Optimizing BitBlt by generating code on the fly

devblogs.microsoft.com/oldnewthing/20180209-00

February 9, 2018

Raymond Chen

The initial implementation of the `BitBlt` function in 16-bit Windows didn't have any special tricks. It was static code that supported the sixteen raster operations that involve a source and destination.

The second version of `BitBlt` generated code on the fly. Specifically, the `BitBlt` function generated code onto the stack which performed the block transfer with the appropriate operation, and then called the code as a subroutine.

This was clearly the days before Data Execution Prevention (DEP).

Internally, the function that did this was called `CBlt`, short for "compiled block transfer." The generated code followed a template, hard-coding the array bounds and bitmap stride into the generated code, thereby transforming variables into constants, which avoided having to consume a register (or worse, accessing memory) to check the value.

The code generator emitted code to loop over the source and destination bitmaps, taking care to check for overlap between the source and destination and looping in the correct direction accordingly. The code generator also had to generate code to handle the so-called "phase mismatch" in the case where the source and destination do not start at the same bit offset within the starting byte. And of course to handle the case where the starting or ending pixels are not on a byte boundary. And then there was code to handle the case of interlaced displays, where the way to move from one scan line to the next depends on whether you're starting from an odd-numbered or even-numbered scan line.[1] Basically, all the stuff that you need to worry about when doing `BitBlt`, but instead of doing it, you are generating code that does it.

Inside the loop body, the code generator inserted a code fragment to perform the block transfer operation. For example, if the operation was `SRCERASE`, then the generated code would do something like

```
; By this point, the source is in AL
; and the destination is in ES:[DI].

not al
and al, es:[di]

; On exit from the fragment, the result is in AL.
```

The fragment is where the donuts are made. All the rest of the generated code is just scaffolding so we can get to this point. And as you can see, the fragment is usually rather anticlimactic.

The code generator had a table of sixteen fragments, so it knew what instructions to place inside the loop body.

The third version of `BitBlt`, known as SuperBlt at the time, extended its support to three-parameter raster operations (source, pattern, and destination). There are 256 possible operations, but to avoid exploding the number of fragments, there was some consolidation. For example, if two raster operations are the same except that one is the bitwise inverse of the other, then the same fragment was used for both, and the compiler appended a `not al` to one of them.

The fragment table also noted which inputs were required by the operation. For example, the `DSTINVERT` operation doesn't use the source at all, so the code generator can avoid generating the code to loop through the source bytes and load them into the `al` register. No point calculating values you're never going to use.

The result of all this compilation was around 120 instructions of machine code to perform a block transfer operation. Each of these custom-generated subroutines handled a particular bitmap size, overlap scenario, phase match, block transfer operation, and interlace state.

The fourth version of `BitBlt` added support for blitting between color bitmaps and monochrome bitmaps. So now you had color conversion as another input to the code generator.

In Windows 3.0, the fifth version added support for bitmaps larger than 64KB. The code generator took advantage of 32-bit registers so that it could index into the entire memory block at once, instead of having to break it up into 64KB pieces.

In Windows 95, the code generator got crazy and used the `esp` register as a general-purpose register. The 80386 has only eight 32-bit registers, so gaining an extra register was a big help. The code doesn't actually use the stack, so the fact that the `esp` register doesn't point to the stack isn't a problem. (Note that normal Win32 code can't get away with this trick

because the stack pointer must remain valid for stack unwinding purposes. But this was special code running under special conditions, and it was in cahoots with the kernel so the kernel didn't freak out when it saw this wacko stack.)

Uh oh, but this means that you can't use the `esp` register to access your local variables. No problem! We'll run the code on a custom stack, too, so that our local variables are at fixed offsets relative to the stack selector register.

Nearly all of GDI was written in assembly language in versions of Windows up to and including the Windows 95 family. In that era, being able to not only read but also write assembly language was a core developer skill.

Bonus reading: The idea of generating block transfer code on the fly has been around for a while. (If impatient, skip to the bottom of page 43.)

[1] The way the code managed this was rather clever. It calculated the stride to go from an odd-numbered scan line to an even-numbered scan line, and then it calculated the stride to go from an even-numbered scan line to an odd-numbered scan line. It then xor'd the two values together to create a toggle value. After each scan line was complete, the current stride was applied, and then the stride was xor'd with the toggle value. This causes the stride to flip back and forth between the two desired values.

Raymond Chen

Follow