

Native vs. Managed Code: GDI Performance

May 2008

Chris Tacke

OpenNETCF Consulting, LLC

Introduction

I recently saw a post in the public Compact Framework newsgroup¹ that is fairly typical. The poster stated that if you want speed and performance for graphics, you should use native (meaning C or C++) code. It's a general sentiment that I often read or hear – native code is faster than managed code for XYZ operation, so don't use managed code. In most cases these claims are being made by developers with little experience in managed development and they rarely have any sort of empirical data to back up their statement.

I knew that I already had some code for measuring GDI performance in native code that I wrote several years ago for testing and tuning a Windows CE display driver, so I decided I'd dig it up and then create a managed application that did the same operations and then compare the results of the two with real evidence and put to rest this silly myth once again.

Figure 1



Native Code Device Test

Before diving into managed code to see how it would perform, I needed to get the baseline test running in native code. The code I had was a pretty simple and straightforward test. It creates an off-screen square bitmap and device context (DC), fills it with a varying shade of grey, and then draws two intersecting green ellipses centered in that square (see Figure 1). The test then repeats the fill and ellipse drawing 10,000 times, each time changing the fill color and the ellipse boundaries to prevent the driver from being able to cache anything and to provide a nice psychedelic pattern to watch as the test runs.

Listing 1 – Native Code for Test Loop

```
...
screenDC = GetWindowDC(NULL);
bufferDC = CreateCompatibleDC(screenDC);
oldPen = (HPEN)SelectObject(bufferDC, CreatePen(PS_NULL, 0, NULL));
oldBrush = (HBRUSH)SelectObject(bufferDC, CreateSolidBrush(0,255,0));
hbufferBmp = CreateCompatibleBitmap(screenDC, BOX_WIDTH, BOX_HEIGHT);
oldObject = SelectObject(bufferDC, hbufferBmp);

rect.top = 0;
rect.left = 0;
rect.right = BOX_WIDTH;
rect.bottom = BOX_HEIGHT;

for(i = 0 ; i < GDI_ITERATIONS_PER_REPORT ; i++)
{
    width += xop;
    height += yop;
    if((width >= BOX_WIDTH) || (width <= 0))
        xop *= -1;

    if((height >= BOX_HEIGHT) || (height <= 0))
        yop *= -1;

    r += rop;
    g += gop;
    b += bop;

    if((r > 254) || (r < 1))
        rop *= -1;

    if((g > 254) || (g < 1))
        gop *= -1;

    if((b > 254) || (b < 1))
        bop *= -1;

    boxBrush = CreateSolidBrush(0, g, b);
    FillRect(bufferDC, &rect, boxBrush);
    DeleteObject(boxBrush);

    Ellipse(bufferDC, width, height, BOX_WIDTH - width, BOX_HEIGHT - height);
    Ellipse(bufferDC, height, width, BOX_WIDTH - height, BOX_HEIGHT - width);

    BitBlt(screenDC, left, top, left + BOX_WIDTH, top + BOX_HEIGHT,
           bufferDC, 0, 0, SRCCOPY);
}
...
```

The general code loop can be seen in Listing 1 (the full code accompanies this article).

¹ microsoft.public.framework.compactframework

In the tests I ran, the loop executed for 10,000 iterations. On the Windows Mobile 5.0 emulator that ships with Visual Studio 2008 running on my desktop PC this test executes at about 490 iterations per second (one iteration being a fill, two ellipse draws and a blit to the screen). On a Windows Mobile 5.0 Dell Axim x51 (PXA270 processor) it achieves about 250 iterations per second.

Compact Framework Device Test

The next step was to “port” the native code to managed code. The code is straightforward and most of the concepts in the native application have direct parallels in managed code. Listing 2 shows the essence of the managed version of the loop code (again, the full source accompanies this article).

When I ran this test against Compact Framework 3.5 it was rather surprised to see that on the Windows Mobile 5.0 emulator it only achieved 425 iterations per second (15% slower than the native code) and on the Axim it only achieved 210 iterations per second (19% slower than native code).

Listing 2 – Managed Code for Test Loop

```
...
Bitmap backBuffer = new Bitmap(BOX_WIDTH, BOX_HEIGHT);
Brush boxBrush;
Brush ellipseBrush = new SolidBrush(Color.Green);
Graphics screenGraphics = GDI.CreateGraphics(); // Graphics.FromHdc(screenDC);
Graphics backGraphics = Graphics.FromImage(backBuffer);

for (i = 0; i < GDI_ITERATIONS_PER_REPORT; i++)
{
    width += xop;
    height += yop;

    if ((width >= BOX_WIDTH) || (width <= 0))
        xop *= -1;
    if ((height >= BOX_HEIGHT) || (height <= 0))
        yop *= -1;

    r += rop;
    g += gop;
    b += bop;

    if ((r > 254) || (r < 1))
        rop *= -1;
    if ((g > 254) || (g < 1))
        gop *= -1;
    if ((b > 254) || (b < 1))
        bop *= -1;

    boxBrush = new SolidBrush(Color.FromArgb(r, g, b));
    backGraphics.FillRectangle(boxBrush, 0, 0, BOX_WIDTH, BOX_HEIGHT);
    boxBrush.Dispose();

    backGraphics.FillEllipse(ellipseBrush, width, height,
        BOX_WIDTH - 2 * width, BOX_HEIGHT - 2 * height);
    backGraphics.FillEllipse(ellipseBrush, height, width,
        BOX_WIDTH - 2 * height, BOX_HEIGHT - 2 * width);
}

screenGraphics.DrawImage(backBuffer, 0, 0);
}
...
```

Testing on the Desktop

To say the least, the results from the device tests surprised me. In my gut I had expected the managed code performance to be within 5% of the native code – 10% at the outside – but to see 15-20% was amazing. I then wondered if it was maybe something specific to the Compact Framework and that maybe we’d not see such a large disparity on the desktop. If I were asked to guess at the desktop results *before* seeing the device results I would have almost certainly said that the managed code results would be within 5% of the native, but after seeing the results from the device I had to actually run it and see.

I modified both the device and desktop code slightly to get single code bases for both languages that would compile for either target and then re-ran the tests. The managed code test achieved 825 iterations per second on the same PC that the emulator was running on – so roughly twice as fast. The native code, however, surprised me again and this time it was more “shock and awe” than mere surprise. The native desktop version of the test achieved 6200 iterations per second. Yes, 625% faster than the managed code on the same target.

Updating the Managed code

The managed code performed substantially worse than I expected (especially on the desktop). The question that still remains is “why?” What would account for this performance degradation? I would suspect that something like a call to `Graphics.FillRectangle` would just be a thin wrapper around a P/Invoke to the Win32 `FillRect` call so does that mean the performance problems we see are purely an artifact of the performance penalty of crossing the P/Invoke boundary²? If so we should be able to remove the calls to the framework Drawing classes, directly P/Invoke the GDI functions and get the same, or at least similar results.

I went back and reworked the managed code to be able to run using only framework-provided drawing operations within the test loop or to P/Invoke all of them. I then re-ran the test on the Windows Mobile 5.0 Emulator and the Axim and got 455 and 225 iterations per second respectively. That’s a 7% speed improvement over using purely the framework-provided GDI calls, and generally cut the difference between the native and managed tests in half.

This means that the penalty for crossing the P/Invoke boundary is definitely part of the performance difference between native and managed code in our tests, but it only accounts for about half of it. I can’t readily account for the other half, but if I were to make an educated guess it’s likely because the framework is doing parameter and bounds checking on all calls, which my P/Invokes aren’t doing, and it may be doing something different for data marshaling to the native side as well.

To be complete I then updated the code again so it would compile and run on the desktop as well as the device (the P/Invoke declarations are different between the two platforms). What really surprised me here was that using the P/Invokes on the desktop resulted in the managed test achieving 6150 iterations per second, or within 1% of the native test.

Obviously there’s something I don’t fully understand about GDI under the full framework, but the updated tests show that managed code certainly can perform just as fast as native code. Since we at OpenNETCF tend to focus on Windows CE and Windows Mobile, I decided to leave investigating the huge disparity on the desktop to a later time.

Conclusion

So I set out to gather some hard data and debunk a myth about managed code being slow but I ended up confirming that managed code is indeed slower, sometime *a lot* slower, than native code - at least for basic GDI operations. Below is a summary table of my results as well as results from another tech reviewer that ran tests on his devices for me (thanks to Peter Nowak).

² See my January 2008 article entitled “Performance Implication of Crossing the P/Invoke Boundary”
<http://community.opennetcf.com/articles/cf/archive/2008/01/31/performance-implications-of-crossing-the-p-invoke-boundary.aspx>

	Managed: Framework Calls	Managed: P/Invoke Calls	Native	Native is <i>n</i> % Faster (framework / P/invoke)
Desktop				
2.66GHz Intel Core2 Duo	890	6150	6200	597% / 1%
Device				
Windows Mobile 5.0 Emulator	425	455	490	15% / 8%
WinMo 5.0 Dell Axim x51 PXA270	210	225	250	19% / 11%
HTC TyTn* Samsung 2442A	70	74	75	7% / 1%
HTC P6300* Samsung 2442	73	76	77	6% / 1%
HTC Athena* PXA270	118	119	128	9% / 8%
HTC TyTn II* Qualcomm 7200	155	168	178	15% / 6%
HTC Sedna* Qualcomm 7200	121	129	131	9% / 2%
HTC Charmer* OMAP850	134	151	154	15% / 2%

* = Results from testing 1,000 iterations instead of 10,000

We can see that across the board managed code is always slower, but that with a little extra effort and code we can squeeze a bit more performance out of the managed code. If you really need that extra speed it's probably worth the effort. Making the changes on the desktop yields an enormous improvement, but again there could be something fundamentally wrong with the desktop test, so do some more investigation before just running wild with the numbers I came up with.

So does this mean that native code is "superior" to managed code and that you shouldn't still consider using managed code for your development projects? I'm still going to adamantly answer "no." Managed code is not the answer to all development problems, but neither is native code. Each has their place, their strengths and their weaknesses.

Consider the rapid development nature of managed code and the nice variety of test and continuous integration tools available for managed code developers. It's very difficult to make a business case that something like a typical enterprise data collection and reporting application should be developed using anything but managed code. It will get you to market faster, with lower cost, more features and less support headache. What's not to like?

Similarly you'd be hard pressed to make a reasonable case that something like an Ethernet driver should be written in anything but native code. The more difficult decisions are in the grey area between. In

these cases you have to weigh the application requirements against your schedule, budget and available resources. Sure, the simple tests I ran here show that for when it comes to raw throughput for simple GDI operations, native code is faster but the human eye doesn't really need anything faster than 30 frames per second to see it as smooth – so is it really a problem? What *else* is your application going to do?

If you're making a flight simulator, then your app is all about drawing and sure – native code would be the choice, but what about a card game? How about a drawing program or a text reader? The point is that only you can decide which is better for what you need to get done. Hopefully this article, instead of providing opinion and conjecture, provides you another piece of real information with hard data that can help you make that decision.