BOOK

The following sample is taken from the *OpenGL Programming Guide*.

MAKEFILE (BOOK Sample)

```
# Nmake macros for building Windows 32-Bit apps for OpenGL

!include <ntwin32.mak>

opengllibs     = opengl32.lib glu32.lib glaux.lib

EXES = accanti.exe  \
          accnot.exe   \
          accpersp.exe \
          accum.exe    \
          aim.exe      \
          alpha.exe    \
          alpha3d.exe  \
          anti.exe     \
          antindx.exe  \
          antipndx.exe \
          antipnt.exe  \
          antipoly.exe \
          bezcurve.exe \
          bezmesh.exe  \
          bezsurf.exe  \
          checker.exe  \
          checker2.exe \
          chess.exe    \
          clip.exe     \
          colormat.exe \
          cone.exe     \
          cube.exe     \
          curve.exe    \
          depthcue.exe \
          disk.exe     \
          dof.exe      \
          dofnot.exe   \
          double.exe   \
          drawf.exe    \
          feedback.exe \
          fog.exe      \
          fogindex.exe \
          font.exe     \
          light.exe    \
          linelist.exe \
          lines.exe    \
          list.exe     \
          list2.exe    \
          maplight.exe \
          material.exe \
          mipmap.exe   \
          model.exe    \
          movelght.exe \
          nurbs.exe    \
          pickdpth.exe \
          pickline.exe \
```

```
           picksqr.exe  \
           plane.exe    \
           planet.exe   \
           planetup.exe \
           polys.exe    \
           robot.exe    \
           scclrlt.exe  \
           scene.exe    \
           scenebmb.exe \
           sceneflt.exe \
           select.exe   \
           simple.exe   \
           smooth.exe   \
           sphere.exe   \
           stencil.exe  \
           stroke.exe   \
           surface.exe  \
           tea.exe      \
           teaamb.exe   \
           teapots.exe  \
           texgen.exe   \
           texsurf.exe  \
           trim.exe

all: $(EXES)

.c.exe:
    $(cc) $(cflags) $(cdebug) -DSILENT_WARN $(cvars) $<
    $(link) $(linkdebug) $(guiflags) -subsystem:windows
-entry:mainCRTStartup -out:$*.exe $*.obj $(opengllibs) $(guilibs)
```

ACCANTI.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  accanti.c
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize lighting and other values.
 */
void myinit(void)
{
    GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
    GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void displayObjects(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
```

```
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

    glPushMatrix ();
    glRotatef (30.0, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glTranslatef (-0.80, 0.35, 0.0);
    glRotatef (100.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (-0.75, -0.50, 0.0);
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
    auxSolidCube (1.5);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.75, 0.60, 0.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
    auxSolidSphere (1.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.70, -0.90, 0.25);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
    auxSolidOctahedron (1.0);
    glPopMatrix ();

    glPopMatrix ();
}

#define ACSIZE  8

void CALLBACK display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glPushMatrix ();
/*  Note that 4.5 is the distance in world space between
 *  left and right and bottom and top.
 *  This formula converts fractional pixel movement to
 *  world coordinates.
 */
```

```
        glTranslatef (j8[jitter].x*4.5/viewport[2],
            j8[jitter].y*4.5/viewport[3], 0.0);
        displayObjects ();
        glPopMatrix ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
    else
    glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 250, 250);
    auxInitWindow ("Jittering - Orthographic");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ACCNOT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  accnot.c
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);


/*  Initialize lighting and other values.
 */

void myinit(void)
{
    GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
    GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glShadeModel (GL_FLAT);
    glPushMatrix ();
    glRotatef (30.0, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glTranslatef (-0.80, 0.35, 0.0);
    glRotatef (100.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (-0.75, -0.50, 0.0);
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
    auxSolidCube (1.5);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.75, 0.60, 0.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
    auxSolidSphere (1.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.70, -0.90, 0.25);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
    auxSolidOctahedron (1.0);
    glPopMatrix ();

    glPopMatrix ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
    else
        glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 250, 250);
    auxInitWindow ("Auxillary 3-D Models");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ACCPERSP.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  accpersp.c
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void displayObjects(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);


#define PI_ 3.14159265358979323846

/*  accFrustum()
 *  The first 6 arguments are identical to the glFrustum() call.
 *
 *  pixdx and pixdy are anti-alias jitter in pixels.
 *  Set both equal to 0.0 for no anti-alias jitter.
 *  eyedx and eyedy are depth-of field jitter in pixels.
 *  Set both equal to 0.0 for no depth of field effects.
 *
 *  focus is distance from eye to plane in focus.
 *  focus must be greater than, but not equal to 0.0.
 *
 *  Note that accFrustum() calls glTranslatef().  You will
 *  probably want to insure that your ModelView matrix has been
 *  initialized to identity before calling accFrustum().
 */
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top,
```

```
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble xwsize, ywsize;
    GLdouble dx, dy;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);

    xwsize = right - left;
    ywsize = top - bottom;

    dx = -(pixdx*xwsize/(GLdouble) viewport[2] + eyedx*znear/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] + eyedy*znear/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy, znear, zfar);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

/*  accPerspective()
 *
 *  The first 4 arguments are identical to the gluPerspective() call.
 *  pixdx and pixdy are anti-alias jitter in pixels.
 *  Set both equal to 0.0 for no anti-alias jitter.
 *  eyedx and eyedy are depth-of field jitter in pixels.
 *  Set both equal to 0.0 for no depth of field effects.
 *
 *  focus is distance from eye to plane in focus.
 *  focus must be greater than, but not equal to 0.0.
 *
 *  Note that accPerspective() calls accFrustum().
 */
void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2,left,right,bottom,top;

    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = znear / (cos(fov2) / sin(fov2));
    bottom = -top;

    right = top * aspect;
    left = -right;

    accFrustum (left, right, bottom, top, znear, zfar,
    pixdx, pixdy, eyedx, eyedy, focus);
}

/*  Initialize lighting and other values.
```

```
 */
void myinit(void)
{
    GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
    GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel (GL_FLAT);

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void displayObjects(void)
{
    GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
    GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
    GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
    GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };

    glPushMatrix ();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glTranslatef (-0.80, 0.35, 0.0);
    glRotatef (100.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (-0.75, -0.50, 0.0);
    glRotatef (45.0, 0.0, 0.0, 1.0);
    glRotatef (45.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
    auxSolidCube (1.5);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.75, 0.60, 0.0);
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
    auxSolidSphere (1.0);
```

```
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.70, -0.90, 0.25);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
    auxSolidOctahedron (1.0);
    glPopMatrix ();

    glPopMatrix ();
}

#define ACSIZE  8

void CALLBACK display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        accPerspective (50.0,
            (GLdouble) viewport[2]/(GLdouble) viewport[3],
            1.0, 15.0, j8[jitter].x, j8[jitter].y,
            0.0, 0.0, 1.0);
        displayObjects ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 250, 250);
    auxInitWindow ("Scene Antialiasing");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ACCUM.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  accum.c
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void loaddxdy(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLsizei width, height;

void myinit(void)
{
    GLfloat ambient[] = { 0.4, 0.4, 0.4, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 2.0, 2.0, 0.0 };
    GLfloat mat_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

```
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);

    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

#define ACSIZE 16

GLfloat dx[ACSIZE], dy[ACSIZE];

GLfloat jitter3[3][2] = {
    {0.5, 0.5},
    {1.35899e-05, 0.230369},
    {0.000189185, 0.766878},
};

GLfloat jitter11[11][2] = {
    {0.5, 0.5}, {0.406537, 0.135858},
    {0.860325, 0.968558}, {0.680141, 0.232877},
    {0.775694, 0.584871}, {0.963354, 0.309056},
    {0.593493, 0.864072}, {0.224334, 0.415055},
    {0.0366643, 0.690884}, {0.139685, 0.0313988},
    {0.319861, 0.767097},
};

GLfloat jitter16[16][2] = {
    {0.4375, 0.4375}, {0.1875, 0.5625},
    {0.9375, 1.1875}, {0.4375, 0.9375-1},
    {0.6875, 0.5625}, {0.1875, 0.0625},
    {0.6875, 0.3125}, {0.1875, 0.3125},
    {0.4375, 0.1875}, {0.9375-1, 0.4375},
    {0.6875, 0.8125}, {0.4375, 0.6875},
    {0.6875, 0.0625}, {0.9375, 0.9375},
    {1.1875, 0.8125}, {0.9375, 0.6875},
};

GLfloat jitter29[29][2] = {
    {0.5, 0.5}, {0.498126, 0.141363},
    {0.217276, 0.651732}, {0.439503, 0.954859},
    {0.734171, 0.836294}, {0.912454, 0.79952},
    {0.406153, 0.671156}, {0.0163892, 0.631994},
    {0.298064, 0.843476}, {0.312025, 0.0990405},
    {0.98135, 0.965697}, {0.841999, 0.272378},
    {0.559348, 0.32727}, {0.809331, 0.638901},
    {0.632583, 0.994471}, {0.00588314, 0.146344},
    {0.713365, 0.437896}, {0.185173, 0.246584},
    {0.901735, 0.474544}, {0.366423, 0.296698},
    {0.687032, 0.188184}, {0.313256, 0.472999},
    {0.543195, 0.800044}, {0.629329, 0.631599},
```

```
    {0.818263, 0.0439354}, {0.163978, 0.00621497},
    {0.109533, 0.812811}, {0.131325, 0.471624},
    {0.0196755, 0.331813},
};

GLfloat jitter90[90][2] = {
    {0.5, 0.5}, {0.784289, 0.417355},
    {0.608691, 0.678948}, {0.546538, 0.976002},
    {0.972245, 0.270498}, {0.765121, 0.189392},
    {0.513193, 0.743827}, {0.123709, 0.874866},
    {0.991334, 0.745136}, {0.56342, 0.0925047},
    {0.662226, 0.143317}, {0.444563, 0.928535},
    {0.248017, 0.981655}, {0.100115, 0.771923},
    {0.593937, 0.559383}, {0.392095, 0.225932},
    {0.428776, 0.812094}, {0.510615, 0.633584},
    {0.836431, 0.00343328}, {0.494037, 0.391771},
    {0.617448, 0.792324}, {0.688599, 0.48914},
    {0.530421, 0.859206}, {0.0742278, 0.665344},
    {0.979388, 0.626835}, {0.183806, 0.479216},
    {0.151222, 0.0803998}, {0.476489, 0.157863},
    {0.792675, 0.653531}, {0.0990416, 0.267284},
    {0.776667, 0.303894}, {0.312904, 0.296018},
    {0.288777, 0.691008}, {0.460097, 0.0436075},
    {0.594323, 0.440751}, {0.876296, 0.472043},
    {0.0442623, 0.0693901}, {0.355476, 0.00442787},
    {0.391763, 0.361327}, {0.406994, 0.696053},
    {0.708393, 0.724992}, {0.925807, 0.933103},
    {0.850618, 0.11774}, {0.867486, 0.233677},
    {0.208805, 0.285484}, {0.572129, 0.211505},
    {0.172931, 0.180455}, {0.327574, 0.598031},
    {0.685187, 0.372379}, {0.23375, 0.878555},
    {0.960657, 0.409561}, {0.371005, 0.113866},
    {0.29471, 0.496941}, {0.748611, 0.0735321},
    {0.878643, 0.34504}, {0.210987, 0.778228},
    {0.692961, 0.606194}, {0.82152, 0.8893},
    {0.0982095, 0.563104}, {0.214514, 0.581197},
    {0.734262, 0.956545}, {0.881377, 0.583548},
    {0.0560485, 0.174277}, {0.0729515, 0.458003},
    {0.719604, 0.840564}, {0.325388, 0.7883},
    {0.26136, 0.0848927}, {0.393754, 0.467505},
    {0.425361, 0.577672}, {0.648594, 0.0248658},
    {0.983843, 0.521048}, {0.272936, 0.395127},
    {0.177695, 0.675733}, {0.89175, 0.700901},
    {0.632301, 0.908259}, {0.782859, 0.53611},
    {0.0141421, 0.855548}, {0.0437116, 0.351866},
    {0.939604, 0.0450863}, {0.0320883, 0.962943},
    {0.341155, 0.895317}, {0.952087, 0.158387},
    {0.908415, 0.820054}, {0.481435, 0.281195},
    {0.675525, 0.25699}, {0.585273, 0.324454},
    {0.156488, 0.376783}, {0.140434, 0.977416},
    {0.808155, 0.77305}, {0.282973, 0.188937},
};

void loaddxdy(void)
{
```

```
    long i;
    for (i = 0; i < ACSIZE; i++) {
        dx[i] = jitter16[i][0]*10/width;
        dy[i] = jitter16[i][1]*10/height;
    }
}

void CALLBACK display(void)
{
    int i;

    glClear(GL_ACCUM_BUFFER_BIT);
    loaddxdy();
    for (i = 0; i < (ACSIZE); i++) {
        //printf("Pass %d\n", i);
        glPushMatrix();
        glTranslatef(dx[i], dy[i], 0.0);
        glRotatef(45.0, 1.0, 1.0, 1.0);
        glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
        auxSolidTeapot(1.0);
        glPopMatrix();
        glAccum(GL_ACCUM, 1.0/(ACSIZE));
    }
    //printf("final job\n");
    glAccum(GL_RETURN, 1.0);
    //printf("done\n");
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    width = w; height = h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-3.0, 3.0, -3.0*(GLfloat) h/(GLfloat) w,
                3.0*(GLfloat) h/(GLfloat) w, -15.0, 15.0);
    else
        glOrtho(-3.0*(GLfloat) w/(GLfloat) h,
                3.0*(GLfloat) w/(GLfloat) h, -3.0, 3.0, -15.0, 15.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 300, 300);
    auxInitWindow ("Accumulation Buffer");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
```

```
    return(0);
}
```

AIM.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  aim.c
 *  This program calculates the fovy (field of view angle
 *  in the y direction), by using trigonometry, given the
 *  size of an object and its size.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
GLdouble calculateAngle (double size, double distance);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

void myinit (void) {
    glShadeModel (GL_FLAT);
}

/*  Clear the screen.  Set the current color to white.
 *  Draw the wire frame cube and sphere.
 */
void CALLBACK display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);

    glLoadIdentity ();
/*  glTranslatef() as viewing transformation    */
    glTranslatef (0.0, 0.0, -5.0);
    auxWireCube(2.0);
    auxWireSphere(1.0);
    glFlush();
}

#define PI  3.1415926535

/*  atan2 () is a system math routine which calculates
```

```
 *  the arctangent of an angle, given length of the
 *  opposite and adjacent sides of a right triangle.
 *  atan2 () is not an OpenGL routine.
 */
GLdouble calculateAngle (double size, double distance)
{
    GLdouble radtheta, degtheta;

    radtheta = 2.0 * atan2 (size/2.0, distance);
    degtheta = (180.0 * radtheta) / PI;
    //printf ("degtheta is %lf\n", degtheta);
    return ((GLdouble) degtheta);
}

/*  Called when the window is first opened and whenever
 *  the window is reconfigured (moved or resized).
 */
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    GLdouble theta;

    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    theta = calculateAngle (2.0, 5.0);
    gluPerspective(theta, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Field of View Angle");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ALPHA.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  alpha.c
 *  This program draws several overlapping filled polygons
 *  to demonstrate the effect order has on alpha blending results.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize alpha blending function.
 */
void myinit(void)
{
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glShadeModel (GL_FLAT);
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glColor4f (1.0, 1.0, 0.0, 0.75);
    glRectf (0.0, 0.0, 0.5, 1.0);

    glColor4f (0.0, 1.0, 1.0, 0.75);
    glRectf (0.0, 0.0, 1.0, 0.5);
/*  draw colored polygons in reverse order in upper right  */
    glColor4f (0.0, 1.0, 1.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);

    glColor4f (1.0, 1.0, 0.0, 0.75);
    glRectf (0.5, 0.5, 1.0, 1.0);

    glFlush();
```

```
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    gluOrtho2D (0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
    else
    gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Alpha Blending");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ALPHA3D.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  alpha3D.c
 *  This program demonstrates how to intermix opaque and
 *  alpha blended polygons in the same scene, by using glDepthMask.
 *  Pressing the left mouse button toggles the eye position.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK toggleEye (AUX_EVENTREC *event);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
    GLfloat mat_ambient[] = { 0.0, 0.0, 0.0, 0.15 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
    GLfloat mat_shininess[] = { 15.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

GLboolean eyePosition = GL_FALSE;

void CALLBACK toggleEye (AUX_EVENTREC *event)
{
    if (eyePosition)
    eyePosition = GL_FALSE;
    else
    eyePosition = GL_TRUE;
}
```

```
void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.0, 1.0 };
    GLfloat mat_torus[] = { 0.75, 0.75, 0.0, 1.0 };
    GLfloat mat_cylinder[] = { 0.0, 0.75, 0.75, 0.15 };

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLightfv (GL_LIGHT0, GL_POSITION, position);
    glPushMatrix ();
    if (eyePosition)
        gluLookAt (0.0, 0.0, 9.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    else
        gluLookAt (0.0, 0.0, -9.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, 1.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_torus);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();

    glEnable (GL_BLEND);
    glDepthMask (GL_FALSE);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cylinder);
    glTranslatef (0.0, 0.0, -1.0);
    auxSolidCylinder (1.0, 2.0);
    glDepthMask (GL_TRUE);
    glDisable (GL_BLEND);
    glPopMatrix ();

    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Intermixing Opaque and Alpha Blending");
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleEye);
    myinit();
```

```
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ANTI.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  anti.c
 *  This program draws antialiased lines in RGBA mode.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize antialiasing for RGBA mode, including alpha
 *  blending, hint, and line width.  Print out implementation
 *  specific info on line width granularity and width.
 */
void myinit(void)
{
    GLfloat values[2];
    glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
    //printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n", values[0]);

    glGetFloatv (GL_LINE_WIDTH_RANGE, values);

    //printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
    //values[0], values[1]);

    glEnable (GL_LINE_SMOOTH);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth (1.5);

    glShadeModel(GL_FLAT);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}
```

```
/*  display() draws an icosahedron with a large alpha value, 1.0.
 */
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor4f (1.0, 1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0);  /*  move object into view   */
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("Antialiased Lines Using RGBA");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ANTINDX.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  antiindex.c
 *  The program draws a wireframe icosahedron with
 *  antialiased lines, in color index mode.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

#define RAMPSIZE 16
#define RAMPSTART 32

/*  Initialize antialiasing for color index mode,
 *  including loading a grey color ramp starting
 *  at RAMPSTART, which must be a multiple of 16.
 */
void myinit(void)
{
    int i;

    for (i = 0; i < RAMPSIZE; i++) {
    GLfloat shade;
    shade = (GLfloat) i/(GLfloat) RAMPSIZE;
    auxSetOneColor(RAMPSTART+(GLint)i, shade, shade, shade);
    }

    glEnable (GL_LINE_SMOOTH);
    glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
    glLineWidth (1.5);

    glClearIndex ((GLfloat) RAMPSTART);
    glShadeModel(GL_FLAT);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}
```

```
/*  display() draws an icosahedron.
 */
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glIndexi(RAMPSTART);
    auxWireIcosahedron(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0);  /*  move object into view   */
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  color index display mode, depth buffer,
 *  and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("Antialiased Lines Using Color Index");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ANTIPNDX.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  antipindex.c
 *  The program draws antialiased points,
 *  in color index mode.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

#define RAMPSIZE 16
#define RAMPSTART 32

/*  Initialize point antialiasing for color index mode,
 *  including loading a grey color ramp starting at
 *  RAMPSTART, which must be a multiple of 16.
 */
void myinit(void)
{
    int i;

    for (i = 0; i < RAMPSIZE; i++) {
    GLfloat shade;
    shade = (GLfloat) i/(GLfloat) RAMPSIZE;
    auxSetOneColor (RAMPSTART+(GLint)i, shade, shade, shade);
    }
    glEnable (GL_POINT_SMOOTH);
    glHint (GL_POINT_SMOOTH_HINT, GL_FASTEST);
    glPointSize (3.0);
    glClearIndex ((GLfloat) RAMPSTART);
}

/*  display() draws several points.  */

void CALLBACK display(void)
{
    int i;
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glIndexi (RAMPSTART);
    glBegin (GL_POINTS);
    for (i = 1; i < 10; i++) {
        glVertex2f ((GLfloat) i * 10.0, (GLfloat) i * 10.0);
    }
    glEnd ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w < h)
    glOrtho (0.0, 100.0, 0.0,
        100.0*(GLfloat) h/(GLfloat) w, -1.0, 1.0);
    else
    glOrtho (0.0, 100.0*(GLfloat) w/(GLfloat) h,
        0.0, 100.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  color index display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow ("Points");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ANTIPNT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  antipoint.c
 *  The program draws antialiased points, in RGBA mode.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize point anti-aliasing for RGBA mode, including alpha
 *  blending, hint, and point size.  These points are 3.0 pixels big.
 */
void myinit(void)
{
    glEnable (GL_POINT_SMOOTH);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glHint (GL_POINT_SMOOTH_HINT, GL_DONT_CARE);
    glPointSize (3.0);

    glClearColor(0.0, 0.0, 0.0, 0.0);
}

/*  display() draws several points.
 */
void CALLBACK display(void)
{
    int i;

    glClear (GL_COLOR_BUFFER_BIT);
    glColor4f (1.0, 1.0, 1.0, 1.0);
    glBegin (GL_POINTS);
    for (i = 1; i < 10; i++) {
        glVertex2f ((GLfloat) i * 10.0, (GLfloat) i * 10.0);
    }
    glEnd ();
    glFlush();
```

```
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w < h)
    glOrtho (0.0, 100.0,
        0.0, 100.0*(GLfloat) h/(GLfloat) w, -1.0, 1.0);
    else
    glOrtho (0.0, 100.0*(GLfloat) w/(GLfloat) h,
        0.0, 100.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow ("Points");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

ANTIPOLY.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  antipoly.c
 *  This program draws filled polygons with antialiased
 *  edges.  The special GL_SRC_ALPHA_SATURATE blending
 *  function is used.
 *  Pressing the left mouse button turns the antialiasing
 *  on and off.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK toggleSmooth (AUX_EVENTREC *event);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLboolean polySmooth;

void myinit(void)
{
    GLfloat mat_ambient[] = { 0.0, 0.0, 0.0, 1.00 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.00 };
    GLfloat mat_shininess[] = { 15.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
    glEnable (GL_BLEND);
    glCullFace (GL_BACK);
    glEnable (GL_CULL_FACE);
    glEnable (GL_POLYGON_SMOOTH);
    polySmooth = GL_TRUE;

    glClearColor (0.0, 0.0, 0.0, 0.0);
}
```

```
void CALLBACK toggleSmooth (AUX_EVENTREC *event)
{
    if (polySmooth) {
    polySmooth = GL_FALSE;
    glDisable (GL_BLEND);
    glDisable (GL_POLYGON_SMOOTH);
    glEnable (GL_DEPTH_TEST);
    }
    else {
    polySmooth = GL_TRUE;
    glEnable (GL_BLEND);
    glEnable (GL_POLYGON_SMOOTH);
    glDisable (GL_DEPTH_TEST);
    }
}

/*  Note:  polygons must be drawn from back to front
 *  for proper blending.
 */
void CALLBACK display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.0, 0.0 };
    GLfloat mat_cube1[] = { 0.75, 0.75, 0.0, 1.0 };
    GLfloat mat_cube2[] = { 0.0, 0.75, 0.75, 1.0 };

    if (polySmooth)
    glClear (GL_COLOR_BUFFER_BIT);
    else
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
    glTranslatef (0.0, 0.0, -8.0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);

    glBlendFunc (GL_SRC_ALPHA_SATURATE, GL_ONE);

    glPushMatrix ();
    glRotatef (30.0, 1.0, 0.0, 0.0);
    glRotatef (60.0, 0.0, 1.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube1);
    auxSolidCube (1.0);
    glPopMatrix ();

    glTranslatef (0.0, 0.0, -2.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_cube2);
    glRotatef (30.0, 0.0, 1.0, 0.0);
    glRotatef (60.0, 1.0, 0.0, 0.0);
    auxSolidCube (1.0);

    glPopMatrix ();

    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
```

```
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_ALPHA | AUX_DEPTH16);
    auxInitPosition (0, 0, 200, 200);
    auxInitWindow ("Antialiased Edges");
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, toggleSmooth);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

BEZCURVE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  bezcurve.c
 *  This program uses evaluators to draw a Bezier curve.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat ctrlpoints[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void myinit(void)
{
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINE_STRIP);
    for (i = 0; i <= 30; i++)
        glEvalCoord1f((GLfloat) i/30.0);
    glEnd();
    /* The following code displays the control points as dots. */
    glPointSize(5.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
    for (i = 0; i < 4; i++)
        glVertex3fv(&ctrlpoints[i][0]);
    glEnd();
```

```
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
        5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
    else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
        5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Bezier Curves Using Evaluators");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

BEZMESH.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  bezsurf.c
 *  This program renders a lighted, filled Bezier surface,
 *  using two-dimensional evaluators.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void initlights(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
    {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
    {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
    {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
    {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void initlights(void)
{
    GLfloat ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat position[] = { 0.0, 0.0, 2.0, 1.0 };
    GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

```
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
    glPopMatrix();
    glFlush();
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glEnable (GL_DEPTH_TEST);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    initlights();   /* for lighted version only */
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
        4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
    glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
        4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Lighted and Filled Bezier Surface");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

BEZSURF.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  bezmesh.c
 *  This program renders a wireframe Bezier surface,
 *  using two-dimensional evaluators.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void CALLBACK display(void)
{
    int i, j;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    for (j = 0; j <= 8; j++) {
     glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
     glEnd();
     glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
         glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
     glEnd();
```

```
    }
    glPopMatrix ();
    glFlush();
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
         0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
     glOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
         4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
    else
     glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
         4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Wireframe Bezier Surface");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CHECKER.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  checker.c
 *  This program texture maps a checkerboard image onto
 *  two rectangles.  This program clamps the texture, if
 *  the texture coordinates fall outside 0.0 and 1.0.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void makeCheckImage(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageWidth][checkImageHeight][3];

void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageWidth; i++) {
    for (j = 0; j < checkImageHeight; j++) {
        c = ((((i&0x8)==0)^((j&0x8))==0))*255;
        checkImage[i][j][0] = (GLubyte) c;
        checkImage[i][j][1] = (GLubyte) c;
        checkImage[i][j][2] = (GLubyte) c;
    }
    }
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    makeCheckImage();
```

```
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
    checkImageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
    &checkImage[0][0][0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
    glEnd();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Texture Map");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CHECKER2.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  checker2.c
 *  This program texture maps a checkerboard image onto
 *  two rectangles.  This program repeats the texture, if
 *  the texture coordinates fall outside 0.0 and 1.0.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void makeCheckImage(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageWidth][checkImageHeight][3];

void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageWidth; i++) {
    for (j = 0; j < checkImageHeight; j++) {
        c = ((((i&0x8)==0)^((j&0x8))==0))*255;
        checkImage[i][j][0] = (GLubyte) c;
        checkImage[i][j][1] = (GLubyte) c;
        checkImage[i][j][2] = (GLubyte) c;
    }
    }
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    makeCheckImage();
```

```
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
    checkImageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
    &checkImage[0][0][0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
    glEnd();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Texture Map");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CHESS.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*  chess.c
 *  This program texture maps a checkerboard image onto
 *  two rectangles.  The texture coordinates for the
 *  rectangles are 0.0 to 3.0.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <math.h>

void myinit(void);
void makeCheckImage(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageWidth][checkImageHeight][3];

void makecheckimage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageWidth; i++) {
    for (j = 0; j < checkImageHeight; j++) {
        c = ((((i&0x8)==0)^((j&0x8))==0))*255;
        checkImage[i][j][0] = (GLubyte) c;
        checkImage[i][j][1] = (GLubyte) c;
        checkImage[i][j][2] = (GLubyte) c;
    }
    }
}

void myinit(void)
{
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    makecheckimage();
    glTexImage2D(GL_TEXTURE_2D, 0, 3,
```

```
    checkImageWidth, checkImageHeight, 0,
    GL_RGB, GL_UNSIGNED_BYTE, &checkImage[0][0][0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
    glEnd();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Texture Map");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CLIP.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  clip.c
 *  This program demonstrates arbitrary clipping planes.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void CALLBACK display(void)
{
    GLdouble eqn[4] = {0.0, 1.0, 0.0, 0.0};
    GLdouble eqn2[4] = {1.0, 0.0, 0.0, 0.0};

    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f (1.0, 1.0, 1.0);
    glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);

/*    clip lower half -- y < 0      */
    glClipPlane (GL_CLIP_PLANE0, eqn);
    glEnable (GL_CLIP_PLANE0);
/*    clip left half -- x < 0       */
    glClipPlane (GL_CLIP_PLANE1, eqn2);
    glEnable (GL_CLIP_PLANE1);

    glRotatef (90.0, 1.0, 0.0, 0.0);
    auxWireSphere(1.0);
    glPopMatrix();
    glFlush();
}

void myinit (void) {
    glShadeModel (GL_FLAT);
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Arbitrary Clipping Planes");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

COLORMAT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  colormat.c
 *  After initialization, the program will be in
 *  ColorMaterial mode.  Pressing the mouse buttons
 *  will change the color of the diffuse reflection.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK changeRedDiffuse (void);
void CALLBACK changeGreenDiffuse (void);
void CALLBACK changeBlueDiffuse (void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };

/*  Initialize values for material property, light source,
 *  lighting model, and depth buffer.
 */
void myinit(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

    glColorMaterial(GL_FRONT, GL_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);
}
```

```
void CALLBACK changeRedDiffuse (void)
{
    diffuseMaterial[0] += 0.1;
    if (diffuseMaterial[0] > 1.0)
    diffuseMaterial[0] = 0.0;
    glColor4fv(diffuseMaterial);
}

void CALLBACK changeGreenDiffuse (void)
{
    diffuseMaterial[1] += 0.1;
    if (diffuseMaterial[1] > 1.0)
    diffuseMaterial[1] = 0.0;
    glColor4fv(diffuseMaterial);
}

void CALLBACK changeBlueDiffuse (void)
{
    diffuseMaterial[2] += 0.1;
    if (diffuseMaterial[2] > 1.0)
    diffuseMaterial[2] = 0.0;
    glColor4fv(diffuseMaterial);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
        1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
    glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
        1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("ColorMaterial Mode");
    myinit();
    auxKeyFunc(AUX_R, changeRedDiffuse);
```

```
    auxKeyFunc(AUX_r, changeRedDiffuse);
    auxKeyFunc(AUX_G, changeGreenDiffuse);
    auxKeyFunc(AUX_g, changeGreenDiffuse);
    auxKeyFunc(AUX_B, changeBlueDiffuse);
    auxKeyFunc(AUX_b, changeBlueDiffuse);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CONE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 * cone.c
 * This program demonstrates the use of the GL lighting model.
 * A sphere is drawn using a grey material characteristic.
 * A single light source illuminates the object.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize material property, light source, and lighting model.
 */
void myinit(void)
{
    GLfloat mat_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat mat_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
/*  mat_specular and mat_shininess are NOT default values   */
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };

    GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
/*  light_position is NOT default value */
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

```
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glTranslatef (0.0, -1.0, 0.0);
    glRotatef (250.0, 1.0, 0.0, 0.0);
    auxSolidCone(1.0, 2.0);
    glPopMatrix();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
                1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
                1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Lighting");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CUBE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  cube.c
 *  Draws a 3-D cube, viewed with perspective, stretched
 *  along the y-axis.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Clear the screen.  Set the current color to white.
 *  Draw the wire frame cube.
 */
void CALLBACK display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glLoadIdentity ();  /*  clear the matrix    */
    glTranslatef (0.0, 0.0, -5.0);  /*  viewing transformation  */
    glScalef (1.0, 2.0, 1.0);   /*  modeling transformation */
    auxWireCube(1.0);   /*  draw the cube   */
    glFlush();
}

void myinit (void) {
    glShadeModel (GL_FLAT);
}

/*  Called when the window is first opened and whenever
 *  the window is reconfigured (moved or resized).
 */
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glMatrixMode (GL_PROJECTION);   /*  prepare for and then  */
    glLoadIdentity ();  /*  define the projection  */
    glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);    /*  transformation  */
    glMatrixMode (GL_MODELVIEW);    /*  back to modelview matrix    */
```

```
    glViewport (0, 0, w, h);    /*  define the viewport */
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Perspective 3-D Cube");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

CURVE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  curve.c
 *  This program uses the Utility Library NURBS routines to
 *  draw a one-dimensional NURBS curve.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLUnurbsObj *theNurb;

void myinit(void)
{
    glShadeModel (GL_FLAT);
    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 10.0);
}

/*  This routine draws a B-spline curve.  Try a different
 *  knot sequence for a Bezier curve.  For example,
    GLfloat knots[8] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
 */

void CALLBACK display(void)
{
    GLfloat ctlpoints[4][3] = {{-.75, -.75, 0.0},
    {-.5, .75, 0.0}, {.5, .75, 0.0}, {.75, -.75, 0.0}};
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    gluBeginCurve(theNurb);
    gluNurbsCurve(theNurb,
        8, knots,
        3,
        &ctlpoints[0][0],
```

```
        4,
        GL_MAP1_VERTEX_3);
    gluEndCurve(theNurb);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    gluOrtho2D (-1.0, 1.0, -1.0 * (GLfloat) h/(GLfloat) w,
        1.0 * (GLfloat) h/(GLfloat) w);
    else
    gluOrtho2D (-1.0 * (GLfloat) w/(GLfloat) h,
        1.0 * (GLfloat) w/(GLfloat) h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("NURBS Curve");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

DEPTHCUE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  depthcue.c
 *  This program draws a wireframe model, which uses
 *  intensity (brightness) to give clues to distance.
 *  Fog is used to achieve this effect.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize linear fog for depth cueing.
 */
void myinit(void)
{
    GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
    glFogf (GL_FOG_START, 3.0);
    glFogf (GL_FOG_END, 5.0);
    glFogfv (GL_FOG_COLOR, fogColor);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

/*  display() draws an icosahedron.
 */
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
```

```
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0);  /*  move object into view   */
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("Distance Using Fog");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

DISK.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  disk.c
 *  This program demonstrates the use of the quadrics
 *  Utility Library routines to draw circles and arcs.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLUquadricObj * quadObj;

/*  Clear the screen.  For each triangle, set the current
 *  color and modify the modelview matrix.
 */
void CALLBACK display(void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glClear (GL_COLOR_BUFFER_BIT);

    glPushMatrix();
    gluQuadricDrawStyle (quadObj, GLU_FILL);
    glColor3f (1.0, 1.0, 1.0);
    glTranslatef (10.0, 10.0, 0.0);
    gluDisk (quadObj, 0.0, 5.0, 10, 2);
    glPopMatrix();

    glPushMatrix();
    glColor3f (1.0, 1.0, 0.0);
    glTranslatef (20.0, 20.0, 0.0);
    gluPartialDisk (quadObj, 0.0, 5.0, 10, 3, 30.0, 120.0);
    glPopMatrix();

    glPushMatrix();
    gluQuadricDrawStyle (quadObj, GLU_SILHOUETTE);
    glColor3f (0.0, 1.0, 1.0);
    glTranslatef (30.0, 30.0, 0.0);
```

```
    gluPartialDisk (quadObj, 0.0, 5.0, 10, 3, 135.0, 270.0);
    glPopMatrix();

    glPushMatrix();
    gluQuadricDrawStyle (quadObj, GLU_LINE);
    glColor3f (1.0, 0.0, 1.0);
    glTranslatef (40.0, 40.0, 0.0);
    gluDisk (quadObj, 2.0, 5.0, 10, 10);
    glPopMatrix();
    glFlush();
}

void myinit (void) {
    quadObj = gluNewQuadric ();
    glShadeModel(GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (0.0, 50.0,
        0.0, 50.0*(GLfloat)h/(GLfloat)w, -1.0, 1.0);
    else
    glOrtho (0.0, 50.0*(GLfloat)w/(GLfloat)h,
        0.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Quadrics");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

DOF.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  dof.c
 *  This program demonstrates use of the accumulation buffer to
 *  create an out-of-focus depth-of-field effect.  The teapots
 *  are drawn several times into the accumulation buffer.  The
 *  viewing volume is jittered, except at the focal point, where
 *  the viewing volume is at the same position, each time.  In
 *  this case, the gold teapot remains in focus.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include "jitter.h"

void myinit(void);
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
    GLdouble top, GLdouble znear, GLdouble zfar, GLdouble pixdx,
    GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus);
void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
    GLfloat ambr, GLfloat ambg, GLfloat ambb,
    GLfloat difr, GLfloat difg, GLfloat difb,
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

#define PI_ 3.14159265358979323846

/*  accFrustum()
 *  The first 6 arguments are identical to the glFrustum() call.
 *
 *  pixdx and pixdy are anti-alias jitter in pixels.
 *  Set both equal to 0.0 for no anti-alias jitter.
 *  eyedx and eyedy are depth-of field jitter in pixels.
 *  Set both equal to 0.0 for no depth of field effects.
 *
 *  focus is distance from eye to plane in focus.
```

```
 *  focus must be greater than, but not equal to 0.0.
 *
 *  Note that accFrustum() calls glTranslatef().  You will
 *  probably want to insure that your ModelView matrix has been
 *  initialized to identity before calling accFrustum().
 */
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
    GLdouble top, GLdouble znear, GLdouble zfar, GLdouble pixdx,
    GLdouble pixdy, GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble xwsize, ywsize;
    GLdouble dx, dy;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);

    xwsize = right - left;
    ywsize = top - bottom;

    dx = -(pixdx*xwsize/(GLdouble) viewport[2] + eyedx*znear/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] + eyedy*znear/focus);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy, znear, zfar);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}

/*  accPerspective()
 *
 *  The first 4 arguments are identical to the gluPerspective() call.
 *  pixdx and pixdy are anti-alias jitter in pixels.
 *  Set both equal to 0.0 for no anti-alias jitter.
 *  eyedx and eyedy are depth-of field jitter in pixels.
 *  Set both equal to 0.0 for no depth of field effects.
 *
 *  focus is distance from eye to plane in focus.
 *  focus must be greater than, but not equal to 0.0.
 *
 *  Note that accPerspective() calls accFrustum().
 */
void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble znear, GLdouble zfar, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
    GLdouble fov2,left,right,bottom,top;

    fov2 = ((fovy*PI_) / 180.0) / 2.0;

    top = znear / (cos(fov2) / sin(fov2));
    bottom = -top;

    right = top * aspect;
```

```
    left = -right;

    accFrustum (left, right, bottom, top, znear, zfar,
    pixdx, pixdy, eyedx, eyedy, focus);
}

void myinit(void)
{
    GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat local_view[] = { 0.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClearAccum(0.0, 0.0, 0.0, 0.0);
}

void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
    GLfloat ambr, GLfloat ambg, GLfloat ambb,
    GLfloat difr, GLfloat difg, GLfloat difb,
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
    float mat[3];

    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = ambr; mat[1] = ambg; mat[2] = ambb;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = difr; mat[1] = difg; mat[2] = difb;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = specr; mat[1] = specg; mat[2] = specb;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
```

```
    auxSolidTeapot(0.5);
    glPopMatrix();
}

/*  display() draws 5 teapots into the accumulation buffer
 *  several times; each time with a jittered perspective.
 *  The focal point is at z = 5.0, so the gold teapot will
 *  stay in focus.  The amount of jitter is adjusted by the
 *  magnitude of the accPerspective() jitter; in this example, 0.33.
 *  In this example, the teapots are drawn 8 times.  See jitter.h
 */
void CALLBACK display(void)
{
    int jitter;
    GLint viewport[4];

    glGetIntegerv (GL_VIEWPORT, viewport);
    glClear(GL_ACCUM_BUFFER_BIT);

    for (jitter = 0; jitter < 8; jitter++) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    accPerspective (45.0,
        (GLdouble) viewport[2]/(GLdouble) viewport[3],
        1.0, 15.0, 0.0, 0.0,
        0.33*j8[jitter].x, 0.33*j8[jitter].y, 5.0);
/*  ruby, gold, silver, emerald, and cyan teapots   */
    renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175, 0.01175,
        0.61424, 0.04136, 0.04136, 0.727811, 0.626959, 0.626959, 0.6);
    renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995, 0.0745,
        0.75164, 0.60648, 0.22648, 0.628281, 0.555802, 0.366065, 0.4);
    renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225, 0.19225,
        0.50754, 0.50754, 0.50754, 0.508273, 0.508273, 0.508273, 0.4);
    renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
        0.07568, 0.61424, 0.07568, 0.633, 0.727811, 0.633, 0.6);
    renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0, 0.50980392,
        0.50980392, 0.50196078, 0.50196078, 0.50196078, .25);
    glAccum (GL_ACCUM, 0.125);
    }

    glAccum (GL_RETURN, 1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, depth buffer, and handle input events.
 */
int main(int argc, char** argv)
{
```

```
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB
            | AUX_ACCUM | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("Depth-of-Field");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

DOFNOT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  dofnot.c
 *  This program demonstrates the same scene as dof.c, but
 *  without use of the accumulation buffer, so everything
 *  is in focus.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>

void myinit(void);
void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
    GLfloat ambr, GLfloat ambg, GLfloat ambb,
    GLfloat difr, GLfloat difg, GLfloat difb,
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

void myinit(void)
{
    GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };

    GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat local_view[] = { 0.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glFrontFace (GL_CW);
```

```
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    glClearColor(0.0, 0.0, 0.0, 0.0);
}

void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
    GLfloat ambr, GLfloat ambg, GLfloat ambb,
    GLfloat difr, GLfloat difg, GLfloat difb,
    GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
    float mat[3];

    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = ambr; mat[1] = ambg; mat[2] = ambb;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = difr; mat[1] = difg; mat[2] = difb;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = specr; mat[1] = specg; mat[2] = specb;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
    auxSolidTeapot(0.5);
    glPopMatrix();
}

/*  display() draws 5 teapots into the accumulation buffer
 *  several times; each time with a jittered perspective.
 *  The focal point is at z = 5.0, so the gold teapot will
 *  stay in focus.  The amount of jitter is adjusted by the
 *  magnitude of the accPerspective() jitter; in this example, 0.33.
 *  In this example, the teapots are drawn 8 times.  See jitter.h
 */
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
/*  ruby, gold, silver, emerald, and cyan teapots   */
    renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175, 0.01175,
        0.61424, 0.04136, 0.04136, 0.727811, 0.626959, 0.626959, 0.6);
    renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995, 0.0745,
        0.75164, 0.60648, 0.22648, 0.628281, 0.555802, 0.366065, 0.4);
    renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225, 0.19225,
        0.50754, 0.50754, 0.50754, 0.508273, 0.508273, 0.508273, 0.4);
    renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
        0.07568, 0.61424, 0.07568, 0.633, 0.727811, 0.633, 0.6);
    renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0, 0.50980392,
        0.50980392, 0.50196078, 0.50196078, 0.50196078, .25);

    glPopMatrix ();
    glFlush();
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h,
        1.0, 15.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, depth buffer, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow ("Depth-of-Field");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

DOUBLE.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  double.c
 *  This program demonstrates double buffering for
 *  flicker-free animation.  The left and middle mouse
 *  buttons start and stop the spinning motion of the square.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK spinDisplay (void);
void CALLBACK startIdleFunc (AUX_EVENTREC *event);
void CALLBACK stopIdleFunc (AUX_EVENTREC *event);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

static GLfloat spin = 0.0;

void CALLBACK display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);

    glPushMatrix ();
    glRotatef (spin, 0.0, 0.0, 1.0);
    glRectf (-25.0, -25.0, 25.0, 25.0);
    glPopMatrix ();

    glFlush();
    auxSwapBuffers();
}

void CALLBACK spinDisplay (void)
{
    spin = spin + 2.0;
    if (spin > 360.0)
    spin = spin - 360.0;
    display();
}
```

```
void CALLBACK startIdleFunc (AUX_EVENTREC *event)
{
    auxIdleFunc(spinDisplay);
}

void CALLBACK stopIdleFunc (AUX_EVENTREC *event)
{
    auxIdleFunc(0);
}

void myinit (void)
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f (1.0, 1.0, 1.0);
    glShadeModel (GL_FLAT);
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-50.0, 50.0, -50.0*(GLfloat)h/(GLfloat)w,
        50.0*(GLfloat)h/(GLfloat)w, -1.0, 1.0);
    else
    glOrtho (-50.0*(GLfloat)w/(GLfloat)h,
        50.0*(GLfloat)w/(GLfloat)h, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_DOUBLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Double Buffering");
    myinit ();
    auxReshapeFunc (myReshape);
    auxIdleFunc (spinDisplay);
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, startIdleFunc);
    auxMouseFunc (AUX_RIGHTBUTTON, AUX_MOUSEDOWN, stopIdleFunc);
    auxMainLoop(display);
    return(0);
}
```

DRAWF.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  drawf.c
 *  Draws the bitmapped letter F on the screen (several times).
 *  This demonstrates use of the glBitmap() call.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLubyte rasters[24] = {
    0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
    0xff, 0xc0, 0xff, 0xc0};

void myinit(void)
{
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1);
    glClearColor (0.0, 0.0, 0.0, 0.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glRasterPos2i (20.5, 20.5);
    glBitmap (10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap (10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glBitmap (10, 12, 0.0, 0.0, 12.0, 0.0, rasters);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    glOrtho (0, w, 0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Bitmap");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

FEEDBACK.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  feedback.c
 *  This program demonstrates use of OpenGL feedback.  First,
 *  a lighting environment is set up and a few lines are drawn.
 *  Then feedback mode is entered, and the same lines are
 *  drawn.  The results in the feedback buffer are printed.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void drawGeometry (GLenum mode);
void print3DcolorVertex (GLint size, GLint *count, GLfloat *buffer);
void printBuffer(GLint size, GLfloat *buffer);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize lighting.
 */
void myinit(void)
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

/*  Draw a few lines and two points, one of which will
 *  be clipped.  If in feedback mode, a passthrough token
 *  is issued between the each primitive.
 */
void drawGeometry (GLenum mode)
{
    glBegin (GL_LINE_STRIP);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (30.0, 30.0, 0.0);
    glVertex3f (50.0, 60.0, 0.0);
    glVertex3f (70.0, 40.0, 0.0);
    glEnd ();
    if (mode == GL_FEEDBACK)
```

```
    glPassThrough (1.0);
    glBegin (GL_POINTS);
    glVertex3f (-100.0, -100.0, -100.0);    /*  will be clipped  */
    glEnd ();
    if (mode == GL_FEEDBACK)
    glPassThrough (2.0);
    glBegin (GL_POINTS);
    glNormal3f (0.0, 0.0, 1.0);
    glVertex3f (50.0, 50.0, 0.0);
    glEnd ();
}

/*  Write contents of one vertex to stdout. */
void print3DcolorVertex (GLint size,
    GLint *count, GLfloat *buffer)
{
    int i;

    //printf ("  ");
    for (i = 0; i < 7; i++) {
    //printf ("%4.2f ", buffer[size-(*count)]);
    *count = *count - 1;
    }
    //printf ("\n");
}

/*  Write contents of entire buffer.  (Parse tokens!)   */
void printBuffer(GLint size, GLfloat *buffer)
{
    GLint count;
    GLfloat token;

    count = size;
    while (count) {
    token = buffer[size-count]; count--;
    if (token == GL_PASS_THROUGH_TOKEN) {
        //printf ("GL_PASS_THROUGH_TOKEN\n");
        //printf ("  %4.2f\n", buffer[size-count]);
        count--;
    }
    else if (token == GL_POINT_TOKEN) {
        //printf ("GL_POINT_TOKEN\n");
        print3DcolorVertex (size, &count, buffer);
    }
    else if (token == GL_LINE_TOKEN) {
        //printf ("GL_LINE_TOKEN\n");
        print3DcolorVertex (size, &count, buffer);
        print3DcolorVertex (size, &count, buffer);
    }
    else if (token == GL_LINE_RESET_TOKEN) {
        //printf ("GL_LINE_RESET_TOKEN\n");
        print3DcolorVertex (size, &count, buffer);
        print3DcolorVertex (size, &count, buffer);
    }
    }
```

```
}

void CALLBACK display(void)
{
    GLfloat feedBuffer[1024];
    GLint size;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawGeometry (GL_RENDER);

    glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
    (void) glRenderMode (GL_FEEDBACK);
    drawGeometry (GL_FEEDBACK);

    size = glRenderMode (GL_RENDER);
    printBuffer (size, feedBuffer);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow ("Feedback");
    myinit ();
    auxMainLoop(display);
    return(0);
}
```

FOG.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  fog.c
 *  This program draws 5 red teapots, each at a different
 *  z distance from the eye, in different types of fog.
 *  Pressing the left mouse button chooses between 3 types of
 *  fog:  exponential, exponential squared, and linear.
 *  In this program, there is a fixed density value, as well
 *  as fixed start and end values for the linear fog.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glaux.h>
#include <stdio.h>

void myinit(void);
void renderRedTeapot (GLfloat x, GLfloat y, GLfloat z);
void CALLBACK cycleFog (AUX_EVENTREC *event);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLint fogMode;

void CALLBACK cycleFog (AUX_EVENTREC *event)
{
    if (fogMode == GL_EXP) {
    fogMode = GL_EXP2;
    //printf ("Fog mode is GL_EXP2\n");
    }
    else if (fogMode == GL_EXP2) {
    fogMode = GL_LINEAR;
    //printf ("Fog mode is GL_LINEAR\n");
    glFogf (GL_FOG_START, 1.0);
    glFogf (GL_FOG_END, 5.0);
    }
    else if (fogMode == GL_LINEAR) {
    fogMode = GL_EXP;
    //printf ("Fog mode is GL_EXP\n");
    }
    glFogi (GL_FOG_MODE, fogMode);
```

```
}

/*  Initialize z-buffer, projection matrix, light source,
 *  and lighting model.  Do not specify a material property here.
 */
void myinit(void)
{
    GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
    GLfloat local_view[] = { 0.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glEnable(GL_FOG);
    {
    GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

    fogMode = GL_EXP;
    glFogi (GL_FOG_MODE, fogMode);
    glFogfv (GL_FOG_COLOR, fogColor);
    glFogf (GL_FOG_DENSITY, 0.35);
    glHint (GL_FOG_HINT, GL_DONT_CARE);
    glClearColor(0.5, 0.5, 0.5, 1.0);
    }
}

void renderRedTeapot (GLfloat x, GLfloat y, GLfloat z)
{
    float mat[3];

    glPushMatrix();
    glTranslatef (x, y, z);
    mat[0] = 0.1745; mat[1] = 0.01175; mat[2] = 0.01175;
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
    mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
    mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
    glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
    glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
    auxSolidTeapot(1.0);
    glPopMatrix();
}

/*  display() draws 5 teapots at different z positions.
 */
void CALLBACK display(void)
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    renderRedTeapot (-4.0, -0.5, -1.0);
    renderRedTeapot (-2.0, -0.5, -2.0);
    renderRedTeapot (0.0, -0.5, -3.0);
    renderRedTeapot (2.0, -0.5, -4.0);
    renderRedTeapot (4.0, -0.5, -5.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= (h*3))
    glOrtho (-6.0, 6.0, -2.0*((GLfloat) h*3)/(GLfloat) w,
        2.0*((GLfloat) h*3)/(GLfloat) w, 0.0, 10.0);
    else
    glOrtho (-6.0*(GLfloat) w/((GLfloat) h*3),
        6.0*(GLfloat) w/((GLfloat) h*3), -2.0, 2.0, 0.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, depth buffer, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 450, 150);
    auxInitWindow ("Fog");
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, cycleFog);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

FOGINDEX.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  fogindex.c
 *  This program demonstrates fog in color index mode.
 *  Three cones are drawn at different z values in a linear
 *  fog.  32 contiguous colors (from 16 to 47) are loaded
 *  with a color ramp.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize color map and fog.  Set screen clear color
 *  to end of color ramp.
 */
#define NMCOLORS 32
#define RAMPSTART 16

void myinit(void)
{
    int i;

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    for (i = 0; i < NMCOLORS; i++) {
    GLfloat shade;
    shade = (GLfloat) (NMCOLORS-i)/(GLfloat) NMCOLORS;
    auxSetOneColor (16 + i, shade, shade, shade);
    }
    glEnable(GL_FOG);

    glFogi (GL_FOG_MODE, GL_LINEAR);
    glFogi (GL_FOG_INDEX, NMCOLORS);
    glFogf (GL_FOG_START, 0.0);
    glFogf (GL_FOG_END, 4.0);
    glHint (GL_FOG_HINT, GL_NICEST);
    glClearIndex((GLfloat) (NMCOLORS+RAMPSTART-1));
```

```
}

/*  display() renders 3 cones at different z positions.
 */
void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (-1.0, -1.0, -1.0);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (0.0, -1.0, -2.25);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();

    glPushMatrix ();
    glTranslatef (1.0, -1.0, -3.5);
    glRotatef (-90.0, 1.0, 0.0, 0.0);
    glIndexi (RAMPSTART);
    auxSolidCone(1.0, 2.0);
    glPopMatrix ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w,
        2.0*(GLfloat)h/(GLfloat)w, 0.0, 10.0);
    else
    glOrtho (-2.0*(GLfloat)w/(GLfloat)h,
        2.0*(GLfloat)w/(GLfloat)h, -2.0, 2.0, 0.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, depth buffer, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH16);
    auxInitPosition (0, 0, 200, 200);
    auxInitWindow ("Fog Using Color Index");
```

```
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

FONT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  font.c
 *
 *  Draws some text in a bitmapped font.  Uses glBitmap()
 *  and other pixel routines.  Also demonstrates use of
 *  display lists.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void makeRasterFont(void);
void printString(char *s);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

GLubyte rasters[][13] = {
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36, 0x36, 0x36,
0x36},
{0x00, 0x00, 0x00, 0x66, 0x66, 0xff, 0x66, 0x66, 0xff, 0x66, 0x66, 0x00,
0x00},
{0x00, 0x00, 0x18, 0x7e, 0xff, 0x1b, 0x1f, 0x7e, 0xf8, 0xd8, 0xff, 0x7e,
0x18},
{0x00, 0x00, 0x0e, 0x1b, 0xdb, 0x6e, 0x30, 0x18, 0x0c, 0x76, 0xdb, 0xd8,
0x70},
{0x00, 0x00, 0x7f, 0xc6, 0xcf, 0xd8, 0x70, 0x70, 0xd8, 0xcc, 0xcc, 0x6c,
0x38},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x1c, 0x0c,
0x0e},
{0x00, 0x00, 0x0c, 0x18, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x18,
0x0c},
{0x00, 0x00, 0x30, 0x18, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x18,
0x30},
{0x00, 0x00, 0x00, 0x00, 0x99, 0x5a, 0x3c, 0xff, 0x3c, 0x5a, 0x99, 0x00,
0x00},
```

```
{0x00, 0x00, 0x00, 0x18, 0x18, 0x18, 0xff, 0xff, 0x18, 0x18, 0x18, 0x00,
0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x60, 0x60, 0x30, 0x30, 0x18, 0x18, 0x0c, 0x0c, 0x06, 0x06, 0x03,
0x03},
{0x00, 0x00, 0x3c, 0x66, 0xc3, 0xe3, 0xf3, 0xdb, 0xcf, 0xc7, 0xc3, 0x66,
0x3c},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x78, 0x38,
0x18},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0xe7,
0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0x07, 0x03, 0x03, 0xe7,
0x7e},
{0x00, 0x00, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0xff, 0xcc, 0x6c, 0x3c, 0x1c,
0x0c},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0,
0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc0, 0xc0, 0xc0, 0xe7,
0x7e},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x03, 0x03,
0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7, 0x7e, 0xe7, 0xc3, 0xc3, 0xe7,
0x7e},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x03, 0x7f, 0xe7, 0xc3, 0xc3, 0xe7,
0x7e},
{0x00, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x38, 0x38, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x30, 0x18, 0x1c, 0x1c, 0x00, 0x00, 0x1c, 0x1c, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x06, 0x0c, 0x18, 0x30, 0x60, 0xc0, 0x60, 0x30, 0x18, 0x0c,
0x06},
{0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0xff, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x60, 0x30, 0x18, 0x0c, 0x06, 0x03, 0x06, 0x0c, 0x18, 0x30,
0x60},
{0x00, 0x00, 0x18, 0x00, 0x00, 0x18, 0x18, 0x0c, 0x06, 0x03, 0xc3, 0xc3,
0x7e},
{0x00, 0x00, 0x3f, 0x60, 0xcf, 0xdb, 0xd3, 0xdd, 0xc3, 0x7e, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c,
0x18},
{0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7,
0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7,
0x7e},
{0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce,
0xfc},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0,
0xff},
```

```
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0,
0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7,
0x7e},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x7e},
{0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06,
0x06},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6,
0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0,
0xc0},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xdb, 0xff, 0xff, 0xe7,
0xc3},
{0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3,
0xe3},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xe7,
0x7e},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7,
0xfe},
{0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x66,
0x3c},
{0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7,
0xfe},
{0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7,
0x7e},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0xff},
{0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3,
0xc3},
{0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66,
0xc3},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66,
0xc3},
{0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x03,
0xff},
{0x00, 0x00, 0x3c, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30,
0x3c},
{0x00, 0x03, 0x03, 0x06, 0x06, 0x0c, 0x0c, 0x18, 0x18, 0x30, 0x30, 0x60,
0x60},
{0x00, 0x00, 0x3c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
0x3c},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc3, 0x66, 0x3c,
0x18},
{0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x38, 0x30,
0x70},
```

```
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0x7f, 0x03, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0, 0xc0,
0xc0},
{0x00, 0x00, 0x7e, 0xc3, 0xc0, 0xc0, 0xc0, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x03, 0x03, 0x03, 0x03,
0x03},
{0x00, 0x00, 0x7f, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x30, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30, 0x33,
0x1e},
{0x7e, 0xc3, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0x7e, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0xc0, 0xc0, 0xc0,
0xc0},
{0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x00, 0x00, 0x18,
0x00},
{0x38, 0x6c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x00, 0x00, 0x0c,
0x00},
{0x00, 0x00, 0xc6, 0xcc, 0xf8, 0xf0, 0xd8, 0xcc, 0xc6, 0xc0, 0xc0, 0xc0,
0xc0},
{0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x78},
{0x00, 0x00, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xdb, 0xfe, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xfc, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x7c, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x7c, 0x00, 0x00, 0x00,
0x00},
{0xc0, 0xc0, 0xc0, 0xfe, 0xc3, 0xc3, 0xc3, 0xc3, 0xfe, 0x00, 0x00, 0x00,
0x00},
{0x03, 0x03, 0x03, 0x7f, 0xc3, 0xc3, 0xc3, 0xc3, 0x7f, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xe0, 0xfe, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xfe, 0x03, 0x03, 0x7e, 0xc0, 0xc0, 0x7f, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x1c, 0x36, 0x30, 0x30, 0x30, 0x30, 0xfc, 0x30, 0x30, 0x30,
0x00},
{0x00, 0x00, 0x7e, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0xc6, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc3, 0xe7, 0xff, 0xdb, 0xc3, 0xc3, 0xc3, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xc3, 0x66, 0x3c, 0x18, 0x3c, 0x66, 0xc3, 0x00, 0x00, 0x00,
0x00},
{0xc0, 0x60, 0x60, 0x30, 0x18, 0x3c, 0x66, 0x66, 0xc3, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0xff, 0x60, 0x30, 0x18, 0x0c, 0x06, 0xff, 0x00, 0x00, 0x00,
0x00},
{0x00, 0x00, 0x0f, 0x18, 0x18, 0x18, 0x38, 0xf0, 0x38, 0x18, 0x18, 0x18,
0x0f},
```

```
{0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18,
0x18},
{0x00, 0x00, 0xf0, 0x18, 0x18, 0x18, 0x1c, 0x0f, 0x1c, 0x18, 0x18, 0x18,
0xf0},
{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x06, 0x8f, 0xf1, 0x60, 0x00, 0x00,
0x00}
};

GLuint fontOffset;

void makeRasterFont(void)
{
    GLuint i;
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    fontOffset = glGenLists (128);
    for (i = 32; i < 127; i++) {
    glNewList(i+fontOffset, GL_COMPILE);
    glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, rasters[i-32]);
    glEndList();
    }
}

void myinit(void)
{
    glShadeModel (GL_FLAT);
    makeRasterFont();
}

void printString(char *s)
{
    glPushAttrib (GL_LIST_BIT);
    glListBase(fontOffset);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
    glPopAttrib ();
}

/* Everything above this line could be in a library that defines a font.
 * To make it work, you've got to call makeRasterFont() before you start
 * making calls to printString().
 */
void CALLBACK display(void)
{
    GLfloat white[3] = { 1.0, 1.0, 1.0 };
    int i, j;
    char teststring[33];

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3fv(white);
    for (i = 32; i < 127; i += 32) {
    glRasterPos2i(20, 200 - 18*i/32);
    for (j = 0; j < 32; j++)
        teststring[j] = (char) (i+j);
    teststring[32] = 0;
    printString(teststring);
```

```
    }
    glRasterPos2i(20, 100);
    printString("The quick brown fox jumps");
    glRasterPos2i(20, 82);
    printString("over a lazy dog.");
    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Display Lists");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

GLOS.H (BOOK Sample)

```
// GLOS.H
//
// This is an OS specific header file

#include <windows.h>

// disable data conversion warnings

#pragma warning(disable : 4244)     // MIPS
#pragma warning(disable : 4136)     // X86
#pragma warning(disable : 4051)     // ALPHA
```

JITTER.H (BOOK Sample)

```
/*
jitter.h

This file contains jitter point arrays for 2,3,4,8,15,24 and 66 jitters.

The arrays are named j2, j3, etc. Each element in the array has the form,
for example, j8[0].x and j8[0].y

Values are floating point in the range -.5 < x < .5, -.5 < y < .5, and
have a gaussian distribution around the origin.

Use these to do model jittering for scene anti-aliasing and view volume
jittering for depth of field effects. Use in conjunction with the
accwindow() routine.
*/

typedef struct
{
     GLfloat x, y;
} jitter_point;

#define MAX_SAMPLES  66


/* 2 jitter points */
jitter_point j2[] =
{
     { 0.246490,  0.249999},
     {-0.246490, -0.249999}
};


/* 3 jitter points */
jitter_point j3[] =
{
     {-0.373411, -0.250550},
     { 0.256263,  0.368119},
     { 0.117148, -0.117570}
};


/* 4 jitter points */
jitter_point j4[] =
{
     {-0.208147,  0.353730},
     { 0.203849, -0.353780},
     {-0.292626, -0.149945},
     { 0.296924,  0.149994}
};


/* 8 jitter points */
```

```
jitter_point j8[] =
{
     {-0.334818,  0.435331},
     { 0.286438, -0.393495},
     { 0.459462,  0.141540},
     {-0.414498, -0.192829},
     {-0.183790,  0.082102},
     {-0.079263, -0.317383},
     { 0.102254,  0.299133},
     { 0.164216, -0.054399}
};


/* 15 jitter points */
jitter_point j15[] =
{
     { 0.285561,  0.188437},
     { 0.360176, -0.065688},
     {-0.111751,  0.275019},
     {-0.055918, -0.215197},
     {-0.080231, -0.470965},
     { 0.138721,  0.409168},
     { 0.384120,  0.458500},
     {-0.454968,  0.134088},
     { 0.179271, -0.331196},
     {-0.307049, -0.364927},
     { 0.105354, -0.010099},
     {-0.154180,  0.021794},
     {-0.370135, -0.116425},
     { 0.451636, -0.300013},
     {-0.370610,  0.387504}
};


/* 24 jitter points */
jitter_point j24[] =
{
     { 0.030245,  0.136384},
     { 0.018865, -0.348867},
     {-0.350114, -0.472309},
     { 0.222181,  0.149524},
     {-0.393670, -0.266873},
     { 0.404568,  0.230436},
     { 0.098381,  0.465337},
     { 0.462671,  0.442116},
     { 0.400373, -0.212720},
     {-0.409988,  0.263345},
     {-0.115878, -0.001981},
     { 0.348425, -0.009237},
     {-0.464016,  0.066467},
     {-0.138674, -0.468006},
     { 0.144932, -0.022780},
     {-0.250195,  0.150161},
     {-0.181400, -0.264219},
     { 0.196097, -0.234139},
```

```
     {-0.311082, -0.078815},
     { 0.268379,  0.366778},
     {-0.040601,  0.327109},
     {-0.234392,  0.354659},
     {-0.003102, -0.154402},
     { 0.297997, -0.417965}
};


/* 66 jitter points */
jitter_point j66[] =
{
     { 0.266377, -0.218171},
     {-0.170919, -0.429368},
     { 0.047356, -0.387135},
     {-0.430063,  0.363413},
     {-0.221638, -0.313768},
     { 0.124758, -0.197109},
     {-0.400021,  0.482195},
     { 0.247882,  0.152010},
     {-0.286709, -0.470214},
     {-0.426790,  0.004977},
     {-0.361249, -0.104549},
     {-0.040643,  0.123453},
     {-0.189296,  0.438963},
     {-0.453521, -0.299889},
     { 0.408216, -0.457699},
     { 0.328973, -0.101914},
     {-0.055540, -0.477952},
     { 0.194421,  0.453510},
     { 0.404051,  0.224974},
     { 0.310136,  0.419700},
     {-0.021743,  0.403898},
     {-0.466210,  0.248839},
     { 0.341369,  0.081490},
     { 0.124156, -0.016859},
     {-0.461321, -0.176661},
     { 0.013210,  0.234401},
     { 0.174258, -0.311854},
     { 0.294061,  0.263364},
     {-0.114836,  0.328189},
     { 0.041206, -0.106205},
     { 0.079227,  0.345021},
     {-0.109319, -0.242380},
     { 0.425005, -0.332397},
     { 0.009146,  0.015098},
     {-0.339084, -0.355707},
     {-0.224596, -0.189548},
     { 0.083475,  0.117028},
     { 0.295962, -0.334699},
     { 0.452998,  0.025397},
     { 0.206511, -0.104668},
     { 0.447544, -0.096004},
     {-0.108006, -0.002471},
     {-0.380810,  0.130036},
```

```
    {-0.242440,  0.186934},
    {-0.200363,  0.070863},
    {-0.344844, -0.230814},
    { 0.408660,  0.345826},
    {-0.233016,  0.305203},
    { 0.158475, -0.430762},
    { 0.486972,  0.139163},
    {-0.301610,  0.009319},
    { 0.282245, -0.458671},
    { 0.482046,  0.443890},
    {-0.121527,  0.210223},
    {-0.477606, -0.424878},
    {-0.083941, -0.121440},
    {-0.345773,  0.253779},
    { 0.234646,  0.034549},
    { 0.394102, -0.210901},
    {-0.312571,  0.397656},
    { 0.200906,  0.333293},
    { 0.018703, -0.261792},
    {-0.209349, -0.065383},
    { 0.076248,  0.478538},
    {-0.073036, -0.355064},
    { 0.145087,  0.221726}
};
```

LIGHT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  light.c
 *  This program demonstrates the use of the OpenGL lighting
 *  model.  A sphere is drawn using a grey material characteristic.
 *  A single light source illuminates the object.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);
void CALLBACK display(void);

/*  Initialize material property, light source, lighting model,
 *  and depth buffer.
 */
void myinit(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}
```

```
void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
        1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
    glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
        1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Lighting");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

LINELIST.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  linelist.c
 *  This program demonstrates using display lists to call
 *  different line stipples.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

GLuint offset;

void myinit (void)
{
/*  background to be cleared to black   */
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);

    offset = glGenLists (3);
    glNewList (offset, GL_COMPILE);
    glDisable (GL_LINE_STIPPLE);
    glEndList ();
    glNewList (offset+1, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0F0F);
    glEndList ();
    glNewList (offset+2, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x1111);
    glEndList ();
}

void CALLBACK display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
```

```
/*  draw all lines in white */
    glColor3f (1.0, 1.0, 1.0);

    glCallList (offset);
    drawOneLine (50.0, 125.0, 350.0, 125.0);
    glCallList (offset+1);
    drawOneLine (50.0, 100.0, 350.0, 100.0);
    glCallList (offset+2);
    drawOneLine (50.0, 75.0, 350.0, 75.0);
    glCallList (offset+1);
    drawOneLine (50.0, 50.0, 350.0, 50.0);
    glCallList (offset);
    drawOneLine (50.0, 25.0, 350.0, 25.0);
    glFlush ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 400, 150);
    auxInitWindow ("Display Lists");
    myinit ();
    auxMainLoop(display);
    return(0);
}
```

LINES.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  lines.c
 *  This program demonstrates different line stipples and widths.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);

#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();

void myinit (void) {
    /*  background to be cleared to black   */
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void CALLBACK display(void)
{
    int i;

    glClear (GL_COLOR_BUFFER_BIT);
/*  draw all lines in white */
    glColor3f (1.0, 1.0, 1.0);

/*  in 1st row, 3 lines drawn, each with a different stipple    */
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0101);  /*  dotted  */
    drawOneLine (50.0, 125.0, 150.0, 125.0);
    glLineStipple (1, 0x00FF);  /*  dashed  */
    drawOneLine (150.0, 125.0, 250.0, 125.0);
    glLineStipple (1, 0x1C47);  /*  dash/dot/dash   */
    drawOneLine (250.0, 125.0, 350.0, 125.0);

/*  in 2nd row, 3 wide lines drawn, each with different stipple */
    glLineWidth (5.0);
    glLineStipple (1, 0x0101);
```

```
    drawOneLine (50.0, 100.0, 150.0, 100.0);
    glLineStipple (1, 0x00FF);
    drawOneLine (150.0, 100.0, 250.0, 100.0);
    glLineStipple (1, 0x1C47);
    drawOneLine (250.0, 100.0, 350.0, 100.0);
    glLineWidth (1.0);

/*  in 3rd row, 6 lines drawn, with dash/dot/dash stipple,  */
/*  as part of a single connect line strip          */
    glLineStipple (1, 0x1C47);
    glBegin (GL_LINE_STRIP);
    for (i = 0; i < 7; i++)
    glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
    glEnd ();

/*  in 4th row, 6 independent lines drawn,  */
/*  with dash/dot/dash stipple          */
    for (i = 0; i < 6; i++) {
    drawOneLine (50.0 + ((GLfloat) i * 50.0),
        50.0, 50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
    }

/*  in 5th row, 1 line drawn, with dash/dot/dash stipple    */
/*  and repeat factor of 5          */
    glLineStipple (5, 0x1C47);
    drawOneLine (50.0, 25.0, 350.0, 25.0);
    glFlush ();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 400, 150);
    auxInitWindow ("Lines");
    myinit ();
    auxMainLoop(display);
    return(0);
}
```

LIST.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  list.c
 *  This program demonstrates how to make and execute a
 *  display list.  Note that attributes, such as current
 *  color and matrix, are changed.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void drawLine (void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLuint listName = 1;

void myinit (void)
{
    glNewList (listName, GL_COMPILE);
    glColor3f (1.0, 0.0, 0.0);
    glBegin (GL_TRIANGLES);
    glVertex2f (0.0, 0.0);
    glVertex2f (1.0, 0.0);
    glVertex2f (0.0, 1.0);
    glEnd ();
    glTranslatef (1.5, 0.0, 0.0);
    glEndList ();
    glShadeModel (GL_FLAT);
}

void drawLine (void)
{
    glBegin (GL_LINES);
    glVertex2f (0.0, 0.5);
    glVertex2f (15.0, 0.5);
    glEnd ();
}

void CALLBACK display(void)
```

```
{
    GLuint i;

    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (0.0, 1.0, 0.0);
    glPushMatrix();
    for (i = 0; i < 10; i++)
        glCallList (listName);
    drawLine ();
    glPopMatrix();
    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,
                1.5 * (GLfloat) h/(GLfloat) w);
    else
        gluOrtho2D (0.0, 2.0 * (GLfloat) w/(GLfloat) h, -0.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 400, 50);
    auxInitWindow ("Display List");
    myinit ();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

LIST2.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  list2.c
 *  This program demonstrates glGenList() and glPushAttrib().
 *  The matrix and color are restored, before the line is drawn.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void drawLine (void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

GLuint listName;

void myinit (void)
{
    GLfloat color_vector[3] = {1.0, 0.0, 0.0};

    listName = glGenLists (1);
    glNewList (listName, GL_COMPILE);
    glPushAttrib (GL_CURRENT_BIT);
    glColor3fv (color_vector);
    glBegin (GL_TRIANGLES);
    glVertex2f (0.0, 0.0);
    glVertex2f (1.0, 0.0);
    glVertex2f (0.0, 1.0);
    glEnd ();
    glTranslatef (1.5, 0.0, 0.0);
    glPopAttrib ();
    glEndList ();
    glShadeModel (GL_FLAT);
}

void drawLine (void)
{
    glBegin (GL_LINES);
    glVertex2f (0.0, 0.5);
    glVertex2f (15.0, 0.5);
```

```
    glEnd ();
}

void CALLBACK display(void)
{
    GLuint i;
    GLfloat new_color[3] = {0.0, 1.0, 0.0};

    glClear (GL_COLOR_BUFFER_BIT);
    glColor3fv (new_color);
    glPushMatrix ();
    for (i = 0; i < 10; i++)
    glCallList (listName);
    glPopMatrix ();
    drawLine ();
    glFlush ();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,
        1.5 * (GLfloat) h/(GLfloat) w);
    else
    gluOrtho2D (0.0, 2.0 * (GLfloat) w/(GLfloat) h, -0.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB);
    auxInitPosition (0, 0, 400, 50);
    auxInitWindow ("Display List");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

MAPLIGHT.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 *  maplight.c
 *  This program demonstrates the use of the GL lighting model.
 *  A sphere is drawn using a magenta diffuse reflective and
 *  white specular material property.
 *  A single light source illuminates the object.  This program
 *  illustrates lighting in color map mode.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

/*  Initialize material property, light source, and lighting model.
 */
void myinit(void)
{
    GLint i;

    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat mat_colormap[] = { 16.0, 48.0, 79.0 };
    GLfloat mat_shininess[] = { 10.0 };

    glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

    for (i = 0; i < 32; i++) {
    auxSetOneColor (16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
    auxSetOneColor (48 + i, 1.0, 1.0 * (i/32.0), 1.0);
    }
```

```
    glClearIndex(0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    auxSolidSphere(1.0);
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
        1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
    glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
        1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/*  Main Loop
 *  Open window with initial window size, title bar, color
 *  index display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_INDEX | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow ("Lighting in Color Map Mode");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

MATERIAL.C (BOOK Sample)

```
/*
 * (c) Copyright 1993, Silicon Graphics, Inc.
 *               1993, 1994 Microsoft Corporation
 *
 * ALL RIGHTS RESERVED
 *
 * Please refer to OpenGL/readme.txt for additional information
 *
 */

/*
 * material.c
 * This program demonstrates the use of the GL lighting model.
 * Several objects are drawn using different material characteristics.
 * A single light source illuminates the objects.
 */
#include "glos.h"

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

/*  Initialize z-buffer, projection matrix, light source,
 *  and lighting model.  Do not specify a material property here.
 */
void myinit(void)
{
    GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat position[] = { 0.0, 3.0, 2.0, 0.0 };
    GLfloat lmodel_ambient[] = { 0.4, 0.4, 0.4, 1.0 };
    GLfloat local_view[] = { 0.0 };

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glClearColor(0.0, 0.1, 0.1, 0.0);
}
```

```
/*  Draw twelve spheres in 3 rows with 4 columns.
 *  The spheres in the first row have materials with no ambient reflection.
 *  The second row has materials with significant ambient reflection.
 *  The third row has materials with colored ambient reflection.
 *
 *  The first column has materials with blue, diffuse reflection only.
 *  The second column has blue diffuse reflection, as well as specular
 *  reflection with a low shininess exponent.
 *  The third column has blue diffuse reflection, as well as specular
 *  reflection with a high shininess exponent (a more concentrated
highlight).
 *  The fourth column has materials which also include an emissive
component.
 *
 *  glTranslatef() is used to move spheres to their appropriate locations.
 */

void CALLBACK display(void)
{
    GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
    GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
    GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat no_shininess[] = { 0.0 };
    GLfloat low_shininess[] = { 5.0 };
    GLfloat high_shininess[] = { 100.0 };
    GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*  draw sphere in first row, first column
 *  diffuse reflection only; no ambient or specular
 */
    glPushMatrix();
    glTranslatef (-3.75, 3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in first row, second column
 *  diffuse and specular reflection; low shininess; no ambient
 */
    glPushMatrix();
    glTranslatef (-1.25, 3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
```

```
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in first row, third column
 *  diffuse and specular reflection; high shininess; no ambient
 */
    glPushMatrix();
    glTranslatef (1.25, 3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in first row, fourth column
 *  diffuse reflection; emission; no ambient or specular reflection
 */
    glPushMatrix();
    glTranslatef (3.75, 3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in second row, first column
 *  ambient and diffuse reflection; no specular
 */
    glPushMatrix();
    glTranslatef (-3.75, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in second row, second column
 *  ambient, diffuse and specular reflection; low shininess
 */
    glPushMatrix();
    glTranslatef (-1.25, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();
```

```
/*  draw sphere in second row, third column
 *  ambient, diffuse and specular reflection; high shininess
 */
    glPushMatrix();
    glTranslatef (1.25, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in second row, fourth column
 *  ambient and diffuse reflection; emission; no specular
 */
    glPushMatrix();
    glTranslatef (3.75, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row, first column
 *  colored ambient and diffuse reflection; no specular
 */
    glPushMatrix();
    glTranslatef (-3.75, -3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row, second column
 *  colored ambient, diffuse and specular reflection; low shininess
 */
    glPushMatrix();
    glTranslatef (-1.25, -3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row, third column
 *  colored ambient, diffuse and specular reflection; high shininess
 */
```

```
    glPushMatrix();
    glTranslatef (1.25, -3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
    auxSolidSphere(1.0);
    glPopMatrix();

/*  draw sphere in third row, fourth column
 *  colored ambient and diffuse reflection; emission; no specular
 */
    glPushMatrix();
    glTranslatef (3.75, -3.0, 0.0);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient_color);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
    glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    auxSolidSphere(1.0);
    glPopMatrix();

    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    h = (h == 0) ? 1 : h;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= (h * 2))
    glOrtho (-6.0, 6.0, -3.0*((GLfloat)h*2)/(GLfloat)w,
        3.0*((GLfloat)h*2)/(GLfloat)w, -10.0, 10.0);
    else
    glOrtho (-6.0*(GLfloat)w/((GLfloat)h*2),
        6.0*(GLfloat)w/((GLfloat)h*2), -3.0, 3.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 600, 450);
    auxInitWindow ("Lighting");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
    return(0);
}
```

PIPES

The following sample is for the Pipes screensaver included with Windows® NT™.

MAKEFILE (PIPES Sample)

```
!include <ntwin32.mak>

all: sspipes.scr

.c.obj:
    $(cc) $(cdebug) $(cflags) $(cvars) $(scall) $*.c

sspipes.res: sspipes.rc sspipes.h
    $(rc) -r sspipes.rc

sspipes.scr: sspipes.res \
                material.obj    \
                objects.obj     \
                pipes.obj       \
                texture.obj     \
                util.obj        \
                sspipes.obj
  $(link) $(linkdebug) $(lflags) -subsystem:windows -entry:mainCRTStartup
-machine:$(CPU) -out:$*.scr $** $(guilibs) opengl32.lib glu32.lib glaux.lib
scrnsave.lib comdlg32.lib
```

MATERIAL.H (PIPES Sample)

```
/*****************************Module*Header*******************************\
* Module Name: material.h
*
* Externals from material.c
*
* Copyright (c) 1994 Microsoft Corporation
*
\*************************************************************************/

// 24 materials (from teapots)

enum {
    EMERALD = 0,
    JADE,
    OBSIDIAN,
    PEARL,
    RUBY,
    TURQUOISE,
    BRASS,
    BRONZE,
    CHROME,
    COPPER,
    GOLD,
    SILVER,
    BLACK_PLASTIC,
    CYAN_PLASTIC,
    GREEN_PLASTIC,
    RED_PLASTIC,
    WHITE_PLASTIC,
    YELLOW_PLASTIC,
    BLACK_RUBBER,
    CYAN_RUBBER,
    GREEN_RUBBER,
    RED_RUBBER,
    WHITE_RUBBER,
    YELLOW_RUBBER
};

// other materials

enum {
    BRIGHT_WHITE = 0,
    LESS_BRIGHT_WHITE,
    WARM_WHITE,
    COOL_WHITE
};

extern void ChooseMaterial();
extern void SpecifyMaterial( unsigned int mIndex );
```

MATERIAL.C (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: material.c
*
* Material selection functions.
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#include <windows.h>
#include <GL/gl.h>

#include "pipes.h"
#include "objects.h"
#include "util.h"
#include "material.h"


#define NUM_MATERIALS 24

#define NUM_GOOD_MATS 16  // good ones among the 24

#define NUM_TEX_MATS 4  // materials for texture

int goodMaterials[NUM_GOOD_MATS] = {
        EMERALD, JADE, PEARL, RUBY, TURQUOISE, BRASS, BRONZE,
        COPPER, GOLD, SILVER, CYAN_PLASTIC, WHITE_PLASTIC, YELLOW_PLASTIC,
        CYAN_RUBBER, GREEN_RUBBER, WHITE_RUBBER };

/*  materials:  emerald, jade, obsidian, pearl, ruby, turquoise
 *              brass, bronze, chrome, copper, gold, silver
 *              black, cyan, green, red, white, yellow plastic
 *              black, cyan, green, red, white, yellow rubber

    description: ambient(RGB), diffuse(RGB), specular(RGB), shininess
 *
 */
// 'tea' materials, from aux teapots program
static GLfloat teaMaterial[NUM_MATERIALS][10] = {
     0.0215f, 0.1745f, 0.0215f,
        0.07568f, 0.61424f, 0.07568f, 0.633f, 0.727811f, 0.633f, 0.6f,
     0.135f, 0.2225f, 0.1575f,
        0.54f, 0.89f, 0.63f, 0.316228f, 0.316228f, 0.316228f, 0.1f,
     0.05375f, 0.05f, 0.06625f, // XX
        0.18275f, 0.17f, 0.22525f, 0.332741f, 0.328634f, 0.346435f, 0.3f,
     0.25f, 0.20725f, 0.20725f,
```

```
        1.0f, 0.829f, 0.829f, 0.296648f, 0.296648f, 0.296648f, 0.088f,
     0.1745f, 0.01175f, 0.01175f,
        0.61424f, 0.04136f, 0.04136f, 0.727811f, 0.626959f, 0.626959f, 0.6f,
     0.1f, 0.18725f, 0.1745f,
        0.396f, 0.74151f, 0.69102f, 0.297254f, 0.30829f, 0.306678f, 0.1f,
     0.329412f, 0.223529f, 0.027451f,
        0.780392f, 0.568627f, 0.113725f, 0.992157f, 0.941176f, 0.807843f,
        0.21794872f,
     0.2125f, 0.1275f, 0.054f,
        0.714f, 0.4284f, 0.18144f, 0.393548f, 0.271906f, 0.166721f, 0.2f,
     0.25f, 0.25f, 0.25f,  // XX
        0.4f, 0.4f, 0.4f, 0.774597f, 0.774597f, 0.774597f, 0.6f,
     0.19125f, 0.0735f, 0.0225f,
        0.7038f, 0.27048f, 0.0828f, 0.256777f, 0.137622f, 0.086014f, 0.1f,
     0.24725f, 0.1995f, 0.0745f,
        0.75164f, 0.60648f, 0.22648f, 0.628281f, 0.555802f, 0.366065f, 0.4f,
     0.19225f, 0.19225f, 0.19225f,
        0.50754f, 0.50754f, 0.50754f, 0.508273f, 0.508273f, 0.508273f, 0.4f,
     0.0f, 0.0f, 0.0f, 0.01f, 0.01f, 0.01f,
        0.50f, 0.50f, 0.50f, .25f,
     0.0f, 0.1f, 0.06f, 0.0f, 0.50980392f, 0.50980392f,
        0.50196078f, 0.50196078f, 0.50196078f, .25f,
     0.0f, 0.0f, 0.0f,
        0.1f, 0.35f, 0.1f, 0.45f, 0.55f, 0.45f, .25f,
     0.0f, 0.0f, 0.0f, 0.5f, 0.0f, 0.0f, // XX
        0.7f, 0.6f, 0.6f, .25f,
     0.0f, 0.0f, 0.0f, 0.55f, 0.55f, 0.55f,
        0.70f, 0.70f, 0.70f, .25f,
     0.0f, 0.0f, 0.0f, 0.5f, 0.5f, 0.0f,
        0.60f, 0.60f, 0.50f, .25f,
     0.02f, 0.02f, 0.02f, 0.01f, 0.01f, 0.01f, // XX
        0.4f, 0.4f, 0.4f, .078125f,
     0.0f, 0.05f, 0.05f, 0.4f, 0.5f, 0.5f,
        0.04f, 0.7f, 0.7f, .078125f,
     0.0f, 0.05f, 0.0f, 0.4f, 0.5f, 0.4f,
        0.04f, 0.7f, 0.04f, .078125f,
     0.05f, 0.0f, 0.0f, 0.5f, 0.4f, 0.4f,
        0.7f, 0.04f, 0.04f, .078125f,
     0.05f, 0.05f, 0.05f, 0.5f, 0.5f, 0.5f,
        0.7f, 0.7f, 0.7f, .078125f,
     0.05f, 0.05f, 0.0f, 0.5f, 0.5f, 0.4f,
        0.7f, 0.7f, 0.04f, .078125f };

// generally white materials for texturing

static GLfloat texMaterial[NUM_TEX_MATS][10] = {
// bright white
     0.2f, 0.2f, 0.2f,
        1.0f, 1.0f, 1.0f, 0.8f, 0.8f, 0.8f, 0.4f,
// less bright white
     0.2f, 0.2f, 0.2f,
        0.8f, 0.8f, 0.8f, 0.5f, 0.5f, 0.5f, 0.35f,
// warmish white
     0.3f, 0.2f, 0.2f,
        1.0f, 0.9f, 0.8f, 0.4f, 0.2f, 0.2f, 0.35f,
```

```
// coolish white
     0.2f, 0.2f, 0.3f,
        0.8f, 0.9f, 1.0f, 0.2f, 0.2f, 0.4f, 0.35f
};

static void SetMaterial( GLfloat *pMat )
{
    glMaterialfv (GL_FRONT, GL_AMBIENT, pMat);
    glMaterialfv (GL_BACK, GL_AMBIENT, pMat);
    pMat += 3;
    glMaterialfv (GL_FRONT, GL_DIFFUSE, pMat);
    glMaterialfv (GL_BACK, GL_DIFFUSE, pMat);
    pMat += 3;
    glMaterialfv (GL_FRONT, GL_SPECULAR, pMat);
    glMaterialfv (GL_BACK, GL_SPECULAR, pMat);
    pMat++;
    glMaterialf (GL_FRONT, GL_SHININESS, *pMat*128.0f);
    glMaterialf (GL_BACK, GL_SHININESS, *pMat*128.0f);
}

void SpecifyMaterial( unsigned int mIndex )
{
    if( mIndex >= NUM_MATERIALS )
     mIndex = NUM_MATERIALS - 1;
    SetMaterial( &teaMaterial[mIndex][0] );
}

void ChooseMaterial()
{
    static int numMat = NUM_GOOD_MATS;

    if( bTexture ) // choose a white material
     SetMaterial( &texMaterial[mfRand(NUM_TEX_MATS)][0] );
    else {
        // randomnly pick new material
        SetMaterial( &teaMaterial[ goodMaterials[mfRand(numMat)] ][0] );
    }
}
```

OBJECTS.H (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: objects.h
*
* Externals from objects.c
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

extern GLint ball;
extern GLint elbows[4], balls[4];
extern GLint shortPipe, longPipe;
extern void BuildLists(void);
```

OBJECTS.C (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: object.c
*
* Creates command lists for pipe primitives
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <windows.h>
#include <GL/gl.h>
#include "objects.h"
#include "pipes.h"
#include "texture.h"

#define PI 3.14159265358979323846f
#define ROOT_TWO 1.414213562373f

GLint ball;
GLint shortPipe, longPipe;
GLint elbows[4];
GLint balls[4];

GLint Slices;

typedef struct strpoint3d {
    GLfloat x;
    GLfloat y;
    GLfloat z;
} POINT3D;

typedef struct _MATRIX {
    GLfloat M[4][4];
} MATRIX;

#define ZERO_EPS    0.00000001

// most of these transform routines from 'flying objects' screen saver

void xformPoint(POINT3D *ptOut, POINT3D *ptIn, MATRIX *mat)
{
    double x, y, z;

    x = (ptIn->x * mat->M[0][0]) + (ptIn->y * mat->M[0][1]) +
        (ptIn->z * mat->M[0][2]) + mat->M[0][3];

    y = (ptIn->x * mat->M[1][0]) + (ptIn->y * mat->M[1][1]) +
        (ptIn->z * mat->M[1][2]) + mat->M[1][3];
```

```
    z = (ptIn->x * mat->M[2][0]) + (ptIn->y * mat->M[2][1]) +
        (ptIn->z * mat->M[2][2]) + mat->M[2][3];

    ptOut->x = (float) x;
    ptOut->y = (float) y;
    ptOut->z = (float) z;
}

static void zeroMatrix( MATRIX *m )
{
    GLint i, j;

    for( j = 0; j < 4; j ++ ) {
     for( i = 0; i < 4; i ++ ) {
         m->M[i][j] = 0.0f;
     }
    }
}

static void matrixMult( MATRIX *m1, MATRIX *m2, MATRIX *m3 )
{
    GLint i, j;

    for( j = 0; j < 4; j ++ ) {
     for( i = 0; i < 4; i ++ ) {
         m1->M[j][i] = m2->M[j][0] * m3->M[0][i] +
                   m2->M[j][1] * m3->M[1][i] +
                   m2->M[j][2] * m3->M[2][i] +
                   m2->M[j][3] * m3->M[3][i];
     }
    }
}

void matrixIdent(MATRIX *mat)
{
    mat->M[0][0] = 1.0f; mat->M[0][1] = 0.0f;
    mat->M[0][2] = 0.0f; mat->M[0][3] = 0.0f;

    mat->M[1][0] = 0.0f; mat->M[1][1] = 1.0f;
    mat->M[1][2] = 0.0f; mat->M[1][3] = 0.0f;

    mat->M[2][0] = 0.0f; mat->M[2][1] = 0.0f;
    mat->M[2][2] = 1.0f; mat->M[2][3] = 0.0f;

    mat->M[3][0] = 0.0f; mat->M[3][1] = 0.0f;
    mat->M[3][2] = 0.0f; mat->M[3][3] = 1.0f;
}

void matrixTranslate(MATRIX *m, float xTrans, float yTrans,
                     float zTrans)
{
    m->M[0][3] = (float) xTrans;
    m->M[1][3] = (float) yTrans;
    m->M[2][3] = (float) zTrans;
```

```
}

void matrixRotate(MATRIX *m, double xTheta, double yTheta, double zTheta)
{
    float xScale, yScale, zScale;
    float sinX, cosX;
    float sinY, cosY;
    float sinZ, cosZ;

    xScale = m->M[0][0];
    yScale = m->M[1][1];
    zScale = m->M[2][2];
    sinX = (float) sin(xTheta);
    cosX = (float) cos(xTheta);
    sinY = (float) sin(yTheta);
    cosY = (float) cos(yTheta);
    sinZ = (float) sin(zTheta);
    cosZ = (float) cos(zTheta);

    m->M[0][0] = (float) ((cosZ * cosY) * xScale);
    m->M[0][1] = (float) ((cosZ * -sinY * -sinX + sinZ * cosX) * yScale);
    m->M[0][2] = (float) ((cosZ * -sinY * cosX + sinZ * sinX) * zScale);

    m->M[1][0] = (float) (-sinZ * cosY * xScale);
    m->M[1][1] = (float) ((-sinZ * -sinY * -sinX + cosZ * cosX) * yScale);
    m->M[1][2] = (float) ((-sinZ * -sinY * cosX + cosZ * sinX) * zScale);

    m->M[2][0] = (float) (sinY * xScale);
    m->M[2][1] = (float) (cosY * -sinX * yScale);
    m->M[2][2] = (float) (cosY * cosX * zScale);
}


// rotate circle around x-axis, with edge attached to anchor

static void TransformCircle(
    float angle,
    POINT3D *inPoint,
    POINT3D *outPoint,
    GLint num,
    POINT3D *anchor )
{
    MATRIX matrix1, matrix2, matrix3;
    int i;

    // translate anchor point to origin
    matrixIdent( &matrix1 );
    matrixTranslate( &matrix1, -anchor->x, -anchor->y, -anchor->z );

    // rotate by angle, cw around x-axis
    matrixIdent( &matrix2 );
    matrixRotate( &matrix2, (double) -angle, 0.0, 0.0 );

    // concat these 2
    matrixMult( &matrix3, &matrix2, &matrix1 );
```

```
    // translate back
    matrixIdent( &matrix2 );
    matrixTranslate( &matrix2,  anchor->x,  anchor->y,  anchor->z );

    // concat these 2
    matrixMult( &matrix1, &matrix2, &matrix3 );

    // transform all the points, + center
    for( i = 0; i < num; i ++, outPoint++, inPoint++ ) {
     xformPoint( outPoint, inPoint, &matrix1 );
    }
}

static void normalize( POINT3D *n )
{
    float len;

    len = (n->x * n->x) + (n->y * n->y) + (n->z * n->z);
    if (len > ZERO_EPS)
        len = (float) (1.0 / sqrt(len));
    else
        len = 1.0f;

    n->x *= len;
    n->y *= len;
    n->z *= len;
}

static void CalcNormals( POINT3D *p, POINT3D *n, POINT3D *center,
                  int num )
{
    int i;

    for( i = 0; i < num; i ++, n++, p++ ) {
     n->x = p->x - center->x;
     n->y = p->y - center->y;
     n->z = p->z - center->z;
     normalize( n );
    }
}

/*-------------------------------------------------------------------------\
|    MakeQuadStrip()                                                      |
|    - builds quadstrip between 2 rows of points. pA points to one        |
|      row of points, and pB to the next rotated row.  Because            |
|      the rotation has previously been defined CCW around the            |
|      x-axis, using an A-B sequence will result in CCW quads             |
|                                                                         |
\-------------------------------------------------------------------------*/
static void MakeQuadStrip
(
    POINT3D *pA,
    POINT3D *pB,
    POINT3D *nA,
```

```
    POINT3D *nB,
    GLfloat *tex_s,
    GLfloat *tex_t,
    GLint slices
)
{
    GLint i;

    glBegin( GL_QUAD_STRIP );

    for( i = 0; i < slices; i ++ ) {
     glNormal3fv( (GLfloat *) nA++ );
     if( bTextureCoords )
         glTexCoord2f( tex_s[0], *tex_t );
     glVertex3fv( (GLfloat *) pA++ );
     glNormal3fv( (GLfloat *) nB++ );
     if( bTextureCoords )
         glTexCoord2f( tex_s[1], *tex_t++ );
     glVertex3fv( (GLfloat *) pB++ );
    }

    glEnd();
}

#define CACHE_SIZE     100


/*----------------------------------------------------------------------\
|    BuildElbows()                                                      |
|    - builds elbows, by rotating a circle in the y=r plane             |
|      centered at (0,r,-r), CW around the x-axis at anchor pt.  |
|      (r = radius of the circle)                                       |
|    - rotation is 90.0 degrees, ending at circle in z=0 plane,  |
|      centered at origin.                                              |
|    - in order to 'mate' texture coords with the cylinders             |
|      generated with glu, we generate 4 elbows, each corresponding    |
|      to the 4 possible CW 90 degree orientations of the start point|
|      for each circle.                                                 |
|    - We call this start point the 'notch'.  If we characterize |
|      each notch by the axis it points down in the starting and |
|      ending circles of the elbow, then we get the following axis     |
|      pairs for our 4 notches:                                         |
|          - +z,+y                                                      |
|          - +x,+x                                                      |
|          - -z,-y                                                      |
|          - -x,-x                                                      |
|      Since the start of the elbow always points down +y, the 4 |
|      start notches give all possible 90.0 degree orientations  |
|      around y-axis.                                   |
|    - We can keep track of the current 'notch' vector to provide       |
|      proper mating between primitives.                                |
|    - Each circle of points is described CW from the start point,      |
|      assuming looking down the +y axis(+y direction).          |
|    - texture 's' starts at 0.0, and goes to 2.0*r/divSize at   |
|      end of the elbow.  (Then a short pipe would start with this      |
```

```
|      's', and run it to 1.0).                                  |
|                                                              |
\--------------------------------------------------------------------------*/
static void BuildElbows(void)
{
    GLfloat radius = vc.radius;
    GLfloat angle, startAng;
    GLint slices = Slices;
    GLint stacks = Slices / 2;
    GLint numPoints;
    GLfloat start_s, end_s, delta_s;
    POINT3D pi[CACHE_SIZE]; // initial row of points + center
    POINT3D p0[CACHE_SIZE]; // 2 rows of points
    POINT3D p1[CACHE_SIZE];
    POINT3D n0[CACHE_SIZE]; // 2 rows of normals
    POINT3D n1[CACHE_SIZE];
    GLfloat tex_t[CACHE_SIZE];// 't' texture coords
    GLfloat tex_s[2];  // 's' texture coords
    POINT3D center;  // center of circle
    POINT3D anchor;  // where circle is anchored
    POINT3D *pA, *pB, *nA, *nB;
    int i, j;

    // 's' texture range
    start_s = 0.0f;
    end_s = texRep.x * 2.0f * vc.radius / vc.divSize;
    delta_s = end_s;

    // calculate 't' texture coords
    for( i = 0; i <= slices; i ++ ) {
     tex_t[i] = (GLfloat) i * texRep.y / slices;
    }

    numPoints = slices + 1;

    for( j = 0; j < 4; j ++ ) {
     // starting angle increment 90.0 degrees each time
     startAng = j * PI / 2;

        // calc initial circle of points for circle centered at 0,r,-r
        // points start at (0,r,0), and rotate circle CCW

        for( i = 0; i <= slices; i ++ ) {
         angle = startAng + (2 * PI * i / slices);
         pi[i].x = radius * (float) sin(angle);
         pi[i].y = radius;
         // translate z by -r, cuz these cos calcs are for circle at origin
         pi[i].z = radius * (float) cos(angle) - radius;
        }

     // center point, tacked onto end of circle of points
     pi[i].x =  0.0f;
     pi[i].y =  radius;
     pi[i].z = -radius;
     center = pi[i];
```

```
      // anchor point
      anchor.x = anchor.z = 0.0f;
      anchor.y = radius;

      // calculate initial normals
      CalcNormals( pi, n0, &center, numPoints );

      // initial 's' texture coordinate
      tex_s[0] = start_s;

      // setup pointers
      pA = pi;
      pB = p0;
      nA = n0;
      nB = n1;

      // now iterate throught the stacks

      glNewList(elbows[j], GL_COMPILE);

         for( i = 1; i <= stacks; i ++ ) {
          // ! this angle must be negative, for correct vertex orientation !
          angle = - 0.5f * PI * i / stacks;

          // transform to get next circle of points + center
          TransformCircle( angle, pi, pB, numPoints+1, &anchor );

          // calculate normals
          center = pB[numPoints];
          CalcNormals( pB, nB, &center, numPoints );

          // calculate next 's' texture coord
          tex_s[1] = (GLfloat) start_s + delta_s * i / stacks;

          // now we've got points and normals, ready to be quadstrip'd
          MakeQuadStrip( pA, pB, nA, nB, tex_s, tex_t, numPoints );

          // reset pointers
          pA = pB;
          nA = nB;
          pB = (pB == p0) ? p1 : p0;
          nB = (nB == n0) ? n1 : n0;
          tex_s[0] = tex_s[1];
         }

         glEndList();
    }
}

/*-------------------------------------------------------------------------\
|    BuildBallJoints()                                                     |
|    - These are very similar to the elbows, in that the starting          |
|      and ending positions are almost identical.   The difference         |
|      here is that the circles in the sweep describe a sphere as          |
```

```
|      they are rotated.                                          |
|    - The starting circle has the same diameter as a pipe, but   |
|      instead of starting at a distance of 1.0*r from the node   |
|      centre, as in the elbow, it starts at the circle that            |
|      is the intersection of the standard Ball object and the    |
|      pipe.  The Ball's size was chosen to have a radius such    |
|      that it would encompass all intersection points between 2 |
|      pipes at right angles, so that there wouldn't be 'any sharp     |
|      stuff sticking out of the Ball'.  But we'll try using a    |
|      starting and ending circle coincident with the ends of the       |
|      pipes, since this will make life easier.  If it doesn't work,    |
|      we can then increase the sphere radius and futz around with      |
|      the starting and ending texture coords (which is why we're       |
|      going to all this trouble in the first place)              |
|      same as the standard Ball radius.  Therefore we use sphere       |
|      radius of (2)**0.5 * r.                                    |
|                                                        |
\-----------------------------------------------------------------------*/
static void BuildBallJoints(void)
{
    GLfloat ballRadius;
    GLfloat angle, delta_a, startAng, theta;
    GLint slices = Slices;
    GLint stacks = Slices; // same # as standard Ball
    GLint numPoints;
    GLfloat start_s, end_s, delta_s;
    POINT3D pi0[CACHE_SIZE]; // 2 circles of untransformed points
    POINT3D pi1[CACHE_SIZE];
    POINT3D p0[CACHE_SIZE]; // 2 rows of transformed points
    POINT3D p1[CACHE_SIZE];
    POINT3D n0[CACHE_SIZE]; // 2 rows of normals
    POINT3D n1[CACHE_SIZE];
    float   r[CACHE_SIZE];  // radii of the circles
    GLfloat tex_t[CACHE_SIZE];// 't' texture coords
    GLfloat tex_s[2];  // 's' texture coords
    POINT3D center;  // center of circle
    POINT3D anchor;  // where circle is anchored
    POINT3D *pA, *pB, *nA, *nB;
    int i, j, k;

    // calculate the radii for each circle in the sweep, where
    // r[i] = y = sin(angle)/r

    angle = PI / 4;  // first radius always at 45.0 degrees
    delta_a = (PI / 2.0f) / stacks;

    ballRadius = ROOT_TWO * vc.radius;
    for( i = 0; i <= stacks; i ++, angle += delta_a ) {
     r[i] = (float) sin(angle) * ballRadius;
    }

    // calculate 't' texture coords
    for( i = 0; i <= slices; i ++ ) {
     tex_t[i] = (GLfloat) i * texRep.y / slices;
    }
```

```
    // 's' texture range
    start_s = 0.0f;
    end_s = texRep.x * 2.0f * vc.radius / vc.divSize;
    delta_s = end_s;

    numPoints = slices + 1;

    // unlike the elbow, the center for the ball joint is constant
    center.x = center.y = 0.0f;
    center.z = -vc.radius;

    for( j = 0; j < 4; j ++ ) {
     // starting angle along circle, increment 90.0 degrees each time
     startAng = j * PI / 2;

        // calc initial circle of points for circle centered at 0,r,-r
        // points start at (0,r,0), and rotate circle CCW

     delta_a = 2 * PI / slices;
        for( i = 0, theta = startAng; i <= slices; i ++, theta += delta_a )
{
         pi0[i].x = r[0] * (float) sin(theta);
         pi0[i].y = vc.radius;
         // translate z by -r, cuz these cos calcs are for circle at origin
         pi0[i].z = r[0] * (float) cos(theta) - r[0];
        }

     // anchor point
     anchor.x = anchor.z = 0.0f;
     anchor.y = vc.radius;

     // calculate initial normals
     CalcNormals( pi0, n0, &center, numPoints );

     // initial 's' texture coordinate
     tex_s[0] = start_s;

     // setup pointers
     pA = pi0; // circles of transformed points
     pB = p0;
     nA = n0; // circles of transformed normals
     nB = n1;

     // now iterate throught the stacks

     glNewList(balls[j], GL_COMPILE);

        for( i = 1; i <= stacks; i ++ ) {
         // ! this angle must be negative, for correct vertex orientation !
         angle = - 0.5f * PI * i / stacks;

            // calc the next circle of untransformed points into pi1[]
```

```
            for( k = 0, theta = startAng; k <= slices; k ++,
theta+=delta_a ) {
             pi1[k].x = r[i] * (float) sin(theta);
             pi1[k].y = vc.radius;
             // translate z by -r, cuz calcs are for circle at origin
             pi1[k].z = r[i] * (float) cos(theta) - r[i];
            }

         // rotate cirle of points to next position
         TransformCircle( angle, pi1, pB, numPoints, &anchor );

         // calculate normals
         CalcNormals( pB, nB, &center, numPoints );

         // calculate next 's' texture coord
         tex_s[1] = (GLfloat) start_s + delta_s * i / stacks;

         // now we've got points and normals, ready to be quadstrip'd
         MakeQuadStrip( pA, pB, nA, nB, tex_s, tex_t, numPoints );

         // reset pointers
         pA = pB;
         nA = nB;
         pB = (pB == p0) ? p1 : p0;
         nB = (nB == n0) ? n1 : n0;
         tex_s[0] = tex_s[1];
        }

        glEndList();
    }
}

// 'glu' routines

#ifdef _EXTENSIONS_
#define COS cosf
#define SIN sinf
#define SQRT sqrtf
#else
#define COS cos
#define SIN sin
#define SQRT sqrt
#endif


/*-----------------------------------------------------------------------\
|    pipeCylinder()                                          |
|                                                    |
\-----------------------------------------------------------------------*/
static void pipeCylinder( GLfloat radius, GLfloat height, GLint slices,
     GLint stacks, GLfloat start_s, GLfloat end_s )
{
    GLint i,j;
    GLfloat sinCache[CACHE_SIZE];
    GLfloat cosCache[CACHE_SIZE];
```

```
    GLfloat sinCache2[CACHE_SIZE];
    GLfloat cosCache2[CACHE_SIZE];
    GLfloat angle;
    GLfloat zLow, zHigh;
    GLfloat zNormal;
    GLfloat delta_s;

    if (slices >= CACHE_SIZE) slices = CACHE_SIZE-1;

    zNormal = 0.0f;

    delta_s = end_s - start_s;

    for (i = 0; i < slices; i++) {
     angle = 2 * PI * i / slices;
     sinCache2[i] = (float) SIN(angle);
     cosCache2[i] = (float) COS(angle);
     sinCache[i] = (float) SIN(angle);
     cosCache[i] = (float) COS(angle);
    }

    sinCache[slices] = sinCache[0];
    cosCache[slices] = cosCache[0];
    sinCache2[slices] = sinCache2[0];
    cosCache2[slices] = cosCache2[0];

     for (j = 0; j < stacks; j++) {
         zLow = j * height / stacks;
         zHigh = (j + 1) * height / stacks;

         glBegin(GL_QUAD_STRIP);
         for (i = 0; i <= slices; i++) {
               glNormal3f(sinCache2[i], cosCache2[i], zNormal);
               if (bTextureCoords) {
                 glTexCoord2f( (float) start_s + delta_s * j / stacks,
                             (float) i * texRep.y / slices );
               }
               glVertex3f(radius * sinCache[i],
                     radius * cosCache[i], zLow);
               if (bTextureCoords) {
                 glTexCoord2f( (float) start_s + delta_s*(j+1) / stacks,
                             (float) i * texRep.y / slices );
               }
               glVertex3f(radius * sinCache[i],
                     radius * cosCache[i], zHigh);
         }
         glEnd();
     }
}


/*-----------------------------------------------------------------------\
|    pipeSphere()                                                       |
|                                                                       |
\-----------------------------------------------------------------------*/
```

```
void pipeSphere(GLfloat radius, GLint slices, GLint stacks,
           GLfloat start_s, GLfloat end_s)
{
    GLint i,j;
    GLfloat sinCache1a[CACHE_SIZE];
    GLfloat cosCache1a[CACHE_SIZE];
    GLfloat sinCache2a[CACHE_SIZE];
    GLfloat cosCache2a[CACHE_SIZE];
    GLfloat sinCache1b[CACHE_SIZE];
    GLfloat cosCache1b[CACHE_SIZE];
    GLfloat sinCache2b[CACHE_SIZE];
    GLfloat cosCache2b[CACHE_SIZE];
    GLfloat angle;
    GLfloat zLow, zHigh;
    GLfloat sintemp1, sintemp2, sintemp3, sintemp4;
    GLfloat costemp3, costemp4;
    GLfloat delta_s;
    GLint start, finish;

    if (slices >= CACHE_SIZE) slices = CACHE_SIZE-1;
    if (stacks >= CACHE_SIZE) stacks = CACHE_SIZE-1;

    // invert sense of s - it seems the glu sphere is not built similarly
    // to the glu cylinder
    // (this probably means stacks don't grow along +z - check it out)
    delta_s = start_s;
    start_s = end_s;
    end_s = delta_s;

    delta_s = end_s - start_s;

    /* Cache is the vertex locations cache */
    /* Cache2 is the various normals at the vertices themselves */

    for (i = 0; i < slices; i++) {
     angle = 2 * PI * i / slices;
     sinCache1a[i] = (float) SIN(angle);
     cosCache1a[i] = (float) COS(angle);
         sinCache2a[i] = sinCache1a[i];
         cosCache2a[i] = cosCache1a[i];
    }

    for (j = 0; j <= stacks; j++) {
     angle = PI * j / stacks;
           sinCache2b[j] = (float) SIN(angle);
           cosCache2b[j] = (float) COS(angle);
     sinCache1b[j] = radius * (float) SIN(angle);
     cosCache1b[j] = radius * (float) COS(angle);
    }
    /* Make sure it comes to a point */
    sinCache1b[0] = 0.0f;
    sinCache1b[stacks] = 0.0f;

    sinCache1a[slices] = sinCache1a[0];
    cosCache1a[slices] = cosCache1a[0];
```

```
sinCache2a[slices] = sinCache2a[0];
cosCache2a[slices] = cosCache2a[0];

/* Do ends of sphere as TRIANGLE_FAN's (if not bTextureCoords)
** We don't do it when bTextureCoords because we need to respecify the
** texture coordinates of the apex for every adjacent vertex (because
** it isn't a constant for that point)
*/
if (!bTextureCoords) {
    start = 1;
    finish = stacks - 1;

    /* Low end first (j == 0 iteration) */
    sintemp2 = sinCache1b[1];
    zHigh = cosCache1b[1];
      sintemp3 = sinCache2b[1];
      costemp3 = cosCache2b[1];
      glNormal3f(sinCache2a[0] * sinCache2b[0],
            cosCache2a[0] * sinCache2b[0],
            cosCache2b[0]);

    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0f, 0.0f, radius);

      for (i = slices; i >= 0; i--) {
            glNormal3f(sinCache2a[i] * sintemp3,
                  cosCache2a[i] * sintemp3,
                  costemp3);
          glVertex3f(sintemp2 * sinCache1a[i],
                sintemp2 * cosCache1a[i], zHigh);
      }
    glEnd();

    /* High end next (j == stacks-1 iteration) */
    sintemp2 = sinCache1b[stacks-1];
    zHigh = cosCache1b[stacks-1];
      sintemp3 = sinCache2b[stacks-1];
      costemp3 = cosCache2b[stacks-1];
      glNormal3f(sinCache2a[stacks] * sinCache2b[stacks],
            cosCache2a[stacks] * sinCache2b[stacks],
            cosCache2b[stacks]);
    glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0.0f, 0.0f, -radius);
      for (i = 0; i <= slices; i++) {
            glNormal3f(sinCache2a[i] * sintemp3,
                  cosCache2a[i] * sintemp3,
                  costemp3);
          glVertex3f(sintemp2 * sinCache1a[i],
                sintemp2 * cosCache1a[i], zHigh);
      }
    glEnd();
} else {
    start = 0;
    finish = stacks;
}
```

```
     for (j = start; j < finish; j++) {
         zLow = cosCache1b[j];
         zHigh = cosCache1b[j+1];
         sintemp1 = sinCache1b[j];
         sintemp2 = sinCache1b[j+1];
               sintemp3 = sinCache2b[j+1];
               costemp3 = cosCache2b[j+1];
               sintemp4 = sinCache2b[j];
               costemp4 = cosCache2b[j];

         glBegin(GL_QUAD_STRIP);
         for (i = 0; i <= slices; i++) {
               glNormal3f(sinCache2a[i] * sintemp3,
                     cosCache2a[i] * sintemp3,
                     costemp3);
               if (bTextureCoords) {
                 glTexCoord2f( (float) start_s + delta_s*(j+1) / stacks,
                             (float) i * texRep.y / slices );
               }
               glVertex3f(sintemp2 * sinCache1a[i],
                     sintemp2 * cosCache1a[i], zHigh);
               glNormal3f(sinCache2a[i] * sintemp4,
                     cosCache2a[i] * sintemp4,
                     costemp4);
               if (bTextureCoords) {
                 glTexCoord2f( (float) start_s + delta_s * j / stacks,
                             (float) i * texRep.y / slices );
               }
               glVertex3f(sintemp1 * sinCache1a[i],
                     sintemp1 * cosCache1a[i], zLow);
         }
         glEnd();
     }
}

/*-------------------------------------------------------------------------\
|    BuildBall()                                          |
|    - builds sphere along the +z axis, by calling pipesSphere() |
|    - What is calculated here is the starting and ending 's' values   |
|      for texturing.  This primitive will only be used for start      |
|      and end caps when texturing.  As such, the intersection points|
|      where the sphere meets the pipe must be calculated to provide   |
|      a smooth texturing transition.  Therefore, at the initial |
|      intersection point, we want s=0.0, and at the final point |
|      we want s= texRep.x * 2.0f * vc.radius / vc.divSize;            |
|      But, since this will result in a negative start_s, we have      |
|      to adjust these values to be positive - therefore add           |
|      a value to each 's' to compensate for this.              |
|                                                         |
\-------------------------------------------------------------------------*/
static void BuildBall(void)
{
    GLfloat radius = ROOT_TWO*vc.radius;
    GLint stacks = Slices;
    GLfloat start_s;
```

```
    GLfloat comp_s;
    GLfloat end_s;

    if( bTextureCoords ) {
     start_s = - texRep.x * (ROOT_TWO - 1.0f) * vc.radius / vc.divSize;
     end_s = texRep.x * (2.0f + (ROOT_TWO - 1.0f)) * vc.radius / vc.divSize;
     comp_s = (int) ( - start_s ) + 1.0f;
     start_s += comp_s;
     end_s += comp_s;
    }

    glNewList(ball, GL_COMPILE);
        pipeSphere( radius, Slices, stacks, start_s, end_s );
    glEndList();
}

/*--------------------------------------------------------------------------\
|    BuildShortPipe()                                                      |
|    - builds a long cylinder along the +z axis, from the origin           |
|    - a quad stack starts at point along +y axis, and rotate CW           |
|    - texture coord 't' is 1.0 at starting point, reducing to 0.0         |
|      after full rotation.                                                |
|    - texture 's' is 0.0 at start of cylinder, to 1.0 at end              |
|                                                                          |
\--------------------------------------------------------------------------*/
static void BuildShortPipe(void)
{
    GLfloat radius = vc.radius;
    GLfloat height = vc.divSize - 2*vc.radius;
    GLint stacks = Slices;
    GLfloat start_s;
    float end_s = (float) texRep.y;

    //start_s = 1.0f - height / vc.divSize;
    start_s = texRep.x * (1.0f - height / vc.divSize);

    glNewList(shortPipe, GL_COMPILE);
    pipeCylinder( radius, height, Slices, stacks, start_s, end_s );
    glEndList();
}

static void BuildLongPipe(void)
{
    GLfloat radius = vc.radius;
    GLfloat height = vc.divSize;
    GLint stacks = Slices;
    float end_s = (float) texRep.y;

    glNewList(longPipe, GL_COMPILE);
    pipeCylinder( radius, height, Slices, stacks, 0.0f, end_s );
    glEndList();
}

void BuildLists(void)
{
```

```
    GLint i;
    Slices = (tessLevel+2) * 4;

    ball = glGenLists(1);
    shortPipe = glGenLists(1);
    longPipe = glGenLists(1);
    for( i = 0; i < 4; i ++ )
     elbows[i] = glGenLists(1);

    if( bTextureCoords ) {
        for( i = 0; i < 4; i ++ )
         balls[i] = glGenLists(1);
        BuildBallJoints();
    }
    BuildElbows();
    BuildBall();
    BuildShortPipe();
    BuildLongPipe();
}
```

PIPES.H (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: pipes.h
*
* Include file for accessing pipelib functionality
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

typedef struct {
   float x,y,z;
} mfPoint3df;

typedef struct {
   int x,y,z;
} mfPoint3di;

typedef struct {
   float width, height, depth;
} mfSize3df;

typedef struct {
   int width, height, depth;
} mfSize3di;

typedef struct {
   float x,y;
} mfPoint2df;

typedef struct {
   int x,y;
} mfPoint2di;

typedef struct {
   float width, height;
} mfSize2df;

typedef struct {
   int width, height;
} mfSize2di;

enum {
    MF_MANUAL = 0,
    MF_STEP,
    MF_AUTO
};


// These are absolute directions, with origin in center of screen,
// looking down -z

#define NUM_DIRS 6
```

```
enum {
    PLUS_X = 0,
    MINUS_X,
    PLUS_Y,
    MINUS_Y,
    PLUS_Z,
    MINUS_Z
};

#define RANDOM_DIR     -1

// styles for pipe joints
enum {
    ELBOWS = 0,
    BALLS,
    EITHER
};

#define NUM_JOINT_STYLES     3


typedef struct {
    GLboolean   empty;
} Node;

#define MAX_TESS 3

// texture quality level
enum {
    TEX_LOW = 0,
    TEX_MID,
    TEX_HIGH
};

// viewing context stuff

typedef struct {
    float viewAngle;         // field of view angle for height
    float aspectRatio;       // width/height
    float zNear;       // near z clip value
    float zFar;              // far z clip value
} Perspective;  // perspective view description

typedef struct {
    float  zTrans;           // z translation
    float  viewDist;   // viewing distance, usually -zTrans
    int    numDiv;           // # grid divisions in x,y,z
    float  divSize;    // distance between divisions
    mfSize2di    winSize;        // window size in pixels
    mfPoint3di   numNodes;       // number of nodes in x,y,z
    mfPoint3df   world;          // view area in world space
    Perspective persp;       // perspective view description
    float  radius;           // pipe radius
    mfPoint3di   curPos;         // current x,y stray in cylinder units
```

```
    Node         *curNode;       // ptr to current node
    int    numPipes;   // number of pipes/frame
    float  yRot;       // current yRotation
} VC;  // viewing context

extern VC vc;
extern GLenum polyMode;
extern GLenum dithering;
extern GLenum shade;
extern GLenum doStipple;
extern GLenum projMode;
extern int drawMode;
extern int jointStyle;
extern int bCycleJointStyles;
extern int tessLevel;
extern int textureQuality;
extern int bTexture;
extern int bTextureCoords;
extern GLubyte stipple[];

int (*drawNext)( int );

extern void ChooseMaterial();
extern void InitPipes(int mode);
//extern void ResetPipes();
extern void SetProjMatrix();
extern void ReshapePipes(int width, int height);
extern void DrawPipes(void);
extern int InitBMPTexture( char *, int );
```

PIPES.C (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: pipes.c
*
* Core pipes code
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

#include "gl/glaux.h"

#include "pipes.h"
#include "objects.h"
#include "material.h"
#include "util.h"

GLenum polyMode;
GLenum dithering;
GLenum shade;
GLenum projMode;
int drawMode;
int bTexture;
int bTextureCoords;

enum {
    ELBOW_JOINT = 0,
    BALL_JOINT
};

#define TEAPOT 66

// these attributes correspond to screen-saver dialog items
int jointStyle;
int bCycleJointStyles;
int tessLevel;
int textureQuality;

// initial viewing context
#define NUM_DIV 16              // divisions in window in longest dimension
#define NUM_NODE (NUM_DIV - 1)  // num nodes in longest dimension
VC vc;
VC *vp;
```

```
// for now, static array
static Node node[NUM_NODE][NUM_NODE][NUM_NODE];

static int lastDir;  // last direction taken by pipe
static int notchVec; // current pipe notch vector

// forward decl'ns
void ResetPipes(void);
void DrawPipes(void);
int (*drawNext)( int );


/*-----------------------------------------------------------------------
|                                                                       |
|    InitPipes( int mode ):                                             |
|       - One time init stuff                                           |
|                                                                       |
-----------------------------------------------------------------------*/

void InitPipes( int mode )
{
    static float ambient[] = {0.1f, 0.1f, 0.1f, 1.0f};
    static float diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    static float position[] = {90.0f, 90.0f, 150.0f, 0.0f};
    static float lmodel_ambient[] = {1.0f, 1.0f, 1.0f, 1.0f};
    time_t timer;

    time( &timer );
    srand( timer );

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    glFrontFace(GL_CCW);

    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);

    glEnable( GL_AUTO_NORMAL ); // needed for GL_MAP2_VERTEX (tea)

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    ChooseMaterial();

    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);

    dithering = GL_TRUE;
    shade = GL_TRUE;
    polyMode = GL_BACK;
```

```
    projMode = GL_TRUE;

    drawMode = mode;

    vp = &vc;

    // set some initial viewing and size params

    vc.zTrans = -75.0f;
    vc.viewDist = -vc.zTrans;

    vc.numDiv = NUM_DIV;
    vc.radius = 1.0f;
    vc.divSize = 7.0f;

    vc.persp.viewAngle = 90.0f;
    vc.persp.zNear = 1.0f;

    vc.yRot = 0.0f;

    if( bTexture )
     vc.numPipes = 3;
    else
     vc.numPipes = 5;

    // Build objects
    BuildLists();
}

/*-----------------------------------------------------------------------
|                                                                       |
|    SetProjMatrix();                                                   |
|       - sets ortho or perspective viewing dimensions                  |
|                                                                       |
-----------------------------------------------------------------------*/

void SetProjMatrix()
{
    mfPoint3df *world = &vc.world;
    Perspective *persp = &vc.persp;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    persp->aspectRatio = world->x / world->y;
    persp->zFar = vc.viewDist + world->z*2;
    if( projMode ) {
        gluPerspective( persp->viewAngle,
                 persp->aspectRatio,
                 persp->zNear, persp->zFar );
    }
    else {
        glOrtho( -world->x/2, world->x/2, -world->y/2, world->y/2,
                          -world->z, world->z );
    }
```

```
    glMatrixMode(GL_MODELVIEW);

    // reset drawing state
    ResetPipes();
}

/*---------------------------------------------------------------------
|                                                                     |
|    ReshapePipes( width, height ):                                   |
|       - called on resize, expose                                    |
|       - always called on app startup                                |
|       - use it to reset projection matrix, model dimensions, etc.   |
|                                                                     |
---------------------------------------------------------------------*/

void ReshapePipes(int width, int height)
{
    float ratio;

    glViewport(0, 0, (GLint)width, (GLint)height);
    vc.winSize.width = width;
    vc.winSize.height = height;

    // adjust world dimensions to fit viewport, and adjust node counts
    if( width >= height ) {
        ratio = (float)height/width;
        vc.world.x = vc.numDiv * vc.divSize;
        vc.world.y = ratio * vc.world.x;
        vc.world.z = vc.world.x;
        vc.numNodes.x = vc.numDiv - 1;
        vc.numNodes.y = (int) (ratio * vc.numNodes.x);
        vc.numNodes.z = vc.numNodes.x;
    }
    else {
        ratio = (float)width/height;
        vc.world.y = vc.numDiv * vc.divSize;
        vc.world.x = ratio * vc.world.y;
        vc.world.z = vc.world.y;
        vc.numNodes.y = vc.numDiv - 1;
        vc.numNodes.x = (int) (ratio * vc.numNodes.y);
        vc.numNodes.z = vc.numNodes.y;
    }

    // reset stuff not done on PipeReset()
    vc.yRot = 0.0f;

    // set view matrix (resets pipe drawing as well)
    SetProjMatrix();
}


/*---------------------------------------------------------------------
|                                                                     |
|    getNeighbourNodes( pos ):                                        |
|       - get addresses of the neigbour nodes,                        |
```

```
|         and put them in supplied matrix                                |
|       - boundary hits are returned as NULL                             |
|                                                                        |
------------------------------------------------------------------------*/

static void getNeighbourNodes( mfPoint3di *pos, Node **nNode )
{
    Node *theNode = &node[pos->z][pos->y][pos->x];

    if( pos->x == 0 )
        nNode[MINUS_X] = (Node *) NULL;
    else
        nNode[MINUS_X] = theNode - 1;

    if( pos->x == (vc.numNodes.x - 1) )
        nNode[PLUS_X] = (Node *) NULL;
    else
        nNode[PLUS_X] = theNode + 1;

    if( pos->y == 0 )
        nNode[MINUS_Y] = (Node *) NULL;
    else
        nNode[MINUS_Y] = theNode - NUM_NODE;

    if( pos->y == (vc.numNodes.y - 1) )
        nNode[PLUS_Y] = (Node *) NULL;
    else
        nNode[PLUS_Y] = theNode + NUM_NODE;

    if( pos->z == 0 )
        nNode[MINUS_Z] = (Node *) NULL;
    else
        nNode[MINUS_Z] = theNode - NUM_NODE*NUM_NODE;

    if( pos->z == (vc.numNodes.z - 1) )
        nNode[PLUS_Z] = (Node *) NULL;
    else
        nNode[PLUS_Z] = theNode + NUM_NODE*NUM_NODE;
}

/*------------------------------------------------------------------------
|                                                                        |
|    getEmptyNeighbourNodes()                                            |
|       - get list of direction indices of empty node neighbours,        |
|         and put them in supplied matrix                                |
|       - return number of empty node neighbours                         |
|    - currently, if we find a node that is empty in the current |
|      direction, we duplicate it in the empty set, thereby making     |
|      it a bit more likely to go straight.                      |
|                                                                        |
------------------------------------------------------------------------*/

static int getEmptyNeighbours( Node **nNode, int *nEmpty )
{
    int i, count = 0;
```

```
    for( i = 0; i < NUM_DIRS; i ++ ) {
        if( nNode[i] && nNode[i]->empty )
     // weight straight
     {
            nEmpty[count++] = i;
         if( i == lastDir )
                 nEmpty[count++] = i;
     }
    }
    return count;
}


/*---------------------------------------------------------------------------
|                                                                           |
|    updateCurrentPosition( newDir ):
|
|                                                                           |
---------------------------------------------------------------------------*/
static void updateCurrentPosition( int newDir )
{
    switch( newDir ) {
        case PLUS_X:
            vc.curPos.x += 1;
            break;
        case MINUS_X:
            vc.curPos.x -= 1;
            break;
        case PLUS_Y:
            vc.curPos.y += 1;
            break;
        case MINUS_Y:
            vc.curPos.y -= 1;
            break;
        case PLUS_Z:
            vc.curPos.z += 1;
            break;
        case MINUS_Z:
            vc.curPos.z -= 1;
            break;
    }
}


/*---------------------------------------------------------------------------
|                                                                           |
|    align_plusz( int newDir )                                      |
|       - Aligns the z axis along specified direction               |
|                                                                           |
---------------------------------------------------------------------------*/

static void align_plusz( int newDir )
{
    // align +z along new direction
```

```
    switch( newDir ) {
        case PLUS_X:
            glRotatef( 90.0f, 0.0f, 1.0f, 0.0f);
            break;
        case MINUS_X:
            glRotatef( -90.0f, 0.0f, 1.0f, 0.0f);
            break;
        case PLUS_Y:
            glRotatef( -90.0f, 1.0f, 0.0f, 0.0f);
            break;
        case MINUS_Y:
            glRotatef( 90.0f, 1.0f, 0.0f, 0.0f);
            break;
        case PLUS_Z:
            glRotatef( 0.0f, 0.0f, 1.0f, 0.0f);
            break;
        case MINUS_Z:
            glRotatef( 180.0f, 0.0f, 1.0f, 0.0f);
            break;
    }

}

static float RotZ[NUM_DIRS][NUM_DIRS] = {
       0.0f,       0.0f,      90.0f,  90.0f,    90.0f,     -90.0f,
       0.0f,       0.0f,     -90.0f,     -90.0f,     -90.0f,      90.0f,
     180.0f,     180.0f,   0.0f,     0.0f, 180.0f,   180.0f,
       0.0f,       0.0f,       0.0f,       0.0f,   0.0f,     0.0f,
     -90.0f,  90.0f,     0.0f,     180.0f,   0.0f,   0.0f,
      90.0f, -90.0f, 180.0f,   0.0f,   0.0f,   0.0f };



/*---------------------------------------------------------------------
|                                                                      |
|    align_plusy( int lastDir, int newDir )                          |
|       - Assuming +z axis is already aligned with newDir, align |
|      +y axis BACK along lastDir                                 |
|                                                                      |
---------------------------------------------------------------------*/

static void align_plusy( int oldDir, int newDir )
{
    GLfloat rotz;

    rotz = RotZ[oldDir][newDir];
    glRotatef( rotz, 0.0f, 0.0f, 1.0f );
}

// defCylNotch shows where the notch for the default cylinder will be,
//  in absolute coords, once we do an align_plusz

static GLint defCylNotch[NUM_DIRS] =
     { PLUS_Y, PLUS_Y, MINUS_Z, PLUS_Z, PLUS_Y, PLUS_Y };
```

```
// given a dir, determine how much to rotate cylinder around z to match
notches
// format is [newDir][notchVec]

#define fXX 1.0f // float don't care value
#define iXX -1         // int don't care value

static GLfloat alignNotchRot[NUM_DIRS][NUM_DIRS] = {
     fXX,  fXX,  0.0f, 180.0f,      90.0f,     -90.0f,
     fXX,  fXX,  0.0f, 180.0f,      -90.0f, 90.0f,
     -90.0f,     90.0f,      fXX,  fXX,  180.0f,     0.0f,
     -90.0f,     90.0f,      fXX,  fXX,  0.0f, 180.0f,
     -90.0f,     90.0f,      0.0f, 180.0f,     fXX,  fXX,
     90.0f,      -90.0f,     0.0f, 180.0f,     fXX,  fXX
};

/*----------------------------------------------------------------------
|                                                                      |
|    align_notch( int newDir )                                        |
|    - a cylinder is notched, and we have to line this up              |
|      with the previous primitive's notch which is maintained as      |
|      notchVec.                                                       |
|    - this adds a rotation around z to achieve this                   |
|                                                                      |
----------------------------------------------------------------------*/

static void align_notch( int newDir, int notch )
{
    GLfloat rotz;
    GLint curNotch;

    // figure out where notch is presently after +z alignment
    curNotch = defCylNotch[newDir];
    // (don't need this now we have lut)

    // look up rotation value in table
    rotz = alignNotchRot[newDir][notch];

    if( rotz != 0.0f )
        glRotatef( rotz, 0.0f, 0.0f, 1.0f );
}

/*----------------------------------------------------------------------
|                                                                      |
|    ChooseJointType                                          |
|       - Decides which type of joint to draw                          |
|                                                                      |
----------------------------------------------------------------------*/

#define BLUE_MOON 153

static int ChooseJointType()
{
    switch( jointStyle ) {
     case ELBOWS:
```

```
        return ELBOW_JOINT;
    case BALLS:
        return BALL_JOINT;
    case EITHER:
        // draw a teapot once in a blue moon
        if( mfRand(1000) == BLUE_MOON )
          return( TEAPOT );
        // otherwise an elbow or a ball
        return( mfRand( 2 ) );
   }
}

// this array supplies the sequence of elbow notch vectors, given
//  oldDir and newDir  (0's are don't cares's)
// it is also used to determine the ending notch of an elbow
static GLint notchElbDir[NUM_DIRS][NUM_DIRS][4] = {
// oldDir = +x
    iXX,        iXX,        iXX,        iXX,
    iXX,        iXX,        iXX,        iXX,
    PLUS_Y,          MINUS_Z,    MINUS_Y,    PLUS_Z,
    MINUS_Y,    PLUS_Z,          PLUS_Y,          MINUS_Z,
    PLUS_Z,          PLUS_Y,          MINUS_Z,    MINUS_Y,
    MINUS_Z,    MINUS_Y,    PLUS_Z,          PLUS_Y,
// oldDir = -x
    iXX,        iXX,        iXX,        iXX,
    iXX,        iXX,        iXX,        iXX,
    PLUS_Y,          PLUS_Z,          MINUS_Y,    MINUS_Z,
    MINUS_Y,    MINUS_Z,    PLUS_Y,          PLUS_Z,
    PLUS_Z,          MINUS_Y,    MINUS_Z,    PLUS_Y,
    MINUS_Z,    PLUS_Y,          PLUS_Z,          MINUS_Y,
// oldDir = +y
    PLUS_X,          PLUS_Z,          MINUS_X,    MINUS_Z,
    MINUS_X,    MINUS_Z,    PLUS_X,          PLUS_Z,
    iXX,        iXX,        iXX,        iXX,
    iXX,        iXX,        iXX,        iXX,
    PLUS_Z,          MINUS_X,    MINUS_Z,    PLUS_X,
    MINUS_Z,    PLUS_X,          PLUS_Z,          MINUS_X,
// oldDir = -y
    PLUS_X,          MINUS_Z,    MINUS_X,    PLUS_Z,
    MINUS_X,    PLUS_Z,          PLUS_X,          MINUS_Z,
    iXX,        iXX,        iXX,        iXX,
    iXX,        iXX,        iXX,        iXX,
    PLUS_Z,          PLUS_X,          MINUS_Z,    MINUS_X,
    MINUS_Z,    MINUS_X,    PLUS_Z,          PLUS_X,
// oldDir = +z
    PLUS_X,          MINUS_Y,    MINUS_X,    PLUS_Y,
    MINUS_X,    PLUS_Y,          PLUS_X,          MINUS_Y,
    PLUS_Y,          PLUS_X,          MINUS_Y,    MINUS_X,
    MINUS_Y,    MINUS_X,    PLUS_Y,          PLUS_X,
    iXX,        iXX,        iXX,        iXX,
    iXX,        iXX,        iXX,        iXX,
// oldDir = -z
    PLUS_X,          PLUS_Y,          MINUS_X,    MINUS_Y,
    MINUS_X,    MINUS_Y,    PLUS_X,          PLUS_Y,
    PLUS_Y,          MINUS_X,    MINUS_Y,    PLUS_X,
```

```
     MINUS_Y,    PLUS_X,           PLUS_Y,           MINUS_X,
     iXX,        iXX,        iXX,        iXX,
     iXX,        iXX,        iXX,        iXX
};


// this array tells you which way the notch will be once you make
// a turn
// format: notchTurn[oldDir][newDir][notchVec]
static GLint notchTurn[NUM_DIRS][NUM_DIRS][NUM_DIRS] = {
// oldDir = +x
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  MINUS_X,PLUS_X, PLUS_Z, MINUS_Z,
     iXX,  iXX,  PLUS_X, MINUS_X,PLUS_Z, MINUS_Z,
     iXX,  iXX,  PLUS_Y, MINUS_Y,MINUS_X,PLUS_X,
     iXX,  iXX,  PLUS_Y, MINUS_Y,PLUS_X, MINUS_X,
// oldDir = -x
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  PLUS_X, MINUS_X,PLUS_Z, MINUS_Z,
     iXX,  iXX,  MINUS_X,PLUS_X, PLUS_Z, MINUS_Z,
     iXX,  iXX,  PLUS_Y, MINUS_Y,PLUS_X, MINUS_X,
     iXX,  iXX,  PLUS_Y, MINUS_Y,MINUS_X,PLUS_X,
// oldDir = +y
     MINUS_Y,PLUS_Y, iXX,    iXX,  PLUS_Z, MINUS_Z,
     PLUS_Y, MINUS_Y,iXX,    iXX,  PLUS_Z, MINUS_Z,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     PLUS_X, MINUS_X,iXX,    iXX,  MINUS_Y,PLUS_Y,
     PLUS_X, MINUS_X,iXX,    iXX,  PLUS_Y, MINUS_Y,
// oldDir = -y
     PLUS_Y, MINUS_Y,iXX,    iXX,  PLUS_Z, MINUS_Z,
     MINUS_Y,PLUS_Y, iXX,    iXX,  PLUS_Z, MINUS_Z,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     PLUS_X, MINUS_X,iXX,    iXX,  PLUS_Y, MINUS_Y,
     PLUS_X, MINUS_X,iXX,    iXX,  MINUS_Y,PLUS_Y,
// oldDir = +z
     MINUS_Z,PLUS_Z, PLUS_Y, MINUS_Y,iXX,      iXX,
     PLUS_Z, MINUS_Z,PLUS_Y, MINUS_Y,iXX,      iXX,
     PLUS_X, MINUS_X,MINUS_Z,PLUS_Z, iXX,      iXX,
     PLUS_X, MINUS_X,PLUS_Z, MINUS_Z,iXX,      iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
// oldDir = -z
     PLUS_Z, MINUS_Z,PLUS_Y, MINUS_Y,iXX,      iXX,
     MINUS_Z,PLUS_Z, PLUS_Y, MINUS_Y,iXX,      iXX,
     PLUS_X, MINUS_X,PLUS_Z, MINUS_Z,iXX,      iXX,
     PLUS_X, MINUS_X,MINUS_Z,PLUS_Z, iXX,      iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX,
     iXX,  iXX,  iXX,  iXX,  iXX,  iXX
};

static GLint oppositeDir[NUM_DIRS] =
```

```
    { MINUS_X, PLUS_X, MINUS_Y, PLUS_Y, MINUS_Z, PLUS_Z };


/*-------------------------------------------------------------------------
|                                                                         |
|    ChooseElbow( int newDir, int oldDir )                                |
|       - Decides which elbow to draw                                     |
|    - The beginning of each elbow is aligned along +y, and we have       |
|      to choose the one with the notch in correct position                |
|    - The 'primary' start notch (elbow[0]) is in same direction as       |
|      newDir,                                                            |
|      and successive elbows rotate this notch CCW around +y              |
|                                                                         |
-------------------------------------------------------------------------*/
static GLint ChooseElbow( int oldDir, int newDir )
{
    int i;

    // precomputed table supplies correct elbow orientation
    for( i = 0; i < 4; i ++ ) {
     if( notchElbDir[oldDir][newDir][i] == notchVec )
         return i;
    }
    // we shouldn't arrive here
    return -1;
}

/*-------------------------------------------------------------------------\
|                                                                         |
| drawPipeSection(int dir)                                                |
|                                                                         |
|    - Draws a continuous pipe section                                    |
|       - if turning, draws a joint and a short cylinder, otherwise       |
|         draws a long cylinder.                                          |
|       - int dir: new absolute direction, or RANDOM_DIR                  |
|       - the 'current node' is set as the one we draw thru the NEXT      |
|         time around.                                                    |
|       - return: 0 if couldn't do it                                     |
|                 1 if successful                                         |
|                                                                         |
|                                                                         |
\-------------------------------------------------------------------------*/

static int drawPipeSection( int dir )
{
    static Node *nNode[NUM_DIRS];
    // weight going straight a little more
    static int nEmpty[NUM_DIRS+1];
    int numEmpty, newDir;
    int jointType;
    int iElbow;
    VC *vp = &vc;

    // find out who the neighbours are: this returns addresses of the  6
    // neighbours, or NULL, if they're out of bounds
```

```
    getNeighbourNodes( &vc.curPos, nNode );

    // determine new direction to go in

    if( dir != RANDOM_DIR ) {  // choose absolute direction
        if( (nNode[dir] != NULL) && nNode[dir]->empty ) {
            newDir = dir;
        }
        else {  // can't go in that direction
            return 0;
        }
    }
    else {  // randomnly choose one of the empty nodes
        // who's empty ?: returns the number of empty nodes, and fills the
        // nEmpty matrix with the direction indices of the empty nodes
        numEmpty = getEmptyNeighbours( nNode, nEmpty );
        if( numEmpty == 0 ) {  // no empty nodes - nowhere to go
            return 0;
        }
        // randomnly choose an empty node: by direction index
        newDir = nEmpty[mfRand( numEmpty )];
    }

    // push matrix that has initial zTrans and rotation
    glPushMatrix();

    // translate to current position
    glTranslatef( (vc.curPos.x - (vc.numNodes.x - 1)/2.0f )*vc.divSize,
                  (vc.curPos.y - (vc.numNodes.y - 1)/2.0f )*vc.divSize,
                  (vc.curPos.z - (vc.numNodes.z - 1)/2.0f )*vc.divSize );

    // draw joint if necessary, and pipe

    if( newDir != lastDir ) { // turning! - we have to draw joint
     jointType = ChooseJointType();

     switch( jointType ) {
       case BALL_JOINT:
         if( bTexture ) {
           // use special texture-friendly balls

             align_plusz( newDir );
             glPushMatrix();

             align_plusy( lastDir, newDir );

             // translate forward 1.0*r along +z to get set for drawing
elbow
             glTranslatef( 0.0f, 0.0f, vc.radius );
             // decide which elbow orientation to use
             iElbow = ChooseElbow( lastDir, newDir );
             glCallList( balls[iElbow] );

             glPopMatrix();
```

```
      }
      else {
        // draw ball in default orientation
          glCallList( ball );
          align_plusz( newDir );
      }
      // move ahead 1.0*r to draw pipe
      glTranslatef( 0.0f, 0.0f, vc.radius );
      break;

    case ELBOW_JOINT:
      align_plusz( newDir );

      // the align_plusy() here will screw up our notch calcs, so
      //  we push-pop

      glPushMatrix();

      align_plusy( lastDir, newDir );

      // translate forward 1.0*r along +z to get set for drawing elbow
      glTranslatef( 0.0f, 0.0f, vc.radius );
      // decide which elbow orientation to use
      iElbow = ChooseElbow( lastDir, newDir );
      if( iElbow == -1 ) {
        iElbow = 0; // recover
      }
      glCallList( elbows[iElbow] );

      glPopMatrix();

      glTranslatef( 0.0f, 0.0f, vc.radius );
      break;

    default:
         // Horrors! It's the teapot!
      glFrontFace( GL_CW );
      glEnable( GL_NORMALIZE );
      auxSolidTeapot(2.5 * vc.radius);
      glDisable( GL_NORMALIZE );
      glFrontFace( GL_CCW );
      align_plusz( newDir );
      // move ahead 1.0*r to draw pipe
      glTranslatef( 0.0f, 0.0f, vc.radius );
  }

  // update the current notch vector
  notchVec = notchTurn[lastDir][newDir][notchVec];

     // draw short pipe
  align_notch( newDir, notchVec );
  glCallList( shortPipe );
}
else {  // no turn
     // draw long pipe, from point 1.0*r back
```

```
     align_plusz( newDir );
     align_notch( newDir, notchVec );
     glTranslatef( 0.0f, 0.0f, -vc.radius );
     glCallList( longPipe );
    }

    glPopMatrix();

    // mark new node as non-empty
    nNode[newDir]->empty = GL_FALSE;
    vc.curNode = nNode[newDir];

    updateCurrentPosition( newDir );

    lastDir = newDir;

    return 1;
}

/*-------------------------------------------------------------------------\
|                                                                          |
| drawFirstPipeSection(int dir)                                            |
|                                                                          |
|    - Draws a starting cap and a short pipe section             |
|       - int dir: new absolute direction, or RANDOM_DIR                   |
|       - the 'current node' is set as the one we draw thru the NEXT       |
|         time around.                                                     |
|       - return: 0 if couldn't do it                                      |
|                 1 if successful                                          |
|                                                                          |
|                                                                          |
\-------------------------------------------------------------------------*/

static int drawFirstPipeSection( int dir )
{
    static Node *nNode[NUM_DIRS];
    // weight going straight a little more
    static int nEmpty[NUM_DIRS+1];
    int numEmpty, newDir;
    VC *vp = &vc;

    // find out who the neighbours are: this returns addresses of the  6
    // neighbours, or NULL, if they're out of bounds

    getNeighbourNodes( &vc.curPos, nNode );

    // determine new direction to go in

    if( dir != RANDOM_DIR ) {  // choose absolute direction
        if( (nNode[dir] != NULL) && nNode[dir]->empty ) {
            newDir = dir;
        }
        else {  // can't go in that direction
            return 0;
        }
```

```
    }
    else {  // randomnly choose one of the empty nodes
        numEmpty = getEmptyNeighbours( nNode, nEmpty );
        if( numEmpty == 0 ) {  // no empty nodes - nowhere to go
            return 0;
        }
        // randomnly choose an empty node: by direction index
        newDir = nEmpty[mfRand( numEmpty )];
    }

    // push matrix that has initial zTrans and rotation
    glPushMatrix();

    // translate to current position
    glTranslatef( (vc.curPos.x - (vc.numNodes.x - 1)/2.0f )*vc.divSize,
                  (vc.curPos.y - (vc.numNodes.y - 1)/2.0f )*vc.divSize,
                  (vc.curPos.z - (vc.numNodes.z - 1)/2.0f )*vc.divSize );

    // draw ball

    if( bTexture ) {
     align_plusz( newDir );
     glCallList( ball );
    }
    else {
     // draw ball in default orientation
     glCallList( ball );
     align_plusz( newDir );
    }

    // set initial notch vector
    notchVec = defCylNotch[newDir];

    // move ahead 1.0*r to draw pipe
    glTranslatef( 0.0f, 0.0f, vc.radius );

    // draw short pipe
    align_notch( newDir, notchVec );
    glCallList( shortPipe );

    glPopMatrix();

    // mark new node as non-empty
    nNode[newDir]->empty = GL_FALSE;
    vc.curNode = nNode[newDir];

    updateCurrentPosition( newDir );

    lastDir = newDir;

    // set normal drawing mode
    drawNext = drawPipeSection;

    return 1;
}
```

```
/*-----------------------------------------------------------------------
|                                                                       |
|    DrawEndCap():                                                      |
|       - Draws a ball, used to cap end of a pipe                |      |
|                                                                       |
-----------------------------------------------------------------------*/
void DrawEndCap()
{
    glPushMatrix();

    // translate to current position
    glTranslatef( (vc.curPos.x - (vc.numNodes.x - 1)/2.0f )*vc.divSize,
                  (vc.curPos.y - (vc.numNodes.y - 1)/2.0f )*vc.divSize,
                  (vc.curPos.z - (vc.numNodes.z - 1)/2.0f )*vc.divSize );

    if( bTexture ) {
     glPushMatrix();
     align_plusz( lastDir );
     align_notch( lastDir, notchVec );
        glCallList( ball );
     glPopMatrix();
    }
    else
        glCallList( ball );

    glPopMatrix();
}

/*-----------------------------------------------------------------------
|                                                                       |
|    ResetPipes():                                                      |
|       - Resets drawing parameters                                     |
|                                                                       |
-----------------------------------------------------------------------*/

void ResetPipes(void)
{
    int i;
    Node *pNode;

    // ! flush gl cmds before calling gdi !
    glFlush();
    RectWipeClear( vc.winSize.width, vc.winSize.height );

    glClear(GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, vc.zTrans);

    // Rotate Scene
    glRotatef( vc.yRot, 0.0f, 1.0f, 0.0f );
    vc.yRot += 9.73156f;
```

```
    vc.viewDist = -vc.zTrans;
    vc.curPos.x = vc.numNodes.x / 2;
    vc.curPos.y = vc.numNodes.y / 2;
    vc.curPos.z = vc.numNodes.z / 2;

    vc.curNode = (Node *) &node[vc.curPos.z][vc.curPos.y][vc.curPos.x];

    // Reset the node states
    pNode = &node[0][0][0];
    for( i = 0; i < (NUM_NODE)*(NUM_NODE)*(NUM_NODE); i++, pNode++ )
        pNode->empty = GL_TRUE;

    // Mark starting node as taken
    node[vc.curPos.z][vc.curPos.y][vc.curPos.x].empty = GL_FALSE;

    // Set the joint style
    if( bCycleJointStyles ) {
        if( ++jointStyle >= NUM_JOINT_STYLES )
         jointStyle = 0;
    }

    drawNext = drawFirstPipeSection;
}


/*-------------------------------------------------------------------------
|                                                                         |
|    DrawPipes(void):                                                     |
|       - Draws next section in random orientation                        |
|                                                                         |
-------------------------------------------------------------------------*/

// If random search takes longer than twice the total number
// of nodes, give up the random search.  There may not be any
// empty nodes.

#define INFINITE_LOOP   (2 * NUM_NODE * NUM_NODE * NUM_NODE)

void DrawPipes(void)
{
    static int pipeCount = 0;
    int infLoopDetect = 0;
    int count = 0;
    int x,y,z;
    BOOL bFoundNode = FALSE;

    if( drawMode != MF_MANUAL ) {

        if( !drawNext(RANDOM_DIR) ) {  // deadlock -- nowhere for current
pipe to go.
            // Cap off last pipe.

            DrawEndCap();

            ChooseMaterial();
```

```
            // If number pipes exceeds max, then reset the pipes and exit.
            // Otherwise, start a new pipe.

            if( pipeCount++ >= vc.numPipes ) {
                pipeCount = 0;
                ResetPipes();
                return;
            }
            else {
                // Find empty node

                while( !bFoundNode ) {

                    // Pick a random node.

                    x = mfRand( vc.numNodes.x );
                    y = mfRand( vc.numNodes.y );
                    z = mfRand( vc.numNodes.z );

                    // If its empty, we're done.

                    if( node[z][y][x].empty )
                        bFoundNode = TRUE;
                    else {
                        // Watch out for infinite loops!  After trying for
                        // awhile, give up on the random search and look
                        // for the first empty node.

                        if ( infLoopDetect++ > INFINITE_LOOP ) {

                            // Search for first empty node.

                            for ( x = 0; !bFoundNode && (x < vc.numNodes.x);
x++ )
                                for ( y = 0; !bFoundNode && (y <
vc.numNodes.y); y++ )
                                    for ( z = 0; !bFoundNode && (z <
vc.numNodes.z); z++ )
                                        if ( node[z][y][x].empty )
                                            bFoundNode = TRUE;

                            // If nothing found, there are no more empty
nodes.
                            // Reset the pipes and exit.

                            if ( !bFoundNode ) {
                                pipeCount = 0;
                                ResetPipes();
                                return;
                            }
                        }
                    }
                }
```

```
                // start drawing at new node next time we come thru
                vc.curPos.x = x;
                vc.curPos.y = y;
                vc.curPos.z = z;
                node[z][y][x].empty = GL_FALSE;
                vc.curNode =
                    (Node *) &node[vc.curPos.z][vc.curPos.y][vc.curPos.x];

            drawNext = drawFirstPipeSection;

                return;
             }
        }
    }

    glFlush();
}
```

RESOURCE.H (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: resource.h
*
* AppStudio generated header used by sspipes.rc.
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

//{{NO_DEPENDENCIES}}
// App Studio generated include file.
// Used by sspipes.rc
//
#define IDI_ICON1                       103
#define IDC_STATIC                      -1

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS

#define _APS_NEXT_RESOURCE_VALUE        104
#define _APS_NEXT_COMMAND_VALUE         101
#define _APS_NEXT_CONTROL_VALUE         1011
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

SSPIPES.H (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: sspipes.h
*
* Global header for 3D Pipes screen saver.
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

extern float fTesselFact;
extern ULONG ulJointType;
extern ULONG ulSurfStyle;
extern WCHAR awchTexPathname[];
extern int nTexFileOffset;

// Resource constants

#define IDS_COPYRIGHT           1001
#define IDS_SAVERNAME           1002
#define IDS_GENNAME             1003
#define IDS_INIFILE             1004
#define IDS_TESSELATION         1005
#define IDS_TEXTURE             1006
#define IDS_TEXTURE_FILE_OFFSET 1007
#define IDS_JOINT               1008
#define IDS_SURF                1009
#define IDS_TEXQUAL             1010
#define IDS_WARNING             1011
#define IDS_BITMAP_SIZE         1012
#define IDS_BITMAP_INVALID      1013
#define IDS_RUN_CONTROL_PANEL   1014
#define IDS_TEXTUREFILTER       1015
#define IDS_TEXTUREDIALOGTITLE  1016
#define IDS_BMP                 1017
#define IDS_DOTBMP              1018
#define IDS_STARDOTBMP          1019

#define DLG_SETUP_TESSEL        2001
#define DLG_SETUP_TEXTURE       2002
#define IDC_STATIC_TESS         2003
#define IDC_STATIC_TESS_MIN     2004
#define IDC_STATIC_TESS_MAX     2005
#define IDC_STATIC_TESS_GRP     2006
#define IDC_STATIC_TEXQUAL_GRP  2007

// In order for the IDC_TO_JOINT conversion macro to work, the radio buttons
// for joint types must be kept contiguous.

#define IDC_RADIO_ELBOW         2101
#define IDC_RADIO_BALL          2102
#define IDC_RADIO_MIXED         2103
#define IDC_RADIO_ALT           2104
```

```
#define JOINT_ELBOW             0
#define JOINT_BALL              1
#define JOINT_MIXED             2
#define JOINT_ALT               3

#define IDC_TO_JOINT(n)         ( (n) - IDC_RADIO_ELBOW )

// In order for the IDC_TO_SURF conversion macro to work, the radio buttons
// for surface styles must be kept contiguous.

#define IDC_RADIO_SOLID         2111
#define IDC_RADIO_TEX           2112

#define SURF_SOLID              0
#define SURF_TEX                1

#define IDC_TO_SURF(n)          ( (n) - IDC_RADIO_SOLID )

// In order for the IDC_TO_TEXQUAL conversion macro to work, the radio
buttons
// for texture quality must be kept contiguous.

#define IDC_RADIO_TEXQUAL_LOW   2121
#define IDC_RADIO_TEXQUAL_HIGH  2122

#define TEXQUAL_LOW             0
#define TEXQUAL_HIGH            1

#define IDC_TO_TEXQUAL(n)       ( (n) - IDC_RADIO_TEXQUAL_LOW )
```

SSPIPES.RC (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: sspipes.rc
*
* Resource file for OpenGL-based 3D Pipes screen saver.
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

//Microsoft App Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
#include "scrnsave.h"
#include "sspipes.h"

////////////////////////////////////////////////////////////////////////////
/////////
#undef APSTUDIO_READONLY_SYMBOLS


////////////////////////////////////////////////////////////////////////////
//
//
// Dialog
//

DLG_SCRNSAVECONFIGURE DIALOG DISCARDABLE  28, 25, 297, 113
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "3D Pipes Setup"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",1,240,5,50,14,WS_GROUP
    PUSHBUTTON      "Cancel",2,240,25,50,14,WS_GROUP
    GROUPBOX        "Joint Style",IDC_STATIC,5,0,110,50,WS_GROUP
    CONTROL         "&Elbow",IDC_RADIO_ELBOW,"Button",BS_AUTORADIOBUTTON |
                    WS_GROUP | WS_TABSTOP,15,15,35,13
    CONTROL         "&Ball",IDC_RADIO_BALL,"Button",BS_AUTORADIOBUTTON |
                    WS_GROUP | WS_TABSTOP,15,30,30,13
    CONTROL         "&Mixed",IDC_RADIO_MIXED,"Button",BS_AUTORADIOBUTTON |
                    WS_GROUP | WS_TABSTOP,55,15,38,13
    CONTROL         "&Cycle styles",IDC_RADIO_ALT,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP,55,30,50,13
    GROUPBOX        "Surface Style",IDC_STATIC,120,0,110,50,WS_GROUP
```

```
    CONTROL         "&Solid",IDC_RADIO_SOLID,"Button",BS_AUTORADIOBUTTON |
                    WS_GROUP | WS_TABSTOP,132,15,35,13
    CONTROL         "B&itmap texture",IDC_RADIO_TEX,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP,132,30,60,14
    PUSHBUTTON      "Te&xture...",DLG_SETUP_TEXTURE,240,45,50,14
    GROUPBOX        "Texture Quality",IDC_STATIC_TEXQUAL_GRP,5,55,110,50
    CONTROL         "&Default quality",IDC_RADIO_TEXQUAL_LOW,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP,15,70,65,15
    CONTROL         "&High quality",IDC_RADIO_TEXQUAL_HIGH,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP | WS_TABSTOP,15,85,65,15
    GROUPBOX        "&Resolution",IDC_STATIC_TESS_GRP,120,55,111,50
    LTEXT           "Min",IDC_STATIC_TESS_MIN,135,70,30,7
    SCROLLBAR       DLG_SETUP_TESSEL,135,80,80,11,WS_GROUP | WS_TABSTOP
    RTEXT           "Max",IDC_STATIC_TESS_MAX,185,70,30,7
END


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include ""windows.h""\r\n"
    "#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
    "#include ""scrnsave.h""\r\n"
    "#include ""sspipes.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

//////////////////////////////////////////////////////////////////////////////
/////////
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
//
// String Table
//
```

```
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDS_DESCRIPTION         "3D Pipes (OpenGL)"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDS_TEXTUREFILTER       "Bitmap files"
    IDS_TEXTUREDIALOGTITLE  "Choose Texture Bitmap"
    IDS_BMP                 "BMP"
    IDS_DOTBMP              ".BMP"
    IDS_STARDOTBMP          "*.BMP"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDS_COPYRIGHT           "Copyright (c) 1994 Microsoft Corporation"
    IDS_SAVERNAME           "Screen Saver.3DPipes"
    IDS_GENNAME             "ScreenSaver"
    IDS_INIFILE             "control.ini"
    IDS_TESSELATION         "Tesselation"
    IDS_TEXTURE             "Texture"
    IDS_TEXTURE_FILE_OFFSET "TextureFileOffset"
END

// This string table is used for registry keys -- localization not required
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDS_JOINT               "JointType"
    IDS_SURF                "SurfStyle"
    IDS_TEXQUAL             "TextureQuality"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDS_WARNING             "Warning"
    IDS_BITMAP_SIZE         "The bitmap must not be larger than %ld by %ld
pixels."
    IDS_BITMAP_INVALID      "The bitmap you selected is not a valid
texture."
    IDS_RUN_CONTROL_PANEL   "Please run the Control Panel Desktop Applet and
select another bitmap.  %ws is not a valid bitmap texture for this screen
saver."
END


#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
////
//
// Generated from the TEXTINCLUDE 3 resource.
//
```

```
//////////////////////////////////////////////////////////////////////////////
/////////
#endif    // not APSTUDIO_INVOKED
```

SSPIPES.C (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: sspipes.c
*
* Message loop and dialog box for the OpenGL-based 3D Pipes screen saver.
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

#include <windows.h>
#include <commdlg.h>
#include <scrnsave.h>
#include <GL\gl.h>
#include <math.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys\timeb.h>
#include <time.h>
#include "sspipes.h"
#include "pipes.h"
#include "gl\glaux.h"

extern void tkErrorPopups(GLboolean);

// Global strings.
#define GEN_STRING_SIZE 64
WCHAR wszTextureDialogTitle[GEN_STRING_SIZE];
WCHAR wszTextureFilter[2*GEN_STRING_SIZE];
WCHAR wszBmp[GEN_STRING_SIZE];
WCHAR wszDotBmp[GEN_STRING_SIZE];

HWND hWnd;
HANDLE hInst;
HGLRC gHrc = (HGLRC) 0;
HDC ghdcMem = 0;
HPALETTE ghpalOld, ghPalette = (HPALETTE) 0;
UINT guiOldStaticUse = SYSPAL_STATIC;
BOOL gbUseStatic = FALSE;

// Global string buffers for message box.

WCHAR gawch[2 * MAX_PATH];      // May contain a pathname
WCHAR gawchFormat[MAX_PATH];
WCHAR gawchTitle[MAX_PATH];

ULONG totalMem = 0;

// ulJointType controls the style of the elbows.
```

```
ULONG ulJointType = JOINT_ELBOW;

// ulSurfStyle determines whether the pipe surfaces are textured.

ULONG ulSurfStyle = SURF_SOLID;

// ulTexQuality control the texture quality.

ULONG ulTexQuality = TEXQUAL_LOW;

// fTesselFact controls the how finely the surface is tesselated.  It
// varies from very course (0.0) to very fine (2.0).

float fTesselFact = 1.0f;
static float fTesselSave = 1.0f;

// If ulSurfStyle indicates a textured surface, awchTexPathname specifies
// the bitmap chosen as the texture.

WCHAR awchTexPathname[MAX_PATH];
int nTexFileOffset;    // If 0, awchTexPathname is not full pathname.

#define TEX_WIDTH_MAX   1280
#define TEX_HEIGHT_MAX  1024

extern HPALETTE Hpal;

static int xSize;
static int ySize;
static int wxSize;
static int wySize;
static int updates = 0;

BOOL bVerifyDIB(WCHAR *fileName, ULONG *pWidth, ULONG *pHeight);
void TimerProc(HWND);

char *IDString(int id)
{
    static char strings[2][128];
    static int count = 0;
    char *str;

    str = strings[count];
    LoadString(hMainInstance, id, str, 128);
    count++;
    count &= 1;
    return str;
}

/****************************Public*Routine*****************************\
* getIniSettings
*
* Get the screen saver configuration options from .INI file/registry.
*
```

```
\*************************************************************************/

static void getIniSettings()
{
    WCHAR  sectName[30];
    WCHAR  itemName[30];
    WCHAR  fname[30];
    HKEY   hkey;
    WCHAR  awchDefaultBitmap[MAX_PATH];
    LONG   cjDefaultBitmap = MAX_PATH;
    int    tessel;
    LPWSTR pwsz;

    LoadString(hMainInstance, IDS_GENNAME, szScreenSaver,
sizeof(szScreenSaver) / sizeof(TCHAR));
    LoadStringW(hMainInstance, IDS_TEXTUREDIALOGTITLE,
wszTextureDialogTitle, GEN_STRING_SIZE);
    LoadStringW(hMainInstance, IDS_BMP, wszBmp, GEN_STRING_SIZE);
    LoadStringW(hMainInstance, IDS_DOTBMP, wszDotBmp, GEN_STRING_SIZE);

// wszTextureFilter requires a little more work.  We need to assemble the
file
// name filter string, which is composed of two strings separated by a NULL
// and terminated with a double NULL.

    LoadStringW(hMainInstance, IDS_TEXTUREFILTER, wszTextureFilter,
GEN_STRING_SIZE);
    pwsz = &wszTextureFilter[lstrlenW(wszTextureFilter)+1];
    LoadStringW(hMainInstance, IDS_STARDOTBMP, pwsz, GEN_STRING_SIZE);
    pwsz[lstrlenW(pwsz)+1] = L'\0';

    if (LoadStringW(hMainInstance, IDS_SAVERNAME, sectName, 30) &&
        LoadStringW(hMainInstance, IDS_INIFILE, fname, 30))
    {
        if (LoadStringW(hMainInstance, IDS_JOINT, itemName, 30))
            ulJointType = GetPrivateProfileIntW(sectName, itemName,
                                                JOINT_ELBOW, fname);

        if (LoadStringW(hMainInstance, IDS_SURF, itemName, 30))
            ulSurfStyle = GetPrivateProfileIntW(sectName, itemName,
                                                SURF_SOLID, fname);

        if (LoadStringW(hMainInstance, IDS_TEXQUAL, itemName, 30))
            ulTexQuality = GetPrivateProfileIntW(sectName, itemName,
                                                 TEXQUAL_LOW, fname);

        if (LoadStringW(hMainInstance, IDS_TESSELATION, itemName, 30))
            tessel = GetPrivateProfileIntW(sectName, itemName,
                                0, fname);
        fTesselFact  = (float)tessel / 100.0f;

    // Startup bitmap name not in the registry (yet).  We have to
    // synthesis the name from the ProductType registry entry.

        if ( RegOpenKeyExW(HKEY_LOCAL_MACHINE,
```

```
                            (PCWSTR) L"System\\CurrentControlSet\\Control\
\ProductOptions",
                            0,
                            KEY_QUERY_VALUE,
                            &hkey) == ERROR_SUCCESS )
        {

            if ( RegQueryValueExW(hkey,
                                  L"ProductType",
                                  (LPDWORD) NULL,
                                  (LPDWORD) NULL,
                                  (LPBYTE) awchDefaultBitmap,
                                  (LPDWORD) &cjDefaultBitmap) ==
ERROR_SUCCESS
                 && (cjDefaultBitmap / sizeof(WCHAR) + 4) <= MAX_PATH )
                lstrcatW(awchDefaultBitmap, wszDotBmp);
            else
                lstrcpyW(awchDefaultBitmap, L"winnt.bmp");

            RegCloseKey(hkey);
        }
        else
            lstrcpyW(awchDefaultBitmap, L"winnt.bmp");

        if (LoadStringW(hMainInstance, IDS_TEXTURE, itemName, 30))
            GetPrivateProfileStringW(sectName, itemName, awchDefaultBitmap,
                                     awchTexPathname, MAX_PATH, fname);

        if (LoadStringW(hMainInstance, IDS_TEXTURE_FILE_OFFSET, itemName,
30))
            nTexFileOffset = GetPrivateProfileIntW(sectName, itemName,
                                                   0, fname);
    }
}

/*****************************Public*Routine******************************\
* saveIniSettings
*
* Save the screen saver configuration option to the .INI file/registry.
*
\*************************************************************************/

static void saveIniSettings(HWND hDlg)
{
    WCHAR sectName[30];
    WCHAR itemName[30];
    WCHAR fname[30];
    WCHAR tmp[30];
    int pos;

    pos = GetScrollPos(GetDlgItem(hDlg,DLG_SETUP_TESSEL),SB_CTL);
    fTesselFact  = (float)pos / 100.0f;

    if (LoadStringW(hMainInstance, IDS_SAVERNAME, sectName, 30) &&
        LoadStringW(hMainInstance, IDS_INIFILE, fname, 30)) {
```

```
        if (LoadStringW(hMainInstance, IDS_JOINT, itemName, 30)) {
            wsprintfW(tmp, L"%ld", ulJointType);
            WritePrivateProfileStringW(sectName, itemName,
                                       tmp, fname);
        }
        if (LoadStringW(hMainInstance, IDS_SURF, itemName, 30)) {
            wsprintfW(tmp, L"%ld", ulSurfStyle);
            WritePrivateProfileStringW(sectName, itemName,
                                       tmp, fname);
        }
        if (LoadStringW(hMainInstance, IDS_TEXQUAL, itemName, 30)) {
            wsprintfW(tmp, L"%ld", ulTexQuality);
            WritePrivateProfileStringW(sectName, itemName,
                                       tmp, fname);
        }
        if (LoadStringW(hMainInstance, IDS_TESSELATION, itemName, 30)) {
            wsprintfW(tmp, L"%d", pos);
            WritePrivateProfileStringW(sectName, itemName,
                                       tmp, fname);
        }
        if (LoadStringW(hMainInstance, IDS_TEXTURE, itemName, 30)) {
            WritePrivateProfileStringW(sectName, itemName,
                                      awchTexPathname, fname);
        }
        if (LoadStringW(hMainInstance, IDS_TEXTURE_FILE_OFFSET, itemName,
30)) {
            wsprintfW(tmp, L"%ld", nTexFileOffset);
            WritePrivateProfileStringW(sectName, itemName,
                                       tmp, fname);
        }
    }

}

/*****************************Public*Routine******************************\
* setupDialogControls
*
* Setup the dialog controls based on the current global state.
*
\*************************************************************************/

static void setupDialogControls(HWND hDlg)
{
    int pos;

    CheckDlgButton(hDlg, IDC_RADIO_ELBOW, ulJointType == JOINT_ELBOW);
    CheckDlgButton(hDlg, IDC_RADIO_BALL , ulJointType == JOINT_BALL );
    CheckDlgButton(hDlg, IDC_RADIO_MIXED, ulJointType == JOINT_MIXED);
    CheckDlgButton(hDlg, IDC_RADIO_ALT  , ulJointType == JOINT_ALT  );

    CheckDlgButton(hDlg, IDC_RADIO_SOLID, ulSurfStyle == SURF_SOLID );
    CheckDlgButton(hDlg, IDC_RADIO_TEX  , ulSurfStyle == SURF_TEX   );

    CheckDlgButton(hDlg, IDC_RADIO_TEXQUAL_LOW ,
                   ulSurfStyle == SURF_TEX && ulTexQuality == TEXQUAL_LOW);
```

```
    CheckDlgButton(hDlg, IDC_RADIO_TEXQUAL_HIGH,
                   ulSurfStyle == SURF_TEX && ulTexQuality == TEXQUAL_HIGH);

    EnableWindow(GetDlgItem(hDlg, DLG_SETUP_TEXTURE),
                 ulSurfStyle == SURF_TEX);
    EnableWindow(GetDlgItem(hDlg, IDC_RADIO_TEXQUAL_LOW),
                 ulSurfStyle == SURF_TEX);
    EnableWindow(GetDlgItem(hDlg, IDC_RADIO_TEXQUAL_HIGH),
                 ulSurfStyle == SURF_TEX);
    EnableWindow(GetDlgItem(hDlg, IDC_STATIC_TEXQUAL_GRP),
                 ulSurfStyle == SURF_TEX);

    EnableWindow(GetDlgItem(hDlg, DLG_SETUP_TESSEL), TRUE);
    EnableWindow(GetDlgItem(hDlg, IDC_STATIC_TESS_MIN), TRUE);
    EnableWindow(GetDlgItem(hDlg, IDC_STATIC_TESS_MAX), TRUE);
    EnableWindow(GetDlgItem(hDlg, IDC_STATIC_TESS_GRP), TRUE);

    SetScrollRange(GetDlgItem(hDlg, DLG_SETUP_TESSEL), SB_CTL, 0, 200,
FALSE);
    pos = (int)(fTesselFact * 100.0f);
    SetScrollPos(GetDlgItem(hDlg, DLG_SETUP_TESSEL), SB_CTL, pos, TRUE);
}

/*****************************Public*Routine******************************\
* getTextureBitmap
*
* Use the common dialog GetOpenFileName to get the name of a bitmap file
* for use as a texture.  This function will not return until the user
* either selects a valid bitmap or cancels.  If a valid bitmap is selected
* by the user, the global array awchTexPathname will have the full path
* to the bitmap file and the global value nTexFileOffset will have the
* offset from the beginning of awchTexPathname to the pathless file name.
*
* If the user cancels, awchTexPathname and nTexFileOffset will remain
* unchanged.
*
\*************************************************************************/

void getTextureBitmap(HWND hDlg)
{
    OPENFILENAMEW ofn;
    WCHAR dirName[MAX_PATH];
    WCHAR fileName[MAX_PATH];
    PWSTR pwsz;
    BOOL bTryAgain;

// Make dialog look nice by parsing out the initial path and
// file name from the full pathname.  If this isn't done, then
// dialog has a long ugly name in the file combo box and
// directory will end up with the default current directory.

    if (nTexFileOffset) {
    // Separate the directory and file names.

        lstrcpyW(dirName, awchTexPathname);
```

```
    dirName[nTexFileOffset-1] = L'\0';
    lstrcpyW(fileName, &awchTexPathname[nTexFileOffset]);
}
else {
// If nTexFileOffset is zero, then awchTexPathname is not a full path.
// Attempt to make it a full path by calling SearchPathW.

    if ( SearchPathW(NULL, awchTexPathname, NULL, MAX_PATH,
                     dirName, &pwsz) )
    {
    // Successful.  Go ahead a change awchTexPathname to the full path
    // and compute the filename offset.

        lstrcpyW(awchTexPathname, dirName);
        nTexFileOffset = pwsz - dirName;

    // Break the filename and directory paths apart.

        dirName[nTexFileOffset-1] = L'\0';
        lstrcpyW(fileName, pwsz);
    }

// Give up and use the Windows system directory.

    else
    {
        GetWindowsDirectoryW(dirName, MAX_PATH);
        lstrcpyW(fileName, awchTexPathname);
    }
}

ofn.lStructSize = sizeof(OPENFILENAMEW);
ofn.hwndOwner = hDlg;
ofn.hInstance = NULL;
ofn.lpstrFilter = wszTextureFilter;
ofn.lpstrCustomFilter = (LPWSTR) NULL;
ofn.nMaxCustFilter = 0;
ofn.nFilterIndex = 1;
ofn.lpstrFile = fileName;
ofn.nMaxFile = MAX_PATH;
ofn.lpstrFileTitle = (LPWSTR) NULL;
ofn.nMaxFileTitle = 0;
ofn.lpstrInitialDir = dirName;
ofn.lpstrTitle = wszTextureDialogTitle;
ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;
ofn.nFileOffset = 0;
ofn.nFileExtension = 0;
ofn.lpstrDefExt = wszBmp;
ofn.lCustData = 0;
ofn.lpfnHook = (LPOFNHOOKPROC) NULL;
ofn.lpTemplateName = (LPWSTR) NULL;

do {
// Invoke the common file dialog.  If it succeeds, then validate
// the bitmap file.  If not valid, make user try again until either
```

```
    // they pick a good one or cancel the dialog.

        bTryAgain = FALSE;

        if ( GetOpenFileNameW(&ofn) ) {
            ULONG w, h;

            if (bVerifyDIB(fileName, &w, &h)) {
                if ( w <= TEX_WIDTH_MAX && h <= TEX_HEIGHT_MAX )
                {
                    lstrcpyW(awchTexPathname, fileName);
                    nTexFileOffset = ofn.nFileOffset;
                }
                else {
                    if ( LoadStringW(hMainInstance, IDS_WARNING, gawchTitle,
MAX_PATH) &&
                         LoadStringW(hMainInstance, IDS_BITMAP_SIZE,
gawchFormat, MAX_PATH) )
                    {
                        wsprintfW(gawch, gawchFormat, TEX_WIDTH_MAX,
TEX_HEIGHT_MAX);
                        MessageBoxW(NULL, gawch, gawchTitle, MB_OK);
                    }
                    bTryAgain = TRUE;
                }
            }
            else {
                if ( LoadStringW(hMainInstance, IDS_WARNING, gawchTitle,
MAX_PATH) &&
                     LoadStringW(hMainInstance, IDS_BITMAP_INVALID,
gawchFormat, MAX_PATH) )
                {
                    MessageBoxW(NULL, gawchFormat, gawchTitle, MB_OK);
                }
                bTryAgain = TRUE;
            }
        }

    // If need to try again, recompute dir and file name so dialog
    // still looks nice.

        if (bTryAgain && ofn.nFileOffset) {
            lstrcpyW(dirName, fileName);
            dirName[ofn.nFileOffset-1] = L'\0';
            lstrcpyW(fileName, &fileName[ofn.nFileOffset]);
        }

    } while (bTryAgain);
}

BOOL WINAPI RegisterDialogClasses(HANDLE hinst)
{
    return TRUE;
}
```

```
/*****************************Public*Routine******************************\
* ScreenSaverConfigureDialog
*
* Screen saver setup dialog box procedure.
\*************************************************************************/

BOOL ScreenSaverConfigureDialog(HWND hDlg, UINT message,
                                WPARAM wParam, LPARAM lParam)
{
    int wTmp;
    int optMask = 1;

    switch (message)
    {
        case WM_INITDIALOG:
            getIniSettings();
            setupDialogControls(hDlg);
            return TRUE;

        case WM_HSCROLL:
            if (lParam == (LPARAM) GetDlgItem(hDlg,DLG_SETUP_TESSEL)) {
                wTmp =
GetScrollPos(GetDlgItem(hDlg,DLG_SETUP_TESSEL),SB_CTL);
                switch(LOWORD(wParam))
                {
                    case SB_PAGEDOWN:
                        wTmp += 10;
                    case SB_LINEDOWN:
                        wTmp += 1;
                        wTmp = min(200, wTmp);
                        break;
                    case SB_PAGEUP:
                        wTmp -= 10;
                    case SB_LINEUP:
                        wTmp -= 1;
                        wTmp = max(0, (int)wTmp);
                        break;
                    case SB_THUMBPOSITION:
                        wTmp = HIWORD(wParam);
                        break;
                    default:
                        break;
                }
                SetScrollPos(GetDlgItem(hDlg, DLG_SETUP_TESSEL), SB_CTL,
wTmp,
                             TRUE);
                fTesselFact  = (float)wTmp / 100.0f;
            }
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                case IDC_RADIO_ELBOW:
```

```
                case IDC_RADIO_BALL:
                case IDC_RADIO_MIXED:
                case IDC_RADIO_ALT:
                    ulJointType = IDC_TO_JOINT(LOWORD(wParam));
                    setupDialogControls(hDlg);
                    break;

                case IDC_RADIO_SOLID:
            case IDC_RADIO_TEX:
                    ulSurfStyle = IDC_TO_SURF(LOWORD(wParam));
                    setupDialogControls(hDlg);
                    break;

                case IDC_RADIO_TEXQUAL_LOW:
                case IDC_RADIO_TEXQUAL_HIGH:
                    ulTexQuality = IDC_TO_TEXQUAL(LOWORD(wParam));
                    setupDialogControls(hDlg);
                    break;

                case DLG_SETUP_TEXTURE:
                getTextureBitmap(hDlg);
                setupDialogControls(hDlg);
                    break;

                case IDOK:
                    saveIniSettings(hDlg);
                    EndDialog(hDlg, TRUE);
                    break;

                case IDCANCEL:
                    EndDialog(hDlg, FALSE);
                    break;

                default:
                    break;
            }
            return TRUE;

        default:
            return 0;
    }
    return 0;
}

// These are in GLAUX.LIB
extern unsigned char threeto8[];
extern unsigned char twoto8[];
extern unsigned char oneto8[];

unsigned char
jComponentFromIndex(i, nbits, shift)
{
    unsigned char val;

    val = i >> shift;
```

```
    switch (nbits) {

    case 1:
        val &= 0x1;
        return oneto8[val];

    case 2:
        val &= 0x3;
        return twoto8[val];

    case 3:
        val &= 0x7;
        return threeto8[val];

    default:
        return 0;
    }
}


void
vCreateRGBPalette(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    LOGPALETTE *pPal;
    int n, i;

    ppfd = &pfd;
    n = GetPixelFormat(hdc);
    DescribePixelFormat(hdc, n, sizeof(PIXELFORMATDESCRIPTOR), ppfd);

    if (ppfd->dwFlags & PFD_NEED_PALETTE) {
        n = 1 << ppfd->cColorBits;
        pPal = (PLOGPALETTE)LocalAlloc(LMEM_FIXED, sizeof(LOGPALETTE) +
                n * sizeof(PALETTEENTRY));
        pPal->palVersion = 0x300;
        pPal->palNumEntries = n;
        for (i=0; i < n; i++) {
            pPal->palPalEntry[i].peRed =
                    jComponentFromIndex(i, ppfd->cRedBits, ppfd->cRedShift);
            pPal->palPalEntry[i].peGreen =
                    jComponentFromIndex(i, ppfd->cGreenBits, ppfd-
>cGreenShift);
            pPal->palPalEntry[i].peBlue =
                    jComponentFromIndex(i, ppfd->cBlueBits, ppfd-
>cBlueShift);
            pPal->palPalEntry[i].peFlags = ((n == 256) && (i == 0 || i ==
255))
                                            ? 0 : PC_NOCOLLAPSE;
        }

        ghPalette = CreatePalette(pPal);
        LocalFree(pPal);

        if ( n == 256 )
```

```
        {
            gbUseStatic = TRUE;
            guiOldStaticUse = SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
        }

        ghpalOld = SelectPalette(hdc, ghPalette, FALSE);
        n = RealizePalette(hdc);
    }
}


BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL;

    ppfd->iLayerType = PFD_MAIN_PLANE;
    ppfd->cAlphaBits = 0;
    ppfd->cAuxBuffers = 0;
    ppfd->bReserved = 0;

    ppfd->iPixelType = PFD_TYPE_RGBA;
    ppfd->cColorBits = 24;

    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 ||
         SetPixelFormat(hdc, pixelformat, ppfd) == FALSE )
        return FALSE;

    return TRUE;
}

HGLRC hrcInitGL(HWND hwnd, HDC hdc)
{
    HGLRC hrc;
    RECT rect;

    GetClientRect(hwnd, &rect);
    wxSize = rect.right - rect.left;
    wySize = rect.bottom - rect.top;

    // communicate window size to pipelib:
    vc.winSize.width = wxSize;
    vc.winSize.height= wySize;

    bSetupPixelFormat(hdc);
```

```
    vCreateRGBPalette(hdc);
    hrc = wglCreateContext(hdc);
    if (!hrc || !wglMakeCurrent(hdc, hrc))
        return NULL;

    glViewport(0, 0, wxSize, wySize);

    return hrc;
}


void
vCleanupGL(HGLRC hrc, HWND hwnd)
{
    HDC hdc = wglGetCurrentDC();

    if (ghPalette)
    {
        if ( gbUseStatic )
        {
            SetSystemPaletteUse(hdc, guiOldStaticUse);
            PostMessage(HWND_BROADCAST, WM_SYSCOLORCHANGE, 0, 0);
        }
        DeleteObject(SelectPalette(hdc, ghpalOld, FALSE));
    }

    /*  Destroy our context and release the DC */
    ReleaseDC(hwnd, hdc);
    wglDeleteContext(hrc);
}


/******************************Public*Routine******************************\
* PipesGetHWND()
*
* Get HWND for drawing.
*
* Note: done here so that pipes.lib is tk-independent.
*
\**************************************************************************/

HWND PipesGetHWND()
{
    return( hMainWindow );
}

/******************************Public*Routine******************************\
* TimerProc
*
* Procedure called on each screen saver timer tick to draw the next section
* of pipe.
*
\**************************************************************************/

void TimerProc(HWND hwnd)
```

```
{
    static int busy = FALSE;

    if (busy)
        return;
    busy = TRUE;

    updates++;

// Put stuff to be done on each update here:

    DrawPipes();

    busy = FALSE;
}


/*****************************Public*Routine******************************\
* InitPipeSettings
*
* - translates variables set from the dialog boxes, into
*   variables used in pipelib.
*
\*************************************************************************/

void InitPipeSettings()
{
    // tesselation from fTesselFact(0.0-2.0) to tessLevel(0-MAX_TESS)

    tessLevel = (int) (fTesselFact * (MAX_TESS+1) / 2.0001f);

    // texturing state

    switch( ulTexQuality ) {
     case TEXQUAL_LOW:
         textureQuality = TEX_LOW;
         break;
     case TEXQUAL_HIGH:
     default:
         textureQuality = TEX_MID;
         break;
    }

    if( ulSurfStyle == SURF_TEX ) {
     // try to load the bmp file
     // if nTexFileOffset is 0, then awchTexPathname is not a valid
     // pathname and we should not call to load texture (InitBMPTexture
     // calls glaux.lib which may try and put a MessageBox without focus
     // -- this causes a problem and the screen saver will have to be
     // killed).
     if( nTexFileOffset &&
         InitBMPTexture( (char *) awchTexPathname, 1 ) ) {
         bTextureCoords = 1;
         bTexture = 1;
     }
```

```
     else {  // couldn't open .bmp file
         bTextureCoords = 0;
         bTexture = 0;
     }
    }

    // joint types

    bCycleJointStyles = 0;

    switch( ulJointType ) {
     case JOINT_ELBOW:
         jointStyle = ELBOWS;
         break;
     case JOINT_BALL:
            jointStyle = BALLS;
         break;
     case JOINT_MIXED:
         jointStyle = EITHER;
         break;
     case JOINT_ALT:
         bCycleJointStyles = 1;
         jointStyle = EITHER;
         break;
     default:
         break;
    }

}


/******************************Public*Routine******************************\
* ScreenSaverProc
*
* Screen saver window procedure.
*
\**************************************************************************/

LONG ScreenSaverProc(HWND hwnd, UINT message, WPARAM wParam,
                     LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    static POINT point;
    static WORD wTimer = 0;
    static BOOL bInited = FALSE;

    switch (message) {

    case WM_CREATE:
     getIniSettings();

    // Make sure the selected texture file is OK.

     if ( ulSurfStyle == SURF_TEX )
```

```
     {
         ULONG w, h;
         WCHAR fileName[MAX_PATH];
         PWSTR pwsz;

         lstrcpyW(fileName, awchTexPathname);

         if ( SearchPathW(NULL, fileName, NULL, MAX_PATH,
                     awchTexPathname, &pwsz)
            && bVerifyDIB(awchTexPathname, &w, &h)
                 && w <= TEX_WIDTH_MAX && h <= TEX_HEIGHT_MAX
            )
         {
           nTexFileOffset = pwsz - awchTexPathname;
         }
         else
         {
                lstrcpyW(awchTexPathname, fileName);    // restore

           nTexFileOffset = 0; // A valid pathname will never have this 0
                           // so this is a good error indicator.

                if ( LoadStringW(hMainInstance, IDS_WARNING, gawchTitle,
MAX_PATH) &&
                     LoadStringW(hMainInstance, IDS_RUN_CONTROL_PANEL,
gawchFormat, MAX_PATH) )
                {
                    wsprintfW(gawch, gawchFormat, awchTexPathname);
                    MessageBoxW(NULL, gawch, gawchTitle, MB_OK);
                }
         }
     }

        break;

    case WM_DESTROY:
        if (wTimer) {
            KillTimer(hwnd, wTimer);
        }

        // Put stuff to be done on app close here:

        if (gHrc)
            vCleanupGL(gHrc, hwnd);
        break;

    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        EndPaint(hwnd, &ps);

    // Normally do this in WM_CREATE, but in this case if we do, you see
    // the system colors get changed.  By doing it here, we delay the
    // initialization until after the screen has already been cleared.

        if (!bInited)
```

```
        {
        // Put initial setup stuff here:

            bInited = TRUE;

            /* do not release the dc until we quit */
            if (hdc = GetDC(hwnd)) {
                if ( gHrc = hrcInitGL(hwnd, hdc) ) {
                    tkErrorPopups(FALSE);   // disable message boxes in
GLAUX

                    InitPipeSettings();
                    InitPipes( MF_AUTO );
                    ReshapePipes( wxSize, wySize );

                    wTimer = SetTimer( hwnd, 1, 16, NULL );
                }
                else {
                    ReleaseDC(hwnd, hdc);
                }
            }
        }

        break;

    case WM_TIMER:
        TimerProc(hwnd);
        break;

    }
    return DefScreenSaverProc(hwnd, message, wParam, lParam);
}


#define BFT_BITMAP  0x4d42  // 'BM' -- indicates structure is
BITMAPFILEHEADER

// struct BITMAPFILEHEADER {
//      WORD  bfType
//      DWORD bfSize
//      WORD  bfReserved1
//      WORD  bfReserved2
//      DWORD bfOffBits
// }
#define OFFSET_bfType       0
#define OFFSET_bfSize       2
#define OFFSET_bfReserved1  6
#define OFFSET_bfReserved2  8
#define OFFSET_bfOffBits    10
#define SIZEOF_BITMAPFILEHEADER 14

// Read a WORD-aligned DWORD.  Needed because BITMAPFILEHEADER has
// WORD-alignment.
#define READDWORD(pv)    ( (DWORD)((PWORD)(pv))[0]                \
                         | ((DWORD)((PWORD)(pv))[1] << 16) )    \
```

```
// Computes the number of BYTES needed to contain n number of bits.
#define BITS2BYTES(n)   ( ((n) + 7) >> 3 )

/*****************************Public*Routine******************************\
* bVerifyDIB
*
* Stripped down version of tkDIBImageLoadAW that verifies that a bitmap
* file is valid and, if so, returns the bitmap dimensions.
*
* Returns:
*   TRUE if valid bitmap file; otherwise, FALSE.
*
\*************************************************************************/

BOOL bVerifyDIB(WCHAR *fileName, ULONG *pWidth, ULONG *pHeight)
{
    BOOL bRet = FALSE;
    BITMAPFILEHEADER *pbmf;         // Ptr to file header
    BITMAPINFOHEADER *pbmihFile;    // Ptr to file's info header (if it
exists)
    BITMAPCOREHEADER *pbmchFile;    // Ptr to file's core header (if it
exists)

    // These need to be cleaned up when we exit:
    HANDLE     hFile = INVALID_HANDLE_VALUE;        // File handle
    HANDLE     hMap = (HANDLE) NULL;                // Mapping object handle
    PVOID      pvFile = (PVOID) NULL;               // Ptr to mapped file

// Map the DIB file into memory.

    hFile = CreateFileW((LPWSTR) fileName, GENERIC_READ, FILE_SHARE_READ,
NULL, OPEN_EXISTING, 0, 0);
    if (hFile == INVALID_HANDLE_VALUE)
        goto bVerifyDIB_cleanup;

    hMap = CreateFileMappingW(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (!hMap)
        goto bVerifyDIB_cleanup;

    pvFile = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);
    if (!pvFile)
        goto bVerifyDIB_cleanup;

// Check the file header.  If the BFT_BITMAP magic number is there,
// then the file format is a BITMAPFILEHEADER followed immediately
// by either a BITMAPINFOHEADER or a BITMAPCOREHEADER.  The bitmap
// bits, in this case, are located at the offset bfOffBits from the
// BITMAPFILEHEADER.
//
// Otherwise, this may be a raw BITMAPINFOHEADER or BITMAPCOREHEADER
// followed immediately with the color table and the bitmap bits.

    pbmf = (BITMAPFILEHEADER *) pvFile;
```

```
    if ( pbmf->bfType == BFT_BITMAP )
        pbmihFile = (BITMAPINFOHEADER *) ((PBYTE) pbmf +
SIZEOF_BITMAPFILEHEADER);
    else
        pbmihFile = (BITMAPINFOHEADER *) pvFile;

// Get the width and height from whatever header we have.
//
// We distinguish between BITMAPINFO and BITMAPCORE cases based upon
// BITMAPINFOHEADER.biSize.

    // Note: need to use safe READDWORD macro because pbmihFile may
    // have only WORD alignment if it follows a BITMAPFILEHEADER.

    switch (READDWORD(&pbmihFile->biSize))
    {
    case sizeof(BITMAPINFOHEADER):

        *pWidth  = READDWORD(&pbmihFile->biWidth);
        *pHeight = READDWORD(&pbmihFile->biHeight);
        bRet = TRUE;

        break;

    case sizeof(BITMAPCOREHEADER):
        pbmchFile = (BITMAPCOREHEADER *) pbmihFile;

    // Convert BITMAPCOREHEADER to BITMAPINFOHEADER.

        *pWidth  = (DWORD) pbmchFile->bcWidth;
        *pHeight = (DWORD) pbmchFile->bcHeight;
        bRet = TRUE;

        break;

    default:
        break;
    }

bVerifyDIB_cleanup:

    if (pvFile)
        UnmapViewOfFile(pvFile);

    if (hMap)
        CloseHandle(hMap);

    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);

    return bRet;
}
```

TEXTURE.H (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: texture.h
*
* Externals from texture.c
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

extern mfPoint2di texRep;  // texture repetition
```

TEXTURE.C (PIPES Sample)

```
/*****************************Module*Header*******************************\
* Module Name: texture.c
*
* Handles texture initialization
*
* Copyright (c) 1994 Microsoft Corporation
*
\*************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#include <windows.h>
#include <GL/gl.h>
#include <GL/glaux.h>

#include "pipes.h"
#include "texture.h"

mfPoint2di texRep; // texture repetition factors

// creates a texture from a BMP file

int InitBMPTexture( char *bmpfile, int wflag )
{
    AUX_RGBImageRec *image = (AUX_RGBImageRec *) NULL;
    double xPow2, yPow2;
    int ixPow2, iyPow2;
    int xSize2, ySize2;
    BYTE *pData;
    float fxFact, fyFact;
    GLint glMaxTexDim;

    // load the bmp file

    if( wflag )
     image = auxDIBImageLoadW( (LPCWSTR) bmpfile);
    else
     image = auxDIBImageLoadA( bmpfile );

    if( !image )
     return 0; // failure


    glEnable(GL_TEXTURE_2D);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

```
switch( textureQuality ) {
 case TEX_HIGH:
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR_MIPMAP_NEAREST);
     break;
 case TEX_MID:
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
     break;
 case TEX_LOW:
 default:
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
     break;
}

glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);
glMaxTexDim = min(256, glMaxTexDim);

if (image->sizeX <= glMaxTexDim)
    xPow2 = log((double)image->sizeX) / log((double)2.0);
else
    xPow2 = log((double)glMaxTexDim) / log((double)2.0);

if (image->sizeY <= glMaxTexDim)
    yPow2 = log((double)image->sizeY) / log((double)2.0);
else
    yPow2 = log((double)glMaxTexDim) / log((double)2.0);

ixPow2 = (int)xPow2;
iyPow2 = (int)yPow2;

if (xPow2 != (double)ixPow2)
    ixPow2++;
if (yPow2 != (double)iyPow2)
    iyPow2++;

xSize2 = 1 << ixPow2;
ySize2 = 1 << iyPow2;

pData = LocalAlloc(LMEM_FIXED, xSize2 * ySize2 * 3 * sizeof(BYTE));
if (!pData)
    return 0;

gluScaleImage(GL_RGB, image->sizeX, image->sizeY,
              GL_UNSIGNED_BYTE, image->data,
              xSize2, ySize2, GL_UNSIGNED_BYTE,
              pData);

if( textureQuality == TEX_HIGH ) {
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3, image->sizeX, image->sizeY,
                      GL_RGB, GL_UNSIGNED_BYTE, image->data);
```

```
    }
    else {
           glTexImage2D(GL_TEXTURE_2D, 0, 3, xSize2, ySize2,
                        0, GL_RGB, GL_UNSIGNED_BYTE, pData);
    }

    // Figure out repetition factor of texture, based on bitmap size and
    // screen size.
    //
    // We arbitrarily decide to repeat textures that are smaller than
    // 1/8th of screen width or height.

    texRep.x = texRep.y = 1;

    if( (fxFact = vc.winSize.width / xSize2 / 8.0f) >= 1.0f)
     texRep.x = (int) (fxFact+0.5f);

    if( (fyFact = vc.winSize.height / ySize2 / 8.0f) >= 1.0f)
     texRep.y = (int) (fyFact+0.5f);

    LocalFree(image->data);
    image->data = pData;

    return 1; // success
}
```

UTIL.H (PIPES Sample)

```
/*****************************Module*Header*******************************\
* Module Name: util.h
*
* Externals from util.c
*
* Copyright (c) 1994 Microsoft Corporation
*
\*************************************************************************/

extern int  mfRand( int );
extern void RectWipeClear( int width, int height );
```

UTIL.C (PIPES Sample)

```
/******************************Module*Header*******************************\
* Module Name: util.c
*
* Misc. utility functions
*
* Copyright (c) 1994 Microsoft Corporation
*
\**************************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include <GL/gl.h>

extern HWND PipesGetHWND();

/*-------------------------------------------------------------------------
|                                                       |
|    mfRand(numVal):                                          |
|        Generates random number 0..(numVal-1)                    |
|                                                       |
-------------------------------------------------------------------------*/

int mfRand( int numVal )
{
    int num;

    num = (int) ( numVal * ( ((float)rand()) / ((float)(RAND_MAX+1)) ) );
    return num;
}


/*-------------------------------------------------------------------------
|                                                       |
|    RectWipeClear(width, height):                                          |
|       - Does a rectangular wipe (or clear) by drawing in a sequence   |
|         of rectangles using Gdi                                           |
|    MOD: add calibrator capability to adjust speed for different      |
|         architectures                                       |
|                                                       |
-------------------------------------------------------------------------*/
void RectWipeClear( int width, int height )
{
    HWND hwnd;
    HDC hdc;
    HBRUSH hbr;
    RECT rect;
    int i, j, xinc, yinc, numDivs = 500;
    int xmin, xmax, ymin, ymax;
    int repCount = 3;
```

```
    xinc = 1;
    yinc = 1;
    numDivs = height;
    xmin = ymin = 0;
    xmax = width;
    ymax = height;

    hwnd = PipesGetHWND();
    hdc = GetDC( hwnd );

    hbr = CreateSolidBrush( RGB( 0, 0, 0 ) );

    for( i = 0; i < (numDivs/2 - 1); i ++ ) {
      for( j = 0; j < repCount; j ++ ) {
     rect.left = xmin; rect.top = ymin;
     rect.right = xmax; rect.bottom = ymin + yinc;
     FillRect( hdc, &rect, hbr );
     rect.top = ymax - yinc;
     rect.bottom = ymax;
     FillRect( hdc, &rect, hbr );
     rect.top = ymin + yinc;
     rect.right = xmin + xinc; rect.bottom = ymax - yinc;
     FillRect( hdc, &rect, hbr );
     rect.left = xmax - xinc; rect.top = ymin + yinc;
     rect.right = xmax; rect.bottom = ymax - yinc;
     FillRect( hdc, &rect, hbr );
      }

     xmin += xinc;
     xmax -= xinc;
     ymin += yinc;
     ymax -= yinc;
    }

    // clear last square in middle

    rect.left = xmin; rect.top = ymin;
    rect.right = xmax; rect.bottom = ymax;
    FillRect( hdc, &rect, hbr );

    ReleaseDC( hwnd, hdc );

    GdiFlush();
}
```

glAccum

The **glAccum** function operates on the accumulation buffer.

void glAccum(
GLenum *op*,
GLfloat *value*
);

Parameters

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.

Remarks

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 **glGetIntegerv** four times, with arguments **GL_ACCUM_RED_BITS**, **GL_ACCUM_GREEN_BITS**, **GL_ACCUM_BLUE_BITS**, and **GL_ACCUM_ALPHA_BITS**, respectively. Regardless of the number of bits per component, however, the range of values stored by each component is
[- 1, 1]. The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

The **glAccum** function 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 are applied identically to the red, green, blue, and alpha components of each pixel. The contents of an accumulation buffer pixel component are undefined if the **glAccum** operation results in a value outside the range [- 1,1]. The operations are as follows:

GL_ACCUM
Obtains R, G, B, and A values from the buffer currently selected for reading (see **glReadBuffer**). 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 [0,1], 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.

The accumulation buffer is cleared by specifying R, G, B, and A values to set it to with the **glClearAccum** directive, and issuing a **glClear** command with the accumulation buffer enabled.

Only those pixels within the current scissor box are updated by any **glAccum** operation.

The following functions retrieve information related to the **glAccum** function:

glGet with argument **GL_ACCUM_RED_BITS**

glGet with argument **GL_ACCUM_GREEN_BITS**

glGet with argument **GL_ACCUM_BLUE_BITS**

glGet with argument **GL_ACCUM_ALPHA_BITS**

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 **glAccum** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBlendFunc, **glClear**, **glClearAccum**, **glCopyPixels**, **glGet**, **glLogicOp**, **glPixelStore**, **glPixelTransfer**, **glReadPixels**, **glReadBuffer**, **glScissor**, **glStencilOp**

glAlphaFunc

The **glAlphaFunc** function specifies the alpha test function.

void glAlphaFunc(
  GLenum *func*,
  GLclampf *ref*
  );

Parameters

func
  Specifies the alpha comparison function. Symbolic constants **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS** are accepted. The default function is **GL_ALWAYS**.

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 default reference is 0.

Remarks

The alpha test discards fragments depending on the outcome of a comparison between the incoming fragments alpha value and a constant reference value. The **glAlphaFunc** function specifies the reference and comparison function. The comparison is performed only if alpha testing is enabled. (See **glEnable** of **GL_ALPHA_TEST**.)

The *func* and *ref* parameters 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 comparison passes, the incoming fragment is drawn, conditional on subsequent stencil and depth buffer tests. If the comparison fails, 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.

The **glAlphaFunc** function operates on all pixel writes, including those resulting from the scan conversion of points, lines, polygons, and bitmaps, and from pixel draw and copy operations. **glAlphaFunc** does not affect screen clear operations.

Alpha testing is done only in RGBA mode.

The following functions retrieve information related to the **glAlphaFunc** function:

glGet with argument **GL_ALPHA_TEST_FUNC**

glGet with argument **GL_ALPHA_TEST_REF**

glIsEnabled with argument **GL_ALPHA_TEST**

Errors

GL_INVALID_ENUM is generated if *func* is not an accepted value.

GL_INVALID_OPERATION is generated if **glAlphaFunc** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBlendFunc, **glClear**, **glDepthFunc**, **glEnable**, **glStencilFunc**

glBegin, glEnd

The **glBegin** and **glEnd** functions delimit the vertexes of a primitive or a group of like primitives.

void glBegin(
  GLenum *mode*
  );

Parameters

mode
  Specifies the primitive or primitives that will be created from vertexes presented between **glBegin** and the subsequent **glEnd**. Ten symbolic constants are accepted: **GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP,** and **GL_POLYGON**.

void glEnd(
  void
  );

Remarks

The **glBegin** and **glEnd** functions delimit the vertexes that define a primitive or a group of like primitives. The **glBegin** function accepts a single argument that specifies which of ten ways the vertexes are interpreted. Taking *n* as an integer count starting at one, and *N* as the total number of vertexes 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 vertexes as an independent line segment. Vertexes *2n - 1* and *2n* 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. Vertexes *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. Vertexes *n* and *n+1* define line *n*. The last line, however, is defined by vertexes *N* and *1*. *N* lines are drawn.

GL_TRIANGLES
  Treats each triplet of vertexes as an independent triangle. Vertexes *3n - 2*, *3n-1*, and *3n* 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 vertexes. For odd *n*, vertexes *n*, *n+1*, and *n+2* define triangle *n*. For even *n*, vertexes *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 vertexes. Vertexes *1*, *n+1*, and *n+2* define triangle *n*. *N - 2* triangles are drawn.

GL_QUADS
  Treats each group of four vertexes as an independent quadrilateral. Vertexes *4n - 3*, *4n - 2*, *4n - 1*, and *4n* 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 vertexes presented after the first pair. Vertexes *2n - 1*, *2n*, *2n+2*, and *2n+1* define quadrilateral *n*. *N* quadrilaterals are drawn. Note that the order in which vertexes are used to construct a quadrilateral

from strip data is different from that used with independent data.

GL_POLYGON

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

Only a subset of GL commands can be used between **glBegin** and **glEnd**. The commands are **glVertex**, **glColor**, **glIndex**, **glNormal**, **glTexCoord**, **glEvalCoord**, **glEvalPoint**, **glMaterial**, and **glEdgeFlag**. Also, it is acceptable to use **glCallList** or **glCallLists** to execute display lists that include only the preceding commands. If any other GL command is called between **glBegin** and **glEnd**, 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 vertexes that can be defined between **glBegin** and **glEnd**. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertexes are provided to specify even a single primitive or when an incorrect multiple of vertexes is specified. The incomplete primitive is ignored; the rest are drawn.

The minimum specification of vertexes 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 vertexes are **GL_LINES (2), GL_TRIANGLES (3)**, **GL_QUADS (4),** and **GL_QUAD_STRIP (2)**.

Errors

GL_INVALID_ENUM is generated if *mode* is set to an unaccepted value.

GL_INVALID_OPERATION is generated if a command other than **glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, glCallList,** or **glCallLists** is called between **glBegin** and the corresponding **glEnd**.

GL_INVALID_OPERATION is generated if **glEnd** is called before the corresponding **glBegin** is called, or if **glBegin** is called within a **glBegin**/**glEnd** sequence.

See Also

glCallList, **glCallLists**, **glColor**, **glEdgeFlag**, **glEvalCoord**, **glEvalPoint**, **glIndex**, **glMaterial**, **glNormal**, **glTexCoord**, **glVertex**

glBitmap

The **glBitmap** function draws a bitmap.

void glBitmap(
  GLsizei *width*,
  GLsizei *height*,
  GLfloat *xorig*,
  GLfloat *yorig*,
  GLfloat *xmove*,
  GLfloat *ymove*,
  const GLubyte **bitmap*
  );

Parameters

width, height
  Specify the pixel width and height of the bitmap image.

xorig, yorig
  Specify the location of the 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, ymove
  Specify the *x* and *y* offsets to be added to the current raster position after the bitmap is drawn.

bitmap
  Specifies the address of the bitmap image.

Remarks

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

The **glBitmap** function takes seven arguments. The first pair specify the width and height of the bitmap image. The second pair specify the location of the bitmap origin relative to the lower left corner of the bitmap image. The third pair of arguments specify *x* and *y* offsets to be added to the current raster position after the bitmap has been drawn. The final argument is a pointer to the bitmap image itself.

The bitmap image is interpreted like image data for the **glDrawPixels** command, with *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 **glPixelStore** affect the interpretation of bitmap image data; modes specified using **glPixelTransfer** do not.

If the current raster position is invalid, **glBitmap** is ignored. Otherwise, the lower left corner of the bitmap image is positioned at the window coordinates

$$x_{(w)} = \lfloor x_{(r)} - x_{(o)} \rfloor$$
$$y_{(w)} = \lfloor y_{(r)} - y_{(o)} \rfloor$$

where (*xr, yr)* is the raster position and (*xo, yo*) is the bitmap origin. Fragments are then generated for each pixel corresponding to a 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, index, or texture coordinates.

The following functions retrieve information related to the **glBitmap** function:

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_COLOR**

glGet with argument **GL_CURRENT_RASTER_INDEX**

glGet with argument **GL_CURRENT_RASTER_TEXTURE_COORDS**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**

Errors

GL_INVALID_VALUE is generated if *width* or *height* is negative.

GL_INVALID_OPERATION is generated if **glBitmap** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glDrawPixels**, **glRasterPos**, **glPixelStore**, **glPixelTransfer**

glBlendFunc

The **glBlendFunc** function specifies pixel arithmetic.

void glBlendFunc(
  GLenum *sfactor*,
  GLenum *dfactor*,
  );

Parameters

sfactor
  Specifies how the red, green, blue, and alpha source-blending factors are computed. Nine symbolic constants are accepted: **GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA,** and **GL_SRC_ALPHA_SATURATE**.

dfactor
  Specifies how the red, green, blue, and alpha destination blending factors are computed. Eight symbolic constants are accepted: **GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA,** and **GL_ONE_MINUS_DST_ALPHA.**

Remarks

In RGB 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). By default, blending is disabled. Use **glEnable** and **glDisable** with argument **GL_BLEND** to enable and disable blending.

The **glBlendFunc** function defines the operation of blending when it is enabled. The *sfactor* parameter specifies which of nine methods is used to scale the source color components. The *dfactor* parameter specifies which of eight methods is used to scale the destination color components. The eleven possible methods are described in the table below. 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 ($R_{(s)}, G_{(s)}, B_{(s)}, A_{(s)}$) and ($R_{(d)}, G_{(d)}, B_{(d)}, A_{(d)}$). They are understood to have integer values between zero and ($k_{(R)}, k_{(G)}, k_{(B)}, k_{(A)}$), where

$$k_{(c)} = 2^{m_{(c)}} - 1$$

and ($m_{(R)}, m_{(G)}, m_{(B)}, m_{(A)}$) is the number of red, green, blue, and alpha bitplanes.

Source and destination scale factors are referred to as ($s_{(R)}, s_{(G)}, s_{(B)}, s_{(A)}$) and ($d_{(R)}, d_{(G)}, d_{(B)}, d_{(A)}$). The scale factors described in the table, denoted ($f_{(R)}, f_{(G)}, f_{(B)}, f_{(A)}$), represent either source or destination factors. All scale factors have range [0,1].

Parameter	***(f (R) ,f (G) ,f (B) ,f (A))***
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	($R_{(s)}/k_{(R)}, G_{(s)}/k_{(G)}, B_{(s)}/k_{(B)}, A_{(s)}/k_{(A)}$)
GL_ONE_MINUS_SRC_COLOR	(1,1,1,1). ($R_{(s)}/k_{(R)}, G_{(s)}/k_{(G)}, B_{(s)}/k_{(B)}, A_{(s)}/k_{(A)}$)
GL_DST_COLOR	($R_{(d)}/k_{(R)}, G_{(d)}/k_{(G)}, B_{(d)}/k_{(B)}, A_{(d)}/k_{(A)}$)
GL_ONE_MINUS_DST_COLOR	(1,1,1,1)
GL_SRC_ALPHA	($R_{(d)}/k_{(R)}, G_{(d)}/k_{(G)}, B_{(d)}/k_{(B)}, A_{(d)}/k$

	$_{(A)}$) ($A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$)
GL_ONE_MINUS_SRC_ALPHA	(1,1,1,1) ($A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$,$A_{(s)}/k_{(A)}$)
GL_DST_ALPHA	($A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$)
GL_ONE_MINUS_DST_ALPHA	(1,1,1,1) ($A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$,$A_{(d)}/k_{(A)}$)
GL_SRC_ALPHA_SATURATE	($i,i,i,$1)

In the table,

$$i = \min(A_{(s)}, k_{(A)} - A_{(d)}) / kA$$

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

$$R_{(d)} = \min(kR, RssR+RddR)$$
$$G_{(d)} = \min(kG, GssG+GddG)$$
$$B_{(d)} = \min(kB, BssB+BddB)$$
$$A_{(d)} = \min(kA, AssA+AddA)$$

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 one is guaranteed not to modify its multiplicand, and a blend factor equal to zero reduces its multiplicand to zero. Thus, for example, when *sfactor* is **GL_SRC_ALPHA**, *dfactor* is **GL_ONE_MINUS_SRC_ALPHA**, and $A_{(s)}$ is equal to $k_{(A)}$, the equations reduce to simple replacement:

$$R_{(d)} = R_{(s)}$$
$$G_{(d)} = G_{(s)}$$
$$B_{(d)} = B_{(s)}$$
$$A_{(d)} = A_{(s}$$

) Examples

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 **glEnable** and the **GL_POLYGON_SMOOTH** argument for information on polygon antialiasing.) 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 ($K_{(A)}$), representing complete opacity, to 0.0 (0), representing complete transparency.

When more than one color buffer is enabled for drawing, blending is done separately for each enabled buffer, using for destination color the contents of that buffer. (See **glDrawBuffer**.)

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

The following functions retrieve information related to the **glBlendFunc** function:

glGet with argument **GL_BLEND_SRC**

glGet with argument **GL_BLEND_DST**

glIsEnabled with argument **GL_BLEND**

Errors

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

GL_INVALID_OPERATION is generated if **glBlendFunc** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBegin**, **glClear**, **glDrawBuffer**, **glEnable**, **glLogicOp**, **glStencilFunc**

glCallLists

The **glCallLists** function executes a list of display lists.

void glCallLis0s(
  GLsizei *n*,
  GLenum *type*,
  const GLvoid **lists*
  );

Parameters

n
  Specifies the number of display lists to be executed.

type
  Specifies the type of values in *lists*. Symbolic constants **GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT**, **GL_2_BYTES**, **GL_3_BYTES**, and **GL_4_BYTES** are accepted.

lists
  Specifies the address of an array of name offsets in the display list. The pointer type is void because the offsets can be bytes, shorts, ints, or floats, depending on the value of *type*.

Remarks

The **glCallLists** function 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.

The **glCallLists** function provides an efficient means for executing display lists.The *n* parameter allows lists with various name formats to be accepted. The formats are as follows:

GL_BYTE
  The *lists* parameter is treated as an array of signed bytes, each in the range - 128 through 127.

GL_UNSIGNED_BYTE
  The *lists* parameter is treated as an array of unsigned bytes, each in the range 0 through 255.

GL_SHORT
  The *lists* parameter is treated as an array of signed two-byte integers, each in the range - 32768 through 32767.

GL_UNSIGNED_SHORT
  The *lists* parameter is treated as an array of unsigned two-byte integers, each in the range 0 through 65535.

GL_INT
  The *lists* parameter is treated as an array of signed four-byte integers.

GL_UNSIGNED_INT
  The *lists* parameter is treated as an array of unsigned four-byte integers.

GL_FLOAT
  The *lists* parameter is treated as an array of four-byte floating-point values.

GL_2_BYTES
  The *lists* parameter is treated as an array of unsigned bytes. Each pair of bytes specifies a single display-list name. The value of the pair is computed as 256 times the unsigned value of the first byte plus the unsigned value of the second byte.

GL_3_BYTES
  The *lists* parameter is treated as an array of unsigned bytes. Each triplet of bytes specifies a single display-list name. The value of the triplet is computed as 65536 times the unsigned value of the first byte, plus 256 times the unsigned value of the second byte, plus the unsigned value of the third

byte.

GL_4_BYTES

The *lists* parameter is treated as an array of unsigned bytes. Each quadruplet of bytes specifies a single display-list name. The value of the quadruplet is computed as 16777216 times the unsigned value of the first byte, plus 65536 times the unsigned value of the second byte, plus 256 times the unsigned value of the third byte, plus the unsigned value of the fourth byte.

The list of display list names is not null-terminated. Rather, *n* specifies how many names are to be taken from *lists*.

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

The **glCallLists** function 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 must be at least 64, and it depends on the implementation.

GL state is not saved and restored across a call to **glCallLists**. Thus, changes made to GL state during the execution of the display lists remain after execution is completed. Use **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix** to preserve GL state across **glCallLists** 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.

The following functions retrieve information related to the **glCallLists** function:

glGet with argument **GL_LIST_BASE**

glGet with argument **GL_MAX_LIST_NESTING**

glIsList

See Also

glCallList, **glDeleteLists**, **glGenLists**, **glListBase**, **glNewList**, **glPushAttrib**, **glPushMatrix**

glCallList

The **glCallList** function executes a display list.

void glCallList(
  GLuint *list*
  );

Parameters

list
  Specifies the integer name of the display list to be executed.

Remarks

The **glCallList** function 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, **glCallList** is ignored.

The **glCallList** function 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 **glCallList**. Thus, changes made to GL state during the execution of a display list remain after execution of the display list is completed. Use **glPushAttrib**, **glPopAttrib**, **glPushMatrix**, and **glPopMatrix** to preserve GL state across **glCallList** 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.

The following functions retrieve information related to the **glCallList** function:

glGet with argument **GL_MAX_LIST_NESTING**

glIsList

See Also

glBegin, **glCallLists**, **glDeleteLists**, **glGenLists**, **glNewList**, **glPushAttrib**, **glPushMatrix**

glClear

The **glClear** function clears buffers within the viewport.

void glClear(
  GLbitfield *mask*
  );

Parameters

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**.

Remarks

The **glClear** function sets the bitplane area of the window to values previously selected by **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil**, and **glClearAccum**. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using **glDrawBuffer**.

The pixel ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of **glClear**. The scissor box bounds the cleared region. Alpha function, blend function, logical operation, stenciling, texture mapping, and *z*-buffering are ignored by **glClear**.

The **glClear** function 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 **glClear** call directed at that buffer has no effect.

The following functions retrieve information related to the **glClear** function:

glGet with argument **GL_ACCUM_CLEAR_VALUE**

glGet with argument **GL_DEPTH_CLEAR_VALUE**

glGet with argument **GL_INDEX_CLEAR_VALUE**

glGet with argument **GL_COLOR_CLEAR_VALUE**

glGet with argument **GL_STENCIL_CLEAR_VALUE**

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 **glClear** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClearAccum, glClearColor, glClearDepth, glClearIndex, glClearStencil, glDrawBuffer, glScissor

glClearAccum

The **glClearAccum** function clears values for the accumulation buffer.

void glClearAccum(
  GLfloat *red*,
  GLfloat *green*,
  GLfloat *blue*,
  GLfloat *alpha*
  );

Parameters

red, green, blue, alpha
  Specify the red, green, blue, and alpha values used when the accumulation buffer is cleared. The default values are all zero.

Remarks

The **glClearAccum** function specifies the red, green, blue, and alpha values used by **glClear** to clear the accumulation buffer.

Values specified by **glClearAccum** are clamped to the range [- 1,1].

The following function retrieves information related to the **glClearAccum** function:

glGet with argument **GL_ACCUM_CLEAR_VALUE**

Errors

GL_INVALID_OPERATION is generated if **glClearAccum** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClear

glClearColor

The **glClearColor** function specifies clear values for the color buffers.

void glClearColor(
  GLclampf *red*,
  GLclampf *green*,
  GLclampf *blue*,
  GLclampf *alpha*
  );

Parameters

red, green, blue, alpha
  Specify the red, green, blue, and alpha values used when the color buffers are cleared. The default values are all zero.

Remarks

The **glClearColor** function specifies the red, green, blue, and alpha values used by **glClear** to clear the color buffers. Values specified by **glClearColor** are clamped to the range [0,1].

The following functions retrieve information related to the **glClearColor** function:

glGet with argument **GL_ACCUM_CLEAR_VALUE**

glGet with argument **GL_COLOR_CLEAR_VALUE**

Errors

GL_INVALID_OPERATION is generated if **glClearColor** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClear

glClearDepth

The **glClearDepth** function specifies the clear value for the depth buffer.

void glClearDepth(
  GLclampd *depth*
  );

Parameters

depth
  Specifies the depth value used when the depth buffer is cleared.

Remarks

The **glClearDepth** function specifies the depth value used by **glClear** to clear the depth buffer. Values specified by **glClearDepth** are clamped to the range [0,1].

The following function retrieves information related to the **glClearDepth** function:

glGet with argument **GL_DEPTH_CLEAR_VALUE**

Errors

GL_INVALID_OPERATION is generated if **glClearDepth** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClear

glClearIndex

The **glClearIndex** function specifies the clear value for the color index buffers.

void glClearIndex(
  GLfloat *c*
  );

Parameters

c
  Specifies the index used when the color index buffers are cleared. The default value is zero.

Remarks

The **glClearIndex** function specifies the index used by **glClear** 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.

The following functions retrieve information related to the **glClearIndex** function:

glGet with argument **GL_INDEX_CLEAR_VALUE**

glGet with argument **GL_INDEX_BITS**

Errors

GL_INVALID_OPERATION is generated if **glClearIndex** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClear

glClearStencil

The **glClearStencil** function specifies the clear value for the stencil buffer.

void glClearStencil(
  GLint *s*
  );

Parameters

s
  Specifies the index used when the stencil buffer is cleared. The default value is zero.

Remarks

The **glClearStencil** function specifies the index used by **glClear** to clear the stencil buffer. The *s* parameter is masked with 2^*m* - 1, where *m* is the number of bits in the stencil buffer.

The following functions retrieve information related to the **glCearStencil** function:

glGet with argument **GL_STENCIL_CLEAR_VALUE**

glGet with argument **GL_STENCIL_BITS**

Errors

GL_INVALID_OPERATION is generated if **glClearStencil** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClear

glClipPlane

The **glClipPlane** function specifies a plane against which all geometry is clipped.

void glClipPlane(
  GLenum *plane*,
  const GLdouble **equation*
  );

Parameters

plane
  Specifies which clipping plane is being positioned. Symbolic names of the form **GL_CLIP_PLANE***i*, where *i* is an integer between 0 and **GL_MAX_CLIP_PLANES** - 1, are accepted.

equation
  Specifies the address of an array of four double-precision floating-point values. These values are interpreted as a plane equation.

Remarks

Geometry is always clipped against the boundaries of a six-plane frustum in *x*, *y*, and *z*. **glClipPlane** allows the specification of additional planes, not necessarily perpendicular to the *x*, *y*, or *z* axis, against which all geometry is clipped. Up to **GL_MAX_CLIP_PLANES** planes can be specified, where **GL_MAX_CLIP_PLANES** is at least six in all implementations. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

The **glClipPlane** function specifies a half-space using a four-component plane equation. When **glClipPlane** 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*.

Clipping planes are enabled and disabled with **glEnable** and **glDisable**, and called with the argument **GL_CLIP_PLANE***i*, where *i* is the plane number.

By default, all clipping planes are defined as (0,0,0,0) in eye coordinates and are disabled.

It is always the case that **GL_CLIP_PLANE***i* = **GL_CLIP_PLANE0** + *i*.

The following functions retrieve information related to the **glClipPlane** function:

glGetClipPlane

glIsEnabled with argument **GL_CLIP_PLANE***i*

Errors

GL_INVALID_ENUM is generated if *plane* is not an accepted value.

GL_INVALID_OPERATION is generated if **glClipPlane** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable

glColor

glColor3b, glColor3d, glColor3f, glColor3i, glColor3s, glColor3ub, glColor3ui, glColor3us, glColor4b, glColor4d, glColor4f, glColor4i, glColor4s, glColor4ub, glColor4ui, glColor4us, glColor3bv, glColor3dv, glColor3fv, glColor3iv, glColor3sv, glColor3ubv, glColor3uiv, glColor3usv, glColor4bv, glColor4dv, glColor4fv, glColor4iv, glColor4sv, glColor4ubv, glColor4uiv, glColor4usv

These functions set the current color.

void glColor3b(
GLbyte *red*,
GLbyte *green*,
GLbyte *blue*
);

void glColor3d(
GLdouble *red*,
GLdouble *green*,
GLdouble *blue*
);

void glColor3f(
GLfloat *red*,
GLfloat *green*,
GLfloat *blue*
);

void glColor3i(
GLint *red*,
GLint *green*,
GLint *blue*
);

void glColor3s(
GLshort *red*,
GLshort *green*,
GLshort *blue*
);

void glColor3ub(
GLubyte *red*,
GLubyte *green*,
GLubyte *blue*
);

void glColor3ui(
GLuint *red*,
GLuint *green*,
GLuint *blue*
);

void glColor3us(
GLushort *red*,
GLushort *green*,
GLushort *blue*
);

void glColor4b(

GLbyte *red*,
GLbyte *green*,
GLbyte *blue*,
GLbyte *alpha*
);

void glColor4d(
GLdouble *red*,
GLdouble *green*,
GLdouble *blue*,
GLdouble *alpha*
);

void glColor4f(
GLfloat *red*,
GLfloat *green*,
GLfloat *blue*,
GLfloat *alpha*
);

void glColor4i(
GLint *red*,
GLint *green*,
GLint *blue*,
GLint *alpha*
);

void glColor4s(
GLshort *red*,
GLshort *green*,
GLshort *blue*,
GLshort *alpha*
);

void glColor4ub(
GLubyte *red*,
GLubyte *green*,
GLubyte *blue*,
GLubyte *alpha*
);

void glColor4ui(
GLuint *red*,
GLuint *green*,
GLuint *blue*,
GLuint *alpha*
);

void glColor4us(
GLushort *red*,
GLushort *green*,
GLushort *blue*,
GLushort *alpha*
);

Parameters

red, green, blue
Specify new red, green, and blue values for the current color.

alpha
Specifies a new alpha value for the current color. Included only in the four-argument **glColor4** command.

void glColor3bv(
const GLbyte *v
);

void glColor3dv(
const GLdouble *v
);

void glColor3fv(
const GLfloat *v
);

void glColor3iv(
const GLint *v
);

void glColor3sv(
const GLshort *v
);

void glColor3ubv(
const GLubyte *v
);

void glColor3uiv(
const GLuint *v
);

void glColor3usv(
const GLushort *v
);

void glColor4bv(
const GLbyte *v
);

void glColor4dv(
const GLdouble *v
);

void glColor4fv(
const GLfloat *v
);

void glColor4iv(
const GLint *v
);

void glColor4sv(
const GLshort *v
);

void glColor4ubv(
const GLubyte *v
);

void glColor4uiv(

const GLuint *v
);

void glColor4usv(
const GLushort *v
);

Parameters

v
Specifies a pointer to an array that contains red, green, blue, and (sometimes) alpha values.

Remarks

The GL stores both a current single-valued color index and a current four-valued RGBA color. **glColor** sets a new four-valued RGBA color. **glColor** has two major variants: **glColor3 and glColor4. glColor3** variants specify new red, green, and blue values explicitly, and set the current alpha value to 1.0 implicitly. **glColor4** variants specify all four color components explicitly.

The **glColor3b**, **glColor4b**, **glColor3s**, **glColor4s**, **glColor3i**, and **glColor4i** functions take three or four signed byte, short, or long integers as arguments. When *v* is appended to the name, the color commands can take a pointer to an array of such values.

Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to - 1.0. Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0,1] before updating the current color. However, color components are clamped to this range before they are interpolated or written into a color buffer.

The current color can be updated at any time. In particular, **glColor** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following functions retrieve information related to the **glColor** functions:

glGet with argument **GL_CURRENT_COLOR**

glGet with argument **GL_RGBA_MODE**

See Also

glIndex

glColorMask

The **glColorMask** function enables and disables writing of frame buffer color components.

void glColorMask(
  GLboolean *red*,
  GLboolean *green*,
  GLboolean *blue*,
  GLboolean *alpha*
  );

Parameters

red, green, blue, alpha
  Specify whether red, green, blue, and alpha can or cannot be written into the frame buffer. The default values are all **GL_TRUE**, indicating that the color components can be written.

Remarks

The **glColorMask** function 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.

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

The following functions retrieve information related to the **glColorMask** function:

glGet with argument **GL_COLOR_WRITEMASK**

glGet with argument **GL_RGBA_MODE**

Errors

GL_INVALID_OPERATION is generated if **glColorMask** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColor, **glIndex**, **glIndexMask**, **glDepthMask**, **glStencilMask**

glColorMaterial

The **glColorMaterial** function causes a material color to track the current color.

void glColorMaterial(
GLenum *face*,
GLenum *mode*
);

Parameters

face
Specifies whether front, back, or both front and back material parameters should track the current color. Accepted values are **GL_FRONT**, **GL_BACK**, and **GL_FRONT_AND_BACK**. The default value is **GL_FRONT_AND_BACK**.

mode
Specifies which of several material parameters track the current color. Accepted values are **GL_EMISSION**, **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, and **GL_AMBIENT_AND_DIFFUSE**. The default value is **GL_AMBIENT_AND_DIFFUSE**.

Remarks

The **glColorMaterial** function 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. **GL_COLOR_MATERIAL** is enabled and disabled using the commands **glEnable** and **glDisable**, called with **GL_COLOR_MATERIAL** as their argument. By default, it is disabled.

The **glColorMaterial** function allows a subset of material parameters to be changed for each vertex using only the **glColor** command, without calling **glMaterial**. If only such a subset of parameters is to be specified for each vertex, **glColorMaterial** is preferred over calling **glMaterial**.

The following functions retrieve information related to the **glColorMaterial** function:

glGet with argument **GL_COLOR_MATERIAL_PARAMETER**

glGet with argument **GL_COLOR_MATERIAL_FACE**

glIsEnabled with argument **GL_COLOR_MATERIAL**

Errors

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

GL_INVALID_OPERATION is generated if **glColorMaterial** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColor, **glEnable**, **glLight**, **glLightModel**, **glMaterial**

glCopyPixels

The **glCopyPixels** function copies pixels in the frame buffer.

void glCopyPixels(
GLint *x*,
GLint *y*,
GLsizei *width*,
GLsizei *height*,
GLenum *type*
);

Parameters

x, y
Specify the window coordinates of the lower left corner of the rectangular region of pixels to be copied.

width, 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.

Remarks

The **glCopyPixels** function 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.

The *x* and *y* parameters 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: **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This topic describes the effects on **glCopyPixels** of most, but not all, of the parameters specified by these three commands.

The **glCopyPixels** function copies values from each pixel with the lower left-hand corner at (*x* + i, *y* + *j*) for $0 \leq i < width$ and $0 \leq j < height$. This pixel is said to be the *i*th pixel in the *j*th row. Pixels are copied in row order from the lowest to the highest row, left to right in each row.

The *type* parameter 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 **glReadBuffer**). 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 zero 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**, respectively.

The resulting indices or RGBA colors are then converted to fragments by attaching the current raster position *z* coordinate and texture coordinates to each pixel, then assigning window coordinates ($x_r + i$, $y_r + j$), where (x_r y_r) 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 resulting depth components are then converted to fragments by attaching the current raster position color or color index and texture coordinates to each pixel, then assigning window coordinates ($x_r + i$, $y_r + j$), where (x_r , y_r) 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 *i*th location of the *j*th row is written to location ($x_r + i$, $y_r + j$), where (x_r , y_r) is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these writes.

The rasterization described thus far assumes pixel zoom factors of 1.0. If **glPixelZoom** is used to change the *x* and *y* pixel zoom factors, pixels are converted to fragments as follows. If (x_r , y_r) is the current raster position, and a given pixel is in the *i*th location in the *j*th row of the source pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$$(x_r + zoom_x\ i,\ y_r + zoom_y\ j)$$

and

$$(x_r + zoom_x\ (i + 1),\ y_r + zoom_y\ (j + 1))$$

where $zoom_x$ is the value of **GL_ZOOM_X** and $zoom_y$ is the value of **GL_ZOOM_Y**.

Modes specified by **glPixelStore** have no effect on the operation of **glCopyPixels**.

The following functions retrieve information related to the **glCopyPixels** function:

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**

To copy the color pixel in the lower left corner of the window to the current raster position, use

glCopyPixels(0, 0, 1, 1, **GL_COLOR**);

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 **glCopyPixels** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDepthFunc, **glDrawBuffer**, **glDrawPixels**, **glPixelMap**, **glPixelTransfer**, **glPixelZoom**, **glRasterPos**, **glReadBuffer**, **glReadPixels**, **glStencilFunc**

glCullFace

The **glCullFace** function specifies whether front- or back-facing facets can be culled.

void glCullFace(
  GLenum *mode*
  );

Parameters

mode
  Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants **GL_FRONT** and **GL_BACK** are accepted. The default value is **GL_BACK**.

Remarks

The **glCullFace** function specifies whether front- or back-facing facets are culled (as specified by *mode*) when facet culling is enabled. Facet culling is enabled and disabled using the **glEnable** and **glDisable** commands with the argument **GL_CULL_FACE**. Facets include triangles, quadrilaterals, polygons, and rectangles.

The **glFrontFace** function specifies which of the clockwise and counterclockwise facets are front-facing and back-facing.

The following functions retrieve information related to the **glCullFace** function:

glGet with argument **GL_CULL_FACE_MODE**

glIsEnabled with argument **GL_CULL_FACE**

Errors

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

GL_INVALID_OPERATION is generated if **glCullFace** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable, **glFrontFace**

glDeleteLists

The **glDeleteLists** function deletes a contiguous group of display lists.

void glDeleteLists(
  GLuint *list*,
  GLsizei *range*
  );

Parameters

list
  Specifies the integer name of the first display list to delete.

range
  Specifies the number of display lists to delete.

Remarks

The **glDeleteLists** function causes a contiguous group of display lists to be deleted. The *list* parameter 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* $\leq$ *d* $\leq$ *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 zero, nothing happens.

Errors

GL_INVALID_VALUE is generated if *range* is negative.

GL_INVALID_OPERATION is generated if **glDeleteLists** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallList, **glCallLists**, **glGenLists**, **glIsList**, **glNewList**

glDepthFunc

The **glDepthFunc** function specifies the value used for depth buffer comparisons.

void glDepthFunc(
  GLenum *func*
  );

Parameters

func
  Specifies the depth comparison function. Symbolic constants **GL_NEVER**, **GL_LESS**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS** are accepted. The default value is **GL_LESS**.

Remarks

The **glDepthFunc** function specifies the function used to compare each incoming pixel *z* value with the *z* value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See **glEnable** of **GL_DEPTH_TEST**.)

The *func* parameter 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 *z* value is less than the stored *z* value.

GL_EQUAL
  Passes if the incoming *z* value is equal to the stored *z* value.

GL_LEQUAL
  Passes if the incoming *z* value is less than or equal to the stored *z* value.

GL_GREATER
  Passes if the incoming *z* value is greater than the stored *z* value.

GL_NOTEQUAL
  Passes if the incoming *z* value is not equal to the stored *z* value.

GL_GEQUAL
  Passes if the incoming *z* value is greater than or equal to the stored *z* value.

GL_ALWAYS
  Always passes.

The default value of *func* is **GL_LESS**. Initially, depth testing is disabled.

The following functions retrieve information related to the **glDepthFunc** function:

glGet with argument **GL_DEPTH_FUNC**

glIsEnabled with argument **GL_DEPTH_TEST**

Errors

GL_INVALID_ENUM is generated if *func* is not an accepted value.

GL_INVALID_OPERATION is generated if **glDepthFunc** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDepthRange, glEnable

glDepthMask

The **glDepthMask** function enables or disables writing into the depth buffer.

void glDepthMask(
  GLboolean *flag*
  );

Parameters

flag
  Specifies whether the depth buffer is enabled for writing. If *flag* is zero, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

Remarks

The **glDepthMask** function specifies whether the depth buffer is enabled for writing. If *flag* is zero, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

The following function retrieves information related to the **glDepthMask** function:

glGet with argument **GL_DEPTH_WRITEMASK**

Errors

GL_INVALID_OPERATION is generated if **glDepthMask** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColorMask, **glDepthFunc**, **glDepthRange**, **glIndexMask**, **glStencilMask**

glDepthRange

The **glDepthRange** function specifies the mapping of *z*values from normalized device coordinates to window coordinates.

void glDepthRange(
  GLclampd *near*,
  GLclampd *far*
  );

Parameters

near
  Specifies the mapping of the near clipping plane to window coordinates. The default value is 0.

far
  Specifies the mapping of the far clipping plane to window coordinates. The default value is 1.

Remarks

After clipping and division by *w*, *z* coordinates range from -1.0 to 1.0, corresponding to the near and far clipping planes. The **glDepthRange** function specifies a linear mapping of the normalized *z* coordinates in this range to window *z* coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0.0 through 1.0 (like color components). Thus, the values accepted by **glDepthRange** are both clamped to this range before they are accepted.

The default mapping 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 *near* be less than *far*. Reverse mappings such as 1,0 are acceptable.

The following function retrieves information related to the **glDepthRange** function:

glGet with argument **GL_DEPTH_RANGE**

Errors

GL_INVALID_OPERATION is generated if **glDepthRange** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDepthFunc, **glViewport**

glDrawBuffer

The **glDrawBuffer** function specifies which color buffers are to be drawn into.

void glDrawBuffer(
  GLenum *mode*
  );

Parameters

mode
  Specifies up to four color buffers to be drawn into. Symbolic constants **GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_AND_BACK**, and **GL_AUX*i***, where *i* is between 0 and **GL_AUX_BUFFERS** -1, are accepted. (**GL_AUX_BUFFERS** is not the upper limit; use **glGet** to query the number of available aux buffers.) The default value is **GL_FRONT** for single-buffered contexts, and **GL_BACK** for double-buffered contexts.

Remarks

When colors are written to the frame buffer, they are written into the color buffers specified by **glDrawBuffer**. The specifications are as follows:

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_AUX*i*
  Only auxiliary color buffer *i* is written.

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.

It is always the case that **GL_AUX***i* = **GL_AUX0** + *i*.

The following functions retrieve information related to the **glDrawBuffer** function:

glGet with argument **GL_DRAW_BUFFER**

glGet with argument **GL_AUX_BUFFERS**

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 **glDrawBuffer** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBlendFunc, **glColorMask**, **glIndexMask**, **glLogicOp**, **glReadBuffer**

glDrawPixels

The **glDrawPixels** function writes a block of pixels to the frame buffer.

void glDrawPixels(
  GLsizei *width*,
  GLsizei *height*,
  GLenum *format*,
  GLenum *type*,
  const GLvoid **pixels*
  );

Parameters

width, height
  Specify the dimensions of the pixel rectangle that will be written into the frame buffer.

format
  Specifies the format of the pixel data. Symbolic constants **GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA**, **GL_RGB**, **GL_LUMINANCE**, and **GL_LUMINANCE_ALPHA** are accepted.

type
  Specifies the data type for *pixels*. Symbolic constants **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT**, **GL_INT**, and **GL_FLOAT** are accepted.

pixels
  Specifies a pointer to the pixel data.

Remarks

The **glDrawPixels** function reads pixel data from memory and writes it into the frame buffer relative to the current raster position. Use **glRasterPos** to set the current raster position, and use **glGet** 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: **glPixelStore**, **glPixelTransfer**, **glPixelMap**, and **glPixelZoom**. This topic describes the effects on **glDrawPixels** 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*. 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 **glPixelStore**).

The *width* x *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 **glPixelStore** 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.

The *width* x *height* pixels that are read from memory are each operated on in the same way, based on the values of several parameters specified by **glPixelTransfer** and **glPixelMap**. 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 eleven 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. Signed and unsigned integer data is converted with all fraction bits set to zero. Bitmap data convert to either 0.0 or 1.0.

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 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. 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 resulting indices or RGBA colors are then converted 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

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$$

where $(x_{(r)}, y_{(r)})$ 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. Signed and unsigned integer data is converted with all fraction bits set to zero. Bitmap data convert to either 0.0 or 1.0.

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

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n/width \rfloor$$

where $(x_{(r)}, y_{(r)})$ is the current raster position. Only the pixel ownership test, the scissor test, and the stencil writemask affect these writes.

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. Signed integer data is mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to -1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps to 0.0. 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 resulting depth components are then converted 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

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n / width \rfloor$$

where $(x_{(r)}, y_{(r)})$ 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: red 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. Signed integer values are mapped linearly to the internal floating-point format such that the most positive representable integer value maps to 1.0, and the most negative representable value maps to
- 1.0. Unsigned integer data is mapped similarly: the largest integer value maps to 1.0, and zero maps 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**, respectively.

The resulting RGBA colors are then converted 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

$$x_{(n)} = x_{(r)} + n \bmod width$$

$$y_{(n)} = y_{(r)} + \lfloor n / width \rfloor$$

where $(x_{(r)}, y_{(r)})$ 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 as the red component of an RGBA pixel is, then it is converted to an RGBA pixel with green and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just 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 as the green component of an RGBA pixel is, then it is converted to an RGBA pixel with red and blue set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just 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 as the blue component of an RGBA pixel is, then it is converted to an RGBA pixel with red and green set to 0.0, and alpha set to 1.0. After this conversion, the pixel is treated just 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 as the alpha component of an RGBA pixel is, then it is converted to an RGBA pixel with red, green, and blue set to 0.0. After this conversion, the pixel is treated just 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 as the red, green, and blue components of an RGBA pixel are. The color triple is converted to an RGBA pixel with alpha set to 1.0. After this conversion, the pixel is treated just 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 as the red component of an RGBA pixel is, then it is converted to an RGBA pixel with red, green, and blue set to the converted luminance value, and alpha set to 1.0. After this conversion, the pixel is treated just 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 as the red component of an RGBA pixel is, then they are 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 just as if it had been read as an RGBA pixel.

The following table summarizes the meaning of the valid constants for the *type* parameter:

Type	Corresponding Type
GL_UNSIGNED_BYTE	unsigned 8-bit integer
GL_BYTE	signed 8-bit integer
GL_BITMAP	single bits in unsigned 8-bit integers
GL_UNSIGNED_SHORT	unsigned 16-bit integer
GL_SHORT	signed 16-bit integer
GL_UNSIGNED_INT	unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	single-precision floating-point

The rasterization described thus far assumes pixel zoom factors of 1.0. If **glPixelZoom** is used to change the x and y pixel zoom factors, pixels are converted to fragments as follows. If $(x_{(r)}, y_{(r)})$ is the current raster position, and a given pixel is in the *n*th column and mth row of the pixel rectangle, then fragments are generated for pixels whose centers are in the rectangle with corners at

$$(x_{(r)} + zoom_{(x)}\ n,\ y_{(r)} + zoom_{(y)}\ m)$$

$$(x_{(r)} + zoom_{(x)}\ (n + 1),\ y_{(r)} + zoom_{(y)}\ (m + 1))$$

where $zoom_{(x)}$ is the value of **GL_ZOOM_X** and $zoom_{(y)}$ is the value of **GL_ZOOM_Y**.

The following functions retrieve information related to the **glDrawPixels** function:

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**

Errors

GL_INVALID_VALUE is generated if either *width* or *height* is negative.

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

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_ENUM is generated if *type* is **GL_BITMAP** and *format* is not either **GL_COLOR_INDEX** or **GL_STENCIL_INDEX**.

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

GL_INVALID_OPERATION is generated if **glDrawPixels** is called between a call to **glBegin** and the

corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBlendFunc**, **glCopyPixels**, **glDepthFunc**, **glLogicOp**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glPixelZoom**, **glRasterPos**, **glReadPixels**, **glScissor**, **glStencilFunc**

glEdgeFlag, glEdgeFlagv

The **glEdgeFlag** and **glEdgeFlagv** functions flag edges as either boundary or nonboundary.

void glEdgeFlag(
  GLboolean *flag*
  );

Parameters

flag
  Specifies the current edge flag value, either true or false.

void glEdgeFlagv(
  const GLboolean **flag*
  );

Parameters

flag
  Specifies a pointer to an array that contains a single Boolean element, which replaces the current edge flag value.

Remarks

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a **glBegin**/**glEnd** 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. The **glEdgeFlag** function sets the edge flag to true if *flag* is nonzero, false otherwise.

The vertexes 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 vertexes are significant only if **GL_POLYGON_MODE** is set to **GL_POINT** or **GL_LINE**. See **glPolygonMode**.

Initially, the edge flag bit is true.

The current edge flag can be updated at any time. In particular, **glEdgeFlag** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to the **glEdgeFlag** function:

glGet with argument **GL_EDGE_FLAG**

See Also

glBegin, **glPolygonMode**

glEnable, glDisable

The **glEnable** and **glDisable** functions enable or disable GL capabilities.

void glEnable(
  GLenum *cap*
  );

Parameters

cap
  Specifies a symbolic constant indicating a GL capability.

void glDisable(
  GLenum *cap*
  );

Parameters

cap
  Specifies a symbolic constant indicating a GL capability.

Remarks

The **glEnable** and **glDisable** functions enable and disable various capabilities. Use **glIsEnabled** or **glGet** to determine the current setting of any capability.

Both the **glEnable** and **glDisable** functions take a single argument, *cap*, which can assume one of the following values:

GL_ALPHA_TEST
  If enabled, do alpha testing. See **glAlphaFunc**.

GL_AUTO_NORMAL
  If enabled, compute surface normal vectors analytically when either **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4** is used to generate vertexes. See **glMap2**.

GL_BLEND
  If enabled, blend the incoming RGBA color values with the values in the color buffers. See **glBlendFunc**.

GL_CLIP_PLANE*i*
  If enabled, clip geometry against user-defined clipping plane *i*. See **glClipPlane**.

GL_COLOR_MATERIAL
  If enabled, have one or more material parameters track the current color. See **glColorMaterial**.

GL_CULL_FACE
  If enabled, cull polygons based on their winding in window coordinates. See **glCullFace**.

GL_DEPTH_TEST
  If enabled, do depth comparisons and update the depth buffer. See **glDepthFunc** and **glDepthRange**.

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

GL_FOG
  If enabled, blend a fog color into the posttexturing color. See **glFog**.

GL_LIGHT*i*
  If enabled, include light *i* in the evaluation of the lighting equation. See **glLightModel** and **glLight**.

GL_LIGHTING
  If enabled, use the current lighting parameters to compute the vertex color or index. Otherwise, simply associate the current color or index with each vertex. See **glMaterial**, **glLightModel**, and **glLight**.

GL_LINE_SMOOTH
If enabled, draw lines with correct filtering. Otherwise, draw aliased lines. See **glLineWidth**.

GL_LINE_STIPPLE
If enabled, use the current line stipple pattern when drawing lines. See **glLineStipple**.

GL_LOGIC_OP
If enabled, apply the currently selected logical operation to the incoming and color buffer indices. See **glLogicOp**.

GL_MAP1_COLOR_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate RGBA values. See **glMap1**.

GL_MAP1_INDEX
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate color indices. See **glMap1**.

GL_MAP1_NORMAL
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate normals. See **glMap1**.

GL_MAP1_TEXTURE_COORD_1
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s* texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_2
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s* and *t* texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_3
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s*, *t*, and *r* texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *s*, *t*, *r*, and *q* texture coordinates. **glMap1**.

GL_MAP1_VERTEX_3
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate *x*, *y*, and *z* vertex coordinates. See **glMap1**.

GL_MAP1_VERTEX_4
If enabled, calls to **glEvalCoord1**, **glEvalMesh1**, and **glEvalPoint1** will generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See **glMap1**.

GL_MAP2_COLOR_4
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate RGBA values. See **glMap2**.

GL_MAP2_INDEX
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate color indices. See **glMap2**.

GL_MAP2_NORMAL
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate normals. See **glMap2**.

GL_MAP2_TEXTURE_COORD_1
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s* texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_2
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s* and *t* texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_3
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s*, *t*, and *r* texture

coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_4
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *s*, *t*, *r*, and *q* texture coordinates. See **glMap2**.

GL_MAP2_VERTEX_3
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate *x*, *y*, and *z* vertex coordinates. See **glMap2**.

GL_MAP2_VERTEX_4
If enabled, calls to **glEvalCoord2**, **glEvalMesh2**, and **glEvalPoint2** will generate homogeneous *x*, *y*, *z*, and *w* vertex coordinates. See **glMap2**.

GL_NORMALIZE
If enabled, normal vectors specified with **glNormal** are scaled to unit length after transformation. See **glNormal**.

GL_POINT_SMOOTH
If enabled, draw points with proper filtering. Otherwise, draw aliased points. See **glPointSize**.

GL_POLYGON_SMOOTH
If enabled, draw polygons with proper filtering. Otherwise, draw aliased polygons. See **glPolygonMode**.

GL_POLYGON_STIPPLE
If enabled, use the current polygon stipple pattern when rendering polygons. See **glPolygonStipple**.

GL_SCISSOR_TEST
If enabled, discard fragments that are outside the scissor rectangle. See **glScissor**.

GL_STENCIL_TEST
If enabled, do stencil testing and update the stencil buffer. See **glStencilFunc** and **glStencilOp**.

GL_TEXTURE_1D
If enabled, one-dimensional texturing is performed (unless two-dimensional texturing is also enabled). See **glTexImage1D**.

GL_TEXTURE_2D
If enabled, two-dimensional texturing is performed. See **glTexImage2D**.

GL_TEXTURE_GEN_Q
If enabled, the *q* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise, the current *q* texture coordinate is used.

GL_TEXTURE_GEN_R
If enabled, the *r* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise, the current *r* texture coordinate is used.

GL_TEXTURE_GEN_S
If enabled, the *s* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise, the current *s* texture coordinate is used.

GL_TEXTURE_GEN_T
If enabled, the *t* texture coordinate is computed using the texture generation function defined with **glTexGen**. Otherwise, the current *t* texture coordinate is used.

Errors

GL_INVALID_ENUM is generated if *cap* is not one of the values listed above.

GL_INVALID_OPERATION is generated if **glEnable** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBlendFunc**, **glClipPlane**, **glColorMaterial**, **glCullFace**, **glDepthFunc**,

glDepthRange, glFog, glGet, glIsEnabled, glLight, glLightModel, glLineWidth, glLineStipple, glLogicOp, glMap1, glMap2, glMaterial, glNormal, glPointSize, glPolygonMode, glPolygonStipple, glScissor, glStencilFunc, glStencilOp, glTexGen, glTexImage1D, glTexImage2D

glEvalCoord

glEvalCoord1d, glEvalCoord1f, glEvalCoord2d, glEvalCoord2f, glEvalCoord1dv, glEvalCoord1fv, glEvalCoord2dv, glEvalCoord2fv

These functions evaluate enabled one- and two-dimensional maps.

void glEvalCoord1d(
GLdouble *u*
);

void glEvalCoord1f(
GLfloat *u*
);

void glEvalCoord2d(
GLdouble *u*,
GLdouble *v*
);

void glEvalCoord2f(
GLfloat *u*,
GLfloat *v*
);

Parameters

u
Specifies a value that is the domain coordinate *u* to the basis function defined in a previous **glMap1** or **glMap2** command.

v
Specifies a value that is the domain coordinate *v* to the basis function defined in a previous **glMap2** command. This argument is not present in an **glEvalCoord1** command.

void glEvalCoord1dv(
const GLdouble **u*
);

void glEvalCoord1fv(
const GLfloat **u*
);

void glEvalCoord2dv(
const GLdouble **u*
);

void glEvalCoord2fv(
const GLfloat **u*
);

Parameters

u
Specifies a pointer to an array containing either one or two domain coordinates. The first coordinate is *u*. The second coordinate is *v*, which is present only in **glEvalCoord2** versions.

Remarks

The **glEvalCoord1** function evaluates enabled one-dimensional maps at argument *u*. **glEvalCoord2** does the same for two-dimensional maps using two domain values, *u* and *v*. Maps are defined with

glMap1 and **glMap2** and enabled and disabled with **glEnable** and **glDisable**.

When one of the **glEvalCoord** commands 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 was issued with the computed value. That is, if **GL_MAP1_INDEX** or **GL_MAP2_INDEX** is enabled, a **glIndex** command is simulated. If **GL_MAP1_COLOR_4** or **GL_MAP2_COLOR_4** is enabled, a **glColor** command is simulated. If **GL_MAP1_NORMAL** or **GL_MAP2_NORMAL** is enabled, a normal vector is produced, and if any of **GL_MAP1_TEXTURE_COORD_1**, **GL_MAP1_TEXTURE_COORD_2**, **GL_MAP1_TEXTURE_COORD_3**, **GL_MAP1_TEXTURE_COORD_4**, **GL_MAP2_TEXTURE_COORD_1**, **GL_MAP2_TEXTURE_COORD_2**, **GL_MAP2_TEXTURE_COORD_3**, or **GL_MAP2_TEXTURE_COORD_4** is enabled, then an appropriate **glTexCoord** command is simulated.

The GL uses evaluated values instead of current values for those evaluations that are enabled, and current values otherwise, for color, color index, normal, and texture coordinates. However, the evaluated values do not update the current values. Thus, if **glVertex** commands are interspersed with **glEvalCoord** commands, the color, normal, and texture coordinates associated with the **glVertex** commands are not affected by the values generated by the **glEvalCoord** commands, but rather only by the most recent **glColor**, **glIndex**, **glNormal**, and **glTexCoord** 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 four-component vertex map is enabled for the specified dimension, the **glEvalCoord** command is ignored.

If automatic normal generation is enabled, by calling **glEnable** with argument **GL_AUTO_NORMAL**, **glEvalCoord2** generates surface normals analytically, regardless of the contents or enabling of the **GL_MAP2_NORMAL** map. Let

{ewc msdncd, EWGraphic, group10282 0 /a "SDK.bmp"}

Then the generated normal **n** is

{ewc msdncd, EWGraphic, group10282 1 /a "SDK.bmp"}

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 **glEvalCoord2** commands.

The following functions retrieve information related to the **glEvalCoord** functions:

glIsEnabled with argument **GL_MAP1_VERTEX_3**

glIsEnabled with argument **GL_MAP1_VERTEX_4**

glIsEnabled with argument **GL_MAP1_INDEX**

glIsEnabled with argument **GL_MAP1_COLOR_4**

glIsEnabled with argument **GL_MAP1_NORMAL**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_1**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_2**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_3**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_4**

glIsEnabled with argument **GL_MAP2_VERTEX_3**

glIsEnabled with argument **GL_MAP2_VERTEX_4**

glIsEnabled with argument **GL_MAP2_INDEX**

glIsEnabled with argument **GL_MAP2_COLOR_4**

glIsEnabled with argument **GL_MAP2_NORMAL**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_1**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_2**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_3**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_4**

glIsEnabled with argument **GL_AUTO_NORMAL**

glGetMap

See Also

glBegin, **glColor**, **glEnable**, **glEvalMesh**, **glEvalPoint**, **glIndex**, **glMap1**, **glMap2**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glEvalMesh1, glEvalMesh2

The **glEvalMesh1** and **glEvalMesh2** functions compute a one- or two-dimensional grid of points or lines.

void glEvalMesh1(
GLenum *mode*,
GLint *i1*,
GLint *i2*
);

Parameters

mode
In **glEvalMesh1**, specifies whether to compute a one-dimensional mesh of points or lines. Symbolic constants **GL_POINT** and **GL_LINE** are accepted.

i1, i2
Specify the first and last integer values for grid domain variable *i*.

void glEvalMesh2(
GLenum *mode*,
GLint *i1*,
GLint *i2*,
GLint *j1*,
GLint *j2*
);

Parameters

mode
In **glEvalMesh2**, specifies whether to compute a two-dimensional mesh of points, lines, or polygons. Symbolic constants **GL_POINT**, **GL_LINE**, and **GL_FILL** are accepted.

i1, i2
Specify the first and last integer values for grid domain variable *i*.

j1, j2
Specify the first and last integer values for grid domain variable *j*.

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. **glEvalMesh** steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**. *mode* determines whether the resulting vertexes are connected as points, lines, or filled polygons.

In the one-dimensional case, **glEvalMesh1**, the mesh is generated as if the following code fragment were executed:

glBegin(*type*) ;
for (*i* = *i1*; *i* <= *i2*; *i* += 1)
glEvalCoord1($i \cdot \Delta u + u_{(1)}$) (
) **glEnd**( );
where
$\Delta u = (u_{(2)} - u_{(1)}) / n$
and n, $u_{(1)}$, and $u_{(2)}$ are the arguments to the most recent **glMapGrid1** 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 $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$.

In the two-dimensional case, **glEvalMesh2**, let

$$\Delta u = (u_{(2)} - u_{(1)})/n$$
$$\Delta v = (v_{(2)} - v_{(1)})/m,$$

where n, $u_{(1)}$, $u_{(2)}$, m, $v_{(1)}$, and $v_{(2)}$ are the arguments to the most recent **glMapGrid2** command. Then, if *mode* is **GL_FILL,** the **glEvalMesh2** command is equivalent to:

```
for (j = j1;   j < j2; j += 1) {
    glBegin (GL_QUAD_STRIP);
    for (i = i1; i <= i2; i += 1) {
        glEvalCoord2(i·Δ u + u(1,) j ·Δ v + v(1) );
        glEvalCoord2(i·Δ u + u(1,)  (j+1) ·Δ v + v(1) );
    }
    glEnd( );
}
```

If *mode* is **GL_LINE**, then a call to **glEvalMesh2** is equivalent to:

```
for (j = j1; j <= j2; j += 1) {
    glBegin (GL_LINE_STRIP);
    for (i = i1; i <= i2; i += 1)
        glEvalCoord2(i·Δ u + u(1) , j·Δ v + v(1);)
    glEnd( );
}
    for (i = i1;   i <= i2; i += 1) {
    glBegin (GL_LINE_STRIP);
    for (j = j1; j <= j1; j += 1)
        glEvalCoord2(i·Δ u + u(1) , j·Δ v + v(1) );
    glEnd( );
}
```

And finally, if *mode* is **GL_POINT,** then a call to **glEvalMesh2** is equivalent to:

```
glBegin (GL_POINTS);
for (j = j1;   j <= j2; j += 1) {
    for (i = i1; i <= i2; i += 1) {
        glEvalCoord2(i·Δ u   +   u(1) , j·Δ v + v(1) );
    }
}
glEnd( );
```

In all three cases, the only absolute numeric requirements are that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$, and if $j = m$, then the value computed from $j \cdot \Delta v + v_{(1)}$ is exactly $v_{(2)}$.

The following functions retrieve information relating to the **glEvalMesh** function:

glGet with argument **GL_MAP1_GRID_DOMAIN**

glGet with argument **GL_MAP2_GRID_DOMAIN**

glGet with argument **GL_MAP1_GRID_SEGMENTS**

glGet with argument **GL_MAP2_GRID_SEGMENTS**

Errors

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

GL_INVALID_OPERATION is generated if **glEvalMesh** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEvalCoord**, **glEvalPoint**, **glMap1**, **glMap2**, **glMapGrid**

glEvalPoint1, glEvalPoint2

The **glEvalPoint1** and **glEvalPoint2** functions generate and evaluate a single point in a mesh.

void glEvalPoint1(
  GLint *i*
  );

void glEvalPoint2(
  GLint *i*,
  GLint *j*
  );

Parameters

i
  Specifies the integer value for grid domain variable *i*.

j
  Specifies the integer value for grid domain variable *j* (**glEvalPoint2** only).

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. The **glEvalPoint** function can be used to evaluate a single grid point in the same gridspace that is traversed by **glEvalMesh**. Calling **glEvalPoint1** is equivalent to calling

glEvalCoord1(i·$\Delta u + u_{(1)}$);

where

$$\Delta u = (u_{(2)} - u_{(1)}) / n$$

and n, $u_{(1)}$, and $u_{(2)}$ are the arguments to the most recent **glMapGrid1** command. The one absolute numeric requirement is that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$.

In the two-dimensional case, **glEvalPoint2**, let.

$$\Delta u = (u_{(2)} - u_{(1)})/n$$

$$\Delta v = (v_{(2)} - v_{(1)})/m$$

where n, $u_{(1)}$, $u_{(2)}$, m, $v_{(1)}$, and $v_{(2)}$ are the arguments to the most recent **glMapGrid2** command. Then the **glEvalPoint2** command is equivalent to calling

glEvalCoord2(i·$\Delta u + u_{(1)}$, j·$\Delta v + v_{(1)}$);

The only absolute numeric requirements are that if $i = n$, then the value computed from $i \cdot \Delta u + u_{(1)}$ is exactly $u_{(2)}$, and if $j = m$, then the value computed from $j \cdot \Delta v + v1$ is exactly $v_{(2)}$.

The following functions retrieve information relating to the **glEvalPoint1** and **glEvalPoint2** functions:

glGet with argument **GL_MAP1_GRID_DOMAIN**

glGet with argument **GL_MAP2_GRID_DOMAIN**

glGet with argument **GL_MAP1_GRID_SEGMENTS**

glGet with argument **GL_MAP2_GRID_SEGMENTS**

See Also

glEvalCoord, **glEvalMesh**, **glMap1**, **glMap2**, **glMapGrid**

glFeedbackBuffer

The **glFeedbackBuffer** function controls feedback mode.

void glFeedbackBuffer(
  GLsizei *size*,
  GLenum *type*,
  GLfloat **buffer*
  );

Parameters

size
  Specifies the maximum number of values that can be written into *buffer*.

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.

buffer
  Returns the feedback data.

Remarks

The **glFeedbackBuffer** function controls feedback. Feedback, like selection, is a GL mode. The mode is selected by calling **glRenderMode** 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.

The **glFeedbackBuffer** function has three arguments: *buffer* is a pointer to an array of floating-point values into which feedback information is placed. *size* indicates the size of the array. *type* is a symbolic constant describing the information that is fed back for each vertex. The **glFeedbackBuffer** function must be issued before feedback mode is enabled (by calling **glRenderMode** with argument **GL_FEEDBACK**). Setting **GL_FEEDBACK** without establishing the feedback buffer, or calling **glFeedbackBuffer** while the GL is in feedback mode, is an error.

The GL is taken out of feedback mode by calling **glRenderMode** with a parameter value other than **GL_FEEDBACK**. When this is done while the GL is in feedback mode, **glRenderMode** returns the number of entries placed in the feedback array. The returned value never exceeds *size*. If the feedback data required more room than was available in *buffer*, **glRenderMode** returns a negative value.

While in feedback mode, each primitive that would be rasterized generates a block of values that get 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 primitives vertexes and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling and **glPolyMode** 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 **glPassThrough** command can be used to insert a marker into the feedback buffer.

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 vertexes. Polygon entries include an integer value indicating how many vertexes 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.

  feedbackList ← feedbackItem feedbackList | feedbackItem

feedbackItem ← point | lineSegment | polygon | bitmap | pixelRectangle | passThru
point ← **GL_POINT_TOKEN** vertex
lineSegment ← **GL_LINE_TOKEN** vertex vertex | **GL_LINE_RESET_TOKEN** vertex vertex
polygon ← **GL_POLYGON_TOKEN** n polySpec
polySpec ← polySpec vertex | vertex vertex vertex
bitmap ← **GL_BITMAP_TOKEN** vertex
pixelRectangle ← **GL_DRAW_PIXEL_TOKEN** vertex | **GL_COPY_PIXEL_TOKEN** vertex
passThru ← **GL_PASS_THROUGH_TOKEN** value
vertex ← 2d | 3d | 3dColor | 3dColorTexture | 4dColorTexture
2d ← value value
3d ← value value value
3dColor ← value value value color
3dColorTexture ← value value value color tex
4dColorTexture ← value value value value color tex
color ← rgba | index
rgba ← value value value value
index ← value
tex ← value value value value

The *value* parameter is a floating-point number, and *n* is a floating-point integer giving the number of vertexes in the polygon. **GL_POINT_TOKEN**, **GL_LINE_TOKEN**, **GL_LINE_RESET_TOKEN**, **GL_POLYGON_TOKEN**, **GL_BITMAP_TOKEN**, **GL_DRAW_PIXEL_TOKEN**, **GL_COPY_PIXEL_TOKEN** and **GL_PASS_THROUGH_TOKEN** are symbolic floating-point constants. **GL_LINE_RESET_TOKEN** is returned whenever the line stipple pattern is reset. The data returned as a vertex depends on the feedback *type*.

The following table gives the correspondence between *type* and the number of values per vertex. *k* is 1 in color index mode and 4 in RGBA mode.

Type	Coordinates	Color	Texture	Total Number of Values
GL_2D	*x*, *y*			2
GL_3D	*x*, *y*, *z*			3
GL_3D_COLOR	*x*, *y*, *z*	k		3 + k
GL_3D_COLOR_TEXTURE	*x*, *y*, *z*,	k	4	7 + k
GL_4D_COLOR_TEXTURE	*x*, *y*, *z*, *w*	k	4	8 + k

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.

The **glFeedbackBuffer** function, when used in a display list, is not compiled into the display list but rather is executed immediately.

The following function retrieves information related to the **glFeedbackBuffer** function:

glGet with argument **GL_RENDER_MODE**

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 **glFeedbackBuffer** is called while the render mode is **GL_FEEDBACK**, or if **glRenderMode** is called with argument **GL_FEEDBACK** before **glFeedbackBuffer** is called at least once.

GL_INVALID_OPERATION is generated if **glFeedbackBuffer** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glLineStipple**, **glPassThrough**, **glPolygonMode**, **glRenderMode**, **glSelectBuffer**

glFinish

The **glFinish** function blocks until all GL execution is complete.

void glFinish(
  void
  );

Remarks

The **glFinish** function 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.

The **glFinish** function requires a round trip to the server.

Errors

GL_INVALID_OPERATION is generated if **glFinish** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFlush

glFlush

The **glFlush** function forces execution of GL commands in finite time.

void glFlush(
  void
  );

Remarks

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. The **glFlush** function 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 **glFlush** whenever they count on having all of their previously issued commands completed. For example, call **glFlush** before waiting for user input that depends on the generated image.

The **glFlush** function can return at any time. It does not wait until the execution of all previously issued OpenGL commands is complete.

Errors

GL_INVALID_OPERATION is generated if **glFlush** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFinish

glFogf, glFogi, glFogfv, glFogiv

These functions specify fog parameters.

void glFogf(
  GLenum *pname*,
  GLfloat *param*
  );

void glFogi(
  GLenum *pname*,
  GLint *param*
  );

Parameters

pname
  Specifies a single-valued fog parameter. **GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END,** and **GL_FOG_INDEX** are accepted.

param
  Specifies the value that *pname* will be set to.

void glFogfv(
  GLenum *pname*,
  const GLfloat **params*
  );

void glFogiv(
  GLenum *pname*,
  const GLint **params*
  );

Parameters

pname
  Specifies a fog parameter. **GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END**, **GL_FOG_INDEX**, and **GL_FOG_COLOR** are accepted.

params
  Specifies the value or values to be assigned to *pname*. **GL_FOG_COLOR** requires an array of four values. All other parameters accept an array containing only a single value.

Remarks

Fog is enabled and disabled with **glEnable** and **glDisable** using the argument **GL_FOG**. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer clear operations.

The **glFog** function assigns the value or values in *params* to the fog parameter specified by *pname*. The accepted values for *pname* are as follows:

GL_FOG_MODE
  The *params* parameter is a single integer or 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 default fog mode is **GL_EXP**.

GL_FOG_DENSITY
  The *params* parameter is a single integer or floating-point value that specifies *density*, the fog density used in both exponential fog equations. Only nonnegative densities are accepted. The default fog density is 1.0.

GL_FOG_START

The *params* parameter is a single integer or floating-point value that specifies *start*, the near distance used in the linear fog equation. The default near distance is 0.0.

GL_FOG_END

The *params* parameter is a single integer or floating-point value that specifies *end*, the far distance used in the linear fog equation. The default far distance is 1.0.

GL_FOG_INDEX

The *params* parameter is a single integer or floating-point value that specifies $i_{(f)}$, the fog color index. The default fog index is 0.0.

GL_FOG_COLOR

The *params* parameter contains four integer or floating-point values that specify $C_{(f)}$, the fog color. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. After conversion, all color components are clamped to the range [0,1]. The default fog color is (0,0,0,0).

Fog blends a fog color with each rasterized pixel fragments posttexturing color using a blending factor *f*. Factor *f* is computed in one of three ways, depending on the fog mode. Let *z* be the distance in eye coordinates from the origin to the fragment being fogged. The equation for **GL_LINEAR** fog is

{ewc msdncd, EWGraphic, group10288 0 /a "SDK.bmp"}

The equation for **GL_EXP** fog is

{ewc msdncd, EWGraphic, group10288 1 /a "SDK.bmp"}

The equation for **GL_EXP2** fog is

{ewc msdncd, EWGraphic, group10288 2 /a "SDK.bmp"}

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 fragments color $C_{(r)}$ is replaced by

{ewc msdncd, EWGraphic, group10288 3 /a "SDK.bmp"}

In color index mode, the fragments color index $i_{(r)}$ is replaced by

{ewc msdncd, EWGraphic, group10288 4 /a "SDK.bmp"}

The following functions retrieve information related to the **glFog** functions:

glGet with argument **GL_FOG_COLOR**

glGet with argument **GL_FOG_INDEX**

glGet with argument **GL_FOG_DENSITY**

glGet with argument **GL_FOG_START**

glGet with argument **GL_FOG_END**

glGet with argument **GL_FOG_MODE**

glIsEnabled with argument **GL_FOG**

Errors

GL_INVALID_ENUM is generated if pname is not an accepted value.

GL_INVALID_OPERATION is generated if **glFog** is called between a call to glBegin and the corresponding call to glEnd.

glFrontFace

The **glFrontFace** function defines front- and back-facing polygons.

void glFrontFace(
  GLenum *mode*
  );

Parameters

mode
  Specifies the orientation of front-facing polygons. **GL_CW** and **GL_CCW** are accepted. The default value is **GL_CCW**.

Remarks

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. Elimination of back-facing polygons is enabled and disabled with **glEnable** and **glDisable** using 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 polygons 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. The **glFrontFace** function 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.

The following function retrieves information about the **glFrontface** function:

glGet with argument GL_FRONT_FACE

Errors

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

GL_INVALID_OPERATION is generated if **glFrontFace** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCullFace, **glLightModel**

glFrustum

The **glFrustum** function multiplies the current matrix by a perspective matrix.

void glFrustum(
  GLdouble *left*,
  GLdouble *right*,
  GLdouble *bottom*,
  GLdouble *top*,
  GLdouble *near*,
  GLdouble *far*
  );

Parameters

left, right
  Specify the coordinates for the left and right vertical clipping planes.

bottom, top
  Specify the coordinates for the bottom and top horizontal clipping planes.

near, far
  Specify the distances to the near and far depth clipping planes. Both distances must be positive.

Remarks

The **glFrustum** function describes a perspective matrix that produces a perspective projection. (*left*, *bottom*, *near*) and (*right*, *top*, *near*) 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). *far* specifies the location of the far clipping plane. Both *near* and *far* must be positive. The corresponding matrix is

{ewc msdncd, EWGraphic, group10290 0 /a "SDK.bmp"}

{ewc msdncd, EWGraphic, group10290 1 /a "SDK.bmp"}

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 **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.

Depth buffer precision is affected by the values specified for *near* and *far*. 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

{ewc msdncd, EWGraphic, group10290 2 /a "SDK.bmp"}

roughly $\log_{(2)} r$ bits of depth buffer precision are lost. Because *r* approaches infinity as *near* approaches zero, *near* must never be set to zero.

The following functions retrieve information about the **glFrustum** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_VALUE is generated if *near* or *far* is not positive.

GL_INVALID_OPERATION is generated if **glFrustum** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glOrtho, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glViewport**

glGenLists

The **glGenLists** function generates a contiguous set of empty display lists.

GLuint glGenLists(
  GLsizei *range*
  );

Parameters

range
  Specifies the number of contiguous empty display lists to be generated.

Remarks

The **glGenLists** function 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 zero, if there is no group of *range* contiguous names available, or if any error is generated, no display lists are generated, and zero is returned.

The following function retrieves information related to the **glGenLists** function:

glIsList

Errors

GL_INVALID_VALUE is generated if *range* is negative.

GL_INVALID_OPERATION is generated if **glGenLists** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallList, **glCallLists**, **glDeleteLists**, **glNewList**

glGetBooleanv, glGetDoublev, glGetFloatv, glGetIntegerv

These functions return the value or values of a selected parameter.

void glGetBooleanv(
  GLenum *pname*,
  GLboolean **params*
  );

void glGetDoublev(
  GLenum *pname*,
  GLdouble **params*
  );

void glGetFloatv(
  GLenum *pname*,
  GLfloat **params*
  );

void glGetIntegerv(
  GLenum *pname*,
  GLint **params*
  );

Parameters

pname
  Specifies the parameter value to be returned. The symbolic constants in the list below are accepted.

params
  Returns the value or values of the specified parameter.

Remarks

These four commands return values for simple state variables in GL. *pname* is a symbolic constant indicating the state variable to be returned, and *params* is a pointer to an array of the indicated type in which to place the returned data.

Type conversion is performed if *params* has a different type than the state variable value being requested. If **glGetBooleanv** is called, a floating-point or integer value is converted to **GL_FALSE** if and only if it is zero. Otherwise, it is converted to **GL_TRUE**. If **glGetIntegerv** is called, Boolean values are returned as **GL_TRUE** or **GL_FALSE**, and most floating-point values are rounded to the nearest integer value. Floating-point colors and normals, however, are returned with a linear mapping that maps 1.0 to the most positive representable integer value, and -1.0 to the most negative representable integer value. If **glGetFloatv** or **glGetDoublev** is called, Boolean values are returned as **GL_TRUE** or **GL_FALSE**, and integer values are converted to floating-point values.

The following symbolic constants are accepted by *pname*:

GL_ACCUM_ALPHA_BITS
  The *params* parameter returns one value, the number of alpha bitplanes in the accumulation buffer.

GL_ACCUM_BLUE_BITS
  The *params* parameter returns one value, the number of blue bitplanes in the accumulation buffer.

GL_ACCUM_CLEAR_VALUE
  The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the accumulation buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearAccum**.

GL_ACCUM_GREEN_BITS

The *params* parameter returns one value, the number of green bitplanes in the accumulation buffer.

GL_ACCUM_RED_BITS
The *params* parameter returns one value, the number of red bitplanes in the accumulation buffer.

GL_ALPHA_BIAS
The *params* parameter returns one value, the alpha bias factor used during pixel transfers. See **glPixelTransfer**.

GL_ALPHA_BITS
The *params* parameter returns one value, the number of alpha bitplanes in each color buffer.

GL_ALPHA_SCALE
The *params* parameter returns one value, the alpha scale factor used during pixel transfers. See **glPixelTransfer**.

GL_ALPHA_TEST
The *params* parameter returns a single Boolean value indicating whether alpha testing of fragments is enabled. See **glAlphaFunc**.

GL_ALPHA_TEST_FUNC
The *params* parameter returns one value, the symbolic name of the alpha test function. See **glAlphaFunc**.

GL_ALPHA_TEST_REF
The *params* parameter returns one value, the reference value for the alpha test. See **glAlphaFunc**. An integer value, if requested, is linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value.

GL_ATTRIB_STACK_DEPTH
The *params* parameter returns one value, the depth of the attribute stack. If the stack is empty, zero is returned. See **glPushAttrib**.

GL_AUTO_NORMAL
The *params* parameter returns a single Boolean value indicating whether 2-D map evaluation automatically generates surface normals. See **glMap2**.

GL_AUX_BUFFERS
The *params* parameter returns one value, the number of auxiliary color buffers.

GL_BLEND
The *params* parameter returns a single Boolean value indicating whether blending is enabled. See **glBlendFunc**.

GL_BLEND_DST
The *params* parameter returns one value, the symbolic constant identifying the destination blend function. See **glBlendFunc**.

GL_BLEND_SRC
The *params* parameter returns one value, the symbolic constant identifying the source blend function. See **glBlendFunc**.

GL_BLUE_BIAS
The *params* parameter returns one value, the blue bias factor used during pixel transfers. See **glPixelTransfer**.

GL_BLUE_BITS
The *params* parameter returns one value, the number of blue bitplanes in each color buffer.

GL_BLUE_SCALE
The *params* parameter returns one value, the blue scale factor used during pixel transfers. See **glPixelTransfer**.

GL_CLIP_PLANE*i*
The *params* parameter returns a single Boolean value indicating whether the specified clipping plane is enabled. See **glClipPlane**.

GL_COLOR_CLEAR_VALUE
The *params* parameter returns four values: the red, green, blue, and alpha values used to clear the color buffers. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glClearColor**.

GL_COLOR_MATERIAL
The *params* parameter returns a single Boolean value indicating whether one or more material parameters are tracking the current color. See **glColorMaterial**.

GL_COLOR_MATERIAL_FACE
The *params* parameter returns one value, a symbolic constant indicating which materials have a parameter that is tracking the current color. See **glColorMaterial**.

GL_COLOR_MATERIAL_PARAMETER
The *params* parameter returns one value, a symbolic constant indicating which material parameters are tracking the current color. See **glColorMaterial**.

GL_COLOR_WRITEMASK
The *params* parameter returns four Boolean values: the red, green, blue, and alpha write enables for the color buffers. See **glColorMask**.

GL_CULL_FACE
The *params* parameter returns a single Boolean value indicating whether polygon culling is enabled. See **glCullFace**.

GL_CULL_FACE_MODE
The *params* parameter returns one value, a symbolic constant indicating which polygon faces are to be culled. See **glCullFace**.

GL_CURRENT_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the current color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glColor**.

GL_CURRENT_INDEX
The *params* parameter returns one value, the current color index. See **glIndex**.

GL_CURRENT_NORMAL
The *params* parameter returns three values: the *x*, *y*, and *z* values of the current normal. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glNormal**.

GL_CURRENT_RASTER_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the current raster position. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glRasterPos**.

GL_CURRENT_RASTER_DISTANCE
The *params* parameter returns one value, the distance from the eye to the current raster position. See **glRasterPos**.

GL_CURRENT_RASTER_INDEX
The *params* parameter returns one value, the color index of the current raster position. See **glRasterPos**.

GL_CURRENT_RASTER_POSITION
The *params* parameter 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 **glRasterPos**.

GL_CURRENT_RASTER_TEXTURE_COORDS
The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current raster texture coordinates. See

glRasterPos and **glTexCoord**.

GL_CURRENT_RASTER_POSITION_VALID
The *params* parameter returns a single Boolean value indicating whether the current raster position is valid. See **glRasterPos**.

GL_CURRENT_TEXTURE_COORDS
The *params* parameter returns four values: the *s*, *t*, *r*, and *q* current texture coordinates. See **glTexCoord**.

GL_DEPTH_BIAS
The *params* parameter returns one value, the depth bias factor used during pixel transfers. See **glPixelTransfer**.

GL_DEPTH_BITS
The *params* parameter returns one value, the number of bitplanes in the depth buffer.

GL_DEPTH_CLEAR_VALUE
The *params* parameter returns one value, the value that is used to clear the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and - 1.0 returns the most negative representable integer value. See **glClearDepth**.

GL_DEPTH_FUNC
The *params* parameter returns one value, the symbolic constant that indicates the depth comparison function. See **glDepthFunc**.

GL_DEPTH_RANGE
The *params* parameter returns two values: the near and far mapping limits for the depth buffer. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glDepthRange**.

GL_DEPTH_SCALE
The *params* parameter returns one value, the depth scale factor used during pixel transfers. See **glPixelTransfer**.

GL_DEPTH_TEST
The *params* parameter returns a single Boolean value indicating whether depth testing of fragments is enabled. See **glDepthFunc** and **glDepthRange**.

GL_DEPTH_WRITEMASK
The *params* parameter returns a single Boolean value indicating if the depth buffer is enabled for writing. See **glDepthMask**.

GL_DITHER
The *params* parameter returns a single Boolean value indicating whether dithering of fragment colors and indices is enabled.

GL_DOUBLEBUFFER
The *params* parameter returns a single Boolean value indicating whether double buffering is supported.

GL_DRAW_BUFFER
The *params* parameter returns one value, a symbolic constant indicating which buffers are being drawn to. See **glDrawBuffer**.

GL_EDGE_FLAG
The *params* parameter returns a single Boolean value indication whether the current edge flag is true or false. See **glEdgeFlag**.

GL_FOG
The *params* parameter returns a single Boolean value indicating whether fogging is enabled. See **glFog**.

GL_FOG_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha components of the fog

color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glFog**.

GL_FOG_DENSITY
The *params* parameter returns one value, the fog density parameter. See **glFog**.

GL_FOG_END
The *params* parameter returns one value, the end factor for the linear fog equation. See **glFog**.

GL_FOG_HINT
The *params* parameter returns one value, a symbolic constant indicating the mode of the fog hint. See **glHint**.

GL_FOG_INDEX
The *params* parameter returns one value, the fog color index. See **glFog**.

GL_FOG_MODE
The *params* parameter returns one value, a symbolic constant indicating which fog equation is selected. See **glFog**.

GL_FOG_START
The *params* parameter returns one value, the start factor for the linear fog equation. See **glFog**.

GL_FRONT_FACE
The *params* parameter returns one value, a symbolic constant indicating whether clockwise or counterclockwise polygon winding is treated as front-facing. See **glFrontFace**.

GL_GREEN_BIAS
The *params* parameter returns one value, the green bias factor used during pixel transfers.

GL_GREEN_BITS
The *params* parameter returns one value, the number of green bitplanes in each color buffer.

GL_GREEN_SCALE
The *params* parameter returns one value, the green scale factor used during pixel transfers. See **glPixelTransfer**.

GL_INDEX_BITS
The *params* parameter returns one value, the number of bitplanes in each color index buffer.

GL_INDEX_CLEAR_VALUE
The *params* parameter returns one value, the color index used to clear the color index buffers. See **glClearIndex**.

GL_INDEX_MODE
The *params* parameter returns a single Boolean value indicating whether the GL is in color index mode (true) or RGBA mode (false).

GL_INDEX_OFFSET
The *params* parameter returns one value, the offset added to color and stencil indices during pixel transfers. See **glPixelTransfer**.

GL_INDEX_SHIFT
The *params* parameter returns one value, the amount that color and stencil indices are shifted during pixel transfers. See **glPixelTransfer**.

GL_INDEX_WRITEMASK
The *params* parameter returns one value, a mask indicating which bitplanes of each color index buffer can be written. See **glIndexMask**.

GL_LIGHT*i*
The *params* parameter returns a single Boolean value indicating whether the specified light is enabled. See **glLight** and **glLightModel**.

GL_LIGHTING
The *params* parameter returns a single Boolean value indicating whether lighting is enabled. See **glLightModel**.

GL_LIGHT_MODEL_AMBIENT
The *params* parameter returns four values: the red, green, blue, and alpha components of the ambient intensity of the entire scene. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glLightModel**.

GL_LIGHT_MODEL_LOCAL_VIEWER
The *params* parameter returns a single Boolean value indicating whether specular reflection calculations treat the viewer as being local to the scene. See **glLightModel**.

GL_LIGHT_MODEL_TWO_SIDE
The *params* parameter returns a single Boolean value indicating whether separate materials are used to compute lighting for front- and back-facing polygons. See **glLightModel**.

GL_LINE_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of lines is enabled. See **glLineWidth**.

GL_LINE_SMOOTH_HINT
The *params* parameter returns one value, a symbolic constant indicating the mode of the line antialiasing hint. See **glHint**.

GL_LINE_STIPPLE
The *params* parameter returns a single Boolean value indicating whether stippling of lines is enabled. See **glLineStipple**.

GL_LINE_STIPPLE_PATTERN
The *params* parameter returns one value, the 16-bit line stipple pattern. See **glLineStipple**.

GL_LINE_STIPPLE_REPEAT
The *params* parameter returns one value, the line stipple repeat factor. See **glLineStipple**.

GL_LINE_WIDTH
The *params* parameter returns one value, the line width as specified with **glLineWidth**.

GL_LINE_WIDTH_GRANULARITY
The *params* parameter returns one value, the width difference between adjacent supported widths for antialiased lines. See **glLineWidth**.

GL_LINE_WIDTH_RANGE
The *params* parameter returns two values: the smallest and largest supported widths for antialiased lines. See **glLineWidth**.

GL_LIST_BASE
The *params* parameter returns one value, the base offset added to all names in arrays presented to **glCallLists**. See **glListBase**.

GL_LIST_INDEX
The *params* parameter returns one value, the name of the display list currently under construction. Zero is returned if no display list is currently under construction. See **glNewList**.

GL_LIST_MODE
The *params* parameter returns one value, a symbolic constant indicating the construction mode of the display list currently being constructed. See **glNewList**.

GL_LOGIC_OP
The *params* parameter returns a single Boolean value indicating whether fragment indexes are merged into the framebuffer using a logical operation. See **glLogicOp**.

GL_LOGIC_OP_MODE
The *params* parameter returns one value, a symbolic constant indicating the selected logic operational mode. See **glLogicOp**.

GL_MAP1_COLOR_4
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates colors. See **glMap1**.

GL_MAP1_GRID_DOMAIN

The *params* parameter returns two values: the endpoints of the 1-D maps grid domain. See **glMapGrid**.

GL_MAP1_GRID_SEGMENTS
The *params* parameter returns one value, the number of partitions in the 1-D maps grid domain. See **glMapGrid**.

GL_MAP1_INDEX
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates color indices. See **glMap1**.

GL_MAP1_NORMAL
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates normals. See **glMap1**.

GL_MAP1_TEXTURE_COORD_1
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 1D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_2
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 2D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_3
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 3D texture coordinates. See **glMap1**.

GL_MAP1_TEXTURE_COORD_4
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 4D texture coordinates. See **glMap1**.

GL_MAP1_VERTEX_3
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 3D vertex coordinates. See **glMap1**.

GL_MAP1_VERTEX_4
The *params* parameter returns a single Boolean value indicating whether 1D evaluation generates 4D vertex coordinates. See **glMap1**.

GL_MAP2_COLOR_4
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates colors. See **glMap2**.

GL_MAP2_GRID_DOMAIN
The *params* parameter returns four values: the endpoints of the 2-D maps *i* and *j* grid domains. See **glMapGrid**.

GL_MAP2_GRID_SEGMENTS
The *params* parameter returns two values: the number of partitions in the 2-D maps *i* and *j* grid domains. See **glMapGrid**.

GL_MAP2_INDEX
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates color indices. See **glMap2**.

GL_MAP2_NORMAL
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates normals. See **glMap2**.

GL_MAP2_TEXTURE_COORD_1
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 1D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_2
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 2D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_3

The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 3D texture coordinates. See **glMap2**.

GL_MAP2_TEXTURE_COORD_4
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 4D texture coordinates. See **glMap2**.

GL_MAP2_VERTEX_3
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 3D vertex coordinates. See **glMap2**.

GL_MAP2_VERTEX_4
The *params* parameter returns a single Boolean value indicating whether 2D evaluation generates 4D vertex coordinates. See **glMap2**.

GL_MAP_COLOR
The *params* parameter returns a single Boolean value indicating if colors and color indices are to be replaced by table lookup during pixel transfers. See **glPixelTransfer**.

GL_MAP_STENCIL
The *params* parameter returns a single Boolean value indicating if stencil indices are to be replaced by table lookup during pixel transfers. See **glPixelTransfer**.

GL_MATRIX_MODE
The *params* parameter 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
The *params* parameter returns one value, the maximum supported depth of the attribute stack. See **glPushAttrib**.

GL_MAX_CLIP_PLANES
The *params* parameter returns one value, the maximum number of application-defined clipping planes. See **glClipPlane**.

GL_MAX_EVAL_ORDER
The *params* parameter returns one value, the maximum equation order supported by 1-D and 2-D evaluators. See **glMap1** and **glMap2**.

GL_MAX_LIGHTS
The *params* parameter returns one value, the maximum number of lights. See **glLight**.

GL_MAX_LIST_NESTING
The *params* parameter returns one value, the maximum recursion depth allowed during display-list traversal. See **glCallList**.

GL_MAX_MODELVIEW_STACK_DEPTH
The *params* parameter returns one value, the maximum supported depth of the modelview matrix stack. See **glPushMatrix**.

GL_MAX_NAME_STACK_DEPTH
The *params* parameter returns one value, the maximum supported depth of the selection name stack. See **glPushName**.

GL_MAX_PIXEL_MAP_TABLE
The *params* parameter returns one value, the maximum supported size of a **glPixelMap** lookup table. See **glPixelMap**.

GL_MAX_PROJECTION_STACK_DEPTH
The *params* parameter returns one value, the maximum supported depth of the projection matrix stack. See **glPushMatrix**.

GL_MAX_TEXTURE_SIZE
The *params* parameter returns one value, the maximum width or height of any texture image (without borders). See **glTexImage1D** and **glTexImage2D**

GL_MAX_TEXTURE_STACK_DEPTH
The *params* parameter returns one value, the maximum supported depth of the texture matrix stack.

See **glPushMatrix**.

GL_MAX_VIEWPORT_DIMS
The *params* parameter returns two values: the maximum supported width and height of the viewport. See **glViewport**.

GL_MODELVIEW_MATRIX
The *params* parameter returns sixteen values: the modelview matrix on the top of the modelview matrix stack. See **glPushMatrix**.

GL_MODELVIEW_STACK_DEPTH
The *params* parameter returns one value, the number of matrices on the modelview matrix stack. See **glPushMatrix**.

GL_NAME_STACK_DEPTH
The *params* parameter returns one value, the number of names on the selection name stack. See **glPushMatrix**.

GL_NORMALIZE
The *params* parameter returns a single Boolean value indicating whether normals are automatically scaled to unit length after they have been transformed to eye coordinates. See **glNormal**.

GL_PACK_ALIGNMENT
The *params* parameter returns one value, the byte alignment used for writing pixel data to memory. See **glPixelStore**.

GL_PACK_LSB_FIRST
The *params* parameter 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 **glPixelStore**.

GL_PACK_ROW_LENGTH
The *params* parameter returns one value, the row length used for writing pixel data to memory. See **glPixelStore**.

GL_PACK_SKIP_PIXELS
The *params* parameter returns one value, the number of pixel locations skipped before the first pixel is written into memory. See **glPixelStore**.

GL_PACK_SKIP_ROWS
The *params* parameter returns one value, the number of rows of pixel locations skipped before the first pixel is written into memory. See **glPixelStore**.

GL_PACK_SWAP_BYTES
The *params* parameter 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 **glPixelStore**.

GL_PERSPECTIVE_CORRECTION_HINT
The *params* parameter returns one value, a symbolic constant indicating the mode of the perspective correction hint. See **glHint**.

GL_PIXEL_MAP_A_TO_A_SIZE
The *params* parameter returns one value, the size of the alpha-to-alpha pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_B_TO_B_SIZE
The *params* parameter returns one value, the size of the blue-to-blue pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_G_TO_G_SIZE
The *params* parameter returns one value, the size of the green-to-green pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_A_SIZE
The *params* parameter returns one value, the size of the index-to-alpha pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_B_SIZE
The *params* parameter returns one value, the size of the index-to-blue pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_G_SIZE
The *params* parameter returns one value, the size of the index-to-green pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_I_SIZE
The *params* parameter returns one value, the size of the index-to-index pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_I_TO_R_SIZE
The *params* parameter returns one value, the size of the index-to-red pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_R_TO_R_SIZE
The *params* parameter returns one value, the size of the red-to-red pixel translation table. See **glPixelMap**.

GL_PIXEL_MAP_S_TO_S_SIZE
The *params* parameter returns one value, the size of the stencil-to-stencil pixel translation table. See **glPixelMap**.

GL_POINT_SIZE
The *params* parameter returns one value, the point size as specified by **glPointSize**.

GL_POINT_SIZE_GRANULARITY
The *params* parameter returns one value, the size difference between adjacent supported sizes for antialiased points. See **glPointSize**.

GL_POINT_SIZE_RANGE
The *params* parameter returns two values: the smallest and largest supported sizes for antialiased points. See **glPointSize**.

GL_POINT_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of points is enabled. See **glPointSize**.

GL_POINT_SMOOTH_HINT
The *params* parameter returns one value, a symbolic constant indicating the mode of the point antialiasing hint. See **glHint**.

GL_POLYGON_MODE
The *params* parameter returns two values: symbolic constants indicating whether front-facing and back-facing polygons are rasterized as points, lines, or filled polygons. See **glPolygonMode**.

GL_POLYGON_SMOOTH
The *params* parameter returns a single Boolean value indicating whether antialiasing of polygons is enabled. See **glPolygonMode**.

GL_POLYGON_SMOOTH_HINT
The *params* parameter returns one value, a symbolic constant indicating the mode of the polygon antialiasing hint. See **glHint**.

GL_POLYGON_STIPPLE
The *params* parameter returns a single Boolean value indicating whether stippling of polygons is enabled. See **glPolygonStipple**.

GL_PROJECTION_MATRIX
The *params* parameter returns sixteen values: the projection matrix on the top of the projection matrix stack. See **glPushMatrix**.

GL_PROJECTION_STACK_DEPTH
The *params* parameter returns one value, the number of matrices on the projection matrix stack. See **glPushMatrix**.

GL_READ_BUFFER

The *params* parameter returns one value, a symbolic constant indicating which color buffer is selected for reading. See **glReadPixels** and **glAccum**.

GL_RED_BIAS
The *params* parameter returns one value, the red bias factor used during pixel transfers.

GL_RED_BITS
The *params* parameter returns one value, the number of red bitplanes in each color buffer.

GL_RED_SCALE
The *params* parameter returns one value, the red scale factor used during pixel transfers. See **glPixelTransfer**.

GL_RENDER_MODE
The *params* parameter returns one value, a symbolic constant indicating whether the GL is in render, select, or feedback mode. See **glRenderMode**.

GL_RGBA_MODE
The *params* parameter returns a single Boolean value indicating whether the GL is in RGBA mode (true) or color index mode (false). See **glColor**.

GL_SCISSOR_BOX
The *params* parameter returns four values: the *x* and *y* window coordinates of the scissor box, followed by its width and height. See **glScissor**.

GL_SCISSOR_TEST
The *params* parameter returns a single Boolean value indicating whether scissoring is enabled. See **glScissor**.

GL_SHADE_MODEL
The *params* parameter returns one value, a symbolic constant indicating whether the shading mode is flat or smooth. See **glShadeModel**.

GL_STENCIL_BITS
The *params* parameter returns one value, the number of bitplanes in the stencil buffer.

GL_STENCIL_CLEAR_VALUE
The *params* parameter returns one value, the index to which the stencil bitplanes are cleared. See **glClearStencil**.

GL_STENCIL_FAIL
The *params* parameter returns one value, a symbolic constant indicating what action is taken when the stencil test fails. See **glStencilOp**.

GL_STENCIL_FUNC
The *params* parameter returns one value, a symbolic constant indicating what function is used to compare the stencil reference value with the stencil buffer value. See **glStencilFunc**.

GL_STENCIL_PASS_DEPTH_FAIL
The *params* parameter returns one value, a symbolic constant indicating what action is taken when the stencil test passes, but the depth test fails. See **glStencilOp**.

GL_STENCIL_PASS_DEPTH_PASS
The *params* parameter returns one value, a symbolic constant indicating what action is taken when the stencil test passes and the depth test passes. See **glStencilOp**.

GL_STENCIL_REF
The *params* parameter returns one value, the reference value that is compared with the contents of the stencil buffer. See **glStencilFunc**.

GL_STENCIL_TEST
The *params* parameter returns a single Boolean value indicating whether stencil testing of fragments is enabled. See **glStencilFunc** and **glStencilOp**.

GL_STENCIL_VALUE_MASK
The *params* parameter 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 **glStencilFunc**.

GL_STENCIL_WRITEMASK
The *params* parameter returns one value, the mask that controls writing of the stencil bitplanes. See **glStencilMask**.

GL_STEREO
The *params* parameter returns a single Boolean value indicating whether stereo buffers (left and right) are supported.

GL_SUBPIXEL_BITS
The *params* parameter 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
The *params* parameter returns a single Boolean value indicating whether 1D texture mapping is enabled. See **glTexImage1D**.

GL_TEXTURE_2D
The *params* parameter returns a single Boolean value indicating whether 2D texture mapping is enabled. See **glTexImage2D**.

GL_TEXTURE_ENV_COLOR
The *params* parameter returns four values: the red, green, blue, and alpha values of the texture environment color. Integer values, if requested, are linearly mapped from the internal floating-point representation such that 1.0 returns the most positive representable integer value, and -1.0 returns the most negative representable integer value. See **glTexEnv**.

GL_TEXTURE_ENV_MODE
The *params* parameter returns one value, a symbolic constant indicating what texture environment function is currently selected. See **glTexEnv**.

GL_TEXTURE_GEN_S
The *params* parameter returns a single Boolean value indicating whether automatic generation of the S texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_T
The *params* parameter returns a single Boolean value indicating whether automatic generation of the T texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_R
The *params* parameter returns a single Boolean value indicating whether automatic generation of the R texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_GEN_Q
The *params* parameter returns a single Boolean value indicating whether automatic generation of the Q texture coordinate is enabled. See **glTexGen**.

GL_TEXTURE_MATRIX
The *params* parameter returns sixteen values: the texture matrix on the top of the texture matrix stack. See **glPushMatrix**.

GL_TEXTURE_STACK_DEPTH
The *params* parameter returns one value, the number of matrices on the texture matrix stack. See **glPushMatrix**.

GL_UNPACK_ALIGNMENT
The *params* parameter returns one value, the byte alignment used for reading pixel data from memory. See **glPixelStore**.

GL_UNPACK_LSB_FIRST
The *params* parameter 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 **glPixelStore**.

GL_UNPACK_ROW_LENGTH
The *params* parameter returns one value, the row length used for reading pixel data from memory. See **glPixelStore**.

GL_UNPACK_SKIP_PIXELS

The *params* parameter returns one value, the number of pixel locations skipped before the first pixel is read from memory. See **glPixelStore**.

GL_UNPACK_SKIP_ROWS
The *params* parameter returns one value, the number of rows of pixel locations skipped before the first pixel is read from memory. See **glPixelStore**.

GL_UNPACK_SWAP_BYTES
The *params* parameter 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 **glPixelStore**.

GL_VIEWPORT
The *params* parameter returns four values: the *x* and *y* window coordinates of the viewport, followed by its width and height. See **glViewport**.

GL_ZOOM_X
The *params* parameter returns one value, the *x* pixel zoom factor. See **glPixelZoom**.

GL_ZOOM_Y
The *params* parameter returns one value, the *y* pixel zoom factor. See **glPixelZoom**.

Many of the Boolean parameters can also be queried more easily using **glIsEnabled**.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGet** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glGetClipPlane, **glGetError**, **glGetLight**, **glGetMap**, **glGetMaterial**, **glGetPixelMap**,**glGetPolygonStipple**, **glGetString**, **glGetTexEnv**, **glGetTexGen**, **glGetTexImage**, **glGetTexLevelParameter**, **glGetTexParameter**, **glIsEnabled**

glGetClipPlane

The **glGetClipPlane** function returns the coefficients of the specified clipping plane.

void glGetClipPlane(
  GLenum *plane*,
  GLdouble **equation*
  );

Parameters

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_PLANE**i where 0 ≤ *i* < **GL_MAX_CLIP_PLANES**.

equation
  Returns four double-precision values that are the coefficients of the plane equation of *plane* in eye coordinates.

Remarks

The **glGetClipPlane** function returns in *equation* the four coefficients of the plane equation for *plane*.

It is always the case that **GL_CLIP_PLANE***i* = **GL_CLIP_PLANE0** + *i*.

If an error is generated, no change is made to the contents of *equation*.

Errors

GL_INVALID_ENUM is generated if *plane* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetClipPlane** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glClipPlane

glGetError

The **glGetError** function returns error information.

GLenum glGetError(
  void
  );

Remarks

The **glGetError** function 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 **glGetError** is called, the error code is returned, and the flag is reset to **GL_NO_ERROR**. If a call to **glGetError** returns **GL_NO_ERROR**, there has been no detectable error since the last call to **glGetError**, 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 **glGetError** is called. If more than one flag has recorded an error, **glGetError** returns and clears an arbitrary error flag value. Thus, **glGetError** 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 currently defined errors are as follows:

GL_NO_ERROR
  No error has been recorded. The value of this symbolic constant is guaranteed to be zero.

GL_INVALID_ENUM
  An unacceptable value is specified for an enumerated argument. The offending command is ignored, having no side effect other than to set the error flag.

GL_INVALID_VALUE
  A numeric argument is out of range. The offending command is ignored, having no side effect other than to set the error flag.

GL_INVALID_OPERATION
  The specified operation is not allowed in the current state. The offending command is ignored, having no side effect other than to set the error flag.

GL_STACK_OVERFLOW
  This command would cause a stack overflow. The offending command is ignored, having no side effect other than to set the error flag.

GL_STACK_UNDERFLOW
  This command would cause a stack underflow. The offending command is ignored, having no side effect other 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.

Errors

GL_INVALID_OPERATION is generated if **glGetError** is called between a call to **glBegin** and the corresponding call to **glEnd**.

glGetLightfv, glGetLightiv

The **glGetLightfv** and **glGetLightiv** functions return light source parameter values.

void glGetLightfv(
  GLenum *light*,
  GLenum *pname*,
  GLfloat **params*
  );

void glGetLightiv(
  GLenum *light*,
  GLenum *pname*,
  GLint **params*
  );

Parameters

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_LIGHT***i* where 0 ≤ *i* < < **GL_MAX_LIGHTS**.

pname
  Specifies a light source parameter for *light*. Accepted symbolic names are **GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION** , and **GL_QUADRATIC_ATTENUATION**.

params
  Returns the requested data.

Remarks

The **glGetLight** function returns in *params* the value or values of a light source parameter. *light* names the light and is a symbolic name of the form **GL_LIGHT***i* for 0≤*i*<**GL_MAX_LIGHTS**, where **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 parameters are as follows:

GL_AMBIENT
  The *params* parameter returns four integer or floating-point values representing the ambient intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_DIFFUSE
  The *params* parameter returns four integer or floating-point values representing the diffuse intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SPECULAR
  The *params* parameter returns four integer or floating-point values representing the specular intensity of the light source. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_POSITION
The *params* parameter returns four integer or floating-point values representing the position of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identity at the time **glLight** was called.

GL_SPOT_DIRECTION
The *params* parameter returns three integer or floating-point values representing the direction of the light source. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer value. The returned values are those maintained in eye coordinates. They will not be equal to the values specified using **glLight**, unless the modelview matrix was identity at the time **glLight** 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.

GL_SPOT_EXPONENT
The *params* parameter returns a single integer or floating-point value representing the spot exponent of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_SPOT_CUTOFF
The *params* parameter returns a single integer or floating-point value representing the spot cutoff angle of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_CONSTANT_ATTENUATION
The *params* parameter returns a single integer or floating-point value representing the constant (not distance related) attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_LINEAR_ATTENUATION
The *params* parameter returns a single integer or floating-point value representing the linear attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

GL_QUADRATIC_ATTENUATION
The *params* parameter returns a single integer or floating-point value representing the quadratic attenuation of the light. An integer value, when requested, is computed by rounding the internal floating-point representation to the nearest integer.

It is always the case that **GL_LIGHT***i* = **GL_LIGHT0** + *i*.

If an error is generated, no change is made to the contents of *params*.

Errors

GL_INVALID_ENUM is generated if *light* or *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetLight** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLight

glGetMapdv, glGetMapfv, glGetMapiv

These functions return evaluator parameters.

void glGetMapdv(
  GLenum *target*,
  GLenum *query*,
  GLdouble **v*
  );

void glGetMapfv(
  GLenum *target*,
  GLenum *query*,
  GLfloat **v*
  );

void glGetMapiv(
  GLenum *target*,
  GLenum *query*,
  GLint **v*
  );

Parameters

target
  Specifies the symbolic name of a map. Accepted values are **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**, and **GL_MAP2_VERTEX_4**.

query
  Specifies which parameter to return. Symbolic names **GL_COEFF**, **GL_ORDER**, and **GL_DOMAIN** are accepted.

v
  Returns the requested data.

Remarks

The **glGetMap** function returns evaluator parameters. (The **glMap1** and **glMap2** functions define evaluators.) *target* chooses a map, *query* selects a specific parameter, and *v* points to storage where the values will be returned.

The acceptable values for the *target* parameter are described in **glMap1** and **glMap2**

The *query* parameter can assume the following values:

GL_COEFF
  The *v* parameter returns the control points for the evaluator function. One-dimensional evaluators return *order* control points, and two-dimensional evaluators return *uorder*x*vorder* control points. Each control point consists of one, two, three, or four integer, single-precision floating-point, or double-precision floating-point values, depending on the type of the evaluator. Two-dimensional control points are returned in row-major order, incrementing the *uorder* index quickly, and the *vorder* index after each row. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

GL_ORDER

v returns the order of the evaluator function. One-dimensional evaluators return a single value, *order*. Two-dimensional evaluators return two values, *uorder* and *vorder*.

GL_DOMAIN
v returns the linear u and v mapping parameters. One-dimensional evaluators return two values, *u*1 and *u*2, as specified by **glMap1**. Two-dimensional evaluators return four values (*u*1, *u*2, *v*1, and *v*2) as specified by **glMap2**. Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

If an error is generated, no change is made to the contents of *v*.

Errors

GL_INVALID_ENUM is generated if either *target* or *query* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetMap** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEvalCoord, **glMap1**, **glMap2**

glGetMaterialfv, glGetMaterialiv

The **glGetMaterialfv** and **glGetMaterialiv** functions return material parameters.

void glGetMaterialfv(
  GLenum *face*,
  GLenum *pname*,
  GLfloat **params*
  );

void glGetMaterialiv(
  GLenum *face*,
  GLenum *pname*,
  GLint **params*
  );

Parameters

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. **GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS,** and **GL_COLOR_INDEXES** are accepted.

params
  Returns the requested data.

Remarks

The **glGetMaterial** function returns in *params* the value or values of parameter *pname* of material *face*. Six parameters are defined:

GL_AMBIENT
  The *params* parameter returns four integer or floating-point values representing the ambient reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_DIFFUSE
  The *params* parameter returns four integer or floating-point values representing the diffuse reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SPECULAR
  The *params* parameter returns four integer or floating-point values representing the specular reflectance of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_EMISSION
  The *params* parameter returns four integer or floating-point values representing the emitted light intensity of the material. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer value, and -1.0 maps to the most negative representable integer value. If the internal value is outside the range [-1,1], the corresponding integer return value is undefined.

GL_SHININESS
The *params* parameter returns one integer or floating-point value representing the specular exponent of the material. Integer values, when requested, are computed by rounding the internal floating-point value to the nearest integer value.

GL_COLOR_INDEXES
The *params* parameter returns three integer or floating-point values representing the ambient, diffuse, and specular indices of the material. These indices are used only for color index lighting. (The other parameters are all used only for RGBA lighting.) Integer values, when requested, are computed by rounding the internal floating-point values to the nearest integer values.

If an error is generated, no change is made to the contents of *params*.

Errors

GL_INVALID_ENUM is generated if *face* or *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetMaterial** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glMaterial

glGetPixelMapfv, glGetPixelMapuiv, glGetPixelMapusv

These functions return the specified pixel map.

void glGetPixelMapfv(
  GLenum *map*,
  GLfloat **values*
  );

void glGetPixelMapuiv(
  GLenum *map*,
  GLuint **values*
  );

void glGetPixelMapusv(
  GLenum *map*,
  GLushort **values*
  );

Parameters

map
Specifies the name of the pixel map to return. Accepted values are **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**, and **GL_PIXEL_MAP_A_TO_A***values*
  Returns the pixel map contents.

Remarks

See **glPixelMap** for a description of the acceptable values for the *map* parameter. The **glGetPixelMap** function returns in *values* the contents of the pixel map specified in *map*. Pixel maps are used during the execution of **glReadPixels**, **glDrawPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** to map color indices, stencil indices, color components, and depth components to other values.

Unsigned integer values, if requested, are linearly mapped from the internal fixed or floating-point representation such that 1.0 maps to the largest representable integer value, and 0.0 maps to zero. Return unsigned integer values are undefined if the map value was not in the range [0,1].

To determine the required size of *map*, call **glGet** with the appropriate symbolic constant.

If an error is generated, no change is made to the contents of *values*.

The following functions retrieve information related to the **glGetPixelMap** function:

glGet with argument **GL_PIXEL_MAP_I_TO_I_SIZE**

glGet with argument **GL_PIXEL_MAP_S_TO_S_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_R_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_G_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_B_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_A_SIZE**

glGet with argument **GL_PIXEL_MAP_R_TO_R_SIZE**

glGet with argument **GL_PIXEL_MAP_G_TO_G_SIZE**

glGet with argument **GL_PIXEL_MAP_B_TO_B_SIZE**

glGet with argument **GL_PIXEL_MAP_A_TO_A_SIZE**

glGet with argument **GL_MAX_PIXEL_MAP_TABLE**

Errors

GL_INVALID_ENUM is generated if *map* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetPixelMap** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glDrawPixels**, **glPixelMap**, **glPixelTransfer**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glGetPolygonStipple

The **glGetPolygonStipple** function returns the polygon stipple pattern.

void glGetPolygonStipple(
  GLubyte **mask*
  );

Parameters

mask
  Returns the stipple pattern.

Remarks

The **glGetPolygonStipple** function returns to *mask* a 32x32 polygon stipple pattern. The pattern is packed into memory as if **glReadPixels** 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 32x32 color index buffer. Unlike **glReadPixels**, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image.

If an error is generated, no change is made to the contents of *mask*.

Errors

GL_INVALID_OPERATION is generated if **glGetPolygonStipple** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glPixelStore, **glPixelTransfer**, **glPolygonStipple**, **glReadPixels**

glGetString

The **glGetString** function returns a string describing the current GL connection.

const GLubyte * **glGetString**(
GLenum *name*
);

Parameters

name
Specifies a symbolic constant, one of **GL_VENDOR**, **GL_RENDERER**, **GL_VERSION**, or **GL_EXTENSIONS**.

Remarks

The **glGetString** function returns a pointer to a static string describing some aspect of the current GL connection. The *name* parameter 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 GL does not include queries for the performance characteristics of an implementation, it is expected that some applications will be written to recognize known platforms and will modify their GL usage based on known performance characteristics of these platforms. Strings **GL_VENDOR** and **GL_RENDERER** together uniquely specify a platform, and will not change from release to release. They should be used by such platform recognition algorithms.

The format and contents of the string that **glGetString** returns depend on the implementation, except that extension names will not include space characters and will be separated by space characters in the **GL_EXTENSIONS** string, and that all strings are null-terminated.

If an error is generated, **glGetString** returns zero.

Errors

GL_INVALID_ENUM is generated if *name* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetString** is called between a call to **glBegin** and the corresponding call to **glEnd**.

glGetTexEnvfv, glGetTexEnviv

The **glGetTexEnvfv** and **glGetTexEnviv** functions return texture environment parameters.

void glGetTexEnvfv(
  GLenum *target*,
  GLenum *pname*,
  GLfloat **params*
  );

void glGetTexEnviv(
  GLenum *target*,
  GLenum *pname*,
  GLint **params*
  );

Parameters

target
  Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

pname
  Specifies the symbolic name of a texture environment parameter. Accepted values are **GL_TEXTURE_ENV_MODE** and **GL_TEXTURE_ENV_COLOR**.

params
  Returns the requested data.

Remarks

The **glGetTexEnv** function returns in *params* selected values of a texture environment that was specified with **glTexEnv**. *target* specifies a texture environment. Currently, only one texture environment is defined and supported: **GL_TEXTURE_ENV**.

The *pname* parameter names a specific texture environment parameter. The two parameters are as follows:

GL_TEXTURE_ENV_MODE
  The *params* parameter returns the single-valued texture environment mode, a symbolic constant.

GL_TEXTURE_ENV_COLOR
  The *params* parameter returns four integer or floating-point values that are the texture environment color. Integer values, when requested, are linearly mapped from the internal floating-point representation such that 1.0 maps to the most positive representable integer, and -1.0 maps to the most negative representable integer.

If an error is generated, no change is made to the contents of *params*.

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetTexEnv** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexEnv

glGetTexGendv, glGetTexGenfv, glGetTexGeniv

These functions return texture coordinate generation parameters.

void glGetTexGendv(
  GLenum *coord*,
  GLenum *pname*,
  GLdouble **params*
  );

void glGetTexGenfv(
  GLenum *coord*,
  GLenum *pname*,
  GLfloat **params*
  );

void glGetTexGeniv(
  GLenum *coord*,
  GLenum *pname*,
  GLint **params*
  );

Parameters

coord
  Specifies a texture coordinate. Must be **GL_S**, **GL_T**, **GL_R**, or **GL_Q**.

pname
  Specifies the symbolic name of the value(s) to be returned. Must be either **GL_TEXTURE_GEN_MODE** or the name of one of the texture generation plane equations: **GL_OBJECT_PLANE** or **GL_EYE_PLANE**.

params
  Returns the requested data.

Remarks

The **glGetTexGen** funtion returns in *params* selected parameters of a texture coordinate generation function that was specified using **glTexGen**. *coord* names one of the (*s*,*t*,*r*,*q*) texture coordinates, using the symbolic constant **GL_S**, **GL_T**, **GL_R**, or **GL_Q**.

The *pname* parameter specifies one of three symbolic names:

GL_TEXTURE_GEN_MODE
  params returns the single-valued texture generation function, a symbolic constant.

GL_OBJECT_PLANE
  The *params* parameter returns the four plane equation coefficients that specify object linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation.

GL_EYE_PLANE
  The *params* parameter returns the four plane equation coefficients that specify eye linear-coordinate generation. Integer values, when requested, are mapped directly from the internal floating-point representation. The returned values are those maintained in eye coordinates. They are not equal to the values specified using **glTexGen**, unless the modelview matrix was identified at the time **glTexGen** was called.

If an error is generated, no change is made to the contents of *params*.

Errors

GL_INVALID_ENUM is generated if *coord* or *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetTexGen** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexGen

glGetTexImage

The **glGetTexImage** function returns a texture image.

void glGetTexImage(
  GLenum *target*,
  GLint *level*,
  GLenum *format*,
  GLenum *type*,
  GLvoid **pixels*
  );

Parameters

target
  Specifies which texture is to be obtained. **GL_TEXTURE_1D** and **GL_TEXTURE_2D** are accepted.

level
  Specifies the level-of-detail number of the desired image. Level 0 is the base image level. Level *n* is the *n*th 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**.

type
  Specifies a pixel type for the returned data. The supported types are **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, and **GL_FLOAT**.

pixels
  Returns the texture image. Should be a pointer to an array of the type specified by *type*.

Remarks

The **glGetTexImage** function returns a texture image into *pixels*. *target* specifies whether the desired texture image is one specified by **glTexImage1D** (**GL_TEXTURE_1D**) or by **glTexImage2D** (**GL_TEXTURE_2D**). *level* specifies the level-of-detail number of the desired image. *format* and *type* specify the format and type of the desired image array. Please see **glTexImage1D** and **glDrawPixels** for a description of the acceptable values for the *format* and *type* parameters, respectively.

Operation of **glGetTexImage** is best understood by considering the selected internal four-component texture image to be an RGBA color buffer the size of the image. The semantics of **glGetTexImage** are then identical to those of **glReadPixels** called with the same *format* and *type*, with *x* and *y* set to zero, *width* set to the width of the texture image (including border if one was specified), and *height* set to one for 1-D images, or to the height of the texture image (including border if one was specified) for 2-D 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, and green, blue, and alpha set to zero. 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 zero. 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 zero.

To determine the required size of *pixels*, use **glGetTexLevelParameter** to ascertain 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**.

If an error is generated, no change is made to the contents of *pixels*.

The following functions retrieve information related to the **glGetTexImage** function:

glGetTexLevelParameter with argument **GL_TEXTURE_WIDTH**

glGetTexLevelParameter with argument **GL_TEXTURE_HEIGHT**

glGetTexLevelParameter with argument **GL_TEXTURE_BORDER**

glGetTexLevelParameter with argument **GL_TEXTURE_COMPONENTS**

glGet with arguments **GL_PACK_ALIGNMENT** and others

Errors

GL_INVALID_ENUM is generated if *target*, *format*, or *type* is not an accepted value.

GL_INVALID_VALUE is generated if *level* is less than zero or greater than $\log_{(2)}$ max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_OPERATION is generated if **glGetTexImage** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glGetTexLevelParameterfv, glGetTexLevelParameteriv

The **glGetTexLevelParameterfv** and **glGetTexLevelParameteriv** functions return texture parameter values for a specific level of detail.

void glGetTexLevelParameterfv(
GLenum *target*,
GLint *level*,
GLenum *pname*,
GLfloat **params*
);

void glGetTexLevelParameteriv(
GLenum *target*,
GLint *level*,
GLenum *pname*,
GLint **params*
);

Parameters

target
Specifies the symbolic name of the target texture, either **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 *n*th mipmap reduction image.

pname
Specifies the symbolic name of a texture parameter. **GL_TEXTURE_WIDTH**, **GL_TEXTURE_HEIGHT**, **GL_TEXTURE_COMPONENTS**, and **GL_TEXTURE_BORDER** are accepted.

params
Returns the requested data.

Remarks

The **glGetTexLevelParameter** function returns in *params* texture parameter values for a specific level-of-detail value, specified as *level*. *target* defines the target texture, either **GL_TEXTURE_1D** or **GL_TEXTURE_2D**, to specify one- or two-dimensional texturing. *pname* specifies the texture parameter whose value or values will be returned.

The accepted parameter names are as follows:

GL_TEXTURE_WIDTH
The *params* parameter returns a single value, the width of the texture image. This value includes the border of the texture image.

GL_TEXTURE_HEIGHT
The *params* parameter returns a single value, the height of the texture image. This value includes the border of the texture image.

GL_TEXTURE_COMPONENTS
The *params* parameter returns a single value, the number of components in the texture image.

GL_TEXTURE_BORDER
The *params* parameter returns a single value, the width in pixels of the border of the texture image.

If an error is generated, no change is made to the contents of *params*.

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 zero or greater than $\log_{(2)}$ max, where max is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_OPERATION is generated if **glGetTexLevelParameter** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glGetTexParameter, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glGetTexParameterfv, glGetTexParameteriv

The **glGetTexParameterfv** and **glGetTexParameteriv** functions return texture parameter values.

void glGetTexParameterfv(
  GLenum *target*,
  GLenum *pname*,
  GLfloat **params*
  );

void glGetTexParameteriv(
  GLenum *target*,
  GLenum *pname*,
  GLint **params*
  );

Parameters

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. **GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_WRAP_S**, **GL_TEXTURE_WRAP_T**, and **GL_TEXTURE_BORDER_COLOR** are accepted.

params
  Returns the texture parameters.

Remarks

The **glGetTexParameter** function returns in *params* the value or values of the texture parameter specified as *pname*. The *target* parameter defines the target texture, either **GL_TEXTURE_1D** or **GL_TEXTURE_2D**, to specify one- or two-dimensional texturing. The *pname* parameter accepts the same symbols as **glTexParameter**, with the same interpretations:

GL_TEXTURE_MAG_FILTER
  Returns the single-valued texture magnification filter, a symbolic constant.

GL_TEXTURE_MIN_FILTER
  Returns the single-valued texture minification filter, a symbolic constant.

GL_TEXTURE_WRAP_S
  Returns the single-valued wrapping function for texture coordinate *s*, a symbolic constant.

GL_TEXTURE_WRAP_T
  Returns the single-valued wrapping function for texture coordinate *t*, a symbolic constant.

GL_TEXTURE_BORDER_COLOR
  Returns four integer or floating-point numbers that comprise the RGBA color of the texture border. Floating-point values are returned in the range [0,1]. Integer values are returned as a linear mapping of the internal floating-point representation such that 1.0 maps to the most positive representable integer and -1.0 maps to the most negative representable integer.

If an error is generated, no change is made to the contents of *params*.

Errors

GL_INVALID_ENUM is generated if *target* or *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glGetTexParameter** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexParameter

glAddSwapHintRectWIN

The **glAddSwapHintRectWIN** function specifies a set of rectangles that are to be copied by **SwapBuffers**.

void glAddSwapHintRectWIN(
  GLint *x*,
  GLint *y*,
  GLsizei *width*,
  GLsizei *height*
  );

Parameters

x
  Specifies the x coordinate (in window coordinates) of lower-left hand corner of the hint region rectangle.

y
  Specifies the y coordinate (in window coordinates) of lower-left hand corner of the hint region rectangle.

width
  Specifies the width of the hint region rectangle.

height
  Specifies the height of the hint region rectangle.

Remarks

The **glAddSwapHintRectWIN** function enables your application to speed up animation by reducing the amount of repainting between frames. With **glAddSwapHintRectWIN** you specify a set of rectangular areas that you want copied when you call the **SwapBuffers** function. When no rectangles are specified with **glAddSwapHintRectWIN** before calling **SwapBuffers**, the entire frame buffer is swapped. Using **glAddSwapHintRectWIN** to copy only parts of the buffer that changed can increase the performance of **SwapBuffers** significantly, especially when **SwapBuffers** is implemented in software.

The **glAddSwapHintRectWIN** function adds a rectangle to the hint region. When the **PFD_SWAP_COPY** flag of the **PIXELFORMATDESCRIPTOR** pixel format structure is set, **SwapBuffers** uses this region to clip the copying of the back buffer to the front buffer. You don't specify **PFD_SWAP_COPY**, it is set by capable hardware. The hint region is cleared after each call to **SwapBuffers**. With some hardware configurations, **SwapBuffers** can ignore the hint region and exchange the entire buffer. **SwapBuffers** is implemented by the system, not by the application.

OpenGL maintains a separate hint region for each window. When you call **glAddSwapHintRectWIN** on any rendering contexts associated with a window, the hint rectangles are combined into a single region.

Call **glAddSwapHintRectWIN** with a bounding rectangle for each object drawn for a frame and for each rectangle cleared to erase previous frame objects.

Note
Note The **glAddSwapHintRectWIN** function is an extension function that is not part of the standard OpenGL library but is part of the GL_WIN_swap_hint extension. To check whether your implementation of OpenGL supports **glAddSwapHintRectWIN**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_WIN_swap_hint, **glAddSwapHintRectWIN** is supported. To obtain the address of an extension function, call **wglGetProcAddress**.

See Also

PIXELFORMATDESCRIPTOR, **SwapBuffers**, **glGetString**

glArrayElementEXT

The **glArrayElementEXT** function specifies the array elements used to render a vertex.

void glArrayElementEXT(
  GLint *index*
  );

Parameters

index
  Specifies an index in the enabled arrays.

Remarks

Use the **glArrayElementEXT** function within **glBegin** and **glEnd** pairs to specify vertex and attribute data for point, line and polygon primitives. The **glArrayElementEXT** function specifies the data for a single vertex using vertex and attribute data located at the *index* of the enabled vertex arrays.

You can use **glArrayElementEXT** to construct primitives by indexing vertex data, rather than by streaming through arrays of data in first-to-last order. Because **glArrayElementEXT** specifies a single vertex only, you can explicitly specify attributes for individual primitives. For example, you can set a single normal for each individual triangle.

When you include calls to **glArrayElementEXT** in display lists, the necessary array data, determined by the array pointers and enable values, is entered in the display list also. Array pointer and enable values are determined when display lists are created, not when display lists are executed.

The implementation can read and cache static array data at any time with **glArrayElementEXT**. When you modify the elements of a static array without specifying the array again, the results of any subsequent calls to **glArrayElementEXT** are undefined.

Note The **glArrayElementEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_vertex_array extension. To check whether your implementation of OpenGL supports **glArrayElementEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glArrayElementEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

When you call **glArrayElementEXT** without first calling **glEnable**(**GL_VERTEX_ARRAY_EXT**), no drawing occurs but the attributes corresponding to enabled arrays are modified. Although no error is generated when you specify an array within **glBegin** and **glEnd** pairs, the results are undefined.

See Also

glColorPointerEXT, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glColorPointerEXT

The **glColorPointerEXT** function defines an array of colors.

void glColorPointerEXT(
GLint *size*,
GLenum *type*,
GLsizei *stride*,
GLsizei *count*,
const GLvoid **pointer*
);

Parameters

size
Specifies the number of components per color. The value must be either 3 or 4.

type
Specifies the data type of each color component in a color array. Acceptable data types are specified with the following constants: **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT**, **GL_FLOAT**, or **GL_DOUBLE_EXT**.

stride
Specifies the byte offset between consecutive colors. When *stride* is zero, the colors are tightly packed in the array.

count
Specifies the number of static colors, counting from the first color.

pointer
Specifies a pointer to the first component of the first color element in a color array.

Remarks

The **glColorPointerEXT** function specifies the location and data format of an array of color components to use when rendering. The *stride* parameter determines the byte offset from one color to the next, enabling the packing of vertex atributes in a single array or storage in separate arrays. In some implementations storing vertex attributes in a single array can be more efficient than the use of separate arrays. Starting from the first color array element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static color array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

The color array is enabled when you specify the **GL_COLOR_ARRAY_EXT** constant with **glEnable**. When enabled in this way, the color array is used when you call **glDrawArraysEXT** or **glArrayElementEXT**. By default, the color array is disabled. **glColorPointerEXT** calls are not entered in display lists.

When you specify a color array using **glColorPointerEXT**, the values of all the function's color array parameters are saved in a client-side state and static array elements can be cached. Because the color array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you specify the color array within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note The **glColorPointerEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_vertex_array extension. To check whether your implementation of OpenGL supports **glColorPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glColorPointerEXT** is supported. To obtain the function address of an extension function, call **wglGetProcAddress**.

The following functions retrieve information related to the **glColorPointerEXT** function:

glIsEnabled with argument **GL_COLOR_ARRAY_EXT**

glGet with argument **GL_COLOR_ARRAY_SIZE_EXT**

glGet with argument **GL_COLOR_ARRAY_TYPE_EXT**

glGet with argument **GL_COLOR_ARRAY_STRIDE_EXT**

glGet with argument **GL_COLOR_ARRAY_COUNT_EXT**

glGetPointervEXT with argument **GL_COLOR_ARRAY_POINTER_EXT**

Errors

GL_INVALID_VALUE is generated if *size* is not 3 or 4.

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString** **wglGetProcAddress**

glDrawArraysEXT

The **glDrawArraysEXT** function specifies multiple primitives to render.

void glDrawArraysEXT(
  GLenum *mode*,
  GLint *first*,
  GLsizei *count*
  );

Parameters

mode
  Specifies the kind of primitives to render. Acceptable types of primitives are specified with the following 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**.

first
  Specifies the starting index in the enabled arrays.

count
  Specifies the number of indexes to render.

Remarks

The **glDrawArraysEXT** function enables you to specify multiple geometric primitives to render. Instead of calling separate OpenGL functions to pass each individual vertex, normal, or color, you can specify separate arrays of vertexes, normals, and colors to define a sequence of primitives (all the same kind) with a single call to **glDrawArraysEXT**.

When you call **glDrawArraysEXT**, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with the *first* element. The *mode* parameter specifies what kind of primitive to construct and how to use the array elements to construct the primitives.

After **glDrawArraysEXT** returns, the values of vertex attributes that are modified by **glDrawArraysEXT** are undefined. For example, if **GL_COLOR_ARRAY_EXT** is enabled, the value of the current color is undefined after **glDrawArraysEXT** returns. Attributes not modified by **glDrawArraysEXT** remain defined. When **GL_VERTEX_ARRAY_EXT** is not enabled, no geometric primitives are generated but the attributes corresponding to enabled arrays are modified.

You can include **glDrawArraysEXT** in display lists. When you include **glDrawArraysEXT** in a display list, the necessary array data, determined by the array pointers and the enables, are generated and entered in the display list. The values of array pointers and enables are determined during the creation of display lists.

Static array data can be read at any time. If any static array elements are modified and the array is not specified again, the results of any subsequent calls to **glDrawArraysEXT** are undefined.

Although no error is generated when you specify an array more than once within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note The **glDrawArraysEXT** function is an extension function that is not part of the standard OpenGL library but is part of the GL_EXT_vertex_array extension. To check whether your implementation of OpenGL supports **glDrawArraysEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glDrawArraysEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

Errors

GL_INVALID_VALUE is generated if *count* is negative.

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

GL_INVALID_OPERATION is generated if **glDrawArraysEXT** is called between **glBegin** and **glEnd** pairs.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glEdgeFlagPointerEXT

The **glEdgeFlagPointerEXT** function defines an array of edge flags.

void glEdgeFlagPointerEXT(
  GLsizei *stride*,
  GLsizei *count*,
  const GLboolean **pointer*
  );

Parameters

stride
  Specifies the byte offset between consecutive edge flags. When *stride* is zero, the edge flags are tightly packed in the array.

count
  Specifies the number of edge flags, counting from the first, that are static.

pointer
  Specifies a pointer to the first edge flag in the array.

Remarks

The **glEdgeFlagPointerEXT** function specifies the location and data of an array of Boolean edge flags to use when rendering. The *stride* parameter determines the byte offset from one edge flag to the next, which enables the packing of vertexes and attributes in a single array or storage in separate arrays. In some implementations storing the vertexes and attributes in a single array can be more efficient than using separate arrays. Starting from the first edge-flag array element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

Static array data can be read at any time. If any static array elements are modified and the array is not specified again, the results of any subsequent calls to **glEdgeFlagPointerEXT** are undefined.

An edge-flag array is enabled when you specify the **GL_EDGE_FLAG_ARRAY_EXT** constant with **glEnable**. When enabled, **glDrawArraysEXT** and **glArrayElementEXT** use the edge-flag array. By default the edge-flag array is disabled.

You cannot include **glEdgeFlagPointerEXT** in display lists.

When you specify an edge-flag array using **glEdgeFlagPointerEXT**, the values of all the function's edge-flag array parameters are saved in a client-side state and static array elements can be cached. Because the edge-flag array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glEdgeFlagPointerEXT** within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note The **glEdgeFlagPointerEXT** function is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glEdgeFlagPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glEdgeFlagPointerEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

The following functions retrieve information related to the **glEdgeFlagPointerEXT** function:

glIsEnabled with argument **GL_EDGE_FLAG_ARRAY_EXT**

glGet with argument **GL_EDGE_FLAG_ARRAY_STRIDE_EXT**

glGet with argument **GL_EDGE_FLAG_ARRAY_COUNT_EXT**

glGetPointervEXT with argument **GL_EDGE_FLAG_ARRAY_POINTER_EXT**

Errors

GL_INVALID_ENUM is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glGetPointervEXT

The **glGetPointervEXT** function returns the address of a vertex data array.

void glGetPointervEXT(
  GLenum *pname*,
  GLvoid **params*
  );

Parameters

pname
  Specifies the type of array pointer to return from the following list symbolic constants:
  GL_VERTEX_ARRAY_POINTER_EXT, **GL_NORMAL_ARRAY_POINTER_EXT**,
  GL_COLOR_ARRAY_POINTER_EXT, **GL_INDEX_ARRAY_POINTER_EXT**,
  GL_TEXTURE_COORD_ARRAY_POINTER_EXT, **GL_EDGE_FLAG_ARRAY_POINTER_EXT**

params
  Returns the value of the array pointer specified by *pname*.

Remarks

The **glGetPointervEXT** function returns array pointer information. The *pname* parameter is a symbolic constant specifying the kind of array pointer to return and *params* is a pointer to a location to place the returned data.

Note The **glGetPointervEXT** function is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glGetPointervEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glGetPointervEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glIndexPointerEXT

The **glIndexPointerEXT** function defines an array of color indexes.

void glIndexPointerEXT(
  GLenum *type*,
  GLsizei *stride*,
  GLsizei *count*,
  const GLvoid **pointer*
  );

Parameters

type
  Specifies the datatype of each color index in the array using the following symbolic constants: **GL_SHORT**, **GL_INT**, **GL_FLOAT**, **GL_DOUBLE_EXT**

stride
  Specifies the byte offset between consecutive color indexes. When *stride* is zero, the color indexes are tightly packed in the array.

count
  Specifies the number of color indexes, counting from the first, that are static.

pointer
  Specifies a pointer to the first color index in the array.

Remarks

The **glIndexPointerEXT** function specifies the location and data of an array of color indexes to use when rendering. The *type* parameter specifies the datatype of each color index and *stride* determines the byte offset from one color index to the next, enabling the packing of vertexes and attributes in a single array or storage in separate arrays. In some implementations storing the vertexes and attributes in a single array can be more efficient than using separate arrays. Starting from the first color index element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

A color-index array is enabled when you specify the **GL_INDEX_ARRAY_EXT** constant with **glEnable**. When enabled, **glDrawArraysEXT** and **glArrayElementEXT** use the color index array. By default the color-index array is disabled.

You cannot include **glIndexPointerEXT** in display lists.

When you specify a color-index array using **glIndexPointerEXT**, the values of all the function's color-index array parameters are saved in a client-side state and static array elements can be cached. Because the color-index array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glIndexPointerEXT** within **glBegin** and **glEnd** pairs, the results are undefined.

Note The **glIndexPointerEXT** function is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glIndexPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns GL_EXT_vertex_array, **glIndexPointerEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

The following functions retrieve information related to the **glIndexPointerEXT** function:

glIsEnabled with argument **GL_INDEX_ARRAY_EXT**

glGet with argument **GL_INDEX_ARRAY_STRIDE_EXT**

glGet with argument **GL_INDEX_ARRAY_COUNT_EXT**

glGet with argument **GL_INDEX_ARRAY_TYPE_EXT**

glGet with argument **GL_INDEX_ARRAY_SIZE_EXT**

glGetPointervEXT with argument **GL_INDEX_ARRAY_POINTER_EXT**

Errors

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glNormalPointerEXT

The **glNormalPointerEXT** function defines an array of normals.

void glNormalPointerEXT(
  GLenum *type*,
  GLsizei *stride*,
  GLsizei *count*,
  const GLvoid **pointer*
  );

Parameters

type
  Specifies the datatype of each coordinate in the array using the following symbolic constants: **GL_BYTE**, **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE_EXT**.

stride
  Specifies the byte offset between consecutive normals. When *stride* is zero, the normals are tightly packed in the array.

count
  Specifies the number of normals, counting from the first, that are static.

pointer
  Specifies a pointer to the first normal in the array.

Remarks

The **glNormalPointerEXT** function specifies the location and data of an array of normals to use when rendering. The *type* parameter specifies the datatype of each normal coordinate and *stride* determines the byte offset from one normal to the next, enabling the packing of vertexes and attributes in a single array or storage in separate arrays. In some implementations storing the vertexes and attributes in a single array can be more efficient than using separate arrays. Starting from the first normal element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

A normal arrays is enabled when you specify the **GL_NORMAL_ARRAY_EXT** constant with **glEnable**. When enabled, **glDrawArraysEXT** and **glArrayElementEXT** use the normal array. By default the normal array is disabled.

You cannot include **glNormalPointerEXT** in display lists.

When you specify a normal array using **glNormalPointerEXT**, the values of all the function's normal array parameters are saved in a client-side state and static array elements can be cached. Because the normal array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glNormalPointerEXT** within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note The **glNormalPointerEXT** function is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glNormalPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns **GL_EXT_vertex_array**, **glNormalPointerEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

The following functions are associated with the **glNormalPointerEXT** function:

glIsEnabled with argument **GL_NORMAL_ARRAY_EXT**

glGet with argument **GL_NORMAL_ARRAY_STRIDE_EXT**

glGet with argument **GL_NORMAL_ARRAY_COUNT_EXT**

glGet with argument **GL_NORMAL_ARRAY_TYPE_EXT**

glGetPointervEXT with argument **GL_NORMAL_ARRAY_POINTER_EXT**

Errors

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glTexCoordPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glTexCoordPointerEXT

The **glTexCoordPointerEXT** function defines an array of texture coordinates.

void glTexCoordPointerEXT(
  GLint *size*,
  GLenum *type*,
  GLsizei *stride*,
  GLsizei *count*,
  const GLvoid **pointer*
  );

Parameters

size
  Specifies the number of coordinates per array element. The value of *size* must be 1, 2, 3, or 4.

type
  Specifies the datatype of each texture coordinate in the array using the following symbolic constants: **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE_EXT**.

stride
  Specifies the byte offset between consecutive array elements. When *stride* is zero, the array elements are tightly packed in the array.

count
  Specifies the number of array elements, counting from the first, that are static.

pointer
  Specifies a pointer to the first coordinate of the first element in the array.

Remarks

The **glTexCoordPointerEXT** function specifies the location and data of an array of texture coordinates to use when rendering.The *size* parameter specifies the number of coordinates used for each element of the array.The *type* parameter specifies the datatype of each texture coordinate and the *stride* parameter determines the byte offset from one array element to the next, enabling the packing of vertexes and attributes in a single array or storage in separate arrays. In some implementations storing the vertexes and attributes in a single array can be more efficient than using separate arrays. Starting from the first array element, *count* indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

A texture coordinate array is enabled when you specify the **GL_TEXTURE_COORD_ARRAY_EXT** constant with **glEnable**. When enabled, **glDrawArraysEXT** and **glArrayElementEXT** use the texture coordinate array. By default the texture coordinate array is disabled.

You cannot include **glTexCoordPointerEXT** in display lists.

When you specify a texture coordinate array using **glTexCoordPointerEXT**, the values of all the function's texture coordinate array parameters are saved in a client-side state, and static array elements can be cached. Because the texture coordinate array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glTexCoordPointerEXT** within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note The **glTexCoordPointerEXT** function is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glTexCoordPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If

it returns **GL_EXT_vertex_array**, **glTexCoordPointerEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

The following functions retrieve information related to the **glTexCoordPointerEXT** function:

glIsEnabled with argument **GL_TEXTURE_COORD_ARRAY_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_SIZE_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_STRIDE_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_COUNT_EXT**

glGet with argument **GL_TEXTURE_COORD_ARRAY_TYPE_EXT**

glGetPointervEXT with argument **GL_TEXTURE_COORD_ARRAY_POINTER_EXT**

Errors

GL_INVALID_VALUE is generated if *size* is not 1, 2, 3, or 4.

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glVertexPointerEXT**, **glGetString**, **wglGetProcAddress**

glVertexPointerEXT

The **glVertexPointerEXT** function defines an array of vertex data.

void glVertexPointerEXT(
  GLint *size*,
  GLenum *type*,
  GLsizei *stride*,
  GLsizei *count*,
  const GLvoid **pointer*
  );

Parameters

size
  Specifies the number of coordinates per vertex. The value of *size* must be 2, 3, or 4.

type
  Specifies the datatype of each coordinate in the array using the following symbolic constants: **GL_SHORT**, **GL_INT**, **GL_FLOAT**, and **GL_DOUBLE_EXT**.

stride
  Specifies the byte offset between consecutive vertexes. When *stride* is zero, the vertexes are tightly packed in the array.

count
  Specifies the number of vertexes, counting from the first, that are static.

pointer
  Specifies a pointer to the first coordinate of the first vertex in the array.

Remarks

The **glVertexPointerEXT** function specifies the location and data of an array of vertex coordinates to use when rendering. The *size* parameter specifies the number of coordinates per vertex. The *type* parameter specifies the datatype of each vertex coordinate and the *stride* parameter determines the byte offset from one vertex to the next, enabling the packing of vertexes and attributes in a single array or storage in separate arrays. In some implementations storing the vertexes and attributes in a single array can be more efficient than using separate arrays. Starting from the first vertex element, the *count* parameter indicates the total number of static elements. Your application can modify static elements, but once the elements are modified, the application must explicitly specify the array again before using the array for any rendering. Non-static array elements are not accessed until you call **glDrawArraysEXT** or **glArrayElementEXT**.

A vertex array is enabled when you specify the **GL_VERTEX_ARRAY_EXT** constant with **glEnable**. When enabled, **glDrawArraysEXT** and **glArrayElementEXT** use the vertex array. By default, the vertex array is disabled.

You cannot include **glVertexPointerEXT** in display lists.

When you specify a vertex array using **glVertexPointerEXT**, the values of all the function's vertex array parameters are saved in a client-side state and static array elements can be cached. Because the vertex array parameters are client-side state, their values are not saved or restored by **glPushAttrib** and **glPopAttrib**.

Although no error is generated when you call **glVertexPointerEXT** within **glBegin** and **glEnd** pairs, the results are undefined.

Note
Note **glVertexPointerEXT** is an extension function that is not part of the standard OpenGL library but is part of the **GL_EXT_vertex_array** extension. To check whether your implementation of OpenGL supports **glVertexPointerEXT**, call **glGetString**(**GL_EXTENSIONS**). If it returns

GL_EXT_vertex_array, **glVertexPointerEXT** is supported.

To obtain the address of an extension function, call **wglGetProcAddress**.

The following functions retrieve information related to the **glVertexPointerEXT** function:

glIsEnabled with argument **GL_VERTEX_ARRAY_EXT**

glGet with argument **GL_VERTEX_ARRAY_SIZE_EXT**

glGet with argument **GL_VERTEX_ARRAY_STRIDE_EXT**

glGet with argument **GL_VERTEX_ARRAY_COUNT_EXT**

glGet with argument **GL_VERTEX_ARRAY_TYPE_EXT**

glGetPointervEXT with argument **GL_VERTEX_ARRAY_POINTER_EXT**

Errors

GL_INVALID_VALUE is generated if *size* is not 2, 3, or 4.

GL_INVALID_ENUM is generated if *type* is not an accepted value.

GL_INVALID_VALUE is generated if *stride* or *count* is negative.

See Also

glArrayElementEXT, **glColorPointerEXT**, **glDrawArraysEXT**, **glEdgeFlagPointerEXT**, **glGetPointervEXT**, **glIndexPointerEXT**, **glNormalPointerEXT**, **glTexCoordPointerEXT**, **glGetString**, **wglGetProcAddress**

glHint

The **glHint** function specifies implementation-specific hints.

void glHint(
GLenum *target*,
GLenum *mode*
);

Parameters

target
Specifies a symbolic constant indicating the behavior to be controlled. **GL_FOG_HINT, GL_LINE_SMOOTH_HINT, GL_PERSPECTIVE_CORRECTION_HINT, GL_POINT_SMOOTH_HINT,** and **GL_POLYGON_SMOOTH_HINT** are accepted.

mode
Specifies a symbolic constant indicating the desired behavior. **GL_FASTEST**, **GL_NICEST**, and **GL_DONT_CARE** are accepted.

Remarks

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. *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
The client doesn't have a 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. Hinting **GL_NICEST** can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

GL_PERSPECTIVE_CORRECTION_HINT
Indicates the quality of color and texture 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. Hinting **GL_NICEST** can result in more pixel fragments being generated during rasterization, if a larger filter function is applied.

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. **glHint** can be ignored.

Errors

GL_INVALID_ENUM is generated if either *target* or *mode* is not an accepted value.

GL_INVALID_OPERATION is generated if **glHint** is called between a call to **glBegin** and the corresponding call to **glEnd**.

glIndex

glIndexd, glIndexf, glIndexi, glIndexs, glIndexdv, glIndexfv, glIndexiv, glIndexsv

These functions set the current color index.

void glIndexd(
  GLdouble *c*
  );

void glIndexf(
  GLfloat *c*
  );

void glIndexi(
  GLint *c*
  );

void glIndexs(
  GLshort *c*
  );

Parameters

c
  Specifies the new value for the current color index.

void glIndexdv(
  const GLdouble **c*
  );

void glIndexfv(
  const GLfloat **c*
  );

void glIndexiv(
  const GLint **c*
  );

void glIndexsv(
  const GLshort **c*
  );

Parameters

c
  Specifies a pointer to a one-element array that contains the new value for the current color index.

Remarks

The **glIndex** function 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. Integer values are converted directly to floating-point values, with no special mapping.

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, **glIndex** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to the **glIndex** function:

glGet with argument **GL_CURRENT_INDEX**

See Also

glColor

glIndexMask

The **glIndexMask** function controls the writing of individual bits in the color index buffers.

void glIndexMask(
  GLuint *mask*
  );

Parameters

mask
  Specifies a bit mask to enable and disable the writing of individual bits in the color index buffers. Initially, the mask is all ones.

Remarks

The **glIndexMask** function 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. Wherever a one appears in the mask, the corresponding bit in the color index buffer (or buffers) is made writable. Where a zero appears, the bit is write-protected.

This mask is used only in color index mode, and it affects only the buffers currently selected for writing (see **glDrawBuffer**). Initially, all bits are enabled for writing.

The following function retrieves information related to the **glIndexMask** function.

glGet with argument **GL_INDEX_WRITEMASK**

Errors

GL_INVALID_OPERATION is generated if **glIndexMask** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDepthMask, **glDrawBuffer**, **glIndex**, **glStencilMask**

glInitNames

The **glInitNames** function initializes the name stack.

void glInitNames(
  void
  );

Remarks

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. The **glInitNames** function 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 **glInitNames** while the render mode is not **GL_SELECT** are ignored.

The following functions retrieve information related to the **glInitNames** function:

glGet with argument **GL_NAME_STACK_DEPTH**

glGet with argument **GL_MAX_NAME_STACK_DEPTH**

Errors

GL_INVALID_OPERATION is generated if **glInitNames** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLoadName, **glPushName**, **glRenderMode**, **glSelectBuffer**

glIsEnabled

The **glIsEnabled** function tests whether a capability is enabled.

GLboolean glIsEnabled(
  GLenum *cap*
  );

Parameters

cap
  Specifies a symbolic constant indicating a GL capability.

Remarks

The **glIsEnabled** function returns **GL_TRUE** if *cap* is an enabled capability and returns **GL_FALSE** otherwise. The following capabilities are accepted for *cap*:

GL_ALPHA_TEST	See **glAlphaFunc**
GL_AUTO_NORMAL	See **glEvalCoord**
GL_BLEND	See **glBlendFunc**
GL_CLIP_PLANE*i*	See **glClipPlane**
GL_COLOR_MATERIAL	See **glColorMaterial**
GL_CULL_FACE	See **glCullFace**
GL_DEPTH_TEST	See **glDepthFunc** and **glDepthRange**
GL_DITHER	See **glEnable**
GL_FOG	See **glFog**
GL_LIGHT*i*	See **glLightModel** and **glLight**
GL_LIGHTING	See **glMaterial** **glLightModel** and **glLight**
GL_LINE_SMOOTH	See **glLineWidth**
GL_LINE_STIPPLE	See **glLineStipple**
GL_LOGIC_OP	See **glLogicOp**
GL_MAP1_COLOR_4	See **glMap1**
GL_MAP1_INDEX	See **glMap1**
GL_MAP1_NORMAL	See **glMap1**
GL_MAP1_TEXTURE_COORD_1	See **glMap1**
GL_MAP1_TEXTURE_COORD_2	See **glMap1**
GL_MAP1_TEXTURE_COORD_3	See **glMap1**
GL_MAP1_TEXTURE_COORD_4	See **glMap1**
GL_MAP1_VERTEX_3	See **glMap1**
GL_MAP1_VERTEX_4	See **glMap1**
GL_MAP2_COLOR_4	See **glMap2**
GL_MAP2_INDEX	See **glMap2**
GL_MAP2_NORMAL	See **glMap2**
GL_MAP2_TEXTURE_COORD_1	See **glMap2**
GL_MAP2_TEXTURE_COORD_2	See **glMap2**

GL_MAP2_TEXTURE_COORD_3	See **glMap2**
GL_MAP2_TEXTURE_COORD_4	See **glMap2**
GL_MAP2_VERTEX_3	See **glMap2**
GL_MAP2_VERTEX_4	See **glMap2**
GL_NORMALIZE	See **glNormal**
GL_POINT_SMOOTH	See **glPointSize**
GL_POLYGON_SMOOTH	See **glPolygonMode**
GL_POLYGON_STIPPLE	See **glPolygonStipple**
GL_SCISSOR_TEST	See **glScissor**
GL_STENCIL_TEST	See **glStencilFunc** and **glStencilOp**
GL_TEXTURE_1D	See **glTexImage1D**
GL_TEXTURE_2D	See **glTexImage2D**
GL_TEXTURE_GEN_Q	See **glTexGen**
GL_TEXTURE_GEN_R	See **glTexGen**
GL_TEXTURE_GEN_S	See **glTexGen**
GL_TEXTURE_GEN_T	See **glTexGen**

If an error is generated, **glIsEnabled** returns zero.

Errors

GL_INVALID_ENUM is generated if *cap* is not an accepted value.

GL_INVALID_OPERATION is generated if **glIsEnabled** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable

glIsList

The **glIsList** function tests for display-list existence.

GLboolean glIsList(
  GLuint *list*
  );

Parameters

list
  Specifies a potential display-list name.

Remarks

The **glIsList** function returns **GL_TRUE** if *list* is the name of a display list and returns **GL_FALSE** otherwise.

Errors

GL_INVALID_OPERATION is generated if **glIsList** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallList, **glCallLists**, **glDeleteLists**, **glGenLists**, **glNewList**

glLightf, glLighti, glLightfv, glLightiv

These functions set light source parameters.

void glLightf(
  GLenum *light*,
  GLenum *pname*,
  GLfloat *param*
  );

void glLighti(
  GLenum *light*,
  GLenum *pname*,
  GLint *param*
  );

Parameters

light
  Specifies a light. The number of lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form **GL_LIGHT**i where $0 \leq i <$ **GL_MAX_LIGHTS**.

pname
  Specifies a single-valued light source parameter for *light*. **GL_SPOT_EXPONENT, GL_SPOT_CUTOFF, GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION**, and **GL_QUADRATIC_ATTENUATION** are accepted.

param
  Specifies the value to which parameter *pname* of light source *light* will be set.

void glLightfv(
  GLenum *light*,
  GLenum *pname*,
  const GLfloat **params*
  );

void glLightiv(
  GLenum *light*,
  GLenum *pname*,
  const GLint **params*
  );

Parameters

light
  Specifies a light. The number of lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form **GL_LIGHT***i* where $0 \leq i <$ **GL_MAX_LIGHTS**.

pname
  Specifies a light source parameter for *light*. **GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION**, **GL_SPOT_EXPONENT**, **GL_SPOT_CUTOFF**, **GL_CONSTANT_ATTENUATION**, **GL_LINEAR_ATTENUATION**, and **GL_QUADRATIC_ATTENUATION** are accepted.

params
  Specifies a pointer to the value or values to which parameter *pname* of light source *light* will be set.

Remarks

The **glLight** function sets the values of individual light source parameters. *light* names the light and is a

symbolic name of the form

GL_LIGHT*i*, where $0 \leq i <$ **GL_MAX_LIGHTS**.

The *pname* parameter specifies one of ten light source parameters, again by symbolic name. The *params* parameter is either a single value or a pointer to an array that contains the new values.

Lighting calculation is enabled and disabled using **glEnable** and **glDisable** with argument **GL_LIGHTING**. When lighting is enabled, light sources that are enabled contribute to the lighting calculation. Light source *i* is enabled and disabled using **glEnable** and **glDisable** with argument **GL_LIGHT***i*.

The ten light parameters are as follows:

GL_AMBIENT
The *params* parameter contains four integer or floating-point values that specify the ambient RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient light intensity is (0.0, 0.0, 0.0, 1.0).

GL_DIFFUSE
The *params* parameter contains four integer or floating-point values that specify the diffuse RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).

GL_SPECULAR
The *params* parameter contains four integer or floating-point values that specify the specular RGBA intensity of the light. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).

GL_POSITION
The *params* parameter contains four integer or floating-point values that specify the position of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The position is transformed by the modelview matrix when **glLight** is called (just as if it were a point), and it is stored in eye coordinates. If the *w* component of the position is 0.0, the light is treated as a directional source. Diffuse and specular lighting calculations take the lights 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 default position is (0,0,1,0); thus, the default light source is directional, parallel to, and in the direction of the -*z* axis.

GL_SPOT_DIRECTION
The *params* parameter contains three integer or floating-point values that specify the direction of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly. Neither integer nor floating-point values are clamped.

The spot direction is transformed by the inverse of the modelview matrix when **glLight** is called (just as it it were a normal), and it is stored in eye coordinates. It is significant only when **GL_SPOT_CUTOFF** is not 180, which it is by default. The default direction is (0,0,-1).

GL_SPOT_EXPONENT
The *params* parameter is a single integer or floating-point value that specifies the intensity distribution of the light. Integer and floating-point values are mapped directly. 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 the following paragraph). The default spot exponent is 0, resulting in uniform light distribution.

GL_SPOT_CUTOFF

The *params* parameter is a single integer or floating-point value that specifies the maximum spread angle of a light source. Integer and floating-point values are mapped directly. 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 default spot cutoff is 180, resulting in uniform light distribution.

GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION

The *params* parameter is a single integer or floating-point value that specifies one of the three light attenuation factors. Integer and floating-point values are mapped directly. Only nonnegative 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 default attenuation factors are (1,0,0), resulting in no attenuation.

It is always the case that **GL_LIGHT***i* = **GL_LIGHT0** + *i*.

The following functions retrieve information related to the **glLight** function:

glGetLight

glIsEnabled with argument **GL_LIGHTING**

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 **glLight** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColorMaterial, **glLightModel**, **glMaterial**

glLightModelf, glLightModeli, glLightModelfv, glLightModeliv

These functions set the lighting model parameters.

void glLightModelf(
  GLenum *pname*,
  GLfloat *param*
  );

void glLightModeli(
  GLenum *pname*,
  GLint *param*
  );

Parameters

pname
  Specifies a single-valued lighting model parameter. **GL_LIGHT_MODEL_LOCAL_VIEWER** and **GL_LIGHT_MODEL_TWO_SIDE** are accepted.

param
  Specifies the value to which *param* will be set.

void glLightModelfv(
  GLenum *pname*,
  const GLfloat **params*
  );

void glLightModeliv(
  GLenum *pname*,
  const GLint **params*
  );

Parameters

pname
  Specifies a lighting model parameter. **GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER,** and **GL_LIGHT_MODEL_TWO_SIDE** are accepted.

params
  Specifies a pointer to the value or values to which *params* will be set.

Remarks

The **glLightModel** function 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
  The *params* parameter contains four integer or floating-point values that specify the ambient RGBA intensity of the entire scene. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient scene intensity is (0.2, 0.2, 0.2, 1.0).

GL_LIGHT_MODEL_LOCAL_VIEWER
  The *params* parameter is a single integer or floating-point value that specifies how specular reflection angles are computed. If *params* is 0 (or 0.0), 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 default is 0.

GL_LIGHT_MODEL_TWO_SIDE

The *params* parameter is a single integer or 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 *params* is 0 (or 0.0), 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, vertexes of back-facing polygons are lighted using the *back* material parameters, and have their normals reversed before the lighting equation is evaluated. Vertexes of front-facing polygons are always lighted using the *front* material parameters, with no change to their normals. The default 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 lights ambient intensity. The diffuse light source contribution is the product of the material diffuse reflectance, the lights 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 lights 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 zero 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 **glMaterial** 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.

The following functions retrieve information related to the **glLightModel** function:

glGet with argument **GL_LIGHT_MODEL_AMBIENT**

glGet with argument **GL_LIGHT_MODEL_LOCAL_VIEWER**

glGet with argument **GL_LIGHT_MODEL_TWO_SIDE**

glIsEnabled with argument **GL_LIGHTING**

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glLightModel** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLight, **glMaterial**

glLineStipple

The **glLineStipple** function specifies the line stipple pattern.

void glLineStipple(
  GLint *factor***,**
  GLushort *pattern*
  );

Parameters

factor
  Specifies a multiplier for each bit in the line stipple pattern. If *factor* is 3, for example, each bit in the pattern will be used three times before the next bit in the pattern is used. *factor* is clamped to the range [1, 256] and defaults to one.

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, and the default pattern is all ones.

Remarks

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 zero whenever **glBegin** is called, and before each line segment of a **glBegin**(**GL_LINES**)/**glEnd** 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*) mod 16 is zero, 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 1x*width* rectangles for purposes of stippling. Rectangle *s* is rasterized or not based on the fragment rule described for aliased lines, counting rectangles rather than groups of fragments.

Line stippling is enabled or disabled using **glEnable** and **glDisable** 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 ones. Initially, line stippling is disabled.

The following functions retrieve information related to the **glLineStipple** function:

glGet with argument **GL_LINE_STIPPLE_PATTERN**

glGet with argument **GL_LINE_STIPPLE_REPEAT**

glIsEnabled with argument **GL_LINE_STIPPLE**

Errors

GL_INVALID_OPERATION is generated if **glLineStipple** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLineWidth, **glPolygonStipple**

glLineWidth

The **glLineWidth** function specifies the width of rasterized lines.

void glLineWidth(
  GLfloat *width*
  );

Parameters

width
  Specifies the width of rasterized lines. The default is 1.0.

Remarks

The **glLineWidth** function specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1.0 has different effects, depending on whether line antialiasing is enabled. Line antialiasing is controlled by calling **glEnable** and **glDisable** with argument **GL_LINE_SMOOTH**.

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 $|\Delta x| \geq |\Delta y|$, *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 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported widths and the size difference between supported widths within the range can be queried by calling **glGet** with arguments **GL_LINE_WIDTH_RANGE** and **GL_LINE_WIDTH_GRANULARITY**.

The line width specified by **glLineWidth** is always returned when **GL_LINE_WIDTH** is queried. Clamping and rounding for aliased and antialiased lines have no effect on the specified value.

Non-antialiased 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.

The following functions retrieve information related to the **glLineWidth** function:

glGet with argument **GL_LINE_WIDTH**

glGet with argument **GL_LINE_WIDTH_RANGE**

glGet with argument **GL_LINE_WIDTH_GRANULARITY**

glIsEnabled with argument **GL_LINE_SMOOTH**

Errors

GL_INVALID_VALUE is generated if *width* is less than or equal to zero.

GL_INVALID_OPERATION is generated if **glLineWidth** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable

glListBase

The **glListBase** function sets the display-list base for **glCallLists.**

void glListBase(
  GLuint *base*
  );

Parameters

base
  Specifies an integer offset that will be added to **glCallLists** offsets to generate display-list names. Initial value is zero.

Remarks

The **glListBase** function 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.

The following function retrieves information related to the **glListBase** function:

glGet with argument **GL_LIST_BASE**

Errors

GL_INVALID_OPERATION is generated if **glListBase** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallLists

glLoadIdentity

The **glLoadIdentity** function replaces the current matrix with the identity matrix.

void glLoadIdentity(
  void
  );

Remarks

The **glLoadIdentity** function replaces the current matrix with the identity matrix. It is semantically equivalent to calling **glLoadMatrix** with the identity matrix

{ewc msdncd, EWGraphic, group10327 0 /a "SDK.bmp"}

but in some cases it is more efficient.

The following functions retrieve information related to the **glLoadIdentity** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glLoadIdentity** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLoadMatrix, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

glLoadMatrixd, glLoadMatrixf

The **glLoadMatrixd** and **glLoadMatrixf** functions replace the current matrix with an arbitrary matrix.

void glLoadMatrixd(
  const GLdouble **m*
  );

void glLoadMatrixf(
  const GLfloat **m*
  );

Parameters

m
  Specifies a pointer to a 4 times 4 matrix stored in column-major order as sixteen consecutive values.

Remarks

The **glLoadMatrix** function replaces the current matrix with the one specified in *m*. The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as follows:

{ewc msdncd, EWGraphic, group10328 0 /a "SDK.bmp"}

The following functions retrieve information related to the **glLoadMatrix** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glLoadMatrix** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLoadIdentity, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**

glLoadName

The **glLoadName** function loads a name onto the name stack.

void glLoadName(
  GLuint *name*
  );

Parameters

name
  Specifies a name that will replace the top value on the name stack.

Remarks

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. The **glLoadName** function causes *name* to replace the value on the top of the name stack, which is initially empty.

The name stack is always empty while the render mode is not **GL_SELECT**. Calls to **glLoadName** while the render mode is not **GL_SELECT** are ignored.

The following functions retrieve information related to the **glLoadName** function:

glGet with argument **GL_NAME_STACK_DEPTH**

glGet with argument **GL_MAX_NAME_STACK_DEPTH**

Errors

GL_INVALID_OPERATION is generated if **glLoadName** is called while the name stack is empty.

GL_INVALID_OPERATION is generated if **glLoadName** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glInitNames, **glPushName**, **glRenderMode**, **glSelectBuffer**

glLogicOp

The **glLogicOp** function specifies a logical pixel operation for color index rendering.

void glLogicOp(
  GLenum *opcode*
  );

Parameters

opcode
  Specifies a symbolic constant that selects a logical operation. The following symbols are accepted: **GL_CLEAR, GL_SET, GL_COPY, GL_COPY_INVERTED, GL_NOOP, GL_INVERT, GL_AND, GL_NAND, GL_OR, GL_NOR, GL_XOR, GL_EQUIV, GL_AND_REVERSE, GL_AND_INVERTED, GL_OR_REVERSE,** and **GL_OR_INVERTED**.

Remarks

The **glLogicOp** function specifies a logical operation that, when enabled, is applied between the incoming color index and the color index at the corresponding location in the frame buffer. The logical operation is enabled or disabled with **glEnable** and **glDisable** using the symbolic constant **GL_LOGIC_OP**.

The *opcode* parameter is a symbolic constant chosen from the list below. 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.

Opcode	Resulting Value
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

Logical pixel operations are not applied to RGBA color buffers.

When more than one color index buffer is enabled for drawing, logical operations are done separately for each enabled buffer, using for the destination index the contents of that buffer (see **glDrawBuffer**).

The *opcode* parameter must be one of the sixteen accepted values. Other values result in an error.

The following functions retrieve information related to the **glLogicOp** function:

glGet with argument **GL_LOGIC_OP_MODE**

glIsEnabled with argument **GL_LOGIC_OP**

Errors

GL_INVALID_ENUM is generated if *opcode* is not an accepted value.

GL_INVALID_OPERATION is generated if **glLogicOp** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBlendFunc**, **glDrawBuffer**, **glEnable**, **glStencilOp**

glMap1d, glMap1f

The **glMap1d** and **glMap1f** functions define a one-dimensional evaluator.

void glMap1d(
  GLenum *target*,
  GLdouble *u1*,
  GLdouble *u2*,
  GLint *stride*,
  GLint *order*,
  const GLdouble **points*
  );

void **glMap1f**(
  GLenum *target*,
  GLfloat *u1*,
  GLfloat *u2*,
  GLint *stride*,
  GLint *order*,
  const GLfloat **points*
  );

Parameters

target
  Specifies the kind of values that are generated by the evaluator. Symbolic constants **GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP1_INDEX, GL_MAP1_COLOR_4, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2**, **GL_MAP1_TEXTURE_COORD_3**, and **GL_MAP1_TEXTURE_COORD_4** are accepted.

u1, u2
  Specify a linear mapping of u, as presented to **glEvalCoord1**, to $\hat{u}$, the variable that is evaluated by the equations specified by this command.

stride
  Specifies the number of floats or doubles between the beginning of one control point and the beginning of the next one in the data structure referenced in *points*. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

order
  Specifies the number of control points. Must be positive.

points
  Specifies a pointer to the array of control points.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertexes, 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 **glVertex**, **glNormal**, **glTexCoord**, and **glColor** 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, including B-splines, Bezier curves, Hermite splines, and so on.

Evaluators define curves based on Bernstein polynomials. Define $\mathbf{p}(\hat{u})$ as

{ewc msdncd, EWGraphic, group10331 0 /a "SDK.bmp"}

where **R** (i) is a control point and $B_i^n(\hat{u})$ is the *i*th Bernstein polynomial of degree *n* (*order* = *n* + 1):

{ewc msdncd, EWGraphic, group10331 1 /a "SDK.bmp"}

Recall that

{ewc msdncd, EWGraphic, group10331 2 /a "SDK.bmp"}

The **glMap1** function 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 **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target* described below. **glEvalCoord1** evaluates the one-dimensional maps that are enabled. When **glEvalCoord1** presents a value *u*, the Bernstein functions are evaluated using $\hat{u}$, where

{ewc msdncd, EWGraphic, group10331 3 /a "SDK.bmp"}

The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

GL_MAP1_VERTEX_3
Each control point is three floating-point values representing *x, y,* and *z*. Internal **glVertex3** 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 **glVertex4** 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 **glIndex** commands are generated when the map is evaluated. The current index is not updated with the value of these **glIndex** commands, however.

GL_MAP1_COLOR_4
Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** commands are generated when the map is evaluated. The current color is not updated with the value of these **glColor4** commands, however.

GL_MAP1_NORMAL
Each control point is three floating-point values representing the *x, y,* and *z* components of a normal vector. Internal **glNormal** commands are generated when the map is evaluated. The current normal is not updated with the value of these **glNormal** commands, however.

GL_MAP1_TEXTURE_COORD_1
Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP1_TEXTURE_COORD_2
Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP1_TEXTURE_COORD_3
Each control point is three floating-point values representing the *s, t,* and *r* texture coordinates. Internal **glTexCoord3** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP1_TEXTURE_COORD_4

Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

The *stride*, *order*, and *points* parameters define the array addressing for accessing the control points. *points* is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. *order* is the number of control points in the array. *stride* tells how many float or double locations to advance the internal memory pointer to reach the next control point.

As is the case with all GL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap1** before it returned. Changes to the contents of *points* have no effect after **glMap1** is called.

The following functions retrieve information related to the **glMap1** function:

glGetMap

glGet with argument **GL_MAX_EVAL_ORDER**

glIsEnabled with argument **GL_MAP1_VERTEX_3**

glIsEnabled with argument **GL_MAP1_VERTEX_4**

glIsEnabled with argument **GL_MAP1_INDEX**

glIsEnabled with argument **GL_MAP1_COLOR_4**

glIsEnabled with argument **GL_MAP1_NORMAL**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_1**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_2**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_3**

glIsEnabled with argument **GL_MAP1_TEXTURE_COORD_4**

Errors

GL_INVALID_ENUM is generated if *target* is not an accepted value.

GL_INVALID_VALUE is generated if *u1* is equal to *u2*.

GL_INVALID_VALUE is generated if *stride* is less than the number of values in a control point.

GL_INVALID_VALUE is generated if *order* is less than one or greater than **GL_MAX_EVAL_ORDER**.

GL_INVALID_OPERATION is generated if **glMap1** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glEnable**, **glEvalCoord**, **glEvalMesh**, **glEvalPoint**, **glMap2**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glMap2d, glMap2f

The **glMap2d** and **glMap2f** functions define a two-dimensional evaluator.

void glMap2d(
  GLenum *target*,
  GLdouble *u1*,
  GLdouble *u2*,
  GLint *ustride*,
  GLint *uorder*,
  GLdouble *v1*,
  GLdouble *v2*,
  GLint *vstride*,
  GLint *vorder*,
  const GLdouble **points*
  );

void glMap2f(
  GLenum *target*,
  GLfloat *u1*,
  GLfloat *u2*,
  GLint *ustride*,
  GLint *uorder*,
  GLfloat *v1*,
  GLfloat *v2*,
  GLint *vstride*,
  GLint *vorder*,
  const GLfloat **points*
  );

Parameters

target
  Specifies the kind of values that are generated by the evaluator. Symbolic constants **GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4, GL_MAP2_INDEX, GL_MAP2_COLOR_4, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2**, **GL_MAP2_TEXTURE_COORD_3**, and **GL_MAP2_TEXTURE_COORD_4** are accepted.

u1, u2
  Specify a linear mapping of *u*, as presented to **glEvalCoord2**, to $\hat{u}$, one of the two variables that is evaluated by the equations specified by this command.

ustride
  Specifies the number of floats or doubles between the beginning of control point $\mathbf{R}_{(ij)}$ and the beginning of control point $\mathbf{R}_{(i+1)j}$, where *i* and *j* are the *u* and *v* control point indices, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

uorder
  Specifies the dimension of the control point array in the *u* axis. Must be positive.

v1, v2
  Specify a linear mapping of *v*, as presented to **glEvalCoord2**, to $\hat{v}$, one of the two variables that is evaluated by the equations specified by this command.

vstride
  Specifies the number of floats or doubles between the beginning of control point $\mathbf{R}_{(ij)}$ and the beginning of control point $\mathbf{R}_{(i(j+1))}$, where *i* and *j* are the *u* and *v* control point indices, respectively. This allows control points to be embedded in arbitrary data structures. The only constraint is that the values for a particular control point must occupy contiguous memory locations.

vorder
Specifies the dimension of the control point array in the *v* axis. Must be positive.

points
Specifies a pointer to the array of control points.

Remarks

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertexes, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of GL processing just as if they had been presented using **glVertex**, **glNormal**, **glTexCoord**, and **glColor** 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 surfaces used in computer graphics, including B-spline surfaces, NURBS surfaces, Bezier surfaces, and so on.

Evaluators define surfaces based on bivariate Bernstein polynomials. Define **p**($\hat{u}$,$\hat{v}$) as

{ewc msdncd, EWGraphic, group10332 0 /a "SDK.bmp"}

where **R** $_{(ij)}$ is a control point, B_i^n ($\hat{u}$) is the *i*th Bernstein polynomial of degree

n (*uorder* = *n* + 1)

{ewc msdncd, EWGraphic, group10332 1 /a "SDK.bmp"}

and ($\hat{v}$) is the *j*th Bernstein polynomial of degree *m* (*vorder* = *m* + 1)

{ewc msdncd, EWGraphic, group10332 2 /a "SDK.bmp"}

Recall that

{ewc msdncd, EWGraphic, group10332 3 /a "SDK.bmp"}

The **glMap2** function 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 **glEnable** and **glDisable** with the map name, one of the nine predefined values for *target*, described below. When **glEvalCoord2** presents values *u* and *v*, the bivariate Bernstein polynomials are evaluated using $\hat{u}$ and $\hat{v}$, where

{ewc msdncd, EWGraphic, group10332 4 /a "SDK.bmp"}

{ewc msdncd, EWGraphic, group10332 5 /a "SDK.bmp"}

The *target* parameter is a symbolic constant that indicates what kind of control points are provided in *points*, and what output is generated when the map is evaluated. It can assume one of nine predefined values:

GL_MAP2_VERTEX_3
Each control point is three floating-point values representing *x, y,* and *z*. Internal **glVertex3** 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 **glVertex4** 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 **glIndex** commands are generated when the map is evaluated. The current index is not updated with the value of these **glIndex** commands, however.

GL_MAP2_COLOR_4
Each control point is four floating-point values representing red, green, blue, and alpha. Internal **glColor4** commands are generated when the map is evaluated. The current color is not updated with the value of these **glColor4** commands, however.

GL_MAP2_NORMAL
Each control point is three floating-point values representing the *x, y,* and *z* components of a normal vector. Internal **glNormal** commands are generated when the map is evaluated. The current normal is not updated with the value of these **glNormal** commands, however.

GL_MAP2_TEXTURE_COORD_1
Each control point is a single floating-point value representing the *s* texture coordinate. Internal **glTexCoord1** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_2
Each control point is two floating-point values representing the *s* and *t* texture coordinates. Internal **glTexCoord2** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_3
Each control point is three floating-point values representing the *s, t,* and *r* texture coordinates. Internal **glTexCoord3** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

GL_MAP2_TEXTURE_COORD_4
Each control point is four floating-point values representing the *s, t, r,* and *q* texture coordinates. Internal **glTexCoord4** commands are generated when the map is evaluated. The current texture coordinates are not updated with the value of these **glTexCoord** commands, however.

The *ustride*, *uorder*, *vstride*, *vorder*, and *points* parameters define the array addressing for accessing the control points. *points* is the location of the first control point, which occupies one, two, three, or four contiguous memory locations, depending on which map is being defined. There are *uorder*x*vorder* control points in the array. *ustride* tells how many float or double locations are skipped to advance the internal memory pointer from control point $\mathbf{R}_{(i\,j)}$ to control point $\mathbf{R}_{((i+1)j)}$. *vstride* tells how many float or double locations are skipped to advance the internal memory pointer from control point $\mathbf{R}_{(i\,j)}$ to control point $\mathbf{R}_{((i(j+1))}$.

As is the case with all GL commands that accept pointers to data, it is as if the contents of *points* were copied by **glMap2** before it returned. Changes to the contents of *points* have no effect after **glMap2** is called.

The following functions retrieve information related to the **glMap2** function:

glGetMap

glGet with argument **GL_MAX_EVAL_ORDER**

glIsEnabled with argument **GL_MAP2_VERTEX_3**

glIsEnabled with argument **GL_MAP2_VERTEX_4**

glIsEnabled with argument **GL_MAP2_INDEX**

glIsEnabled with argument **GL_MAP2_COLOR_4**

glIsEnabled with argument **GL_MAP2_NORMAL**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_1**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_2**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_3**

glIsEnabled with argument **GL_MAP2_TEXTURE_COORD_4**

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_VALUE is generated if either *ustride* or *vstride* is less than the number of values in a control point.

GL_INVALID_VALUE is generated if either *uorder* or *vorder* is less than one or greater than **GL_MAX_EVAL_ORDER**.

GL_INVALID_OPERATION is generated if **glMap2** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glColor**, **glEnable**, **glEvalCoord**, **glEvalMesh**, **glEvalPoint**, **glMap1**, **glMapGrid**, **glNormal**, **glTexCoord**, **glVertex**

glMapGrid1d, glMapGrid1f, glMapGrid2d, glMapGrid2f

These functions define a one- or two-dimensional mesh.

void glMapGrid1d(
  GLint *un*,
  GLdouble *u1*,
  GLdouble *u2*
  );

void glMapGrid1f(
  GLint *un*,
  GLfloat *u1*,
  GLfloat *u2*
  );

void glMapGrid2d(
  GLint *un*,
  GLdouble *u1*,
  GLdouble *u2*,
  GLint *vn*,
  GLdouble *v1*,
  GLdouble *v2*
  );

void glMapGrid2f(
  GLint *un*,
  GLfloat *u1*,
  GLfloat *u2*,
  GLint *vn*,
  GLfloat *v1*,
  GLfloat *v2*
  );

Parameters

un
  Specifies the number of partitions in the grid range interval [*u1*, *u2*]. Must be positive.

u1, u2
  Specify the mappings for integer grid domain values *i* = 0 and *i* = *un*.

vn
  Specifies the number of partitions in the grid range interval [*v1*, *v2*] (**glMapGrid2** only).

v1, v2
  Specify the mappings for integer grid domain values *j* = 0 and *j* = *vn* (**glMapGrid2** only).

Remarks

The **glMapGrid** and **glEvalMesh** functions are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. **glEvalMesh** steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by **glMap1** and **glMap2**.

The **glMapGrid1** and **glMapGrid2** functions specify 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 **glMap1** and **glMap2** for details of how *u* and *v* coordinates are evaluated.

The **glMapGrid1** function 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 such that

$$u = i(u2 - u1)/un + u1$$

The **glMapGrid2** function 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 **glMapGrid** are used identically by **glEvalMesh** and **glEvalPoint**.

The following functions retrieve information related to the **glMapGrid** function:

glGet with argument **GL_MAP1_GRID_DOMAIN**

glGet with argument **GL_MAP2_GRID_DOMAIN**

glGet with argument **GL_MAP1_GRID_SEGMENTS**

glGet with argument **GL_MAP2_GRID_SEGMENTS**

Errors

GL_INVALID_VALUE is generated if either *un* or *vn* is not positive.

GL_INVALID_OPERATION is generated if **glMapGrid** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEvalCoord, **glEvalMesh**, **glEvalPoint**, **glMap1**, **glMap2**

glMaterialf, glMateriali, glMaterialfv, glMaterialiv

These functions specify material parameters for the lighting model.

void glMaterialf(
  GLenum *face*,
  GLenum *pname*,
  GLfloat *param*
  );

void glMateriali(
  GLenum *face*,
  GLenum *pname*,
  GLint *param*
  );

Parameters

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 single-valued material parameter of the face or faces that is being updated. Must be **GL_SHININESS**.

param
  Specifies the value that parameter **GL_SHININESS** will be set to.

void glMaterialfv(
  GLenum *face*,
  GLenum *pname*,
  const GLfloat **params*
  );

void glMaterialiv(
  GLenum *face*,
  GLenum *pname*,
  const GLint **params*
  );

Parameters

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. Must be one of **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, **GL_EMISSION**, **GL_SHININESS**, **GL_AMBIENT_AND_DIFFUSE**, or **GL_COLOR_INDEXES**.

params
  Specifies a pointer to the value or values to which *pname* will be set.

Remarks

The **glMaterial** function 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.

Refer to the **glLightModel** reference page for details concerning one- and two-sided lighting calculations.

The **glMaterial** function 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, *params*, specifies what value or values will be assigned to the specified parameter.

Material parameters are used in the lighting equation that is optionally applied to each vertex. The equation is discussed in **glLightModel**. The parameters that can be specified using **glMaterial**, and their interpretations by the lighting equation, are as follows:

GL_AMBIENT
The *params* parameter contains four integer or floating-point values that specify the ambient RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default ambient reflectance for both front- and back-facing materials is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE
The *params* parameter contains four integer or floating-point values that specify the diffuse RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default diffuse reflectance for both front- and back-facing materials is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR
The *params* parameter contains four integer or floating-point values that specify the specular RGBA reflectance of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default specular reflectance for both front- and back-facing materials is (0.0, 0.0, 0.0, 1.0).

GL_EMISSION
The *params* parameter contains four integer or floating-point values that specify the RGBA emitted light intensity of the material. Integer values are mapped linearly such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly. Neither integer nor floating-point values are clamped. The default emission intensity for both front- and back-facing materials is (0.0, 0.0, 0.0, 1.0).

GL_SHININESS
The *params* parameter is a single integer or floating-point value that specifies the RGBA specular exponent of the material. Integer and floating-point values are mapped directly. Only values in the range [0,128] are accepted. The default specular exponent for both front- and back-facing materials is 0.

GL_AMBIENT_AND_DIFFUSE
Equivalent to calling **glMaterial** twice with the same parameter values, once with **GL_AMBIENT** and once with **GL_DIFFUSE**.

GL_COLOR_INDEXES
The *params* parameter contains three integer or 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. Refer to **glLightModel** for a discussion of color index lighting.

The material parameters can be updated at any time. In particular, **glMaterial** can be called between a call to **glBegin** and the corresponding call to **glEnd**. If only a single material parameter is to be changed per vertex, however, **glColorMaterial** is preferred over **glMaterial**.

The following function retrieves information related to the **glMaterial** function:

glGetMaterial

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.

See Also

glColorMaterial, **glLight**, **glLightModel**

glMatrixMode

The **glMatixMode** function specifies which matrix is the current matrix.

void glMatrixMode(
  GLenum *mode*
  );

Parameters

mode
  Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: **GL_MODELVIEW**, **GL_PROJECTION**, and **GL_TEXTURE**.

Remarks

The **glMatixMode** function sets the current matrix mode. The *mode* parameter 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.

The following function retrieves information related to the **glMatrixMode** function:

glGet with argument **GL_MATRIX_MODE**

Errors

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

GL_INVALID_OPERATION is generated if **glMatrixMode** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glLoadMatrix, **glPushMatrix**

glMultMatrixd, glMultMatrixf

The **glMultMatrixd** and **glMultMatrixf** functions multiply the current matrix by an arbitrary matrix.

void glMultMatrixd(
  const GLdouble **m*
  );

void glMultMatrixf(
  const GLfloat **m*
  );

Parameters

m
  Specifies a pointer a to 4x4 matrix stored in column-major order as sixteen consecutive values.

Remarks

The **glMultMatrix** function multiplies the current matrix with the one specified in *m*. That is, if M is the current matrix and T is the matrix passed to **glMultMatrix**, then M is replaced with M • T.

The current matrix is the projection matrix, modelview matrix, or texture matrix, determined by the current matrix mode (see **glMatrixMode**).

The *m* parameter points to a 4x4 matrix of single- or double-precision floating-point values stored in column-major order. That is, the matrix is stored as

{ewc msdncd, EWGraphic, group10336 0 /a "SDK.bmp"}

The following functions retrieve information related to the **glMultMatrix** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glMultMatrix** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glMatrixMode, **glLoadIdentity**, **glLoadMatrix**, **glPushMatrix**

glNewList, glEndList

The **glNewList** and **glEndList** functions create or replace a display list.

void glNewList(
  GLuint *list*,
  GLenum *mode*
  );

void glEndList(
  void
  );

Parameters

list
  Specifies the display list name.

mode
  Specifies the compilation mode, which can be **GL_COMPILE** or **GL_COMPILE_AND_EXECUTE**.

Remarks

Display lists are groups of GL commands that have been stored for subsequent execution. The display lists are created with **glNewList**. All subsequent commands are placed in the display list, in the order issued, until **glEndList** is called.

The **glNewList** function 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 **glGenLists** and tested for uniqueness with **glIsList**. 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. These commands are **glIsList**, **glGenLists**, **glDeleteLists**, **glFeedbackBuffer**, **glSelectBuffer**, **glRenderMode**, **glReadPixels**, **glPixelStore**, **glFlush**, **glFinish**, **glIsEnabled**, and all of the **glGet** routines.

When the **glNewList** function is encountered, the display-list definition is completed by associating the list with the unique name *list* (specified in the **glNewList** command). If a display list with name *list* already exists, it is replaced only when **glEndList** is called.

glCallList and **glCallLists** can be entered into display lists. The commands in the display list or lists executed by **glCallList** or **glCallLists** are not included in the display list being created, even if the list creation mode is **GL_COMPILE_AND_EXECUTE**.

The following function retrieves information related to the **glNewList** function:

glGet with argument **GL_MATRIX_MODE**

Errors

GL_INVALID_VALUE is generated if *list* is zero.

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

GL_INVALID_OPERATION is generated if **glEndList** is called without a preceding **glNewList**, or if **glNewList** is called while a display list is being defined.

GL_INVALID_OPERATION is generated if **glNewList** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallList, glCallLists, glDeleteLists, glGenLists, glIsList

glNormal

glNormal3b, glNormal3d, glNormal3f, glNormal3i, glNormal3s, glNormal3bv, glNormal3dv, glNormal3fv, glNormal3iv, glNormal3sv

These functions set the current normal vector.

void glNormal3b(
  GLbyte *nx*,
  GLbyte *ny*,
  GLbyte *nz*
  );

void glNormal3d(
  GLdouble *nx*,
  GLdouble *ny*,
  GLdouble *nz*
  );

void glNormal3f(
  GLfloat *nx*,
  GLfloat *ny*,
  GLfloat *nz*
  );

void glNormal3i(
  GLint *nx*,
  GLint *ny*,
  GLint *nz*
  );

void glNormal3s(
  GLshort *nx*,
  GLshort *ny*,
  GLshort *nz*
  );

Parameters

nx, ny, nz
  Specify the *x, y,* and *z* coordinates of the new current normal. The initial value of the current normal is (0,0,1).

void glNormal3bv(
  const GLbyte **v*
  );

void glNormal3dv(
  const GLdouble **v*
  );

void glNormal3fv(
  const GLfloat **v*
  );

void glNormal3iv(
  const GLint **v*
  );

void glNormal3sv(
const GLshort **v*
);

Parameters

v

Specifies a pointer to an array of three elements:

the *x, y,* and *z* coordinates of the new current normal.

Remarks

The current normal is set to the given coordinates whenever **glNormal** is issued. Byte, short, or integer arguments are converted to floating-point format with a linear mapping that maps the most positive representable integer value to 1.0, and the most negative representable integer value to -1.0.

Normals specified with **glNormal** need not have unit length. If normalization is enabled, then normals specified with **glNormal** are normalized after transformation. Normalization is controlled using **glEnable** and **glDisable** with the argument **GL_NORMALIZE**. By default, normalization is disabled.

The current normal can be updated at any time. In particular, **glNormal** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following functions retrieve information related to the **glNormal** function:

glGet with argument **GL_CURRENT_NORMAL**

glIsEnable with argument **GL_NORMALIZE**

See Also

glBegin, **glColor**, **glIndex**, **glTexCoord**, **glVertex**

glOrtho

The **glOrtho** function multiplies the current matrix by an orthographic matrix.

void glOrtho(
  GLdouble *left*,
  GLdouble *right*,
  GLdouble *bottom*,
  GLdouble *top*,
  GLdouble *near*,
  GLdouble *far*
  );

Parameters

left, right
  Specify the coordinates for the left and right vertical clipping planes.

bottom, top
  Specify the coordinates for the bottom and top horizontal clipping planes.

near, far
  Specify the distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.

Remarks

The **glOrtho** function describes a perspective matrix that produces a parallel projection. (*left*, *bottom*, -*near*) and (*right*, *top*, - *near*) 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). -*far* specifies the location of the far clipping plane. Both *near* and *far* can be either positive or negative. The corresponding matrix is

{ewc msdncd, EWGraphic, group10339 0 /a "SDK.bmp"}

where

{ewc msdncd, EWGraphic, group10339 1 /a "SDK.bmp"}

The current matrix is multiplied by this matrix with the result replacing the current matrix. That is, if M is the current matrix and O is the ortho matrix, then M is replaced with M • O.

Use **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.

The following functions retrieve information related to the **glOrtho** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glOrtho** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFrustum, **glMatrixMode**, **glMultMatrix**, **glPushMatrix**, **glViewport**

glPassThrough

The **glPassThrough** function places a marker in the feedback buffer.

void glPassThrough(
  GLfloat *token*
  );

Parameters

token
  Specifies a marker value to be placed in the feedback buffer following a **GL_PASS_THROUGH_TOKEN**.

Remarks

Feedback is a GL render mode. The mode is selected by calling **glRenderMode** 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 **glFeedbackBuffer** for a description of the feedback buffer and the values in it.

The **glPassThrough** function 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 **glPassThrough** commands with respect to the specification of graphics primitives is maintained.

The **glPassThrough** function is ignored if the GL is not in feedback mode.

The following function retrieves information related to the **glPassThrough** function:

glGet with argument **GL_RENDER_MODE**

Errors

GL_INVALID_OPERATION is generated if **glPassThrough** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFeedbackBuffer, **glRenderMode**

glPixelMapfv, glPixelMapuiv, glPixelMapusv

These functions set up pixel transfer maps.

void glPixelMapfv(
  GLenum *map*,
  GLint *mapsize*,
  const GLfloat **values*
  );

void glPixelMapuiv(
  GLenum *map*,
  GLint *mapsize*,
  const GLuint **values*
  );

void glPixelMapusv(
  GLenum *map*,
  GLint *mapsize*,
  const GLushort **values*
  );

Parameters

map
  Specifies a symbolic map name. Must be one of the following: **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,** or **GL_PIXEL_MAP_A_TO_A.**

mapsize
  Specifies the size of the map being defined.

values
  Specifies an array of *mapsize* values.

Remarks

The **glPixelMap** function sets up translation tables, or *maps*, used by **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D**. Use of these maps is described completely in the **glPixelTransfer** 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 topic.

The *map* parameter is a symbolic map name, indicating one of ten maps to set. *mapsize* specifies the number of entries in the map, and *values* is a pointer to an array of *mapsize* map values.

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 can be specified as single-precision floating-point numbers, unsigned short integers, or unsigned long integers. 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 **glPixelMapfv** are converted directly to the internal floating-point format of these maps, then clamped to the range [0,1]. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** are converted linearly such that the largest representable integer maps to 1.0, and zero maps to 0.0.

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 **glPixelMapfv** are converted directly to the internal fixed-point format of these maps. Unsigned integer values specified by **glPixelMapusv** and **glPixelMapuiv** specify integer values, with all zeros to the right of the binary point.

The table below shows the initial sizes and values for each of the maps. Maps that are indexed by either color or stencil indices must have *mapsize* = 2^*n* for some *n* or 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, and it is at least 32.

GL_PIXEL_MAP_I_TO_I

Lookup Index: color index
Lookup Value: color index
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_S_TO_S

Lookup Index: stencil index
Lookup Value: stencil index
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_I_TO_R

Lookup Index: color index
Lookup Value: R
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_I_TO_G

Lookup Index: color index
Lookup Value: G
Initial Size: 1

Initial Value: 0.0

GL_PIXEL_MAP_I_TO_B

Lookup Index: color index
Lookup Value: B
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_I_TO_A

Lookup Index: color index
Lookup Value: A
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_R_TO_R

Lookup Index: R
Lookup Value: R
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_G_TO_G

Lookup Index: G
Lookup Value: G
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_B_TO_B

Lookup Index: B
Lookup Value: B
Initial Size: 1
Initial Value: 0.0

GL_PIXEL_MAP_A_TO_A

Lookup Index: A
Lookup Value: A
Initial Size: 1
Initial Value: 0.0

The following functions retrieve information related to the **glPixelMap** function:

glGet with argument **GL_PIXEL_MAP_I_TO_I_SIZE**

glGet with argument **GL_PIXEL_MAP_S_TO_S_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_R_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_G_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_B_SIZE**

glGet with argument **GL_PIXEL_MAP_I_TO_A_SIZE**

glGet with argument **GL_PIXEL_MAP_R_TO_R_SIZE**

glGet with argument **GL_PIXEL_MAP_G_TO_G_SIZE**

glGet with argument **GL_PIXEL_MAP_B_TO_B_SIZE**

glGet with argument **GL_PIXEL_MAP_A_TO_A_SIZE**

glGet with argument **GL_MAX_PIXEL_MAP_TABLE**

Errors

GL_INVALID_ENUM is generated if *map* is not an accepted value.

GL_INVALID_VALUE is generated if *mapsize* is negative 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 **glPixelMap** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glDrawPixels**, **glPixelStore**, **glPixelTransfer**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelStoref, glPixelStorei

The **glPixelStoref** and **glPixelStorei** functions set pixel storage modes.

void glPixelStoref(
  GLenum *pname*,
  GLfloat *param*
  );

void glPixelStorei(
  GLenum *pname*,
  GLint *param*
  );

Parameters

pname
  Specifies the symbolic name of the parameter to be set. Six values affect the packing of pixel data into memory: **GL_PACK_SWAP_BYTES, GL_PACK_LSB_FIRST, GL_PACK_ROW_LENGTH, GL_PACK_SKIP_PIXELS, GL_PACK_SKIP_ROWS**, and **GL_PACK_ALIGNMENT**. Six more affect the unpacking of pixel data *from* memory: **GL_UNPACK_SWAP_BYTES, GL_UNPACK_LSB_FIRST, GL_UNPACK_ROW_LENGTH, GL_UNPACK_SKIP_PIXELS, GL_UNPACK_SKIP_ROWS,** and **GL_UNPACK_ALIGNMENT**.

param
  Specifies the value that *pname* is set to.

Remarks

The **glPixelStore** function sets pixel storage modes that affect the operation of subsequent **glDrawPixels** and **glReadPixels** as well as the unpacking of polygon stipple patterns (see **glPolygonStipple**), bitmaps (see **glBitmap**), and texture patterns (see **glTexImage1D** and **glTexImage2D**).

The *pname* parameter 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, and are therefore significant only for **glReadPixels** commands. 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 is made up of bytes $b_{(0)}$, $b_{(1)}$, $b_{(2)}$, $b_{(3)}$, it is stored in memory as $b_{(3)}$, $b_{(2)}$, $b_{(1)}$, $b_{(0)}$ 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 zero, **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

{ewc msdncd, EWGraphic, group10342 0 /a "SDK.bmp"}

  components or indices, where *n* is the number of components or indices in a pixel, *l* is the number of pixels in a row (**GL_PACK_ROW_LENGTH** if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of **GL_PACK_ALIGNMENT**, and *s* is the size, in bytes, of a single

component (if $a < s$, then it is as if $a = s$). In the case of 1-bit values, the location of the next row is obtained by skipping

{ewc msdncd, EWGraphic, group10342 1 /a "SDK.bmp"}

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format **GL_RGB**, for example, has three components per pixel: first red, then green, and finally blue.

GL_PACK_SKIP_PIXELS and **GL_PACK_SKIP_ROWS**
These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glReadPixels**. Setting **GL_PACK_SKIP_PIXELS** to *i* is equivalent to incrementing the pointer by *i n* components or indices, where *n* is the number of components or indices in each pixel. Setting **GL_PACK_SKIP_ROWS** to *j* is equivalent to incrementing the pointer by *j k* components or indices, where *k* is the number of components or indices per row, as computed above 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 significant for **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, and **glPolygonStipple**. 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 is made up of bytes $b_{(0)}$, $b_{(1)}$, $b_{(2)}$, $b_{(3)}$, it is taken from memory as $b_{(3)}$, $b_{(2)}$, $b_{(1)}$, $b_{(0)}$ 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 significant for bitmap data only.

GL_UNPACK_ROW_LENGTH
If greater than zero, **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

{ewc msdncd, EWGraphic, group10342 2 /a "SDK.bmp"}

components or indices, where n is the number of components or indices in a pixel, *l* is the number of pixels in a row (**GL_UNPACK_ROW_LENGTH** if it is greater than zero, the width argument to the pixel routine otherwise), *a* is the value of **GL_UNPACK_ALIGNMENT**, and *s* is the size, in bytes, of a single component (if $a < s$, then it is as if $a = s$). In the case of 1-bit values, the location of the next row is obtained by skipping

{ewc msdncd, EWGraphic, group10342 3 /a "SDK.bmp"}

components or indices.

The word *component* in this description refers to the nonindex values red, green, blue, alpha, and depth. Storage format **GL_RGB**, for example, has three components per pixel: first red, then green,

and finally blue.

GL_UNPACK_SKIP_PIXELS and **GL_UNPACK_SKIP_ROWS**
These values are provided as a convenience to the programmer; they provide no functionality that cannot be duplicated simply by incrementing the pointer passed to **glDrawPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple**. Setting **GL_UNPACK_SKIP_PIXELS** to *i* is equivalent to incrementing the pointer by *i n* components or indices, where *n* is the number of components or indices in each pixel. Setting **GL_UNPACK_SKIP_ROWS** to *j* is equivalent to incrementing the pointer by *j k* components or indices, where *k* is the number of components or indices per row, as computed above 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 of the storage parameters that can be set with **glPixelStore**.

Pname	Type	Initial Value	Valid Range
GL_PACK_SWAP_BYTES	Boolean	false	true or false
GL_PACK_SWAP_BYTES	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 **glPixelStoref** function can be used to set any pixel store parameter. If the parameter type is Boolean, and if *param* is 0.0, the parameter is false; otherwise it is set to true. If *pname* is an integer type parameter, *param* is rounded to the nearest integer.

Likewise, the **glPixelStorei** function can also be used to set any of the pixel store parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. *param* is converted to floating point before being assigned to real-valued parameters.

The pixel storage modes in effect when **glDrawPixels**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**, **glBitmap**, or **glPolygonStipple** 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.

The following functions retrieve information related to the **glPixelStore** function:

glGet with argument **GL_PACK_SWAP_BYTES**

glGet with argument **GL_PACK_LSB_FIRST**

glGet with argument **GL_PACK_ROW_LENGTH**

glGet with argument **GL_PACK_SKIP_ROWS**

glGet with argument **GL_PACK_SKIP_PIXELS**

glGet with argument **GL_PACK_ALIGNMENT**

glGet with argument **GL_UNPACK_SWAP_BYTES**

glGet with argument **GL_UNPACK_LSB_FIRST**

glGet with argument **GL_UNPACK_ROW_LENGTH**

glGet with argument **GL_UNPACK_SKIP_ROWS**

glGet with argument **GL_UNPACK_SKIP_PIXELS**

glGet with argument **GL_UNPACK_ALIGNMENT**

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 **glPixelStore** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBitmap, **glDrawPixels**, **glPixelMap**, **glPixelTransfer**, **glPixelZoom**, **glPolygonStipple**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelTransferf, glPixelTransferi

The **glPixelTransferf** and **glPixelTransferi** functions set pixel transfer modes.

void glPixelTransferf(
  GLenum *pname*,
  GLfloat *param*
  );

void glPixelTransferi(
  GLenum *pname*,
  GLint *param*
  );

Parameters

pname
  Specifies the symbolic name of the pixel transfer parameter to be set. Must be one of the following: **GL_MAP_COLOR, GL_MAP_STENCIL, GL_INDEX_SHIFT, GL_INDEX_OFFSET, GL_RED_SCALE, GL_RED_BIAS, GL_GREEN_SCALE, GL_GREEN_BIAS, GL_BLUE_SCALE, GL_BLUE_BIAS, GL_ALPHA_SCALE, GL_ALPHA_BIAS, GL_DEPTH_SCALE,** or **GL_DEPTH_BIAS**.

param
  Specifies the value that *pname* is set to.

Remarks

The **glPixelTransfer** function sets pixel transfer modes that affect the operation of subsequent **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, and **glTexImage2D** commands. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (**glReadPixels** and **glCopyPixels**) or unpacked from client memory (**glDrawPixels**, **glTexImage1D**, and **glTexImage2D**). 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 (**glPixelStore**) 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 are made up of four floating-point values with unspecified mantissa and exponent sizes, scaled such that 0.0 represents zero intensity and 1.0 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 **glPixelTransfer**.

  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 **glPixelTransfer**. The contents of the various maps are specified with **glPixelMap**.

Color index

Each color index is shifted left by **GL_INDEX_SHIFT** bits, filling with zeros any bits beyond the number of fraction bits carried by the fixed-point index. 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 **glPixelTransfer**.

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 **glPixelTransfer**. The contents of the index map are specified with **glPixelMap**.

If the resulting pixels are to be written to an RGBA color buffer, or if they are being 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 are specified with **glPixelMap**.

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.

The following table gives the type, initial value, and range of valid values for each of the pixel transfer parameters that are set with **glPixelTransfer**.

Pname	Type	Initial Value	Valid Range
GL_MAP_COLOR	Boolean	false	true/false
GL_MAP_STENCIL	Boolean	false	true/false
GL_INDEX_SHIFT	integer	0	(- oo,oo)
GL_INDEX_OFFSET	integer	0	(- oo,oo)
GL_RED_SCALE	float	1.0	(- oo,oo)
GL_GREEN_SCALE	float	1.0	(- oo,oo)
GL_BLUE_SCALE	float	1.0	(- oo,oo)
GL_ALPHA_SCALE	float	1.0	(- oo,oo)
GL_DEPTH_SCALE	float	1.0	(- oo,oo)
GL_RED_BIAS	float	0.0	(- oo,oo)
GL_GREEN_BIAS	float	0.0	(- oo,oo)

GL_BLUE_BIAS	float	0.0	(- ∞,∞)
GL_ALPHA_BIAS	float	0.0	(- ∞,∞)
GL_DEPTH_BIAS	float	0.0	(- ∞,∞)

The **glPixelTransferf** function can be used to set any pixel transfer parameter. If the parameter type is Boolean, 0.0 implies false and any other value implies true. If *pname* is an integer parameter, *param* is rounded to the nearest integer.

Likewise, **glPixelTransferi** can also be used to set any of the pixel transfer parameters. Boolean parameters are set to false if *param* is 0 and true otherwise. *param* is converted to floating point before being assigned to real-valued parameters.

If a **glDrawPixels**, **glReadPixels**, **glCopyPixels**, **glTexImage1D**, or **glTexImage2D** command is placed in a display list (see **glNewList** and **glCallList**), the pixel transfer mode settings in effect when the display list is *executed* are the ones that are used. They may be different from the settings when the command was compiled into the display list.

The following functions retrieve information related to the **glPixelTransfer** function:

glGet with argument **GL_MAP_COLOR**

glGet with argument **GL_MAP_STENCIL**

glGet with argument **GL_INDEX_SHIFT**

glGet with argument **GL_INDEX_OFFSET**

glGet with argument **GL_RED_SCALE**

glGet with argument **GL_RED_BIAS**

glGet with argument **GL_GREEN_SCALE**

glGet with argument **GL_GREEN_BIAS**

glGet with argument **GL_BLUE_SCALE**

glGet with argument **GL_BLUE_BIAS**

glGet with argument **GL_ALPHA_SCALE**

glGet with argument **GL_ALPHA_BIAS**

glGet with argument **GL_DEPTH_SCALE**

glGet with argument **GL_DEPTH_BIAS**

Errors

GL_INVALID_ENUM is generated if *pname* is not an accepted value.

GL_INVALID_OPERATION is generated if **glPixelTransfer** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCallList, **glCopyPixels**, **glDrawPixels**, **glNewList**, **glPixelMap**, **glPixelStore**, **glPixelZoom**, **glReadPixels**, **glTexImage1D**, **glTexImage2D**

glPixelZoom

The **glPixelZoom** function specifies the pixel zoom factors.

void glPixelZoom(
  GLfloat *xfactor*,
  GLfloat *yfactor*
  );

Parameters

xfactor, yfactor
  Specify the *x* and *y* zoom factors for pixel write operations.

Remarks

The **glPixelZoom** function specifies values for the *x* and *y* zoom factors. During the execution of **glDrawPixels** or **glCopyPixels**, if ($x_{(r)}$, $y_{(r)}$) is the current raster position, and a given element is in the *n*th row and mth column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

{ewc msdncd, EWGraphic, group10344 0 /a "SDK.bmp"}

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.

The following functions retrieve information related to the **glPixelZoom** function:

glGet with argument **GL_ZOOM_X**

glGet with argument **GL_ZOOM_Y**

Errors

GL_INVALID_OPERATION is generated if **glPixelZoom** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glDrawPixels**

glPointSize

The **glPointSize** function specifies the diameter of rasterized points.

void glPointSize(
  GLfloat *size*
  );

Parameters

size
  Specifies the diameter of rasterized points. The default is 1.0.

Remarks

The **glPointSize** function specifies the rasterized diameter of both aliased and antialiased points. Using a point size other than 1.0 has different effects, depending on whether point antialiasing is enabled. Point antialiasing is controlled by calling **glEnable** and **glDisable** with argument **GL_POINT_SMOOTH**.

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

$$(\lfloor x_{(w)} \rfloor + .5 , \lfloor y_{(w)} \rfloor + .5)$$

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

$$(\lfloor x_{(w)} + .5 \rfloor, \lfloor y_{(w)} + .5 \rfloor)$$

and the rasterized fragments 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 nonantialiased 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 points ($x_{(w)}$, $y_{(w)}$). 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 1.0 is guaranteed to be supported; others depend on the implementation. The range of supported sizes and the size difference between supported sizes within the range can be queried by calling **glGet** with arguments **GL_POINT_SIZE_RANGE** and **GL_POINT_SIZE_GRANULARITY**.

The point size specified by **glPointSize** is always returned when **GL_POINT_SIZE** is queried. Clamping and rounding for aliased and antialiased points have no effect on the specified value.

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.

The following functions retrieve information related to the **glPointSize** function:

glGet with argument **GL_POINT_SIZE**

glGet with argument **GL_POINT_SIZE_RANGE**

glGet with argument **GL_POINT_SIZE_GRANULARITY**

glIsEnabled with argument **GL_POINT_SMOOTH**

Errors

GL_INVALID_VALUE is generated if *size* is less than or equal to zero.

GL_INVALID_OPERATION is generated if **glPointSize** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable

glPolygonMode

The **glPolygonMode** function selects a polygon rasterization mode.

void glPolygonMode(
  GLenum *face*,
  GLenum *mode*
  );

Parameters

face
  Specifies the polygons that *mode* applies to. Must be **GL_FRONT** for front-facing polygons, **GL_BACK** for back-facing polygons, or **GL_FRONT_AND_BACK** for front- and back-facing polygons.

mode
  Specifies the way polygons will be rasterized. Accepted values are **GL_POINT**, **GL_LINE**, and **GL_FILL**. The default is **GL_FILL** for both front- and back-facing polygons.

Remarks

The **glPolygonMode** function 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 vertexes 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 vertexes 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.

To draw a surface with filled back-facing polygons and outlined front-facing polygons, call

glPolygonMode(**GL_FRONT**, **GL_LINE**);

Vertexes are marked as boundary or nonboundary with an edge flag. Edge flags are generated internally by the GL when it decomposes polygons, and they can be set explicitly using **glEdgeFlag**.

The following function retrieves information related to the **glPolygonMode** function:

glGet with argument **GL_POLYGON_MODE**

Errors

GL_INVALID_ENUM is generated if either *face* or *mode* is not an accepted value.

GL_INVALID_OPERATION is generated if **glPolygonMode** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glEdgeFlag**, **glLineStipple**, **glLineWidth**, **glPointSize**, **glPolygonStipple**

glPolygonStipple

The **glPolygonStipple** function sets the polygon stippling pattern.

void glPolygonStipple(
  const GLubyte **mask*
  );

Parameters

mask
  Specifies a pointer to a 32x32 stipple pattern that will be unpacked from memory in the same way that **glDrawPixels** unpacks pixels.

Remarks

Polygon stippling, like line stippling (see **glLineStipple**), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

The *mask* parameter is a pointer to a 32x32 stipple pattern that is stored in memory just like the pixel data supplied to a **glDrawPixels** 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. **glPixelStore** 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.

Polygon stippling is enabled and disabled with **glEnable** and **glDisable**, using argument **GL_POLYGON_STIPPLE**. If enabled, a rasterized polygon fragment with window coordinates $x_{(w)}$ and $y_{(w)}$ is sent to the next stage of the GL if and only if the ($x_{(w)}$ mod 32)th bit in the ($y_{(w)}$ mod 32)th row of the stipple pattern is one. When polygon stippling is disabled, it is as if the stipple pattern were all ones.

The following functions retrieve information related to the **glPolygonStipple** function:

glGetPolygonStipple

glIsEnabled with argument **GL_POLYGON_STIPPLE**

Errors

GL_INVALID_OPERATION is generated if **glPolygonStipple** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glLineStipple**, **glPixelStore**, **glPixelTransfer**

glPushAttrib, glPopAttrib

The **glPushAttrib** and **glPopAttrib** functions push and pop the attribute stack.

void glPushAttrib(
  GLbitfield *mask*
  );

Parameters

mask
  Specifies a mask that indicates which attributes to save. Values for *mask* are listed below.

void glPopAttrib(
  void
  );

Remarks

The **glPushAttrib** function 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 (the indented paragraphs list which attributes are saved):

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
  GL_DITHER enable bit
  GL_DRAW_BUFFER setting
  GL_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_LIGHT*i* where 0<= *i*<**GL_MAX_LIGHTS**
GL_LIGHTING flag
GL_LINE_SMOOTH flag
GL_LINE_STIPPLE flag
GL_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_SMOOTH flag
GL_POLYGON_STIPPLE flag
GL_SCISSOR_TEST flag
GL_SENCIL_TESTT 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
1-D grid endpoints and divisions
2-D grid endpoints and divisions
GL_AUTO_NORMAL enable bit

GL_FOG_BIT
GL_FOG enable flag
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_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**
glTexGen plane equations for **S**, **T**, **R**, and **Q**

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

The **glPopAttrib** function restores the values of the state variables saved with the last **glPushAttrib** command. Those not saved are left unchanged.

It is an error to push attributes onto a full stack, or to pop attributes off an empty stack. In either 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, pixel pack and unpack state, render mode state, and select and feedback state cannot be saved.

The depth of the attribute stack depends on the implementation, but it must be at least 16.

The following functions retrieve information related to the **glPushAttrib** and **glPopAttrib** functions:

glGet with argument **GL_ATTRIB_STACK_DEPTH**.

glGet with argument **GL_MAX_ATTRIB_STACK_DEPTH**.

Errors

GL_STACK_OVERFLOW is generated if **glPushAttrib** is called while the attribute stack is full.

GL_STACK_UNDERFLOW is generated if **glPopAttrib** is called while the attribute stack is empty.

GL_INVALID_OPERATION is generated if **glPushAttrib** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glGet, **glGetClipPlane**, **glGetError**, **glGetLight**, **glGetMap**, **glGetMaterial**, **glGetPixelMap**, **glGetPolygonStipple**, **glGetString**, **glGetTexEnv**, **glGetTexGen**, **glGetTexImage**, **glGetTexLevelParameter**, **glGetTexParameter**, **glIsEnabled**

glPushMatrix, glPopMatrix

The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

void glPushMatrix(
  void
  );

void glPopMatrix(
  void
  );

Remarks

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 two modes, **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.

The **glPushMatrix** function pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on the top of the stack is identical to the one below it. The **glPushMatrix** function 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 push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set and no other change is made to GL state.

The following functions retrieve information related to the **glPushMatrix** and **glPopMatrix** functions:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

glGet with argument **GL_MODELVIEW_STACK_DEPTH**

glGet with argument **GL_PROJECTION_STACK_DEPTH**

glGet with argument **GL_TEXTURE_STACK_DEPTH**

glGet with argument **GL_MAX_MODELVIEW_STACK_DEPTH**

glGet with argument **GL_MAX_PROJECTION_STACK_DEPTH**

glGet with argument **GL_MAX_TEXTURE_STACK_DEPTH**

Errors

GL_STACK_OVERFLOW is generated if **glPushMatrix** is called while the current matrix stack is full.

GL_STACK_UNDERFLOW is generated if **glPopMatrix** is called while the current matrix stack contains only a single matrix.

GL_INVALID_OPERATION is generated if **glPushMatrix** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFrustum, **glLoadIdentity**, **glLoadMatrix**, **glMatrixMode**, **glMultMatrix**, **glOrtho**, **glRotate**, **glScale**, **glTranslate**, **glViewport**

glPushName, glPopName

The **glPushName** and **glPopName** functions push and pop the name stack.

void glPushName(
  GLuint *name*
  );

Parameters

name
  Specifies a name that will be pushed onto the name stack.

void glPopName(
  void
  );

Remarks

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. The **glPushName** function causes *name* to be pushed onto the name stack, which is initially empty. **glPopName** pops one name off the top of the stack.

It is an error to push a name onto a full stack, or to pop a name off an empty stack. It is also an error to manipulate the name stack between a call to **glBegin** and the corresponding call to **glEnd**. 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 **glPushName** or **glPopName** while the render mode is not **GL_SELECT** are ignored.

The following functions retrieve information related to the **glPushName** and **glPopName** functions:

glGet with argument **GL_NAME_STACK_DEPTH**

glGet with argument **GL_MAX_NAME_STACK_DEPTH**

Errors

GL_STACK_OVERFLOW is generated if **glPushName** is called while the name stack is full.

GL_STACK_UNDERFLOW is generated if **glPopName** is called while the name stack is empty.

GL_INVALID_OPERATION is generated if **glPushName** or **glPopName** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glInitNames, **glLoadName**, **glRenderMode**, **glSelectBuffer**

glRasterPos

glRasterPos2d, glRasterPos2f, glRasterPos2i, glRasterPos2s, glRasterPos3d, glRasterPos3f, glRasterPos3i, glRasterPos3s, glRasterPos4d, glRasterPos4f, glRasterPos4i, glRasterPos4s, glRasterPos2dv, glRasterPos2fv, glRasterPos2iv, glRasterPos2sv, glRastePos3dv, glRasterPos3fv, glRasterPos3iv, glRasterPos3sv, glRasterPos4dv, glRasterPos4fv, glRasterPos4iv, glRasterPos4sv

These functions specify the raster position for pixel operations.

void glRasterPos2d(
  GLdouble *x*,
  GLdouble *y*
  );

void glRasterPos2f(
  GLfloat *x*,
  GLfloat *y*
  );

void glRasterPos2i(
  GLint *x*,
  GLint *y*
  );

void glRasterPos2s(
  GLshort *x*,
  GLshort *y*
  );

void glRasterPos3d(
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*
  );

void glRasterPos3f(
  GLfloat *x*,
  GLfloat *y*,
  GLfloat *z*
  );

void glRasterPos3i(
  GLint *x*,
  GLint *y*,
  GLint *z*
  );

void glRasterPos3s(
  GLshort *x*,
  GLshort *y*,
  GLshort *z*
  );

void glRasterPos4d(
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*,
  GLdouble *w*

);

void glRasterPos4f(
GLfloat *x*,
GLfloat *y*,
GLfloat *z*,
GLfloat *w*
);

void glRasterPos4i(
GLint *x*,
GLint *y*,
GLint *z*,
GLint *w*
);

void glRasterPos4s(
GLshort *x*,
GLshort *y*,
GLshort *z*,
GLshort *w*
);

Parameters

x, y, z, w
Specify the x, y, z, and w object coordinates (if present) for the raster position.

void glRasterPos2dv(
const GLdouble **v*
);

void glRasterPos2fv(
const GLfloat **v*
);

void glRasterPos2iv(
const GLint **v*
);

void glRasterPos2sv(
const GLshort **v*
);

void glRasterPos3dv(
const GLdouble **v*
);

void glRasterPos3fv(
const GLfloat **v*
);

void glRasterPos3iv(
const GLint **v*
);

void glRasterPos3sv(
const GLshort **v*
);

void glRasterPos4dv(
const GLdouble *v
);

void glRasterPos4fv(
const GLfloat *v
);

void glRasterPos4iv(
const GLint *v
);

void glRasterPos4sv(
const GLshort *v
);

Parameters

v
Specifies a pointer to an array of two, three, or four elements, specifying *x, y, z,* and *w* coordinates, respectively.

Remarks

The GL maintains a 3-D position in window coordinates. This position, called the raster position, is maintained with subpixel accuracy. It is used to position pixel and bitmap write operations. See **glBitmap**, **glDrawPixels**, and **glCopyPixels**.

The current raster position consists of three window coordinates (*x, y, z*), a clip coordinate *w* value, 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. **glRasterPos4** specifies object coordinates *x, y, z,* and *w* explicitly. **glRasterPos3** specifies object coordinate *x, y,* and *z* explicitly, while *w* is implicitly set to one. **glRasterPos2** uses the argument values for *x* and *y* while implicitly setting *z* and *w* to zero and one.

The object coordinates presented by **glRasterPos** are treated just like those of a **glVertex** 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 the **GL_CURRENT_RASTER_INDEX**, in color index mode, is set to the color produced by the lighting calculation (see **glLight**, **glLightModel**, and **glShadeModel**). 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 **glTexGen**). 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.

Notes

The raster position is modified both by **glRasterPos** and by **glBitmap**.

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).

The following functions retrieve information related to the **glRasterPos** function:

glGet with argument **GL_CURRENT_RASTER_POSITION**

glGet with argument **GL_CURRENT_RASTER_POSITION_VALID**

glGet with argument **GL_CURRENT_RASTER_DISTANCE**

glGet with argument **GL_CURRENT_RASTER_COLOR**

glGet with argument **GL_CURRENT_RASTER_INDEX**

glGet with argument **GL_CURRENT_RASTER_TEXTURE_COORDS**

Errors

GL_INVALID_OPERATION is generated if **glRasterPos** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBitmap, **glCopyPixels**, **glDrawPixels**, **glLight**, **glLightModel**, **glShadeModel**, **glTexCoord**, **glTexGen**, **glVertex**

glReadBuffer

The **glReadBuffer** function selects a color buffer source for pixels.

void glReadBuffer(
  GLenum *mode*
  );

Parameters

mode
  Specifies a color buffer. Accepted values are **GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK**, **GL_LEFT**, **GL_RIGHT**, and **GL_AUX*i***, where ***i*** is between 0 and **GL_AUX_BUFFERS** -1.

Remarks

The **glReadBuffer** function specifies a color buffer as the source for subsequent **glReadPixels** and **glCopyPixels** 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.

Nonstereo 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 nonexistent buffer to **glReadBuffer**.

By default, *mode* is **GL_FRONT** in single-buffered configurations, and **GL_BACK** in double-buffered configurations.

The following function retrieves information related to the **glReadBuffer** function:

glGet with argument **GL_READ_BUFFER**

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 **glReadBuffer** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glDrawBuffer**, **glReadPixels**

glReadPixels

The **glReadPixels** function reads a block of pixels from the frame buffer.

void glReadPixels(
GLint *x*,
GLint *y*,
GLsizei *width*,
GLsizei *height*,
GLenum *format*,
GLenum *type*,
GLvoid **pixels*
);

Parameters

x, y
Specify the window coordinates of the first pixel that is read from the frame buffer. This location is the lower left corner of a rectangular block of pixels.

width, height
Specify the dimensions of the pixel rectangle. The *width* and *height* parameters of one correspond to a single pixel.

format
Specifies the format of the pixel data. The following symbolic values are accepted: **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**.

type
Specifies the data type of the pixel data. Must be one of **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** or **GL_FLOAT**.

pixels
Returns the pixel data.

Remarks

The **glReadPixels** function returns pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (*x*, *y*), into client memory starting at location *pixels*. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with three commands: **glPixelStore**, **glPixelTransfer**, and **glPixelMap**. This topic describes the effects on **glReadPixels** of most, but not all of the parameters specified by these three commands.

The **glReadPixels** function returns values from each pixel with lower left-hand corner at (*x* + i, *y* + j) for $0 \leq i < width$ and $0 \leq j < height$. This pixel is said to be the *i*th pixel in the *j*th row. Pixels are returned in row order from the lowest to the highest row, left to right in each row.

The *format* parameter specifies the format for the returned pixel values. Accepted values for *format* are as follows:

GL_COLOR_INDEX
Color indices are read from the color buffer selected by **glReadBuffer**. 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 0.0 and the maximum value maps to 1.0. Each component is then multiplied by **GL_DEPTH_SCALE**, added to **GL_DEPTH_BIAS**, and finally clamped to the range [0,1].

GL_RED
GL_GREEN
GL_BLUE
GL_ALPHA
GL_RGB
GL_RGBA
GL_LUMINANCE
GL_LUMINANCE_ALPHA
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 **glReadBuffer**. 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 **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 RGBA color components are stored in the color buffers, they are read from the color buffer selected by **glReadBuffer**. 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 **GL_RED**, **GL_GREEN**, **GL_BLUE**, and **GL_ALPHA**. Each component is clamped to the range [0,1]. Finally, if **GL_MAP_COLOR** is **GL_TRUE**, each color component *c* is replaced by its mapping in the table **GL_PIXEL_MAP_c_TO_c**, where *c* again is **GL_RED**, **GL_GREEN**, **GL_BLUE**, and **GL_ALPHA**. Each component is scaled to the size of its corresponding table before the lookup is performed.

Finally, unneeded data is 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 shift, scale, bias, and lookup factors described above are all specified by **glPixelTransfer**. The lookup table contents themselves are specified by **glPixelMap**.

The final step involves converting the indices or components to the proper format, as specified by *type*. If *format* is **GL_COLOR_INDEX** or **GL_STENCIL_INDEX** and *type* is not **GL_FLOAT**, each index is masked with the mask value given in the following table. If *type* is **GL_FLOAT**, then each integer index is converted to single-precision floating-point format.

If *format* is **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_RGB**, **GL_RGBA**, **GL_LUMINANCE**, or **GL_LUMINANCE_ALPHA** and *type* is not **GL_FLOAT**, each component is multiplied by the multiplier shown in the following table. If type is **GL_FLOAT**, then each component is passed as is (or converted to the clients single-precision floating-point format if it is different from the one used by the GL).

Type	Index Mask	Component Conversion
GL_UNSIGNED_BYTE	2^*8*-1	(2^*8*-1)*c*
GL_BYTE	2^*7*-1	[2^*7*-1]*c*-1]/2
GL_BITMAP	1	1
GL_UNSIGNED_SHORT	2^*16*-1	(2^*16*-1) *c*
GL_SHORT	2^*15*-1	[(2^*15*-1) *c*-1] / 2

GL_UNSIGNED_INT	2^*32*-1	(2^*32*-1) *c*
GL_INT	2^*31*-1	[(2^*31*-1) *c*-1] / 2
GL_FLOAT	none	*c*

Return values are placed in memory 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 value is returned and the data for the ith pixel in the jth row is placed in location (*j*) *width* + *i*. **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 *pixels*. Storage parameters set by **glPixelStore**, such as **GL_PACK_SWAP_BYTES** and **GL_PACK_LSB_FIRST**, affect the way that data is written into memory. See **glPixelStore** for a description.

Values for pixels that lie outside the window connected to the current GL context are undefined.

If an error is generated, no change is made to the contents of *pixels*.

The following function retrieves information related to the **glReadPixels** function:

glGet with argument **GL_INDEX_MODE**

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 **glReadPixels** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glCopyPixels, **glDrawPixels**, **glPixelMap**, **glPixelStore**, **glPixelTransfer**, **glReadBuffer**

glRectd, glRectf, glRecti, glRects, glRectdv, glRectfv, glRectiv, glRectsv

These functions draw a rectangle.

void glRectd(
  GLdouble *x1*,
  GLdouble *y1*,
  GLdouble *x2*,
  GLdouble *y2*
  );

void glRectf(
  GLfloat *x1*,
  GLfloat *y1*,
  GLfloat *x2*,
  GLfloat *y2*
  );

void glRecti(
  GLint *x1*,
  GLint *y1*,
  GLint *x2*,
  GLint *y2*
  );

void glRects(
  GLshort *x1*,
  GLshort *y1*,
  GLshort *x2*,
  GLshort *y2*
  );

Parameters

x1, y1
  Specify one vertex of a rectangle.

x2, y2
  Specify the opposite vertex of the rectangle.

void glRectdv(
  const GLdouble **v1*,
  const GLdouble **v2*
  );

void glRectfv(
  const GLfloat **v1*,
  const GLfloat **v2*
  );

void glRectiv(
  const GLint **v1*,
  const GLint **v2*
  );

void glRectsv(
  const GLshort **v1*,
  const GLshort **v2*
  );

Parameters

v1
Specifies a pointer to one vertex of a rectangle.

v2
Specifies a pointer to the opposite vertex of the rectangle.

Remarks

The **glRect** function supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (*x*, *y*) coordinates, or as two pointers to arrays, each containing an (*x*,*y*) pair. The resulting rectangle is defined in the *z* = 0 plane.

The **glRect**(*x1*, *y1*, *x2*, *y2*) function is exactly equivalent to the following sequence:

```
glBegin(GL_POLYGON);
glVertex2(x1, y1);
glVertex2(x2, y1);
glVertex2(x2, y2);
glVertex2(x1, y2);
glEnd( );
```

Notice that if the second vertex is above and to the right of the first vertex, the rectangle is constructed with a counterclockwise winding.

Errors

GL_INVALID_OPERATION is generated if **glRect** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glBegin, **glVertex**

glRenderMode

The **glRenderMode** function sets the rasterization mode.

GLint glRenderMode(
  GLenum *mode*
  );

Parameters

mode
  Specifies the rasterization mode. Three values are accepted: **GL_RENDER, GL_SELECT,** and **GL_FEEDBACK**.
  The default value is **GL_RENDER**.

Remarks

The **glRenderMode** function 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 was **GL_RENDER** is returned in a select buffer, which must be created (see **glSelectBuffer**) 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 vertexes that would have been drawn had the render mode been **GL_RENDER** are returned in a feedback buffer, which must be created (see **glFeedbackBuffer**) before feedback mode is entered.

The return value of the **glRenderMode** function is determined by the render mode at the time **glRenderMode** is called, rather than by *mode*. The values returned for the three render modes are as follows:

GL_RENDER
  Zero.

GL_SELECT
  The number of hit records transferred to the select buffer.

GL_FEEDBACK
  The number of values (not vertexes) transferred to the feedback buffer.

Refer to **glSelectBuffer** and **glFeedbackBuffer** for more details concerning selection and feedback operation.

If an error is generated, the **glRenderMode** function returns zero regardless of the current render mode.

The following function retrieves information related to the **glRenderMode** function:

glGet with argument **GL_RENDER_MODE**

Errors

GL_INVALID_ENUM is generated if *mode* is not one of the three accepted values.

GL_INVALID_OPERATION is generated if **glSelectBuffer** is called while the render mode is **GL_SELECT**, or if **glRenderMode** is called with argument **GL_SELECT** before **glSelectBuffer** is called at least once.

GL_INVALID_OPERATION is generated if **glFeedbackBuffer** is called while the render mode is **GL_FEEDBACK**, or if **glRenderMode** is called with argument **GL_FEEDBACK** before **glFeedbackBuffer** is called at least once.

GL_INVALID_OPERATION is generated if **glRenderMode** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFeedbackBuffer, **glInitNames**, **glLoadName**, **glPassThrough**, **glPushName**, **glSelectBuffer**

glRotated, glRotatef

The **glRotated** and **glRotatef** functions multiply the current matrix by a rotation matrix.

void glRotated(
  GLdouble *angle*,
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*
)

void glRotatef(
  GLfloat *angle*,
  GLfloat *x*,
  GLfloat *y*,
  GLfloat *z*
)

Parameters

angle
  Specifies the angle of rotation, in degrees.

x, y, z
  Specify the *x*, *y*, and *z* coordinates of a vector, respectively.

Remarks

The **glRotate** function computes a matrix that performs a counterclockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

The current matrix (see **glMatrixMode**) is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if M is the current matrix and R is the translation matrix, then M is replaced with M•R.

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glRotate** is called are rotated. Use **glPushMatrix** and **glPopMatrix** to save and restore the unrotated coordinate system.

The following functions retrieve information related to the **glRotate** function:

glGet with argument **GL_RENDER_MODE**

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glRotate** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glMatrixMode, **glMultMatrix**, **glPushMatrix**, **glScale**, **glTranslate**

glScaled, glScalef

The **glScaled** and **glScalef** functions multiply the current matrix by a general scaling matrix.

void glScaled(
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*
  );

void glScalef(
  GLfloat *x*,
  GLfloat *y*,
  GLfloat *z*
  );

Parameters

x, y, z
  Specify scale factors along the *x*, *y*, and *z* axes, respectively.

Remarks

The **glScale** function produces a general scaling along the *x*, *y*, and *z* axes. The three arguments indicate the desired scale factors along each of the three axes. The resulting matrix is

{ewc msdncd, EWGraphic, group10357 0 /a "SDK.bmp"}

The current matrix (see **glMatrixMode**) is multiplied by this scale matrix, with the product replacing the current matrix. That is, if M is the current matrix and S is the scale matrix, then M is replaced with M•S.

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glScale** is called are scaled. Use **glPushMatrix** and **glPopMatrix** to save and restore the unscaled coordinate system.

If scale factors other than 1.0 are applied to the modelview matrix and lighting is enabled, automatic normalization of normals should probably also be enabled (**glEnable** and **glDisable** with argument **GL_NORMALIZE**).

The following functions retrieve information related to the **glScale** function:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glScale** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glMatrixMode, **glMultMatrix**, **glPushMatrix**, **glRotate**, **glTranslate**

glScissor

The **glScissor** function defines the scissor box.

void glScissor(
  GLint *x*,
  GLint *y*,
  GLsizei *width*,
  GLsizei *height*
  );

Parameters

x, y
  Specify the lower left corner of the scissor box. Initially (0,0).

width, height
  Specify the width and height of the scissor box. When a GL context is *first* attached to a window, *width* and *height* are set to the dimensions of that window.

Remarks

The **glScissor** function 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.

The scissor test is enabled and disabled using **glEnable** and **glDisable** with argument **GL_SCISSOR_TEST**. While the scissor 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, so **glScissor**(0,0,1,1) allows only the lower-left pixel in the window to be modified, and **glScissor**(0,0,0,0) disallows modification to all pixels in the window.

When the scissor test is disabled, it is as though the scissor box includes the entire window.

The following functions retrieve information related to the **glScissor** function:

glGet with argument **GL_SCISSOR_BOX**

glIsEnabled with argument **GL_SCISSOR_TEST**

Errors

GL_INVALID_VALUE is generated if either *width* or *height* is negative.

GL_INVALID_OPERATION is generated if **glScissor** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glEnable, **glViewport**

glSelectBuffer

The **glSelectBuffer** function establishes a buffer for selection mode values.

void glSelectBuffer(
  GLsizei *size*,
  GLuint **buffer*
  );

Parameters

size
  Specifies the size of *buffer*.

buffer
  Returns the selection data.

Remarks

The **glSelectBuffer** function has two arguments: *buffer* is a pointer to an array of unsigned integers, and *size* indicates the size of the array. *buffer* returns values from the name stack (see **glInitNames**, **glLoadName**, **glPushName**) when the rendering mode is **GL_SELECT** (see **glRenderMode**). The **glSelectBuffer** function must be issued before selection mode is enabled, and it must not be issued while the rendering mode is **GL_SELECT**.

Selection is used by a programmer 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 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 **glRenderMode** is called, a hit record is copied to *buffer* if any hits have occurred since the last such event (name stack change or **glRenderMode** 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 vertexes that hit since the previous event, followed by the name stack contents, bottom name first.

Returned depth values are mapped such that the largest unsigned integer value corresponds to window coordinate depth 1.0, and zero corresponds to window coordinate depth 0.0.

An internal index into *buffer* is reset to zero 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 zero followed by the minimum and maximum depth values.

Selection mode is exited by calling **glRenderMode** with an argument other than **GL_SELECT**. Whenever **glRenderMode** 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 **glRenderMode** was called, a negative hit record count is returned.

The contents of *buffer* are undefined until **glRenderMode** is called with an argument other than **GL_SELECT**.

The **glBegin**/**glEnd** primitives and calls to **glRasterPos** can result in hits.

The following function retrieves information related to the **glSelectBuffer** function:

glGet with argument **GL_NAME_STACK_DEPTH**

Errors

GL_INVALID_VALUE is generated if *size* is negative.

GL_INVALID_OPERATION is generated if **glSelectBuffer** is called while the render mode is **GL_SELECT**, or if **glRenderMode** is called with argument **GL_SELECT** before **glSelectBuffer** is called at least once.

GL_INVALID_OPERATION is generated if **glSelectBuffer** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glFeedbackBuffer, **glInitNames**, **glLoadName**, **glPushName**, **glRenderMode**

glShadeModel

The **glShadeModel** function selects flat or smooth shading.

void glShadeModel(
  GLenum *mode*
  );

Parameters

mode
  Specifies a symbolic value representing a shading technique. Accepted values are **GL_FLAT** and **GL_SMOOTH**. The default is **GL_SMOOTH**.

Remarks

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertexes 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. Counting vertexes and primitives from one starting when **glBegin** is issued, each flat-shaded line segment i is given the computed color of vertex *i* + 1, its second vertex. Counting similarly from one, each flat-shaded polygon is given 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	3*i*
Quad strip	2*i* + 2
Independent quad	4*i*

Flat and smooth shading are specified by **glShadeModel** with *mode* set to **GL_FLAT** and **GL_SMOOTH**, respectively.

The following function retrieves information related to the **glShadeModel** function:

glGet with argument **GL_SHADE_MODEL**

Errors

GL_INVALID_ENUM is generated if *mode* is any value other than **GL_FLAT** or **GL_SMOOTH**.

GL_INVALID_OPERATION is generated if **glShadeModel** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColor, **glLight**, **glLightModel**

glStencilFunc

The **glStencilFunc** function sets the function and reference value for stencil testing.

void glStencilFunc(
  GLenum *func*,
  GLint *ref*,
  GLuint *mask*
  );

Parameters

func
  Specifies the test function. Eight tokens are valid: **GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL,** and **GL_ALWAYS**.

ref
  Specifies the reference value for the stencil test. *ref* is clamped to the range *[0,2^n - 1],* where *n* is the number of bitplanes in the stencil buffer.

mask
  Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done.

Remarks

Stenciling, like *z*-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 reference value and the value in the stencil buffer. The test is enabled by **glEnable** and **glDisable** with argument **GL_STENCIL**. Actions taken based on the outcome of the stencil test are specified with **glStencilOp**.

The *func* parameter is a symbolic constant that determines the stencil comparison function. It accepts one of eight values, shown below. *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 **glStencilOp**). 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.

Here are the values accepted by *func*:

GL_NEVER
  Always fails.

GL_LESS
  Passes if (*ref* & *mask*) < (*stencil* & *mask*).

GL_LEQUAL
  Passes if (*ref* & *mask*) $\leq$ (*stencil* & *mask*).

GL_GREATER
  Passes if (*ref* & *mask*) > (*stencil* & *mask*).

GL_GEQUAL
  Passes if (*ref* & *mask*) $\geq$ (*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.

The following functions retrieve information related to the **glStencilFunc** function:

glGet with argument **GL_STENCIL_FUNC**

glGet with argument **GL_STENCIL_VALUE_MASK**

glGet with argument **GL_STENCIL_REF**

glGet with argument **GL_STENCIL_BITS**

glIsEnabled with argument **GL_STENCIL_TEST**

Errors

GL_INVALID_ENUM is generated if *func* is not one of the eight accepted values.

GL_INVALID_OPERATION is generated if **glStencilFunc** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glIsEnabled**, **glLogicOp**, **glStencilOp**

glStencilMask

The **glStencilMask** function controls the writing of individual bits in the stencil planes.

void glStencilMask(
  GLuint *mask*
  );

Parameters

mask
  Specifies a bit mask to enable and disable writing of individual bits in the stencil planes. Initially, the mask is all ones.

Remarks

The **glStencilMask** function 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. Wherever a one appears in the mask, the corresponding bit in the stencil buffer is made writable. Where a zero appears, the bit is write-protected. Initially, all bits are enabled for writing.

The following functions retrieve information related to the **glStencilMask** function:

glGet with argument **GL_STENCIL_WRITEMASK**

glGet with argument **GL_STENCIL_BITS**

Errors

GL_INVALID_OPERATION is generated if **glStencilMask** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glColorMask, **glDepthMask**, **glIndexMask**, **glStencilFunc**, **glStencilOp**

glStencilOp

The **glStencilOp** function sets the stencil test actions.

void glStencilOp(
  GLenum *fail*,
  GLenum *zfail*,
  GLenum *zpass*
  );

Parameters

fail
  Specifies the action to take when the stencil test fails. Six symbolic constants are accepted: **GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR,** and **GL_INVERT**.

zfail
  Specifies stencil action when the stencil test passes, but the depth test fails. Accepts the same symbolic constants as *fail.*

zpass
  Specifies 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. Accepts the same symbolic constants as *fail*.

Remarks

Stenciling, like *z*-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. The test is enabled with **glEnable** and **glDisable** calls with argument **GL_STENCIL**, and controlled with **glStencilFunc**.

The **glStencilOp** function 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 pixels color or depth buffers, and *fail* specifies what happens to the stencil buffer contents. The six possible actions are as follows:

GL_KEEP
  Keeps the current value.

GL_ZERO
  Sets the stencil buffer value to zero.

GL_REPLACE
  Sets the stencil buffer value to *ref*, as specified by **glStencilFunc**.

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 zero.

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 **glStencilOp** specify stencil buffer actions should subsequent depth buffer tests succeed (*zpass*) or fail (*zfail*). (See **glDepthFunc**.) They are specified using the same six

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 **glStencilOp**.

The following functions retrieve information related to the **glStencilOp** function:

glGet with argument **GL_STENCIL_FAIL**

glGet with argument **GL_STENCIL_PASS_DEPTH_PASS**

glGet with argument **GL_STENCIL_PASS_DEPTH_FAIL**

glGet with argument **GL_STENCIL_BITS**

glIsEnabled with argument **GL_STENCIL_TEST**

Errors

GL_INVALID_ENUM is generated if *fail*, *zfail*, or *zpass* is any value other than the six defined constant values.

GL_INVALID_OPERATION is generated if **glStencilOp** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glAlphaFunc, **glBlendFunc**, **glDepthFunc**, **glEnable**, **glLogicOp**, **glStencilFunc**

glTexCoord

glTexCoord1d, glTexCoord1f, glTexCoord1i, glTexCoord1s, glTexCoord2d, glTexCoord2f, glTexCoord2i, glTexCoord2s, glTexCoord3d, glTexCoord3f, glTexCoord3i, glTexCoord3s, glTexCoord4d, glTexCoord4f, glTexCoord4i, glTexCoord4s, glTexCoord1dv, glTexCoord1fv, glTexCoord1iv, glTexCoord1sv, glTexCoord2dv, glTexCoord2fv, glTexCoord2iv, glTexCoord2sv, glTexCoord3dv, glTexCoord3fv, glTexCoord3iv, glTexCoord3sv, glTexCoord4dv, glTexCoord4fv, glTexCoord4iv, glTexCoord4sv

These functions set the current texture coordinates.

void glTexCoord1d(
  GLdouble *s*
  );

void glTexCoord1f(
  GLfloat *s*
  );

void glTexCoord1i(
  GLint *s*
  );

void glTexCoord1s(
  GLshort *s*
  );

void glTexCoord2d(
  GLdouble *s*,
  GLdouble *t*
  );

void glTexCoord2f(
  GLfloat *s*,
  GLfloat *t*
  );

void glTexCoord2i(
  GLint *s*,
  GLint *t*
  );

void glTexCoord2s(
  GLshort *s*,
  GLshort *t*
  );

void glTexCoord3d(
  GLdouble *s*,
  GLdouble *t*,
  GLdouble *r*
  );

void glTexCoord3f(
  GLfloat *s*,
  GLfloat *t*,
  GLfloat *r*
  );

void glTexCoord3i(

GLint *s*,
GLint *t*,
GLint *r*
);

void glTexCoord3s(
GLshort *s*,
GLshort *t*,
GLshort *r*
);

void glTexCoord4d(
GLdouble *s*,
GLdouble *t*,
GLdouble *r*,
GLdouble *q*
);

void glTexCoord4f(
GLfloat *s*,
GLfloat *t*,
GLfloat *r*,
GLfloat *q*
);

void glTexCoord4i(
GLint *s*,
GLint *t*,
GLint *r*,
GLint *q*
);

void glTexCoord4s(
GLshort *s*,
GLshort *t*,
GLshort *r*,
GLshort *q*
);

Parameters

s, t, r, q
Specify *s*, *t*, *r*, and *q* texture coordinates. Not all parameters are present in all forms of the command.

void glTexCoord1dv(
const GLdouble **v*
);

void glTexCoord1fv(
const GLfloat **v*
);

void glTexCoord1iv(
const GLint **v*
);

void glTexCoord1sv(
const GLshort **v*

);

void glTexCoord2dv(
const GLdouble *v
);

void glTexCoord2fv(
const GLfloat *v
);

void glTexCoord2iv(
const GLint *v
);

void glTexCoord2sv(
const GLshort *v
);

void glTexCoord3dv(
const GLdouble *v
);

void glTexCoord3fv(
const GLfloat *v
);

void glTexCoord3iv(
const GLint *v
);

void glTexCoord3sv(
const GLshort *v
);

void glTexCoord4dv(
const GLdouble *v
);

void glTexCoord4fv(
const GLfloat *v
);

void glTexCoord4iv(
const GLint *v
);

void glTexCoord4sv(
const GLshort *v
)

Parameters

v
Specifies a pointer to an array of one, two, three, or four elements, which in turn specify the *s, t*, *r,* and *q* texture coordinates.

Remarks

The current texture coordinates are part of the data that is associated with polygon vertexes. They are set with **glTexCoord**.

The **glTexCoord** function specifies texture coordinates in one, two, three, or four dimensions. **glTexCoord1** sets the current texture coordinates to (*s*, 0, 0, 1); a call to **glTexCoord2** sets them to (*s*,

t, 0, 1). Similarly, **glTexCoord3** specifies the texture coordinates as (*s*, *t*, *r*, 1), and **glTexCoord4** defines all four components explicitly as (*s*, *t*, *r*, *q*).

The current texture coordinates can be updated at any time. In particular, **glTexCoord** can be called between a call to **glBegin** and the corresponding call to **glEnd**.

The following function retrieves information related to the **glTexCoord** function:

glGet with argument **GL_CURRENT_TEXTURE_COORDS**

See Also

glVertex

glTexEnvf, glTexEnvi, glTexEnvfv, glTexEnviv

These functions set texture environment parameters.

void glTexEnvf(
  GLenum *target*,
  GLenum *pname*,
  GLfloat *param*
  );

void glTexEnvi(
  GLenum *target*,
  GLenum *pname*,
  GLint *param*
  );

Parameters

target
  Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

pname
  Specifies the symbolic name of a single-valued texture environment parameter. Must be **GL_TEXTURE_ENV_MODE**.

param
  Specifies a single symbolic constant, one of **GL_MODULATE**, **GL_DECAL**, or **GL_BLEND**.

void glTexEnvfv(
  GLenum *target*,
  GLenum *pname*,
  const GLfloat **params*
  );

void glTexEnviv(
  GLenum *target*,
  GLenum *pname*,
  const GLint **params*
  );

Parameters

target
  Specifies a texture environment. Must be **GL_TEXTURE_ENV**.

pname
  Specifies the symbolic name of a texture environment parameter. Accepted values are **GL_TEXTURE_ENV_MODE** and **GL_TEXTURE_ENV_COLOR**.

params
  Specifies a pointer to an array of parameters: either a single symbolic constant or an RGBA color.

Remarks

A texture environment specifies how texture values are interpreted when a fragment is textured. *target* must be **GL_TEXTURE_ENV**. *pname* can be either **GL_TEXTURE_ENV_MODE** or **GL_TEXTURE_ENV_COLOR**.

If *pname* is **GL_TEXTURE_ENV_MODE**, then *params* is (or points to) the symbolic name of a texture function. Three texture functions are defined: **GL_MODULATE**, **GL_DECAL**, and **GL_BLEND**.

A texture function acts on the fragment to be textured using the texture image value that applies to the fragment (see **glTexParameter**) and produces an RGBA color for that fragment. The following table shows how the RGBA color is produced for each of the three texture functions that can be chosen. *C* is a triple of color values (RGB) and *A* is the associated alpha value. RGBA values extracted from a texture image are in the range [0,1]. The subscript *f* refers to the incoming fragment, the subscript *t* to the texture image, the subscript *c* to the texture environment color, and subscript *v* indicates a value produced by the texture function.

A texture image can have up to four components per texture element (see **glTexImage1D** and **glTexImage2D**). In a one-component image, $L_{(t)}$ indicates that single component. A two-component image uses $L_{(t)}$ and $A_{(t)}$. A three-component image has only a color value, $C_{(t)}$. A four-component image has both a color value $C_{(t)}$ and an alpha value $A_{(t)}$.

{ewc msdncd, EWGraphic, group10365 0 /a "SDK.bmp"}

If *pname* is **GL_TEXTURE_ENV_COLOR**, *params* is a pointer to an array that holds an RGBA color consisting of four values. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. $C_{(c)}$ takes these four values.

GL_TEXTURE_ENV_MODE defaults to **GL_MODULATE** and **GL_TEXTURE_ENV_COLOR** defaults to (0,0,0,0).

The following function retrieves information related to the **glTexEnv** function:

glGetTexEnv

Errors

GL_INVALID_ENUM is generated when *target* or *pname* is not one of the accepted defined values, or when *params* should have a defined constant value (based on the value of *pname*) and does not.

GL_INVALID_OPERATION is generated if **glTexEnv** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexImage1D, **glTexImage2D**, **glTexParameter**

glTexGend, glTexGenf, glTexGeni, glTexGendv, glTexGenfv, glTexGeniv

These functions control the generation of texture coordinates.

void glTexGend(
  GLenum *coord*,
  GLenum *pname*,
  GLdouble *param*
  );

void glTexGenf(
  GLenum *coord*,
  GLenum *pname*,
  GLfloat *param*
  );

void glTexGeni(
  GLenum *coord*,
  GLenum *pname*,
  GLint *param*
  );

Parameters

coord
  Specifies a texture coordinate. Must be one of the following: **GL_S, GL_T, GL_R,** or **GL_Q**.

pname
  Specifies the symbolic name of the texture-coordinate generation function. Must be **GL_TEXTURE_GEN_MODE**.

param
  Specifies a single-valued texture generation parameter, one of **GL_OBJECT_LINEAR**, **GL_EYE_LINEAR**, or **GL_SPHERE_MAP**.

void glTexGendv(
  GLenum *coord*,
  GLenum *pname*,
  const GLdouble **params*
  );

void glTexGenfv(
  GLenum *coord*,
  GLenum *pname*,
  const GLfloat **params*
  );

void glTexGeniv(
  GLenum *coord*,
  GLenum *pname*,
  const GLint **params*
  );

Parameters

coord
  Specifies a texture coordinate. Must be one of the following: **GL_S, GL_T, GL_R,** or **GL_Q**.

pname

Specifies the symbolic name of the texture-coordinate generation function or function parameters. Must be **GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE,** or **GL_EYE_PLANE**.

params

Specifies a pointer to an array of texture generation parameters. If *pname* is **GL_TEXTURE_GEN_MODE**, then the array must contain a single symbolic constant, one of **GL_OBJECT_LINEAR**, **GL_EYE_LINEAR**, or **GL_SPHERE_MAP**. Otherwise, params holds the coefficients for the texture-coordinate generation function specified by *pname*.

Remarks

The **glTexGen** function 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, and it must be one of these 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 *params* 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**, *params* contains coefficients for the corresponding texture generation function.

If the texture generation function is **GL_OBJECT_LINEAR**, the function

{ewc msdncd, EWGraphic, group10366 0 /a "SDK.bmp"}

is used, where *g* is the value computed for the coordinate named in *coord*, $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$ are the four values supplied in *params*, and $x_{(o)}$, $y_{(o)}$, $z_{(o)}$, and $w_{(o)}$ are the object coordinates of the vertex. This function can be used to texture-map terrain using sea level as a reference plane (defined by $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$). The altitude of a terrain vertex is computed by the **GL_OBJECT_LINEAR** coordinate generation function as its distance from sea level; that altitude is used to index the texture image to map white snow onto peaks and green grass onto foothills, for example.

If the texture generation function is **GL_EYE_LINEAR**, the function

is used, where

{ewc msdncd, EWGraphic, group10366 1 /a "SDK.bmp"}

and $x_{(e)}$, $y_{(e)}$, $z_{(e)}$, and $w_{(e)}$ are the eye coordinates of the vertex, $p_{(1)}$, $p_{(2)}$, $p_{(3)}$, and $p_{(4)}$ are the values supplied in *params*, and *M* is the modelview matrix when **glTexGen** 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 *params* 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 vertexes 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 $\mathbf{n}'$ be the current normal, after transformation to eye coordinates. Let $f = (f_{(x)}\, f_{(y)}\, f_{(z)})^T$ be the reflection vector such that

{ewc msdncd, EWGraphic, group10366 2 /a "SDK.bmp"}

Finally, let

{ewc msdncd, EWGraphic, group10366 3 /a "SDK.bmp"}

Then the values assigned to the *i* and *t* texture coordinates are

{ewc msdncd, EWGraphic, group10366 4 /a "SDK.bmp"}

A texture-coordinate generation function is enabled or disabled using **glEnable** or **glDisable** 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 vertexes 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).

The following functions retrieve information related to the **glTexGen** function:

glGetTexGen
glIsEnabled with argument **GL_TEXTURE_GEN_S**
glIsEnabled with argument **GL_TEXTURE_GEN_T**
glIsEnabled with argument **GL_TEXTURE_GEN_R**
glIsEnabled with argument **GL_TEXTURE_GEN_Q**

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 *params* is not an accepted defined value.

GL_INVALID_ENUM is generated when *pname* is **GL_TEXTURE_GEN_MODE**, *params* is **GL_SPHERE_MAP**, and *coord* is either **GL_R** or **GL_Q**.

GL_INVALID_OPERATION is generated if **glTexGen** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexEnv, **glTexImage1D**, **glTexImage2D**, **glTexParameter**

glTexImage1D

The **glTexImage1D** function specifies a one-dimensional texture image.

void glTexImage1D(
  GLenum *target*,
  GLint *level*,
  GLint *components*,
  GLsizei *width*,
  GLint *border*,
  GLenum *format*,
  GLenum *type*,
  const GLvoid **pixels*
  );

Parameters

target
  Specifies the target texture. Must be **GL_TEXTURE_1D**.

level
  Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

components
  Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.

width
  Specifies the width of the texture image. Must be 2^*n* + 2(*border*) for some integer *n*. The height of the texture image is 1.

border
  Specifies the width of the border. Must be either 0 or 1.

format
  Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE,** and **GL_LUMINANCE_ALPHA**.

type
  Specifies the data type of the pixel data. The following symbolic values are accepted: **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** and **GL_FLOAT**.

pixels
  Specifies a pointer to the image data in memory.

Remarks

Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. One-dimensional texturing is enabled and disabled using **glEnable** and **glDisable** with argument **GL_TEXTURE_1D**.

Texture images are defined with **glTexImage1D**. The arguments describe the parameters of the texture image, such as width, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe the way the image is represented in memory, and 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*. 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

glPixelStore).

The *format* parameter 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. It is converted 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 glPixelTransfer). 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. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 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 **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 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 glPixelTransfer).

GL_BLUE
Each element is a single blue component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 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 **glPixelTransfer**).

GL_ALPHA
Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 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 glPixelTransfer).

GL_RGB
Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 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 **glPixelTransfer**).

GL_RGBA
Each element is a complete RGBA element. It is converted to floating point. 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 **glPixelTransfer**).

GL_LUMINANCE
Each element is a single luminance value. It is converted to floating point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1.0 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 glPixelTransfer).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. It is converted to floating point, then assembled 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 **glPixelTransfer**).

A texture image can have up to four components per texture element, depending on *components*. 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 **glDrawPixels** command, except that **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** cannot be used. **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to the **glTexImage1D** function:

glGetTexImage

glIsEnabled with argument **GL_TEXTURE_1D**

Errors

GL_INVALID_ENUM is generated when *target* is not **GL_TEXTURE_1D**.

GL_INVALID_ENUM is generated when *format* is not an accepted *format* constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated when *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 zero or greater than $\log_{(2)} max$, where *max* is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *components* is not 1, 2, 3, or 4.

GL_INVALID_VALUE is generated if *width* is less than zero or greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if it 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 **glTexImage1D** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage2D**, **glTexParameter**

glTexImage2D

The **glTexImage2D** function specifies a two-dimensional texture image.

void glTexImage2D(
  GLenum *target*,
  GLint *level*,
  GLint *components*,
  GLsizei *width*,
  GLsizei *height*,
  GLint *border*,
  GLenum *format*,
  GLenum *type*,
  const GLvoid **pixels*
  );

Parameters

target
  Specifies the target texture. Must be **GL_TEXTURE_2D**.

level
  Specifies the level-of-detail number. Level 0 is the base image level. Level *n* is the *n*th mipmap reduction image.

components
  Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.

width
  Specifies the width of the texture image. Must be 2^*n* + 2(*border*) for some integer *n*.

height
  Specifies the height of the texture image. Must be 2^*m* + 2(*border*) for some integer *m*.

border
  Specifies the width of the border. Must be either 0 or 1.

format
  Specifies the format of the pixel data. The following symbolic values are accepted: **GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE,** and **GL_LUMINANCE_ALPHA.**

type
  Specifies the data type of the pixel data. The following symbolic values are accepted: **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** and **GL_FLOAT**.

pixels
  Specifies a pointer to the image data in memory.

Remarks

Texturing maps a portion of a specified *texture image* onto each graphical primitive for which texturing is enabled. Two-dimensional texturing is enabled and disabled using **glEnable** and **glDisable** with argument **GL_TEXTURE_2D**.

Texture images are defined with **glTexImage2D**. The arguments describe the parameters of the texture image, such as height, width, width of the border, level-of-detail number (see **glTexParameter**), and number of color components provided. The last three arguments describe the way the image is represented in memory, and 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*. 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 **glPixelStore**).

The *format* parameter 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. It is converted 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 **glPixelTransfer**). 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. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for green and blue, and 1.0 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 **glPixelTransfer**).

GL_GREEN
Each element is a single green component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and blue, and 1.0 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 **glPixelTransfer**).

GL_BLUE
Each element is a single blue component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 for red and green, and 1.0 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 **glPixelTransfer**).

GL_ALPHA
Each element is a single red component. It is converted to floating point and assembled into an RGBA element by attaching 0.0 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 **glPixelTransfer**).

GL_RGB
Each element is an RGB triple. It is converted to floating point and assembled into an RGBA element by attaching 1.0 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 **glPixelTransfer**).

GL_RGBA
Each element is a complete RGBA element. It is converted to floating point. 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 **glPixelTransfer**).

GL_LUMINANCE
Each element is a single luminance value. It is converted to floating point, then assembled into an RGBA element by replicating the luminance value three times for red, green, and blue and attaching 1.0 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 **glPixelTransfer**).

GL_LUMINANCE_ALPHA
Each element is a luminance/alpha pair. It is converted to floating point, then assembled 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 **glPixelTransfer**).

Please see **glDrawPixels** for a description of the acceptable values for the *type* parameter. A texture image can have up to four components per texture element, depending on *components*. 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 **glDrawPixels** command, except that **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** cannot be used. **glPixelStore** and **glPixelTransfer** modes affect texture images in exactly the way they affect **glDrawPixels**.

A texture image with zero height or width indicates the null texture. If the null texture is specified for level-of-detail 0, it is as if texturing were disabled.

The following functions retrieve information related to the **glTexImage2D** function:

glGetTexImage
glIsEnabled with argument **GL_TEXTURE_2D**

Errors

GL_INVALID_ENUM is generated when *target* is not **GL_TEXTURE_2D**.

GL_INVALID_ENUM is generated when *format* is not an accepted *format* constant. Format constants other than **GL_STENCIL_INDEX** and **GL_DEPTH_COMPONENT** are accepted.

GL_INVALID_ENUM is generated when *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 zero or greater than $\log_{(2)}$ max, where *max* is the returned value of **GL_MAX_TEXTURE_SIZE**.

GL_INVALID_VALUE is generated if *components* is not 1, 2, 3, or 4.

GL_INVALID_VALUE is generated if *width* or *height* is less than zero or greater than 2 + **GL_MAX_TEXTURE_SIZE**, or if either cannot be represented as 2^*k* + 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 **glTexImage2D** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDrawPixels, **glFog**, **glPixelStore**, **glPixelTransfer**, **glTexEnv**, **glTexGen**, **glTexImage1D**, **glTexParameter**

glTexParameterf, glTexParameteri, glTexParameterfv, glTexParameteriv

These functions set texture parameters.

void glTexParameterf(
  GLenum *target*,
  GLenum *pname*,
  GLfloat *param*
  );

void glTexParameteri(
  GLenum *target*,
  GLenum *pname*,
  GLint *param*
  );

Parameters

target
  Specifies the target texture, which must be either **GL_TEXTURE_1D** or **GL_TEXTURE_2D**.

pname
  Specifies the symbolic name of a single-valued texture parameter. *pname* can be one of the following: **GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S,** or **GL_TEXTURE_WRAP_T**.

param
  Specifies the value of *pname*.

void glTexParameterfv(
  GLenum *target*,
  GLenum *pname*,
  const GLfloat **params*
  );

void glTexParameteriv(
  GLenum *target*,
  GLenum *pname*,
  const GLint **params*
  );

Parameters

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.The *pname* parameter can be one of the following: **GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER,**
  GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, or **GL_TEXTURE_BORDER_COLOR.**

params
  Specifies a pointer to an array where the value or values of *pname* are stored.

Remarks

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.

The **glTexParameter** function assigns the value or values in *params* 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. If the texture has dimensions 2^*n*x2^*m* there are **max**(*n*, *m*) + 1 mipmaps. The first mipmap is the original texture, with dimensions 2^*n*x2^*m*. Each subsequent mipmap has dimensions 2^*k* (-1) x2^*l* (-1) where 2^*k*x2^*l* are the dimensions of the previous mipmap, until either *k* = 0 or *l* = 0. At that point, subsequent mipmaps have dimension 1x2^*l-1* or 2^*k-1*x1 until the final mipmap, which has dimension 1x1. Mipmaps are defined using **glTexImage1D** or **glTexImage2D** with the level-of-detail argument indicating the order of the mipmaps. Level 0 is the original texture; level bold max(*n, m*) is the final 1x1 mipmap.

The *params* parameter 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 default 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 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 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 default value of **GL_TEXTURE_MAG_FILTER** is **GL_LINEAR**.

GL_TEXTURE_WRAP_S
Sets the wrap parameter for texture coordinate *s* to either **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 either **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_BORDER_COLOR
Sets a border color. The *params* parameter contains four values that comprise the RGBA color of the texture border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. The values are clamped to the range [0,1] when they are specified. Initially, the border color is (0, 0, 0, 0).

Suppose texturing is enabled (by calling **glEnable** with argument **GL_TEXTURE_1D** or **GL_TEXTURE_2D**) and **GL_TEXTURE_MIN_FILTER** is set to one of the functions that requires a mipmap. If either the dimensions of the texture images currently defined (with previous calls to **glTexImage1D** or **glTexImage2D**) do not follow the proper sequence for mipmaps, 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 2-D textures. In 1-D textures, linear filtering accesses the two nearest texture elements.

The following functions retrieve information related to the **glTexParameterf, glTexParameteri, glTexParameterfv,** and **glTexParameteriv** functions:

glGetTexParameter

glGetTexLevelParameter

Errors

GL_INVALID_ENUM is generated when *target* or *pname* is not one of the accepted defined values, or when *params* should have a defined constant value (based on the value of *pname*) and does not.

GL_INVALID_OPERATION is generated if **glTexParameter** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glTexEnv, **glTexImage1D**, **glTexImage2D**, **glTexGen**

glTranslated, glTranslatef

The **glTranslated** and **glTranslatef** functions multiply the current matrix by a translation matrix.

void glTranslated(
GLdouble *x*,
GLdouble *y*,
GLdouble *z*
);

void glTranslatef(
GLfloat *x*,
GLfloat *y*,
GLfloat *z*
);

Parameters

x, y, z
Specify the *x*, *y*, and *z* coordinates of a translation vector.

Remarks

The **glTranslate** function moves the coordinate system origin to the point specified by (*x*,*y*,*z*). The translation vector is used to compute a 4x4 translation matrix:

{ewc msdncd, EWGraphic, group10370 0 /a "SDK.bmp"}

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M•T.

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glTranslate** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to the **glTranslated** and **glTranslatef** functions:

glGet with argument **GL_MATRIX_MODE**

glGet with argument **GL_MODELVIEW_MATRIX**

glGet with argument **GL_PROJECTION_MATRIX**

glGet with argument **GL_TEXTURE_MATRIX**

Errors

GL_INVALID_OPERATION is generated if **glTranslate** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glMatrixMode, **glMultMatrix**, **glPushMatrix**, **glRotate**, **glScale**

glVertex

glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d, glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f, glVertex4i, glVertex4s, glVertex2dv, glVertex2fv, glVertex2iv, glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv, glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv

These functions specify a vertex.

void glVertex2d(
  GLdouble *x*,
  GLdouble *y*
  );

void glVertex2f(
  GLfloat *x*,
  GLfloat *y*
  );

void glVertex2i(
  GLint *x*,
  GLint *y*
  );

void glVertex2s(
  GLshort *x*,
  GLshort *y*
  );

void glVertex3d(
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*
  );

void glVertex3f(
  GLfloat *x*,
  GLfloat *y*,
  GLfloat *z*
  );

void glVertex3i(
  GLint *x*,
  GLint *y*,
  GLint *z*
  );

void glVertex3s(
  GLshort *x*,
  GLshort *y*,
  GLshort *z*
  );

void glVertex4d(
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*,
  GLdouble *w*
  );

void glVertex4f(
GLfloat *x*,
GLfloat *y*,
GLfloat *z*,
GLfloat *w*
);

void glVertex4i(
GLint *x*,
GLint *y*,
GLint *z*,
GLint *w*
);

void glVertex4s(
GLshort *x*,
GLshort *y*,
GLshort *z*,
GLshort *w*
);

Parameters

x, y, z, w
Specify *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

void glVertex2dv(
const GLdouble **v*
);

void glVertex2fv(
const GLfloat **v*
);

void glVertex2iv(
const GLint **v*
);

void glVertex2sv(
const GLshort **v*
);

void glVertex3dv(
const GLdouble **v*
);

void glVertex3fv(
const GLfloat **v*
);

void glVertex3iv(
const GLint **v*
);

void glVertex3sv(
const GLshort **v*
);

void glVertex4dv(
  const GLdouble *v
  );

void glVertex4fv(
  const GLfloat *v
  );

void glVertex4iv(
  const GLint *v
  );

void glVertex4sv(
  const GLshort *v
  );

Parameters

v
  Specifies a pointer to an array of two, three, or four elements. The elements of a two-element array are *x* and *y*; of a three-element array, *x*, *y*, and *z*; and of a four-element array, *x*, *y*, *z*, and *w*.

Remarks

The **glVertex** function commands are used within **glBegin**/**glEnd** pairs to specify point, line, and polygon vertexes. The current color, normal, and texture coordinates are associated with the vertex when **glVertex** is called.

When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x, y,* and *z* are specified, *w* defaults to 1.0.

Invoking **glVertex** outside of a **glBegin**/**glEnd** pair results in undefined behavior.

See Also

glBegin, **glCallList**, **glColor**, **glEdgeFlag**, **glEvalCoord**, **glIndex**, **glMaterial**, **glNormal**, **glRect**, **glTexCoord**

glViewport

The **glViewport** function sets the viewport.

void glViewport(
  GLint *x*,
  GLint *y*,
  GLsizei *width*,
  GLsizei *height*
  );

Parameters

x, y
  Specify the lower-left corner of the viewport rectangle, in pixels. The default is (0,0).

width, height
  Specify the width and height, respectively, of the viewport. When a GL context is *first* attached to a window, *width* and *height* are set to the dimensions of that window.

Remarks

The **glViewport** function specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let ($x_{(nd)}$, $y_{(nd)}$) be normalized device coordinates. Then the window coordinates ($x_{(w)}$, $y_{(w)}$) are computed as follows:

{ewc msdncd, EWGraphic, group10372 0 /a "SDK.bmp"}

Viewport width and height are silently clamped to a range that depends on the implementation. This range is queried by calling **glGet** with argument **GL_MAX_VIEWPORT_DIMS**.

The following functions retrieve information related to the **glViewport** function:

glGet with argument **GL_VIEWPORT**

glGet with argument **GL_MAX_VIEWPORT_DIMS**

Errors

GL_INVALID_VALUE is generated if either *width* or *height* is negative.

GL_INVALID_OPERATION is generated if **glViewport** is called between a call to **glBegin** and the corresponding call to **glEnd**.

See Also

glDepthRange

OpenGL Glossary

A

aliasing
A rendering technique that assigns to pixels the color of the primitive being rendered, regardless of whether that primitive covers all of the pixel's area or only a portion of the pixel's area. This results in jagged edges, or **jaggies**

alpha
A fourth color component typically used to control color blending. The alpha component is never displayed directly. By convention, OpenGL alpha corresponds to opacity rather than transparency, meaning an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 implies complete transparency.

animation
Generating repeated renderings of a scene, with smoothly changing viewpoint and/or object positions, quickly enough that the illusion of motion is achieved. OpenGL animation almost always is done using double-buffering.

antialiasing
A rendering technique that assigns pixel colors based on the fraction of the pixel's area that's covered by the primitive being rendered. Antialiased rendering reduces or eliminates the jaggies that result from aliased rendering.

application-specific clipping
Clipping of primitives against planes in eye coordinates. The planes are specified by the application using **glClipPlane( )**.

B

back face
See **face**

bit
Binary digit. A state variable that has only two possible values: 0 or 1. Binary numbers are constructions of one or more bits.

bitmap
A rectangular array of bits. Also, the primitive rendered by the **glBitmap( )** command, which uses its *bitmap* parameter as a mask.

bitplane
A rectangular array of bits mapped one-to-one with pixels. The frame buffer is a stack of bitplanes.

blending
Reduction of two color components to one component, usually as a linear interpolation between the two components.

buffer
A group of bitplanes that store a single component (such as depth or green) or a single index

(such as the color index or the stencil index). Sometimes the red, green, blue, and alpha buffers together are referred to as the color buffer, rather than the color buffers.

C

client
The computer from which OpenGL commands are issued. The computer that issues OpenGL commands can be connected through a network to a different computer that executes the commands, or commands can be issued and executed on the same computer. See also **server**

client memory
The main memory (where program variables are stored) of the client computer.

clip coordinates
The coordinate system that follows transformation by the projection matrix and that precedes perspective division. View-volume clipping is done in clip coordinates, but application-specific clipping is not.

clipping
Elimination of the portion of a geometric primitive that's outside the half-space defined by a clipping plane. Points are simply rejected if outside. The portion of a line or of a polygon that's outside the half-space is eliminated, and additional vertices are generated as necessary to complete the primitive within the clipping half-space. Geometric primitives and the current raster position (when specified) are always clipped against the six half-spaces defined by the left, right, bottom, top, near, and far planes of the view volume. Applications can specify optional application-specific clipping planes to be applied in eye coordinates.

color index
A single value that represents a color by name, rather than by value. OpenGL color indexes are treated as continuous values (for example, floating-point numbers) while operations such as interpolation and dithering are performed on them. Color indexes stored in the frame buffer are always integer values, however. Floating-point indexes are converted to integers by rounding to the nearest integer value.

color-index mode
An OpenGL context is in color index mode if its color buffers store color indexes, rather than red, green, blue, and alpha color components.

color map
A table of index-to-RGB mappings that's accessed by the display hardware. Each color index is read from the color buffer, converted to an RGB triple by lookup in the color map, and sent to the monitor.

component
A single, continuous (for example, floating-point) value that represents an intensity or quantity. Usually, a component value of zero represents the minimum value or intensity, and a component value of one represents the maximum value or intensity, though other normalizations are sometimes used. Because component values are interpreted in a normalized range, they are specified independent of actual resolution. For example, the RGB triple (1, 1, 1) is white, regardless of whether the color buffers store 4, 8, or 12 bits each.

Out-of-range components are typically clamped to the normalized range, not truncated or

otherwise interpreted. For example, the RGB triple (1.4, 1.5, 0.9) is clamped to (1.0, 1.0, 0.9) before it's used to update the color buffer. Red, green, blue, alpha, and depth are always treated as components, never as indexes.

context
A complete set of OpenGL state variables. Note that frame buffer contents are not part of the OpenGL state, but that the configuration of the frame buffer is.

convex
A polygon is convex if no straight line in the plane of the polygon intersects the polygon more than twice.

convex hull
The smallest convex region enclosing a specified group of points. In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band.

coordinate system
In n-dimensional space, a set of n linearly independent vectors anchored to a point (called the origin). A group of coordinates specifies a point in space (or a vector from the origin) by indicating how far to travel along each vector to reach the point (or tip of the vector).

culling
The process of eliminating a front face or back face of a polygon so that it isn't drawn.

current matrix
A matrix that transforms coordinates in one coordinate system to coordinates of another system. There are three current matrices in OpenGL: the modelview matrix transforms object coordinates (coordinates specified by the programmer) to eye coordinates; the perspective matrix transforms eye coordinates to clip coordinates; the texture matrix transforms specified or generated texture coordinates as described by the matrix. Each current matrix is the top element on a stack of matrices. Each of the three stacks can be manipulated with OpenGL matrix-manipulation commands.

current raster position
A window coordinate position that specifies the placement of an image primitive when it's rasterized. The current raster position, and other current raster parameters, are updated when **glRasterpos( )** is called.

D

depth
Generally refers to the *z* window coordinate.

depth-cueing
A rendering technique that assigns color based on distance from the viewpoint.

display list
A named list of OpenGL commands. Display lists are always stored on the server, so display lists can be used to reduce the network traffic in client-server environments. The contents of a display list may be preprocessed, and might therefore execute more efficiently than the same set of OpenGL commands executed in immediate mode. Such preprocessing is especially important for

computing intensive commands such as **glTexImage( )**.

dithering
A technique for increasing the perceived range of colors in an image at the cost of spatial resolution. Adjacent pixels are assigned differing color values. When viewed from a distance, these colors seem to blend into a single intermediate color. The technique is similar to the half-toning used in black-and-white publications to achieve shades of gray.

double-buffering
OpenGL contexts with both front and back color buffers are double-buffered. Smooth animation is accomplished by rendering into only the back buffer (which isn't displayed), then causing the front and back buffers to be swapped.

E

element
A single component or index.

evaluation
The OpenGL process of generating object-coordinate vertices and parameters from previously specified Bézier equations.

execute
An OpenGL command is executed when it's called in immediate mode or when the display list that it's a part of is called.

eye coordinates
The coordinate system that follows transformation by the modelview matrix and that precedes transformation by the projection matrix. Lighting and application-specific clipping are done in eye coordinates.

F

face
One side of a polygon. Each polygon has two faces: a front face and a back face. Only one face or the other is ever visible in the window. Whether the back or front face is visible is effectively determined after the polygon is projected onto the window. After this projection, if the polygon's edges are directed clockwise, one of the faces is visible; if directed counterclockwise, the other face is visible. Whether clockwise corresponds to front or back (and counterclockwise corresponds to back or front) is determined by the OpenGL programmer.

flat shading
Refers to coloring a primitive with a single, constant color across its extent, rather than smoothly interpolating colors across the primitive. See **Gouraud shading**

fog
A rendering technique that can be used to simulate atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer. Fog also aids in the perception of distance from the viewer, giving a **depth cue**

font
A group of graphical character representations usually used to display strings of text. The

characters may be roman letters, mathematical symbols, Asian ideograms, Egyptian hieroglyphs, and so on.

fragment
Fragments are generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes color, depth, and sometimes texture-coordinate values.

frame buffer
All the buffers of a given window or context. Sometimes includes all the pixel memory of the graphics hardware accelerator.

front face
See **face**

frustum
The view volume warped by perspective division.

G

gamma correction
A function applied to colors stored in the frame buffer to correct for the nonlinear response of the eye (and sometimes of the monitor) to linear changes in color-intensity values.

geometric model
The object-coordinate vertices and parameters that describe an object. Note that OpenGL doesn't define a syntax for geometric models, but rather a syntax and semantics for the rendering of geometric models.

geometric object
Geometric model.

geometric primitive
A point, a line, or a polygon.

Gouraud shading
Smooth interpolation of colors across a polygon or line segment. Colors are assigned at vertices and linearly interpolated across the primitive to produce a relatively smooth variation in color. Also called *smooth shading*.

group
Each pixel of an image in client memory is represented by a group of one, two, three, or four elements. Thus, in the context of a client memory image, a group and a pixel are the same thing.

H

half-space
A plane divides space into two half-spaces.

homogenous coordinates
A set of n+1 coordinates used to represent points in n-dimensional projective space. Points in projective space can be thought of as points in Euclidean space together with some points at

infinity. The coordinates are homogenous because a scaling of each of the coordinates by the same non-zero constant doesn't alter the point to which the coordinates refer. Homogeneous coordinates are useful in the calculations of projective geometry, and thus in computer graphics, where scenes must be projected onto a window.

I

image
A rectangular array of pixels, either in client memory or in the frame buffer.

image primitive
A bitmap or an image.

immediate mode
Execution of OpenGL commands when they're called, rather than from a display list. No immediate-mode bit exists; the *mode* in immediate mode refers to usage of OpenGL, rather than to a specific bit of OpenGL state.

index
A single value that's interpreted as an absolute value, rather than as a normalized value in a specified range (as is a component). Color indexes are the names of colors, which are dereferenced by the display hardware using the color map. Indexes are typically masked, rather than clamped, when out of range. For example, the index 0xf7 is masked to 0x7 when written to a 4-bit buffer (color or stencil). Color indexes and stencil indexes are always treated as indexes, never as components.

IRIS GL
Silicon Graphics' proprietary graphics library, developed from 1982 through 1992. OpenGL was designed with IRIS GL as a starting point.

JK

jaggies
Artifacts of aliased rendering. The edges of primitives that are rendered with aliasing are jagged rather than smooth. A near-horizontal aliased line, for example, is rendered as a set of horizontal lines on adjacent pixel rows, rather than as a smooth, continuous line.

L

lighting
The process of computing the color of a vertex based on current lights, material properties, and lighting-model modes.

line
A straight region of finite width between two vertices. (Unlike mathematical lines, OpenGL lines have finite width and length.) Each segment of a strip of lines is itself a line.

luminance
The perceived brightness of a surface. Often refers to a weighted average of red, green, and blue color values that gives the perceived brightness of the combination.

M

matrices
Plural of *matrix*.

matrix
A two-dimensional array of values. OpenGL matrices are all 4x4, though when they are stored in client memory they're treated as 1x16 single-dimension arrays.

modelview matrix
The 4x4 matrix that transforms points, lines, polygons, and raster positions from object coordinates to eye coordinates.

monitor
The device that displays the image in the frame buffer.

motion blurring
A technique that simulates what you get on a piece of film when you take a picture of a moving object, or when you move the camera when you take a picture of a stationary object. In animations without motion blur, moving objects can appear jerky.

N

network
A connection between two or more computers that allows each to transfer data to and from the others.

nonconvex
A polygon is nonconvex if there exists a line in the plane of the polygon that intersects the polygon more than twice.

normal
A three-component plane equation that defines the angular orientation, but not position, of a plane or surface.

normalize
Divide each of the components of a normal by the square root of the sum of their squares. Then, if the normal is thought of as a vector from the origin to the point (nx´, ny´, nx´), this vector has unit length

nx´ = nx/factor
ny´ = ny/factor
nz´ = nz/factor

normal vector
Same as *normal*.

NURBS
Non-Uniform Rational B-Spline. A common way to specify parametric curves and surfaces.

O

object

An object-coordinate model that's rendered as a collection of primitives.

object coordinates
Coordinate system prior to any OpenGL transformation.

orthographic
Nonperspective projection, as in some engineering drawings, with no foreshortening.

P

parameter
A value passed as an argument to an OpenGL command. Sometimes one of the values passed by reference to an OpenGL command.

perspective division
The division of *x*, *y*, and *z* by *w*, carried out in clip coordinates.

pixel
Picture element. The bits at location (*x*, *y*) of all the bitplanes in the frame buffer constitute the single pixel (*x*, *y*). In an image in client memory, a pixel is one group of elements. In OpenGL window coordinates, each pixel corresponds to a 1.0x1.0 screen area. The coordinates of the lower left corner of the pixel names *x*, *y* are (*x*, *y*), and the upper right corner are (*x*+1, *y*+1).

point
An exact location in space, which is rendered as a finite-diameter dot.

polygon
A near-planar surface bounded by edges specified by vertices. Each triangle of a triangle mesh is a polygon, as is each quadrilateral of a quadrilateral mesh. The rectangle specified by **glRect*( )** is also a polygon.

primitive
A shape (such as a point, line, polygon, bitmap or image) that can be drawn, stored, and manipulated as a discrete entity; elements from which large graphic designs are created.

projection matrix
The 4x4 matrix that transforms points, lines, polygons, and raster positions from eye coordinates to clip coordinates.

Q

quadrilateral
A polygon with four edges.

R

rasterize
Convert a projected point, line, or polygon, or the pixels of a bitmap or image, to fragments, each corresponding to a pixel in the frame buffer. Note that all primitives are rasterized, not just points, lines, and polygons.

rectangle

A quadrilateral whose alternate edges are parallel to each other in object coordinates. Polygons specified with **glRect*( )** are always rectangles; other quadrilaterals might be rectangles.

rendering
Conversion of primitives specified in object coordinates to an image in the frame buffer. Rendering is the primary operation of OpenGL.

RGBA
Red, Green, Blue, Alpha

RGBA mode
An OpenGL context is in RGBA mode if its color buffers store red, green, blue, and alpha color components, rather than color indexes.

S

server
The computer on which OpenGL commands are executed. This might differ from the computer from which commands are issued. See **client**

shading
The process of interpolating color within the interior of a polygon, or between the vertices of a line, during rasterization.

single-buffering
OpenGL contexts that don't have back color buffers are single-buffered.

stipple
A one- or two-dimensional binary pattern that defeats the generation of fragments where its value is zero. Line stipples are one-dimensional and are applied relative to the start of a line. Polygon stipples are two-dimensional and are applied with a fixed orientation to the window.

T

tessellation
Reduction of a portion of an analytic surface to a mesh of polygons, or of a portion of an analytic curve to a sequence of lines.

texel
A texture element. A texel is obtained from texture memory and represents the color of the texture to be applied to a corresponding fragment.

texture
A one- or two-dimensional image used to modify the color of fragments produced by rasterization.

texture mapping
The process of applying an image (the texture) to a primitive. Texture mapping is often used to add realism to a scene. For example, you could apply a picture of a building facade to a polygon representing a wall.

texture matrix

The 4x4 matrix that transforms texture coordinates from the coordinates that they're specified in to the coordinates that are used for interpolation and texture lookup.

transformation
A warping of space. In OpenGL, transformations are limited to projective transformations that include anything that can be represented by a 4x4 matrix. Such transformations include rotations, translations, (nonuniform) scalings along the coordinate axes, perspective transformations, and combinations of these.

triangle
A polygon with three edges. Triangles are always convex.

U V

vertex
A point in three-dimensional space.

vertices
Preferred plural of vertex.

viewpoint
The origin of either the eye- or the clip-coordinate system, depending on context. (For example, when discussing lighting, the viewpoint is the origin of the eye-coordinate system. When discussing projection, the viewpoint is the origin of the clip-coordinate system.) With a typical projection matrix, the eye-coordinate and clip-coordinate origins are at the same location.

view volume
The volume in clip coordinates whose coordinates satisfy the three conditions
- $-w \leq x \leq w$
- $-w \leq y \leq w$
- $-w \leq z \leq w$

W

window
A subregion of the frame buffer, usually rectangular, whose pixels all have the same buffer configuration. An OpenGL context renders to a single window at a time.

window-aligned
When referring to line segments or polygon edges, implies that these are parallel to the window boundaries. (In OpenGL, the window is rectangular, with horizontal and vertical edges). When referring to a polygon pattern, implies that the pattern is fixed relative to the window origin.

window coordinates
The coordinate system of a window. It's important to distinguish between the names of pixels, which are discrete, and the window-coordinate system, which is continuous. For example, the pixel at the lower-left corner of a window is a pixel (0, 0); the window coordinates of the center of this pixel are (0.5, 0.5, *z*). Note that window coordinates include a depth, or *z*, component, and that this component is continuous as well.

wireframe
A representation of an object that contains line segments only. Typically, the line segments

indicate polygon edges.

X Y Z

X Window System
A window system used by many of the machines on which OpenGL is implemented.

gluBeginCurve, gluEndCurve

The **gluBeginCurve** and **gluEndCurve** functions delimit a NURBS curve definition.

void gluBeginCurve(
  GLUnurbsObj **nobj*
  );

void gluEndCurve(
  GLUnurbsObj **nobj*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

Remarks

Use the **gluBeginCurve** function to mark the beginning of a NURBS curve definition. After calling **gluBeginCurve**, make one or more calls to **gluNurbsCurve** to define the attributes of the curve. Exactly one of the calls to **gluNurbsCurve** must have a curve type of **GL_MAP1_VERTEX_3** or **GL_MAP1_VERTEX_4**. To mark the end of the NURBS curve definition, call **gluEndCurve**.

OpenGL evaluators are used to render the NURBS curve as a series of line segments. Evaluator state is preserved during rendering with **glPushAttrib**(**GL_EVAL_BIT**) and **glPopAttrib**( ). See **glPushAttrib** for details on exactly what state these calls preserve.

Example

The following commands render a textured NURBS curve with normals; texture coordinates and normals are also specified as NURBS curves:

```
gluBeginCurve(nobj);
  gluNurbsCurve(nobj, . . ., GL_MAP1_TEXTURE_COORD_2);
  gluNurbsCurve(nobj, . . ., GL_MAP1_NORMAL);
  gluNurbsCurve(nobj, . . ., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

See Also

gluBeginSurface, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **glPopAttrib**, **glPushAttrib**

gluBeginPolygon, gluEndPolygon

The **gluBeginPolygon** and **gluEndPolygon** functions delimit a polygon description.

void gluBeginPolygon(
  GLUtesselator **tess*
  );

void gluEndPolygon(
  GLUtesselator **tess*
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

Remarks

Use the **gluBeginPolygon** and **gluEndPolygon** functions to delimit the definition of a nonconvex polygon. To define such a polygon, first call **gluBeginPolygon**. Then define the contours of the polygon by calling **gluTessVertex** for each vertex and **gluNextContour** to start each new contour. Finally, call **gluEndPolygon** to signal the end of the definition. See **gluTessVertex** and **gluNextContour** for more details.

Once **gluEndPolygon** is called, the polygon is tessellated, and the resulting triangles are described through callbacks. See **gluTessCallback** for descriptions of the callback functions.

Note The **gluBeginPolygon** and **gluEndPolygon** functions are obsolete and are provided for backward compatibility only. **gluBeginPolygon** is mapped to **gluTessBeginPolygon** followed by **gluTessBeginContour**; **gluEndPolygon** is mapped to **gluTessEndPolygon** followed by **gluTessEndContour**.

Example

A quadrilateral with a triangular hole in it can be described like this:

```
gluBeginPolygon(tess);
  gluTessVertex(tess, v1, v1);
   gluTessVertex(tess, v2, v2);
   gluTessVertex(tess, v3, v3);
   gluTessVertex(tess, v4, v4);
gluNextContour(tess, GLU_INTERIOR);
   gluTessVertex(tess, v5, v5);
   gluTessVertex(tess, v6, v6);
   gluTessVertex(tess, v7, v7);
gluEndPolygon(tess);
```

See Also

gluNewTess, **gluNextContour**, **gluTessCallback**, **gluTessVertex**, **gluTessBeginPolygon**, **gluTessBeginContour**

gluBeginSurface, gluEndSurface

The **gluBeginSurface** and **gluEndSurface** functions delimit a NURBS surface definition.

void gluBeginSurface(
  GLUnurbsObj **nobj*
  );

void gluEndSurface(
  GLUnurbsObj **nobj*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

Remarks

Use the **gluBeginSurface** function to mark the beginning of a NURBS surface definition. After calling **gluBeginSurface**, make one or more calls to **gluNurbsSurface** to define the attributes of the surface. Exactly one of these calls to **gluNurbsSurface** must have a surface type of **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4**. To mark the end of the NURBS surface definition, call **gluEndSurface**.

Trimming of NURBS surfaces is supported with **gluBeginTrim**, **gluPwlCurve**, **gluNurbsCurve**, and **gluEndTrim**. See **gluBeginTrim** for details.

OpenGL evaluators are used to render the NURBS surface as a set of polygons. Evaluator state is preserved during rendering with **glPushAttrib**(**GL_EVAL_BIT**) and **glPopAttrib**( ). See **glPushAttrib** for details on exactly what state these calls preserve.

Example

The following commands render a textured NURBS surface with normals; the texture coordinates and normals are also described as NURBS surfaces:

```
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, . . ., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, . . ., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, . . ., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

See Also

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **gluNurbsSurface**, **gluPwlCurve**

gluBeginTrim, gluEndTrim

The **gluBeginTrim** and **gluEndTrim** functions delimit a NURBS trimming loop definition.

void gluBeginTrim(
  GLUnurbsObj **nobj*
  );

void gluEndTrim(
  GLUnurbsObj **nobj*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

Remarks

Use the **gluBeginTrim** function to mark the beginning of a trimming loop, and the **gluEndTrim** function 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 **gluBeginSurface** and **gluEndSurface**.

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 **gluBeginTrim**/**gluEndTrim** pair.

The definition of a single closed trimming loop can consist of multiple curve segments, each described as a piecewise linear curve (see **gluPwlCurve**) or as a single NURBS curve (see **gluNurbsCurve**), 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 **gluBeginTrim** and **gluEndTrim**) are **gluPwlCurve** and **gluNurbsCurve**.

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 counterclockwise, 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 **gluNurbsCallback**).

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. Trimming curves cannot be self-intersecting, nor can they intersect one another (or an error results).

If no trimming information is given for a NURBS surface, the entire surface is drawn.

Example

This code fragment defines a trimming loop that consists of one piecewise linear curve, and two NURBS curves:

```
gluBeginTrim(nobj);
```

```
    gluPwlCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_2);
    gluNurbsCurve(. . ., GLU_MAP1_TRIM_3);
gluEndTrim(nobj);
```

See Also

gluBeginSurface, **gluNewNurbsRenderer**, **gluNurbsCallback**, **gluNurbsCurve**, **gluPwlCurve**

gluBuild1DMipmaps

The **gluBuild1DMipmaps** function creates 1-D mipmaps.

int gluBuild1DMipmaps(
GLenum *target*,
GLint *components*,
GLint *width*,
GLenum *format*,
GLenum *type*,
const void **data*
);

Parameters

target
Specifies the target texture. Must be **GL_TEXTURE_1D**.

components
Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.

width
Specifies the width of the texture image.

format
Specifies the format of the pixel data. Must be one of **GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE,** and **GL_LUMINANCE_ALPHA.**

type
Specifies the data type for *data*. Must be one of **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, or **GL_FLOAT**.

data
Specifies a pointer to the image data in memory.

Remarks

The **gluBuild1DMipmaps** function obtains the input image and generates all mipmap images (using **gluScaleImage**) so that the input image can be used as a mipmapped texture image. **glTexImage1D** is then called to load each of the images. If the width of the input image is not a power of two, then the image is scaled to the nearest power of two before the mipmaps are generated.

A return value of zero indicates success. Otherwise, a GLU error code is returned (see **gluErrorString**).

See **glTexImage1D** for a description of the acceptable values for the *format* parameter. See **glDrawPixels** for a description of the acceptable values for the *type* parameter.

See Also

glTexImage1D, **gluBuild2DMipmaps**, **gluErrorString**, **gluScaleImage**

gluBuild2DMipmaps

The **gluBuild2DMipmaps** function creates 2-D mipmaps.

int gluBuild2DMipmaps(
GLenum *target*,
GLint *components*,
GLint *width*,
GLint *height*,
GLenum *format*,
GLenum *type*,
const void **data*
);

Parameters

target
Specifies the target texture. Must be **GL_TEXTURE_2D**.

components
Specifies the number of color components in the texture. Must be 1, 2, 3, or 4.

width, height
Specifies the width and height, respectively, of the texture image.

format
Specifies the format of the pixel data. Must be one of: **GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_LUMINANCE,** and **GL_LUMINANCE_ALPHA.**

type
Specifies the data type for *data*. Must be one of: **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** or **GL_FLOAT.**

data
Specifies a pointer to the image data in memory.

Remarks

The **gluBuild2DMipmaps** function obtains the input image and generates all mipmap images (using **gluScaleImage**) so that the input image can be used as a mipmapped texture image. The **glTexImage2D** function is then called to load each of the images. If the dimensions of the input image are not powers of two, then the image is scaled so that both the width and height are powers of two before the mipmaps are generated.

A return value of 0 indicates success. Otherwise, a GLU error code is returned (see **gluErrorString**).

See **glTexImage1D** for a description of the acceptable values for the *format* parameter. See **glDrawPixels** for a description of the acceptable values for the *type* parameter.

See Also

glDrawPixels, **glTexImage1D**, **glTexImage2D**, **gluBuild1DMipmaps**, **gluErrorString**, **gluScaleImage**

gluCylinder

The **gluCylinder** function draws a cylinder.

void gluCylinder(
  GLUquadricObj **qobj*,
  GLdouble *baseRadius*,
  GLdouble *topRadius*,
  GLdouble *height*,
  GLint *slices*,
  GLint *stacks*
  );

Parameters

qobj
  Specifies the quadrics object (created with **gluNewQuadric**).

baseRadius
  Specifies the radius of the cylinder at *z* = 0.

topRadius
  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.

Remarks

The **gluCylinder** function 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.

Notice that if *topRadius* is set to zero, then this routine will generate a cone.

If the orientation is set to **GLU_OUTSIDE** (with **gluQuadricOrientation**), then any generated normals point away from the *z* axis. Otherwise, they point toward the *z* axis.

If texturing is turned on (with **gluQuadricTexture**), 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.

See Also

gluDisk, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricTexture**, **gluSphere**

gluDeleteNurbsRenderer

The **gluDeleteNurbsRenderer** function destroys a NURBS object.

void gluDeleteNurbsRenderer(
  GLUnurbsObj **nobj*
  );

Parameters

nobj
  Specifies the NURBS object to be destroyed (created with **gluNewNurbsRenderer**).

Remarks

The **gluDeleteNurbsRenderer** function destroys the NURBS object and frees any memory used by it. Once **gluDeleteNurbsRenderer** has been called, *nobj* cannot be used again.

See Also

gluNewNurbsRenderer

gluDeleteQuadric

The **gluDeleteQuadric** function destroys a quadrics object.

void gluDeleteQuadric(
  GLUquadricObj **state*
  );

Parameters

state
  Specifies the quadrics object to be destroyed (created with **gluNewQuadric**).

Remarks

The **gluDeleteQuadric** function destroys the quadrics object and frees any memory used by it. Once **gluDeleteQuadric** has been called, *state* cannot be used again.

See Also

gluNewQuadric

gluDeleteTess

The **gluDeleteTess** function destroys a tessellation object.

void gluDeleteTess(
  GLUtesselator **tess*
  );

Parameters

tess
  Specifies the tessellation object to destroy (created with **gluNewTess**).

Remarks

The **gluDeleteTess** function destroys the indicated tessellation object and frees any memory that it used.

See Also

gluTessBeginPolygon, **gluNewTess**, **gluTessCallback**

gluDisk

The **gluDisk** function draws a disk.

void gluDisk(
  GLUquadricObj **qobj*,
  GLdouble *innerRadius*,
  GLdouble *outerRadius*,
  GLint *slices*,
  GLint *loops*
  );

Parameters

qobj
  Specifies the quadrics object (created with **gluNewQuadric**).

innerRadius
  Specifies the inner radius of the disk (may be 0).

outerRadius
  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.

Remarks

The **gluDisk** function renders a disk on the *z* = 0 plane. The disk has a radius of *outerRadius*, and contains a concentric circular hole with a radius of *innerRadius*. If *innerRadius* 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 **gluQuadricOrientation**). 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 **gluQuadricTexture**), texture coordinates are generated linearly such that where r = *outerRadius* , 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).

See Also

gluCylinder, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

gluErrorString

The **gluErrorString** function produces an error string from an OpenGL or GLU error code. The error string is ANSI only.

const GLubyte* **gluErrorString**(
    GLenum *errorCode*
    );

Parameters

errorCode
    Specifies an OpenGL or GLU error code.

Remarks

The **gluErrorString** function produces an error string from an OpenGL or GLU error code. The string is in an ISO Latin 1 format. For example, **gluErrorString**(**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 **glGetError** for the list of OpenGL error codes.

The **gluErrorString** function produces error strings in ANSI only. Whenever possible you should use **gluErrorStringWIN**, which allows ANSI or Unicode error strings. This makes it easier to internationalize your program.

See Also

glGetError, **gluNurbsCallback**, **gluQuadricCallback**, **gluTessCallback**

gluGetNurbsProperty

The **gluGetNurbsProperty** function gets a NURBS property.

void gluGetNurbsProperty(
  GLUnurbsObj **nobj*,
  GLenum *property*,
  GLfloat **value*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

property
  Specifies the property whose value is to be fetched. Valid values are **GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX**, **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, and **GLU_V_STEP**.

value
  Specifies a pointer to the location into which the value of the named property is written.

Remarks

The **gluGetNurbsProperty** function is used to retrieve properties stored in a NURBS object. These properties affect the way that NURBS curves and surfaces are rendered. See **gluNurbsProperty** for information about what the properties are and what they do.

See Also

gluNewNurbsRenderer, **gluNurbsProperty**

gluGetString

The **gluGetString** function gets a string that describes the GLU version number or supported GLU extension calls.

const GLubyte ***gluGetString**(
    GLenum *name*
    );

Parameters

name
    Specifies either the version number of GLU, **GL_VERSION**, or available vendor-specific extension calls, **GL_EXTENSIONS**.

Remarks

The **gluGetString** function returns a pointer to a static, null-terminated string. When *name* is **GL_VERSION**, the returned string is a value that represents the version number of GLU. The format of the version number is as follows:

<version number><space><vendor-specific information>
(for example, "1.2.11 Microsoft Windows NT")

The version number has the form "major_number.minor_number" or "major_number.minor_number.release_number". The vendor-specific information is optional and the format and contents depend on the implementation.

When *name* is **GL_EXTENSIONS**, the returned string contains a list of names of supported GLU extensions that are separated by spaces. The format of the returned list of strings is as follows:

<extension_name><space><extension_ name><space> . . .
    (for example, "GLU_NURBS GL_TESSELATION")

The extension names cannot contain any spaces.

The **gluGetString** function is valid for GLU version 1.1 or later.

gluGetTessProperty

The **gluGetTessProperty** function gets a tessellation object property.

void gluTessProperty(
  GLUtesselator **tess*,
  GLenum *which*,
  GLdouble **value*
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

which
  Specifies the property whose value is to be retrieved. Valid values are **GLU_TESS_WINDING_RULE**, **GLU_TESS_BOUNDARY_ONLY**, and **GLU_TESS_TOLERANCE**.

value
  Specifies a pointer to the location of where the value of the named property is written.

Remarks

Use **gluGetTessProperty** to retrieve properties stored in a tessellation object. These properties affect the way that tessellation objects are interpreted and rendered. For information about what the properties are and what they do, see **gluTessProperty**.

See Also

gluNewTess

gluLoadSamplingMatrices

The **gluLoadSamplingMatices** function loads NURBS sampling and culling matrices.

void gluLoadSamplingMatrices(
  GLUnurbsObj **nobj*,
  const GLfloat *modelMatrix[16]*,
  const GLfloat *projMatrix[16]*,
  const GLint *viewport[4]*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

modelMatrix
  Specifies a modelview matrix (as from a **glGetFloatv** call).

projMatrix
  Specifies a projection matrix (as from a **glGetFloatv** call).

viewport
  Specifies a viewport (as from a **glGetIntegerv** call).

Remarks

The **gluLoadSamplingMatrices** function uses *modelMatrix*, *projMatrix*, and *viewport;* to recompute the sampling and culling matrices stored in *nobj*. 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).

The **gluLoadSamplingMatrices** function is necessary only if the **GLU_AUTO_LOAD_MATRIX** property is turned off (see **gluNurbsProperty**). 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 OpenGL server is needed to fetch the current values of the modelview matrix, projection matrix, and viewport.)

See Also

gluGetNurbsProperty, **gluNewNurbsRenderer**, **gluNurbsProperty**

gluLookAt

The **gluLookAt** function defines a viewing transformation.

void gluLookAt(
  GLdouble *eyex*,
  GLdouble *eyey*,
  GLdouble *eyez*,
  GLdouble *centerx*,
  GLdouble *centery*,
  GLdouble *centerz*,
  GLdouble *upx*,
  GLdouble *upy*,
  GLdouble *upz*
  );

Parameters

eyex, eyey, eyez
  Specifies the position of the eye point.

centerx, centery, centerz
  Specifies the position of the reference point.

upx, upy, upz
  Specifies the direction of the up vector.

Remarks

The **gluLookAt** function 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, so that, when a typical projection matrix is used, the center of the scene 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 to the reference point.

The matrix generated by **gluLookAt** postmultiplies the current matrix.

See Also

glFrustum, **gluPerspective**

gluNewNurbsRenderer

The **gluNewNurbsRenderer** function creates a NURBS object.

GLUnurbsObj* gluNewNurbsRenderer(
  void
  );

Remarks

The **gluNewNurbsRenderer** function creates and returns a pointer to a new NURBS object. This object must be referred to when calling NURBS rendering and control functions. A return value of zero means that there is not enough memory to allocate the object.

See Also

gluBeginCurve, **gluBeginSurface**, **gluBeginTrim**, **gluDeleteNurbsRenderer**, **gluNurbsCallback**, **gluNurbsProperty**

gluNewQuadric

The **gluNewQuadric** function creates a quadrics object.

GLUquadricObj* gluNewQuadric(
    void
    );

Remarks

The **gluNewQuadric** function creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of zero means that there is not enough memory to allocate the object.

See Also

gluCylinder, **gluDeleteQuadric**, **gluDisk**, **gluPartialDisk**, **gluQuadricCallback**, **gluQuadricDrawStyle**, **gluQuadricNormals**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

gluNewTess

The **gluNewTess** function creates a tessellation object.

GLUtesselator* gluNewTess(
    void
    );

Remarks

The **gluNewTess** function creates and returns a pointer to a new tessellation object. This object must be referred to when calling tessellation functions. A return value of zero means that there is not enough memory to allocate the object.

See Also

gluTessBeginPolygon, **gluDeleteTess**, **gluTessCallback**

gluNextContour

The **gluNextContour** function marks the beginning of another contour.

void gluNextContour(
GLUtesselator *tess,
GLenum *type*
);

Parameters

tess
Specifies the tessellation object (created with **gluNewTess**).

type
Specifies the type of the contour being defined. Valid values are **GLU_EXTERIOR**, **GLU_INTERIOR**, **GLU_UNKNOWN**, **GLU_CCW**, and **GLU_CW**.

Remarks

The **gluNextContour** function is used in describing polygons with multiple contours. After the first contour has been described through a series of **gluTessVertex** calls, a **gluNextContour** call indicates that the previous contour is complete and that the next contour is about to begin. Another series of **gluTessVertex** calls is then used to describe the new contour. This process can be repeated until all contours have been described.

The *type* parameter defines what type of contour follows. The legal contour types are as follows:

GLU_EXTERIOR
An exterior contour defines an exterior boundary of the polygon.

GLU_INTERIOR
An interior contour defines an interior boundary of the polygon (such as a hole).

GLU_UNKNOWN
An unknown contour is analyzed by the library to determine if it is interior or exterior.

GLU_CCW, GLU_CW
The first **GLU_CCW** or **GLU_CW** contour defined is considered to be exterior. All other contours are considered to be exterior if they are oriented in the same direction (clockwise or counterclockwise) as the first contour, and interior if they are not.

If one contour is of type **GLU_CCW** or **GLU_CW**, then all contours must be of the same type (if they are not, then all **GLU_CCW** and **GLU_CW** contours will be changed to **GLU_UNKNOWN**). Note that there is no real difference between the **GLU_CCW** and **GLU_CW** contour types**.**

The **gluNextContour** function can be called before the first contour is described to define the type of the first contour. If **gluNextContour** is not called before the first contour, then the first contour is marked **GLU_EXTERIOR**.

Note The **gluNextContour** function is obsolete and is provided for backwards compatibility only. The **gluNextContour** function is mapped to **gluTessEndContour** followed by **gluTessBeginContour**.

Example

A quadrilateral with a triangular hole in it can be described as follows:

```
gluBeginPolygon(tess);
    gluTessVertex(tess, v1, v1);
    gluTessVertex(tess, v2, v2);
    gluTessVertex(tess, v3, v3);
    gluTessVertex(tess, v4, v4);
gluNextContour(tess, GLU_INTERIOR);
```

```
    gluTessVertex(tess, v5, v5);
    gluTessVertex(tess, v6, v6);
    gluTessVertex(tess, v7, v7);
gluEndPolygon(tess);
```

See Also

gluTessBeginPolygon, gluNewTess, gluTessCallback, gluTessVertex, gluTessBeginContour

gluNurbsCallback

The **gluNurbsCallback** function defines a callback for a NURBS object.

void gluNurbsCallback(
GLUnurbsObj **nobj*,
GLenum *which*,
void (**fn*)( )
);

Parameters

nobj
Specifies the NURBS object (created with **gluNewNurbsRenderer**).

which
Specifies the callback being defined. The only valid value is **GLU_ERROR**.

fn
Specifies the function that the callback calls.

Remarks

The **gluNurbsCallback** function 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 *fn* is NULL, then any existing callback is erased.

The one legal callback is **GLU_ERROR**:

GLU_ERROR
The error function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. There are 37 errors unique to NURBS named **GLU_NURBS_ERROR1** through **GLU_NURBS_ERROR37.** Character strings describing these errors can be retrieved with **gluErrorString.**

See Also

gluErrorString, **gluNewNurbsRenderer**

gluNurbsCurve

The **gluNurbsCurve** function defines the shape of a NURBS curve.

void gluNurbsCurve(
  GLUnurbsObj **nobj*,
  GLint *nknots*,
  GLfloat **knot*,
  GLint *stride*,
  GLfloat **ctlarray*,
  GLint *order*,
  GLenum *type*
  );

Parameters

nobj
  Specifies the NURBS object (created with **gluNewNurbsRenderer**).

nknots
  Specifies the number of knots in *knot*. *nknots* equals the number of control points plus the order.

knot
  Specifies an array of *nknots* nondecreasing knot values.

stride
  Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.

ctlarray
  Specifies a pointer to an array of control points. The coordinates must agree with *type*, specified below.

order
  Specifies the order of the NURBS curve. *order* equals degree + 1, hence a cubic curve has an order of 4.

type
  Specifies the type of the curve. If this curve is defined within a **gluBeginCurve**/**gluEndCurve** pair, then the type can be any of the valid one-dimensional evaluator types (such as **GL_MAP1_VERTEX_3** or **GL_MAP1_COLOR_4**). Between a **gluBeginTrim**/**gluEndTrim** pair, the only valid types are **GLU_MAP1_TRIM_2** and **GLU_MAP1_TRIM_3**.

Remarks

Use the **gluNurbsCurve** function to describe a NURBS curve.

When **gluNurbsCurve** appears between a **gluBeginCurve**/**gluEndCurve** pair, it is used to describe a curve to be rendered. Positional, texture, and color coordinates are associated by presenting each as a separate **gluNurbsCurve** between a **gluBeginCurve**/**gluEndCurve** pair. No more than one call to **gluNurbsCurve** for each of color, position, and texture data can be made within a single **gluBeginCurve**/**gluEndCurve** pair. Exactly one call must be made to describe the position of the curve (a *type* of **GL_MAP1_VERTEX_3** or **GL_MAP1_VERTEX_4**).

When **gluNurbsCurve** appears between a **gluBeginTrim**/**gluEndTrim** 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 **gluBeginTrim** for more discussion about trimming curves.

Example

The following commands render a textured NURBS curve with normals:

```
gluBeginCurve(nobj);
    gluNurbsCurve(nobj, ..., GL_MAP1_TEXTURE_COORD_2);
    gluNurbsCurve(nobj, ..., GL_MAP1_NORMAL);
    gluNurbsCurve(nobj, ..., GL_MAP1_VERTEX_4);
gluEndCurve(nobj);
```

See Also

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluPwlCurve**

gluNurbsProperty

The **gluNurbsProperty** function sets a NURBS property.

void gluNurbsProperty(
GLUnurbsObj **nobj*,
GLenum *property*,
GLfloat *value*
);

Parameters

nobj
Specifies the NURBS object (created with **gluNewNurbsRenderer**).

property
Specifies the property to be set. Valid values are **GLU_SAMPLING_TOLERANCE, GLU_DISPLAY_MODE, GLU_CULLING, GLU_AUTO_LOAD_MATRIX**, **GLU_PARAMETRIC_TOLERANCE**, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, and **GLU_V_STEP**.

value
Specifies the value to which to set the indicated property. *value* can be a numeric value or one of three values **GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR,** or **GLU_DOMAIN_DISTANCE**.

Remarks

The **gluNurbsProperty** function is used to control properties stored in a NURBS object. These properties affect the way that a NURBS curve is rendered. The legal values for *property* are as follows:

GLU_SAMPLING_TOLERANCE
Specifies the maximum length, in pixels, to use when the sampling method is set to the **GLU_PATH_LENGTH** sampling method. The default value is 50.0 pixels.

GLU_DISPLAY_MODE
The *value* parameter defines how a NURBS surface should be rendered. *value* can be set to **GLU_FILL, GLU_OUTLINE_POLYGON,** or **GLU_OUTLINE_PATCH**. When set to **GLU_FILL**, the surface is rendered as a set of polygons. **GLU_OUTLINE_POLYGON** instructs the NURBS library to draw only the outlines of the polygons created by tessellation. **GLU_OUTLINE_PATCH** causes just the outlines of patches and trim curves defined by the user to be drawn. The default value is **GLU_FILL**.

GLU_CULLING
The *value* parameter is a Boolean value that, when set to **GL_TRUE**, indicates that a NURBS curve should be discarded prior to tessellation if its control points lie outside the current viewport. The default is **GL_FALSE** (because a NURBS curve cannot fall entirely within the convex hull of its control points).

GLU_PARAMETRIC_TOLERANCE
Specifies the maximum distance, in pixels, to use when the sampling method is set to **GLU_PARAMETRIC_ERROR**. The default value is 0.5.

GLU_SAMPLING_METHOD
Specifies how to tessallate a NURBS surface. **GLU_SAMPLING_METHOD** can have one of three values:

GLU_PATH_LENGTH
Specifies that surfaces rendered with the maximum length, in pixels, of the edges of the tessallation polygons are no greater than the value specified by **GLU_SAMPLING_TOLERANCE**.

GLU_PARAMETRIC_ERROR
Specifies that the surface is rendered with the value of **GLU_PARAMETRIC_TOLERANCE** specifying the maximum distance, in pixels, between the tessellation polygons and the surfaces

they approximate.

GLU_DOMAIN_DISTANCE
Specifies, in parametric coordinates, how many sample points per unit length to take in the u and v dimensions.

The default value is **GLU_PATH_LENGTH**.

GLU_U_STEP
Specifies the number of sample points per unit length taken along the u dimension in parametric coordinates. The value of **GLU_U_STEP** is used when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The default value is 100.

GLU_V_STEP
Specifies the number of sample points per unit length taken along the v dimension in parametric coordinates. The value of **GLU_V_STEP** is used when **GLU_SAMPLING_METHOD** is set to **GLU_DOMAIN_DISTANCE**. The default value is 100.

GLU_AUTO_LOAD_MATRIX
The *value* parameter is a Boolean value. When set to **GL_TRUE**, the NURBS code downloads the projection matrix, the modelview matrix, and the viewport from the OpenGL 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 of the viewport. If this mode is set to **GL_FALSE**, then the user 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 **gluLoadSamplingMatrices** function. The default for this mode is **GL_TRUE**. Changing this mode from **GL_TRUE** to **GL_FALSE** does not affect the sampling and culling matrices until **gluLoadSamplingMatrices** is called.

Note
Note The following *property* parameters are supported GLU version 1.1 or later and are not valid for GLU version 1.0:

GLU_PARAMETRIC_TOLERANCE, **GLU_SAMPLING_METHOD**, **GLU_U_STEP**, and **GLU_V_STEP**

The following *value* parameters are supported GLU version 1.1 or later and are not valid for GLU version 1.0:

GLU_PATH_LENGTH, **GLU_PARAMETRIC_ERROR**, **GLU_DOMAIN_DISTANCE**

See Also

gluGetNurbsProperty, **gluGetString**, **gluLoadSamplingMatrices**, **gluNewNurbsRenderer**

gluNurbsSurface

The **gluNurbsSurface** defines the shape of a NURBS surface.

void gluNurbsSurface(
GLUnurbsObj **nobj*,
GLint *uknot_count*,
GLfloat **uknot*,
GLint *vknot_count*,
GLfloat **vknot*,
GLint *u_stride*,
GLint *v_stride*,
GLfloat **ctlarray*,
GLint *uorder*,
GLint *vorder*,
GLenum *type*
);

Parameters

nobj
Specifies the NURBS object (created with **gluNewNurbsRenderer**).

uknot_count
Specifies the number of knots in the parametric *u* direction.

uknot
Specifies an array of *uknot_count* nondecreasing knot values in the parametric *u* direction.

vknot_count
Specifies the number of knots in the parametric *v* direction.

vknot
Specifies an array of *vknot_count* nondecreasing knot values in the parametric *v* direction.

u_stride
Specifies the offset (as a number of single-precision floating point values) between successive control points in the parametric *u* direction in *ctlarray*.

v_stride
Specifies the offset (in single-precision floating-point values) between successive control points in the parametric *v* direction in *ctlarray*.

ctlarray
Specifies an array containing control points for the NURBS surface. The offsets between successive control points in the parametric *u* and *v* directions are given by *u_stride* and *v_stride*.

uorder
Specifies the order of the NURBS surface in the parametric *u* direction. The order is one more than the degree, hence a surface that is cubic in *u* has a *u* order of 4.

vorder
Specifies the order of the NURBS surface in the parametric *v* direction. The order is one more than the degree, hence a surface that is cubic in *v* has a *v* order of 4.

type
Specifies type of the surface. *type* can be any of the valid two-dimensional evaluator types (such as **GL_MAP2_VERTEX_3** or **GL_MAP2_COLOR_4**).

Remarks

Use **gluNurbsSurface** 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 **gluBeginSurface** command. To mark the end of a NURBS surface definition, use

the **gluEndSurface** command. Call **gluNurbsSurface** within a NURBS surface definition only.

Positional, texture, and color coordinates are associated with a surface by presenting each as a separate **gluNurbsSurface** between a **gluBeginSurface**/**gluEndSurface** pair. No more than one call to **gluNurbsSurface** for each of color, position, and texture data can be made within a single **gluBeginSurface**/**gluEndSurface** pair. Exactly one call must be made to describe the position of the surface (a *type* of **GL_MAP2_VERTEX_3** or **GL_MAP2_VERTEX_4**).

A NURBS surface can be trimmed by using the commands **gluNurbsCurve** and **gluPwlCurve** between calls to **gluBeginTrim** and **gluEndTrim**.

Notice that a **gluNurbsSurface** with *uknot_count* knots in the u direction and *vknot_count* knots in the *v* direction with orders *uorder* and *vorder* must have (*uknot_count* - *uorder*) times (*vknot_count* - *vorder*) control points.

Example

The following commands render a textured NURBS surface with normals; the texture coordinates and normals are also NURBS surfaces:

```
gluBeginSurface(nobj);
    gluNurbsSurface(nobj, . . ., GL_MAP2_TEXTURE_COORD_2);
    gluNurbsSurface(nobj, . . ., GL_MAP2_NORMAL);
    gluNurbsSurface(nobj, . . ., GL_MAP2_VERTEX_4);
gluEndSurface(nobj);
```

See Also

gluBeginSurface, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**, **gluPwlCurve**

gluOrtho2D

The **gluOrtho2D** function defines a 2-D orthographic projection matrix.

void gluOrtho2D(
  GLdouble *left*,
  GLdouble *right*,
  GLdouble *bottom*,
  GLdouble *top*
  );

Parameters

left, right
  Specify the coordinates for the left and right vertical clipping planes.

bottom, top
  Specify the coordinates for the bottom and top horizontal clipping planes.

Remarks

The **gluOrtho2D** function sets up a two-dimensional orthographic viewing region. This is equivalent to calling **glOrtho** with near = -1 . and far = 1.

See Also

glOrtho, **gluPerspective**

gluPartialDisk

The **gluPartialDisk** function draws an arc of a disk.

void gluPartialDisk(
  GLUquadricObj **qobj*,
  GLdouble *innerRadius*,
  GLdouble *outerRadius*,
  GLint *slices*,
  GLint *loops*,
  GLdouble *startAngle*,
  GLdouble *sweepAngle*
  );

Parameters

qobj
  Specifies a quadrics object (created with **gluNewQuadric**).

innerRadius
  Specifies the inner radius of the partial disk (can be zero).

outerRadius
  Specifies the outer radius of the partial disk.

slices
  Specfies 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.

startAngle
  Specifies the starting angle, in degrees, of the disk portion.

sweepAngle
  Specifies the sweep angle, in degrees, of the disk portion.

Remarks

The **gluPartialDisk** function 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 *startAngle* through *startAngle* + *sweepAngle* is included (where 0 degrees is along the +*y* axis, 90 degrees along the +*x* axis, 180 along the - *y* axis, and 270 along the - *x* . axis).

The partial disk has a radius of *outerRadius*, and contains a concentric circular hole with a radius of *innerRadius*. If *innerRadius* is zero, 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 **gluQuadricOrientation**). 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 **gluQuadricTexture**), texture coordinates are generated linearly such that where *r* = *outerRadius*, 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).

See Also

gluCylinder, **gluDisk**, **gluNewQuadric**, **gluQuadricOrientation**, **gluQuadricTexture**, **gluSphere**

gluPerspective

The **gluPerspective** function sets up a perspective projection matrix.

void gluPerspective(
  GLdouble *fovy*,
  GLdouble *aspect*,
  GLdouble *zNear*,
  GLdouble *zFar*
  );

Parameters

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).

zNear
  Specifies the distance from the viewer to the near clipping plane (always positive).

zFar
  Specifies the distance from the viewer to the far clipping plane (always positive).

Remarks

The **gluPerspective** function specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in **gluPerspective** should match the aspect ratio of the associated viewport. For example, *aspect* = 2.0 means the viewers 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 **gluPerspective** is multipled by the current matrix, just as if **glMultMatrix** were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to **gluPerspective** with a call to **glLoadIdentity**.

See Also

glFrustum, **glLoadIdentity**, **glMultMatrix**, **gluOrtho2D**

gluPickMatrix

The **gluPickMatrix** function defines a picking region.

void gluPickMatrix(**GLdouble** *x*,
GLdouble *y*,
GLdouble *width*,
GLdouble *height*,
GLint *viewport[4]*
);

Parameters

x, y
Specify the center of a picking region in window coordinates.

width, height
Specify the width and height, respectively, of the picking region in window coordinates.

viewport
Specifies the current viewport (as from a **glGetIntegerv** call).

Remarks

The **gluPickMatrix** function 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 **gluPickMatrix** to restrict drawing to a small region around the cursor. Then, enter selection mode (with **glRenderMode** 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 **gluPickMatrix** is multiplied by the current matrix just as if **glMultMatrix** is called with the generated matrix. To effectively use the generated pick matrix for picking, first call **glLoadIdentity** to load an identity matrix onto the perspective matrix stack. Then call **gluPickMatrix**, and finally, call a command (such as **gluPerspective**) to multiply the perspective matrix by the pick matrix.

When using **gluPickMatrix** 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.

Example

When rendering a scene as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
gluPerspective(. . .);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

a portion of the viewport can be selected as a pick region like this:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
gluPickMatrix(x, y, width, height, viewport);
gluPerspective(. . .);
glMatrixMode(GL_MODELVIEW);
/* Draw the scene */
```

See Also

glGet, glLoadIdentity, glMultMatrix, glRenderMode, gluPerspective

gluProject

The **gluProject** function maps object coordinates to window coordinates.

int gluProject(
  GLdouble *objx*,
  GLdouble *objy*,
  GLdouble *objz*,
  const GLdouble *modelMatrix[16]*,
  const GLdouble *projMatrix[16]*,
  const GLint *viewport[4]*,
  GLdouble **winx*,
  GLdouble **winy*,
  GLdouble **winz*
  );

Parameters

objx, objy, objz
  Specify the object coordinates.

modelMatrix
  Specifies the current modelview matrix (as from a **glGetDoublev** call).

projMatrix
  Specifies the current projection matrix (as from a **glGetDoublev** call).

viewport
  Specifies the current viewport (as from a **glGetIntegerv** call).

winx, winy, winz
  Return the computed window coordinates.

Remarks

The **gluProject** function transforms the specified object coordinates into window coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *winx*, *winy*, and *winz*. A return value of **GL_TRUE** indicates success, and **GL_FALSE** indicates failure.

See Also

glGet, **gluUnProject**

gluPwlCurve

The **gluPwlCurve** function describes a piecewise linear NURBS trimming curve.

void gluPwlCurve(
GLUnurbsObj *nobj,
GLint *count*,
GLfloat *array*,
GLint *stride*,
GLenum *type*
);

Parameters

nobj
Specifies the NURBS object (created with **gluNewNurbsRenderer**).

count
Specifies the number of points on the curve.

array
Specifies an array containing the curve points.

stride
Specifies the offset (a number of single-precision floating-point values) between points on the curve.

type
Specifies the type of curve. Must be either **GLU_MAP1_TRIM_2** or **GLU_MAP1_TRIM_3**.

Remarks

The **gluPwlCurve** function 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 real curve, the points should be close enough 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 **gluBeginTrim** for more information about trimming curves.

See Also

gluBeginCurve, **gluBeginTrim**, **gluNewNurbsRenderer**, **gluNurbsCurve**

gluQuadricCallback

The **gluQuadricCallback** function defines a callback for a quadrics object.

void gluQuadricCallback(
  GLUquadricObj **qobj*,
  GLenum *which*,
  void *(*fn)(*)
  );

Parameters

qobj
  Specifies the quadrics object (created with **gluNewQuadric**).

which
  Specifies the callback being defined. The only valid value is **GLU_ERROR**.

fn
  Specifies the function to be called.

Remarks

The **gluQuadricCallback** function is used to define a new callback to be used by a quadrics object. If the specified callback is already defined, then it is replaced. If *fn* is NULL, then any existing callback is erased.

The one legal callback is **GLU_ERROR**:

GLU_ERROR
  The function is called when an error is encountered. Its single argument is of type GLenum, and it indicates the specific error that occurred. Character strings describing these errors can be retrieved with the **gluErrorString call.**

See Also

gluErrorString, **gluNewQuadric**

gluQuadricDrawStyle

The **gluQuadricDrawStyle** function specifies the draw style desired for quadrics.

void gluQuadricDrawStyle(
GLUquadricObj **qobj*,
GLenum *drawStyle*
);

Parameters

qobj
Specifies the quadrics object (created with **gluNewQuadric**).

drawStyle
Specifies the desired draw style. Valid values are **GLU_FILL**, **GLU_LINE**, **GLU_SILHOUETTE**, and **GLU_POINT**.

Remarks

The **gluQuadricDrawStyle** function specifies the draw style for quadrics rendered with *qobj.* The legal values are as follows:

GLU_FILL
Quadrics are rendered with polygon primitives. The polygons are drawn in a counterclockwise fashion with respect to their normals (as defined with **gluQuadricOrientation**).

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.

See Also

gluNewQuadric, **gluQuadricNormals**, **gluQuadricOrientation**, **gluQuadricTexture**

gluQuadricNormals

The **gluQuadricNormals** function specifies what kind of normals are desired for quadrics.

void gluQuadricNormals(**GLUquadricObj** **qobj*,
GLenum *normals*
);

Parameters

qobj
Specifes the quadrics object (created with **gluNewQuadric**).

normals
Specifies the desired type of normals. Valid values are **GLU_NONE**, **GLU_FLAT**, and **GLU_SMOOTH**.

Remarks

The **gluQuadricNormals** function specifies what kind of normals are desired for quadrics rendered with *qobj*. 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 default.

See Also

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricOrientation**, **gluQuadricTexture**

gluQuadricOrientation

The **gluQuadricOrientation** function specifies inside/outside orientation for quadrics.

void gluQuadricOrientation(
  GLUquadricObj **qobj*,
  GLenum *orientation*
  );

Parameters

qobj
  Specifies the quadrics object (created with **gluNewQuadric**).

orientation
  Specifies the desired orientation. Valid values are **GLU_OUTSIDE** and **GLU_INSIDE**.

Remarks

The **gluQuadricOrientation** function specifies what kind of orientation is desired for quadrics rendered with *qobj*. The *orientation* values are as follows:

GLU_OUTSIDE
  Quadrics are drawn with normals pointing outward.

GLU_INSIDE
  Normals point inward. The default is **GLU_OUTSIDE**.

Notice that the interpretation of *outward* and *inward* depends on the quadric being drawn.

See Also

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricNormals**, **gluQuadricTexture**

gluQuadricTexture

The **gluQuadricTexture** function specifies if texturing is desired for quadrics.

void gluQuadricTexture(**GLUquadricObj** **qobj*,
GLboolean *textureCoords*
);

Parameters

qobj
Specifies the quadrics object (created with **gluNewQuadric**).

textureCoords
Specifies a flag indicating if texture coordinates should be generated.

Remarks

The **gluQuadricTexture** function specifies if texture coordinates should be generated for quadrics rendered with *qobj*. If the value of *textureCoords* is **GL_TRUE**, then texture coordinates are generated, and if *textureCoords* is **GL_FALSE**, they are not. The default is **GL_FALSE**.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

See Also

gluNewQuadric, **gluQuadricDrawStyle**, **gluQuadricNormals**,

gluScaleImage

The **gluScaleImage** function scales an image to an arbitrary size.

int gluScaleImage(
  GLenum *format*,
  GLint *widthin*,
  GLint *heightin*,
  GLenum *typein*,
  const void **datain*,
  GLint *widthout*,
  GLint *heightout*,
  GLenum *typeout*,
  void **dataout*
  );

Parameters

format
  Specifies the format of the pixel data. The following symbolic values are valid: **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**.

widthin, heightin
  Specify the width and height, respectively, of the source image that is scaled.

typein
  Specifies the data type for *datain*. Must be one of **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** or **GL_FLOAT**.

datain
  Specifies a pointer to the source image.

widthout, heightout
  Specify the width and height, respectively, of the destination image.

typeout
  Specifies the data type for *dataout*. Must be one of **GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT,** or **GL_FLOAT**.

dataout
  Specifies a pointer to the destination image.

Remarks

The **gluScaleImage** function 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, **gluScaleImage** uses a box filter to sample the source image and create pixels for the destination image. When magnifying an image, the pixels from the source image are linearly interpolated to create the destination image.

A return value of zero indicates success, otherwise a GLU error code is returned indicating what the problem was (see **gluErrorString**).

See **glReadPixels** for a description of the acceptable values for the *format*, *typein*, and *typeout* parameters.

See Also

glDrawPixels, **glReadPixels**, **gluBuild1DMipmaps**, **gluBuild2DMipmaps**, **gluErrorString**

gluSphere

The **gluSphere** function draws a sphere.

void gluSphere(
  GLUquadricObj **qobj*,
  GLdouble *radius*,
  GLint *slices*,
  GLint *stacks*
  );

Parameters

qobj
  Specifies the quadrics object (created with **gluNewQuadric**).

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).

Remarks

The **gluSphere** function 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 **gluQuadricOrientation**), 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 **gluQuadricTexture**), 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.

See Also

gluCylinder, **gluDisk**, **gluNewQuadric**, **gluPartialDisk**, **gluQuadricOrientation**, **gluQuadricTexture**

gluTessBeginContour, gluTessEndContour

The **gluTessBeginContour** and **gluTessEndContour** functions delimit a contour description.

void gluTessBeginContour(
  GLUtesselator **tess*
  );

void gluTessEndContour(
  GLUtesselator **tess*
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

Remarks

The **gluTessBeginContour** and **gluTessEndContour** functions delimit the definition of a polygon contour. Within each **gluTessBeginContour/gluTessEndContour** pair, there can be zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first). You can call **gluTessBeginContour** between **gluTessBeginPolygon** and **gluTessEndPolygon** only.

See Also

gluNewTess, **gluTessBeginPolygon**, **gluTessVertex**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**, **gluTessEndPolygon**

gluTessBeginPolygon, gluTessEndPolygon

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit a polygon description.

void gluTessBeginPolygon(
  GLUtesselator **tess*,
  void **polygon_data*
  );

void gluTessEndPolygon(
  GLUtesselator **tess*
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

polygon_data
  Specifies a pointer to user-defined polygon data.

Remarks

The **gluTessBeginPolygon** and **gluTessEndPolygon** functions delimit the definition of a non-convex polygon. Within each **gluTessBeginPolygon/gluTessEndPolygon** pair, you must include one or more calls to **gluTessBeginContour**/**gluTessEndContour**. Within each contour, there are zero or more calls to **gluTessVertex**. The vertexes specify a closed contour (the last vertex of each contour is automatically linked to the first).

The *polygon_data* parameter is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see **gluTessCallback**), this pointer is returned to the callback function(s), making it a convenient way to store per-polygon information.

When you call **gluTessEndPolygon**, the polygon is tessellated, and the resulting triangles are described through callbacks. For descriptions of the callback functions, see **gluTessCallback**.

Example

A quadrilateral with a triangular hole in it can be described as follows:

```
gluTessBeginPolygon(tobj, NULL);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v1, v1);
    gluTessVertex(tobj, v2, v2);
    gluTessVertex(tobj, v3, v3);
    gluTessVertex(tobj, v4, v4);
  gluTessEndContour(tobj);
  gluTessBeginContour(tobj);
    gluTessVertex(tobj, v5, v5);
    gluTessVertex(tobj, v6, v6);
    gluTessVertex(tobj, v7, v7);
  gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
```

See Also

gluNewTess, **gluTessBeginContour**, **gluTessVertex**, **gluTessCallback**, **gluTessProperty**, **gluTessNormal**

gluTessCallback

The **gluTessCallback** function defines a callback for a tessellation object.

void gluTessCallback(
  GLUtesselator **tess*,
  GLenum *which*,
  void (**fn*)( )
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

which
  Specifies the callback being defined. The following values are valid: **GLU_TESS_BEGIN**, **GLU_TESS_BEGIN_DATA**, **GLU_TESS_EDGE_FLAG**, **GLU_TESS_EDGE_FLAG_DATA**, **GLU_TESS_VERTEX**, **GLU_TESS_VERTEX_DATA**, **GLU_TESS_END**, **GLU_TESS_END_DATA**, **GLU_TESS_COMBINE**, **GLU_TESS_COMBINE_DATA**, **GLU_TESS_ERROR**, and **GLU_TESS_ERROR_DATA**.

fn
  Specifies the function to be called.

Remarks

Use **gluTessCallback** to specify a callback to be used by a tessellation object. If the specified callback is already defined, then it is replaced. If *fn* is NULL, then the existing callback becomes undefined.

The tessallation object uses these callbacks to describe how a polygon you specify is broken into triangles. Note that there are two versions of each callback: one with polygon data that you can define and one without. If both versions of a particular callback are specified then the callback with the polygon data you specify will be used. The *polygon_data* parameter of **gluTessBeginPolygon** is a copy of the pointer that was specified when **gluTessBeginPolygon** was called.

The legal callbacks are as follows:

GLU_TESS_BEGIN
  The **GLU_TESS_BEGIN** callback is invoked like **glBegin** to indicate the start of a (triangle) primitive. The function takes a single argument of type GLenum. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_FALSE** then the argument is set to either **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, or **GL_TRIANGLES**. If the **GLU_TESS_BOUNDARY_ONLY** property is set to **GL_TRUE** then the argument will be set to **GL_LINE_LOOP**. The function prototype for this callback is as follows:

```
void begin ( GLenum type );
```

GLU_TESS_BEGIN_DATA
  The **GLU_TESS_BEGIN_DATA** is same as the **GLU_TESS_BEGIN** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is as follows:

```
void beginData ( GLenum type, void *polygon_data );
```

GLU_TESS_EDGE_FLAG
  The **GLU_TESS_EDGE_FLAG** callback is similar to **glEdgeFlag**. The function takes a single Boolean flag that indicates which edges lie on the polygon boundary. If the flag is **GL_TRUE**, then each vertex that follows begins an edge that lies on the polygon boundary; that is, an edge which separates an interior region from an exterior one. If the flag is **GL_FALSE**, then each vertex that follows begins an edge that lies in the polygon interior. The **GLU_TESS_EDGE_FLAG** callback (if

defined) is invoked before the first vertex callback is made.

Because triangle fans and triangle strips do not support edge flags, the begin callback is not called with **GL_TRIANGLE_FAN** or **GL_TRIANGLE_STRIP** if an edge flag callback is provided. Instead, the fans and strips are converted to independent triangles. The function prototype for this callback is as follows:

```
void edgeFlag ( GLboolean flag );
```

GLU_TESS_EDGE_FLAG_DATA

The **GLU_TESS_EDGE_FLAG_DATA** callback is the same as the **GLU_TESS_EDGE_FLAG** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is as follows:

```
void edgeFlagData ( GLboolean flag, void *polygon_data );
```

GLU_TESS_VERTEX

The **GLU_TESS_VERTEX** callback is invoked between the begin and end callbacks. It is similar to **glVertex**, and it defines the vertexes of the triangles created by the tessellation process. The function takes a pointer as its only argument. This pointer is identical to the opaque pointer that you provided when you defined the vertex (see **gluTessVertex**). The function prototype for this callback is as follows:

```
void vertex ( void *vertex_data );
```

GLU_TESS_VERTEX_DATA

The **GLU_TESS_VERTEX_DATA** is the same as the **GLU_TESS_VERTEX** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is as follows:

```
void vertexData ( void *vertex_data, void *polygon_data );
```

GLU_TESS_END

The **GLU_TESS_END** callback serves the same purpose as **glEnd**. It indicates the end of a primitive and it takes no arguments. The function prototype for this callback is as follows:

```
void end ( void );
```

GLU_TESS_END_DATA

The **GLU_TESS_END_DATA** callback is the same as the **GLU_TESS_END** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**.The function prototype for this callback is as follows:

```
void endData ( void *polygon_data);
```

GLU_TESS_COMBINE

The **GLU_TESS_COMBINE** callback is called to create a new vertex when the tessellation detects an intersection, or to merge features. The function takes four arguments: an array of three elements each of type GLdouble, an array of four pointers, an array of four elements each of type GLfloat, and a pointer to a pointer. The prototype is as follows:

```
void combine( GLdouble coords[3], void *vertex_data[4],
                      GLfloat weight[4], void **outData );
```

The vertex is defined as a linear combination of up to 4 existing vertexes, stored in *vertex_data*. The coefficients of the linear combination are given by *weight*; these weights always sum to 1.0. All vertex pointers are valid even when some of the weights are zero. *coords* gives the location of the new vertex.

You must allocate another vertex, interpolate parameters using *vertex_data* and weight, and return the new vertex pointer in *outData*. This handle is supplied during rendering callbacks. You are

responsible for freeing the memory sometime after calling **gluTessEndPolygon**.

For example, if the polygon lies in an arbitrary plane in 3-space, and you associate a color with each vertex, the **GLU_TESS_COMBINE** callback might look like the following:

```
void myCombine( GLdouble coords[3], VERTEX *d[4],
                        GLfloat w[4], VERTEX **dataOut )
{
   VERTEX *new = new_vertex();
   new->x = coords[0];
   new->y = coords[1];
   new->z = coords[2];
   new->r = w[0]*d[0]->r + w[1]*d[1]->r + w[2]*d[2]->r +
                    w[3]*d[3]->r;
   new->g = w[0]*d[0]->g + w[1]*d[1]->g + w[2]*d[2]->g +
                    w[3]*d[3]->g;
   new->b = w[0]*d[0]->b + w[1]*d[1]->b + w[2]*d[2]->b +
                    w[3]*d[3]->b;
   new->a = w[0]*d[0]->a + w[1]*d[1]->a + w[2]*d[2]->a +
                    w[3]*d[3]->a;
         *dataOut = new;
}
```

When the tessellation detects an intersection, the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback (see below) must be defined, and it must write a non-NULL pointer into *dataOut*. Otherwise the **GLU_TESS_NEED_COMBINE_CALLBACK** error occurs, and no output is generated. (This is the only error that can occur during tessellation and rendering.)

GLU_TESS_COMBINE_DATA

The **GLU_TESS_COMBINE_DATA** callback is the same as the **GLU_TESS_COMBINE** callback except that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is as follows:

```
void combineData ( GLdouble coords[3], void *vertex_data[4],
                                GLfloat weight[4], void **outData,
                                void *polygon_data );
```

GLU_TESS_ERROR

The **GLU_TESS_ERROR** callback is called when an error is encountered. The one argument is of type GLenum; it indicates the specific error that occurred and will be set to one of **GLU_TESS_MISSING_BEGIN_POLYGON, GLU_TESS_MISSING_END_POLYGON, GLU_TESS_MISSING_BEGIN_CONTOUR, GLU_TESS_MISSING_END_CONTOUR, GLU_TESS_COORD_TOO_LARGE, GLU_TESS_NEED_COMBINE_CALLBACK**. You can call **gluErrorString** to retrieve character strings describing these errors.The function prototype for this callback is as follows:

```
void error ( GLenum errno );
```

The GLU library recovers from the first four errors by inserting the missing call(s). **GLU_TESS_COORD_TOO_LARGE** indicates that some vertex coordinate exceeded the predefined constant **GLU_TESS_MAX_COORD** in absolute value, and that the value has been clamped. (Coordinate values must be small enough so that two can be multiplied together without overflow.) **GLU_TESS_NEED_COMBINE_CALLBACK** indicates that the tessellation detected an intersection between two edges in the input data, and the **GLU_TESS_COMBINE** or **GLU_TESS_COMBINE_DATA** callback was not provided. No output will be generated.

GLU_TESS_ERROR_DATA

The **GLU_TESS_ERROR_DATA** callback is the same as the **GLU_TESS_ERROR** callback except

that it takes an additional pointer argument. This pointer is identical to the opaque pointer provided when you call **gluTessBeginPolygon**. The function prototype for this callback is as follows:

```
void errorData ( GLenum errno, void *polygon_data );
```

Example

You can render tessallated polygons directly as follows:

```
gluTessCallback(tess, GLU_TESS_BEGIN, glBegin);
gluTessCallback(tess, GLU_TESS_VERTEX, glVertex3dv);
gluTessCallback(tess, GLU_TESS_END, glEnd);
gluTessBeginPolygon(tess, NULL);
   gluTessBeginContour(tess);
   gluTessVertex(tess, v, v);
   . . .
   gluTessEndContour(tess);
gluTessEndPolygon(tess);
```

Typically, you should store the tessellated polygon in a display list so that it does not need to be tessellated every time it is rendered.

See Also

glBegin, **glEdgeFlag**, **glVertex**, **gluDeleteTess**, **gluErrorString**, **gluNewTess**, **gluTessVertex**

gluTessNormal

The **gluTessNormal** function specifies a normal for a polygon.

void gluTessNormal(
  GLUtesselator **tess*,
  GLdouble *x*,
  GLdouble *y*,
  GLdouble *z*
);

Parameters

tess
Specifies the tessellation object (created with **gluNewTess**).

x
Specifies the x-coordinate component of a normal.

y
Specifies the y-coordinate component of a normal.

z
Specifies the z-coordinate component of a normal.

Remarks

The **gluTessNormal** function describes a normal for a polygon that you define. All input data is projected onto a plane perpendicular to one of the three coordinate axes before tessellation, and all output triangles are oriented counterclockwise with respect to the normal. (You can obtain clockwise orientation by reversing the sign of the supplied normal). For example, if you know that all polygons lie in the x-y plane, call **gluTessNormal**(tess, 0.0, 0.0, 1.0) before rendering any polygons.

If the supplied normal is (0,0,0) (the default value), the normal is determined as follows. The direction of the normal, up to its sign, is found by fitting a plane to the vertexes, without regard to how the vertexes are connected. It is expected that the input data lies approximately in the plane; otherwise projection perpendicular to one of the three coordinate axes can change the geometry substantially. The sign of the normal is chosen so that the sum of the signed areas of all input contours is non-negative (where a counter-clock-wise contour has positive area).

The supplied normal persists until it is changed by another call to **gluTessNormal**.

See Also

gluTessBeginPolygon, **gluTessEndPolygon**

gluTessProperty

The **gluTessProperty** function sets the property of a tessellation object.

void gluTessProperty(
  GLUtesselator **tess*,
  GLenum *which*,
  GLdouble *value*
);

Parameters

tess
Specifies the tessellation object (created with **gluNewTess**).

which
Specifies the property value to set. Valid values are **GLU_TESS_WINDING_RULE**, **GLU_TESS_BOUNDARY_ONLY**, and **GLU_TESS_TOLERANCE**.

value
Specifies the value of the indicated property.

Remarks

The **gluTessProperty** function controls properties stored in a tessellation object. These properties affect the way that the polygons are interpreted and rendered. The legal values are as follows:

GLU_TESS_WINDING_RULE
Determines which parts of the polygon are on the "interior." The *value* parameter may be set to one of **GLU_TESS_WINDING_ODD**, **GLU_TESS_WINDING_NONZERO**, **GLU_TESS_WINDING_POSITIVE**, or **GLU_TESS_WINDING_NEGATIVE**, or **GLU_TESS_WINDING_ABS_GEQ_TWO**.

To understand how the winding rule works, first consider that the input contours partition the plane into regions. The winding rule determines which of these regions are inside the polygon.

For a single contour C, the winding number of a point *x* is simply the signed number of revolutions we make around *x* as we travel once around C (where counterclockwise is positive). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point *x* in the plane. Note that the winding number is the same for all points in a single region.

The winding rule classifies a region as "inside" if its winding number belongs to the chosen category (odd, non-zero, positive, negative, or absolute value of at least two). The previous GLU tessellator (prior to GLU 1.2) used the "odd" rule. The "non-zero" rule is another common way to define the interior. The other three rules are useful for polygon CSG operations.

GLU_TESS_BOUNDARY_ONLY
Specifies a Boolean value (*value* should be set to **GL_TRUE** or **GL_FALSE**). When set to **GL_TRUE**, a set of closed contours separating the polygon interior and exterior are returned instead of a tessellation. Exterior contours are oriented counterclockwise with respect to the normal, interior contours are oriented clockwise. The **GLU_TESS_BEGIN** and **GLU_TESS_BEGIN_DATA** callbacks use the type **GL_LINE_LOOP** for each contour.

GLU_TESS_TOLERANCE
Specifies a tolerance for merging features to reduce the size of the output. For example, two vertexes that are very close to each other might be replaced by a single vertex. The tolerance is multiplied by the largest coordinate magnitude of any input vertex; this specifies the maximum distance that any feature can move as the result of a single merge operation. If a single feature takes part in several merge operations, the total distance moved can be larger.

Feature merging is completely optional; the tolerance is only a hint. The implementation is free to merge in some cases and not in others, or to never merge features at all. The default tolerance is

zero.

The current implementation merges vertexes only if they are exactly coincident, regardless of the current tolerance. A vertex is spliced into an edge only if the implementation is unable to distinguish which side of the edge the vertex lies on. Two edges are merged only when both endpoints are identical.

See Also

gluGetTessProperty

gluTessVertex

The **gluTessVertex** function specifies a vertex on a polygon.

void gluTessVertex(
  GLUtesselator **tess*,
  GLdouble *v[3]*,
  void **data*
  );

Parameters

tess
  Specifies the tessellation object (created with **gluNewTess**).

v
  Specifies the location of the vertex.

data
  Specifies an opaque pointer passed back to the user with the vertex callback (as specified by **gluTessCallback**).

Remarks

The **gluTessVertex** function describes a vertex on a polygon that the user is defining. Successive **gluTessVertex** calls describe a closed contour. For example, if the user wants to describe a quadrilateral, then **gluTessVertex** should be called four times. The **gluTessVertex** function can only be called between **gluTessBeginContour** and **gluTessEndContour**.

The *data* parameter normally points to a structure containing the vertex location, as well as other per-vertex attributes such as color and normal. This pointer is passed back to the user through the **GLU_VERTEX** callback after tessellation (see **gluTessCallback**).

Example

A quadrilateral with a triangular hole in it can be described as follows:

```
gluTessBeginPolygon(tess, NULL);
  gluTessBeginContour(tess);
   gluTessVertex(tess, v1, v1);
   gluTessVertex(tess, v2, v2);
   gluTessVertex(tess, v3, v3);
   gluTessVertex(tess, v4, v4);
  gluTessEndContour(tess);
gluNextContour(tess, GLU_INTERIOR);
  gluTessBeginContour(tess);
   gluTessVertex(tess, v5, v5);
   gluTessVertex(tess, v6, v6);
   gluTessVertex(tess, v7, v7);
  gluTessEndContour(tess);
gluTessEndPolygon(tess);
```

See Also

gluTessBeginPolygon, **gluNewTess**, **gluTessCallback**, **gluTessBeginContour**, **gluTessProperty**, **gluTessNormal**

gluUnProject

The **gluUnProject** function maps window coordinates to object coordinates.

int gluUnProject(
GLdouble *winx*,
GLdouble *winy*,
GLdouble *winz*,
const GLdouble *modelMatrix[16]*,
const GLdouble *projMatrix[16]*,
const GLint *viewport[4]*,
GLdouble **objx*,
GLdouble **objy*,
GLdouble **objz*
);

Parameters

winx, winy, winz
Specify the window coordinates to be mapped.

modelMatrix
Specifies the modelview matrix (as from a **glGetDoublev** call).

projMatrix
Specifies the projection matrix (as from a **glGetDoublev** call).

viewport
Specifies the viewport (as from a **glGetIntegerv** call).

objx, objy, objz
Returns the computed object coordinates.

Remarks

The **gluUnProject** function maps the specified window coordinates into object coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *objx*, *objy*, and *objz*. A return value of **GL_TRUE** indicates success, and **GL_FALSE** indicates failure.

See Also

glGet, **gluProject**

The OpenGL Utility Library

OpenGL provides a small but powerful set of drawing operations, and all higher-level drawing requires their use. To help simplify some of your programming tasks, the OpenGL Utility Library (GLU) includes several routines that encapsulate OpenGL commands. Consult the *OpenGL Reference Manual* for more detailed descriptions of these routines. This section groups them functionally as follows:

- Initialization
- Manipulating Images for Use in Texturing
- Transforming Coordinates
- Tessellation
- Rendering Spheres, Cylinders, and Disks
- NURBS Curves and Surfaces
- Describing Errors

Initialization

With GLU version 1.1 or later you can use a routine that returns the version number of GLU library or that returns the version number and any vendor-specific GLU extension calls (**gluGetString**).

const GLubyte ***gluGetString**(GLenum *name*);

Manipulating Images for Use in Texturing

As you set up texture mapping in your application, you'll probably want to take advantage of mipmapping, which requires a series of reduced images (or texture maps). To support mipmapping, the GLU includes a general routine that scales images (**gluScaleImage**) and routines that generate a complete set of mipmaps given an original image in one or two dimensions (**gluBuild1DMipmaps** and **gluBuild2DMipmaps**).

GLint **gluScaleImage(**GLenum *format***,** GLint *widthin***,** GLint *heightin***,**
GLenum *typein***,** const void *******datain***,**
GLint *widthout***,** GLint *heightout***,**
GLenum *typeout***,** void *******dataout***);**

GLint **gluBuild1DMipmaps(**GLenum *target***,** GLint *components***,**
GLint *width***,** GLenum *format***,** GLenum *type***,**
void *******data***);**

GLint **gluBuild2DMipmaps(**GLenum *target***,** GLint *components***,**
GLint *width***,** GLint *height***,** GLenum *format***,**
GLenum *type***,** void *******data***);**

Transforming Coordinates

The GLU includes routines that create matrices for standard perspective and orthographic viewing (**gluPerspective** and **gluOrtho2D**). In addition, a viewing routine allows you to position your eye at any point in space and look at any other point (**gluLookAt**). In addition, the GLU includes a routine to help you create a picking matrix (**gluPickMatrix**). The prototypes for these four routines are listed here.

void **gluPerspective(**GLdouble *fovy***,** GLdouble *aspect***,** GLdouble *zNear***,**
GLdouble *zFar***);**

void **gluOrtho2D(**GLdouble *left***,** GLdouble *right***,** GLdouble *bottom***,**
GLdouble *top***);**

void **gluLookAt(**GLdouble *eyex***,** GLdouble *eyey***,** GLdouble *eyez***,**
GLdouble *centerx***,** GLdouble *centery***,**
GLdouble *centerz***,** GLdouble *upx***,** GLdouble *upy***,**
GLdouble *upz*);

void **gluPickMatrix(**GLdouble *x***,** GLdouble *y***,** GLdouble *width***,**
GLdouble *height***,** GLint *viewport*[4]);

In addition, GLU provides two routines that convert between object coordinates and screen coordinates, **gluProject** and **gluUnProject**.

GLint **gluProject(**GLdouble *objx***,** GLdouble *objy***,** GLdouble *objz***,**
const GLdouble *modelMatrix*[16]**,**
const GLdouble *projMatrix*[16]**,**
const GLint *viewport*[4]**,** GLdouble ***winx***,**
GLdouble ****winy***, GLdouble ****winz***);**

Transforms the specified object coordinates *objx*, *objy*, and *objz* into window coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *winx*, *winy*, and *winz*. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

GLint **gluUnProject(**GLdouble *winx***,** GLdouble *winy***,** GLdouble *winz***,**
const GLdouble *modelMatrix*[16]**,**
const GLdouble *projMatrix*[16]**,**
const GLint *viewport*[4]**,** GLdouble ***objx***,**
GLdouble ****objy***, GLdouble ****objz***);**

Transforms the specified window coordinates *winx*, *winy*, and *winz* into object coordinates using *modelMatrix*, *projMatrix*, and *viewport*. The result is stored in *objx*, *objy*, and *objz*. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

Tessellation

OpenGL can directly display only simple convex polygons. A polygon is simple if the edges intersect only at vertices, there are no duplicate vertices, and exactly two edges meet at any vertex. If your application requires the display of simple nonconvex polygons or of simple polygons containing holes, those polygons must first be subdivided into convex polygons before they can be displayed. Such subdivision is called *tessellation*. GLU provides a collection of routines that perform tessellation. Note that the GLU tessellation routines can't handle nonsimple polygons; there's no standard OpenGL method to handle such polygons.

Because tessellation is often required and can be rather tricky, this section describes the GLU tessellation routines in detail. These routines take as input arbitrary simple polygons that might include holes, and they return some combination of triangles, triangle meshes, and triangle fans. You can insist on triangles only if you don't want to have to deal with meshes or fans. If you care about performance, however, you should probably take advantage of any available mesh or fan information.

The Callback Mechanism

To tessellate a polygon using the GLU, first create a tessellation object, then provide a series of callback routines to be called at appropriate times during the tessellation. After specifying the callbacks, describe the polygon and any holes using GLU routines, which are similar to the OpenGL polygon routines. When the polygon description is complete, the tessellation facility invokes your callback routines as necessary.

The callback routines typically save the data for the triangles, triangle meshes, and triangle fans in user-defined data structures, or in OpenGL display lists. To render the polygons, other code traverses the data structures or calls the display lists. Although the callback routines could call OpenGL commands to display them directly, this is usually not done, as tessellation can be computationally expensive. It's a good idea to save the data if there is any chance that you want to display it again. The GLU tessellation routines are guaranteed never to return any new vertices, so interpolation of vertices, texture coordinates, or colors is never required.

The Tessellation Object

As a complex polygon is being described and tessellated, it has associated data, such as the vertices, edges, and callback functions. All this data is tied to a single tessellation object. To tessellate, your program must first create a tessellation object using the routine **gluNewTess**.

GLUtriangulatorObj* **gluNewTess(**void**);**

Creates a new tessellation object and returns a pointer to it. A null pointer is returned if the creation fails.

If you no longer need a tessellation object, you can delete it and free all associated memory with **gluDeleteTess**.

void **gluDeleteTess(**GLUtriangulatorObj **tessobj***)***;**

Deletes the specified tessellation object, *tessobj*, and frees all associated memory.

A single tessellation object can be reused for all your tessellations. This object is required only because library routines might be required to do their own tessellations, and they should be able to do so without interfering with any tessellation that your program is doing. It might also be useful to have multiple tessellation objects if you want to use different sets of callbacks for different tessellations. A typical program, however, allocates a single tessellation object and uses it for all its tessellations. There's no real need to free it because it uses a small amount of memory. On the other hand, if you're writing a library routine that uses the GLU tessellation, you'll want to be careful to free any tessellation objects you create.

Specifying Callbacks

You can specify up to five callback functions for a tessellation. Any functions that are omitted are simply not called during the tessellation, and any information they might have returned to your program is lost. All are specified by the single routine **gluTessCallback**.

void **gluTessCallback(**GLUtriangulatorObj ***tessobj*, GLenum *type*,
    void **(*****fn*****)( ));**

Associates the callback function fn with the tessellation object *tessobj*. The type of the callback is determined by the parameter *type*, which can be GLU_BEGIN, GLU_EDGE_FLAG, GLU_VERTEX, GLU_END, or GLU_ERROR. The five possible callback functions have the following prototypes:

GLU_BEGIN	void **begin(**GLenum *type***);**
GLU_EDGE_FLAG	void **edgeFlag(**GLboolean *flag***);**
GLU_VERTEX	void **vertex(**void ***data***);**
GLU_END	void **end(***void***);**
GLU_ERROR	void **error(**GLenum *errno***);**

To change a callback routine, simply call **gluTessCallback** with the new routine. To eliminate a callback routine without replacing it with a new one, pass **gluTessCallback** a null pointer for the appropriate function.

As tessellation proceeds, these routines are called in a manner similar to the way you would use the OpenGL commands **glBegin**, **glEdgeFlag**, **glVertex**, and **glEnd**. The error callback is invoked during the tessellation only if something goes wrong.

The GLU_BEGIN callback is invoked with one of three possible parameters: GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES. After this routine is called, and before the callback associated with GLU_END is called, some combination of the GLU_EDGE_FLAG and GLU_VERTEX callbacks is invoked. The associated vertices and edge flags are interpreted exactly as they are in OpenGL between **glBegin(GL_TRIANGLE_FAN)**, **glBegin(GL_TRIANGLE_STRIP)**, or **glBegin(GL_TRIANGLES)** and the matching **glEnd**. Since edge flags make no sense in a triangle fan or triangle strip, if there is a callback associated with GLU_EDGE_FLAG, the GLU_BEGIN callback is called only with GL_TRIANGLES. The GLU_EDGE_FLAG callback works exactly analogously to the OpenGL **glEdgeFlag** call.

The error callback is passed a GLU error number. A character string describing the error can be obtained using the routine **gluErrorString**.

Describing the Polygon to Be Tessellated

The polygon to be tessellated, possibly containing holes, is specified using the following four routines: **gluBeginPolygon**, **gluTessVertex**, **gluNextContour**, and **gluEndPolygon**. For polygons without holes, the specification is exactly as in OpenGL: start with **gluBeginPolygon**, call **gluTessVertex** for each vertex in the boundary, and end the polygon with a call to **gluEndPolygon**. If a polygon consists of multiple contours, including holes and holes within holes, the contours are specified one after the other, each preceded by **gluNextContour**. When **gluEndPolygon** is called, it signals the end of the final contour and starts the tessellation. You can omit the call to **gluNextContour** before the first contour. The detailed descriptions of these functions follow.

void **gluBeginPolygon(**GLUtriangulatorObj **tessobj***);*

Begins the specification of a polygon to be tessellated and associates a tessellation object, *tessobj*, with it. The callback functions to be used are those that were bound to the tessellation object using the routine **gluTessCallback**.

void **gluTessVertex(**GLUtriangulatorObj **tessobj***,*
GLdouble *v*[3]**,** void ***data*);

Specifies a vertex in the polygon to be tessellated. Call this routine for each vertex in the polygon to be tessellated. *tessobj* is the tessellation object to use, *v* contains the three-dimensional vertex coordinates, and *data* is an arbitrary pointer that is sent to the callback associated with GLU_VERTEX. Typically, it contains vertex data, texture coordinates, color information, or whatever else the application may require.

void **gluNextContour(**GLUtriangulatorObj **tessobj***,* GLenum *type***);**

Marks the beginning of the next contour when multiple contours make up the boundary of the polygon to be tessellated. *type* can be GLU_EXTERIOR, GLU_INTERIOR, GLU_CCW, GLU_CW, or GLU_UNKNOWN. These serve only as hints to the tessellation. If you get them right, the tessellation might go faster. If you get them wrong, they're ignored, and the tessellation still works. For a polygon with holes, one contour is the exterior contour and the others interior. **gluNextContour** can be called immediately after **gluBeginPolygon**, but if it isn't, the first contour is assumed to be of type GLU_EXTERIOR. GLU_CW and GLU_CCW indicate clockwise- and counterclockwise- oriented polygons. Choosing which are clockwise and which are counterclockwise is arbitrary in three dimensions, but in any plane, there are two different orientations, and the GLU_CW and GLU_CCW types should be used consistently. Use GLU_UNKNOWN if you don't know which to use.

void **gluEndPolygon(**GLUtriangulatorObj **tessobj***);*

Indicates the end of the polygon specification and that the tessellation can begin using the tessellation object *tessobj*.

Rendering Spheres, Cylinders, and Disks

The GLU includes a set of routines for drawing various simple surfaces (spheres, cylinders, disks, and parts of disks) in a variety of styles and orientations. These routines are described in detail in the *OpenGL Reference Manual*; their use is discussed briefly in the following paragraphs, and their prototypes are also listed.

To create a quadric object, use **gluNewQuadric**. (To destroy this object when you're finished with it, use **gluDeleteQuadric**.) Then specify the desired rendering style, as follows, with the appropriate routine (unless you're satisfied with the default values):

- Whether surface normals should be generated, and if so, whether there should be one normal per vertex or one normal per face: **gluQuadricNormals**.
- Whether texture coodinates should be generated: **gluQuadricTexture**.
- Which side of the quadric should be considered the outside and which the inside: **gluQuadricOrientation**.
- Whether the quadric should be drawn as a set of polygons, lines, or points: **gluQuadricDrawStyle**.

After you've specified the rendering style, simply invoke the rendering routine for the desired type of quadric object: **gluSphere**, **gluCylinder**, **gluDisk**, or **gluPartialDisk**. If an error occurs during rendering, the error-handling routine you've specified with **gluQuadricCallBack** is invoked.

It's better to use the **Radius*, *height*, and similar arguments to scale the quadrics rather than the **glScale** command, so that unit-length normals that are generated don't have to be renormalized. Set the *loops* and *stacks* arguments to values other than 1 to force lighting calculations at a finer granularity, especially if the material specularity is high.

The prototypes are listed in three categories.

Manage quadric objects:

GLUquadricObj* **gluNewQuadric** (void);
  void **gluDeleteQuadric (**GLUquadricObj **state***);**
  void **gluQuadricCallback (**GLUquadricObj **qobj***,* GLenum *which***,**
    void **(***fn***)( ));**

Control the rendering:

void **gluQuadricNormals (**GLUquadricObj **quadObject***,*
    GLenum *normals***);**
  void **gluQuadricTexture (**GLUquadricObj **quadObject***,*
    GLboolean *textureCoords***);**
  void **gluQuadricOrientation (**GLUquadricObj **quadObject***,*
    GLenum *orientation***);**
  void **gluQuadricDrawStyle (**GLUquadricObj **quadObject***,*
    GLenum *drawStyle***);**

Specify a quadric primitive:

void **gluCylinder (**GLUquadricObj **qobj***,* GLdouble *baseRadius***,**
    GLdouble *topRadius***,** GLdouble *height***,** GLint *slices***,**
    GLint *stacks***);**
  void **gluDisk (**GLUquadricObj **qobj***,* GLdouble *innerRadius***,**
    GLdouble *outerRadius***,** *GLint* slices**,** GLint *loops***);**
  void **gluPartialDisk (**GLUquadricObj **qobj***,* GLdouble *innerRadius***,**
    GLdouble *outerRadius***,** GLint *slices***,** GLint *loops***,**
    GLdouble *startAngle***,** GLdouble *sweepAngle***);**
  void **gluSphere (**GLUquadricObj **qobj***,* GLdouble *radius***,** GLint *slices***,**
    GLint *stacks***);**

NURBS Curves and Surfaces

NURBS routines provide general and powerful descriptions of curves and surfaces in two and three dimensions. They're used to represent geometry in many computer-aided mechanical design systems. The GLU NURBS routines can render such curves and surfaces in a variety of styles, and they can automatically handle adaptive subdivision that tessellates the domain into smaller triangles in regions of high curvature and near silhouette edges.

Manage a NURBS object:

GLUnurbsObj* **gluNewNurbsRenderer (**void**);**
  void **gluDeleteNurbsRenderer (**GLUnurbsObj **nobj***);*
  void **gluNurbsCallback (**GLUnurbsObj **nobj***, GLenum *which*,
    void **(***fn***)( ));**

Create a NURBS curve:

void **gluBeginCurve (**GLUnurbsObj **nobj***);*
  void **gluEndCurve (**GLUnurbsObj **nobj***);*
  void **gluNurbsCurve (**GLUnurbsObj **nobj***, GLint *nknots*, GLfloat **knot***,*
    GLint *stride*, GLfloat **ctlarray*,
    GLint *order*, GLenum *type*);

Create a NURBS surface:

void **gluBeginSurface (**GLUnurbsObj **nobj***);*
  void **gluEndSurface (**GLUnurbsObj **nobj***);*
  void **gluNurbsSurface (**GLUnurbsObj **nobj***, GLint *uknot_count*,
    GLfloat **uknot***, GLint *vknot_count*, GLfloat **vknot***,*
    GLint *u_stride*, GLint *v_stride*, GLfloat **ctlarray*,
    GLint *uorder*, GLint *vorder*, GLenum *type*);

Define a trimming region:

void **gluBeginTrim (**GLUnurbsObj **nobj***);*
  void **gluEndTrim (**GLUnurbsObj **nobj***);*
  void **gluPwlCurve (**GLUnurbsObj **nobj***, GLint *count*, GLfloat **array*,
    GLint *stride*, GLenum *type*);

Control NURBS rendering:

void **gluLoadSamplingMatrices (**GLUnurbsObj **nobj*,
    const GLfloat *modelMatrix*[16],
    const GLfloat *projMatrix*[16],
    const GLint *viewport*[4]**);**
  void **gluNurbsProperty (**GLUnurbsObj **nobj***, GLenum *property*,
    GLfloat *value*);
  void **gluGetNurbsProperty (**GLUnurbsObj **nobj***, GLenum *property*,
    GLfloat **value*);

Describing Errors

The GLU provides a routine for obtaining a descriptive string for an error code.

const GLubyte* **gluErrorString**(GLenum *errorCode*);

Returns a pointer to a descriptive string that corresponds to the OpenGL, GLU, or GLX error number passed in *errorCode*. The defined error codes are described in the *OpenGL Reference Manual* along with the command or routine that can generate them.

Introduction to OpenGL

As a software interface for graphics hardware, OpenGL's main purpose is to render two- and three-dimensional objects into a frame buffer. These objects are described as sequences of vertexes (which define geometric objects) or pixels (which define images). OpenGL performs several processing steps on this data to convert it to pixels to form the final desired image in the frame buffer.

The following topics present a global view of how OpenGL works:

- **OpenGL Fundamentals** briefly explains basic OpenGL concepts, such as what a graphic primitive is and how OpenGL implements a client-server execution model.
- **Basic OpenGL Operation** gives a high-level description of how OpenGL processes data and produces a corresponding image in the frame buffer.

OpenGL Fundamentals

The following topics explain some of the concepts inherent in OpenGL.

Primitives and Commands

OpenGL draws primitives—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of one another; that is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the frame buffer). Primitives are specified, modes are set, and other OpenGL operations are described by issuing commands in the form of function calls.

Primitives are defined by a group of one or more vertexes. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exception to this rule is if the group of vertexes must be clipped so that a particular primitive fits within a specified region. In this case, vertex data may be modified and new vertexes created. The type of clipping depends on which primitive the group of vertexes represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that is consistent with complete execution of all previously issued OpenGL commands.

Procedural versus Descriptive

OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. However, it doesn't provide you with a means for describing or modeling complex geometric objects. Thus, the OpenGL commands you issue specify how a certain result should be produced (what procedure should be followed) rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. Because of this procedural nature, it helps to know how OpenGL works—the order in which it carries out its operations, for example—to fully understand how to use it.

Execution Model

The model for interpretation of OpenGL commands is client-server. An application (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several GL contexts, each of which is an encapsulated GL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

The effects of OpenGL commands on the frame buffer are ultimately controlled by the window system that allocates frame buffer resources. The window system determines which portions of the frame buffer OpenGL may access at any given time and communicates to OpenGL how those portions are structured. Therefore, there are no OpenGL commands to configure the frame buffer or initialize OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system; OpenGL initialization takes place when the window system allocates a window for OpenGL rendering.

Basic OpenGL Operation

The figure shown below gives an abstract, high-level block diagram of how OpenGL processes data. In the diagram, commands enter from the left and proceed through what can be thought of as a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during the various processing stages.

{ewc msdncd, EWGraphic, group10420 0 /a "SDK.bmp"}

As shown in the diagram's first block, rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.

The evaluator stage of processing provides an efficient means for approximating curve and surface geometry by evaluating polynomial commands of input values. During the next stage, per-vertex operations and primitive assembly, OpenGL processes geometric primitives—points, line segments, and polygons, all of which are described by vertexes. Vertexes are transformed and lit, and primitives are clipped to the viewport in preparation for the next stage.

Rasterization produces a series of frame buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations, which performs the final operations on the data before it's stored as pixels in the frame buffer. These operations include conditional updates to the frame buffer based on incoming and previously stored z-values (for z-buffering) and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

Input data can be in the form of pixels rather than vertexes. Such data, which might describe an image for use in texture mapping, skips the first stage of processing described above and instead is processed as pixels, in the pixel operations stage. The result of this stage is either stored as texture memory, for use in the rasterization stage, or rasterized and the resulting fragments merged into the frame buffer just as if they were generated from geometric data.

All elements of OpenGL state, including the contents of the texture memory and even of the frame buffer, can be obtained by an OpenGL application.

Overview of Commands and Routines

Many OpenGL commands pertain specifically to drawing objects such as points, lines, polygons, and bitmaps. Other commands control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Still other commands are specifically concerned with frame buffer manipulation. This section describes how all the OpenGL commands work together to create the OpenGL processing pipeline. Brief overviews are also given of the routines comprising the OpenGL Utility Library (GLU).

- **OpenGL Processing Pipeline** expands on the discussion in **Introduction to OpenGL** by explaining how specific OpenGL commands control the processing of data.
- **Additional OpenGL Commands** discusses several sets of OpenGL commands not covered in the previous section.
- **OpenGL Utility Library** describes the GLU routines that are available.

OpenGL Processing Pipeline

This section takes a closer look at the stages in which data is actually processed, and ties these stages to OpenGL commands. Scroll to the end of this topic to see a more detailed block diagram of the OpenGL processing pipeline.

For most of the pipeline, you can see three vertical arrows between the major stages. These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates. Also note that vertices are assembled into primitives, then to fragments, and finally to pixels in the frame buffer. This progression is discussed in more detail in **Vertices**, **Primitives**, **Fragments**, and **Pixels**.

As you continue reading, be aware that we've taken some liberties with command names. Many OpenGL commands are simple variations of each other, differing mostly in the data type of arguments. Some commands differ in the number of related arguments and whether those arguments can be specified as a vector or whether they must be specified separately in a list. For example, if you use the **glVertex2f** command, you need to supply *x* and *y* coordinates as 32-bit floating-point numbers; with **glVertex3sv**, you must supply an array of three short (16-bit) integer values for *x*, *y*, and *z*. For simplicity, only the base name of the command is used in the topics that follow, and an asterisk is included to indicate that there may be more to the actual command name than is shown. For example, **glVertex** stands for all variations of the command you use to specify vertices.

Also keep in mind that the effect of an OpenGL command may vary depending on whether certain modes are enabled. For example, you need to enable lighting if the lighting-related commands are to produce a properly lit object. To enable a particular mode, use the **glEnable** command and supply the appropriate constant to identify the mode (for example, GL_LIGHTING). Refer to **glEnable** for a complete list of the modes that can be enabled. Modes are disabled with **glDisable**.

{ewc msdncd, EWGraphic, group10421 0 /a "SDK.bmp"}

{ewc msdncd, EWGraphic, group10421 1 /a "SDK.bmp"}

Vertices

This topic relates the OpenGL commands that perform per-vertex operations to the processing stages shown in **OpenGL Processing Pipeline**.

Input Data

You must provide several types of input data to the OpenGL pipeline:

- Vertices—Vertices describe the shape of the desired geometric object. To specify vertices, use **glVertex** commands in conjunction with **glBegin** and **glEnd** to create a point, line, or polygon. You can also use **glRect** to describe an entire rectangle at once.
- Edge flag—By default, all edges of polygons are boundary edges. Use the **glEdgeFlag** command to explicitly set the edge flag.
- Current raster position—Specified with **glRasterPos**, the current raster position is used to determine raster coordinates for pixel and bitmap drawing operations.
- Current normal—A normal vector associated with a particular vertex determines how a surface at that vertex is oriented in three-dimensional space; this in turn affects how much light that particular vertex receives. Use **glNormal** to specify a normal vector.
- Current color—The color of a vertex, together with the lighting conditions, determine the final, lit color. Color is specified with **glColor** if in RGBA mode or with **glIndex** if in color index mode.
- Current texture coordinates—Specified with **glTexCoord**, texture coordinates determine the location in a texture map that should be associated with a vertex of an object.
- When **glVertex** is called, the resulting vertex inherits the current edge flag, normal, color, and texture coordinates. Therefore, **glEdgeFlag**, **glNormal**, **glColor**, and **glTexCoord** must be called before **glVertex** if they are to affect the resulting vertex.

Matrix Transformations

Vertices and normals are transformed by the modelview and projection matrices before they're used to produce an image in the frame buffer. You can use commands such as **glMatrixMode**, **glMultMatrix**, **glRotate**, **glTranslate**, and **glScale** to compose the desired transformations, or you can directly specify matrices with **glLoadMatrix** and **glLoadIdentity**. Use **glPushMatrix** and **glPopMatrix** to save and restore modelview and projection matrices on their respective stacks.

Lighting and Coloring

In addition to specifying colors and normal vectors, you may define the desired lighting conditions with **glLight** and **glLightModel**, and the desired material properties with **glMaterial**. Related commands for controlling how lighting calculations are performed include **glShadeModel**, **glFrontFace**, and **glColorMaterial**.

Generating Texture Coordinates

Rather than explicitly supplying texture coordinates, you can have OpenGL generate them as a function of other vertex data. This is what the **glTexGen** command does. After the texture coordinates have been specified or generated, they are transformed by the texture matrix. This matrix is controlled with the same commands for matrix transformations.

Primitive Assembly

Once all necessary calculations have been performed, vertices are assembled into **primitives**—points, line segments, or polygons—together with the relevant edge flag, color, and texture information for each vertex.

Primitives

Primitives are converted to pixel fragments in several steps: primitives are clipped appropriately, whatever corresponding adjustments are necessary are made to the color and texture data, and the relevant coordinates are transformed to window coordinates. Finally, rasterization converts the clipped primitives to pixel fragments.

Clipping

Points, line segments, and polygons are handled slightly differently during clipping. Points are either retained in their original state (if they're inside the clip volume) or discarded (if they're outside). If portions of line segments or polygons are outside the clip volume, new vertices are generated at the clip points. For polygons, an entire edge may need to be constructed between such new vertices. For both line segments and polygons that are clipped, the edge flag, color, and texture information is assigned to all new vertices.

Clipping actually happens in two steps:

1. Application-specific clipping—Immediately after primitives are assembled, they're clipped in eye coordinates as necessary for any arbitrary clipping planes you've defined for your application with **glClipPlane**. (OpenGL requires support for at least six such application-specific clipping planes.)
2. View volume clipping—Next, primitives are transformed by the projection matrix (into clip coordinates) and clipped by the corresponding viewing volume. This matrix can be controlled by the previously mentioned matrix transformation commands but is most typically specified by **glFrustum** or **glOrtho**.

Transforming to Window Coordinates

Before clip coordinates can be converted to window coordinates, they are normalized by dividing by the value of *w* to yield normalized device coordinates. After that, the viewport transformation applied to these normalized coordinates produces window coordinates. You control the viewport, which determines the area of the on-screen window that displays an image, with **glDepthRange** and **glViewport**.

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color, depth, and texture data. Together, a point and its associated information are called a fragment. The current raster position (as specified with **glRasterPos**) is used in various ways during this stage for pixel drawing and bitmaps. As discussed below, different issues arise when rasterizing the three different types of primitives. In addition, pixel rectangles and bitmaps need to be rasterized.

Primitives. Control how primitives are rasterized with commands that allow you to choose dimensions and stipple patterns: **glPointSize**, **glLineWidth**, **glLineStipple**, and **glPolygonStipple**. In addition, you can control how the front and back faces of polygons are rasterized with **glCullFace**, **glFrontFace**, and **glPolygonMode**.

Pixels. Several commands control pixel storage and transfer modes. The command **glPixelStore** controls the encoding of pixels in client memory, and **glPixelTransfer** and **glPixelMap** control how pixels are processed before being placed in the frame buffer. A pixel rectangle is specified with **glDrawPixels**; its rasterization is controlled with **glPixelZoom**.

Bitmaps. Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. A bitmap is specified using **glBitmap**.

Texture Memory. Texturing maps a portion of a specified texture image onto each primitive when texturing is enabled. This mapping is accomplished by using the color of the texture image at the location indicated by a fragment's texture coordinates to modify the fragment's RGBA color. A texture image is specified using **glTexImage2D** or **glTexImage1D**. The commands **glTexParameter** and **glTexEnv** control how texture values are interpreted and applied to a fragment.

Fog. You can have OpenGL blend a fog color with a rasterized fragment's post-texturing color using a blending factor that depends on the distance between the eyepoint and the fragment. Use **glFog** to specify the fog color and blending factor.

Fragments

OpenGL allows a fragment produced by rasterization to modify the corresponding pixel in the frame buffer only if it passes a series of tests. If it does pass, the fragment's data can be used directly to replace the existing frame buffer values, or it can be combined with existing data in the frame buffer, depending on the state of certain modes.

Pixel Ownership Test

The first test is to determine whether the pixel in the frame buffer corresponding to a particular fragment is owned by the current OpenGL context. If so, the fragment proceeds to the next test. If not, the window system determines whether the fragment is discarded or whether any further fragment operations will be performed with that fragment. This test allows the window system to control OpenGL's behavior when, for example, an OpenGL window is obscured.

Scissor Test

With the **glScissor** command, you can specify an arbitrary screen-aligned rectangle outside of which fragments will be discarded.

Alpha Test

The alpha test (which is performed only in RGBA mode) discards a fragment depending on the outcome of a comparison between the fragment's alpha value and a constant reference value. The comparison command and reference value are specified with **glAlphaFunc**.

Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer and a reference value. The command **glStencilFunc** specifies the comparison command and the reference value. Whether the fragment passes or fails the stencil test, the value in the stencil buffer is modified according to the instructions specified with **glStencilOp**.

Depth Buffer Test

The depth buffer test discards a fragment if a depth comparison fails; **glDepthFunc** specifies the comparison command. The result of the depth comparison also affects the stencil buffer update value if stenciling is enabled.

Blending

Blending combines a fragment's R, G, B, and A values with those stored in the frame buffer at the corresponding location. The blending, which is performed only in RGBA mode, depends on the alpha value of the fragment and that of the corresponding currently stored pixel; it might also depend on the RGB values. You control blending with **glBlendFunc**, which allows you to indicate the source and destination blending factors.

Dithering

If dithering is enabled, a dithering algorithm is applied to the fragment's color or color index value. This algorithm depends only on the fragment's value and its *x* and *y* window coordinates.

Logical Operations

Finally, a logical operation can be applied between the fragment and the value stored at the corresponding location in the frame buffer; the result replaces the current frame buffer value. You choose the desired logical operation with **glLogicOp**. Logical operations are performed only on color indices, never on RGBA values.

Pixels

During the previous stage of the OpenGL pipeline, fragments are converted to pixels in the frame buffer. The frame buffer is actually organized into a set of logical buffers—the color, depth, stencil, and accumulation buffers. The color buffer itself consists of a front left, front right, back left, back right, and some number of auxiliary buffers. You can issue commands to control these buffers, and you can directly read or copy pixels from them. (Note that the particular OpenGL context you're using may not provide all of these buffers.)

Frame Buffer Operations

You can select into which buffer color values are written with **glDrawBuffer**. In addition, four different commands are used to mask the writing of bits to each of the logical frame buffers after all per-fragment operations have been performed: **glIndexMask**, **glColorMask**, **glDepthMask**, and **glStencilMask**. The operation of the accumulation buffer is controlled with **glAccum**. Finally, **glClear** sets every pixel in a specified subset of the buffers to the value specified with **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil**, or **glClearAccum**.

Reading or Copying Pixels

You can read pixels from the frame buffer into memory, encode them in various ways, and store the encoded result in memory with **glReadPixels**. In addition, you can copy a rectangle of pixel values from one region of the frame buffer to another with **glCopyPixels**. The command **glReadBuffer** controls from which color buffer the pixels are read or copied.

Additional OpenGL Commands

The following topics briefly describe special groups of commands that weren't explicitly shown as part of OpenGL's processing pipeline. These commands accomplish such diverse tasks as evaluating polynomials, using display lists, and obtaining the values of OpenGL state variables.

Using Evaluators

OpenGL's evaluator commands allow you to use a polynomial mapping to produce vertices, normals, texture coordinates, and colors. These calculated values are then passed on to the pipeline as if they had been directly specified. The evaluator facility is also the basis for the NURBS (Non-Uniform Rational B-Spline) commands, which allow you to define curves and surfaces, as described in **OpenGL Utility Library**.

The first step involved in using evaluators is to define the appropriate one- or two-dimensional polynomial mapping using **glMap**. The domain values for this map can then be specified and evaluated in one of two ways:

- By defining a series of evenly spaced domain values to be mapped using **glMapGrid** and then evaluating a rectangular subset of that grid with **glEvalMesh**. A single point of the grid can be evaluated using **glEvalPoint**.
- By explicitly specifying a desired domain value as an argument to **glEvalCoord**, which evaluates the maps at that value.

Performing Selection and Feedback

Selection, feedback, and rendering are mutually exclusive modes of operation. Rendering is the normal, default mode during which fragments are produced by rasterization; in selection and feedback modes, no fragments are produced and therefore no frame buffer modification occurs. In selection mode, you can determine which primitives would be drawn into some region of a window; in feedback mode, information about primitives that would be rasterized is fed back to the application. You select among these three modes with **glRenderMode**.

Selection

Selection works by returning the current contents of the name stack, which is an array of integer-valued names. You assign the names and build the name stack within the modeling code that specifies the geometry of objects you want to draw. Then, in selection mode, whenever a primitive intersects the clip volume, a selection hit occurs. The hit record, which is written into the selection array you've supplied with **glSelectBuffer**, contains information about the contents of the name stack at the time of the hit. (Note that **glSelectBuffer** needs to be called before OpenGL is put into selection mode with **glRenderMode**. Also, the entire contents of the name stack isn't guaranteed to be returned until **glRenderMode** is called to take OpenGL out of selection mode.) You manipulate the name stack with **glInitNames**, **glLoadName**, **glPushName**, and **glPopName**. In addition, you might want to use an OpenGL Utility Library routine for selection, **gluPickMatrix**, which is described in **OpenGL Utility Library**.

Feedback

In feedback mode, each primitive that would be rasterized generates a block of values that is copied into the feedback array. You supply this array with **glFeedbackBuffer**, which must be called before OpenGL is put into feedback mode. Each block of values 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. Values are not guaranteed to be written into the feedback array until **glRenderMode** is called to take OpenGL out of feedback mode. You can use **glPassThrough** to supply a marker that's returned in feedback mode as if it were a primitive.

Using Display Lists

A display list is simply a group of OpenGL commands that has been stored for subsequent execution. The **glNewList** command begins the creation of a display list, and **glEndList** ends it. With few exceptions, OpenGL commands called between **glNewList** and **glEndList** are appended to the display list, and optionally executed as well. (**glNewList** lists the commands that can't be stored and executed from within a display list.) To trigger the execution of a list or set of lists, use **glCallList** or **glCallLists** and supply the identifying number of a particular list or lists. You can manage the indices used to identify display lists with **glGenLists**, **glListBase**, and **glIsList**. Finally, you can delete a set of display lists with **glDeleteLists**.

Managing Modes and Execution

The effect of many OpenGL commands depends on whether a particular mode is in effect. You use **glEnable** and **glDisable** to set such modes and **glIsEnabled** to determine whether a particular mode is set.

You can control the execution of previously issued OpenGL commands with **glFinish**, which forces all such commands to complete, or **glFlush**, which ensures that all such commands will be completed in a finite time.

A particular implementation of OpenGL may allow certain behaviors to be controlled with hints, by using the **glHint** command. Possible behaviors are the quality of color and texture coordinate interpolation, the accuracy of fog calculations, and the sampling quality of antialiased points, lines, or polygons.

Obtaining State Information

OpenGL maintains many state variables that affect the behavior of many commands. Some of these variables have specialized query commands:

glGetLight	**glGetTexEnv**	**glGetTexParameter**
glGetMaterial	**glGetTexGen**	**glGetMap**
glGetClipPlane	**glGetTexImage**	**glGetPixelMap**
glGetPolygonStipple	**glGetTexLevelParameter**	

The value of other state variables can be obtained with **glGetBooleanv**, **glGetDoublev**, **glGetFloatv**, or **glGetIntegerv**, as appropriate. See **glGet** for information about how to use these commands. Other query commands you might want to use are **glGetError**, **glGetString**, and **glIsEnabled**. (See **Handling Errors** for more information about routines related to error handling.) Finally, you can save and restore sets of state variables with **glPushAttrib** and **glPopAttrib**.

OpenGL Utility Library

The OpenGL Utility Library (GLU) contains several groups of commands that complement the core OpenGL interface by providing support for auxiliary features. These utility routines make use of core OpenGL commands, so any OpenGL implementation is guaranteed to support the utility routines. Note that the prefix for Utility Library routines is "glu" rather than "gl."

Manipulating Images for Use in Texturing

The OpenGL Utility Library (GLU) provides image scaling and automatic mipmapping routines to simplify the specification of texture images. The routine **gluScaleImage** scales a specified image to an accepted texture size; the resulting image can then be passed to OpenGL as a texture. The automatic mipmapping routines **gluBuild1DMipmaps** and **gluBuild2DMipmaps** create mipmapped texture images from a specified image and pass them to **glTexImage1D** and **glTexImage2D**, respectively.

Transforming Coordinates

The OpenGL Utility Library (GLU) provides several commonly used matrix transformation routines. You can set up a two-dimensional orthographic viewing region with **gluOrtho2D**, a perspective viewing volume using **gluPerspective**, or a viewing volume that is centered on a specified eyepoint with **gluLookAt**. Each of these routines creates the desired matrix and applies it to the current matrix using **glMultMatrix**.

The **gluPickMatrix** routine simplifies selection by creating a matrix that restricts drawing to a small region of the viewport. If you rerender the scene in selection mode after this matrix has been applied, all objects that would be drawn near the cursor will be selected, and information about them will be stored in the selection buffer. See **Performing Selection and Feedback** for more information about selection mode.

To determine where in the window an object is being drawn, use the **gluProject** function, which converts specified coordinates from object coordinates to window coordinates. The **gluUnProject** function performs the inverse conversion.

Polygon Tessellation

The polygon tessellation routines triangulate a concave polygon with one or more contours. To use this GLU feature, first create a tessellation object with **gluNewTess**, and define callback routines that will be used to process the triangles generated by the tessellator (with **gluTessCallBack**). Then use **gluBeginPolygon**, **gluTessVertex**, **gluNextContour**, and **gluEndPolygon** to specify the concave polygon to be tessellated. Unneeded tessellation objects can be destroyed with **gluDeleteTess**.

Rendering Simple Surfaces

You can render spheres, cylinders, and disks using the GLU quadric routines. To do this, create a quadric object with **gluNewQuadric**. (To destroy this object when you're finished with it, use **gluDeleteQuadric**.) Then specify the desired rendering style, as listed below, with the appropriate routine (unless you're satisfied with the default values):

- Whether surface normals should be generated, and if so, whether there should be one normal per vertex or one normal per face: **gluQuadricNormals**
- Whether texture coodinates should be generated: **gluQuadricTexture**
- Which side of the quadric should be considered the outside and which the inside: **gluQuadricOrientation**
- Whether the quadric should be drawn as a set of polygons, lines, or points: **gluQuadricDrawStyle**

After you've specified the rendering style, simply invoke the rendering routine for the desired type of quadric object: **gluSphere**, **gluCylinder**, **gluDisk**, or **gluPartialDisk**. If an error occurs during rendering, the error-handling routine you've specified with **gluQuadricCallBack** is invoked.

NURBS Curves and Surfaces

Non-Uniform Rational B-Spline (NURBS) curves and surfaces are converted to OpenGL evaluators by the routines described in this section. You can create and delete a NURBS object with **gluNewNurbsRenderer** and **gluDeleteNurbsRenderer**, and establish an error-handling routine with **gluNurbsCallback**.

You specify the desired curves and surfaces with different sets of routines – **gluBeginCurve**, **gluNurbsCurve**, and **gluEndCurve** for curves or **gluBeginSurface**, **gluNurbsSurface**, and **gluEndSurface** for surfaces. You can also specify a trimming region, which defines a subset of the NURBS surface domain to be evaluated, thereby allowing you to create surfaces that have smooth boundaries or that contain holes. The trimming routines are **gluBeginTrim**, **gluPwlCurve**, **gluNurbsCurve**, and **gluEndTrim**.

As with quadric objects, you can control how NURBS curves and surfaces are rendered:

- Whether a curve or surface should be discarded if its control polyhedron lies outside the current viewport
- What the maximum length should be (in pixels) of edges of polygons used to render curves and surfaces
- Whether the projection matrix, modelview matrix, and viewport should be taken from the OpenGL server or whether you'll supply them explictly with **gluLoadSamplingMatrices**.

Use **gluNurbsProperty** to set these properties, or use the default values. You can query a NURBS object about its rendering style with **gluGetNurbsProperty**.

Handling Errors

The routine **gluErrorString** is provided for retrieving an error string that corresponds to an OpenGL or GLU error code. The currently defined OpenGL error codes are described in **glGetError**. The GLU error codes are listed in **gluErrorString**, **gluTessCallback**, **gluQuadricCallback**, and **gluNurbsCallback**.

IRIS GL and OpenGL Differences

This appendix lists the differences between OpenGL and IRIS GL. A term for each difference is given, followed by a description.

accumulation wrapping
: The OpenGL accumulation buffer operation is not defined when component values exceed 1.0 or drop below - 1.0.

antialiased lines
: OpenGL stipples antialiased lines. IRIS GL does not.

arc
: OpenGL supports arcs in its utility library.

attribute lists
: The attributes pushed by IRIS GL **pushattributes** differ from any of the attribute sets pushed by OpenGL **glPushAttrib**. Because all OpenGL states can be read back, however, you can implement any desired push/pop semantics using OpenGL.

automatic texture scaling
: The OpenGL texture interface does not support automatic scaling of images to power-of-two dimensions. However, the GLU supports image scaling.

bbox
: OpenGL doesn't support conditional execution of display lists.

callfunc
: OpenGL doesn't support callback from display lists. Note that IRIS GL doesn't support this functionality either, when client and server are on different platforms.

circle
: OpenGL supports circles with the GLU. In OpenGL both circles and arcs (disks and partial disks) can have holes. Also, you can change subdivision of the primitives in OpenGL, and the primitives' surface normals are available for lighting.

clear options
: OpenGL actually clears buffers. It doesn't apply currently specified pixel operations, such as blending and logicop, regardless of their modes. To clear using such features, you must render a window-size polygon.

closed lines
: OpenGL renders all single-width aliased lines such that abutting lines share no pixels. This means that the "last" pixel of an independent line is not drawn.

color/normal flag
: OpenGL lighting is explicitly enabled or disabled. When enabled, it is effective regardless of the order in which colors and normals are specified.

  You cannot enable or disable lighting between OpenGL **glBegin** and **glEnd** commands. To disable lighting between **glBegin** and **glEnd**, specify zero ambient, diffuse, and specular material reflectance and then set the material emission to the desired color.

concave polygons
: The core OpenGL API doesn't handle concave polygons, but the GLU support decomposing concave, non-self-intersecting contours into triangles. These triangles can either be drawn immediately or returned.

current computed color
: OpenGL has no equivalent to a current computed color. If you're using OpenGL as a lighting engine, you can use feedback to obtain colors generated by lighting calculations.

current graphics position
: OpenGL doesn't maintain a current graphics position. IRIS GL commands that depends on current graphics position, such as relative lines and polygons, are not included in OpenGL.

curves
: OpenGL does not support IRIS GL curves. Use of NURBS curves is recommended.

defs/binds
: OpenGL doesn't have the concept of material, light, or texture objects, only of material, light, and texture properties. You can use display lists to create their own objects, however.

depthcue
: OpenGL provides no direct support for depth cueing, but its fog support is a more general capability that you can easily use to emulate the IRIS GL **depthcue** function.

display list editing
: OpenGL display lists can't be edited, only created and destroyed. Because you specify display list names, however, you can redefine individual display lists in a hierarchy.

  OpenGL display lists are designed for data caching, not for database management. They are guaranteed to be stored on the server in client/server environments, so they are not limited by network bandwidth during execution.

  OpenGL display lists can be called between **glBegin** and **glEnd** commands, so the display list hierarchy can be made fine enough that it can, in effect, be edited.

error checking
: OpenGL checks for errors more carefully than IRIS GL. For example, all OpenGL functions that are not accepted between **glBegin** and **glEnd** are detected as errors, and have no other effect.

error return values
: When an OpenGL command that returns a value detects an error, it always returns zero. OpenGL commands that return data through passed pointers make no change to the array contents if an error is detected.

error side effects
: When an OpenGL command results in an error, its only side effect is to update the error flag to the appropriate value. No other state changes are made. (An exception is the OUT_OF_MEMORY error, which is fatal.)

feedback
: Feedback is standardized in OpenGL so it doesn't change from machine to machine.

fonts and strings
: OpenGL requires character glyphs to be manipulated as individual display lists. It provides a display list calling function that accepts a list of display list names, each name represented as 1, 2, or 4 bytes. **glCallLists** adds a separately specified offset to each display list name before the call, allowing lists of display list names to be treated as strings.

  This mechanism provides all the functionality of IRIS GL fonts, and considerably more. For example, characters comprised of triangles can be easily manipulated.

frontbuffer
: IRIS GL has complex rules for rendering to the front buffer in single buffer mode. OpenGL handles rendering to the front buffer in a straightforward way.

hollow polygons
: You can use the OpenGL stencil capacity to render hollow polygons. OpenGL doesn't support other means for creating hollow polygons.

index clamping
: Where possible, OpenGL treats color and stencil indexes as bit fields rather than numbers. Thus indexes are masked, rather than clamped, to the supported range of the framebuffer.

integer colors
: Signed integer color components (red, green, blue, or alpha) are mapped linearly to floating point such that the most negative integer maps to -1.0 and the most positive integer maps to 1.0. This mapping occurs when you specify the color, before OpenGL replaces the current color.

  Unsigned integer color components are mapped linearly to floating points such that 0 maps to 0.0 and the largest integer maps to 1.0. This mapping occurs when you specify the color, before

OpenGL replaces the current color.

integer normals

Integer normal components are mapped just like signed color components. The most negative integer maps to -1.0, and the most positive integer maps to 1.0. pixel fragments.

Pixels drawn by **glDrawPixels** or **glCopyPixels** are always rasterized and converted to fragments. The resulting fragments are textured, fogged, depth buffered, blended, and so on, just as if they were generated from geometric points. Fragment data that isn't provided by the source pixels is augmented from the current raster position. For example, RGBA pixels take the raster position Z and texture coordinates. Depth pixels take the raster position color and texture coordinates.

invariance

OpenGL guarantees certain consistency that IRIS GL doesn't. For example, OpenGL guarantees that identical code sequences sent to the same system, differing only in the specified blending function, will generate the same pixel fragments. (The fragments differ, however, if blending is enabled and then disabled.)

lighting equation

The OpenGL lighting equation differs slightly from the IRIS GL equation. OpenGL supports separate attenuation for each light source, rather than a single attenuation for all the light sources like IRIS GL. OpenGL adjusts the equation so that ambient, diffuse, and specular lighting contributions are all attenuated. Also, OpenGL allows you to specify separate colors for the ambient, diffuse, and specular intensities of light sources, as well as for the ambient, diffuse, and specular reflectance of materials. All OpenGL light and material colors include alpha.

Setting the specular exponent to zero does not defeat specular lighting in OpenGL.

mapw

OpenGL utilities support mapping between object and window coordinates.

matrix mode

Where the IRIS GL **ortho**, **ortho2**, **perspective**, and **window** functions operate on a particular matrix, all OpenGL matrix operations work on the current matrix. All OpenGL matrix operations except **glLoadIdentity** and **glLoadMatrix** multiply the current matrix rather than replacing it (as do **ortho**, **ortho2**, **perspective**, and **window** in the IRIS GL).

mipmaps, automatic generation

The OpenGL texture interface does not support automatic generation of mipmap images. However, the GLU supports the automatic generation of mipmap images for both 1D and 2D textures.

move/draw/pmove/pdraw/pclos

OpenGL supports only Begin/End style graphics, because it does not maintain a current graphics position. The scalar parameter specification of the old move/draw commands is accepted by OpenGL for all vertex related commands, however.

mprojection mode

IRIS GL doesn't transform geometry by the ModelView matrix while in Projection matrix mode. OpenGL always transforms by both the ModelView and the Projection matrix, regardless of matrix mode.

multi-buffer drawing

OpenGL renders to each color buffer individually, rather than computing a single new color value based on the contents of one color buffer and writing it to all the enabled color buffers, as IRIS GL does.

NURBS

OpenGL supports NURBS with a combination of core capability (evaluators) and GLU support. All IRIS GL NURBS capabilities are supported.

old polygon mode

Aliased OpenGL polygons are always point-sampled. IRIS GL's polygon compatibility mode, where pixels outside the polygon perimeter are included in its rasterization, is not supported. If your code uses this polygon mode, it's probably for rectangles. Old polygon mode rectangles appear one pixel wider and higher.

packed color formats
: OpenGL accepts colors as 8-bit components, but these components are treated as an array of bytes rather than as bytes packed into larger words. By encouraging array indexing rather than shifting, OpenGL promotes endian-invariant programming.

  Just as IRIS GL accepts packed colors both for geometric and pixel rendering, OpenGL accepts arrays of color components for geometric and pixel rendering.

patches
: OpenGL doesn't support IRIS GL patches.

per-bit color writemask
: OpenGL writemasks for color components enable or disable changes to the entire component (red, green, blue, or alpha), not to individual bits of components. Note that per-bit writemasks are supported for both color indexes and stencil indexes, however.

per-bit depth writemask
: OpenGL writemasks for depth components enable or disable changes to the entire component, not to individual bits of the depth component.

pick
: The OpenGL utility library includes support for generating a pick matrix.

pixel coordinates
: In both OpenGL and IRIS GL, the origin of a window's coordinate system is at its lower left corner. OpenGL places the origin at the lower left corner of this pixel, however, while IRIS GL places it at the center of the lower left pixel.

pixel zoom
: OpenGL negative zoom factors reflect about the current graphics position. IRIS GL doesn't define the operation of negative zoom factors, and instead provides RIGHT_TO_LEFT and TOP_TO_BOTTOM reflection pixmodes. These reflection modes reflect in place, rather than about the current raster position. OpenGL doesn't define reflection modes.

pixmode
: OpenGL pixel transfers operate on individual color components, rather than on packed groups of four 8-bit components as does IRIS GL. While OpenGL provides substantially more pixel capability than IRIS GL, it doesn't support packed color constructs, and it doesn't enable color components to be reassigned (red to green, red to blue, etc.) during pixel copy operations.

polf/poly
: OpenGL provides no direct support for vertex lists other than display lists. Functions like **polf** and **poly** can be implemented easily using the OpenGL API, however.

polygon provoking vertex
: Flat shaded IRIS GL polygons take the color of the last vertex specified, while OpenGL polygons take the color of the first vertex specified.

polygon stipple
: With IRIS GL the polygon stipple pattern is relative to the screen. With OpenGL it is relative to a window.

polygon vertex count
: There is no limit to the number of vertexes between **glBegin** and **glEnd** with OpenGL, even for **glBegin(POLYGON)**. With IRIS GL polygons are limited to no more than 255 vertexes.

readdisplay
: Reading pixels outside window boundaries is properly a window system capability, rather than a rendering capability. Use Win32 functions to replace the IRIS GL readdisplay command.

relative move/draw/pmove/pdraw/pclos
: OpenGL doesn't maintain a current graphics position, and therefore doesn't support relative vertex operations.

RGBA logicop()
: OpenGL does not support logical operations on RGBA buffers.

sbox()
: **sbox** is an IRIS GL rectangle primitive that is well-defined only if transformed without rotation. It is designed to be rendered faster than standard rectangles. While OpenGL doesn't support such a primitive, it can be tuned to render rectangles very quickly when the matrixes and other modes are in states that simplify calculations.

scalar arguments
: All OpenGL commands that are accepted between **glBegin** and **glEnd** have entry points that accept scalar arguments. For example, **glColor4f(red,green,blue,alpha)**.

scissor
: OpenGL **glScissor** doesn't track the viewport. The IRIS GL **viewport** command automatically updates the **scrmask**.

scrbox()
: OpenGL doesn't support bounding box computation.

scrsubdivide()
: OpenGL doesn't support screen subdivision.

single matrix mode
: OpenGL always maintains two matrixes: ModelView and Projection. While an OpenGL implementation can consolidate these into a single matrix for performance reasons, it must always present the 2-matrix model to the programmer.

subpixel mode
: All OpenGL rendering is subpixel positioned—subpixel mode is always on.

swaptmesh()
: OpenGL doesn't support the **swaptmesh** capability. It does offer two types of triangle meshes, however: one that corresponds to the default "strip" behavior of the IRIS GL, and another that corresponds to calling **swaptmesh** prior to the third and all subsequent vertexes when using IRIS GL.

vector arguments
: All OpenGL commands that are accepted between **glBegin** and **glEnd** have entry points that accept vector arguments. For example, **glColor4fv(v)**.

window management
: OpenGL includes no window system commands. It is always supported as an extension to a window or operating system that includes capability for device and window control. Each extension provides a system-specific mechanism for creating, destroying, and manipulating OpenGL rendering contexts. For example, the OpenGL extension to the X window system (GLX) includes roughly 10 commands for this purpose.

  IRIS GL commands such as **gconfig** and **drawmode** are not implemented by OpenGL.

window offset
: IRIS GL returns viewport and character positions in screen, rather than window, coordinates. OpenGL always uses window coordinates.

z rendering
: OpenGL doesn't support rendering colors to the depth buffer. It does allow for additional color buffers, which can be implemented using the same memory that is used for depth buffers in other window configurations. But these additional color buffers cannot share memory with the depth buffer in any single configuration.

OpenGL Commands and Their IRIS GL Equivalents

This appendix lists IRIS GL functions and their equivalent OpenGL functions. The first column is an alphabetical list of IRIS GL functions, the second column contains the corresponding functions to use in OpenGL.

Note In many cases the OpenGL commands listed will function somewhat differently from the IRIS GL commands, and the parameters may be different as well. For more information on the differences between IRIS GL and OpenGL, see "IRIS GL and OpenGL Differences."

IRIS GL Call	OpenGL/GLU/Win32 Equivalent
acbuf	glAccum
acsize	ChoosePixelFormat
addtopup	Use Win32 for menus.
afunction	glAlphaFunc
arc	gluPartialDisk
backbuffer	glDrawBuffer(GL_BACK)
backface	glCullFace(GL_BACK)
bbox2	Not supported.
bgnclosedline	glBegin(GL_LINE_LOOP)
bgncurve	gluBeginCurve
bgnline	glBegin(GL_LINE_STRIP)
bgnpoint	glBegin(GL_POINTS)
bgnpolygon	glBegin(GL_POLYGON)
bgnqstrip	glBegin(GL_QUAD_STRIP)
bgnsurface	gluBeginSurface
bgntmesh	glBegin(GL_TRIANGLE_STRIP)
bgntrim	gluBeginTrim
blankscreen	Use Win32 for windowing.
blanktime	Use Win32 for windowing.
blendfunction	glBlendFunc
blink	Use Win32 for color maps.
blkqread	Use Win32 for event handling.
c	glColor
callfunc	Not supported.
callobj	glCallList
charstr	glCallLists
chunksize	Not needed.
circ	gluDisk
clear	glClear(GL_COLOR_BUFFER_BIT)
clearhitcode	Not supported.
clipplane	glClipPlane
clkon	Use Win32 for keyboard management.
clkoff	Use Win32 for keyboard management.
closeobj	glEndList
cmode	ChoosePixelFormat

cmov	glRasterPos3
cmov2	glRasterPos2
color	glIndex
compactify	Not needed.
concave	gluBeginPolygon
cpack	glColor
crv	Not supported.
crvn	Not supported.
curorigin	Use Win32 for cursors.
cursoff	Use Win32 for cursors.
curson	Use Win32 for cursors.
curstype	Use Win32 for cursors.
curvebasis	glMap1
curveit	glEvalMesh1
curveprecision	Not supported.
cyclemap	Use Win32 for color maps.
czclear	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
dbtext	Not supported.
defbasis	glMap1
defcursor	Use Win32 for cursors.
deflinestyle	glLineStipple
defpattern	glPolygonStipple
defpup	Use Win32 for menus.
defrasterfont	wglUseFontBitmaps
delobj	glDeleteLists
deltag	Not supported.
depthcue	glFog
dglclose	Not needed. (OpenGL is network transparent.)
dglopen	Not needed. (OpenGL is network transparent.)
dither	glEnable(GL_DITHER)
dopup	Use Win32 for menus.
doublebuffer	ChoosePixelFormat
draw	glBegin(GL_LINES)
drawmode	wglMakeCurrent
editobj	Not supported.
endclosedline	glEnd
endcurve	gluEndCurve
endfeedback	glRenderMode(GL_RENDER)
endfullscreen	Not supported.
endline	glEnd
endpick	glRenderMode(GL_RENDER)
endpoint	glEnd

endpolygon	glEnd
endpupmode	Use Win32 for menus.
endqstrip	glEnd
endselect	glRenderMode(GL_RENDER)
endsurface	gluEndSurface
endtmesh	glEnd
endtrim	gluEndTrim
feedback	glFeedbackBuffer
finish	glFinish
fogvertex	glFog
font	See glListBase.
foreground	Use Win32 for windowing.
freepup	Use Win32 for menus.
frontbuffer	glDrawBuffer(GL_FRONT)
frontface	See glCullFace.
fudge	Use Win32 for windowing.
fullscrn	Not supported.
gammaramp	Use Win32 for color maps.
gbegin	Use Win32 for windowing.
gconfig	No equivalent. (Not needed.)
genobj	glGenLists
gentag	Not supported.
getbackface	glGet
getbuffer	glGet
getbutton	Use Win32 for windowing.
getcmmode	wglGetCurrentContext
getcolor	glGet
getcpos	glGet
getcursor	Not supported.
getdcm	glIsEnabled
getdepth	glGet
getdescender	Use Win32 for fonts.
getdev	Not supported.
getdisplaymode	glGet wglGetCurrentContext
getdrawmode	wglGetCurrentContext
getfont	Use Win32 for fonts.
getgdesc	glGet DescribePixelFormat wglGetCurrentContext wglGetCurrentDC
getgpos	Not supported.
getheight	Use Win32 for fonts.
gethitcode	Not supported.
getlsbackup	Not supported.

getlsrepeat	glGet
getlstyle	glGet
getlwidth	glGet
getmap(void)	Not supported.
getmatrix	glGet(GL_MODELVIEW_MATRIX) glGet(GL_PROJECTION_MATRIX)
getmcolor	Not supported.
getmmode	glGet(GL_MATRIX_MODE)
getmonitor	Not supported.
getnurbsproperty	gluGetNurbsProperty
getopenobj	Not supported.
getorigin	Use Win32 for windowing.
getpattern	glGetPolygonStipple
getplanes	glGet(GL_RED_BITS) glGet(GL_GREEN_BITS) glGet(GL_BLUE_BITS)
getport	Use Win32 for windowing.
getresetls	Not supported.
getscrbox	Not supported.
getscrmask	glGet(GL_SCISSOR_BOX)
getshade	glGet(GL_CURRENT_INDEX)
getsize	Use Win32 for windowing.
getsm	glGet(GL_SHADE_MODEL)
getvaluator	Use Win32 for event handling
getvideo	Not supported.
getviewport	glGet(GL_VIEWPORT)
getwritemask	glGet(GL_INDEX_WRITEMASK)
getwscrn	Use Win32 for windowing.
getzbuffer	glIsEnabled(GL_DEPTH_TEST)
gexit	Use Win32 for windowing.
gflush	glFlush
ginit	Use Win32 for windowing.
glcompat	Not supported.
greset	Not supported.
gRGBcolor	glGet(GL_CURRENT_RASTER_COLOR)
gRGBcursor	Use Win32 for cursors.
gRGBmask	glGet(GL_COLOR_WRITEMASK)
gselect	glSelectBuffer
gsync	Use Win32 for windowing.
gversion	glGetString(GL_RENDERER)
iconsize	Use Win32.
icontitle	Use Win32.
imakebackground	Use Win32 for event handling.
initnames	glInitNames
ismex	Not supported.

isobj	glIsList
isqueued	Use Win32 for event handling.
istag	Not supported.
keepaspect	Use Win32 for windowing.
lampoff	Not supported.
lampon	Not supported.
linesmooth	glEnable(GL_LINE_SMOOTH)
linewidth	glLineWidth
linewidthf	glLineWidth
lmbind	glEnable(GL_LIGHTING) glEnable(GL_LIGHT)
lmcolor	glColorMaterial
lmdef	glMaterial glLight glLightModel
loadmatrix	glLoadMatrix
loadname	glLoadName
logicop	glLogicOp
lookat	gluLookAt
lrectread	glReadPixels
lrectwrite(])	glDrawPixels
lRGBrange	Not supported. (See glFog)
lsbackup	Not supported.
lsetdepth	glDepthRange
lshaderange	Not supported. (See glFog)
lsrepeat	glLineStipple
makeobj	glNewList
maketag	Not supported.
mapcolor	Use Win32 for color maps.
mapw	gluProject
maxsize	Use Win32 for windowing.
minsize	Use Win32 for windowing.
mmode	glMatrixMode
move	Not supported.
mswapbuffers	Use Win32 for windowing.
multimap	Use Win32 for color maps.
multmatrix	glMultMatrix
n3f	glNormal3fv
newpup	Use Win32 for Menus.
newtag	Not supported.
nmode	glEnable(GL_NORMALIZE)
noborder	Use Win32 for windowing.
noise	Use Win32 for event handling.
noport	Use Win32 for windowing.
normal	glNormal3fv

nurbscurve	gluNurbsCurve
nurbssurface	gluNurbsSurface
objdelete	Not supported.
objinsert	Not supported.
objreplace	Not supported.
onemap	Use Win32 for color maps.
ortho	glOrtho
ortho2	gluOrtho2D
overlay	Use Win32.
pagecolor	Not supported.
passthrough	glPassThrough
patch	glEvalMesh2
patchbasis	glMap2
patchcurves	glMap2
patchprecision	Not supported.
pclos	Not supported. (See glEnd)
pdr	Not supported. (See glVertex)
perspective	gluPerspective
pick	gluPickMatrix glRenderMode(GL_SELECT)
picksize	gluPickMatrix
pixmode	glPixelTransfer and glPixelStore
pmv	Not supported. (See glBegin and glVertex)
pnt	glBegin(GL_POINTS)
pntsize	glPointSize
pntsizef	glPointSize
pntsmooth	glEnable(GL_POINT_SMOOTH)
polarview	Not supported. (See glRotate and glTranslate)
polf	Not supported.
poly	Not supported.
polymode	glPolygonMode
polysmooth	glEnable(GL_POLYGON_SMOOTH)
popattributes	glPopAttrib
popmatrix	glPopMatrix
popname	glPopName
popviewport	glPopAttrib
prefposition	Use Win32 for windowing.
prefsize	Use Win32 for windowing.
pupmode	Use Win32 for windowing.
pushattributes	glPushAttrib
pushmatrix	glPushMatrix
pushname	glPushName
pushviewport	glPushAttrib(GL_VIEWPORT)
pwlcurve	gluPWLCurve

qcontrol	Use Win32 for event handling.
qdevice	Use Win32 for event handling.
qenter	Use Win32 for event handling.
qgetfd	Use Win32 for event handling.
qread	Use Win32 for event handling.
qreset	Use Win32 for event handling.
qtest	Use Win32 for event handling.
rcrv	Not supported.
rcrvn	Not supported.
rdr	Not supported.
readdisplay	Not supported.
readRGB	Not supported.
readsource	glReadBuffer
rect	See glRect and glPolygonMode
rectf	glRect
rectcopy	glCopyPixels
rectread	glReadPixels
rectwrite	glDrawPixels
rectzoom	glPixelZoom
resetls	Not supported.
reshapeviewport	Not supported.
RGBcolor	glColor
RGBcursor	Use Win32 for cursors.
RGBmode	Use Win32 for windowing.
RGBrange	Not supported.
RGBwritemask	glColorMask
ringbell	Not supported.
rmv	Not supported.
rot	glRotate
rotate	glRotate
rpatch	Not supported.
rpdr	Not supported.
rpmv	Not supported.
sbox	glRect
scale	glScale
sclear	glClear(GL_STENCIL_BUFFER_BIT)
scrbox	Not supported.
screenspace	Not supported.
scrmask	glScissor
scrnattach	Use Win32 for windowing.
scrnselect	Use Win32 for windowing.
scrsubdivide	Not supported.
select	glRenderMode
setbell	Not supported.

setcursor	Use Win32 for cursors.
setdblights	Not supported.
setdepth	glDepthRangea
setlinestyle	glLineStipple
setmap	Use Win32 for color maps.
setmonitor	Not supported.
setnurbsproperty	gluNurbsProperty
setpattern	glPolygonStipple
setpup	Use Win32 for menus.
setvaluator	Use Win32 for devices.
setvideo	Not supported.
shademodel	glShadeModel
shaderange	glFog
singlebuffer	Use Win32 for windowing.
smoothline	glEnable(GL_LINE_SMOOTH)
spclos	Not supported.
splf	Not supported. (See glBegin)
stencil	glStencilFunc glStencilOp
stensize	glStencilMask
stepunit	Use Win32 for windowing.
strwidth	Use Win32 for fonts and strings.
subpixel	Not needed.
swapbuffers	SwapBuffers
swapinterval	Use Win32 for windowing.
swaptmesh	Not supported. (See glBegin(GL_TRIANGLE_FAN))
swinopen	Use Win32 for windowing.
swritemask	glStencilMask
t2	glTexCoord2
tevbind	glTexEnv
tevdef	glTexEnv
texbind	glTexImage2D glTexParameter gluBuild2DMipmaps,
texdef2d	glTexImage2D glTexParameter gluBuild2DMipmaps
texgen	glTexGen
textcolor	Not supported.
textinit	Not supported.
textport	Not supported.
tie	Use Win32 for event handling.
tpoff	Not supported.
tpon	Not supported.
translate	glTranslate

underlay	ChoosePixelFormat
unqdevice	Use Win32 for event handling.
v	glVertex
videocmd	Not supported.
viewport	glViewport
winattach	Use Win32 for windowing.
winclose	wglDeleteContext CloseWindow
winconstraints	Use Win32 for windowing.
windepth	Use Win32 for windowing.
window	glFrustum
winget	wglGetCurrentContext
winmove	Use Win32 for windowing.
winopen	Use Win32 for windowing.
winpop	Use Win32 for windowing.
winposition	Use Win32 for windowing.
winpush	Use Win32 for windowing.
winset	Use Win32 for windowing.
wintitle	Use Win32 for windowing.
wmpack	glColorMask
writemask	glIndexMask
writepixels	glDrawPixels
writeRGB	glDrawPixels
xfpt	Not supported.
zbuffer	glEnable(GL_DEPTH_TEST)
zclear	glClear(GL_DEPTH_BUFFER_BIT)
zdraw	Not supported.
zfunction	glDepthFunc
zsource	Not supported.
zwritemask	glDepthMask

Introduction to Porting to OpenGL for Windows NT

OpenGL is designed for compatibility across hardware and operating systems. This design makes it easier for programmers to port OpenGL programs from one system to another. While each operating system has unique requirements, much of the OpenGL code in your current programs can be used as is. To port your OpenGL application to Windows NT you'll have to modify your programs to work with the Windows NT windowing system.

In general applications are ported to OpenGL for Windows NT from one of two platforms:

- OpenGL applications developed for the X Window System and the X library (Xlib).
- IRIS GL applications.

The following topics describe how to port your applications from each of these platforms. The topics discuss porting OpenGL and window management code only; there is no discussion of other operating system port issues such as reading files, messaging, thread creation, and so on. This porting guide focuses on specific porting issues and assumes that you have an understanding of OpenGL and Windows NT programming.

Porting X Window System Applications

Like Windows NT, the X Window System is an event-handling, message-based system that uses windows controls and menus. The OpenGL code in your X Window System application probably is located in areas that roughly correspond to where it will appear when you port it to Windows NT. Most of your OpenGL code will not change, but you must rewrite any code that is specific to the X Window System. For more information on Win 32 User and GDI calls, refer to the *Win32 Programmer's Reference*. For more information on the X Window System and UNIX, refer to your X Window System and UNIX operating system documentation.

In general, you'll use the following procedure to port your X Window System OpenGL programs to Windows NT.

1. Rewrite the X Window System specific code using equivalent Win32 code. Locate window-creation and event-handling code. The X Window System and Windows NT are both event-handling, message-based windowing systems, which makes it easier to determine where to make the appropriate changes. (However, especially for large applications, rewriting an application from one operating system to another can be a complex and difficult undertaking.)
2. Locate any code that uses GLX functions. These are the functions you'll translate to their corresponding Win32 functions.
3. Translate GLX pixel format functions and Visual/Drawable functions to appropriate Win32/OpenGL pixel format and device context functions.
4. Translate GLX rendering context functions to Win32/OpenGL rendering context functions.
5. Translate GLX Pixmap functions to corresponding Win32 functions.
6. Translate GLX framebuffer and other GLX functions to the appropriate Win32 functions.

Translating the GLX Library

OpenGL X Window System programs use the OpenGL Extension with the X Windows System (GLX) library. The library is a set of functions and routines that initialize the pixel format, control rendering, and do other OpenGL specific tasks. It connects the OpenGL library to the X Window System by managing window handles and rendering contexts. You must translate these functions to their corresponding Windows NT functions. The following table lists the X Window System GLX functions and their corresponding Win32 functions.

GLX/Xlib Functions	Win32 Functions
glXChooseVisual	ChoosePixelFormat
glXCopyContext	---
glXCreateContext	wglCreateContext
glXCreateGLXPixmap	CreateDIBitmap/CreateDIBSection
glXDestroyContext	wglDeleteContext
glXDestroyGLXPixmap	DeleteObject
glXGetConfig	DescribePixelFormat
glXGetCurrentContext	wglGetCurrentContext
glXGetCurrentDrawable	wglGetCurrentDC
glXIsDirect	---
glXMakeCurrent	wglMakeCurrent
glXQueryExtension	GetVersion
glXQueryVersion	GetVersion
glXSwapBuffers	SwapBuffers
glXUseXFont	wglUseFontBitmaps
XGetVisualInfo	GetPixelFormat
XCreateWindow	CreateWindow/CreateWindowEx and GetDC/BeginPaint
XSync	GdiFlush
---	SetPixelFormat

Some GLX functions don't have a corresponding Win32 function. To port these functions to Win32, rewrite your code to achieve the same functionality. For example, **glXWaitGL** has no corresponding Win32 function but you can achieve the same result, executing any pending OpenGL commands, by calling **glFinish**.

The following topics describe how to port GLX functions that set the pixel format, manage rendering contexts, and manage pixmaps and bitmaps.

Porting Device Contexts and Pixel Formats

Each window in Microsoft's implementation of OpenGL for Windows NT has its own current pixel format. A pixel format is defined by a **PIXELFORMATDESCRIPTOR** data structure. Because each window has its own pixel format, you obtain a device context, set the pixel format of the device context, and then create an OpenGL rendering context for the device context. The rendering context automatically uses the pixel format of its device context.

The X Window System also uses a data structure, **XVisualInfo**, to specify the properties of pixels in a window. XVisualInfo structures contain a **Visual** data structure that describes how color resources are used in a specific screen.

In the X Window System, **XVisualInfo** is used to create windows by setting it to the pixel format you want. The returned structure is used to create the window and a rendering context. In Windows NT, you first create a window and get a handle to a device context (HDC) of the window. The HDC is then used to set the pixel format for the window. The rendering context uses the pixel format of the window.

The following table compares the X Window System and GLX visual functions with their corresponding Win32 pixel format functions.

X Window/GLX Visual Functions	Win32 Pixel Format Functions
XVisualInfo* **glXChooseVisual**(Display **dpy*, int *screen*, int **attribList*)	int **ChoosePixelFormat**(HDC *hdc*, PIXELFORMATDESCRIPTOR **ppfd*)
int **glXGetConfig**(Display **dpy*, XVisualInfo **vis*, int **attribList*, int **value*)	int **DescribePixelFormat**(HDC *hdc*, int *iPixelFormat*, UINT *nBytes*, LPPIXELFORMATDESCRIPTOR *ppfd*)
XVisualInfo* **XGetVisualInfo**(Display **dpy*, long *vinfo_mask*, XVisualInfo **vinfo_templ*, int **nitems*)	int **GetPixelFormat**(HDC *hdc*)
The *visual* returned by **glxChooseVisual** is used when a window is created.	BOOL **SetPixelFormat**(HDC *hdc*, int *iPixelFormat*, PIXELFORMATDESCRIPTOR **ppfd*)

The following sections give examples of pixel format code fragments for an X Window System program, and the same code after it has been ported to Windows NT.

For more information on pixel formats, see Pixel Formats.

GLX Pixel Format Code Sample

The code fragment below shows how an X Windows System OpenGL program uses GLX visual/pixel formatting functions.

```
/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};
           .
           .
           .

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }
```

The Visual can be used to create a window and a rendering context.

Win32 Pixel Format Code Sample

The following code fragment shows a function that sets the pixel format using Win32 functions.

```
BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                                PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;
    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
}
```

Porting Rendering Contexts

Both the X Window System and Windows NT render through rendering contexts. Six GLX functions manage rendering contexts and five of them have a corresponding Win32 function.

The following table lists the GLX rendering functions and their corresponding Win32 functions.

GLX Rendering Context Functions	**Win32 Rendering Context Functions**
GLXContext **glXCopyContext**(Display **dpy*, GLXContext *src*, GLXContext *dst*, GLuint *mask*)	No corresponding Win32 function.
GLXContext **glXCreateContext**(Display **dpy*, XVisualInfo **vis*, GLXContext *shareList*, Bool *direct*)	HGLRC **wglCreateContext**(HDC *hdc*)
void **glXDeleteContext** (Display **dpy*, GLXContext *ctx*)	BOOL **wglDeleteContext**(HGLRC *hglrc*)
GLXContext **glXGetCurrentContext** (*void*)	HGLRC **wglGetCurrentContext**(*VOID*)
GLXDrawable **glXGetCurrentDrawable**(*void*)	HDC **wglGetCurrentDC**(*VOID*)
Bool **glXMakeCurrent**(Display **dpy*, GLXDrawable *draw*, GLXContext *ctx*)	BOOL **wglMakeCurrent**(HDC *hdc*, HGLRC *hglrc*)

The X Window System and Windows NT use different names for return types and other types. You can search for occurrences of GLXContext to help find parts of your code that need to be ported.

The following sections compare rendering context code fragment samples in an X Window System program and the same code after it has been ported to Windows NT.

For more information on rendering contexts, see Rendering Contexts.

GLX Rendering Context Code Sample

The code fragment below shows how an X Windows System OpenGL program uses GLX rendering context functions.

```
Display *dpy;                       /* display variable */
XVisualInfo *vi;                    /* visual variable */
Window win;                         /* window variable */
GLXDrawable drawable;         /* drawable variable */
GLXContext cx, cxTemp;              /* rendering context variables */

/* Code to open a display and get a visual. */
           .
           .
           .
/* Create a GLX context. */
cx = glXCreateContext(dpy, vi, 0, GL_FALSE);
if (!cx) {
     fprintf(stderr, "Cannot create context.\n");
     exit(-1);
}
           .
           .
           .
/* Connect the context to the window. */
glXMakeCurrent(dpy, win, cx);
           .
           .
           .
/* When it's time to destroy the rendering context. . . */
cx = glXGetCurrentContext;
glXDestroyContext(dpy, cx);
```

Win32 Rendering Context Code Sample

The following code fragment shows how the GLX rendering context code in the previous section looks when it has been ported to Windows NT using Win32 functions.

```
HGLRC hRC;       // rendering context variable

/* Create and initialize a window */
           .
           .
           .
/* Window message switch in a window procedure */
case WM_CREATE:        // Message when window is created.
{
     HDC hDC, hDCTemp;       // device context handles

     /* Get the handle of the windows device context. */
     hDC = GetDC(hWnd);

     /* Create a rendering context and make it the current context*/
     hRC = wglCreateContext(hDC);
     if (!hRC)
     {
           MessageBox(NULL, "Cannot create context.", "Error", MB_OK);
           return FALSE;
     }
     wglMakeCurrent(hDC, hRC);
}
break;
           .
           .
           .
case WM_DESTROYED:           // Message when window is destroyed.
{
     HGLRC hRC   // rendering context handle
     HDC hDC;          // device context handle

     /* Release and free the device context and rendering context. */
     hDC = wglGetCurrentDC;
     hRC = wglGetCurrentContext;

     wglMakeCurrent(NULL, NULL);

     if (hRC)
           wglDeleteContext(hRC);

     if (hDC)
           ReleaseDC(hWnd, hDC);

     PostQuitMessage (0);
}
break;
```

Porting GLX Pixmap Code

The X Window System uses *pixmaps*, which are off-screen virtual drawing surfaces in the form of a three-dimensional array of bits. You can think of a pixmap as a stack of bitmaps: a two-dimensional array of pixels with each pixel having a value from 0 to 2 (N) -1 where N is the depth of the pixmap.

For OpenGL programs you use the GLX functions **glXCreateGLXPixmap** and **glXDestroyGLXPixmap** to create and destroy GLX pixmaps used for off-screen rendering.

Windows NT uses device-independent bitmaps that serve the same function as X Window System pixmaps. Use the standard Win32 bitmap functions to create and destroy bitmaps.

The following table lists the GLX pixmap functions and their corresponding Win32 bitmap functions.

GLX Pixmap and Font Functions	**Win32 Bitmap and Font Functions**
GLXPixmap **glXCreateGLXPixmap**(Display **dpy*, XVisualInfo **vis*, Pixmap *pixmap*)	HBITMAP **CreateDIBitmap**(HDC *hdc*, LPBITMAPINFOHEADER *lpbmih*, DWORD *fdwInit*, CONST BYTE **lpbInit*, LPBITMAPINFO *lpbmi*, UINT *fuUsage*) HBITMAP **CreateDIBSection**(HDC *hdc*, LPBITMAPINFO *lpbmi*, DWORD *fInit*, DWORD *iUsage*)
void **glXDestroyGLXPixmap**(Display **dpy*, GLXPixmap *pix*)	BOOL **DeleteObject**(HGDIOBJ *hObject*)

Porting Other GLX Code

In addition to the Xlib and GLX functions described in the preceding sections, your program probably contains some of the other GLX or Xlib functions listed in Translating the GLX Library. Rewrite your X Window System code to Windows NT code substituting the appropriate functions.

A Porting Sample

It's easier to understand how to modify your X Window System OpenGL program for a Windows NT program if you can compare before and after samples, and you can better see how the translated code is used in the proper context. This section presents an example of an X Windows System OpenGL program and then shows how the program looks after it has been ported to Windows NT. Note that the OpenGL code is the same in both programs.

An X Windows System OpenGL Program

The following program is an X Windows System OpenGL program with the same OpenGL code used in the AUXEDEMO.C sample program supplied with the Win32 SDK. Compare this program with the Win32 OpenGL program in the next section.

```
/*
 * Example of an X Windows System OpenGL program.
 * OpenGL code is taken from auxdemo.c in the Win32 SDK.
 */
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

/* X globals, defines, and prototypes */
Display *dpy;
Window glwin;
static int attributes[] = {GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER, None};

#define SWAPBUFFERS glXSwapBuffers(dpy, glwin)
#define BLACK_INDEX     0
#define RED_INDEX       1
#define GREEN_INDEX     2
#define BLUE_INDEX      4
#define WIDTH           300
#define HEIGHT          200


/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

static Bool WaitForMapNotify(Display *d, XEvent *e, char *arg)
{
    if ((e->type == MapNotify) && (e->xmap.window == (Window)arg)) {
     return GL_TRUE;
    }
    return GL_FALSE;
}

void
main(int argc, char **argv)
```

```
{
    XVisualInfo    *vi;
    Colormap        cmap;
    XSetWindowAttributes swa;
    GLXContext      cx;
    XEvent          event;
    GLboolean       needRedraw = GL_FALSE, recalcModelView = GL_TRUE;
    int              dummy;

    dpy = XOpenDisplay(NULL);
    if (dpy == NULL){
        fprintf(stderr, "could not open display\n");
        exit(1);
    }

    if(!glXQueryExtension(dpy, &dummy, &dummy)){
        fprintf(stderr, "could not open display");
        exit(1);
    }

    /* find an OpenGL-capable Color Index visual with depth buffer */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributes);
    if (vi == NULL) {
        fprintf(stderr, "could not get visual\n");
        exit(1);
    }

    /* create an OpenGL rendering context */
    cx = glXCreateContext(dpy, vi,  None, GL_TRUE);
    if (cx == NULL) {
        fprintf(stderr, "could not create rendering context\n");
        exit(1);
    }

    /* create an X colormap since probably not using default visual */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                                                vi->visual, AllocNone);
    swa.colormap = cmap;
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | KeyPressMask | StructureNotifyMask;
    glwin = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, WIDTH,
                                    HEIGHT, 0, vi->depth, InputOutput, vi-
>visual,                                  CWBorderPixel | CWColormap |
CWEventMask, &swa);
    XSetStandardProperties(dpy, glwin, "xogl", "xogl", None, argv,
                                                argc, NULL);

    glXMakeCurrent(dpy, glwin, cx);

    XMapWindow(dpy, glwin);
    XIfEvent(dpy,  &event,  WaitForMapNotify,  (char *)glwin);

    initializeGL(WIDTH, HEIGHT);
    resize(WIDTH, HEIGHT);
```

```
    /* Animation loop */
    while (1) {
    KeySym key;

     while (XPending(dpy)) {
         XNextEvent(dpy, &event);
         switch (event.type) {
         case KeyPress:
                XLookupString((XKeyEvent *)&event, NULL, 0, &key, NULL);
           switch (key) {
           case XK_Left:
               longinc += 0.5;
               break;
           case XK_Right:
               longinc -= 0.5;
               break;
           case XK_Up:
               latinc += 0.5;
               break;
           case XK_Down:
               latinc -= 0.5;
               break;
           }
           break;
         case ConfigureNotify:
           resize(event.xconfigure.width, event.xconfigure.height);
           break;
         }
     }
     drawScene();
    }
}

/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
```

```
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
    GLfloatmaxObjectSize, aspect;
    GLdouble     near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );

    glEnable(GL_DEPTH_TEST);

    glMatrixMode( GL_PROJECTION );
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
            GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
```

```
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

     latitude += latinc;
     longitude += longinc;

     polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

     glIndexi(GREEN_INDEX);
     glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();

    SWAPBUFFERS;
}
```

The Program Ported to Win32

The following program is a Win32 OpenGL program with the same OpenGL code used in the AUXDEMO.C sample program supplied with the Win32 SDK. Compare this program with the X Windows System OpenGL program in the above section.

```
/*
 * Example of a Win32 OpenGL program.
 * The OpenGL code is the same as that used in
 * the X Windows System sample.
 */
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

/* Windows globals, defines, and prototypes */
CHAR szAppName[]="Win OpenGL";
HWND  ghWnd;
HDC   ghDC;
HGLRC ghRC;

#define SWAPBUFFERS SwapBuffers(ghDC)
#define BLACK_INDEX     0
#define RED_INDEX       13
#define GREEN_INDEX     14
#define BLUE_INDEX      16
#define WIDTH           300
#define HEIGHT          200

LONG WINAPI MainWndProc (HWND, UINT, WPARAM, LPARAM);
BOOL bSetupPixelFormat(HDC);

/* OpenGL globals, defines, and prototypes */
GLfloat latitude, longitude, latinc, longinc;
GLdouble radius;

#define GLOBE    1
#define CYLINDER 2
#define CONE     3

GLvoid resize(GLsizei, GLsizei);
GLvoid initializeGL(GLsizei, GLsizei);
GLvoid drawScene(GLvoid);
void polarView( GLdouble, GLdouble, GLdouble, GLdouble);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
    MSG        msg;
    WNDCLASS   wndclass;

    /* Register the frame class */
    wndclass.style         = 0;
    wndclass.lpfnWndProc   = (WNDPROC)MainWndProc;
    wndclass.cbClsExtra    = 0;
```

```
    wndclass.cbWndExtra    = 0;
    wndclass.hInstance     = hInstance;
    wndclass.hIcon         = LoadIcon (hInstance, szAppName);
    wndclass.hCursor       = LoadCursor (NULL,IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wndclass.lpszMenuName  = szAppName;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass (&wndclass) )
        return FALSE;

    /* Create the frame */
    ghWnd = CreateWindow (szAppName,
            "Generic OpenGL Sample",
          WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
             CW_USEDEFAULT,
             CW_USEDEFAULT,
             WIDTH,
             HEIGHT,
             NULL,
             NULL,
             hInstance,
             NULL);

    /* make sure window was created */
    if (!ghWnd)
        return FALSE;

    /* show and update main window */
    ShowWindow (ghWnd, nCmdShow);

    UpdateWindow (ghWnd);

    /* animation loop */
    while (1) {
        /*
         *  Process all pending messages
         */

        while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
        {
            if (GetMessage(&msg, NULL, 0, 0) )
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            } else {
                return TRUE;
            }
        }
        drawScene();
    }
}

/* main window procedure */
LONG WINAPI MainWndProc (
```

```
    HWND    hWnd,
    UINT    uMsg,
    WPARAM  wParam,
    LPARAM  lParam)
{
    LONG    lRet = 1;
    PAINTSTRUCT    ps;
    RECT rect;

    switch (uMsg) {

    case WM_CREATE:
        ghDC = GetDC(hWnd);
        if (!bSetupPixelFormat(ghDC))
            PostQuitMessage (0);

        ghRC = wglCreateContext(ghDC);
        wglMakeCurrent(ghDC, ghRC);
        GetClientRect(hWnd, &rect);
        initializeGL(rect.right, rect.bottom);
        break;

    case WM_PAINT:
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;

    case WM_SIZE:
        GetClientRect(hWnd, &rect);
        resize(rect.right, rect.bottom);
        break;

    case WM_CLOSE:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);
        ghRC = 0;
        ghDC = 0;

        DestroyWindow (hWnd);
        break;

    case WM_DESTROY:
        if (ghRC)
            wglDeleteContext(ghRC);
        if (ghDC)
            ReleaseDC(hWnd, ghDC);

        PostQuitMessage (0);
        break;

    case WM_KEYDOWN:
        switch (wParam) {
        case VK_LEFT:
```

```
            longinc += 0.5F;
            break;
        case VK_RIGHT:
            longinc -= 0.5F;
            break;
        case VK_UP:
            latinc += 0.5F;
            break;
        case VK_DOWN:
            latinc -= 0.5F;
            break;
        }

    default:
        lRet = DefWindowProc (hWnd, uMsg, wParam, lParam);
        break;
    }

    return lRet;
}

BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                                PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;
    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
}
```

```
/* OpenGL code */

GLvoid resize( GLsizei width, GLsizei height )
{
    GLfloat aspect;

    glViewport( 0, 0, width, height );

    aspect = (GLfloat) width / height;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );
}

GLvoid createObjects()
{
    GLUquadricObj *quadObj;

    glNewList(GLOBE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_LINE);
        gluSphere (quadObj, 1.5, 16, 16);
    glEndList();

    glNewList(CONE, GL_COMPILE);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder(quadObj, 0.3, 0.0, 0.6, 15, 10);
    glEndList();

    glNewList(CYLINDER, GL_COMPILE);
        glPushMatrix ();
        glRotatef ((GLfloat)90.0, (GLfloat)1.0, (GLfloat)0.0, (GLfloat)0.0);
        glTranslatef ((GLfloat)0.0, (GLfloat)0.0, (GLfloat)-1.0);
        quadObj = gluNewQuadric ();
        gluQuadricDrawStyle (quadObj, GLU_FILL);
        gluQuadricNormals (quadObj, GLU_SMOOTH);
        gluCylinder (quadObj, 0.3, 0.3, 0.6, 12, 2);
        glPopMatrix ();
    glEndList();
}

GLvoid initializeGL(GLsizei width, GLsizei height)
{
     GLfloat     maxObjectSize, aspect;
     GLdouble    near_plane, far_plane;

    glClearIndex( (GLfloat)BLACK_INDEX);
    glClearDepth( 1.0 );

    glEnable(GL_DEPTH_TEST);
```

```
    glMatrixMode( GL_PROJECTION );
    aspect = (GLfloat) width / height;
    gluPerspective( 45.0, aspect, 3.0, 7.0 );
    glMatrixMode( GL_MODELVIEW );

    near_plane = 3.0;
    far_plane = 7.0;
    maxObjectSize = 3.0F;
    radius = near_plane + maxObjectSize/2.0;

    latitude = 0.0F;
    longitude = 0.0F;
    latinc = 6.0F;
    longinc = 2.5F;

    createObjects();
}

void polarView(GLdouble radius, GLdouble twist, GLdouble latitude,
            GLdouble longitude)
{
    glTranslated(0.0, 0.0, -radius);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-latitude, 1.0, 0.0, 0.0);
    glRotated(longitude, 0.0, 0.0, 1.0);

}

GLvoid drawScene(GLvoid)
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glPushMatrix();

     latitude += latinc;
     longitude += longinc;

     polarView( radius, 0, latitude, longitude );

        glIndexi(RED_INDEX);
        glCallList(CONE);

        glIndexi(BLUE_INDEX);
        glCallList(GLOBE);

     glIndexi(GREEN_INDEX);
     glPushMatrix();
            glTranslatef(0.8F, -0.65F, 0.0F);
            glRotatef(30.0F, 1.0F, 0.5F, 1.0F);
            glCallList(CYLINDER);
        glPopMatrix();

    glPopMatrix();
```

```
    SWAPBUFFERS;
}
```

Porting Applications from IRIS GL

This chapter lists important differences between IRIS GL and OpenGL and describes the basic steps for porting code from IRIS GL to OpenGL. For a complete list of the differences between IRIS GL and Open GL, see IRIS GL and OpenGL Differences.

Porting IRIS GL programs to OpenGL for Windows NT requires considerably more work than converting OpenGL programs from the X Window System. While IRIS GL programs are designed to run with specific hardware and software, OpenGL was designed for portability among various systems.

Here are some of the key differences between IRIS GL and OpenGL programs:

OpenGL code	IRIS GL code
Operating system independent; contains no functions for windowing, event handling, buffer allocation/management, and so on.	Dependent on operating system; windowing-system functions are mixed with rendering functions. There is no windows manager in IRIS GL.
Uses a standard, common naming convention. OpenGL functions and defined types begin with a "gl" prefix to prevent conflicts with other libraries.	Does not use a common naming convention for functions and defined types.
Manages state variables (such as color, fog, texture, lighting, and so on) directly and consistently. Does not use tables to load state-variable values.	Uses tables to manage state variables and must bind variables to table values.
Display lists cannot be edited.	Display lists can be edited.
Does not provide a file format for fonts.	Provides functions to handle fonts and text strings and a file format for fonts.
Includes a GL Utility Library (GLU) that contains additional functions and routines (such as NURBS and quadratic rendering routines).	Does not support the GLU library.

{ewl msdncd, EWGraphic, group10424 0 /a "SDK.BMP"} Use the following general procedure to port your IRIS GL programs to OpenGL:

1. Rewrite any code that makes calls to a window manager, window configuration, device, or event, or where you load a color map to equivalent Win32 code. Rewriting an application from one operating system to another can be a complex and difficult undertaking. This subject is beyond the scope of this chapter.
2. Locate any code that uses IRIS GL functions and routines. You'll translate these functions to their corresponding OpenGL functions. For a complete listing of IRIS GL functions and routines and their equivalent OpenGL counterparts, see "IRIS GL Commands and Their OpenGL Equivalents."
3. Change IRIS GL code as described in "Special IRIS GL Porting Issues."

Special IRIS GL Porting Issues

The following sections describe techniques for porting specific parts of your IRIS GL code to OpenGL code.

Porting greset

OpenGL replaces the IRIS GL function **greset** with the functions **glPushAttrib** and **glPopAttrib**. Use these functions to save and restore groups of state variables. For example,

```
void glPushAttrib( GLbitfield mask );
```

This example takes a bitwise OR of symbolic constants, indicating which groups of state variables to push onto an attribute stack. Each constant refers to a group of state variables. The following table shows the attribute groups with their corresponding symbolic constant names. For a complete list of the OpenGL state variables associated with each constant, see **glPushAttrib**.

Attribute	**Constant**
accumulation buffer clear value	GL_ACCUM_BUFFER_BIT
color buffer	GL_COLOR_BUFFER_BIT
current	GL_CURRENT_BIT
depth buffer	GL_DEPTH_BUFFER_BIT
enable	GL_ENABLE_BIT
evaluators	EGL_VAL_BIT
fog	GL_FOG_BIT
GL_LIST_BASE setting	GL_LIST_BIT
hint variables	GL_HINT_BIT
lighting variables	GL_LIGHTING_BIT
line drawing mode	GL_LINE_BIT
pixel mode variables	GL_PIXEL_MODE_BIT
point variables	GL_POINT_BIT
polygon	GL_POLYGON_BIT
polygon stipple	GL_POLYGON_STIPPLE_BIT
scissor	GL_SCISSOR_BIT
stencil buffer	GL_STENCIL_BUFFER_BIT
texture	GL_TEXTURE_BIT
transform	GL_TRANSFORM_BIT
viewport	GL_VIEWPORT_BIT
–	GL_ALL_ATTRIB_BITS

To restore the values of the state variables to those saved with the last **glPushAttrib**, simply call **glPopAttrib**. The variables you didn't save will remain unchanged. The attribute stack has a finite depth of at least 16.

Porting IRIS GL 'Get' Functions

IRIS GL "get" functions take the following form:

```
int getthing();
```

and

```
int getthings( int *a, int *b);
```

Your IRIS GL code probably includes get function calls that look something like:

```
thing = getthing();
if (getthing() == THING) { /* some stuff here */ }
getthings (&a, &b);
```

In OpenGL you use one of the following four types of **glGet** functions in place of corresponding IRIS GL get functions:

- **glGetBooleanv**
- **glGetIntegerv**
- **glGetFloatv**
- **glGetDoublev**

The functions have the following syntax:

```
glGet<Datatype>v( value, *data );
```

where *value* is of type *GLenum* and data is of type *GLdatatype*. When you call **glGet** and it returns a type different from the type expected, the type is converted appropriately. For a complete list of **glGet** parameters, see **glGet**.

Porting Code that Requires a Current Graphics Position

OpenGL does not maintain a current graphics position. IRIS GL functions that depend on the current graphics position, such as **move**, **draw**, and **rmv**, have no equivalents in OpenGL.

Older versions of IRIS GL included drawing commands that relied upon the current graphics position, though their use has been discouraged. You will need to rewrite your code if you relied on the current graphics position in any way, or used any of the following routines:

- **draw** and **move**
- **pmv**, **pdr**, and **pclos**
- **rdr**, **rmv**, **rpdr**, and **rpmv**
- **getgpos**

OpenGL has a concept of raster position that corresponds to IRIS GL's current character position. For more information on raster positioning, see Porting Pixel Operations.

Porting Screen and Buffer Clearing Commands

OpenGL replaces a variety of IRIS GL **clear** functions (such as **zclear**, **aclear**, **sclear**, and so on) with a single function **glClear**. Specify exactly what you want to clear by passing masks to **glClear**.

Keep the following points in mind when porting screen and buffer commands:

- OpenGL maintains clearing colors separately from drawing colors, with calls like **glClearColor** and **glClearIndex**. Be sure to set the clear color for each buffer before doing a clear.
- Instead of using one of several differently named clear calls, you now clear several buffers with one call, **glClear**, by OR-ing together buffer masks. For example, **czclear** is replaced by:

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
```

- IRIS GL references the polygon stipple and the color write mask. OpenGL ignores the polygon stipple but references the write mask. (**czclear** ignores both the polygon stipple and the write mask.)

The table below lists the various IRIS GL clear calls with their OpenGL equivalents.

IRIS GL Call	OpenGL Call	Meaning
acbuf(AC_CLEAR)	glClear(GL_ACCUM_BUFFER_BIT)	Clear the accumulation buffer.
–	glClearColor	Set the RGBA clear color.
–	glClearIndex	Set the clear-color index.
clear	glClear(GL_COLOR_BUFFER_BIT)	Clear the color buffer.
–	glClearDepth	Specify the clear value for the depth buffer.
zclear	glClear(GL_DEPTH_BUFFER_BIT)	Clear the depth buffer.
czclear	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)	Clear the color buffer and the depth buffer.
–	glClearAccum	Specify clear values for the accumulation buffer.
–	glClearStencil	Specify the clear value for the stencil buffer.
sclear	glClear(GL_STENCIL_BUFFER_BIT)	Clear the stencil buffer.

When your IRIS GL code uses both **gclear** and **sclear**, you can combine them into a single **glClear** call—this can improve your program's performance.

Porting Matrix and Transformation Functions

IRIS GL and OpenGL handle matrixes and transformations in a similar manner. But there are several differences to keep in mind when porting code from IRIS GL:

- In OpenGL you are always in double-matrix mode; there is no single-matrix mode.
- Angles are measured in degrees, instead of tenths of degrees.
- Projection matrix calls, like **glFrustum** and **glOrtho**, now multiply onto the current matrix, instead of being loaded onto the current matrix.
- The OpenGL function **glRotate** is very different from **rotate**. You can rotate around any arbitrary axis, instead of being confined to the x, y, and z axes. For example, you can translate:

  ```
  rotate(200*(i+1), 'z');
  ```

  to:

  ```
  glRotate(.1*(200*(i+1), 0.0, 0.0, 1.0);
  ```

  When translating from **rotate** to **glRotate** you switch to degrees from tenths of degrees and replace 'z' with a vector for the z-axis.
- OpenGL has no equivalent to the **polarview** function. You can replace it easily with a translation and three rotations. For example, you can translate:

  ```
  polarview(distance, azimuth, incidence, twist);
  ```

  to:

  ```
  glTranslatef( 0.0, 0.0, -distance);
  glRotatef( -twist * 10.0, 0.0, 0.0, 1.0);
  glRotatef( -incidence * 10.0, 1.0, 0.0, 0.0);
  glRotatef( -azimuth * 10.0, 0.0, 0.0, 1.0);
  ```

The following table lists the OpenGL matrix functions and their IRIS GL equivalents.

IRIS GL Function	OpenGL Function	Meaning
mmode	glMatrixMode	Set current matrix mode.
–	glLoadIdentity	Replace current matrix with the identity matrix.
loadmatrix	glLoadMatrixf, glLoadMatrixd	Replace current matrix with the specified matrix.
multmatrix	glMultMatrixf, glMultMatrixd	Post-multiply current matrix with the specified matrix (note that **multmatrix** pre-multiplied).
mapw, mapw2	gluUnProject	Project world-space coordinates to object space (see also **gluProject**).
ortho	glOrtho	Multiply current matrix by an orthographic projection matrix.
ortho2	gluOrtho2D	Define a two-dimensional orthographic projection matrix.
perspective	gluPerspective	Define a perspective projection matrix.
picksize	gluPickMatrix	Define a picking region.

popmatrix	glPopMatrix	Pop current matrix stack, replacing the current matrix with the one below it.
pushmatrix	glPushMatrix	Push current matrix stack down by one, duplicating the current matrix.
rotate, rot	glRotated, glRotatef	Rotate current coordinate system by the given angle about the vector from the origin through the given point. Note that **rotate** rotated only about the x, y, and z axes.
scale	glScaled, glScalef	Multiply current matrix by a scaling matrix.
translate	glTranslatef, glTranslated	Move coordinate-system origin to the point specified, by multiplying the current matrix by a translation matrix.
window	glFrustum	Given coordinates for clipping planes, multiply the current matrix by a perspective matrix.

OpenGL has three matrix modes, which are set with **glMatrixMode**. The following table lists the modes available as parameters for **glMatrixMode**.

IRIS GL Matrix Mode	OpenGL Mode	Meaning	Min Stack Depth
MTEXTURE	GL_TEXTURE	Operate on the texture matrix stack.	2
MVIEWING	GL_MODELVIEW	Operate on the model view matrix stack.	32
MPROJECTION	GL_PROJECTION	Operate on the projection matrix stack.	2

Porting MSINGLE Mode Code

OpenGL has no equivalent for MSINGLE, single-matrix mode. Though use of this mode has been discouraged, it is the default for IRIS GL. If your IRIS GL program uses the single-matrix mode, you need to rewrite it to use double-matrix mode only. OpenGL is always in double-matrix mode, and is initially in GL_MODELVIEW mode.

Most IRIS GL code in MSINGLE mode looks like this:

```
projectionmatrix();
```

where **projectionmatrix** is one of: **ortho**, **ortho2**, **perspective**, or **window**. To port to OpenGL, replace the MSINGLE-mode **projectionmatrix** function with:

```
glMatrixMode( GL_PROJECTION );
glLoadMatrix( identity matrix );

/* call one of these functions here: */
/* glFrustrum(), glOrtho(), glOrtho2(), gluPerspective()};  */

glMatrixMode( GL_MODELVIEW );
glLoadMatrix( identity matrix );
```

Porting Functions that Get Matrixes and Transformations

The following table lists the IRIS GL functions that get the state of matrixes and transformations and their OpenGL equivalents.

IRIS GL Matrix Query	OpenGL glGet Matrix Query	Meaning
getmmode	GL_MATRIX_MODE	Return the current matrix mode.
getmatrix in MVIEWING mode	GL_MODELVIEW_MATRIX	Return a copy of the current model-view matrix.
getmatrix in MPROJECTION mode	GL_PROJECTION_MATRIX	Return a copy of the current projection matrix.
getmatrix in MTEXTURE mode	GL_TEXTURE_MATRIX	Return a copy of the current texture matrix.
Not applicable.	GL_MAX_MODELVIEW_ STACK_DEPTH	Return maximum supported depth of the model-view matrix stack.
Not applicable.	GL_MAX_PROJECTION_ STACK_DEPTH	Return maximum supported depth of the projection matrix stack.
Not applicable.	GL_MAX_TEXTURE_ STACK_DEPTH	Return maximum supported depth of the texture matrix stack.
Not applicable.	GL_MODELVIEW_ STACK_DEPTH	Returns number of matrices on the model view stack.
Not applicable.	GL_PROJECTION_ STACK_DEPTH	Returns number of matrices on the projection stack.
Not applicable.	GL_TEXTURE_STACK_ DEPTH	Returns number of matrices on the texture stack.

Porting Viewports, Screenmasks, and Scrboxes

The following IRIS GL viewport functions have no OpenGL equivalent:

- **reshapeviewport**
- **scrbox**
- **getscrbox**

With the IRIS GL **viewport** function, you specify the x coordinates (in pixels) for the left and right of a viewport rectangle and the y coordinates for the top and bottom. With the OpenGL **glViewport** function, however, you specify the x and y coordinates (in pixels) of the lower-left corner of the viewport rectangle along with its width and height.

The following table lists IRIS GL viewport functions and their equivalent OpenGL functions.

IRIS GL Call	OpenGL Call	Meaning
viewport(left, right, bottom, top)	glViewport(x, y, width, height)	Set the viewport.
popviewport	glPopAttrib	Push and pop the stack.
pushviewport	glPushAttrib(GL_VIEWPORT_BIT)	
getviewport	glGet(GL_VIEWPORT)	Returns viewport dimensions.

Porting Clipping Planes

OpenGL implements clipping planes similarly to IRIS GL. In addition, in OpenGL you can query clipping planes. The following table lists the OpenGL equivalents to IRIS GL functions.

IRIS GL Call	OpenGL Call	Meaning
clipplane(i, CP_ON, params)	glEnable(GL_CLIP_PLANEi)	Enable clipping on plane i.
clipplane(i, CP_DEFINE, plane)	glClipPlane(GL_CLIP_PLANEi, plane)	Define clipping plane.
–	glGetClipPlane	Returns clipping plane equation.
–	glIsEnabled(GL_CLIP_PLANEi)	Returns true if clip plane i is enabled.
scrmask	glScissor	Defines the scissor box.
getscrmask	glGet(GL_SCISSOR_BOX)	Return the current scissor box.

To turn on the scissor test, call **glEnable** using GL_SCISSOR_BOX as the parameter.

Porting Drawing Functions

The following sections discuss how to port IRIS GL drawing primitives.

The IRIS GL Sphere Library

OpenGL doesn't support the IRIS GL sphere library. You can replace your sphere library calls with quadrics routines from the GLU library. For more information about the GLU library, see the *Open GL Programming Guide* and GLU.

The following table lists the OpenGL quadrics functions.

OpenGL Call	Meaning
gluNewQuadric	Create a new quadric object.
gluDeleteQuadric	Delete a quadric object.
gluQuadricCallback	Associate a callback with a quadric object, for error handling.
gluQuadricNormals	Specify normals: no normals, one per face, or one per vertex.
gluQuadricOrientation	Specify direction of normals: outward or inward.
gluQuadricTexture	Turn texture-coordinate generation on or off.
gluQuadricDrawstyle	Specify drawing style: polygons, lines, points, and so on.
gluSphere	Draw a sphere.
gluCylinder	Draw a cylinder or cone.
gluPartialDisk	Draw an arc.
gluDisk	Draw a circle or disk.

You can use one quadric object for all quadrics you want to render in similar ways. The following code fragment uses two quadric objects to draw four quadrics, two of them textured.

```
GLUquadricObj    *texturedQuad, *plainQuad;

texturedQuad = gluNewQuadric(void);
gluQuadricTexture(texturedQuad, GL_TRUE);
gluQuadricOrientation(texturedQuad, GLU_OUTSIDE);
gluQuadricDrawStyle(texturedQuad, GLU_FILL);

plainQuad = gluNewQuadric(void);
gluQuadricDrawStyle(plainQuad, GLU_LINE);

glColor3f (1.0, 1.0, 1.0);

gluSphere(texturedQuad, 5.0, 20, 20);
glTranslatef(10.0, 10.0, 0.0);
gluCylinder(texturedQuad, 2.5, 5, 5, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluDisk(plainQuad, 2.0, 5.0, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluSphere(plainQuad, 5.0, 20, 20);
```

Porting v Functions

In IRIS GL, you use variations on the **v** function to specify vertices. The equivalent OpenGL function is **glVertex**: Below are examples of **glVertex**.

```
glVertex2[d|f|i|s][v]( x, y );
glVertex3[d|f|i|s][v]( x, y, z);
glVertex4[d|f|i|s][v]( x, y, z, w);
```

glVertex takes suffixes the same way other OpenGL calls do. The vector versions of the call take arrays of the proper size as arguments. In the 2D version, z=0 and w=1. In the 3D version, w=1.

Porting bgn/end Commands

IRIS GL uses the begin/end paradigm but has a different function for each graphics primitive. For example, you probably use **bgnpolygon** and **endpolygon** to draw polygons, and **bgnline** and **endline** to draw lines. In OpenGL, you use the **glBegin**/**glEnd** structure for both. In OpenGL you draw most geometric objects by enclosing a series of functions that specify vertices, normals, textures, and colors between pairs of **glBegin** and **glEnd** calls. For example:

```
void glBegin( GLenum mode) ;
   /* vertex list, colors, normals, textures, materials */
void glEnd( void );
```

glBegin takes a single parameter that specifies the drawing mode, and thus the primitive. Here's an OpenGL code fragment that draws a polygon and then a line:

```
glBegin( GL_POLYGON ) ;
   glVertex2f(20.0, 10.0);
   glVertex2f(10.0, 30.0);
   glVertex2f(20.0, 50.0);
   glVertex2f(40.0, 50.0);
   glVertex2f(50.0, 30.0);
   glVertex2f(40.0, 10.0);
glEnd();
glBegin( GL_LINES ) ;
   glVertex2i(100,100);
   glVertex2i(500,500);
glEnd();
```

With OpenGL, you draw different geometric objects by specifying different parameters for **glBegin**. The following table lists the OpenGL **glBegin** parameters that correspond to equivalent IRIS GL functions.

IRIS GL Call	Value of glBegin Mode	Meaning
bgnpoint	GL_POINTS	Individual points.
bgnline	GL_LINE_STRIP	Series of connected line segments.
bgnclosedline	GL_LINE_LOOP	Series of connected line segments, with a segment added between first and last vertices.
–	GL_LINES	Pairs of vertices interpreted as individual line segments.
bgnpolygon	GL_POLYGON	Boundary of a simple convex polygon.
–	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh	GL_TRIANGLE_STRIP	Linked strips of triangles.
–	GL_TRIANGLE_FAN	Linked fans of triangles.
–	GL_QUADS	Quadruples of vertices interpreted as quadrilaterals.
bgnqstrip	GL_QUAD_STRIP	Linked strips of quadrilaterals.

For a detailed discussion of the differences between triangle meshes, strips, and fans, see "Triangles."

There is no limit to the number of vertices you can specify between a **glBegin**/**glEnd** pair.

In addition to specifying vertices inside a **glBegin**/**glEnd** pair, you can specify a current normal, current texture coordinates, and a current color. The following table lists the commands valid inside a **glBegin**/**glEnd** pair.

IRIS GL Function	OpenGL Equivalent	Meaning
v2, v3, v4	**glVertex**	Set vertex coordinates.
RGBcolor, cpack	**glColor**	Set current color.
color	**glIndex**	Set current color index.
n3f	**glNormal**	Set normal vector coordinates.
–	**glEvalCoord**	Evaluate enabled one- and two-dimensional maps.
callobj	**glCallList**, **glCallLists**	Execute display list(s).
t2	**glTexCoord**	Set texture coordinates.
–	**glEdgeFlag**	Control drawing edges.
lmbind	**glMaterial**	Set material properties.

If you use any OpenGL function other than those listed in the preceding table inside a **glBegin**/**glEnd** pair, you'll get unpredictable results, or possibly an error.

Porting Points

OpenGL has no command to draw a single point. Otherwise, porting point functions is straightforward. The following table lists functions for drawing points.

IRIS GL Call	OpenGL Equivalent	Meaning
pnt	–	Draw a single point.
bgnpoint, endpoint	glBegin(GL_POINTS), glEnd	Interpret vertices as points.
pntsize	glPointSize	Set point size in pixels.
pntsmooth	glEnable(GL_POINT_SMOOTH)	Turn on point antialiasing. (For more information on point antialiasing, see "Porting Antialiasing Calls.")

For information about related get functions, see **glPointSize**.

Porting Lines

Porting IRIS GL code that draws lines is fairly straightforward, though you should note the differences in the way OpenGL stipples. The following table lists functions for drawing lines.

IRIS GL Function	OpenGL Function	Meaning
bgnclosedline, endclosedline	glBegin(GL_LINE_LOOP) glEnd	Draw a closed line.
bgnline	glBegin(GL_LINE_STRIP)	Draw line segments.
linewidth	glLineWidth	Set line width.
getlwidth	glGet(GL_LINE_WIDTH)	Return current line width.
deflinestyle, setlinestyle	glLineStipple(factor, pattern)	Specify a line stipple pattern.
lsrepeat	factor argument of glLineStipple	Set a repeat factor for the line style.
getlstyle	glGet(GL_LINE_STIPPLE_PATTERN)	Return line stipple pattern.
getlsrepeat	glGet(GL_LINE_STIPPLE_REPEAT)	Return repeat factor.
linesmooth, smoothline	glEnable(GL_LINE_SMOOTH)	Turn on line antialiasing (For more information on antialiasing, see "Porting Antialiasing Calls.")

OpenGL doesn't use tables for line stipples; it maintains only one line-stipple pattern. You can use **glPushAttrib** and **glPopAttrib** to switch between different stipple patterns.

Older IRIS GL line style functions (such as **draw**, **lsbackup**, **getlsbackup**, and so on) are not supported by OpenGL.

For information on drawing antialiased lines, see "Porting Antialiasing Calls."

Porting Polygons and Quadrilaterals

- Keep the following points in mind when porting polygons and quadrilaterals:
- There is no direct equivalent for **concave(TRUE)**. Instead you can use the tessellation routines in the GLU, described in "Polygon Tessellation."
- Polygon modes are set differently.
- These polygon drawing functions have no direct equivalents in OpenGL:
  - The **poly** family of routines
  - The **polf** family of routines
  - **pmv**, **pdr**, and **pclos**
  - **rpmv** and **rpdr**
  - **splf**
  - **spclos**

If your IRIS GL code uses these functions, you'll have to rewrite the code using **glBegin(GL_POLYGON)**.

The following table lists the OpenGL equivalents to IRIS GL polygon drawing calls.

IRIS GL Function	OpenGL Equivalent	Meaning
bgnpolygon endpolygon	glBegin(GL_POLYGON) glEnd	Vertices define boundary of a simple convex polygon.
–	glBegin(GL_QUADS) glEnd	Interpret quadruples of vertices as quadrilaterals.
bgnqstrip endqstrip	glBegin(GL_QUAD_STRIP) glEnd	Interpret vertices as linked strips of quadrilaterals.
–	glEdgeFlag	
polymode	glPolygonMode	Set polygon drawing mode.
rect rectf	glRect	Draw a rectangle.
sbox sboxf	–	Draw a screen-aligned rectangle.

Porting Polygon Modes

The OpenGL function **glPolygonMode** lets you specify which side of a polygon (the back or the front) that the mode applies to. Its syntax is:

```
void glPolygonMode( GLenum face, GLenum mode );
```

where face is one of:

GL_FRONT	mode which applies to front-facing polygons
GL_BACK	mode which applies to back-facing polygons
GL_FRONT_AND_BACK	mode which applies to both front- and back-facing polygons

The GL_FRONT_AND_BACK mode is equivalent to the IRIS GL **polymode** function. The following table lists IRIS GL polygon modes and the corresponding OpenGL modes.

IRIS GL Mode	OpenGL Mode	Meaning
PYM_POINT	GL_POINT	Draw vertices as points.
PYM_LINE	GL_LINE	Draw boundary edges as line segments.
PYM_FILL	GL_FILL	Draw polygon interior filled.
PYM_HOLLOW	–	Fill only interior pixels at the boundaries.

Porting Polygon Stipples

When porting IRIS GL polygon stipples, keep the following point in mind:

- OpenGL doesn't use tables for polygon stipples; only one stipple pattern is kept. You can use display lists to store different stipple patterns.
- The OpenGL polygon stipple bitmap size is always a 32x32 bit pattern.
- Stipple encoding is affected by **glPixelStore**.

For more information on porting polygon stipples, see Porting Pixel Operations.

The following table lists IRIS GL polygon stipple functions and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
defpattern	glPolygonStipple	Set the stipple pattern.
setpattern	–	OpenGL keeps only one polygon stipple pattern.
getpattern	glGetPolygonStipple	Return the stipple bitmap (used to return an index).

In OpenGL, you enable and disable polygon stippling by passing GL_POLYGON_STIPPLE as a parameter for **glEnable** and **glDisable**.

Here's an example OpenGL code fragment that demonstrates polygon stippling:

```
void display(void)
{
    GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
```

```
    };
    GLubyte halftone[] = {
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55
    };

    glClear (GL_COLOR_BUFFER_BIT);

/*  draw all polygons in white*/
    glColor3f (1.0, 1.0, 1.0);

/*  draw one solid, unstippled rectangle,*/
/*  then two stippled rectangles*/
    glRectf (25.0, 25.0, 125.0, 125.0);
    glEnable (GL_POLYGON_STIPPLE);
    glPolygonStipple (fly);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glPolygonStipple (halftone);
    glRectf (225.0, 25.0, 325.0, 125.0);
    glDisable (GL_POLYGON_STIPPLE);

    glFlush ();
}
```

Porting Tessellated Polygons

In IRIS GL, you use **concave(TRUE)** and then **bgnpolygon** to draw concave polygons. The OpenGL GLU includes functions you can use to draw concave polygons.

{ewl msdncd, EWGraphic, group10424 1 /a "SDK.BMP"} To draw a concave polygon with OpenGL, follow these steps

1. Create a tessellation object.
2. Define callbacks that will be used to process the triangles generated by the tessellator.
3. Specify the concave polygon to be tessellated.

The following table lists the OpenGL functions for drawing tessellated polygons.

OpenGL GLU Function	Meaning
gluNewTess	Create a new tessellation object.
gluDeleteTess	Delete a tessellation object.

gluTessCallback	–
gluBeginPolygon	Begin the polygon specification.
gluTessVertex	Specify a polygon vertex in a contours.
gluNextContour	Indicate that the next series of vertices describe a new contour.
gluEndPolygon	End the polygon specification.

Porting Triangles

You can draw three types of triangles in OpenGL: separate triangles, triangle strips, and triangle fans.

OpenGL has no equivalent for the IRIS GL **swaptmesh** function. You can achieve the same effect using a combination of triangles, triangle strips, and triangle fans.

The following table lists the IRIS GL functions for drawing triangles and their OpenGL equivalents.

IRIS GL Function	Equivalent glBegin Parameter	Meaning
–	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh endtmesh	GL_TRIANGLE_STRIP	Linked strips of triangles.
–	GL_TRIANGLE_FAN	Linked fans of triangles.

Porting Arcs and Circles

With OpenGL, filled arcs and circles are drawn with the same calls as unfilled arcs and circles. The following table lists the IRIS GL arc and circle functions and their corresponding OpenGL (GLU) functions.

IRIS GL Function	OpenGL Function	Meaning
arc arcf	gluPartialDisk	Draw an arc.
circ circf	gluDisk	Draw a circle or disk.

You can do some things with OpenGL arcs and circles that you can't do with IRIS GL. OpenGL calls arcs and circles, disks and partial disks respectively.

When porting arcs and circles, keep the following points about OpenGL in mind:

- Angles are measured in degrees not in tenths of degrees.
- The start angle is measured from the positive y-axis, and not from the x-axis.
- The sweep angle is now clockwise instead of counterclockwise.

Porting Spheres

When porting spheres to OpenGL, keep the following point in mind:

- You cannot control the type of primitives used to draw the sphere. You can control drawing precision in another way: use the slices and stacks parameters. Slices are longitudinal; stacks are latitudinal.
- Spheres are drawn centered at the origin. Instead of specifying the location, as you do in **sphdraw**, precede a call to **gluSphere** with a translation.
- The sphere library is not yet available for OpenGL.

The following table lists the IRIS GL functions for drawing spheres and their corresponding GLU calls where available.

IRIS GL Function	GLU Function	Meaning
sphobj	**gluNewQuadric**	Create a new sphere object.
sphfree	**gluDeleteQuadric**	Delete sphere object and free memory used.
sphdraw	**gluSphere**	Draw a sphere.
sphmode	–	Set sphere attributes.
sphrotmatrix	–	Control sphere orientation.
sphgnpolys	–	Return number of polygons in current sphere.

Porting Color, Shading, and Writemask Code

When porting color, shading, and writemask code to OpenGL keep the following points in mind:

- Though you can set color-map indices with the OpenGL **glIndex** function, OpenGL does not have a function for loading color-map indices.
- Color values are normalized to their data type. (For information about color values, see **glColor**).
- There is no simple equivalent for **cpack**.
- You may have to translate code that includes the **c** or **color** functions to **glClearColor** or **glClearIndex** instead of **glColor** or **glIndex**.
- The RGBA writemask applies to each component but not for each bit.
- IRIS GL provides defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL does not provide these constants.

Porting Color Calls

The following table lists IRIS GL color functions and their OpenGL equivalents.

IRIS GL Call	OpenGL Call	Meaning
c	glColor	Sets RGB color.
color	glIndex	Sets the color index.
getcolor	glGet(GL_CURRENT_INDEX)	Returns the current color index.
getmcolor	–	Gets a copy of the RGB values for a color map entry.
gRGBcolor	glGet(GL_CURRENT_COLOR)	Gets the current RGB color values.
mapcolor	–	
RGBcolor	glColor	Sets RGB color.
writemask	glIndexMask	Sets the color-index mode color mask.
wmpack RGBwritemask	glColorMask	Sets the RGB color mode mask.
getwritemask	glGet(GL_COLOR_WRITEMASK) **glGet**(GL_INDEX_WRITEMASK)	Gets the color mask.
gRGBmask	**glGet**(GL_COLOR_WRITEMASK)	Gets the color mask.
zwritemask	glDepthMask	–

Note Be careful when replacing **zwritemask** with **glDepthMask**; **glDepthMask** takes a Boolean argument, whereas **zwritemask** takes a bit field.

If you want to use multiple color maps, you need to use the appropriate Win32 colormap functions. Therefore, **multimap**, **onemap**, **getcmmode**, **setmap**, and **getmap** have no OpenGL equivalents.

Porting Shading Models

Like IRIS GL, OpenGL lets you switch between smooth (Gouraud) shading and flat shading. The following table lists the IRIS GL shading and dithering functions and their OpenGL equivalents.

IRIS GL Call	OpenGL Call	Meaning
shademodel(FLAT)	glShadeModel(GL_FLAT)	Do flat shading.
shademodel(GOURAUD)	glShadeModel(GL_SMOOTH)	Do smooth shading.
getsm	glGet(GL_SHADE_MODEL)	Return current shade model.
dither(DT_ON)	glEnable(GL_DITHER)	Turn dithering on/off.
dither(DT_OFF)	glDisable(GL_DITHER)	

Porting Pixel Operations

When porting code that involves pixel operations, keep the following points in mind:

- Logical pixel operations are not applied to RGBA color buffers. For more information, see **glLogicOp**.
- In general, IRIS GL uses the format ABGR for pixels, whereas OpenGL uses RGBA. You can change the format with **glPixelStore**.
- When porting **lrectwrite** functions, be careful to note where **lrectwrite** is writing (for instance, it could be writing to the depth buffer).

OpenGL gives you some additional flexibility in pixel operations. The following table lists IRIS GL functions for pixel operations and their OpenGL equivalents.

IRIS GL Call	OpenGL Call	Meaning
lrectread, rectread, readRGB	glReadPixels	Read a block of pixels from the frame buffer.
lrectwrite, rectwrite	glDrawPixels	Write a block of pixels to the frame buffer.
rectcopy	glCopyPixels	Copy pixels in the frame buffer.
rectzoom	glPixelZoom	Specify pixel zoom factors for glDrawPixels and glCopyPixels.
cmov	glRasterPos	Specify raster position for pixel operations.
readsource	glReadBuffer	Select a color buffer source for pixels.
pixmode	glPixelStore	Set pixel storage modes.
pixmode	glPixelTransfer	Set pixel transfer modes.
logicop	glLogicOp	Specify a logical operation for pixel writes.
–	glEnable(GL_LOGIC_OP)	Turn on pixel logic operations.

For a complete list of possible logical operations, see **glLogicOp**.

This IRIS GL code fragment shows a typical pixel write:

```
unsigned long *packedRaster;
...
packedRaster[k] = 0x00000000;
...
lrectwrite(0, 0, xSize, ySize, packedRaster);
```

The preceding code looks like this when translated to OpenGL:

```
glRasterPos2i( 0, 0);
glDrawPixels( xSize + 1, ySize + 1, GL_RGBA, GL_UNSIGNED_BYTE,
                        packedRaster);
```

Porting Depth Cueing and Fog Commands

When porting depth-cueing and fog code, keep the following points in mind:

- The IRIS GL call **fogvertex** sets a mode and parameters affecting that mode. In OpenGL, you call **glFog** once to set the mode, then again twice or more to set various parameters.
- In OpenGL, depth cueing is not a separate feature. Use linear fog instead of depth cueing. (This section gives an example of how to do this.) The following IRIS GL functions have no direct OpenGL equivalent:
  - **depthcue**
  - **lRGBrange**
  - **lshaderange**
  - **getdcm**
- To adjust fog quality, use **glHint(GL_FOG_HINT)**.

The following table lists the IRIS GL functions for managing fog and their corresponding OpenGL calls.

IRIS GL Call	OpenGL Call	Meaning
fogvertex	glFog	Set various fog parameters.
fogvertex(FG_ON)	glEnable(GL_FOG)	Turn fog on.
fogvertex(FG_OFF)	glDisable(GL_FOG)	Turn fog off.
depthcue	glFog(GL_FOG_MOD, GL_LINEAR)	Use linear fog for depth cueing.

The following table lists the arguments you can pass to **glFog**.

Fog Parameter	Meaning	Default
GL_FOG_DENSITY	Fog density.	1.0
GL_FOG_START	Near distance for linear fog.	0.0
GL_FOG_END	Far distance for linear fog.	1.0
GL_FOG_INDEX	Fog color index.	0.0
GL_FOG_COLOR	Fog RGBA color.	(0, 0, 0, 0)
GL_FOG_MODE	Fog mode.	See the table below.

The fog-density parameter of OpenGL differs from the one in IRIS GL. They are related as follows:

- if fogMode = EXP2
  openGLfogDensity = (irisGLfogDensity) (sqrt(- log(1 / 255)))
- if fogMode = EXP
  openGLfogDensity = (irisGLfogDensity) (- log(1 / 255))

where **sqrt** is the square root operation, **log** is the natural logarithm, *irisGLfogDensity* is the IRIS GL fog density, and *openGLfogDensity* is the OpenGL fog density.

To switch between calculating fog in per-pixel mode and per-vertex mode, use **glHint(GL_FOG_HINT, hintMode)**. Two hint modes are available:

- GL_NICEST per-pixel fog calculation
- GL_FASTEST per-vertex fog calculation

The following table lists the IRIS GL fog modes and their OpenGL equivalents.

IRIS GL Fog Mode	OpenGL Fog Mode	Hint Mode	Meaning
FG_VTX_EXP, FG_PIX_EXP	GL_EXP	GL_FASTEST, GL_NICEST	Heavy fog mode (default).
FG_VTX_EXP2, FG_PIX_EXP2	GL_EXP2	GL_FASTEST, GL_NICEST	Haze mode.
FG_VTX_LIN, FG_PIX_LIN	GL_LINEAR	GL_FASTEST, GL_NICEST	Linear fog mode. (Use for depth cueing).

Here's an example that demonstrates depth cueing in OpenGL:

```
/*
 *  depthcue.c
 *  This program draws a wire frame model, which uses
 *  intensity (brightness) to give clues to distance.
 *  Fog is used to achieve this effect.
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

/*  Initialize linear fog for depth cueing.
 */
void myinit(void)
{
    GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

    glEnable(GL_FOG);
    glFogi (GL_FOG_MODE, GL_LINEAR);
    glHint (GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
    glFogf (GL_FOG_START, 3.0);
    glFogf (GL_FOG_END, 5.0);
    glFogfv (GL_FOG_COLOR, fogColor);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

/*  display() draws an icosahedron.
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    auxWireIcosahedron(1.0);
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
```

```
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef (0.0, 0.0, -4.0); /*move object into view*/
}
/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Curve and Surface Functions

OpenGL doesn't support equivalents to the IRIS GL curves and surface patches. You'll need to rewrite your code if it includes any of the following calls:

- **defbasis**
- **curvebasis**, **curveprecision**, **crv**, c**rvn**, **rcrv**, **rcrvn**, and **curveit**
- **patchbasis**, **patchcurves**, **patchprecision**, **patch**, and **rpatch**

Porting NURBS Objects

OpenGL treats NURBS as objects, similar to the way it treats quadrics: you create a NURBS object and then specify how it should be rendered. The following table lists the OpenGL GLU functions for managing NURBS objects.

OpenGL Call	Meaning
gluNewNurbsRenderer	Create a new NURBS object.
gluDeleteNurbsRenderer	Delete a NURBS object.
gluNurbsCallback	Associate a callback with a NURBS object, for error handling.

When porting IRIS GL NURBS code to OpenGL, keep the following points in mind:

- NURBS control points are floats, not doubles.
- The stride parameter is counted in floats, not bytes.
- If you're using lighting and you're not specifying normals, call **glEnable** with GL_AUTO_NORMAL as the parameter to generate normals automatically.

Porting NURBS Curves

The OpenGL calls for drawing NURBS are very similar to the IRIS GL calls. You specify knot sequences and control points using a **gluNurbsCurve** call, which must be contained within a **gluBeginCurve**/**gluEndCurve** pair.

The following table lists the IRIS GL functions for drawing NURBS curves and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
bgncurve	gluBeginCurve	Begin a curve definition.
nurbscurve	gluNurbsCurve	Specify curve attributes.
endcurve	gluEndCurve	End a curve definition.

Associate position, texture, and color coordinates by presenting each as a separate **gluNurbsCurve** inside the begin/end pair. You can make no more than one call to **gluNurbsCurve** for each piece of color, position, and texture data within a single **gluBeginCurve**/**gluEndCurve** pair. You must make exactly one call to describe the position of the curve (a GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 description). When you call **gluEndCurve**, the curve is tessellated into line segments and then rendered.

The following table lists IRIS GL and OpenGL NURBS curve types.

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP1_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP1_VERTEX_4	Rational curve.
–	GL_MAP1_TEXTURE_COORD_*	Control points are texture coordinates.
–	GL_MAP1_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap1**.

Porting Trimming Curves

OpenGL trimming curves are very similar to IRIS GL trimming curves. The following table lists the IRIS GL functions for defining trimming curves and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
bgntrim	gluBeginTrim	Begin trimming-curve definition.
pwlcurve	gluPwlCurve	Define a piecewise linear curve.
nurbscurve	gluNurbsCurve	Specify trimming-curve attributes.
endtrim	gluEndTrim	End trimming-curve definition.

Porting NURBS Surfaces

The following table lists the IRIS GL functions for drawing NURBS surfaces and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
bgnsurface	gluBeginSurface	Begin a surface definition.
nurbssurface	gluNurbsSurface	Specify surface attributes.
endsurface	gluEndSurface	End a surface definition.

The following table lists IRIS GL parameters for surface types and their OpenGL equivalents.

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP2_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP2_VERTEX_4	Rational curve.
N_C4D	GL_MAP2_COLOR_4	Control points define color surface in (R,G,B,A) form.
N_C4DR	–	–
N_T2D	GL_MAP2_TEXTURE_COORD_2	Control points are texture coordinates.
N_T2DR	GL_MAP2_TEXTURE_COORD_3	Control points are texture coordinates.
–	GL_MAP2_NORMAL	Control points are normals.

For more information on available evaluator types, see **glMap2**.

This sample program draws a trimmed NURBS surface:

```
/*
 *  trim.c
 *  This program draws a NURBS surface in the shape of a
 *  symmetrical hill, using both a NURBS curve and pwl
 *  (piecewise linear) curve to trim part of the surface.
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

GLfloat ctlpoints[4][4][3];

GLUnurbsObj *theNurb;

/*
 *  Initializes the control points of the surface to
 *  a small hill. The control points range from -3 to
 *  +3 in x, y, and z
 */
void init_surface(void)
{
    int u, v;
    for (u = 0; u < 4; u++) {
```

```
        for (v = 0; v < 4; v++) {
            ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
            ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);

            if ( (u == 1 || u == 2) && (v == 1 || v == 2))
                ctlpoints[u][v][2] = 3.0;
            else
                ctlpoints[u][v][2] = -3.0;
        }
    }
}

/*  Initialize material property and depth buffer.
 */
void myinit(void)
{
    GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat mat_shininess[] = { 128.0 };

    glClearColor (0.0, 0.0, 0.0, 1.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 50.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat edgePt[5][2] = /* counter clockwise */
    {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
     {0.0, 0.0}};
    GLfloat curvePt[4][2] = /* clockwise */
    {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
    GLfloat curveKnots[8] =
        {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat pwlPt[4][2] = /* clockwise */
        {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(330.0, 1.,0.,0.);
```

```
    glScalef (0.5, 0.5, 0.5);

    gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
            8, knots,
            8, knots,
            4 * 3,
            3,
            &ctlpoints[0][0][0],
            4, 4,
            GL_MAP2_VERTEX_3);
    gluBeginTrim (theNurb);
        gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluBeginTrim (theNurb);
        gluNurbsCurve (theNurb, 8, curveKnots, 2,
                &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
        gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
                     GLU_MAP1_TRIM_2);
    gluEndTrim (theNurb);
    gluEndSurface(theNurb);

    glPopMatrix();
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -5.0);
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Antialiasing Calls

In OpenGL the subpixel mode is always on, consequently the IRIS GL call **subpixel (TRUE)** is not necessary and has no OpenGL equivalent. The following sections describe aspects of porting IRIS GL antialiasing code.

Porting Blending Code

In IRIS GL, when drawing to both front and back buffers, blending is done by reading one of the buffers, blending with that color, and then writing the result to both buffers. In OpenGL, however, each buffer is read in turn, blended, and then written.

The following table lists IRIS GL blending functions and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
–	glEnable(GL_BLEND)	Turn on blending.
blendfunction	glBlendFunc	Specify a blend function.

The OpenGL **glBlendFunc** function and the IRIS GL **blendfunction** function are almost identical. The following table lists the OpenGL equivalents to the IRIS GL blend factors.

IRIS GL	OpenGL	Notes
BF_ZERO	GL_ZERO	
BF_ONE	GL_ONE	
BF_SA	GL_SRC_ALPHA	
BF_MSA	GL_ONE_MINUS_SRC_ALPHA	
BF_DA	GL_DST_ALPHA	
BF_MDA	GL_ONE_MINUS_DST_ALPHA	
BF_SC	GL_SRC_COLOR	
BF_MSC	GL_ONE_MINUS_SRC_COLOR	Destination only.
BF_DC	GL_DST_COLOR	Source only.
BF_MDC	GL_ONE_MINUS_DST_COLOR	Source only.
BF_MIN_SA_MDA	GL_SRC_ALPHA_SATURATE	

Porting afunction Test Functions

The following table lists the available IRIS GL alpha test functions and their OpenGL equivalents.

afunction	glAlphaFunc
AF_NOTEQUAL	GL_NOTEQUAL
AF_ALWAYS	GL_ALWAYS
AF_NEVER	GL_NEVER
AF_LESS	GL_LESS
AF_EQUAL	GL_EQUAL
AF_LEQUAL	GL_LEQUAL
AF_GREATER	GL_GREATER
AF_GEQUAL	GL_GEQUAL

Antialiasing

OpenGL has direct equivalents to IRIS GL's antialiasing calls; the following table lists them.

IRIS GL Call	OpenGL Call	Meaning
pntsmooth	glEnable(GL_POINT_SMOOTH)	Enable antialiasing of points.
linesmooth	glEnable(GL_LINE_SMOOTH)	Enable antialiasing of lines.
polysmooth	glEnable(GL_POLYGON_SMOOTH)	Enable antialiasing of polygons.

Use the corresponding **glDisable** calls to turn off antialiasing.

In IRIS GL, you can control the quality of the antialiasing by calling:

```
linesmooth(SML_ON + SML_SMOOTHER);
```

OpenGL provides similar control—use **glHint**:

```
glHint(GL_POINT_SMOOTH_HINT, hintMode);
glHint(GL_LINE_SMOOTH_HINT, hintMode);
glHint(GL_POLYGON_SMOOTH_HINT, hintMode);
```

where *hintMode* is one of the following:

- GL_NICEST (Use the highest quality smoothing.)
- GL_FASTEST (Use the most efficient smoothing.)
- GL_DONT_CARE

IRIS GL also permits end-correction by calling:

```
linesmooth(SML_ON +  SML_END_CORRECT);
```

OpenGL has no equivalent for this function.

Porting Accumulation Buffer Calls

You must allocate your accumulation buffer by requesting the appropriate pixel format with **auxInitDisplayMode** or **ChoosePixelFormat**. The following table lists IRIS GL functions that affect the accumulation buffer and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
acsize	**auxInitDisplayMode** or **ChoosePixelFormat**	Specify number of bitplanes per color component in the accumulation buffer.
acbuf	**glAccum**	Operate on the accumulation buffer.
–	**glClearAccum**	Set clear values for accumulation buffer.
acbuf(AC_CLEAR)	**glClear**(GL_ACCUM_BUFFER_BIT)	Clear the accumulation buffer.

The following table lists IRIS GL's **acbuf** parameters along with the corresponding parameters to OpenGL's **glAccum**.

IRIS GL Parameter	OpenGL Parameter
AC_ACCUMULATE	GL_ACCUM
AC_CLEAR_ACCUMULATE	GL_LOAD
AC_RETURN	GL_RETURN
AC_MULT	GL_MULT
AC_ADD	GL_ADD

Porting Stencil Plane Calls

In OpenGL, you allocate stencil planes by requesting the appropriate pixel format with **auxInitDisplayMode** or **ChoosePixelFormat**. The following table lists IRIS GL functions that affect the stencil planes and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
stensize	**ChoosePixelFormat**	–
stencil(TRUE, ...)	glEnable(GL_STENCIL_TEST)	Enable stencil tests.
stencil	glStencilOp	Set stencil test actions.
stencil(... func, ...)	glStencilFunc	Set function and reference value for stencil testing.
swritemask	glStencilMask	Specify which stencil bits can be written.
–	glClearStencil	Specify the clear value for the stencil buffer.
sclear	glClear(GL_STENCIL_BUFFER_BIT)	–

Stencil-comparison functions and stencil pass/fail operations are nearly equivalent in OpenGL and IRIS GL. The IRIS GL stencil-function flags are prefaced with SF, the OpenGL flags with GL. IRIS GL pass/fail operation flags are prefaced with ST, the OpenGL flags with GL.

Porting Display Lists

The OpenGL implementation of display lists is similar to the IRIS GL implementation, with two exceptions: in OpenGL you can't edit display lists once you've created them and you can't call functions from within display lists.

Because you can't edit or call functions from within display lists, these IRIS GL functions have no equivalent in OpenGL:

- **editobj**
- **objdelete**, **objinsert**, and **objreplace**
- **maketag**, **gentag**, **istag**, and **deltag**
- **callfunc**

In IRIS GL, you use the **makeobj** and **closeobj** functions to create display lists. In OpenGL, you use **glNewList** and **glEndList**.

The following table lists the IRIS GL display list commands with the corresponding OpenGL commands.

IRIS GL Function	OpenGL Function	Meaning
makeobj	glNewList	Create a new display list.
closeobj	glEndList	Signal end of display list.
callobj	glCallList, glCallLists	Execute display list(s).
isobj	glIsList	Test for display list existence.
delobj	glDeleteLists	Delete contiguous group of display lists.
genobj	glGenLists	Generate the given number of contiguous empty display lists.

Porting the bbox2 Function

There is no OpenGL equivalent for the IRIS GL **bbox2** function.

{ewl msdncd, EWGraphic, group10424 2 /a "SDK.BMP"} To port code that contains bbox2 functions

1. Create a new (OpenGL) display list that contains everything in the corresponding IRIS GL display list except the call to **bbox2**.
2. Use appropriate Win32 code to draw a rectangle the same size as the IRIS GL bbox.

Porting Edited Display Lists

Although you can't edit OpenGL display lists, you can get similar results by nesting display lists and then destroying and creating new versions of the sublists. For example:

```
glNewList (1, GL_COMPILE);
     glIndexi(MY_RED);
glEndlist();

glNewList(2, GL_COMPILE);
     glScalef(1.2, 1.2, 1.0);
glEndList();

glNewList(3, GL_COMPILE);
     glCallList(1);
     glCallList(2);
glEndList();
     .
     .
     .
glDeleteLists(1, 2);
glNewList(1, GL_COMPILE);
     glIndexi(MY_CYAN);
glEndList();
glNewList(2, GL_COPILE);
     glScalef(0.5, 0.5, 1.0);
glEndList;
```

A Sample Port of a Display Lists

This topic gives an IRIS GL sample of code that defines three display lists; one of the display lists refers to the others in its definition. Following the IRIS GL sample is a sample of what the code looks like when translated to OpenGL.

IRIS GL Sample Display List Code

```
makeobj(10);  // 10 object.
     cpack(0x0000FF);
     recti(164, 33, 364, 600);  // Hollow rectangle.
closeobj();

makeobj(20);  // 20 object.
     cpack(0xFFFF00);
     circle(0, 0, 25); // Unfilled circle.
     recti(100, 100, 200, 200);  // Filled rectangle.
closeobj();

makeobj(30); // 30 object.
     callobj(10);
     cpack(0xFFFFFF);
     recti(400, 100, 500, 300); // Draw filled rectangle.
     callobj(20);
closeobj();

// Now draw by calling the lists.
call(30);
```

OpenGL Sample Display List Code

Here is the preceding IRIS GL code translated to OpenGL:

```
glNewList(10, GL_COMPILE);         // List #10.
     glColor3f(1, 0, 0);
     glRecti(164, 33, 364, 600);
glEndList();

glNewList(20, GL_COMPILE);         //List #20.
     glColor3f(1, 1, 0);           // Set color to YELLOW.
     glPolygonMode(GL_BOTH, GL_LINE);  // Unfilled mode.
     glBegin(GL_POLYGON);  // Use polygon to approximate a circle.
           for(i=0; i<100; i++) {
                 cosine = 25 * cos(i * 2 * PI/100.0);
                 sine =   25 * sin(i * 2 * PI/100.0);
                 glVertex2f(cosine, sine);
           }
     glEnd();
     glBegin(GL_QUADS);
           glColorf(0, 1, 1);  // Set color ro CYAN.
           glVertex2i(100, 100);
           glVertex2i(100, 200);
           glVertex2i(200, 200);
           glVertex2i(100, 200);
     glEnd();
glEndList();
```

```
glNewList(30, GL_COMPILE);  // List #30.
     glCallList(10);
           glColorf(1, 1, 1);      // Set color to WHITE.
           glRecti(400, 100, 500, 300);
     glCallList(20);
glEndList();

// Execute the display lists.
glCallList(30);
```

Porting Defs, Binds, and Sets

OpenGL doesn't have tables of stored definitions; you can't define lighting models, material, textures, line styles, or patterns as separate objects as you can in IRIS GL. Thus OpenGL has no direct equivalents to the following IRIS GL functions:

- **lmdef** and **lmbind**
- **tevdef** and **tevbind**
- **textdef** and **textbind**
- **definestyle** and **setstyle**
- **defpattern** and **setpattern**

You can use OpenGL display lists to mimic the IRIS GL def/bind mechanism. For example, here is a material definition in IRIS GL:

```
float mat() = {
     AMBIENT, .1, .1, .1,
     DIFFUSE, 0, .369, .165,
     SPECULAR, .5, .5, .5,
     SHININESS, 10,
     LMNULL
};
lmdef(DEFMATERIAL, 1, 0, mat);
lmbind(MATERIAL, 1);
```

The following OpenGL code fragment defines the same material in a display list that is referred to by the list number defined by MYMATERIAL.

```
#define MYMATERIAL 10

GLfloat          mat_amb[] = {.1, .1, .1, 1.0};
GLfloat          mat_dif[] = {0, .369, .165, 1.0};
GLfloat          mat_spec[] = {.5, .5, .5, 1.0};

glNewList(MYMATERIAL, GL_COMPILE);
     glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
     glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_dif);
     glMaterialfv(GL_FRONT, GL_SHININESS, 10);
glEndList();

glCallList( MYMATERIAL );
```

Porting Lighting and Materials Functions

OpenGL functions for lighting and materials differ substantially from the IRIS GL functions. Unlike IRIS GL, OpenGL has separate functions for setting lights, light models and materials.

Keep the following points in mind when porting lighting and materials functions:

- OpenGL has no table of stored definitions. You can use display lists to mimic the IRIS GL def/bind mechanism. For more information on defs and binds, see "Porting Defs, Binds, and Sets."
- With OpenGL attenuation is associated with each light source, rather than the overall lighting model.
- Diffuse and specular components are separated in OpenGL light sources.
- OpenGL light sources have an alpha component. When porting your IRIS GL code set this alpha component to 1.0, indicating 100% opaque. The alpha values are then determined by the alpha component of your materials only, so the objects in your scene will look the same as they did in IRIS GL.

The following table lists IRIS GL lighting and materials functions and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
lmdef(DEFLIGHT, ...)	glLight	Define a light source.
lmdef(DEFMODEL, ...)	glLightModel	Define a lighting model.
lmbind	glEnable(GL_LIGHT*i*)	Enable light *i*.
lmbind	**glEnable**(GL_LIGHTING)	Enable lighting.
lmdef(DEFMATERIAL, ...)	glMaterial	Define a material.
lmcolor	glColorMaterial	Change the effect of color commands while lighting is active.
--	glGetMaterial	Get material parameters.

The following table lists various IRIS GL material parameters and their OpenGL equivalents.

lmdef index	glMaterial parameter	Default	Meaning
ALPHA	GL_DIFFUSE	--	The fourth value in the GL_DIFFUSE parameter specifies the alpha value.
AMBIENT	GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color
DIFFUSE	GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color
SPECULAR	GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	Emissive color
SHININESS	GL_SHININESS GL_AMBIENT_AND_DIFFUSE	0.0	Specular exponent Equivalent to calling glMaterial

			twice with the same values.
COLORINDEXES	GL_COLOR_INDEXES	--	Color indices for ambient, diffuse, and specular lighting.

When the first parameter of **lmdef** is DEFLMODEL, the equivalent OpenGL translation is the function **glLightModel**. The exception is when the parameter following DEFMODEL is ATTENUATION: then the equivalent OpenGL function is **glLight**.

The following table lists the equivalent lighting model parameters for IRIS GL and OpenGL.

lmdef Index	glLightModel parameter	Default	Meaning
AMBIENT	GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of scene.
ATTENUATION			See **glLight**.
LOCALVIEWER	GL_LIGHT_MODEL_LOCAL_VIEWER	GL_FALSE	Viewer local (TRUE) or infinite (FALSE).
TWOSIDE	GL_LIGHTMODEL_TWO_SIDE	GL_FALSE	Use two-sided lighting when TRUE.

When the first parameter of **lmdef** is DEFLIGHT, the equivalent OpenGL translation is the function **glLight**.

The following table lists the equivalent light parameters for IRIS GL and OpenGL.

lmdef Index	glLight Parameter	Default	Meaning
AMBIENT	GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	Ambient intensity.
	GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	Diffuse intensity.
	GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	Specular intensity.
LCOLOR	No equivalent.	--	--
POSITION	GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Position of light.
SPOTDIRECTION	GL_SPOT_DIRECTION	(0, 0, -1)	Direction of spotlight.
SPOTLIGHT	GL_SPOT_EXPONENT	0	Intensity distribution.
	GL_SPOT_CUTOFF	180	Maximum spread angle of light source.
DEFLMODE, ATTENUATION	GL_CONSTANT_ATTENUATION	(1, 0, 0)	Attenuation factors.
	GL_LINEAR_ATTENUATION		

GL_QUADRATIC_
ATTENUATION

Porting Texture Functions

When porting IRIS GL texture functions to OpenGL, keep the following points in mind:

- OpenGL doesn't maintain tables of textures; it uses either 1D texture and 2D texture only. To reuse the textures from your IRIS GL code, put them in a display list.
- OpenGL doesn't automatically generate mipmaps. If you're using mipmaps, you must first call **gluBuild2DMipmaps**.
- In OpenGL, you use **glEnable** and **glDisable** to turn texturing capabilities on and off.
- In OpenGL, texture size is more strictly regulated than with IRIS GL. The size of an OpenGL texture must be:

  $2^{(n)}+2b$

  where (n) is an integer and b is:
  - 0, if the texture has no border
  - 1, if the texture has a border pixel (OpenGL textures can have 1-pixel borders.)

The following table lists IRIS GL texture functions and their general OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
textdef2d	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**	Specify a 2D texture image.
textbind	**glTexImage2D** **glTexParameter** **gluBuild2DMipmaps**	Select a texture function.
tevdef	**glTexEnv**	Define a texture-mapping environment.
tevbind	**glTexEnv** **glTexImage1D**	Select a texture environment.
t2	**glTexCoord**	Set the current texture coordinates.
texgen	**glTexGen** **glGetTexParameter** **gluBuild1DMipmaps** **gluBuild2DMipmaps** **gluScaleImage**	Control generation of texture coordinates. Scale an image to an arbitrary size.

For more information on texturing, see the *OpenGL Programming Guide*.

Translating tevdef

Here is an example of an IRIS GL texture-environment definition that specifies the TV_DECAL texture-environment option:

```
float tevprops[] = {TV_DECAL, TV_NULL};

tevdef(1, 0, tevprops);
```

and the same code translated to OpenGL:

glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

The following table lists the IRIS GL texture-environment options and their OpenGL equivalents.

IRIS GL Option	OpenGL Option
TV_MODULATE	GL_MODULATE
TV_DECAL	GL_DECAL
TV_BLEND	GL_BLEND
TV_COLOR	GL_TEXTURE_ENV_COLOR
TV_ALPHA	No direct OpenGL equivalent.
TV_COMPONENT_SELECT	No direct OpenGL equivalent.

For more information about texture-environment options, see **glTexEnv**.

Translating texdef

Here is an example of an IRIS GL texture definition:

```
float texprops[] = {    TX_MINFILTER, TX_POINT,
                                      TX_MAGFILTER, TX_POINT,
                                      TX_WRAP_S, TX_REPEAT,
                                      TX_WRAP_T, TX_REPEAT,
                                      TX_NULL };

textdef2d(1, 1, 6, 6, ganite_texture, 7, texprops);
```

In the example above, **texdef** specifies the TX_POINT filter as both the magnification and the minimizing filter, and TX_REPEAT as the wrapping mechanism. It also specifies the texture image: *granite_texture*.

In OpenGL, **glTexImage** specifies the image and **glTexParameter** sets the property. To translate IRIS GL texture definitions, replace the **textdef** function with **glTexImage** and one or more calls to **glTexParameter**.

The preceding IRIS GL code looks like this when translated to OpenGL:

```
GLfloat nearest[] = {GL_NEAREST};
GLfloat repeat = {GL_REPEAT};
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MAGILTER, nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, repeat);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T, nearest);
glTexImage1D( GL_TEXTURE_1D, 0, 1, 6, 0, GL_RGB,
                              GL_UNSIGNED_SHORT, granite_tex);
```

The following table lists the IRIS GL texture parameters and their OpenGL equivalents.

texdef(..., np, ...) Option	**glTexParameter Parameter**
TX_MINFILTER	GL_TEXTURE_MIN_FILTER
TX_MAGFILTER	GL_TEXTURE_MAG_FILTER
TX_WRAP, TX_WRAP_S	GL_TEXTURE_WRAP_S
TX_WRAP, TX_WRAP_T	GL_TEXTURE_WRAP_T
	GL_TEXTURE_BORDER_COLOR

The following table lists the possible values of the IRIS GL texture parameters and their OpenGL equivalents.

IRIS GL Texture Parameter	**OpenGL Texture Parameter**
TX_POINT	GL_NEAREST
TX_BILINEAR	GL_LINEAR
TX_MIPMAP_POINT	GL_NEAREST_MIPMAP_NEAREST
TX_MIPMAP_BILINEAR	GL_LINEAR_MIPMAP_NEAREST
TX_MIPMAP_LINEAR	GL_NEAREST_MIPMAP_LINEAR
TX_TRILINEAR	GL_LINEAR_MIPMAP_LINEAR

Translating texgen

The IRIS GL function **texgen** is translated to **glTexGen** for OpenGL.

With IRIS GL, you call **texgen** twice: once to set the mode and plane equation simultaneously, and once to enable texture-coordinate generation. For example:

```
texgen(TX_S, TG_LINEAR, planeParams);
texgen(TX_S, TG_ON, NULL);
```

With OpenGL, you make three calls: two to **glTexGen** (once to set the mode and once to set the plane equation), and one to **glEnable**. For example, the OpenGL equivalent to the IRIS GL code above is:

```
glTexGen(GL_S, GLTEXTURE_GEN_MODE, modeName);
glTextGen(GL_S, GL_OBJECT_PLANE, planeParameters);
glEnable(GL_TEXTURE_GEN_S);
```

The following table lists the IRIS GL texture-coordinate names and their OpenGL equivalents.

IRIS GL Texture Coordinate	OpenGL Texture Coordinate	glEnable Argument
TX_S	GL_S	GL_TEXTURE_GEN_S
TX_T	GL_T	GL_TEXTURE_GEN_T
TX_R	GL_R	GL_TEXTURE_GEN_R
TX_Q	GL_Q	GL_TEXTURE_GEN_Q

The following table lists the IRIS GL texture-generation mode and plane names and their OpenGL equivalents.

IRIS GL Texture Mode	OpenGL Texture Mode	OpenGL Plane Name
TG_LINEAR	GL_OBJECT_LINEAR	GL_OBJECT_PLANE
TG_CONTOUR	GL_EYE_LINEAR	GL_EYE_PLANE
TG_SPHEREMAP	GL_SPHERE_MAP	

Porting Picking Functions

All IRIS GL picking functions have OpenGL equivalents, with the exception of **clearhitcode**. The following table lists the IRIS GL picking functions and their OpenGL counterparts.

IRIS GL Function	OpenGL Function	Meaning
clearhitcode	Not supported.	Clears global variable and hitcode.
pick, select	glRenderMode(GL_SELECT)	Switch to selection or picking mode.
endpick endselect	glRenderMode(GL_RENDER)	Switch to rendering mode.
picksize	gluPickMatrix glSelectBuffer	Set the return array.
initnames	glInitNames	
pushname	glPushName	
popname	glPopName	
loadname	glLoadName	

For more information on picking, see **gluPickMatrix**.

Porting Feedback Functions

With IRIS GL, the way that feedback is handled differs depending on the computer running IRIS GL. OpenGL standardizes feedback functions so you can rely on consistent feedback among various hardware platforms. The following table lists the IRIS GL feedback functions and their OpenGL equivalents.

IRIS GL Function	OpenGL Function	Meaning
feedback	glRenderMode(GL_FEEDBACK)	Switch to feedback mode.
endfeedback	glRenderMode(GL_RENDER) glFeedbackBuffer	Switch to rendering mode.
passthrough	glPassThrough	Place a token marker in the feedback buffer.

Programming Tips

This section lists some tips and guidelines that you might find useful. Keep in mind that these tips are based on the intentions of the designers of the OpenGL, not on any experience with actual applications and implementations! This section has the following major topics:

- OpenGL Correctness Tips
- OpenGL Performance Tips

OpenGL Correctness Tips

- Do not count on the error behavior of an OpenGL implementation – it might change in a future release of OpenGL. For example, OpenGL 1.0 ignores matrix operations invoked between **glBegin** and **glEnd** commands, but OpenGL 1.1 might not. OpenGL error semantics may change between upward-compatible revisions.
- Use the projection matrix to collapse all geometry to a single plane. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling **glRotate** with an incremental angle. Rather, use **glLoadIdentity** to initialize the given matrix for each frame, then call **glRotate** with the desired complete angle for that frame.
- Count on multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. Otherwise, a different set of fragments might be generated.
- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- Place the near frustum plane as far from the viewpoint as possible to optimize the operation of the depth buffer.
- Call **glFlush** to force all previous OpenGL commands to be executed. Do not count on **glGet** or **glIs** to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't guarantee to complete all pending rendering commands.
- Turn dithering off when rendering predithered images (for example, when **glCopyPixels** is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it's accumulated.
- If exact two-dimensional rasterization is desired, you must carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. The orthographic projection should be specified with integer coordinates, as shown in the following example:

  ```
  gluOrtho2D(0, width, 0, height);
  ```

  where *width* and *height* are the dimensions of the viewport. Given this projection matrix, polygon vertices and pixel image positions should be placed at integer coordinates to rasterize predictably. For example, **glRecti(0, 0, 1, 1)** reliably fills the lower-left pixel of the viewport, and **glRasterPos2i(0, 0)** reliably positions an unzoomed image at the lower left of the viewport. Point vertices, line vertices, and bitmap positions should be placed at half-integer locations, however. For example, a line drawn from ($x_{(1)}$, 0.5) to ($x_{(2)}$, 0.5) will be reliably rendered along the bottom row of pixels int the viewport, and a point drawn at (0.5, 0.5) will reliably fill the same pixel as **glRecti(0, 0, 1, 1)**.

  An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate *x* and *y* by 0.375, as shown in the following code fragment. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

  ```
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity( );
  gluOrtho2D(0, width, 0, height);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity( );
  glTranslatef(0.375, 0.375, 0.0);
  /* render all primitives at integer positions */
  ```

- Avoid using negative *w* vertex coordinates and negative *q* texture coordinates. OpenGL might not clip such coordinates correctly and might make interpolation errors when shading primitives defined by such coordinates.

OpenGL Performance Tips

- Use **glColorMaterial** when only a single material property is being varied rapidly (at each vertex, for example). Use **glMaterial** for infrequent changes, or when more than a single material property is being varied rapidly.
- Use **glLoadIdentity** to initialize a matrix, rather than loading your own copy of the identity matrix.
- Use specific matrix calls such as **glRotate**, **glTranslate**, and **glScale**, rather than composing your own rotation, translation, and scale matrices and calling **glMultMatrix**.
- Use **glPushAttrib** and **glPopAttrib** to save and restore state values. Use query functions only when your application requires the state values for its own computations.
- Use display lists to encapsulate potentially expensive state changes. For example, place all the **glTexImage** calls required to completely specify a texture, and perhaps the associated **glTexParameter**, **glPixelStore**, and **glPixelTransfer** calls as well, into a single display list. Call this display list to select the texture.
- Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.
- Use evaluators even for simple surface tessellations to minimize network bandwidth in client-server environments.
- Provide unit-length normals if possible, and avoid the overhead of GL_NORMALIZE. Avoid using **glScale** when doing lighting because it almost always requies that GL_NORMALIZE be enabled.
- Set **glShadeModel** to GL_FLAT if smooth shading isn't required.
- Use a single **glClear** call per frame if possible. Do not use **glClear** to clear small subregions of the buffers; use it only for complete or near-complete clears.
- Use a single call to **glBegin(GL_TRIANGLES)** to draw multiple independent triangles, rather than calling **glBegin(GL_TRIANGLES)** multiple times, or calling **glBegin(GL_POLYGON)**. Even if only a single triangle is to be drawn, use GL_TRIANGLES rather than GL_POLYGON. Use a single call to **glBegin(GL_QUADS)** in the same manner, rather than calling **glBegin(GL_POLYGON)** repeatedly. Likewise, use a single call to **glBegin(GL_LINES)** to draw multiple independent line segments, rather than calling **glBegin(GL_LINES)** multiple times.
- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat-shaded polygon.
- Be sure to disable expensive rasterization and per-fragment operations when drawing or copying images. OpenGL will apply textures to pixel images if asked to do so.

Legal Notice

The Query Commands

There are four commands for obtaining simple state variables, and one for determining whether a particular state is enabled or disabled.

void **glGetBooleanv**(GLenum *pname*, GLboolean **params*);

void **glGetIntegerv**(GLenum *pname*, GLint **params*);

void **glGetFloatv**(GLenum *pname*, GLfloat **params*);

void **glGetDoublev**(GLenum *pname*, GLdouble **params*);

Obtains Boolean, integer, floating-point, or double-precision state variables. The *pname* argument is a symbolic constant indicating the state variable to return, and *params* is a pointer to an array of the indicated type in which to place the returned data. The possible values for *pname* are listed in **OpenGL State Variables**. A type conversion is performed if necessary to return the desired variable as the requested data type.

GLboolean **glIsEnabled**(GLenum *cap*);

Returns GL_TRUE if the mode specified by *cap* is enabled; otherwise, returns GL_FALSE. The possible values for *cap* are listed in **OpenGL State Variables**.

Other specialized commands return specific state variables. The prototypes for these commands are listed here. To find out when to use these commands, see **OpenGL State Variables**. Also see the *OpenGL Reference Manual*. OpenGL's error handling facility and the **glGetError** command are described in more detail in **Error Handling**.

void **glGetClipPlane**(GLenum *plane*,GLdouble **equation*);

GLenum **glGetError**(void);

void **glGetLight**{if}**v**(GLenum *light*,GLenum *pname*, *TYPE *params*);

void **glGetMap**{ifd}**v**(GLenum *target*,GLenum *query TYPE *v*);

void **glGetMaterial**{if}**v**(GLenum *face*,GLenum *pname*, *TYPE *params*);

void **glGetPixelMap**{if ui us}**v**(GLenum *map*, *TYPE *values*);

void **glGetPolygonStipple**(GLubyte **mask*);

void **glGetPolygonStipple**(GLubyte **mask*);

const GLubyte ***glGetString**(GLenum *name*);

void **glGetTexEnv**{if}**v**(GLenum *target*, GLenum *pname*, *TYPE *params*);

void **glGetTexGen**{ifd}**v**(GLenum *coord,* GLenum *pname*, *TYPE *params*);

void **glGetTexImage**(GLenum *target*, GLint *level*, GLenum *format,* GLenum *type,* GLvoid **pixels*);

void **glGetTexLevelParameter**{if}**v**(GLenum *target*, GLint *level*, GLenum *pname*, *TYPE *params*);

void **glGetTexParameter**{if}**v**(GLenum *target*,GLenum *pname*, *TYPE *params*);

Error Handling

When OpenGL detects an error, it records a current error code. The command that caused the error is ignored, so it has no effect on OpenGL state or on the frame-buffer contents. (If the error recorded was GL_OUT_OF_MEMORY, however, the results of the command are undefined.) Once recorded, the current error code isn't cleared until you call the query command **glGetError**, which returns the current error code.

Distributed implementations of OpenGL may return multiple current error codes, each of which remains set until queried. The **glGetError** function returns GL_NO_ERROR once you've queried all the current error codes, or if there's no error, so if you obtain an error code, it's a good practice to call **glGetError** until GL_NO_ERROR is returned to be sure you've discovered all the errors. See **OpenGL error codes**.

You can use the GLU routine **gluErrorString** to obtain a descriptive string corresponding to the error code passed in. This routine is described in more detail in **Describing Errors**. Notice that GLU routines often return error values if an error is detected. Also, the GLU defines the error codes GLU_INVALID_ENUM, GLU_INVALID_VALUE, and GLU_OUT_OF_MEMORY, which have the same meaning as the related OpenGL codes.

OpenGL Error Codes

Error Code	Description
GL_INVALID_ENUM	GLenum argument out of range
GL_INVALID_VALUE	Numeric argument out of range
GL_INVALID_OPERATION	Operation illegal in current state
GL_STACK_OVERFLOW	Command would cause a stack overflow
GL_STACK_UNDERFLOW	Command would cause a stack underflow
GL_OUT_OF_MEMORY	Not enough memory left to execute command

Saving and Restoring Sets of State Variables

You can save and restore the values of a collection of state variables on an attribute stack with the commands **glPushAttrib** and **glPopAttrib**. The attribute stack has a depth of at least 16, and the actual depth can be obtained using GL_MAX_ATTRIB_STACK_DEPTH with **glGetIntegerv**. Pushing a full stack or popping an empty one generates an error.

In general, it's faster to use **glPushAttrib** and **glPopAttrib** than to get and restore the values yourself. Some values might be pushed and popped in the hardware, and saving and restoring them might be expensive. Also, if you're operating on a remote client, all the attribute data must be transferred across the network connection and back as it's saved and restored. However, your OpenGL implementation keeps the attribute stack on the server, avoiding unnecessary network delays.

void **glPushAttrib**(GLbitfield *mask*);

Saves all the attributes indicated by bits in *mask* by pushing them onto the attribute stack. The following Attribute Groups table lists the possible mask bits than can be logically ORed together to save any combination of attributes. Each bit corresponds to a collection of individual state variables. For example, GL_LIGHTING_BIT refers to all the state variables related to lighting, which include the current material color, the ambient, diffuse, specular, and emitted light, a list of the lights that are enabled, and the directions of the spotlights. When **glPopAttrib** is called, all those variables are restored. To find out exactly which attributes are saved for particular mask values, see OpenGL State Variables.

Attribute Groups

Mask bit	Attribute group
GL_ACCUM_BUFFER_BIT	accum-buffer
GL_ALL_ATTRIB_BITS	–
GL_COLOR_BUFFER_BIT	color-buffer
GL_CURRENT_BIT	current
GL_DEPTH_BUFFER_BIT	depth-buffer
GL_ENABLE_BIT	enable
GL_EVAL_BIT	eval
GL_FOG_BIT	fog
GL_HINT_BIT	hint
GL_LIGHTING_BIT	lighting
GL_LINE_BIT	line
GL_LIST_BIT	list
GL_PIXEL_MODE_BIT	pixel
GL_POINT_BIT	point
GL_POLYGON_BIT	polygon
GL_POLYGON_STIPPLE_BIT	polygon-stipple
GL_SCISSOR_BIT	scissor
GL_STENCIL_BUFFER_BIT	stencil-buffer
GL_TEXTURE_BIT	texture
GL_TRANSFORM_BIT	transform
GL_VIEWPORT_BIT	viewport

void **glPopAttrib**(void);

Restores the values of those state variables that were saved with the last **glPushAttrib**.

OpenGL State Variables

The following topics list the names of queryable state variables:

For each variable, the topic lists a description, attribute group, initial or minimum value, and the suggested **glGet*** command to use for obtaining it. State variables that can be obtained using **glGetBooleanv**, **glGetIntegerv**, **glGetFloatv**, or **glGetDoublev** are listed with just one of these commands – the one that's most appropriate give the type of data to be returned. These state variables can't be obtained using **glIsEnabled**. However, state variables for which **glIsEnabled** is listed as the query command can also be obtained using **glGetBooleanv( )**, **glGetIntegerv**, **glGetFloatv**, and **glGetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command. If no attribute group is listed, the variable doesn't belong to any group. All queryable state variables except the implementation-dependent ones have initial values. If no initial value is listed, consult the topic where that variable is discussed.

Rasterization State Variables

GL_POINT_SIZE

Description:	Point size
Attribute Group:	point
Initial Value:	1.0
Get Command:	glGetFloatv

GL_POINT_SMOOTH

Description:	Point aliasing on
Attribute Group:	point/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_LINE_WIDTH

Description:	Line width
Attribute Group:	line
Initial Value:	1.0
Get Command:	glGetFloatv

GL_LINE_SMOOTH

Description:	Line antialiasing on
Attribute Group:	line/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_LINE_STIPPLE_PATTERN

Description:	Line stipple
Attribute Group:	line
Initial Value:	1's
Get Command:	glGetIntegerv

GL_LINE_STIPPLE_REPEAT

Description:	Line stipple repeat
Attribute Group:	line
Initial Value:	1
Get Command:	glGetIntegerv

GL_LINE_STIPPLE

Description:	Line stipple enable

Attribute Group: line/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_CULL_FACE
Description: Polygon culling enabled
Attribute Group: polygon/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_CULL_FACE_MODE
Description: Cull front-/back-facing polygons
Attribute Group: polygon
Initial Value: GL_BACK
Get Command: glGetIntegerv

GL_FRONT_FACE
Description: Polygon front-face CW/CCW indicator
Attribute Group: polygon
Initial Value: GL_CCW
Get Command: glGetIntegerv

GL_POLYGON_SMOOTH
Description: Polygon antialiasing on
Attribute Group: polygon/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_POLYGON_MODE
Description: Polygon rasterization mode (front and back)
Attribute Group: polygon
Initial Value: GL_FILL
Get Command: glGetIntegerv

GL_POLYGON_STIPPLE
Description: Polygon stipple enable
Attribute Group: polygon/enable
Initial Value: GL_FALSE

Get Command:	glIsEnabled

–

Description:	Polygon stipple pattern
Attribute Group:	polygon-stipple
Initial Value:	1's
Get Command:	glGetPolygon-Stipple

Texturing State Variables

GL_TEXTURE_*x*

Description:	True if *x*-D texturing enabled (*x* is 1D or 2D)
Attribute Group:	texture/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_TEXTURE

Description:	*x*-D texture image at level of detail *i*
Attribute Group:	–
Initial Value:	–
Get Command:	glGetTexImage

GL_TEXTURE_WIDTH

Description:	*x*-D texture image *i*'s width
Attribute Group:	–
Initial Value:	0
Get Command:	glGetTexLevelParameter

GL_TEXTURE_HEIGHT

Description:	*x*-D texture image *i*'s height
Attribute Group:	–
Initial Value:	0
Get Command:	glGetTexLevelParameter

GL_TEXTURE_BORDER

Description:	*x*-D texture image *i*'s border
Attribute Group:	–
Initial Value:	0
Get Command:	glGetTexLevelParameter

GL_TEXTURE_COMPONENTS

Description:	Texture image components
Attribute Group:	–
Initial Value:	1
Get Command:	glGetTexLevelParameter

GL_TEXTURE_BORDER_COLOR

Description:	Texture border color

Attribute Group: texture

Initial Value: 0,0,0,0

Get Command: glGetTexParameter

GL_TEXTURE_MIN_FILTER

Description: Texture minification function

Attribute Group: texture

Initial Value: GL_NEAREST_MIPMAP_LINEAR

Get Command: glGetGetTexParameter

GL_TEXTURE_MAG_FILTER

Description: Texture magnification function

Attribute Group: texture

Initial Value: GL_LINEAR

Get Command: glGetTexParameter

GL_TEXTURE_WRAP_*x*

Description: Texture wrap mode (*x* is S or T)

Attribute Group: texture

Initial Value: GL_REPEAT

Get Command: glGetTexParameter

GL_TEXTURE_ENV_MODE

Description: Texture application function

Attribute Group: texture

Initial Value: GL_MODULATE

Get Command: glGetTexEnviv

GL_TEXTURE_ENV_COLOR

Description: Texture environment color

Attribute Group: texture

Initial Value: 0,0,0,0

Get Command: glGetTexEnvfv

GL_TEXTURE_GEN_*x*

Description: Texgen is enabled (*x* is S, T, R, or Q)

Attribute Group: texture/enable

Initial Value: GL_FALSE

Get Command: glIsEnabled

GL_EYE_LINEAR

Description: Texgen plane equation coefficients

Attribute Group: texture

Initial Value: –

Get Command: glGetTexGenfv

GL_OBJECT_LINEAR

Description: Texgen object linear coefficients

Attribute Group: texture

Initial Value: –

Get Command: glGetTexGenfv

GL_TEXTURE_GEN_MODE

Description: Function used for texgen

Attribute Group: texture

Initial Value: GL_EYTE_LINEAR

Get Command: glGetTexGeniv

Framebuffer Control State Variables

GL_DRAW_BUFFER

Description:	Buffers selected for drawing
Attribute Group:	color-buffer
Initial Value:	–
Get Command:	glGetIntegerv

GL_INDEX_WRITEMASK

Description:	Color-index writemask
Attribute Group:	color-buffer
Initial Value:	1's
Get Command:	glGetIntegerv

GL_COLOR_WRITEMASK

Description:	Color write enables; R, G, B, or A
Attribute Group:	color-buffer
Initial Value:	GL_TRUE
Get Command:	glGetBooleanv

GL_DEPTH_WRITEMASK

Description:	Depth buffer enabled for writing
Attribute Group:	depth-buffer
Initial Value:	GL_TRUE
Get Command:	glGetBooleanv

GL_STENCIL_WRITEMASK

Description:	Stencil-buffer writemask
Attribute Group:	stencil-buffer
Initial Value:	1's
Get Command:	glGetIntegerv

GL_COLOR_CLEAR_VALUE

Description:	Color-buffer clear value (RGBA mode)
Attribute Group:	color-buffer
Initial Value:	0, 0, 0, 0
Get Command:	glGetFloatv

GL_INDEX_CLEAR_VALUE

Description:	Color-buffer clear value (color-index mode)

Attribute Group:	color-buffer
Initial Value:	0
Get Command:	glGetFloatv

GL_DEPTH_CLEAR_VALUE

Description:	Depth-buffer clear value
Attribute Group:	depth-buffer
Initial Value:	1
Get Command:	glGetIntegerv

GL_STENCIL_CLEAR_VALUE

Description:	Stencil-buffer clear value
Attribute Group:	stencil-buffer
Initial Value:	0
Get Command:	glGetIntegerv

GL_ACCUM_CLEAR_VALUE

Description:	Accumulation-buffer clear value
Attribute Group:	accum-buffer
Initial Value:	0
Get Command:	glGetFloatv

Pixel State Variables

GL_UNPACK_SWAP_BYTES

Description: Value of GL_UNPACK_SWAP_BYTES
Attribute Group: –
Initial Value: GL_FALSE
Get Command: glGetBooleanv

GL_UNPACK_LSB_FIRST

Description: Value of GL_UNPACK_LSB_FIRST
Attribute Group: –
Initial Value: GL_FALSE
Get Command: glGetBooleanv

GL_UNPACK_ROW_LENGTH

Description: Value of GL_UNPACK_ROW_LENGTH
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_UNPACK_SKIP_ROWS

Description: Value of GL_UNPACK_SKIP_ROWS
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_UNPACK_SKIP_PIXELS

Description: Value of GL_UNPACK_SKIP_PIXELS
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_UNPACK_ALIGNMENT

Description: Value of GL_UNPACK_ALIGNMENT
Attribute Group: –
Initial Value: 4
Get Command: glGetIntegerv

GL_PACK_SWAP_BYTES

Description: Value of GL_PACK_SWAP_BYTES

Attribute Group: –
Initial Value: GL_FALSE
Get Command: glGetBooleanv

GL_PACK_LSB_FIRST
Description: Value of GL_PACK_LSB_FIRST
Attribute Group: –
Initial Value: GL_FALSE
Get Command: glGetBooleanv

GL_PACK_ROW_LENGTH
Description: Value of GL_PACK_ROW_LENGTH
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_PACK_SKIP_ROWS
Description: Value of GL_PACK_SKIP_ROWS
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_PACK_SKIP_PIXELS
Description: Value of GL_PACK_SKIP_PIXELS
Attribute Group: –
Initial Value: 0
Get Command: glGetIntegerv

GL_PACK_ALIGNMENT
Description: Value of GL_PACK_ALIGNMENT
Attribute Group: –
Initial Value: 4
Get Command: glGetIntegerv

GL_MAP_COLOR
Description: True if colors are mapped
Attribute Group: pixel
Initial Value: GL_FALSE

Get Command: glGetBooleanv

GL_MAP_STENCIL

Description: True if stencil values are mapped

Attribute Group: pixel

Initial Value: GL_FALSE

Get Command: glGetBooleanv

GL_INDEX_SHIFT

Description: Value of GL_INDEX_SHIFT

Attribute Group: pixel

Initial Value: 0

Get Command: glGetIntegerv

GL_INDEX_OFFSET

Description: Value of GL_INDEX_OFFSET

Attribute Group: pixel

Initial Value: 0

Get Command: glGetIntegerv

GL_*x*_SCALE

Description: Value of GL_*x*_SCALE; *x* IS GL_RED GL_BLUE, GL_ALPHA, or GL_DEPTH

Attribute Group: pixel

Initial Value: 1

Get Command: glGetFloatv

GL_*x*_BIAS

Description: Value of GL_*x*_BIAS; *x* IS GL_RED GL_BLUE, GL_ALPHA, or GL_DEPTH

Attribute Group: pixel

Initial Value: 0

Get Command: glGetFloatv

GL_ZOOM_X

Description: *x* zoom factor

Attribute Group: pixel

Initial Value: 1.0

Get glGetFloat

Command:

GL_ZOOM_Y

Description: *y* zoom factor
Attribute Group: pixel
Initial Value: 1.0
Get Command: glGetFloatv

GL_*x*

Description: glPixelMap translation tables
Attribute Group: pixel
Initial Value: 0's
Get Command: glGetPixelMap

GL_*x*_SIZE

Description: Size of table *x*
Attribute Group: pixel
Initial Value: 1
Get Command: glGetIntegerv

GL_READ_BUFFER

Description: Read source buffer
Attribute Group: pixel
Initial Value: −
Get Command: glGetIntegerv

Evaluator State Variables

GL_ORDER

Description:	1D map order
Attribute Group:	–
Initial Value:	1
Get Command:	glGetMapiv

GL_ORDER

Description:	2D map orders
Attribute Group:	–
Initial Value:	1, 1
Get Command:	glGetMapiv

GL_COEFF

Description:	1D control points
Attribute Group:	–
Initial Value:	–
Get Command:	glGetMapfv

GL_COEFF

Description:	2D control points
Attribute Group:	–
Initial Value:	–
Get Command:	glGetMapfv

GL_DOMAIN

Description:	1D domain endpoints
Attribute Group:	–
Initial Value:	–
Get Command:	glGetMapfv

GL_DOMAIN

Description:	2D domain endpoints
Attribute Group:	–
Initial Value:	–
Get Command:	glGetMapfv

GL_MAP1_x

Description:	1D map enables: *x* is map type

Attribute Group: eval/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_MAP2_x
Description: 2D map enables: *x* is map type
Attribute Group: eval/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_MAP1_GRID_DOMAIN
Description: 1D grid endpoints
Attribute Group: eval
Initial Value: 0, 1
Get Command: glGetFloatv

GL_MAP2_GRID_DOMAIN
Description: 2D grid endpoints
Attribute Group: eval
Initial Value: 0, 1; 0, 1
Get Command: glGetFloatv

GL_MAP1_GRID_SEGMENTS
Description: 1D grid divisions
Attribute Group: eval
Initial Value: 1
Get Command: glGetFloatv

GL_MAP1_GRID_SEGMENTS
Description: 2D grid segments
Attribute Group: eval
Initial Value: 1, 1
Get Command: glGetFloatv

GL_AUTO_NORMAL
Description: True if automatic normal generation enabled
Attribute Group: eval
Initial Value: GL_FALSE

Get Command:	glIsEnabled

Hint State Variables

GL_PERSPECTIVE_CORRECTION_HINT

Description:	Perspective correction hint
Attribute Group:	hint
Initial Value:	GL_DON'T CARE
Get Command:	glGetIntegerv

GL_POINT_SMOOTH_HINT

Description:	Point smooth hint
Attribute Group:	hint
Initial Value:	GL_DON'T CARE
Get Command:	glGetIntegerv

GL_LINE_SMOOTH_HINT

Description:	Line smooth hint
Attribute Group:	hint
Initial Value:	GL_DON'T CARE
Get Command:	glGetIntegerv

GL_POLYGON_SMOOTH_HINT

Description:	Polygon smooth hint
Attribute Group:	hint
Initial Value:	GL_DON'T CARE
Get Command:	glGetIntegerv

GL_FOG_HINT

Description:	Fog hint
Attribute Group:	hint
Initial Value:	GL_DON'T CARE
Get Command:	glGetIntegerv

Implementation-Dependent State Variables

GL_MAX_LIGHTS

Description:	Maximum number of lights
Attribute Group:	–
Initial Value:	8
Get Command:	glGetIntegerv

GL_MAX_CLIP_PLANES

Description:	Maximum number of user clipping planes
Attribute Group:	–
Initial Value:	6
Get Command:	glGetIntegerv

GL_MAX_MODELVIEW_STACK_DEPTH

Description:	Maximum modelview-matrix stack depth
Attribute Group:	–
Initial Value:	32
Get Command:	glGetIntegerv

GL_MAX_PROJECTION_STACK_DEPTH

Description:	Maximum projection-matrix stack depth
Attribute Group:	–
Initial Value:	2
Get Command:	glGetIntegerv

GL_MAX_MAX_TEXTURE_STACK_DEPTH

Description:	Maximum depth of texture matrix stack
Attribute Group:	–
Initial Value:	2
Get Command:	glGetIntegerv

GL_SUBPIXEL_BITS

Description:	Number of bits of subpixel precision in *x* and *y*
Attribute Group:	–
Initial Value:	4
Get Command:	glGetIntegerv

GL_MAX_TEXTURE_SIZE

Description:	Maximum height or width of a texture image (w/o

borders)

Attribute Group: –

Initial Value: 64

Get Command: glGetIntegerv

GL_MAX_PIXEL_MAP_TABLE

Description: Maximum size of a glPixelMap translation table

Attribute Group: –

Initial Value: 32

Get Command: glGetIntegerv

GL_MAX_NAME_STACK_DEPTH

Description: Maximum selection-name stack depth

Attribute Group: –

Initial Value: 64

Get Command: glGetIntegerv

GL_MAX_LIST_NESTING

Description: Maximum display-list call nesting

Attribute Group: –

Initial Value: 64

Get Command: glGetIntegerv

GL_MAX_EVAL_ORDER

Description: Maximum evaluator polynomial order

Attribute Group: –

Initial Value: 8

Get Command: glGetIntegerv

GL_MAX_VIEWPORT_DIMS

Description: Maximum viewport dimensions

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_MAX_ATTRIB_STACK_DEPTH

Description: Maximum depth of the attribute stack

Attribute Group: –

Initial Value: 16

Get Command: glGetIntegerv

GL_AUX_BUFFERS

Description: Number of auxiliary buffers

Attribute Group: –

Initial Value: 0

Get Command: glGetBooleanv

GL_RGBA_MODE

Description: True if color buffers store RGBA

Attribute Group: –

Initial Value: –

Get Command: glGetBooleanv

GL_INDEX_MODE

Description: True if color buffers store indices

Attribute Group: –

Initial Value: –

Get Command: glGetBooleanv

GL_DOUBLEBUFFER

Description: True if front and back buffers exist

Attribute Group: –

Initial Value: –

Get Command: glGetBooleanv

GL_STEREO

Description: True if left and right buffers exist

Attribute Group: –

Initial Value: –

Get Command: glGetFloatv

GL_POINT_SIZE_RANGE

Description: Range (low to hight) of antialiased point sizes

Attribute Group: –

Initial Value: 1, 1

Get Command: glGetFloatv

GL_POINT_SIZE_GRANULARITY

Description:	Antialiased point size granularity
Attribute Group:	–
Initial Value:	–
Get Command:	glGetFloatv

GL_LINE_WIDTH_RANGE

Description:	Range (low to high) of antialiased line widths
Attribute Group:	–
Initial Value:	1, 1
Get Command:	glGetFloatv

GL_LINE_WIDTH_GRANULARITY

Description:	Antialiased line-width granularity
Attribute Group:	–
Initial Value:	–
Get Command:	glGetFloatv

Implementation-Dependent Pixel-Depth State Variables

GL_RED_BITS

Description: Number of bits per red component in color buffers

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_GREEN_BITS

Description: Number of bits per green component in color buffers

Attribute Group: –

Get Command: glGetIntegerv

Initial Value: –

GL_BLUE_BITS

Description: Number of bits per blue component in color buffers

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_ALPHA_BITS

Description: Number of bits per alpha component in color buffers

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_INDEX_BITS

Description: Number of bits per index in color buffers

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_DEPTH_BITS

Description: Number of depth-buffer bitplanes

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_STENCIL_BITS

Description: Number of stencil bitplanes

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_ACCUM_RED_BITS

Description: Number of bits per red component in the accumulation buffer

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_ACCUM_GREEN_BITS

Description: Number of bits per green component in the accumulation buffer

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_ACCUM_BLUE_BITS

Description: Number of bits per blue component in the accumulation buffer

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

GL_ACCUM_ALPHA_BITS

Description: Number of bits per alpha component in the accumulation buffer

Attribute Group: –

Initial Value: –

Get Command: glGetIntegerv

Miscellaneous State Variables

GL_LIST_BASE

Description:	Setting of glListBase
Attribute Group:	list
Initial Value:	0
Get Command:	glGetIntegerv

GL_LIST_INDEX

Description:	Number of display list under construction; 0 if none
Attribute Group:	–
Initial Value:	0
Get Command:	glGetIntegerv

GL_LIST_MODE

Description:	Mode of display list under construction; undefined if none
Attribute Group:	–
Initial Value:	0
Get Command:	glGetIntegerv

GL_ATTRIB_STACK_DEPTH

Description:	Attribute stack pointer
Attribute Group:	–
Initial Value:	0
Get Command:	glGetIntegerv

GL_NAME_STACK_DEPTH

Description:	Name stack depth
Attribute Group:	–
Initial Value:	0
Get Command:	glGetIntegerv

GL_RENDER_MODE

Description:	glRenderMode setting
Attribute Group:	–
Initial Value:	GL_RENDER
Get Command:	glGetIntegerv

–

Description:	Current error code(s)
Attribute Group:	–
Initial Value:	0
Get Command:	glGetError

State Variables for Current Values and Associated Data

GL_CURRENT_COLOR

Description:	Current color
Attribute Group:	current
Initial Value:	1,1,1,1
Get Command:	glGetIntegerv glGetFloatv

GL_CURRENT_INDEX

Description:	Current color index
Attribute Group:	current
Initial Value:	1
Get Command:	glGetIntegerv glGetFloatv

GL_CURRENT_TEXTURE_COORDS

Description:	Current texture coordinates
Attribute Group:	current
Initial Value:	0,0,0,1
Get Command:	glGetFloatv

GL_CURRENT_NORMAL

Description:	Current normal
Attribute Group:	current
Initial Value:	0,0,1
Get Command:	glGetFloatv

GL_CURRENT_RASTER POSITION

Description:	Current raster position
Attribute Group:	current
Initial Value:	0,0,0,1
Get Command:	glGetFloatv

GL_CURRENT_RASTER_DISTANCE

Description:	Current raster distance
Attribute Group:	current
Initial Value:	0
Get Command:	glGetFloatv

GL_CURRENT_RASTER_COLOR

Description:	Color associated with raster position

Attribute Group: current
Initial Value: 1,1,1,1
Get Command: glGetIntegerv glGetFloatv

GL_CURRENT_RASTER_INDEX
Description: Color index associated with raster position
Attribute Group: current
Initial Value: 1
Get Command: glGetIntegerv glGetFloatv

GL_CURRENT_RASTER_TEXTURE_COORDS
Description: Texture coordinates associated with raster position
Attribute Group: current
Initial Value: 0,0,0,1
Get Command: glGetFloatv

GL_CURRENT_RASTER_POSITION_VALID
Description: Raster position valid bit
Attribute Group: current
Initial Value: GL_TRUE
Get Command: glGetBoolean

GL_EDGE_FLAG
Description: Edge flag
Attribute Group: current
Initial Value: GL_TRUE
Get Command: glGetBoolean

Transformation State Variables

GL_MODELVIEW_MATRIX

Description:	Modelview matrix stack
Attribute Group:	–
Initial Value:	Identity
Get Command:	glGetFloatv

GL_PROJECTION_MATRIX

Description:	Projection matrix stack
Attribute Group:	–
Initial Value:	Identity
Get Command:	glGetFloatv

GL_TEXTURE_MATRIX

Description:	Texture matrix stack
Attribute Group:	–
Initial Value:	Identity
Get Command:	glGetFloatv

GL_VIEWPORT

Description:	Viewport origin and extent
Attribute Group:	viewport
Initial Value:	–
Get Command:	glGetIntegerv

GL_DEPTH_RANGE

Description:	Depth range near and far
Attribute Group:	viewport
Initial Value:	0,1
Get Command:	glGetFloatv

GL_MODELVIEW_STACK_DEPTH

Description:	Modelview matrix stack pointer
Attribute Group:	–
Initial Value:	1
Get Command:	glGetIntegerv

GL_PROJECTION_STACK_DEPTH

Description:	Projection matrix stack pointer

Attribute Group: –
Initial Value: 1
Get Command: glGetIntegerv

GL_TEXTURE_STACK_DEPTH
Description: Texture matrix stack pointer
Attribute Group: –
Initial Value: 1
Get Command: glGetIntegerv

GL_MATRIX_MODE
Description: Current matrix mode
Attribute Group: transform
Initial Value: GL_MODELVIEW
Get Command: glGetIntegerv

GL_NORMALIZE
Description: Current normal normalization on/off
Attribute Group: transform/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

GL_CLIP_PLANE*i*
Description: User clipping plane coefficients
Attribute Group: transform
Initial Value: 0,0,0,0
Get Command: glGetClipPlane

GL_CLIP_PLANE*i*
Description: *i*th user clipping plane enabled
Attribute Group: transform/enable
Initial Value: GL_FALSE
Get Command: glIsEnabled

Coloring State Variables

GL_FOG_COLOR

Description:	Fog color
Attribute Group:	fog
Initial Value:	0,0,0,0
Get Command:	glGetFloatv

GL_FOG_INDEX

Description:	Fog index
Attribute Group:	fog
Initial Value:	0
Get Command:	glGetFloatv

GL_FOG_DENSITY

Description:	Exponential fog density
Attribute Group:	fog
Initial Value:	1.0
Get Command:	glGetFloatv

GL_FOG_START

Description:	Linear fog start
Attribute Group:	fog
Initial Value:	0.0
Get Command:	glGetFloatv

GL_FOG_END

Description:	Linear fog end
Attribute Group:	fog
Initial Value:	1.0
Get Command:	glGetFloatv

GL_FOG_MODE

Description:	Fog mode
Attribute Group:	fog
Initial Value:	GL_EXP
Get Command:	glGetIntegerv

GL_FOG

Description:	True if fog enabled

Attribute Group:	fog/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_SHADE_MODEL

Description:	glShadeModel setting
Attribute Group:	lighting
Initial Value:	GL_SMOOTH
Get Command:	glGetIntegerv

Lighting State Variables

GL_LIGHTING

Description:	True if lighting is enabled
Attribute Group:	lighting/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_COLOR_MATERIAL

Description:	True if color tracking is enabled
Attribute Group:	lighting
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_COLOR_MATERIAL_PARAMETER

Description:	Material properties tracking current color
Attribute Group:	lighting
Initial Value:	GL_AMBIENT_AND_DIFFUSE
Get Command:	glGetIntegerv

GL_COLOR_MATERIAL_FACE

Description:	Face(s) affected by color tracking
Attribute Group:	lighting
Initial Value:	GL_FRONT_AND_BACK
Get Command:	glGetIntegerv

GL_AMBIENT

Description:	Ambient material color
Attribute Group:	lighting
Initial Value:	(0.2, 0.2, 0.2, 1.0)
Get Command:	glGetMaterialfv

GL_DIFFUSE

Description:	Diffuse material color
Attribute Group:	lighting
Initial Value:	(0.8, 0.8, 0.8, 1.0)
Get Command:	glGetMaterialfv

GL_SPECULAR

Description:	Specular material color

Attribute Group: lighting
Initial Value: (0.0, 0.0, 0.0, 1.0)
Get Command: glGetMaterialfv

GL_EMISSION
Description: Emissive material color
Attribute Group: lighting
Initial Value: (0.0, 0.0, 0.0, 1.0)
Get Command: glGet

GL_SHININESS
Description: Specular exponent of material
Attribute Group: lighting
Initial Value: 0.0
Get Command: glGetMaterialfv

GL_LIGHT_MODEL_AMBIENT
Description: Ambient scene color
Attribute Group: lighting
Initial Value: (0.2, 0.2, 0.2, 0.1)
Get Command: glGetFloatv

GL_LIGHT_MODEL_LOCAL_VIEWER
Description: Viewer is local
Attribute Group: lighting
Initial Value: GL_FALSE
Get Command: glGetBoolean

GL_LIGHT_MODEL_TWO_SIDE
Description: Use two-sided lighting
Attribute Group: lighting
Initial Value: GL_FALSE
Get Command: glGetBooleanv

GL_AMBIENT
Description: Ambient intensity of light *i*
Attribute Group: lighting
Initial Value: (0.0, 0.0, 0.0, 1.0)

Get Command: glGetLightfv

GL_DIFFUSE

Description: Diffuse intensity of light *i*

Attribute Group: lighting

Initial Value: –

Get Command: glGetLightfv

GL_SPECULAR

Description: Specular intensity of light *i*

Attribute Group: lighting

Initial Value: –

Get Command: glGetLightfv

GL_POSITION

Description: Position of light *i*

Attribute Group: lighting

Initial Value: (0.0, 0.0, 1.0, 0.0)

Get Command: glGetLightfv

GL_CONSTANT_ATTENUATION

Description: Constant attenuation factor

Attribute Group: lighting

Initial Value: 1.0

Get Command: glGetLightfv

GL_LINEAR_ATTENUATION

Description: Linear attenuation factor

Attribute Group: lighting

Initial Value: 0.0

Get Command: glGetLightfv

GL_QUADRATIC_ATTENUATION

Description: Quadratic attenuation factor

Attribute Group: lighting

Initial Value: 0.0

Get Command: glGetLightfv

GL_SPOT_DIRECTION

Description:	Spotlight direction of light *i*
Attribute Group:	lighting
Initial Value:	(0.0, 0.0, -1.0)
Get Command:	glGetLightfv

GL_SPOT_EXPONENT

Description:	Spotlight exponent of light *i*
Attribute Group:	lighting
Initial Value:	0.0
Get Command:	glGetLightfv

GL_SPOT_CUTOFF

Description:	Spotlight angle of light *i*
Attribute Group:	lighting
Initial Value:	180.0
Get Command:	glGetLightfv

GL_LIGHT*i*

Description:	True if light *i* enabled
Attribute Group:	lighting/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_COLOR_INDEXES

Description:	$C_{(a)}$, $C_{(d)}$, and $C_{(s)}$ for color-index lighting
Attribute Group:	lighting/enable
Initial Value:	0, 1, 1
Get Command:	glGetFloatv

Pixel Operations

GL_SCISSOR_TEST

Description:	Scissoring enabled
Attribute Group:	scissor/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_SCISSOR_BOX

Description:	Scissor box
Attribute Group:	scissor
Initial Value:	–
Get Command:	glGetGetIntegerv

GL_STENCIL_TEST

Description:	Stenciling enabled
Attribute Group:	stencil-buffer/enable
Initial Value:	GL_FALSE
Get Command:	glIsEnabled

GL_STENCIL_FUNC

Description:	Stencil function
Attribute Group:	stencil-buffer
Initial Value:	GL_ALWAYS
Get Command:	glGetIntegerv

GL_STENCIL_VALUE_MASK

Description:	Stencil mask
Attribute Group:	stencil-buffer
Initial Value:	1's
Get Command:	glGetIntegerv

GL_STENCIL_REF

Description:	Stencil reference value
Attribute Group:	stencil-buffer
Initial Value:	0
Get Command:	glGetIntegerv

GL_STENCIL_FAIL

Description:	Stencil fail action

Attribute Group: stencil-buffer

Initial Value: GL_KEEP

Get Command: glGetIntegerv

GL_STENCIL_PASS_DEPTH_FAIL

Description: Stencil depth buffer fail action

Attribute Group: stencil-buffer

Initial Value: GL_KEEP

Get Command: glGetIntegerv

GL_STENCIL_PASS_DEPTH_PASS

Description: Stencil depth buffer pass action

Attribute Group: stencil-buffer

Initial Value: GL_KEEP

Get Command: glGetIntegerv

GL_ALPHA_TEST

Description: Alpha test enabled

Attribute Group: color-buffer/enable

Initial Value: GL_FALSE

Get Command: glIsEnabled

GL_ALPHA_TEST_FUNC

Description: Alpha test function

Attribute Group: color-buffer

Initial Value: GL_ALWAYS

Get Command: glGetIntegerv

GL_ALPHA_TEST_REF

Description: Alpha test reference value

Attribute Group: color-buffer

Initial Value: 0

Get Command: glGetIntegerv

GL_DEPTH_TEST

Description: Depth buffer enabled

Attribute Group: depth-buffer/enable

Initial Value: GL_FALSE

Get Command: glIsEnabled

GL_DEPTH_FUNC

Description: Depth buffer test function

Attribute Group: depth-buffer

Initial Value: GL_LESS

Get Command: glGetIntegerv

GL_BLEND

Description: Blending enabled

Attribute Group: color-buffer/enable

Initial Value: GL_FALSE

Get Command: glIsEnabled

GL_BLENC_SRC

Description: Blending source function

Attribute Group: color-buffer

Initial Value: GL_ONE

Get Command: glGetIntegerv

GL_BLEND_DST

Description: Blending destination function

Attribute Group: color-buffer

Initial Value: GL_ZERO

Get Command: glGetIntegerv

GL_LOGIC_OP

Description: Logical operation enabled

Attribute Group: color-buffer/enable

Initial Value: GL_FALSE

Get Command: glIsEnabled

GL_LOGIC_OP_MODE

Description: Logical operation function

Attribute Group: color-buffer

Initial Value: GL_COPY

Get Command: glGetIntegerv

GL_DITHER

Description:	Dithering enabled
Attribute Group:	color-buffer/enable
Initial Value:	GL_TRUE
Get Command:	glIsEnabled

OpenGL on Windows NT

The Microsoft® implementation of OpenGL™ in Windows NT™ is an implementation of the industry-standard OpenGL three-dimensional (3D) graphics software interface that lets programmers create high-quality still and animated 3D color images. This overview describes the Windows NT implementation of OpenGL.

About OpenGL

OpenGL, originally developed by Silicon Graphics Incorporated (SGI) for their graphics workstations, lets applications create high-quality color images independent of windowing systems, operating systems, and hardware.

The OpenGL Architecture Review Board (ARB), an industry consortium, is currently responsible for defining OpenGL. Members of the ARB include Silicon Graphics Incorporated, Microsoft Corporation, Intel, IBM, and Digital Equipment Corporation.

The official reference document for OpenGL, version 1, is the *OpenGL Reference Manual*, by the OpenGL Architecture Review Board (ISBN 0-201-63276-4). The official guide to learning OpenGL, version 1, is the *OpenGL Programming Guide*, by Jackie Neider, Tom Davis, and Mason Woo (ISBN 0-201-63274-8). Both books are published by Addison-Wesley.

OpenGL in Windows NT

With Microsoft's implementation of OpenGL in Windows NT, application developers can use OpenGL in Windows NT applications, and existing applications that use OpenGL can be ported to Windows NT.

Components

Microsoft's implementation of OpenGL in Windows NT includes the following components:

- the full set of current OpenGL commands

  The core of OpenGL, this library contains 115 distinct functions for 3D graphics operations. These basic functions deal with such things as object shape description, matrix transformation, lighting, coloring, texture, clipping, bitmaps, fog, and anti-aliasing. The names for these core functions have a "gl" prefix.

  Many of the 115 distinct OpenGL commands have several variants. The variants differ in the number and type of their parameters. Counting all the variants, there are more than 300 OpenGL commands.

- the OpenGL Utility (GLU) library

  This is a library of 43 auxiliary functions that complement the core OpenGL functions. The commands deal with texture support, coordinate transformation, polygon tessellation, rendering spheres, cylinders and disks, NURBS (Non-Uniform Rational B-Spline) curves and surfaces, and error handling.

- the OpenGL Programming Guide Auxiliary Library

  This is a simple, platform-independent library of 31 functions for managing windows, handling input events, drawing classic 3D objects, managing a background process, and running a program. The window management and input routines provide a base level of functionality that let you quickly get started programming in OpenGL, but they should not be used in a production application. Here are some reasons for this warning:

  - The message loop is in the library code.
  - There is no way to add handlers for additional WM* messages.
  - There is very little support for logical palettes.

  The library is described and used in the *OpenGL Programming Guide*.

- the WGL APIs

  This is a set of nine functions that connects OpenGL to the Windows NT windowing system. The functions manage rendering contexts, display lists, extension functions, and font bitmaps. The WGL functions are analogous to the GLX extensions that connect OpenGL to the X-Windows windowing system. The names for these functions have a "wgl" prefix.

- new Win32 APIs for pixel formats and double buffering

  This is a set of five functions that support per-window pixel formats and double buffering (for smooth image changes) of windows. These new functions apply only to OpenGL graphics windows.

Generic Implementation and Hardware Implementations

This overview discusses the current generic implementation of OpenGL in Windows NT. The generic implementation is the Microsoft Windows NT software implementation of OpenGL. Hardware manufacturers may enhance parts of OpenGL in their drivers, and may support some features not supported by the generic implementation.

Limitations

The generic implementation has some limitations:

- Printing.

  An application cannot print an OpenGL image directly to a monochrome printer. There is, however, a workaround for this situation: see **Printing an OpenGL Image**. An application can print an OpenGL image directly to a color printer that offers four or more bits of color information per pixel.
- OpenGL and GDI graphics cannot be mixed in a double-buffered window.

  An application can draw both OpenGL graphics and GDI graphics directly into a single-buffered window, but not into a double-buffered window.
- There are no per-window hardware color palettes.

  Windows NT has a single system hardware color palette, which applies to the whole screen. An OpenGL window cannot have its own hardware palette. It can have its own logical palette. To do so, it must become a palette-aware application. For more information, see **OpenGL Color Modes and Windows Palette Management**.
- There is no direct support for the Clipboard, DDE, metafiles, or OLE.

  A window with OpenGL graphics does not directly support these Windows NT capabilities. There are workarounds, however, for working with the Clipboard; see **Copying an OpenGL Image to the Clipboard**.
- The Inventor 2.0 C++ class library is not included.

  The Inventor class library, built on top of OpenGL, provides higher-level constructs for programming 3D graphics. It is not included in the current version of Microsoft's implementation of OpenGL for Windows NT.
- There is no support for several pixel format features: overlay and underlay layers, stereoscopic images, alpha bitplanes, and auxiliary buffers.

  There is, however, support for several ancillary buffers: stencil buffer, accumulation buffer, back buffer (double buffering), and depth (z-axis) buffer.

Guide To Documentation

There are five elements in the documentation set for OpenGL in Windows NT.

The first two elements are the official OpenGL books previously mentioned: the *OpenGL Reference Manual* and the *OpenGL Programming Guide*. Due to the dominant colors of their covers, these books are known respectively as "the Blue book" and "the Red book."

Note *OpenGL Reference Manual* and the *OpenGL Programming Guide* are not included with the Win32 SDK.

The *OpenGL Reference Manual*, the Blue book, includes an overview of how OpenGL works and a set of detailed reference pages. The reference pages cover all of the 115 distinct OpenGL functions, as well as the 43 functions in the OpenGL Utility (GLU) library.

The *OpenGL Programming Guide*, the Red book, explains how to create graphics programs using OpenGL. It includes discussions of the following major topics:

- drawing geometric shapes
- pixels, bitmaps, fonts, and images
- viewing and matrix transformations
- texture mapping
- display lists
- advanced composite techniques
- color
- evaluators and NURBS
- lighting
- selection and feedback
- blending, anti-aliasing, and fog
- advanced techniques

In addition, the *OpenGL Programming Guide* contains appendixes that discuss the OpenGL Utility Library and the OpenGL Programming Guide Auxiliary Library.

The third documentation element is this overview chapter. It describes the Windows NT implementation of OpenGL and provides an overview of its components. It discusses several important concepts: rendering contexts, pixel formats, and buffers. It discusses the six new WGL APIs that connect OpenGL to the Windows NT windowing system, and the five new Win32 APIs that support per-window pixel formats and double buffering of windows for OpenGL graphics windows. The six WGL APIs and the five new Win32 APIs are specific to the Windows NT implementation of OpenGL.

The fourth documentation element is the set of reference pages for the six WGL APIs, the five new Win32 APIs just mentioned, and the PIXELFORMATDESCRIPTOR data structure.

The fifth documentation element is *Porting to OpenGL*. It discusses moving existing OpenGL code from other environments into Windows NT.

Rendering Contexts

An *OpenGL rendering context* is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls needs to have a current rendering context. Rendering contexts link OpenGL to the Windows NT windowing system.

An application specifies a Windows device context when it creates a rendering context. The rendering context created is suitable for drawing on the device that the specified device context references. In particular, the rendering context has the same pixel format as the device context. For more information, see **Rendering Context Functions**. Despite this relationship, a rendering context is not the same as a device context. A device context contains information pertinent to the graphics component of Windows, GDI. A rendering context contains information pertinent to OpenGL. A device context must be explicitly specified in a GDI call. A rendering context is implicit in an OpenGL call. An application should set a device context's pixel format before creating a rendering context.

A thread that makes OpenGL calls must have a current rendering context. If an application makes OpenGL calls from a thread that lacks a current rendering context, nothing happens; the call has no effect. An application commonly creates a rendering context, sets it as a thread's current rendering context, then calls OpenGL functions. When it finishes calling OpenGL functions, the application uncouples the rendering context from the thread, and then deletes the rendering context. An application can use multiple rendering contexts drawing to the same window, but a thread can have one current, active rendering context.

A current rendering context has an associated device context. That device context need not be the same device context as that used when the rendering context was created, but it must reference the same device and have the same pixel format.

A thread can have only one current rendering context. A rendering context can be current to only one thread.

Rendering Context Functions

Five WGL functions manage rendering contexts:

WGL Function	Description
wglCreateContext	Creates a new rendering context.
wglMakeCurrent	Sets a thread's current rendering context.
wglGetCurrentContext	Obtains a handle to a thread's current rendering context.
wglGetCurrentDC	Obtains a handle to the device context associated with a thread's current rendering context.
wglDeleteContext	Deletes a rendering context.

An application creates a rendering context by calling **wglCreateContext**. The **wglCreateContext** function takes a device context handle as its parameter and returns a rendering context handle. The created rendering context is suitable for drawing on the device referenced by the device context handle. In particular, its pixel format is the same as the device context's pixel format. After the rendering context is created, an application can release or dispose of the device context. See the **Device Contexts** overview for more details on creating, obtaining, releasing, and disposing of a device context. One important note: the device context sent to **wglCreateContext** must be a display device context, a memory device context, or a color printer device context that uses four or more bits per pixel. The device context cannot be a monochrome printer device context.

An application gives a thread a current rendering context by calling **wglMakeCurrent**. The **wglMakeCurrent** function takes a rendering context handle and a device context handle as parameters. All subsequent OpenGL calls made by the thread are made through that rendering context, and are drawn on the device referenced by that device context. The device context does not have to be the same one passed to **wglCreateContext** when the rendering context was created, but it must be on the same device and have the same pixel format. The call to **wglMakeCurrent** creates an association between the supplied rendering context and device context. An application cannot release or dispose of the device context associated with a current rendering context until the rendering context is made not current.

Once a thread has a current rendering context, it can make OpenGL graphics calls. All calls must pass through a rendering context. Nothing happens if an application makes OpenGL graphics calls from a thread that lacks a current rendering context.

An application obtains a handle to a thread's current rendering context by calling **wglGetCurrentContext**. The **wglGetCurrentContext** function takes no parameters, and returns a handle to the calling thread's current rendering context. If the thread has no current rendering context, the return value is NULL.

An application obtains a handle to the device context associated with a thread's current rendering context by calling **wglGetCurrentDC**. As noted above, such associations are created when a rendering context is made current.

There are two ways for an application to break the association between a current rendering context and a thread: calling **wglMakeCurrent** with a null rendering context handle, and calling **wglMakeCurrent** with a handle other than the one originally called.

After calling **wglMakeCurrent** with the rendering context handle parameter set to NULL, the calling thread has no current rendering context. The rendering context is released from its connection to the thread, and the rendering context's association to a device context ends. OpenGL flushes all drawing commands, and may release some resources. No OpenGL drawing will be done until the next call to **wglMakeCurrent**, since the thread can make no OpenGL graphics calls until it regains a current rendering context.

The second way an application can break the association between a rendering context and a thread is to call **wglMakeCurrent** with a different rendering context. After such a call, the calling thread has a new current rendering context, the former current rendering context is released from its connection to the thread, and the former current rendering context's association to a device context ends.

An application deletes a rendering context by calling **wglDeleteContext**. The **wglDeleteContext** function takes a single parameter, the handle to the rendering context to be deleted. Before calling **wglDeleteContext** an application should make the rendering context not current by calling **wglMakeCurrent**, and delete or release the associated device context by calling **DeleteDC** or **ReleaseDC** as appropriate.

It is an error for a thread to delete a rendering context that is another thread's current rendering context. However, if a rendering context is the calling thread's current rendering context, **wglDeleteContext** will flush all OpenGL drawing commands and make the rendering context not current before deleting it. In this case, relying on **wglDeleteContext** to make a rendering context not current, it is the programmer's responsibility to delete or release the associated device context.

Pixel Formats

A *pixel format* specifies several properties of an OpenGL drawing surface. Some of the properties specified by a pixel format are:

- whether the pixel buffer is single- or double-buffered
- whether the pixel data is in RGBA or color-indexed form
- the number of bits used to store color data
- the number of bits used for the depth (z-axis) buffer
- the number of bits used for the stencil buffer
- various visibility masks

Microsoft's implementation of OpenGL for Windows NT uses the **PIXELFORMATDESCRIPTOR** data structure to convey pixel format data. The structure's members specify the preceding properties and several others.

A given device context can support several pixel formats. Windows NT identifies the pixel formats that a device context supports with consecutive one-based index values (1, 2, 3, 4, and so on). A device context can have just one current pixel format, chosen from the set of pixel formats it supports.

Each window has its own current pixel format in OpenGL in WIndows NT. This means, for example, that an application can simultaneously display RGBA and color-indexed OpenGL windows, or single- and double-buffer OpenGL windows. This per-window pixel format capability is limited to OpenGL windows.

An application typically obtains a device context, sets the device context's pixel format, and then creates an OpenGL rendering context suitable for that device. Notice that the pixel format is set before creating a rendering context. The rendering context inherits the device context's pixel format.

Pixel Format Functions

Four new Win32 functions manage pixel formats:

Win32 Function	Description
ChoosePixelFormat	Obtains a device context's pixel format that is the closest match to a specified pixel format.
SetPixelFormat	Sets a device context's current pixel format to the pixel format specified by a pixel format index.
GetPixelFormat	Obtains the pixel format index of a device context's current pixel format.
DescribePixelFormat	Given a device context and a pixel format index, fills in a PIXELFORMATDESCRIPTOR data structure with the pixel format's properties.

An application calls the **ChoosePixelFormat** function to obtain a device context's best match to a specified pixel format. The **ChoosePixelFormat** function returns a one-based pixel format index that identifies the best match from the device context's supported pixel formats.

An application sets a specified device context's current pixel format by calling the **SetPixelFormat** function. The **SetPixelFormat** function identifies the desired format using a one-based pixel format index. Typically, an application calls **ChoosePixelFormat** to find a best-match pixel format, then calls **SetPixelFormat** with the result of **ChoosePixelFormat**.

If **SetPixelFormat** is called for a device context that references a window, the function also changes the pixel format of the window. Setting the pixel format of a window more than once can lead to significant complications for the window manager and for multi-threaded applications, so it is not allowed. An application can set the pixel format of a window only one time. Once a window's pixel format is set, it cannot be changed.

An application obtains a device context's current pixel format by calling the **GetPixelFormat** function. The **GetPixelFormat** function returns a one-based pixel format index.

An application obtains pixel format data by calling the **DescribePixelFormat** function. The **DescribePixelFormat** function takes a handle to a device context, a pixel format index, and a pointer to a **PIXELFORMATDESCRIPTOR** data structure as parameters, and returns with the members of **PIXELFORMATDESCRIPTOR** appropriately set.

Front, Back, and Other Buffers

OpenGL stores and manipulates pixel data in a frame buffer. The frame buffer consists of a set of logical buffers: color, depth, accumulation, and stencil buffers. The color buffer itself consists of a set of logical buffers; this set can include a front left, a front right, a back left, a back right, and some number of auxiliary buffers. A particular pixel format or OpenGL implementation may not supply all of these buffers. For example, the current version of Microsoft's implementation of OpenGL in Windows NT does not support stereoscopic images, and so a pixel format cannot have left and right color buffers. In addition, the current version does not support auxiliary buffers. Refer to the Red and Blue books for further details on OpenGL buffers and the OpenGL functions that operate on them.

Microsoft's implementation of OpenGL in Windows NT supports double buffering of images. This is a technique in which an application draws pixels to an off-screen buffer, and then, when that image is ready for display, the application copies the contents of the off-screen buffer to an on-screen buffer. Double buffering enables smooth image changes, which are especially important for animated images.

Two color buffers are available to applications that use double buffering: a front buffer and a back buffer. By default, drawing commands are directed to the back buffer, while the front buffer is displayed on the screen. When the off-screen buffer is ready for display, an application calls **SwapBuffers**, and Windows NT copies the contents of the off-screen buffer to the on-screen buffer.

The generic implementation uses a device-independent bitmap (DIB) as a back buffer and the screen display as a front buffer. Hardware devices and their drivers may use different approaches.

Double buffering is a pixel format property. An application requests double buffering for a pixel format by setting the PFD_DOUBLEBUFFER flag in the **PIXELFORMATDESCRIPTOR** data structure in a call to **ChoosePixelFormat**.

The OpenGL core function **glDrawBuffer** selects buffers for writing and clearing.

Buffer Functions

One new Win32 function manages buffers:

Win32 Function	Description
SwapBuffers	Copies the contents of the off-screen buffer to the on-screen buffer if the current pixel format for the specified device context includes a back buffer. By default, the back buffer is off-screen, and the front buffer is on-screen. After the operation, the contents of the off-screen buffer are undefined.

An application copies the contents of an off-screen buffer to an on-screen buffer by calling the **SwapBuffers** function. The **SwapBuffers** function takes a handle to a device context. The current pixel format for the specified device context must include a back buffer.

Notice carefully that the **SwapBuffers** function does not really swap the contents of the two buffers, but rather copies the contents of one buffer to another. The contents of the off-screen buffer are undefined after calling this function. Thus, the result of two consecutive calls to **SwapBuffers** is undefined. The following illustrates this situation:

{ewc msdncd, EWGraphic, group10446 0 /a "SDK.bmp"}

Several OpenGL core functions also deal with buffers. The **glDrawBuffer** function previously mentioned is the one most relevant to double buffering; it specifies the frame buffer(s) that OpenGL draws into. In addition, there are these functions: **glReadBuffer**, **glReadPixels**, **glCopyPixels**, **glAccum**, **glColorMask**, **glDepthMask**, **glIndexMask**, **glStencilMask**, and **glClearAccum**, **glClearColor**, **glClearDepth**, **glClearIndex**, and **glClearStencil**.

Fonts and Text

Microsoft's implementation of OpenGL in Windows NT supports GDI graphics in a single-buffered OpenGL window. It does not support GDI graphics in a double-buffered OpenGL window. Thus, an application can call only the standard GDI font and text functions to draw text in a single-buffered OpenGL window; it cannot call those functions to draw text in a double-buffered OpenGL window.

There is a workaround for this restriction on text in double-buffered windows. An application can build OpenGL display lists for bitmap images of characters, and then execute those display lists to draw characters. There are three main steps in this process:

- Select a font for a device context, setting the font's properties as desired.
- Create a set of bitmap display lists based on the glyphs in the device context's font, one display list for each glyph that the application will draw.
- Draw each glyph in a string, using those bitmap display lists.

An application calls the **wglUseFontBitmaps** and **wglUseFontOutlines** functions to create the display lists, and uses **glCallLists** to draw characters in a string using those display lists. .

To create applications that are easy to localize and that use resources sparingly, a programmer must carefully manage the creation and storage of these glyph image display lists. Many languages, unlike English, have alphabets whose character codes range over a relatively large set of values.

Font and Text Functions

Two WGL functions deal with fonts and text:

Win32 Function	Description
wglUseFontBitmaps	Creates a set of character bitmap display lists. Characters come from a specified device context's current font. Characters are specified as a consecutive run within the font's glyph set.
wglUseFontOutlines	Creates a set of display lists based on the glyphs of the currently selected outline font of a device context for use with the current rendering context. The display lists are used to draw 3D characters of TrueType fonts.

An application creates a set of character bitmap display lists by calling the **wglUseFontBitmaps** function or the **wglUseFontOutlines** function. These functions take a handle to a device context, and use that device context's current font as a source for the bitmaps. Thus, an application should set the device context's font and the font's properties before calling **wglUseFontBitmaps** or **wglUseFontOutlines**.

The wglUseFontBitmaps and wglUseFontOutlines functions also take a parameter that turns the first glyph in the font into a bitmap display list, and a parameter that specifies how many glyphs to turn into display lists. The function then creates display lists for the specified consecutive run of glyphs. For example:

- An application can set these two parameters to 32 and 224, respectively, to create a set of 224 bitmap display lists for all of the Windows character set glyphs.
- An application can set these two parameters to 0 and 256, respectively, to create a set of 256 bitmap display lists for all of the OEM character set glyphs.
- An application can set the second of these parameters to 1 to create a single bitmap display list for any single character set glyph.

The **wglUseFontBitmaps** and **wglUseFontOutlines** functions represent a null glyph in a font with an empty display list.

The display lists created by a call to **wglUseFontBitmaps** or **wglUseFontOutlines** are automatically numbered consecutively.

After calling wglUseFontBitmaps or wglUseFontOutlines, an application calls the glCallLists function to draw a string of characters. See Drawing Text in a Double-Buffered OpenGL Window for some sample code. In this context, the **glCallLists** function uses each character in a string as an index into the array of consecutively numbered display lists created by calling **wglUseFontBitmaps** or **wglUseFontOutlines**.

When an application finishes drawing text, it calls glDeleteLists to release the contiguous set of display lists created by **wglUseFontBitmaps** and **wglUseFontOutlines**.

OpenGL Color Modes and Windows Palette Management

Microsoft's implementation of OpenGL in Windows NT supports two color pixel data modes: RGBA and color-indexed mode. Windows and Windows NT provide two analogous ways of handling color: true color and palette management.

True-color devices, able to accept 16, 24, or more bits of color information per pixel, can display tens of thousands, tens of millions, or more colors simultaneously. Complexities arise, however, when an application has to deal with RGBA or color-indexed mode on a palette-type device. Palette-type devices, such as a 256-color VGA display, are limited in the number of colors they can display simultaneously. Applications must handle a number of tricky details to successfully use palette-type devices. Because color-index mode programs don't use a hardware palette, they are more difficult to use with true-color devices than programs using the RGBA mode.

If you are unfamiliar with using true-color devices or the Palette Manager, refer to the articles "Palette Awareness," "The Palette Manager: How and Why," and "Using True-color devices" on the Microsoft Development Library CDs for an introduction to the essentials of Windows color management.

Palettes and the Palette Manager

The Windows NT Palette Manager, which is part of the GDI, specifically targets 8-bit display adapters with a *hardware palette* of 256 color entries. Pixels on the screen are stored as an 8-bit index into the hardware palette. Each entry in the hardware palette usually defines a 24-bit color (eight each of red, green, and blue).

The Palette Manager maintains a *system palette* that is a copy of the hardware palette. The system palette is divided into two sections: 20 reserved colors and the remaining 236 colors, which you can set using the Palette Manager

A default 20-color logical palette is selected and realized into a device context. You can create and use a new logical palette. By selecting and realizing that palette you can change the system palette.

You'll probably create a *logical palette* to specify the colors you want display in your OpenGL application. Using certain GDI calls, your application can temporarily replace most of the system palette with a logical palette. Using a logical palette you can define pixel colors for the GDI using either the RGBA or the color-index mode. The maximum size of a logical palette is 256 colors for 8-bit devices and 4,096 colors on a true-color device (16, 24, and 32 bits).

For more information on the RGBA and color-index modes, see RGBA Mode and Palette Management and Color Modes and Windows Palette Management.

Palette Awareness

Your application must respond to the WM_PALETTECHANGED, WM_QUERYNEWPALETTE, and WM_ACTIVATE messages to be aware of and use palettes. Design your application to select and realize palettes in response to these messages.

For more information on palettes and palette awareness, refer to the articles "Palette Awareness," and "The Palette Manager: How and Why" on the Microsoft Development Library CDs.

Reading Color Values from the Frame Buffer

When using functions that read back color values from the frame buffer, be aware of the differences between reading RGBA values and color-index values on true-color devices and on palette-based devices:

- On a true-color device:
  - RGBA values are limited to the channel in the device.
  - Color-index values are stored a RGBA values in the frame buffer. When using these values, you must perform an inverse translation from RGBA to the logical palette index. If two logical indexes have the same RGB value, the wrong index can be returned.
- On a palette-based device:
  - RGBA values are read from an index in the system palette. The logical index is obtained from an inverse table and the RGBA components are extracted.
  - Color-index values are read from an index into the system palette and an inverse table is used to get the logical palette index.

Choosing Between RGBA and Color-Index Mode

In general, use the RGBA mode for your OpenGL applications; it provides more flexibility than the color-index mode for effects such as shading, lighting, color mapping, fog, anti-aliasing, and blending. Consider using the color-index mode in the following cases:

- If you have a limited number of bitplanes available, the color-index mode can produce less coarse shading than the RGBA mode.
- If you are not concerned about using "real" colors; for example, using several colors in a topographic map to designate relative elevations.
- When you're porting an existing application that uses color-index mode extensively.
- When you want to use colormap animation and effects in your application. (This is not possible on true-color devices.)

RGBA Mode and Palette Management

While most GDI applications tend to use color-indexing with logical palettes, for OpenGL applications you'll usually choose the RGBA mode. It works better than color mapping for several effects, such as shading, lighting, fog, and texture mapping.

The RGBA mode uses red, green, and blue (R, G, and B) color values that together specify the color of each pixel in the display. The R, G, and B values specify the intensity of each color (red, green, or blue); the values range from 0.0 (least intense) to 1.0 (most intense). The number of bits for each component varies depending on the hardware used (2, 3, 5, 6, and 8 bits are possible).The color displayed is a result of the sum of the three color values. If all three values are 0.0, the result is black. If they are all 1.0, the result is white. Other colors are a result of a combination of values of R, G, and B that fall between 0 and 1.0. The A (alpha) bit isn't used to specify color.

The standard super-VGA display uses palettes with eight color-bits per pixel. The eight bits are read from the from buffer and used as an index in the system palette to get the R, G, and B values. When an RGB palette is selected and realized in a device context, OpenGL can render using the RGBA mode.

Because there are eight color-bits per pixel, OpenGL emphasizes the use of a three-three-two RGB palette. The three-three-two refers to how the color-bit data is handled by the hardware or physical palette. Red (R) and green (G) are each specified by three bits while blue (B) is specified by two bits. Red is the least-significant bit and blue is the most-significant bit.

You determine the colors of your application's logical palette with **PALETTEENTRY** structures. Typically you create an array of PALETTEENTRY structures to specify entire palette entry table of the logical palette.

RGBA Mode Palette Sample

The following code fragment shows how you can create a three-three-two RGBA palette.

```
/*
 * win8map.c - program to create an 8-bit RGB color map for
 * use with OpenGL
 *
 * For OpenGL RGB rendering you need to know red, green, & blue
 * component bit sizes and positions. On 8 bit palette devices you need
 * to create a logical palette that has the correct RGB values for all
 * 256 possible entries. This program creates an 8 bit RGB color cube
 * with a default gamma of 1.4.
 *
 * Unfortunately, because the standard 20 colors in the system palette
 * cannot be changed,if you select this palette into an 8-bit display
 * DC, you will not realize all of the logical palette. The program
 * changes some of the entries in the logical palette to match enties in
 * the system palette using a least-squares calculation to find which
 * entries to replace.
  *
 * Note: Three bits for red & green and two bits for blue; red is the
 * least-significant bit and blue is the most-significant bit.
 */

#include <stdio.h>
#include <math.h>

#define DEFAULT_GAMMA 1.4F
```

```
#define MAX_PAL_ERROR (3*256*256L)

struct colorentry {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
};

struct rampentry {
    struct colorentry color;
    long defaultindex;
    unsigned char flags;
};


struct defaultentry {
    struct colorentry color;
    long rampindex;
    unsigned char flags;
};

/* values for flags */
#define EXACTMATCH      0x01
#define CHANGED         0x02        /* one of the default entries is close
*/

/*
 * These arrays hold bit arrays with a gamma of 1.0
 * used to convert n bit values to 8 bit values
 */

unsigned char threeto8[8] = {
    0, 0111>>1, 0222>>1, 0333>>1, 0444>>1, 0555>>1, 0666>>1, 0377
};

unsigned char twoto8[4] = {
    0, 0x55, 0xaa, 0xff
};

unsigned char oneto8[2] = {
    0, 255
};

struct defaultentry defaultpal[20] = {
    { 0,   0,   0 },
    { 0x80,0,   0 },
    { 0,   0x80,0 },
    { 0x80,0x80,0 },
    { 0,   0,   0x80 },
    { 0x80,0,   0x80 },
    { 0,   0x80,0x80 },
    { 0xC0,0xC0,0xC0 },

    { 192, 220, 192 },
```

```
    { 166, 202, 240 },
    { 255, 251, 240 },
    { 160, 160, 164 },

    { 0x80,0x80,0x80 },
    { 0xFF,0,   0 },
    { 0,   0xFF,0 },
    { 0xFF,0xFF,0 },
    { 0,   0,   0xFF },
    { 0xFF,0,   0xFF },
    { 0,   0xFF,0xFF },
    { 0xFF,0xFF,0xFF }
};

struct rampentry rampmap[256];

void
gammacorrect(double gamma)
{
    int i;
    unsigned char v, nv;
    double dv;

    for (i=0; i<8; i++) {
        v = threeto8[i];
        dv = (255.0 * pow(v/255.0, 1.0/gamma)) + 0.5;
    nv = (unsigned char)dv;
    printf("Gamma correct %d to %d (gamma %.2f)\n", v, nv, gamma);
        threeto8[i] = nv;
    }
    for (i=0; i<4; i++) {
        v = twoto8[i];
        dv = (255.0 * pow(v/255.0, 1.0/gamma)) + 0.5;
    nv = (unsigned char)dv;
    printf("Gamma correct %d to %d (gamma %.2f)\n", v, nv, gamma);
        twoto8[i] = nv;
    }
    printf("\n");
}

main(int argc, char *argv[])
{
    long i, j, error, min_error;
    long error_index, delta;
    double gamma;
    struct colorentry *pc;

    if (argc == 2)
        gamma = atof(argv[1]);
    else
        gamma = DEFAULT_GAMMA;

    gammacorrect(gamma);

    /* First create a 256 entry RGB color cube */
```

```
for (i = 0; i < 256; i++) {
    /* BGR: 2:3:3 */
    rampmap[i].color.red = threeto8[(i&7)];
    rampmap[i].color.green = threeto8[((i>>3)&7)];
    rampmap[i].color.blue = twoto8[(i>>6)&3];
}

/* Go through the default palette and find exact matches */
for (i=0; i<20; i++) {
    for(j=0; j<256; j++) {
        if ( (defaultpal[i].color.red == rampmap[j].color.red) &&
             (defaultpal[i].color.green == rampmap[j].color.green) &&
             (defaultpal[i].color.blue == rampmap[j].color.blue)) {

            rampmap[j].flags = EXACTMATCH;
            rampmap[j].defaultindex = i;
            defaultpal[i].rampindex = j;
            defaultpal[i].flags = EXACTMATCH;
            break;
        }
    }
}

/* Now find close matches */
for (i=0; i<20; i++) {
    if (defaultpal[i].flags == EXACTMATCH)
        continue;         /* skip entries w/ exact matches */
    min_error = MAX_PAL_ERROR;

    /* Loop through RGB ramp and calculate least square error */
    /* if an entry has already been used, skip it */
    for(j=0; j<256; j++) {
        if (rampmap[j].flags != 0)      /* Already used */
            continue;

        delta = defaultpal[i].color.red - rampmap[j].color.red;
        error = (delta * delta);
        delta = defaultpal[i].color.green - rampmap[j].color.green;
        error += (delta * delta);
        delta = defaultpal[i].color.blue - rampmap[j].color.blue;
        error += (delta * delta);
        if (error < min_error) {        /* New minimum? */
            error_index = j;
            min_error = error;
        }
    }
    defaultpal[i].rampindex = error_index;
    rampmap[error_index].flags = CHANGED;
    rampmap[error_index].defaultindex = i;
}

/* First print out the color cube */

printf("Standard 8 bit RGB color cube with gamma %.2f:\n", gamma);
```

```
for (i=0; i<256; i++) {
    pc = &rampmap[i].color;
    printf("%3ld: (%3d, %3d, %3d)\n", i, pc->red,
            pc->green, pc->blue);

}
printf("\n");

/* Now print out the default entries that have an exact match */

for (i=0; i<20; i++) {
    if (defaultpal[i].flags == EXACTMATCH) {
        pc = &defaultpal[i].color;
        printf("Default entry %2ld exactly matched RGB ramp entry
                  %3ld", i, defaultpal[i].rampindex);
        printf(" (%3d, %3d, %3d)\n", pc->red, pc->green, pc->blue);
    }
}
printf("\n");

/* Now print out the closet entries for rest of
 * the default entries */

for (i=0; i<20; i++) {
    if (defaultpal[i].flags != EXACTMATCH) {
        pc = &defaultpal[i].color;
        printf("Default entry %2ld (%3d, %3d, %3d) is close to ",
            i, pc->red, pc->green, pc->blue);
        pc = &rampmap[defaultpal[i].rampindex].color;
        printf("RGB ramp entry %3ld (%3d, %3d, %3d)\n",
            defaultpal[i].rampindex, pc->red, pc->green, pc->blue);
    }
}
printf("\n");

/* Print out code to initialize a logical palette
 * that will not overflow */

printf("Here is code you can use to create a logical palette\n");

printf("static struct {\n");
printf("    WORD        palVersion;\n");
printf("    WORD        palNumEntries;\n");
printf("    PALETTEENTRY palPalEntries[256];\n");
printf("} rgb8palette = {\n");
printf("    0x300,\n");
printf("    256,\n");

for (i=0; i<256; i++) {
    if (rampmap[i].flags == 0)
        pc = &rampmap[i].color;
    else
        pc = &defaultpal[rampmap[i].defaultindex].color;

    printf("    %3d, %3d, %3d,   0,    /* %ld",
```

```
                pc->red, pc->green, pc->blue, i);
        if (rampmap[i].flags == EXACTMATCH)
            printf(" - Exact match with default %d",
                        rampmap[i].defaultindex);
        if (rampmap[i].flags == CHANGED)
            printf(" - Changed to match default %d",
                        rampmap[i].defaultindex);
        printf(" */\n");
    }

    printf("};\n");
    printf("\n    * * *\n\n");
    printf("    hpal = CreatePalette((LOGPALETTE *)&rgb8palette);\n");

    return 0;
}
```

Color-Indexed Mode and Palette Management

The color-index mode specifies colors in a logical palette with an index to a specific logical-palette entry. Most GDI programs use color-index palettes, but the RGBA mode works better for OpenGL for several effects, such as shading, lighting, fog, and texture mapping. If having the truest color isn't critical for your OpenGL application, you might choose to use the color-index mode (for example, a topographic map that uses "false color" to emphasize the elevation gradient).

Color-Index Mode Palette Sample

The following code sets up a **PIXELFORMATDESCRIPTOR** that sets the flag of the *iPixelType* field to PFD_TYPE_COLORINDEX. This specifies that the application use a color-index palette.

```
BOOL bSetupPixelFormat(HDC hdc)
{
    PIXELFORMATDESCRIPTOR pfd, *ppfd;
    int pixelformat;

    ppfd = &pfd;

    ppfd->nSize = sizeof(PIXELFORMATDESCRIPTOR);
    ppfd->nVersion = 1;
    ppfd->dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                                PFD_DOUBLEBUFFER;
    ppfd->dwLayerMask = PFD_MAIN_PLANE;

     /* Set to color-index mode and use the default color palette. */
    ppfd->iPixelType = PFD_TYPE_COLORINDEX;

    ppfd->cColorBits = 8;
    ppfd->cDepthBits = 16;
    ppfd->cAccumBits = 0;
    ppfd->cStencilBits = 0;

    pixelformat = ChoosePixelFormat(hdc, ppfd);

    if ( (pixelformat = ChoosePixelFormat(hdc, ppfd)) == 0 )
    {
        MessageBox(NULL, "ChoosePixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    if (SetPixelFormat(hdc, pixelformat, ppfd) == FALSE)
    {
        MessageBox(NULL, "SetPixelFormat failed", "Error", MB_OK);
        return FALSE;
    }

    return TRUE;
}
```

Overlay, Underlay, and Main Planes

In other environments OpenGL may support drawing in several layers of planes. In Microsoft's implementation of OpenGL in Windows NT, all drawing occurs on one layer: the main plane. There is no support for overlay or underlay planes.

Sharing Display Lists

When you create a rendering context, it has its own display-list space. The **wglShareLists** function enables a rendering context to share the display-list space of another rendering context. Any number of rendering contexts can share a single display-list space.

Extending OpenGL Functions

The OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context aren't necessarily available in a different rendering context. For a given rendering context in an application using extension functions, use the function addresses returned by **wglGetProcAddress** only.

GLX and WGL/Win32

Some of the WGL functions, new Win32 functions, and other Win32 functions are more or less analogous to GLX X-windows functions. The following list shows some of the relationships.

GLX Functions	WGL/Win32 Functions
glXChooseVisual	**ChoosePixelFormat**
glXCopyContext	---
glXCreateContext	**wglCreateContext**
glXCreateGLXPixmap	**CreateDIBitmap**/ **CreateDIBSection**
glXDestroyContext	**wglDeleteContext**
glXDestroyGLXPixmap	**DeleteObject**
glXGetConfig	**DescribePixelFormat**
glXGetCurrentContext	**wglGetCurrentContext**
glXGetCurrentDrawable	**wglGetCurrentDC**
glXIsDirect	---
glXMakeCurrent	**wglMakeCurrent**
glXQueryExtension	**GetVersion**
glXQueryVersion	**GetVersion**
glXSwapBuffers	**SwapBuffers**
glXUseXFont	**wglUseFontBitmaps**/ **wglUseFontOutlines**
glXWaitGL	**---**
glXWaitX	**---**
XGetVisualInfo	**GetPixelFormat**
XCreateWindow	**CreateWindow**/**CreateWindowEx** and **GetDC**/**BeginPaint**
XSync	**GdiFlush**
---	**SetPixelFormat**
---	**wglGetProcAddress**
---	**wglShareLists**

For further details, refer to the *Porting Guide*.

Using OpenGL on Windows NT

The following topics explain how to use several Windows NT - specific features of Microsoft's implementation of OpenGL in Windows NT.

Header Files

Applications that use the core OpenGL functions must include the header file <GL\GL.H>.

Applications that use the OpenGL utility library must include the header file <GL\GLU.H>.

Applications that use the OpenGL Programming Guide Auxiliary Library must include the header file <GL\GLAUX.H>.

Applications that use the WGL functions must include the header file WINDOWS.H.

Applications that use the new Win32 functions that support Microsoft's implementation of OpenGL in Windows NT must include the header file WINDOWS.H.

Some Pixel Format Tasks

Before an application creates an OpenGL rendering context, it must set a device context pixel format.

Choosing and Setting a Best-Match Pixel Format

An application can obtain a device context's best match to a pixel format by specifying the desired pixel format in a **PIXELFORMATDESCRIPTOR** data structure, then calling the **ChoosePixelFormat** function. That function returns a pixel format index; which an application can then pass to **SetPixelFormat** to set the best match as the device context's current pixel format. The following is a sample code fragment:

```
PIXELFORMATDESCRIPTOR pfd = {
  sizeof(PIXELFORMATDESCRIPTOR),   // size of this pfd
  1,                               // version number
  PFD_DRAW_TO_WINDOW |             // support window
  PFD_SUPPORT_OPENGL |             // support OpenGL
  PFD_DOUBLEBUFFER,                // double buffered
  PFD_TYPE_RGBA,                   // RGBA type
  24,                              // 24-bit color depth
  0, 0, 0, 0, 0, 0,                // color bits ignored
  0,                               // no alpha buffer
  0,                               // shift bit ignored
  0,                               // no accumulation buffer
  0, 0, 0, 0,                      // accum bits ignored
  32,                              // 32-bit z-buffer
  0,                               // no stencil buffer
  0,                               // no auxiliary buffer
  PFD_MAIN_PLANE,                  // main layer
  0,                               // reserved
  0, 0, 0                          // layer masks ignored
};
HDC  hdc;
int  iPixelFormat;

// get the device context's best-available-match pixel format
iPixelFormat = ChoosePixelFormat(hdc, &pfd);

// make that the device context's current pixel format
SetPixelFormat(hdc, iPixelFormat, &pfd);
```

Examining A Device Context's Current Pixel Format

An application uses the **GetPixelFormat** and **DescribePixelFormat** functions to examine a device context's current pixel format. Here is a code fragment:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

// if the DC has a current pixel format ...
if (iPixelFormat = GetPixelFormat(hdc)) {

     // obtain a detailed description of that pixel format
     DescribePixelFormat(hdc, iPixelFormat,
                              sizeof(PIXELFORMATDESCRIPTOR), &pfd);
     }
```

Examining A Device's Supported Pixel Formats

The **DescribePixelFormat** function obtains device context pixel format data. It also returns an integer that is the device context's maximum pixel format index. The following code fragment shows how an application can use that result to step through and examine the pixel formats supported by a device.

```
// local variables
int                                     iMax ;
PIXELFORMATDESCRIPTOR        pfd;
int                                     iPixelFormat ;

// initialize a pixel format index variable
iPixelFormat = 1;

// keep obtaining and examining pixel format data ...
do {
     // try to obtain some pixel format data
     iMax = DescribePixelFormat(hdc, iPixelFormat, sizeof(pfd), &pfd);

     // if there was some problem with that...
     if (iMax == 0)

           // return indicating failure
           return(FALSE);

     // we have successfully obtained pixel format data

     // let's examine the pixel format data...
     myPixelFormatExaminer (&pfd);
     }

// ...until we've looked at all the device context's pixel formats
while (++iPixelFormat <= iMax);
```

Some Rendering Context Tasks

All calls pass through a rendering context. After an application sets a device context's pixel format, it can create a rendering context.

Creating a Rendering Context and Making It Current

The following code fragment shows how an application might create an OpenGL rendering context in response to a WM_CREATE message. Notice that the pixel format is set up (details not included in this code fragment) before creating the rendering context. Also note that in this scenario the device context is not released locally; the application will release it when the window is closed, after the rendering context is made not current. See **Deleting a Rendering Context**. Finally, notice that an application can use local variables for the device context and rendering context handles, since **wglGetCurrentContext** and **wglGetCurrentDC** let an application obtain handles to those contexts as needed.

```
// a window has been created, but is not yet visible
case WM_CREATE:
     {
     // local variables
     HDC         hdc ;
     HGLRC hglrc ;

     // obtain a device context for the window
     hdc = GetDC(hWnd);

     // set an appropriate pixel format
     myPixelFormatSetupFunction(hdc);

     // if we can create a rendering context ...
     if (hglrc = wglCreateContext( hdc ) ) {

           // try to make it the thread's current rendering context
           bHaveCurrentRC = wglMakeCurrent(hdc, hglrc) ;

           }

     // perform miscellaneous other WM_CREATE chores ....

     }
     break ;
```

Making a Rendering Context Not Current

An application detaches a rendering context from a thread by making it not current. You can do this by calling **wglMakeCurrent** with the parameters set to NULL. The following is a sample of this simple task:

```
// detach the current rendering context from the thread
wglMakeCurrent(NULL, NULL);
```

Deleting a Rendering Context

The following code fragment shows how an application might delete an OpenGL rendering context when an OpenGL window is closed. It is a continuation of the scenario used in **Creating a Rendering Context and Making It Current**:

```
// a window is about to cease its glorious OpenGL existence
case WM_DESTROY:
     {
     // local variables
     HGLRC       hglrc;
     HDC         hdc ;

     // if the thread has a current rendering context ....
     if(hglrc = wglGetCurrentContext()) {

           // obtain its associated device context
           hdc = wglGetCurrentDC() ;

           // make the rendering context not current
           wglMakeCurrent(NULL, NULL) ;

           // release the device context
           ReleaseDC (hwnd, hdc) ;

           // nuke the rendering context
         wglDeleteContext(hglrc);

           }
```

Drawing with Double Buffers

An application uses double buffers to smooth the transition between images. Swapping buffers typically comes at the end of a sequence of drawing commands. By default, Microsoft's implementation of OpenGL in Windows NT draws to the off-screen buffer; when drawing is complete, an application can copy the off-screen buffer to the on-screen buffer by calling the **SwapBuffers** function. In the following code fragment, an application prepares to draw, calls a drawing function, and then copies the completed image on-screen if it is using double buffering:

```
void myRedraw(void)
{
      // set up for drawing commands
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(45, 1.0, 0.1, 100.0);

      // draw our objects
      myDrawAllObjects(GL_FALSE);

      // if we're double-buffering ...
      if (bDoubleBuffering)

            // ...copy the image on-screen
            SwapBuffers(hdc);
}
```

Here is another example. An application obtains a window device context, renders a scene, copies the image on-screen (to show the rendering), and then releases the device context:

```
  hdc = GetDC(hwnd);
  mySceneRenderingFunction();
  SwapBuffers(hdc);
  ReleaseDC(hWnd, hdc);
```

Drawing Text in a Double-Buffered OpenGL Window

As previously mentioned, an application draws text in a double-buffered OpenGL window by creating display lists for selected characters in a font, then executing the appropriate display list for each character it wishes to draw. In the following sample fragment, an application creates a rendering context, draws a red triangle, and then labels it with text. The sample assumes that the application has obtained a device context and set up a font and pixel format for it.

```
// we enter with a device context, font, and pixel format

// create an OpenGL rendering context
hglrc = wglCreateContext(hdc);

// make it this thread's current rendering context
wglMakeCurrent(hdc, hglrc);

// we'll be clearing to a lovely and quiescent deep blue hue
glClearColor(0.0F, 0.0F, 0.4F, 1.0F);

// we want smooth shading
glShadeModel(GL_SMOOTH);

// clear the color buffers
glClear(GL_COLOR_BUFFER_BIT);

// specify a red triangle
glBegin(GL_TRIANGLES);
    glColor3f(1.0F, 0.0F, 0.0F);
    glVertex2f(10.0F, 10.0F);
    glVertex2f(250.0F, 50.0F);
    glVertex2f(105.0F, 280.0F);
glEnd();

// create bitmaps for the device context's font's first 256 glyphs
wglUseFontBitmaps(hdc, 0, 256, 1000);

// move bottom left, southwest of the red triangle
glRasterPos2f(30.0F, 300.0F);

// set up for a string-drawing display list call
glListBase(1000);

// draw a string using font display lists
glCallLists(12, GL_UNSIGNED_BYTE, "Red Triangle");

// get all those commands to execute
glFlush();

// delete our 256 glyphic display lists
glDeleteLists(1000, 256) ;

// make the rendering context not current
wglMakeCurrent (NULL, NULL) ;

// release the device context
```

```
ReleaseDC(hdc) ;

// delete the rendering context
wglDeleteContext(hglrc);
```

Printing an OpenGL Image

The current version of Microsoft's implementation of OpenGL in Windows NT provides direct support for printing on color printers that provide four or more bits of color information per pixel. To print on such a device, an application calls **wglCreateContext** with a printer device context to create a printing rendering context.

The current version of Microsoft's implementation of OpenGL in Windows NT does not provide direct support for printing on monochrome printers. More specifically: an application cannot call **wglCreateContext** with a monochrome printer device context. There is a workaround for this situation, as illustrated in **Printing on a Monochrome Printer**.

Printing on a Monochrome Printer

There is an indirect way to print an image on a monochrome printer. The main points of the algorithm are as follows:

1. Create a printer device context.
2. Create a memory device context compatible with that printer device context.
3. Create a rendering context, passing the **wglCreateContext** function the handle to the memory device context.
4. Make that rendering context a thread's current rendering context.
5. Make OpenGL calls, which will draw into that rendering context.
6. Disconnect and delete the rendering context.
7. Bitblt from the memory device context's bitmap to the printer device context's bitmap, banding to conserve memory.
8. Delete the memory device context.
9. Delete the printer device context.

Copying an OpenGL Image to the Clipboard

Although the current version of Microsoft's implementation of OpenGL in Windows NT does not directly support the Clipboard, an application can copy a Windows OpenGL image to the Clipboard by following these steps:

1. Draw the image to a memory bitmap.
2. Copy that bitmap to the Clipboard.

Multi-Threaded OpenGL Drawing Strategies

The GDI does not support multiple threads. Your OpenGL application must use a distinct device context and a distinct rendering context (RC) for each thread. This tends to limit the performance advantages of using multiple threads with single-processor systems running OpenGL applications. However, there are ways to use threads with a single processor system to greatly increase performance. For example, your application can use a separate thread to pass OpenGL rendering calls to dedicated 3D hardware.

Symmetric multi-processing (SMP) systems can greatly benefit from using multiple threads. An obvious strategy is to use a separate thread for each processor to handle OpenGL rendering in separate windows. For example, in a flight simulation application you could use a separate processor and thread to render the front, back, and side views.

A thread can have one current, active rendering context only. When your application uses multiple threads and multiple rendering contexts, you must be careful to synchronize their use. For example, use one thread only to call **SwapBuffers** after all threads complete drawing.

Using the Auxiliary Library

Using OpenGL, SGI created the Auxiliary Library to write simple sample programs for the "OpenGL Programming Guide" (the "red book"). The source code for the auxiliary library is supplied with the SDK along with the OpenGL samples. You can examine the source code to help you understand how the Auxiliary Library was developed from OpenGL functions and routines, and you can use auxiliary library functions in your own programs. For a description of the Auxiliary Library, refer to the "OpenGL Programming Guide."

Lists of Functions and Structures

WGL Functions

- wglCreateContext
- wglDeleteContext
- wglGetCurrentContext
- wglGetCurrentDC
- wglGetProcAddress
- wglMakeCurrent
- wglShareLists
- wglUseFontBitmaps
- wglUseFontOutlines

Win32 Functions

- ChoosePixelFormat
- DescribePixelFormat
- GetPixelFormat
- SetPixelFormat
- SwapBuffers

Structures

- GLYPHMETRICSFLOAT
- PIXELFORMATDESCRIPTOR
- POINTFLOAT

ChoosePixelFormat

The **ChoosePixelFormat** function attempts to find the pixel format supported by a device context that is the best match to a given pixel format specification.

```
int ChoosePixelFormat(
  HDC   hdc,                                //Device context to search for a best pixel format match
  CONST PIXELFORMATDESCRIPTOR *             //Pixel format for which a best match is sought
ppfd
  );
```

Parameters

hdc

Specifies the device context that the function shall examine for a pixel format that best matches the pixel format descriptor pointed to by *ppfd*.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure that specifies the requested pixel format. In this particular context, the members of the pointed-to **PIXELFORMATDESCRIPTOR** structure are used as follows:

nSize

Specifies the size of the **PIXELFORMATDESCRIPTOR** data structure. Set this member to **sizeof**(**PIXELFORMATDESCRIPTOR**).

nVersion

Specifies the version number of the **PIXELFORMATDESCRIPTOR** data structure. Set this member to 1.

dwFlags

A set of bit flags that specify properties of the pixel buffer. An application can combine the following bit flag constants by bitwise-ORing.

If any of the following flags is set, the function attempts to choose a pixel format that also has that flag set. Otherwise, it ignores that flag in the pixel formats:

PFD_DRAW_TO_WINDOW
PFD_DRAW_TO_BITMAP
PFD_SUPPORT_GDI
PFD_SUPPORT_OPENGL

If any of the following flags are set, the function attempts to match pixel formats with that flag set. Otherwise, it attempts to match pixel formats without that flag set:

PFD_DOUBLEBUFFER
PFD_STEREO

If the following flag is set, the function ignores the PFD_DOUBLEBUFFER flag in the pixel formats:

PFD_DOUBLEBUFFER_DONTCARE

If the following flag is set, the function ignores the PFD_STEREO flag in the pixel formats:

PFD_STEREO_DONTCARE

iPixelType

The function only considers pixel formats of the specified type. It is one of the following types:

PFD_TYPE_RGBA
PFD_TYPE_COLORINDEX

cColorBits

0 or greater.

cRedBits

Not used.

cRedShift
Not used.

cGreenBits
Not used.

cGreenShift
Not used.

cBlueBits
Not used.

cBlueShift
Not used.

cAlphaBits
0 or greater.

cAlphaShift
Not used.

cAccumBits
0 or greater.

cAccumRedBits
Not used.

cAccumGreenBits
Not used.

cAccumBlueBits
Not used.

cAccumAlphaBits
Not used.

cDepthBits
0 or greater.

cStencilBits
0 or greater.

cAuxBuffers
0 or greater.

iLayerType
One of the following values:

PFD_MAIN_PLANE
PFD_OVERLAY_PLANE
PFD_UNDERLAY_PLANE

bReserved
Not used.

dwLayerMask
Not used.

dwVisibleMask
Not used.

dwDamageMask
Not used.

Return Value

If the function succeeds, the return value is a pixel format index (one-based) that is the closest match to the specification of the pixel format descriptor.

If the function fails, the return value is zero. Call **GetLastError** for extended error information.

Remarks

It is an application's responsibility to ensure that the pixel format selected by the **ChoosePixelFormat** function satisfies its requirements. For example, if an application requests a pixel format with a 24-bit RGB color buffer, but the device context offers only 8-bit RGB color buffers, then the function returns a pixel format with an 8-bit RGB color buffer.

Here is a code fragment sample:

```
PIXELFORMATDESCRIPTOR pfd = {
     sizeof(PIXELFORMATDESCRIPTOR),  // size of this pfd
     1,                     // version number
     PFD_DRAW_TO_WINDOW |   // support window
     PFD_SUPPORT_OPENGL |   // support OpenGL
     PFD_DOUBLEBUFFER,      // double buffered
     PFD_TYPE_RGBA,         // RGBA type
     24,                    // 24-bit color depth
     0, 0, 0, 0, 0, 0,      // color bits ignored
     0,                     // no alpha buffer
     0,                     // shift bit ignored
     0,                     // no accumulation buffer
     0, 0, 0, 0,            // accum bits ignored
     32,                    // 32-bit z-buffer
     0,                     // no stencil buffer
     0,                     // no auxiliary buffer
     PFD_MAIN_PLANE,        // main layer
     0,                     // reserved
     0, 0, 0                // layer masks ignored
     };
     HDC  hdc;
     int  iPixelFormat;

iPixelFormat = ChoosePixelFormat(hdc, &pfd);
```

See Also

DescribePixelFormat, **GetPixelFormat**, **SetPixelFormat**

DescribePixelFormat

The **DescribePixelFormat** function obtains information about the pixel format identified by *iPixelFormat* of the device associated with *hdc*. The function sets the members of the **PIXELFORMATDESCRIPTOR** structure pointed to by *ppfd* with that pixel format information.

```
int DescribePixelFormat(
    HDC    hdc,                          //Device context of interest
    int    iPixelFormat,                 //Pixel format selector
    UINT   nBytes,                       //Size of buffer pointed to by ppfd
    LPPIXELFORMATDESCRIPTOR              //Pointer to structure to receive pixel format data
ppfd
  );
```

Parameters

hdc
Specifies the device context of interest.

iPixelFormat
Index that specifies the pixel format of interest. The pixel formats that a device context supports are identified by positive one-based integer indices.

nBytes
Specifies the size, in bytes, of the structure pointed to by *ppfd*. The function stores data to that structure; it stores no more than *nBytes* bytes. Set this value to **sizeof**(**PIXELFORMATDESCRIPTOR**).

ppfd
Pointer to a **PIXELFORMATDESCRIPTOR** structure whose members the function sets with pixel format data. The function stores the number of bytes copied to the structure in the structure's **nSize** field. If, upon entry, *ppfd* is NULL, the function writes no data to the structure. This is useful when an application only wants to obtain a device context's maximum pixel format index.

Return Value

If the function succeeds, the return value is the device context's maximum pixel format index. In addition, the function sets the members of the **PIXELFORMATDESCRIPTOR** structure pointed to by *ppfd* according to the specified pixel format.

If the function fails, the return value is zero. Call **GetLastError** for extended error information.

Remarks

Here's a skeletal example of **DescribePixelFormat** usage:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

iPixelFormat = 1;

// obtain detailed information about
// the device context's 1st pixel format
DescribePixelFormat(hdc, iPixelFormat,
        sizeof(PIXELFORMATDESCRIPTOR), &pfd);
```

See Also

ChoosePixelFormat, **GetPixelFormat**, **SetPixelFormat**

GetPixelFormat

The **GetPixelFormat** function obtains the index of the specified device context's currently selected pixel format.

int GetPixelFormat(
HDC //Device context whose currently selected pixel format index is
hdc sought
);

Parameter

hdc
Specifies the device context whose currently selected pixel format index the function returns.

Return Value

If the function succeeds, the return value is the specified device context's currently selected pixel format index. This is a positive, one-based index value.

If the function fails, the return value is zero. Call **GetLastError** for extended error information.

Remarks

Here is a skeletal example of **GetPixelFormat** usage:

```
PIXELFORMATDESCRIPTOR  pfd;
HDC  hdc;
int  iPixelFormat;

// get the current pixel format index
iPixelFormat = GetPixelFormat(hdc);

// obtain a detailed description of that pixel format
DescribePixelFormat(hdc, iPixelFormat,
        sizeof(PIXELFORMATDESCRIPTOR), &pfd);
```

See Also

ChoosePixelFormat, **DescribePixelFormat**, **SetPixelFormat**

PIXELFORMATDESCRIPTOR

```
typedef struct tagPIXELFORMATDESCRIPTOR { // pfd
    WORD  nSize;
    WORD  nVersion;
    DWORD dwFlags;
    BYTE  iPixelType;
    BYTE  cColorBits;
    BYTE  cRedBits;
    BYTE  cRedShift;
    BYTE  cGreenBits;
    BYTE  cGreenShift;
    BYTE  cBlueBits;
    BYTE  cBlueShift;
    BYTE  cAlphaBits;
    BYTE  cAlphaShift;
    BYTE  cAccumBits;
    BYTE  cAccumRedBits;
    BYTE  cAccumGreenBits;
    BYTE  cAccumBlueBits;
    BYTE  cAccumAlphaBits;
    BYTE  cDepthBits;
    BYTE  cStencilBits;
    BYTE  cAuxBuffers;
    BYTE  iLayerType;
    BYTE  bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

The **PIXELFORMATDESCRIPTOR** structure describes the pixel format of a drawing surface.

Members

nSize
Specifies the size of this data structure. This value should be set to **sizeof**(**PIXELFORMATDESCRIPTOR**).

nVersion
Specifies the version of this data structure. This value should be set to 1.

dwFlags
A set of bit flags that specify properties of the pixel buffer. The properties are generally not mutually exclusive; an application can set any combination of bit flags, with the exceptions noted. The following bit flag constants are defined:

Value	Meaning
PFD_DRAW_TO_WINDOW	The buffer can draw to a window or device surface.
PFD_DRAW_TO_BITMAP	The buffer can draw to a memory bitmap.
PFD_SUPPORT_GDI	The buffer supports GDI drawing. This flag and PFD_DOUBLEBUFFER are mutually exclusive in the current generic implementation.

PFD_SUPPORT_OPENGL	The buffer supports OpenGL drawing.
PFD_GENERIC_FORMAT	The pixel format is supported by the GDI software implementation. That implementation is also known as the generic implementation. If this bit is clear, the pixel format is supported by a device driver or hardware.
PFD_NEED_PALETTE	The buffer uses RGBA pixels on a palette-managed device. A logical palette is required to achieve the best results for this pixel type. Colors in the palette should be specified according to the values of the **cRedBits**, **cRedShift**, **cGreenBits**, **cGreenShift**, **cBluebits**, and **cBlueShift** members. The palette should be created and realized in the device context (DC) before calling **wglMakeCurrent**.
PFD_NEED_SYSTEM_PALETTE	Used with systems with OpenGL hardware that supports one hardware palette only. For such systems to use hardware acceleration, the hardare palette must be in a fixed order (for example, 3-3-2) when in RGBA mode or must match the logical palette when in color-index mode. When you set this flag, you should call **SetSystemPaletteUse** in your program to force a one-to-one mapping of the logical palette and the system palette. If your OpenGL harware supports multiple hardware palettes and the device driver can allocate spare hardware palettes for OpenGL, you don't need to set PFD_NEED_SYSTEM_PALETTE. This flag is not set in the generic pixel formats.
PFD_DOUBLEBUFFER	The buffer is double-buffered. This flag and PFD_SUPPORT_GDI are mutually exclusive in the current generic implementation.
PFD_STEREO	The buffer is stereoscopic. This flag is not supported in the current generic implementation.

In addition, the following bit flags can be specified when calling **ChoosePixelFormat**:

Value	**Meaning**
PFD_DOUBLE_BUFFER_DONTCARE	The requested pixel format can be either single- or double-buffered.
PFD_STEREO_DONTCARE	The requested pixel format can be either monoscopic or stereoscopic.

With the **glAddSwapHintRectWIN** extension function two new flags are included for the

PIXELFORMATDESCRIPTOR pixel format structure:

Value	Meaning
PFD_SWAP_COPY	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the back-buffer content to be copied to the front buffer. The content of the back buffer is not affected by the swap. **PFD_SWAP_COPY** is a hint only and might not be provided by a driver.
PFD_SWAP_EXCHANGE	Specifies the content of the back buffer in the double-buffered main color plane following a buffer swap. Swapping the color buffers causes the exchange of back-buffer content with the front-buffer content. Following the swap, the back-buffer content contains the front-buffer content before the swap. **PFD_SWAP_EXCHANGE** is a hint only and might not be provided by a driver.

iPixelType

Specifies the type of pixel data. The following types are defined:

Value	Meaning
PFD_TYPE_RGBA	RGBA pixels. Each pixel has four components: red, green, blue, and alpha.
PFD_TYPE_COLORINDEX	Color index pixels. Each pixel uses a color index value

cColorBits

Specifies the number of color bitplanes in each color buffer. For RGBA pixel types, it is the size of the color buffer excluding the alpha bitplanes. For color index pixels, it is the size of the color index buffer.

cRedBits

Specifies the number of red bitplanes in each RGBA color buffer.

cRedShift

Specifies the shift count for red bitplanes in each RGBA color buffer.

cGreenBits

Specifies the number of green bitplanes in each RGBA color buffer.

cGreenShift

Specifies the shift count for green bitplanes in each RGBA color buffer.

cBlueBits

Specifies the number of blue bitplanes in each RGBA color buffer.

cBlueShift

Specifies the shift count for blue bitplanes in each RGBA color buffer.

cAlphaBits

Specifies the number of alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAlphaShift
Specifies the shift count for alpha bitplanes in each RGBA color buffer. Alpha bitplanes are not supported.

cAccumBits
Specifies the total number of bitplanes in the accumulation buffer.

cAccumRedBits
Specifies the number of red bitplanes in the accumulation buffer.

cAccumGreenBits
Specifies the number of green bitplanes in the accumulation buffer.

cAccumBlueBits
Specifies the number of blue bitplanes in the accumulation buffer.

cAccumAlphaBits
Specifies the number of alpha bitplanes in the accumulation buffer.

cDepthBits
Specifies the depth of the depth (z-axis) buffer.

cStencilBits
Specifies the depth of the stencil buffer.

cAuxBuffers
Specifies the number of auxiliary buffers. Auxiliary buffers are not supported.

iLayerType
Specifies the type of layer. Although the following values are defined, the current version supports only the main plane (there is no support for overlay or underlay planes):

Value	Meaning
PFD_MAIN_PLANE	The layer is the main plane.
PFD_OVERLAY_PLANE	The layer is the overlay plane.
PFD_UNDERLAY_PLANE	The layer is the underlay plane.

bReserved
Not used. Must be zero.

dwLayerMask
Specifies the layer mask. The layer mask is used in conjunction with the visible mask to determine if one layer overlays another.

dwVisibleMask
Specifies the visible mask. The visible mask is used in conjunction with the layer mask to determine if one layer overlays another. If the result of the bitwise-AND of the visible mask of a layer and the layer mask of a second layer is nonzero, then the first layer overlays the second layer, and a transparent pixel value exists between the two layers. If the visible mask is 0, the layer is opaque.

dwDamageMask
Specifies whether more than one pixel format shares the same frame buffer. If the result of the bitwise-AND of the damage masks between two pixel formats is non-zero, then they share the same buffers.

Remarks

Please notice carefully, as documented above, that certain pixel format properties are not supported in the current generic implementation. The generic implementation is the Microsoft GDI software implementation of OpenGL. Hardware manufacturers may enhance parts of OpenGL, and may support some pixel format properties not supported by the generic implementation.

See Also

ChoosePixelFormat, **DescribePixelFormat**, **GetPixelFormat**, **SetPixelFormat**

SetPixelFormat

The **SetPixelFormat** function sets the specified device context's pixel format to the format specified by the *iPixelFormat* index.

```
BOOL SetPixelFormat(
   HDC   hdc,                              //Device context whose pixel format the function attempts to set
   int   iPixelFormat,                     //Pixel format index (one-based)
   CONST PIXELFORMATDESCRIPTOR *           //Pointer to logical pixel format specification
ppfd
  );
```

Parameters

hdc

Specifies the device context whose pixel format the function attempts to set.

iPixelFormat

Index that identifies the pixel format to set. The various pixel formats supported by a device context are identified by one-based indices.

ppfd

Pointer to a **PIXELFORMATDESCRIPTOR** structure that contains the logical pixel format specification. The system's metafile component uses this structure to record the logical pixel format specification. The structure has no other effect upon the behavior of the **SetPixelFormat** function.

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** for extended error information.

Remarks

If *hdc* references a window, calling the **SetPixelFormat** function also changes the pixel format of the window. Setting the pixel format of a window more than once can lead to significant complications for the window manager and for multithreaded applications, so it is not allowed. An application may only set the pixel format of a window one time. Once a window's pixel format is set, it cannot be changed.

An application should select a pixel format in the device context before calling the **wglCreateContext** function. The **wglCreateContext** function creates a rendering context for drawing on the device in the device context's selected pixel format.

An OpenGL window has its own pixel format. Because of this, only device contexts retrieved for the client area of an OpenGL window are allowed to draw into the window. As a result, an OpenGL window should be created with the WS_CLIPCHILDREN and WS_CLIPSIBLINGS styles. Additionally, the window class attribute should not include the CS_PARENTDC style.

Here is a skeletal example of **SetPixelFormat** usage:

```
PIXELFORMATDESCRIPTOR pfd = {
     sizeof(PIXELFORMATDESCRIPTOR),   // size of this pfd
     1,                     // version number
     PFD_DRAW_TO_WINDOW |   // support window
     PFD_SUPPORT_OPENGL |   // support OpenGL
     PFD_DOUBLEBUFFER,      // double buffered
     PFD_TYPE_RGBA,         // RGBA type
     24,                    // 24-bit color depth
     0, 0, 0, 0, 0, 0,      // color bits ignored
     0,                     // no alpha buffer
     0,                     // shift bit ignored
```

```
    0,                    // no accumulation buffer
    0, 0, 0, 0,           // accum bits ignored
    32,                   // 32-bit z-buffer
    0,                    // no stencil buffer
    0,                    // no auxiliary buffer
    PFD_MAIN_PLANE,       // main layer
    0,                    // reserved
    0, 0, 0               // layer masks ignored
};
HDC  hdc;
int  iPixelFormat;

// get the device context's best-available-match pixel format
iPixelFormat = ChoosePixelFormat(hdc, &pfd);

// make that the device context's current pixel format
SetPixelFormat(hdc, iPixelFormat, &pfd);
```

See Also

ChoosePixelFormat, DescribePixelFormat, GetPixelFormat

SwapBuffers

The **SwapBuffers** function exchanges the front and back buffers if the current pixel format for the window referenced by the specified device context includes a back buffer.

BOOL SwapBuffers(
  HDC //Device context whose buffers get swapped
hdc
);

Parameter

hdc
Specifies a device context. If the current pixel format for the window referenced by this device context includes a back buffer, the function exchanges the front and back buffers.

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** for extended error information.

Remarks

If the current pixel format for the window referenced by the device context does not include a back buffer, then this call has no effect. The content of the back buffer is undefined when the function returns.

A multithreaded application should flush the drawing commands in any other threads drawing to the same window before calling the **SwapBuffers** function.

wglCreateContext

The **wglCreateContext** function creates a new OpenGL rendering context. The rendering context is suitable for drawing on the device referenced by *hdc*. The rendering context has the same pixel format as the device context.

```
HGLRC wglCreateContext(
  HDC   hdc    //Device context of device that the rendering context will be
               suitable for
);
```

Parameter

hdc

Handle to a device context. The function creates an OpenGL rendering context suitable for the device the device context references.

Return Value

If the function succeeds, the return value is a valid handle to an OpenGL rendering context.

If the function fails, the return value is NULL. Call **GetLastError** for extended error information.

Remarks

A rendering context, or GLRC, is a port through which all OpenGL commands pass. Every thread that makes OpenGL calls needs to have a current OpenGL rendering context.

A rendering context is not the same as a device context. A device context contains information pertinent to GDI, and a rendering context contains information pertinent to OpenGL.

An application should set the device context's pixel format before creating a rendering context. See the **SetPixelFormat** function.

To use OpenGL, an application creates a rendering context, selects it as a thread's current rendering context, then calls OpenGL functions. When the application is finished with the rendering context, it disposes of it by calling **wglDeleteContext**. Here's a code example:

```
// The barest of OpenGL skeletons

HDC  hdc;
HGLRC  hglrc;

// create a rendering context
hglrc = wglCreateContext (hdc);

// make it the calling thread's current rendering context
wglMakeCurrent (hdc, hglrc);

// call OpenGL APIs as desired ...
.
.
.
// when the rendering context is no longer needed ...

// make the rendering context not current
wglMakeCurrent (NULL, NULL) ;

// delete the rendering context
```

```
wglDeleteContext (hglrc);
```

See Also

SetPixelFormat, wglDeleteContext, wglGetCurrentContext, wglGetCurrentDC, wglMakeCurrent

wglDeleteContext

The **wglDeleteContext** function deletes a specified OpenGL rendering context.

```
BOOL wglDeleteContext(
  HGLRC   hglrc    //Handle to the OpenGL rendering context to delete
);
```

Parameter

hglrc
Handle to an OpenGL rendering context that the function is to delete.

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** for extended error information.

Remarks

It is an error to delete an OpenGL rendering context that is another thread's current context. However, if a rendering context is the calling thread's current context, the function makes the rendering context not current before deleting it.

The **wglDeleteContext** function does not delete the device context associated with the OpenGL rendering context via the **wglMakeCurrent** function. After calling **wglDeleteContext**, an application should call **DeleteDC** to delete the associated device context.

See Also

wglCreateContext, **wglGetCurrentContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglGetCurrentContext

The **wglGetCurrentContext** function obtains a handle to the calling thread's current OpenGL rendering context.

HGLRC wglGetCurrentContext(VOID**);**

Parameter

This function has no parameters.

Return Value

If the calling thread has a current OpenGL rendering context, the function returns a handle to that rendering context. Otherwise, the function return value is NULL.

Remarks

A thread's current OpenGL rendering context is associated with a device context via the **wglMakeCurrent** function. An application can use the **wglGetCurrentDC** function to obtain a handle to the device context associated with the current OpenGL rendering context.

See Also

wglCreateContext, **wglDeleteContext**, **wglGetCurrentDC**, **wglMakeCurrent**

wglGetCurrentDC

The **wglGetCurrentDC** function obtains a handle to the device context that is associated with the calling thread's current OpenGL rendering context.

HDC wglGetCurrentDC(VOID**);**

Parameter

This function has no parameters.

Return Value

If the calling thread has a current OpenGL rendering context, the function returns a handle to the device context associated with that rendering context via the **wglMakeCurrent** function. Otherwise, the function return value is NULL.

Remarks

An application associates a device context with an OpenGL rendering context when it calls the **wglMakeCurrent** function. An application can use the **wglGetCurrentContext** function to obtain a handle to calling thread's current OpenGL rendering context.

See Also

wglCreateContext, **wglDeleteContext**, **wglGetCurrentContext**, **wglMakeCurrent**

wglGetProcAddress

The **wglGetProcAddress** function returns the address of an OpenGL extension function for use with the current OpenGL rendering context.

```
PROC wglGetProcAddress(
  LPCSTR  lpszProc     //Name of the extension function
);
```

Parameter

lpszProc

Points to a null-terminated string that is the name of the extension function. The name of the extension function must be identical to a corresponding function implemented by OpenGL.

Return Value

When the function succeeds, the return value is the address of the extension function. When no current rendering context exists or the function fails, the return value is NULL. Call **GetLastError** for extended error information.

Remarks

The OpenGL library supports multiple implementations of its functions. Extension functions supported in one rendering context are not necessarily available in a separate rendering context. Thus for a given rendering context in an application use the function addresses returned by **wglGetProcAddress** only.

The spelling and the case of the extension function pointed to by *lpszProc* must be identical to that of a function supported and implemented by OpenGL. Because extension functions are not exported by the OpenGL library, you must use **wglGetProcAddress** to get the addresses of vendor-specific extension functions.

The extension function addresses are unique for each pixel format. All rendering contexts of a given pixel format share the same extension function addresses.

See Also

glGetString, **wglMakeCurrent**

wglMakeCurrent

The **wglMakeCurrent** function makes a specified OpenGL rendering context the calling thread's current rendering context. All subsequent OpenGL calls made by the thread are drawn on the device identified by *hdc*. The function can also be used to make the calling thread's current rendering context not current.

```
BOOL wglMakeCurrent(
  HDC   hdc,      //Device context of device that OpenGL calls are to be drawn on
  HGLRC hglrc     //OpenGL rendering context to be made calling thread's current
                  rendering context
);
```

Parameters

hdc

Handle to a device context. Subsequent OpenGL calls made by the calling thread are drawn on the device identified by *hdc*.

hglrc

Handle to an OpenGL rendering context that the function sets as the calling thread's rendering context.

If *hglrc* is NULL, the function makes the calling thread's current rendering context not current, and releases the device context that is used by the rendering context. In this situation, *hdc* is ignored.

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** for extended error information.

Remarks

The *hdc* parameter must refer to a drawing surface supported by OpenGL. It need not be the same *hdc* that was passed to **wglCreateContext** when *hglrc* was created, but it must be on the same device and it must have the same pixel format. GDI transformation and clipping in *hdc* are not supported by the rendering context. The current rendering context uses the *hdc* device context until the rendering context is made not current.

Before switching to the new rendering context, OpenGL flushes any previous rendering context that was current to the calling thread.

A thread can have one current rendering context. A process can have multiple rendering contexts via multi-threading. A thread must set a current rendering context before calling any OpenGL functions. Otherwise, all OpenGL calls are ignored.

A rendering context can be current to only one thread at a time. It is an error to make a rendering context current to multiple threads.

An application can perform multi-threaded drawing by making different rendering contexts current to different threads, supplying each thread with its own rendering context and device context.

If an error occurs, the **wglMakeCurrent** function makes the thread's current rendering context, if any, not current before returning.

See Also

wglCreateContext, **wglDeleteContext**, **wglGetCurrentContext**, **wglGetCurrentDC**

wglShareLists

The **wglShareLists** function enables multiple OpenGL rendering context to share a single display-list space.

BOOL wlgMakeShareLists(
HGLRC //OpenGL rendering context with which to share display lists
hglrc1,
HGLRC *hglrc2* //OpenGL rendering context to share display lists
);

Parameters

hglrc1
Specifies the OpenGL rendering context with which to share display lists.

hglrc2
Specifies the OpenGL rendering context to share display lists with *hglrc1*. *hglrc2* should not contain any existing display lists when **wglShareLists** is called.

Return Value

When the function succeeds, the return value is TRUE.

When the function fails, the return value is FALSE and the display lists are not shared. Call **GetLastError** for extended error information.

Remarks

When you create an OpenGL rendering context, it has its own display-list space. **wlgShareLists** enables a rendering context to share the display-list space of another rendering context and any number of rendering contexts can share a single display-list space. Once a rendering context shares a display-list space, the rendering context always uses the display-list space until the rendering context is deleted. When the last rendering context of a shared display-list space is deleted, the shared display-list spaced is deleted. All the indexes and definitions of display lists in a shared display-list space are shared.

You can only share display lists with rendering contexts within the same process. However, not all rendering contexts in a process can share display lists. Rendering contexts can share display lists if they use the same implementation of OpenGL functions only. All client rendering contexts of a given pixel format can always share display lists.

Note **wlgShareLists** is available with OpenGL version 1.01 or later only. To determine the version number of the implementation of OpenGL, call **glGetString**.

See Also

glGetString

wglUseFontBitmaps

The **wglUseFontBitmaps** function creates a set of bitmap display lists based on the glyphs in a device context's currently selected font for use in the current OpenGL rendering context. These bitmaps can then be used to draw characters in an OpenGL image.

The **wglUseFontBitmaps** function creates *count* display lists, one for each of a run of *count* glyphs that begins with *hdc*'s selected font's *first* glyph.

```
BOOL wglUseFontBitmaps(
    HDC   hdc,        //Device context whose font will be used
  DWORD   first,      //Glyph that is the first of a run of glyphs to be turned into bitmap
                      display lists
  DWORD   count,      //Number of glyphs to turn into bitmap display lists
  DWORD   listBase    //Specifies starting display list
);
```

Parameters

hdc
Specifies the device context whose currently selected font will be used to form the glyph bitmap display lists in the current OpenGL rendering context.

first
Specifies the glyph of the font that is the first element of a run of glyphs that will be used to form glyph bitmap display lists.

count
Specifies the number of glyphs in the run of glyphs that will be used to form glyph bitmap display lists. The function creates *count* display lists, one for each glyph in the run.

listBase
Specifies a starting display list.

Return Value

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. Call **GetLastError** for extended error information.

Remarks

The **wglUseFontBitmaps** function defines *count* display lists in the current OpenGL rendering context. Each display list has an identifying number, starting at *listBase*. Each display list consists of a single call to **glBitmap**. The definition of bitmap *listBase+i* is taken from the glyph *first+i* of the font currently selected in the device context specified by *hdc*. If a glyph is not defined, then the function defines an empty display list for it.

The function creates bitmap text in the plane of the screen. It enables the labeling of objects in OpenGL.

The current version of Microsoft's implementation of OpenGL in Windows NT does not allow GDI calls to be made to a device context whose pixel format is double-buffered. That precludes an application from using the GDI fonts and text functions with such device contexts. The **wglUseFontBitmaps** function lets an application circumvent this limitation and draw text in a double-buffered device context.

The function determines the parameters of each call to **glBitmap** as follows:

glBitmap parameter	Determination
width	The width of the glyph's bitmap, as returned in the *gmBlackBoxX* field of the glyph's **GLYPHMETRICS** structure.

height	The height of the glyph's bitmap, as returned in the *gmBlackBoxY* field of the glyph's **GLYPHMETRICS** structure.
xorig	The x offset of the glyph's origin, as returned in the *gmptGlyphOrigin.x* field of the glyph's **GLYPHMETRICS** structure.
yorig	The y offset of the glyph's origin, as returned in the *gmptGlyphOrigin.y* field of the glyph's **GLYPHMETRICS** structure.
xmove	The horizontal distance to the origin of the next character cell, as returned in the *gmCellIncX* field of the glyph's **GLYPHMETRICS** structure.
ymove	The vertical distance to the origin of the next character cell as returned in the *gmCellIncY* field of the glyph's **GLYPHMETRICS** structure.
bitmap	The bitmap for the glyph, as returned by **GetGlyphOutline** with *uFormat* equal to 1.

Here's a skeletal example that shows how to draw some text:

```
HDC    hdc;
HGLRC    hglrc;

// create a rendering context
hglrc = wglCreateContext (hdc);

// make it the calling thread's current rendering context
wglMakeCurrent (hdc, hglrc);

// now we can call OpenGL API

// make the system font the device context's selected font
SelectObject (hdc, GetStockObject (SYSTEM_FONT));

// create the bitmap display lists
// we're making images of glyphs 0 thru 255
// the display list numbering starts at 1000, an arbitrary choice
wglUseFontBitmaps (hdc, 0, 255, 1000);

// display a string:
// indicate start of glyph display lists
glListBase (1000);
// now draw the characters in a string
glCallLists (24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World");
```

See Also

glListBase, **glCallLists**, **wglUseFontOutlines**

wglUseFontOutlines

The **wglUseFontOutlines** function creates a set of display lists based on the glyphs of the currently selected outline font of a device context for use with the current rendering context. The display lists are used to draw 3D characters of True Type fonts.

wglUseFontOutlines creates *count* display lists, one for each glyph of a run of *count* glyphs The run of glyphs begins with the *first* glyph of the font of the specified device context, *hdc*. Each display list describes a glyph outline in floating point coordinates. The em square size of the font, the notional grid size of the original font outline from which the font is fitted, is mapped to 1.0 in the *x* and *y* directions in the display lists. The extrusion parameter sets how much depth the font has in the *z* direction.

The parameter *lpgmf* returns a **GLYPHMETRICSFLOAT** structure that contains information about the placement and orientation of each glyph in a character cell.

```
BOOL wglUseFontOutlines(
   HDC    hdc,                    //Device context of the outline font
   DWORD  first,                  //First glyph to be turned into a display list
   DWORD  count,                  //Number of glyphs to be turned into display lists
   DWORD  listBase,               //Specifies the starting display list
   FLOAT  deviation,              //Specifies the maximum chordal deviation from the true outlines
   FLOAT  extrusion,              //Extrusion value in the negative z direction
   int    format,                 //Specifies line segments or polygons in display lists
   LPGLYPHMETRICSFLOAT  lpgmf     //Address of buffer to receive glyphs metric data
  );
```

Parameters

hdc
Specifies the device context with the desired outline font. The outline font of *hdc* is used to create the display lists in the current rendering context.

first
Specifies the glyph of the font that is the first element of the set of glyphs that form the font outline display lists.

count
Specifies the number of glyphs in the set of glyphs used to form the font outline display lists. **wglUseFontOutlines** creates *count* display lists, one display list for each glyph in a set of glyphs.

listBase
Specifies a starting display list.

deviation
Specifies the maximum chordal deviation from the original outlines. When deviation is 0, the chordal deviation is equivalent to one design unit of the original font. The value of *deviation* must be equal to or greater than 0

extrusion
Specifies the amount a font is extruded in the negative *z* direction. The value must be equal to or greater than 0. When *extrusion* is 0, the display lists are not extruded.

format
Specifies the format, either WGL_FONT_LINES or WGL_FONT_POLYGONS, to use in the display lists. When *format* is WGL_FONT_LINES, **wglUseFontOutlines** creates fonts with line segments and when *format* is WGL_FONT_POLYGONS, **wglUseFontOutlines** creates fonts with polygons.

lpgmf
Points to an array of *count* **GLYPHMETRICSFLOAT** structures that is to receive the metrics of the glyphs. When *lpgmf* is NULL, no glyph metrics are returned.

Return Value

When the function succeeds, the return value is TRUE.

When the function fails, the return value is FALSE and no display lists are generated. Call **GetLastError** for extended error information.

Remarks

The **wglUseFontOutlines** function defines the glyphs of an outline font with display lists in the current rendering context. The outline font of the currently selected device context, *hdc*, is the model **wglUseFontOutlines** uses to define glyphs. **wglUseFontOutlines** works with True Type fonts only; stroke and raster fonts are not supported.

The parameter *count* specifies the total number of glyphs generated, each with its own display list. Each display list consists of either line segments or polygons; *format* specifies which method is used. Each display list has a unique identifying number starting with the number *listBase*. The first glyph in the set of glyphs is specified with the *first* parameter.

wglUseFontOutlines approximates glyph outlines by subdividing the quadratic B-spline curves of the outline into line segments until the distance between the outline and the interpolated midpoint is within the value specified by *deviation*. This is the final format used when *format* is WGL_FONT_LINES. When you specify *format* as WGL_FONT_POLYGONS the outlines are further tessellated into separate triangles, triangle fans, triangle strips, or quadrilateral strips to create the surface of each glyph.

A GLYPHMETRICSFLOAT structure contains information about the placement and orientation of each glyph in a character cell. The parameter *lpgmf* is an array of GLYPHMETRICSFLOAT structures holding the entire set of glyphs for a font. Each display list ends with a translation specified with the *gmfCellIncX* and *gmfCellIncY* fields of the corresponding GLYPHMETRICSFLOAT structure. The translation enables the drawing of successive characters in their natural direction with a single call to **glCallLists**.

Note The current release of OpenGL for Windows NT does not permit you to make GDI calls to a device context when a pixel format is double-buffered. You can get around this limitation using **wglUseFontOutlines** and **wglUseFontBitmaps**, when using double-buffered device contexts.

The following sample code shows how to draw text using **wglUseFontOutlines**:

```
HDC  hdc;  // A TrueType font has already been selected.
HGLRC  hglrc;
GLYPHMETRICSFLOAT agmf[256];

// Make hglrc the calling thread's current rendering context.
wglMakeCurrent(hdc, hglrc);

//create display lists for glyphs 0 through 255 with 0.1 extrusion
// and default deviation. The display list numbering starts at 1000
// (it could be any number).
wglUseFontOutlines(hdc, 0, 255, 1000, 0.0f, 0.1f,
            WGL_FONT_POLYGONS, &agmf);

// Set up transformation to draw the string.
glLoadIdentity();
glTranslate(0.0f, 0.0f, -5.0f)
glScalef(2.0f, 2.0f, 2.0f);

// Display a string:
glListBase(1000); // indicate the start of display lists for the glyphs.
// Draw the characters in a string.
```

```
glCallLists(24, GL_UNSIGNED_BYTE, "Hello Win32 OpenGL World.");
```

See Also

GLYPHMETRICSFLOAT, wglUseFontBitmaps, glListBase, glCallLists, glTexGen

GLYPHMETRICSFLOAT

```
typedef struct _GLYPHMETRICSFLOAT { // gmf
    FLOAT      gmfBlackBoxX;
    FLOAT      gmfBlackBoxY;
    POINTFLOAT gmfptGlyphOrigin;
    FLOAT      gmfCellIncX;
    FLOAT      gmfCellIncY;
} GLYPHMETRICSFLOAT;
```

The **GLYPHMETRICSFLOAT** structure contains information about the placement and orientation of a glyph in a character cell.

Members

gmfBlackBoxX
Specifies the width of the smallest rectangle (its black box) that completely encloses the glyph.

gmfBlackBox
Specifies the height of the smallest rectangle (its black box) that completely encloses the glyph.

gmfptGlyphOrigin
Specifies the x and y coordinates of the upper left corner of the smallest rectangle that completely encloses the glyph..

gmfCellIncX
Specifies the horizontal distance from the origin of the current character cell to the origin of the next character cell.

gmfCellIncY
Specifies the vertical distance from the origin of the current character cell to the origin of the next character cell.:

Remarks

The values of **GLYPHMETRICSFLOAT** are specified as notional units.

See Also

wglUseFontOutlines, **POINTFLOAT**

POINTFLOAT

```
typedef struct _POINTFLOAT { // ptf
    FLOAT      x;
    FLOAT      y;
} POINTFLOAT;
```

The **POINTFLOAT** structure contains the x and y coordinates of a point.

Members

x
Specifies the horizontal (x) coordinate of a point.

y
Specifies the vertical (y) coordinate of a point.

See Also

GLYPHMETRICSFLOAT