Beating the System:
Using OpenGL In Delphi Apps

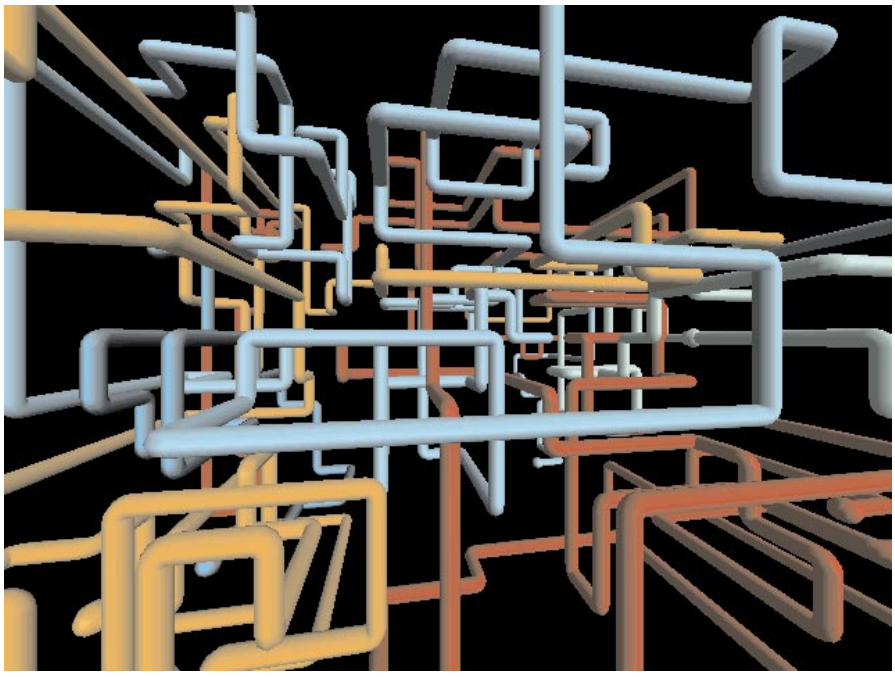
Hasta La Vista, Baby!

by Dave Jewell

Anyone who has watched movies such as Arnold Schwarzenegger's *Terminator II: Judgement Day*, or the somewhat more innocuous *Toy Story*, will have been impressed with the capabilities of modern day computer graphics technology, which can often produce photo-realistic rendering effects given time, sufficiently powerful hardware and the right software. OpenGL is an example of a high quality graphics library which is routinely used for this type of work. It provided the liquid-metal terminator in *Terminator II* and many of the monsters in *Jurassic Park*. It was originally developed by Silicon Graphics for use on their high-end workstations but, because it was written in portable C, has become something of an industry standard with implementations on many different types of hardware. Nowadays, the development of OpenGL is managed by the OpenGL Architecture Review Board, a consortium of companies which includes Silicon Graphics, Microsoft, IBM, Intel and DEC.

Microsoft originally implemented OpenGL under Windows NT only, reasoning that a relatively low-end machine running Windows 95 (or Windows 3.1!) would not be equal to the task of supporting a professional 3D rendering library. This probably made sense two or three years ago, but with today's more powerful and relatively cheap hardware, most Windows 95 machines are quite capable of supporting simple OpenGL programs. So, Microsoft quietly slipped a Windows 95 implementation of the OpenGL libraries into the latest release of Windows 95 (OSR2).

➤ *Figure 1: Journalists often get paid by the word. Now imagine a world where plumbers get paid according to the amount of pipe-work they install! This is the OpenGL Pipes screen saver which comes with Windows NT and Windows 95 OSR2.*



An Introduction To OpenGL

The OpenGL libraries comprise two DLLs: OPENGL32.DLL and GLU32.DLL, both of which should be resident in your Windows\System directory. If you're not sure whether or not you're set up for OpenGL, just look for the OpenGL screen savers in the Control Panel's Display Screen-Saver dialog. My favourite is the 3D Pipes demo (see Figure 1) which bears more than a superficial resemblance to the pipe work in our airing cupboard!

The lowest level DLL, OPENGL32.DLL, provides the core functionality of OpenGL itself, while the GLU32.DLL library contains the so-called 'utility' routines which provide a somewhat higher level of functionality. Traditionally, OpenGL ships with a third library, the GLAUX or auxiliary library, which is intended to help those who are new to OpenGL. The GLAUX library provides the highest level of functionality and it takes care of issues such as window creation and management, keyboard and event handling. Because of this, parts of the GLAUX library need to be specially written for each target platform. In the case of Delphi programmers, Borland provide the OPENGL.PAS file which contains all the declarations for the core and utility libraries. However, neither Microsoft nor Borland provide declarations or libraries for GLAUX.

In actual fact, Microsoft provide source code for the GLAUX library as part of the Win32 SDK, but, naturally, this source is in C. It would certainly be possible to convert

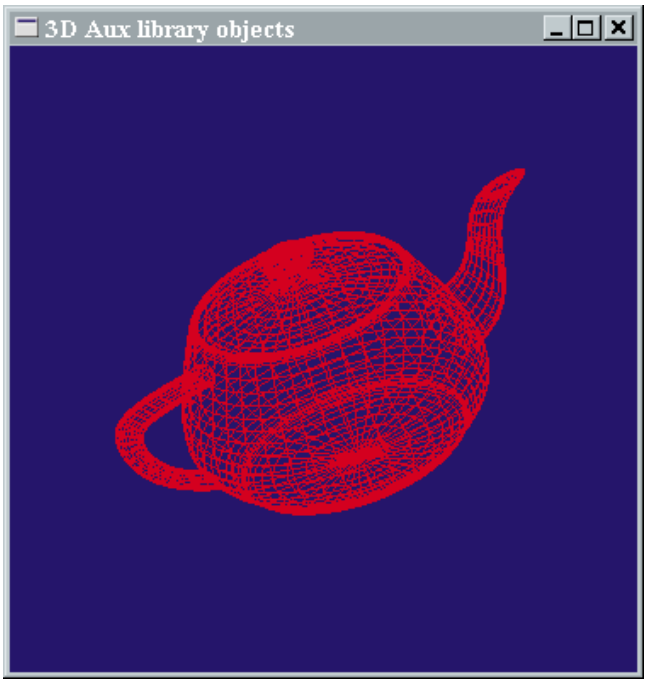
this code into Pascal but might be quite a time consuming exercise (any volunteers?!).

This is a real shame because the GLAUX library is such a valuable starting-out point for OpenGL novices. But never fear, all is not lost. I recently came across what I believe is a public-domain implementation of GLAUX, packaged as a DLL. Because this DLL was designed to work with Borland C compilers, it will also work very nicely with 32-bit Delphi. This month's disk includes a copy of the DLL, which is named, logically enough, GLAUX.DLL. I've also provided a file called GLAUX.PAS which is my translation of the various GLAUX routines into Delphi Pascal. Bear in mind that the translation isn't quite complete: the stuff that follows the final `end.` will be ignored by the compiler and represents the translation work that's still outstanding.

In the October issue, Andrew Kern and Noel Rice discussed a minimum size DirectX 3D program. I've tried to do the same with an OpenGL animation. Listing 1 shows a fairly minimalist Delphi program which makes use of the GLAUX library to do its stuff. You can see it running in Figure 2.

In order to run this program, you'll need to have GLAUX present on the path, and you'll obviously also need to have OpenGL itself installed.

Did I say Delphi program? Well... not exactly. For the purposes of simplicity, GLAUX applications are normally written as console applications which collect parameters and display debug information in the console window while displaying any graphics in a separate window. Not wishing to break with tradition, I've written this program



➤ *Figure 2: The ubiquitous teapot demo, courtesy of the GLAUX program (Listing 1). The teapot is spinning in three dimensions, but we can't demonstrate this on the printed page...*

➤ *Listing 1*

```
program AuxTest;
uses
  Windows, OpenGL, GLAux;
{$R *.RES}
var
  cSelection: Char;
{ Callback routine: called when window changes size }
procedure ChangeSize (w, h: GLsizei); stdcall;
begin
  // Prevent a divide by zero
  if h = 0 then h := 1;
  // Set viewport and clipping volume
  glViewport (0, 0, w, h);
  glLoadIdentity;
  if w <= h then
    glOrtho(-100.0, 100.0, -100.0, 100.0 * h / w, -100.0,
      100.0)
  else
    glOrtho (-100.0, 100 * w / h, -100.0, 100.0, -100.0,
      100.00);
end;
{ Callback routine: called by AUX library to render scene }
procedure RenderScene; stdcall;
begin
  // Clear the screen
  glClear (GL_Color_Buffer_Bit);
  // Rotate one degree around each axis
  // (Note we don't call LoadIdentity() so
  // this rotation is cumulative
  glRotatef (1.0,1.0,0.0,0.0);
  glRotatef (1.0,0.0,1.0,0.0);
  glRotatef (1.0,0.0,0.0,1.0);
  // Draw the selected object
  case cSelection of
    'a':    auxWireCone (30.0, 75.0);
    'b':    auxWireCylinder (30.0, 75.0);
    'c':    auxWireDodecahedron (75.0);
    'd':    auxWireIcosahedron (75.0);
    'e':    auxWireOctahedron (75.0);
    'f':    auxWireSphere (75.0);
    'g':    auxWireTeapot (50.0);
    'h':    auxWireTetrahedron (75.0);
    'i':    auxWireTorus (20.0, 50.0);
    'j':    auxWireCube (75.0);
    'k':    auxWireBox (75.0,75.0,75.0);
  end;
  glFlush;
  // Swap drawing to screen
  auxSwapBuffers;
end;
procedure Main;
begin
  // Display a menu of objects to draw
  Writeln ('Select Wire object to draw:');
  Writeln;
  Writeln ('a - Cone');
  Writeln ('b - Cylinder');
  Writeln ('c - Dodecahedron');
  Writeln ('d - Icosahedron');
  Writeln ('e - Octahedron');
  Writeln ('f - Sphere');
  Writeln ('g - Teapot');
  Writeln ('h - Tetrahedron');
  Writeln ('i - Torus');
  Writeln ('j - Cube');
  Writeln ('k - Box');
  // Validate and accept selection
  cSelection := #0;
  while not (cSelection in ['a'..'k']) do begin
    Writeln;
    Write ('Selection: ');
    Readln (cSelection);
  end;
  // Setup the AUX library window for double buffer
  auxInitDisplayMode (Aux_Double or Aux_RGBA);
  auxInitPosition (100, 100, 250, 250);
  auxInitWindow ('3D Aux library objects');
  // Set background to blue
  glClearColor (0.0, 0.0, 1.0, 1.0);
  // Set drawing color to Red
  glColor3f (1.0, 0.0, 0.0);
  // Establish window resize function
  auxReshapeFunc (@ChangeSize);
  // Establish idle function
  auxIdleFunc (@RenderScene);
  // Start main loop
  auxMainLoop (@RenderScene);
end;
begin
  Main;
end.
```

the same way. In fact, it's a straight port of a Microsoft SDK sample. It can only be called a Delphi program by virtue of the fact that I used Delphi to write it! There are no forms, nor any use of the VCL library. Listing 1 simply corresponds to the .DPR file itself (the program with source code and DLL can be found in AUXTEST.ZIP on this month's disk).

GLAUX For Beginners...

When you run the code, you'll see a console window from where you can select one of several geometric shapes. Enter the character for the required shape, hit `Enter` and you'll see a wire-frame representation of the selected shape rotating in all three dimensions. You should end the program by closing the graphical window, *not* the console window.

So how does it work? As mentioned earlier, the GLAUX provides a relatively high level view of reality. It allows us to easily set up an OpenGL runtime environment and experiment with different OpenGL calls. You can easily distinguish between different calls because of a naming convention that's used throughout OpenGL: API routines in the GLAUX library are always preceded by the letters `aux`, those in the utility library are preceded by `glu` and those in the core library by `gl`. Easy!

The first thing our `AuxTest` program does is ask the user what type of shape they want to draw. This is just classical Pascal and I won't bore you with the details! Once that's done, a call is made to `auxInitDisplayMode`. This tells OpenGL that we want to use double-buffering so as to provide smooth animation. If you're not familiar with the term, it's a common technique whereby the next frame of an animation is drawn into an off-screen bitmap while the current frame is being displayed. When it comes time to change frames, the off-screen bitmap instantly replaces the displayed frame. In this way, the user never sees the individual stages in the drawing of a complex graphic: the whole thing appears in one fast operation and flicker is eliminated. Following this, the `auxInitPosition` call is used to specify an initial X,Y position for the graphics window and an initial width and height. Lastly, the `auxInitWindow` call is used to set the window title.

At this point, we've initialised our window. Now we need to specify what appears inside it. `glClearColor` is the first core-level OpenGL call we've used. It tells OpenGL what colour to use when clearing the window. Because we specified a colour mode of `Aux_RGBA` when setting the display mode, any colour values we supply will be interpreted as separate RGB intensities with an additional alpha channel. In the same way, the `glColor3f` call indicates the colour we want to use for drawing.

It's worth noting that some OpenGL calls are functional duplicates of one another, they have the same job, but differ in the type of parameters they accept. The name `glColor3f` might look odd, but it's telling you that this routine is all about setting colour and that this variant takes three floating point numbers as input.

At this point, we're almost done. The `Main` procedure now sets up the address of three call-back routines. One is a routine to execute when the size of the graphics window is changed. A second is a routine to call during idle time, and thirdly there's a routine which OpenGL calls in order to render the required scene. In this particular case, the second and third routines are one and the same: that's because we want our animation to execute as often as possible. When writing call-back routines with Delphi Pascal, *always*, *always*

OpenGL Versus DirectX 3D

As you'll no doubt appreciate, the October issue of *The Delphi Magazine* carried an article on DirectX 3D graphics. It's tempting to ask why we need two different and incompatible rendering libraries? The answer, of course, is that we probably don't. As with most such battles, only one system will ultimately triumph: I have an old Betamax VCR gathering dust in a corner of my office to prove it!

Nevertheless, each system has its own unique strengths and weaknesses. I'm not an expert with either library, but, put simply, DirectX 3D has the advantage of speed whereas OpenGL is a portable standard that's available on a wide variety of platforms. In addition, few would disagree that OpenGL offers a higher degree of photo-realism than can currently be obtained from DirectX 3D.

My view is that 3D graphics on the IBM PC platform is still an immature technology. At the present time, it's undeniable that DirectX has the edge in terms of market dominance, fuelled by the need for game developers to get every last ounce of speed from the available hardware. However, hardware performance continues to rise while prices plummet. Just as my collection of books always expands to fill the available space, so the complexity of modern software (and the end-user's expectations of graphic performance) always expands to match the capabilities of the available hardware. Once people are happy with animation speed, they will want animation which is more photo-realistic. Want to create something like *Toy Story* on your home PC? I'm sure it'll be a reality within the next ten years.

This being the case, my view of OpenGL is that it's experiencing something of a temporary hiatus while the hardware catches up. Sales of Windows NT never really took off until the hardware was not only powerful enough but also cheap enough for the mass market. So it is with OpenGL. For the moment, DirectX rules the roost, but stay tuned...

Note: for an interesting (but probably somewhat biased) technical comparison of OpenGL and DirectX3D, you can check out the OpenGL area of the Silicon Graphics Web site at www.sgi.com. While you're at it, feast your eyes on those tasty graphics workstations and ask yourself what would make a suitable Xmas pressie for yours truly....

remember this one golden rule: call-back routines must be defined using the `stdcall` directive. If you forget to do this, your code will compile perfectly and then crash inexplicably!

The `ChangeSize` routine is called to adjust OpenGL's co-ordinate system so that the size of the drawing is consistent with the new window size. I'm not going to attempt to explain what `glLoadIdentity` does or those mysterious looking calls to `glOrtho`. Doing so would require a lot of background detail on clipping volumes and viewports.

As I'm sure you appreciate, my mission here isn't to explain every single possible OpenGL call but rather to give you a flavour for writing OpenGL code with Delphi. If you want to learn more about the details, I recommend a few books at the end of this article.

More interesting is the `RenderScene` call. This starts off by clearing the screen, by which I mean the background bitmap: as previously mentioned, this is a double-buffering example and therefore all drawing operations are relative to the hidden buffer.

Next, it issues three calls to `glRotatef`. These calls effectively rotate the displayed shape one 'step' along each of the three axes. Without these calls, the program would be pretty boring. Having done this, the rendering code calls one of the assorted `auxWireXXX` routines to draw the chosen shape. The GLAUX library includes a number of pre-defined shapes, including the ubiquitous teapot. Listen you DirectX guys, OpenGL had teapots first!

Finally, `glFlush` is called to flush any pending drawing commands. Bear in mind that OpenGL has its origins in graphics hardware which pipelines drawing primitives for improved performance. Last but not least, `auxSwapBuffers` does the all-important job of swapping the background buffer with the displayed buffer. This means that the next call to `RenderScene` will be operating on what was the displayed buffer, but is now the off-screen buffer.

➤ *Figure 3: Here's the results of running the 'native-mode' OpenGL program from Listing 2. This dispenses with the need for GLAUX.DLL, but requires more complexity in the program code.*

Doing It The Delphi Way

Of course, it would somewhat remiss of me if I left things there. To use OpenGL properly, we want to be able to integrate it into a 'normal' Delphi application without having to mess around with console applications or anything like that. Listing 2 is a complete OpenGL program, written using Delphi 3. You can see the program running in Figure 3. Again, I converted this code from a Microsoft SDK C-based sample.

If you're interested in other sample OpenGL sources, you can find many interesting snippets in Microsoft's Win32 SDK. This provides the source code to numerous sample programs together with the code for various screen savers, including the Pipes saver and the 3D Maze program.

In order to access the full power of OpenGL, we have to dispense with the GLAUX library. This means, however, that the program now has to take more responsibility itself. In particular, quite a bit of the code in Listing 2 is given over to palette management. However, if you look carefully you will see that there are many similarities between this and the GLAUX code. For example, in the `FormPaint` method, a routine called `RenderScene` is called to create the scene via a series of OpenGL drawing commands. You'll also notice similarities in the code which responds to window size changes.

Of particular interest in this routine is the deeply cool `wglUseFontOutlines` routine. This call creates a set of display lists that correspond to the outline of the selected TrueType font. You can think of display lists as being OpenGL's equivalent of a Windows metafile: it's a way of grouping together a series of drawing operations for rapid execution when needed. The `wglUseFontOutlines` routine works in conjunction with the font engine and converts the font's outlines into a series of low-level drawing operations. The result can then be 'extruded' rotated and lit as shown in Figure 3.

Over To You

There's a lot more that I could say about the code in Listing 2 and about OpenGL itself. I've barely scratched the surface and I'm fresh out of space. What I set out to do was write a motivational article to give you a feel for OpenGL programming in Delphi, and to that extent I hope I've succeeded.

If you want to explore OpenGL further, I recommend that you start out with the GLAUX library, experiment by adding functionality to that and then , when you're feeling confident with OpenGL itself, you can move over to a full-fledged application which doesn't require the 'hand-holding' offered by the GLAUX code.

Have fun!

Books

What you definitely will need to get very much further with OpenGL is a decent book on the subject. Let me finish by giving you my own recommendations:

- *OpenGL Super-Bible* by Richard S. Wright Jr. and Michael Sweet. Waite Group, ISBN: 1-57169-073-5. Expensive but worth it.
- *OpenGL Reference Manual, 2nd Edition* by the OpenGL Architecture Review Board. Addison-Wesley, ISBN: 0-201-46140-4. Purely reference material.
- *OpenGL Programming Guide, 2nd Edition* by OpenGL Architecture Review Board. Addison-Wesley, ISBN: 0-201-46138-2.

The second and third books constitute the official OpenGL Bible and they are not platform-specific. You'll need them if you really get into OpenGL. The first book is more readable, very practical and is targeted specifically at Windows 95 and NT developers.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave as Dave@HexManiac.com.

➤ *Listing 2*

```
unit ugltest;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, OpenGL;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    hRC: hGLRC;              // permanent Rendering context
    dc: hDC;                 // private GDI Device context
    hPal: hPalette;          // global palette handle
    procedure WMQueryNewPalette(var Message:
      TWMQueryNewPalette); message WM_QueryNewPalette;
    procedure WMPaletteChanged(var Message:
      TWMPaletteChanged); message WM_PaletteChanged;
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
// Select the pixel format for a given device context
procedure SetDCPixelFormat (dc: hDC);
var
  PixelFormat: Integer;
  pfd: TPixelFormatDescriptor;
begin
  FillChar (pfd, sizeof (pfd), 0);
  with pfd do begin
    nSize      := sizeof (pfd);
    nVersion   := 1;
    dwFlags    := PFD_Draw_To_Window or PFD_DoubleBuffer
                  or PFD_Support_OpenGL;
    iPixelType := PFD_Type_RGBA;
    cColorBits := 24;
    cDepthBits := 32;
    iLayerType := PFD_Main_Plane;
  end;
  // Choose pixel format best matching that described in pfd
  PixelFormat := ChoosePixelFormat (dc, @pfd);
  // Set the pixel format for the device context
  SetPixelFormat (dc, PixelFormat, @pfd);
end;
// If needed creates 3-3-2 palette for
// the device context listed
function GetOpenGLPalette (dc: hDC): hPalette;
var
  pPal: ^TLogPalette;
  pfd: TPixelFormatDescriptor;
  i, Colors, PixelFormat: Integer;
  RedRange,GreenRange,BlueRange: Byte;
begin
  Result := 0;
  // Get pixel format index,
  // retrieve pixel format description
  PixelFormat := GetPixelFormat (dc);
  DescribePixelFormat (dc, PixelFormat, sizeof (pfd), pfd);
  // Does this pixel format require a palette?
  // If not, do not create a palette and just return 0
  if (pfd.dwFlags and PFD_Need_Palette) = 0 then Exit;
  // Number of entries in palette: 8 bits yields 256 entries
  Colors := 1 shl pfd.cColorBits;
  // Allocate space for logical palette structure
  // +  palette entries
  GetMem (pPal, sizeof (TLogPalette) +
    (Colors * sizeof (TPaletteEntry)));
  try
    // Fill in palette header
    pPal^.palVersion := $300;
    pPal^.palNumEntries := Colors;
    { Build mask of all 1s. This creates a number
      represented by having the low order x bits set, where
      x = pfd.cRedBits, pfd.cGreenBits, and pfd.cBlueBits }
    RedRange   := (1 shl pfd.cRedBits)   - 1;
    GreenRange := (1 shl pfd.cGreenBits) - 1;
    BlueRange  := (1 shl pfd.cBlueBits)  - 1;
    // Loop through all the palette entries
    for i := 0 to Colors - 1 do begin
      // Fill in the 8-bit equivalents for each component
      pPal^.palPalEntry[i].peRed :=
        (i shr pfd.cRedShift) and RedRange;
      pPal^.palPalEntry[i].peRed :=
        Trunc(pPal^.palPalEntry[i].peRed*255.0/RedRange);
      pPal^.palPalEntry[i].peGreen :=
        (i shr pfd.cGreenShift) and GreenRange;
      pPal^.palPalEntry[i].peGreen :=
        Trunc(pPal^.palPalEntry[i].peGreen *
          255.0 / GreenRange);
      pPal^.palPalEntry[i].peBlue :=
        (i shr pfd.cBlueShift) and BlueRange;
      pPal^.palPalEntry[i].peBlue :=
        Trunc(pPal^.palPalEntry[i].peBlue*255.0/BlueRange);
      pPal^.palPalEntry[i].peFlags := 0;
    end;
    // Create the palette
    Result := CreatePalette (pPal^);
    // Select and realize the palette for device context
    SelectPalette (dc, Result, False);
    RealizePalette (dc);
  finally
    // Free memory used for the logical palette structure
    FreeMem (pPal, sizeof (TLogPalette) +
     (Colors * sizeof (TPaletteEntry)));
  end;
end;
// Perform any needed initialization on rendering context.
//  Here it sets up and initializes the lighting for scene.
procedure SetupRC (dc: hDC);
const
  // Light values and coordinates
  whiteLight: array[0..3] of GLfloat =
    (0.4, 0.4, 0.4, 1.0);
  diffuseLight: array[0..3] of GLfloat =
    (0.8, 0.8, 0.8, 1.0);
  specular: array[0..3] of GLfloat =
    (0.9, 0.9, 0.9, 1.0);
  lightPos: array[0..3] of GLfloat =
    (-100.0, 200.0, 50.0, 1.0);
var
  font: hFont;
  logFont: TLogFont;
  agmf: array[0..127] of TGlyphMetricsFloat;
begin
  // Setup the Font characteristics
  FillChar (logfont, sizeof (logfont), 0);
  logfont.lfHeight := -10;
  logfont.lfWeight := FW_Bold;
  logfont.lfCharSet := Ansi_CharSet;
  logfont.lfOutPrecision := Out_Default_Precis;
  logfont.lfClipPrecision := Clip_Default_Precis;
  logfont.lfQuality := Default_Quality;
  logfont.lfPitchAndFamily := Default_Pitch;
  lstrcpy (logfont.lfFaceName, 'Times New Roman');
  // Create the font and display list
  Font := CreateFontIndirect (logfont);
  try
    SelectObject (dc, Font);
    { create display lists for glyphs with 0.1 extrusion and
      default deviation. Display list numbering starts at
      1000 (it could be any number) }
    wglUseFontOutlines (dc, 0, 128, 1000, 0.0, 1.0,
      WGL_Font_Polygons, @agmf);
{ continued on next page... }
```

```
{ Listing 2 continued }
  finally
    DeleteObject (Font);
  end;
  // Hidden surface removal
  glEnable (gl_Depth_Test);
  glEnable (gl_Color_Material);
  glClearColor (0.0, 0.0, 0.0, 1.0);
  glEnable (gl_Lighting);
  glLightfv (gl_Light0, gl_Ambient, @whiteLight);
  glLightfv (gl_Light0, gl_Diffuse, @diffuseLight);
  glLightfv (gl_Light0, gl_Specular, @specular);
  glLightfv (gl_Light0, gl_Position, @lightPos);
  glEnable (gl_Light0);
  glColorMaterial (gl_Front, gl_Ambient_And_Diffuse);
  glMaterialfv (gl_Front, gl_Specular, @specular);
  glMateriali (gl_Front, GL_Shininess, 128);
  glColor3ub (0, 255, 0);   // Green 3D Text
  glClearColor (0.0, 0.0, 0.0, 1.0);  // Black background
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Window creation, setup for OpenGL
  dc := GetDC (Handle);
  SetDCPixelFormat (dc);
  hPal := GetOpenGLPalette (dc);
  // Create the rendering context and make it current
  hRC := wglCreateContext (dc);
  wglMakeCurrent (dc, hRC);
  SetupRC (dc);
end;
procedure RenderScene (const TheText: String);
begin
  // Clear the window with current clearing color
  glClear (gl_Color_Buffer_Bit or gl_Depth_Buffer_Bit);
  // Save the matrix state and do the rotations
  glMatrixMode (gl_ModelView);
  // Draw the string
  glListBase (1000);
  glPushMatrix;
  glRotatef (6.0, 1.0, 0.0, 0.0);
  glRotatef (350.0, 0.0, 1.0, 0.0);
  glCallLists(Length(TheText), gl_Unsigned_Byte,
    PChar(TheText));
  glPopMatrix;
  // Flush drawing commands
  glFlush;
end;
procedure TForm1.WMQueryNewPalette(
  var Message: TWMQueryNewPalette);
begin
  Inherited;
  if hPal <> 0 then begin
    // Selects the palette into the current device context
    SelectPalette (dc, hPal, False);
    // Map entries from current palette to system palette
    RealizePalette (dc);
    // Repaint, forces remap of palette in current window
    InvalidateRect (Handle, Nil, False);
  end;
end;
procedure TForm1.WMPaletteChanged(
  var Message: TWMPaletteChanged);
begin
  Inherited;
{ continued opposite... }
```

Dealing With 'Rogue' DLLs

In time honoured fashion, this month's column includes a bug-fix for last month's code. The fix is particularly bizarre because, believe it or not, it isn't *my* bug; rather, it's one of those weird programming scenarios that happen from time to time.

Let me explain. You'll remember that I developed a little utility which mimicked the behaviour of Merlin's Executable Viewer, allowing you to view the units compiled into a particular executable, whether it be an application or DLL. By chance, I was browsing around my hard disk using the Merlin utility and I happened to come across GExperts (yet another Delphi add-on, written by a chap called Gerald Nunn of Canada). To my surprise, I discovered that whenever I tried running the Executable Viewer on the GEXPERT3.DLL (the heart of GExperts), it consistently crashed with an access violation. No other DLL seemed to produce this behaviour.

"Oh well", I thought, "If you want the job doing properly, do it yourself!" Confidently, I fired up my own little utility and pointed it at the offending DLL, fully expecting it to work flawlessly. But pride comes before a fall! To my amazement, last month's program crashed in exactly the same way. Obviously, there was something deeply odd happening here... I fired up Delphi and tried single-stepping through my code. Everything went fine until I got to the `FreeLibrary` call. At this point, the access violation was generated.

I spent some time chewing over this problem with Mike and John (the authors of Merlin) and more time peeking inside the GExperts DLL. To cut a long story short, it turns out that GEXPERT3.DLL is an example of a "rogue" DLL: it doesn't play by the rules. It's easy to prove this for yourself by simply reducing the code for last month's program to just two statements: a call to `LoadLibrary`, followed immediately by the corresponding call to `FreeLibrary`. Make these changes and you'll find that the program will still crash. By no stretch of the imagination should this ever happen when working with a well-behaved DLL.

So how can a badly-behaved DLL cause this sort of crash to happen? This is conjecture on my parts, but I suspect that internally, within its own initialisation code, GEXPERT3.DLL is loading *itself* (via a call to the `System` unit routine `LoadResourceModule`, in this particular case) and that's what is causing problems when `FreeLibrary` is called to unload the library.

Fortunately, there's a simple fix which can be used to protect against this type of problem. Bear in mind that this has general applicability outside of my unit peek program: I'd advise you to use it whenever you want to access a third-party DLL or executable with a view to getting a resource handle. The trick is simply to use a new routine, `LoadLibraryEx`. Simply find the `LoadLibrary` statement in last month's code and replace it with this:

```
hLib := LoadLibraryEx(
  PChar(OpenDialog.FileName), 0,
  Load_Library_As_DataFile);
```

By using `LoadLibraryEx` and specifying `Load_Library_As_DataFile` as the third parameter, Windows will load the specified library (or executable) without executing any initialisation code which it might contain. This fixes the problem and should protect against any future encounters with other rogue DLLs.

In fairness to Gerald, I understand that the latest version of GExperts fixes the problem, but it's obviously important to protect your code against any other 'rogues' that might be out there...

```
{ Listing 2 continued }
  // Don't do anything if palette does not exist, or if
  // this is the window that changed the palette.
  if (hPal <> 0) and (Handle <> Message.PalChg) then begin
    // Select the palette into the device context
    SelectPalette (dc, hPal, False);
    // Map entries to system palette
    RealizePalette (dc);
    // Remap current colors to the newly realized palette
    UpdateColors (dc);
  end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  // The painting function. This message sent by Windows
  // whenever the screen needs updating.
  RenderScene ('Delphi 3.0');
  SwapBuffers (dc);
  ValidateRect (Handle, Nil);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  // Deselect the current rendering context and delete it
  wglMakeCurrent (dc, 0);
  wglDeleteContext (hRC);
  if hPal <> 0 then DeleteObject (hPal);
  ReleaseDC (Handle, dc);
end;

{ Change viewing volume + viewport.
  Called when window resized }
procedure ChangeSize (w, h: GLsizei);
const
  nRange: GLFloat = 125.0;
begin
  // Prevent a divide by zero
  if h = 0 then h := 1;
  // Set Viewport to window dimensions
  glViewport (0, 0, w, h);
  // Reset coordinate system
  glMatrixMode (gl_Projection);
  glLoadIdentity;
  // Establish clipping volume
  // (left, right, bottom, top, near, far)
  if w <= h then
    glOrtho(-nRange, nRange, -nRange*h/w, nRange*h/w,
      -nRange*2.0, nRange*2.0)
  else
    glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange,
      -nRange*2.0, nRange*2.0);
  // Set up transformation to draw the string.
  glTranslatef (-110.0, 0.0, -5.0);
  glScalef(60.0, 60.0, 60.0);
  glMatrixMode (gl_ModelView);
  glLoadIdentity;
end;

procedure TForm1.FormResize (Sender: TObject);
begin
  ChangeSize (ClientWidth, ClientHeight);
  InvalidateRect (Handle, Nil, False);
end;

end.
```