Pseudo-random ramblings about programming and other geeky stuff

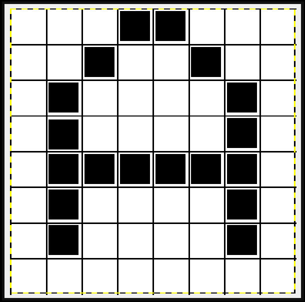
Friday, 18 April 2014

Low-level Graphics on Raspberry Pi (text)

Someone asked me how to go about rendering text - as we are using a framebuffer in graphics mode and can only plot pixels, text output does not come for free and has to be done extending the pixel plotting.

There are basically two different ways to render text: raster fonts and vector fonts. A font is a collection of character representations or 'images' of characters (letters and symbols). As should be obvious: raster font represents the characters as raster images - vector font represents them as collections of lines/polygons. Using the utility functions presented in previous posts, it would be possible to choose either approach.

Let's take a look at simple monospace raster fonts. Each character would be a block of pixels: some filled - some empty. For example a simple version of the letter 'A' as a 8x8 pixel image could be:



...the image occupies 64 pixels: 8 across and 8 down - the pixels representing the character outline would be filled (in this case black). A naive implementation could just plot the pixels:

```
void draw_char(char a, int x, int y, int c) {
  switch (a) {
    case 'A':
      // top row
      put_pixel(x + 3, y, c);
      put_pixel(x + 4, y, c);
      // next row
      put_pixel(x + 2, y, c);
      put_pixel(x + 5, y, c);
    ...
}
```

...which obviously would not be very efficient coding wise.

A better solution would be to use a pixel map - a raster image - and copy this image to the desired spot on the screen. The pixel map could be drawn in a drawing program like

Blog Archive

- ► 2016 (6)
- ► 2015 (3)
- ▼ 2014 (9)
  - ► September (1)
  - ▼ April (3)
    - Low-level Graphics on Raspberry Pi (text)
    - Low-level Graphics on Raspberry Pi (shapes)
    - Rant "javac: invalid target release: 1.6"
  - ► March (5)
- ► 2013 (9)
- ► 2012 (2)

Code Repository

- Low-level Graphics on RPi

Discussion

- Low-level Graphics on RPi
- Python Programming on RPi
- Java Programming on RPi

Links

- Raspberry Pi
- Python

GIMP, saved to a file, loaded into memory in our program and the character pixel blocks copied from it to the screen. This might be the best way to go, but dealing with image file formats is another story... Alternative way would be to define the pixel maps in code - something along:

```
char A[] = {
    0, 0, 0, 1, 1, 0, 0, 0,
    0, 0, 1, 0, 0, 1, 0, 0,
    0, 1, 0, 0, 0, 0, 1, 0,
    0, 1, 0, 0, 0, 0, 1, 0,
    0, 1, 1, 1, 1, 1, 1, 0,
    0, 1, 0, 0, 0, 0, 1, 0,
    0, 1, 0, 0, 0, 0, 1, 0,
    0, 0, 0, 0, 0, 0, 0, 0
};
```

Now how to avoid the long switch statement... Typically computer systems would use a character encoding scheme like ASCII, which in fact is used by the C language in Linux by default. The scheme defines numerical codes for characters: for example space character is 32 (decimal) and the exclamation mark 33 - a quick test to see this in effect would be:

```
  printf("%c", 33);
  printf("\n");
  printf("%d", ' ');
```

...which should print out the exclamation mark and number 32. As this is pretty much a standard and, if leaving out the first 32 non-printable control characters, quite convenient for our purpose, we could define all character pixel maps in one two dimensional array:

```
#define FONTW 8
#define FONTH 8

char fontImg[][FONTW * FONTH] = {
    { // ' ' (space)
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0
    },
    { // !
            0, 0, 0, 1, 0, 0, 0, 0,
            0, 0, 0, 1, 0, 0, 0, 0,
            0, 0, 0, 1, 0, 0, 0, 0,
            0, 0, 0, 1, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 1, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0
    },
    ...
}
```

...where the outer array size is left to be decided (by the compiler) based on the actual entries defined, so we can do some of the first characters for testing and leave rest for later.

So we have the characters defined in the ASCII order, but leaving out the first 32 and we can get the appropriate pixel map using the C language `char/integer` interchangeability:

```
    char c = '!';
    char *img = fontImg[c - 32];
```

...now the `imgc` variable would point to the 8x8 bytes pixel map for the exclamation mark.

Drawing a single character using the `put_pixel` function familiar form before could be implemented as:

```
        // loop through pixel rows
        for (y = 0; y < FONTH; y++) {
            // loop through pixel columns
            for (x = 0; x < FONTW; x++) {
                // get the pixel value
                char b = img[y * FONTW + x];
                if (b > 0) { // plot the pixel
                    put_pixel(textX + i * FONTW + x, textY + y,
                              textC);
                }
                else {
                    // leave empty (or maybe plot 'text backgr color')
                }
            } // end "for x"
        } // end "for y"
```

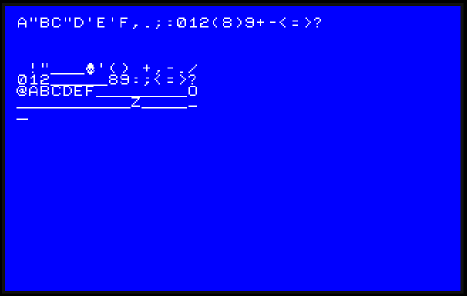
And drawing a string (an array of characters):

```
    // loop through all characters in the text string
    l = strlen(text);
    for (i = 0; i < l; i++) {
        // get the 'image' index for this character
        int ix = font_index(text[i]);
        // get the font 'image'
        char *img = fontImg[ix];
        // loop through pixel rows
        ...
        } // end "for y"
    } // end "for i"
```

Putting all this together with setting up the graphics mode etc (full code in GitHub - note the .c and .h both required), compiling with `gcc -o fbtestfnt fbtestfnt.c` and running with a command like:

```
$ ./fbtestfnt "A\"BC\"D'E'F,.;:012(8)9+-<=>?"
```

...should give us a display similar to:



```
A"BC"D'E'F,.;:012(8)9+-<=>?


 !"___●'() +,-./
012_____89:;<=>?
@ABCDEF________O
__________Z_____-
_
```

As one can see, the .h file contains only part of the full character set - rest are left as an exercise for the reader ;) Also obviously the 8x8 pixel font is quite small for higher resolutions - the code should provide for easy customisation to support more character sizes and/or multi-colored (possibly anti-aliased) fonts ...or to get creative with additional characters (127+ indexes 'extended ASCII' or maybe PETSCII ...or Easter eggs :P).

[Continued in next part Palette]

Posted by Unknown at 18:34

Labels: C, graphics, Linux, Raspberry Pi

No comments:

Post a Comment

Note: only a member of this blog may post a comment.

Enter your comment...

Comment as: Lhunden (Googl ▾ Sign out

Publish Preview ☐ Notify me

Subscribe to: Post Comments (Atom)