Simple Scene Graph Management And 3D Rendering With OpenGL

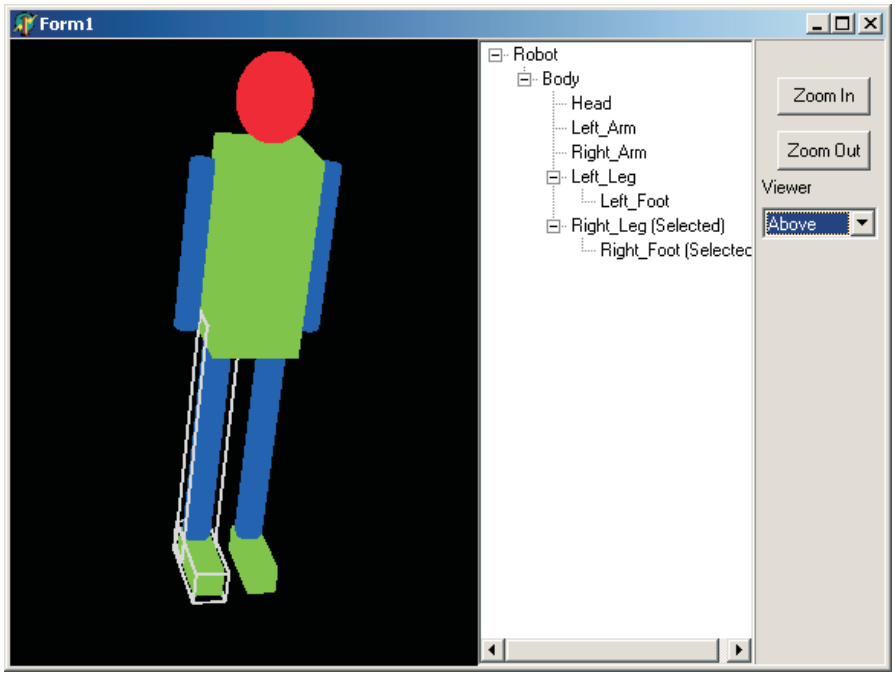
by George Black

Opportunities exist for Delphi developers to produce a huge range of applications based on 3D rendering technology. What follows is an explanation of scene graph management and the OpenGL rendering engine, an understanding of which is essential in order to create your own 3D rendering applications.

Today, computer generated 3D rendered photo-realistic images are all around us. From computer generated animation films, computer augmented feature films, advertisements and weather maps on the nightly news to pictures in magazines, posters and the internet. Complex information can be visualized and more readily understood using sophisticated volumetric rendering techniques. For example, doctors can view the MRI scans of patients to assist in diagnosis, scientists can educate and demonstrate complex concepts using smart 3D graphics, engineers can model complex problems without the cost of mockups or prototypes. A 3D computer game is not cool unless it sports realistic and fast 3D rendering. The core of all these applications is the scene graph management, the rendering engine and the hardware.

Scene graph management is a well developed and understood technology (see the references at the end of this article) which can be easily and effectively applied in the Delphi and object oriented programming environment. Microsoft provides the OpenGL and DirectX 3D rendering engines as standard with 32-bit Windows versions, that can be accessed directly from Delphi. These days a Windows system can easily have the 'grunt' to match what was, in the past, the realm of the graphics workstation. Plenty of graphics card manufactures will accelerate the performance of the OpenGL and DirectX rendering engines, via hardware, to startling levels. Delphi developers, therefore, now have the same opportunity to provide RAD for 3D rendering applications as with other more traditional areas. Kylix will also provide the same opportunities in the Linux environment.

➤ *Figure 1*



This article will demonstrate the basic principles of scene management using some simple 3D objects to create a robot-shaped figure.

A scene graph is composed of a hierarchy of 3D objects that is induced by a bottom-up construction process. Simple 3D objects are used as building blocks to create higher-level 3D objects, with those creating higher-level objects, and so on. If each 3D object is used only once, then the scene graph can be constructed as a tree, with the 3D objects as nodes and the inclusion relationships as the connections. The structure is a directed acyclic graph. The scene graph structure is used directly for the rendering of the 3D objects. Using appropriate techniques, efficient scene graph traversal during the rendering process will optimize the speed of rendering while keeping the structure of the 3D objects simple.

Efficient scene management also requires the efficient culling of 3D objects not currently in view and the selection of 3D objects using the traditional selection techniques with the mouse cursor. Techniques for achieving 3D object view culling and user selection of 3D objects are discussed in this article along with demonstrations.

Also included are the basics for creating a rendering window using the OpenGL API to display the 3D scene graph. Working sample code, based on Delphi 5, to create a basic OpenGL window control along with a demonstration scene graph is included and discussed throughout the article. Refer to Figure 1 for a screenshot of the demonstration application.

It is assumed the reader has some understanding of 3D graphics. Excellent texts exist for an introduction of this topic (see *References* for examples). This article aims to provide the reader with information to enable 3D rendering and scene graph management to be performed from within the Delphi environment.

OpenGL Rendering Basics

Page 2 of the *OpenGL Programming Guide* says 'OpenGL is a software interface to the graphics hardware. This interface consists of about 250 distinct commands that you use to specify the 3D objects and operations needed to produce interactive three-dimensional applications.' An excellent reference to the OpenGL command set and the implementation of OpenGL in the Microsoft Windows environment is the *OpenGL Super Bible* (see *References*).

The OpenGL API is command-driven rather than COM-driven (like DirectX 3D). The core OpenGL functionality is provided by Windows within OpenGL32.DLL. A Delphi prototypes unit was supplied with Delphi 3. However, an improved and updated prototypes unit, OpenGL12.PAS, is at

```
www.delphi-jedi.org/
 DelphiGraphics/jedi-index.htm
```

and is included on this month's disk. As the OpenGL engine is platform-independent, the vendor (Microsoft in our case) must supply interface code into the specific 'windowing' system used. Microsoft has supplied these commands and the prototypes can be found in the Windows.PAS unit.

OpenGL commands are prefaced with `gl`. OpenGL utility commands are prefaced with `glu`. Windows-specific implementation commands are usually prefaced with `wgl`.

The Demonstration Code

Included with this article is a simple demonstration of the topics discussed. Two classes have been created which will do the work. The first is called `TBasicOpenGL` and is a descendant of the `TWinControl`. This class will take care of the OpenGL session details and will be the viewing window for the OpenGL session.

The second class is called `TOpenGLObject` and descends from a `TObject`. This class is the core to the scene graph implementation and will also be responsible for the final rendering of the 3D object into the OpenGL window.

Both classes and some support definitions are found in the BasicOpenGL.pas unit.

Set Up The OpenGL Window

An OpenGL session requires a Windows handle and device context to associate with. `TBasicOpenGL` is a new class descended from the `TWinControl`, as the handle is made available within this class. This class also implements the basic mouse feedback that will be important in the user interaction with the objects.

The implementation of the OpenGL session through the `TBasicOpenGL` class has been kept simple for readability: the supplied code would not be suitable for a working application.

To initialize an OpenGL session a number of steps need to be carried out. Once the OpenGL session is running and active, then any OpenGL commands will be directed to this session and carried out. The majority of this is in the `GLStartUp` call and in the specific call `CreateRenderingContext` found in the OpenGL12.PAS unit.

The sequence of creation is as follows:

Create a window handle. The `TWinControl` takes care of this. The `HandleAllocated` and `HandleRequired` methods provide the standard mechanism for ensuring a handle is available when required.

Create a device context. A Windows device context is created and held for the life of the window. The `GetDC` Windows API call provides the device context and is called within the `GLStartup` method.

Set the pixel format for the window. A valid pixel format is selected which will support the OpenGL session requested. Generally a set of values are requested and the pixel format which best fits the request will be supplied. If hardware acceleration is present, this will be selected in favor of the software only mode. If you want to know more about pixel formats, refer to Microsoft's documentation and the *OpenGL Super Bible*. You can only set the pixel format *once* for any given window handle.

Create a rendering context. The rendering context is a value of pre-defined type `HGLRC` that will associate the OpenGL session data with a device context and pixel format. Only one rendering context can be active per thread. A rendering context can be associated with a number of device contexts as long as each window format, pixel format and device context are exactly the same.

Activate the rendering context. Once activated, the rendering context will receive all OpenGL commands. Hence the reason for one rendering context active per thread. OpenGL does support multi threading. Each thread can have one active rendering context. We must initialize the state of the OpenGL session associated with the rendering context. The OpenGL implementation is a state machine, which will maintain the current state until changed.

To initialize and manage the OpenGL session in the `TBasicOpenGL` six of the `TWinControl's` calls are overridden. Refer to the source code for listings of these modifications.

Rendering A Window

The functionality to render in the window requires some startup and shut-down. Some of this can be done once-per-session, or only when the window is resized or moved, but for readability of the demonstration code this functionality is included in the `RenderWindow` call. Refer to Listing 1 for a listing of the code.

Rendering of the window will be triggered by the `WMPaint` message. All the standard `TWinControl` paint functionality is ignored with the `RenderWindow` method doing all the

work including clearing the window. The rendering process will call the `DoRender` of the Root Object which is the root of the scene graph, if one has been assigned.

The setup of the render requires the initialization of the three transformation matrices involved in data transfer through the OpenGL graphics pipeline. The matrices are the `ModelView` matrix, the `Projection` matrix and the `ViewPort` matrix. Transformation matrices provide the mechanism for projection of 3D points onto a 2D screen or window. 3D objects can be made to appear to rotate, move and scale simply by adjusting the appropriate matrix. The `ModelView` is primary to this function and converts 3D coordinates into eye relative coordinates. This matrix is usually the default for the OpenGL session and will be manipulated often by every 3D object being rendered. The `Projection` matrix establishes the viewing volume against which 3D objects will be clipped and sized. The `Projection` matrix will not change unless the view volume is altered, for example, the viewer wants to zoom in. The `ViewPort` scales the final output into window coordinates. The `ViewPort` matrix will only change if the window is resized or moved.

Retained Mode (Scene Graphs) And Immediate Mode

There are two approaches to programming for 3D graphics. The first is called *retained mode*.

With *retained mode* you provide the API with a description of your objects and the scene and the API will create the screen image for you.

The second approach is called *immediate mode*. With *immediate mode* you will issue low level rendering commands to produce the screen image. These modes could be likened to working with Delphi. Dropping buttons onto a form is similar to *retained mode* action where drawing lines on a Tcanvas is *immediate mode*. Most *retained mode* APIs use an *immediate mode* internally to finally create the image.

```
procedure TBasicOpenGL.RenderWindow;
begin
  ClearWindow;                    //clear the window and fill with black pixels
  If Height=0 then exit;          //exit if too small
  glViewport(0,0,width,height);   //set the viewport size
  glMatrixMode(GL_PROJECTION);    //switch to Projection Matrix
  glLoadIdentity;                 //fill matrix with unity matrix
  gluperspective(fViewAngle,width/height,1,500);    //set perspective view
  glMatrixMode(GL_MODELVIEW);     //switch to ModelView matrix
  glLoadIdentity;                 //fill matrix with unity matrix
  glTranslatef(-9,-5,-50);        //move the viewer to sensible view point
  If fRootObject<>Nil then
     fRootObject.DoRender;        //allow the Root Object to render items
  glFlush;                        //flush the GL pipeline
  SwapBuffers(fRenderDC);         //swap the buffers to make it appear
end;
```

➤ *Listing 1: The RenderWindow method.*

OpenGL is an *immediate mode* API. As there is no supplied *retained mode* structure with OpenGL we will need to build one. 'Wait!', I hear you cry. 'Doesn't DirectX3D have a *retained* and *immediate mode*?' Yes, but not as defined above. Microsoft and Silicon Graphics (the creators of OpenGL) were working on a joint *retained mode* API (Fahrenheit Scene Graph) but I believe this project has folded.

Retained mode or scene graphs, as I will now call the approach, provide the essential mechanism for managing our 3D data in an orderly fashion, provides the framework for user interaction and, possibly more importantly, the potential for performance optimization.

Scene Graph Structure

As mentioned earlier, a scene graph can be constructed as a tree with the objects as nodes. It seemed logical that the `TTreeNode` should be used as the basic building block for the scene graph and that a new class should descend from this with the added functionality. However, the `TTreeNodes` class does not support the extension of the `TTreeNode` class as you can add only `TTreeNode` instances to the `TTreeNodes` structure. A `TTreeView` has been linked in to display the scene graph tree structure only.

The basic building block for a scene graph is the one-to-many management of the parent and children objects. A `TList` is used to hold the children while a `ParentObject` pointer holds the pointer back to the parent for each child. The 'root object' will have a `nil` parent. It is assumed that each parent owns the children and will be responsible for freeing the instances.

`ChildCount` and `Child[index]` properties can be used to access the child fields of a node. Recursive calls have been used to implement most of the functionality.

For example, when a 3D object is selected (or de-selected) all children will also be selected (or de-selected). The `Selected` property is implemented with a write method called `SetSelected` (refer to Listing 2 for the code listing). The method will call all children and set their values via their `Selected` property and will do so recursively all the way down the tree.

The demonstration program builds a 3D object hierarchy comprising a root object, which is shaped after a robot. The robot is composed of a body, head, arms, legs and feet. Simple shapes are used for the construction and include solid cubes (green), cylinders (blue) and spheres (red). All

➤ *Listing 2: The SetSelected method.*

```
procedure TOpenGLObject.SetSelected(const Value: Boolean);
  Var i:Integer;
begin
  //selection is passed down the tree
  FSelected := Value;
  If ChildCount>0 then
     for i:=0 to ChildCount-1 do
       Child[i].Selected:= FSelected;
end;
```

these shapes are part of the supplied Delphi OpenGL Utility Library (refer to dglut.pas in the supplied source code). The head, arms, legs and feet are built as children of the body. The feet are built as children of the legs. The ownership concept means that if the leg is selected then so is the attached foot. However, if the foot is selected, the owning leg will not be selected.

The ownership concept flows onto 3D object action. If the leg were moved when it were selected then the leg would also move the foot. The inherent structure of the objects and the rendering process means that this will all happen automatically.

The parts of the robot can be user-selected via the screen view or the tree view, using the mouse.

The key method for the 3D objects is the rendering method in which each 3D object will render itself and, if required, pass the method along to the children.

The `DoRender` method shown in Listing 3 follows these steps:

Firstly, if the 3D object is flagged as 'culled' then exit. Next, save the current `ModelView` matrix as this must be restored when finished. OpenGL provides a matrix stack and the calls to push and pop matrices. When pushing onto the stack a copy of the current matrix is left behind. Then carry out any 3D object-related translations, rotations and scaling. The 3D objects are only translated in this demonstration. The sphere 3D objects are translated along the X axis. The cube 3D objects will be translated by their owning sphere along the X axis and will translate them selves along the Y axis (in OpenGL the Y axis is always up the screen). Matrix manipulations are the recommended way of modifying the position, size and rotation of 3D objects within the scene graph. Next, push the name (instance pointer in this case) onto the OpenGL name stack. The stack is used during 3D object culling and selection and will be discussed a little later. Now render the 3D object using the `DoRender` method. 3D objects can be modified via polymorphism to create new basic units. If selected, draw a selection bounding box to indicate the object's selection state. Pop the name back off the name stack. Pass the `DoRender` call to the children. Finally, restore the `ModelView` matrix.

View Culling The Scene

View culling is the technique to optimize the rendering process by only rendering 3D objects that appear in the current view. If the 3D object is behind the viewer or out of the field of view then there is no point in carrying out the rendering operations that are costly in CPU cycles and which will eventually discard the 3D object's primitives.

A number of methods exist for the predictive culling of 3D objects and the detail can become very complex (refer to *Computer Graphics: Principles and Practice*, in the *References* section). However, I have found that I can use the OpenGL engine directly to tag 3D objects that do not appear in the view. Please note this is not the preferred method of scene graph culling, but does serve to demonstrate the principles for this article.

The culling is only required if the view changes or the 3D objects change in shape or position. Thus the culling routine can be triggered post one of these modifications. Then whenever the scene is rendered it will do so with only those 3D objects within the view. This method of view culling does require one rendering cycle, albeit that the complete OpenGL pipeline is not used, and so should be used only as required.

The key to view culling (and selecting 3D objects) is the capability of the OpenGL engine to be set to a selection state. In this state, OpenGL will not directly render to the window buffers but will instead create lists of hit records representing 3D objects which have identified themselves during the rendering and produce valid and seen rendering primitives.

The OpenGL engine will return a list of hit records. The structure is basically an array of integers in which 'n' hits have been recorded. Each hit is composed of the following:

➤ *Listing 3: The DoRender method of the TOpenGL Object class.*

```
procedure TOpenGLObject.DoRender;
  Var i:Integer;
begin
  If FViewCulled then exit;     // no need to render further
  glPushMatrix;                 //copy current matrix
  glTranslated(fTranslation.X,fTranslation.Y, fTranslation.Z);
  glRotated(fRotation.X,1,0,0);
  glRotated(fRotation.Y,0,1,0);
  glRotated(fRotation.Z,0,0,1);
  glPushMatrix;                 //copy current matrix
  glScaled(fScale.X,fScale.Y,fScale.Z);//only scale the local object
  glPushName(Integer(self));    //push Self as Object name/locator
  Case fMode of
    1: Begin
         glColor3fv(@glRed);
         glutSolidSphere(0.5,20,20);  //render as a sphere
       end;
    2: Begin
         glColor3fv(@glGreen);
         glutSolidCube(1);          //render as a cube
       end;
    3: Begin
         glColor3fv(@glBlue);
         glutSolidCylinder(0.5,1,10,10);   //render as a cylinder
       end;
  end; //case
  if FSelected then begin
    // if the object tagged as selected then draw a bounding box
    glColor3fv(@glGray);
    glutWireCube(1.2);
  end;
  glPopName;  //pop name back of name stack
  glPopMatrix;  //restore the original matrix
  If FChildrenList.Count>0 then begin
    //if child count>0 then render the children
    For i:=0 to FChildrenList.count-1 do
      Child[i].DoRender;  //recursive call down the tree
  end;
  glPopMatrix;  //restore the original matrix  end;
  glPopMatrix;  //restore the original matrix
end;
```

- The number of names on the stack at the time of the hit. In this example it will always be 1.
- The Minimum Z depth of the 3D objects in the hit. In this example this value is ignored but could be used to sort objects in distance from the viewer.
- The Maximum Z depth of the 3D objects in the hit. In this example this value is ignored.
- The name/s of the 3D objects. In this example there will only be one name, which will be the pointer to the instance that created the hit.

3D object culling (and 3D object selection) is carried out in the `GetSelectList` method, which is shown in Listing 4. To handle both view culling and 3D object selection, pass in a `SelectMode` to separate the approaches. Selection of 3D objects is covered in the next section.

The `GetSelectList` method shown in Listing 4 follows the following logic:

Firstly, provide the OpenGL engine with a suitably sized buffer to store the hit data. Next, set the OpenGL render state to `GL_SELECT`, initialize the OpenGL name stack and push a dummy zero name. Set all 3D objects `cull =False` so they will then render into the scene. Now set the OpenGL engine to `GL_RENDER` state and OpenGL will return the number of hits and the hit data. Set all 3D objects to `cull =True` to cull the complete scene. Finally, process the hit data, utilising the 'name' which is the pointer to the 3D object to set the cull flag to individually set the cull flag to false and turn on only those objects seen.

The current `ModelView`, `Projection` and `ViewPort` matrices, representing the current view, are used for this approach, and thus any 3D object which appears in the view will generate a hit record.

OpenGL provides a name stack for identifying 3D objects. The name is user supplied and of type Integer. Use the pointer to the 3D object instance as the name, as this provides an easy path back to the 3D object itself. OpenGL also provides a name stack onto which the full 3D object hierarchy can be pushed. As this is cumbersome, this demonstration will always push the object name, render the 3D object and then pop the name again before, rendering any owned children. The hits records will then contain records of the individual 3D objects that are seen (or selected).

The demonstration program highlights this technique. When run, all the robot's parts will be displayed in the view. If the view is zoomed in, using the `Zoom In` button, those parts that fall outside the view will gradually be culled. The tree view will also be updated to show those parts currently not in the view. Zooming back out, using the `Zoom Out` button, will bring the parts back into the rendering hierarchy. As the 3D object count is small, it is unlikely that you will notice any improvement in rendering time, but as the number of 3D objects grows this efficiency improvement will become observable.

Listing 4: GetSelectList method.

```
procedure TBasicOpenGL.GetSelectList(X, Y: Integer;
  SelectMode: TSelectMode);
  Var
    SelectArray  : Array[0..256] of TGLuint;
    Hits         : Integer;
    fviewport    : TVector4i;
    Procedure  ProcessHits;
      Var
        I,ArrayCount:Integer;
        aPtr:Pointer;
      Procedure ProcessHit;
        Var HitCount,I:Integer;
        Begin
          HitCount:=SelectArray[ArrayCount];
          Inc(ArrayCount);
          If HitCount=0 then exit;
          Inc(ArrayCount);//Z value min
          Inc(ArrayCount);//Z Value Max
          For i:=1 to HitCount do begin
            aPtr:=Ptr(SelectArray[ArrayCount]);
            Inc(ArrayCount);
            If (aPtr<>Nil) and
               (TObject(aPtr) is TOpenGLObject) then
              Case SelectMode of
                tsMouseSelect: TOpenGLObject(
                  aPtr).Selected:=True;
                tsWindowCull : TOpenGLObject(
                  aPtr).ViewCulled:=False;
              end;//case
          end;//hitcount loop
        end;
      Begin
        ArrayCount:=0;
        For i:=1 to Hits do
          //step through the hits and process records
          ProcessHit;
      end;
begin
  If height<=0 then exit;
  Hits:=0 ;
  //set up the hit record data array
  FillChar (SelectArray[0],SizeOf(SelectArray),0);
  glSelectBuffer(256,@SelectArray[0]);
  //set to GLSelect mode
  glRenderMode(GL_Select);
  //if selecting under mouse build special Projection Matrix
  If (SelectMode= tsMouseSelect) then begin
  //setup the size of the current viewport
    fviewport[0]:=0;
    fviewport[1]:=0;
    fviewport[2]:=Width;
    fviewport[3]:=Height;
    //set up the projection matrix
    glMatrixMode(GL_PROJECTION);
    glPushMatrix;
    glLoadIdentity;
    gluPickMatrix(X, Height-Y, 20, 20, fViewPort);
    gluperspective(fViewAngle,width/height,1,500)
  end;
  //initialize glNaming functionality
  glInitNames;
  glLoadName(0);
  //reset to ModelView
  glMatrixMode(GL_MODELVIEW);
  //If culling then cull ALL objects
  If SelectMode=tsWindowCull then
    If fRootObject<>Nil then
      fRootObject.CullAllObjects(false);
  //render scene into select pipeline
  If fRootObject<>Nil then
    fRootObject.DoRender;
  //get the hits data
  Hits:= glRenderMode(GL_Render);
  //tidy up
  If (SelectMode= tsMouseSelect) then begin
    glMatrixMode(GL_PROJECTION);
    glPopMatrix;
    glMatrixMode(GL_MODELVIEW);
  end;
  If SelectMode=tsMouseSelect then
    If FRootObject<>Nil then
      FRootObject.Selected:=False;
  //If found objects then process hits
  If Hits>0 then begin
    If SelectMode=tsWindowCull then
      If fRootObject<>Nil then
        fRootObject.CullAllObjects(True);
    ProcessHits;
  end;
  //update the tree view if required
  If Assigned(FOnUpdateTreeView) then
    FOnUpdateTreeView;
  //invalidate the view to trgger a repaint.
  Invalidate;
end;
```

Selecting 3D Objects

The selection of 3D objects currently under the cursor follows the same logic as the scene culling. However, to find only those 3D objects that are under the cursor, we must modify the `Projection` matrix to represent a small view space under the cursor. OpenGL utility library provides a method for setting this up given the current mouse position as supplied by the `MouseUp` event and the logic is as follows:

Firstly, save the current projection matrix. Next, load the identity matrix (a do-nothing matrix). Using the current mouse position, required size and window dimensions, create the pick matrix using the `gluPickMatrix` call. In the demonstration the mouse pick is set at 10 pixels by 10 pixels around the hot spot of the cursor. Finally, load the perspective settings.

By reducing the size of the viewing area to a frustrum (or truncated prism) sitting under the mouse, the selection system will now return a list of 3D objects currently under the mouse. Simply set the selected flag for those found. This selection process is relatively fast, and can be done for mouse moves thus allowing for the identification of which 3D object the mouse is currently rolling over. Obviously objects culled out of the view need not be considered for selection by the mouse.

The hit record data generated during the `GetSelectedList` method also contains information about how far the 3D object is away from the viewer, the Z depth. This can be useful in deciding which 3D object is closer to the user and thus more likely the one the user wishes to select.

Previous Articles

The OpenGL rendering API has been covered in *The Delphi Magazine* in the past. John Hutchings (Issue 42, February 1999) wrote an article covering some of the basics of establishing an OpenGL session and basic rendering at the *immediate mode* level. He outlined a set of basic components for interacting with OpenGL API. Ian Ringrose and Joseph Steel (Issue 34, April 1998) describe a *retained mode* (scene graph) component set that uses the VRML/Open Inventor style scene structure. They provide no detail of the scene graph implementation in the article. An earlier article by Dave Jewell (Issue 28, December 1997) was an introduction to the API. Interestingly his views of the future of OpenGL versus DirectX3D were partly correct. Hardware has developed in both processor speed and graphics capability to lower the cost of achieving acceptable 3D rendering performance of the 3D APIs. However, his prediction of a single API winner has not proved correct. Both APIs are still with us and strongly supported by graphics card and game manufacturers. Microsoft appears to be reluctantly supporting OpenGL (surprise, surprise) and has yet to move to the current OpenGL standard release of 1.2. With the growing interest in Linux and now having Kylix with us, there may be a strengthening in interest in OpenGL because of the cross-platform capability of the API.

Conclusion

Delphi developers have all the tools to build 3D rendering applications. OpenGL and DirectX hardware support is fast becoming the norm because of the extensive game support. The basics of a scene graph will allow fast and flexible management of 3D objects with which to build and maintain 3D capable applications.

I have been asked 'so what can I do with all this?' I have spent a number of years producing applications that convert 3-dimensional data into manageable 3D objects. People can appreciate a 3D rendered view of their new home much better than a set of 2D plans. This logic applies to other fields. Where the traditional 2D representation fails to convey the full picture a 3D representation, with suitable viewer manipulation, may do the job. Schematics and flow charts may benefit from the addition of the 3rd dimension. Tree views could be developed in 3D. Buttons and switches can be developed in 3D and the user provided with a 3D desktop.

Haptik devices and stereo vision are both maturing technologies which in the not to distant future will provide PC users with a true 3D work space with which virtual 3D objects can be manipulated: for examples, have a look at

```
www.cmis.csiro.au/imvs/
  level2/imvsimages.htm
```


References

JD Foley, A van Dam, SK Feiner, and JF Hughes (1990) *Computer Graphics: Principles and Practice* 2nd Edn. Addison-Wesley: Massachusetts.

W Schroeder, K Martin, and B Lorensen (1997) *The Visualisation Toolkik* 2nd Edn. Prentice-Hall: New Jersey.

M Woo, J Neider, T Davis, and D Shreiner (1997) *OpenGL Programming Guide* 3rd Edn. Addison-Wesley: Massachusetts.

J Wernecke, (1994) *The Inventor Mentor* Addison-Wesley: Massachusetts.

RS Wright and M Sweet (1999) *OpenGL Super Bible* 2nd Edn. Waite Group: Indianapolis.

Included Third Party Code

OpenGL12.pas. The OpenGL prototypes and some helper routines. Mike Lischke from the Delph-Jedi website.

Geometry.pas. Basic 3D geometry routines and data structures. Mike Lischke from the Delph-Jedi website.

DGLUT.pas. DGLUT is an Object Pascal translation of a small part of Mark Kilgard's GLUT library for OpenGL. Bob Crawford.

George Black has spent the last 7 years developing specialist computer aided design, visualisation and rendering based applications for the PC using Delphi. He is currently employed by Datavis (www.datavis.com.au) where he has developed 3DShapes Toolkit, a set of native Delphi components providing 3D scene management and rendering. He can be reached at georgeb@datavis.com.au