Optimizing Visual Basic

Presented by: **Scott Swanson**
Product Manager
Microsoft Corporation

Optimization Philosophy

Optimization is not a single set of tricks or techniques. It's not a simple process you can tack on at the end of your development cycle: "There, it works. Now I'll speed it up and make it smaller." To create a truly optimized application you must be optimizing it all the time while it is being developed. You choose your algorithms carefully, weighing speed against size and other constraints; you form hypotheses about what parts of your application will be fast or slow, large or compact, and you test those hypotheses as you go.

And you remember that optimization is not always completely beneficial. Sometimes the changes you make to speed up or trim down your application result in code that is hard to maintain or debug. And some optimization techniques fly in the face of structured coding practice, which may cause problems when you try to expand it in the future or incorporate it into other programs (not to mention aggravating fellow programmers if you're working as part of a team).

Understand the Real Problems

You can waste a lot of time optimizing the wrong things. This is particularly true in Visual Basic because so much goes on "behind the scenes" and there isn't a true profiler available that would enable you to find the proverbial 10% of your code that takes 90% of the time or uses 90% of the space.

Nevertheless, you can learn an awful lot by simply stepping through your code and thinking carefully about what's actually happening. It sounds obvious, but I've discovered things in my own code that "seemed like a good idea" when I coded them, but turned out to be amazing memory or CPU hogs in practice. You often forget that setting properties causes events to occur, and if there is a lot of code in those event procedures an innocuous line of code can cause a tremendous delay in your program.

Know When to Stop

Sometimes things aren't worth optimizing. For example, writing an elaborate but fast sorting routine is pointless if you're only sorting a dozen items. In fact, I've seen programs that sort things by adding them to a sorted list box and then reading them back out in order. In absolute terms this is horribly inefficient, but if there aren't a lot of items it is just as quick as any other method, and the code is admirably simple (if a bit obscure).

There are other cases where optimization is wasted effort. If your application is ultimately bound by the speed of your disk or network, there is little you can do in your code to speed things up.

Optimization

You can optimize your program for many different characteristics:

- Real speed (how fast your application actually calculates or performs other operations).
- Display speed (how fast your application paints the screen).
- Apparent speed (how fast your application appears to run; this is often related to display speed but not always to real speed).
- Size in memory.
- Size of graphics (this directly affects size in memory, but often has additional ramifications when working in Microsoft Windows).

Rarely, however, can you optimize for *multiple* characteristics. Typically, an approach that optimizes size compromises on speed; likewise an application that is optimized for speed is often larger than its slower cousin.

Optimizing Actual Speed

Unless you're doing things like generating fractals, your applications are unlikely to be limited by actual processing speed. Typically other factors, such as video speed, network delays, or disk activities are the limiting factor in your applications. However, you may find points in your program where the speed of your code is the gating factor. When that's the case, there are techniques you can use to increase the real speed of your applications:

- Use **Integer** variables and integer math.
- Cache properties in variables.

Use **Integer** and **Long** integer variables whenever you can, particularly in loops. It's surprising how much of your programs you can write using only **Integer** variables. And you can often tweak things so that you can use integers when a floating point value would otherwise be required. For example, if you always set the Scalemode of all your forms and Picture controls to either Twips or Pixels you can use integers for all the size and position values for controls and graphics methods. In a similar vein, it's often possible to modify calculations so that they can be performed entirely with integers. As an example, Microsoft

QuickBasic included a sample program that generated the Mandelbrot set entirely using integer math.

Never get the value of any given property more than once in a procedure unless you know the value has changed. Instead, you should assign the value of the property to a variable and use the variable. Variables are generally 10 to 20 times faster than properties. For example, code like this is very slow:

```
For i = 0 To 10
        picIcon(i).Left = picPallete.Left
Next i
```

Rewritten, this code is much faster:

```
picLeft = picPallete.Left
For i = 0 To 10
        picIcon(i).Left = picLeft
Next i
```

Likewise, code like this:

```
Do Until EOF(F)
        Line Input #F, nextLine
        Text1.Text = Text1.Text + nextLine
Loop
```

...is much slower than this:

```
Do Until EOF(F)
        Line Input #F, nextLine
        bufferVar = bufferVar + nextLine
Loop
Text1.Text = bufferVar
```

However, this code does the equivalent job and is even faster:

```
Text1.Text = Input(F, LOF(F))
```

Yet another example of a better algorithm being the best optimization.

Optimizing Display Speed

Because of the graphical nature of Microsoft Windows, the speed of graphics and other display operations contributes greatly to the *perception* of the speed of your application. In many cases, you can make your application seem faster simply by making your forms repaint faster -- even if the actual speed of your application hasn't changed at all. There are several techniques you can use to speed up the apparent speed of your application:

- Turn off ClipControls.
- Use AutoRedraw appropriately.

- Use Image instead of Picture box.
- Use Line instead of PSet.
- Hide controls when setting properties to avoid multiple repaints.

Unless you are using graphics methods (Line, PSet, Circle, and Print) you should set ClipControls to False for the form and for all Frame and Picture box controls. When ClipControls is False, Visual Basic does not do the extra work required to avoid overpainting controls with the background before repainting the controls themselves. On forms that contain a lot of controls, the resulting speed improvements are significant.

The right setting for the AutoRedraw property varies, depending on what is being displayed. If you can quickly redraw the contents of the form or picture control using graphics methods, you should set AutoRedraw to False and perform the graphics in the Paint event. If you have a complicated display that doesn't change very often, you should set AutoRedraw to True and allow Visual Basic to do the redrawing for you. Note, however, that when AutoRedraw is True Visual Basic maintains a bitmap it uses to redraw the picture, and this bitmap can take up a considerable amount of memory.

Image controls always paint faster than Picture controls. Unless you need some of the capabilities unique to Picture controls (such as DDE and graphics methods) you should use Image controls exclusively.

Speaking of graphics methods, a little experimentation will demonstrate that the Line method is much faster than a series of PSet methods. Avoid using the PSet method, and batch up the points into a single Line method.

Every repaint is expensive. The fewer repaints Visual Basic must perform, the faster your application will appear. One way to reduce the number of repaints is to make controls invisible while you are manipulating them. For example, suppose you want to resize several List boxes in the Resize event for the form:

```
Sub Form_Resize ()
Dim i As Integer, sHeight As Integer
        sHeight = ScaleHeight / 4
        For i = 0 To 3
                lstDisplay(i).Move 0, i * sHeight, ScaleWidth, sHeight
        Next
End Sub
```

This creates four separate repaints, one for each List box. You can reduce the number of repaints by placing all the List boxes within a Picture box, and hiding the Picture box before you move and size the List boxes. Then, when you make the Picture box visible again, all of the List boxes are painted in a single pass:

```
Sub Form_Resize ()
Dim i As Integer, sHeight As Integer
        picContainer.Visible = False
        picContainer.Move 0, 0, ScaleWidth, ScaleHeight
        sHeight = ScaleHeight / 4
        For i = 0 To 3
                lstDisplay(i).Move 0, i * sHeight, ScaleWidth, sHeight
        Next
        picContainer.Visible = True
End Sub
```

Optimizing Apparent Speed

Often the subjective speed of your application has little to do with how quickly it actually gets through the meat of its task. To the user, an application that starts up rapidly, repaints quickly, and provides continuous feedback feels "snappier" than an application that just "hangs up" while it churns through its work. You can use a variety of techniques to give your application that "snap":

- Keep forms hidden but loaded.
- Use progress indicators.
- Pre-load data you expect to need.
- Use timers to work in the background.

Hiding forms instead of unloading them is a trick that has been around since the early days of Visual Basic 1.0, but it is still effective. Version 2 improved form load speed, but it still isn't as fast as simply making a previously-loaded form visible. The obvious downside to this technique is the amount of memory the loaded forms consume, but it can't be beat if you can afford the memory cost and making forms appear quickly is of the highest importance.

Progress indicators are turning up in almost every significant application, and with good reason: when a process takes a long time, you need to give the user some indication that your application hasn't simply hung. There are a variety of custom controls that can be used as progress indicators, but a simple and effective one can be created using just overlapped label and shape controls. To see it in action, examine the system resources display in the CallDLLs sample application included in Visual Basic 2.0

You can also improve the apparent speed of your application by pre-fetching data. For example, if you need to go to disk to load the first of several files, why not load as many of them as you can? Unless the files are extremely small, the user is going to see a delay anyway. The incremental time spent loading the additional files will probably go unnoticed, and you won't have to delay the user again.

In some applications you can do considerable work while you are waiting for the user. The best way to accomplish this is through a timer control. Use static (or module-level) variables to keep track of your progress, and do a very small piece of work each time the timer goes off. If you keep the amount of work done in each timer event very small, users won't see any effect on the responsiveness of the application and you can pre-fetch data or do other things that further speed up your application

First Impressions

Apparent speed is most important when your application starts. Users' first impression of the speed of an application is measured by how quickly they see something after double-clicking on the EXE in File Manager. While some delay is unavoidable with any significant application, there are some things you can do to give a response to the user as quickly as possible:

- Use Show in Form_Load event.
- Simplify your Startup form.
- Don’t load modules you don’t need.

When a form is first loaded, all of the code in the Form_Load event occurs before the form is displayed. You can alter this behavior by using the Show method in the Form_Load code, giving the user something to look at while the rest of the code in the event executes:

```
Sub Form_Load()
        Show                    ' Display startup form.
        Load MainForm           ' Load main application fom.
        Unload Me               ' Unload startup form.
        MainForm.Show           ' Display main form.
End Sub
```

The more complicated a form is, the longer it takes to load. From this observation comes a rule: keep your startup form simple. Most applications for Microsoft Windows display a simple copyright screen at startup: your application can do the same. The fewer controls on the startup form, and the less code it contains, the quicker it will load and appear. Even if it immediately loads another, more complicated form, it gives the user immediate feedback that the application has started.

Visual Basic loads code modules on demand, rather than all at once at startup. This means that if you never call a procedure in a module, that module will never be loaded. Conversely, if your startup form calls procedures in several modules then all of those modules will be loaded as your application starts up, which slows things down. For this

reason, you should avoid calling procedures in other modules from your startup form.

Keeping It Small

You can reduce the size of your application in memory by:

- Reclaiming space used by strings, arrays, and object variables.
- Avoiding Variant variables.
- Avoiding fixed-length String variables.
- Removing dead code.

The space used by (non-Static) local string and array variables is reclaimed automatically when the procedure ends. However, global and module-level string and array variables remain in existence for as long as your program is running. If you are trying to keep your application as small as possible, you should reclaim the space used by these variables as soon as you can. You reclaim string space by assigning the zero-length string to it:

```
SomeStringVar = ""  ' Reclaim space
```

You reclaim the space used by a dynamic array with the **Erase** statement:

```
Erase LargeArray
```

The **Erase** statement completely eliminates an array; if you want to make an array smaller without losing all of its contents, you can use the **ReDim Preserve** statement:

```
ReDim Preserve LargeArray(10, smallernum)
```

Similarly, you can reclaim some (but not all) of the space used by an object variable by setting it to **Nothing**. For example:

```
Global F As New StatusForm
...
        F.Show 1            ' Form is loaded and shown modally.
        X = F.Text1.Text
        Set F = Nothing     ' Reclaim space.
```

Even if you don't use explicit form variables, you should take care to **Unload** (rather than simply hiding) forms you are no longer using.

Variant variables are another size hog: each **Variant** takes 16 bytes, compared to two for an **Integer** or eight for a **Double**. Variable-length **String** variables use four bytes plus one per character in the string, but each **Variant** containing a string takes 16 plus one per character in the string. Because they are so large, **Variant** variables are particularly troublesome when used as local variables or arguments to procedures, because they quickly consume stack space.

Local fixed-length string variables are another culprit in stack space exhaustion. Unlike variable-length strings, which only use four bytes on the stack (the string itself is allocated out of another segment), the entire contents of a fixed-length string variable is allocated off the stack. For this reason you should avoid local fixed-length string variables in your code.

Finally, if your applications are anything like mine, by the time they are close to being finished they've been redesigned several times. In the process you've probably left behind variables that you're no longer using, and sometimes even whole procedures that aren't being called from anywhere. Unfortunately, Visual Basic does not detect and remove this "dead code" so you have to look for and remove it yourself.

Cutting Back on Graphics

In many Visual Basic applications, the space used by graphics dwarfs the memory used by everything else combined. However, there are also opportunities to accomplish significant savings using some simple techniques:

- Reclaim graphics memory with LoadPicture() and Cls.
- Replace Picture box with Image controls.
- Load Pictures only when needed, and share pictures and icons.
- Use RLE bitmaps or metafiles.

Reclaim memory with LoadPicture() and Cls. If you aren't going to use a Picture or Image control again, don't just hide it: remove the bitmap it contains:

```
Image1.Picture = LoadPicture()
```

Another technique reclaims the memory used by the AutoRedraw bitmap in forms and picture controls (the AutoRedraw bitmap is the bitmap that Visual Basic uses if you set AutoRedraw to True, or if you reference the Image property of forms or picture controls). You can reclaim this memory using code like this:

```
mypic.AutoRedraw = True     ' Turn on AutoRedraw bitmap.
myPic.Cls                   ' Clear it.
myPic.AutoRedraw = False    ' Turn off bitmap.
```

The Picture controls in many Visual Basic applications exist merely to be clicked, or to be dragged and dropped. If this is all you're doing with a Picture control, you are wasting a lot of Windows resources. For these purposes, Image controls are superior to Picture controls. Each Picture control is an actual window, and uses significant system resources. The Image control is a "lightweight" control rather than a

window, and doesn't use nearly as much resources. In fact, you can typically use five to 10 times as many Image controls as Picture controls. Moreover, Image controls repaint faster than Picture controls. Only use a Picture controls when you need a feature only it provides, such as DDE, graphics methods, or the ability to contain other controls.

Obviously you use less memory if you only load pictures as you need them at run time, rather than storing them in your application at design time. What may be less obvious is that you can share the same picture between multiple Picture controls, Image controls, and forms. If you use code like this you only maintain one copy of the picture:

```
Picture = LoadPicture("C:\Windows\Chess.BMP")
Image1.Picture = Picture        ' Use the same picture
Picture1.Picture = Picture      ' Use the same picture
```

Contrast that with this code, which causes three copies of the bitmap to be loaded, taking more memory and time:

```
Picture = LoadPicture("C:\Windows\Chess.BMP")
Image1.Picture = LoadPicture("C:\Windows\Chess.BMP")
Picture1.Picture = LoadPicture("C:\Windows\Chess.BMP")
```

Finally, try to use smaller picture data. Run Length Encoded (RLE) bitmaps can be several times smaller than their uncompressed counterparts, and aren't appreciably slower to load or display. The savings found by using metafiles can be even more significant -- ten times or more in some cases. The "Introduction to Visual Basic" application that ships with Visual Basic 2.0 uses metafiles for this reason.

When All Else Fails...

...be creative. There are a variety of extreme measures you can take when nothing else will work. Most of these are too complicated to relate here, but I can give a couple examples that should give you some ideas.

Multi-Apps

While Visual Basic 2.0 greatly increased the capacity limits over what version 1.0 provided, I've occasionally seen applications that have outgrown even the capabilities of 2.0. Rather than cutting back on the application, the developers of these applications have resorted to creating several applications that appear to all be part of one larger application. These can run and quit independently, allowing for extensive control over the size of the application at any one time. The various parts of this "multi-app" communicate with each other using DDE or -- when the amount of data is large and speed is not as critical

-- through shared files. While the amount of work involved is anything but trivial, and adds considerable complexity to the application, it makes truly enormous applications possible.

Lurking Apps

A variation on the above technique can be used when quick response is required and memory space is not at a premium. Instead of actually quitting, an application can just hide (or unload) all visible forms. The application then "lurks" until it is needed again (signaled via DDE, a timer, or by periodically checking for specific conditions), whereupon it seems to start up instantly by showing a form once again.

Use DLLs

Finally, you can go outside Visual Basic completely. By isolating the portion of your application that is slowest or uses the most memory, and rewriting that portion in another language and compiling it as a DLL, you can take advantage of some of the optimization techniques offered by other languages without giving up the productivity of Visual Basic. In fact, a quick look through the huge number of third-party products may turn up a DLL or custom control that does exactly what you need.

OLE Automation

Using CreateObject()

Opening an instance of an OLE aware application, always start it by using CreateObject(). ByUse this instead of ‘Shell’ or DDE. The reason for this can best be seen in the case of Microsoft Excel 5.0. Starting an instance of Excel using CreateObject() does not force Excel to load all of its add-ins. However, starting Excel using ‘Shell’ or DDE will result in Excel loading all of it’s add-ins. This can significantly increase the load time of Excel as well as causing a delay in program responsiveness that might be noticed by the end user.