The program running in a console decides what appears in that console

devblogs.microsoft.com/oldnewthing/20101115-00

November 15, 2010

Raymond Chen

James Risto asks, "Is there a way to change the behavior of the CMD.EXE window? I would like to add a status line."

The use of the phrase "the CMD.EXE window" is ambiguous. James could be referring to the console itself, or he could be referring to the CMD.EXE progarm.

The program running in a console decides what appears in the console. If you want to devote a line of text to a status bar, then feel free to code one up. But if you didn't write the program that's running, then you're at the mercy of whatever that program decided to display.

Just to show that it can be done, here's a totally useless console program that contains a status bar.

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <strsafe.h> // for StringCchPrintf
void DrawStatusBar(HANDLE hScreen)
{
 CONSOLE_SCREEN_BUFFER_INFO sbi;
 if (!GetConsoleScreenBufferInfo(hScreen, &sbi)) return;
 TCHAR szBuf[80];
 StringCchPrintf(szBuf, 80, TEXT("Pos = %3d, %3d"),
                 sbi.dwCursorPosition.X,
                 sbi.dwCursorPosition.Y);
 DWORD dwWritten;
 COORD coDest = { 0, sbi.srWindow.Bottom };
 WriteConsoleOutputCharacter(hScreen, szBuf, lstrlen(szBuf),
    coDest, &dwWritten);
}
```

Our lame-o status bar consists of the current cursor position. Notice that the console subsystem does not follow the GDI convention of endpoint-exclusive rectangles.

```
int __cdecl wmain(int argc, WCHAR **argv)
{
 HANDLE hConin = CreateFile(TEXT("CONIN$"),
                            GENERIC_READ | GENERIC_WRITE,
                            FILE_SHARE_READ | FILE_SHARE_WRITE,
                            NULL, OPEN_EXISTING, 0, NULL);
 if (hConin == INVALID_HANDLE_VALUE) return 1;
 HANDLE hConout = CreateFile(TEXT("CONOUT$"),
                       GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL, OPEN_EXISTING, 0, NULL);
 if (hConout == INVALID_HANDLE_VALUE) return 1;
```

We start by getting the handles to the current console. Since we are a fullscreen program, we don't rely on stdin and stdout. (How do you position the cursor on a redirected output stream?)

```
 HANDLE hScreen = CreateConsoleScreenBuffer(
                       GENERIC_READ | GENERIC_WRITE,
                       0, NULL, CONSOLE_TEXTMODE_BUFFER, NULL);
 if (!hScreen) return 1;
 SetConsoleActiveScreenBuffer(hScreen);
```

We create a new screen buffer and switch to it, so that our work doesn't disturb what was previously on the screen.

```
 DWORD dwInMode;
 GetConsoleMode(hConin, &dwInMode);
```

We start by retrieving the original console input mode before we start fiddling with it, so we can restore the mode when our program is finished.

```
 SetConsoleCtrlHandler(NULL, TRUE);
 SetConsoleMode(hConin, ENABLE_MOUSE_INPUT |
                        ENABLE_EXTENDED_FLAGS);
```

We set our console control handler to `NULL` (which means "don't terminate on Ctrl+C") and enable mouse input on the console because we're going to be tracking the mouse position in our status bar.

```
 CONSOLE_SCREEN_BUFFER_INFO sbi;
 if (!GetConsoleScreenBufferInfo(hConout, &sbi)) return 1;
 COORD coDest = { 0, sbi.srWindow.Bottom - sbi.srWindow.Top };
 DWORD dw;
 FillConsoleOutputAttribute(hScreen,
     BACKGROUND_BLUE |
     FOREGROUND_BLUE | FOREGROUND_RED |
     FOREGROUND_GREEN | FOREGROUND_INTENSITY,
     sbi.srWindow.Right - sbi.srWindow.Left + 1,
     coDest, &dw);
```

We retrieve the screen buffer dimensions and draw a blue status bar at the bottom of the screen. Notice that the endpoint-inclusive rectangles employed by the console subsystem result in what look like off-by-one errors. The bottom line of the screen is `Bottom - Top`, which in an endpoint-exclusive world would be the height of the screen, but since the rectangle is endpoint-inclusive, this is actually the height of the screen *minus 1*, which puts us at the bottom line of the screen. Similarly `Right - Left` is the width of the screen *minus 1*, so we have to add one back to get the width.

```
 DrawStatusBar(hScreen);
```

Draw our initial status bar.

```
 INPUT_RECORD ir;
 BOOL fContinue = TRUE;
 while (fContinue && ReadConsoleInput(hConin, &ir, 1, &dw)) {
  switch (ir.EventType) {
  case MOUSE_EVENT:
   if (ir.Event.MouseEvent.dwEventFlags & MOUSE_MOVED) {
     SetConsoleCursorPosition(hScreen,
         ir.Event.MouseEvent.dwMousePosition);
     DrawStatusBar(hScreen);
   }
   break;
  case KEY_EVENT:
   if (ir.Event.KeyEvent.wVirtualKeyCode == VK_ESCAPE) {
    fContinue = FALSE;
   }
   break;
  }
 }
```

This is the console version of a “message loop”: We read input events from the console and respond to them. If the mouse moves, we move the cursor to the mouse position and update the status bar. If the user hits the Escape key, we exit the program.

```
 SetConsoleMode(hConin, dwInMode);
 SetConsoleActiveScreenBuffer(hConout);
 return 0;
}
```

And when the program ends, we clean up: Restore the original input mode and restore the original screen buffer.

If you run this program, you’ll see a happy little status bar at the bottom whose contents continuously reflect the cursor position, which you can move by just waving the mouse around.

If you want a status bar in your console program, go ahead and draw it yourself. Of course, since it's a console program, your status bar is going to look console-y since all you have to work with are rectangular character cells. Maybe you can make use of those fancy line-drawing characters. Party like it's 1989!

Raymond Chen

Follow