dot *software* ltd

39 High Street
Weston
Bath
B&NES BA1 4BX
ENGLAND

Tel +44 (0)1225 481100
Fax +44 (0)1225 482100
web http://www.dotsoftware.co.uk
email info@dotsoftware.co.uk
Registered No. 3387287

hyper graphics

Article

HyperGraphics WAV Viewer Example

Issued to

Address

Article : , issue 1, 4 September 2003
Prepared by : Andy Noton

Introduction

The idea of Rapid Application Development aided by re-usable software components has been around for a long time now. Delphi puts RAD theory into practice very elegantly and there are thousands of third-party Delphi components available to fulfil every imaginable need.

That being the case, it is surprising that anyone needs to write code any more. But they do and, for real-world projects in science and engineering, code continues to be written in large quantities, often without the help of ready-made components. There are several reasons for this, but one principle issue is that available components aren't flexible enough to meet customer requirements exactly.

Using **HyperGraphics**, an object-oriented graphics library written by **Dot Software Ltd**, this article illustrates how some of the traditional limitations of off-the-shelf software can be overcome to meet the challenges of more demanding, technical, graphics applications: in this case a viewer application for WAV audio files.

HyperGraphics: The software behind the scenes

HyperGraphics (or **HG** as it is more commonly referred to by its developers) was created to allow engineers and scientists to present professional looking graphical data in Windows applications at a time when there were few other tools around to do the job.

The library provides basic objects representing lines, boxes, shapes, axes, text, pictures, arrows (to name a few). Other more complex pre-defined objects, for example, connector, table, text box, print-preview, zoom control, key, matrix line, are provided by combining the basic objects. The developer is free to create his/her own objects based on existing ones and **HG** supports full, interactive, visual editing of objects at design-time and runtime.

The result is ready-to-use professional plotting and diagramming functionality that is as fast and easy to use as a simple charting component. Plus, with its open OO framework, **HG** provides the flexibility and extensibility that is crucial in fulfilling those awkward last few requirements. In short, if **HG** doesn't do it 'out of the box', or your client doesn't like the way something looks, you can tweak the software to make it do exactly what you want, without having to start from scratch.

You can download a fully functional evaluation copy of **HyperGraphics** from www.hypergraphics.co.uk. The latest version supports Delphi 5, 6 & 7, C++ Builder, Visual C++ and Visual Basic. The download includes all the code for this example.

WAV Viewer

The example application lets you open any 16-bit uncompressed PCM format WAV audio file and plots signal level against time for left and right channels. You can zoom in, down to the level of individual samples, zoom out to display the entire file and you can play the currently displayed portion of the file to your sound card.

Sounds trivial, so why a WAV file viewer?

Firstly, as an audio enthusiast, I've never been able to find an entirely satisfactory (and free) viewer/editor, despite the wide range of programs available. I haven't evaluated them all by any means, but ones I have tried are too expensive, or cumbersome to use, or too complicated, or limited in zooming capabilities. So I was looking for an excuse to write my own. Even the limited functionality described here is a significant step towards the viewing and editing functions that I want to use myself.

Viewing WAV files also presents some interesting technical challenges that can be effectively tackled using the speed and flexibility of **HG**. WAV files can be very large indeed and one of my own requirements is to be able to view the whole file in one go. For audio tracks this means plotting tens of millions of points and doing it quickly, so in this example I explore some techniques for optimising the drawing process.

Finally, the WAV Viewer example demonstrates how to use **HG** objects to build custom features like interactive zooming, a status/progress bar and a play cursor, which indicates the progress through the WAV file as we listen.

I'll start by going through the process of setting up the **HG** objects at design-time. Then I'll look at some of the interesting aspects of the code.

First Steps

The example uses the **HyperGraphics** `THGPanel` VCL component, so the first step is to download and install the Delphi version of **HyperGraphics**. This is fairly straightforward, but make sure you follow the instructions in the file "Installing packages.txt", which appears at the end of the install. The file covers the process of getting the **HG** help files properly integrated into Delphi, which is well worth it to be able to use the standard F1 context help from the code editor.

Now start a new application and drop a `THGPanel` on the form. The **HyperGraphics** components can be found on the "HG" page in the Component Palette. `THGPanel` derives from `TCustomPanel` and starts off looking like a blank white panel. We want to use the whole form client area, so set the `Align` property to `alClient`.

To transform the blank panel into something more powerful, we use the **HyperGraphics** *Designer*. Double-clicking on the `THGPanel`, the *Designer* tool window appears, showing a single object "HGwindow". This is the object represented by the white background area of the `THGPanel` component. An **HG** window forms the root of a parent-child hierarchy of **HyperGraphics** objects, which the *Designer* presents in a tree view. Right-click on "HGwindow" and choose *Add* > *Views* > *View*. Doing so creates a rectangle with coloured selection handles; this will be the plotting area for the left audio channel. The object appears in the *Designer* tree view as a child of the **HG** window. As with other tree views, you can click on the items to rename them, so we'll change the name from "HGview1" to "LeftView".

We're aiming to show two plots, for left and right channels, so we need to make room for the right channel. The red selection handle is for moving and the blue ones are for resizing. Resizing from centre-top or centre-right changes height or width; using the corner selection handles preserves the aspect ratio.

Reduce the height of the view and move it up to the top half of the panel. Now add a second view in the same way as the first, name it "RightView" and reduce its height so the two views sit one above the other as shown in Figure 1.

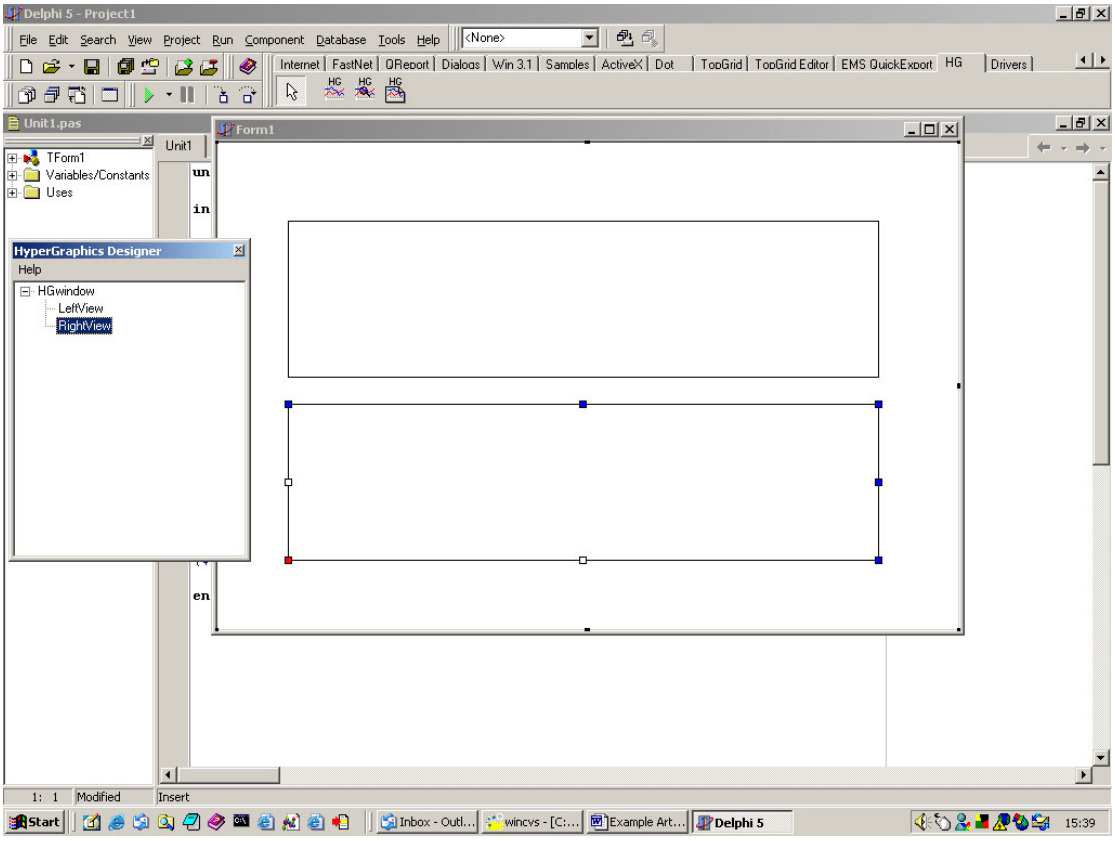

Figure 1: The first two **HyperGraphics** objects

We are using view objects here because they can have axes attached to them, allowing us to plot the WAV data. But, before adding axis and line objects there is some fine-tuning of the views to do.

Right click on the upper view (left channel) and select *Properties* from the pop-up menu. Figure 2 shows the first page of the view's *Properties* dialog. So far the view objects we have added have behaved rather like Delphi components – we dropped them on the window and moved and resized them. There is a bit more to positioning and sizing with **HyperGraphics** though. One important feature of **HyperGraphics** is that the origin for position information is bottom-left, in keeping with mathematical coordinate systems, rather than the top-left system more generally used in Windows. Another is that you can change the units that objects use for their position and size. As Figure 2 shows, the default *units* for the view's position is *CM*, so the bottom left corner of the view is positioned 1.7 cm from the left and 6.2 cm from the bottom of the parent "HGwindow" object.

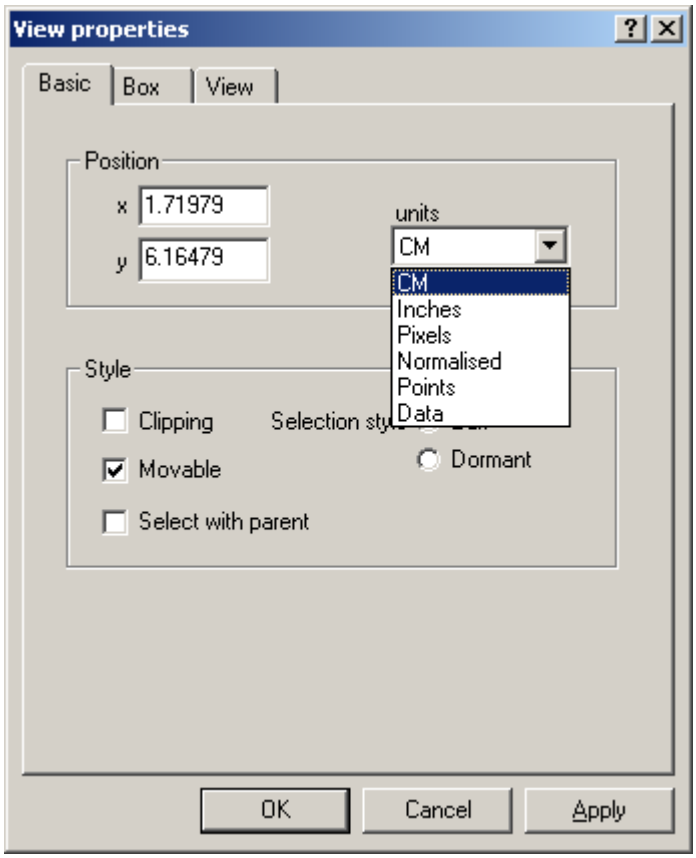

Figure 2: Basic View properties

Change the *Position units* to *Normalised* and press *Apply*. You will notice that the *x* and *y* values in the dialog change. Normalised units indicate that the object is positioned relative to the extent of its parent, so a position of (0, 0) represents bottom-left, while (1, 1) represents top-right. Changing the units means that the views will keep the same relative size and position whatever the size of the application window. Click on the *Box* tab and change the size units to *Normalised* and press *OK*. If you run the application now and maximize the window, you will see the effect of using normalised vs cm units.

Now change the *Position units* and *Size units* to *Normalised* for the lower view (right channel).

Progressing further

After that careful start, we can pick up the pace a bit and add the remaining objects needed for this example. To each of the views we want to add an x-axis, a y-axis and a data line object. You can add these one at a time if you like, but it is useful to know that adding a data line will cause axes to be added automatically. Right click on each view and choose *Add* > *Lines* > *Data line*. We will come back to the data line and axes shortly, but for now just name them “LeftXAxis”, “LeftYAxis”, “LeftLine”, “RightXAxis” and so on.

I also wanted to create a progress and status bar and after a couple of minutes messing around with VCL components, I thought - why not build it with **HyperGraphics** just to show off? To do this we can add another view to the “HGwindow” object, name it “StatusBar”, change its *Size units* and *Position units* to *Normalised* and change its *Vertical Alignment* to *Top* (in the *Box* tab of the *Properties* dialog). Then position the view as a thin bar across the top of the window, tall enough for a piece of text, as Figure 3 demonstrates.

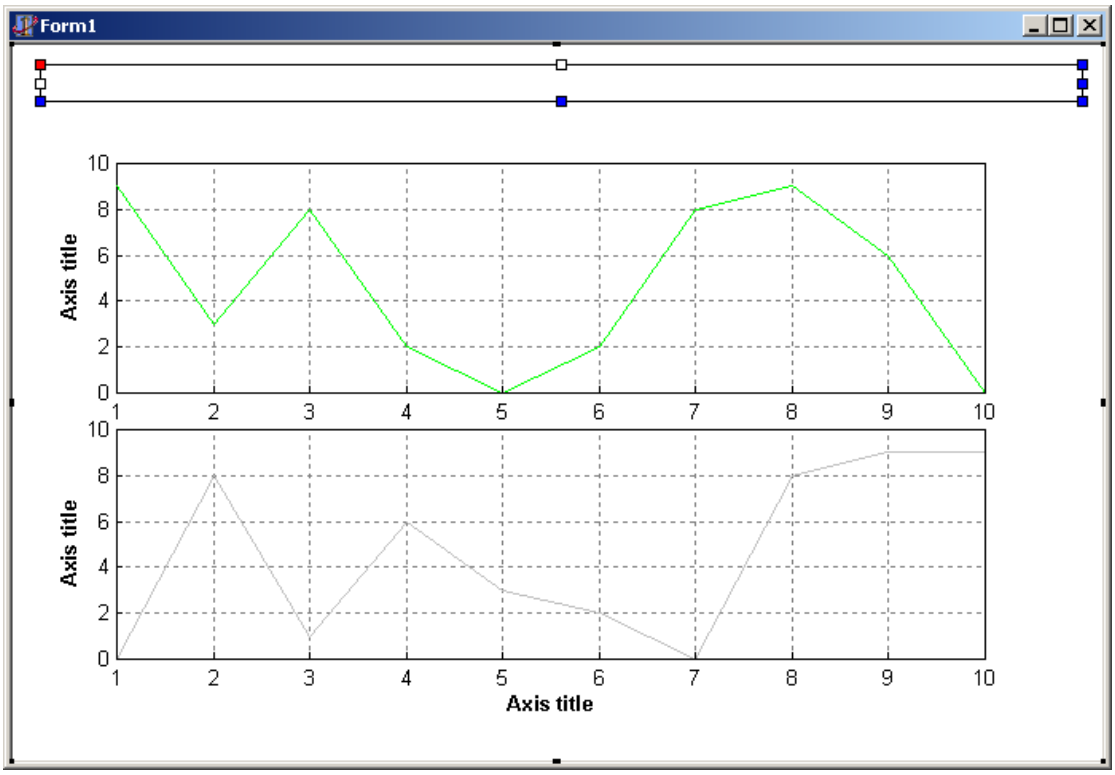

Figure 3: Status bar in progress…

Right-click on the "StatusBar" view and choose *Add* > *Text* > *Text*. As you would expect by now, the text appears at the bottom left of its parent. Name the text object "Status". I want to centre the text in the status bar, so open the text object's Properties dialog and change the *Position units* to *Normalised* and set *x*, *y* to (0.5, 0.5). Then on the *Text* tab of the *Properties* dialog change *Vertical Alignment* to *Middle* and *Horizontal Alignment* to *Centre*.

The text object implements the status function of the bar. For the progress indicator, we will use a filled rectangle that advances across the status bar. This will be drawn in XOR mode so that the text remains visible. Right-click on the "StatusBar" view and select *Add* > *Shapes* > *Rectangle*. Name the rectangle object "Progress" and change the following properties: *Position units* to *Normalised*, *Size units* to *Normalised*, *width* to 0, *height* to 1, *Alignment* to bottom-left, and *Fill colour* to pale yellow. Progress bars are usually blue on my machine and that is what you get if you XOR pale yellow on to a white background. If that is a bit too obscure then you can set the desired fill colour at design-time and modify it later on in the code with `Progress.Fillcolour := Progress.Fillcolour xor clWhite`. In fact because we can't set it at design-time, we have to write another line of code to set XOR mode, but I'll cover that later on. Otherwise the progress bar is completed.

At this point, we can turn our attention back to the axes and lines. The data line uses *Data* units by default, which means that the line's points are positioned according to the scales of the parent view's axes. So adding a data line automatically created a pair of axes in the view because axes are required to support data units. On the *Line* tab of the line's *Properties* dialog, set the *Line colour*. I have used the standard colours for stereo connectors - black for left channel and red for right.

By default, axes auto-scale to accommodate all objects in the view that use data units. Alternatively you can set the axis limits manually. In this example the y-axes will auto-scale, whilst the x-axes' limits will be set programmatically as we zoom in and out at runtime. In the *Properties* dialog of each x-axis, uncheck the *Auto limits* box and set the *Low* limit to 0. It is now up to the code to set the x-axis limits. Both x-axes always use the same limits, so to tidy things up we can hide the "LeftXAxis" object. To do this, right click on "LeftXAxis" and uncheck the *Visible* option.

To change the axis titles to something more meaningful, first select the view then move the mouse over an axis title. You'll notice the cursor changes to an `I` bar, indicating that you can click on the text to do in-place editing. To finish editing, click again. Change the x-axis title to "Time (s)" and the y-axis titles to "Left" and "Right".

The final **HG** objects to add are two play cursors, which are simple vertical lines. Right click each view and choose *Add* > *Lines* > *Line*. Name the objects "LeftCursor" and "RightCursor". The blue selection handles on the line indicate that the points can be moved. Move the first point to the bottom left corner of the view and the second point to the top left corner, so that the line covers the left hand edge of the view. Now change the line's position units to Data, to allow the code to position the cursor using time values. Finally set both cursors to be invisible, because we only want to draw them during playback of the WAV file.

A *Main Menu*, a *File Open Dialog* and a *Timer* are also required. Note that to allow in-place editing at design-time `THGPanel` captures mouse events when it is selected and the **HG** *Designer* tool window is visible. To be able to drop components onto the form on top of a selected `THGPanel`, either press escape first to select the parent form, or close the *Designer* window.

The main menu has a *File* menu that should include *Open*, *Play*, *Stop* and *Exit* options and a *View* menu that should feature a *Zoom Out* option.

All the design time objects are set up now and should look similar to Figure 4.

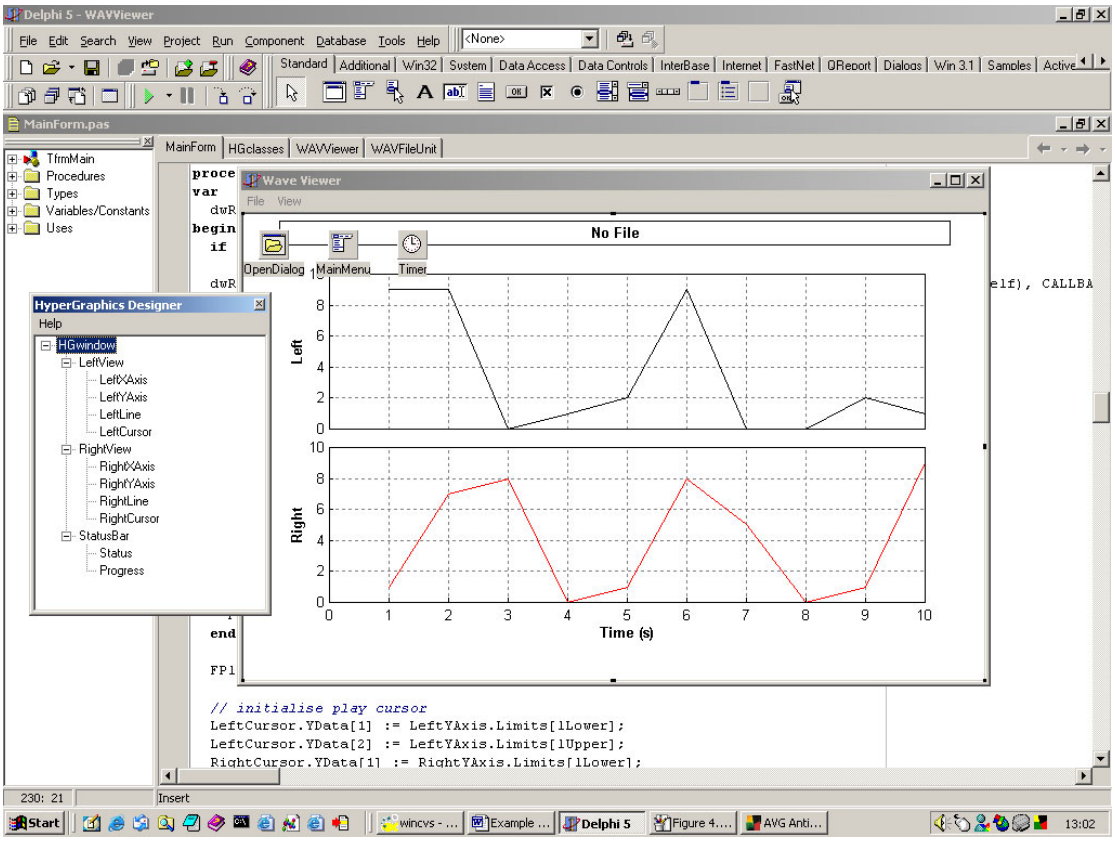

Figure 4: Completed main form at design-time

The Code

As well as the main form that we have been creating, the WAV Viewer project includes two units for dealing with WAV files. These come from **Dot Software**'s Useful Code Archive. I won't dwell on their implementation here as they simply use the standard Win32 functions from the Delphi MMSystem unit to read and play WAV data.

We can begin the discussion of the code with some event handlers. The main form's `FormShow` event handler calls a function called `HGInit`, see Listing 1, which does some useful initialisation. First it sets the XOR drawing mode for the play cursor and progress bar objects, as mentioned earlier.

I used data units for the play cursor so that its x position can be the number of seconds of playback. For the y positions, we need to make sure that the cursor covers the full height of the auto-scaling plot, so we set values that are outside the possible range of the 16-bit integer data. I mentioned earlier that auto-scaling axes set their limits to accommodate any object in the view that uses data units, so you would expect that setting these y positions for the play cursor would just expand the y-axis range. That is true by default, but **HG** gives any drawing object the option of opting out of this via the `OnGetYLimits` event. Here we assign an event handler that returns `False` to indicate that the play cursors don't get included in the auto-scaling.

The final feature of Listing 1 is the assignment of a handler for the `LeftView.OnDraw` event. With this line of code we are doing something very powerful. In contrast to the VCL event model, `OnDraw` is not just an event that is called before the view is drawn; the event handler actually assumes full responsibility for drawing the view to the device context. We have the choice of either doing the default drawing [using the `DefDrawObj` method to draw the object and all its children] or doing custom drawing [for example, calling `DrawObj`, which draws a particular object only (perhaps a different object entirely)]. Alternatively we can do no drawing at all. The "HGwindow" object gives access to the device context via its `Hdc` property, so we can even call GDI functions directly.

This event model is applied throughout **HyperGraphics** to give the developer full control to modify the default behaviour. In addition to events we have encountered to override the drawing and auto-scaling, every drawing object has a host of other events, including moving, resizing, selection, property editing, using connectors, mouse events and serialisation.

```
procedure TfrmMain.HGInit;
begin
  // adjust properties not set at design-time
  LeftCursor.Drawmode := HG_XOR;
  RightCursor.Drawmode := HG_XOR;
  Progress.Lineobj.Drawmode := HG_XOR;
  // set play cursor height to exceed possible y-axis limits
  LeftCursor.YData[1] := 2*Low(SmallInt);
  LeftCursor.YData[2] := 2*High(SmallInt);
  RightCursor.YData[1] := 2*Low(SmallInt);
  RightCursor.YData[2] := 2*High(SmallInt);
  // exclude cursor from autoscale
  LeftCursor.OnGetylimits := GetCursorLimits;
  RightCursor.OnGetylimits := GetCursorLimits;
  // set up event to perform 'intelligent' data reduction
  LeftView.OnDraw := LeftViewDraw;
end;
```

Listing 1: `HGInit` method

The reason for intervening in the drawing process is to deal with the potentially huge amount of data to be drawn. Some WAV file viewers get round this by displaying the file in zoomed in sections, but I have decided I want to see the whole thing and be able to zoom in and out quickly. These requirements imply keeping all the data in memory.

The `miOpenClick` method, which responds to the *File* | *Open* menu item allocates an array for the 16-bit integer WAV data and reads the entire file in chunks from the `WavFile` object, which is basically a file stream. After each chunk the `ShowStatus` method updates the progress bar.

```
procedure TfrmMain.ShowStatus(const Msg: string; Percent: Integer);
begin
  Status.Textstring := Msg;
  Progress.Size[sWidth] := Percent /100.0;
  // immediate redraw
```

```
  StatusBar.DrawObject;
end;
```

Listing 2: `ShowStatus` method

It is worth mentioning `ShowStatus` (Listing 2) because it is so simple and effective. The percentage value is handled effortlessly because we chose to use normalised units for the "Progress" rectangle. We have also demonstrated another very useful feature of **HyperGraphics** - the ability to draw each object independently. I could have used `HGPanel.Invalidate` here, which would queue a WM_PAINT message and eventually redraw the whole panel. But calling `StatusBar.DrawObject` immediately redraws the status bar leaving the rest of the panel intact.

Compressing the WAV data

To speed up drawing and reduce memory usage the `LeftViewDraw` event handler compresses the WAV data into a form that looks exactly the same as the real data. I have mentioned that we have full control here to draw anything we like, but, in fact for this example, we want to draw the view in the normal way and this is done by the call to `DefDrawObj` at the end of the routine. The purpose of the code preceding the draw is to reduce the amount of data passed to the line objects for plotting. Although we only assign the left channel view's `OnDraw` event, the routine sets up the data lines for both channels. Because it was defined before "RightView", "LeftView" is drawn first.

As well as compressing the points that are visible, we can save a lot of time and memory by simply ignoring the points that are off the scales. The main form's `SetXLimits` method makes sure the x range is never smaller than one sample interval. At this sort of level it is easy to see that the x-axis limits are not necessarily integral sample numbers, so the properties `FirstSample` and `LastSample` are used to return the sample number at or below the lower x limit and the sample number at or above the upper x limit.

A **HyperGraphics** line object uses double precision data to store its points. When the points are scaled down into integer pixel values, a large data set often contains repeated pixel points. To optimise the drawing, **HG** automatically throws away the redundant points. In a practical test, when I plot a very detailed sine wave consisting of 1 million double precision points, the number of unique pixel points comes out at something like 800.

So what if I try 1 million points of random y data? Disappointingly, things start to slow down a little here. Because the y values are not changing gradually like the sine wave, the random function will result in far fewer repeated pixel positions. We will get a similar effect for audio data with a wide frequency content.

Figure 5 shows a 20ms section of data from a WAV file. In Figure 6 we see the same file with the time axis zoomed out by a factor of roughly 500, so we can imagine that all the points in Figure 5 are represented by a single pixel position in Figure 6, indicated by the green arrows.

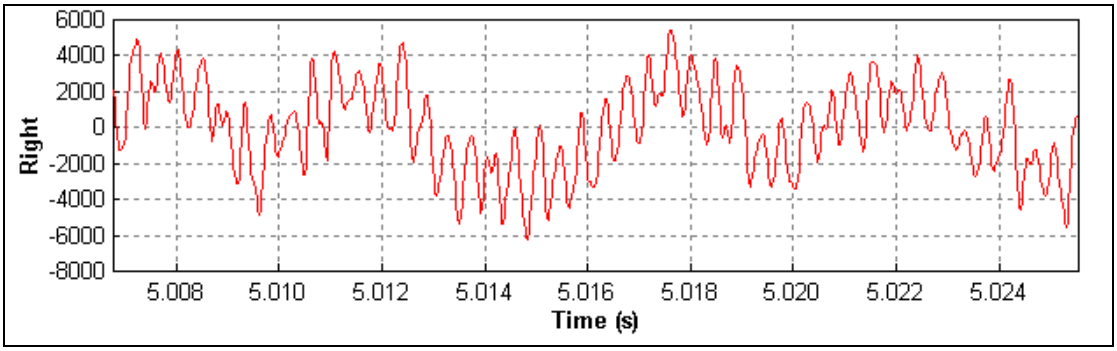

Figure 5: Zoomed-in WAV data

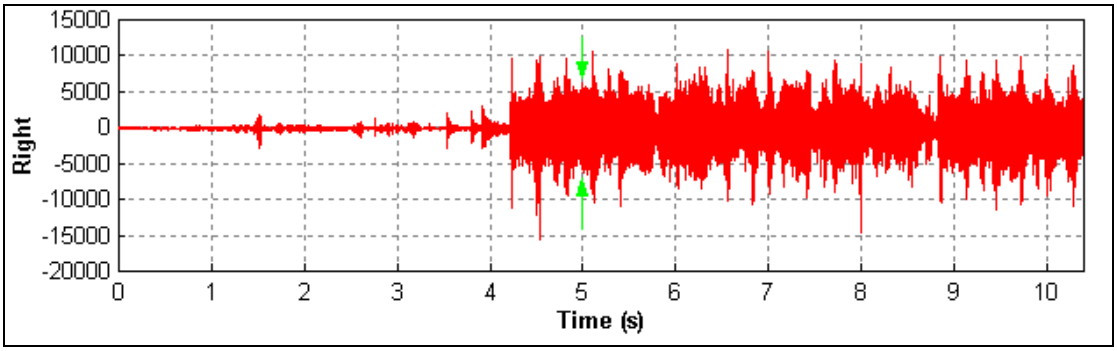

Figure 6: Zoomed-out WAV data

What begins as a complex waveform in Figure 5 appears as a single vertical line in Figure 6. If we were to draw Figure 6 using all the available data we would be drawing hundreds of points up and down the same vertical line. Not only would it be slow, the memory usage would also be much greater. The WAV data alone require 2 bytes per channel per sample. The line objects require 16 bytes (8 bytes each for x and y) per channel per sample. My largest WAV file contains 55 million samples. We probably don't need to do the maths to know that the cost in memory would be prohibitive if we put the data directly into the line objects.

However, if we know that all the points from Figure 5 contribute to a single vertical line on Figure 6, we can simplify the data as I have shown in Figure 7. To get the same vertical line on Figure 6, we provide equivalent data using just four points – the start point, minimum y point, maximum y point and the finish point. All we need to know in order to do the compression is which samples convert to which pixels.

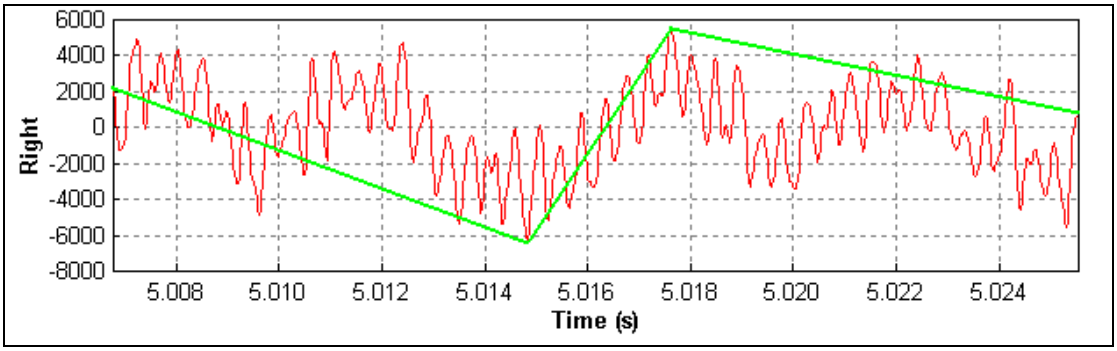

Figure 7: Equivalent data after compression

Pixels differ from other units in **HyperGraphics** because they are discrete rather than continuous. I always find that a diagram is very helpful here. Figure 8 shows a line crossing a very small view, consisting horizontally of four pixels, and an x-axis that shows the mapping to data units. Each coloured group of points in the line scales down to one of the four pixels in the x direction. The view's left hand pixel is n and the right hand pixel is n+3. The x-axis lower limit maps to the centre of the view's left hand pixel and the upper limit to the centre of the right-hand pixel.

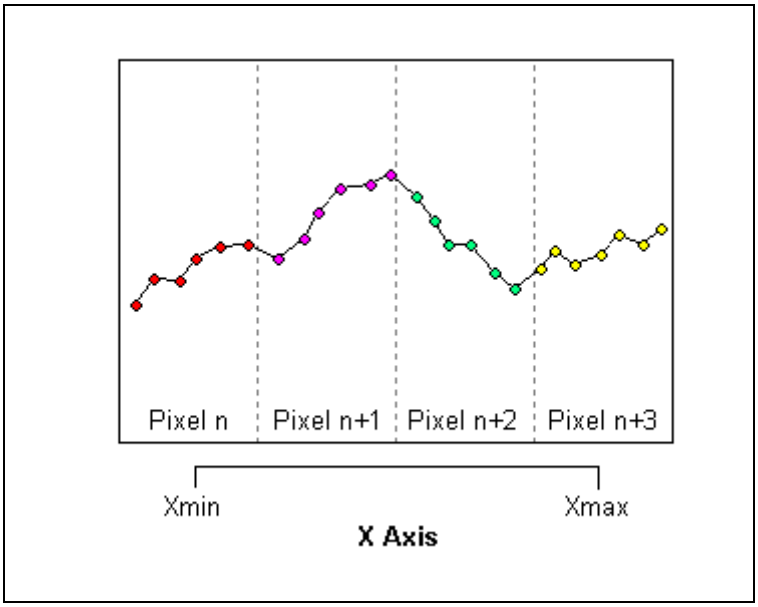

Figure 8: Pixel mapping in **HyperGraphics**

In Listing 3 we see how the `LeftViewDraw` routine expresses this mapping. The view's `GetExtent` method returns the left and right hand pixel values and the difference is divided by the axis range to give the conversion factor `PixInc`. Now the axis limits are not necessarily integral sample numbers, so `PixOfs` is also needed to enable us to find the left and right boundaries of a pixel in terms of sample numbers.

```
// find sample no. to pixel conversion so that Pixel = PixOfs +PixInc*SampleNo
if LastSample>=FirstSample then
begin // get pixel conversion
  // get pixel extent of view
  LeftView.GetExtent( Extent );
  // pixels/sample = pixel width of view / axis range in s / samples per s
  PixInc := (Extent.Right-Extent.Left) /
            (LeftXAxis.Limits[lUpper]-LeftXAxis.Limits[lLower]) /
             FFormat.nSamplesPerSec;
  // PixOfs is Pixel value of Sample zero
  PixOfs:= Extent.Left -
           PixInc*LeftXAxis.Limits[lLower]*Format.nSamplesPerSec;
end;
```

Listing 3: Pixel conversion

If there are more that four samples per pixel, the next step is to generate the four equivalent data points per pixel, which give the same appearance as the original data. Listing 4 is another extract from the `LeftViewDraw` routine that shows how this is done. Using `PixInc` and `PixOfs`, the lower and upper boundaries are calculated for each pixel, yielding the group of points that contribute to the pixel. The groups are represented in Figure 8 by the different colours. Then the `DataMaxMin` method is used to return the maximum and minimum y values for each group of points.

```
// get pixel range
FirstPix := PixRound(PixOfs + PixInc*FirstSample);
LastPix := PixRound(PixOfs + PixInc*LastSample);

iPt := 1; // HG indexes are one-based
// set four points per pixel
LeftLine.Npoints := 4*(LastPix-FirstPix+1);
if FFormat.nChannels>1 then
  RightLine.Npoints := 4*(LastPix-FirstPix+1)
else
  RightLine.Npoints := 0;
```

```
for iPix := FirstPix to LastPix do
begin // create four data points per pixel
  // get sample no corresponding to lower boundary of pixel
  SampleBdy := (iPix-0.5-PixOfs)/PixInc;
  // select first sample point for this pixel
  iFirstOfPix := Max( Ceil(SampleBdy), 0 );

  // get sample no corresponding to upper boundary of pixel
  SampleBdy := (iPix+0.5-PixOfs)/PixInc;
  // select last sample point for this pixel
  iLastOfPix := Min( Floor(SampleBdy), FSampleCount-1 );
  // last sample no must be less than (not equal to) boundary value
  if iLastOfPix = SampleBdy then Dec(iLastOfPix);
  // find max/min data values for this pixel
  DataMaxMin(iFirstOfPix+1, iLastOfPix-1, chLeft, LMaxData, LMinData);
  if FFormat.nChannels>1 then
    DataMaxMin(iFirstOfPix+1, iLastOfPix-1, chRight, RMaxData, RMinData);

  // add data to HG line
  ...
end;
```

Listing 4: Generating visually equivalent data

Finally, the four equivalent data points for each pixel are added to the line objects. The code also caters for the case where we are zoomed in, and there are fewer than four WAV data points per pixel; in this case all the points between `FirstSample` and `LastSample` are added to the line objects.

Either way, the `LeftViewDraw` routine manages to reduce the data set to a manageable maximum of a few thousand points whilst preserving an exact visual representation of the data.

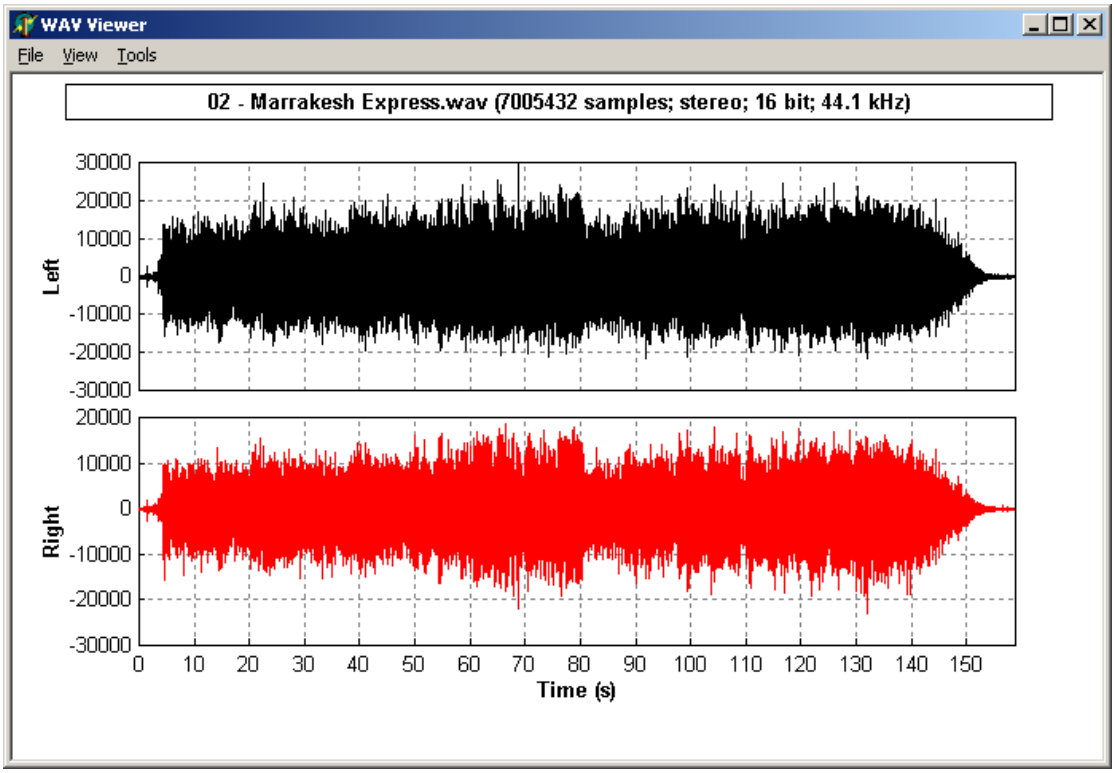

Figure 9: Seven million points of Crosby, Stills and Nash plotted in 0.2 s

To demonstrate the effect of the compression algorithm, I added a Tools menu with a Compression on/off switch and a Drawing stats option, which displays the time taken to prepare the line data and

draw the plots. Using a single WAV file and gradually zooming in, these tools along with Task Manager gave the results in Figure 10.

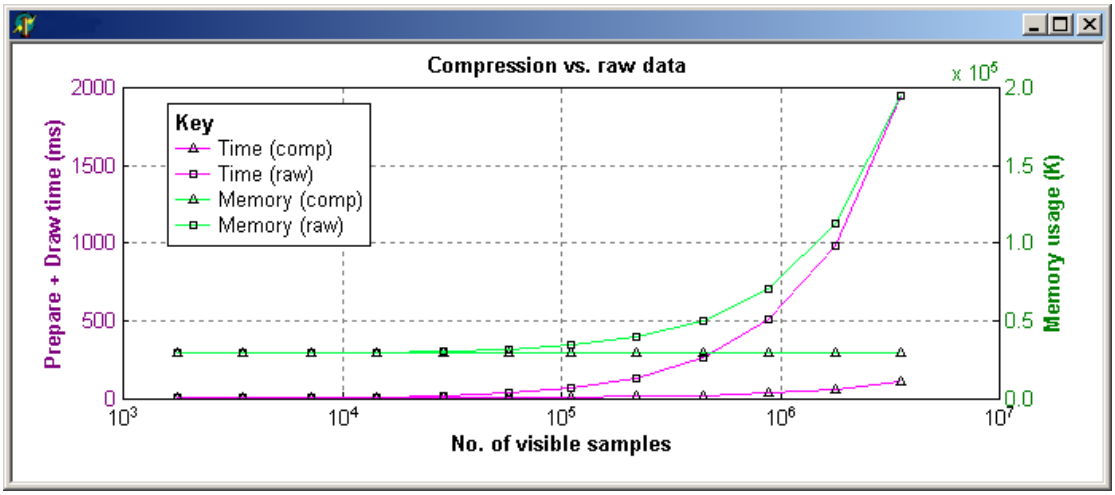

Figure 10: Compression vs. raw data

For compressed operation the memory usage is constant – about 95% of which was the array of WAV data. You can see that the speed improvement with compression is dramatic, even with a few hundred thousand samples visible. The `PixInc < 0.25` criterion used in the code is not necessarily optimal, but unfortunately the timing is not accurate enough to tell us at what point using the data raw becomes quicker than compressing it.

User interaction

Zooming in is achieved using a rubber-band box. **HG** provides some high-level functions to do this and Listing 5 shows them put to use in the **HG** Panel's `MouseDown` and `MouseUp` event handlers.

```
procedure TfrmMain.HGPanelMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  HGPanel.HGWindow.RubberbandStart;
end;

procedure TfrmMain.HGPanelMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  pos: THGPositionArray;
  size: THGSizeArray;
begin
  if HGPanel.HGWindow.RubberbandFinish then
  begin
    // set axes from rubber band box
    LeftView.GetRubberBand(pos, size, HG_DATA);
    SetXAxisLimits(pos[pX], pos[pX]+size[sWidth]);
    // redraw
    HGPanel.Invalidate;
  end;
end;
```

Listing 5: Rubber-band selection of axis limits

The play cursor is implemented with the help of a timer. `miPlayClick` moves the cursors to the start position and draws them. Then each time timer updates, the current playback position is queried and the cursors are redrawn. I look at pixel values here too, using the cursor line object's `GetPosAnyunits` and `ConvertPos` methods. By checking if the cursor's on-screen position has changed, we can eliminate any unnecessary redraws.

Conclusion

In engineering and scientific software, it is rare to find an application that is without special or unusual requirements of some sort. Where these challenges are graphical ones, **HyperGraphics** allows us to customise behaviour to give optimum performance from a minimum amount of code. The example I have used here illustrates this approach well, and yet only covers a small range of **HyperGraphics**' capabilities.

There are plenty of additional features that we could have added using **HyperGraphics**, such as interactive block selection using a rectangle object with cursor objects, or a record function with real-time display. Yet more features, such as diagramming objects and serialisation, are left to other examples.

Specialist software development does not have to mean writing everything from the ground up. For skilful developers needing re-usable graphical components that offer real power and flexibility, **HyperGraphics** is an essential tool.