OpenGL Course, Part 1

HELLO 3D WORLD

THOMAS G E RUGE

Here we present the first in a series covering an OpenGL programming course. Using simple, easy-to-understand examples, we will lead you into the exciting and complex world of 3D programming.

After a broad overview and a little bit of theory, the first part of the course shows how OpenGL programs are constructed, using Hello World as an example.

OpenGL was originally developed by SGI and in ages long past was called IrisGL. The primary aim of developing IrisGL was to develop a programmable interface for the graphics workstations of SGI. This programmable interface was initially intended to be hardware-independent, future-proof and to meet the special demands of 3D graphics programming.

It soon emerged that IrisGL is not only of interest for SGI workstations, but can also help other workstations to get up and running in terms of graphics. The OpenGL Architectural Review Board (ARB) was founded in 1992 by major manufacturers of graphics workstations such as SGI, Sun, Hewlett-Packard and others. The task of the ARB is to guide the further development of OpenGL and introduce new functionalities in later versions. OpenGL has now become the commonest hardware-independent interface for 3D programming.

The latest version is OpenGL 1.2. There are OpenGL implementations for Irix, Solaris, HP-UX, Linux, Windows and a few other operating systems. In the Linux world the OpenGL clone Mesa is the most common one. The American Brian Paul breathed life into Mesa, and the library has since been further developed by him and many other developers all over the world.

Principles

OpenGL is always in a defined state, which is set by condition variables; this means that OpenGL is a State Machine. An example of a condition variable is the current colour with which the individual primitives are rendered (drawn). These are set using the command *glColor3f(red, green, blue)* and this then applies for all drawn objects until a second *glColor3f(..)* command changes the colour.

All objects under OpenGL must be composed from 10 different primitives. The instruction set in the library comprises some 200 commands, all starting with *gl*. OpenGL does in fact contain all the simple operations for programming interactive 3D graphics, but it cannot do real 3D bodies such as cylinders and dice, only points, lines and polygons.

All complex bodies have to be constructed by the 3D developer out of simple primitives. To make the work a bit simpler for the applications developer, hard-working 3D experts have developed some programming libraries, a few of which we shall be introducing in the course of this series of articles. The two most important expansions are

068-opengl.qxd 28.03.2001 9:01 Uhr Seite 69

3D transformations

The elementary transformations in 3D graphics programming are translation, rotation and scaling of points in 3D space. So that these operations are shown in a standard format and can be programmed easily, transformations take the general form: b = M • a. So as to display not only rotations or scalings, but also translations in this linear form, a homogenous co-ordinate xw is inserted into the space of the transformations. In mathematical jargon, we have embedded the affine space R3 in the projective space P(R4) and can thereby display any affine figure (transformation) through a matrix multiplication.

$$a = \begin{pmatrix} a_x \\ a_y \\ a_z \\ a_w \end{pmatrix}$$

$$b = \begin{pmatrix} b_x \\ b_y \\ b_z \\ b_w \end{pmatrix}$$

$$M = \begin{pmatrix} t_{xx} & t_{xy} & t_{xz} & t_{xw} \\ t_{yx} & t_{yy} & t_{yz} & t_{yw} \\ t_{zx} & t_{zy} & t_{zz} & t_{zw} \\ t_{wx} & t_{wy} & t_{wz} & t_{ww} \end{pmatrix}$$

The three transformations referred to above then look like this:

Translations

We can displace objects in space at will with translations

$$b = T(t)a = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} a_x + t_x \\ a_y + t_y \\ a_z + t_z \\ 1 \end{pmatrix}$$

Scalings

Scalings make objects bigger and smaller

$$b = Sa = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_x \\ a_y \\ a_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x \cdot a_x \\ s_y \cdot a_y \\ s_z \cdot a_z \\ 1 \end{pmatrix}$$

Rotations about the origin

Rotations change the orientation of objects. If we want to rotate the point a about an axis p by the angle a, the associated transformation matrix R(p,a) in its commonest form looks like this:

$$R(p,\alpha) = \begin{pmatrix} r_{xx} & r_{xy} & r_{xz} & 0 \\ r_{yx} & r_{yy} & r_{yz} & 0 \\ r_{zx} & r_{zy} & r_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations about the three axes of the co-ordinate grid by the angle a:

$$R(e_x,\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R(e_y,\alpha) = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R(e_z,\alpha) = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations about any axis can be assembled from rotations about the co-ordinate axes:

$$R(p,\alpha) = R(e_x,\alpha) \cdot R(e_y,\alpha) \cdot R(e_z,\alpha)$$

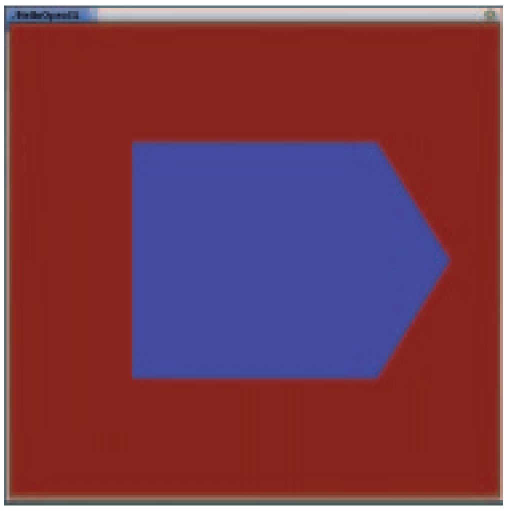
Rotation about any point

The rotation matrices described above define rotations about the origin, but often an object needs to be rotated about a different point u. To do this, we link together translations and rotation in such a way that the point about which the rotation takes place serves as origin. This means that we make a change to the co-ordinate system:

$$b = T(u) \cdot R(p,\alpha) \cdot T(u)^{-1} \cdot a$$

Figuratively speaking, we have first displaced the point a by the vector -u then rotated it by a and finally pushed the point back again.
In graphics programming the programmer needs such assembled matrices over and over again, so that the 3D objects appear where they belong. OpenGL relieves the developer of a great deal of work in the creation and treatment of transformation matrices.
After all the transformations we finally want to work against with three-dimensional co-ordinates:
Where aw = 0, the point lies at infinity.

$$a_3 = \begin{pmatrix} a_x / a_w \\ a_y / a_w \\ a_z / a_w \end{pmatrix}$$


How the output of the program HelloOpenGL ought to look

GLU (OpenGL Utility Library) and GLUT (OpenGL Utility Toolkit).

GLU makes it easier to create projection calculations and to construct complex surfaces, using, among others, Nurbs (Non-uniform-rational B-Splines). GLUT is an operating system-independent tool for simpler handling of OpenGL windows and provides the application developer with routines to respond to user events from the mouse or keyboard.

Anyone wanting to become a real 3D graphics expert will have to overcome a few mathematical obstacles. For a better understanding, some knowledge of linear algebra is useful and terms such as vectors or matrices should not be a complete and utter mystery to the reader. The most important terms that crop up time and time again in OpenGL are set out in the 3D transformations boxout. Anyone with a yen for higher mathematics can learn more with the Open GL Programming Guide

Installing the OpenGL clone Mesa doesn't take much. In most distributions Mesa is also installed by default. Those users who have previously had no OpenGL/Mesa environment on their computers, or want to use the latest version of Mesa can find the

Listing 1: HelloOpenGL.c

```
#include /stdio.h>
#include /stdlib.h>

#include /GL/glut.h>

void DrawScene(void)
{
        //set background colour (dark red)
 glClearColor (0.5, 0.0, 0.0, 0.0);
 glClear(GL_COLOR_BUFFER_BIT);

 // Set colour of pentagon, (blue)
 glColor3f(0.0, 0.1, 1.0);

        //Polygon progression of the pentagon
        glBegin(GL_POLYGON);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0.5,  0.5);
                glVertex2f( 0.5,  0.5);
                glVertex2f( 0.8,  0.0);
                glVertex2f( 0.5, -0.5);
        glEnd();

        //draw previous GL commands
        glFlush();
}

int main(int argc, char *argv[])
{
        // initialises GLUT
glutInit(argc, argv);
        // initialise output window (Single Buffer, RBG colour model)
 glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        // set window position and size
 glutInitWindowPosition (100, 100);
 glutInitWindowSize (500, 500);
        // create window
 glutCreateWindow (argv[0]);

        // Set callback function to draw GL object
 glutDisplayFunc(DrawScene);
        // Main loop
 glutMainLoop();

 return EXIT_SUCCESS;
}
```

latest binary and source packets at the Mesa homepage.

Hello OpenGL with the OpenGL Utility Toolkit

The first program has not much to do with 3D as yet, as OpenGL can in fact also draw two-dimensional objects – and that's enough for a start. To simplify window management we are making use of the OpenGL expansion GLUT. OpenGL puts together all 2D and 3D objects from primitives and in the first program example we are simply going to draw an arrow in an OpenGL window.

The program (Listing: Hello-OpenGL.c) consists of the functions *DrawScene(..)* and *main(..).main (..)* contains only calls from GLUT and helps us to draw the graphics, which are created in *DrawScene(..)*, in a window.

The first command *glutInit(argc, argv)* initialises the GLUT library and takes over X-parameters such as *-display* or *-iconic*. With *glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)* we are agreeing that our OpenGL window uses only one graphics buffer and uses RGB as the colour model. Buffer is that area of memory in which drawing takes place. *GLUT_SINGLE* tells the function that we only want to use one buffer. The memory that is visible on the monitor and the area in which we are drawing are identical. This means that the onlooker sees how the individual OpenGL objects are drawn.

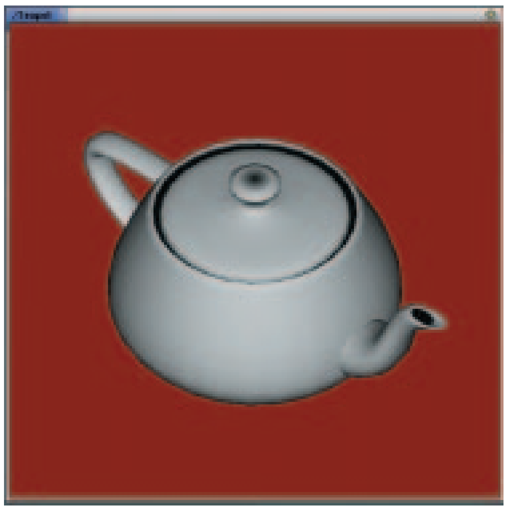
glutInitWindowPosition(100,100) sets the OpenGL window to the position (x = 100, y = 100) of the X-server. *glutInitWindowSize(500,500)* sets the size of the OpenGL window. The window is created by the command *glutCreateWindow(argv[0])*, but is not yet visible. As the result of the argument *argv[0]* the window receives as name the program invocation.

The instruction *glutDisplayFunc (DrawScene)* ensures that every time the OpenGL window is to be redrawn, the callback function *DrawScene()* is called up. Events which make such a redrawing necessary are for example the displacement, enlargement or reduction in size of the window and the explicit call up of the function *glutPostRedisplay()*.

The main loop is invoked using *glutMainLoop()*. Within this loop the OpenGL program responds to all possible user events and intercepts mouse and keyboard events.

Thanks to the seven GLUT commands, we no longer need to bother about making any changes whatsoever to our window or about actions by the user. GLUT takes over these tasks. The programmer only has to write the callback functions, so that their program responds as required to external events. We can thus now concentrate completely on the actual OpenGL programming which occurs in *DrawScene()*.

The command *glClearColor(0.5, 0.0, 0.0, 0.0)* in



DrawScene() specifies that the graphics buffer is cleared with a dark red. *glClear(GL_COLOR_BUFFER_BIT)* performs the clearing of the graphics buffer. The next command *glColor3f(0.0, 0.1, 1.0)* makes sure that henceforth all subsequent objects are coloured blue. The definition of the arrow starts with *glBegin(GL_POLYGON)*. *GL_POLYGON* firstly specifies that the following co-ordinates are to be interpreted as a closed polygon progression.

The following *glVertex2f(x, y)* commands define the position of the corners of the polygon. *glVertex2f* defines the arrow in only two dimensions, and in order to describe the object with three-dimensional co-ordinates, *glVertex2f(x, y)* would just have to be replaced by *glVertex3f (x, y ,z)*; *glEnd()* ends the definition of the arrow.

With *glFlush* we ensure that all previous OpenGL commands are executed, in our example the buffer is first coloured in red, then the arrow is drawn. After the program stops, it is time for the compilation:

```
gcc -I . -c HelloOpenGL.c

gcc -o HelloOpenGL HelloOpenGL.o \
-lGL -lglut -lGLU
```

When executing *./HelloOpenGL* a window with a blue arrow on a red background should be visible (see Figure).

The first example is, nothing particularly spectacular and nor is a 3D scene visible, but it already includes all the important components of an OpenGL program. GLUT has firstly defined which characteristics the OpenGL window possesses and how it responds to events. We have then defined the appearance of the scene within *DrawScene()*.

A full 3D pot

The sample program Teapot.c (Listing 2), with the aid of the GLU library, paints a three-dimensional teapot and responds to events from mouse and keyboard. The program consists, apart from the

AD

main function *main(..)* and the initialisation function *myinit()*, of the callback functions *mouse(..)*, *motion(..)*, *keyb(..)*, *recalcModelPos()* and *DrawScene()*. In comparison with the first sample program *main()* uses three new callback functions and evaluates the depth information.

glutInitDisplayMode(GLUT_DOUBLE I *GLUT_RGB* I *GLUT_DEPTH)* initialises the OpenGL-window with foreground and background buffer and a depth buffer. The foreground buffer is shown in the OpenGL window. Drawing is done in the invisible background buffer. When rendering is

Listing 2:Teapot.c

```
#include <stdio.h>
#include <stdlib.h>

#include <GL/glut.h>

// Condition variable
// mouse movement
int mousemotion;
int mousex, mousey;

// Initialise model orientation
GLfloat xangle = 4;   /* for rotation */
GLfloat yangle = 120;
//Model position
GLfloat posx = 0, posy = 0, posz = 0;

//Callback Function: Reaction of mouse clicks
void mouse(int button, int state, int x, int y)
{
  if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
    mousemotion = 1;
    mousex = x;
    mousey = y;
  }
  if (button == GLUT_LEFT_BUTTON && state == GLUT_UP) {
    mousemotion = 0;
  }
}

//Callback Function: Reaction of mouse movement
void motion(int x, int y)
{
  if (mousemotion) {
    xangle = xangle - (y - mousey);
    yangle = yangle - (x - mousex);
    mousex = x;
    mousey = y;

    // Draw a new display
    glutPostRedisplay();
  }
}

//Callback Function: Reaction of keypress
void keyb( unsigned char keyPressed, int x, int y )
{
        switch( keyPressed ) {

                case ‘l’:
                               // Light activation
                               glEnable(GL_LIGHTING);
                               glutPostRedisplay();
                               break;
                case ‘o’:
                               // Light deactivation
                               glDisable(GL_LIGHTING);
                               glutPostRedisplay();
                               break;

        }
}
//new Model position calculation
void recalcModelPos(void)
{
  glLoadIdentity();
  glTranslatef(posx, posy, posz);
  glRotatef(xangle, 1.0, 0.0, 0.0);
  glRotatef(yangle, 0.0, 1.0, 0.0);
}

void DrawScene(void)
{
          // Delete buffers
  glClearColor (0.5, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

          // Calculation of new model coordinates
  recalcModelPos();
          // Teapot painting
  glutSolidTeapot(0.6);

          // Foreground and background buffer change
  glutSwapBuffers();
}

//Initialising function
void myinit()
{
          GLfloat light_position[] = {0.0, 0.0, -1.0, 1.0 };

          //First GL-Light fall

glLightfv( GL_LIGHT0, GL_POSITION, light_position );

          glEnable(GL_LIGHT0);

          //Z-buffer activation
          glDepthFunc(GL_LEQUAL);
          glEnable(GL_DEPTH_TEST);
}

int main(int argc, char *argv[])
{
          //GLUT initialisation window
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);

          // Initialisation
  myinit();

          //Callbacks set: Reaction of mouse clicks,
  movements and keyboard activity
  glutMouseFunc(mouse);
  glutMotionFunc(motion);
  glutKeyboardFunc(keyb);

          //Callback for drawing the GL-Function
  glutDisplayFunc(DrawScene);

  glutMainLoop();

  return EXIT_SUCCESS;
}
```

complete, we swap foreground and background buffers, so the image just drawn can be seen.

The depth or z-buffer ensures that areas covered by other areas cannot be seen. In the z-buffer each pixel in the OpenGL window is assigned a z-value. If a new pixel is to be set during rendering, a check is made as to whether its z-value is greater than that of a pixel that has already been set. If so, the new pixel comes before the old one and can be drawn, otherwise not.

If all parameters have been set and if the OpenGL window has been created, an initialisation function *myinit()* is called up. This function was introduced for better legibility of the program.

The next step is to set the diverse callback functions. So *glutMouseFunc(mouse)* sets the function to be called up when pressing or releasing a mouse button. *glutMotionFunc(motion)* sets the callback function for mouse movements. So that the program also responds to keyboard events, we also call up *glutKeyboardFunc(keyb)*. The rest of *main()* is equivalent to *HelloOpenGL*.

Callback functions in detail

In *void mouse(int button, int state, int x, int y)* the position of the mouse in the OpenGL window is queried, buffered in *x* and *y* and registers whether the left mouse button was pressed. The function makes a note with *mousemotion* of the condition of the left mouse button and takes it into account later in the function *motion(int x, int y)*. The function *motion(..)* calculates, from the current mouse position (*x*,*y*) and the mouse position at the time when the left mouse button has been pressed (*mousex*, *mousey*), the angle about which the object should be rotated (*xangle*, *yangle*). The call up of *glutPostRedisplay()* ensures that the scene is actually redrawn and by calling up *DrawScene()* the new position of the teapot is calculated in *recalcModelPos()*.

The callback function *keyb(unsigned char keyPressed, int x, int y)* responds to the two letters *l* and *o*. When the user enters *l* the light calculation is activated with *glEnable(GL_LIGHTING)*, *o* deactivates the light again with the aid of *glDisable(GL_LIGHTING)*. These three callback functions cover all events that the program has to process.

The next function *recalcModelPos()* is called up by *DrawScene()* to calculate the new position of the teapot. Generally every transformation (rotation, displacement, scaling) can be described using four-dimensional matrix-vector multiplications. This is explained in more detail in the 3D transformations boxout. OpenGL uses the matrix stack for such multiplications.

In *recalcModelPos()* with *glLoadIdentity()* the identity matrix is loaded onto the stack. *glTranslatef (posx, posy, posz)* multiplies this matrix by a matrix which displaces an object by the co-ordinates *posx*, *posy*, *posz*. But since they all contain *0* this step could also be omitted. These displacements will also be discussed later on in this series of articles.

With *glRotatef(xangle, 1.0, 0.0, 0.0)* and *glRotatef(yangle, 0.0, 1.0, 0.0)* we multiply the matrix again with rotations about the x and the y-axis. This matrix is then used later in *DrawScene()*. In this function, the buffer must first be cleared, as well as the z-buffer, so as not to receive any incorrect depth information. Next, with *recalcModelPos()* the matrix is calculated with the new model co-ordinates. This matrix is then to be multiplied by all points of the teapot.

The teapot is created, as the result of the instruction *glutSolidTeapot(0.6)*, with a size 0.6. There is nothing behind this command but the placing of polygons, similar to the way we called up the arrow in the first program with *glVertex2f*. GLUT has more fixed, pre-defined objects. If you'd like to experiment with other objects, try out *glutWireTorus(..)* or *glutSolidDodecahedron(..)*.

The teapot was drawn in the invisible background buffer and appears with the command *glutSwapBuffers()* in the foreground. In the initialisation function *myinit()* we create, using *glLightfv(GL_ LIGHT0, GL_POSITION, light_position)*, a light at the position (*0*, *0*, *-1*) and switch it on using *glEnable(GL_ LIGHT0)*.

The last two commands set the z-buffer mode. *glDepthFunc(GL_LEQUAL)* draws only pixels whose z-value is less than that of the pixel already drawn. *glEnable(GL_DEPTH_TEST)* activates the z-buffer calculation. Finally, the compilation:

```
gcc -I . -c Teapot.c
gcc -o Teapot Teapot.o -lGL \
-lGL -lglut -lGLU
```

If everything fits, when you call up teapot, as in the picture, a white teapot should appear, which responds to the input of the letters *l* and *o* and to mouse actions.

What's next?

That's the end of our journey into the realms of OpenGL and GLUT. We have looked at the options for responses to the various inputs by the user and conjuring and moving polygons on the screen. The next installment will be about the primitives of OpenGL, how the world can become a bit more colourful and how edges can be smoothed with the aid of normal vectors. ■

The author

Thomas G E Ruge has been a converted Linux fan since Kernel 0.98 and is interested in everything to do with 3D or virtual reality. He has been driving forward the technical construction of the Virtual Reality Laboratory in the research department of Siemens AG for nearly six years and is involved in the possibilities of high-end VR with Linux clusters.

Info

Woo, Neider, Davis, Shreiner: Open GL Programming Guide 3rd ed., Verlag Addison Wesley Longman Inc.,1999
J. Encarnacao : Graphische DV 1, Oldenbourg Verlag GmbH
OpenGL homepage: http://www.opengl.org
Mesa homepage: http://www.mesa3d.org

■