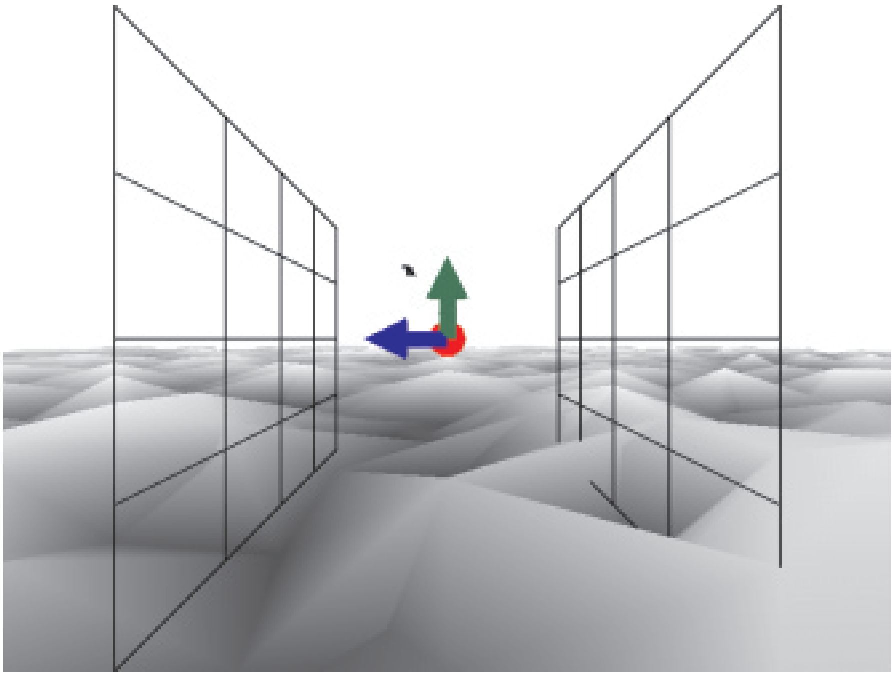
# OpenGL: Under The Hood

*by John Hutchings*

If you have a need for 3D graphics and the good fortune to own a powerful workstation then applications which use 3D APIs, such as OpenGL, produce pretty impressive results. The mining industry needs to manipulate complex 3D data sets representing a mine as part of the mine planning process and I have often envied the spectacular graphics performance achieved while 'flying through' an open cut mine or rotating an ore body. Recently, with a move to Windows NT and the addition of some decent hardware (dual 233Mhz Pentium Pro with a Diamond Fire GL3000 graphics card sporting the OpenGL accelerating GLINT chip set and 16Mb of RAM), I too can almost achieve that impressive graphics performance. The demonstration program included with this month's disk will run at a smooth 25 to 30 frames per second using the above hardware and with a reasonably high polygon count (the 'random terrain' scene seen in Figure 1). Of course, if I drop back to a system without the hardware acceleration the scene will run at a more pedestrian 2 to 10 frames per second.

I am involved with the writing of customised 3D CAD based software for mining related industries. My need is for a powerful and flexible 3D API which will handle basic 3D primitives (including a 3D point, 3D line and 3D triangle) with the potential to do a lot more in the future. To date I have handled the 3D graphic pipeline by using a simple 3D API to project 3D points onto the screen and produce the equivalent 2D screen coordinate points. Then I use the Windows GDI, through a `Canvas`, to draw my 3D objects. OpenGL will now allow me to pass the 3D point directly to the OpenGL API, which will handle the data right through to drawing it onto the screen. The API is highly optimised and will obviously make appropriate use of any hardware acceleration which may be present.



➤ *Figure 1*

I have chosen OpenGL for a number of reasons. Firstly, the API seems to have a future with Microsoft and with graphics card manufacturers. Secondly, the API is function and procedure driven, which means an easier interface from Pascal (the Inprise folk provide the v1.1 interface in the OPENGL.PAS unit). Thirdly, the system handles the basic 3D points and 3D lines as well as 3D triangles (polygons). Other Graphics APIs do not handle them all. If I do not need blistering animation speed (as basic CAD systems don't) then the quality of the render is higher than, say, Direct3D. It is always possible (and seemingly acceptable) to swap to a lower resolution of rendering while moving. Lastly, the system is tried, tested and well thought out. I have always succeeded in doing what I need to do, even if discovering just how to do it can be frustrating.

For the rest of this article I will discuss some of the implementation issues which arise in using OpenGL, together with my solutions. I have encapsulated the work into a set of Delphi 4 components. My aim was to produce a very general set which would wrap the OpenGL API into a Delphi-esque form. My focus for this exercise is on managing the primitives within Delphi. Just as a Delphi user can revert to the Windows API when required, so a user of these components can revert to the OpenGL API. I will assume you have read the previous articles in *The Delphi Magazine* articles on OpenGL (in Issue 28, December 1997, and Issue 34, June 1998).

Note that the code included with this article is my development code. It has a number of areas still undeveloped and a number of areas not discussed here. The demonstration application has a simple control form which allows the testing of the components and also demonstrates how to interface with the components. It is fair to say that the code is not as widely tested as I would like (maybe you could give me some feedback). However, the basic OpenGL management is in place and I feel it is worth sharing at this stage.

## Some Of The Basics

I wanted to construct a basic Delphi component which would allow me to use OpenGL painlessly. To do this, it is first necessary to understand the OpenGL system as implemented in Windows.

OpenGL is a 32-bit API developed and released in 1992 by Silicon Graphics Inc (SGI). OpenGL is now an open standard with enhancements decided by the OpenGL Architecture Review Board (ARB), whose founding members included SGI, Digital Equipment, IBM, Intel and Microsoft.

OpenGL has 120 core functions which are common across all the supported platforms. The convention adopted for the calls is to begin them with `gl`, for example `glVertex3d`. This core is supplemented by a set called the GL Utilities, which can are identified by the `glu` prefix. For Microsoft to implement OpenGL in Windows (95/98/NT) they needed to supply a further set which would manage the basics of the screen buffers and the key interface to the OpenGL session. These are known as the *wiggle* functions, as they begin with a `wgl` prefix.

The OpenGL library comes as two DLLs: OPENGL32.DLL and GLU32.DLL. On NT and Windows 98 these are installed automatically with the operating system. On earlier versions of Win95 you had to download them from Microsoft's website. The writers of OpenGL, Silicon Graphics Inc, have also written their own Windows versions. You can find these library DLLs on the web and, if you wanted to test them, you would have to 'hook' them into the OPENGL.PAS unit. They are called OPENGL.DLL and GLU.DLL. SGI have put in some speed improvements which are also linked to their Cosmo Worlds initiative.

To connect to the OpenGL library the Inprise folk have provided a unit called OPENGL.PAS. This unit combines interfaces to the OPENGL32.DLL and GLU32.DLL libraries. The only shortcomings I have found with this interface are firstly that the OpenGL library is at version 1.2 while the OPENGL.PAS interface supports only 1.1, and secondly that some of the declared data structures are not to my personal taste. I have written an interface for the extra functionality provided in 1.2, which I have not yet fully tested (see OPENGL12.PAS on the disk).

To interact with an OpenGL session you need to cover the following areas. The main interaction is through a window. Since Windows developers know all about device contexts, Microsoft introduced the *rendering context*. Just as the device context links to a drawing device so the rendering context links to an OpenGL session. You may want to build more than one window which is running an OpenGL session within your application. Each OpenGL window session will have its own unique rendering context. However, only one rendering context can be current at any one time. This means that before carrying out any rendering the rendering context needs to be made current. OpenGL also supports multithreading and client/server operation (which are beyond the scope of this article).

One strategy for managing OpenGL is to generate a rendering context each time a window needs to repaint. The disadvantage of this is speed. A second approach, which is more memory hungry, is to create a rendering context during window creation and hold onto it for the life of the window. This also implies holding onto the device context (a no-no with Windows 3.x but not such a problem in Windows 95/98/NT).

To create a rendering context you first need a window handle then the device context. If you have a valid window handle, device context and rendering context then your OpenGL session is up and running.

The basic Windows events to manage include the following.

*On Create*. Before the window is created the correct style and format need to be chosen. I have tackled this by overriding the `CreateParams` procedure and setting the style flags, as shown in Listing 1.

Once the window, associated handle and device context are created then the rendering context needs to be created. To do this first the pixel format of the window needs to be set. At first I thought I'd handle this process by overriding the `CreateHandle` procedure, as this is the point where a valid window handle exists. However, I have found it more effective if I override the `CMShowingChanged` message and build the rendering context just prior to the window being shown for the first time.

The process of building a pixel format is to first set up a basic request record with stuff like how many colours we would like. The pixel format can only be set once for a given window. Listing 2 shows the basic settings and calls.

The software chooses the closest display that the hardware can manage. Your hardware may only support 8 bits for colour, so an appropriate pixel format will be chosen. There may also be a need to set up a colour palette. Set the pixel format and you are ready to create the rendering context with `wglCreateContext`, passing the device context of the new window.

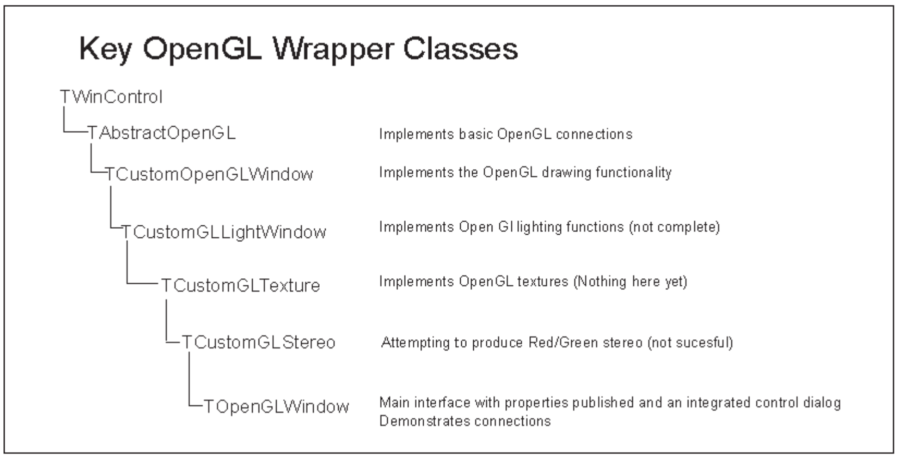
*On EraseBackground*. This event must be handled to let Windows know that OpenGL will clear the window to prevent flicker.

*On Paint*. This is the core to the creation of the OpenGL image and will be covered later. The windows rendering context must be made current otherwise no rendering will take place.

*On ReSize*. When the window is resized the OpenGL session must

➤ *Listing 1*

```
procedure TAbstractOpenGL.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params); { call the inherited first }
  Params.Style := WS_CHILD + WS_CLIPCHILDREN + WS_CLIPSIBLINGS + WS_BORDER;
  //set up the windows style flags MUST have ClipChildren and ClipSiblings
  Params.WindowClass.style := CS_VREDRAW + CS_HREDRAW + CS_DBLCLKS + CS_OWNDC;
  // set up the windowclass style MUST have VRedraw, HRedraw and OwnDC
end;
```


Key OpenGL Wrapper Classes
TWinControl
TAbstractOpenGL
Implements basic OpenGL connections
TCustomOpenGLWindow
Implements the OpenGL drawing functionality
TCustomGLLightWindow
Implements Open Gl lighting functions (not complete)
TCustomGLTexture
Implements OpenGL textures (Nothing here yet)
TCustomGLStereo
Attempting to produce Red/Green stereo (not sucesful)
TOpenGLWindow
Main interface with properties published and an integrated control dialog
Demonstrates connections

➤ *Figure 2*

be notified of the new location and dimensions. This is done by overriding the `WMSize` event and setting the `glViewport` appropriately.

*On Destroy*. Prior to the shutdown of an OpenGL window, the rendering context must be made not current then shut down:

```
wglMakeCurrent(...);
wglDeleteContext(...);
```

The normal window shutdown can then take place.

## Delphi OpenGL Components

Figure 2 contains a basic listing of the components created in the GLDemo application on the disk, which demonstrates the use of the main component. This application has a simple tool form which allows you to explore the current features. A number of test scenes are pre-built. I have put this together as a basic testbed. Obviously these components would be linked differently into production applications, probably as components dropped onto a form.

## 3D Drawing With TCustomOpenGLWindow

I believe that the approach to, say, drawing a 3D line on the screen should be similar to drawing a 2D line. However, to get this functionality implemented I needed to step back a little.

Generally, if a new component needs a window handle and device context, as an OpenGL session does, the ancestor would be a `TCustomControl`. I found, though, that to manage the specifics of drawing to a high resolution bitmap it was better to descend from the `TWinControl` and manage a GDI canvas myself.

If a Delphi programmer wants to draw onto a graphics-capable component then a `Canvas` will implement all the detail of the drawing. I have taken this tack with a `GLCanvas` implementing the 3D rendering and a standard `Canvas` handling the normal painting.

Why do I need both? The drawing logic I have implemented allows a programmer to draw onto the 3D OpenGL window in any or all of the following steps.

*Step 1: OpenGL 2D background rendering*. This is used to draw things at the back of the scene, for example a reference grid. Drawing here may be hidden by later drawing in front and so appears to be always at the back of the rendered scene. It is not a true 2D drawing surface, but a 3D surface aligned to the screen and with a depth of 1 GL unit. Objects will appear if the Z value is between -1 and 1 and the X and Y fall within the screen boundaries.

*Step 2: OpenGL 3D main rendering*. All 3D objects within the scene are projected onto the screen. The OpenGL pipeline has the depth culling enabled, so sorting objects by distance from the viewer.

*Step 3: OpenGL 2D foreground rendering*. Any foreground 2D objects always sit over, or at the front of, the scene, for example title blocks or text labels. The same logic applies as for the 2D background.

*Step 4: Standard Windows 2D GDI draw*. This is normally thought of as the 'paint' of the window. This can be used instead of step 3 above for text labels. This painting will overlay all drawing done in steps 1 to 3. Most OpenGL texts state that you cannot use the GDI if you are double buffering; however, the trick is to finish all OpenGL rendering, swap the buffer into the screen (or 'front' buffer in OpenGL parlance), then make the GDI calls.

The steps above also indicate the sequence of drawing. Listing 3 is the implementation of the rendering process.

All the rendering steps (steps 1 to 3 inclusive) are handled by the OpenGL pipeline, which involves, without some serious intervention, the recalculation and redrawing of the entire screen. This is not terribly efficient if we want to draw only a temporary mouse drag image or a zoom-to rectangle. By using the GDI the programmer can draw temporary data directly onto the `Canvas` without the penalty of a complete screen rebuild. If you enable the basic scene in the demo then you will see a line and text which will never move. This is an example of using the Windows GDI.

I have also implemented a Head-Up-Display (HUD) hook

➤ *Listing 2*

```
with fAPPFD do begin              // this is a pre-decalared data type
  nSize:=SizeOf(fAPPFD);
  nVersion:=1;                    // Must Be  this value
  dwFlags:=(PFD_Draw_To_Window or // rendering to a window
    Pfd_Support_OpenGL or         // supporting OpenGL
    pfd_swap_copy or              // swapcopy will swap in a buffer but keep copy
    PFD_DOUBLEBUFFER) ;           // for smooth animation and screen redraws
  iPixelType:=PFD_TYPE_RGBA;      // see current settings in GLFuncs
  cColorBits:=24;                 // 24 bit colour (can be 8-32)
  cDepthBits:=16;                 // manage the depth buffer with 16 or 32 bits
  cStencilBits:=1;                //need only 1 bit for the stencil buffer
  cAccumBits:=32;        //32 bits for extra precision in the accumulation buffer
  iLayerType:=PFD_Main_Plane;     //no choice in Windows
end;
// find the closest fit to the requested
fPixelFormat:=ChoosePixelFormat(fRenderDC,@fAPPFD);
Result:=SetPixelFormat(fRenderDC,fPixelFormat,@faPPFD); //set the pixel format
```

```
Procedure TCustomOpenGLWindow.GLRenderWindow(
  DoSwap : Boolean);
Begin
  fCanvas.Lock;  // lock the canvas from others
  Try
    If not fRebuildneeded then begin
     If DoSwap then begin
       //If swap copy is enabled then swap buffers
       // else need to rebuild
       If fpfd_Swap_Copy and fValidBuffer
         and not fDrawToOther then
         //some systems allow for a buffer swap others not
         // swapbuffers is the Windows implementation
         SwapBuffers(fRenderDC)
       else
         fRebuildNeeded:=True;
      end;
    end;
    If fRebuildNeeded then begin
      Clear3DCursor;    //Tidy up extra construction lines
      DoMoveTidyUp;
      if f3DCursorOn and not fGDIGeneric then
        Cursor := crnone;  //set up the cursors
      fRebuildNeeded := False;
      fValidBuffer := False;  //reset the flags
      If doswap then
        ClearScreen;  // clear the Open GL screen buffers
      SetUpViewingTransform;
      GetViewPortGrid(glGridType(fViewmode),20);
      // calculate the reference grid data
      SaveState(stDrawing);
      GLRender2DBackGround;
      RestoreState;
      // draw background, note save and restore of GL state
      glPushMatrix();
      SaveState(stAll);
      // set up the modelview transform for 3D drawing
      glListBase(0);
      // make sure Display list base is zero
      Case  fRenderMode of
       rmQuick : CallList(fGeneralLists+dlQuickRenderMode);
      else
        CallList(fGeneralLists+dlFullRenderMode);
      end;
      // set GL state to handle current render mode
      Do3DRenderScene;      // do the 3D rendering
      DrawSelectedPoints;   //draw points selected via tools
      DrawSimpleAxis;
      RestoreState;
      glPopMatrix();        // tidy up after 3D render
      SaveState(stDrawing);
      GLRender2DForeGround;
      RestoreState;    // Call 2D paper space render routine
      SaveState(stDrawing);
      // do HUD draw before the glflush swapbuffer
      DrawHUDDisplay;
      RestoreState;
      glFlush;              // Flush the OpenGL Pipeline
      If DoSwap then begin
        If not fDrawToOther then begin
          If SwapBuffers(fRenderDC) and fGDIGeneric
            and fpfd_Swap_Copy then
            fValidBuffer:=True;
        end;
      end;
      // when all rendering to Back Buffer fiished need to
      // swap rendered scene into front buffer
    end;
  Finally
    if f3DCursorOn and not fGDIGeneric then
      Cursor:=crdefault;
    fCanvas.UnLock;       // unlock the canvas
  end;
end;
```

➤ *Listing 3*

which can be used to display relevant data (such as frame rate) over the scene. By switching the demo's move mode to rotate and clicking and dragging you can see the application of the HUD and GDI. Notice also that the window will respond as normal to mouse messages.

To implement the rendering steps I surfaced individual calls through appropriate event hooks and coded the drawing onto the owning form (or the user could create a descendant of the class and override the specific calls). The events have custom event procedures defined in the header of the `GLWin` unit.

An important thing to remember is that the four rendering steps use different coordinate systems. Firstly, the OpenGL 2D background and 2D foreground use pixel coordinates, which are the same as the `Canvas`. However, OpenGL defaults to the origin being the lower left corner of the window with positive Y *up* the screen. The screen units are OpenGL units (type `double`) which will be rounded to effective pixel coordinates. The `GLCanvas` has been set up to handle draws to both 2D and 3D OpenGL modes.

Secondly, the 3D rendering uses a cartesian coordinate system comprising X, Y and Z (`double`) values encapsulated in a `TGLPoint` record type. X corresponds to the East direction, Y corresponds to the North direction while Z is the Up direction. How these appear on the screen will be controlled by the viewer position. A simple axis can be turned on in the demo which will display the current North, East and Up directions.

Thirdly, the `Canvas` will use the standard windows coordinate system with the origin at the top left corner and the positive Y direction down the screen.

In the `GLFuncs` unit you will find a simple class (`TLinkPoint`) to help handle the three modes.

## OpenGL Primitives

The basic 3D constructs in OpenGL are the 3D point, 3D line, 3D triangle (sometimes called a polygon), bitmap and texture. Everything OpenGL does in 3D rendering is based on multiples or combinations of these. The same type of structure will apply for all primitives, so I will only discuss the 3D line (implementations of the other primitives are in the `TGLCanvas`).

Listing 4 shows the code to draw a 3D line into an OpenGL session. Three basic OpenGL functions are called to draw the line. The `glBegin(GL_LINES)` call informs the OpenGL session that a sequence of OpenGL drawing functions are to follow which form the basic 3D line primitive. Although I have only included the vertex points of the line in the `glVertex3d(X,Y,Z)` command, I could have specified the colour of each vertex and could even have included a normal for each vertex. A feature which the GDI doesn't offer is the ability to set the colour of each vertex of a line. OpenGL will smoothly change the colour along the line. The `GL_LINES` sequence is ended by the `glEnd` function. As the name suggests, I could also specify a large number of lines enclosed within one `glBegin..glEnd` sequence.

This is all very well if you want to get down to the API and really play some tunes. However, to draw a basic line in the GDI I simply call `TCanvas`' `MoveTo` and `LineTo` command sequence. This is how I have implemented the basic 3D point, 3D line and 3D triangle drawing in the `TGLCanvas`. A call to the `TGLCanvas` with a `MoveTo` and `LineTo` will produce a 3D line from the `MoveTo` location in 3D to the `LineTo` location (see Listing 4). There is a little more housekeeping done in the `TGLCanvas` class and so I would recommend using it where possible, or at least studying the code. This is important when creating a metafile (as I discuss later in the article).

I have tried to model the TGLCanvas on the TCanvas, with the exception that you don't need to deal with pens, brushes or fonts. OpenGL does not use these concepts. The OpenGL session is a state machine which will retain the current state until the value is changed. Most state values can be queried and changed as required. Also the current state can be saved and later restored. To draw with a green colour I would set the state using, for example, the function glColor4fv(@glGreen). Any further drawing would be in this colour until the colour is changed. I have allowed the user to set the drawing colour, the line style and the line width as properties of the TGLCanvas. This class will then manage the setting of the appropriate values when drawing a line.

A point to note is to always consider the maximum or minimum values of states within the OpenGL session. If you try to set a state to outside the valid range, the session will always 'clamp' the values for you, but this can be a little disconcerting. For example, in some OpenGL implementations the maximum size of the viewport may be restricted (my Diamond FireGL 3000 allows a maximum viewport of 4096 by 4096 pixels).

## Bitmaps And Textures

I won't spend much time on this subject. Bitmaps and textures can provide detail to the scene being rendered. They can also be used to optimise the rendering process. Replacing a complex set of polygons with a bitmap or texture can speed scene processing while still producing an impressive image.

The OpenGL graphic pipeline is all about converting a set of 3D constructs to a screen image. In essence this is achieved by projecting the 3D constructs onto the screen and calculating which of the screen pixels the construct image will fall on. This calculation includes, among other things, deciding if the construct can be seen in this window, the colour and even which construct is in front of another. At the end of the process is a display buffer containing the final screen pixels. The folk at SGI have provided direct access into this buffer for both reading and writing.

Thus it is possible to add a bitmap directly into this buffer, effectively drawing a bitmap onto the screen. Textures are a little more complex as the texture can be attached to a polygon and thus wrapped or warped depending on the view of the construct at the time.

I have not included any bitmap drawing in the demonstration code. However, I do read a bitmap from the screen buffer when I create a clipboard bitmap image and the basic concepts are the same.

The procedure getBitMapImage will return a TBitmap filled with the current scene read directly from the screen buffer. The things which are important here are the basic OpenGL settings to be able to unpack the image and the fact that the RGB values are stored in the opposite order for a bitmap and need to be reversed. The bitmap data is read into a temporary array for the RGB swap then a bitmap compatible stream is created.

## Text

As I mentioned, OpenGL does not directly handle Windows fonts. The reason for this (and many other differences) stems from the requirement for the OpenGL API to be portable between operating systems and platforms.

Another consideration is that generally text is a 2D construct (that is, it has no depth) which could be problematic to manage in a 3D environment. However, Microsoft have provided two different mechanisms for constructing text within the OpenGL session.

The first provides 3D text constructs which are either a series of polygons or a series of lines constructed from an existing TrueType font. The polygons/lines include a depth component and thus are 3D. In the demonstration program (Figure 1) the *OpenGL* text is constructed using this technique. If you rotate the view then you will see the side and back of the text string.

The font is based on Arial. As with the 3D line the colour is the current colour, the shading is the current shading setting, etc.

Behind the scenes a Microsoft OpenGL function wglUseFontOutlines constructs a set of display lists (display lists allow the pre-building of 3D objects thus saving time when recalculating a scene) and a set of glyph metrics. To display a text string the OpenGL function:

```
glCallLists(length(aST),
  GL_Unsigned_Byte,@aST[1]);
```

is used, with aST being the text string.

Of course there is a little housekeeping involved. As with the 3D line I have provided a TGLCanvas method of drawing text. The TextOut3D takes care of this. This call currently takes a 3D position as the start point for the text and a scale value for the size in 3D units.

➤ *Listing 4*

```
Procedure TOpenGLCanvas.LineTo(aPt:tGLPoint);
Begin
  glLineWidth(fLineWidth);  //set line width (stored by the GLCanvas)
  glColor4fv(@fColor);      //set line colour  (stored by the GLCanvas)
  //use the glPassthrough to signal a line width when creating a metafile
  glPassThrough(1000+fLinewidth);
  glBegin(GL_Lines);        //call the OpenGL Start line
  If f3DMode then begin
    //uses current point as start point of line (stored by the GLCanvas)
    glVertex3dv(@fCurrentPoint);
    glVertex3dv(@aPt);      //pass pointers to the 3D points data
    //passing pointers to data is faster than passing the data!!!!
  end else begin
    glVertex2dv(@fCurrentPoint);
    glVertex2dv(@aPt);      //pass through pointers signalling 2D data
  end;
  glEnd;                    //close the OpenGL begin
  MoveTo(aPt);  // set the glCanvas current point to the end point of the line
end;
```

It should also take some rotation information as the text string can be rotated into any position. Currently the text is drawn lying in the XY plane and parallel to the X axis.

The second method of providing text within the OpenGL session is to use text bitmaps. Using the `wglUseFontBitmaps` call a display list of text bitmaps can be created. This text will be a 2D text and will be always drawn in the plane of the screen. I have implemented this text in the text values that can be toggled on and off with the reference grid. Again I have provided the functionality through a `TGLCanvas` call `TextOut2D`.

The downsides of using OpenGL text include the memory it takes to store a set of complex polygons or lines for the 3D font, the memory needed for the font bitmaps and the time it takes to recalculate the data should you want to reset the font style. I have implemented a one-at-a-time text font so that I can provide some text yet keep the overhead to a minimum. There isn't the font flexibility you get with the GDI.

However, don't forget that should you choose to write text with the normal GDI then all the font and text capability is available at this 'painting' level.

## Saving Rendered Images

So far we have focused attention on producing an image on the

➤ *Listing 5*

```
Procedure TCustomOpenGLWindow.getBitMapImage(aBP:tBitMap);
var
  BitsMem     : pointer;
  BmInfo      : tBitmapInfo;
  bitsize, WinWidth, WinHeight, scanWidth, T1,T2 : DWord;
  aRGB        : pGLRGB;
  temp        : GLUByte;
  tDC         : HDC;
  TempBitMap  : HBitMap;
  aMem        : TMemoryStream;
  Info        : TBitmapFileHeader;
  InfoSize    : DWord;
  InfoHeader  : TBitMapInfoHeader;
  Procedure SwapTheRGBValues;
  Var
    iVal,jVal : DWord;
  Begin
    //swap bytes as RGB values are in reverse order
    T1:=LongInt( Bitsmem);
    For ival:=0 to WinHeight-1 do begin
      T2:=T1+(ival*ScanWidth);
      aRGB:=pGLRGB(ptr(T2));
      For jval:=0 to WinWidth-1 do begin
        If aRGB^[1]<>aRGB^[3] then begin
          //only swap if the values are different
          Temp:=aRGB^[1];
          aRGB^[1]:=aRGB^[3];
          aRGB^[3]:=Temp;
        end;
        t2:=t2+3;  // move to the next set
        aRGB:=pGLRGB(ptr(T2));
      end;
    end;
  end;
  Begin {getBitMapImage}
    //quit if not valid to build
    If not assigned(aBP) then
      exit;
    If (fRenderDC=0) or (fHRC=0) then
      exit;
    // ensure the GL session is enabled
    If not enableGL then
      exit;
    //set up the BMF info and data structures
    FillChar(BmInfo,SizeOf(BmInfo),0);
    WinWidth:=  fviewport[3]; //width of current GL screen
    WinHeight:= fviewport[4]; //height of current GL screen
    ScanWidth:=(WinWidth)*3; // scan width for the bitmap
    //need to fix alignment to 4 byte
    ScanWidth:=(ScanWidth+3) and $FFFFFFFC;
    //calculate the memory size needed for the bitmap
    BitSize:=ScanWidth*(WinHeight);
    glFinish;  // flush the GDI pipeline
    // set up the gl  read
    If not fDrawToOther then
      glReadBuffer(GL_Back)
    else
      glReadBuffer(GL_Front);
    glPixelStorei(GL_PACK_ALIGNMENT,4);
    glPixelStorei(GL_PACK_ROW_LENGTH,0);
    glPixelStorei(GL_PACK_SKIP_ROWS,0);
    glPixelStorei(GL_PACK_SKIP_PIXELS,0);
    Try
      // read the glpixels from the video buffer
      // Allocate memory to read pixels into
      GetMem(Bitsmem,bitsize);
    Except
      on EOutOfMemory do
        Bitsmem:=nil
      else
        Raise;
    end;
    If BitsMem<>Nil then begin
      // get the bits data
      glReadPixels(0,         //X
                   0,         //Y
                   WinWidth,  //Width
                   WinHeight, //Height
                   GL_RGB,    //Format of data read
                   GL_UNSIGNED_BYTE, //Type of data
                   Bitsmem);  // pointer to memory storage
      SwapTheRGBValues;
      // reverse the order of the RGB values
      TDC:=CreateDC('Display',nil,nil,nil);
      // attempt to create a DIB bitmap handle
      If TDC<>0 then begin
        With BmInfo.bmiheader do begin
          biSize:=SizeOf(TBitMapInfoHeader);
          biWidth:=WinWidth;      //width of the bitmap
          biHeight:=WinHeight;    //height of the bitmap
          biPlanes:=1;            //always 1
          biBitCount:=24;         //24 bit colour for bitmap
          biCompression:=BI_RGB;  //No compression
          biSizeImage:=BitSize;   //size of the image
          biXPelsPermeter:=2952;  //75dpi
          biYPelsPermeter:=2952;  //75dpi
          biClrUsed:=0;
          biClrImportant:=0;
        end;
        //set up the Bitmap info header
        TempBitMap:= CreateDIBitmap(tDC,BmInfo.bmiheader,
          cbm_Init, Bitsmem, bmInfo,DIB_RGB_COLORS)
      end else
        TempBitMap:=0;
      try
        If tempBitMap<>0 then begin
          //select bitmap into the DC
          SelectObject(TDC,TempBitMap);
          //assign the bitmap to the tBitmap handle
          aBP.Handle:=TempBitMap;
        end else begin
          //fail on the BID create handle then
          // manually build the bitmap
          FillChar(Info,SizeOf(Info),0);
          FillChar(InfoHeader,SizeOf(InfoHeader),0);
          With Info do Begin
            bfType:=$4D42;
            InFoSize:=SizeOf(InfoHeader);
            bfSize:=sizeOf(info)+ InfoSize+ bitsize;
            bfOffBits:=sizeOf(info)+ Infosize;
          end;
          With InfoHeader do Begin
            biSize:=SizeOf(InfoHeader);
            biWidth:=WinWidth;
            biHeight:=WinHeight;
            biPlanes:=1;
            biBitCount:=24;
            biCompression:=BI_RGB;
            biSizeImage:=BitSize;
            biXPelsPermeter:=2952;//75dpi
            biYPelsPermeter:=2952;//75dpi
            biClrUsed:=0;
            biClrImportant:=0;
          end;
          aMem:=TMemoryStream.Create;
          aMem.Write(Info,SizeOf(info)); // write info block
          //write the information header block
          aMem.Write(InfoHeader,SizeOf(InfoHeader));
          aMem.Write(BitsMem^,BitSize); //write pixels data
          aMem.Position:=0;  //reset the stream
          //load theimage into the tBitMap
          aBP.LoadFromStream(aMem);
          aMem.Free;  //tidy up
        end;
      Finally
        //Tidy up
        If TDC<>0 then
          DeleteDC(TDC);
        FreeMem(Bitsmem,bitsize);
      end;
    end;
    GetError;    // check for GLErrors
  end;
```

screen. Most Windows applications support the clipboard for transferring data or saving data. It would be sensible if our rendered image could be transferred to the clipboard. This logic will also provide the mechanism for saving the graphic image to disk.

To implement this type of action firstly the data types need to be selected. A graphic image can be copied to the clipboard using a bitmap or metafile format. I have implemented both as there are advantages and disadvantages to be considered.

A bitmap represents the pixel data and so is reasonably easy to read from the screen buffer using the supplied `glReadBuffer` routine. The downside is the potential size and detail of the image. Higher detail means the data set needs to be larger. Of course there are compression techniques which can be applied later, but initially we need to manage the raw pixels. Although an 800x600 image looks good on a screen it is small when copied to a printer with a resolution of 600dpi and will suffer badly if it is stretched.

I have implemented the creation of a bitmap from the current screen data in the `getBitMapImage` call. This will take a `TBitmap` and fill it with the current scene data (see Listing 5). There are three key aspects to this call. Firstly, set the OpenGL session variables for correct pixel reading. Next, read the pixels into a suitable buffer and then swap the red and blue values. Finally, copy the data into the supplied `TBitMap`.

Often an image comprises large areas of a single colour, or is sparsely populated with lines, as is the case of basic engineering drawings. This leads into the second method of image transfer, the metafile. In essence this comprises a list of drawing instructions. The problem here is that OpenGL drawing instructions mean nothing to a metafile or the device which will receive the instructions. We need a means of translating OpenGL drawing calls into standard GDI calls. The benefit of the metafile is size but the downside is that the GDI can't handle the subtle rendering achieved by OpenGL. This can mean a loss of detail in the final image. OpenGL bitmaps, such as bitmap text, need special handling.

I have managed the creation of a metafile in the `getMetaFileImage` call. As with the `getBitMapImage`, this call will return the metafile filled with the current image. There are three key aspects to this call. Firstly, set up the OpenGL state variables including the `RenderMode`. By setting the `RenderMode` to `GL_FEEDBACK` the OpenGL session will render to a buffer rather than the screen. The buffer is filled with the basic screen data from which GDI calls can be created. For example, a 3D Line will be projected onto the screen viewport. If any part of the line appears on the screen then the buffer will contain the location of the start and end of the line correctly clipped, if needed, to the screen boundary. Next, re-render the image to the feedback buffer rather than the screen. The

## References

*Computer Graphics*. Foley, James, and Andries van Dam et al. Pub: Addison-Wesley, 1990. An excellent text on the real basics of 3D graphics.

*OpenGL SuperBible.* Wright and Sweet. Pub: Waite Group Press, 1996. An excellent reference although all the demo code is in C.

*OpenGL Programming Guide: The Official Reference Document for OpenGL, Release 1.* Neider, Jackie, Davis, and Mason Woo. Pub: Addison-Wesley, 1993 (sometimes called the red book). A good general reference.

*OpenGL Reference Manual: The Official Reference Document for OpenGL Release 1.* OpenGL Architecture Review Board. Pub: Addison-Wesley, 1992 (sometimes called the blue book). The online help in Delphi 3 and 4 probably has the same information as this text.

## Information on the internet:

The official OpenGL site is www.opengl.org with details of current and proposed specifications. The OpenGL newsgroup (comp. graphics.api.opengl) is an excellent source of tips and tricks, with heaps of undocumented details. Members of the ARB often participate, providing real insight into the API.

current screen buffers (that is the screen image) will be unaffected by this re-rendering. Finally, convert the feedback buffer data to GDI calls. This is carried out in the utility call `DrawFeedBackDataToCanvas` at the bottom of Listing 6. An important part of this conversion is the `GL_PASS_THROUGH_TOKEN` which is a user-defined value passed through to the buffer. I use this to set some of the GDI values which the feedback buffer does not contain (eg line width). I have not yet implemented a bitmap nor the pixel functions into the GDI.

➤ *Below and facing : Listing 6*

```
Function TCustomOpenGLWindow.getMetaFileImage(aMF:tMetaFile;
  UseMFHeight:Integer; XScale,YScale: Double): Boolean;
var
  TempMFC         : TMetaFileCanvas;
  Buffer          : Pointer;
  fFeedBackData, Step : Integer;
  GotAllTheData   : Boolean;
  BufSize         : Longint;
  tHt             : Integer;
  oldCanvas       : TCanvas;
Begin
  Result:=False;
  If not assigned(aMF) then
    exit;
  Case  UseMFHeight of
    0 : tHt:=Height;
    1 : tHt:=aMF.Height;
    2 : tHt:=aMF.mmHeight;
  else
    tHt:=Height;
  end;
  TempMFC := TMetaFileCanvas.CreateWithComment(
    aMF, 0, 'OpenGL App','GL Scene');
  //create a metafile canvas
  GotAllTheData:=False;
  Step:=1;
  oldCanvas:=fCanvas;
  fCanvas:=TempMFC;  // swap in a temporary canvas
  GetViewPortGrid(glGridType(fViewmode),20);
  BufSize:=fbBufferSizetiny;
  Repeat
    If Step>1 then
      BufSize:=BufSize*2;
    GetMem(Buffer,BufSize*SizeOf(Single));
    glFeedbackBuffer(BufSize,GL_3D_COLOR,Buffer);
    // set the render to the feedback buffer
    glRenderMode(GL_FEEDBACK);
    fRebuildNeeded:=True;
    GLRenderWindow(False);    // render the window
    fFeedBackData:=glRenderMode(GL_RENDER);
    If (fFeedBackData>=0) then begin
      GotAllTheData:=True;
      DrawFeedBackDataToCanvas(TempMFC, fFeedbackdata,
        pFeedBackArray(Buffer), GL_3D_COLOR, tHt, nil,
        XScale,YScale);
    end;
    FreeMem(Buffer,BufSize*SizeOf(Single));
    Inc(step);
  until GotAllTheData or (Step=4);
  Result:=GotAllTheData;
  TempMFC.Free;
  fCanvas:=oldCanvas;
end;
Procedure DrawFeedBackDataToCanvas(aCanvas: TCanvas;
  aSize: Integer; FeedBack: pFeedBackArray;
  FeedBackType: Integer; aHeight: Integer;
  aBitMaps : tList;       //list of bitmaps to be drawn
  XScale,YScale:Double);  // point scale factor
Var
  Count,CVal:Integer;
  OldPenCol:TColor;
  OldBrushCol:TColor;
  CUserVal:Longint;
  T1,T2:TPoint;
  C1,C2:tColor;
  UnitVal, TenVal, HundredVal, ThousVal,
    CharVal, CharSize: Single;
Procedure ScalePoint(aPt:tPoint);
// scale point value based on supplied XScale,YScale
Begin
  aPt.X:=round(aPt.X*XScale);
  aPt.Y:=Round(aPt.Y*YScale);
end;
Procedure ReadPoint(Var aPt:TPoint;Var aCol:Tcolor);
Var
  R,G,B:Byte;
  R1,G1,B1:Single;
Begin
  aCol:=clBlack;
  aPt.X:=0;
  aPt.Y:=0;
  aPt.X:=Round(FeedBack^[Count]);
  Inc(Count); //jump the XVal
  aPt.Y:=aHeight-Round(FeedBack^[count]);
  Inc(Count); //jump the Y Val
  If FeedBackType>=GL_3D then begin
    inc(Count); // jump the Z val
    If FeedBackType>=GL_3D_COLOR then Begin
      R1:= FeedBack^[Count];
      inc(Count); // jump the R val
      G1:= FeedBack^[Count];
      inc(Count); // jump the G val
      B1:= FeedBack^[Count];
      inc(Count); // jump the B val
      R := Round(R1*255);
      G:=Round(G1*255);
      B:=Round(B1*255);
      // set to black if white
      If (R=255)and (G=255) and (B=255) then
        aCol:=clBlack
      else
        aCol:=PaletteRGB(R,G,B);
      inc(Count); // jump the A Level
      If FeedBackType>=GL_3D_COLOR_TEXTURE then Begin
        inc(Count); // jump the  val
        inc(Count); // jump the  val
        inc(Count); // jump the  val
        inc(Count); // jump the  val
      end; //GL_3D_COLOR_TEXTURE
    end;//GL_3D_COLOR
  end; //GL_3D
  ScalePoint(aPt);
end;
Procedure ExtractValues(aVal:LongInt);
Begin
  UnitVal:=0;
  TenVal:=0;
  HundredVal:=0;
  ThousVal:=0;
  CharVal:=0;
  CharSize:=0;
  If aVal=0 then
    exit;
  UnitVal:=Frac(aVal/10)*10;
  TenVal  :=Frac(aVal/100)*100-UnitVal;
  CharVal:=Frac(aVal/1000)*1000;
  HundredVal:=CharVal-TenVal-UnitVal;
  ThousVal:=Frac(aVal/10000)*10000-charVal;
  CharSize:=ThousVal;
end;
Procedure  MakePoint;
Begin
  Inc(Count);
  ReadPoint(T1,C1);
  With aCanvas do Begin
    Pen.Color:=C1;
    MoveTo(T1.X,T1.Y);
    LineTo(T1.X+1,T1.Y+1);
  end;
  CUserVal:=0;
end;
Procedure MakeLine;
Begin
  Inc(Count);
  ReadPoint(T1,C1);
  ReadPoint(T2,C2);
  With aCanvas do Begin
    Pen.Color:=C1;
    MoveTo(T1.X,T1.Y);
    LineTo(T2.X,T2.Y);
  end;
  CUserVal:=0;
end;
Procedure MakePolygon;
Var
  PolyC,j: Integer;
  PolyPts: Array of TPoint;
Begin
  Inc(Count);
  PolyC:= Round(FeedBack^[Count]);
  inc(Count);
  SetLength(PolyPts,PolyC);
  For j:=0 to PolyC-1 Do Begin
    ReadPoint(PolyPts[j],C1);
  end;
  With aCanvas do Begin
    Brush.Color :=C1;
    If CUserVal>0 then
      Brush.Style :=bsClear;
    Pen.Color   :=C1;
    PolyGon(PolyPts);
  end;
  Finalize(PolyPts);
  CUserVal:=0;
end;
Procedure MakeBitMap ;
Begin
  Inc(Count);
  ReadPoint(T1,C1);
  If assigned(aBitMaps) and (aBitMaps.Count>0) then
    With aCanvas do Begin
    Pen.Color:=C1;
    MoveTo(T1.X,T1.Y);
  end;
  CUserVal:=0;
end;
Procedure MakeDraw ;
Begin
  Inc(Count);
  CUserVal:=0;
end;
Procedure MakeCopy ;
Begin
  Inc(Count);
  CUserVal:=0;
end;
Begin
  If not Assigned(aCanvas) then
    exit;
  If aSize=0 then
    exit;
  Count:=0;
  OldPenCol  := aCanvas.Pen.Color;
  OldBrushCol:= aCanvas.Brush.Color;
  CUserVal:=0;
  Repeat
    CVal:=Round(FeedBack^[Count]);
    Case CVal of
      GL_PASS_THROUGH_TOKEN :
        Begin
          Inc(Count);
          CUserVal:=Round(FeedBack^[Count]);
          ExtractValues(CUserVal);
          Inc(Count);
        end ;
      GL_POINT_TOKEN       : MakePoint;
      GL_LINE_TOKEN        : MakeLine;
      GL_POLYGON_TOKEN     : MakePolyGon;
      GL_BITMAP_TOKEN      : MakeBitMap;
      GL_DRAW_PIXEL_TOKEN  : MakeDraw;
      GL_COPY_PIXEL_TOKEN  : MakeCopy;
      GL_LINE_RESET_TOKEN  :
        Begin
          Case CUserVal of
            1001..1009 : aCanvas.Pen.Width:=CUserVal-1000;
          else
            aCanvas.Pen.Width:=1;
          end;
          MakeLine;
        end;
    else
      inc(Count);
    end;
  until (Count>=aSize-2);
  aCanvas.Pen.Color   :=OldPenCol;
  aCanvas.Brush.Color :=OldBrushCol;
end;
```

Both a bitmap and a metafile format are achievable. At the moment my need is for engineering style scaled line drawings and the metafile is essential, with the bitmap providing the gee-wizz.

The copy-to-clipboard routines use the screen for the reference. I have also implemented a routine to provide a sized bitmap and a larger than screen metafile for scale drawing. The former requires the building of a special rendering context, to render to a bitmap rather than a window, and the latter requires the a large viewport.

## Conclusion

I hope I have lifted the hood a little on OpenGL for you. One of the criticisms of OpenGL is the difficulty of finding out about the API's capabilities. I tend to agree. However, I have not yet been disappointed in OpenGL's capability to meet my needs. With this article, the code presented, and some coaching from the references, I believe you can make a good start. If you need to work with 3D graphics, OpenGL merits serious consideration.

John Hutchings is a Senior Mining Engineer working on applying PC technology to mining. He has spent the last 5 years working with a small team building customised CAD based mining design applications. He is keen to have feedback from those interested in this area and can be contacted at jxh2@orica.com.au