Add Gauges to Your Applications

Doug Hennig

They say a picture is worth a thousand words. That's especially true with a gauge control, which allows a user to see at a glance how some value compares to a goal amount. In this month's article, Doug presents an easy-to-use gauge control you can use in your VFP applications.

People like visual images. Most people would rather see a chart than columns of raw numbers because it's easier to see the relationships between items visually. Adding analysis tools like charts and gauges to your applications make them much more valuable to your users.

What is a gauge? A gauge is an image that shows how a single value compares to a maximum or goal value. The two values can be anything: current sales compared to budget, volume to date compared to the maximum allowable volume, and so on. For example, many charitable organizations show the current status of their fund raising campaigns as a thermometer. The top of the thermometer represents the value of the fund raising goal and the height of the bar inside the thermometer represents how much money has been raised so far.

The most common type of gauge looks like a speedometer in a car; see **Figure 1** for an example. The end of the gauge represents the maximum value and the position of the needle represents the current value. Color bands around the outside edge of the gauge show the ranges of certain categories. For example, in Figure 1, the red band indicates where sales are too low, the yellow band where they're acceptable but not great, and the green band where sales should be to make the boss happy.

We recently added support for gauges to my company's flagship product, Stonefield Query (www.stonefieldquery.com). In this article, I'll show you how we did it.

Figure 1. It's easy to draw beautiful gauges using the Gauge class (although this is printed in black and white, the actual gauge is full color).

The Gauge class

There's just a single class used to draw gauges: Gauge in Gauge.VCX (additional components are also required as I'll discuss later). It's a subclass of Custom so it has no visible appearance at runtime. How does the gauge appear then? After you call the DrawGauge method, the cImage property is set to the bytes for the gauge image. You can use FILETOSTR() to write the contents of cImage to a file, such as if you want to use the gauge in a report, or set the PictureVal property of an Image object if you want it to appear in a form. For example, SampleGauge.SCX, one of the forms included in the downloads for this article, uses this code to have the Gauge object referenced in the oGauge property draw a gauge in an Image named imgGauge:

```
with This
    .oGauge.nSize = .imgGauge.Width
    .oGauge.DrawGauge()
    .imgGauge.PictureVal = .oGauge.cImage
endwith
```

Table 1 lists the properties of the Gauge class. As you can see, there are quite a few of them. Most of them affect the appearance of the gauge. The best way to check out how the various properties work is by running SampleGauge.SCX, shown in Figure 2. Each control has a tooltip specifying which property it controls. Changing any setting immediately redraws the gauge so you can instantly see the effect.

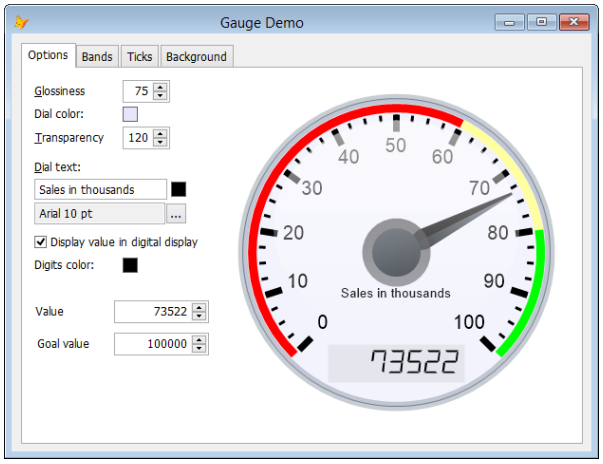

Figure 2. SampleGauge.SCX allows you to experiment with the properties of the Gauge object.

Here are comments about some of the properties:

- nGlossiness controls how bright the "glossy" ellipse that appears at the top of the gauge is. This ellipse gives the illusion of reflected light, as if the gauge was made of glass or plastic.
- The gauge size is determined by the nSize property; since a gauge is drawn as a square, the height and width of the image are both set to nSize.
- By default, the labels indicating the values around the gauge are sized based on the size of the gauge: the larger the gauge, the bigger the labels. Set lAdjustLabelSize to .F. if you want to use the font size specified in nLabelFontSize instead.
- cFormat indicates how the labels are formatted. It has to use .NET syntax for number formats for reasons that will be obvious later. .NET syntax uses "#" as an optional digit placeholder, "0" as a digit placeholder that displays 0 if there is no digit, "." for a decimal separator, and "," to use a thousands separator (unlike VFP, only one is needed in the format even when numbers exceed one million). The format string is surrounded with "{0:" and "}", so the default of "{0:#,##0}" specifies no leading zeros, no decimal places, and thousands separators.
- nBand1End indicates where on the outer rim of the gauge the first color band appears, such as the red band in Figure 1. Only the end value is needed, since the band starts at 0. Similarly, nBand2End and nBand3End indicate the ending positions of the second and third color bands, with the starting positions being the end of the previous band.
- By default, nBand1End, nBand2End, and nBand3End are assumed to be percentages, so a value of 35 indicates an ending position of 35% of the gauge arc. If you want to use amounts instead, such as 25,000, set lValuesAsPercentages to .F.
- Since the gauge can't go above 100%, the needle is pegged at the maximum value when nValue is greater than nMaxValue. Because you may want the maximum value to be a goal that could be exceeded (for example, a salesperson's monthly quota may be $10,000 but they certainly could sell more than that), you can set nGoalPosition to a lower value than 100. For example, if you set it to 75, then the nMaxValue value appears at 75%.
- For larger numbers (over 1,000), the labels can overlap the major tick marks, so increase the value of nLabelDistance accordingly. Alternatively, you can set nLabelFactor to a value to divide the labels by so smaller numbers are shown. For example, if you set nLabelFactor to 10000, the 5,000 position on the gauge appears as "5."

Creating a dashboard

Dashboards are all the rage these days. A dashboard is a form displaying multiple panels of information, such as charts, reports, and of course gauges. Another sample form that comes with the downloads for this article, Dashboard.SCX (Figure 3), is a simple demo of how a dashboard might work.

This form is actually quite simple. Its Init method adds eight Image controls and sizes and positions them so they take up two rows of four images. A Timer on the form calls the form's DrawGauges method, which uses a single Gauge control to do the drawing. DrawGauges sets the properties of the Gauge control to different values for each gauge (different dial colors and different current and maximum values), draws the gauge,

and sets the PictureVal property of each Image control to the resulting image. Just for fun, the timer fires every 2 seconds and shows a random value so you can see the needles move.

Figure 3. A dashboard consists of multiple gauges.

Table 1. The properties of the Gauge class.

Property	Description
cDialText	The text for the dial
cDialTextFontName	The font for the dial text
cErrorMessage	The text of any error that occurs
cFormat	The format for the labels in .NET syntax: {0:#,##0} by default
cImage	The gauge image
cLabelFontName	The font for the labels
lAdjustLabelSize	.T. to adjust the label size based on the gauge size, .F. to use the size specified in nLabelFontSize
lDialTextFontBold	.T. if the dial text is bold
lDialTextFontItalic	.T. if the dial text is italics
lDisplayDigitalValue	.T. to display the value in a digital display
lLabelFontBold	.T. if the label text is bold
lLabelFontItalic	.T. if the label text is italics
lValuesAsPercentages	.T. to use percentages for values, .F. to use amounts for values
nBackColor	The background color or the starting color for a background gradient
nBackColor2	The end color for a background gradient; if it's the same as nBackColor, there is no gradient
nBackColorAlpha	The alpha for the background color
nBackGradientMode	The mode for a background gradient: 0 = left to right, 1 = top to bottom, 2 = from top left, 3 = from top right
nBand1Color	The color for band 1
nBand1End	The ending position for band 1
nBand2Color	The color for band 2
nBand2End	The ending position for band 2
nBand3Color	The color for band 3
nDialAlpha	The alpha for the dial color
nDialColor	The color to use for the dial
nDialTextColor	The color for the dial text
nDialTextFontSize	The font size for the dial text

nDigitsColor	The color for digital digits
nGlossiness	The glossiness value (0 – 100)
nGoalPosition	The position where the goal value appears on the gauge; defaults to 100
nLabelColor	The color to use for labels
nLabelDistance	The distance between labels and major tick marks
nLabelFactor	The factor to use for labels: 1, 1000, 10000, etc.
nLabelFontSize	The font size for the labels
nMajorTickColor	The color of major tick marks
nMajorTickCount	The number of major ticks
nMaxValue	The goal value for the gauge
nMinorTickColor	The color of minor tick marks
nMinorTickCount	The number of minor ticks
nSize	The height and width of the gauge (it's a square so they're the same)
nValue	The current value for the gauge
oBridge	A reference to a wwDotNetBridge object
oGauge	A reference to a .NET GaugeControl object

How Gauge works

The Gauge class is actually a wrapper for a .NET DLL that does all the work. I'll discuss the .NET class later.

To avoid COM registration and other issues, I use Rick Strahl's wwDotNetBridge utility, which I discussed in the January 2013 issue of FoxRockX. As you can see in **Listing 1**, the Init method of Gauge instantiates wwDotNetBridge into the oBridge property. Since you usually only want a single instance of wwDotNetBridge in an application, you can pass an existing instance to Init instead. Init also loads the Gauge.DLL .NET assembly and instantiates the Gauge.GaugeControl class into the oGauge property.

Listing 1. The Init method sets up the helper objects needed by the class.

```
lparameters toBridge

* If we were passed a wwDotNetBridge object,
* use it. Otherwise, create one.

if vartype(toBridge) = 'O'
  This.oBridge = toBridge
else
  This.oBridge = newobject('wwDotNetBridge', ;
    'wwDotNetBridge.prg', '', 'V2')
endif vartype(toBridge) = 'O'
loBridge = This.oBridge

* Load the Gauge assembly: it must be in the
* current directory or path.

if not loBridge.LoadAssembly('Gauge.dll')
  This.cErrorMessage = 'Gauge.dll could ' + ;
    'not be loaded: ' + loBridge.cErrorMsg
  return
endif not loBridge.LoadAssembly('Gauge.dll')

* Instantiate a GaugeControl object.

This.oGauge = ;
 loBridge.CreateInstance('Gauge.GaugeControl')

* Set cFormat to a default we can't set in the
* Properties window.

This.cFormat = '{0:#,##0}'
```

Since the .NET DLL does all the work, all the DrawGauge method of the Gauge class has to do is populate the properties of the .NET object with the values of its own properties, call the .NET object's DrawGauge method, and put the return value, which is the bytes of the gauge image, into cImage. The code for DrawGauge is shown in **Listing 2**.

Listing 2. The DrawGauge method uses the GaugeControl object to draw the gauge.

```
local lnEnd1, ;
    lnEnd2, ;
    lnMaxValue
with This

* Get the band positions.

    lnEnd1     = .nBand1End
    lnEnd2     = .nBand2End
    lnMaxValue = .nMaxValue/100

* If the band values were entered as amounts,
* convert to percentages.

    do case
        case .lValuesAsPercentages
        case .nMaxValue = 100
            && max value hasn't been set
            lnEnd1 = 35
            lnEnd2 = 70
        otherwise
            lnEnd1 = int(lnEnd1/lnMaxValue)
            lnEnd2 = int(lnEnd2/lnMaxValue)
    endcase
endwith
with This.oGauge

* Set the appearance properties.

    .AdjustLabelSize     = ;
        This.lAdjustLabelSize
    .BackColor           = ;
        This.GetColor(This.nBackColor, ;
        This.nBackColorAlpha)
    .BackColor2          = ;
        This.GetColor(This.nBackColor2, ;
        This.nBackColorAlpha)
    .BackGradientMode    = ;
        This.nBackGradientMode
```

```
    .Band1Color          = ;
        This.GetColor(This.nBand1Color)
    .Band1End            = lnEnd1
    .Band2Color          = ;
        This.GetColor(This.nBand2Color)
    .Band2End            = lnEnd2
    .Band3Color          = ;
        This.GetColor(This.nBand3Color)
    .DialColor           = ;
        This.GetColor(This.nDialColor, ;
        This.nDialAlpha)
    .DialText            = This.cDialText
    .DialTextColor       = ;
        This.GetColor(This.nDialTextColor)
    .DialTextFontName    = ;
        This.cDialTextFontName
    .DialTextFontSize    = ;
        This.nDialTextFontSize
    .DialTextFontBold    = ;
        This.lDialTextFontBold
    .DialTextFontItalic  = ;
        This.lDialTextFontItalic
    .DigitsColor         = ;
        This.GetColor(This.nDigitsColor)
    .DisplayDigitalValue = ;
        This.lDisplayDigitalValue
    .LabelFontBold       = This.lLabelFontBold
    .LabelFontItalic     = ;
        This.lLabelFontItalic
    .LabelFontName       = This.cLabelFontName
    .LabelFontSize       = This.nLabelFontSize
    .Format              = This.cFormat
    .Glossiness          = This.nGlossiness
    .Height              = This.nSize
    .LabelColor          = ;
        This.GetColor(This.nLabelColor)
    .LabelDistance       = This.nLabelDistance
    .LabelFactor         = This.nLabelFactor
    .MajorTickColor      = ;
        This.GetColor(This.nMajorTickColor)
    .MajorTicks          = ;
        This.nMajorTickCount
    .MinorTickColor      = ;
        This.GetColor(This.nMinorTickColor)
    .MinorTicks          = ;
        This.nMinorTickCount

* Set the value properties.

    .MaxValue = This.nMaxValue * ;
        100/This.nGoalPosition
    .Value    = This.nValue

* Draw the image and set our cImage property
* to the image bytes.

    This.cImage = .DrawGauge()
endwith
```

The only complication in DrawGauge is that VFP color values don't match up with .NET color values: the .NET values have the red, green, and blue components reversed, and also support an alpha, or transparency, value. So, DrawGauge calls a helper method named GetColor (**Listing 3**), which pulls out the color components and puts them into the order needed for .NET.

Listing 3. The GetColor method converts a VFP color number to the .NET equivalent.

```
lparameters tnColor, ;
    tnAlpha
local lnRed, ;
    lnGreen, ;
    lnBlue, ;
    lnAlpha
lnRed   = mod(tnColor, 256)
lnGreen = mod(bitrshift(tnColor, 8), 256)
lnBlue  = mod(bitrshift(tnColor, 16), 256)
lnAlpha = iif(vartype(tnAlpha) = 'N', ;
    tnAlpha, 255)
return rgb(lnBlue, lnGreen, lnRed) + ;
    bitlshift(lnAlpha, 24)
```

The .NET component

GaugeControl.cs, included with the downloads for this article, is the source code for the .NET gauge component in Gauge.DLL. I started with code created by Ambalavanar Thirugnanam, available from http://tinyurl.com/ppl44uy, and made a number of changes to it:

- I modified it to be a simple .NET class that returns an image as a string rather than a Windows Forms User Control that displays the gauge. This allows the image to be written to a file or displayed in a VFP Image control without having to worry about registering the .NET control as an ActiveX control and adding it to a VFP form.
- I added properties for various colors, such as the background color, rather than using hard-coded values.
- I added support for a background gradient in addition to a solid color.
- Because it's difficult to create a .NET Font object in VFP, even with wwDotNetBridge, I added properties for the name, size, bold, and italics settings of fonts used for the dial text and labels. GaugeControl uses these properties to instantiate a Font object with the specified settings.

If you're interested in how the .NET component works, I recommend reading the article at http://tinyurl.com/ppl44uy as it discusses the logic and math involved in drawing the gauge. Then examine the C# source code in GaugeControl.cs to see how it's implemented.

If you build the Gauge solution that includes GaugeControl.cs, you'll find that it has a post-build event that copies the DLL to the parent folder of the solution, which is the same folder as Gauge.VCX is located. Note that if you've used the VFP Gauge control and VFP is still open, you have to close VFP before building the .NET solution because the .NET DLL is still open in VFP.

Since GaugeControl.cs uses GDI+ to do all of the drawing, why didn't I convert the C# code to VFP code using the VFPX GDIPlusX project? After all, that would give us a 100% VFP solution

with no need for wwDotNetBridge or Gauge.DLL. The reason I didn't is two-fold:

- Why reinvent the wheel? It would've taken several hours to convert the C# code into the equivalent VFP code and there'd be lots of debugging to make sure it works the same.
- I've run into some performance issues with GDIPlusX on some machines. In fact, this is what prompted me to look at this solution in the first place. I was using the VFPX FoxCharts project to draw gauges but found that on some systems, it was taking a minute or more to draw the gauge. In tracking the problem down, I found that on those systems, some of the GDI+ function calls were taking an order of magnitude longer to execute, and these functions were called thousands of times for each gauge. I don't know why some systems have this performance problem with GDIPlusX but the .NET component has no such problems on those systems.

Deploying Gauge

Deploying Gauge is straightforward:

- Add Gauge.VCX and wwDotNetBridge.PRG to your project.
- Use the Gauge class as you see fit: to create image files for reports (cImage is in PNG format) or for the source of images in forms.
- Include wwDotNetBridge.DLL, ClrHost.DLL, and Gauge.DLL in your installer or copy those files to the client's system. No registration is required for any of these components.

Gauge.DLL requires version 2.0 of the .NET framework. Windows Vista and later come with .NET 2.0 so this is only an issue for Windows XP and earlier. If you use Inno Setup as your application installer, you can make your installer detect whether .NET 2.0 is missing and automatically download and install it by adding #INCLUDE DotNet2Install.iss to your Inno script file. DotNet2Install.iss is included in the downloads for this article, as is Isxdl.DLL, a component used by DotNet2Install.iss.

Summary

The Gauge class and supporting components make it easy to add beautiful, customizable gauges to your applications. I look forward to any feedback you have to enhance this tool.

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 to 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).