GL Galore 1.1

Super smooth OpenGL® scripting with Hollywood

Andreas Falkenhahn

Table of Contents

1 General information ... 1

1.1 Introduction ... 1
1.2 Terms and conditions ... 2
1.3 Requirements ... 3
1.4 Installation ... 3

2 About GL Galore ... 5

2.1 Credits ... 5
2.2 Frequently asked questions ... 5
2.3 Known issues ... 6
2.4 Future ... 6
2.5 History ... 6

3 Usage ... 7

3.1 Activating GL Galore ... 7
3.2 Accessing OpenGL from Hollywood ... 7
3.3 Using a hardware double buffer ... 8
3.4 Drawing graphics ... 9
3.5 Using hardware brushes ... 10
3.6 Multiple displays ... 10
3.7 Working with pointers ... 11
3.8 Hollywood bridge ... 11
3.9 Increasing execution speed ... 11
3.10 Handling mode switches ... 12
3.11 GL Galore as a helper plugin ... 12
3.12 Internal pixel formats ... 13

4 Tutorial ... 15

4.1 OpenGL tutorial ... 15

5 Examples ... 17

5.1 Examples ... 17

6 GL reference ... 19

6.1 gl.Accum ... 19
6.2 gl.AlphaFunc ... 20
6.3 gl.AreTexturesResident ... 22
6.4 gl.ArrayElement ... 23
6.5 gl.Begin ... 23
6.6 gl.BindTexture ... 25
6.7 gl.Bitmap ... 27

6.8 gl.BlendFunc . . . 28
6.9 gl.CallList . . . 30
6.10 gl.CallLists . . . 31
6.11 gl.Clear . . . 31
6.12 gl.ClearAccum . . . 33
6.13 gl.ClearColor . . . 33
6.14 gl.ClearDepth . . . 34
6.15 gl.ClearIndex . . . 34
6.16 gl.ClearStencil . . . 35
6.17 gl.ClipPlane . . . 35
6.18 gl.Color . . . 36
6.19 gl.ColorMask . . . 37
6.20 gl.ColorMaterial . . . 38
6.21 gl.ColorPointer . . . 39
6.22 gl.CopyPixels . . . 40
6.23 gl.CopyTexImage . . . 43
6.24 gl.CopyTexSubImage . . . 44
6.25 gl.CullFace . . . 46
6.26 gl.DeleteLists . . . 47
6.27 gl.DeleteTextures . . . 47
6.28 gl.DepthFunc . . . 48
6.29 gl.DepthMask . . . 49
6.30 gl.DepthRange . . . 49
6.31 gl.Disable . . . 50
6.32 gl.DisableClientState . . . 55
6.33 gl.DrawArrays . . . 56
6.34 gl.DrawBuffer . . . 57
6.35 gl.DrawElements . . . 58
6.36 gl.DrawPixels . . . 59
6.37 gl.DrawPixelsRaw . . . 60
6.38 gl.EdgeFlag . . . 65
6.39 gl.EdgeFlagPointer . . . 65
6.40 gl.Enable . . . 66
6.41 gl.EnableClientState . . . 67
6.42 gl.End . . . 67
6.43 gl.EndList . . . 67
6.44 gl.EvalCoord . . . 68
6.45 gl.EvalMesh . . . 69
6.46 gl.EvalPoint . . . 71
6.47 gl.FeedbackBuffer . . . 72
6.48 gl.Finish . . . 74
6.49 gl.Flush . . . 74
6.50 gl.Fog . . . 75
6.51 gl.FreeFeedbackBuffer . . . 77
6.52 gl.FreeSelectBuffer . . . 77
6.53 gl.FrontFace . . . 77
6.54 gl.Frustum . . . 78
6.55 gl.GenLists . . . 79

6.56 gl.GenTextures ... 80
6.57 gl.Get ... 81
6.58 gl.GetArray ... 95
6.59 gl.GetClipPlane ... 96
6.60 gl.GetError ... 97
6.61 gl.GetLight ... 98
6.62 gl.GetMap ... 100
6.63 gl.GetMaterial ... 101
6.64 gl.GetPixelMap ... 102
6.65 gl.GetPointer ... 103
6.66 gl.GetPolygonStipple ... 104
6.67 gl.GetSelectBuffer ... 105
6.68 gl.GetString ... 105
6.69 gl.GetTexEnv ... 106
6.70 gl.GetTexGen ... 107
6.71 gl.GetTexImage ... 108
6.72 gl.GetTexImageRaw ... 109
6.73 gl.GetTexLevelParameter ... 111
6.74 gl.GetTexParameter ... 112
6.75 gl.Hint ... 113
6.76 gl.Index ... 115
6.77 gl.IndexMask ... 115
6.78 gl.IndexPointer ... 116
6.79 gl.InitNames ... 117
6.80 gl.InterleavedArrays ... 117
6.81 gl.IsEnabled ... 119
6.82 gl.IsList ... 122
6.83 gl.IsTexture ... 123
6.84 gl.Light ... 123
6.85 gl.LightModel ... 125
6.86 gl.LineStipple ... 127
6.87 gl.LineWidth ... 128
6.88 gl.ListBase ... 129
6.89 gl.LoadIdentity ... 130
6.90 gl.LoadMatrix ... 130
6.91 gl.LoadName ... 131
6.92 gl.LogicOp ... 131
6.93 gl.Map ... 133
6.94 gl.MapGrid ... 137
6.95 gl.Material ... 138
6.96 gl.MatrixMode ... 139
6.97 gl.MultMatrix ... 140
6.98 gl.NewList ... 141
6.99 gl.Normal ... 142
6.100 gl.NormalPointer ... 143
6.101 gl.Ortho ... 144
6.102 gl.PassThrough ... 145
6.103 gl.PixelMap ... 146

6.104 gl.PixelStore ... 148
6.105 gl.PixelTransfer ... 152
6.106 gl.PixelZoom ... 154
6.107 gl.PointSize ... 155
6.108 gl.PolygonMode ... 156
6.109 gl.PolygonOffset ... 157
6.110 gl.PolygonStipple ... 158
6.111 gl.PopAttrib ... 159
6.112 gl.PopClientAttrib ... 160
6.113 gl.PopMatrix ... 160
6.114 gl.PopName ... 161
6.115 gl.PrioritizeTextures ... 162
6.116 gl.PushAttrib ... 163
6.117 gl.PushClientAttrib ... 168
6.118 gl.PushMatrix ... 169
6.119 gl.PushName ... 169
6.120 gl.RasterPos ... 170
6.121 gl.ReadBuffer ... 172
6.122 gl.ReadPixels ... 173
6.123 gl.ReadPixelsRaw ... 175
6.124 gl.Rect ... 176
6.125 gl.RenderMode ... 177
6.126 gl.Rotate ... 178
6.127 gl.Scale ... 179
6.128 gl.Scissor ... 180
6.129 gl.SelectBuffer ... 181
6.130 gl.ShadeModel ... 183
6.131 gl.StencilFunc ... 184
6.132 gl.StencilMask ... 185
6.133 gl.StencilOp ... 186
6.134 gl.TexCoord ... 187
6.135 gl.TexCoordPointer ... 188
6.136 gl.TexEnv ... 189
6.137 gl.TexGen ... 190
6.138 gl.TexImage ... 192
6.139 gl.TexImage1D ... 192
6.140 gl.TexImage2D ... 196
6.141 gl.TexParameter ... 199
6.142 gl.TexSubImage ... 202
6.143 gl.TexSubImage1D ... 203
6.144 gl.TexSubImage2D ... 204
6.145 gl.Translate ... 206
6.146 gl.Vertex ... 207
6.147 gl.VertexPointer ... 207
6.148 gl.Viewport ... 208

7 GLU reference 211

7.1 glu.BuildMipmaps 211
7.2 glu.Build1DMipmaps 211
7.3 glu.Build2DMipmaps 213
7.4 glu.Build3DMipmaps 214
7.5 glu.ErrorString 215
7.6 glu.GetString 216
7.7 glu.LookAt 217
7.8 glu.NewNurbsRenderer 217
7.9 glu.NewQuadric 218
7.10 glu.Ortho2D 218
7.11 glu.Perspective 219
7.12 glu.PickMatrix 219
7.13 glu.Project 220
7.14 glu.ScaleImage 221
7.15 glu.ScaleImageRaw 221
7.16 glu.UnProject 223
7.17 nurb:BeginCurve 223
7.18 nurb:BeginSurface 224
7.19 nurb:BeginTrim 224
7.20 nurb:Callback 225
7.21 nurb:Curve 227
7.22 nurb:EndCurve 228
7.23 nurb:EndSurface 228
7.24 nurb:EndTrim 228
7.25 nurb:GetProperty 229
7.26 nurb:LoadSamplingMatrices 229
7.27 nurb:PwlCurve 230
7.28 nurb:SetProperty 230
7.29 nurb:Surface 233
7.30 quad:Cylinder 233
7.31 quad:Disk 234
7.32 quad:DrawStyle 235
7.33 quad:Normals 235
7.34 quad:Orientation 236
7.35 quad:PartialDisk 236
7.36 quad:Texture 237
7.37 quad:Sphere 237

8 GLFW reference 239

8.1 glfw.GetJoystickAxes 239
8.2 glfw.GetJoystickButtons 239
8.3 glfw.GetJoystickName 240
8.4 glfw.JoystickPresent 240

9 Hollywood bridge ... 241
9.1 gl.BitmapFromBrush ... 241
9.2 gl.DrawPixelsFromBrush ... 241
9.3 gl.GetCurrentContext ... 242
9.4 gl.GetTexImageToBrush ... 242
9.5 gl.ReadPixelsToBrush ... 243
9.6 gl.SetCurrentContext ... 243
9.7 gl.TexImageFromBrush ... 244
9.8 gl.TexSubImageFromBrush ... 244
9.9 glu.BuildMipmapsFromBrush ... 244

Appendix A Licenses ... 247
A.1 LuaGL license ... 247
A.2 GLFW license ... 247
A.3 SGI Free Software B license ... 247
A.4 LGPL license ... 248

Index ... 257

1 General information

1.1 Introduction

GL Galore is a plugin for Hollywood that allows you to access the OpenGL® 1.1 command set directly from Hollywood. This makes it possible to write scripts that utilize the host system's 3D hardware to create high-performance, butter-smooth 2D and 3D animation that is calculated completely in hardware by the GPU of your graphics board. This leads to a huge performance boost over the classic Hollywood graphics API which is mostly implemented in software. Especially systems with slower CPUs will benefit greatly from hardware-accelerated drawing offered by OpenGL.

OpenGL is a portable software interface to graphics hardware. It is available for almost every platform in a variety of flavours. On AmigaOS and compatibles, OpenGL is available as MiniGL on AmigaOS 4, TinyGL on MorphOS, StormMesa on AmigaOS 3, and Mesa 3D on AROS. Windows, Mac OS X, and Linux systems are usually shipped with an OpenGL driver already installed. More information about OpenGL can be obtained from http://www.opengl.org. You can find good tutorials about learning OpenGL all over the web.

There are two ways of using GL Galore: You can either access the OpenGL 1.1 API directly or you can use Hollywood's hardware brush functions without making any direct calls to the OpenGL API. Whenever GL Galore is activated, Hollywood hardware brushes are mapped directly to OpenGL textures so they can be drawn and transformed in an extremely fast way on all supported systems. This is especially useful on Windows, Mac OS X, and Linux because Hollywood doesn't support hardware double buffers and brushes on these platforms by default. With GL Galore, however, hardware double buffers and brushes can be used on these platforms now too. So GL Galore can also act as a helper plugin here which adds this functionality to Hollywood without having you write a single line of OpenGL code to utilize it!

On top of that, GL Galore offers wrapper functions for most commands of the OpenGL 1.1 API. These commands are wrapped directly with little to no changes to their original syntax. The only exception concerns OpenGL commands that expect a pointer: In this case, GL Galore usually offers a variant of the command so that it works with Hollywood tables. However, the original pointer variant is also available in GL Galore and can be used for time-critical scripts. Additionally, GL Galore also offers some bridging functions that allow you to convert Hollywood brushes into OpenGL textures and vice versa.

GL Galore can also be useful for rapidly prototyping software written in OpenGL. People who used to program OpenGL using C will greatly appreciate Hollywood's convenient multimedia API which offers functions for almost all common tasks. For example, by using GL Galore to write OpenGL programs you can avoid all the hassle of managing a GL window using one of the many different toolkits out there. Also, jobs like image loading, sound or video playback, font handling and image manipulation become ridiculously easy now thanks to Hollywood's powerful command set which covers almost 700 functions.

GL Galore utilizes the new display adapter plugin interface introduced with Hollywood 6.0. Thus, the plugin will not work with any older versions of Hollywood. It requires at least Hollywood 6.0. Whenever GL Galore is activated, all graphics output will automatically be routed through OpenGL. To benefit from hardware acceleration, however, Hollywood scripts have to follow some rules as described in this manual.

GL Galore comes with extensive documentation in various formats like PDF, HTML, AmigaGuide, and CHM that contains a full OpenGL reference and information about special functions in GL Galore. On top of that, many example scripts are included in the distribution archive to get you started really quickly.

All of this makes GL Galore the ultimate OpenGL scripting experience combining the best of both worlds into one powerful plugin: Hollywood's extensive and convenient multimedia function set and OpenGL's raw graphics power!

1.2 Terms and conditions

GL Galore is © Copyright 2014-2017 by Andreas Falkenhahn (in the following referred to as "the author"). All rights reserved.

The program is provided "as-is" and the author cannot be made responsible of any possible harm done by it. You are using this program absolutely at your own risk. No warranties are implied or given by the author.

This plugin may be freely distributed as long as the following three conditions are met:

1. No modifications must be made to the plugin.
2. It is not allowed to sell this plugin.
3. If you want to put this plugin on a coverdisc, you need to ask for permission first.

This software uses LuaGL by Fabio Guerra, Cleyde Marlyse, and Antonio Scuri. See Section A.1 [LuaGL license], page 247, for details.

This software uses GLFW by Marcus Geelnard and Camilla Berglund. See Section A.2 [GLFW license], page 247, for details.

This software uses StormMesa by Sam Jordan and Brian Paul. See Section A.4 [LGPL license], page 248, for details.

This documentation is based on the OpenGL® 2.1 reference manual (C) 1991-2006 Silicon Graphics, Inc. See Section A.3 [SGI Free Software B license], page 247, for details.

OpenGL® and the oval logo are trademarks or registered trademarks of Silicon Graphics, Inc. in the United States and/or other countries worldwide.

Amiga is a registered trademark of Amiga, Inc.

All other trademarks belong to their respective owners.

DISCLAIMER: THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDER AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED 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 PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU

FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (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 PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1.3 Requirements

- Hollywood 6.0 or better
- Windows: requires at least Windows 2000
- Mac OS X: requires at least 10.5 on PowerPC or 10.6 on Intel Macs
- MorphOS: requires TinyGL with MorphOS 3.8 or better
- AmigaOS 3: requires StormMesa, 68040 or 68060, and an FPU
- AmigaOS 4: requires MiniGL
- AROS: requires Mesa 3D

1.4 Installation

Installing GL Galore is straightforward and simple: Just copy the file `glgalore.hwp` for the platform to Hollywood's plugins directory. On all systems except on AmigaOS and compatibles, plugins must be stored in a directory named `Plugins` that is in the same directory as the main Hollywood program. On AmigaOS and compatible systems, plugins must be installed to `LIBS:Hollywood` instead. On Mac OS X, the `Plugins` directory must be inside the `Resources` directory of the application bundle, i.e. inside the `HollywoodInterpreter.app/Contents/Resources` directory. Note that `HollywoodInterpreter.app` is stored inside the `Hollywood.app` application bundle itself, namely in `Hollywood.app/Contents/Resources`.

Afterwards merge the contents of the `Examples` folder with the `Examples` folder that is part of your Hollywood installation. All GL Galore examples will then appear in Hollywood's GUI and you can launch and view them conveniently from the Hollywood GUI or IDE.

On Windows you should also copy the file `GLGalore.chm` to the `Docs` directory of your Hollywood installation. Then you will be able to get online help by pressing F1 when the cursor is over a GL Galore function in the Hollywood IDE.

On Linux and Mac OS copy the `GLGalore` directory that is inside the `Docs` directory of the GL Galore distribution archive to the `Docs` directory of your Hollywood installation. Note that on Mac OS the `Docs` directory is within the `Hollywood.app` application bundle, i.e. in `Hollywood.app/Contents/Resources/Docs`.

2 About GL Galore

2.1 Credits

GL Galore was written by Andreas Falkenhahn. Additional coding by Fabio Guerra, Cleyde Marlyse, and Antonio Scuri. Work on this project was started in January 2014 as a proof-of-concept demonstration of Hollywood 6.0's powerful new display adapter API which allows plugins to take over Hollywood's entire display handler and replace it with a custom driver. It was then successively expanded into a full wrapper for the OpenGL 1.1 API including some bridging functions between Hollywood and OpenGL as well as a hardware brush and double-buffer driver for all the platforms that are not supported by Hollywood's inbuilt hardware brush and double-buffer driver.

Thanks go to Frank Mariak, Mark Olsen, Hans de Ruiter, Frank Wille, Krzysztof Smiechowicz, and Sam Jordan.

If you need to contact me, you can either send an e-mail to `andreas@airsoftsoftwair.de` or use the contact form on `http://www.hollywood-mal.com`.

2.2 Frequently asked questions

This section covers some frequently asked questions. Please read them first before asking on the mailing list or forum because your problem might have been covered here.

Q: Why does GL Galore report all keyboard events in upper case notation?

A: That's a limitation of GLFW which is used by GL Galore. It currently doesn't allow listeners to distinguish between upper and lower case key presses when using raw keyboard listeners. See Section 2.3 [Known issues], page 6, for details. If you have Hollywood 7.0 or better, you can just listen to the `VanillaKey` event handler to get the real keyboard events with full Unicode support.

Q: Why are the Y and Z keys swapped on German keyboards?

A: That's because GLFW's raw key listener is based on the US keyboard layout. See Section 2.3 [Known issues], page 6, for details. If you are on Hollywood 7.0 or better, you can just listen to the `VanillaKey` event handler instead. This will give you the real keyboard events with full Unicode support.

Q: Is there a Hollywood forum where I can get in touch with other users?

A: Yes, please check out the "Community" section of the official Hollywood Portal online at `http://www.hollywood-mal.com`.

Q: How do I quit scripts that run in fullscreen mode?

A: Just press CTRL+C. This will always work except when CTRL+C has been explicitly disabled using Hollywood's `CtrlCQuit()` function.

Q: Where can I ask for help?

A: There's a lively forum at http://forums.hollywood-mal.com and we also have a mailing list which you can access at airsoft_hollywood@yahoogroups.com. Visit http://www.hollywood-mal.com for information on how to join the mailing list.

Q: I have found a bug.

A: Please post about it in the dedicated sections of the forum or the mailing list.

2.3 Known issues

Here is a list of things that GL Galore doesn't support yet or that may be confusing in some way:

- menus are unsupported
- the mouse wheel is unsupported
- the keyboard listener that is mapped to Hollywood's `OnKeyDown` and `OnKeyUp` event handlers currently only supports raw key codes based on the US keyboard layout; this means that all character keys will be returned as upper-case and on German keyboards the position of the Y and Z keys will be swapped. This is because of a limitation in GLFW which doesn't support fine-tuned listening (i.e. key down, key repeat, key up) using international keyboards; people who are on Hollywood 7.0 or better can just use the `VanillaKey` event handler instead; this event handler will deliver real keyboard events including full Unicode support
- not all display styles are supported

2.4 Future

Here are some things that are on my to do list:

- add support for tesselation
- improve support for OpenGL 1.2
- integrate with FTGL for 3D text effects
- add some more convenience functions that make it easier to use OpenGL
- add support for embedding OpenGL displays in MUI GUIs via MUI Royale

Don't hesitate to contact me if GL Galore lacks a certain feature that is important for your project.

2.5 History

Please see the file `history.txt` for a complete change log of GL Galore.

3 Usage

3.1 Activating GL Galore

All you have to do to make your script use OpenGL instead of Hollywood's inbuilt graphics driver is adding the following line to the top of your script:

```
@REQUIRE "glgalore"
```

Alternatively, if you are using Hollywood from a console, you can also start your script like this:

```
Hollywood test.hws -requireplugins glgalore
```

Once the GL Galore plugin has been activated for your script, it will reroute all of Hollywood's graphics output through OpenGL. Note that this will usually be slower than Hollywood's inbuilt graphics driver for scripts that aren't optimized for OpenGL. To get an optimal performance with OpenGL, your script needs to use a hardware-accelerated double buffer. See Section 3.3 [Using a hardware double buffer], page 8, for details.

GL Galore accepts the following arguments in its `@REQUIRE` call:

`ForceFullRefresh:`
: If this tag is set to `False`, GL Galore will only refresh the parts of the display that have actually changed. This is quicker but it doesn't work correctly with older OpenGL implementations (especially on Amiga) because they often do not offer pixel perfect positioning of graphics. That is why this tag defaults to `True`, which means that GL Galore will always refresh the full display whenever something is drawn. Note that this tag is only used when GL Galore is running without a hardware double buffer. In hardware double-buffered mode, front and back buffers cause a full refresh anyway.

Here is an example of how to pass arguments to the `@REQUIRE` preprocessor command:

```
@REQUIRE "glgalore", {ForceFullRefresh = False}
```

Alternatively, you can also use the `-requiretags` console argument to pass these arguments. See the Hollywood manual for more information.

3.2 Accessing OpenGL from Hollywood

GL Galore directly wraps all OpenGL commands to Hollywood with little to no changes to their original syntax. After calling `@REQUIRE` on GL Galore, all GL functions will be made available inside a "gl" table and the GLU functions will be made available inside a "glu" table.

Calling OpenGL functions from Hollywood is much simpler than using OpenGL directly because the argument specification (e.g., '2d', '3f', '4sv') at the end of most OpenGL functions names has been removed. For example, GL Galore's `gl.Light()` function binds the OpenGL functions: `glLightf`, `glLightfv`, `glLighti`, `glLightiv`. The number of parameters passed to `gl.Light()` defines the correct function to use.

GL Galore usually uses the floating point versions of all the OpenGL functions with the highest possible precision. Some functions that have a type parameter simply use the most

precise possible (usually `#GL_DOUBLE` or `#GL_FLOAT`) and the format parameter is not used. When `stride` is not used, then it is assumed to be 0.

The color and the vector data can be passed as a Hollywood table or as multiple parameters. A vector can have 2, 3 or 4 values (x, y, z, w), and colors can have 3 or 4 values (red, green, blue, alpha).

For example:

```
v1 = {0, 0}
v2 = {1, 1}
yellow = {1, 1, 0}
gl.Color(yellow)
gl.Vertex(v1)
gl.Vertex(v2)
```

Or you can also do:

```
gl.Color(1, 1, 0)
gl.Vertex(0, 0)
gl.Vertex(1, 1)
```

There are some OpenGL commands that expect a pointer. In this case, GL Galore usually offers a variant of the command so that it works with Hollywood tables. However, the original pointer variant is also available in GL Galore and can be used for time-critical scripts. For example, `gl.ReadPixels()` reads pixel data from the frame buffer into a Hollywood table whereas `gl.ReadPixelsRaw()` reads pixel data into a memory buffer directly which is much faster of course, but requires you to work with pointers. See Section 3.7 [Working with pointers], page 11, for details.

3.3 Using a hardware double buffer

If you want your script to benefit from OpenGL's hardware-accelerated drawing functions, you need to use a hardware double buffer and do all your drawing within that double buffer. Using a hardware double buffer will also ensure that graphics output is synchronized with your monitor's refresh rate to prevent any flickering. To get an optimal performance with OpenGL, your main loop should always look like this:

```
@REQUIRE "glgalore"

BeginDoubleBuffer(True)  ; set up a hardware double buffer

Repeat
    ....           ; draw the next frame here
    Flip()         ; wait for vertical refresh, then flip buffers
    CheckEvent()   ; run event callbacks
Forever
```

The call to `CheckEvent()` is only necessary if your script needs to listen to event handlers that have been installed using `InstallEventHandler()`. Note that you should not draw the next frame in an interval callback that runs at a constant frame rate (say 50fps) because such a setup won't guarantee that drawing is synchronized with the vertical refresh as different monitors use different refresh rates so you might get flickery graphics. If you do

your drawing like above, you can be sure that front and back buffers will be flipped in perfect synchronization with the monitor's vertical refresh.

Additionally, you need to take care of how you actually draw your graphics because most of Hollywood's drawing commands operate entirely in software mode and thus do not benefit from hardware acceleration. See Section 3.4 [Drawing graphics], page 9, for details.

Important: OpenGL is designed to be used with double buffers. Thus, all OpenGL drawing commands must be called within a double buffer. Drawing outside a double buffer with OpenGL is unsupported.

3.4 Drawing graphics

For an optimal performance you need to be very careful concerning the way you draw your graphics. Most of Hollywood's drawing commands are implemented in software only, i.e. they draw using the CPU instead of the GPU. This can become quite a bottleneck especially on slower CPUs. Thus, you should draw directly using the OpenGL commands offered by GL Galore whenever and whereever possible.

Nevertheless, there are a few Hollywood commands which are redirected to use OpenGL directly when GL Galore has been activated. These are the following:

```
Box()
Cls()
Line()
Plot()
DisplayBrush()
```

You can use these commands with OpenGL without any performance penalty. However, there are some restrictions: `Box()`, `Line()`, and `WritePixel()` will only be redirected to OpenGL in case the fill style is either `#FILLNONE` or `#FILLCOLOR` and no other form styles like `#EDGE` or `#SHADOW` are active. As soon as you want to draw with other fill or form styles, these commands will fall back to their software counterparts and thus will be very slow.

`DisplayBrush()` will only use OpenGL directly when called with a hardware brush. See Section 3.5 [Using hardware brushes], page 10, for details. When used with a software brush, i.e. a brush that doesn't reside in video memory, `DisplayBrush()` will draw the brush using the CPU which is much slower.

When mixing Hollywood and OpenGL drawing commands, however, there is another potential problem that you have to be aware of: Since OpenGL is a state machine, changes to the GL state made by one of Hollywood's drawing commands can affect subsequent calls to OpenGL commands. Thus, you might need to restore certain states after calling a Hollywood command which is redirected to OpenGL, e.g. the current color, transformation matrix, matrix mode, enable texturing, blending or depth test again, etc. This can get quite tedious so it is often easier to use only OpenGL commands in order to avoid having to restore states after calling Hollywood commands.

Finally, don't forget that you should do all your drawing inside a hardware double buffer loop. See Section 3.3 [Using a hardware double buffer], page 8, for details.

3.5 Using hardware brushes

GL Galore supports the creation of hardware brushes. Hardware brushes reside in GPU memory and thus can be drawn in no time. On most graphics boards, they can also be scaled and transformed by the GPU in an extremely efficient way. To make Hollywood create a hardware brush, all you have to do is set the optional "Hardware" tag to `True`. This tag is supported by most of the Hollywood commands which create brushes.

Here is an example:

```
@REQUIRE "glgalore"  ; make sure this line is first
@BRUSH 1, "sprites.png", {Hardware = True}
```

In the code above, GL Galore will create brush 1 in video memory. It can then be drawn using the GPU at almost no cost. Keep in mind, though, that hardware brushes can only be drawn to hardware double buffers. See Section 3.3 [Using a hardware double buffer], page 8, for details.

To transform a hardware brush, you can use the `ScaleBrush()`, `RotateBrush()`, and `TransformBrush()` commands. Transformations of hardware brushes are usually also GPU-accelerated and thus many times faster than transformations done by the CPU.

Note that hardware brushes can only be drawn to the display that was specified when allocating them. Thus, if your script uses multiple displays, you need to tell Hollywood the identifier of the display you want to use this hardware brush with. This can be done by specifying the "Display" tag along the "Hardware" tag. Here is an example:

```
@REQUIRE "glgalore"  ; make sure this line is first
@DISPLAY 1, {Title = "First display"}
@DISPLAY 2, {Title = "Second display"}
@BRUSH 1, "sprites.png", {Hardware = True, Display = 1}
@BRUSH 2, "sprites.png", {Hardware = True, Display = 2}
```

The code above will allocate brush 1 in a way that it can be drawn to display 1 and it will allocate brush 2 in a way that it can be drawn to display 2. It won't be possible, however, to draw brush 2 to display 1 or brush 1 to display 2! OpenGL hardware brushes are always display-dependent and can only be drawn to the display they were allocated for.

Please see the Hollywood manual for more information on hardware brushes and hardware double buffers.

You can also use GL Galore as a helper plugin to add hardware brush support to Hollywood on Windows, Mac OS X, and Linux. By default, Hollywood doesn't support hardware brushes on these systems but GL Galore can add this feature to Hollywood. See Section 3.11 [GL Galore as a helper plugin], page 12, for details.

The `SmoothScroll.hws` example script that comes with GL Galore demonstrates how to use hardware brushes and a hardware double buffer without any calls to OpenGL.

3.6 Multiple displays

When using multiple displays, GL Galore maintains a separate OpenGL context for each display. Thus, you need to tell OpenGL which context all calls to the GL should operate on. This is done by calling the `gl.SetCurrentContext()` function which makes the GL context of the specified Hollywood display the current context. See Section 9.6 [gl.SetCurrentContext], page 243, for details.

When dealing with hardware brushes, you also need to be careful when using multiple displays because hardware brushes in OpenGL are display-dependent. They can only be drawn to the display that was used to allocate them. See Section 3.5 [Using hardware brushes], page 10, for details.

3.7 Working with pointers

Several OpenGL functions expect you to pass a pointer to a raw memory buffer to them. Working with pointers directly is the most efficient way to interact with OpenGL since it avoids any overhead created by having to read the contents of Hollywood tables into memory buffers first.

You can use the functions of Hollywood's memory block library to allocate memory buffers, read or write to them, and obtain a pointer to their raw memory buffer. To allocate a memory buffer, you use the `AllocMem()` function, to read from a memory buffer you use `Peek()` while `Poke()` can be used to write to a memory buffer. Finally, `GetMemPointer()` returns the pointer of a memory block object. You can pass the return value of `GetMemPointer()` to all OpenGL functions which expect a pointer argument.

Here is an example:

```
AllocMem(1, 640*480*4)
Local ptr = GetMemPointer(1)
gl.ReadPixelsRaw(0, 0, 640, 480, #GL_BGRA, #GL_UNSIGNED_BYTE, ptr)
... ; do something with the data
FreeMem(1)
```

The code above reads 480 rows of 640 pixels into memory block object 1. You could then write the data to a file using `WriteMem()`, you could convert it to a table using `MemToTable()` or read individual values from it using `Peek()`. See the documentation of Hollywood's memory block library for more information.

3.8 Hollywood bridge

GL Galore offers some additional functions that are not part of the official OpenGL API. These functions allow you to conveniently use Hollywood objects like brushes with OpenGL. For example, the `gl.TexImageFromBrush()` function allows you to upload a Hollywood brush as an OpenGL texture and the `gl.GetTexImageToBrush()` functions allows you to convert an OpenGL texture back into a Hollywood brush.

See the chapter "Hollywood bridge" for all available functions.

3.9 Increasing execution speed

To increase the raw execution speed of your script, you can disable Hollywood's line hook using the `DisableLineHook()` and `EnableLineHook()` commands. This will improve your script's execution speed significantly in case lots of Hollywood code needs to be run to draw the next frame. Keep in mind, though, that you have to enable the line hook for every frame you draw or your window will become unresponsive. Here's what a speed-optimized implemention of the main loop could look like:

```
@REQUIRE "glgalore"
```

```
BeginDoubleBuffer(True)  ; set up a hardware double buffer

Repeat
    DisableLineHook() ; disable line hook while drawing the next frame
    p_DrawFrame()     ; draw the next frame here
    EnableLineHook()  ; enable line hook again
    Flip()            ; wait for vertical refresh, then flip buffers
    CheckEvent()      ; run event callbacks
Forever
```

Note that you'll only notice a speed difference here if `p_DrawFrame()` executes many lines of Hollywood code. If `p_DrawFrame()` only consists of 20 lines of code, you won't notice any difference. It's only noticeable with hundreds of code lines or long loops.

See the documentation of `DisableLineHook()` and `EnableLineHook()` in the Hollywood manual for more information.

3.10 Handling mode switches

As you might know, Hollywood offers a hotkey to switch between windowed and fullscreen mode. Whenever the user presses ALT+RETURN or COMMAND+RETURN Hollywood will automatically switch modes between windowed and fullscreen. This behaviour is enabled by default. It can be disabled by setting the `ModeTag` to `False` in the `@DISPLAY` preprocessor command.

When using Hollywood's inbuilt display driver, mode switches are handled automatically by Hollywood and there is nothing your script needs to do. This is different with GL Galore. With GL Galore you will have to reinitialize your GL context after a mode switch. This is necessary because the old GL context will be destroyed when Hollywood switches modes. Thus, all current GL states will be lost in a mode switch. This also includes textures and display lists that your script has allocated. After a mode switch your script's GL context will be replaced by a vanilla GL context that is identical to the one your script is started with.

In order to support mode switches with GL Galore, you have to install a listener on the `ModeSwitch` tag using `InstallEventHandler()`. Whenever this event handler triggers, you will have to reinitialize your GL context and set all states to the desired values. Normally, you just have to run your initialization code, that sets up your GL context at the beginning of the script, again whenever the `ModeSwitch` event triggers. If this is too much hassle for you, you can also just disable automatic mode switching.

Please note that it's not necessary to handle mode switches manually in case you're not using the OpenGL API directly. Hardware brushes allocated by GL Galore will be automatically transferred to the new GL context by GL Galore so you don't have to do anything about them. It is only necessary when programming OpenGL directly.

3.11 GL Galore as a helper plugin

GL Galore can also be used as a helper plugin to work around the problem that Hollywood only supports hardware-accelerated double buffers and brushes on AmigaOS and compatibles. They aren't supported on Windows, Mac OS X, or Linux. If you install and

`@REQUIRE` GL Galore, however, hardware double buffer and hardware brush support will also be available on Windows, Mac OS X, and Linux because GL Galore supports this.

Thus, you can also use GL Galore as a helper plugin just to get hardware-accelerated double buffer support on Windows, Mac OS X, and Linux. You don't even have to use any of the OpenGL commands directly. You can just `@REQUIRE` GL Galore, set up a hardware double buffer and then draw to it using hardware brushes. This allows you to utilize hardware acceleration without having to write a single line of OpenGL code!

On AmigaOS and compatibles this isn't necessary since Hollywood already supports hardware accelerated double buffers and brushes by default. Still, using GL Galore on AmigaOS as a hardware double buffer driver can be of benefit in full screen mode because GL Galore uses drawing which is perfectly synchronized with the monitor's vertical refresh so it usually looks better than double buffers managed by Hollywood directly.

See Section 3.3 [Using a hardware double buffer], page 8, for details.

See Section 3.5 [Using hardware brushes], page 10, for details.

The `SmoothScroll.hws` example script that comes with GL Galore demonstrates how to use hardware brushes and a hardware double buffer without any calls to OpenGL.

3.12 Internal pixel formats

OpenGL commands which create textures, e.g. `gl.TexImage2D()` or `gl.CopyTexImage()` accept an `internalFormat` parameter which allows you to specify the internal format of the texture. The following format constants are currently supported by GL Galore:

```
#GL_ALPHA
#GL_ALPHA4
#GL_ALPHA8
#GL_ALPHA12
#GL_ALPHA16
#GL_LUMINANCE
#GL_LUMINANCE4
#GL_LUMINANCE8
#GL_LUMINANCE12
#GL_LUMINANCE16
#GL_LUMINANCE_ALPHA
#GL_LUMINANCE4_ALPHA4
#GL_LUMINANCE6_ALPHA2
#GL_LUMINANCE8_ALPHA8
#GL_LUMINANCE12_ALPHA4
#GL_LUMINANCE12_ALPHA12
#GL_LUMINANCE16_ALPHA16
#GL_INTENSITY
#GL_INTENSITY4
#GL_INTENSITY8
#GL_INTENSITY12
#GL_INTENSITY16
#GL_RGB
#GL_R3_G3_B2
```

```
#GL_RGB4
#GL_RGB5
#GL_RGB8
#GL_RGB10
#GL_RGB12
#GL_RGB16
#GL_RGBA
#GL_RGBA2
#GL_RGBA4
#GL_RGB5_A1
#GL_RGBA8
#GL_RGB10_A2
#GL_RGBA12
#GL_RGBA16
#GL_DEPTH_COMPONENT
```

Note that `gl.TexImage1D()` and `gl.TexImage2D()` also accept the special values 1, 2, 3, and 4 as valid internal pixel formats but `gl.CopyTexImage()` doesn't support this.

4 Tutorial

4.1 OpenGL tutorial

Unfortunately, there is currently no tutorial to get you started with GL Galore. The internet, however, is full of beginner's tutorials for OpenGL which you can use to get into the engine. Since GL Galore just wraps the OpenGL API, it is mostly simple and straightforward to port code written for other programming languages to GL Galore. The examples that are shipped with GL Galore can also help you to get started with GL Galore.

Finally, the Hollywood forums are always a good place to ask your question if you're stuck programming with GL Galore. Just visit `http://forums.hollywood-mal.com` and ask.

5 Examples

5.1 Examples

GL Galore comes with a number of examples that demonstrate certain features and should allow you to get started really quickly. Here's a list of examples that are distributed with GL Galore:

BlockTube
: A swirling, falling tunnel of reflective slabs which fade from hue to hue. Original code by Lars Damerow

Boing
: A clone of the first graphics demo for the Amiga 1000, which was written by Dale Luck and RJ Mical during a break at the 1984 CES. Original code by Jim Brooks

Cel shading
: Demonstrates cel shading. Original code by Jeff Molofee

Cityflow
: Waves move across a sea of boxes. The city swells. The walls are closing in. Original code by Jamie Zawinski

Cube
: The OpenGL equivalent of "Hello World"

Gears
: The classic OpenGL gears demo. Original code by Brian Paul

Gears 2
: A variant of the OpenGL gears demo based on code found in the MiniGL SDK

Gears 3
: The OpenGL gears demo, this time with a texture

GLMatrix
: The 3D digital rain effect, as seen in the title sequence of a popular movie. Based on code by Jamie Zawinski

Morph3D
: Platonic solids that turn inside out and get spikey. Based on code by Marcelo F. Vianna

MultiDisplays
: Demonstrates how to use multiple displays with GL Galore

Simple
: Simple rotating GL triangle. Based on code by Camilla Berglund

SmoothScroll
: Uses OpenGL for hardware-accelerated 2D drawing of Hollywood brushes

SplitView
: Renders four views of the same scene in one window. Based on code by Camilla Berglund

Spots
: Demonstrates GL lights. Based on code by Mark J. Kilgard

Sproingies
: Slinky-like creatures walk down an infinite staircase and occasionally explode! Based on code by Ed Mackey

Warp
: Example of what an extreme field of view can do

Wave
: Wave simulation in OpenGL. Based on code by Jakob Thomsen

6 GL reference

6.1 gl.Accum

NAME

gl.Accum – operate on the accumulation buffer

SYNOPSIS

```
gl.Accum(op, value)
```

FUNCTION

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. Effects such as antialiasing (of points, lines, and polygons), motion blur, and depth of field can be created by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling `gl.Get()` four times, with arguments `#GL_ACCUM_RED_BITS`, `#GL_ACCUM_GREEN_BITS`, `#GL_ACCUM_BLUE_BITS`, and `#GL_ACCUM_ALPHA_BITS`. Regardless of the number of bits per component, the range of values stored by each component is -1 through 1 . The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

`gl.Accum()` operates on the accumulation buffer. The first argument, `op`, is a symbolic constant that selects an accumulation buffer operation. The second argument, `value`, is a floating-point value to be used in that operation. Five operations are specified: `#GL_ACCUM`, `#GL_LOAD`, `#GL_ADD`, `#GL_MULT`, and `#GL_RETURN`.

All accumulation buffer operations are limited to the area of the current scissor box and applied identically to the red, green, blue, and alpha components of each pixel. If a `gl.Accum()` operation results in a value outside the range -1 through 1, the contents of an accumulation buffer pixel component are undefined.

The operations are as follows:

`#GL_ACCUM`
: Obtains R, G, B, and A values from the buffer currently selected for reading (See Section 6.121 [gl.ReadBuffer], page 172, for details.). Each component value is divided by 2^n - 1 , where n is the number of bits allocated to each color component in the currently selected buffer. The result is a floating-point value in the range 0 through 1 , which is multiplied by value and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.

`#GL_LOAD`
: Similar to `#GL_ACCUM`, except that the current value in the accumulation buffer is not used in the calculation of the new value. That is, the R, G, B, and A values from the currently selected buffer are divided by 2^n - 1 , multiplied by value, and then stored in the corresponding accumulation buffer cell, overwriting the current value.

`#GL_ADD` Adds value to each R, G, B, and A in the accumulation buffer.

`#GL_MULT` Multiplies each R, G, B, and A in the accumulation buffer by value and returns the scaled component to its corresponding accumulation buffer location.

`#GL_RETURN`
Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by value, then multiplied by 2^n - 1 , clamped to the range 0 2 n - 1 , and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

To clear the accumulation buffer, call `gl.ClearAccum()` with R, G, B, and A values to set it to, then call `gl.Clear()` with the accumulation buffer enabled.

Only pixels within the current scissor box are updated by a `gl.Accum()` operation.

Please consult an OpenGL reference manual for more information.

INPUTS

`op` specifies the accumulation buffer operation. Symbolic constants `#GL_ACCUM`, `#GL_LOAD`, `#GL_ADD`, `#GL_MULT`, and `#GL_RETURN` are accepted

`value` specifies a floating-point value used in the accumulation buffer operation. `op` determines how `value` is used

ERRORS

`#GL_INVALID_ENUM` is generated if op is not an accepted value.

`#GL_INVALID_OPERATION` is generated if there is no accumulation buffer.

`#GL_INVALID_OPERATION` is generated if `gl.Accum()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ACCUM_RED_BITS`

`gl.Get()` with argument `#GL_ACCUM_GREEN_BITS`

`gl.Get()` with argument `#GL_ACCUM_BLUE_BITS`

`gl.Get()` with argument `#GL_ACCUM_ALPHA_BITS`

6.2 gl.AlphaFunc

NAME

gl.AlphaFunc – specify the alpha test function

SYNOPSIS

`gl.AlphaFunc(func, ref)`

FUNCTION

The alpha test discards fragments depending on the outcome of a comparison between an incoming fragment's alpha value and a constant reference value. `gl.AlphaFunc()`

specifies the reference value and the comparison function. The comparison is performed only if alpha testing is enabled. By default, it is not enabled. (See `gl.Enable()` and `gl.Disable()` of `#GL_ALPHA_TEST`.)

`func` and `ref` specify the conditions under which the pixel is drawn. The incoming alpha value is compared to `ref` using the function specified by `func`. If the value passes the comparison, the incoming fragment is drawn if it also passes subsequent stencil and depth buffer tests. If the value fails the comparison, no change is made to the frame buffer at that pixel location. The comparison functions are as follows:

`#GL_NEVER`
: Never passes.

`#GL_LESS` Passes if the incoming alpha value is less than the reference value.

`#GL_EQUAL`
: Passes if the incoming alpha value is equal to the reference value.

`#GL_LEQUAL`
: Passes if the incoming alpha value is less than or equal to the reference value.

`#GL_GREATER`
: Passes if the incoming alpha value is greater than the reference value.

`#GL_NOTEQUAL`
: Passes if the incoming alpha value is not equal to the reference value.

`#GL_GEQUAL`
: Passes if the incoming alpha value is greater than or equal to the reference value.

`#GL_ALWAYS`
: Always passes (initial value).

`gl.AlphaFunc()` operates on all pixel write operations, including those resulting from the scan conversion of points, lines, polygons, and bitmaps, and from pixel draw and copy operations. `gl.AlphaFunc()` does not affect screen clear operations.

Please consult an OpenGL reference manual for more information.

INPUTS

`func` specifies the alpha comparison function (see above)

`ref` specifies the reference value that incoming alpha values are compared to. This value is clamped to the range 0 through 1, where 0 represents the lowest possible alpha value and 1 the highest possible value (the initial reference value is 0)

ERRORS

`#GL_INVALID_ENUM` is generated if `func` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.AlphaFunc()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ALPHA_TEST_FUNC`

`gl.Get()` with argument `#GL_ALPHA_TEST_REF`

`gl.IsEnabled()` with argument `#GL_ALPHA_TEST`

6.3 gl.AreTexturesResident

NAME

gl.AreTexturesResident – determine if textures are loaded in texture memory

SYNOPSIS

```
residencesArray = gl.AreTexturesResident(texturesArray)
```

FUNCTION

GL establishes a working set of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

`gl.AreTexturesResident()` queries the texture residence status of the n textures named by the elements of `texturesArray` and returns their status in the table `residencesArray`.

The residence status of a single bound texture may also be queried by calling `gl.GetTexParameter()` with the target argument set to the target to which the texture is bound, and the `pname` argument set to `#GL_TEXTURE_RESIDENT`. This is the only way that the residence status of a default texture can be queried.

`gl.AreTexturesResident()` returns the residency status of the textures at the time of invocation. It does not guarantee that the textures will remain resident at any other time.

If textures reside in virtual memory (there is no texture memory), they are considered always resident.

Some implementations may not load a texture until the first use of that texture.

Please consult an OpenGL reference manual for more information.

INPUTS

`texturesArray`
: specifies an array containing the names of the textures to be queried

RESULTS

`residencesArray`
: an array in which the texture residence status is returned

ERRORS

`#GL_INVALID_VALUE` is generated if any element in `texturesArray` is 0 or does not name a texture. In that case, the function returns `Nil`.

`#GL_INVALID_OPERATION` is generated if `gl.AreTexturesResident()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexParameter()` with parameter name `#GL_TEXTURE_RESIDENT` retrieves the residence status of a currently bound texture.

6.4 gl.ArrayElement

NAME

gl.ArrayElement – render a vertex using the specified vertex array element

SYNOPSIS

`gl.ArrayElement(i)`

FUNCTION

`gl.ArrayElement()` commands are used within `gl.Begin()` / `gl.End()` pairs to specify vertex and attribute data for point, line, and polygon primitives. If `#GL_VERTEX_ARRAY` is enabled when `gl.ArrayElement()` is called, a single vertex is drawn, using vertex and attribute data taken from location `i` of the enabled arrays. If `#GL_VERTEX_ARRAY` is not enabled, no drawing occurs but the attributes corresponding to the enabled arrays are modified.

Use `gl.ArrayElement()` to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because each call specifies only a single vertex, it is possible to explicitly specify per-primitive attributes such as a single normal for each triangle.

Changes made to array data between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` may affect calls to `gl.ArrayElement()` that are made within the same `gl.Begin()` / `gl.End()` period in nonsequential ways. That is, a call to `gl.ArrayElement()` that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

`gl.ArrayElement()` is included in display lists. If `gl.ArrayElement()` is entered into a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client-side state, their values affect display lists when the lists are created, not when the lists are executed.

Please consult an OpenGL reference manual for more information.

INPUTS

`i` specifies an index into the enabled vertex data arrays

ERRORS

`#GL_INVALID_VALUE` may be generated if `i` is negative.

`#GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

6.5 gl.Begin

NAME

gl.Begin – delimit the vertices of a primitive or a group of like primitives

SYNOPSIS

`gl.Begin(mode)`

FUNCTION

`gl.Begin()` and `gl.End()` delimit the vertices that define a primitive or a group of like primitives. `gl.Begin()` accepts a single argument that specifies in which of ten ways the vertices are interpreted. Taking n as an integer count starting at one, and N as the total number of vertices specified, the interpretations are as follows:

`#GL_POINTS`

Treats each vertex as a single point. Vertex n defines point n. N points are drawn.

`#GL_LINES`

Treats each pair of vertices as an independent line segment. Vertices 2^n - 1 and 2^n define line n. N/2 lines are drawn.

`#GL_LINE_STRIP`

Draws a connected group of line segments from the first vertex to the last. Vertices n and n + 1 define line n. N - 1 lines are drawn.

`#GL_LINE_LOOP`

Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n + 1 define line n. The last line, however, is defined by vertices N and 1. N lines are drawn.

`#GL_TRIANGLES`

Treats each triplet of vertices as an independent triangle. Vertices 3^n - 2 , 3^n - 1 , and 3^n define triangle n. N/3 triangles are drawn.

`#GL_TRIANGLE_STRIP`

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n, vertices n, n + 1 , and n + 2 define triangle n. For even n, vertices n + 1 , n, and n + 2 define triangle n. N - 2 triangles are drawn.

`#GL_TRIANGLE_FAN`

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1, n + 1 , and n + 2 define triangle n. N - 2 triangles are drawn.

`#GL_QUADS`

Treats each group of four vertices as an independent quadrilateral. Vertices 4^n - 3 , 4^n - 2 , 4^n - 1 , and 4^n define quadrilateral n. N/4 quadrilaterals are drawn.

`#GL_QUAD_STRIP`

Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices 2^n - 1 , 2^n , 2^n + 2 , and 2^n + 1 define quadrilateral n. N/2 - 1 quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

`#GL_POLYGON`

Draws a single, convex polygon. Vertices 1 through N define this polygon.

Only a subset of GL commands can be used between `gl.Begin()` and `gl.End()`. The commands are `gl.Vertex()`, `gl.Color()`, `gl.Index()`, `gl.Normal()`, `gl.TexCoord()`, and `gl.Material()`, `gl.EvalCoord()`, `gl.EvalPoint()`, `gl.EdgeFlag()`, and `gl.ArrayElement()`. Also, it is acceptable to use `gl.CallList()` or `gl.CallLists()` to execute display lists that include only the preceding commands. If any other GL command is executed between `gl.Begin()` and `gl.End()`, the error flag is set and the command is ignored.

Regardless of the value chosen for mode, there is no limit to the number of vertices that can be defined between `gl.Begin()` and `gl.End()`. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the rest are drawn.

The minimum specification of vertices for each primitive is as follows: 1 for a point, 2 for a line, 3 for a triangle, 4 for a quadrilateral, and 3 for a polygon. Modes that require a certain multiple of vertices are `#GL_LINES` (2), `#GL_TRIANGLES` (3), `#GL_QUADS` (4), and `#GL_QUAD_STRIP` (2).

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies the primitive or primitives that will be created from vertices presented between `gl.Begin()` and the subsequent gl.End() (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is set to an unaccepted value.

`#GL_INVALID_OPERATION` is generated if `gl.Begin()` is executed between a `gl.Begin()` and the corresponding execution of `gl.End()`.

`#GL_INVALID_OPERATION` is generated if `gl.End()` is executed without being preceded by a glBegin.

`#GL_INVALID_OPERATION` is generated if an unsupported command is executed between the execution of `gl.Begin()` and the corresponding execution `gl.End()`. See your OpenGL reference manual for commands that can be executed between `gl.Begin()` and `gl.End()`.

6.6 gl.BindTexture

NAME

gl.BindTexture – bind a named texture to a texturing target

SYNOPSIS

`gl.BindTexture(target, texture)`

FUNCTION

`gl.BindTexture()` lets you create or use a named texture. Calling `gl.BindTexture()` with target set to `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D` and texture set to the name of the new texture binds the texture name to the target. When a texture is bound to a target, the previous binding for that target is automatically broken.

Texture names are unsigned integers. The value zero is reserved to represent the default texture for each texture target. Texture names and the corresponding texture contents are local to the shared display-list space of the current GL rendering context; two rendering contexts share texture names only if they also share display lists.

You may use `gl.GenTextures()` to generate a set of new texture names.

When a texture is first bound, it assumes the specified target: A texture first bound to `#GL_TEXTURE_1D` becomes one-dimensional texture, a texture first bound to `#GL_TEXTURE_2D` becomes two-dimensional texture. The state of a one-dimensional texture immediately after it is first bound is equivalent to the state of the default `#GL_TEXTURE_1D` at GL initialization, and similarly for two-dimensional textures.

While a texture is bound, GL operations on the target to which it is bound affect the bound texture, and queries of the target to which it is bound return state from the bound texture. If texture mapping is active on the target to which a texture is bound, the bound texture is used. In effect, the texture targets become aliases for the textures currently bound to them, and the texture name zero refers to the default textures that were bound to them at initialization.

A texture binding created with `gl.BindTexture()` remains active until a different texture is bound to the same target, or until the bound texture is deleted with `gl.DeleteTextures()`.

Once created, a named texture may be re-bound to its same original target as often as needed. It is usually much faster to use `gl.BindTexture()` to bind an existing named texture to one of the texture targets than it is to reload the texture image using `gl.TexImage1D()` or `gl.TexImage2D()`. For additional control over performance, use `gl.PrioritizeTextures()`.

`gl.BindTexture()` is included in display lists.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the target to which the texture is bound. Must be either `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`

`texture` specifies the name of a texture

ERRORS

`#GL_INVALID_ENUM` is generated if `target` is not one of the allowable values.

`#GL_INVALID_OPERATION` is generated if texture was previously created with a target that doesn't match that of `target`.

`#GL_INVALID_OPERATION` is generated if `gl.BindTexture()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_TEXTURE_BINDING_1D`

`gl.Get()` with argument `#GL_TEXTURE_BINDING_2D`

6.7 gl.Bitmap

NAME

gl.Bitmap – draw a bitmap

SYNOPSIS

```
gl.Bitmap(xorig, yorig, xmove, ymove[, bitmapArray])
```

FUNCTION

A bitmap is a binary image. When drawn, the bitmap is positioned relative to the current raster position, and frame buffer pixels corresponding to 1's in the bitmap are written using the current raster color or index. Frame buffer pixels corresponding to 0's in the bitmap are not modified.

`gl.Bitmap()` takes up to five arguments. The first pair specifies the location of the bitmap origin relative to the lower left corner of the bitmap image. The second pair of arguments specifies x and y offsets to be added to the current raster position after the bitmap has been drawn. The final argument is a table containing pixel data of the bitmap image itself.

The bitmap image is interpreted like image data for the `gl.DrawPixels()` command, with the bitmap's width and height corresponding to the width and height arguments of that command, and with type set to `GL_BITMAP` and format set to `GL_COLOR_INDEX`. Modes specified using `gl.PixelStore()` affect the interpretation of bitmap image data; modes specified using `gl.PixelTransfer()` do not.

If the current raster position is invalid, `gl.Bitmap()` is ignored. Otherwise, the lower left corner of the bitmap image is positioned at the window coordinates xw = xr - xo and yw = yr - yo where (xr,yr) is the raster position and (xo,yo) is the bitmap origin. Fragments are then generated for each pixel corresponding to a 1 (one) in the bitmap image. These fragments are generated using the current raster z coordinate, color or color index, and current raster texture coordinates. They are then treated just as if they had been generated by a point, line, or polygon, including texture mapping, fogging, and all per-fragment operations such as alpha and depth testing.

After the bitmap has been drawn, the x and y coordinates of the current raster position are offset by `xmove` and `ymove`. No change is made to the z coordinate of the current raster position, or to the current raster color, texture coordinates, or index.

To set a valid raster position outside the viewport, first set a valid raster position inside the viewport, then call `gl.Bitmap()` without the bitmap parameter and with `xmove` and `ymove` set to the offsets of the new raster position. This technique is useful when panning an image around the viewport.

Please consult an OpenGL reference manual for more information.

INPUTS

`xorig` specify the location of the x origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.

`yorig` specify the location of the y origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.

`xmove` specify the x offset to be added to the current raster position after the bitmap is drawn

`ymove` specify the y offset to be added to the current raster position after the bitmap is drawn

`bitmapArray`
optional: table containing bitmap data

ERRORS

`#GL_INVALID_OPERATION` is generated if glBitmap is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` with argument `#GL_CURRENT_RASTER_COLOR`

`gl.Get()` with argument `#GL_CURRENT_RASTER_SECONDARY_COLOR`

`gl.Get()` with argument `#GL_CURRENT_RASTER_DISTANCE`

`gl.Get()` with argument `#GL_CURRENT_RASTER_INDEX`

`gl.Get()` with argument `#GL_CURRENT_RASTER_TEXTURE_COORDS`

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.8 gl.BlendFunc

NAME

gl.BlendFunc – specify pixel arithmetic

SYNOPSIS

```
gl.BlendFunc(sfactor, dfactor)
```

FUNCTION

In RGBA mode, pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use `gl.Enable()` and `gl.Disable()` with argument `#GL_BLEND` to enable and disable blending.

`gl.BlendFunc()` defines the operation of blending when it is enabled. `sfactor` specifies which of nine methods is used to scale the source color components. `dfactor` specifies which of eight methods is used to scale the destination color components. The eleven possible methods are described in the following table. Each method defines four scale factors, one each for red, green, blue, and alpha.

In the table and in subsequent equations, source and destination color components are referred to as (Rs, Gs, Bs, As) and (Rd, Gd, Bd, Ad). They are understood to have integer values between 0 and (kR, kG, kB, kA), where

```
kc = 2^mc - 1
```

and (mR, mG, mB, mA) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as (sR, sG, sB, sA) and (dR, dG, dB, dA). The scale factors described in the table, denoted (fR, fG, fB, fA), represent either source or destination factors. All scale factors have range [0,1].

```
Parameter                |                (fR, fG, fB, fA)
-------------------------------------------------------------------------
#GL_ZERO                  |                  (0, 0, 0, 0)
#GL_ONE                   |                  (1, 1, 1, 1)
#GL_SRC_COLOR             |           (Rs/kR, Gs/kG, Bs/kB, As/kA)
#GL_ONE_MINUS_SRC_COLOR | (1, 1, 1, 1) - (Rs/kR, Gs/kG, Bs/kB, As/kA)
#GL_DST_COLOR             |          (Rd/kR, Gd/kG, Bd/kB, Ad/kA)
#GL_ONE_MINUS_DST_COLOR | (1, 1, 1, 1) - (Rd/kR, Gd/kG, Bd/kB, Ad/kA)
#GL_SRC_ALPHA             |          (As/kA, As/kA, As/kA, As/kA)
#GL_ONE_MINUS_SRC_ALPHA | (1, 1, 1, 1) - (As/kA, As/kA, As/kA, As/kA)
#GL_DST_ALPHA             |          (Ad/kA, Ad/kA, Ad/kA, Ad/kA)
#GL_ONE_MINUS_DST_ALPHA | (1, 1, 1, 1) - (Ad/kA, Ad/kA, Ad/kA, Ad/kA)
#GL_SRC_ALPHA_SATURATE  |                  (i, i, i, 1)
```

In the table,

```
i = min(As, kA  - Ad) / kA
```

To determine the blended RGBA values of a pixel when drawing in RGBA mode, the system uses the following equations:

```
Rd = min(kR, Rs sR + Rd dR)
Gd = min(kG, Gs sG + Gd dG)
Bd = min(kB, Bs sB + Bd dB)
Ad = min(kA, As sA + Ad dA)
```

Despite the apparent precision of the above equations, blending arithmetic is not exactly specified, because blending operates with imprecise integer color values. However, a blend factor that should be equal to 1 is guaranteed not to modify its multiplicand, and a blend factor equal to 0 reduces its multiplicand to 0. For example, when `sfactor` is `#GL_SRC_ALPHA`, `dfactor` is `#GL_ONE_MINUS_SRC_ALPHA`, and As is equal to kA, the equations reduce to simple replacement:

```
Rd = Rs
Gd = Gs
Bd = Bs
Ad = As
```

Transparency is best implemented using blend function (`#GL_SRC_ALPHA`, `#GL_ONE_MINUS_SRC_ALPHA`) with primitives sorted from farthest to nearest. Note that this transparency calculation does not require the presence of alpha bitplanes in the frame buffer. Blend function (`#GL_SRC_ALPHA`, `#GL_ONE_MINUS_SRC_ALPHA`) is also useful for rendering antialiased points and lines in arbitrary order.

Polygon antialiasing is optimized using blend function (`#GL_SRC_ALPHA_SATURATE`, `#GL_ONE`) with polygons sorted from nearest to farthest. (See Section 6.40 [gl.Enable], page 66, for information on polygon antialiasing. Look for `#GL_POLYGON_SMOOTH`) Destination alpha bitplanes, which must be present for this blend function to operate correctly, store the accumulated coverage.

Incoming (source) alpha is correctly thought of as a material opacity, ranging from 1.0 (KA), representing complete opacity, to 0.0 (0), representing complete transparency.

When more than one color buffer is enabled for drawing, the GL performs blending separately for each enabled buffer, using the contents of that buffer for destination color. (See Section 6.34 [gl.DrawBuffer], page 57, for details.)

Blending affects only RGBA rendering. It is ignored by color index renderers.

Please consult an OpenGL reference manual for more information.

INPUTS

`sfactor` specifies how the red, green, blue, and alpha source blending factors are computed (see above)

`dfactor` specifies how the red, green, blue, and alpha destination blending factors are computed (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if either `sfactor` or `dfactor` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.BlendFunc()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_BLEND_SRC`

`gl.Get()` with argument `#GL_BLEND_DST`

`gl.IsEnabled()` with argument `#GL_BLEND`

6.9 gl.CallList

NAME

gl.CallList – execute a display list

SYNOPSIS

```
gl.CallList(list)
```

FUNCTION

`gl.CallList()` causes the named display list to be executed. The commands saved in the display list are executed in order, just as if they were called without using a display list. If list has not been defined as a display list, `gl.CallList()` is ignored.

`gl.CallList()` can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, a limit is placed on the nesting level of display lists during display-list execution. This limit is at least 64, and it depends on the implementation.

GL state is not saved and restored across a call to `gl.CallList()`. Thus, changes made to GL state during the execution of a display list remain after execution of the display list is completed. Use `gl.PushAttrib()`, `gl.PopAttrib()`, `gl.PushMatrix()`, and `gl.PopMatrix()` to preserve GL state across `gl.CallList()` calls.

Display lists can be executed between a call to glBegin and the corresponding call to glEnd, as long as the display list includes only commands that are allowed in this interval.

Please consult an OpenGL reference manual for more information.

INPUTS

`list` specifies the integer name of the display list to be executed

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MAX_LIST_NESTING`

6.10 gl.CallLists

NAME

gl.CallLists – execute a list of display lists

SYNOPSIS

`gl.CallLists(listArray)`

FUNCTION

`gl.CallLists()` causes each display list in the list of names passed as lists to be executed. As a result, the commands saved in each display list are executed in order, just as if they were called without using a display list. Names of display lists that have not been defined are ignored.

`gl.CallLists()` provides an efficient means for executing more than one display list.

An additional level of indirection is made available with the `gl.ListBase()` command, which specifies an unsigned offset that is added to each display-list name specified in lists before that display list is executed.

`gl.CallLists()` can appear inside a display list. To avoid the possibility of infinite recursion resulting from display lists calling one another, a limit is placed on the nesting level of display lists during display-list execution. This limit is at least 64, and it depends on the implementation.

GL state is not saved and restored across a call to `gl.CallLists()`. Thus, changes made to GL state during the execution of a display list remain after execution of the display list is completed. Use `gl.PushAttrib()`, `gl.PopAttrib()`, `gl.PushMatrix()`, and `gl.PopMatrix()` to preserve GL state across `gl.CallLists()` calls.

Please consult an OpenGL reference manual for more information.

INPUTS

`listArray`
specifies an array of name offsets in the display list

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LIST_BASE`

`gl.Get()` with argument `#GL_MAX_LIST_NESTING`

6.11 gl.Clear

NAME

gl.Clear – clear buffers to preset values

SYNOPSIS

`gl.Clear(mask)`

FUNCTION

`gl.Clear()` sets the bitplane area of the window to values previously selected by `gl.ClearColor()`, `gl.ClearIndex()`, `gl.ClearDepth()`, `gl.ClearStencil()`, and `gl.ClearAccum()`. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using `gl.DrawBuffer()`.

The pixel ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of `gl.Clear()`. The scissor box bounds the cleared region. Alpha function, blend function, logical operation, stenciling, texture mapping, and depth-buffering are ignored by `gl.Clear()`.

`gl.Clear()` takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are as follows:

`#GL_COLOR_BUFFER_BIT`
: Indicates the buffers currently enabled for color writing.

`#GL_DEPTH_BUFFER_BIT`
: Indicates the depth buffer.

`#GL_ACCUM_BUFFER_BIT`
: Indicates the accumulation buffer.

`#GL_STENCIL_BUFFER_BIT`
: Indicates the stencil buffer.

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

If a buffer is not present, then a `gl.Clear()` directed at that buffer has no effect.

Please consult an OpenGL reference manual for more information.

INPUTS

`mask` bitwise OR of masks that indicate the buffers to be cleared. The four masks are `#GL_COLOR_BUFFER_BIT`, `#GL_DEPTH_BUFFER_BIT`, `#GL_ACCUM_BUFFER_BIT`, and `#GL_STENCIL_BUFFER_BIT`

ERRORS

`#GL_INVALID_VALUE` is generated if any bit other than the four defined bits is set in mask.

`#GL_INVALID_OPERATION` is generated if `gl.Clear()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ACCUM_CLEAR_VALUE`

`gl.Get()` with argument `#GL_DEPTH_CLEAR_VALUE`

`gl.Get()` with argument `#GL_INDEX_CLEAR_VALUE`

`gl.Get()` with argument `#GL_COLOR_CLEAR_VALUE`

`gl.Get()` with argument `#GL_STENCIL_CLEAR_VALUE`

6.12 gl.ClearAccum

NAME

gl.ClearAccum – specify clear values for the accumulation buffer

SYNOPSIS

`gl.ClearAccum(red, green, blue, alpha)`

FUNCTION

`gl.ClearAccum()` specifies the red, green, blue, and alpha values used by `gl.Clear()` to clear the accumulation buffer.

Values specified by `gl.ClearAccum()` are clamped to the range -1 through 1.

Please consult an OpenGL reference manual for more information.

INPUTS

`red` specify the red value used when the accumulation buffer is cleared; the initial value is 0

`green` specify the green value used when the accumulation buffer is cleared; the initial value is 0

`blue` specify the blue value used when the accumulation buffer is cleared; the initial value is 0

`alpha` specify the alpha value used when the accumulation buffer is cleared; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ClearAccum()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ACCUM_CLEAR_VALUE`

6.13 gl.ClearColor

NAME

gl.ClearColor – specify clear values for the color buffers

SYNOPSIS

`gl.ClearColor(red, green, blue, alpha)`

FUNCTION

`gl.ClearColor()` specifies the red, green, blue, and alpha values used by `gl.Clear()` to clear the color buffers. Values specified by `gl.ClearColor()` are clamped to the range 0 through 1.

Please consult an OpenGL reference manual for more information.

INPUTS

`red` specify the red value used when the color buffers are cleared; the initial value is 0

`green` specify the green value used when the color buffers are cleared; the initial value is 0

`blue` specify the blue value used when the color buffers are cleared; the initial value is 0

`alpha` specify the alpha value used when the color buffers are cleared; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ClearColor()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_COLOR_CLEAR_VALUE`

6.14 gl.ClearDepth

NAME

gl.ClearDepth – specify the clear value for the depth buffer

SYNOPSIS

`gl.ClearDepth(depth)`

FUNCTION

`gl.ClearDepth()` specifies the depth value used by `gl.Clear()` to clear the depth buffer. Values specified by `gl.ClearDepth()` are clamped to the range 0 through 1.

Please consult an OpenGL reference manual for more information.

INPUTS

`depth` specifies the depth value used when the depth buffer is cleared; the initial value is 1

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ClearDepth()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_DEPTH_CLEAR_VALUE`

6.15 gl.ClearIndex

NAME

gl.ClearIndex – specify the clear value for the color index buffers

SYNOPSIS

`gl.ClearIndex(c)`

FUNCTION

`gl.ClearIndex()` specifies the index used by `gl.Clear()` to clear the color index buffers. `c` is not clamped. Rather, `c` is converted to a fixed-point value with unspecified precision

to the right of the binary point. The integer part of this value is then masked with 2^m-1 , where m is the number of bits in a color index stored in the frame buffer.

Please consult an OpenGL reference manual for more information.

INPUTS

`c` specifies the index used when the color index buffers are cleared; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ClearIndex()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_INDEX_CLEAR_VALUE`

`gl.Get()` with argument `#GL_INDEX_BITS`

6.16 gl.ClearStencil

NAME

gl.ClearStencil – specify the clear value for the stencil buffer

SYNOPSIS

```
gl.ClearStencil(s)
```

FUNCTION

`gl.ClearStencil()` specifies the index used by `gl.Clear()` to clear the stencil buffer. `s` is masked with 2^m-1 , where m is the number of bits in the stencil buffer.

Please consult an OpenGL reference manual for more information.

INPUTS

`s` specifies the index used when the stencil buffer is cleared; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ClearStencil()` is executed between the execution of glBegin and the corresponding execution of glEnd

ASSOCIATED GETS

`gl.Get()` with argument `#GL_STENCIL_CLEAR_VALUE`

`gl.Get()` with argument `#GL_STENCIL_BITS`

6.17 gl.ClipPlane

NAME

gl.ClipPlane – specify a plane against which all geometry is clipped

SYNOPSIS

```
gl.ClipPlane(plane, equationArray)
```

FUNCTION

Geometry is always clipped against the boundaries of a six-plane frustum in x, y, and z. `gl.ClipPlane()` allows the specification of additional planes, not necessarily perpendicular to the x, y, or z axis, against which all geometry is clipped. To determine the maximum number of additional clipping planes, call `gl.Get()` with argument `#GL_MAX_CLIP_PLANES`. All implementations support at least six such clipping planes. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

`gl.ClipPlane()` specifies a half-space using a four-component plane equation. When `gl.ClipPlane()` is called, equation is transformed by the inverse of the modelview matrix and stored in the resulting eye coordinates. Subsequent changes to the modelview matrix have no effect on the stored plane-equation components. If the dot product of the eye coordinates of a vertex with the stored plane equation components is positive or zero, the vertex is in with respect to that clipping plane. Otherwise, it is out.

To enable and disable clipping planes, call `gl.Enable()` and `gl.Disable()` with the argument `#GL_CLIP_PLANEi`, where i is the plane number.

All clipping planes are initially defined as (0, 0, 0, 0) in eye coordinates and are disabled.

It is always the case that `#GL_CLIP_PLANEi` = `#GL_CLIP_PLANE0 +` i.

Please consult an OpenGL reference manual for more information.

INPUTS

`plane` specifies which clipping plane is being positioned; symbolic names of the form `#GL_CLIP_PLANEi`, where i is an integer between 0 and `#GL_MAX_CLIP_PLANES` -1 , are accepted

`equationArray`
specifies an array of four double-precision floating-point values; these values are interpreted as a plane equation

ERRORS

`#GL_INVALID_ENUM` is generated if plane is not an accepted value

`#GL_INVALID_OPERATION` is generated if `gl.ClipPlane()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.GetClipPlane()`

`gl.IsEnabled()` with argument `#GL_CLIP_PLANEi`

6.18 gl.Color

NAME

gl.Color – set the current color

SYNOPSIS

```
gl.Color(red, green, blue[, alpha])
```

FUNCTION

The GL stores both a current single-valued color index and a current four-valued RGBA color. `gl.Color()` sets a new four-valued RGBA color. If the optional alpha argument is omitted, it will be set to 1.0.

Current color values are stored in floating-point format such that the largest representable value maps to 1.0 (full intensity), and 0 maps to 0.0 (zero intensity).

Alternatively, you can also pass a table containing three or four floating-point values specifying the red, green, blue, and alpha values for the color.

The initial value for the current color is (1, 1, 1, 1).

The current color can be updated at any time. In particular, `gl.Color()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`red`	specify new red value for the current color
`green`	specify new green value for the current color
`blue`	specify new blue value for the current color
`alpha`	optional: specify new alpha value for the current color (defaults to 1.0)

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_COLOR`

`gl.Get()` with argument `#GL_RGBA_MODE`

6.19 gl.ColorMask

NAME

gl.ColorMask – enable and disable writing of frame buffer color components

SYNOPSIS

```
gl.ColorMask(red, green, blue, alpha)
```

FUNCTION

`gl.ColorMask()` specifies whether the individual color components in the frame buffer can or cannot be written. If red is `#GL_FALSE`, for example, no change is made to the red component of any pixel in any of the color buffers, regardless of the drawing operation attempted. The initial values are all `#GL_TRUE`, indicating that the color components can be written.

Changes to individual bits of components cannot be controlled. Rather, changes are either enabled or disabled for entire color components.

Please consult an OpenGL reference manual for more information.

INPUTS

`red`	specify whether red can or cannot be written into the frame buffer
`green`	specify whether green can or cannot be written into the frame buffer

`blue` specify whether blue can or cannot be written into the frame buffer

`alpha` specify whether alpha can or cannot be written into the frame buffer

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ColorMask()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_COLOR_WRITEMASK`

`gl.Get()` with argument `#GL_RGBA_MODE`

6.20 gl.ColorMaterial

NAME

gl.ColorMaterial – cause a material color to track the current color

SYNOPSIS

```
gl.ColorMaterial(face, mode)
```

FUNCTION

`gl.ColorMaterial()` specifies which material parameters track the current color. When `#GL_COLOR_MATERIAL` is enabled, the material parameter or parameters specified by `mode`, of the material or materials specified by `face`, track the current color at all times. `face` can be set to `#GL_FRONT`, `#GL_BACK`, or `#GL_FRONT_AND_BACK`. The initial value is `#GL_FRONT_AND_BACK`.

The following values can be passed in the `mode` parameter: `#GL_EMISSION`, `#GL_AMBIENT`, `#GL_DIFFUSE`, `#GL_SPECULAR`, and `#GL_AMBIENT_AND_DIFFUSE`. The initial value is `#GL_AMBIENT_AND_DIFFUSE`.

To enable and disable `#GL_COLOR_MATERIAL`, call `gl.Enable()` and `gl.Disable()` with argument `#GL_COLOR_MATERIAL`. `#GL_COLOR_MATERIAL` is initially disabled.

`gl.ColorMaterial()` makes it possible to change a subset of material parameters for each vertex using only the `gl.Color()` command, without calling `gl.Material()`. If only such a subset of parameters is to be specified for each vertex, calling `gl.ColorMaterial()` is preferable to calling `gl.Material()`.

Call `gl.ColorMaterial()` before enabling `#GL_COLOR_MATERIAL`.

Calling `gl.DrawElements()` or `gl.DrawArrays()` may leave the current color indeterminate, if the color array is enabled. If `gl.ColorMaterial()` is enabled while the current color is indeterminate, the lighting material state specified by face and mode is also indeterminate.

If the GL version is 1.1 or greater, and `#GL_COLOR_MATERIAL` is enabled, evaluated color values affect the results of the lighting equation as if the current color were being modified, but no change is made to the tracking lighting parameter of the current color.

Please consult an OpenGL reference manual for more information.

INPUTS

`face` specifies whether front, back, or both front and back material parameters should track the current color

`mode` specifies which of several material parameters track the current color (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if face or mode is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.ColorMaterial()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_COLOR_MATERIAL`

`gl.Get()` with argument `#GL_COLOR_MATERIAL_PARAMETER`

`gl.Get()` with argument `#GL_COLOR_MATERIAL_FACE`

6.21 gl.ColorPointer

NAME

gl.ColorPointer – define an array of colors

SYNOPSIS

`gl.ColorPointer(colorArray[, size])`

FUNCTION

`gl.ColorPointer()` specifies an array of color components to use when rendering. `colorArray` can be either a one-dimensional table consisting of an arbitrary number of consecutive color values or a two-dimensional table consisting of an arbitrary number of subtables which contain 3 or 4 color values each. If `colorArray` is a one-dimensional table, you need to pass the optional `size` argument as well to define the size of each color component in `colorArray`. `size` must be either 3 or 4. If `colorArray` is a two-dimensional table, `size` is automatically determined by the number of items in the first subtable, which must be either three or four as well.

When using a two-dimensional table, please keep in mind that the number of color values in each subtable must be constant. It is not allowed to use differing numbers of color values in the individual subtables. The number of color values is defined by the number of elements in the first subtable and all following subtables must use the very same number of color values.

If you pass `Nil` in `colorArray`, the color array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the color array or defining a new one.

To enable and disable the color array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_COLOR_ARRAY`. If enabled, the color array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

The color array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.ColorPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.ColorPointer()` is typically implemented on the client side.

Color array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`colorArray` one- or two-dimensional table containing color values or `Nil` (see above)

`size` optional: size of each color component; must be either 3 or 4 and is only used with one-dimensional tables (see above)

ERRORS

`#GL_INVALID_VALUE` is generated if size is not 3 or 4.

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_COLOR_ARRAY`

`gl.Get()` with argument `#GL_COLOR_ARRAY_SIZE`

`gl.Get()` with argument `#GL_COLOR_ARRAY_TYPE`

`gl.Get()` with argument `#GL_COLOR_ARRAY_STRIDE`

`gl.Get()` with argument `#GL_COLOR_ARRAY_POINTER`

6.22 gl.CopyPixels

NAME

gl.CopyPixels – copy pixels in the frame buffer

SYNOPSIS

`gl.CopyPixels(x, y, width, height, type)`

FUNCTION

`gl.CopyPixels()` copies a screen-aligned rectangle of pixels from the specified frame buffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware dependent and undefined.

`x` and `y` specify the window coordinates of the lower left corner of the rectangular region to be copied. `width` and `height` specify the dimensions of the rectangular region to be copied. Both `width` and `height` must not be negative.

Several parameters control the processing of the pixel data while it is being copied. These parameters are set with three commands: `gl.PixelTransfer()`, `gl.PixelMap()`, and `gl.PixelZoom()`. This reference page describes the effects on `gl.CopyPixels()` of most, but not all, of the parameters specified by these three commands.

`gl.CopyPixels()` copies values from each pixel with the lower left-hand corner at (x+i,y+j) for 0 <= i < width and 0 <= j < height. This pixel is said to be the ith pixel in

the jth row. Pixels are copied in row order from the lowest to the highest row, left to right in each row.

`type` specifies whether color, depth, or stencil data is to be copied. The details of the transfer for each data type are as follows:

`#GL_COLOR`

Indices or RGBA colors are read from the buffer currently specified as the read source buffer (See Section 6.121 [gl.ReadBuffer], page 172, for details.). If the GL is in color index mode, each index that is read from this buffer is converted to a fixed-point format with an unspecified number of bits to the right of the binary point. Each index is then shifted left by `#GL_INDEX_SHIFT` bits, and added to `#GL_INDEX_OFFSET`. If `#GL_INDEX_SHIFT` is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If `#GL_MAP_COLOR` is true, the index is replaced with the value that it references in lookup table `#GL_PIXEL_MAP_I_TO_I`. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b-1 , where b is the number of bits in a color index buffer.

If the GL is in RGBA mode, the red, green, blue, and alpha components of each pixel that is read are converted to an internal floating-point format with unspecified precision. The conversion maps the largest representable component value to 1.0, and component value 0 to 0.0. The resulting floating-point color values are then multiplied by `#GL_c_SCALE` and added to `#GL_c_BIAS`, where c is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1]. If `#GL_MAP_COLOR` is true, each color component is scaled by the size of lookup table `#GL_PIXEL_MAP_c_TO_c`, then replaced by the value that it references in that table. c is R, G, B, or A.

If the ARB_imaging extension is supported, the color values may be additionally processed by color-table lookups, color-matrix transformations, and convolution filters.

The GL then converts the resulting indices or RGBA colors to fragments by attaching the current raster position z coordinate and texture coordinates to each pixel, then assigning window coordinates (xr+i,yr+j) , where (xr,yr) is the current raster position, and the pixel was the ith pixel in the jth row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

`#GL_DEPTH`

Depth values are read from the depth buffer and converted directly to an internal floating-point format with unspecified precision. The resulting floating-point depth value is then multiplied by `#GL_DEPTH_SCALE` and added to `#GL_DEPTH_BIAS`. The result is clamped to the range [0,1].

The GL then converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordi-

nates to each pixel, then assigning window coordinates (xr+i,yr+j), where (xr,yr) is the current raster position, and the pixel was the ith pixel in the jth row. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

`#GL_STENCIL`

Stencil indices are read from the stencil buffer and converted to an internal fixed-point format with an unspecified number of bits to the right of the binary point. Each fixed-point index is then shifted left by `#GL_INDEX_SHIFT` bits, and added to `#GL_INDEX_OFFSET`. If `#GL_INDEX_SHIFT` is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If `#GL_MAP_STENCIL` is true, the index is replaced with the value that it references in lookup table `#GL_PIXEL_MAP_S_TO_S`. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b-1 , where b is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the index read from the ith location of the jth row is written to location (xr+i,yr+j), where (xr,yr) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these write operations.

The rasterization described thus far assumes pixel zoom factors of 1.0. If `gl.PixelZoom()` is used to change the x and y pixel zoom factors, pixels are converted to fragments as follows. If (xr,yr) is the current raster position, and a given pixel is in the ith location in the jth row of the source pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

```
(xr + zoomx_i, yr + zoomy_j)
```

and

```
(xr + zoomx_(i + 1), yr + zoomy_(j + 1))
```

where `zoomx` is the value of `#GL_ZOOM_X` and `zoomy` is the value of `#GL_ZOOM_Y`.

Modes specified by `gl.PixelStore()` have no effect on the operation of the command `gl.CopyPixels()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`x` specify the x coordinate of the lower left corner of the rectangular region of pixels to be copied

`y` specify the y coordinate of the lower left corner of the rectangular region of pixels to be copied

`width` specify the dimensions of the rectangular region of pixels to be copied; both must be nonnegative

`height` specify the dimensions of the rectangular region of pixels to be copied; both must be nonnegative

`type` specifies whether color values, depth values, or stencil values are to be copied; symbolic constants `#GL_COLOR`, `#GL_DEPTH`, and `#GL_STENCIL` are accepted

ERRORS

`#GL_INVALID_ENUM` is generated if type is not an accepted value.

`#GL_INVALID_VALUE` is generated if either width or height is negative.

`#GL_INVALID_OPERATION` is generated if type is `#GL_DEPTH` and there is no depth buffer.

`#GL_INVALID_OPERATION` is generated if type is `#GL_STENCIL` and there is no stencil buffer.

`#GL_INVALID_OPERATION` is generated if `gl.CopyPixels()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.23 gl.CopyTexImage

NAME

gl.CopyTexImage – copy pixels into a texture image

SYNOPSIS

`gl.CopyTexImage(level, internalFormat, border, x, y, width[, height])`

FUNCTION

`gl.CopyTexImage()` defines a one- or two-dimensional texture image with pixels from the current `#GL_READ_BUFFER`. If the optional `height` argument is omitted, a one-dimensional texture will be defined, otherwise a two-dimensional texture will be defined.

The screen-aligned pixel rectangle with lower left corner at (x, y) and with a width of width+2*border and a height of height+2*border defines the texture array at the mipmap level specified by `level`.

`internalformat` specifies the internal format of the texture array. See Section 3.12 [Internal pixel formats], page 13, for details. Note that in contrast to `gl.TexImage1D()` and `gl.TexImage2D()` the values 1, 2, 3, and 4 are not supported by the `internalFormat` parameter with `gl.CopyTexImage()`.

The pixels in the rectangle are processed exactly as if `gl.CopyPixels()` had been called, but the process stops just before final conversion. At this point all pixel component values are clamped to the range [0,1] and then converted to the texture's internal format for storage in the texel array.

Pixel ordering is such that lower x and y screen coordinates correspond to lower s and t texture coordinates.

If any of the pixels within the specified rectangle of the current `#GL_READ_BUFFER` are outside the window associated with the current rendering context, then the values obtained for those pixels are undefined.

Texturing has no effect in color index mode.

An image with height or width of 0 indicates a `NULL` texture.

Please consult an OpenGL reference manual for more information.

INPUTS

`level` specifies the level-of-detail number. Level 0 is the base image level. Level n is the nth mipmap reduction image

`internalFormat` specifies the internal format of the texture; must be one of the pixel format constants (see above)

`border` specifies the width of the border; must be either 0 or 1

`x` specify the x coordinate of the lower left corner of the rectangular region of pixels to be copied

`y` specify the y coordinate of the lower left corner of the rectangular region of pixels to be copied

`width` specifies the width of the texture image. Must be 0 or 2^n+2*border for some integer n

`height` optional: specifies the height of the texture image. Must be 0 or 2^n+2*border for some integer n (defaults to 1)

ERRORS

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than log2(max), where `max` is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if `internalformat` is not an allowable value.

`#GL_INVALID_VALUE` is generated if `width` is less than 0 or greater than 2 + `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if non-power-of-two textures are not supported and the width cannot be represented as 2^n+2*border for some integer value of `n`.

`#GL_INVALID_VALUE` is generated if border is not 0 or 1.

`#GL_INVALID_OPERATION` is generated if `gl.CopyTexImage()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.GetTexImage()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_2D` or `#GL_TEXTURE_1D`

6.24 gl.CopyTexSubImage

NAME

gl.CopyTexSubImage – copy a two-dimensional texture subimage

SYNOPSIS

`gl.CopyTexSubImage(level, x, y, xoffset, width[, yoffset, height])`

FUNCTION

`gl.CopyTexSubImage()` replaces a rectangular portion of a one- or two-dimensional texture image with pixels from the current `#GL_READ_BUFFER` (rather than from main memory, as is the case for `gl.TexSubImage2D()`). If the last two arguments are omitted, a rectangular portion of a one-dimension texture image is replaced, otherwise a two-dimensional texture image is the target.

The screen-aligned pixel rectangle with lower left corner at (x,y) and with width `width` and height `height` replaces the portion of the texture array with `x` indices `xoffset` through `xoffset + width - 1`, inclusive, and `y` indices `yoffset` through `yoffset + height - 1`, inclusive, at the mipmap level specified by `level`.

The pixels in the rectangle are processed exactly as if `gl.CopyPixels()` had been called, but the process stops just before final conversion. At this point, all pixel component values are clamped to the range [0,1] and then converted to the texture's internal format for storage in the texel array.

The destination rectangle in the texture array may not include any texels outside the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

If any of the pixels within the specified rectangle of the current `#GL_READ_BUFFER` are outside the read window associated with the current rendering context, then the values obtained for those pixels are undefined.

No change is made to the `internalformat`, `width`, `height`, or `border` parameters of the specified texture array or to texel values outside the specified subregion.

Texturing has no effect in color index mode.

`gl.PixelStore()` and `gl.PixelTransfer()` modes affect texture images in exactly the way they affect `gl.DrawPixels()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`level`	specifies the level-of-detail number; level 0 is the base image level; level n is the nth mipmap reduction image
`x`	specify the x coordinate of the lower left corner of the rectangular region of pixels to be copied
`y`	specify the y coordinate of the lower left corner of the rectangular region of pixels to be copied
`xoffset`	specifies a texel offset in the x direction within the texture array
`width`	specifies the width of the texture subimage
`yoffset`	optional: specifies a texel offset in the y direction within the texture array
`height`	optional: specifies the height of the texture subimage

ERRORS

`#GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous `gl.TexImage2D()` or `gl.CopyTexImage()` operation.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

#GL_INVALID_VALUE may be generated if level > log2(max), where max is the returned value of #GL_MAX_TEXTURE_SIZE.

#GL_INVALID_VALUE is generated if xoffset < -b , xoffset + width > w - b , yoffset < -b , or yoffset + height > h -b , where w is the #GL_TEXTURE_WIDTH, h is the #GL_TEXTURE_HEIGHT, and b is the #GL_TEXTURE_BORDER of the texture image being modified. Note that w and h include twice the border width.

#GL_INVALID_OPERATION is generated if gl.CopyTexSubImage() is executed between the execution of gl.Begin() and the corresponding execution of gl.End()

ASSOCIATED GETS

gl.GetTexImage()

gl.IsEnabled() with argument #GL_TEXTURE_2D or #GL_TEXTURE_1D

6.25 gl.CullFace

NAME

gl.CullFace – specify whether front- or back-facing facets can be culled

SYNOPSIS

gl.CullFace(mode)

FUNCTION

gl.CullFace() specifies whether front- or back-facing facets are culled (as specified by mode) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the gl.Enable() and gl.Disable() commands with the argument #GL_CULL_FACE. Facets include triangles, quadrilaterals, polygons, and rectangles.

gl.FrontFace() specifies which of the clockwise and counterclockwise facets are front-facing and back-facing. See Section 6.53 [gl.FrontFace], page 77, for details.

If mode is #GL_FRONT_AND_BACK, no facets are drawn, but other primitives such as points and lines are drawn.

Please consult an OpenGL reference manual for more information.

INPUTS

mode — specifies whether front- or back-facing facets are candidates for culling; symbolic constants #GL_FRONT, #GL_BACK, and #GL_FRONT_AND_BACK are accepted; the initial value is #GL_BACK

ERRORS

#GL_INVALID_ENUM is generated if mode is not an accepted value.

#GL_INVALID_OPERATION is generated if gl.CullFace() is executed between the execution of gl.Begin() and the corresponding execution of gl.End()

ASSOCIATED GETS

gl.IsEnabled() with argument #GL_CULL_FACE

gl.Get() with argument #GL_CULL_FACE_MODE

6.26 gl.DeleteLists

NAME

gl.DeleteLists – delete a contiguous group of display lists

SYNOPSIS

`gl.DeleteLists(list, range)`

FUNCTION

`gl.DeleteLists()` causes a contiguous group of display lists to be deleted. list is the name of the first display list to be deleted, and range is the number of display lists to delete. All display lists d with list <= d <= list + range - 1 are deleted.

All storage locations allocated to the specified display lists are freed, and the names are available for reuse at a later time. Names within the range that do not have an associated display list are ignored. If range is 0, nothing happens.

Please consult an OpenGL reference manual for more information.

INPUTS

`list` specifies the integer name of the first display list to delete

`range` specifies the number of display lists to delete

ERRORS

`#GL_INVALID_VALUE` is generated if range is negative.

`#GL_INVALID_OPERATION` is generated if `gl.DeleteLists()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

6.27 gl.DeleteTextures

NAME

gl.DeleteTextures – delete named textures

SYNOPSIS

`gl.DeleteTextures(texturesArray)`

FUNCTION

`gl.DeleteTextures()` deletes all textures passed in the table `texturesArray`. After a texture is deleted, it has no contents or dimensionality, and its name is free for reuse (for example by `gl.GenTextures()`). If a texture that is currently bound is deleted, the binding reverts to 0 (the default texture).

`gl.DeleteTextures()` silently ignores 0's and names that do not correspond to existing textures.

Please consult an OpenGL reference manual for more information.

INPUTS

`texturesArray`
specifies an array of textures to be deleted

ERRORS

`#GL_INVALID_OPERATION` is generated if glDeleteTextures is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS
`gl.IsTexture()`

6.28 gl.DepthFunc

NAME
gl.DepthFunc – specify the value used for depth buffer comparisons

SYNOPSIS
`gl.DepthFunc(func)`

FUNCTION
`gl.DepthFunc()` specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See `gl.Enable()` and `gl.Disable()` of `#GL_DEPTH_TEST`)

`func` specifies the conditions under which the pixel will be drawn. The comparison functions are as follows:

`#GL_NEVER`
: Never passes.

`#GL_LESS` Passes if the incoming depth value is less than the stored depth value.

`#GL_EQUAL`
: Passes if the incoming depth value is equal to the stored depth value.

`#GL_LEQUAL`
: Passes if the incoming depth value is less than or equal to the stored depth value.

`#GL_GREATER`
: Passes if the incoming depth value is greater than the stored depth value.

`#GL_NOTEQUAL`
: Passes if the incoming depth value is not equal to the stored depth value.

`#GL_GEQUAL`
: Passes if the incoming depth value is greater than or equal to the stored depth value.

`#GL_ALWAYS`
: Always passes.

The initial value of func is `#GL_LESS`. Initially, depth testing is disabled. If depth testing is disabled or if no depth buffer exists, it is as if the depth test always passes.

Even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled.

Please consult an OpenGL reference manual for more information.

INPUTS
`func` specifies the depth comparison function (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `func` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.DepthFunc()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_DEPTH_FUNC`

`gl.IsEnabled()` with argument `#GL_DEPTH_TEST`

6.29 gl.DepthMask

NAME

gl.DepthMask – enable or disable writing into the depth buffer

SYNOPSIS

```
gl.DepthMask(flag)
```

FUNCTION

`gl.DepthMask()` specifies whether the depth buffer is enabled for writing. If flag is `#GL_FALSE`, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Please consult an OpenGL reference manual for more information.

INPUTS

`flag` specifies whether the depth buffer is enabled for writing; if flag is `#GL_FALSE`, depth buffer writing is disabled, otherwise, it is enabled; initially, depth buffer writing is enabled

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.DepthMask()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_DEPTH_WRITEMASK`

6.30 gl.DepthRange

NAME

gl.DepthRange – specify mapping of depth values from normalized device coordinates to window coordinates

SYNOPSIS

```
gl.DepthRange(zNear, zFar)
```

FUNCTION

After clipping and division by w, depth coordinates range from -1 to 1, corresponding to the near and far clipping planes. `gl.DepthRange()` specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of

the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). Thus, the values accepted by `gl.DepthRange()` are both clamped to this range before they are accepted.

The setting of (0,1) maps the near plane to 0 and the far plane to 1. With this mapping, the depth buffer range is fully utilized.

It is not necessary that `nearVal` be less than `farVal`. Reverse mappings such as nearVal = 1 , and farVal = 0 are acceptable.

Please consult an OpenGL reference manual for more information.

INPUTS

`zNear` specifies the mapping of the near clipping plane to window coordinates; the initial value is 0

`zFar` specifies the mapping of the far clipping plane to window coordinates; the initial value is 1

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.DepthRange()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_DEPTH_RANGE`

6.31 gl.Disable

NAME

gl.Disable – disable server-side GL capabilities

SYNOPSIS

```
gl.Disable(cap)
```

FUNCTION

`gl.Disable()` disables various capabilities. Use `gl.IsEnabled()` or `gl.Get()` to determine the current setting of any capability. The initial value for each capability with the exception of `#GL_DITHER` is `#GL_FALSE`. The initial value for `#GL_DITHER` is `#GL_TRUE`.

`gl.Disable()` takes a single argument, `cap`, which can assume one of the following values:

`#GL_ALPHA_TEST`
If enabled, do alpha testing. See Section 6.2 [gl.AlphaFunc], page 20, for details..

`#GL_AUTO_NORMAL`
If enabled, generate normal vectors when either `#GL_MAP2_VERTEX_3` or `#GL_MAP2_VERTEX_4` is used to generate vertices. See Section 6.93 [gl.Map], page 133, for details.

`#GL_BLEND`
If enabled, blend the computed fragment color values with the values in the color buffers. See See Section 6.8 [gl.BlendFunc], page 28, for details.

`#GL_CLIP_PLANEi`
: If enabled, clip geometry against user-defined clipping plane i. See See Section 6.17 [gl.ClipPlane], page 35, for details.

`#GL_COLOR_LOGIC_OP`
: If enabled, apply the currently selected logical operation to the computed fragment color and color buffer values. See See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_COLOR_MATERIAL`
: If enabled, have one or more material parameters track the current color. See See Section 6.20 [gl.ColorMaterial], page 38, for details.

`#GL_CULL_FACE`
: If enabled, cull polygons based on their winding in window coordinates. See See Section 6.25 [gl.CullFace], page 46, for details.

`#GL_DEPTH_TEST`
: If enabled, do depth comparisons and update the depth buffer. Note that even if the depth buffer exists and the depth mask is non-zero, the depth buffer is not updated if the depth test is disabled. See See Section 6.28 [gl.DepthFunc], page 48, for details. and See Section 6.30 [gl.DepthRange], page 49, for details.

`#GL_DITHER`
: If enabled, dither color components or indices before they are written to the color buffer.

`#GL_FOG` If enabled and no fragment shader is active, blend a fog color into the post-texturing color. See See Section 6.50 [gl.Fog], page 75, for details.

`#GL_INDEX_LOGIC_OP`
: If enabled, apply the currently selected logical operation to the incoming index and color buffer indices. See See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_LIGHTi`
: If enabled, include light i in the evaluation of the lighting equation. See See Section 6.85 [gl.LightModel], page 125, for details. and See Section 6.84 [gl.Light], page 123, for details.

`#GL_LIGHTING`
: If enabled and no vertex shader is active, use the current lighting parameters to compute the vertex color or index. Otherwise, simply associate the current color or index with each vertex. See Section 6.95 [gl.Material], page 138, for details. See Section 6.85 [gl.LightModel], page 125, for details. See Section 6.84 [gl.Light], page 123, for details.

`#GL_LINE_SMOOTH`
: If enabled, draw lines with correct filtering. Otherwise, draw aliased lines. See Section 6.87 [gl.LineWidth], page 128, for details.

`#GL_LINE_STIPPLE`
: If enabled, use the current line stipple pattern when drawing lines. See Section 6.86 [gl.LineStipple], page 127, for details.

`#GL_MAP1_COLOR_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate RGBA values. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_INDEX`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate color indices. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_NORMAL`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate normals. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_1`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_2`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s and t texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_3`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s, t, and r texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s, t, r, and q texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_VERTEX_3`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate x, y, and z vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_VERTEX_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate homogeneous x, y, z, and w vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_COLOR_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate RGBA values. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_INDEX`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate color indices. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_NORMAL`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate normals. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_1`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_2`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s and t texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_3`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s, t, and r texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate s, t, r, and q texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_VERTEX_3`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate x, y, and z vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_VERTEX_4`
: If enabled, calls to `gl.EvalCoord()`, `gl.EvalMesh()`, and `gl.EvalPoint()` generate homogeneous x, y, z, and w vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_NORMALIZE`
: If enabled and no vertex shader is active, normal vectors are normalized to unit length after transformation and before lighting. See Section 6.99 [gl.Normal], page 142, for details. See Section 6.100 [gl.NormalPointer], page 143, for details.

`#GL_POINT_SMOOTH`
: If enabled, draw points with proper filtering. Otherwise, draw aliased points. See Section 6.107 [gl.PointSize], page 155, for details.

`#GL_POLYGON_OFFSET_FILL`
: If enabled, and if the polygon is rendered in `#GL_FILL` mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See Section 6.109 [gl.PolygonOffset], page 157, for details.

`#GL_POLYGON_OFFSET_LINE`
: If enabled, and if the polygon is rendered in `#GL_LINE` mode, an offset is added to depth values of a polygon's fragments before the depth comparison is performed. See Section 6.109 [gl.PolygonOffset], page 157, for details.

`#GL_POLYGON_OFFSET_POINT`
: If enabled, an offset is added to depth values of a polygon's fragments before the depth comparison is performed, if the polygon is rendered in GL_POINT mode. See glPolygonOffset.

`#GL_POLYGON_SMOOTH`
: If enabled, draw polygons with proper filtering. Otherwise, draw aliased polygons. For correct antialiased polygons, an alpha buffer is needed and the polygons must be sorted front to back.

`#GL_POLYGON_STIPPLE`
: If enabled, use the current polygon stipple pattern when rendering polygons. See Section 6.110 [gl.PolygonStipple], page 158, for details.

`#GL_SCISSOR_TEST`
: If enabled, discard fragments that are outside the scissor rectangle. See Section 6.128 [gl.Scissor], page 180, for details.

`#GL_STENCIL_TEST`
: If enabled, do stencil testing and update the stencil buffer. See Section 6.131 [gl.StencilFunc], page 184, for details. See Section 6.133 [gl.StencilOp], page 186, for details.

`#GL_TEXTURE_1D`
: If enabled and no fragment shader is active, one-dimensional texturing is performed (unless two- or three-dimensional or cube-mapped texturing is also enabled). See Section 6.139 [gl.TexImage1D], page 192, for details.

`#GL_TEXTURE_2D`
: If enabled and no fragment shader is active, two-dimensional texturing is performed (unless three-dimensional or cube-mapped texturing is also enabled). See Section 6.140 [gl.TexImage2D], page 196, for details.

`#GL_TEXTURE_GEN_Q`
: If enabled and no vertex shader is active, the q texture coordinate is computed using the texture generation function defined with `gl.TexGen()`. Otherwise, the current q texture coordinate is used. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_R`
: If enabled and no vertex shader is active, the r texture coordinate is computed using the texture generation function defined with `gl.TexGen()`. Otherwise, the current r texture coordinate is used. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_S`
: If enabled and no vertex shader is active, the s texture coordinate is computed using the texture generation function defined with `gl.TexGen()`. Otherwise, the current s texture coordinate is used. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_T`
: If enabled and no vertex shader is active, the t texture coordinate is computed using the texture generation function defined with `gl.TexGen()`. Otherwise, the current t texture coordinate is used. See Section 6.137 [gl.TexGen], page 190, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`cap` specifies a symbolic constant indicating a GL capability

ERRORS

`#GL_INVALID_ENUM` is generated if `cap` is not one of the values listed previously.

`#GL_INVALID_OPERATION` is generated if `gl.Enable()` or `gl.Disable()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.IsEnabled()`

`gl.Get()`

6.32 gl.DisableClientState

NAME

gl.DisableClientState – disable client-side capability

SYNOPSIS

`gl.DisableClientState(cap)`

FUNCTION

`gl.DisableClientState()` disables individual client-side capabilities. By default, all client-side capabilities are disabled. `gl.DisableClientState()` takes a single argument, `cap`, which can assume one of the following values:

`#GL_COLOR_ARRAY`
: If enabled, the color array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.21 [gl.ColorPointer], page 39, for details.

`#GL_EDGE_FLAG_ARRAY`
: If enabled, the edge flag array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.39 [gl.EdgeFlagPointer], page 65, for details.

`#GL_INDEX_ARRAY`
: If enabled, the index array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.78 [gl.IndexPointer], page 116, for details.

`#GL_NORMAL_ARRAY`
: If enabled, the normal array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.100 [gl.NormalPointer], page 143, for details.

`#GL_TEXTURE_COORD_ARRAY`
: If enabled, the texture coordinate array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.135 [gl.TexCoordPointer], page 188, for details.

`#GL_VERTEX_ARRAY`
: If enabled, the vertex array is enabled for writing and used during rendering when `gl.ArrayElement()`, `gl.DrawArrays()`, or `gl.DrawElements()` is called. See Section 6.147 [gl.VertexPointer], page 207, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`array` specifies the capability to disable (see above for supported constants)

ERRORS

`#GL_INVALID_ENUM` is generated if `cap` is not an accepted value.

`gl.DisableClientState()` is not allowed between the execution of `gl.Begin()` and the corresponding `gl.End()`, but an error may or may not be generated. If no error is generated, the behavior is undefined.

6.33 gl.DrawArrays

NAME

gl.DrawArrays – render primitives from array data

SYNOPSIS

```
gl.DrawArrays(mode, first, count)
```

FUNCTION

`gl.DrawArrays()` specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to `gl.DrawArrays()`.

When `gl.DrawArrays()` is called, it uses `count` sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element first. `mode` specifies what kind of primitives are constructed and how the array elements construct those primitives. If `#GL_VERTEX_ARRAY` is not enabled, no geometric primitives are generated. `mode` can be set to the symbolic constants `#GL_POINTS`, `#GL_LINE_STRIP`, `#GL_LINE_LOOP`, `#GL_LINES`, `#GL_TRIANGLE_STRIP`, `#GL_TRIANGLE_FAN`, `#GL_TRIANGLES`, `#GL_QUAD_STRIP`, `#GL_QUADS`, or `#GL_POLYGON`.

Vertex attributes that are modified by `gl.DrawArrays()` have an unspecified value after `gl.DrawArrays()` returns. For example, if `#GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `gl.DrawArrays()` executes. Attributes that aren't modified remain well defined.

`gl.DrawArrays()` is included in display lists. If `gl.DrawArrays()` is entered into a display list, the necessary array data (determined by the array pointers and enables) is

also entered into the display list. Because the array pointers and enables are client-side state, their values affect display lists when the lists are created, not when the lists are executed.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies what kind of primitives to render (see above)

`first` specifies the starting index in the enabled arrays

`count` specifies the number of indices to be rendered

ERRORS

`#GL_INVALID_ENUM` is generated if mode is not an accepted value.

`#GL_INVALID_VALUE` is generated if count is negative.

`#GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array and the buffer object's data store is currently mapped.

`#GL_INVALID_OPERATION` is generated if glDrawArrays is executed between the execution of glBegin and the corresponding glEnd.

6.34 gl.DrawBuffer

NAME

gl.DrawBuffer – specify which color buffers are to be drawn into

SYNOPSIS

```
gl.DrawBuffer(mode)
```

FUNCTION

When colors are written to the frame buffer, they are written into the color buffers specified by `gl.DrawBuffer()`. The following constants can be passed in `mode`:

`#GL_NONE` No color buffers are written.

`#GL_FRONT_LEFT`
Only the front left color buffer is written.

`#GL_FRONT_RIGHT`
Only the front right color buffer is written.

`#GL_BACK_LEFT`
Only the back left color buffer is written.

`#GL_BACK_RIGHT`
Only the back right color buffer is written.

`#GL_FRONT`
Only the front left and front right color buffers are written. If there is no front right color buffer, only the front left color buffer is written.

`#GL_BACK` Only the back left and back right color buffers are written. If there is no back right color buffer, only the back left color buffer is written.

`#GL_LEFT` Only the front left and back left color buffers are written. If there is no back left color buffer, only the front left color buffer is written.

`#GL_RIGHT`
Only the front right and back right color buffers are written. If there is no back right color buffer, only the front right color buffer is written.

`#GL_FRONT_AND_BACK`
All the front and back color buffers (front left, front right, back left, back right) are written. If there are no back color buffers, only the front left and front right color buffers are written. If there are no right color buffers, only the front left and back left color buffers are written. If there are no right or back color buffers, only the front left color buffer is written.

`#GL_AUXi` Only auxiliary color buffer `i` is written where `i` is between 0 and the value of `#GL_AUX_BUFFERS` minus 1. Note that `#GL_AUX_BUFFERS` is not the upper limit; use `gl.Get()` to query the number of available aux buffers. It is always the case that `#GL_AUXi` = `#GL_AUX0 + i`.

If more than one color buffer is selected for drawing, then blending or logical operations are computed and applied independently for each color buffer and can produce different results in each buffer.

Monoscopic contexts include only left buffers, and stereoscopic contexts include both left and right buffers. Likewise, single-buffered contexts include only front buffers, and double-buffered contexts include both front and back buffers. The context is selected at GL initialization.

The initial value is `#GL_FRONT` for single-buffered contexts, and `#GL_BACK` for double-buffered contexts.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies up to four color buffers to be drawn into (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if mode is not an accepted value.

`#GL_INVALID_OPERATION` is generated if none of the buffers indicated by mode exists.

`#GL_INVALID_OPERATION` is generated if `gl.DrawBuffer()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_DRAW_BUFFER`

`gl.Get()` with argument `#GL_AUX_BUFFERS`

6.35 gl.DrawElements

NAME

gl.DrawElements – render primitives from array data

SYNOPSIS

`gl.DrawElements(mode, indicesArray)`

FUNCTION

`gl.DrawElements()` specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and so on, and use them to construct a sequence of primitives with a single call to `gl.DrawElements()`.

When `gl.DrawElements()` is called, it reads sequential elements from an enabled array and constructs a sequence of geometric primitives. `mode` specifies what kind of primitives are constructed and how the array elements construct these primitives. `mode` can be set to the symbolic constants `#GL_POINTS`, `#GL_LINE_STRIP`, `#GL_LINE_LOOP`, `#GL_LINES`, `#GL_TRIANGLE_STRIP`, `#GL_TRIANGLE_FAN`, `#GL_TRIANGLES`, `#GL_QUAD_STRIP`, `#GL_QUADS`, and `#GL_POLYGON`. If more than one array is enabled, each is used. If `#GL_VERTEX_ARRAY` is not enabled, no geometric primitives are constructed.

Vertex attributes that are modified by `gl.DrawElements()` have an unspecified value after `gl.DrawElements()` returns. For example, if `#GL_COLOR_ARRAY` is enabled, the value of the current color is undefined after `gl.DrawElements()` executes. Attributes that aren't modified maintain their previous values.

`gl.DrawElements()` is included in display lists. If `gl.DrawElements()` is entered into a display list, the necessary array data (determined by the array pointers and enables) is also entered into the display list. Because the array pointers and enables are client-side state, their values affect display lists when the lists are created, not when the lists are executed.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies what kind of primitives to render (see above)

`indicesArray`
specifies an array where the indices are stored; the indices in this array are treated as values of type `#GL_UNSIGNED_INT`

ERRORS

`#GL_INVALID_ENUM` is generated if mode is not an accepted value.

`#GL_INVALID_OPERATION` is generated if a non-zero buffer object name is bound to an enabled array or the element array and the buffer object's data store is currently mapped.

`#GL_INVALID_OPERATION` is generated if `gl.DrawElements()` is executed between the execution of glBegin and the corresponding glEnd.

6.36 gl.DrawPixels

NAME

gl.DrawPixels – write a block of pixels to the frame buffer

SYNOPSIS

`gl.DrawPixels(width, height, format, pixelsArray)`

FUNCTION

This function does the same as `gl.DrawPixelsRaw()` except that the pixel data is not passed as a raw memory buffer but as a table containing width*height number of elements describing a pixel each. This is of course not as efficient as using raw memory buffers because the table's pixel data has to be copied to a raw memory buffer first.

Note that `gl.DrawPixels()` expects data of type `#GL_FLOAT` inside the `pixelsArray` table.

See Section 6.37 [gl.DrawPixelsRaw], page 60, for more details on the parameters accepted by this function.

Please consult an OpenGL reference manual for more information.

INPUTS

`width` specify the width of the pixel rectangle to be written into the frame buffer

`height` specify the height of the pixel rectangle to be written into the frame buffer

`format` specifies the format of the pixel data (see above for supported formats)

`pixelsArray`
specifies an array containing the pixel data; data in this array is treated as `#GL_FLOAT`

6.37 gl.DrawPixelsRaw

NAME

gl.DrawPixelsRaw – write a block of pixels to the frame buffer

SYNOPSIS

```
gl.DrawPixelsRaw(width, height, format, type, pixels)
```

FUNCTION

`gl.DrawPixelsRaw()` reads pixel data from memory and writes it into the frame buffer relative to the current raster position, provided that the raster position is valid. Use `gl.RasterPos()` to set the current raster position; use `gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION_VALID` to determine if the specified raster position is valid, and `gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION` to query the raster position.

Several parameters define the encoding of pixel data in memory and control the processing of the pixel data before it is placed in the frame buffer. These parameters are set with four commands: `gl.PixelStore()`, `gl.PixelTransfer()`, `gl.PixelMap()`, and `gl.PixelZoom()`. This reference page describes the effects on `gl.DrawPixelsRaw()` of many, but not all, of the parameters specified by these four commands.

Data is read from `pixels` as a sequence of signed or unsigned bytes, signed or unsigned shorts, signed or unsigned integers, or single-precision floating-point values, depending on `type` which can be `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`. Each of these bytes, shorts, integers, or floating-point values is interpreted as one color or depth component, or one index, depending on format. Indices are always treated individually. Color components

are treated as groups of one, two, three, or four values, again based on `format`. Both individual indices and groups of components are referred to as pixels. If type is `#GL_BITMAP`, the data must be unsigned bytes, and format must be either `#GL_COLOR_INDEX` or `#GL_STENCIL_INDEX`. Each unsigned byte is treated as eight 1-bit pixels, with bit ordering determined by `#GL_UNPACK_LSB_FIRST` (See Section 6.104 [gl.PixelStore], page 148, for details.).

`width` * `height` pixels are read from memory, starting at location `pixels`. By default, these pixels are taken from adjacent memory locations, except that after all width pixels are read, the read pointer is advanced to the next four-byte boundary. The four-byte row alignment is specified by `gl.PixelStore()` with argument `#GL_UNPACK_ALIGNMENT`, and it can be set to one, two, four, or eight bytes. Other pixel store parameters specify different read pointer advancements, both before the first pixel is read and after all width pixels are read. See Section 6.104 [gl.PixelStore], page 148, for details.

The `width` * `height` pixels that are read from memory are each operated on in the same way, based on the values of several parameters specified by `gl.PixelTransfer()` and `gl.PixelMap()`. The details of these operations, as well as the target buffer into which the pixels are drawn, are specific to the format of the pixels, as specified by `format`. `format` can assume one of 13 symbolic values:

`#GL_COLOR_INDEX`

Each pixel is a single value, a color index. It is converted to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values.

Each fixed-point index is then shifted left by `#GL_INDEX_SHIFT` bits and added to `#GL_INDEX_OFFSET`. If `#GL_INDEX_SHIFT` is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result.

If the GL is in RGBA mode, the resulting index is converted to an RGBA pixel with the help of the `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, and `#GL_PIXEL_MAP_I_TO_A` tables. If the GL is in color index mode, and if `#GL_MAP_COLOR` is true, the index is replaced with the value that it references in lookup table `#GL_PIXEL_MAP_I_TO_I`. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b-1, where `b` is the number of bits in a color index buffer.

The GL then converts the resulting indices or RGBA colors to fragments by attaching the current raster position z coordinate and texture coordinates to each pixel, then assigning x and y window coordinates to the nth fragment such that

```
xn = xr + n % width
yn = yr + n / width
```

where (xr,yr) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

#GL_STENCIL_INDEX
: Each pixel is a single value, a stencil index. It is converted to fixed-point format, with an unspecified number of bits to the right of the binary point, regardless of the memory data type. Floating-point values convert to true fixed-point values.

  Each fixed-point index is then shifted left by #GL_INDEX_SHIFT bits, and added to #GL_INDEX_OFFSET. If #GL_INDEX_SHIFT is negative, the shift is to the right. In either case, zero bits fill otherwise unspecified bit locations in the result. If #GL_MAP_STENCIL is true, the index is replaced with the value that it references in lookup table #GL_PIXEL_MAP_S_TO_S. Whether the lookup replacement of the index is done or not, the integer part of the index is then ANDed with 2^b-1, where b is the number of bits in the stencil buffer. The resulting stencil indices are then written to the stencil buffer such that the nth index is written to location

```
xn = xr + n % width
yn = yr + n / width
```

  where (xr,yr) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these write operations.

#GL_DEPTH_COMPONENT
: Each pixel is a single-depth component. Floating-point data is converted directly to an internal floating-point format with unspecified precision. The resulting floating-point depth value is then multiplied by #GL_DEPTH_SCALE and added to #GL_DEPTH_BIAS. The result is clamped to the range [0,1].

  The GL then converts the resulting depth components to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning x and y window coordinates to the nth fragment such that

```
xn = xr + n % width
yn = yr + n / width
```

  where (xr,yr) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

#GL_RGBA
: Each pixel is a four-component group: For #GL_RGBA, the red component is first, followed by green, followed by blue, followed by alpha. Floating-point values are converted directly to an internal floating-point format with unspecified precision. The resulting floating-point color values are then multiplied by #GL_c_SCALE and added to #GL_c_BIAS, where c is RED, GREEN, BLUE, and ALPHA for the respective color components. The results are clamped to the range [0,1].

  If #GL_MAP_COLOR is true, each color component is scaled by the size of lookup table #GL_PIXEL_MAP_c_TO_c, then replaced by the value that it references in that table. c is R, G, B, or A respectively.

The GL then converts the resulting RGBA colors to fragments by attaching the current raster position z coordinate and texture coordinates to each pixel, then assigning x and y window coordinates to the nth fragment such that

```
xn = xr + n % width
yn = yr + n / width
```

where (xr,yr) is the current raster position. These pixel fragments are then treated just like the fragments generated by rasterizing points, lines, or polygons. Texture mapping, fog, and all the fragment operations are applied before the fragments are written to the frame buffer.

`#GL_RED` Each pixel is a single red component. This component is converted to the internal floating-point format in the same way the red component of an RGBA pixel is. It is then converted to an RGBA pixel with green and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_GREEN`
Each pixel is a single green component. This component is converted to the internal floating-point format in the same way the green component of an RGBA pixel is. It is then converted to an RGBA pixel with red and blue set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_BLUE` Each pixel is a single blue component. This component is converted to the internal floating-point format in the same way the blue component of an RGBA pixel is. It is then converted to an RGBA pixel with red and green set to 0, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_ALPHA`
Each pixel is a single alpha component. This component is converted to the internal floating-point format in the same way the alpha component of an RGBA pixel is. It is then converted to an RGBA pixel with red, green, and blue set to 0. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_RGB` Each pixel is a three-component group: red first, followed by green, followed by blue. Each component is converted to the internal floating-point format in the same way the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_LUMINANCE`
Each pixel is a single luminance component. This component is converted to the internal floating-point format in the same way the red component of an RGBA pixel is. It is then converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

`#GL_LUMINANCE_ALPHA`
: Each pixel is a two-component group: luminance first, followed by alpha. The two components are converted to the internal floating-point format in the same way the red component of an RGBA pixel is. They are then converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to the converted alpha value. After this conversion, the pixel is treated as if it had been read as an RGBA pixel.

The rasterization described so far assumes pixel zoom factors of 1. If `gl.PixelZoom()` is used to change the x and y pixel zoom factors, pixels are converted to fragments as follows. If (xr,yr) is the current raster position, and a given pixel is in the nth column and mth row of the pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

```
(xr + zoomx_n, yr + zoomy_m)
```

and

```
(xr + zoomx_(n + 1), yr + zoomy_(m + 1))
```

where `zoomx` is the value of `#GL_ZOOM_X` and `zoomy` is the value of `#GL_ZOOM_Y`.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 6.36 [gl.DrawPixels], page 59, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`width`	specifies the width of the pixel rectangle to be written into the frame buffer
`height`	specifies the height of the pixel rectangle to be written into the frame buffer
`format`	specifies the format of the pixel data (see above for supported formats)
`type`	specifies the data type of the pixel data (see above)
`pixels`	specifies a pointer to the pixel data

ERRORS

`#GL_INVALID_ENUM` is generated if `format` or `type` is not one of the accepted values.

`#GL_INVALID_ENUM` is generated if `type` is `#GL_BITMAP` and `format` is not either `#GL_COLOR_INDEX` or `#GL_STENCIL_INDEX`.

`#GL_INVALID_VALUE` is generated if either `width` or `height` is negative.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_STENCIL_INDEX` and there is no stencil buffer.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, or `#GL_LUMINANCE_ALPHA`, and the GL is in color index mode.

`#GL_INVALID_OPERATION` is generated if `gl.DrawPixelsRaw()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.38 gl.EdgeFlag

NAME

gl.EdgeFlag – flag edges as either boundary or nonboundary

SYNOPSIS

`gl.EdgeFlag(flag)`

FUNCTION

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a `gl.Begin()` / `gl.End()` pair is marked as the start of either a boundary or nonboundary edge. If the current edge flag is true when the vertex is specified, the vertex is marked as the start of a boundary edge. Otherwise, the vertex is marked as the start of a nonboundary edge. `gl.EdgeFlag()` sets the edge flag bit to `#GL_TRUE` if flag is `#GL_TRUE` and to `#GL_FALSE` otherwise. The initial value is `#GL_TRUE`.

The vertices of connected triangles and connected quadrilaterals are always marked as boundary, regardless of the value of the edge flag.

Boundary and nonboundary edge flags on vertices are significant only if `#GL_POLYGON_MODE` is set to `#GL_POINT` or `#GL_LINE`. See Section 6.108 [gl.PolygonMode], page 156, for details.

The current edge flag can be updated at any time. In particular, `gl.EdgeFlag()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`flag` specifies the current edge flag value (either `#GL_TRUE` or `#GL_FALSE`)

ASSOCIATED GETS

`gl.Get()` with argument `#GL_EDGE_FLAG`

6.39 gl.EdgeFlagPointer

NAME

gl.EdgeFlagPointer – define an array of edge flags

SYNOPSIS

`gl.EdgeFlagPointer(flagsArray)`

FUNCTION

`gl.EdgeFlagPointer()` specifies an array of boolean edge flags to use when rendering. If you pass `Nil` in `flagsArray`, the edge flag array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the edge flag array or defining a new one.

When an edge flag array is specified, it is saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the edge flag array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_EDGE_FLAG_ARRAY`. If enabled, the edge flag array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Edge flags are not supported for interleaved vertex array formats. See Section 6.80 [gl.InterleavedArrays], page 117, for details.

The edge flag array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.EdgeFlagPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.EdgeFlagPointer()` is typically implemented on the client side.

Edge flag array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`flagsArray`
: specifies a table containing an array of edge flags or `Nil`

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_EDGE_FLAG_ARRAY`

`gl.Get()` with argument `#GL_EDGE_FLAG_ARRAY_POINTER`

6.40 gl.Enable

NAME

gl.Enable – enable server-side GL capabilities

SYNOPSIS

`gl.Enable(cap)`

FUNCTION

`gl.Enable()` enables various capabilities. Use `gl.IsEnabled()` or `gl.Get()` to determine the current setting of any capability. The initial value for each capability with the exception of `#GL_DITHER` is `#GL_FALSE`. The initial value for `#GL_DITHER` is `#GL_TRUE`.

See Section 6.31 [gl.Disable], page 50, for a list of supported capabilities.

Please consult an OpenGL reference manual for more information.

INPUTS

`cap` specifies a symbolic constant indicating a GL capability

6.41 gl.EnableClientState

NAME
gl.EnableClientState – enable client-side capability

SYNOPSIS
`gl.EnableClientState(cap)`

FUNCTION
`gl.EnableClientState()` enables individual client-side capabilities. By default, all client-side capabilities are disabled. `gl.EnableClientState()` takes a single argument, `cap`. See Section 6.32 [gl.DisableClientState], page 55, for a list of supported capabilities.

Please consult an OpenGL reference manual for more information.

INPUTS
`cap` specifies the capability to enable

6.42 gl.End

NAME
gl.End – delimit the vertices of a primitive or a group of like primitives

SYNOPSIS
`gl.End()`

FUNCTION
See Section 6.5 [gl.Begin], page 23, for details.

INPUTS
none

ERRORS
ASSOCIATED GETS

6.43 gl.EndList

NAME
gl.EndList – replace a display list

SYNOPSIS
`gl.EndList()`

FUNCTION
See Section 6.98 [gl.NewList], page 141, for details.

Please consult an OpenGL reference manual for more information.

INPUTS
none

ERRORS
`#GL_INVALID_OPERATION` is generated if `gl.EndList()` is called without a preceding `gl.NewList()`

6.44 gl.EvalCoord

NAME

gl.EvalCoord – evaluate enabled one- and two-dimensional maps

SYNOPSIS

```
gl.EvalCoord(u[, v])
```

FUNCTION

`gl.EvalCoord()` evaluates enabled one- or two-dimensional maps at argument `u` or `u` and `v`. To define a map, call `gl.Map()`; to enable and disable it, call `gl.Enable()` and `gl.Disable()`.

When the `gl.EvalCoord()` command is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if the corresponding GL command had been issued with the computed value. That is, if `#GL_MAP1_INDEX` or `#GL_MAP2_INDEX` is enabled, a `gl.Index()` command is simulated. If `#GL_MAP1_COLOR_4` or `#GL_MAP2_COLOR_4` is enabled, a `gl.Color()` command is simulated. If `#GL_MAP1_NORMAL` or `#GL_MAP2_NORMAL` is enabled, a normal vector is produced, and if any of the constants `#GL_MAP1_TEXTURE_COORD_1`, `#GL_MAP1_TEXTURE_COORD_2`, `#GL_MAP1_TEXTURE_COORD_3`, `#GL_MAP1_TEXTURE_COORD_4`, or also the constants `#GL_MAP2_TEXTURE_COORD_1`, `#GL_MAP2_TEXTURE_COORD_2`, `#GL_MAP2_TEXTURE_COORD_3`, or `#GL_MAP2_TEXTURE_COORD_4` is enabled, then the GL will simulate an appropriate `gl.TexCoord()` command.

For color, color index, normal, and texture coordinates the GL uses evaluated values instead of current values for those evaluations that are enabled, and current values otherwise, However, the evaluated values do not update the current values. Thus, if `gl.Vertex()` commands are interspersed with `gl.EvalCoord()` commands, the color, normal, and texture coordinates associated with the `gl.Vertex()` commands are not affected by the values generated by the `gl.EvalCoord()` commands, but only by the most recent `gl.Color()`, `gl.Index()`, `gl.Normal()`, and `gl.TexCoord()` commands.

No commands are issued for maps that are not enabled. If more than one texture evaluation is enabled for a particular dimension (for example, `#GL_MAP2_TEXTURE_COORD_1` and `#GL_MAP2_TEXTURE_COORD_2`), then only the evaluation of the map that produces the larger number of coordinates (in this case, `#GL_MAP2_TEXTURE_COORD_2`) is carried out. `#GL_MAP1_VERTEX_4` overrides `#GL_MAP1_VERTEX_3`, and `#GL_MAP2_VERTEX_4` overrides `#GL_MAP2_VERTEX_3`, in the same manner. If neither a three- nor a four-component vertex map is enabled for the specified dimension, the `gl.EvalCoord()` command is ignored.

If you have enabled automatic normal generation, by calling `gl.Enable()` with argument `#GL_AUTO_NORMAL`, `gl.EvalCoord()` generates surface normals analytically, regardless of the contents or enabling of the `#GL_MAP2_NORMAL` map. If automatic normal generation is disabled, the corresponding normal map `#GL_MAP2_NORMAL`, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map is enabled, no normal is generated for `gl.EvalCoord()` commands.

Alternatively, you can also pass a table containing one or two domain coordinates to `gl.EvalCoord()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`u` specifies a value that is the domain coordinate u to the basis function defined in a previous `gl.Map()` command

`v` optional: specifies a value that is the domain coordinate v to the basis function defined in a previous `gl.Map()` command

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_MAP1_VERTEX_3`

`gl.IsEnabled()` with argument `#GL_MAP1_VERTEX_4`

`gl.IsEnabled()` with argument `#GL_MAP1_INDEX`

`gl.IsEnabled()` with argument `#GL_MAP1_COLOR_4`

`gl.IsEnabled()` with argument `#GL_MAP1_NORMAL`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_1`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_2`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_3`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_4`

`gl.IsEnabled()` with argument `#GL_MAP2_VERTEX_3`

`gl.IsEnabled()` with argument `#GL_MAP2_VERTEX_4`

`gl.IsEnabled()` with argument `#GL_MAP2_INDEX`

`gl.IsEnabled()` with argument `#GL_MAP2_COLOR_4`

`gl.IsEnabled()` with argument `#GL_MAP2_NORMAL`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_1`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_2`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_3`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_4`

`gl.IsEnabled()` with argument `#GL_AUTO_NORMAL`

`gl.GetMap()`

6.45 gl.EvalMesh

NAME

gl.EvalMesh – compute a one- or two-dimensional grid of points or lines

SYNOPSIS

```
gl.EvalMesh(mode, i1, i2[, j1, j2])
```

FUNCTION

This function can be used to compute a one- or two-dimensional grid of points or lines. If you omit the last two parameters, a one-dimensional mesh is computed, otherwise a two-dimensional will be computed.

`gl.MapGrid()` and `gl.EvalMesh()` are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. `gl.EvalMesh()` steps through the

integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by `gl.Map()`. `mode` determines whether the resulting vertices are connected as points, lines, or filled polygons (the latter is only supported for two-dimensional grids). In the one-dimensional case, `gl.EvalMesh()`, the mesh is generated as if the following code fragment were executed:

```
gl.Begin(type)
For Local i = i1 To i2 Do gl.EvalCoord(i*du+u1)
gl.End()
```

where du = (u2-u1)/n and n, u1, and u2 are the arguments to the most recent `gl.MapGrid()` command. `type` is `#GL_POINTS` if mode is `#GL_POINT`, or `#GL_LINES` if mode is `#GL_LINE`. The one absolute numeric requirement is that if i = n, then the value computed from i*du+u1 is exactly u2.

In the two-dimensional case, `gl.EvalMesh()`, let

```
du = (u2-u1)/n
dv = (v2-v1)/m,
```

where n, u1, u2, m, v1, and v2 are the arguments to the most recent `gl.MapGrid()` command. Then, if mode is `#GL_FILL`, the `gl.EvalMesh()` command is equivalent to:

```
For Local j = j1 To j2 - 1
   gl.Begin(#GL_QUAD_STRIP)
   For Local i = i1 To i2
      gl.EvalCoord(i*du+u1, j*dv+v1)
      gl.EvalCoord(i*du+u1, (j+1)*dv+v1)
   Next
   gl.End()
Next
```

If mode is `#GL_LINE`, then a call to `gl.EvalMesh()` is equivalent to:

```
For Local j = j1 To j2
   gl.Begin(#GL_LINE_STRIP)
   For Local i = i1 To i2
      gl.EvalCoord(i*du+u1, j*dv+v1)
   Next
   gl.End()
Next

For Local i = i1 To i2
   gl.Begin(#GL_LINE_STRIP)
   For Local j = j1 To j2
      gl.EvalCoord(i*du+u1, j*dv+v1)
   Next
   gl.End()
Next
```

And finally, if mode is `#GL_POINT`, then a call to `gl.EvalMesh()` is equivalent to:

```
gl.Begin(#GL_POINTS)
For Local j = j1 To j2
   For Local i = i1 To i2
```

```
            gl.EvalCoord(i*du+u1, j*dv+v1)
         Next
      Next
      gl.End()
```

In all three cases, the only absolute.numeric requirements are that if i = n, then the value computed from i*du+u1 is exactly u2, and if j = m, then the value computed from j*dv+v1 is exactly v2.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode`	specifies whether to compute a mesh of points, lines or polygons; symbolic constants `#GL_POINT`, `#GL_LINE`, and in case of a two-dimensional mesh, `#GL_FILL` are accepted
`i1`	specify the first integer value for grid domain variable i
`i2`	specify the last integer value for grid domain variable i
`j1`	optional: specify the first integer value for grid domain variable j
`j2`	optional: specify the last integer value for grid domain variable j

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.EvalMesh()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` with argument `#GL_MAP2_GRID_SEGMENTS`

6.46 gl.EvalPoint

NAME

gl.EvalPoint – generate and evaluate a single point in a mesh

SYNOPSIS

`gl.EvalPoint(i[, j])`

FUNCTION

`gl.MapGrid()` and `gl.EvalMesh()` are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. `gl.EvalPoint()` can be used to evaluate a single grid point in the same gridspace that is traversed by `gl.EvalMesh()`. Calling `gl.EvalPoint()` with a single argument is equivalent to calling

```
gl.EvalCoord(i*du+u1)
```

where

```
du = (u2-u1)/n
```

and n, u1, and u2 are the arguments to the most recent `gl.MapGrid()` command. The one absolute numeric requirement is that if i = n, then the value computed from i*du+u1 is exactly u2.

In the two-dimensional case, `gl.EvalPoint()`, let

```
du = (u2-u1)/n
dv = (v2-v1)/m
```

where n, u1, u2, m, v1, and v2 are the arguments to the most recent `gl.MapGrid()` command. Then the `gl.EvalPoint()` command is equivalent to calling

```
gl.EvalCoord(i*du+u1, j*dv+v1)
```

The only absolute numeric requirements are that if i = n, then the value computed from i*du+u1 is exactly u2, and if j = m, then the value computed from j*dv+v1 is exactly v2.

Please consult an OpenGL reference manual for more information.

INPUTS

`i` specifies the integer value for grid domain variable i

`j` optional: specifies the integer value for grid domain variable j

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` with argument `#GL_MAP2_GRID_SEGMENTS`

6.47 gl.FeedbackBuffer

NAME

gl.FeedbackBuffer – controls feedback mode

SYNOPSIS

```
buffer = gl.FeedbackBuffer(size, type)
```

FUNCTION

The `gl.FeedbackBuffer()` function controls feedback. Feedback, like selection, is a GL mode. The mode is selected by calling `gl.RenderMode()` with `#GL_FEEDBACK`. When the GL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

`gl.FeedbackBuffer()` has two arguments: `size` indicates the size of the array that is to be returned, in items of `#GL_FLOAT`. `type` is a symbolic constant describing the information that is fed back for each vertex. `gl.FeedbackBuffer()` must be issued before feedback mode is enabled (by calling `gl.RenderMode()` with argument `#GL_FEEDBACK`). Setting `#GL_FEEDBACK` without establishing the feedback buffer, or calling `gl.FeedbackBuffer()` while the GL is in feedback mode, is an error.

When `gl.RenderMode()` is called while in feedback mode, it returns the number of entries placed in the feedback array and resets the feedback array pointer to the base of

the feedback buffer. The returned value never exceeds size. If the feedback data required more room than was available in buffer, `gl.RenderMode()` returns a negative value. To take the GL out of feedback mode, call `gl.RenderMode()` with a parameter value other than `#GL_FEEDBACK`.

While in feedback mode, each primitive, bitmap, or pixel rectangle that would be rasterized generates a block of values that are copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all), and an overflow flag is set. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling and `gl.PolygonMode()` interpretation of polygons has taken place, so polygons that are culled are not returned in the feedback buffer. It can also occur after polygons with more than three edges are broken up into triangles, if the GL implementation renders polygons by performing this decomposition.

The `gl.PassThrough()` command can be used to insert a marker into the feedback buffer. See Section 6.102 [gl.PassThrough], page 145, for details.

Following is the grammar for the blocks of values written into the feedback buffer. Each primitive is indicated with a unique identifying value followed by some number of vertices. Polygon entries include an integer value indicating how many vertices follow. A vertex is fed back as some number of floating-point values, as determined by type. Colors are fed back as four values in RGBA mode and one value in color index mode.

Feedback vertex coordinates are in window coordinates, except w, which is in clip coordinates. Feedback colors are lighted, if lighting is enabled. Feedback texture coordinates are generated, if texture coordinate generation is enabled. They are always transformed by the texture matrix

`gl.FeedbackBuffer()`, when used in a display list, is not compiled into the display list but is executed immediately.

Please note that `gl.FeedbackBuffer()` returns only the texture coordinate of texture unit `#GL_TEXTURE0`.

To free a buffer allocated by this function, call `gl.FreeFeedbackBuffer()`. See Section 6.51 [gl.FreeFeedbackBuffer], page 77, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`size` specifies the maximum number of values that will be returned

`type` specifies a symbolic constant that describes the information that will be returned for each vertex; `#GL_2D`, `#GL_3D`, `#GL_3D_COLOR`, `#GL_3D_COLOR_TEXTURE`, and `#GL_4D_COLOR_TEXTURE` are accepted

RESULTS

`buffer` pointer to feedback buffer

ERRORS

`#GL_INVALID_ENUM` is generated if `type` is not an accepted value.

`#GL_INVALID_VALUE` is generated if `size` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.FeedbackBuffer()` is called while the render mode is `#GL_FEEDBACK`, or if `gl.RenderMode()` is called with argument `#GL_FEEDBACK` before `gl.FeedbackBuffer()` is called at least once.

`#GL_INVALID_OPERATION` is generated if `gl.FeedbackBuffer()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_RENDER_MODE`

`gl.Get()` with argument `#GL_FEEDBACK_BUFFER_POINTER`

`gl.Get()` with argument `#GL_FEEDBACK_BUFFER_SIZE`

`gl.Get()` with argument `#GL_FEEDBACK_BUFFER_TYPE`

6.48 gl.Finish

NAME

gl.Finish – block until all GL execution is complete

SYNOPSIS

`gl.Finish()`

FUNCTION

`gl.Finish()` does not return until the effects of all previously called GL commands are complete. Such effects include all changes to GL state, all changes to connection state, and all changes to the frame buffer contents.

`gl.Finish()` requires a round trip to the server.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Finish()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.49 gl.Flush

NAME

gl.Flush – force execution of GL commands in finite time

SYNOPSIS

`gl.Flush()`

FUNCTION

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. glFlush empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any GL program might be executed over a network, or on an accelerator that buffers commands, all programs should call `gl.Flush()` whenever they count on having all of their previously issued commands completed. For example, call `gl.Flush()` before waiting for user input that depends on the generated image.

`gl.Flush()` can return at any time. It does not wait until the execution of all previously issued GL commands is complete.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Flush()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.50 gl.Fog

NAME

gl.Fog – specify fog parameters

SYNOPSIS

```
gl.Fog(pname, param)
```

FUNCTION

Fog is initially disabled. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer clear operations. To enable and disable fog, call `gl.Enable()` and `gl.Disable()` with argument `#GL_FOG`.

`gl.Fog()` assigns the value or values in params to the fog parameter specified by `pname`. The following values are accepted for pname:

`#GL_FOG_MODE`

> `param` is a single floating-point value that specifies the equation to be used to compute the fog blend factor, f. Three symbolic constants are accepted: `#GL_LINEAR`, `#GL_EXP`, and `#GL_EXP2`. The equations corresponding to these symbolic constants are defined below. The initial fog mode is `#GL_EXP`.

`#GL_FOG_DENSITY`

> `param` is a single floating-point value that specifies density, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The initial fog density is 1.

`#GL_FOG_START`

> `param` is a single floating-point value that specifies start, the near distance used in the linear fog equation. The initial near distance is 0.

`#GL_FOG_END`

> `param` is a single floating-point value that specifies end, the far distance used in the linear fog equation. The initial far distance is 1.

#GL_FOG_INDEX

param is a single floating-point value that specifies i f , the fog color index. The initial fog index is 0.

#GL_FOG_COLOR

param must be a table containing four floating-point values that specify Cf, the fog color. All color components are clamped to the range [0,1]. The initial fog color is (0, 0, 0, 0).

Fog blends a fog color with each rasterized pixel fragment's post-texturing color using a blending factor f. Factor f is computed in one of three ways, depending on the fog mode. Let c be the distance in eye coordinates from the origin to the fragment being fogged. The equation for #GL_LINEAR fog is

```
f = (end - c) / (end - start)
```

The equation for #GL_EXP fog is

```
f = e^(-density*c)
```

The equation for #GL_EXP2 fog is

```
f = e^(-density*c)^2
```

Regardless of the fog mode, f is clamped to the range [0,1] after it is computed. Then, if the GL is in RGBA color mode, the fragment's red, green, and blue colors, represented by Cr , are replaced by

```
Cr' = f*Cr+(1-f)*Cf
```

Fog does not affect a fragment's alpha component.

In color index mode, the fragment's color index ir is replaced by

```
ir' = f*ir+(1-f)*if
```

Please consult an OpenGL reference manual for more information.

INPUTS

pname specifies a single-valued fog parameter; #GL_FOG_MODE, #GL_FOG_DENSITY, #GL_FOG_START, #GL_FOG_END, #GL_FOG_INDEX, and #GL_FOG_COLOR are accepted

param specifies the value that pname will be set to

ERRORS

#GL_INVALID_ENUM is generated if pname is not an accepted value, or if pname is #GL_FOG_MODE and param is not an accepted value.

#GL_INVALID_VALUE is generated if pname is #GL_FOG_DENSITY and param is negative.

#GL_INVALID_OPERATION is generated if gl.Fog() is executed between the execution of gl.Begin() and the corresponding execution of gl.End() .

ASSOCIATED GETS

gl.IsEnabled() with argument #GL_FOG

gl.Get() with argument #GL_FOG_COLOR

gl.Get() with argument #GL_FOG_INDEX

gl.Get() with argument #GL_FOG_DENSITY

`gl.Get()` with argument `#GL_FOG_START`

`gl.Get()` with argument `#GL_FOG_END`

`gl.Get()` with argument `#GL_FOG_MODE`

6.51 gl.FreeFeedbackBuffer

NAME

gl.FreeFeedbackBuffer – free feedback mode buffer

SYNOPSIS

```
gl.FreeFeedbackBuffer(buffer)
```

FUNCTION

This function frees a buffer allocated by `gl.FeedbackBuffer()`. See Section 6.47 [gl.FeedbackBuffer], page 72, for details.

Note that this function doesn't detach the buffer from the GL. You need to do this manually, e.g. by changing the render mode using `gl.RenderMode()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`buffer` a buffer allocated by `gl.FeedbackBuffer()`

6.52 gl.FreeSelectBuffer

NAME

gl.FreeSelectBuffer – free selection mode buffer

SYNOPSIS

```
gl.FreeSelectBuffer(buffer)
```

FUNCTION

This function frees a buffer allocated by `gl.SelectBuffer()`. See Section 6.129 [gl.SelectBuffer], page 181, for details.

Note that this function doesn't detach the buffer from the GL. You need to do this manually, e.g. by changing the render mode using `gl.RenderMode()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`buffer` a buffer allocated by `gl.SelectBuffer()`

6.53 gl.FrontFace

NAME

gl.FrontFace – define front- and back-facing polygons

SYNOPSIS

`gl.FrontFace(mode)`

FUNCTION

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call `gl.Enable()` and `gl.Disable()` with argument `#GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `gl.FrontFace()` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `#GL_CCW` to mode selects counterclockwise polygons as front-facing; `#GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies the orientation of front-facing polygons; `#GL_CW` and `#GL_CCW` are accepted; the initial value is `#GL_CCW`

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.FrontFace()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_FRONT_FACE`

6.54 gl.Frustum

NAME

gl.Frustum – multiply the current matrix by a perspective matrix

SYNOPSIS

`gl.Frustum(left, right, bottom, top, zNear, zFar)`

FUNCTION

`gl.Frustum()` describes a perspective matrix that produces a perspective projection. (left,bottom,-zNear) and (right,top,-zNear) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). -zFar specifies the location of the far clipping plane. Both `zNear` and `zFar` must be positive. Consult an OpenGL reference for the corresponding matrix.

The current matrix is multiplied by this matrix with the result replacing the current matrix. That is, if M is the current matrix and F is the frustum perspective matrix, then M is replaced with M*F.

Use `gl.PushMatrix()` and `gl.PopMatrix()` to save and restore the current matrix stack.

Depth buffer precision is affected by the values specified for zNear and zFar. The greater the ratio of far to near is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other. If

```
r = zFar / zNear
```

roughly ld(r) bits of depth buffer precision are lost. Because r approaches infinity as zNear approaches zero, zNear must never be set to zero.

Please consult an OpenGL reference manual for more information.

INPUTS

`left`	specify the coordinate for the left vertical clipping plane
`right`	specify the coordinate for the right vertical clipping plane
`bottom`	specify the coordinate for the bottom horizontal clipping plane
`top`	specify the coordinate for the top horizontal clipping plane
`zNear`	specify the distance to the near depth clipping plane; must be positive
`zFar`	specify the distance to the far depth clipping plane; must be positive

ERRORS

`#GL_INVALID_VALUE` is generated if `zNear` or `zFar` is not positive, or if left = right, or bottom = top, or zNear = zFar.

`#GL_INVALID_OPERATION` is generated if `gl.Frustum()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.55 gl.GenLists

NAME

gl.GenLists – generate a contiguous set of empty display lists

SYNOPSIS

```
num = gl.GenLists(range)
```

FUNCTION

`gl.GenLists()` has one argument, `range`. It returns an integer n such that `range` contiguous empty display lists, named n, n + 1 , ... , n + range - 1, are created. If `range` is 0, if there is no group of range contiguous names available, or if any error is generated, no display lists are generated, and 0 is returned.

Please consult an OpenGL reference manual for more information.

INPUTS

`range` specifies the number of contiguous empty display lists to be generated

RESULTS

`num` name of first empty display list

ERRORS

`#GL_INVALID_VALUE` is generated if `range` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.GenLists()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.IsList()`

6.56 gl.GenTextures

NAME

gl.GenTextures – generate texture names

SYNOPSIS

```
texturesArray = gl.GenTextures(n)
```

FUNCTION

`gl.GenTextures()` generates `n` texture names and returns them in the table `texturesArray`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl.GenTextures()`.

The generated textures have no dimensionality; they assume the dimensionality of the texture target to which they are first bound (See Section 6.6 [gl.BindTexture], page 25, for details.).

Texture names returned by a call to `gl.GenTextures()` are not returned by subsequent calls, unless they are first deleted with `gl.DeleteTextures()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`n` specifies the number of texture names to be generated

RESULTS

`texturesArray`
table containing `n` number of texture names

ERRORS

`#GL_INVALID_VALUE` is generated if `n` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.GenTextures()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.IsTexture()`

6.57 gl.Get

NAME

gl.Get – return the value or values of a selected parameter

SYNOPSIS

```
param, ... = gl.Get(pname)
```

FUNCTION

This command return values for simple state variables in GL. `pname` is a symbolic constant indicating the state variable to be returned. The following symbolic constants are accepted by `pname`:

`#GL_ACCUM_ALPHA_BITS`
: `param` returns one value, the number of alpha bitplanes in the accumulation buffer.

`#GL_ACCUM_BLUE_BIT`
: `param` returns one value, the number of blue bitplanes in the accumulation buffer.

`#GL_ACCUM_CLEAR_VALUE`
: `param` returns four values: the red, greeen, blue, and alpha values used to clear the accumulation buffer. See Section 6.12 [gl.ClearAccum], page 33, for details.

`#GL_ACCUM_GREEN_BITS`
: `param` returns one value, the number of green bitplanes in the accumulation buffer.

`#GL_ACCUM_RED_BITS`
: `param` returns one value, the number of red bitplanes in the accumulation buffer.

`#GL_ALPHA_BIAS`
: `param` returns one value, the alpha bias factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_ALPHA_BITS`
: `param` returns one value, the number of alpha bitplanes in each color buffer.

`#GL_ALPHA_SCALE`
: `param` returns one value, the alpha scale factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_ALPHA_TEST`
: `param` returns a single Boolean value indicating whether alpha testing of fragments is enabled. See Section 6.2 [gl.AlphaFunc], page 20, for details.

`#GL_ALPHA_TEST_FUNC`
: `param` returns one value, the symbolic name of the alpha test function. See Section 6.2 [gl.AlphaFunc], page 20, for details.

`#GL_ALPHA_TEST_REF`
: `param` returns one value, the reference value for the alpha test. See Section 6.2 [gl.AlphaFunc], page 20, for details.

`#GL_ATTRIB_STACK_DEPTH`
: `param` returns one value, the depth of the attribute stack. If the stack is empty, zero is returned. See Section 6.116 [gl.PushAttrib], page 163, for details.

`#GL_AUTO_NORMAL`
: `param` returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See Section 6.93 [gl.Map], page 133, for details.

`#GL_AUX_BUFFERS`
: `param` returns one value, the number of auxiliary color buffers.

`#GL_BLEND`
: `param` returns a single Boolean value indicating whether blending is enabled. See Section 6.8 [gl.BlendFunc], page 28, for details.

`#GL_BLEND_DST`
: `param` returns one value, the symbolic constant identifying the destination blend function. See Section 6.8 [gl.BlendFunc], page 28, for details.

`#GL_BLEND_SRC`
: `param` returns one value, the symbolic constant identifying the source blend function. See Section 6.8 [gl.BlendFunc], page 28, for details.

`#GL_BLUE_BIAS`
: `param` returns one value, the blue bias factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_BLUE_BITS`
: `param` returns one value, the number of blue bitplanes in each color buffer.

`#GL_BLUE_SCALE`
: `param` returns one value, the blue scale factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_CLIP_PLANEi`
: `param` returns a single Boolean value indicating whether the specified clipping plane is enabled. See Section 6.17 [gl.ClipPlane], page 35, for details..

`#GL_COLOR_CLEAR_VALUE`
: `param` returns four values: the red, green, blue, and alpha values used to clear the color buffers. See Section 6.13 [gl.ClearColor], page 33, for details..

`#GL_COLOR_MATERIAL`
: `param` returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See Section 6.20 [gl.ColorMaterial], page 38, for details.

`#GL_COLOR_MATERIAL_FACE`
: `param` returns one value, a symbolic constant indicating which materials have a parameter that is tracking the current color. See Section 6.20 [gl.ColorMaterial], page 38, for details.

`#GL_COLOR_MATERIAL_PARAMETER`
: `param` returns one value, a symbolic constant indicating which material parameters are tracking the current color. See Section 6.20 [gl.ColorMaterial], page 38, for details.

`#GL_COLOR_WRITEMASK`
: `param` returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See Section 6.19 [gl.ColorMask], page 37, for details.

`#GL_CULL_FACE`
: `param` returns a single Boolean value indicating whether polygon culling is enabled. See Section 6.25 [gl.CullFace], page 46, for details.

`#GL_CULL_FACE_MODE`
: `param` returns one value, a symbolic constant indicating which polygon faces are to be culled. See Section 6.25 [gl.CullFace], page 46, for details.

`#GL_CURRENT_COLOR`
: `param` returns four values: the red, green, blue, and alpha values of the current color. See Section 6.18 [gl.Color], page 36, for details.

`#GL_CURRENT_INDEX`
: `param` returns one value, the current color index. See Section 6.76 [gl.Index], page 115, for details.

`#GL_CURRENT_NORMAL`
: `param` returns three values: the x, y, and z values of the current normal. See Section 6.99 [gl.Normal], page 142, for details.

`#GL_CURRENT_RASTER_COLOR`
: `param` returns four values: the red, green, blue, and alpha values of the current raster position. See Section 6.120 [gl.RasterPos], page 170, for details.

`#GL_CURRENT_RASTER_INDEX`
: `param` returns one value, the color index of the current raster position. See Section 6.120 [gl.RasterPos], page 170, for details.

`#GL_CURRENT_RASTER_POSITION`
: `param` returns four values: the x, y, z, and w components of the current raster position. x, y, and z are in window coordinates, and w is in clip coordinates. See Section 6.120 [gl.RasterPos], page 170, for details.

`#GL_CURRENT_RASTER_TEXTURE_COORDS`
: `param` returns four values: the s, t, r, and q current raster texture coordinates. See Section 6.120 [gl.RasterPos], page 170, for details. See Section 6.134 [gl.TexCoord], page 187, for details.

`#GL_CURRENT_RASTER_POSITION_VALID`
: `param` returns a single Boolean value indicating whether the current raster position is valid. See Section 6.120 [gl.RasterPos], page 170, for details.

`#GL_CURRENT_TEXTURE_COORDS`
: `param` returns four values: the s, t, r, and q current texture coordinates. See Section 6.134 [gl.TexCoord], page 187, for details.

`#GL_DEPTH_BITS`
: `param` returns one value, the number of bitplanes in the depth buffer.

`#GL_DEPTH_CLEAR_VALUE`
: `param` returns one value, the value that is used to clear the depth buffer. See Section 6.14 [gl.ClearDepth], page 34, for details.

`#GL_DEPTH_FUNC`
: `param` returns one value, the symbolic constant that indicates the depth comparison function. See Section 6.28 [gl.DepthFunc], page 48, for details.

`#GL_DEPTH_RANGE`
: `param` returns two values: the near and far mapping limits for the depth buffer. See Section 6.30 [gl.DepthRange], page 49, for details.

`#GL_DEPTH_WRITEMASK`
: `param` returns a single Boolean value indicating if the depth buffer is enabled for writing. See Section 6.29 [gl.DepthMask], page 49, for details.

`#GL_DOUBLEBUFFER`
: `param` returns a single Boolean value indicating whether double buffering is supported.

`#GL_DRAW_BUFFER`
: `param` returns one value, a symbolic constant indicating which buffers are being drawn to. See Section 6.34 [gl.DrawBuffer], page 57, for details.

`#GL_EDGE_FLAG`
: `param` returns a single Boolean value indication whether the current edge flag is true or false. See Section 6.38 [gl.EdgeFlag], page 65, for details.

`#GL_FOG`
: `param` returns a single Boolean value indicating whether fogging is enabled. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_COLOR`
: `param` returns four values: the red, green, blue, and alpha components of the fog color. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_DENSITY`
: `param` returns one value, the fog density parameter. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_END`
: `param` returns one value, the end factor for the linear fog equation. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_HINT`
: `param` returns one value, a symbolic constant indicating the mode of the fog hint. See See Section 6.75 [gl.Hint], page 113, for details.

`#GL_FOG_INDEX`
: `param` returns one value, the fog color index. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_MODE`
: `param` returns one value, a symbolic constant indicating which fog equation is selected. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FOG_START`
: `param` returns one value, the start factor for the linear fog equation. See Section 6.50 [gl.Fog], page 75, for details.

`#GL_FRONT_FACE`
: `param` returns one value, a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See Section 6.53 [gl.FrontFace], page 77, for details.

`#GL_GREEN_BIAS`
: `param` returns one value, the green bias factor used during pixel transfers.

`#GL_GREEN_BITS`
: `param` returns one value, the number of green bitplanes in each color buffer.

`#GL_GREEN_SCALE`
: `param` returns one value, the green scale factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_INDEX_BITS`
: `param` returns one value, the number of bitplanes in each color index buffer.

`#GL_INDEX_CLEAR_VALUE`
: `param` returns one value, the color index used to clear the color index buffers. See Section 6.15 [gl.ClearIndex], page 34, for details.

`#GL_INDEX_MODE`
: `param` returns a single Boolean value indicating whether the GL is in color index mode (true) or RGBA mode (false).

`#GL_INDEX_OFFSET`
: `param` returns one value, the offset added to color and stencil indices during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_INDEX_SHIFT`
: `param` returns one value, the amount that color and stencil indices are shifted during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_INDEX_WRITEMASK`
: `param` returns one value, a mask indicating which bitplanes of each color index buffer can be written. See Section 6.77 [gl.IndexMask], page 115, for details.

`#GL_LIGHTi`
: `param` returns a single Boolean value indicating whether the specified light is enabled. See Section 6.84 [gl.Light], page 123, for details. See Section 6.85 [gl.LightModel], page 125, for details.

`#GL_LIGHTING`
: `param` returns a single Boolean value indicating whether lighting is enabled. See Section 6.85 [gl.LightModel], page 125, for details.

`#GL_LIGHT_MODEL_AMBIENT`
: `param` returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. See Section 6.85 [gl.LightModel], page 125, for details.

`#GL_LIGHT_MODEL_LOCAL_VIEWER`
: `param` returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See Section 6.85 [gl.LightModel], page 125, for details.

`#GL_LIGHT_MODEL_TWO_SIDE`
: `param` returns a single Boolean value indicating whether separate materials are used to compute lighting for front- and back-facing polygons. See Section 6.85 [gl.LightModel], page 125, for details.

`#GL_LINE_SMOOTH`
: `param` returns a single Boolean value indicating whether antialiasing of lines is enabled. See Section 6.87 [gl.LineWidth], page 128, for details.

`#GL_LINE_STIPPLE`
: `param` returns a single Boolean value indicating whether stippling of lines is enabled. See Section 6.86 [gl.LineStipple], page 127, for details.

`#GL_LINE_STIPPLE_PATTERN`
: `param` returns one value, the 16-bit line stipple pattern. See Section 6.86 [gl.LineStipple], page 127, for details.

`#GL_LINE_STIPPLE_REPEAT`
: `param` returns one value, the line stipple repeat factor. See Section 6.86 [gl.LineStipple], page 127, for details.

`#GL_LINE_WIDTH`
: `param` returns one value, the line width as specified with `gl.LineWidth()`.

`#GL_LINE_WIDTH_GRANULARITY`
: `param` returns one value, the width difference between adjacent supported widths for antialiased lines. See Section 6.87 [gl.LineWidth], page 128, for details.

`#GL_LINE_WIDTH_RANGE`
: `param` returns two values: the smallest and largest supported widths for antialiased lines. See Section 6.87 [gl.LineWidth], page 128, for details.

`#GL_LIST_BASE`
: `param` returns one value, the base offset added to all names in arrays presented to `gl.CallLists()`. See Section 6.88 [gl.ListBase], page 129, for details.

`#GL_LIST_INDEX`
: `param` returns one value, the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See Section 6.98 [gl.NewList], page 141, for details.

`#GL_LIST_MODE`
: `param` returns one value, a symbolic constant indicating the construction mode of the display list currently being constructed. See Section 6.98 [gl.NewList], page 141, for details.

`#GL_LOGIC_OP`
: `param` returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_LOGIC_OP_MODE`
: `param` returns one value, a symbolic constant indicating the selected logic operational mode. See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_MAP1_COLOR_4`
: `param` returns a single Boolean value indicating whether 1D evaluation generates colors. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_GRID_DOMAIN`
: `param` returns two values: the endpoints of the 1-D map's grid domain. See Section 6.94 [gl.MapGrid], page 137, for details.

`#GL_MAP1_GRID_SEGMENTS`
: `param` returns one value, the number of partitions in the 1-D map's grid domain. See Section 6.94 [gl.MapGrid], page 137, for details.

`#GL_MAP1_INDEX`
: `param` returns a single Boolean value indicating whether 1D evaluation generates color indices. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_NORMAL`
: `param` returns a single Boolean value indicating whether 1D evaluation generates normals. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_1`
: `param` returns a single Boolean value indicating whether 1D evaluation generates 1D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_2`
: `param` returns a single Boolean value indicating whether 1D evaluation generates 2D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_3`
: `param` returns a single Boolean value indicating whether 1D evaluation generates 3D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_4`
: `param` returns a single Boolean value indicating whether 1D evaluation generates 4D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP1_VERTEX_3
: param returns a single Boolean value indicating whether 1D evaluation generates 3D vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP1_VERTEX_4
: param returns a single Boolean value indicating whether 1D evaluation generates 4D vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_COLOR_4
: param returns a single Boolean value indicating whether 2D evaluation generates colors. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_GRID_DOMAIN
: param returns four values: the endpoints of the 2-D map's i and j grid domains. See Section 6.94 [gl.MapGrid], page 137, for details.

#GL_MAP2_GRID_SEGMENTS
: param returns two values: the number of partitions in the 2-D map's i and j grid domains. See Section 6.94 [gl.MapGrid], page 137, for details.

#GL_MAP2_INDEX
: param returns a single Boolean value indicating whether 2D evaluation generates color indices. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_NORMAL
: param returns a single Boolean value indicating whether 2D evaluation generates normals. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_TEXTURE_COORD_1
: param returns a single Boolean value indicating whether 2D evaluation generates 1D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_TEXTURE_COORD_2
: param returns a single Boolean value indicating whether 2D evaluation generates 2D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_TEXTURE_COORD_3
: param returns a single Boolean value indicating whether 2D evaluation generates 3D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_TEXTURE_COORD_4
: param returns a single Boolean value indicating whether 2D evaluation generates 4D texture coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_VERTEX_3
: param returns a single Boolean value indicating whether 2D evaluation generates 3D vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP2_VERTEX_4
: param returns a single Boolean value indicating whether 2D evaluation generates 4D vertex coordinates. See Section 6.93 [gl.Map], page 133, for details.

#GL_MAP_COLOR
: param returns a single Boolean value indicating if colors and color indices are to be replaced by table lookup during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_MAP_STENCIL`
: `param` returns a single Boolean value indicating if stencil indices are to be replaced by table lookup during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_MATRIX_MODE`
: `param` returns one value, a symbolic constant indicating which matrix stack is currently the target of all matrix operations. See glMatrixMode.

`#GL_MAX_ATTRIB_STACK_DEPTH`
: `param` returns one value, the maximum supported depth of the attribute stack. See Section 6.116 [gl.PushAttrib], page 163, for details.

`#GL_MAX_CLIP_PLANES`
: `param` returns one value, the maximum number of application-defined clipping planes. See Section 6.17 [gl.ClipPlane], page 35, for details.

`#GL_MAX_EVAL_ORDER`
: `param` returns one value, the maximum equation order supported by 1-D and 2-D evaluators. See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAX_LIGHTS`
: `param` returns one value, the maximum number of lights. See Section 6.84 [gl.Light], page 123, for details.

`#GL_MAX_LIST_NESTING`
: `param` returns one value, the maximum recursion depth allowed during display-list traversal. See Section 6.9 [gl.CallList], page 30, for details.

`#GL_MAX_MODELVIEW_STACK_DEPTH`
: `param` returns one value, the maximum supported depth of the modelview matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_MAX_NAME_STACK_DEPTH`
: `param` returns one value, the maximum supported depth of the selection name stack. See Section 6.119 [gl.PushName], page 169, for details.

`#GL_MAX_PIXEL_MAP_TABLE`
: `param` returns one value, the maximum supported size of a glPixelMap lookup table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_MAX_PROJECTION_STACK_DEPTH`
: `param` returns one value, the maximum supported depth of the projection matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_MAX_TEXTURE_SIZE`
: `param` returns one value, the maximum width or height of any texture image (without borders). See Section 6.139 [gl.TexImage1D], page 192, for details. See Section 6.140 [gl.TexImage2D], page 196, for details.

`#GL_MAX_TEXTURE_STACK_DEPTH`
: `param` returns one value, the maximum supported depth of the texture matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_MAX_VIEWPORT_DIMS`
: `param` returns two values: the maximum supported width and height of the viewport. See Section 6.148 [gl.Viewport], page 208, for details.

`#GL_MODELVIEW_MATRIX`
: `param` returns sixteen values: the modelview matrix on the top of the modelview matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_MODELVIEW_STACK_DEPTH`
: `param` returns one value, the number of matrices on the modelview matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_NAME_STACK_DEPTH`
: `param` returns one value, the number of names on the selection name stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_NORMALIZE`
: `param` returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See Section 6.99 [gl.Normal], page 142, for details.

`#GL_PACK_ALIGNMENT`
: `param` returns one value, the byte alignment used for writing pixel data to memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PACK_LSB_FIRST`
: `param` returns a single Boolean value indicating whether single-bit pixels being written to memory are written first to the least significant bit of each unsigned byte. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PACK_ROW_LENGTH`
: `param` returns one value, the row length used for writing pixel data to memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PACK_SKIP_PIXELS`
: `param` returns one value, the number of pixel locations skipped before the first pixel is written into memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PACK_SKIP_ROWS`
: `param` returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PACK_SWAP_BYTES`
: `param` returns a single Boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped before being written to memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_PIXEL_MAP_A_TO_A_SIZE`
: `param` returns one value the size of the alpha-to-alpha pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_B_TO_B_SIZE`
: `param` returns one value, the size of the blue-to-blue pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_G_TO_G_SIZE`
: `param` returns one value, the size of the green-to-green pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_I_TO_A_SIZE`
: `param` returns one value, the size of the index-to-alpha pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_I_TO_B_SIZE`
: `param` returns one value, the size of the index-to-blue pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_I_TO_G_SIZE`
: `param` returns one value, the size of the index-to-green pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_I_TO_I_SIZE`
: `param` returns one value, the size of the index-to-index pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_I_TO_R_SIZE`
: `param` returns one value, the size of the index-to-red pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_R_TO_R_SIZE`
: `param` returns one value, the size of the red-to-red pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_PIXEL_MAP_S_TO_S_SIZE`
: `param` returns one value, the size of the stencil-to-stencil pixel translation table. See Section 6.103 [gl.PixelMap], page 146, for details.

`#GL_POINT_SIZE`
: `param` returns one value, the point size as specified by `gl.PointSize()`.

`#GL_POINT_SIZE_GRANULARITY`
: `param` returns one value, the size difference between adjacent supported sizes for antialiased points. See Section 6.107 [gl.PointSize], page 155, for details.

`#GL_POINT_SIZE_RANGE`
: `param` returns two values: the smallest and largest supported sizes for antialiased points. See Section 6.107 [gl.PointSize], page 155, for details.

`#GL_POINT_SMOOTH`
: `param` returns a single Boolean value indicating whether antialiasing of points is enabled. See Section 6.107 [gl.PointSize], page 155, for details.

`#GL_POLYGON_MODE`
: `param` returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See Section 6.108 [gl.PolygonMode], page 156, for details.

`#GL_POLYGON_SMOOTH`
: `param` returns a single Boolean value indicating whether antialiasing of polygons is enabled. See Section 6.108 [gl.PolygonMode], page 156, for details.

`#GL_POLYGON_STIPPLE`
: `param` returns a single Boolean value indicating whether stippling of polygons is enabled. See Section 6.110 [gl.PolygonStipple], page 158, for details.

`#GL_PROJECTION_MATRIX`
: `param` returns sixteen values: the projection matrix on the top of the projection matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_PROJECTION_STACK_DEPTH`
: `param` returns one value, the number of matrices on the projection matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_READ_BUFFER`
: `param` returns one value, a symbolic constant indicating which color buffer is selected for reading. See Section 6.122 [gl.ReadPixels], page 173, for details. See Section 6.1 [gl.Accum], page 19, for details.

`#GL_RED_BIAS`
: `param` returns one value, the red bias factor used during pixel transfers.

`#GL_RED_BITS`
: `param` returns one value, the number of red bitplanes in each color buffer.

`#GL_RED_SCALE`
: `param` returns one value, the red scale factor used during pixel transfers. See Section 6.105 [gl.PixelTransfer], page 152, for details.

`#GL_RENDER_MODE`
: `param` returns one value, a symbolic constant indicating whether the GL is in render, select, or feedback mode. See Section 6.125 [gl.RenderMode], page 177, for details.

`#GL_RGBA_MODE`
: `param` returns a single Boolean value indicating whether the GL is in RGBA mode (true) or color index mode (false). See Section 6.18 [gl.Color], page 36, for details.

`#GL_SCISSOR_BOX`
: `param` returns four values: the x and y window coordinates of the scissor box, follow by its width and height. See Section 6.128 [gl.Scissor], page 180, for details.

`#GL_SCISSOR_TEST`
: `param` returns a single Boolean value indicating whether scissoring is enabled. See Section 6.128 [glScisscor], page 180, for details.

`#GL_SHADE_MODEL`
: `param` returns one value, a symbolic constant indicating whether the shading mode is flat or smooth. See Section 6.130 [gl.ShadeModel], page 183, for details.

`#GL_STENCIL_BITS`
: `param` returns one value, the number of bitplanes in the stencil buffer.

`#GL_STENCIL_CLEAR_VALUE`
: `param` returns one value, the index to which the stencil bitplanes are cleared. See Section 6.16 [gl.ClearStencil], page 35, for details.

`#GL_STENCIL_FAIL`
: `param` returns one value, a symbolic constant indicating what action is taken when the stencil test fails. See Section 6.133 [gl.StencilOp], page 186, for details.

`#GL_STENCIL_FUNC`
: `param` returns one value, a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See Section 6.131 [gl.StencilFunc], page 184, for details.

`#GL_STENCIL_PASS_DEPTH_FAIL`
: `param` returns one value, a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See Section 6.133 [gl.StencilOp], page 186, for details.

`#GL_STENCIL_PASS_DEPTH_PASS`
: `param` returns one value, a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See Section 6.133 [gl.StencilOp], page 186, for details.

`#GL_STENCIL_REF`
: `param` returns one value, the reference value that is compared with the contents of the stencil buffer. See Section 6.131 [gl.StencilFunc], page 184, for details.

`#GL_STENCIL_TEST`
: `param` returns a single Boolean value indicating whether stencil testing of fragments is enabled. See glStencilFunc and glStencilOp.

`#GL_STENCIL_VALUE_MASK`
: `param` returns one value, the mask that is used to mask both the stencil reference value and the stencil buffer value before they are compared. See Section 6.131 [gl.StencilFunc], page 184, for details.

`#GL_STENCIL_WRITEMASK`
: `param` returns one value, the mask that controls writing of the stencil bitplanes. See Section 6.132 [gl.StencilMask], page 185, for details.

`#GL_STEREO`
: `param` returns a single Boolean value indicating whether stereo buffers (left and right) are supported.

`#GL_SUBPIXEL_BITS`
: `param` returns one value, an estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates.

`#GL_TEXTURE_1D`
: `param` returns a single Boolean value indicating whether 1D texture mapping is enabled. See Section 6.139 [gl.TexImage1D], page 192, for details.

`#GL_TEXTURE_2D`
: `param` returns a single Boolean value indicating whether 2D texture mapping is enabled. See Section 6.140 [gl.TexImage2D], page 196, for details.

`#GL_TEXTURE_GEN_S`
: `param` returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_T`
: `param` returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_R`
: `param` returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_Q`
: `param` returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_MATRIX`
: `param` returns sixteen values: the texture matrix on the top of the texture matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_TEXTURE_STACK_DEPTH`
: `param` returns one value, the number of matrices on the texture matrix stack. See Section 6.118 [gl.PushMatrix], page 169, for details.

`#GL_UNPACK_ALIGNMENT`
: `param` returns one value, the byte alignment used for reading pixel data from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_LSB_FIRST`
: `param` returns a single Boolean value indicating whether single-bit pixels being read from memory are read first from the least significant bit of each unsigned byte. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_ROW_LENGTH`
: `param` returns one value, the row length used for reading pixel data from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_SKIP_IMAGES`
: `param` returns one value, the number of images skipped before the first (3D) pixel is read from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_SKIP_PIXELS`
: `param` returns one value, the number of pixel locations skipped before the first pixel is read from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_SKIP_ROWS`
: `param` returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_UNPACK_SWAP_BYTES`
: `param` returns a single Boolean value indicating whether the bytes of two-byte and four-byte pixel indices and components are swapped after being read from memory. See Section 6.104 [gl.PixelStore], page 148, for details.

`#GL_VIEWPORT`
: `param` returns four values: the x and y window coordinates of the viewport, follow by its width and height. See Section 6.148 [gl.Viewport], page 208, for details..

`#GL_ZOOM_X`
: `param` returns one value, the x pixel zoom factor. See Section 6.106 [gl.PixelZoom], page 154, for details.

`#GL_ZOOM_Y`
: `param` returns one value, the y pixel zoom factor. See Section 6.106 [gl.PixelZoom], page 154, for details.

Many of the boolean parameters can also be queried more easily using `gl.IsEnabled()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the parameter value to be returned (see above for supported constants)

RESULTS

`param` value of the specified parameter

`...` additional return values depending on `pname`

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.Get()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.58 gl.GetArray

NAME

gl.GetArray – return the value or values of a selected parameter as an array

SYNOPSIS

```
paramsArray = gl.GetArray(pname)
```

FUNCTION

This function does the same as `gl.Get()` except that the values are returned as an array. See Section 6.57 [gl.Get], page 81, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the parameter value to be returned

RESULTS

`paramsArray`
parameter values in an array

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if gl.Get() is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.59 gl.GetClipPlane

NAME

gl.GetClipPlane – return the coefficients of the specified clipping plane

SYNOPSIS

```
equationArray = gl.GetClipPlane(plane)
```

FUNCTION

`gl.GetClipPlane()` returns in equation the four coefficients of the plane equation for plane.

It is always the case that `#GL_CLIP_PLANEi` = `#GL_CLIP_PLANE0` + i.

If an error is generated, no change is made to the contents of equation.

Please consult an OpenGL reference manual for more information.

INPUTS

`plane` specifies a clipping plane; the number of clipping planes depends on the implementation, but at least six clipping planes are supported; they are identified by symbolic names of the form `#GL_CLIP_PLANEi` where `i` ranges from 0 to the value of `#GL_MAX_CLIP_PLANES` - 1

RESULTS

`equationArray`
table with four double-precision values that are the coefficients of the plane equation of plane in eye coordinates; the initial value is (0, 0, 0, 0)

ERRORS

`#GL_INVALID_ENUM` is generated if plane is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetClipPlane()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.60 gl.GetError

NAME

gl.GetError – return error information

SYNOPSIS

```
error = gl.GetError()
```

FUNCTION

`gl.GetError()` returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until `gl.GetError()` is called, the error code is returned, and the flag is reset to `#GL_NO_ERROR`. If a call to `gl.GetError()` returns `#GL_NO_ERROR`, there has been no detectable error since the last call to `gl.GetError()`, or since the GL was initialized.

To allow for distributed implementations, there may be several error flags. If any single error flag has recorded an error, the value of that flag is returned and that flag is reset to `#GL_NO_ERROR` when `gl.GetError()` is called. If more than one flag has recorded an error, `gl.GetError()` returns and clears an arbitrary error flag value. Thus, `gl.GetError()` should always be called in a loop, until it returns `#GL_NO_ERROR`, if all error flags are to be reset.

Initially, all error flags are set to `#GL_NO_ERROR`.

The following errors are currently defined:

`#GL_NO_ERROR`
: No error has been recorded. The value of this symbolic constant is guaranteed to be 0.

`#GL_INVALID_ENUM`
: An unacceptable value is specified for an enumerated argument. The offending command is ignored and has no other side effect than to set the error flag.

`#GL_INVALID_VALUE`
: A numeric argument is out of range. The offending command is ignored and has no other side effect than to set the error flag.

`#GL_INVALID_OPERATION`
: The specified operation is not allowed in the current state. The offending command is ignored and has no other side effect than to set the error flag.

`#GL_STACK_OVERFLOW`
: This command would cause a stack overflow. The offending command is ignored and has no other side effect than to set the error flag.

`#GL_STACK_UNDERFLOW`
: This command would cause a stack underflow. The offending command is ignored and has no other side effect than to set the error flag.

`#GL_OUT_OF_MEMORY`
: There is not enough memory left to execute the command. The state of the GL is undefined, except for the state of the error flags, after this error is recorded.

When an error flag is set, results of a GL operation are undefined only if `#GL_OUT_OF_MEMORY` has occurred. In all other cases, the command generating the error is ignored and has no effect on the GL state or frame buffer contents. If the generating command returns a value, it returns 0. If `gl.GetError()` itself generates an error, it returns 0.

Please consult an OpenGL reference manual for more information.

INPUTS

none

RESULTS

`error` value of GL's error flag

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.GetError()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` . In this case, `gl.GetError()` returns 0.

6.61 gl.GetLight

NAME

gl.GetLight – return light source parameter values

SYNOPSIS

```
paramsArray = gl.GetLight(light, pname)
```

FUNCTION

`gl.GetLight()` returns in `paramsArray` the value or values of a light source parameter. `light` names the light and is a symbolic name of the form `#GL_LIGHTi` where `i` ranges from 0 to the value of `#GL_MAX_LIGHTS - 1`. `#GL_MAX_LIGHTS` is an implementation dependent constant that is greater than or equal to eight. `pname` specifies one of ten light source parameters, again by symbolic name.

The following parameters are defined:

`#GL_AMBIENT`
: Returns four floating-point values representing the ambient intensity of the light source. The initial value is (0, 0, 0, 1).

`#GL_DIFFUSE`
: Returns four floating-point values representing the diffuse intensity of the light source. The initial value for `#GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

`#GL_SPECULAR`
: Returns four floating-point values representing the specular intensity of the light source. The initial value for `#GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 0).

`#GL_POSITION`
: Returns four floating-point values representing the position of the light source. The returned values are those maintained in eye coordinates. They

will not be equal to the values specified using `gl.Light()`, unless the modelview matrix was identity at the time `gl.Light()` was called. The initial value is (0, 0, 1, 0).

`#GL_SPOT_DIRECTION`
: Returns three floating-point values representing the direction of the light source. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using `gl.Light()`, unless the modelview matrix was identity at the time `gl.Light()` was called. Although spot direction is normalized before being used in the lighting equation, the returned values are the transformed versions of the specified values prior to normalization. The initial value is (0,0,-1).

`#GL_SPOT_EXPONENT`
: Returns a single floating-point value representing the spot exponent of the light. The initial value is 0.

`#GL_SPOT_CUTOFF`
: Returns a single floating-point value representing the spot cutoff angle of the light. The initial value is 180.

`#GL_CONSTANT_ATTENUATION`
: Returns a single floating-point value representing the constant (not distance-related) attenuation of the light. The initial value is 1.

`#GL_LINEAR_ATTENUATION`
: Returns a single floating-point value representing the linear attenuation of the light. The initial value is 0.

`#GL_QUADRATIC_ATTENUATION`
: Returns a single floating-point value representing the quadratic attenuation of the light. The initial value is 0.

It is always the case that `#GL_LIGHT` i = `#GL_LIGHT0 +` i.

Please consult an OpenGL reference manual for more information.

INPUTS

`light` specifies a light source; the number of possible lights depends on the implementation, but at least eight lights are supported; they are identified by symbolic names of the form `#GL_LIGHTi` where `i` ranges from 0 to the value of `#GL_MAX_LIGHTS` - 1

`pname` specifies a light source parameter for `light` (see above for possible parameters)

RESULTS

`paramsArray`
: table containing requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `light` or `pname` is not an accepted value.

#GL_INVALID_OPERATION is generated if gl.GetLight() is executed between the execution of gl.Begin() and the corresponding execution of gl.End() .

6.62 gl.GetMap

NAME

gl.GetMap – return evaluator parameters

SYNOPSIS

```
vArray = gl.GetMap(target, query)
```

FUNCTION

gl.Map() defines an evaluator. gl.GetMap() returns evaluator parameters. target chooses a map, and query selects a specific parameter. The following values are currently supported for target:

```
#GL_MAP1_COLOR_4
#GL_MAP1_INDEX
#GL_MAP1_NORMAL
#GL_MAP1_TEXTURE_COORD_1
#GL_MAP1_TEXTURE_COORD_2
#GL_MAP1_TEXTURE_COORD_3,
#GL_MAP1_TEXTURE_COORD_4
#GL_MAP1_VERTEX_3
#GL_MAP1_VERTEX_4
#GL_MAP2_COLOR_4
#GL_MAP2_INDEX
#GL_MAP2_NORMAL
#GL_MAP2_TEXTURE_COORD_1
#GL_MAP2_TEXTURE_COORD_2
#GL_MAP2_TEXTURE_COORD_3
#GL_MAP2_TEXTURE_COORD_4
#GL_MAP2_VERTEX_3
#GL_MAP2_VERTEX_4
```

See Section 6.93 [gl.Map], page 133, for details.

query can assume the following values:

#GL_COEFF

v returns the control points for the evaluator function. One-dimensional evaluators return order control points, and two-dimensional evaluators return uorder*vorder control points. Each control point consists of one, two, three, or four double-precision floating-point values. The GL returns two-dimensional control points in row-major order, incrementing the uorder index quickly and the vorder index after each row.

#GL_ORDER

v returns the order of the evaluator function. One-dimensional evaluators return a single value, order. The initial value is 1. Two-dimensional evaluators return two values, uorder and vorder. The initial value is (1,1).

`#GL_DOMAIN`
: v returns the linear u and v mapping parameters. One-dimensional evaluators return two values, u1 and u2, as specified by `gl.Map()`. Two-dimensional evaluators return four values (u1, u2, v1, and v2) as specified by `gl.Map()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the symbolic name of a map (see above for possible values)

`query` specifies which parameter to return (see above for possible values)

RESULTS

`vArray` table containing the requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `either` target or `query` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetMap()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.63 gl.GetMaterial

NAME

gl.GetMaterial – return material parameters

SYNOPSIS

```
paramsArray = gl.GetMaterial(face, pname)
```

FUNCTION

`gl.GetMaterial()` returns a table containing the value or values of parameter `pname` of material `face`. The following six parameters can be passed in `pname`:

`#GL_AMBIENT`
: Returns four floating-point values representing the ambient reflectance of the material. The initial value is (0.2, 0.2, 0.2, 1.0)

`#GL_DIFFUSE`
: Returns four floating-point values representing the diffuse reflectance of the material. The initial value is (0.8, 0.8, 0.8, 1.0).

`#GL_SPECULAR`
: Returns four floating-point values representing the specular reflectance of the material. The initial value is (0, 0, 0, 1).

`#GL_EMISSION`
: Returns four floating-point values representing the emitted light intensity of the material. The initial value is (0, 0, 0, 1).

`#GL_SHININESS`
: Returns one floating-point value representing the specular exponent of the material. The initial value is 0.

`#GL_COLOR_INDEXES`
: Returns three floating-point values representing the ambient, diffuse, and specular indices of the material. These indices are used only for color index lighting. (All the other parameters are used only for RGBA lighting.)

Please consult an OpenGL reference manual for more information.

INPUTS

`face` specifies which of the two materials is being queried; `#GL_FRONT` or `#GL_BACK` are accepted, representing the front and back materials, respectively

`pname` specifies the material parameter to return (see above for possible values)

RESULTS

`paramsArray`
: table containing the requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `face` or `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetMaterial()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.64 gl.GetPixelMap

NAME

gl.GetPixelMap – return the specified pixel map

SYNOPSIS

```
valuesArray = gl.GetPixelMap(map)
```

FUNCTION

`gl.GetPixelMap()` returns the contents of the pixel map specified in `map`. This can be one of the following constants:

```
#GL_PIXEL_MAP_I_TO_I
#GL_PIXEL_MAP_S_TO_S
#GL_PIXEL_MAP_I_TO_R
#GL_PIXEL_MAP_I_TO_G
#GL_PIXEL_MAP_I_TO_B
#GL_PIXEL_MAP_I_TO_A
#GL_PIXEL_MAP_R_TO_R
#GL_PIXEL_MAP_G_TO_G
#GL_PIXEL_MAP_B_TO_B
#GL_PIXEL_MAP_A_TO_A
```

See Section 6.103 [gl.PixelMap], page 146, for details.

Pixel maps are used during the execution of `gl.ReadPixels()`, `gl.DrawPixels()`, `gl.CopyPixels()`, and `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage1D()`, `gl.TexSubImage2D()`, `gl.CopyTexImage()` and `gl.CopyTexSubImage()`, to map color indices, stencil indices, color components, and depth components to other values.

Please consult an OpenGL reference manual for more information.

INPUTS

`map` specifies the name of the pixel map to return (see above for possible values)

RESULTS

`valuesArray`
table containing the pixel map contents

ERRORS

`#GL_INVALID_ENUM` is generated if `map` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetPixelMap()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_I_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_S_TO_S_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_R_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_G_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_B_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_A_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_R_TO_R_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_G_TO_G_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_B_TO_B_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_A_TO_A_SIZE`

`gl.Get()` with argument `#GL_MAX_PIXEL_MAP_TABLE`

6.65 gl.GetPointer

NAME

gl.GetPointer – return values of the specified pointer

SYNOPSIS

```
valuesArray = gl.GetPointer(pname, n)
```

FUNCTION

`gl.GetPointer()` returns elements read from a GL pointer. `pname` is a symbolic constant indicating the pointer to be used and `n` specifies how many elements should be read and returned. `pname` can be set to the following values:

```
#GL_COLOR_ARRAY_POINTER
#GL_EDGE_FLAG_ARRAY_POINTER
#GL_FEEDBACK_BUFFER_POINTER
#GL_INDEX_ARRAY_POINTER
#GL_NORMAL_ARRAY_POINTER
#GL_SELECTION_BUFFER_POINTER
#GL_TEXTURE_COORD_ARRAY_POINTER
#GL_VERTEX_ARRAY_POINTER
```

The pointers are all client-side state.

The initial value for each pointer is `NULL`.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the array or buffer pointer to be queried (see above for possible values)

`n` number of items to read from pointer

RESULTS

`valuesArray`
table containing `n` items read from the respective pointer

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

6.66 gl.GetPolygonStipple

NAME

gl.GetPolygonStipple – return the polygon stipple pattern

SYNOPSIS

```
maskArray = gl.GetPolygonStipple()
```

FUNCTION

`gl.GetPolygonStipple()` returns to pattern a 32*32 polygon stipple pattern. The pattern is packed into memory as if `gl.ReadPixels()` with both height and width of 32, type of `#GL_BITMAP`, and format of `#GL_COLOR_INDEX` were called, and the stipple pattern were stored in an internal 32*32 color index buffer. Unlike `gl.ReadPixels()`, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image. Since `#GL_BITMAP` uses only 1-bit per pixel, the table returned by this function will always have exactly 128 elements containing 8 pixels per table element.

Please consult an OpenGL reference manual for more information.

INPUTS

none

RESULTS

`maskArray`
table containing the stipple pattern; the initial value is all 1's

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.GetPolygonStipple()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.67 gl.GetSelectBuffer

NAME

gl.GetSelectBuffer – read value from the selection buffer

SYNOPSIS

```
value = gl.GetSelectBuffer(buffer, index)
```

FUNCTION

This function can be used to read the value at index `index` in the selection buffer passed in `buffer`. This buffer must have been allocated by `gl.SelectBuffer()`. Values are read as unsigned four byte integers starting at index 0.

See Section 6.129 [gl.SelectBuffer], page 181, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`buffer` memory buffer allocated by `gl.SelectBuffer()`

`index` index of the value to read (starting at index 0)

RESULTS

`value` value at the specified index

6.68 gl.GetString

NAME

gl.GetString – return a string describing the current GL connection

SYNOPSIS

```
string = gl.GetString(name)
```

FUNCTION

`gl.GetString()` returns a pointer to a static string describing some aspect of the current GL connection. name can be one of the following:

`#GL_VENDOR`
: Returns the company responsible for this GL implementation. This name does not change from release to release.

`#GL_RENDERER`
: Returns the name of the renderer. This name is typically specific to a particular configuration of a hardware platform. It does not change from release to release.

`#GL_VERSION`
: Returns a version or release number.

`#GL_EXTENSIONS`
: Returns a space-separated list of supported extensions to GL.

Because the GL does not include queries for the performance characteristics of an implementation, some applications are written to recognize known platforms and modify their

GL usage based on known performance characteristics of these platforms. Strings `#GL_VENDOR` and `#GL_RENDERER` together uniquely specify a platform. They do not change from release to release and should be used by platform-recognition algorithms.

Some applications want to make use of features that are not part of the standard GL. These features may be implemented as extensions to the standard GL. The `#GL_EXTENSIONS` string is a space-separated list of supported GL extensions. (Extension names never contain a space character.)

The `#GL_VERSION` string begins with a version number. The version number uses one of these forms:

```
<major_number>.<minor_number>
<major_number>.<minor_number>.<release_number>
```

Vendor-specific information may follow the version number. Its format depends on the implementation, but a space always separates the version number and the vendor-specific information.

The client and server may support different versions or extensions. `gl.GetString()` always returns a compatible version number or list of extensions. The release number always describes the server.

Please consult an OpenGL reference manual for more information.

INPUTS

`name` specifies a symbolic constant (see above for possible values)

RESULTS

`string` string describing the current GL connection

ERRORS

`#GL_INVALID_ENUM` is generated if `name` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetString()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.69 gl.GetTexEnv

NAME

gl.GetTexEnv – return texture environment parameters

SYNOPSIS

```
paramsArray = gl.GetTexEnv(pname)
```

FUNCTION

`gl.GetTexEnv()` returns a table containing selected values of a texture environment that was specified with `gl.TexEnv()`. `pname` names a specific texture environment parameter. The two parameters are as follows:

`#GL_TEXTURE_ENV_MODE`
: Returns the single-valued texture environment mode, a symbolic constant.

`#GL_TEXTURE_ENV_COLOR`
: Returns four floating-point values that are the texture environment color.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the symbolic name of a texture environment parameter (see above for possible values)

RESULTS

`paramsArray`
table containing the requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexEnv()` is called between a call to glBegin and the corresponding execution to glEnd.

6.70 gl.GetTexGen

NAME

gl.GetTexGen – return texture coordinate generation parameters

SYNOPSIS

```
paramsArray = gl.GetTexGen(coord, pname)
```

FUNCTION

`gl.GetTexGen()` returns a table containing selected parameters of a texture coordinate generation function that was specified using `gl.TexGen(). coord` names one of the (s, t, r, q) texture coordinates, using the symbolic constants `#GL_S`, `#GL_T`, `#GL_R`, or `#GL_Q`.

`pname` specifies one of three symbolic names:

`#GL_TEXTURE_GEN_MODE`
Returns the single-valued texture generation function, a symbolic constant. The initial value is `#GL_EYE_LINEAR`.

`#GL_OBJECT_PLANE`
This will return the four plane equation coefficients that specify object linear-coordinate generation.

`#GL_EYE_PLANE`
Returns the four plane equation coefficients that specify eye linear-coordinate generation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using `gl.TexGen()`, unless the modelview matrix was identity when `gl.TexGen()` was called.

Please consult an OpenGL reference manual for more information.

INPUTS

`coord` specifies a texture coordinate

`pname` specifies the symbolic name of the value(s) to be returned

RESULTS

`paramsArray`
table containing the requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `coord` or `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexGen()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.71 gl.GetTexImage

NAME

gl.GetTexImage – return a texture image

SYNOPSIS

```
pixelsArray = gl.GetTexImage(target, level, format)
```

FUNCTION

`gl.GetTexImage()` returns the pixels of a texture image. One-dimensional textures are returned in a one-dimensional table whereas two-dimensional textures are returned in a table that contains subtables for all rows in the texture. The pixels are returned as values of type `#GL_FLOAT` `target` specifies whether the desired texture image is one specified by `gl.TexImage1D()` (`#GL_TEXTURE_1D`) or `gl.TexImage2D()` (`#GL_TEXTURE_2D`). `level` specifies the level-of-detail number of the desired image. `format` specifies the format of the desired image array. See Section 6.140 [gl.TexImage2D], page 196, for a description of the acceptable values for the format parameter.

To understand the operation of `gl.GetTexImage()`, consider the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of `gl.GetTexImage()` are then identical to those of `gl.ReadPixels()`, with the exception that no pixel transfer operations are performed, when called with the same format and type, with x and y set to 0, width set to the width of the texture image (including border if one was specified), and height set to 1 for 1D images, or to the height of the texture image (including border if one was specified) for 2D images. Because the internal texture image is an RGBA image, pixel formats `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX`, and `#GL_DEPTH_COMPONENT` are not accepted, and pixel type `#GL_BITMAP` is not accepted.

If the selected texture image does not contain four components, the following mappings are applied. Single-component textures are treated as RGBA buffers with red set to the single-component value, green set to 0, blue set to 0, and alpha set to 1. Two-component textures are treated as RGBA buffers with red set to the value of component zero, alpha set to the value of component one, and green and blue set to 0. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to 1.

If you want to have fine-tuned control over the pixel type or if you want the pixels to be written into a memory buffer instead of a table, you can use the `gl.GetTexImageRaw()` function instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies which texture is to be obtained (must be `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`)

`level` specifies the level-of-detail number of the desired image; level 0 is the base image level; level n is the nth mipmap reduction image

`format` specifies a pixel format for the returned data; the supported formats are `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, and `#GL_LUMINANCE_ALPHA`

RESULTS

`pixelsArray`
table containing the raw pixels

ERRORS

`#GL_INVALID_ENUM` is generated if `target` or `format` is not an accepted value.

`#GL_INVALID_VALUE` is generated if `level` is less than zero or greater than ld(max), where `max` is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexImage()` is called between a call to glBegin and the corresponding call to glEnd.

ASSOCIATED GETS

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_WIDTH`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_HEIGHT`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_BORDER`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_COMPONENTS`

`gl.Get()` with arguments `#GL_PACK_ALIGNMENT` and others

6.72 gl.GetTexImageRaw

NAME

gl.GetTexImageRaw – return a texture image

SYNOPSIS

```
gl.GetTexImageRaw(target, level, format, type, pixels)
```

FUNCTION

`gl.GetTexImageRaw()` writes the pixels of a texture image to `pixels`. This must be a memory buffer allocated by Hollywood's `AllocMem()` function and returned by `GetMemPointer()`. To determine the required size of `pixels`, use `gl.GetTexLevelParameter()` to determine the dimensions of the internal texture image, then scale the required number of pixels by the storage required for each pixel, based on `format` and `type`. Be sure to take the pixel storage parameters into account, especially `#GL_PACK_ALIGNMENT`.

The supported values for `format` are `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, and `#GL_LUMINANCE_ALPHA`.

Supported data types for `type` are `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, and `#GL_FLOAT`.

The pixels are written to the memory buffer as values of type `type`. `target` specifies whether the desired texture image is one specified by `gl.TexImage1D()` (`#GL_TEXTURE_1D`) or `gl.TexImage2D()` (`#GL_TEXTURE_2D`). `level` specifies the level-of-detail number of the desired image. `format` specifies the format of the desired image array. See Section 6.140 [gl.TexImage2D], page 196, for a description of the acceptable values for the format parameter.

To understand the operation of `gl.GetTexImageRaw()`, consider the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of `gl.GetTexImageRaw()` are then identical to those of `gl.ReadPixels()`, with the exception that no pixel transfer operations are performed, when called with the same format and type, with x and y set to 0, width set to the width of the texture image (including border if one was specified), and height set to 1 for 1D images, or to the height of the texture image (including border if one was specified) for 2D images. Because the internal texture image is an RGBA image, pixel formats `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX`, and `#GL_DEPTH_COMPONENT` are not accepted, and pixel type `#GL_BITMAP` is not accepted.

If the selected texture image does not contain four components, the following mappings are applied. Single-component textures are treated as RGBA buffers with red set to the single-component value, green set to 0, blue set to 0, and alpha set to 1. Two-component textures are treated as RGBA buffers with red set to the value of component zero, alpha set to the value of component one, and green and blue set to 0. Finally, three-component textures are treated as RGBA buffers with red set to component zero, green set to component one, blue set to component two, and alpha set to 1.

If you want to have the pixels returned in a table instead of a memory buffer, you can use the `gl.GetTexImage()` function instead. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies which texture is to be obtained (must be `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`)

`level` specifies the level-of-detail number of the desired image; level 0 is the base image level; level n is the nth mipmap reduction image

`format` specifies a pixel format for the returned data (see above)

`type` specifies a pixel type for the returned data (see above)

`pixels` pointer to a memory buffer to write the pixels to

ERRORS

`#GL_INVALID_ENUM` is generated if `target`, `type` or `format` is not an accepted value.

`#GL_INVALID_VALUE` is generated if `level` is less than zero or greater than ld(max), where `max` is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexImageRaw()` is called between a call to glBegin and the corresponding call to glEnd.

ASSOCIATED GETS

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_WIDTH`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_HEIGHT`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_BORDER`

`gl.GetTexLevelParameter()` with argument `#GL_TEXTURE_COMPONENTS`

`gl.Get()` with arguments `#GL_PACK_ALIGNMENT` and others

6.73 gl.GetTexLevelParameter

NAME

gl.GetTexLevelParameter – return texture parameter values for a specific level of detail

SYNOPSIS

```
param = gl.GetTexLevelParameter(target, level, pname)
```

FUNCTION

`gl.GetTexLevelParameter()` returns texture parameter values for a specific level-of-detail value, specified as `level`. `target` defines the target texture, either `#GL_TEXTURE_1D`, `#GL_TEXTURE_2D`, `#GL_PROXY_TEXTURE_1D`, or `#GL_PROXY_TEXTURE_2D`.

`#GL_MAX_TEXTURE_SIZE` is not really descriptive enough. It has to report the largest square texture image that can be accommodated with mipmaps and borders, but a long skinny texture, or a texture without mipmaps and borders, may easily fit in texture memory. The proxy targets allow the user to more accurately query whether the GL can accommodate a texture of a given configuration. If the texture cannot be accommodated, the texture state variables, which may be queried with `gl.GetTexLevelParameter()`, are set to 0. If the texture can be accommodated, the texture state values will be set as they would be set for a non-proxy target.

`pname` specifies the texture parameter whose value or values will be returned. The accepted parameter names are as follows:

`#GL_TEXTURE_WIDTH`
: params returns a single value, the width of the texture image. This value includes the border of the texture image. The initial value is 0.

`#GL_TEXTURE_HEIGHT`
: params returns a single value, the height of the texture image. This value includes the border of the texture image. The initial value is 0.

`#GL_TEXTURE_DEPTH`
: params returns a single value, the depth of the texture image. This value includes the border of the texture image. The initial value is 0.

`#GL_TEXTURE_INTERNAL_FORMAT`
: params returns a single value, the internal format of the texture image.

`#GL_TEXTURE_BORDER`
: params returns a single value, the width in pixels of the border of the texture image. The initial value is 0.

`#GL_TEXTURE_XXX_SIZE`
: The internal storage resolution of an individual component (XXX can be RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, DEPTH). The resolution chosen by the GL will be a close match for the resolution requested by the user with the component argument of `gl.TexImage1D()`, `gl.TexImage2D()`, and `gl.CopyTexImage()`. The initial value is 0.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the symbolic name of the target texture, either `#GL_TEXTURE_1D`, `#GL_TEXTURE_2D`, `#GL_PROXY_TEXTURE_1D`, or `#GL_PROXY_TEXTURE_2D`

`level` specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level n is the nth mipmap reduction image

`pname` specifies the symbolic name of a texture parameter (see above for possible values)

RESULTS

`param` requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `target` or `pname` is not an accepted value.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than ld(max), where max is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexLevelParameter()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.74 gl.GetTexParameter

NAME

gl.GetTexParameter – return texture parameter values

SYNOPSIS

```
param = gl.GetTexParameter(target, pname)
```

FUNCTION

`gl.GetTexParameter()` returns the value or values of the texture parameter specified as `pname`. `target` defines the target texture, either `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`, to specify one- or two-dimensional texturing. `pname` accepts the same symbols as `gl.TexParameter()` with the same interpretations:

`#GL_TEXTURE_MAG_FILTER`
: Returns the single-valued texture magnification filter, a symbolic constant. The initial value is `#GL_LINEAR`.

`#GL_TEXTURE_MIN_FILTER`
: Returns the single-valued texture minification filter, a symbolic constant. The initial value is `#GL_NEAREST_MIPMAP_LINEAR`.

`#GL_TEXTURE_WRAP_S`
: Returns the single-valued wrapping function for texture coordinate s, a symbolic constant. The initial value is `#GL_REPEAT`.

`#GL_TEXTURE_WRAP_T`
: Returns the single-valued wrapping function for texture coordinate t, a symbolic constant. The initial value is `#GL_REPEAT`.

`#GL_TEXTURE_BORDER_COLOR`
: Returns four floating-point numbers that comprise the RGBA color of the texture border. The initial value is (0, 0, 0, 0).

`#GL_TEXTURE_PRIORITY`
: Returns the residence priority of the target texture (or the named texture bound to it). The initial value is 1. See Section 6.115 [gl.PrioritizeTextures], page 162, for details.

`#GL_TEXTURE_RESIDENT`
: Returns the residence status of the target texture. If the value returned in params is `#GL_TRUE`, the texture is resident in texture memory. See Section 6.3 [gl.AreTexturesResident], page 22, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the symbolic name of the target texture; `#GL_TEXTURE_1D` and `#GL_TEXTURE_2D` are accepted

`pname` specifies the symbolic name of a texture parameter (see above for supported values)

RESULTS

`param` requested data

ERRORS

`#GL_INVALID_ENUM` is generated if `target` or `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.GetTexParameter()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.75 gl.Hint

NAME

gl.Hint – specify implementation-specific hints

SYNOPSIS

```
gl.Hint(target, mode)
```

FUNCTION

Certain aspects of GL behavior, when there is room for interpretation, can be controlled with hints. A hint is specified with two arguments. `target` is a symbolic constant indicating the behavior to be controlled, and `mode` is another symbolic constant indicating

the desired behavior. The initial value for each target is `#GL_DONT_CARE`. `mode` can be one of the following:

`#GL_FASTEST`
: The most efficient option should be chosen.

`#GL_NICEST`
: The most correct, or highest quality, option should be chosen.

`#GL_DONT_CARE`
: No preference.

Though the implementation aspects that can be hinted are well defined, the interpretation of the hints depends on the implementation. The hint aspects that can be specified with `target`, along with suggested semantics, are as follows:

`#GL_FOG_HINT`
: Indicates the accuracy of fog calculation. If per-pixel fog calculation is not efficiently supported by the GL implementation, hinting `#GL_DONT_CARE` or `#GL_FASTEST` can result in per-vertex calculation of fog effects.

`#GL_LINE_SMOOTH_HINT`
: Indicates the sampling quality of antialiased lines. If a larger filter function is applied, hinting `#GL_NICEST` can result in more pixel fragments being generated during rasterization.

`#GL_PERSPECTIVE_CORRECTION_HINT`
: Indicates the quality of color, texture coordinate, and fog coordinate interpolation. If perspective-corrected parameter interpolation is not efficiently supported by the GL implementation, hinting `#GL_DONT_CARE` or `#GL_FASTEST` can result in simple linear interpolation of colors and/or texture coordinates.

`#GL_POINT_SMOOTH_HINT`
: Indicates the sampling quality of antialiased points. If a larger filter function is applied, hinting `#GL_NICEST` can result in more pixel fragments being generated during rasterization.

`#GL_POLYGON_SMOOTH_HINT`
: Indicates the sampling quality of antialiased polygons. Hinting `#GL_NICEST` can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

The interpretation of hints depends on the implementation. Some implementations ignore `gl.Hint()` settings.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies a symbolic constant indicating the behavior to be controlled (see above for possible values)

`mode` specifies a symbolic constant indicating the desired behavior; `#GL_FASTEST`, `#GL_NICEST`, and `#GL_DONT_CARE` are accepted

ERRORS

`#GL_INVALID_ENUM` is generated if either target or mode is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.Hint()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

6.76 gl.Index

NAME

gl.Index – set the current color index

SYNOPSIS

```
gl.Index(c)
```

FUNCTION

`gl.Index()` updates the current (single-valued) color index. It takes one argument, the new value for the current color index.

The current index is stored as a floating-point value. The initial value is 1.

Index values outside the representable range of the color index buffer are not clamped. However, before an index is dithered (if enabled) and written to the frame buffer, it is converted to fixed-point format. Any bits in the integer portion of the resulting fixed-point value that do not correspond to bits in the frame buffer are masked out.

The current index can be updated at any time. In particular, `gl.Index()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`c` specifies the new value for the current color index

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_INDEX`

6.77 gl.IndexMask

NAME

gl.IndexMask – control the writing of individual bits in the color index buffers

SYNOPSIS

```
gl.IndexMask(mask)
```

FUNCTION

`gl.IndexMask()` controls the writing of individual bits in the color index buffers. The least significant n bits of mask, where n is the number of bits in a color index buffer, specify a mask. Where a 1 (one) appears in the mask, it's possible to write to the corresponding bit in the color index buffer (or buffers). Where a 0 (zero) appears, the corresponding bit is write-protected.

This mask is used only in color index mode, and it affects only the buffers currently selected for writing (See Section 6.34 [gl.DrawBuffer], page 57, for details.). Initially, all bits are enabled for writing.

Please consult an OpenGL reference manual for more information.

INPUTS

`mask` specifies a bit mask to enable and disable the writing of individual bits in the color index buffers; initially, the mask is all 1's.

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.IndexMask()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`

ASSOCIATED GETS

`gl.Get()` with argument `#GL_INDEX_WRITEMASK`

6.78 gl.IndexPointer

NAME

gl.IndexPointer – define an array of color indexes

SYNOPSIS

`gl.IndexPointer(indexArray)`

FUNCTION

`gl.IndexPointer()` specifies an array of color indexes to use when rendering. `indexArray` must be an array containing a number of floating-point values describing color indexes.

If you pass `Nil` in `indexArray`, the color index array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the color index array or defining a new one.

When a color index array is specified, it is saved as client-side state, in addition to the current vertex array buffer object binding.

To enable and disable the color index array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_INDEX_ARRAY`. If enabled, the color index array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Color indexes are not supported for interleaved vertex array formats (See Section 6.80 [gl.InterleavedArrays], page 117, for details.).

The color index array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.IndexPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.IndexPointer()` is typically implemented on the client side.

Color index array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`indexArray`
array of color indexes or `Nil` (see above)

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_INDEX_ARRAY`

`gl.Get()` with argument `#GL_INDEX_ARRAY_TYPE`

`gl.Get()` with argument `#GL_INDEX_ARRAY_STRIDE`

`gl.GetPointer()` with argument `#GL_INDEX_ARRAY_POINTER`

6.79 gl.InitNames

NAME

gl.InitNames – initialize the name stack

SYNOPSIS

`gl.InitNames()`

FUNCTION

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers. `gl.InitNames()` causes the name stack to be initialized to its default empty state.

The name stack is always empty while the render mode is not `#GL_SELECT`. Calls to `gl.InitNames()` while the render mode is not `#GL_SELECT` are ignored.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.InitNames()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_NAME_STACK_DEPTH`

6.80 gl.InterleavedArrays

NAME

gl.InterleavedArrays – simultaneously specify and enable several interleaved arrays

SYNOPSIS

`gl.InterleavedArrays(format, stride, data)`

FUNCTION

`gl.InterleavedArrays()` lets you specify and enable individual color, normal, texture and vertex arrays whose elements are part of a larger aggregate array element. For some implementations, this is more efficient than specifying the arrays separately.

`data` must be a pointer to a raw memory buffer allocated by Hollywood's `AllocMem()` function, containing the relevant array data. Use Hollywood's `GetMemPointer()` function to get the raw pointer address of memory blocks allocated by `AllocMem()`. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

If `stride` is 0, the aggregate elements are stored consecutively. Otherwise, `stride` bytes occur between the beginning of one aggregate array element and the beginning of the next aggregate array element.

`format` serves as a key describing the extraction of individual arrays from the aggregate array. If `format` contains a T, then texture coordinates are extracted from the interleaved array. If C is present, color values are extracted. If N is present, normal coordinates are extracted. Vertex coordinates are always extracted. The digits 2, 3, and 4 denote how many values are extracted. F indicates that values are extracted as floating-point values. Colors may also be extracted as 4 unsigned bytes if 4UB follows the C. If a color is extracted as 4 unsigned bytes, the vertex array element which follows is located at the first possible floating-point aligned address. The following symbolic constants are recognized for `format`:

```
#GL_V2F
#GL_V3F
#GL_C4UB_V2F
#GL_C4UB_V3F
#GL_C3F_V3F
#GL_N3F_V3F
#GL_C4F_N3F_V3F
#GL_T2F_V3F
#GL_T4F_V4F
#GL_T2F_C4UB_V3F
#GL_T2F_C3F_V3F
#GL_T2F_N3F_V3F
#GL_T2F_C4F_N3F_V3F
#GL_T4F_C4F_N3F_V4F
```

If `gl.InterleavedArrays()` is called while compiling a display list, it is not compiled into the list, and it is executed immediately.

Execution of `gl.InterleavedArrays()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.InterleavedArrays()` is typically implemented on the client side.

Vertex array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`format` specifies the type of array to enable (see above for supported formats)

`stride` specifies the offset in bytes between each aggregate array element

`data` raw memory pointer containing data

ERRORS

`#GL_INVALID_ENUM` is generated if `format` is not an accepted value.

6.81 gl.IsEnabled

NAME

gl.IsEnabled – test whether a capability is enabled

SYNOPSIS

```
bool = gl.IsEnabled(cap)
```

FUNCTION

`gl.IsEnabled()` returns `#GL_TRUE` if `cap` is an enabled capability and returns `#GL_FALSE` otherwise. Initially all capabilities except `#GL_DITHER` are disabled; `#GL_DITHER` is initially enabled.

The following capabilities are accepted for `cap`:

`#GL_ALPHA_TEST`
: See Section 6.2 [gl.AlphaFunc], page 20, for details.

`#GL_AUTO_NORMAL`
: See Section 6.44 [gl.EvalCoord], page 68, for details.

`#GL_BLEND`
: See Section 6.8 [gl.BlendFunc], page 28, for details. See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_CLIP_PLANEi`
: See Section 6.17 [gl.ClipPlane], page 35, for details.

`#GL_COLOR_ARRAY`
: See Section 6.21 [gl.ColorPointer], page 39, for details.

`#GL_COLOR_LOGIC_OP`
: See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_COLOR_MATERIAL`
: See Section 6.20 [gl.ColorMaterial], page 38, for details.

`#GL_CULL_FACE`
: See Section 6.25 [gl.CullFace], page 46, for details.

`#GL_DEPTH_TEST`
: See Section 6.28 [gl.DepthFunc], page 48, for details. See Section 6.30 [gl.DepthRange], page 49, for details.

`#GL_DITHER`
: See Section 6.40 [gl.Enable], page 66, for details.

`#GL_EDGE_FLAG_ARRAY`
: See Section 6.39 [gl.EdgeFlagPointer], page 65, for details.

`#GL_FOG` See Section 6.50 [gl.Fog], page 75, for details.

`#GL_INDEX_ARRAY`
See Section 6.78 [gl.IndexPointer], page 116, for details.

`#GL_INDEX_LOGIC_OP`
See Section 6.92 [gl.LogicOp], page 131, for details.

`#GL_LIGHTi`
See Section 6.85 [gl.LightModel], page 125, for details. See Section 6.84 [gl.Light], page 123, for details.

`#GL_LIGHTING`
See Section 6.95 [gl.Material], page 138, for details. See Section 6.85 [gl.LightModel], page 125, for details. See Section 6.84 [gl.Light], page 123, for details.

`#GL_LINE_SMOOTH`
See Section 6.87 [gl.LineWidth], page 128, for details.

`#GL_LINE_STIPPLE`
See Section 6.86 [gl.LineStipple], page 127, for details.

`#GL_MAP1_COLOR_4`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_INDEX`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_NORMAL`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_1`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_2`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_3`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP1_TEXTURE_COORD_4`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_COLOR_4`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_INDEX`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_NORMAL`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_1`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_2`
See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_3`
: See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_TEXTURE_COORD_4`
: See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_VERTEX_3`
: See Section 6.93 [gl.Map], page 133, for details.

`#GL_MAP2_VERTEX_4`
: See Section 6.93 [gl.Map], page 133, for details.

`#GL_NORMAL_ARRAY`
: See Section 6.100 [gl.NormalPointer], page 143, for details.

`#GL_NORMALIZE`
: See Section 6.99 [gl.Normal], page 142, for details.

`#GL_POINT_SMOOTH`
: See Section 6.107 [gl.PointSize], page 155, for details.

`#GL_POLYGON_SMOOTH`
: See Section 6.108 [gl.PolygonMode], page 156, for details.

`#GL_POLYGON_OFFSET_FILL`
: See Section 6.109 [gl.PolygonOffset], page 157, for details.

`#GL_POLYGON_OFFSET_LINE`
: See Section 6.109 [gl.PolygonOffset], page 157, for details.

`#GL_POLYGON_OFFSET_POINT`
: See Section 6.109 [gl.PolygonOffset], page 157, for details.

`#GL_POLYGON_STIPPLE`
: See Section 6.110 [gl.PolygonStipple], page 158, for details.

`#GL_RESCALE_NORMAL`
: See Section 6.99 [gl.Normal], page 142, for details.

`#GL_SCISSOR_TEST`
: See Section 6.128 [gl.Scissor], page 180, for details.

`#GL_STENCIL_TEST`
: See Section 6.131 [gl.StencilFunc], page 184, for details. See Section 6.133 [gl.StencilOp], page 186, for details.

`#GL_TEXTURE_1D`
: See Section 6.139 [gl.TexImage1D], page 192, for details.

`#GL_TEXTURE_2D`
: See Section 6.140 [gl.TexImage2D], page 196, for details.

`#GL_TEXTURE_COORD_ARRAY`
: See Section 6.135 [gl.TexCoordPointer], page 188, for details.

`#GL_TEXTURE_GEN_Q`
: See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_R`
: See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_S`
: See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_TEXTURE_GEN_T`
: See Section 6.137 [gl.TexGen], page 190, for details.

`#GL_VERTEX_ARRAY`
: See Section 6.147 [gl.VertexPointer], page 207, for details.

If an error is generated, `gl.IsEnabled()` returns 0.

Please consult an OpenGL reference manual for more information.

INPUTS

`cap` specifies a symbolic constant indicating a GL capability

RESULTS

`bool` `#GL_TRUE` or `#GL_FALSE`

ERRORS

`#GL_INVALID_ENUM` is generated if `cap` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.IsEnabled()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

6.82 gl.IsList

NAME

gl.IsList – determine if a name corresponds to a display list

SYNOPSIS

```
bool = gl.IsList(list)
```

FUNCTION

`gl.IsList()` returns `#GL_TRUE` if list is the name of a display list and returns `#GL_FALSE` if it is not, or if an error occurs.

A name returned by `gl.GenLists()`, but not yet associated with a display list by calling `gl.NewList()`, is not the name of a display list.

Please consult an OpenGL reference manual for more information.

INPUTS

`list` specifies a potential display list name

RESULTS

`bool` `#GL_TRUE` or `#GL_FALSE`

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.IsList()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

6.83 gl.IsTexture

NAME

gl.IsTexture – determine if a name corresponds to a texture

SYNOPSIS

```
bool = gl.IsTexture(texture)
```

FUNCTION

`gl.IsTexture()` returns `#GL_TRUE` if `texture` is currently the name of a texture. If `texture` is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, `gl.IsTexture()` returns `#GL_FALSE`.

A name returned by `gl.GenTextures()`, but not yet associated with a texture by calling `gl.BindTexture()`, is not the name of a texture.

Please consult an OpenGL reference manual for more information.

INPUTS

`texture` specifies a value that may be the name of a texture

RESULTS

`bool` `#GL_TRUE` or `#GL_FALSE`

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.IsTexture()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

6.84 gl.Light

NAME

gl.Light – set light source parameters

SYNOPSIS

```
gl.Light(light, pname, param)
```

FUNCTION

`gl.Light()` sets the values of individual light source parameters. `light` names the light and is a symbolic name of the form `#GL_LIGHTi`, where `i` ranges from 0 to the value of `#GL_MAX_LIGHTS - 1`. `pname` specifies one of ten light source parameters, again by symbolic name. `param` is either a single floating-point value or a table that contains several floating-point values. This depends on the `pname` parameter.

To enable and disable lighting calculation, call `gl.Enable()` and `gl.Disable()` with argument `#GL_LIGHTING`. Lighting is initially disabled. When it is enabled, light sources that are enabled contribute to the lighting calculation. Light source i is enabled and disabled using `gl.Enable()` and `gl.Disable()` with argument `#GL_LIGHTi`.

The ten light parameters are as follows:

`#GL_AMBIENT`

: `param` must contain four floating-point values that specify the ambient RGBA intensity of the light. The initial ambient light intensity is (0, 0, 0, 1).

`#GL_DIFFUSE`

`param` must contain four floating-point values that specify the diffuse RGBA intensity of the light. The initial value for `#GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1).

`#GL_SPECULAR`

`param` must contain four floating-point values that specify the specular RGBA intensity of the light. The initial value for `#GL_LIGHT0` is (1, 1, 1, 1); for other lights, the initial value is (0, 0, 0, 1).

`#GL_POSITION`

`param` must contain four floating-point values that specify the position of the light in homogeneous object coordinates. The position is transformed by the modelview matrix when `gl.Light()` is called (just as if it were a point), and it is stored in eye coordinates. If the w component of the position is 0, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is (0, 0, 1, 0); thus, the initial light source is directional, parallel to, and in the direction of the -z axis.

`#GL_SPOT_DIRECTION`

`param` must contain three floating-point values that specify the direction of the light in homogeneous object coordinates. The spot direction is transformed by the upper 3x3 of the modelview matrix when `gl.Light()` is called, and it is stored in eye coordinates. It is significant only when `#GL_SPOT_CUTOFF` is not 180, which it is initially. The initial direction is (0, 0, -1).

`#GL_SPOT_EXPONENT`

`param` must be a single floating-point value that specifies the intensity distribution of the light. Only values in the range (0, 128) are accepted. Effective light intensity is attenuated by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lighted, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source, regardless of the spot cutoff angle (see `#GL_SPOT_CUTOFF`, next paragraph). The initial spot exponent is 0, resulting in uniform light distribution.

`#GL_SPOT_CUTOFF`

`param` must be a single floating-point value that specifies the maximum spread angle of a light source. Only values in the range (0, 90) and the special value 180 are accepted. If the angle between the direction of the light and the direction from the light to the vertex being lighted is greater than the spot cutoff angle, the light is completely masked. Otherwise, its intensity is controlled by the spot exponent and the attenuation factors. The initial spot cutoff is 180, resulting in uniform light distribution.

`#GL_CONSTANT_ATTENUATION`

`param` must be a single floating-point value that specifies one of the three light attenuation factors. Only non-negative values are accepted. If the

light is positional, rather than directional, its intensity is attenuated by the reciprocal of the sum of the constant factor, the linear factor times the distance between the light and the vertex being lighted, and the quadratic factor times the square of the same distance. The initial attenuation factors are (1, 0, 0), resulting in no attenuation.

`#GL_LINEAR_ATTENUATION`
: See the documentation of `#GL_CONSTANT_ATTENUATION` above.

`#GL_QUADRATIC_ATTENUATION`
: See the documentation of `#GL_CONSTANT_ATTENUATION` above.

It is always the case that `#GL_LIGHTi` = `#GL_LIGHT0` + i.

Please consult an OpenGL reference manual for more information.

INPUTS

`light` specifies a light (see above)

`pname` specifies a single-valued light source parameter for light (see above)

`param` a single floating-point value or a table containing multiple floating-point values (depends on the `pname` parameter, see above)

ERRORS

`#GL_INVALID_ENUM` is generated if either `light` or `pname` is not an accepted value.

`#GL_INVALID_VALUE` is generated if a spot exponent value is specified outside the range (0, 128) , or if spot cutoff is specified outside the range (0, 90) (except for the special value 180), or if a negative attenuation factor is specified.

`#GL_INVALID_OPERATION` is generated if `gl.Light()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetLight()`

`gl.IsEnabled()` with argument `#GL_LIGHTING`

6.85 gl.LightModel

NAME

gl.LightModel – set the lighting model parameters

SYNOPSIS

`gl.LightModel(pname, param)`

FUNCTION

`gl.LightModel()` sets the lighting model parameter. `pname` names a parameter and params gives the new value. There are three lighting model parameters:

`#GL_LIGHT_MODEL_AMBIENT`
: `param` must contain four floating-point values that specify the ambient RGBA intensity of the entire scene. The initial ambient scene intensity is (0.2, 0.2, 0.2, 1.0).

`#GL_LIGHT_MODEL_LOCAL_VIEWER`

`param` must be a single floating-point value that specifies how specular reflection angles are computed. If `param` is 0, specular reflection angles take the view direction to be parallel to and in the direction of the -z axis, regardless of the location of the vertex in eye coordinates. Otherwise, specular reflections are computed from the origin of the eye coordinate system. The initial value is 0.

`#GL_LIGHT_MODEL_TWO_SIDE`

`param` must be a single floating-point value that specifies whether one- or two-sided lighting calculations are done for polygons. It has no effect on the lighting calculations for points, lines, or bitmaps. If `param` is 0, one-sided lighting is specified, and only the front material parameters are used in the lighting equation. Otherwise, two-sided lighting is specified. In this case, vertices of back-facing polygons are lighted using the back material parameters and have their normals reversed before the lighting equation is evaluated. Vertices of front-facing polygons are always lighted using the front material parameters, with no change to their normals. The initial value is 0.

In RGBA mode, the lighted color of a vertex is the sum of the material emission intensity, the product of the material ambient reflectance and the lighting model full-scene ambient intensity, and the contribution of each enabled light source. Each light source contributes the sum of three terms: ambient, diffuse, and specular. The ambient light source contribution is the product of the material ambient reflectance and the light's ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the light's diffuse intensity, and the dot product of the vertex's normal with the normalized vector from the vertex to the light source. The specular light source contribution is the product of the material specular reflectance, the light's specular intensity, and the dot product of the normalized vertex-to-eye and vertex-to-light vectors, raised to the power of the shininess of the material. All three light source contributions are attenuated equally based on the distance from the vertex to the light source and on light source direction, spread exponent, and spread cutoff angle. All dot products are replaced with 0 if they evaluate to a negative value.

The alpha component of the resulting lighted color is set to the alpha value of the material diffuse reflectance.

In color index mode, the value of the lighted index of a vertex ranges from the ambient to the specular values passed to `gl.Material()` using `#GL_COLOR_INDEXES`. Diffuse and specular coefficients, computed with a (.30, .59, .11) weighting of the lights' colors, the shininess of the material, and the same reflection and attenuation equations as in the RGBA case, determine how much above ambient the resulting index is.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies a single-valued lighting model parameter (see above)

`param` a single floating-point value or a table containing multiple floating-point values (depends on the `pname` parameter, see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.LightModel()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LIGHT_MODEL_AMBIENT`

`gl.Get()` with argument `#GL_LIGHT_MODEL_LOCAL_VIEWER`

`gl.Get()` with argument `#GL_LIGHT_MODEL_TWO_SIDE`

`gl.IsEnabled()` with argument `#GL_LIGHTING`

6.86 gl.LineStipple

NAME

gl.LineStipple – specify the line stipple pattern

SYNOPSIS

```
gl.LineStipple(factor, pattern)
```

FUNCTION

Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern `pattern`, the repeat count `factor`, and an integer stipple counter s.

Counter s is reset to 0 whenever `gl.Begin()` is called and before each line segment of a

```
gl.Begin(#GL_LINES)
gl.End()
```

sequence is generated. It is incremented after each fragment of a unit width aliased line segment is generated or after each i fragments of an i width line segment are generated. The i fragments associated with count s are masked out if

```
pattern bit s factor % 16
```

is 0, otherwise these fragments are sent to the frame buffer. Bit zero of pattern is the least significant bit.

Antialiased lines are treated as a sequence of 1*width rectangles for purposes of stippling. Whether rectangle s is rasterized or not depends on the fragment rule described for aliased lines, counting rectangles rather than groups of fragments.

To enable and disable line stippling, call `gl.Enable()` and `gl.Disable()` with argument `#GL_LINE_STIPPLE`. When enabled, the line stipple pattern is applied as described above. When disabled, it is as if the pattern were all 1's. Initially, line stippling is disabled.

Alternatively, you can also pass a string consisting of 16 characters that are either 0 or 1 in `pattern`, e.g. `"1111000011110000"`.

Please consult an OpenGL reference manual for more information.

INPUTS

`factor` specifies a multiplier for each bit in the line stipple pattern; if factor is 3, for example, each bit in the pattern is used three times before the next bit in the pattern is used; factor is clamped to the range [1, 256] and defaults to 1.

`pattern` specifies a 16-bit integer whose bit pattern determines which fragments of a line will be drawn when the line is rasterized; bit zero is used first; the default pattern is all 1's.

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.LineStipple()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LINE_STIPPLE_PATTERN`

`gl.Get()` with argument `#GL_LINE_STIPPLE_REPEAT`

`gl.IsEnabled()` with argument `#GL_LINE_STIPPLE`

6.87 gl.LineWidth

NAME

gl.LineWidth – specify the width of rasterized lines

SYNOPSIS

`gl.LineWidth(width)`

FUNCTION

`gl.LineWidth()` specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1 has different effects, depending on whether line antialiasing is enabled. To enable and disable line antialiasing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_LINE_SMOOTH`. Line antialiasing is initially disabled.

If line antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer. (If the rounding results in the value 0, it is as if the line width were 1.) If delta x >= delta y , i pixels are filled in each column that is rasterized, where i is the rounded value of width. Otherwise, i pixels are filled in each row that is rasterized.

If antialiasing is enabled, line rasterization produces a fragment for each pixel square that intersects the region lying within the rectangle having width equal to the current line width, length equal to the actual length of the line, and centered on the mathematical line segment. The coverage value for each fragment is the window coordinate area of the intersection of the rectangular region with the corresponding pixel square. This value is saved and used in the final rasterization step.

Not all widths can be supported when line antialiasing is enabled. If an unsupported width is requested, the nearest supported width is used. Only width 1 is guaranteed to be supported; others depend on the implementation. Likewise, there is a range for aliased line widths as well. To query the range of supported widths and the size difference between supported widths within the range, call `gl.Get()` with arguments `#GL_LINE_WIDTH_RANGE` and `#GL_LINE_WIDTH_GRANULARITY`.

The line width specified by `gl.LineWidth()` is always returned when `#GL_LINE_WIDTH` is queried. Clamping and rounding for aliased and antialiased lines have no effect on the specified value.

Nonantialiased line width may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased lines, rounded to the nearest integer value.

Please consult an OpenGL reference manual for more information.

INPUTS

`width` specifies the width of rasterized lines; the initial value is 1

ERRORS

`#GL_INVALID_VALUE` is generated if `width` is less than or equal to 0.

`#GL_INVALID_OPERATION` is generated if `gl.LineWidth()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LINE_WIDTH`

`gl.Get()` with argument `#GL_LINE_WIDTH_RANGE`

`gl.Get()` with argument `#GL_LINE_WIDTH_GRANULARITY`

`gl.IsEnabled()` with argument `#GL_LINE_SMOOTH`

6.88 gl.ListBase

NAME

gl.ListBase – set the display-list base for gl.CallLists()

SYNOPSIS

```
gl.ListBase(base)
```

FUNCTION

`gl.CallLists()` specifies an array of offsets. Display-list names are generated by adding base to each offset. Names that reference valid display lists are executed; the others are ignored.

Please consult an OpenGL reference manual for more information.

INPUTS

`base` specifies an integer offset that will be added to `gl.CallLists()` offsets to generate display-list names; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.ListBase()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LIST_BASE`

6.89 gl.LoadIdentity

NAME
gl.LoadIdentity – replace the current matrix with the identity matrix

SYNOPSIS
`gl.LoadIdentity()`

FUNCTION
`gl.LoadIdentity()` replaces the current matrix with the identity matrix. It is semantically equivalent to calling `gl.LoadMatrix()` with the identity matrix but in some cases it is more efficient.

Please consult an OpenGL reference manual for more information.

INPUTS
none

ERRORS
`#GL_INVALID_OPERATION` is generated if `gl.LoadIdentity()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS
`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.90 gl.LoadMatrix

NAME
gl.LoadMatrix – replace the current matrix with the specified matrix

SYNOPSIS
`gl.LoadMatrix(mArray)`

FUNCTION
`gl.LoadMatrix()` replaces the current matrix with the one whose elements are specified in `mArray`. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (See Section 6.96 [gl.MatrixMode], page 139, for details.). `mArray` must store its values in column-major order.

Please consult an OpenGL reference manual for more information.

INPUTS

`mArray` specifies an array containing 16 consecutive values, which are used as the elements of a 4*4 column-major matrix

ERRORS
`#GL_INVALID_OPERATION` is generated if `gl.LoadMatrix()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()` .

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.91 gl.LoadName

NAME

gl.LoadName – load a name onto the name stack

SYNOPSIS

`gl.LoadName(name)`

FUNCTION

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

`gl.LoadName()` causes name to replace the value on the top of the name stack.

The name stack is always empty while the render mode is not `#GL_SELECT`. Calls to `gl.LoadName()` while the render mode is not `#GL_SELECT` are ignored.

Please consult an OpenGL reference manual for more information.

INPUTS

`name` specifies a name that will replace the top value on the name stack

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.LoadName()` is called while the name stack is empty.

`#GL_INVALID_OPERATION` is generated if `gl.LoadName()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_NAME_STACK_DEPTH`

6.92 gl.LogicOp

NAME

gl.LogicOp – specify a logical pixel operation for color index rendering

SYNOPSIS

`gl.LogicOp(opcode)`

FUNCTION

`gl.LogicOp()` specifies a logical operation that, when enabled, is applied between the incoming color index or RGBA color and the color index or RGBA color at the corresponding location in the frame buffer. To enable or disable the logical operation, call

`gl.Enable()` and `gl.Disable()` using the symbolic constant `#GL_COLOR_LOGIC_OP` for RGBA mode or `#GL_INDEX_LOGIC_OP` for color index mode. The initial value is disabled for both operations.

`#GL_CLEAR`
: 0

`#GL_SET`
: 1

`#GL_COPY`
: s

`#GL_COPY_INVERTED`
: ~s

`#GL_NOOP`
: d

`#GL_INVERT`
: ~d

`#GL_AND`
: s & d

`#GL_NAND`
: ~(s & d)

`#GL_OR`
: s | d

`#GL_NOR`
: ~(s | d)

`#GL_XOR`
: s ^ d

`#GL_EQUIV`
: ~(s ^ d)

`#GL_AND_REVERSE`
: s & ~d

`#GL_AND_INVERTED`
: ~s & d

`#GL_OR_REVERSE`
: s | ~d

`#GL_OR_INVERTED`
: ~s | d

`opcode` is a symbolic constant chosen from the list above. In the explanation of the logical operations, `s` represents the incoming color index and `d` represents the index in the frame buffer. Standard C-language operators are used. As these bitwise operators suggest, the logical operation is applied independently to each bit pair of the source and destination indices or colors.

Color index logical operations are always supported. RGBA logical operations are supported only if the GL version is 1.1 or greater.

When more than one RGBA color or index buffer is enabled for drawing, logical operations are performed separately for each enabled buffer, using for the destination value the contents of that buffer (See Section 6.34 [gl.DrawBuffer], page 57, for details.).

Please consult an OpenGL reference manual for more information.

INPUTS

`opcode` specifies a symbolic constant that selects a logical operation (see above for supported constants)

ERRORS

`#GL_INVALID_ENUM` is generated if `opcode` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.LogicOp()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_LOGIC_OP_MODE`.

`gl.IsEnabled()` with argument `#GL_COLOR_LOGIC_OP` or `#GL_INDEX_LOGIC_OP`.

6.93 gl.Map

NAME

gl.Map – define a one- or two-dimensional evaluator

SYNOPSIS

```
gl.Map(target, u1, u2, pointsArray)
gl.Map(target, u1, u2, v1, v2, pointsArray)
```

FUNCTION

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of GL processing just as if they had been presented using `gl.Vertex()`, `gl.Normal()`, `gl.TexCoord()`, and `gl.Color()` commands, except that the generated values do not update the current normal, texture coordinates, or color.

All polynomial or rational polynomial splines of any degree (up to the maximum degree supported by the GL implementation) can be described using evaluators. These include almost all splines used in computer graphics: B-splines, Bezier curves, Hermite splines, and so on.

`gl.Map()` is used to define the basis and to specify what kind of values are produced. Once defined, a map can be enabled and disabled by calling `gl.Enable()` and `gl.Disable()` with the map name, one of the nine predefined values for target described below. `gl.EvalCoord()` evaluates the one-dimensional maps that are enabled. When `gl.EvalCoord()` presents a value u or values u and v, the Bernstein functions are evaluated using u^ and v^ , where

```
u^ = (u - u1) / (u2 - u1)
v^ = (v - v1) / (v2 - v1)
```

`target` is a symbolic constant that indicates what kind of control points are provided in `pointsArray`, and what output is generated when the map is evaluated. In one-dimensional mode it can assume one of the following nine predefined values:

`#GL_MAP1_VERTEX_3`
: Each control point is three floating-point values representing x, y, and z. Internal `gl.Vertex()` commands are generated when the map is evaluated.

`#GL_MAP1_VERTEX_4`
: Each control point is four floating-point values representing x, y, z, and w. Internal `gl.Vertex()` commands are generated when the map is evaluated.

`#GL_MAP1_INDEX`
: Each control point is a single floating-point value representing a color index. Internal `gl.Index()` commands are generated when the map is evaluated but the current index is not updated with the value of these `gl.Index()` commands.

`#GL_MAP1_COLOR_4`
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal `gl.Color()` commands are generated when the map is evaluated but the current color is not updated with the value of these `gl.Color()` commands.

`#GL_MAP1_NORMAL`
: Each control point is three floating-point values representing the x, y, and z components of a normal vector. Internal `gl.Normal()` commands are generated when the map is evaluated but the current normal is not updated with the value of these `gl.Normal()` commands.

`#GL_MAP1_TEXTURE_COORD_1`
: Each control point is a single floating-point value representing the s texture coordinate. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP1_TEXTURE_COORD_2`
: Each control point is two floating-point values representing the s and t texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP1_TEXTURE_COORD_3`
: Each control point is three floating-point values representing the s, t, and r texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP1_TEXTURE_COORD_4`
: Each control point is four floating-point values representing the s, t, r, and q texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

In two-dimensional mode the following predefined constants are supported:

`#GL_MAP2_VERTEX_3`
: Each control point is three floating-point values representing x, y, and z. Internal `gl.Vertex()` commands are generated when the map is evaluated.

`#GL_MAP2_VERTEX_4`
: Each control point is four floating-point values representing x, y, z, and w. Internal `gl.Vertex()` commands are generated when the map is evaluated.

`#GL_MAP2_INDEX`
: Each control point is a single floating-point value representing a color index. Internal `gl.Index()` commands are generated when the map is evaluated but the current index is not updated with the value of these `gl.Index()` commands.

`#GL_MAP2_COLOR_4`
: Each control point is four floating-point values representing red, green, blue, and alpha. Internal `gl.Color()` commands are generated when the map is evaluated but the current color is not updated with the value of these `gl.Color()` commands.

`#GL_MAP2_NORMAL`
: Each control point is three floating-point values representing the x, y, and z components of a normal vector. Internal `gl.Normal()` commands are generated when the map is evaluated but the current normal is not updated with the value of these `gl.Normal()` commands.

`#GL_MAP2_TEXTURE_COORD_1`
: Each control point is a single floating-point value representing the s texture coordinate. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP2_TEXTURE_COORD_2`
: Each control point is two floating-point values representing the s and t texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP2_TEXTURE_COORD_3`
: Each control point is three floating-point values representing the s, t, and r texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

`#GL_MAP2_TEXTURE_COORD_4`
: Each control point is four floating-point values representing the s, t, r, and q texture coordinates. Internal `gl.TexCoord()` commands are generated when the map is evaluated but the current texture coordinates are not updated with the value of these `gl.TexCoord()` commands.

Initially, `#GL_AUTO_NORMAL` is enabled. If `#GL_AUTO_NORMAL` is enabled, normal vectors are generated when either `#GL_MAP2_VERTEX_3` or `#GL_MAP2_VERTEX_4` is used to generate vertices.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the kind of values that are generated by the evaluator (see above)

`u1` specify a linear mapping of u, as presented to `gl.EvalCoord()`, to u^ , the variable that is evaluated by the equations specified by this command

`u2` specify a linear mapping of u, as presented to `gl.EvalCoord()`, to u^ , the variable that is evaluated by the equations specified by this command

`v1` specify a linear mapping of v, as presented to `gl.EvalCoord()`, to v^ , one of the two variables that are evaluated by the equations specified by this command

`v2` specify a linear mapping of v, as presented to `gl.EvalCoord()`, to v^ , one of the two variables that are evaluated by the equations specified by this command

`pointsArray`
specifies a table containing a number of control points (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `target` is not an accepted value.

`#GL_INVALID_VALUE` is generated if `u1` is equal to `u2`, or if `v1` is equal to `v2`.

`#GL_INVALID_OPERATION` is generated if `gl.Map()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetMap()`

`gl.Get()` with argument `#GL_MAX_EVAL_ORDER`

`gl.IsEnabled()` with argument `#GL_MAP1_VERTEX_3`

`gl.IsEnabled()` with argument `#GL_MAP1_VERTEX_4`

`gl.IsEnabled()` with argument `#GL_MAP1_INDEX`

`gl.IsEnabled()` with argument `#GL_MAP1_COLOR_4`

`gl.IsEnabled()` with argument `#GL_MAP1_NORMAL`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_1`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_2`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_3`

`gl.IsEnabled()` with argument `#GL_MAP1_TEXTURE_COORD_4`

`gl.IsEnabled()` with argument `#GL_MAP2_VERTEX_3`

`gl.IsEnabled()` with argument `#GL_MAP2_VERTEX_4`

`gl.IsEnabled()` with argument `#GL_MAP2_INDEX`

`gl.IsEnabled()` with argument `#GL_MAP2_COLOR_4`

`gl.IsEnabled()` with argument `#GL_MAP2_NORMAL`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_1`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_2`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_3`

`gl.IsEnabled()` with argument `#GL_MAP2_TEXTURE_COORD_4`

6.94 gl.MapGrid

NAME

gl.MapGrid – define a one- or two-dimensional mesh

SYNOPSIS

```
gl.MapGrid(un, u1, u2[, vn, v1, v2])
```

FUNCTION

`gl.MapGrid()` and `gl.EvalMesh()` are used together to efficiently generate and evaluate a series of evenly-spaced map domain values. `gl.EvalMesh()` steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by `gl.Map()`.

`gl.MapGrid()` specifies the linear grid mappings between the i (or i and j) integer grid coordinates, to the u (or u and v) floating-point evaluation map coordinates. See Section 6.93 [gl.Map], page 133, for details of how u and v coordinates are evaluated.

In one-dimensional mode, `gl.MapGrid()` specifies a single linear mapping such that integer grid coordinate 0 maps exactly to u1, and integer grid coordinate un maps exactly to u2. All other integer grid coordinates i are mapped so that

```
u = i(u2 - u1) / un + u1
```

In two-dimensional mode `gl.MapGrid()` specifies two such linear mappings. One maps integer grid coordinate i = 0 exactly to u1, and integer grid coordinate i = un exactly to u2. The other maps integer grid coordinate j = 0 exactly to v1, and integer grid coordinate j = vn exactly to v2. Other integer grid coordinates i and j are mapped such that

```
u = i(u2 - u1) / un + u1
v = j(v2 - v1) / vn + v1
```

The mappings specified by `gl.MapGrid()` are used identically by `gl.EvalMesh()` and `gl.EvalPoint()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`un`	specifies the number of partitions in the grid range interval [u1, u2]; must be positive
`u1`	specify the mappings for integer grid domain values i = 0 and i = un
`u2`	specify the mappings for integer grid domain values i = 0 and i = un
`vn`	optional: specifies the number of partitions in the grid range interval [v1, v2]
`v1`	optional: specify the mappings for integer grid domain values j = 0 and j = vn
`v2`	optional: specify the mappings for integer grid domain values j = 0 and j = vn

ERRORS

`#GL_INVALID_VALUE` is generated if either `un` or `vn` is not positive.

`#GL_INVALID_OPERATION` is generated if `gl.MapGrid()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` with argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` with argument `#GL_MAP2_GRID_SEGMENTS`

6.95 gl.Material

NAME

gl.Material – specify material parameters for the lighting model

SYNOPSIS

`gl.Material(face, pname, param)`

FUNCTION

`gl.Material()` assigns values to material parameters. There are two matched sets of material parameters. One, the front-facing set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, back-facing, is used to shade back-facing polygons only when two-sided lighting is enabled. See Section 6.85 [gl.LightModel], page 125, for details concerning one- and two-sided lighting calculations.

`gl.Material()` takes three arguments. The first, `face`, specifies whether the `#GL_FRONT` materials, the `#GL_BACK` materials, or both `#GL_FRONT_AND_BACK` materials will be modified. The second, `pname`, specifies which of several parameters in one or both sets will be modified. The third, `param`, specifies what value or values will be assigned to the specified parameter. It can be single floating-point value or a table containing multiple floating-point values.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in the `gl.LightModel()` reference page. The parameters that can be specified using `gl.Material()`, and their interpretations by the lighting equation, are as follows:

`#GL_AMBIENT`

`param` must be a table containing four floating-point values that specify the ambient RGBA reflectance of the material. The initial ambient reflectance for both front- and back-facing materials is (0.2, 0.2, 0.2, 1.0).

`#GL_DIFFUSE`

`param` must be a table containing four floating-point values that specify the diffuse RGBA reflectance of the material. The initial diffuse reflectance for both front- and back-facing materials is (0.8, 0.8, 0.8, 1.0).

`#GL_SPECULAR`

`param` must be a table containing four floating-point values that specify the specular RGBA reflectance of the material. The initial specular reflectance for both front- and back-facing materials is (0, 0, 0, 1).

`#GL_EMISSION`
: `param` must be a table containing four floating-point values that specify the RGBA emitted light intensity of the material. The initial emission intensity for both front- and back-facing materials is (0, 0, 0, 1).

`#GL_SHININESS`
: `param` must be a single floating-point value that specifies the RGBA specular exponent of the material. Only values in the range (0,128) are accepted. The initial specular exponent for both front- and back-facing materials is 0.

`#GL_AMBIENT_AND_DIFFUSE`
: Equivalent to calling `gl.Material()` twice with the same parameter values, once with `#GL_AMBIENT` and once with `#GL_DIFFUSE`.

`#GL_COLOR_INDEXES`
: `param` must be a table containing three floating-point values specifying the color indices for ambient, diffuse, and specular lighting. These three values, and `#GL_SHININESS`, are the only material values used by the color index mode lighting equation. See Section 6.85 [gl.LightModel], page 125, for a discussion of color index lighting.

The material parameters can be updated at any time. In particular, `gl.Material()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`. If only a single material parameter is to be changed per vertex, however, `gl.ColorMaterial()` is preferred over `gl.Material()` (See Section 6.20 [gl.ColorMaterial], page 38, for details.).

While the ambient, diffuse, specular and emission material parameters all have alpha components, only the diffuse alpha component is used in the lighting computation.

Please consult an OpenGL reference manual for more information.

INPUTS

`face` specifies which face or faces are being updated; must be one of `#GL_FRONT`, `#GL_BACK`, or `#GL_FRONT_AND_BACK`

`pname` specifies the material parameter of the face or faces that is being updated (see above)

`param` a floating-point value (or a table containing multiple floating-point values) that `pname` will be set to

ERRORS

`#GL_INVALID_ENUM` is generated if either `face` or `pname` is not an accepted value.

`#GL_INVALID_VALUE` is generated if a specular exponent outside the range (0,128) is specified.

ASSOCIATED GETS

`gl.GetMaterial()`

6.96 gl.MatrixMode

NAME

gl.MatrixMode – specify which matrix is the current matrix

SYNOPSIS

`gl.MatrixMode(mode)`

FUNCTION

`gl.MatrixMode()` sets the current matrix mode. `mode` can assume one of three values:

`#GL_MODELVIEW`
: Applies subsequent matrix operations to the modelview matrix stack.

`#GL_PROJECTION`
: Applies subsequent matrix operations to the projection matrix stack.

`#GL_TEXTURE`
: Applies subsequent matrix operations to the texture matrix stack.

To find out which matrix stack is currently the target of all matrix operations, call `gl.Get()` with argument `#GL_MATRIX_MODE`. The initial value is `#GL_MODELVIEW`.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies which matrix stack is the target for subsequent matrix operations (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.MatrixMode()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

6.97 gl.MultMatrix

NAME

gl.MultMatrix – multiply the current matrix with the specified matrix

SYNOPSIS

`gl.MultMatrix(mArray)`

FUNCTION

`gl.MultMatrix()` multiplies the current matrix with the one specified using mArray, and replaces the current matrix with the product.

The current matrix is determined by the current matrix mode (See Section 6.96 [gl.MatrixMode], page 139, for details.). It is either the projection matrix, modelview matrix, or the texture matrix.

While the elements of the matrix are specified with double precision, the GL may store or operate on these values in less-than-single precision.

In many computer languages, 4x4 arrays are represented in row-major order. The transformations just described represent these matrices in column-major order. The order of the multiplication is important. For example, if the current transformation is a rotation,

and `gl.MultMatrix()` is called with a translation matrix, the translation is done directly on the coordinates to be transformed, while the rotation is done on the results of that translation.

Please consult an OpenGL reference manual for more information.

INPUTS

`mArray` table containing 16 consecutive values that are used as the elements of a 4x4 column-major matrix

ERRORS

`#GL_INVALID_OPERATION` is generated if glMultMatrix is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.98 gl.NewList

NAME

gl.NewList – create or replace a display list

SYNOPSIS

```
gl.NewList(list, mode)
```

FUNCTION

Display lists are groups of GL commands that have been stored for subsequent execution. Display lists are created with `gl.NewList()`. All subsequent commands are placed in the display list, in the order issued, until `gl.EndList()` is called.

`gl.NewList()` has two arguments. The first argument, `list`, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with `gl.GenLists()` and tested for uniqueness with `gl.IsList()`. The second argument, `mode`, is a symbolic constant that can assume one of two values:

`#GL_COMPILE`
: Commands are merely compiled.

`#GL_COMPILE_AND_EXECUTE`
: Commands are executed as they are compiled into the display list.

Certain commands are not compiled into the display list but are executed immediately, regardless of the display list mode that is currently active. These commands are `gl.AreTexturesResident()`, `gl.ColorPointer()`, `gl.DeleteLists()`, `gl.DeleteTextures()`, and `gl.DisableClientState()`, `gl.EdgeFlagPointer()`, `gl.EnableClientState()`, and `gl.FeedbackBuffer()`, `gl.Finish()`, `gl.Flush()`, `gl.GenLists()`, `gl.GenTextures()`, `gl.IndexPointer()`, `gl.InterleavedArrays()`, `gl.IsEnabled()`, and also `gl.IsList()`, `gl.IsTexture()`, `gl.NormalPointer()`,

gl.PopClientAttrib(), and finally also gl.PixelStore(), gl.PushClientAttrib(), gl.ReadPixels(), gl.RenderMode(), gl.SelectBuffer(), gl.TexCoordPointer(), gl.VertexPointer(), and all of the gl.Get() commands.

When gl.EndList() is encountered, the display-list definition is completed by associating the list with the unique name list (specified in the gl.NewList() command). If a display list with name list already exists, it is replaced only when gl.EndList() is called.

gl.CallList() and gl.CallLists() can be entered into display lists. Commands in the display list or lists executed by gl.CallList() or gl.CallLists() are not included in the display list being created, even if the list creation mode is #GL_COMPILE_AND_EXECUTE.

A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in #GL_COMPILE mode, errors are not generated until the list is executed.

Please consult an OpenGL reference manual for more information.

INPUTS

list	specifies the display-list name as an integer
mode	specifies the compilation mode, which can be #GL_COMPILE or #GL_COMPILE_AND_EXECUTE

ERRORS

#GL_INVALID_VALUE is generated if list is 0.

#GL_INVALID_ENUM is generated if mode is not an accepted value.

#GL_INVALID_OPERATION is generated if gl.EndList() is called without a preceding gl.NewList(), or if gl.NewList() is called while a display list is being defined.

#GL_INVALID_OPERATION is generated if gl.NewList() or gl.EndList() is executed between the execution of gl.Begin() and the corresponding execution of gl.End().

#GL_OUT_OF_MEMORY is generated if there is insufficient memory to compile the display list. If the GL version is 1.1 or greater, no change is made to the previous contents of the display list, if any, and no other change is made to the GL state. (It is as if no attempt had been made to create the new display list.)

ASSOCIATED GETS

gl.IsList()

gl.Get() with argument #GL_LIST_INDEX

gl.Get() with argument #GL_LIST_MODE

6.99 gl.Normal

NAME

gl.Normal – set the current normal vector

SYNOPSIS

gl.Normal(nx, ny, nz)

FUNCTION

The current normal is set to the given floating-point coordinates whenever `gl.Normal()` is issued. The initial value of the current normal is the unit vector, (0, 0, 1). Alternatively, `gl.Normal()` can also be called with a single table element containing the x, y, z normal coordinates.

Normals specified with `gl.Normal()` need not have unit length. If `#GL_NORMALIZE` is enabled, then normals of any length specified with `gl.Normal()` are normalized after transformation. If `#GL_RESCALE_NORMAL` is enabled, normals are scaled by a scaling factor derived from the modelview matrix. `#GL_RESCALE_NORMAL` requires that the originally specified normals were of unit length, and that the modelview matrix contain only uniform scales for proper results. To enable and disable normalization, call `gl.Enable()` and `gl.Disable()` with either `#GL_NORMALIZE` or `#GL_RESCALE_NORMAL`. Normalization is initially disabled.

The current normal can be updated at any time. In particular, `gl.Normal()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`nx`	specify the x coordinate of the new current normal
`ny`	specify the y coordinate of the new current normal
`nz`	specify the z coordinate of the new current normal

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_NORMAL`

`gl.IsEnabled()` with argument `#GL_NORMALIZE`

`gl.IsEnabled()` with argument `#GL_RESCALE_NORMAL`

6.100 gl.NormalPointer

NAME

gl.NormalPointer – define an array of normals

SYNOPSIS

`gl.NormalPointer(normalArray[, type])`

FUNCTION

`gl.NormalPointer()` specifies an array of normals to use when rendering. `normalArray` can be either a one-dimensional table consisting of an arbitrary number of consecutive normals or a two-dimensional table consisting of an arbitrary number of subtables which contain a normal each. Every normal must contain x, y, and z coordinates specified as floating-point values. If `normalArray` is a one-dimensional table, you need to set the optional `type` argument to `#GL_TRUE`. If `normalArray` is a two-dimensional table, you can leave out the optional `type` argument or set it to `#GL_FALSE`.

If you pass `Nil` in `normalArray`, the normal array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the normal array or defining a new one.

To enable and disable the normal array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_NORMAL_ARRAY`. If enabled, the normal array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

The normal array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.NormalPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.NormalPointer()` is typically implemented on the client side.

Normal array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`normalArray`
: one- or two-dimensional table containing normal values or `Nil` (see above)

`type`
: optional: `#GL_TRUE` if the table in argument 1 is a one-dimensional table, else `#GL_FALSE` (defaults to `#GL_FALSE`)

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_NORMAL_ARRAY`

`gl.Get()` with argument `#GL_NORMAL_ARRAY_TYPE`

`gl.Get()` with argument `#GL_NORMAL_ARRAY_STRIDE`

`gl.GetPointer()` with argument `#GL_NORMAL_ARRAY_POINTER`

6.101 gl.Ortho

NAME

gl.Ortho – multiply the current matrix with an orthographic matrix

SYNOPSIS

```
gl.Ortho(left, right, bottom, top, zNear, zFar)
```

FUNCTION

`gl.Ortho()` describes a transformation that produces a parallel projection. The current matrix (See Section 6.96 [gl.MatrixMode], page 139, for details.) is multiplied by this matrix and the result replaces the current matrix, as if `gl.MultMatrix()` were called with the following matrix as its argument:

```
A 0 0 tx
0 B 0 ty
0 0 C tz
0 0 0 1
```

where

```
A = 2 / (right - left)
```

```
B = 2 / (top - bottom)
C = -2 / (far  - near)
tx = -(right + left) / (right  - left)
ty = -(top + bottom) / (top  - bottom)
tz = -(zFar + zNear) / (zFar  - zNear)
```

Typically, the matrix mode is `#GL_PROJECTION`, and (left, bottom, -zNear) and (right, top, -zNear) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). -zFar specifies the location of the far clipping plane. Both zNear and zFar can be either positive or negative.

Use `gl.PushMatrix()` and `gl.PopMatrix()` to save and restore the current matrix stack.

Please consult an OpenGL reference manual for more information.

INPUTS

`left`	specify the coordinate for the left vertical clipping plane
`right`	specify the coordinate for the right vertical clipping plane
`bottom`	specify the coordinate for the bottom horizontal clipping plane
`top`	specify the coordinate for the top horizontal clipping plane
`zNear`	specify the distance to the nearer depth clipping plane; this value is negative if the plane is to be behind the viewer
`zFar`	specify the distance to the farther depth clipping plane; this value is negative if the plane is to be behind the viewer

ERRORS

`#GL_INVALID_VALUE` is generated if left = right, or bottom = top, or zNear = zFar.

`#GL_INVALID_OPERATION` is generated if `gl.Ortho()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.102 gl.PassThrough

NAME

gl.PassThrough – place a marker in the feedback buffer

SYNOPSIS

`gl.PassThrough(token)`

FUNCTION

Feedback is a GL render mode. The mode is selected by calling `gl.RenderMode()` with `#GL_FEEDBACK`. When the GL is in feedback mode, no pixels are produced by

rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL. See Section 6.47 [gl.FeedbackBuffer], page 72, for a description of the feedback buffer and the values in it.

`gl.PassThrough()` inserts a user-defined marker in the feedback buffer when it is executed in feedback mode. `token` is returned as if it were a primitive; it is indicated with its own unique identifying value: `#GL_PASS_THROUGH_TOKEN`. The order of `gl.PassThrough()` commands with respect to the specification of graphics primitives is maintained.

`gl.PassThrough()` is ignored if the GL is not in feedback mode.

Please consult an OpenGL reference manual for more information.

INPUTS

`token` specifies a marker value to be placed in the feedback buffer following a `#GL_PASS_THROUGH_TOKEN`

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.PassThrough()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_RENDER_MODE`

6.103 gl.PixelMap

NAME

gl.PixelMap – set up pixel transfer maps

SYNOPSIS

```
gl.PixelMap(map, valuesArray)
```

FUNCTION

`gl.PixelMap()` sets up translation tables, or maps, used by `gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.DrawPixels()`, `gl.ReadPixels()`, `gl.TexImage()`, and also `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()`, and `gl.TexSubImage2D()`. Use of these maps is described completely in the `gl.PixelTransfer()` reference page, and partly in the reference pages for the pixel and texture image commands. Only the specification of the maps is described in this reference page.

`map` is a symbolic map name, indicating one of ten maps to set. `values` is a table that contains an array of values for the specified map name. The ten maps are as follows:

`#GL_PIXEL_MAP_I_TO_I`
: Maps color indices to color indices.

`#GL_PIXEL_MAP_S_TO_S`
: Maps stencil indices to stencil indices.

`#GL_PIXEL_MAP_I_TO_R`
: Maps color indices to red components.

`#GL_PIXEL_MAP_I_TO_G`
: Maps color indices to green components.

`#GL_PIXEL_MAP_I_TO_B`
: Maps color indices to blue components.

`#GL_PIXEL_MAP_I_TO_A`
: Maps color indices to alpha components.

`#GL_PIXEL_MAP_R_TO_R`
: Maps red components to red components.

`#GL_PIXEL_MAP_G_TO_G`
: Maps green components to green components.

`#GL_PIXEL_MAP_B_TO_B`
: Maps blue components to blue components.

`#GL_PIXEL_MAP_A_TO_A`
: Maps alpha components to alpha components.

The entries in a map are specified as floating-point numbers. Maps that store color component values (all but `#GL_PIXEL_MAP_I_TO_I` and `#GL_PIXEL_MAP_S_TO_S`) retain their values in floating-point format, with unspecified mantissa and exponent sizes. Floating-point values specified by `gl.PixelMap()` are converted directly to the internal floating-point format of these maps, then clamped to the range [0,1].

Maps that store indices, `#GL_PIXEL_MAP_I_TO_I` and `#GL_PIXEL_MAP_S_TO_S`, retain their values in fixed-point format, with an unspecified number of bits to the right of the binary point. Floating-point values specified by `gl.PixelMap()` are converted directly to the internal fixed-point format of these maps.

The following table shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indices must have mapsize = 2n for some n or the results are undefined. The maximum allowable size for each map depends on the implementation and can be determined by calling glGet with argument `#GL_MAX_PIXEL_MAP_TABLE`. The single maximum applies to all maps; it is at least 32.

Map	Lookup Index	Lookup Value	Def Size	Def Value
#GL_PIXEL_MAP_I_TO_I	color index	color index	1	0
#GL_PIXEL_MAP_S_TO_S	stencil index	stencil index	1	0
#GL_PIXEL_MAP_I_TO_R	color index	R	1	0
#GL_PIXEL_MAP_I_TO_G	color index	G	1	0
#GL_PIXEL_MAP_I_TO_B	color index	B	1	0
#GL_PIXEL_MAP_I_TO_A	color index	A	1	0
#GL_PIXEL_MAP_R_TO_R	R	R	1	0
#GL_PIXEL_MAP_G_TO_G	G	G	1	0
#GL_PIXEL_MAP_B_TO_B	B	B	1	0
#GL_PIXEL_MAP_A_TO_A	A	A	1	0

Please consult an OpenGL reference manual for more information.

INPUTS

`map` specifies a symbolic map name (see above for supported names)

`valuesArray`
specifies a table containing an array of values

ERRORS

`#GL_INVALID_ENUM` is generated if map is not an accepted value.

`#GL_INVALID_VALUE` is generated if mapsize is less than one or larger than `#GL_MAX_PIXEL_MAP_TABLE`.

`#GL_INVALID_VALUE` is generated if map is `#GL_PIXEL_MAP_I_TO_I`, `#GL_PIXEL_MAP_S_TO_S`, `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, or `#GL_PIXEL_MAP_I_TO_A`, and mapsize is not a power of two.

`#GL_INVALID_OPERATION` is generated if `gl.PixelMap()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetPixelMap()`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_I_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_S_TO_S_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_R_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_G_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_B_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_I_TO_A_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_R_TO_R_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_G_TO_G_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_B_TO_B_SIZE`

`gl.Get()` with argument `#GL_PIXEL_MAP_A_TO_A_SIZE`

`gl.Get()` with argument `#GL_MAX_PIXEL_MAP_TABLE`

6.104 gl.PixelStore

NAME

gl.PixelStore – set pixel storage modes

SYNOPSIS

`gl.PixelStore(pname, param)`

FUNCTION

`gl.PixelStore()` sets pixel storage modes that affect the operation of subsequent `gl.DrawPixels()` and `gl.ReadPixels()` as well as the unpacking of polygon stipple patterns (See Section 6.110 [gl.PolygonStipple], page 158, for details.), bitmaps (See Section 6.7 [gl.Bitmap], page 27, for details.), texture patterns (See Section 6.138 [gl.TexImage], page 192, for details.).

`pname` is a symbolic constant indicating the parameter to be set, and `param` is the new value. Six of the twelve storage parameters affect how pixel data is returned to client memory. They are as follows:

`#GL_PACK_SWAP_BYTES`
: If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists of bytes b0, b1, b2, b3, it is stored in memory as b3, b2, b1, b0 if `#GL_PACK_SWAP_BYTES` is true. `#GL_PACK_SWAP_BYTES` has no effect on the memory order of components within a pixel, only on the order of bytes within components or indices. For example, the three components of a `#GL_RGB` format pixel are always stored with red first, green second, and blue third, regardless of the value of `#GL_PACK_SWAP_BYTES`.

`#GL_PACK_LSB_FIRST`
: If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This parameter is significant for bitmap data only.

`#GL_PACK_ROW_LENGTH`
: If greater than 0, `#GL_PACK_ROW_LENGTH` defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping a certain number of components or indices. See an OpenGL reference manual for details.

`#GL_PACK_SKIP_PIXELS`
: This value is provided as a convenience to the programmer; it provides no functionality that cannot be duplicated simply by incrementing the pointer passed to `gl.ReadPixels()`. Setting `#GL_PACK_SKIP_PIXELS` to i is equivalent to incrementing the pointer by `in` components or indices, where n is the number of components or indices in each pixel.

`#GL_PACK_SKIP_ROWS`
: This value is provided as a convenience to the programmer; it provides no functionality that cannot be duplicated simply by incrementing the pointer passed to `gl.ReadPixels()`. Setting `#GL_PACK_SKIP_ROWS` to j is equivalent to incrementing the pointer by `jm` components or indices, where m is the number of components or indices per row, as just computed in the `#GL_PACK_ROW_LENGTH` section.

`#GL_PACK_ALIGNMENT`
: Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The other six of the twelve storage parameters affect how pixel data is read from client memory. These values are then significant for `gl.DrawPixels()`, `gl.TexImage()`, and furthermore also for `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()`, `gl.TexSubImage2D()`, `gl.Bitmap()`, and `gl.PolygonStipple()`. They are as follows:

#GL_UNPACK_SWAP_BYTES
: If true, byte ordering for multibyte color components, depth components, color indices, or stencil indices is reversed. That is, if a four-byte component consists of bytes b0 , b1 , b2 , b3 , it is taken from memory as b3 , b2 , b1 , b0 if #GL_UNPACK_SWAP_BYTES is true. #GL_UNPACK_SWAP_BYTES has no effect on the memory order of components within a pixel, only on the order of bytes within components or indices. For example, the three components of a #GL_RGB format pixel are always stored with red first, green second, and blue third, regardless of the value of #GL_UNPACK_SWAP_BYTES.

#GL_UNPACK_LSB_FIRST
: If true, bits are ordered within a byte from least significant to most significant; otherwise, the first bit in each byte is the most significant one. This is relevant only for bitmap data.

#GL_UNPACK_ROW_LENGTH
: If greater than 0, #GL_UNPACK_ROW_LENGTH defines the number of pixels in a row. If the first pixel of a row is placed at location p in memory, then the location of the first pixel of the next row is obtained by skipping a certain number of components or indices. See an OpenGL reference manual for details.

#GL_UNPACK_SKIP_PIXELS
: This value is provided as a convenience to the programmer; it provides no functionality that cannot be duplicated by incrementing the pointer passed to gl.DrawPixels(), or to gl.TexImage(), or also to gl.TexImage1D() and gl.TexImage2D(), additionally also to gl.TexSubImage(), or to gl.TexSubImage1D() and gl.TexSubImage2D(), gl.Bitmap(), or to gl.PolygonStipple() Setting #GL_UNPACK_SKIP_PIXELS to i is equivalent to incrementing the pointer by in components or indices, where n is the number of components or indices in each pixel.

#GL_UNPACK_SKIP_ROWS
: This value is provided as a convenience to the programmer; it provides no functionality that cannot be duplicated by incrementing the pointer passed to gl.DrawPixels(), or to gl.TexImage(), or also to gl.TexImage1D() and gl.TexImage2D(), additionally also to gl.TexSubImage(), or to gl.TexSubImage1D() and gl.TexSubImage2D(), gl.Bitmap(), or to gl.PolygonStipple() Setting #GL_UNPACK_SKIP_ROWS to j is equivalent to incrementing the pointer by jk components or indices, where k is the number of components or indices per row, as just computed in the #GL_UNPACK_ROW_LENGTH section.

#GL_UNPACK_ALIGNMENT
: Specifies the alignment requirements for the start of each pixel row in memory. The allowable values are 1 (byte-alignment), 2 (rows aligned to even-numbered bytes), 4 (word-alignment), and 8 (rows start on double-word boundaries).

The following table gives the type, initial value, and range of valid values for each storage parameter that can be set with gl.PixelStore().

pname	Type	Default	Valid Range
#GL_PACK_SWAP_BYTES	boolean	false	true or false
#GL_PACK_LSB_FIRST	boolean	false	true or false
#GL_PACK_ROW_LENGTH	integer	0	[0,oo)
#GL_PACK_SKIP_ROWS	integer	0	[0,oo)
#GL_PACK_SKIP_PIXELS	integer	0	[0,oo)
#GL_PACK_ALIGNMENT	integer	4	1, 2, 4, or 8
#GL_UNPACK_SWAP_BYTES	boolean	false	true or false
#GL_UNPACK_LSB_FIRST	boolean	false	true or false
#GL_UNPACK_ROW_LENGTH	integer	0	[0,oo)
#GL_UNPACK_SKIP_ROWS	integer	0	[0,oo)
#GL_UNPACK_SKIP_PIXELS	integer	0	[0,oo)
#GL_UNPACK_ALIGNMENT	integer	4	1, 2, 4, or 8

The pixel storage modes in effect when `gl.DrawPixels()`, `gl.ReadPixels()`, or `gl.TexImage()`, `gl.TexImage1D()`, or `gl.TexImage2D()`, or `gl.TexSubImage()`, `gl.TexSubImage1D()`, or also GL's `gl.TexSubImage2D()`, or `gl.Bitmap()`, or `gl.PolygonStipple()` is placed in a display list control the interpretation of memory data. The pixel storage modes in effect when a display list is executed are not significant.

Pixel storage modes are client state and must be pushed and restored using `gl.PushClientAttrib()` and `gl.PopClientAttrib()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the symbolic name of the parameter to be set (see above for possible modes)

`param` specifies the value that `pname` is set to

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_VALUE` is generated if a negative row length, pixel skip, or row skip value is specified, or if alignment is specified as other than 1, 2, 4, or 8.

`#GL_INVALID_OPERATION` is generated if `gl.PixelStore()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_PACK_SWAP_BYTES`

`gl.Get()` with argument `#GL_PACK_LSB_FIRST`

`gl.Get()` with argument `#GL_PACK_ROW_LENGTH`

`gl.Get()` with argument `#GL_PACK_SKIP_ROWS`

`gl.Get()` with argument `#GL_PACK_SKIP_PIXELS`

`gl.Get()` with argument `#GL_PACK_ALIGNMENT`

`gl.Get()` with argument `#GL_UNPACK_SWAP_BYTES`

`gl.Get()` with argument `#GL_UNPACK_LSB_FIRST`

`gl.Get()` with argument `#GL_UNPACK_ROW_LENGTH`

`gl.Get()` with argument `#GL_UNPACK_SKIP_ROWS`

`gl.Get()` with argument `#GL_UNPACK_SKIP_PIXELS`

`gl.Get()` with argument `#GL_UNPACK_ALIGNMENT`

6.105 gl.PixelTransfer

NAME

gl.PixelTransfer – set pixel transfer modes

SYNOPSIS

```
gl.PixelTransfer(pname, param)
```

FUNCTION

`gl.PixelTransfer()` sets pixel transfer modes that affect the operation of subsequent `gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.DrawPixels()`, `gl.ReadPixels()`, or also GL's `gl.TexImage()`, `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()`, and `gl.TexSubImage2D()` commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (`gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.ReadPixels()`) or in case they are unpacked from client memory (`gl.DrawPixels()`, `gl.TexImage()`, `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()`, and `gl.TexSubImage2D()`). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (See Section 6.104 [gl.PixelStore], page 148, for details.) control the unpacking of pixels being read from client memory, and the packing of pixels being written back into client memory.

Pixel transfer operations handle four fundamental pixel types: color, color index, depth, and stencil. Color pixels consist of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0 represents zero intensity and 1 represents full intensity. Color indices comprise a single fixed-point value, with unspecified precision to the right of the binary point. Depth pixels comprise a single floating-point value, with unspecified mantissa and exponent sizes, scaled such that 0.0 represents the minimum depth buffer value, and 1.0 represents the maximum depth buffer value. Finally, stencil pixels comprise a single fixed-point value, with unspecified precision to the right of the binary point.

The pixel transfer operations performed on the four basic pixel types are as follows:

Color
: Each of the four color components is multiplied by a scale factor, then added to a bias factor. That is, the red component is multiplied by `#GL_RED_SCALE`, then added to `#GL_RED_BIAS`; the green component is multiplied by `#GL_GREEN_SCALE`, then added to `#GL_GREEN_BIAS`; the blue component is multiplied by `#GL_BLUE_SCALE`, then added to `#GL_BLUE_BIAS`; and the alpha component is multiplied by `#GL_ALPHA_SCALE`, then added to `#GL_ALPHA_BIAS`. After all four color components are scaled and biased, each is clamped to the range [0, 1]. All color, scale, and bias values are specified

with `gl.PixelTransfer()`. If `#GL_MAP_COLOR` is true, each color component is scaled by the size of the corresponding color-to-color map, then replaced by the contents of that map indexed by the scaled component. That is, the red component is scaled by `#GL_PIXEL_MAP_R_TO_R_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_R_TO_R` indexed by itself. The green component is scaled by `#GL_PIXEL_MAP_G_TO_G_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_G_TO_G` indexed by itself. The blue component is scaled by `#GL_PIXEL_MAP_B_TO_B_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_B_TO_B` indexed by itself. And the alpha component is scaled by `#GL_PIXEL_MAP_A_TO_A_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_A_TO_A` indexed by itself. All components taken from the maps are then clamped to the range [0, 1]. `#GL_MAP_COLOR` is specified with `gl.PixelTransfer()`. The contents of the various maps are specified with `gl.PixelMap()`.

Color index

Each color index is shifted left by `#GL_INDEX_SHIFT` bits; any bits beyond the number of fraction bits carried by the fixed-point index are filled with zeros. If `#GL_INDEX_SHIFT` is negative, the shift is to the right, again zero filled. Then `#GL_INDEX_OFFSET` is added to the index. `#GL_INDEX_SHIFT` and `#GL_INDEX_OFFSET` are specified with `gl.PixelTransfer()`.

From this point, operation diverges depending on the required format of the resulting pixels. If the resulting pixels are to be written to a color index buffer, or if they are being read back to client memory in `#GL_COLOR_INDEX` format, the pixels continue to be treated as indices. If `#GL_MAP_COLOR` is true, each index is masked by 2^n - 1, where n is `#GL_PIXEL_MAP_I_TO_I_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_I_TO_I` indexed by the masked value. `#GL_MAP_COLOR` is specified with `gl.PixelTransfer()`. The contents of the index map is specified with glPixelMap.

If the resulting pixels are to be written to an RGBA color buffer, or if they are read back to client memory in a format other than `#GL_COLOR_INDEX`, the pixels are converted from indices to colors by referencing the four maps `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, and `#GL_PIXEL_MAP_I_TO_A`. Before being dereferenced, the index is masked by 2^n - 1, where n is `#GL_PIXEL_MAP_I_TO_R_SIZE` for the red map, `#GL_PIXEL_MAP_I_TO_G_SIZE` for the green map, `#GL_PIXEL_MAP_I_TO_B_SIZE` for the blue map, and `#GL_PIXEL_MAP_I_TO_A_SIZE` for the alpha map. All components taken from the maps are then clamped to the range [0, 1]. The contents of the four maps is specified with `gl.PixelMap()`.

Depth Each depth value is multiplied by `#GL_DEPTH_SCALE`, added to `#GL_DEPTH_BIAS`, then clamped to the range [0, 1].

Stencil Each index is shifted `#GL_INDEX_SHIFT` bits just as a color index is, then added to `#GL_INDEX_OFFSET`. If `#GL_MAP_STENCIL` is true, each index is masked by 2^n - 1, where n is `#GL_PIXEL_MAP_S_TO_S_SIZE`, then replaced by the contents of `#GL_PIXEL_MAP_S_TO_S` indexed by the masked value.

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the symbolic name of the pixel transfer parameter to be set (see above)

`param` specifies the value that `pname` is set to

ERRORS

`#GL_INVALID_ENUM` is generated if `pname` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.PixelTransfer()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MAP_COLOR`

`gl.Get()` with argument `#GL_MAP_STENCIL`

`gl.Get()` with argument `#GL_INDEX_SHIFT`

`gl.Get()` with argument `#GL_INDEX_OFFSET`

`gl.Get()` with argument `#GL_RED_SCALE`

`gl.Get()` with argument `#GL_RED_BIAS`

`gl.Get()` with argument `#GL_GREEN_SCALE`

`gl.Get()` with argument `#GL_GREEN_BIAS`

`gl.Get()` with argument `#GL_BLUE_SCALE`

`gl.Get()` with argument `#GL_BLUE_BIAS`

`gl.Get()` with argument `#GL_ALPHA_SCALE`

`gl.Get()` with argument `#GL_ALPHA_BIAS`

`gl.Get()` with argument `#GL_DEPTH_SCALE`

`gl.Get()` with argument `#GL_DEPTH_BIAS`

6.106 gl.PixelZoom

NAME

gl.PixelZoom – specify the pixel zoom factors

SYNOPSIS

```
gl.PixelZoom(xfactor, yfactor)
```

FUNCTION

`gl.PixelZoom()` specifies values for the x and y zoom factors. During the execution of `gl.DrawPixels()` or `gl.CopyPixels()`, if (xr,yr) is the current raster position, and a given element is in the mth row and nth column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

```
(xr+n*xfactor, yr+m*yfactor)
(xr+(n+1)*xfactor,yr+(m+1)*yfactor)
```

are candidates for replacement. Any pixel whose center lies on the bottom or left edge of this rectangular region is also modified.

Pixel zoom factors are not limited to positive values. Negative zoom factors reflect the resulting image about the current raster position.

Please consult an OpenGL reference manual for more information.

INPUTS

`xfactor` specify the x zoom factor for pixel write operations

`yfactor` specify the y zoom factor for pixel write operations

ERRORS

`#GL_INVALID_OPERATION` is generated if glPixelZoom is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ZOOM_X`

`gl.Get()` with argument `#GL_ZOOM_Y`

6.107 gl.PointSize

NAME

gl.PointSize – specify the diameter of rasterized points

SYNOPSIS

```
gl.PointSize(size)
```

FUNCTION

`gl.PointSize()` specifies the rasterized diameter of both aliased and antialiased points. Using a point size other than 1 has different effects, depending on whether point antialiasing is enabled. To enable and disable point antialiasing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_POINT_SMOOTH`. Point antialiasing is initially disabled.

If point antialiasing is disabled, the actual size is determined by rounding the supplied size to the nearest integer. (If the rounding results in the value 0, it is as if the point size were 1.) If the rounded size is odd, then the center point (x,y) of the pixel fragment that represents the point is computed as

```
(xw + 0.5, yw + 0.5)
```

where w subscripts indicate window coordinates. All pixels that lie within the square grid of the rounded size centered at (x,y) make up the fragment. If the size is even, the center point is

```
(xw + 0.5, yw + 0.5)
```

and the rasterized fragment's centers are the half-integer window coordinates within the square of the rounded size centered at (x,y). All pixel fragments produced in rasterizing a non-antialiased point are assigned the same associated data, that of the vertex corresponding to the point.

If antialiasing is enabled, then point rasterization produces a fragment for each pixel square that intersects the region lying within the circle having diameter equal to the current point size and centered at the point's (xw,yw). The coverage value for each

fragment is the window coordinate area of the intersection of the circular region with the corresponding pixel square. This value is saved and used in the final rasterization step. The data associated with each fragment is the data associated with the point being rasterized.

Not all sizes are supported when point antialiasing is enabled. If an unsupported size is requested, the nearest supported size is used. Only size 1 is guaranteed to be supported; others depend on the implementation. To query the range of supported sizes and the size difference between supported sizes within the range, call `gl.Get()` with arguments `#GL_POINT_SIZE_RANGE` and `#GL_POINT_SIZE_GRANULARITY`.

The point size specified by `gl.PointSize()` is always returned when `#GL_POINT_SIZE` is queried. Clamping and rounding for aliased and antialiased points have no effect on the specified value.

A non-antialiased point size may be clamped to an implementation-dependent maximum. Although this maximum cannot be queried, it must be no less than the maximum value for antialiased points, rounded to the nearest integer value.

Please consult an OpenGL reference manual for more information.

INPUTS

`size` specifies the diameter of rasterized points; the initial value is 1

ERRORS

`#GL_INVALID_VALUE` is generated if size is less than or equal to 0.

`#GL_INVALID_OPERATION` is generated if `gl.PointSize()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_POINT_SIZE`

`gl.Get()` with argument `#GL_POINT_SIZE_RANGE`

`gl.Get()` with argument `#GL_POINT_SIZE_GRANULARITY`

`gl.IsEnabled()` with argument `#GL_POINT_SMOOTH`

6.108 gl.PolygonMode

NAME

gl.PolygonMode – select a polygon rasterization mode

SYNOPSIS

```
gl.PolygonMode(face, mode)
```

FUNCTION

`gl.PolygonMode()` controls the interpretation of polygons for rasterization. `face` describes which polygons mode applies to: front-facing polygons (`#GL_FRONT`), back-facing polygons (`#GL_BACK`), or both (`#GL_FRONT_AND_BACK`). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

Three modes are defined and can be specified in `mode`:

`#GL_POINT`
: Polygon vertices that are marked as the start of a boundary edge are drawn as points. Point attributes such as `#GL_POINT_SIZE` and `#GL_POINT_SMOOTH` control the rasterization of the points. Polygon rasterization attributes other than `#GL_POLYGON_MODE` have no effect.

`#GL_LINE`
: Boundary edges of the polygon are drawn as line segments. They are treated as connected line segments for line stippling; the line stipple counter and pattern are not reset between segments (see glLineStipple). Line attributes such as `#GL_LINE_WIDTH` and `#GL_LINE_SMOOTH` control the rasterization of the lines. Polygon rasterization attributes other than `#GL_POLYGON_MODE` have no effect.

`#GL_FILL`
: The interior of the polygon is filled. Polygon attributes such as `#GL_POLYGON_STIPPLE` and `#GL_POLYGON_SMOOTH` control the rasterization of the polygon.

The initial value is `#GL_FILL` for both front- and back-facing polygons.

Vertices are marked as boundary or nonboundary with an edge flag. Edge flags are generated internally by the GL when it decomposes polygons; they can be set explicitly using `gl.EdgeFlag()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`face`
: specifies the polygons that mode applies to (see above)

`mode`
: specifies how polygons will be rasterized (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if either `face` or `mode` is not an accepted value.

`#GL_INVALID_OPERATION` is generated if `gl.PolygonMode()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_POLYGON_MODE`

6.109 gl.PolygonOffset

NAME

gl.PolygonOffset – set the scale and units used to calculate depth values

SYNOPSIS

```
gl.PolygonOffset(factor, units)
```

FUNCTION

When `#GL_POLYGON_OFFSET_FILL`, `#GL_POLYGON_OFFSET_LINE`, or `#GL_POLYGON_OFFSET_POINT` is enabled, each fragment's depth value will be offset after it is interpolated from the depth values of the appropriate vertices. The value of the offset is

factor * delta(Z) + r * units , where delta(Z) is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

`gl.PolygonOffset()` is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

`gl.PolygonOffset()` has no effect on depth coordinates placed in the feedback buffer.

`gl.PolygonOffset()` has no effect on selection.

Please consult an OpenGL reference manual for more information.

INPUTS

`factor` specifies a scale factor that is used to create a variable depth offset for each polygon; the initial value is 0

`units` is multiplied by an implementation-specific value to create a constant depth offset; the initial value is 0

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.PolygonOffset()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_POLYGON_OFFSET_FILL`, `#GL_POLYGON_OFFSET_LINE`, or `#GL_POLYGON_OFFSET_POINT`.

`gl.Get()` with argument `#GL_POLYGON_OFFSET_FACTOR` or `#GL_POLYGON_OFFSET_UNITS`.

6.110 gl.PolygonStipple

NAME

gl.PolygonStipple – set the polygon stippling pattern

SYNOPSIS

`gl.PolygonStipple(maskArray)`

FUNCTION

Polygon stippling, like line stippling (See Section 6.86 [gl.LineStipple], page 127, for details.), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

`maskArray` is a table containing a 32*32 stipple pattern stored as a monochrome bitmap that uses only 1 bit per pixel. The bitmap is passed in a table that consists of chunks of 8 pixels packed into one byte. Thus, for a 32*32 stipple pattern you'll have to pass a table that contains 128 byte elements containing 8 pixels each. This can be either a one-dimensional table containing 128 byte entries or a two-dimensional table containing 32 subtables of 4 byte entries each (those 4 byte entries describe a row of 32 pixels each.) The data is passed to the GL in a contiguous memory block without any padding or special alignments so make sure that no exotic settings with `gl.PixelStore()` are active because `gl.PolygonStipple()` expects the pattern data to be stored in memory just like the pixel data supplied to a `gl.DrawPixels()` call with height and width both

equal to 32, a pixel format of `#GL_COLOR_INDEX`, and data type of `#GL_BITMAP`. That is, the stipple pattern is represented as a 32x32 array of 1-bit color indices packed in unsigned bytes. `gl.PixelStore()` parameters like `#GL_UNPACK_SWAP_BYTES` and `#GL_UNPACK_LSB_FIRST` affect the assembling of the bits into a stipple pattern. Pixel transfer operations (shift, offset, pixel map) are not applied to the stipple image, however.

To enable and disable polygon stippling, call `gl.Enable()` and `gl.Disable()` with argument `#GL_POLYGON_STIPPLE`. Polygon stippling is initially disabled. If it's enabled, a rasterized polygon fragment with window coordinates xw and yw is sent to the next stage of the GL if and only if the (xw%32)th bit in the (yw%32)th row of the stipple pattern is 1 (one). When polygon stippling is disabled, it is as if the stipple pattern consists of all 1's.

Please consult an OpenGL reference manual for more information.

INPUTS

`maskArray`
: specifies a table that contains a 32x32 stipple pattern

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.PolygonStipple()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetPolygonStipple()`

`gl.IsEnabled()` with argument `#GL_POLYGON_STIPPLE`

6.111 gl.PopAttrib

NAME

gl.PopAttrib – pop the server attribute stack

SYNOPSIS

`gl.PopAttrib()`

FUNCTION

`gl.PopAttrib()` restores the values of the state variables saved with the last `gl.PushAttrib()` command. Those not saved are left unchanged.

See Section 6.116 [gl.PushAttrib], page 163, for a list of supported state variables.

It is an error to pop attributes off an empty stack. In that case, the error flag is set and no other change is made to GL state.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_STACK_UNDERFLOW` is generated if `gl.PopAttrib()` is called while the attribute stack is empty.

`#GL_INVALID_OPERATION` is generated if `gl.PopAttrib()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_ATTRIB_STACK_DEPTH`

6.112 gl.PopClientAttrib

NAME

gl.PopClientAttrib – pop the client attribute stack

SYNOPSIS

`gl.PopClientAttrib()`

FUNCTION

`gl.PopClientAttrib()` restores the values of the client-state variables saved with the last `gl.PushClientAttrib()`. Those not saved are left unchanged.

See Section 6.117 [gl.PushClientAttrib], page 168, for a list of supported client state variables.

It is an error to pop attributes off an empty stack. In that case, the error flag is set, and no other change is made to GL state.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_STACK_UNDERFLOW` is generated if `gl.PopClientAttrib()` is called while the attribute stack is empty.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_CLIENT_ATTRIB_STACK_DEPTH`

6.113 gl.PopMatrix

NAME

gl.PopMatrix – pop the current matrix stack

SYNOPSIS

`gl.PopMatrix()`

FUNCTION

`gl.PopMatrix()` pops the current matrix stack, replacing the current matrix with the one below it on the stack.

Initially, each of the stacks contains one matrix, an identity matrix.

It is an error to pop a matrix stack that contains only a single matrix. In that case, the error flag is set and no other change is made to GL state.

Please consult an OpenGL reference manual for more information.

INPUTS
none

ERRORS
`#GL_STACK_UNDERFLOW` is generated if `gl.PopMatrix()` is called while the current matrix stack contains only a single matrix.

`#GL_INVALID_OPERATION` is generated if `gl.PopMatrix()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS
`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

`gl.Get()` with argument `#GL_MODELVIEW_STACK_DEPTH`

`gl.Get()` with argument `#GL_PROJECTION_STACK_DEPTH`

`gl.Get()` with argument `#GL_TEXTURE_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_MODELVIEW_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_PROJECTION_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_TEXTURE_STACK_DEPTH`

6.114 gl.PopName

NAME
gl.PopName – pop the name stack

SYNOPSIS
`gl.PopName()`

FUNCTION
The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

`gl.PopName()` pops one name off the top of the stack.

The maximum name stack depth is implementation-dependent; call `#GL_MAX_NAME_STACK_DEPTH` to find out the value for a particular implementation. It is an error to pop a name off an empty stack. It is also an error to manipulate the name stack between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`. In any of these cases, the error flag is set and no other change is made to GL state.

The name stack is always empty while the render mode is not `#GL_SELECT`. Calls to `gl.PopName()` while the render mode is not `#GL_SELECT` are ignored.

Please consult an OpenGL reference manual for more information.

INPUTS
none

ERRORS

`#GL_STACK_UNDERFLOW` is generated if `gl.PopName()` is called while the name stack is empty.

`#GL_INVALID_OPERATION` is generated if `gl.PopName()` is executed between a call to glBegin and the corresponding call to glEnd.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_NAME_STACK_DEPTH`

6.115 gl.PrioritizeTextures

NAME

gl.PrioritizeTextures – set texture residence priority

SYNOPSIS

`gl.PrioritizeTextures(texturesArray, prioritiesArray)`

FUNCTION

`gl.PrioritizeTextures()` assigns the texture priorities given in `prioritiesArray` to the textures named in `texturesArray`.

The GL establishes a "working set" of textures that are resident in texture memory. These textures may be bound to a texture target much more efficiently than textures that are not resident. By specifying a priority for each texture, `gl.PrioritizeTextures()` allows applications to guide the GL implementation in determining which textures should be resident.

The priorities given in `prioritiesArray` are clamped to the range (0,1) before they are assigned. 0 indicates the lowest priority; textures with priority 0 are least likely to be resident. 1 indicates the highest priority; textures with priority 1 are most likely to be resident. However, textures are not guaranteed to be resident until they are used.

`gl.PrioritizeTextures()` silently ignores attempts to prioritize texture 0 or any texture name that does not correspond to an existing texture.

`gl.PrioritizeTextures()` does not require that any of the textures named by textures be bound to a texture target. `gl.TexParameter()` may also be used to set a texture's priority, but only if the texture is currently bound. This is the only way to set the priority of a default texture.

Please consult an OpenGL reference manual for more information.

INPUTS

`texturesArray`
: specifies an array containing the names of the textures to be prioritized

`prioritiesArray`
: specifies an array containing the texture priorities; a priority given in an element of priorities applies to the texture named by the corresponding element of textures

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.PrioritizeTextures()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexParameter()` with parameter name `#GL_TEXTURE_PRIORITY` retrieves the priority of a currently bound texture

6.116 gl.PushAttrib

NAME

gl.PushAttrib – push the server attribute stack

SYNOPSIS

```
gl.PushAttrib(mask)
```

FUNCTION

`gl.PushAttrib()` takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. mask is typically constructed by ORing several of these constants together. The special mask `#GL_ALL_ATTRIB_BITS` can be used to save all stackable states.

The symbolic mask constants and their associated GL state are as follows:

`#GL_ACCUM_BUFFER_BIT`
: Accumulation buffer clear value

`#GL_COLOR_BUFFER_BIT`
: `#GL_ALPHA_TEST` enable bit
: Alpha test function and reference value
: `#GL_BLEND` enable bit
: Blending source and destination functions
: Constant blend color
: Blending equation
: `#GL_DITHER` enable bit
: `#GL_DRAW_BUFFER` setting
: `#GL_COLOR_LOGIC_OP` enable bit
: `#GL_INDEX_LOGIC_OP` enable bit
: Logic op function
: Color mode and index mode clear values
: Color mode and index mode writemasks

`#GL_CURRENT_BIT`
: Current RGBA color
: Current color index
: Current normal vector
: Current texture coordinates

Current raster position
#GL_CURRENT_RASTER_POSITION_VALID flag
RGBA color associated with current raster position
Color index associated with current raster position
Texture coordinates associated with current raster position
#GL_EDGE_FLAG flag

#GL_DEPTH_BUFFER_BIT
#GL_DEPTH_TEST enable bit
Depth buffer test function
Depth buffer clear value
#GL_DEPTH_WRITEMASK enable bit

#GL_ENABLE_BIT
#GL_ALPHA_TEST flag
#GL_AUTO_NORMAL flag
#GL_BLEND flag
Enable bits for the user-definable clipping planes
#GL_COLOR_MATERIAL
#GL_CULL_FACE flag
#GL_DEPTH_TEST flag
#GL_DITHER flag
#GL_FOG flag
#GL_LIGHTi where 0 .le. i < #GL_MAX_LIGHTS
#GL_LIGHTING flag
#GL_LINE_SMOOTH flag
#GL_LINE_STIPPLE flag
#GL_COLOR_LOGIC_OP flag
#GL_INDEX_LOGIC_OP flag
#GL_MAP1_x where x is a map type
#GL_MAP2_x where x is a map type
#GL_NORMALIZE flag
#GL_POINT_SMOOTH flag
#GL_POLYGON_OFFSET_LINE flag
#GL_POLYGON_OFFSET_FILL flag
#GL_POLYGON_OFFSET_POINT flag
#GL_POLYGON_SMOOTH flag
#GL_POLYGON_STIPPLE flag
#GL_SCISSOR_TEST flag
#GL_STENCIL_TEST flag
#GL_TEXTURE_1D flag

#GL_TEXTURE_2D flag
Flags #GL_TEXTURE_GEN_x where x is S, T, R, or Q

#GL_EVAL_BIT
#GL_MAP1_x enable bits, where x is a map type
#GL_MAP2_x enable bits, where x is a map type
1D grid endpoints and divisions
2D grid endpoints and divisions
#GL_AUTO_NORMAL enable bit

#GL_FOG_BIT
#GL_FOG enable bit
Fog color
Fog density
Linear fog start
Linear fog end
Fog index
#GL_FOG_MODE value

#GL_HINT_BIT
#GL_PERSPECTIVE_CORRECTION_HINT setting
#GL_POINT_SMOOTH_HINT setting
#GL_LINE_SMOOTH_HINT setting
#GL_POLYGON_SMOOTH_HINT setting
#GL_FOG_HINT setting

#GL_LIGHTING_BIT
#GL_COLOR_MATERIAL enable bit
#GL_COLOR_MATERIAL_FACE value
Color material parameters that are tracking the current color
Ambient scene color
#GL_LIGHT_MODEL_LOCAL_VIEWER value
#GL_LIGHT_MODEL_TWO_SIDE setting
#GL_LIGHTING enable bit
Enable bit for each light
Ambient, diffuse, and specular intensity for each light
Direction, position, exponent, and cutoff angle for each light
Constant, linear, and quadratic attenuation factors for each light
Ambient, diffuse, specular, and emissive color for each material
Ambient, diffuse, and specular color indices for each material
Specular exponent for each material
#GL_SHADE_MODEL setting

#GL_LINE_BIT
: #GL_LINE_SMOOTH flag
: #GL_LINE_STIPPLE enable bit
: Line stipple pattern and repeat counter
: Line width

#GL_LIST_BIT
: #GL_LIST_BASE setting

#GL_PIXEL_MODE_BIT
: #GL_RED_BIAS and #GL_RED_SCALE settings
: #GL_GREEN_BIAS and #GL_GREEN_SCALE values
: #GL_BLUE_BIAS and #GL_BLUE_SCALE
: #GL_ALPHA_BIAS and #GL_ALPHA_SCALE
: #GL_DEPTH_BIAS and #GL_DEPTH_SCALE
: #GL_INDEX_OFFSET and #GL_INDEX_SHIFT values
: #GL_MAP_COLOR and #GL_MAP_STENCIL flags
: #GL_ZOOM_X and #GL_ZOOM_Y factors
: #GL_READ_BUFFER setting

#GL_POINT_BIT
: #GL_POINT_SMOOTH flag Point size

#GL_POLYGON_BIT
: #GL_CULL_FACE enable bit
: #GL_CULL_FACE_MODE value
: #GL_FRONT_FACE indicator
: #GL_POLYGON_MODE setting
: #GL_POLYGON_SMOOTH flag
: #GL_POLYGON_STIPPLE enable bit
: #GL_POLYGON_OFFSET_FILL flag
: #GL_POLYGON_OFFSET_LINE flag
: #GL_POLYGON_OFFSET_POINT flag
: #GL_POLYGON_OFFSET_FACTOR
: #GL_POLYGON_OFFSET_UNITS

#GL_POLYGON_STIPPLE_BIT
: Polygon stipple image

#GL_SCISSOR_BIT
: #GL_SCISSOR_TEST flag
: Scissor box

#GL_STENCIL_BUFFER_BIT
: #GL_STENCIL_TEST enable bit
: Stencil function and reference value

Stencil value mask

Stencil fail, pass, and depth buffer pass actions

Stencil buffer clear value

Stencil buffer writemask

`#GL_TEXTURE_BIT`

Enable bits for the four texture coordinates

Border color for each texture image

Minification function for each texture image

Magnification function for each texture image

Texture coordinates and wrap mode for each texture image

Color and mode for each texture environment

Enable bits `#GL_TEXTURE_GEN_x`, x is S, T, R, and Q

`#GL_TEXTURE_GEN_MODE` setting for S, T, R, and Q

`gl.TexGen()` plane equations for S, T, R, and Q

Current texture bindings (for example, `#GL_TEXTURE_2D_BINDING`)

`#GL_TRANSFORM_BIT`

Coefficients of the six clipping planes

Enable bits for the user-definable clipping planes

`#GL_MATRIX_MODE` value

`#GL_NORMALIZE` flag

`#GL_VIEWPORT_BIT`

Depth range (near and far)

Viewport origin and extent

It is an error to push attributes onto a full stack. In that case, the error flag is set and no other change is made to GL state.

Initially, the attribute stack is empty.

Not all values for GL state can be saved on the attribute stack. For example, render mode state, and select and feedback state cannot be saved. Client state must be saved with `gl.PushClientAttrib()`.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

Please consult an OpenGL reference manual for more information.

INPUTS

`mask` specifies a mask that indicates which attributes to save (see above)

ERRORS

`#GL_STACK_OVERFLOW` is generated if `gl.PushAttrib()` is called while the attribute stack is full.

`#GL_INVALID_OPERATION` is generated if `gl.PushAttrib()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_ATTRIB_STACK_DEPTH`

6.117 gl.PushClientAttrib

NAME

gl.PushClientAttrib – push the client attribute stack

SYNOPSIS

`gl.PushClientAttrib(mask)`

FUNCTION

`gl.PushClientAttrib()` takes one argument, a mask that indicates which groups of client-state variables to save on the client attribute stack. Symbolic constants are used to set bits in the mask. `mask` is typically constructed by specifying the bitwise-or of several of these constants together. The special mask `#GL_CLIENT_ALL_ATTRIB_BITS` can be used to save all stackable client state.

The symbolic mask constants and their associated GL client state are as follows:

`#GL_CLIENT_PIXEL_STORE_BIT`
: Pixel storage modes

`#GL_CLIENT_VERTEX_ARRAY_BIT`
: Vertex arrays (and enables)

It is an error to push attributes onto a full client attribute stack. In that case, the error flag is set, and no other change is made to GL state.

Initially, the client attribute stack is empty.

Not all values for GL client state can be saved on the attribute stack. For example, select and feedback state cannot be saved.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

Use gl.PushAttrib() to push state that is kept on the server. Only pixel storage modes and vertex array state may be pushed with `gl.PushClientAttrib()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`mask` specifies a mask that indicates which attributes to save (see above)

ERRORS

`#GL_STACK_OVERFLOW` is generated if `gl.PushClientAttrib()` is called while the attribute stack is full.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_CLIENT_ATTRIB_STACK_DEPTH`

6.118 gl.PushMatrix

NAME

gl.PushMatrix – push the current matrix stack

SYNOPSIS

`gl.PushMatrix()`

FUNCTION

There is a stack of matrices for each of the matrix modes. In `#GL_MODELVIEW` mode, the stack depth is at least 32. In the other modes, `#GL_COLOR`, `#GL_PROJECTION`, and `#GL_TEXTURE`, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

`gl.PushMatrix()` pushes the current matrix stack down by one, duplicating the current matrix. That is, after a `gl.PushMatrix()` call, the matrix on top of the stack is identical to the one below it.

Initially, each of the stacks contains one matrix, an identity matrix.

It is an error to push a full matrix stack. In that case, the error flag is set and no other change is made to GL state.

Please consult an OpenGL reference manual for more information.

INPUTS

none

ERRORS

`#GL_STACK_OVERFLOW` is generated if `gl.PushMatrix()` is called while the current matrix stack is full.

`#GL_INVALID_OPERATION` is generated if `gl.PushMatrix()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

`gl.Get()` with argument `#GL_MODELVIEW_STACK_DEPTH`

`gl.Get()` with argument `#GL_PROJECTION_STACK_DEPTH`

`gl.Get()` with argument `#GL_TEXTURE_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_MODELVIEW_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_PROJECTION_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_TEXTURE_STACK_DEPTH`

6.119 gl.PushName

NAME

gl.PushName – push the name stack

SYNOPSIS

`gl.PushName(name)`

FUNCTION

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

`gl.PushName()` causes name to be pushed onto the name stack.

The maximum name stack depth is implementation-dependent; call `#GL_MAX_NAME_STACK_DEPTH` to find out the value for a particular implementation. It is an error to push a name onto a full stack. It is also an error to manipulate the name stack between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`. In any of these cases, the error flag is set and no other change is made to GL state.

The name stack is always empty while the render mode is not `#GL_SELECT`. Calls to `gl.PushName()` while the render mode is not `#GL_SELECT` are ignored.

Please consult an OpenGL reference manual for more information.

INPUTS

`name` specifies a name that will be pushed onto the name stack

ERRORS

`#GL_STACK_OVERFLOW` is generated if `gl.PushName()` is called while the name stack is full.

`#GL_INVALID_OPERATION` is generated if `gl.PushName()` is executed between a call to glBegin and the corresponding call to glEnd.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` with argument `#GL_MAX_NAME_STACK_DEPTH`

6.120 gl.RasterPos

NAME

gl.RasterPos – specify the raster position for pixel operations

SYNOPSIS

`gl.RasterPos(x, y[, z, w])`

FUNCTION

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See Section 6.7 [gl.Bitmap], page 27, for details. See Section 6.36 [gl.DrawPixels], page 59, for details. See Section 6.22 [gl.CopyPixels], page 40, for details.

The current raster position consists of three window coordinates (x, y, z), a clip coordinate value (w), an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The w coordinate is a clip coordinate, because w is not projected to window coordinates.

The object coordinates presented by `gl.RasterPos()` are treated just like those of a `gl.Vertex()` command: They are transformed by the current modelview and projection matrices and passed to the clipping stage. If the vertex is not culled, then it is projected and scaled to window coordinates, which become the new current raster position, and the `#GL_CURRENT_RASTER_POSITION_VALID` flag is set. If the vertex is culled, then the valid bit is cleared and the current raster position and associated color and texture coordinates are undefined.

The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then `#GL_CURRENT_RASTER_COLOR` (in RGBA mode) or `#GL_CURRENT_RASTER_INDEX` (in color index mode) is set to the color produced by the lighting calculation (see `gl.Light()`, `gl.LightModel()`, and `gl.ShadeModel()`). If lighting is disabled, current color (in RGBA mode, state variable `#GL_CURRENT_COLOR`) or color index (in color index mode, state variable `#GL_CURRENT_INDEX`) is used to update the current raster color.

Likewise, `#GL_CURRENT_RASTER_TEXTURE_COORDS` is updated as a function of `#GL_CURRENT_TEXTURE_COORDS`, based on the texture matrix and the texture generation functions (See Section 6.137 [gl.TexGen], page 190, for details.). Finally, the distance from the origin of the eye coordinate system to the vertex as transformed by only the modelview matrix replaces `#GL_CURRENT_RASTER_DISTANCE`.

Initially, the current raster position is (0, 0, 0, 1), the current raster distance is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1), the associated color index is 1, and the associated texture coordinates are (0, 0, 0, 1). In RGBA mode, `#GL_CURRENT_RASTER_INDEX` is always 1; in color index mode, the current raster RGBA color always maintains its initial value.

The raster position is modified by `gl.RasterPos()` and `gl.Bitmap()`.

When the raster position coordinates are invalid, drawing commands that are based on the raster position are ignored (that is, they do not result in changes to GL state).

Calling `gl.DrawElements()` may leave the current color or index indeterminate. If `gl.RasterPos()` is executed while the current color or index is indeterminate, the current raster color or current raster index remains indeterminate.

Alternatively, `gl.RasterPos()` can also be called with a single table argument containing two to four coordinates to set as the new raster position.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the x object coordinates for the raster position
`y`	specify the y object coordinates for the raster position
`z`	optional: specify the z object coordinates for the raster position (defaults to 0)
`w`	optional: specify the w object coordinates for the raster position (defaults to 1)

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.RasterPos()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` with argument `#GL_CURRENT_RASTER_POSITION_VALID`

`gl.Get()` with argument `#GL_CURRENT_RASTER_DISTANCE`

`gl.Get()` with argument `#GL_CURRENT_RASTER_COLOR`

`gl.Get()` with argument `#GL_CURRENT_RASTER_INDEX`

`gl.Get()` with argument `#GL_CURRENT_RASTER_TEXTURE_COORDS`

6.121 gl.ReadBuffer

NAME

gl.ReadBuffer – select a color buffer source for pixels

SYNOPSIS

`gl.ReadBuffer(mode)`

FUNCTION

`gl.ReadBuffer()` specifies a color buffer as the source for subsequent `gl.ReadPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, and `gl.CopyPixels()` commands. `mode` accepts one of twelve or more predefined values. `#GL_AUX0` through `#GL_AUX3` are always defined. In a fully configured system, `#GL_FRONT`, `#GL_LEFT`, and `#GL_FRONT_LEFT` all name the front left buffer, `#GL_FRONT_RIGHT` and `#GL_RIGHT` name the front right buffer, and `#GL_BACK_LEFT` and `#GL_BACK` name the back left buffer.

Non-stereo double-buffered configurations have only a front left and a back left buffer. Single-buffered configurations have a front left and a front right buffer if stereo, and only a front left buffer if nonstereo. It is an error to specify a non-existent buffer to `gl.ReadBuffer()`.

`mode` is initially `#GL_FRONT` in single-buffered configurations and `#GL_BACK` in double-buffered configurations.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies a color buffer (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is not one of the twelve (or more) accepted values.

`#GL_INVALID_OPERATION` is generated if `mode` specifies a buffer that does not exist.

`#GL_INVALID_OPERATION` is generated if `gl.ReadBuffer()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_READ_BUFFER`

6.122 gl.ReadPixels

NAME

gl.ReadPixels – read a block of pixels from the frame buffer

SYNOPSIS

```
pixelsArray = gl.ReadPixels(x, y, width, height, format)
```

FUNCTION

`gl.ReadPixels()` returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (x, y), in a table. Several parameters control the processing of the pixel data before it is placed into the table. These parameters are set with three commands: `gl.PixelStore()`, `gl.PixelTransfer()`, and `gl.PixelMap()`. This reference page describes the effects on `gl.ReadPixels()` of most, but not all of the parameters specified by these three commands.

`gl.ReadPixels()` returns values from each pixel with lower left corner at (x + i, y + j) for 0 <= i < width and 0 <= j < height. This pixel is said to be the ith pixel in the jth row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

`gl.ReadPixels()` always uses type `#GL_FLOAT` to read the pixels from the frame buffer. For fine-tuned control over the data type of the pixel data, you can use `gl.ReadPixelsRaw()` instead. See Section 6.123 [gl.ReadPixelsRaw], page 175, for details.

`format` specifies the format for the returned pixel values; accepted values are:

`#GL_COLOR_INDEX`
: Color indices are read from the color buffer selected by gl.ReadBuffer(). Each index is converted to fixed point, shifted left or right depending on the value and sign of `#GL_INDEX_SHIFT`, and added to `#GL_INDEX_OFFSET`. If `#GL_MAP_COLOR` is `#GL_TRUE`, indices are replaced by their mappings in the table `#GL_PIXEL_MAP_I_TO_I`.

`#GL_STENCIL_INDEX`
: Stencil values are read from the stencil buffer. Each index is converted to fixed point, shifted left or right depending on the value and sign of `#GL_INDEX_SHIFT`, and added to `#GL_INDEX_OFFSET`. If `#GL_MAP_STENCIL` is `#GL_TRUE`, indices are replaced by their mappings in the table `#GL_PIXEL_MAP_S_TO_S`.

`#GL_DEPTH_COMPONENT`
: Depth values are read from the depth buffer. Each component is converted to floating point such that the minimum depth value maps to 0 and the maximum value maps to 1. Each component is then multiplied by `#GL_DEPTH_SCALE`, added to `#GL_DEPTH_BIAS`, and finally clamped to the range (0,1).

`#GL_RED`
: Processing differs depending on whether color buffers store color indices or RGBA color components. If color indices are stored, they are read from the color buffer selected by gl.ReadBuffer(). Each index is converted to

fixed point, shifted left or right depending on the value and sign of #GL_INDEX_SHIFT, and added to #GL_INDEX_OFFSET. Indices are then replaced by the red, green, blue, and alpha values obtained by indexing the tables #GL_PIXEL_MAP_I_TO_R, #GL_PIXEL_MAP_I_TO_G, #GL_PIXEL_MAP_I_TO_B, and #GL_PIXEL_MAP_I_TO_A. Each table must be of size 2^n , but n may be different for different tables. Before an index is used to look up a value in a table of size 2^n , it must be masked against 2^n - 1.

If RGBA color components are stored in the color buffers, they are read from the color buffer selected by gl.ReadBuffer(). Each color component is converted to floating point such that zero intensity maps to 0.0 and full intensity maps to 1.0. Each component is then multiplied by #GL_c_SCALE and added to #GL_c_BIAS, where c is RED, GREEN, BLUE, or ALPHA. Finally, if #GL_MAP_COLOR is #GL_TRUE, each component is clamped to the range (0,1), scaled to the size of its corresponding table, and is then replaced by its mapping in the table #GL_PIXEL_MAP_c_TO_c, where c is R, G, B, or A.

Unneeded data is then discarded. For example, #GL_RED discards the green, blue, and alpha components, while #GL_RGB discards only the alpha component. #GL_LUMINANCE computes a single-component value as the sum of the red, green, and blue components, and #GL_LUMINANCE_ALPHA does the same, while keeping alpha as a second value. The final values are clamped to the range (0,1).

#GL_GREEN
: See above in #GL_RED.

#GL_BLUE
: See above in #GL_RED.

#GL_ALPHA
: See above in #GL_RED.

#GL_RGB
: See above in #GL_RED.

#GL_RGBA
: See above in #GL_RED.

#GL_LUMINANCE
: See above in #GL_RED.

#GL_LUMINANCE_ALPHA
: See above in #GL_RED.

The shift, scale, bias, and lookup factors just described are all specified by gl.PixelTransfer(). The lookup table contents themselves are specified by gl.PixelMap().

Return values are placed in the table as follows. If format is #GL_COLOR_INDEX, #GL_STENCIL_INDEX, #GL_DEPTH_COMPONENT, #GL_RED, #GL_GREEN, #GL_BLUE, #GL_ALPHA, or #GL_LUMINANCE, a single floating-point value is returned. #GL_RGB returns three values, #GL_RGBA returns four values, and #GL_LUMINANCE_ALPHA returns two values for each pixel, with all values corresponding to a single pixel occupying contiguous space in data. Storage parameters set by gl.PixelStore(), such as #GL_PACK_LSB_FIRST and #GL_PACK_SWAP_BYTES, affect the way that data is written into memory. See Section 6.104 [gl.PixelStore], page 148, for details.

Values for pixels that lie outside the window connected to the current GL context are undefined.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the left coordinate of a rectangular block of pixels
`y`	specify the lower coordinate of a rectangular block of pixels
`width`	width of the pixel rectangle
`height`	height of the pixel rectangle
`format`	format of the pixel data (see above)

RESULTS

`pixelsArray`
a table containing the pixel data

ERRORS

`#GL_INVALID_ENUM` is generated if `format` is not an accepted value.

`#GL_INVALID_VALUE` is generated if either `width` or `height` is negative.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_COLOR_INDEX` and the color buffers store RGBA color components.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_STENCIL_INDEX` and there is no stencil buffer.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_DEPTH_COMPONENT` and there is no depth buffer.

`#GL_INVALID_OPERATION` is generated if `gl.ReadPixels()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_INDEX_MODE`

6.123 gl.ReadPixelsRaw

NAME

gl.ReadPixelsRaw – read a block of pixels from the frame buffer

SYNOPSIS

```
gl.ReadPixelsRaw(x, y, width, height, format, type, pixels)
```

FUNCTION

This does the same as `gl.ReadPixels()` but doesn't return the pixels in a table. Instead, the pixels are written directly to a memory block that has to be passed in `pixels`. This must be a memory buffer allocated by Hollywood's `AllocMem()` function and returned by `GetMemPointer()`. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

See Section 6.122 [gl.ReadPixels], page 173, for a list of supported types for the format parameter.

Additionally, `gl.ReadPixelsRaw()` also allows you to define the data type that should be used when reading pixels from the frame buffer. `type` can assume the following values: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`. `gl.ReadPixels()` always uses `#GL_FLOAT`. With `gl.ReadPixelsRaw()` you can adjust this parameter to your specific needs.

See Section 6.122 [gl.ReadPixels], page 173, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the left coordinate of a rectangular block of pixels
`y`	specify the lower coordinate of a rectangular block of pixels
`width`	width of the pixel rectangle
`height`	height of the pixel rectangle
`format`	format of the pixel data (see above)
`type`	specifies the data type of the pixel data (see above)
`pixels`	pointer to a memory buffer to write the pixels to

ERRORS

`#GL_INVALID_ENUM` is generated if `format` or `type` is not an accepted value.

`#GL_INVALID_VALUE` is generated if either `width` or `height` is negative.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_COLOR_INDEX` and the color buffers store RGBA color components.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_STENCIL_INDEX` and there is no stencil buffer.

`#GL_INVALID_OPERATION` is generated if `format` is `#GL_DEPTH_COMPONENT` and there is no depth buffer.

`#GL_INVALID_OPERATION` is generated if gl.ReadPixels() is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_INDEX_MODE`

6.124 gl.Rect

NAME

gl.Rect – draw a rectangle

SYNOPSIS

```
gl.Rect(x1, y1, x2, y2)
```

FUNCTION

`gl.Rect()` supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized as two consecutive pairs of (x,y) coordinates. Alternatively, you can also pass two tables containing (x,y) coordinates each to `gl.Rect()`. The resulting rectangle is defined in the z = 0 plane.

gl.Rect(x1, y1, x2, y2) is exactly equivalent to the following sequence:

```
gl.Begin(#GL_POLYGON)
gl.Vertex(x1, y1)
gl.Vertex(x2, y1)
gl.Vertex(x2, y2)
gl.Vertex(x1, y2)
gl.End()
```

Note that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counter-clockwise winding.

Please consult an OpenGL reference manual for more information.

INPUTS

`x1` specifies one vertex of the rectangle

`y1` specifies one vertex of the rectangle

`x2` specifies the opposite vertex of the rectangle

`y2` specifies the opposite vertex of the rectangle

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Rect()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

6.125 gl.RenderMode

NAME

gl.RenderMode – set rasterization mode

SYNOPSIS

```
r = gl.RenderMode(mode)
```

FUNCTION

`gl.RenderMode()` sets the rasterization mode. It takes one argument, `mode`, which can assume one of three predefined values:

`#GL_RENDER`
: Render mode. Primitives are rasterized, producing pixel fragments, which are written into the frame buffer. This is the normal mode and also the default mode.

`#GL_SELECT`
: Selection mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, a record of the names of primitives that would have been drawn if the render mode had been `#GL_RENDER` is returned in a select buffer, which must be created (See Section 6.129 [gl.SelectBuffer], page 181, for details.) before selection mode is entered.

`#GL_FEEDBACK`
: Feedback mode. No pixel fragments are produced, and no change to the frame buffer contents is made. Instead, the coordinates and attributes of vertices that would have been drawn if the render mode had been `#GL_RENDER`

is returned in a feedback buffer, which must be created (See Section 6.47 [gl.FeedbackBuffer], page 72, for details.) before feedback mode is entered.

The return value of `gl.RenderMode()` is determined by the render mode at the time `gl.RenderMode()` is called, rather than by `mode`. The values returned for the three render modes are as follows:

`#GL_RENDER`
0.

`#GL_SELECT`
The number of hit records transferred to the select buffer.

`#GL_FEEDBACK`
The number of values (not vertices) transferred to the feedback buffer.

See Section 6.129 [gl.SelectBuffer], page 181, for more details concerning selection operation.

See Section 6.47 [gl.FeedbackBuffer], page 72, for more details concerning feedback operation.

If an error is generated, `gl.RenderMode()` returns 0 regardless of the current render mode.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies the rasterization mode; the initial value is `#GL_RENDER` (see above)

RESULTS

`r` return code (see above)

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is not one of the three accepted values.

`#GL_INVALID_OPERATION` is generated if `gl.SelectBuffer()` is called while the render mode is `#GL_SELECT`, or if `gl.RenderMode()` is called with argument `#GL_SELECT` before `gl.SelectBuffer()` is called at least once.

`#GL_INVALID_OPERATION` is generated if `gl.FeedbackBuffer()` is called while the render mode is `#GL_FEEDBACK`, or if `gl.RenderMode()` is called with argument `#GL_FEEDBACK` before `gl.FeedbackBuffer()` is called at least once.

`#GL_INVALID_OPERATION` is generated if `gl.RenderMode()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_RENDER_MODE`

6.126 gl.Rotate

NAME

gl.Rotate – multiply the current matrix by a rotation matrix

SYNOPSIS

```
gl.Rotate(angle, x, y, z)
```

FUNCTION

`gl.Rotate()` produces a rotation of angle degrees around the vector (x,y,z). The current matrix (See Section 6.96 [gl.MatrixMode], page 139, for details.) is multiplied by a rotation matrix with the product replacing the current matrix.

If the matrix mode is either `#GL_MODELVIEW` or `#GL_PROJECTION`, all objects drawn after `gl.Rotate()` is called are rotated.

Use `gl.PushMatrix()` and `gl.PopMatrix()` to save and restore the unscaled coordinate system.

This rotation follows the right-hand rule, so if the vector (x,y,z) points toward the user, the rotation will be counterclockwise.

Please consult an OpenGL reference manual for more information.

INPUTS

`angle`	specifies the angle of rotation, in degrees
`x`	specify the x coordinate of a vector
`y`	specify the y coordinate of a vector
`z`	specify the z coordinate of a vector

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Rotate()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.127 gl.Scale

NAME

gl.Scale – multiply the current matrix by a general scaling matrix

SYNOPSIS

```
gl.Scale(x, y, z)
```

FUNCTION

`gl.Scale()` produces a nonuniform scaling along the x, y, and z axes. The three parameters indicate the desired scale factor along each of the three axes.

The current matrix (See Section 6.96 [gl.MatrixMode], page 139, for details.) is multiplied by this scale matrix, and the product replaces the current matrix as if `gl.MultMatrix()` were called with the following matrix as its argument:

```
x 0 0 0
```

```
0 y 0 0
0 0 z 0
0 0 0 1
```

If the matrix mode is either `#GL_MODELVIEW` or `#GL_PROJECTION`, all objects drawn after `gl.Scale()` is called are scaled.

Use `gl.PushMatrix()` and `gl.PopMatrix()` to save and restore the unscaled coordinate system.

If scale factors other than 1 are applied to the modelview matrix and lighting is enabled, lighting often appears wrong. In that case, enable automatic normalization of normals by calling `gl.Enable()` with the argument `#GL_NORMALIZE`.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify scale factor along the x axis
`y`	specify scale factor along the y axis
`z`	specify scale factor along the z axis

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Scale()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.128 gl.Scissor

NAME

gl.Scissor – define the scissor box

SYNOPSIS

```
gl.Scissor(x, y, width, height)
```

FUNCTION

`gl.Scissor()` defines a rectangle, called the scissor box, in window coordinates. The first two arguments, `x` and `y`, specify the lower left corner of the box. `width` and `height` specify the width and height of the box.

To enable and disable the scissor test, call `gl.Enable()` and `gl.Disable()` with argument `#GL_SCISSOR_TEST`. The test is initially disabled. While the test is enabled, only pixels that lie within the scissor box can be modified by drawing commands. Window coordinates have integer values at the shared corners of frame buffer pixels. gl.Scissor(0,0,1,1) allows modification of only the lower left pixel in the window, and gl.Scissor(0,0,0,0) doesn't allow modification of any pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the left corner of the scissor box; initally 0
`y`	specify the lower corner of the scissor box; initially 0
`width`	specify the width of the scissor box
`height`	specify the height of the scissor box

ERRORS

`#GL_INVALID_VALUE` is generated if either `width` or `height` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.Scissor()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_SCISSOR_BOX`

`gl.IsEnabled()` with argument `#GL_SCISSOR_TEST`

6.129 gl.SelectBuffer

NAME

gl.SelectBuffer – establish a buffer for selection mode values

SYNOPSIS

```
buffer = gl.SelectBuffer(size)
```

FUNCTION

`gl.SelectBuffer()` allocates a memory buffer of the size specified in the `size` argument and returns a pointer to this buffer. Values from the name stack will be written to this buffer (see `gl.InitNames()`, `gl.LoadName()`, `gl.PushName()`) when the rendering mode is `#GL_SELECT` (See Section 6.125 [gl.RenderMode], page 177, for details.). `gl.SelectBuffer()` must be issued before selection mode is enabled, and it must not be issued while the rendering mode is `#GL_SELECT`.

You can read values from the memory buffer returned by `gl.SelectBuffer()` by calling `gl.GetSelectBuffer()` or accessing the buffer directly using Hollywood's `GetMemPointer()` and `Peek()` functions. When accessing the buffer directly, please note that the first four bytes in the buffer contain the size of the selection buffer in bytes.

A programmer can use selection to determine which primitives are drawn into some region of a window. The region is defined by the current modelview and perspective matrices.

In selection mode, no pixel fragments are produced from rasterization. Instead, if a primitive or a raster position intersects the clipping volume defined by the viewing frustum and the user-defined clipping planes, this primitive causes a selection hit. (With polygons, no hit occurs if the polygon is culled.) When a change is made to the name

stack, or when `gl.RenderMode()` is called, a hit record is copied to buffer if any hits have occurred since the last such event (name stack change or `gl.RenderMode()` call). The hit record consists of the number of names in the name stack at the time of the event, followed by the minimum and maximum depth values of all vertices that hit since the previous event, followed by the name stack contents, bottom name first.

Depth values (which are in the range [0,1]) are multiplied by 2^32 - 1, before being placed in the hit record.

An internal index into buffer is reset to 0 whenever selection mode is entered. Each time a hit record is copied into buffer, the index is incremented to point to the cell just past the end of the block of names, that is, to the next available cell if the hit record is larger than the number of remaining locations in buffer, as much data as can fit is copied, and the overflow flag is set. If the name stack is empty when a hit record is copied, that record consists of 0 followed by the minimum and maximum depth values.

To exit selection mode, call `gl.RenderMode()` with an argument other than `#GL_SELECT`. Whenever `gl.RenderMode()` is called while the render mode is `#GL_SELECT`, it returns the number of hit records copied to buffer, resets the overflow flag and the selection buffer pointer, and initializes the name stack to be empty. If the overflow bit was set when `gl.RenderMode()` was called, a negative hit record count is returned.

The contents of buffer is undefined until `gl.RenderMode()` is called with an argument other than `#GL_SELECT`.

`gl.Begin()` / `gl.End()` primitives and calls to `gl.RasterPos()` can result in hits.

To free a buffer allocated by this function, call `gl.FreeSelectBuffer()`. See Section 6.52 [gl.FreeSelectBuffer], page 77, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`size` specifies the size of the buffer in bytes

RESULTS

`buffer` memory buffer to use in selection mode

ERRORS

`#GL_INVALID_VALUE` is generated if `size` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.SelectBuffer()` is called while the render mode is `#GL_SELECT`, or if `gl.RenderMode()` is called with argument `#GL_SELECT` before `gl.SelectBuffer()` is called at least once.

`#GL_INVALID_OPERATION` is generated if `gl.SelectBuffer()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` with argument `#GL_SELECTION_BUFFER_SIZE`

`gl.GetPointer()` with argument `#GL_SELECTION_BUFFER_POINTER`

6.130 gl.ShadeModel

NAME

gl.ShadeModel – select flat or smooth shading

SYNOPSIS

`gl.ShadeModel(mode)`

FUNCTION

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Starting when `gl.Begin()` is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment i the computed color of vertex i + 1, its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.

```
Primitive Type of Polygon i  |  Vertex
-------------------------------------
Single polygon (i == 1)      |     1
Triangle strip               |   i + 2
Triangle fan                 |   i + 2
Independent triangle         |    3i
Quad strip                   |  2i + 2
Independent quad             |    4i
```

Flat and smooth shading are specified by `gl.ShadeModel()` with mode set to `#GL_FLAT` and `#GL_SMOOTH`, respectively.

Please consult an OpenGL reference manual for more information.

INPUTS

`mode` specifies a symbolic value representing a shading technique; accepted values are `#GL_FLAT` and `#GL_SMOOTH`; the initial value is `#GL_SMOOTH`

ERRORS

`#GL_INVALID_ENUM` is generated if `mode` is any value other than `#GL_FLAT` or `#GL_SMOOTH`.

`#GL_INVALID_OPERATION` is generated if `gl.ShadeModel()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_SHADE_MODEL`

6.131 gl.StencilFunc

NAME

gl.StencilFunc – set function and reference value for stencil testing

SYNOPSIS

```
gl.StencilFunc(func, ref, mask)
```

FUNCTION

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. Stencil planes are first drawn into using GL drawing primitives, then geometry and images are rendered using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the reference value and the value in the stencil buffer. To enable and disable the test, call `gl.Enable()` and `gl.Disable()` with argument `#GL_STENCIL_TEST`. To specify actions based on the outcome of the stencil test, call `gl.StencilOp()`.

`func` is a symbolic constant that determines the stencil comparison function. It accepts one of eight values, shown in the following list. `ref` is an integer reference value that is used in the stencil comparison. It is clamped to the range (0,2^n-1) , where n is the number of bitplanes in the stencil buffer. `mask` is bitwise ANDed with both the reference value and the stored stencil value, with the ANDed values participating in the comparison.

If `stencil` represents the value stored in the corresponding stencil buffer location, the following list shows the effect of each comparison function that can be specified by func. Only if the comparison succeeds is the pixel passed through to the next stage in the rasterization process (See Section 6.133 [gl.StencilOp], page 186, for details.). All tests treat stencil values as unsigned integers in the range (0,2^n-1) , where n is the number of bitplanes in the stencil buffer.

The following values are accepted by `func`:

`#GL_NEVER`
: Always fails.

`#GL_LESS` Passes if (ref & mask) < (stencil & mask).

`#GL_LEQUAL`
: Passes if (ref & mask) <= (stencil & mask).

`#GL_GREATER`
: Passes if (ref & mask) > (stencil & mask).

`#GL_GEQUAL`
: Passes if (ref & mask) >= (stencil & mask).

`GL_EQUAL` Passes if (ref & mask) = (stencil & mask).

`GL_NOTEQUAL`
: Passes if (ref & mask) != (stencil & mask).

`#GL_ALWAYS`
: Always passes.

Initially, the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil test always passes.

Please consult an OpenGL reference manual for more information.

INPUTS

`func` specifies the test function; the initial value is `#GL_ALWAYS` (see above)

`ref` specifies the reference value for the stencil test; the initial value is 0

`mask` specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done; the initial value is all 1's

ERRORS

`#GL_INVALID_ENUM` is generated if `func` is not one of the eight accepted values.

`#GL_INVALID_OPERATION` is generated if `gl.StencilFunc()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_STENCIL_FUNC`, `#GL_STENCIL_VALUE_MASK`, `#GL_STENCIL_REF`, or `#GL_STENCIL_BITS`

`gl.IsEnabled()` with argument `#GL_STENCIL_TEST`

6.132 gl.StencilMask

NAME

gl.StencilMask – control the writing of individual bits in the stencil planes

SYNOPSIS

```
gl.StencilMask(mask)
```

FUNCTION

`gl.StencilMask()` controls the writing of individual bits in the stencil planes. The least significant n bits of mask, where n is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

Please consult an OpenGL reference manual for more information.

INPUTS

`mask` specifies a bit mask to enable and disable writing of individual bits in the stencil planes; initially, the mask is all 1's

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.StencilMask()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_STENCIL_WRITEMASK`, `#GL_STENCIL_BACK_WRITEMASK`, or `#GL_STENCIL_BITS`

6.133 gl.StencilOp

NAME

gl.StencilOp – set stencil test actions

SYNOPSIS

```
gl.StencilOp(fail, zfail, zpass)
```

FUNCTION

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

The stencil test conditionally eliminates a pixel based on the outcome of a comparison between the value in the stencil buffer and a reference value. To enable and disable the test, call `gl.Enable()` and `gl.Disable()` with argument `#GL_STENCIL_TEST`; to control it, call `gl.StencilFunc()`.

`gl.StencilOp()` takes three arguments that indicate what happens to the stored stencil value while stenciling is enabled. If the stencil test fails, no change is made to the pixel's color or depth buffers, and `fail` specifies what happens to the stencil buffer contents. The following eight actions are possible.

`#GL_KEEP` Keeps the current value.

`#GL_ZERO` Sets the stencil buffer value to 0.

`#GL_REPLACE`
: Sets the stencil buffer value to ref, as specified by `gl.StencilFunc()`.

`#GL_INCR` Increments the current stencil buffer value. Clamps to the maximum representable unsigned value.

`#GL_DECR` Decrements the current stencil buffer value. Clamps to 0.

`#GL_INVERT`
: Bitwise inverts the current stencil buffer value.

Stencil buffer values are treated as unsigned integers. When incremented and decremented, values are clamped to 0 and 2^n - 1, where n is the value returned by querying `#GL_STENCIL_BITS`.

The other two arguments to `gl.StencilOp()` specify stencil buffer actions that depend on whether subsequent depth buffer tests succeed (`zpass`) or fail (`zfail`) (See Section 6.28 [gl.DepthFunc], page 48, for details.). The actions are specified using the same eight symbolic constants as `fail`. Note that `zfail` is ignored when there is no depth buffer, or when the depth buffer is not enabled. In these cases, `fail` and `zpass` specify stencil action when the stencil test fails and passes, respectively.

Initially the stencil test is disabled. If there is no stencil buffer, no stencil modification can occur and it is as if the stencil tests always pass, regardless of any call to `gl.StencilOp()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`fail` specifies the action to take when the stencil test fails; the initial value is `#GL_KEEP` (see above)

`zfail` specifies the stencil action when the stencil test passes, but the depth test fails; `zfail` accepts the same symbolic constants as `fail`; the initial value is `#GL_KEEP`

`zpass` specifies the stencil action when both the stencil test and the depth test pass, or when the stencil test passes and either there is no depth buffer or depth testing is not enabled; `zpass` accepts the same symbolic constants as `fail`; the initial value is `#GL_KEEP`

ERRORS

`#GL_INVALID_ENUM` is generated if `fail`, `zfail`, or `zpass` is any value other than the eight defined constant values.

`#GL_INVALID_OPERATION` is generated if `gl.StencilOp()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with one of the following arguments: `#GL_STENCIL_FAIL`, `#GL_STENCIL_PASS_DEPTH_PASS`, `#GL_STENCIL_PASS_DEPTH_FAIL`, `#GL_STENCIL_BACK_FAIL`, or `#GL_STENCIL_BITS`

`gl.IsEnabled()` with argument `#GL_STENCIL_TEST`

6.134 gl.TexCoord

NAME

gl.TexCoord – set the current texture coordinates

SYNOPSIS

```
gl.TexCoord(s[, t, r, q])
```

FUNCTION

`gl.TexCoord()` specifies texture coordinates in one, two, three, or four dimensions.

The current texture coordinates are part of the data that is associated with each vertex and with the current raster position. Initially, the values for s, t, r, and q are (0, 0, 0, 1).

The current texture coordinates can be updated at any time. In particular, `gl.TexCoord()` can be called between a call to `gl.Begin()` and the corresponding call to `gl.End()`.

Alternatively, you can also pass a table that contains one to four coordinates to `gl.TexCoord()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`s` specify the s texture coordinate

`t` optional: specify the t texture coordinate (defaults to 0)

`r` optional: specify the r texture coordinate (defaults to 0)

`q` optional: specify the q texture coordinate (defaults to 1)

ASSOCIATED GETS

`gl.Get()` with argument `#GL_CURRENT_TEXTURE_COORDS`

6.135 gl.TexCoordPointer

NAME

gl.TexCoordPointer – define an array of texture coordinates

SYNOPSIS

`gl.TexCoordPointer(vArray[, size])`

FUNCTION

`gl.TexCoordPointer()` specifies an array of texture coordinates to use when rendering. `vArray` can be either a one-dimensional table consisting of an arbitrary number of consecutive texture coordinates or a two-dimensional table consisting of an arbitrary number of subtables which contain 1 to 4 texture coordinates each. If `vArray` is a one-dimensional table, you need to pass the optional `size` argument as well to define the number of texture coordinates per array element. `size` must be a value in the range of 1 to 4. If `vArray` is a two-dimensional table, `size` is automatically determined by the number of items in the first subtable, which must be in the range of 1 to 4 as well.

When using a two-dimensional table, please keep in mind that the number of texture coordinates in each subtable must be constant. It is not allowed to use differing numbers of texture coordinates in the individual subtables. The number of texture coordinates is defined by the number of elements in the first subtable and all following subtables must use the very same number of coordinates.

If you pass `Nil` in `vArray`, the texture coordinates array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the texture coordinates array or defining a new one.

To enable and disable a texture coordinate array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_TEXTURE_COORD_ARRAY`. If enabled, the texture coordinate array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Each texture coordinate array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.TexCoordPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.TexCoordPointer()` is typically implemented on the client side.

Texture coordinate array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`vArray` one- or two-dimensional table containing texture coordinates or `Nil` (see above)

`size` optional: texture coordinates per array element; must be between 1 to 4 and is only used with one-dimensional tables (see above)

ERRORS

`#GL_INVALID_VALUE` is generated if `size` is not 1, 2, 3, or 4.

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_TEXTURE_COORD_ARRAY`

`gl.Get()` with argument `#GL_TEXTURE_COORD_ARRAY_SIZE`

`gl.Get()` with argument `#GL_TEXTURE_COORD_ARRAY_TYPE`

`gl.Get()` with argument `#GL_TEXTURE_COORD_ARRAY_STRIDE`

`gl.GetPointer()` with argument `#GL_TEXTURE_COORD_ARRAY_POINTER`

6.136 gl.TexEnv

NAME

gl.TexEnv – set texture environment parameters

SYNOPSIS

`gl.TexEnv(pname, param)`

FUNCTION

A texture environment specifies how texture values are interpreted when a fragment is textured. `pname` can be either `#GL_TEXTURE_ENV_MODE` or `#GL_TEXTURE_ENV_COLOR`. If `pname` is `#GL_TEXTURE_ENV_MODE`, then `param` must be the symbolic name of a texture function. Four texture functions may be specified: `#GL_MODULATE`, `#GL_DECAL`, `#GL_BLEND`, and `#GL_REPLACE`.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (See Section 6.141 [gl.TexParameter], page 199, for details.) and produces an RGBA color for that fragment. See an OpenGL reference manual for information on how the RGBA color is produced for each of the three texture functions that can be chosen.

If `pname` is `#GL_TEXTURE_ENV_COLOR`, `param` must be a table containing an array that holds an RGBA color consisting of four floating-point values.

`#GL_TEXTURE_ENV_MODE` defaults to `#GL_MODULATE` and `#GL_TEXTURE_ENV_COLOR` defaults to (0, 0, 0, 0)

Please consult an OpenGL reference manual for more information.

INPUTS

`pname` specifies the symbolic name of a texture environment parameter (see above)

`param` table or single value specifying the parameter

ERRORS

`#GL_INVALID_ENUM` is generated when `pname` is not one of the accepted defined values, or when `param` should have a defined constant value (based on the value of `pname`) and does not.

`#GL_INVALID_OPERATION` is generated if `gl.TexEnv()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexEnv()`

6.137 gl.TexGen

NAME

gl.TexGen – control the generation of texture coordinates

SYNOPSIS

```
gl.TexGen(coord, pname, param)
```

FUNCTION

`gl.TexGen()` selects a texture-coordinate generation function or supplies coefficients for one of the functions. `coord` names one of the (s, t, r, q) texture coordinates; it must be one of the symbols `#GL_S`, `#GL_T`, `#GL_R`, or `#GL_Q`. `pname` must be one of three symbolic constants: `#GL_TEXTURE_GEN_MODE`, `#GL_OBJECT_PLANE`, or `#GL_EYE_PLANE`. If `pname` is `#GL_TEXTURE_GEN_MODE`, then `param` chooses a mode, one of `#GL_OBJECT_LINEAR`, `#GL_EYE_LINEAR`, or `#GL_SPHERE_MAP`. If `pname` is either `#GL_OBJECT_PLANE` or `#GL_EYE_PLANE`, `param` contains coefficients for the corresponding texture generation function.

If the texture generation function is `#GL_OBJECT_LINEAR`, the function

```
g = p1x0 + p2y0 + p3z0 + p4w0
```

is used, where g is the value computed for the coordinate named in `coord`, p1, p2, p3, and p4 are the four values supplied in `param`, and x0, y0, z0, and w0 are the object coordinates of the vertex. This function can be used, for example, to texture-map terrain using sea level as a reference plane (defined by p1, p2, p3, and p4). The altitude of a terrain vertex is computed by the `#GL_OBJECT_LINEAR` coordinate generation function as its distance from sea level; that altitude can then be used to index the texture image to map white snow onto peaks and green grass onto foothills.

If the texture generation function is `#GL_EYE_LINEAR`, the function

```
g = p1'x0 + p2'y0 + p3'z0 + p4'w0
```

is used, where

```
(p1' p2' p3' p4') = ( p1 p2 p3 p4) M^-1
```

and x0, y0, z0, and w0 are the eye coordinates of the vertex, p1, p2, p3, and p4 are the values supplied in `param`, and M is the modelview matrix when `gl.TexGen()` is invoked. If M is poorly conditioned or singular, texture coordinates generated by the resulting function may be inaccurate or undefined.

Note that the values in `param` define a reference plane in eye coordinates. The modelview matrix that is applied to them may not be the same one in effect when the polygon

vertices are transformed. This function establishes a field of texture coordinates that can produce dynamic contour lines on moving objects.

If `pname` is `#GL_SPHERE_MAP` and `coord` is either `#GL_S` or `#GL_T`, s and t texture coordinates are generated as follows. Let u be the unit vector pointing from the origin to the polygon vertex (in eye coordinates). Let n' be the current normal, after transformation to eye coordinates. Let f = (fx fy fz)^T be the reflection vector such that

```
f = u - 2n' n'^Tu
```

Finally, let m = 2 sqrt(fx^2 + fy^2 + (fz + 1)^2). Then the values assigned to the s and t texture coordinates are

```
s = fx/m + 1/2
t = fy/m + 1/2
```

To enable or disable a texture-coordinate generation function, call `gl.Enable()` or `gl.Disable()` with one of the symbolic texture-coordinate names (`#GL_TEXTURE_GEN_S`, `#GL_TEXTURE_GEN_T`, `#GL_TEXTURE_GEN_R`, or `#GL_TEXTURE_GEN_Q`) as the argument. When enabled, the specified texture coordinate is computed according to the generating function associated with that coordinate. When disabled, subsequent vertices take the specified texture coordinate from the current set of texture coordinates. Initially, all texture generation functions are set to `#GL_EYE_LINEAR` and are disabled. Both s plane equations are (1, 0, 0, 0), both t plane equations are (0, 1, 0, 0), and all r and q plane equations are (0, 0, 0, 0).

Please consult an OpenGL reference manual for more information.

INPUTS

`coord`	specifies a texture coordinate; must be one of `#GL_S`, `#GL_T`, `#GL_R`, or `#GL_Q`
`pname`	specifies the symbolic name of the texture-coordinate generation function or function parameters (see above)
`param`	single value or table containing parameters for `pname` (see above)

ERRORS

`#GL_INVALID_ENUM` is generated when `coord` or `pname` is not an accepted defined value, or when `pname` is `#GL_TEXTURE_GEN_MODE` and `param` is not an accepted defined value.

`#GL_INVALID_ENUM` is generated when `pname` is `#GL_TEXTURE_GEN_MODE`, `param` is `#GL_SPHERE_MAP`, and `coord` is either `#GL_R` or `#GL_Q`.

`#GL_INVALID_OPERATION` is generated if `gl.TexGen()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexGen()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_GEN_S`

`gl.IsEnabled()` with argument `#GL_TEXTURE_GEN_T`

`gl.IsEnabled()` with argument `#GL_TEXTURE_GEN_R`

`gl.IsEnabled()` with argument `#GL_TEXTURE_GEN_Q`

6.138 gl.TexImage

NAME

gl.TexImage – specify a one- or two-dimensional texture image

SYNOPSIS

`gl.TexImage(level, internalformat, format, type, pixels[, border])`

FUNCTION

This function does the same as `gl.TexImage1D()` and `gl.TexImage2D()` except that the pixel data is not passed in a raw memory buffer but as table containing one subtable for each row of pixels. This is of course not as efficient as using raw memory buffers because the table's pixel data has to be copied to a raw memory buffer first.

Width and height of the texture will be automatically determined by the layout of the table in `pixels`. If there is only one subtable within `pixels`, `gl.TexImage()` will define a texture of type `#GL_TEXTURE_1D`. If there are multiple subtables within `pixels`, `#GL_TEXTURE_2D` will be used.

Note that only `#GL_FLOAT` and `#GL_UNSIGNED_BYTE` are currently supported for `type` and `internalformat` only accepts `#GL_RGB`, `#GL_RGBA`, `#GL_ALPHA`, `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_DEPTH_COMPONENT` and the values 1, 2, 3, and 4.

See Section 6.140 [gl.TexImage2D], page 196, for more details on the parameters accepted by this function.

Please consult an OpenGL reference manual for more information.

INPUTS

`level` specifies the level-of-detail number; level 0 is the base image level, level n is the nth mipmap reduction image

`internalformat`
specifies the number of color components in the texture; must be 1, 2, 3, or 4, or a symbolic constant (see above)

`format` specifies the format of the pixel data (see above)

`type` specifies the data type of the pixel data (see above)

`pixels` specifies a two-dimensional table containing pixel data

`border` optional: specifies the width of the border (defaults to 0)

6.139 gl.TexImage1D

NAME

gl.TexImage1D – specify a one-dimensional texture image

SYNOPSIS

`gl.TexImage1D(level, internalformat, width, border, format, type, pixels)`

FUNCTION

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_TEXTURE_1D`.

Texture images are defined with `gl.TexImage1D()`. The arguments describe the parameters of the texture image, such as width, width of the border, level-of-detail number (See Section 6.141 [gl.TexParameter], page 199, for details.), and the internal resolution and format used to store the image. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for `gl.DrawPixels()`.

Data is read from `pixels` as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on `type`. These values are grouped into sets of one, two, three, or four values, depending on `format`, to form elements. If type is `#GL_BITMAP`, the data is considered as a string of unsigned bytes (and format must be `#GL_COLOR_INDEX`). Each data byte is treated as eight 1-bit elements, with bit ordering determined by `#GL_UNPACK_LSB_FIRST` (See Section 6.104 [gl.PixelStore], page 148, for details.).

The first element corresponds to the left end of the texture array. Subsequent elements progress left-to-right through the remaining texels in the texture array. The final element corresponds to the right end of the texture array.

`format` determines the composition of each element in pixels. It can assume one of nine symbolic values:

`#GL_COLOR_INDEX`
: Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of `#GL_INDEX_SHIFT`, and added to `#GL_INDEX_OFFSET` (See Section 6.105 [gl.PixelTransfer], page 152, for details.). The resulting index is converted to a set of color components using the `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, and `#GL_PIXEL_MAP_I_TO_A` tables, and clamped to the range [0,1].

`#GL_RED`
: Each element is a single red component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_GREEN`
: Each element is a single green component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_BLUE`
: Each element is a single blue component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and green, and 1 for alpha. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_ALPHA`
: Each element is a single alpha component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red, green,

and blue. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_RGB` Each element is an RGB triple. The GL converts it to floating point and assembles it into an RGBA element by attaching 1 for alpha. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_RGBA` Each element contains all four components. Each component is multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_LUMINANCE`
Each element is a single luminance value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1 for alpha. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_LUMINANCE_ALPHA`
Each element is a luminance/alpha pair. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor `#GL_c_SCALE`, added to the signed bias `#GL_c_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_DEPTH_COMPONENT`
Each element is a single depth component. It is converted to floating-point, then multiplied by the signed scale factor `#GL_DEPTH_SCALE`, added to the signed bias `#GL_DEPTH_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with `internalformat`. `internalformat` specifies the internal format of the texture array. See Section 3.12 [Internal pixel formats], page 13, for details. The GL will choose an internal representation that closely approximates that requested by `internalformat`, but it may not match exactly. (The representations specified by `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_RGB`, and `#GL_RGBA` must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

A one-component texture image uses only the red component of the RGBA color extracted from pixels. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color index mode.

The texture image can be represented by the same data formats as the pixels in a `gl.DrawPixels()` command, except that `#GL_STENCIL_INDEX` and `#GL_DEPTH_COMPONENT` cannot be used. `gl.PixelStore()` and `gl.PixelTransfer()` modes affect texture images in exactly the way they affect `gl.DrawPixels()`.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 6.138 [gl.TexImage], page 192, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`level` specifies the level-of-detail number; level 0 is the base image level, level n is the nth mipmap reduction image

`internalformat` specifies the number of color components in the texture; must be 1, 2, 3, or 4, or a symbolic constant (see above)

`width` specifies the width of the texture image; must be 2^n + 2*border for some integer n; all implementations support texture images that are at least 64 texels wide

`border` specifies the width of the border; must be either 0 or 1

`format` specifies the format of the pixel data (see above)

`type` specifies the data type of the pixel data (see above)

`pixels` specifies a pointer to the image data in memory

ERRORS

`#GL_INVALID_ENUM` is generated if `format` is not an accepted format constant. Format constants other than `#GL_STENCIL_INDEX` are accepted.

`#GL_INVALID_ENUM` is generated if `type` is not a type constant.

`#GL_INVALID_ENUM` is generated if `type` is `#GL_BITMAP` and `format` is not `#GL_COLOR_INDEX`.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than log2max, where max is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if `internalformat` is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

`#GL_INVALID_VALUE` is generated if `width` or `height` is less than 0 or greater than 2 + `#GL_MAX_TEXTURE_SIZE`, or if either cannot be represented as 2k + 2*border for some integer value of k.

`#GL_INVALID_VALUE` is generated if `border` is not 0 or 1.

`#GL_INVALID_OPERATION` is generated if `gl.TexImage1D()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexImage()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_1D`

6.140 gl.TexImage2D

NAME

gl.TexImage2D – specify a two-dimensional texture image

SYNOPSIS

```
gl.TexImage2D(level, internalformat, w, h, border, format, type, pixels)
```

FUNCTION

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_TEXTURE_2D`.

To define texture images, call `gl.TexImage2D()`. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (See Section 6.141 [gl.TexParameter], page 199, for details.), and number of color components provided. The last three arguments describe how the image is represented in memory; they are identical to the pixel formats used for glDrawPixels.

Data is read from `pixels` as a sequence of signed or unsigned bytes, shorts, or longs, or single-precision floating-point values, depending on `type` which can be `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, and `#GL_FLOAT`. These values are grouped into sets of one, two, three, or four values, depending on `format`, to form elements. If type is `#GL_BITMAP`, the data is considered as a string of unsigned bytes (and `format` must be `#GL_COLOR_INDEX`). Each data byte is treated as eight 1-bit elements, with bit ordering determined by `#GL_UNPACK_LSB_FIRST` (See Section 6.104 [gl.PixelStore], page 148, for details.).

The first element corresponds to the lower left corner of the texture image. Subsequent elements progress left-to-right through the remaining texels in the lowest row of the texture image, and then in successively higher rows of the texture image. The final element corresponds to the upper right corner of the texture image.

`format` determines the composition of each element in pixels. It can assume one of nine symbolic values:

`#GL_COLOR_INDEX`
: Each element is a single value, a color index. The GL converts it to fixed point (with an unspecified number of zero bits to the right of the binary point), shifted left or right depending on the value and sign of `#GL_INDEX_SHIFT`, and added to `#GL_INDEX_OFFSET` (See Section 6.105 [gl.PixelTransfer], page 152, for details.). The resulting index is converted to a set of color components using the `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, and `#GL_PIXEL_MAP_I_TO_A` tables, and clamped to the range [0,1].

`#GL_RED`
: Each element is a single red component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for green and blue, and 1 for alpha. Each component is then multiplied by the signed scale

factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_GREEN
Each element is a single green component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and blue, and 1 for alpha. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_BLUE Each element is a single blue component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red and green, and 1 for alpha. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_ALPHA
Each element is a single alpha component. The GL converts it to floating point and assembles it into an RGBA element by attaching 0 for red, green, and blue. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_RGB Each element is an RGB triple. The GL converts it to floating point and assembles it into an RGBA element by attaching 1 for alpha. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_RGBA Each element contains all four components. Each component is multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_LUMINANCE
Each element is a single luminance value. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1 for alpha. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

#GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. The GL converts it to floating point, then assembles it into an RGBA element by replicating the luminance value three times for red, green, and blue. Each component is then multiplied by the signed scale factor #GL_c_SCALE, added to the signed bias #GL_c_BIAS, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

`#GL_DEPTH_COMPONENT`
: Each element is a single depth component. It is converted to floating-point, then multiplied by the signed scale factor `#GL_DEPTH_SCALE`, added to the signed bias `#GL_DEPTH_BIAS`, and clamped to the range [0, 1] (See Section 6.105 [gl.PixelTransfer], page 152, for details.).

If an application wants to store the texture at a certain resolution or in a certain format, it can request the resolution and format with `internalformat`. `internalformat` specifies the internal format of the texture array. See Section 3.12 [Internal pixel formats], page 13, for details. The GL will choose an internal representation that closely approximates that requested by `internalformat`, but it may not match exactly. (The representations specified by `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_RGB`, and `#GL_RGBA` must match exactly. The numeric values 1, 2, 3, and 4 may also be used to specify the above representations.)

A one-component texture image uses only the red component of the RGBA color extracted from pixels. A two-component image uses the R and A values. A three-component image uses the R, G, and B values. A four-component image uses all of the RGBA components.

Texturing has no effect in color index mode.

The texture image can be represented by the same data formats as the pixels in a `gl.DrawPixels()` command, except that `#GL_STENCIL_INDEX` and `#GL_DEPTH_COMPONENT` cannot be used. `gl.PixelStore()` and `gl.PixelTransfer()` modes affect texture images in exactly the way they affect `gl.DrawPixels()`.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 6.138 [gl.TexImage], page 192, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`level` specifies the level-of-detail number; level 0 is the base image level, level n is the nth mipmap reduction image

`internalformat`
: specifies the number of color components in the texture; must be 1, 2, 3, or 4, or a symbolic constant (see above)

`w` specifies the width of the texture image; must be 2^n + 2*border for some integer n; all implementations support texture images that are at least 64 texels wide

`h` specifies the height of the texture image; must be 2^m + 2*border for some integer m; all implementations support texture images that are at least 64 texels high

`border` specifies the width of the border; must be either 0 or 1

`format` specifies the format of the pixel data (see above)

`type` specifies the data type of the pixel data (see above)

`pixels` specifies a pointer to the image data in memory

ERRORS

`#GL_INVALID_ENUM` is generated if `format` is not an accepted format constant. Format constants other than `#GL_STENCIL_INDEX` are accepted.

`#GL_INVALID_ENUM` is generated if `type` is not a type constant.

`#GL_INVALID_ENUM` is generated if `type` is `#GL_BITMAP` and `format` is not `#GL_COLOR_INDEX`.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than log2max, where max is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if `internalformat` is not 1, 2, 3, 4, or one of the accepted resolution and format symbolic constants.

`#GL_INVALID_VALUE` is generated if `width` or `height` is less than 0 or greater than 2 + `#GL_MAX_TEXTURE_SIZE`, or if either cannot be represented as 2k + 2*border for some integer value of k.

`#GL_INVALID_VALUE` is generated if `border` is not 0 or 1.

`#GL_INVALID_OPERATION` is generated if `gl.TexImage2D()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexImage()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_2D`

6.141 gl.TexParameter

NAME

gl.TexParameter – set texture parameters

SYNOPSIS

`gl.TexParameter(target, pname, param)`

FUNCTION

Texture mapping is a technique that applies an image onto an object's surface as if the image were a decal or cellophane shrink-wrap. The image is created in texture space, with an (s, t) coordinate system. A texture is a one- or two-dimensional image and a set of parameters that determine how samples are derived from the image.

`gl.TexParameter()` assigns the value or values in `param` to the texture parameter specified as `pname`. `target` defines the target texture, either `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`. The following symbols are accepted in `pname`:

`#GL_TEXTURE_MIN_FILTER`

The texture minifying function is used whenever the pixel being textured maps to an area greater than one texture element. There are six defined minifying functions. Two of them use the nearest one or nearest four texture elements to compute the texture value. The other four use mipmaps.

A mipmap is an ordered set of arrays representing the same image at progressively lower resolutions: 2^a for 1D mipmaps and 2^a*2^b for 2D mipmaps. For example, if a 2D texture has dimensions 2^m*2^n, there are max(m, n) + 1 mipmaps. The first mipmap is the original texture, with dimensions 2^m*2^n. Each subsequent mipmap has dimensions 2^k-1*2^l-1, where 2^k*2^l are the dimensions of the previous mipmap, until either k=0 or l=0. At that point, subsequent mipmaps have dimension 1*2^l-1 or 2^k-1*1 until the final mipmap, which has dimension 1x1. To define the mipmaps, call `gl.TexImage1D()`, `gl.TexImage2D()`, or `gl.CopyTexImage()` with the level argument indicating the order of the mipmaps. Level 0 is the original texture; level max(m, n) is the final 1x1 mipmap. `param` supplies a function for minifying the texture as one of the following:

`#GL_NEAREST`
: Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.

`#GL_LINEAR`
: Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of `#GL_TEXTURE_WRAP_S` and `#GL_TEXTURE_WRAP_T`, and on the exact mapping.

`#GL_NEAREST_MIPMAP_NEAREST`
: Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `#GL_NEAREST` criterion (the texture element nearest to the center of the pixel) to produce a texture value.

`#GL_LINEAR_MIPMAP_NEAREST`
: Chooses the mipmap that most closely matches the size of the pixel being textured and uses the `#GL_LINEAR` criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value.

`#GL_NEAREST_MIPMAP_LINEAR`
: Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `#GL_NEAREST` criterion (the texture element nearest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

`#GL_LINEAR_MIPMAP_LINEAR`
: Chooses the two mipmaps that most closely match the size of the pixel being textured and uses the `#GL_LINEAR` criterion (a weighted average of the four texture elements that are closest to the center of the pixel) to produce a texture value from each mipmap. The final texture value is a weighted average of those two values.

As more texture elements are sampled in the minification process, fewer aliasing artifacts will be apparent. While the #GL_NEAREST and #GL_LINEAR minification functions can be faster than the other four, they sample only one or four texture elements to determine the texture value of the pixel being rendered and can produce moire patterns or ragged transitions. The initial value of #GL_TEXTURE_MIN_FILTER is #GL_NEAREST_MIPMAP_LINEAR.

#GL_TEXTURE_MAG_FILTER

The texture magnification function is used when the pixel being textured maps to an area less than or equal to one texture element. It sets the texture magnification function to either #GL_NEAREST or #GL_LINEAR (see below). #GL_NEAREST is generally faster than #GL_LINEAR, but it can produce textured images with sharper edges because the transition between texture elements is not as smooth. The initial value of #GL_TEXTURE_MAG_FILTER is #GL_LINEAR.

#GL_NEAREST

Returns the value of the texture element that is nearest (in Manhattan distance) to the center of the pixel being textured.

#GL_LINEAR

Returns the weighted average of the four texture elements that are closest to the center of the pixel being textured. These can include border texture elements, depending on the values of #GL_TEXTURE_WRAP_S and #GL_TEXTURE_WRAP_T, and on the exact mapping.

#GL_TEXTURE_WRAP_S

Sets the wrap parameter for texture coordinate s to #GL_CLAMP or #GL_REPEAT. #GL_CLAMP causes s coordinates to be clamped to the range [0, 1] and is useful for preventing wrapping artifacts when mapping a single image onto an object. #GL_REPEAT causes the integer part of the s coordinate to be ignored; the GL uses only the fractional part, thereby creating a repeating pattern. Border texture elements are accessed only if wrapping is set to #GL_CLAMP. Initially, #GL_TEXTURE_WRAP_S is set to #GL_REPEAT.

#GL_TEXTURE_WRAP_T

Sets the wrap parameter for texture coordinate t to #GL_CLAMP or #GL_REPEAT. See the discussion under #GL_TEXTURE_WRAP_S. Initially, #GL_TEXTURE_WRAP_T is set to #GL_REPEAT.

#GL_TEXTURE_WRAP_R_EXT

Sets the wrap parameter for texture coordinate r to #GL_CLAMP or #GL_REPEAT. See the discussion under #GL_TEXTURE_WRAP_S. Initially, #GL_TEXTURE_WRAP_R_EXT is set to #GL_REPEAT.

#GL_TEXTURE_BORDER_COLOR

Sets a border color. param must be a table containing four floating-point values that comprise the RGBA color of the texture border. Initially, the border color is (0, 0, 0, 0).

`#GL_TEXTURE_PRIORITY`
: Specifies the texture residence priority of the currently bound texture. Permissible values are in the range [0, 1]. See Section 6.115 [gl.PrioritizeTextures], page 162, for details.

Suppose that a program has enabled texturing (by calling `gl.Enable()` with argument `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`) and has set `#GL_TEXTURE_MIN_FILTER` to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to `gl.TexImage1D()`, `gl.TexImage2D()` or `gl.CopyTexImage()`) do not follow the proper sequence for mipmaps (described above), or there are fewer texture images defined than are needed, or the set of texture images have differing numbers of texture components, then it is as if texture mapping were disabled.

Linear filtering accesses the four nearest texture elements only in 2D textures. In 1D textures, linear filtering accesses the two nearest texture elements.

Please consult an OpenGL reference manual for more information.

INPUTS

`target` specifies the target texture, which must be either `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`

`pname` specifies the symbolic name of a texture parameter (see above)

`param` specifies a single value or a table containing the value for `pname`

ERRORS

`#GL_INVALID_ENUM` is generated if `target` or `pname` is not one of the accepted defined values.

`#GL_INVALID_ENUM` is generated if `param` should have a defined constant value (based on the value of `pname`) and does not.

`#GL_INVALID_OPERATION` is generated if `gl.TexParameter()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexParameter()`

`gl.GetTexLevelParameter()`

6.142 gl.TexSubImage

NAME

gl.TexSubImage – specify a one- or two-dimensional texture subimage

SYNOPSIS

`gl.TexSubImage(level, format, type, pixels, xoffset[, yoffset])`

FUNCTION

This function does the same as `gl.TexSubImage1D()` and `gl.TexSubImage2D()` except that the pixel data is not passed in a raw memory buffer but as table containing one

subtable for each row of pixels. This is of course not as efficient as using raw memory buffers because the table's pixel data has to be copied to a raw memory buffer first.

Width and height of the texture will be automatically determined by the layout of the table in `pixels`. If there is only one subtable within `pixels`, `gl.TexSubImage()` will define a texture of type `#GL_TEXTURE_1D`. If there are multiple subtables within `pixels`, `#GL_TEXTURE_2D` will be used.

Note that only `#GL_FLOAT` and `#GL_UNSIGNED_BYTE` are currently supported for `type`.

See Section 6.144 [gl.TexSubImage2D], page 204, for more details on the parameters accepted by this function.

Please consult an OpenGL reference manual for more information.

INPUTS

`level`	specifies the level-of-detail number; level 0 is the base image level; level n is the nth mipmap reduction image
`format`	specifies the format of the pixel data (see above)
`type`	specifies the data type of the pixel data (see above)
`pixels`	specifies a one- or two-dimensional with pixel data
`xoffset`	specifies a texel offset in the x direction within the texture array
`yoffset`	optional: specifies a texel offset in the y direction within the texture array (only required for two-dimensional textures)

6.143 gl.TexSubImage1D

NAME

gl.TexSubImage1D – specify a two-dimensional texture subimage

SYNOPSIS

```
gl.TexSubImage1D(level, xoffset, width, format, type, pixelsUserData)
```

FUNCTION

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable or disable one-dimensional texturing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_TEXTURE_1D`.

`gl.TexSubImage1D()` redefines a contiguous subregion of an existing one-dimensional texture image. The texels referenced by pixels replace the portion of the existing texture array with X indices xoffset and xoffset `+` width - 1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

Texturing has no effect in color index mode.

`gl.PixelStore()` and `gl.PixelTransfer()` modes affect texture images in exactly the way they affect `gl.DrawPixels()`.

See Section 6.139 [gl.TexImage1D], page 192, for more details on the parameters accepted by this function.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 6.142 [gl.TexSubImage], page 202, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`level` specifies the level-of-detail number; level 0 is the base image level; level n is the nth mipmap reduction image

`xoffset` specifies a texel offset in the x direction within the texture array

`width` specifies the width of the texture subimage

`format` specifies the format of the pixel data (see above)

`type` specifies the data type of the pixel data (see above)

`pixels` specifies a pointer to the image data in memory

ERRORS

`#GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous `gl.TexImage1D()` operation.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than log2max, where max is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if xoffset `<` -b, or if (xoffset `+` width) `>` (w - b), where w is the `#GL_TEXTURE_WIDTH`, and b is the width of the `#GL_TEXTURE_BORDER` of the texture image being modified. Note that w includes twice the border width.

`#GL_INVALID_VALUE` is generated if `width` is less than 0.

`#GL_INVALID_ENUM` is generated if `format` is not an accepted format constant.

`#GL_INVALID_ENUM` is generated if `type` is not a type constant.

`#GL_INVALID_ENUM` is generated if `type` is `#GL_BITMAP` and format is not `#GL_COLOR_INDEX`.

`#GL_INVALID_OPERATION` is generated if `gl.TexSubImage1D()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexImage()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_1D`

6.144 gl.TexSubImage2D

NAME

gl.TexSubImage2D – specify a two-dimensional texture subimage

SYNOPSIS

```
gl.TexSubImage2D(level, xoff, yoff, w, h, format, type, pixels)
```

FUNCTION

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable two-dimensional texturing, call `gl.Enable()` and `gl.Disable()` with argument `#GL_TEXTURE_2D`.

`gl.TexSubImage2D()` redefines a contiguous subregion of an existing two-dimensional texture image. The texels referenced by `pixels` replace the portion of the existing texture array with X indices `xoffset` and xoffset `+` width - 1, inclusive, and Y indices `yoffset` and yoffset `+` height - 1, inclusive. This region may not include any texels outside the range of the texture array as it was originally specified. It is not an error to specify a subtexture with zero width or height, but such a specification has no effect.

Texturing has no effect in color index mode.

`gl.PixelStore()` and `gl.PixelTransfer()` modes affect texture images in exactly the way they affect `gl.DrawPixels()`.

See Section 6.140 [gl.TexImage2D], page 196, for more details on the parameters accepted by this function.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 6.142 [gl.TexSubImage], page 202, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`level`	specifies the level-of-detail number; level 0 is the base image level; level n is the nth mipmap reduction image
`xoffset`	specifies a texel offset in the x direction within the texture array
`yoffset`	specifies a texel offset in the y direction within the texture array
`width`	specifies the width of the texture subimage
`height`	specifies the height of the texture subimage
`format`	specifies the format of the pixel data (see above)
`type`	specifies the data type of the pixel data (see above)
`pixels`	specifies a pointer to the image data in memory

ERRORS

`#GL_INVALID_OPERATION` is generated if the texture array has not been defined by a previous `gl.TexImage2D()` operation.

`#GL_INVALID_VALUE` is generated if `level` is less than 0.

`#GL_INVALID_VALUE` may be generated if `level` is greater than log2max, where max is the returned value of `#GL_MAX_TEXTURE_SIZE`.

`#GL_INVALID_VALUE` is generated if xoffset `<` -b, (xoffset `+` width) `>` (w - b), yoffset `<` -b, or (yoffset `+` height) `>` (h - b), where w is the `#GL_TEXTURE_WIDTH`, h is the `#GL_TEXTURE_HEIGHT`, and b is the border width of the texture image being modified. Note that w and h include twice the border width.

`#GL_INVALID_VALUE` is generated if `width` or `height` is less than 0.

`#GL_INVALID_ENUM` is generated if `format` is not an accepted format constant.

`#GL_INVALID_ENUM` is generated if `type` is not a type constant.

`#GL_INVALID_ENUM` is generated if `type` is `#GL_BITMAP` and `format` is not `#GL_COLOR_INDEX`.

`#GL_INVALID_OPERATION` is generated if `gl.TexSubImage2D()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.GetTexImage()`

`gl.IsEnabled()` with argument `#GL_TEXTURE_2D`

6.145 gl.Translate

NAME

gl.Translate – multiply the current matrix by a translation matrix

SYNOPSIS

```
gl.Translate(x, y, z)
```

FUNCTION

`gl.Translate()` produces a translation by (x,y,z). The current matrix (See Section 6.96 [gl.MatrixMode], page 139, for details.) is multiplied by this translation matrix, with the product replacing the current matrix, as if `gl.MultMatrix()` were called with the following matrix for its argument:

```
1 0 0 x
0 1 0 y
0 0 1 z
0 0 0 1
```

If the matrix mode is either `#GL_MODELVIEW` or `#GL_PROJECTION`, all objects drawn after a call to `gl.Translate()` are translated.

Use `gl.PushMatrix()` and `gl.PopMatrix()` to save and restore the untranslated coordinate system.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the x coordinate of a translation vector
`y`	specify the y coordinate of a translation vector
`z`	specify the z coordinate of a translation vector

ERRORS

`#GL_INVALID_OPERATION` is generated if `gl.Translate()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_MATRIX_MODE`

`gl.Get()` with argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` with argument `#GL_PROJECTION_MATRIX`

`gl.Get()` with argument `#GL_TEXTURE_MATRIX`

6.146 gl.Vertex

NAME

gl.Vertex – specify a vertex

SYNOPSIS

```
gl.Vertex(x, y[, z, w])
```

FUNCTION

`gl.Vertex()` is used within `gl.Begin()` / `gl.End()` pairs to specify point, line, and polygon vertices. The current color, normal, texture coordinates, and fog coordinate are associated with the vertex when `gl.Vertex()` is called.

When only `x` and `y` are specified, `z` defaults to 0 and `w` defaults to 1. When x, y, and z are specified, `w` defaults to 1.

Alternatively, you can also pass a table containing two to four vertex coordinates to this function.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the x coordinate of a vertex
`y`	specify the y coordinate of a vertex
`z`	optional: specify the z coordinate of a vertex (defaults to 0)
`w`	optional: specify the w coordinate of a vertex (defaults to 1)

6.147 gl.VertexPointer

NAME

gl.VertexPointer – define an array of vertex data

SYNOPSIS

```
gl.VertexPointer(vertexArray[, size])
```

FUNCTION

`gl.VertexPointer()` specifies an array of vertex coordinates to use when rendering. `vertexArray` can be either a one-dimensional table consisting of an arbitrary number of consecutive vertex coordinates or a two-dimensional table consisting of an arbitrary number of subtables which contain 2 to 4 texture coordinates each. If `vertexArray` is a one-dimensional table, you need to pass the optional `size` argument as well to define the number of vertex coordinates per array element. `size` must be a value in the range of 2 to 4. If `vertexArray` is a two-dimensional table, `size` is automatically determined by the number of items in the first subtable, which must be in the range of 2 to 4 as well.

When using a two-dimensional table, please keep in mind that the number of vertex coordinates in each subtable must be constant. It is not allowed to use differing numbers of vertex coordinates in the individual subtables. The number of vertex coordinates is defined by the number of elements in the first subtable and all following subtables must use the very same number of coordinates.

If you pass `Nil` in `vertexArray`, the vertex coordinates array buffer will be freed but it won't be removed from OpenGL. You need to do this manually, e.g. by disabling the vertex coordinates array or defining a new one.

In order to enable and disable a vertex array, call `gl.EnableClientState()` and `gl.DisableClientState()` with the argument `#GL_VERTEX_ARRAY`. If enabled, the vertex array is used when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

The vertex array is initially disabled and isn't accessed when `gl.DrawArrays()`, `gl.DrawElements()`, or `gl.ArrayElement()` is called.

Execution of `gl.VertexPointer()` is not allowed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`, but an error may or may not be generated. If no error is generated, the operation is undefined.

`gl.VertexPointer()` is typically implemented on the client side.

Vertex array parameters are client-side state and are therefore not saved or restored by `gl.PushAttrib()` and `gl.PopAttrib()`. Use `gl.PushClientAttrib()` and `gl.PopClientAttrib()` instead.

Please consult an OpenGL reference manual for more information.

INPUTS

`vertexArray`
: one- or two-dimensional table containing vertex coordinates or `Nil` (see above)

`size`
: optional: vertex coordinates per array element; must be between 2 to 4 and is only used with one-dimensional tables (see above)

ERRORS

`#GL_INVALID_VALUE` is generated if `size` is not 2, 3, or 4.

ASSOCIATED GETS

`gl.IsEnabled()` with argument `#GL_VERTEX_ARRAY`

`gl.Get()` with argument `#GL_VERTEX_ARRAY_SIZE`

`gl.Get()` with argument `#GL_VERTEX_ARRAY_TYPE`

`gl.Get()` with argument `#GL_VERTEX_ARRAY_STRIDE`

`gl.GetPointer()` with argument `#GL_VERTEX_ARRAY_POINTER`

6.148 gl.Viewport

NAME

gl.Viewport – set the viewport

SYNOPSIS

`gl.Viewport(x, y, width, height)`

FUNCTION

`gl.Viewport()` specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let (xnd, ynd) be normalized device coordinates. Then the window coordinates (xw, yw) are computed as follows:

```
xw = (xnd + 1) * (width / 2) + x
yw = (ynd + 1) * (height / 2) + y
```

Viewport width and height are silently clamped to a range that depends on the implementation. To query this range, call `gl.Get()` with argument `#GL_MAX_VIEWPORT_DIMS`.

When a GL context is first attached to a window, width and height are set to the dimensions of that window.

Please consult an OpenGL reference manual for more information.

INPUTS

`x`	specify the left corner of the viewport rectangle, in pixels; the initial value is 0
`y`	specify the lower corner of the viewport rectangle, in pixels; the initial value is 0
`width`	specify the width of the viewport
`height`	specify the height of the viewport

ERRORS

`#GL_INVALID_VALUE` is generated if either `width` or `height` is negative.

`#GL_INVALID_OPERATION` is generated if `gl.Viewport()` is executed between the execution of `gl.Begin()` and the corresponding execution of `gl.End()`.

ASSOCIATED GETS

`gl.Get()` with argument `#GL_VIEWPORT`

`gl.Get()` with argument `#GL_MAX_VIEWPORT_DIMS`

7 GLU reference

7.1 glu.BuildMipmaps

NAME

glu.BuildMipmaps – create 1D or 2D mipmaps

SYNOPSIS

```
error = glu.BuildMipmaps(internalformat, format, pixels)
```

FUNCTION

This function does the same as `glu.Build1DMipmaps()` and `gl.Build2DMipmaps()` except that the pixel data is not passed in a raw memory buffer but as table containing one subtable for each row of pixels. This is of course not as efficient as using raw memory buffers because the table's pixel data has to be copied to a raw memory buffer first.

Width and height of the texture will be automatically determined by the layout of the table in `pixels`. If there is only one subtable within `pixels`, gl.BuildMipmaps() will create 1D mipmaps. If there are multiple subtables within `pixels`, 2D mipmaps will be created.

Note that `#GL_UNSIGNED_BYTE` is currently the only supported data type. `glu.BuildMipmaps()` expects all elements in `pixels` to use the `#GL_UNSIGNED_BYTE` data type.

See Section 7.3 [glu.Build2DMipmaps], page 213, for more details on the parameters accepted by this function.

Please consult an OpenGL reference manual for more information.

INPUTS

`internalFormat`
: specifies the internal format of the texture; must be one of the pixel format constants (see above)

`format`
: specifies the format of the pixel data (see above)

`pixels`
: specifies a table containing the pixel data

RESULTS

`error`
: error code or 0 for success

7.2 glu.Build1DMipmaps

NAME

glu.Build1DMipmaps – create 1D mipmaps

SYNOPSIS

```
error = glu.Build1DMipmaps(internalformat, width, format, type, pixels)
```

FUNCTION

`glu.Build1DMipmaps()` builds a series of prefiltered 1D texture maps of decreasing resolution. Mipmaps can be used so that textures don't appear aliased.

A return value of 0 indicates success. Otherwise a GLU error code is returned (See Section 7.5 [glu.ErrorString], page 215, for details.).

`glu.Build1DMipmaps()` first checks whether the width of data is a power of 2. If not, it scales a copy of `pixels` (up or down) to the nearest power of two. This copy is used as the base for subsequent mipmapping operations. For example, if `width` is 57, a copy of `pixels` scales up to 64 before mipmapping takes place. (If `width` is exactly between powers of 2, the copy of `pixels` is scaled upward.)

If the GL version is 1.1 or greater, `glu.Build1DMipmaps()` uses proxy textures (See Section 6.139 [gl.TexImage1D], page 192, for details.) to determine if the implementation can store the requested texture in texture memory. If there isn't enough room, `width` is halved (and halved again) until it fits.

Next, `glu.Build1DMipmaps()` builds a series of mipmap levels; it halves a copy of `pixels` (or a scaled version of `pixels`, if necessary) until size 1 is reached. At each level, each texel in the halved image is an average of the corresponding two texels in the larger image.

`gl.TexImage1D()` is called to load each of these images by level. If `width` is a power of 2 which fits in the implementation, level 0 is a copy of `pixels`, and the highest level is log2(width). For example, if `width` is 64 the following images are built: 64x1, 32x1, 16x1, 8x1, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

`internalformat` specifies the internal format of the texture image. See Section 3.12 [Internal pixel formats], page 13, for details. This can also be one of the special values 1, 2, 3, or 4.

`format` must be one of `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, or `#GL_LUMINANCE_ALPHA`

`type` must be one of `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 7.1 [glu.BuildMipmaps], page 211, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`internalformat`
: specifies the internal format of the texture; must be one of the pixel format constants (see above)

`width`
: specifies the width of the texture image

`format`
: specifies the format of the pixel data (see above)

`type`
: specifies the data type for `pixels` (see above)

`pixels`
: specifies a pointer to the image data in memory

RESULTS

`error`
: error code or 0 for success

7.3 glu.Build2DMipmaps

NAME

glu.Build2DMipmaps – create 2D mipmaps

SYNOPSIS

```
error = glu.Build2DMipmaps(iformat, width, height, format, type, pixels)
```

FUNCTION

`glu.Build2DMipmaps()` builds a series of prefiltered 2D texture maps of decreasing resolution. Mipmaps can be used so that textures don't appear aliased.

A return value of 0 indicates success. Otherwise a GLU error code is returned (See Section 7.5 [glu.ErrorString], page 215, for details.).

`glu.Build2DMipmaps()` first check whether `width` and `height` of `pixels` are both powers of 2. If not, `glu.Build2DMipmaps()` scales a copy of `pixels` up or down to the nearest power of 2. This copy is then used as the base for subsequent mipmapping operations. For example, if `width` is 57 and `height` is 23, then a copy of `pixels` scales up to 64 and down to 16, respectively, before mipmapping takes place. (If `width` or `height` is exactly between powers of 2, the copy of data is scaled upward.)

If the GL version is 1.1 or greater, `glu.Build2DMipmaps()` then uses proxy textures (See Section 6.140 [gl.TexImage2D], page 196, for details.) to determine whether there's enough room for the requested texture in the implementation. If not, `width` is halved (and halved again) until it fits.

`glu.Build2DMipmaps()` then uses proxy textures (See Section 6.140 [gl.TexImage2D], page 196, for details.) to determine if the implementation can store the requested texture in texture memory. If not, both dimensions are continually halved until it fits.

Next, `glu.Build2DMipmaps()` builds a series of images; it halves a copy of `type` (or a scaled version of `type`, if necessary) along both dimensions until size 1x1 is reached. At each level, each texel in the halved mipmap is an average of the corresponding four texels in the larger mipmap. (In the case of rectangular images, halving the images repeatedly eventually results in an n*1 or 1*n configuration. Here, two texels are averaged instead.)

`gl.TexImage2D()` is called to load each of these images by level. If `width` and `height` are both powers of 2 which fit in the implementation, level 0 is a copy of `pixels`, and the highest level is log2(max(width, height)). For example, if `width` is 64 and `height` is 16, the following mipmaps are built: 64x16, 32x8, 16x4, 8x2, 4x1, 2x1 and 1x1. These correspond to levels 0 through 6, respectively.

`iformat` specifies the internal format of the texture image. See Section 3.12 [Internal pixel formats], page 13, for details. This can also be one of the special values 1, 2, 3, or 4.

`format` must be one of `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, or `#GL_LUMINANCE_ALPHA`

`type` must be one of `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`.

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 7.1 [glu.BuildMipmaps], page 211, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`iformat`	specifies the internal format of the texture; must be one of the pixel format constants (see above)
`width`	specifies the width of the texture image
`height`	specifies the height of the texture image
`format`	specifies the format of the pixel data (see above)
`type`	specifies the data type for `pixels` (see above)
`pixels`	specifies a pointer to the image data in memory

RESULTS

`error`	error code or 0 for success

7.4 glu.Build3DMipmaps

NAME

glu.Build3DMipmaps – create 3D mipmaps

SYNOPSIS

```
error = glu.Build3DMipmaps(ifmt, width, height, depth, fmt, type, data)
```

FUNCTION

`glu.Build3DMipmaps()` builds a series of prefiltered 3D texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

A return value of zero indicates success, otherwise a GLU error code is returned (See Section 7.5 [glu.ErrorString], page 215, for details.).

Initially, the `width`, `height` and `depth` of data are checked to see if they are a power of 2. If not, a copy of data is made and scaled up or down to the nearest power of 2. (If `width`, `height`, or `depth` is exactly between powers of 2, then the copy of data will scale upwards.) This copy will be used for subsequent mipmapping operations described below. For example, if width is 57, height is 23, and depth is 24, then a copy of data will scale up to 64 in width, down to 16 in height, and up to 32 in depth before mipmapping takes place.

Then, proxy textures (see gl.TexImage3D()) are used to determine if the implementation can fit the requested texture. If not, all three dimensions are continually halved until it fits.

Next, a series of mipmap levels is built by decimating a copy of data in half along all three dimensions until size 1x1x1 is reached. At each level, each texel in the halved mipmap level is an average of the corresponding eight texels in the larger mipmap level. (If exactly one of the dimensions is 1, four texels are averaged. If exactly two of the dimensions are 1, two texels are averaged.)

gl.TexImage3D() is called to load each of these mipmap levels. Level 0 is a copy of data. The highest level is log2(max(width,height,depth)). For example, if `width` is 64,

`height` is 16, and `depth` is 32, and the implementation can store a texture of this size, the following mipmap levels are built: 64x16x32, 32x8x16, 16x4x8, 8x2x4, 4x1x2, 2x1x1 and 1x1x1. These correspond to levels 0 through 6, respectively.

`ifmt` specifies the internal format of the texture image. See Section 3.12 [Internal pixel formats], page 13, for details. This can also be one of the special values 1, 2, 3, or 4.

`fmt` must be one of `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, or `#GL_LUMINANCE_ALPHA`

`type` must be one of `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`.

Please note that this command operates directly with memory pointers. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`ifmt`	specifies the internal format of the texture; must be one of the pixel format constants (see above)
`width`	specifies the width of the texture image
`height`	specifies the height of the texture image
`depth`	specifies the depth of the texture image
`fmt`	specifies the format of the pixel data (see above)
`type`	specifies the data type for `pixels` (see above)
`data`	specifies a pointer to the image data in memory

RESULTS

`error`	error code or 0 for success

7.5 glu.ErrorString

NAME

glu.ErrorString – produce an error string from a GL or GLU error code

SYNOPSIS

```
string = glu.ErrorString(errorCode)
```

FUNCTION

`glu.ErrorString()` produces an error string from a GL or GLU error code. The string is in ISO Latin 1 format. For example, glu.ErrorString(`#GL_OUT_OF_MEMORY`) returns the string `"Out of memory"`.

The standard GLU error codes are `#GLU_INVALID_ENUM`, `#GLU_INVALID_VALUE`, and `#GLU_OUT_OF_MEMORY`. Certain other GLU functions can return specialized error codes through callbacks. See Section 6.60 [gl.GetError], page 97, for the list of GL error codes.

Please consult an OpenGL reference manual for more information.

INPUTS

`errorCode`
specifies a GL or GLU error code

RESULTS

`string` error string

7.6 glu.GetString

NAME
glu.GetString – return a string describing the GLU version or GLU extensions

SYNOPSIS
```
string = glu.GetString(name)
```

FUNCTION
`glu.GetString()` returns a string describing the GLU version or the GLU extensions that are supported.

The version number is one of the following forms:

```
<major_number>.<minor_number>
<major_number>.<minor_number>.<release_number>
```

The version string is of the following form:

```
<version number><space><vendor-specific information>
```

Vendor-specific information is optional. Its format and contents depend on the implementation.

The standard GLU contains a basic set of features and capabilities. If a company or group of companies wish to support other features, these may be included as extensions to the GLU. If `name` is `#GLU_EXTENSIONS`, then `glu.GetString()` returns a space-separated list of names of supported GLU extensions. (Extension names never contain spaces.)

Please note that `glu.GetString()` only returns information about GLU extensions. Call `gl.GetString()` to get a list of GL extensions.

`glu.GetString()` is an initialization routine. Calling it after a `gl.NewList()` results in undefined behavior.

Please consult an OpenGL reference manual for more information.

INPUTS

`name` specifies a symbolic constant, one of `#GLU_VERSION`, or `#GLU_EXTENSIONS`

RESULTS

`string` string describing the GLU version or the GLU extensions that are supported

7.7 glu.LookAt

NAME

glu.LookAt – define a viewing transformation

SYNOPSIS

```
glu.LookAt(Ex, Ey, Ez, Lx, Ly, Lz, Ux, Uy, Uz)
```

FUNCTION

`glu.LookAt()` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

The matrix maps the reference point to the negative z axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the UP vector projected onto the viewing plane is mapped to the positive y axis so that it points upward in the viewport. The UP vector must not be parallel to the line of sight from the eye point to the reference point.

Please consult an OpenGL reference manual for more information.

INPUTS

`Ex`	specifies the x position of the eye point
`Ey`	specifies the y position of the eye point
`Ez`	specifies the z position of the eye point
`Lx`	specifies the x position of the reference point
`Ly`	specifies the y position of the reference point
`Lz`	specifies the z position of the reference point
`Ux`	specifies the x direction of the up vector
`Uy`	specifies the y direction of the up vector
`Uz`	specifies the z direction of the up vector

7.8 glu.NewNurbsRenderer

NAME

glu.NewNurbsRenderer – create a NURBS object

SYNOPSIS

```
nurb = glu.NewNurbsRenderer()
```

FUNCTION

`glu.NewNurbsRenderer()` creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions.

NURBS objects are automatically deleted by Hollywood's garbage collector when no longer in use.

Please consult an OpenGL reference manual for more information.

INPUTS
none

RESULTS

`nurb`	a new NURBS object

7.9 glu.NewQuadric

NAME
glu.NewQuadric – create a quadrics object

SYNOPSIS
```
quad = glu.NewQuadric()
```

FUNCTION
`glu.NewQuadric()` creates and returns a new quadrics object. This object must be referred to when calling quadrics rendering and control functions.

Quadrics objects are automatically deleted by Hollywood's garbage collector when no longer in use.

Please consult an OpenGL reference manual for more information.

INPUTS
none

RESULTS

`quad`	handle to a new quadrics object

7.10 glu.Ortho2D

NAME
glu.Ortho2D – define a 2D orthographic projection matrix

SYNOPSIS
```
glu.Ortho2D(left, right, bottom, top)
```

FUNCTION
`glu.Ortho2D()` sets up a two-dimensional orthographic viewing region. This is equivalent to calling `gl.Ortho()` with near = -1 and far = 1.

Please consult an OpenGL reference manual for more information.

INPUTS

`left`	specify the coordinate for the left vertical clipping planes
`right`	specify the coordinate for the right vertical clipping planes
`bottom`	specify the coordinates for the bottom horizontal clipping planes
`top`	specify the coordinates for the top horizontal clipping planes

7.11 glu.Perspective

NAME

glu.Perspective – set up a perspective projection matrix

SYNOPSIS

```
glu.Perspective(fovy, aspect, near, far)
```

FUNCTION

`glu.Perspective()` specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in `glu.Perspective()` should match the aspect ratio of the associated viewport. For example, aspect = 2.0 means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by `glu.Perspective()` is multipled by the current matrix, just as if `gl.MultMatrix()` were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to `glu.Perspective()` with a call to `gl.LoadIdentity()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`fovy`	specifies the field of view angle, in degrees, in the y direction
`aspect`	specifies the aspect ratio that determines the field of view in the x direction; the aspect ratio is the ratio of x (width) to y (height)
`near`	specifies the distance from the viewer to the near clipping plane (always positive)
`far`	specifies the distance from the viewer to the far clipping plane (always positive)

7.12 glu.PickMatrix

NAME

glu.PickMatrix – define a picking region

SYNOPSIS

```
glu.PickMatrix(x, y, deltax, deltay, viewportArray)
```

FUNCTION

`glu.PickMatrix()` creates a projection matrix that can be used to restrict drawing to a small region of the viewport. This is typically useful to determine what objects are being drawn near the cursor. Use `glu.PickMatrix()` to restrict drawing to a small region around the cursor. Then, enter selection mode (with `gl.RenderMode()`) and rerender the scene. All primitives that would have been drawn near the cursor are identified and stored in the selection buffer.

The matrix created by `glu.PickMatrix()` is multiplied by the current matrix just as if `gl.MultMatrix()` is called with the generated matrix. To effectively use the generated pick matrix for picking, first call `gl.LoadIdentity()` to load an identity matrix onto

the perspective matrix stack. Then call `glu.PickMatrix()`, and finally, call a command (such as `glu.Perspective()`) to multiply the perspective matrix by the pick matrix.

When using `glu.PickMatrix()` to pick NURBS, be careful to turn off the NURBS property `#GLU_AUTO_LOAD_MATRIX`. If `#GLU_AUTO_LOAD_MATRIX` is not turned off, then any NURBS surface rendered is subdivided differently with the pick matrix than the way it was subdivided without the pick matrix.

The four window coordinates for `viewportArray` can easily be obtained by calling `gl.Get()` with `#GL_VIEWPORT`. See Section 6.57 [gl.Get], page 81, for details.

Please consult an OpenGL reference manual for more information.

INPUTS

`x` specify the x center of a picking region in window coordinates

`y` specify the y center of a picking region in window coordinates

`deltax` specify the width of the picking region in window coordinates

`deltay` specify the height of the picking region in window coordinates

`viewportArray`
the four window coordinates of the viewport in a table

7.13 glu.Project

NAME

glu.Project – map object coordinates to window coordinates

SYNOPSIS

```
e,wx,wy,wz = glu.Project(objx, objy, objz, model, proj, view)
```

FUNCTION

`glu.Project()` transforms the specified object coordinates into window coordinates using `model`, `proj`, and `view`. The result is stored in `wx`, `wy`, and `wz`. A return value of `#GL_TRUE` indicates success, a return value of `#GL_FALSE` indicates failure.

Please consult an OpenGL reference manual for more information.

INPUTS

`objx` specify the object x coordinate

`objy` specify the object y coordinate

`objz` specify the object z coordinate

`model` specifies the current modelview matrix as a table

`proj` specifies the current projection matrix as a table

`view` specifies the current viewport as a table

RESULTS

`e` error code

`wx` computed window x coordinate

`wy` computed window y coordinate

`wz` computed window z coordinate

7.14 glu.ScaleImage

NAME

glu.ScaleImage – scale an image to an arbitrary size

SYNOPSIS

```
err, pixelsOut = glu.ScaleImage(format, widthIn, heightIn, pixelsIn,
                    widthOut, heightOut)
```

FUNCTION

This function does the same as `glu.ScaleImageRaw()` except that the pixel data is not passed and returned as a raw memory buffer but as a table containing width*height*depth number of elements describing a pixel each. This is of course not as efficient as using raw memory buffers because the table's pixel data has to be copied to a raw memory buffer first and then has to be mapped to a table again.

See Section 7.15 [glu.ScaleImageRaw], page 221, for more details on the parameters accepted by this function.

Note that `glu.ScaleImage()` expects data of type `#GL_FLOAT` inside the `pixelsIn` table. `#GL_FLOAT` pixel data will also be written to the return table `pixelsOut`.

Please consult an OpenGL reference manual for more information.

INPUTS

`format` specifies the format of the pixel data (see above)

`widthIn` specifies the width of the source image that is scaled

`heightIn` specifies the height of the source image that is scaled

`pixelsIn` specifies a table containing the pixel data of the source image

`widthOut` specifies the width of the destination image

`heightOut`
specifies the height of the destination image

RESULTS

`error` error code or 0 for success

`pixelsOut`
table containing the scaled image data

7.15 glu.ScaleImageRaw

NAME

glu.ScaleImageRaw – scale an image to an arbitrary size

SYNOPSIS

```
error = glu.ScaleImageRaw(format, widthIn, heightIn, typeIn, pixelsIn,
           widthOut, heightOut, typeOut, pixelsOut)
```

FUNCTION

`glu.ScaleImageRaw()` scales a pixel image using the appropriate pixel store modes to unpack data from the source image and pack data into the destination image.

When shrinking an image, `glu.ScaleImageRaw()` uses a box filter to sample the source image and create pixels for the destination image. When an image is magnified, the pixels from the source image are linearly interpolated to create the destination image.

`format` must be one of `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX`, `#GL_DEPTH_COMPONENT`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, and `#GL_LUMINANCE_ALPHA`.

`typeIn` and `typeOut` must be one of `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, or `#GL_FLOAT`.

A return value of 0 indicates success. Otherwise glu.ScaleImage() returns a GLU error code that indicates what the problem is (See Section 7.5 [glu.ErrorString], page 215, for details.).

Please note that this command operates directly with memory pointers. There is also a version which works with tables instead of memory pointers, but this is slower of course. See Section 7.14 [glu.ScaleImage], page 221, for details. See Section 3.7 [Working with pointers], page 11, for details on how to use memory pointers with Hollywood.

Please consult an OpenGL reference manual for more information.

INPUTS

`format` specifies the format of the pixel data (see above)

`widthIn` specifies the width of the source image that is scaled

`heightIn` specifies the height of the source image that is scaled

`typeIn` specifies the data type for `pixelsIn` (see above)

`pixelsIn` specifies a pointer to the source image

`widthOut` specifies the width of the destination image

`heightOut`
specifies the height of the destination image

`typeOut` specifies the data type for `pixelsOut` (see above)

`pixelsOut`
specifies a pointer to the destination image

RESULTS

`error` error code or 0 for success

7.16 glu.UnProject

NAME

glu.UnProject – map window coordinates to object coordinates

SYNOPSIS

```
e,objx,objy,objz = glu.UnProject(winx, winy, winz, model, proj, view)
```

FUNCTION

`glu.UnProject()` maps the specified window coordinates into object coordinates using `model`, `proj`, and `view`. The result is stored in `objx`, `objy`, and `objz`. A return value of `#GL_TRUE` indicates success; a return value of `#GL_FALSE` indicates failure.

Please consult an OpenGL reference manual for more information.

INPUTS

`winx`	specify the window x coordinate
`winy`	specify the window y coordinate
`winz`	specify the window z coordinate
`model`	specifies the current modelview matrix as a table
`proj`	specifies the current projection matrix as a table
`view`	specifies the current viewport as a table

RESULTS

`e`	error code
`objx`	computed object x coordinate
`objy`	computed object y coordinate
`objz`	computed object z coordinate

7.17 nurb:BeginCurve

NAME

nurb:BeginCurve – mark the beginning of a NURBS curve definition

SYNOPSIS

```
nurb:BeginCurve()
```

FUNCTION

Use nurb:BeginCurve() to mark the beginning of a NURBS curve definition. After calling nurb:BeginCurve(), make one or more calls to `nurb:Curve()` to define the attributes of the curve. Exactly one of the calls to `nurb:Curve()` must have a curve type of `#GLU_MAP1_VERTEX_3` or `#GLU_MAP1_VERTEX_4`. To mark the end of the NURBS curve definition, call `nurb:EndCurve()`.

GL evaluators are used to render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with `gl.PushAttrib()` (`#GLU_EVAL_BIT`) and `gl.PopAttrib()`. See Section 6.116 [gl.PushAttrib], page 163, for details on exactly what state these calls preserve.

Please consult an OpenGL reference manual for more information.

INPUTS
none

7.18 nurb:BeginSurface

NAME
nurb:BeginSurface – mark the beginning of a NURBS surface definition

SYNOPSIS
`nurb:BeginSurface()`

FUNCTION
Use nurb:BeginSurface() to mark the beginning of a NURBS surface definition. After calling nurb:BeginSurface(), make one or more calls to `nurb:Surface()` to define the attributes of the surface. Exactly one of these calls to `nurb:Surface()` must have a surface type of `#GLU_MAP2_VERTEX_3` or `#GLU_MAP2_VERTEX_4`. To mark the end of the NURBS surface definition, call `nurb:EndSurface()`.

Trimming of NURBS surfaces is supported with `nurb:BeginTrim()`, `nurb:PwlCurve()`, `nurb:Curve()`, and `nurb:EndTrim()`. See Section 7.19 [nurb:BeginTrim], page 224, for details.

GL evaluators are used to render the NURBS surface as a set of polygons. Evaluator state is preserved during rendering with `gl.PushAttrib()` (`#GLU_EVAL_BIT`) and `gl.PopAttrib()`. See Section 6.116 [gl.PushAttrib], page 163, for details on exactly what state these calls preserve.

Please consult an OpenGL reference manual for more information.

INPUTS
none

7.19 nurb:BeginTrim

NAME
nurb:BeginTrim – mark the beginning of a NURBS trimming loop definition

SYNOPSIS
`nurb:BeginTrim()`

FUNCTION
Use nurb:BeginTrim() to mark the beginning of a trimming loop and `nurb:EndTrim()` to mark the end of a trimming loop. A trimming loop is a set of oriented curve segments (forming a closed curve) that define boundaries of a NURBS surface. You include these trimming loops in the definition of a NURBS surface, between calls to `nurb:BeginSurface()` and `nurb:EndSurface()`.

The definition for a NURBS surface can contain many trimming loops. For example, if you wrote a definition for a NURBS surface that resembled a rectangle with a hole punched out, the definition would contain two trimming loops. One loop would define the outer edge of the rectangle; the other would define the hole punched out of the

rectangle. The definitions of each of these trimming loops would be bracketed by a nurb:BeginTrim() / `nurb:EndTrim()` pair.

The definition of a single closed trimming loop can consist of multiple curve segments, each described as a piecewise linear curve (See Section 7.27 [nurb:PwlCurve], page 230, for details.) or as a single NURBS curve (See Section 7.21 [nurb:Curve], page 227, for details.), or as a combination of both in any order. The only library calls that can appear in a trimming loop definition (between the calls to nurb:BeginTrim() and `nurb:EndTrim()`) are `nurb:PwlCurve()` and `nurb:Curve()`.

The area of the NURBS surface that is displayed is the region in the domain to the left of the trimming curve as the curve parameter increases. Thus, the retained region of the NURBS surface is inside a counterclockwise trimming loop and outside a clockwise trimming loop. For the rectangle mentioned earlier, the trimming loop for the outer edge of the rectangle runs counter-clockwise, while the trimming loop for the punched-out hole runs clockwise.

If you use more than one curve to define a single trimming loop, the curve segments must form a closed loop (that is, the endpoint of each curve must be the starting point of the next curve, and the endpoint of the final curve must be the starting point of the first curve). If the endpoints of the curve are sufficiently close together but not exactly coincident, they will be coerced to match. If the endpoints are not sufficiently close, an error results (See Section 7.20 [nurb:Callback], page 225, for details.).

If a trimming loop definition contains multiple curves, the direction of the curves must be consistent (that is, the inside must be to the left of all of the curves). Nested trimming loops are legal as long as the curve orientations alternate correctly. If trimming curves are self-intersecting, or intersect one another, an error results.

If no trimming information is given for a NURBS surface, the entire surface is drawn.

Please consult an OpenGL reference manual for more information.

INPUTS

none

7.20 nurb:Callback

NAME

nurb:Callback – define a callback for a NURBS object

SYNOPSIS

```
nurb:Callback(which, func)
```

FUNCTION

nurb:Callback() is used to define a callback to be used by a NURBS object. If the specified callback is already defined, then it is replaced. If `func` is `Nil`, then this callback will not get invoked and the related data, if any, will be lost.

Except the error callback, these callbacks are used by NURBS tessellator (when `#GLU_NURBS_MODE` is set to be `#GLU_NURBS_TESSELLATOR`) to return back the OpenGL polygon primitives resulting from the tessellation. The error callback function is effective no

matter which value that #GLU_NURBS_MODE is set to. All other callback functions are effective only when #GLU_NURBS_MODE is set to #GLU_NURBS_TESSELLATOR.

All callbacks receive a handle to the NURBS object as their first parameter. The second parameter depends on the callback type specified in which.

The legal callbacks are as follows:

#GLU_NURBS_BEGIN
: The begin callback indicates the start of a primitive. The function receives an integer argument, which can be one of #GLU_LINES, #GLU_LINE_STRIP, #GLU_TRIANGLE_FAN, #GLU_TRIANGLE_STRIP, #GLU_TRIANGLES, or #GLU_QUAD_STRIP. The default begin callback function is Nil.

#GLU_NURBS_VERTEX
: The vertex callback indicates a vertex of the primitive. The coordinates of the vertex are passed in a table parameter as four floating-point values. All the generated vertices have dimension 3; that is, homogeneous coordinates have been transformed into affine coordinates. The default vertex callback function is Nil.

#GLU_NURBS_NORMAL
: The normal callback is invoked as the vertex normal is generated. The components of the normal are passed in a table parameter as three floating-point values. In the case of a NURBS curve, the callback function is effective only when the user provides a normal map (#GLU_MAP1_NORMAL). In the case of a NURBS surface, if a normal map (#GLU_MAP2_NORMAL) is provided, then the generated normal is computed from the normal map. If a normal map is not provided, then a surface normal is computed in a manner similar to that described for evaluators when #GLU_AUTO_NORMAL is enabled. The default normal callback function is Nil.

#GLU_NURBS_COLOR
: The color callback is invoked as the color of a vertex is generated. The components of the color are passed in a table parameter. This callback is effective only when the user provides a color map (#GLU_MAP1_COLOR_4 or #GLU_MAP2_COLOR_4). The table passed to this callback contains four floating-point components: R, G, B, A. The default color callback function is Nil.

#GLU_NURBS_TEXTURE_COORD
: The texture callback is invoked as the texture coordinates of a vertex are generated. These coordinates are passed in a table parameter. The number of texture coordinates can be 1, 2, 3, or 4 depending on which type of texture map is specified (#GLU_MAP1_TEXTURE_COORD_1, #GLU_MAP1_TEXTURE_COORD_2, #GLU_MAP1_TEXTURE_COORD_3, #GLU_MAP1_TEXTURE_COORD_4, #GLU_MAP2_TEXTURE_COORD_1, #GLU_MAP2_TEXTURE_COORD_2, #GLU_MAP2_TEXTURE_COORD_3, #GLU_MAP2_TEXTURE_COORD_4). If no texture map is specified, this callback function will not be called. The default texture callback function is Nil.

`#GLU_NURBS_END`
: The end callback is invoked at the end of a primitive. The default end callback function is `Nil`. This callback doesn't receive any parameters except a handle to the NURBS object.

Please consult an OpenGL reference manual for more information.

INPUTS

`which` specifies the callback being defined (see above)

`func` specifies the function that the callback calls

7.21 nurb:Curve

NAME

nurb:Curve – define the shape of a NURBS curve

SYNOPSIS

```
nurb:Curve(knotsArray, controlArray, type)
```

FUNCTION

Use nurb:Curve() to describe a NURBS curve.

When nurb:Curve() appears between a `nurb:BeginCurve()` / `nurb:EndCurve()` pair, it is used to describe a curve to be rendered. Positional, texture, and color coordinates are associated by presenting each as a separate nurb:Curve() between a `nurb:BeginCurve()` / `nurb:EndCurve()` pair. No more than one call to nurb:Curve() for each of color, position, and texture data can be made within a single `nurb:BeginCurve()` / `nurb:EndCurve()` pair. Exactly one call must be made to describe the position of the curve (a type of `#GLU_MAP1_VERTEX_3` or `#GLU_MAP1_VERTEX_4`).

When nurb:Curve() appears between a nurb:BeginTrim() / `nurb:EndTrim()` pair, it is used to describe a trimming curve on a NURBS surface. If type is `#GLU_MAP1_TRIM_2`, then it describes a curve in two-dimensional (u and v) parameter space. If it is `#GLU_MAP1_TRIM_3`, then it describes a curve in two-dimensional homogeneous (u, v, and w) parameter space. See Section 7.19 [nurb:BeginTrim], page 224, for more discussion about trimming curves.

Please consult an OpenGL reference manual for more information.

INPUTS

`knotsArray`
: specifies an array of non-decreasing knot values

`controlArray`
: specifies an array of control points; the coordinates must agree with `type`, specified below

`type` specifies the type of the curve (see above)

7.22 nurb:EndCurve

NAME
nurb:EndCurve – mark the end of a NURBS curve definition

SYNOPSIS
`nurb:EndCurve()`

FUNCTION
nurb:EndCurve() marks the end of a NURBS curve definition. See Section 7.17 [nurb:BeginCurve], page 223, for details.

Please consult an OpenGL reference manual for more information.

INPUTS
none

7.23 nurb:EndSurface

NAME
nurb:EndSurface – mark the end of a NURBS surface definition

SYNOPSIS
`nurb:EndSurface()`

FUNCTION
nurb:EndSurface() marks the end of a NURBS surface definition. See Section 7.18 [nurb:BeginSurface], page 224, for details.

Please consult an OpenGL reference manual for more information.

INPUTS
none

7.24 nurb:EndTrim

NAME
nurb:EndTrim – mark the end of a trimming loop definition

SYNOPSIS
`nurb:EndTrim()`

FUNCTION
nurb:EndTrim() marks the end of a trimming loop definition. See Section 7.19 [nurb:BeginTrim], page 224, for details.

Please consult an OpenGL reference manual for more information.

INPUTS
none

7.25 nurb:GetProperty

NAME

nurb:GetProperty – get a NURBS property

SYNOPSIS

```
value = nurb:GetProperty(property)
```

FUNCTION

nurb:GetProperty() can be used to get the state of a NURBS property. See Section 7.28 [nurb:SetProperty], page 230, for a list of accepted properties.

Please consult an OpenGL reference manual for more information.

INPUTS

`property` specifies the property to be set (see above)

RESULTS

`value` state of the specified property

7.26 nurb:LoadSamplingMatrices

NAME

nurb:LoadSamplingMatrices – load NURBS sampling and culling matrices

SYNOPSIS

```
nurb:LoadSamplingMatrices(modelArray, perspectiveArray, viewArray)
```

FUNCTION

nurb:LoadSamplingMatrices() uses `modelArray`, `perspectiveArray`, and `viewArray` to recompute the sampling and culling matrices stored in `nurb`. The sampling matrix determines how finely a NURBS curve or surface must be tessellated to satisfy the sampling tolerance (as determined by the `#GLU_SAMPLING_TOLERANCE` property). The culling matrix is used in deciding if a NURBS curve or surface should be culled before rendering (when the `#GLU_CULLING` property is turned on).

nurb:LoadSamplingMatrices() is necessary only if the `#GLU_AUTO_LOAD_MATRIX` property is turned off (See Section 7.28 [nurb:SetProperty], page 230, for details.). Although it can be convenient to leave the `#GLU_AUTO_LOAD_MATRIX` property turned on, there can be a performance penalty for doing so. (A round trip to the GL server is needed to fetch the current values of the modelview matrix, projection matrix, and viewport.)

Please consult an OpenGL reference manual for more information.

INPUTS

`modelArray`
: specifies a table containing a modelview matrix

`perspectiveArray`
: specifies a table containing a projection matrix

`viewArray`
: specifies a table containing viewport coordinates

7.27 nurb:PwlCurve

NAME

nurb:PwlCurve – describe a piecewise linear NURBS trimming curve

SYNOPSIS

```
nurb:PwlCurve(dataArray, type)
```

FUNCTION

nurb:PwlCurve() describes a piecewise linear trimming curve for a NURBS surface. A piecewise linear curve consists of a list of coordinates of points in the parameter space for the NURBS surface to be trimmed. These points are connected with line segments to form a curve. If the curve is an approximation to a curve that is not piecewise linear, the points should be close enough in parameter space that the resulting path appears curved at the resolution used in the application.

If `type` is `#GLU_MAP1_TRIM_2`, then it describes a curve in two-dimensional (u and v) parameter space. If it is `#GLU_MAP1_TRIM_3`, then it describes a curve in two-dimensional homogeneous (u, v, and w) parameter space. See Section 7.19 [nurb:BeginTrim], page 224, for more information about trimming curves.

To describe a trim curve that closely follows the contours of a NURBS surface, call `nurb:Curve()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`dataArray`
: specifies an array containing the curve points

`type`
: specifies the type of curve; must be either `#GLU_MAP1_TRIM_2` or `#GLU_MAP1_TRIM_3`

7.28 nurb:SetProperty

NAME

nurb:SetProperty – set a NURBS property

SYNOPSIS

```
nurb:SetProperty(property, value)
```

FUNCTION

nurb:SetProperty() is used to control properties stored in a NURBS object. These properties affect the way that a NURBS curve is rendered. The accepted values for `property` are as follows:

`#GLU_NURBS_MODE`
: `value` should be set to be either `#GLU_NURBS_RENDERER` or `#GLU_NURBS_TESSELLATOR`. When set to `#GLU_NURBS_RENDERER`, NURBS objects are tessellated into OpenGL primitives and sent to the pipeline for rendering. When set to `#GLU_NURBS_TESSELLATOR`, NURBS objects are tessellated into OpenGL primitives but the vertices, normals, colors, and/or textures are retrieved back through a callback interface (See Section 7.20 [nurb:Callback],

page 225, for details.). This allows the user to cache the tessellated results for further processing. The initial value is #GLU_NURBS_RENDERER.

#GLU_SAMPLING_METHOD

Specifies how a NURBS surface should be tessellated. value may be one of #GLU_PATH_LENGTH, #GLU_PARAMETRIC_ERROR, #GLU_DOMAIN_DISTANCE, #GLU_OBJECT_PATH_LENGTH, or #GLU_OBJECT_PARAMETRIC_ERROR. When set to #GLU_PATH_LENGTH, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by #GLU_SAMPLING_TOLERANCE.

#GLU_PARAMETRIC_ERROR specifies that the surface is rendered in such a way that the value specified by #GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.

#GLU_DOMAIN_DISTANCE allows users to specify, in parametric coordinates, how many sample points per unit length are taken in u, v direction.

#GLU_OBJECT_PATH_LENGTH is similar to #GLU_PATH_LENGTH except that it is view independent; that is, the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by #GLU_SAMPLING_TOLERANCE.

#GLU_OBJECT_PARAMETRIC_ERROR is similar to #GLU_PARAMETRIC_ERROR except that it is view independent; that is, the surface is rendered in such a way that the value specified by #GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate.

The initial value of #GLU_SAMPLING_METHOD is #GLU_PATH_LENGTH.

#GLU_SAMPLING_TOLERANCE

Specifies the maximum length, in pixels or in object space length unit, to use when the sampling method is set to #GLU_PATH_LENGTH or #GLU_OBJECT_PATH_LENGTH. The NURBS code is conservative when rendering a curve or surface, so the actual length can be somewhat shorter. The initial value is 50.0 pixels.

#GLU_PARAMETRIC_TOLERANCE

Specifies the maximum distance, in pixels or in object space length unit, to use when the sampling method is #GLU_PARAMETRIC_ERROR or #GLU_OBJECT_PARAMETRIC_ERROR. The initial value is 0.5.

#GLU_U_STEP

Specifies the number of sample points per unit length taken along the u axis in parametric coordinates. It is needed when #GLU_SAMPLING_METHOD is set to #GLU_DOMAIN_DISTANCE. The initial value is 100.

#GLU_V_STEP

Specifies the number of sample points per unit length taken along the v axis in parametric coordinate. It is needed when #GLU_SAMPLING_METHOD is set to #GLU_DOMAIN_DISTANCE. The initial value is 100.

#GLU_DISPLAY_MODE

value can be set to #GLU_OUTLINE_POLYGON, #GLU_FILL, or #GLU_OUTLINE_PATCH. When #GLU_NURBS_MODE is set to be #GLU_NURBS_RENDERER, value defines how a NURBS surface should be rendered. When value is set to #GLU_FILL, the surface is rendered as a set of polygons. When value is set to #GLU_OUTLINE_POLYGON, the NURBS library draws only the outlines of the polygons created by tessellation. When value is set to #GLU_OUTLINE_PATCH just the outlines of patches and trim curves defined by the user are drawn.

When #GLU_NURBS_MODE is set to be #GLU_NURBS_TESSELLATOR, value defines how a NURBS surface should be tessellated. When #GLU_DISPLAY_MODE is set to# GLU_FILL or #GLU_OUTLINE_POLYGON, the NURBS surface is tessellated into OpenGL triangle primitives that can be retrieved back through callback functions. If #GLU_DISPLAY_MODE is set to #GLU_OUTLINE_PATCH, only the outlines of the patches and trim curves are generated as a sequence of line strips that can be retrieved back through callback functions.

The initial value is #GLU_FILL.

#GLU_CULLING

value is a boolean value that, when set to #GLU_TRUE, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The initial value is #GLU_FALSE.

#GLU_AUTO_LOAD_MATRIX

value is a boolean value. When set to #GLU_TRUE, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the GL server to compute sampling and culling matrices for each NURBS curve that is rendered. Sampling and culling matrices are required to determine the tessellation of a NURBS surface into line segments or polygons and to cull a NURBS surface if it lies outside the viewport.

If this mode is set to #GLU_FALSE, then the program needs to provide a projection matrix, a modelview matrix, and a viewport for the NURBS renderer to use to construct sampling and culling matrices. This can be done with the nurb:LoadSamplingMatrices() function. This mode is initially set to #GLU_TRUE. Changing it from #GLU_TRUE to #GLU_FALSE does not affect the sampling and culling matrices until nurb:LoadSamplingMatrices() is called.

If #GLU_AUTO_LOAD_MATRIX is true, sampling and culling may be executed incorrectly if NURBS routines are compiled into a display list.

Please consult an OpenGL reference manual for more information.

INPUTS

property specifies the property to be set (see above)

value specifies the value of the indicated property (see above)

7.29 nurb:Surface

NAME

nurb:Surface – define the shape of a NURBS surface

SYNOPSIS

`nurb:Surface(sKnotsArray, tKnotsArray, controlArray, type)`

FUNCTION

Use nurb:Surface() within a NURBS (Non-Uniform Rational B-Spline) surface definition to describe the shape of a NURBS surface (before any trimming). To mark the beginning of a NURBS surface definition, use the `nurb:BeginSurface()` command. To mark the end of a NURBS surface definition, use the `nurb:EndSurface()` command. Call nurb:Surface() within a NURBS surface definition only.

Positional, texture, and color coordinates are associated with a surface by presenting each as a separate nurb:Surface() between a `nurb:BeginSurface()` / `nurb:EndSurface()` pair. No more than one call to nurb:Surface() for each of color, position, and texture data can be made within a single `nurb:BeginSurface()` / `nurb:EndSurface()` pair. Exactly one call must be made to describe the position of the surface (a type of `#GLU_MAP2_VERTEX_3` or `#GLU_MAP2_VERTEX_4`).

A NURBS surface can be trimmed by using the commands `nurb:Curve()` and `nurb:PwlCurve()` between calls to `nurb:BeginTrim()` and `nurb:EndTrim()`.

Please consult an OpenGL reference manual for more information.

INPUTS

`sKnotsArray`
: specifies an array of non-decreasing knot values in the parametric u direction

`tKnotsArray`
: specifies an array of non-decreasing knot values in the parametric v direction

`controlArray`
: specifies an array containing control points for the NURBS surface

`type`
: specifies type of the surface; can be any of the valid two-dimensional evaluator types (such as `#GLU_MAP2_VERTEX_3` or `#GLU_MAP2_COLOR_4`)

7.30 quad:Cylinder

NAME

quad:Cylinder – draw a cylinder

SYNOPSIS

`quad:Cylinder(base, top, height, slices, stacks)`

FUNCTION

quad:Cylinder() draws a cylinder oriented along the z axis. The base of the cylinder is placed at z = 0 and the top at z = height. Like a sphere, a cylinder is subdivided around the z axis into slices and along the z axis into stacks.

Note that if top is set to 0.0, this routine generates a cone.

If the orientation is set to `#GLU_OUTSIDE` (with `quad:Orientation()`), then any generated normals point away from the z axis. Otherwise, they point toward the z axis.

If texturing is turned on (with `quad:Texture()`), then texture coordinates are generated so that t ranges linearly from 0.0 at z = 0 to 1.0 at z = height, and s ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

Please consult an OpenGL reference manual for more information.

INPUTS

`base`	specifies the radius of the cylinder at z = 0
`top`	specifies the radius of the cylinder at z = height
`height`	specifies the height of the cylinder
`slices`	specifies the number of subdivisions around the z axis
`stacks`	specifies the number of subdivisions along the z axis

7.31 quad:Disk

NAME

quad:Disk – draw a disk

SYNOPSIS

```
quad:Disk(inner, outer, slices, loops)
```

FUNCTION

quad:Disk() renders a disk on the z = 0 plane. The disk has a radius of outer and contains a concentric circular hole with a radius of inner. If `inner` is 0, then no hole is generated. The disk is subdivided around the z axis into slices (like pizza slices) and also about the z axis into rings (as specified by `slices` and `loops`, respectively).

With respect to orientation, the +z side of the disk is considered to be "outside" (See Section 7.34 [quad:Orientation], page 236, for details.). This means that if the orientation is set to `#GLU_OUTSIDE`, then any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing has been turned on (with `quad:Texture()`), texture coordinates are generated linearly such that where r = outer , the value at (r, 0, 0) is (1, 0.5), at (0, r, 0) it is (0.5, 1), at (-r, 0, 0) it is (0, 0.5), and at (0, -r, 0) it is (0.5, 0).

Please consult an OpenGL reference manual for more information.

INPUTS

`inner`	specifies the inner radius of the disk (may be 0)
`outer`	specifies the outer radius of the disk
`slices`	specifies the number of subdivisions around the z axis
`loops`	specifies the number of concentric rings about the origin into which the disk is subdivided

7.32 quad:DrawStyle

NAME

quad:DrawStyle – specify the draw style desired for quadrics

SYNOPSIS

`quad:DrawStyle(draw)`

FUNCTION

quad:DrawStyle() specifies the draw style for quadrics rendered with `quad`. The legal values are as follows:

`#GLU_FILL`
: Quadrics are rendered with polygon primitives. The polygons are drawn in a counter-clockwise fashion with respect to their normals (as defined with `quad:Orientation()`).

`#GLU_LINE`
: Quadrics are rendered as a set of lines.

`#GLU_SILHOUETTE`
: Quadrics are rendered as a set of lines, except that edges separating coplanar faces will not be drawn.

`#GLU_POINT`
: Quadrics are rendered as a set of points.

Please consult an OpenGL reference manual for more information.

INPUTS

`draw` specifies the desired draw style (see above)

7.33 quad:Normals

NAME

quad:Normals – specify what kind of normals are desired for quadrics

SYNOPSIS

`quad:Normals(normal)`

FUNCTION

quad:Normals() specifies what kind of normals are desired for quadrics rendered with `quad`. The legal values are as follows:

`#GLU_NONE`
: No normals are generated.

`#GLU_FLAT`
: One normal is generated for every facet of a quadric.

`#GLU_SMOOTH`
: One normal is generated for every vertex of a quadric. This is the initial value.

Please consult an OpenGL reference manual for more information.

INPUTS

`normal` specifies the desired type of normals (see above)

7.34 quad:Orientation

NAME

quad:Orientation – specify inside/outside orientation for quadrics

SYNOPSIS

```
quad:Orientation(orientation)
```

FUNCTION

quad:Orientation() specifies what kind of orientation is desired for quadrics rendered with `quad`. The orientation values are as follows:

`#GLU_OUTSIDE`
: Quadrics are drawn with normals pointing outward (the initial value).

`#GLU_INSIDE`
: Quadrics are drawn with normals pointing inward.

Note that the interpretation of outward and inward depends on the quadric being drawn.

Please consult an OpenGL reference manual for more information.

INPUTS

`orientation`
: specifies the desired orientation (see above)

7.35 quad:PartialDisk

NAME

quad:PartialDisk – draw an arc of a disk

SYNOPSIS

```
quad:PartialDisk(inner, outer, slices, loops, start, sweep)
```

FUNCTION

quad:PartialDisk() renders a partial disk on the z = 0 plane. A partial disk is similar to a full disk, except that only the subset of the disk from `start` through `start + sweep` is included (where 0 degrees is along the +y axis, 90 degrees along the +x axis, 180 degrees along the -y axis, and 270 degrees along the -x axis).

The partial disk has a radius of `outer` and contains a concentric circular hole with a radius of `inner`. If `inner` is 0, then no hole is generated. The partial disk is subdivided around the z axis into slices (like pizza slices) and also about the z axis into rings (as specified by `slices` and `loops`, respectively).

With respect to orientation, the +z side of the partial disk is considered to be outside (See Section 7.34 [quad:Orientation], page 236, for details.). This means that if the

orientation is set to `#GLU_OUTSIDE`, then any normals generated point along the +z axis. Otherwise, they point along the -z axis.

If texturing is turned on (with `quad:Texture()`), texture coordinates are generated linearly such that where r = outer , the value at (r, 0, 0) is (1.0, 0.5), at (0, r, 0) it is (0.5, 1.0), at (-r, 0, 0) it is (0.0, 0.5), and at (0, -r, 0) it is (0.5, 0.0).

Please consult an OpenGL reference manual for more information.

INPUTS

`inner`	specifies the inner radius of the partial disk (can be 0)
`outer`	specifies the outer radius of the partial disk
`slices`	specifies the number of subdivisions around the z axis
`loops`	specifies the number of concentric rings about the origin into which the partial disk is subdivided
`start`	specifies the starting angle, in degrees, of the disk portion
`sweep`	specifies the sweep angle, in degrees, of the disk portion

7.36 quad:Texture

NAME

quad:Texture – specify if texturing is desired for quadrics

SYNOPSIS

```
quad:Texture(texture)
```

FUNCTION

quad:Texture() specifies if texture coordinates should be generated for quadrics rendered with `quad`. If the value of texture is `#GLU_TRUE`, then texture coordinates are generated, and if texture is `#GLU_FALSE`, they are not. The initial value is `#GLU_FALSE`.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

Please consult an OpenGL reference manual for more information.

INPUTS

`texture`	specifies a flag indicating if texture coordinates should be generated

7.37 quad:Sphere

NAME

quad:Sphere – draw a sphere

SYNOPSIS

```
quad:Sphere(radius, slices, stacks)
```

FUNCTION

quad:Sphere() draws a sphere of the given radius centered around the origin. The sphere is subdivided around the z axis into slices and along the z axis into stacks (similar to lines of longitude and latitude).

If the orientation is set to `#GLU_OUTSIDE` (with `quad:Orientation()`), then any normals generated point away from the center of the sphere. Otherwise, they point toward the center of the sphere.

If texturing is turned on (with `quad:Texture()`), then texture coordinates are generated so that t ranges from 0.0 at z = -radius to 1.0 at z = radius (t increases linearly along longitudinal lines), and s ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

Please consult an OpenGL reference manual for more information.

INPUTS

`radius` specifies the radius of the sphere

`slices` specifies the number of subdivisions around the z axis (similar to lines of longitude)

`stacks` specifies the number of subdivisions along the z axis (similar to lines of latitude)

8 GLFW reference

8.1 glfw.GetJoystickAxes

NAME

glfw.GetJoystickAxes – get states of all joystick axes (V1.1)

SYNOPSIS

```
t, count = glfw.GetJoystickAxes(port)
```

FUNCTION

This function returns the values of all axes of the specified joystick in a table. Each element in the table is a value between -1.0 and 1.0. The `port` argument must be set to a valid joystick identifier between `#GLFW_JOYSTICK_1` and `#GLFW_JOYSTICK_16`.

INPUTS

`port`	the joystick port to query

RESULTS

`t`	table containing axes data
`count`	number of entries in table

8.2 glfw.GetJoystickButtons

NAME

glfw.GetJoystickButtons – get states of all joystick buttons (V1.1)

SYNOPSIS

```
t, count = glfw.GetJoystickButtons(port)
```

FUNCTION

This function returns the states of all buttons of the specified joystick in a table. Each element in the table is either `True` if the button is pressed or `False` if it isn't pressed. The `port` argument must be set to a valid joystick identifier between `#GLFW_JOYSTICK_1` and `#GLFW_JOYSTICK_16`.

INPUTS

`port`	the joystick port to query

RESULTS

`t`	table containing button states
`count`	number of entries in table

8.3 glfw.GetJoystickName

NAME

glfw.GetJoystickName – return joystick name (V1.1)

SYNOPSIS

```
name$ = glfw.GetJoystickName(port)
```

FUNCTION

This function returns the name of the specified joystick. The `port` argument must be set to a valid joystick identifier between `#GLFW_JOYSTICK_1` and `#GLFW_JOYSTICK_16`.

INPUTS

`port` the joystick port to query

RESULTS

`name$` name of joystick

8.4 glfw.JoystickPresent

NAME

glfw.JoystickPresent – check if there is a joystick at the specified port (V1.1)

SYNOPSIS

```
bool = glfw.JoystickPresent(port)
```

FUNCTION

This function returns whether the specified joystick is present. The `port` argument must be set to a valid joystick identifier between `#GLFW_JOYSTICK_1` and `#GLFW_JOYSTICK_16`.

INPUTS

`port` the joystick port to query

RESULTS

`bool` `True` or `False` depending on whether there is a joystick at the specified port

9 Hollywood bridge

9.1 gl.BitmapFromBrush

NAME

gl.BitmapFromBrush – draw a bitmap from a brush's mask

SYNOPSIS

```
gl.BitmapFromBrush(xorig, yorig, xmove, ymove, id)
```

FUNCTION

This function does the same as `gl.Bitmap()` except that the pixel data is fetched from the mask of the Hollywood brush specified by `id`. If the brush specified in `id` doesn't have a mask, an error will be generated.

See Section 6.7 [gl.Bitmap], page 27, for details.

INPUTS

`xorig` specify the location of the x origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.

`yorig` specify the location of the y origin in the bitmap image. The origin is measured from the lower left corner of the bitmap, with right and up being the positive axes.

`xmove` specify the x offset to be added to the current raster position after the bitmap is drawn

`ymove` specify the y offset to be added to the current raster position after the bitmap is drawn

`id` identifier of a Hollywood brush that has a mask

9.2 gl.DrawPixelsFromBrush

NAME

gl.DrawPixelsFromBrush – draw a Hollywood brush to the frame buffer

SYNOPSIS

```
gl.DrawPixelsFromBrush(id)
```

FUNCTION

This function does the same as `gl.DrawPixels()` except that the pixel data is fetched from the Hollywood brush specified by `id`.

See Section 6.36 [gl.DrawPixels], page 59, for details.

INPUTS

`id` identifier of a Hollywood brush

9.3 gl.GetCurrentContext

NAME
gl.GetCurrentContext – get current OpenGL context

SYNOPSIS
```
id = gl.GetCurrentContext()
```

FUNCTION
This function returns the identifier of the display whose OpenGL context is the current one.

You can use `gl.SetCurrentContext()` to set the current GL context.

INPUTS
none

RESULTS

`id`	identifier of the display whose OpenGL context is the current one

9.4 gl.GetTexImageToBrush

NAME
gl.GetTexImageToBrush – return a texture image as a brush

SYNOPSIS
```
[id] = gl.GetTexImageToBrush(target, level, id)
```

FUNCTION
This function does the same as `gl.GetTexImage()` except that the pixel data is converted into the Hollywood brush specified by `id`. If there is already a brush that uses the identifier `id`, it will be freed first. If you specify `Nil` in the `id` argument, `gl.GetTexImageToBrush()` will automatically choose a vacant identifier for this brush and return it to you.

See Section 6.71 [gl.GetTexImage], page 108, for details.

INPUTS

`target`	specifies which texture is to be obtained (must be `#GL_TEXTURE_1D` or `#GL_TEXTURE_2D`)
`level`	specifies the level-of-detail number of the desired image; level 0 is the base image level; level n is the nth mipmap reduction image
`id`	id for the new brush or `Nil` for auto id selection

RESULTS

`id`	optional: identifier of the brush; will only be returned when you pass `Nil` as argument 3 (see above)

9.5 gl.ReadPixelsToBrush

NAME

gl.ReadPixelsToBrush – read pixels from the frame buffer to a brush

SYNOPSIS

```
[id] = gl.ReadPixelsToBrush(x, y, width, height, id)
```

FUNCTION

This function does the same as `gl.ReadPixels()` except that the pixel data is converted into the Hollywood brush specified by `id`. If there is already a brush that uses the identifier `id`, it will be freed first. If you specify `Nil` in the `id` argument, `gl.ReadPixelsToBrush()` will automatically choose a vacant identifier for this brush and return it to you.

See Section 6.122 [gl.ReadPixels], page 173, for details.

INPUTS

`x`	specify the left coordinate of a rectangular block of pixels
`y`	specify the lower coordinate of a rectangular block of pixels
`width`	width of the pixel rectangle
`height`	height of the pixel rectangle
`id`	id for the new brush or `Nil` for auto id selection

RESULTS

`id`	optional: identifier of the brush; will only be returned when you pass `Nil` as argument 5 (see above)

9.6 gl.SetCurrentContext

NAME

gl.SetCurrentContext – set current OpenGL context

SYNOPSIS

```
gl.SetCurrentContext(id)
```

FUNCTION

This function makes the OpenGL context of the display specified by `id` current. With GL Galore, every Hollywood display maintains its own GL context. You can use this function to set the current GL context that all further calls to OpenGL should use.

You can use `gl.GetCurrentContext()` to get the current GL context.

INPUTS

`id`	identifier of a Hollywood display whose GL context should be made current

9.7 gl.TexImageFromBrush

NAME

gl.TexImageFromBrush – specify a 1D or 2D texture image from brush

SYNOPSIS

`gl.TexImageFromBrush(level, id)`

FUNCTION

This command does the same as `gl.TexImage()` but fetches the pixel data from the Hollywood brush specified by `id`. Two-dimensional texturing will be used automatically if the brush has more than one row.

See Section 6.138 [gl.TexImage], page 192, for details.

INPUTS

`level`	specifies the level-of-detail number; level 0 is the base image level, level n is the nth mipmap reduction image
`id`	identifier of a Hollywood brush

9.8 gl.TexSubImageFromBrush

NAME

gl.TexSubImageFromBrush – specify a 1D or 2D texture subimage from brush

SYNOPSIS

`gl.TexSubImageFromBrush(level, id, xoffset[, yoffset])`

FUNCTION

This command does the same as `gl.TexSubImage()` but fetches the pixel data from the Hollywood brush specified by `id`. Two-dimensional texturing will be used automatically if the brush has more than one row.

See Section 6.142 [gl.TexSubImage], page 202, for details.

INPUTS

`level`	specifies the level-of-detail number; level 0 is the base image level; level n is the nth mipmap reduction image
`id`	identifier of a Hollywood brush
`xoffset`	specifies a texel offset in the x direction within the texture array
`yoffset`	optional: specifies a texel offset in the y direction within the texture array; only required for 2D textures

9.9 glu.BuildMipmapsFromBrush

NAME

glu.BuildMipmapsFromBrush – create 1D or 2D mipmaps from brush

SYNOPSIS

```
error = glu.BuildMipmapsFromBrush(id)
```

FUNCTION

This command does the same as `glu.BuildMipmaps()` but fetches the pixel data from the Hollywood brush specified by `id`. Two-dimensional mipmaps will be created automatically if the brush has more than one row.

See Section 7.1 [glu.BuildMipmaps], page 211, for details.

INPUTS

`id`	identifier of a Hollywood brush

RESULTS

`error`	error code or 0 for success

Appendix A Licenses

A.1 LuaGL license

LuaGL is licensed under the terms of the MIT license reproduced below. This means that LuaGL is free software and can be used for both academic and commercial purposes at absolutely no cost.

Copyright (C) 2003-2012 by Fabio Guerra and Cleyde Marlyse.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

A.2 GLFW license

Copyright (c) 2002-2006 Marcus Geelnard

Copyright (c) 2006-2010 Camilla Berglund <elmindreda@elmindreda.org>

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

A.3 SGI Free Software B license

SGI FREE SOFTWARE LICENSE B (Version 2.0, Sept. 18, 2008)

Copyright (C) 1991-2006 Silicon Graphics, Inc. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without

restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice including the dates of first publication and either this permission notice or a reference to http://oss.sgi.com/projects/FreeB/ shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL SILICON GRAPHICS, INC. BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of Silicon Graphics, Inc. shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from Silicon Graphics, Inc.

A.4 LGPL license

GNU Lesser General Public License Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages–typically libraries–of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too,

receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that

uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an

application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than

a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies

directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED 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 LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (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 LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index

G

gl.Accum 19
gl.AlphaFunc 20
gl.AreTexturesResident 22
gl.ArrayElement 22
gl.Begin 23
gl.BindTexture 25
gl.Bitmap 26
gl.BitmapFromBrush 241
gl.BlendFunc 28
gl.CallList 30
gl.CallLists 31
gl.Clear 31
gl.ClearAccum 32
gl.ClearColor 33
gl.ClearDepth 34
gl.ClearIndex 34
gl.ClearStencil 35
gl.ClipPlane 35
gl.Color 36
gl.ColorMask 37
gl.ColorMaterial 38
gl.ColorPointer 39
gl.CopyPixels 40
gl.CopyTexImage 43
gl.CopyTexSubImage 44
gl.CullFace 46
gl.DeleteLists 46
gl.DeleteTextures 47
gl.DepthFunc 48
gl.DepthMask 49
gl.DepthRange 49
gl.Disable 50
gl.DisableClientState 55
gl.DrawArrays 56
gl.DrawBuffer 57
gl.DrawElements 58
gl.DrawPixels 59
gl.DrawPixelsFromBrush 241
gl.DrawPixelsRaw 60
gl.EdgeFlag 65
gl.EdgeFlagPointer 65
gl.Enable 66
gl.EnableClientState 66
gl.End 67
gl.EndList 67
gl.EvalCoord 67
gl.EvalMesh 69
gl.EvalPoint 71
gl.FeedbackBuffer 72
gl.Finish 74
gl.Flush 74
gl.Fog 75
gl.FreeFeedbackBuffer 77
gl.FreeSelectBuffer 77
gl.FrontFace 77
gl.Frustum 78
gl.GenLists 79
gl.GenTextures 80
gl.Get 80
gl.GetArray 95
gl.GetClipPlane 96
gl.GetCurrentContext 241
gl.GetError 96
gl.GetLight 98
gl.GetMap 100
gl.GetMaterial 101
gl.GetPixelMap 102
gl.GetPointer 103
gl.GetPolygonStipple 104
gl.GetSelectBuffer 104
gl.GetString 105
gl.GetTexEnv 106
gl.GetTexGen 107
gl.GetTexImage 108
gl.GetTexImageRaw 109
gl.GetTexImageToBrush 242
gl.GetTexLevelParameter 111
gl.GetTexParameter 112
gl.Hint 113
gl.Index 115
gl.IndexMask 115
gl.IndexPointer 116
gl.InitNames 117
gl.InterleavedArrays 117
gl.IsEnabled 119
gl.IsList 122
gl.IsTexture 122
gl.Light 123
gl.LightModel 125
gl.LineStipple 127
gl.LineWidth 128
gl.ListBase 129
gl.LoadIdentity 129
gl.LoadMatrix 130
gl.LoadName 131
gl.LogicOp 131
gl.Map 133
gl.MapGrid 136
gl.Material 138
gl.MatrixMode 139
gl.MultMatrix 140
gl.NewList 141
gl.Normal 142
gl.NormalPointer 143
gl.Ortho 144
gl.PassThrough 145
gl.PixelMap 146
gl.PixelStore 148

gl.PixelTransfer 152
gl.PixelZoom 154
gl.PointSize 155
gl.PolygonMode 156
gl.PolygonOffset 157
gl.PolygonStipple 158
gl.PopAttrib 159
gl.PopClientAttrib 160
gl.PopMatrix 160
gl.PopName 161
gl.PrioritizeTextures 162
gl.PushAttrib 163
gl.PushClientAttrib 168
gl.PushMatrix 168
gl.PushName 169
gl.RasterPos 170
gl.ReadBuffer 172
gl.ReadPixels 172
gl.ReadPixelsRaw 175
gl.ReadPixelsToBrush 242
gl.Rect 176
gl.RenderMode 177
gl.Rotate 178
gl.Scale 179
gl.Scissor 180
gl.SelectBuffer 181
gl.SetCurrentContext 243
gl.ShadeModel 182
gl.StencilFunc 183
gl.StencilMask 185
gl.StencilOp 185
gl.TexCoord 187
gl.TexCoordPointer 188
gl.TexEnv 189
gl.TexGen 190
gl.TexImage 191
gl.TexImage1D 192
gl.TexImage2D 196
gl.TexImageFromBrush 243
gl.TexParameter 199
gl.TexSubImage 202
gl.TexSubImage1D 203
gl.TexSubImage2D 204
gl.TexSubImageFromBrush 244
gl.Translate 206
gl.Vertex 207
gl.VertexPointer 207
gl.Viewport 208
glfw.GetJoystickAxes 239
glfw.GetJoystickButtons 239
glfw.GetJoystickName 239
glfw.JoystickPresent 240
glu.Build1DMipmaps 211
glu.Build2DMipmaps 212
glu.Build3DMipmaps 214
glu.BuildMipmaps 211
glu.BuildMipmapsFromBrush 244
glu.ErrorString 215
glu.GetString 216
glu.LookAt 216
glu.NewNurbsRenderer 217
glu.NewQuadric 218
glu.Ortho2D 218
glu.Perspective 218
glu.PickMatrix 219
glu.Project 220
glu.ScaleImage 221
glu.ScaleImageRaw 221
glu.UnProject 222

N

nurb:BeginCurve 223
nurb:BeginSurface 224
nurb:BeginTrim 224
nurb:Callback 225
nurb:Curve 227
nurb:EndCurve 227
nurb:EndSurface 228
nurb:EndTrim 228
nurb:GetProperty 228
nurb:LoadSamplingMatrices 229
nurb:PwlCurve 229
nurb:SetProperty 230
nurb:Surface 232

Q

quad:Cylinder 233
quad:Disk 234
quad:DrawStyle 234
quad:Normals 235
quad:Orientation 236
quad:PartialDisk 236
quad:Sphere 237
quad:Texture 237