# Real Time 3D

## *Developing real time animated 3D graphics with OpenGL*

***by Ian Ringrose and Joseph Steel***

The type of 3D animation normally seen in films and on TV is based on the principle that consecutive frames are created, drawn and saved. They are then re-drawn as part of an overall moving sequence: a sort of film strip of stored computer pictures. This form of animation has the advantage that the graphics can be very realistic, but is limited in that there is no real time interaction: you can't change things as the animation sequence happens. The principle of real time animation means that the display process is done 'on the fly'. As one frame is being displayed, the next is being created. When it is complete the frames are switched and the process starts over again. The display can therefore be controlled on a frame-by-frame basis, making real time animation ideal for applications like games or simulation training.

To make displayed motion appear smooth, the frame buffer normally has to be updated at a frequency of 25 frames per second or faster. This depends on the application: flight simulators may work at a frequency of 60 frames/sec but some forms of information graphics update at 5 frames/sec or less. The key factor with real time graphics is that there is always a trade-off between displayed realism and frame update frequency. Adding realism to the picture invariably results in a slower frequency. To increase either requires more efficient software and/or more processing power.

### Development Decisions

A couple of years ago we started looking at the problem of how to design a real time 3D graphics system which would allow the user to build and display relatively complex 3D scale models. We had previously been using PCs with expensive graphics accelerator cards and hand coded all of the graphics according to the application specification (years of development!). The introduction of the Pentium PC meant that for the first time an off the shelf PC had the power to produce detailed imagery at a reasonable frame update frequency. This, coupled with the vast improvement in graphics cards (many having 3D acceleration on-board), allowed us to plan a development based on the PC.

We selected Delphi because of its ease of use and it was then the only Windows development environment to offer re-usable components. So, the system development was organized in two parts: the component part, which provided an interface to all graphics and motion control classes, and the application part, which allowed a user to build and visualise real time 3D models.

We realized that by developing the class interface as a component it could be used as a programmer's tool to develop this application and other applications which required real time 3D. In addition to AiG company projects, the component could be marketed to assist in the development of other products and projects. We decided to use Silicon Graphics' OpenGL for the source 3D programs. This had originally been adapted for use with Windows NT and was, at that time, becoming available for Windows 95. Our task was to provide an easy to use programming interface with OpenGL and then to use this to develop the application.

Dave Jewell discussed using OpenGL with Delphi in Issue 28 (December 1997): a good place to look for a gentle introduction!

### View3D Component

We call our class interface component `View3D`. It comprises Delphi descendant classes which allow the programmer to build graphics realism and motion into a display without the angst which can result from making complex library calls. The programmer instances and creates objects which inherit the properties and methods contained in the `View3D` classes. These properties and methods can be accessed programmatically (at runtime), or through the Object Inspector.

A table showing the outline structure of `View3D` is given in Figure 1. The classes fall into two distinct categories: those which are designed to provide an interface between a Delphi application and OpenGL, and those which are designed to control the position and motion of constituent 3D models. The `TView3D` class has properties and methods which simplify access to OpenGL. The `TView3DModel`, `TView3DReference` and `TView3DRoute` classes allow data to be transformed in 3D space, either individually or collectively, as models and polygonal shapes.

The sample code in Listing 1 shows how `View3D` is programmed in an application. It creates two 3D cylinders and applies textures to them using the `TView3DMaterial` class. These cylinders are then referenced, using `TView3DReference`, as individual children of a combined parent object. Applying transformations to each child using `TView3DPosition` allows the cylinders to move independently. They are, at the same time, moved collectively using the parent reference. Finally, `TView3DRoute` is used with the parent reference to control the motion of the 'collective' model between pre-defined waypoints (`TView3DRouteWayPt`).

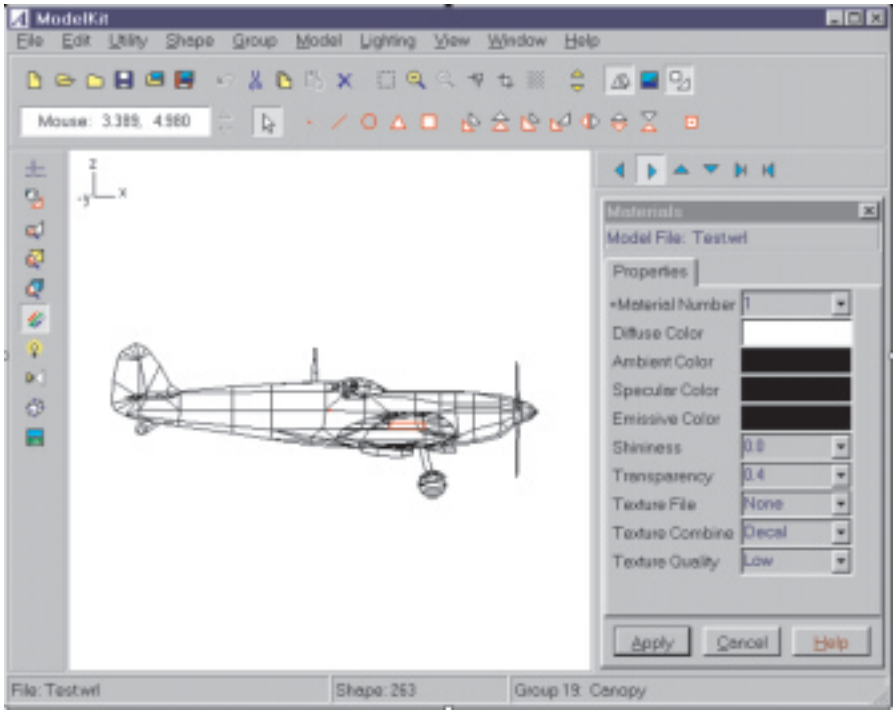
### ModelKit: The Application

ModelKit has two distinct functions: one to allow the user to build

| Class | Properties & Methods | Types |
|---|---|---|
| TView3D | property Ambient: (Ambient lighting) | TView3DColor: (red, green, blue) |
| | property Background: (Background color) | TView3DColor: (red, green, blue) |
| | property Cull: (Back shape culling) | Boolean |
| | property Eye: (Eye position) | TView3DPosition: (x, y, z, roll, pitch, yaw) |
| | property Fill: (Fill style) | TView3DFill: (point, line, solid) |
| | property Fog: (Fog effect) | TView3DFog: (back, front, density, mode, state) |
| | property Lights: (Lighting) | TView3DLights: (azimuth, color, elevation, state) |
| | property Shade: (Shading) | TView3DShade: (flat, smooth) |
| | property Size: (View window size) | TView3DSize: (bottom, height, left, width) |
| | property Volume: (View projection) | TView3DVolume: (angle, back, front, projection, scale) |
| | method AddReference: (Adds ref to view) | PView3DReference: (model, position) |
| | method Draw: (Draws the view) | |
| | method RemoveReference: (removes ref) | PView3DReference: (model, position) |
| | method GetBitmap: (returns view as bitmap) | TBitmap |
| TView3DModel | method Empty: (clears model object) | |
| | method AddPolygon: (adds a shape to the display list) | |
| | method AddLine: (adds a line to the display list) | |
| | method Smooth: (changes the shading characteristics) | |
| TView3DRoute | property Position: (waypoint position) | TView3DPosition: (x, y, z, roll, pitch, yaw) |
| | property Continuous: (route continually loops) | |
| | property Count: (number of waypoints) | |
| | property Period: (time from first to last waypoint) | |
| | property TimeBase: (used to shift the route period) | |
| | method Add: (adds a waypoint) | TView3DRouteWayPt: (x, y, z, pitch, yaw, roll, speed) |
| | method Clear: (clears all waypoints) | |
| | method Insert: (inserts a waypoint) | |
| | method Remove: (remove waypoint) | |
| | method Time: (time at waypoint) | Cardinal |

➤ *Figure 1: Outline of View3D Class Structure*

➤ *Figure 2: ModelKit main form and dialog*




and apply controlled motion to 3D 'scale' models and the other to allow the user to view these models in real time.

It comprises a user interface with drawing canvas, menu items and speed buttons and has a number of dialog forms which are designed to set up and control parameters relating to scaling and 3D viewing. These include calibration, lighting, materials, background color and fog effects. Figure 2 shows ModelKit being used to model a Spitfire aircraft.

The drawing canvas uses Delphi's comprehensive set of built-in graphics functions which are encapsulated in the `TCanvas` object. The canvas analogy is easy to program and includes the `TBitmap` object for importing and

handling bitmaps on the canvas drawing surface. ModelKit also uses `TBitmap` to access textures which are subsequently mapped in real time using `View3DMaterial` class.

The ModelKit software is designed to provide a clearly defined model building sequence. This starts by importing scanned bitmaps of plans or drawings and displaying them on the canvas. These are then scaled to calibrate real world size with the drawing area, and polygonal shapes based on triangles and quadrilaterals are superimposed on the bitmap to construct the model shape. As this all happens in 2D, heights have to be entered separately to provide a third ordinate. A snap-to facility allows shapes to be bound together with common points and edges, like a web of triangular and quadrilateral panels. The model data structure comprises a list of points which contain x, y and z vertex information and a list of shape definitions which reference the points.

The drawing canvas is updated by double buffering background bitmaps. This works for most drawing operations, but Delphi `MouseMove` events for dragging points and shapes around the canvas are very slow and the XOR drawing mode is used as a faster alternative (third party components are available to allow direct access to the Windows frame buffer and hence speed up these events).

Additional dialogs comprise panels, edit and combo boxes and are used to edit color, texture and lighting values. Other general purpose tools which assist the user to construct models are available as menu items or speed buttons on the ModelKit main form. They include fencing, snap-to, zoom, view selection, cut, copy, paste, delete and file handling. A separate 3D graphics library supports these utilities.

After constructing a model and applying all of the lighting and texture effects, it remains to apply position and motion transforms. `Waypoint` properties are created which allow models to follow specific routes in 3D space. These properties define the 6 degrees of freedom of any moving object: x, y, z, yaw, pitch and roll. Waypoint inputs are assigned to `TView3DRoute` properties and class methods are used to interpolate position and attitude values between the `Waypoints` in real time.

The final step in the sequence is to visualize the 'world' as modelled by the user. A visualize form is added to the system and a `View3D` component is placed on it. `View3D` properties and methods allow the previous dialog assignments (material, lighting, fog etc) to be inherited and rendered as a real time display. A virtual mouse facility is added to control eye position and movement, or the eye can be placed on a moving model. The format of the visualize unit is very similar to that of Listing 1.

## ModelKit Data Structure

The ModelKit data structure is based on a subset of VRML (Virtual Reality Modeling Language.) This format is designed for real time animation and is compatible with internet browsers. The subset includes list index definitions as specified in the VRML `Coordinate3` and `IndexedFaceSet` nodes. ModelKit is designed to build objects of any shape and in this context all the predefined shapes part of VRML is ignored. Nevertheless, ModelKit output can be displayed on or off the internet using a browser (with a VRML plugin), but ModelKit can only read VRML files which it creates.

Until now, we have dealt with a two element data hierarchy: models and polygonal shapes. VRML introduces a third element, the group. Group nodes are collections of shapes which form some common part of a model. In the Spitfire example, over 800 shapes are used to create the model aircraft. These are divided into 19 groups, one of which represents the propeller, another the canopy, a third the pilot, etc. The inclusion of groups in the ModelKit data structure allows a model to be broken up into logical parts,

➤ *Facing page: Listing 1*

making modelling easier for the user and also isolating those parts which may be subject to separate transformations (eg the propeller or the wheels.) If we look back to Listing 1 and replace the child references with Spitfire model groups and the parent with the complete Spitfire model, then the parent aircraft would be 'flown' in 3D space whilst rotating the propeller group, lowering and raising wheel groups, etc. Figure 3 shows the example model with texture camouflage, transparent cockpit, lighting and fog effects. Figure 4 illustrates the use of the `TView3DRoute` class, with the Spitfire moving between specified waypoints. Although the model is a realistic representation of a Spitfire, the average update rate on a 100MHz Pentium PC is about 3 frames/sec. This results in very jerky interpolated motion as the aircraft follows its waypoints. Improvements could be made by the modeler by reducing the number of polygons in the model, or using a more powerful PC.

ModelKit currently handles 10 models, each with 20 groups of shapes. This allows complete worlds to be simulated: terrain, buildings, aircraft, etc. It could also be used for more abstract work such as commercial advertising, design or art. These worlds are stored in VRML inline files which comprise a list of all the individual model filenames in the world. In this way the user doesn't have to load a world file by file, but still has the option to load individual models if required. All files are saved as VRML .wrl types.

## Development Problems

We use both Windows 95 and NT for software development, with their respective OpenGL DLLs. Windows 95 runs out of memory and becomes unstable when larger model objects are declared. First indications of this problem are intermittent loss of picture on the visualize form. An intermittent *invalid access* exception also

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, View3D, Mat3D, Model3D, Math3D,
  Ref3D, Route3D, Text3D;
const
  CylinderRadius : Single   = 10.0;
  CylinderHeight : Single   = 20.0;
type
  TForm1 = class(TForm)
    View3D1 : TView3D;
    Timer1  : TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
    procedure FormDestroy(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    private
      procedure ConstructCylinder;
      procedure ConstructRoute;
      procedure Draw;
      procedure LoadTextures;
    public
      child1   : TView3DReference;
      child2   : TView3DReference;
      material : array [0..11] of TView3DMaterial;
      model    : TView3DModel;
      parent   : TView3DReference;
      radius   : Single;
      texture  : array[0..5] of TView3DTexture;
      route    : TView3DRoute;
      time     : Single;
  end;
var
  Form1 : TForm1;
implementation
{$R *.DFM}
procedure TForm1.Draw;
begin
  View3D1.Draw;
end;
procedure TForm1.FormCreate(Sender: TObject);
{ construct the view, called when the form is created }
begin
  model := TView3DModel.Create;
  parent := TView3DReference.Create;
  child1 := TView3DReference.Create;
  child2 := TView3DReference.Create;
  child1.Model := @model;
  child2.Model := @model;
  child1.Position.X := CylinderRadius;
  child2.Position.X := -CylinderRadius;
  parent.Add(@child1);
  parent.Add(@child2);
  LoadTextures();
  ConstructCylinder();
  { === calculate polygon vertex normals === }
  model.Smooth(-180.0, 0.0001);
  View3D1.AddReference(@parent);
  { === initialise eye radius === }
  radius := Sqrt(View3D1.Eye.X * View3D1.Eye.X +
                 View3D1.Eye.Y * View3D1.Eye.Y +
                 View3D1.Eye.Z * View3D1.Eye.Z);
  ConstructRoute();
end;
procedure TForm1.LoadTextures();
var
  bitmap : TBitmap;
  i      : Integer;
  name   : TFileName;
begin
  name := 'textureX.bmp';
  bitmap := TBitmap.Create;
  for i := 0 to 5 do begin
    name[8] := Char(i + Integer('1'));
    texture[i] := TView3DTexture.Create;
    bitmap.LoadFromFile(name);
    texture[i].Bitmap  := bitmap;
    texture[i].Quality := Low;
    texture[i].Combine := Modulate;
  end;
end;
procedure TForm1.ConstructCylinder();
{ construct a 12-sided cylinder }
var
  angle  : Single;
  i      : Integer;
  pt     : array [0..23] of TView3DPoint3;
  vertex : array [0..3] of TView3DVertex;
begin
  for i := 0 to 11 do begin
    angle := i * 2.0 * Pi / 12;
    pt[i].x := CylinderRadius * Sin(angle);
    pt[i].y := -CylinderHeight;
    pt[i].z := CylinderRadius * Cos(angle);
    pt[i + 12].x := pt[i].x;
    pt[i + 12].y := CylinderHeight;
    pt[i + 12].z := pt[i].z;
  end;
  for i := 0 to 11 do begin
    material[i] := TView3DMaterial.Create;
    material[i].Diffuse.Red   := 255;
    material[i].Diffuse.Green := 255;
    material[i].Diffuse.Blue  := 255;
    material[i].texture := @texture[i mod 6];
    vertex[0].point := pt[i];
    vertex[0].texture.x := 0.0;
    vertex[0].texture.y := 0.0;
    vertex[1].point := pt[(i + 1) mod 12];
    vertex[1].texture.x := 1.0;
    vertex[1].texture.y := 0.0;
    vertex[2].point := pt[(i + 1) mod 12 + 12];
    vertex[2].texture.x := 1.0;
    vertex[2].texture.y := 1.0;
    vertex[3].point := pt[i + 12];
    vertex[3].texture.x := 0.0;
    vertex[3].texture.y := 1.0;
    model.AddPolygon(vertex[0], vertex[1], vertex[2],
      @material[i]);
    model.AddPolygon(vertex[0], vertex[2], vertex[3],
      @material[i]);
  end;
end;
procedure TForm1.ConstructRoute();
{ construct a simple route }
var
  w : array[0..3] of TView3DRouteWayPt;
begin
  route := TView3DRoute.Create;
  w[0].x := -50.0;
  w[0].y := 0.0;
  w[0].z := 0.0;
  w[0].pitch := 0.0;
  w[0].yaw   := 0.0;
  w[0].roll  := 0.0;
  w[0].speed := 25.0;
  { ** CODE OMITTED for w[1] to w[3] : SEE DISK ** }
  route.Add(w[0]);
  route.Add(w[1]);
  route.Add(w[2]);
  route.Add(w[3]);
  route.continuous := True; { set route to auto-loop }
  time := 0.0;
end;
procedure TForm1.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
{ change eye point depending on mouse position }
var
  lat  : Single;
  long : Single;
begin
  View3D1.Eye.Yaw :=
    (X - ClientWidth / 2) / ClientWidth * 360.0;
  View3D1.Eye.Pitch :=
    (Y - ClientHeight / 2) / ClientHeight * 180.0;
  { calculate x, y and z values }
  long := DegToRad(View3D1.Eye.Yaw) + Pi;
  lat := DegToRad(View3D1.Eye.Pitch);
  View3D1.Eye.X := radius * Sin(long) * Cos(lat);
  View3D1.Eye.Y := radius * Sin(lat);
  View3D1.Eye.Z := radius * Cos(long) * Cos(lat);
end;
procedure TForm1.FormDestroy(Sender: TObject);
var
  i : Integer;
begin
  model.Empty;
  model.Destroy;
  parent.Destroy;
  child1.Destroy;
  child2.Destroy;
  for i := 0 to 5 do begin
    texture[i].Destroy;
  end;
  for i := 0 to 11 do begin
    material[i].Destroy;
  end;
  route.Destroy;
end;
procedure TForm1.FormResize(Sender: TObject);
begin
  View3D1.Size.Width  := ClientWidth;
  View3D1.Size.Height := ClientHeight;
  Draw;
end;
procedure TForm1.FormPaint(Sender: TObject);
begin
  Draw;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  child1.Position.Pitch := child1.Position.Pitch + 10.0;
  child2.Position.Yaw   := child2.Position.Yaw - 10.0;
  route.Evaluate(time);
  time := time + 0.1;
  parent.Position := route.Position;
  Draw();
end;
end.
```

➤ *Left, Figure 3: Spitfire with textures, transparency and fog*

➤ *Right, Figure 4: Interpolated positions derived from waypoints*

occurs when running OpenGL code in the Delphi IDE with the integrated debugger turned on. Turn it off and this problem disappears! This is an annoyance rather than a hindrance and, although neither ourselves nor Borland Assist could find a solution to the problem, it did not significantly affect the development timeframe. As yet, we have had no memory or debugger problems with NT and we'd recommend NT for larger projects.

We have noticed a problem with OpenGL when using the fog effect. If a model is clipped against the side of the display the surfaces which are being clipped turn completely white. This has been observed in a number of graphics systems which use OpenGL and seems to be present in even the latest versions of the library.

## Real Time Application

Real time 3D imagery is currently used in applications which require user interaction. Typical examples are training simulators or games, where a feedback loop exists between user control inputs and the displayed imagery.

The quality of real time 3D graphics will improve as PC processors and memory get faster, and better 3D graphics cards emerge. We believe that this improvement will be reflected in the popularity and use of the technology, particularly in markets like education and IT.

The `View3D` component is being marketed by AiG Limited as a Delphi 3 developer tool. The component approach has allowed far greater flexibility than with an application. The good news is that a trial version of View3D is included on this month's companion disk, in the VIEW3D directory.

ModelKit is still being developed and will complement `View3D` as a PC-based 3D modeling and visualization tool when released later this year. Versions of `View3D` will have a data reader which will allow models that have been created in ModelKit to be read directly into an application.

Information on both products can be found at www.aignet.co.uk

Ian Ringrose has 15 years experience in real time 3D graphics. He has a PhD in computer simulation and is a Chartered Engineer. Joseph Steel has a degree in applied mathematics with 12 years experience as a real time graphics programmer and analyst. They both consult for AiG Limited on projects relating to computer graphics and can be contacted via info@aignet.co.uk