The case of the image that came out horribly slanted: Negative stride

devblogs.microsoft.com/oldnewthing/20240906-00

September 6, 2024

Last time, we looked at how to account for stride when passing a block of pixel data to GDI. But we failed to take into account the case where the stride is negative.

First of all, what does negative stride even mean?

Recall that the stride is the value to add (in bytes) to get from the start of one row of pixels to the start of the next row of pixels. But there's nothing in the definition that prohibits the value from being negative. A negative stride means that the next row of pixels is stored *before* the current row of pixels.

In other words, the rows of the bitmap are sequenced in reverse.

What "reverse" means depends on what each framework considers to be "forward".

GDI DIB sections are normally bottom-up: The first row in memory represents the bottom row of the bitmap. But you can also specify top-down DIBs, in which case the first row in memory represents the top row of the bitmap. docs.microsoft.com has a comparison of the two, including diagrams showing both the up-and-down-ness and the padding encompassed by the stride.

A negative stride indicates that the in-memory layout puts the rows in reverse order, and the pointer to the first row is actually near the *end* of the bitmap pixel memory, and the stride tells you how far *backward* to go to get to the next row.

Here's a detailed memory layout of a bitmap with positive stride. Assume memory is laid out top to bottom, with lower addresses at the top of the diagram and higher addresses at the bottom.

bitmap pixels

p	➤	• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p + *stride*		• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p + 2 × *stride*		• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p + 3 × *stride*		• → • → • → • → • → • → • → • → • →

And here’s what you get with a negative stride:

		bitmap pixels
p + 3 × *stride*		• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p + 2 × *stride*		• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p + *stride*		• → • → • → • → • → • → • → • → • →
		← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ← ←
p	➤	• → • → • → • → • → • → • → • → • →

From the diagram, we can see how to convert a bitmap with negative stride to one with positive stride or vice versa: Find the last row of the bitmap by adding (*height* − 1) × *stride* to the first row. Use that as the first row of the corresponding bitmap with the opposite-sign stride. The result is an upside-down bitmap, which you can then correct with a flip operation (say by pre-flipping the source, post-flipping the destination, or flipping as part of the block transfer).

Now, it so happens that the stride of a Direct2D bitmap is an unsigned value, so it can never be negative. This means that our D2D-to-GDI converter doesn’t have to worry about negative stride. (Wipes brow in relief.) However, other graphics frameworks do support negative stride, and a general solution would have to take that into account. Let’s add negative stride support to our converter, even though it’s not actually possible for Direct2D, just to show how to do the math.

```
HRESULT D2D1BitmapToGdiBitmap(
    ID2D1Bitmap1* d2dBitmap,
    HDC hdc,
    HBITMAP* result)
{
    *result = nullptr;

    // Verify that it's in PARGB format
    assert(d2dBitmap->GetPixelFormat().format ==
            DXGI_FORMAT_B8G8R8A8_UNORM &&
           d2dBitmap->GetPixelFormat().alphaMode ==
            DXGI_ALPHA_MODE_PREMULTIPLIED);

    auto size = d2dBitmap->GetPixelSize();

    // Get a pointer to the pixel memory for reading.
    D2D1_MAPPED_RECT rect;
    RETURN_IF_FAILED(d2dBitmap->Map(D2D1_MAP_OPTIONS_READ, &rect);

    // Make sure we unmap even on early exit.
    auto unmap = wil::ScopeExit([&]() { d2dBitmap->Unmap(); });

    // Create a GDI HBITMAP of matching size.
    wil::unique_hbitmap gdiBitmap(
        CreateCompatibleBitmap(hdc, size.width, size.height));
    RETURN_IF_NULL_ALLOC(gdiBitmap);

    // D2D bitmaps are top-down. GDI supports both top-down and bottom-up:
    // If the height is positive, then the bitmap is bottom-up.
    // If the height is negative, then the bitmap is top-down.
    BITMAPINFO bmi{};
    bmi.bmiHeader.biSize = sizeof(bmi.bmiHeader);
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 32; // B8G8R8A8_UNORM is 32bpp
    bmi.bmiHeader.biCompression = BI_RGB;
    BYTE* bits;
    if (rect.pitch >= 0) { // always true for D2D since pitch is unsigned
        // Specify a top-down bitmap (negative height)
        bmi.bmiHeader.biHeight = -static_cast<LONG>(size.height);
        bmi.bmiHeader.biWidth = rect.pitch / 4; // 4 bytes per pixel
        bmi.bmiHeader.biHeight = -static_cast<LONG>(size.height);
        bits = rect.bits;
    } else {
        // Specify a bottom-down bitmap (positive height)
        bmi.bmiHeader.biHeight = +static_cast<LONG>(size.height);
        bmi.bmiHeader.biWidth = (-rect.pitch) / 4; // 4 bytes per pixel
        bmi.bmiHeader.biHeight = static_cast<LONG>(size.height);
        bits = rect.bits + (size.height - 1) * rect.pitch;
    }

    RETURN_IF_WIN32_BOOL_FALSE(SetDIBits(hdc, gdiBitmap.get(),
        0, size.height, bits, &bmi; DIB_RGB_COLORS));
    *result = gdiBitmap.release();
    return S_OK;
}
```