19941201 (this version 19961110)
Bo Nordin (bosse@cb.uu.se)
Centre for Image Analysis
018-183470

Introduction to the GL

Introduction

The GL (Graphics Library) is a library of subroutines that you call from a C program (normally, GL also supports C++, Ada, FORTRAN, Pascal, and BASIC) to draw and animate 2D and 3D colour graphics scenes. GL was originally developed by Silicon Graphics for use on the IRIS workstation series. Under the name openGL it is now available on several computer platforms.

GL is integrated with the X-Windows system. In order to be able to run GL programs on a workstation, the X server of the workstations must accept and understand the GL extensions to the X protocol (just as the PEX protocol must be known to the server when running PHIGS programs).

Drawing in GL

When you specify a geometric figure (a *geometry*), the GL draws it on the screen right away (*immediate* mode). Geometries are specified by its *edges* and *corners*. Each corner is a vertex which you specify the co-ordinates of. Edges are then formed from vertex to vertex and the complete geometry (a wireframe figure) is formed by connecting the edges. You mark the beginning and end of a vertex list with special *bgn*... and *end*... subroutines. In general terms, a geometry is specified as follows:

```
bgngeometry();
 vtype( vertex 1);
 vtype( vertex 2);
 .
 vtype( vertex n);
endgeometry();
```

where the italicised part of the subroutine names varies. *geometry* can be one of:

line	- a number of lines
points	- a number of points
closedline	- a number of lines, with the last vertex connected to the first
polygon	- a polygon, connected sequence of vertices all in a plane
qstrip	- a geometric figure consisting only of quadrilateral polygons
surface	- a parametric surface
tmesh	- a geometric figure consisting only of triangular polygons

and type should be one of

2s, 3s, 4s	- 2D, 3D, 4D short (normally 16 bit)
2i, 3i, 4i	- 2D, 3D, 4D integer (normally 32 bit)
2f, 3f, 4f	- 2D, 3D, 4D float (normally 32 bit)

2d, 3d, 4d - 2D, 3D, 4D doubles (normally 64 bit)

The following program demonstrates the use of some of the subroutines. It draws a square with green lines.

```
#include <gl/gl.h>

long vert1[3] = {200, 200, 0};
long vert2[2] = {200, 400};
float vert3[2] = {400.0, 400.0};
double vert4[3] = {400.0, 200.0, 0.0};

main() {
 prefsize(400, 400);       /* SET SCREEN WINDOW SIZE */
 winopen("greensquare");   /* OPEN A WINDOW */
 ortho2(99.5, 500.5, 99.5, 500.5); /*  SET VIEWPORT SIZE */
 color(WHITE);
 clear();                 /* CLEAR THE WINDOW: SET TO WHITE */
 color(GREEN);
 bgnline();               /* BEGIN A POLYLINE */
   v2s(vert1);
   v2i(vert2);
   v2f(vert3);
   v2d(vert4);
   v2s(vert1);
 endline();                /* END OF POLYLINE */
 sleep(10);
 gexit();
 return 0;
}
```

In addition to the simple primitives, GL offers a variety of "basic shape" subroutines. These are (the last letter in the name indicates the datatype of the arguments., and if the penultimate letter is "f", this indicates that the figure is to be filled, except that float is default so rectf, for instance, means a filled rectangle defined by floating-point vertices):

rectfs, rectfi, rectf, rects, recti, rect - 2D rectangles
circfs, circfi, circf, circs, circi, circ - 2D cirlces
arcfs, arcfi, arfcf, arcs, arci, arc - 2D arcs

Strings (as raster characters) are drawn by first "moving" to the desired position and then displaying the text:

```
cmov2(100,100);  /* variants: cmov2s, cmov2i, cmov,
                     cmovs, cmovi, cmov */
chrstr("String");
```

GL defines a number of functions for extracting character information, `getcpos` (returns the current character position - this position is changed by the cmov family of subroutines and by `chrstr` itself), `strwidth` (returns the width of a string in pixels). You can also create your own raster fonts.

Line styles and patterns (which can be used to fill polygons, rectangles, and arcs) can be defined and used. Both are described by bit patterns: xFFFF, for instance, is line style index 0 (predefined, default, and not redefinable) and describes a solid line.

Colour modes

Colour in GL can be either mapped (colour map mode) or unmapped (RGB Mode). You set the colour mode (colour map mode is default) with `RGBMode()` or `cmode()`.

In RGB mode, the values in the bitplanes correspond exactly to the colour to be presented. Regardless of the actual number of bits-per-pixel, GL allows you to treat the hardware as though it stores 24 bits per pixel (8 bits for each primary colour). You set the current colour in RGB mode with one of `c3s, c3i, c3f` (where the last letter indicates the type of the argument as usual). All these take a three-element vector as argument.

Colour map mode provides a level of indirection between the values stored in the bitplanes and the RGB values displayed on the screen. This mode is useful for systems that do not have enough bitplanes for RGB mode, and for applications that require dynamic colour changes. GL's colour map has 4096 (or 256 on some systems) entries. You can select either *onemap* mode, or *multimap* mode. The default mode is onemap; in this mode the colour map is organised as a single map. Multimap, on the other hand, uses only 8 bits from the bitplanes (256 entries in the colour map), but allows up to 16 completely independent 256-entry colour maps.

In colour map mode you specify a colour with `mapcolor()` and set it with `color()`:

```
#define PINK 101
mapcolor( PINK, 255, 80, 80);
color( PINK);
```

GREEN and WHITE in the example program in the previous section are predefined colour map entries. Other predefined entries are BLACK, RED, YELLOW, BLUE, MAGENTA, and CYAN

If the system has too few bitplanes to display colour correctly, *colour dithering* (cf. Foley *et al*, chapter 11.1.2 Halftone Approximation), is used to expand the range of displayed colours.

Gouraud Shading

So far, all lines and polygons have been assumed to be drawn in one uniform colour. In other words, *flat shading*, has been used. One way to improve the appearance of an approximated polygonal surface is to vary the colour across the polygons. *Gouraud shading* (see Foley e*t al*: 14.2.4) is one way to accomplish this. To make GL shade a polygon with Gouraud shading, you set the colour before each vertex as the following program excerpt shows:

```
float blackvect[3] = {0.0, 0.0, 0.0};
float redvect[3] = {1.0, 0.0, 0.0};
float greenvect[3] = {0.0, 1.0, 0.0};
```

```
float bluevect[3] = {0.0, 0.0, 1.0};

long triangle[3][2] = { { 20, 20}, { 20, 380}, {380, 20} };
  bgnpolygon();
  c3f(redvect);
  v2i(triangle[0]);
  c3f(greenvect);
  v2i(triangle[1]);
  c3f(bluevect);
  v2i(triangle[2]);
 endpolygon();
```

The triangle drawn by this program will have vertices of bright red, bright green, and bright blue, respectively. It will be Gouraud shaded.

Input

GL has a large number of input functions. Because GL runs under X-Windows the *event handling* of the X server is used for the queuing of input events (it is possible to set up user input to use only GL, X and GL, or X alone but here I will only treat the GL-alone variety)

The most important input functions are:

qdevice(Device) - enables queuing for a device.
unqdevice(Device) - disables queuing for a device
long qtest(short *) - returns the device of the first event in the queue, returns zero if the queue is empty. Returns the data value in the parameter.
Device qread(short *) - as qtest but waits if the queue is empty
qreset() - removes all entries from the queue

long getvaluator(Device) - polls (samples) the status of a valuator
Boolean getbutton(Device) - polls the status (up or down) of a button

GL predefines many devices (of type Device which is a short): buttons and valuators, for instance. Some are:

Buttons: MOUSE1 (also called RIGHTMOUSE),
MOUSE2 (also called MIDDLEMOUSE)),
MOUSE3 (also called LEFTMOUSE)
AKEY..PADENTER - all keys on the keyboard
Valuators: MOUSEX - x position of the cursor controlled by the mouse
MOUSEY - y position of the cursor
CURSORX - x valuator attached to cursor (default MOUSEX)
CURSORY - y valuator attached to cursor (default MOUSEY)

Window manager devices are abstract devices associated with GL window events. Some examples:

REDRAW - a window must be redrawn (size has changed, becomes visible,..)
WINQUIT - if queued, the window manager sends this instead of killing the window when Quit is selected from a window's frame menu.

The following program uses queuing to control a simple drawing program, that lets you sketch in the window with the left mouse button:

```
#include <gl/gl.h>
#include <gl/device.h> /* definitions of device constants */
#define X       0
#define Y       1

main() {
 short val, mval[2], lastval[2];
 long org[2];
 Device dev, mdev[2];
 int leftmouse_down = 0;
 lastval[X] = -1;
 prefsize(400, 400);
 winopen("input");
 color(BLACK);
 clear();
 getorigin(&org[X], &org[Y]); /* find origin of the window */
 mdev[X] = MOUSEX; mdev[Y] = MOUSEY;
 getdev(2, mdev, lastval); /* getdev polls several devices */
 lastval[X] -= org[X];  lastval[Y] -= org[Y];
 qdevice(LEFTMOUSE);
 qdevice(ESCKEY);      /* use ESC to terminate */
 qdevice(MOUSEX);      /* In effect, we sample the */
 qdevice(MOUSEY);      /* cursor positions      */
 qdevice(REDRAW);
 color(WHITE);         /* prepare to draw white lines */
 while (1) {
  switch (dev = qread(&val)) {
   case LEFTMOUSE: leftmouse_down = val; break;
   case MOUSEX: mval[X] = val - org[X]; break;
   case MOUSEY: mval[Y] = val - org[Y];
                 if (leftmouse_down) {
                   bgnline();
                   v2s(lastval);
                   v2s(mval);
                   endline();
                 }
                 lastval[X] = mval[X]; lastval[Y] = mval[Y];
                 break;
   case REDRAW: color(BLACK); /* REDRAW -> clear window */
                 clear();
                 color(WHITE);
                 break;
   case ESCKEY:  exit(0);
  }
 }
}
```

Picking
Picking identifies items on the screen that are near the cursor. Because GL draws in immediate mode, picking is not very easy to do. In fact, you must be able to rewrite the complete screen when picking is required. In order to perform picking of an item on the screen you must:

1. set the system into picking mode
2. redraw the image on the screen using *pick()* (nothing is actually redrawn).
3. call *endpick()*

You may resize the picking region with *picksize(int deltax, int deltay).* Default is that a region of size (10,10) about the cursor is pick sensitive. Only named objects can be picked. And the programmer must name objects by calling naming functions while drawing in pick mode.

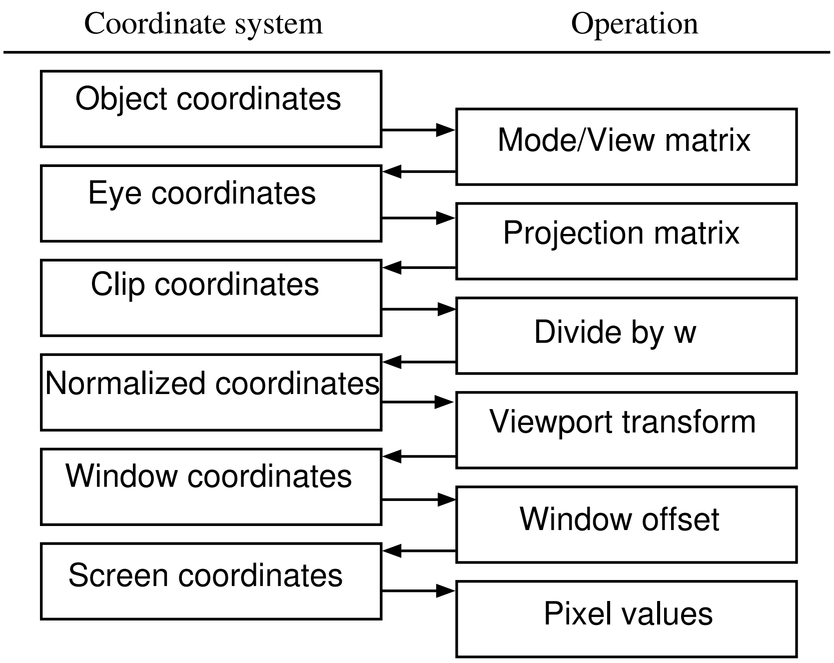
Animation

GL has functions which can help improve animation. The buffer memory (the frame buffer) can be addressed in two different modes: single and double buffer. You select which one you want by calling *singlebuffer()* or *doublebuffer()*. In double buffer mode you can switch back and forth between buffers (making one or the other the visible buffer) by calling *swapbuffers()*. Thus, when you create an animated sequence you draw in one buffer, then make it the visible buffer while you are drawing in the other. There are a multitude of functions in GL for maximising animation performance, but I will not describe these here.

Another way of creating semi-animated sequences is by changing the colour map in multimap mode.

Coordinate transformations

As every graphics package worth its name, GL also supports coordinate transformations. The figure below describes the coordinate systems in GL:



The various systems should be familiar to you by now, The geometry vertices are specified in Object Coordinates. GL concatenates modelling and viewing transformations into the ModelView matrix and thus has no world coordinate system (but the Object coordinates could be called that, of course).

In order to go from geometry vertices to a scene displayed on your screen, you have to specify a *projection transformation* and a *viewing transformation*.

You define a perspective transformation by calling (Coord is a float):

```
void perspective(
 Angle fovy,  /* field of view measured in 10th of degrees */
 float aspect, /* x dimension / y dimension */
 Coord znear,   /* distance to the front clipping plane */
 Coord zfar ); /* distance to the back clipping plane */
```

You define a parallel (ortographic) projection with ortho or ortho2:

```
void ortho (
 Coord left,
 Coord right,
 Coord top,
 Coord znear,
 Coord zfar);
```

The simple ortho2 does not require the last two arguments; it defines a 2D clipping rectangle.

The second type of transformation is the viewing transformation. This transformations allows you to position the eye and define the direction toward which it is looking. There are two different ways to specify this: *polarview()* and *lookat()*:

```
polarview(
 double dist; /* distance from viewpoint to origin */
 int azim,    /* azimuthal angle in the xy plane */
 int inc,     /* incidence angle in the yz plane */
 Angle twist); /* rot.around the z axis in eye system */

lookat(
 Coord vx, vy, vz,  /* viewpoint - looking from */
 Coord px, py, pz, /* reference points - looking at */
 Angle twist); /* rot. around the z axis in eye system */
```

Apart from the transformations that calling the above functions result in, you may also transform your geometries in the Object coordinate system:

```
void rotate( Angle a, char axis);
void translate( Coord x, Coord y, Coord z);
void scale( float x, float y, float z);
```

All geometries drawn after calling one of these routines are affected. You can use the two functions *pushmatrix()* and *popmatrix()* to preserve and restore coordinate systems.

Hidden-surface removal

GL has two basic methods for hidden-surface removal (or visible-surface determination, the more positive-sounding term used by Foley *et al*): z-buffering and backface removal.

By default, z-buffering is turned off. To set up z-buffering, you call two subroutines:

```
zbuffer( TRUE); /* turn z-buffering on */
zclear()        /* clear the buffer: initialise with max z*/
```

In z-buffer mode, the system, before drawing anything, compares the z-value of each incoming pixel to the z-buffer value for that pixel. The values of the znear and zfar planes determine the resolution of the z buffer. You can enhance the resolution of the z-buffer by diminishing the viewing volume in z by setting the near and far values as close together as possible.

The other hidden-surface algorithm in GL is the elimination of backfacing polygons. A backfacing polygon is defined as a polygon whose vertices appear in clockwise order in screen space. In other words, when backfacing polygon removal is on, only polygons whose vertices appear in counter-clockwise order are displayed. When defining polygons, you must make sure that the vertices are specified such that they are drawn in counter-clockwise order. You call b*ackface(Boolean on)* to turn the feature on or off.

Lighting

In GL, the characteristics of a light source determine the direction, intensity, and colour of the incident light. The characteristics of the object geometry and its surface material determine the direction, intensity, and colour of the reflected light.

To use GL's lighting functions, you must define a surface normal for each vertex. GL maintains a current normal which is in effect until changed. Normals should be unit-length and are defined with the function *n3f*, corresponding to *v3f*. Call n3f before calling v3f for the corresponding vertex.

To set up GL lighting you must:
- define the properties for each material, light, and lighting model that you need
- activate (*bind*) the definitions
- draw the scene, providing surface normals

The lmdef() subroutine lets you establish the properties, and the lmbind subroutine binds the properties to a specific material, light source, or lighting model:

```
void lmdef(
 short deftype,/* indicates material, light src, or model */
 short index,    /* an index to be associated */
 short np,        /* count of symbols in props */
 float props[]); ./* array of symbols defining properties */
```

```
void lmbind(
 short target,  /* MATERIAL, LIGHTxx, or LMODEL */
 short index);  /* same as in lmdef *./
```

The first argument to lmbind binds a definition to material, light source, or light model. As always in GL, the settings are modal. In any moment, only one MATERIAL and LMODEL can be active. Up to eight light sources (LIGHT0 - LIGHT7) can be active simultaneously.

The first argument to lmdef should be one of DEFMATERIAL, DEFLMODEL, or DEFLIGHT. The fourth argument to lmdef is quite complex, it consists of keywords (defined constants) and values. There number of values after each keyword differs depending on the keyword. I use an example to show you how it is used.

```
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>

Matrix Identity = { 1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1 };

float mat[] = {
 AMBIENT, .1, .1, .1,    /*r,g,b coeff. for ambient light*/
 DIFFUSE, 0, .369, .165, /* dito for diffuse light */
 SPECULAR, .5, .5, .5,   /* dito for specular refl. */
 SHININESS, 10,          /* 0-128, size & bright of spec. */
 LMNULL   /* must always end a props[] */
};

static float lm[] = {
 AMBIENT, .1, .1, .1,
 LOCALVIEWER, 1,  /* local (1) or infinite viewpoint */
 LMNULL
};

static float lt[] = {
 LCOLOR, 1, 1, 1,   /* white */
 POSITION, 0, 0, 1, 0, /*location of the light source */
 LMNULL
};

main() {
 long xorigin, yorigin, xsize, ysize;
 float rx, ry;
 short val;

 winopen("cylinder");
 qdevice(ESCKEY);
 getorigin(&xorigin, &yorigin);
 getsize(&xsize, &ysize);
 RGBmode();
```

```
 doublebuffer();
 gconfig();
 lsetdepth(getgdesc(GD_ZMIN), getgdesc(GD_ZMAX));
 zbuffer(1);
 mmode(MVIEWING);
 loadmatrix(Identity);
 perspective(600, xsize/(float)ysize, .25, 15.0);
 lmdef(DEFMATERIAL, 1, 0, mat);
 lmdef(DEFLIGHT, 1, 0, lt);
 lmdef(DEFLMODEL, 1, 0, lm);
 lmbind(MATERIAL, 1);
 lmbind(LMODEL, 1);
 lmbind(LIGHT0, 1);
 translate(0, 0, -4);

 while (!(qtest() && qread(&val) == ESCKEY && val == 0)) {
  ry = 300 * (2.0*(getvaluator(MOUSEX)-xorigin)/xsize-1.0);
  rx = -300 * (2.0*(getvaluator(MOUSEY)-yorigin)/ysize-1.0);
  czclear(0x404040, getgdesc(GD_ZMAX));
  pushmatrix();
  rot(ry, 'y');
  rot(rx, 'x');
  drawcyl();
  popmatrix();
  swapbuffers();
 }
}

drawcyl() {
 double dy = .2;
 double theta, dtheta = 2*M_PI/20;
 double x, y, z; float n[3], v[3]; int i, j;

 for (i = 0, y = -1; i < 10; i++, y += dy) {
  bgntmesh();
  for (j = 0, theta = 0; j <= 20; j++, theta += dtheta) {
   if (j == 20) theta = 0;
   x = cos(theta);
   z = sin(theta);
   n[0] = x; n[1] = 0; n[2] = z;
   n3f(n);
   v[0] = x; v[1] = y; v[2] = z;
   v3f(v);
   v[1] = y + dy;
   v3f(v);
  }
  endtmesh();
 }
}
```

The value after SHININESS controls the size and apparent brightness of a specular highlight. It can range from 0-128 where higher values result in smaller, more focused specular highlights. A value of zero turns off specular reflection.

POSITION is defined in homogeneous coordinates. A zero w component defines an infinite light source, while a non-zero component defines a point light source.

Advanced features

Some (or even most) of the capabilities I briefly describe here are not available on low-cost systems

Pixels

GL allows you to address pixels directly and to perform pixel operations. Like text in raster fonts, pixels are not easy to transform, though. You can read and write screen pixels as well as pixels in the z-buffer,

Depth-Cueing

GL has two techniques that contribute to the perception of 3D in your scenes: depth-cueing and the simulation of atmospheric effects. Depth-cueing is simply the modification of an object's colour depending on its distance from the viewer.

You can also simulate fog, smoke, haze, and air pollution by varying the colour and density of the atmosphere in your scene. In order to introduce fog, you call *fogvertex()* which sets up the fog characteristics.

Curves and Surfaces

You can define B-spline curves and surfaces in GL. GL supports NURBS (**N**on-**U**niform **R**ational **B**-**S**pline) curves and surfaces. You can *trim* a NURBS surface to define its visible regions.

Graphics Objects

You can define graphics objects. These objects can be edited with functions similar to those of PHIGS. Unlike PHIGS, however, GL works in immediate mode, i.e. you must call a function (*callobj(Object obj*)) to draw the object yourself; GL has no database storage of objects. And you must remember that all attribute settings (linestyle, colour, etc.) are modal, i.e. if you change an attribute in an object you must save the status before calling it yourself (GL has functions to facilitate this: *pushattributes()* and *popattributes()*).

Antialiasing

GL includes methods that make objects drawn on discrete devices - such as a display screen - appear smooth. You can use antialiasing techniques to smooth lines and polygons. This is accomplished by setting modes before drawing the primitives: you call *polysmooth()* in order to draw a polygon in antialiasing mode, for instance. Reducing aliasing effects is especially important in animation applications because the "motions" created by aliasing is easily confused with the motion we want to show.

Apart from conventional antialiasing techniques (using pixel coverage, as described in Foley *et al*, chapter 3.14), GL also supports blending. If blending mode is on, the value of the pixel to be written is affected by not only the new pixel but also by the old pixel on the screen.

The conventional technique for antialiasing used in GL affects vertex positions (by allowing subpixel positioning) and the scan conversion process. When antialiasing is off, vertices, after having been transformed and projected are rounded to the nearest pixel centre. When antialiasnig is on, vertices keep their subpixel position in order to facilitate accurate scan conversion: the exact position is made available to the sampling hardware (or software).

Texture maps

GL has subroutines for the texture for added realism. Textures may be defined and mapped to the surface of objects.

Concluding remarks

This has been a very short description of some of the features of GL. It is obviously not intended to be a comprehensive account of GL, but I hope that I have managed to give you at least a flavour of the system, and pointed out some of the similarities with and differences from other systems, such as SPHIGS or PHIGS.

Appendix: Practical advice

This appendix gives you some hints about where to get (Open) GL and how to compile and link programs.

OpenGL versus GL

Open GL is the extension of the GL (Graphics Library) to other platforms, such as DEC/Alpha, SUN Sparc etc. Originally, GL was written by the computer company SGI (Silicon Graphics) about ten years ago, and ran only on their own platforms. Advanced Computer Graphics, by the way, has always (?) been SGI:s profile -- the name SGI perhaps gives some hint in that direction.

Later, they developed Open GL. The extension "Open" correctly suggests that this is available not only for SGI, but for any company that wants to use these graphics routines. So, why bother if the library has the extension "Open" or not? The reason is that the syntax has changed quite a bit. So even though the feeling of and idea behind GL remain unchanged, one must learn new function names, new flags etc., in order to use Open GL.

Libraries you will need to compile Open GL programs

You will need the GL libraries (GL, GLU, GLw) installed at your workstation. The GLX-extension for X-windows *must* be available on the workstation that is used as display. That is, even if you have compiled your programs on a computer with the GLX-extension, you won't be able to redirect output to another X-server that hasn't got this extension (i.e. "setenv DISPLAY ..." won't do).

Furthermore, you will most probably want to install the GL toolkit libraries (tk, aux) to make Open GL a useful package. But please be aware of the following: The GL tk-library has got nothing to do with John Ousterhouts tk-library (*Tcl and the Tk toolkit)*. They are totally different things, but unfortunately they have the very same name.

To compile simple GL programs, you can use a command like the following:

```
gcc -o simple simple.c -lX11 -lXm -lXt -lGL -L. -ltk -laux
```

`gcc` is the c-compiler. `X11`, `Xm`, and `Xt` are the X-libraries. `GL` is the GL-library (surprise), and `tk` and `aux` are the toolkit libraries. Please note that those libraries are placed in my current directory, hence the `-L.`-directive (which directs the compiler to look for libraries in the directory '.', the current one).

A very simple Open GL program

This program is similar to the "square" program of page 2. Please note that it is quite a big difference between the source code for this Open GL program and the GL version. I have excluded the copyright notice from SGI, so you may *not* use this code in any programs, but instead copy the full version available somewhere at the Internet.

```
#include <GL/gl.h>
#include "aux.h"
#include <unistd.h> /*-DEC- Alpha-workstation */
/*gcc -o simple -simple.c -lX11  -lXm -lXt -lGL L.-ltk -laux */

int main( int argc, char **argv ) {
  auxInitDisplayMode ( AUX_SINGLE | AUX_RGB);
  auxInitPosition ( 0, 0, 500, 500);
  auxInitWindow ( argv[0]);

  glClearColor ( 0.0, 0.0, 0.0, 0.0);
  glClear( GL_COLOR_BUFFER_BIT);
  glColor3f( 1.0, 1.0, 1.0);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  glOrtho(-1.0, -1.0, -1.0, 1.0, -1.0, 1.0);
  glBegin(GL_POLYGON);
    glVertex2f( -0.5, -0.5);
    glVertex2f( -0.5, 0.5);
    glVertex2f( 0.5,  0.5);
    glVertex2f( 0.5, -0.5);
  glEnd();
  glFlush();
  sleep (10);
}
```

I suggest that anybody interested in this topic, installs (or ask the system manager to install) Open GL including the example programs. Then try to compile and run the examples to gain experience. Maybe find some book to read. Or take a look at the Internet