Pseudo-random ramblings about programming and other geeky stuff

Wednesday, 3 April 2013

Low-level Graphics on Raspberry Pi (part eight)

Last time, we looked at the differences between display modes regarding color depths. Another attribute of a display mode is the resolution - by default, RPi uses the full HD resolution 1920x1080 as we saw in the first part (of course depending on the actual display attached).

Basically this can be changed to (almost) anything if needed/wanted - one obvious reason would be to limit the number of pixels RPi has to handle. You might have noticed that the color gradients in the previous example appear on the screen somewhat slowly (especially if using the full HD resolution)?

Let's experiment a bit... As briefly mentioned before, there is an utility for changing the frame buffer parameters called `fbset` (see man fbset). Now if we assume the original resolution is the full HD, running `fbtest7` would look like:

```
$ ./fbtest7
The framebuffer device was opened successfully.
Original 1920x1080, 16bpp
$
```

...and the image appears like a roller blind drawn down. Now try the following sequence of commands (presented with output, so type only the text on lines starting with $ sign, not including the sign):

```
$ fbset -g 960 540 960 540 16
$ ./fbtest7
The framebuffer device was opened successfully.
Original 960x540, 16bpp
$ fbset -g 640 360 640 360 16
$ ./fbtest7
The framebuffer device was opened successfully.
Original 640x360, 16bpp
$ fbset -g 448 256 448 256 16
$ ./fbtest7
The framebuffer device was opened successfully.
Original 448x256, 16bpp
$
```

...with about the same image appearing faster and faster. Also note that even though we change the resolution to smaller and smaller, the RPI VideoCore GPU scales the image up to fill the entire display - hardware accelerated up-scaling for free ;)

The effect on the image quality may not be that apparent with the color gradient example, so let's try something possibly a bit more descriptive:

```
#include <unistd.h>
#include <stdio.h>
```

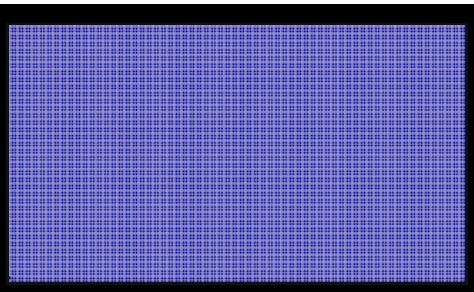
```c
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>

// 'global' variables to store screen info
char *fbp = 0;
struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;

// helper function to 'plot' a pixel in given color
void put_pixel(int x, int y, int c)
{
    // calculate the pixel's byte offset inside the buffer
    unsigned int pix_offset = x + y * finfo.line_length;

    // now this is about the same as 'fbp[pix_offset] = value'
    *((char*)(fbp + pix_offset)) = c;

}

// helper function for drawing - no more need to go mess with
// the main function when just want to change what to draw...
void draw() {

    int x, y;

    // fill the screen with blue
    memset(fbp, 1, vinfo.xres * vinfo.yres);

    // white horizontal lines every 10 pixel rows
    for (y = 0; y < (vinfo.yres); y+=10) {
        for (x = 0; x < vinfo.xres; x++) {
            put_pixel(x, y, 15);
        }
    }

    // white vertical lines every 10 pixel columns
    for (x = 0; x < vinfo.xres; x+=10) {
        for (y = 0; y < (vinfo.yres); y++) {
            put_pixel(x, y, 15);
        }
    }

    int n;
    // select smaller extent
    // (just in case of a portrait mode display)
    n = (vinfo.xres < vinfo.yres) ? vinfo.xres : vinfo.yres;
    // red diagonal line from top left
    for (x = 0; x < n; x++) {
        put_pixel(x, x, 4);
    }

}

// application entry point
int main(int argc, char* argv[])
{

    int fbfd = 0;
    struct fb_var_screeninfo orig_vinfo;
    long int screensize = 0;


    // Open the file for reading and writing
    fbfd = open("/dev/fb0", O_RDWR);
    if (!fbfd) {
      printf("Error: cannot open framebuffer device.\n");
      return(1);
    }
    printf("The framebuffer device was opened successfully.\n");

    // Get variable screen information
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
```

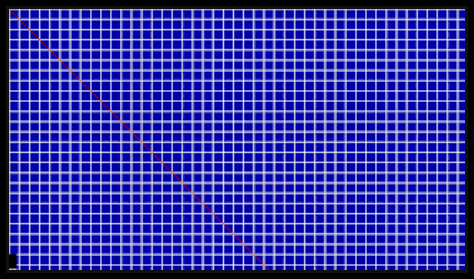
```
      printf("Error reading variable information.\n");
    }
    printf("Original %dx%d, %dbpp\n", vinfo.xres, vinfo.yres,
       vinfo.bits_per_pixel );

    // Store for reset (copy vinfo to vinfo_orig)
    memcpy(&orig_vinfo, &vinfo, sizeof(struct fb_var_screeninfo));

    // Change variable info - force 8 bit
    vinfo.bits_per_pixel = 8;
    if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo)) {
      printf("Error setting variable information.\n");
    }

    // Get fixed screen information
    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
      printf("Error reading fixed information.\n");
    }

    // map fb to user mem
    screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
    fbp = (char*)mmap(0,
              screensize,
              PROT_READ | PROT_WRITE,
              MAP_SHARED,
              fbfd,
              0);

    if ((int)fbp == -1) {
        printf("Failed to mmap.\n");
    }
    else {
        // draw...
        draw();
        sleep(5);
    }

    // cleanup
    munmap(fbp, screensize);
    if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &orig_vinfo)) {
        printf("Error re-setting variable information.\n");
    }
    close(fbfd);

    return 0;

}
```

Save that code as `fbtest8.c`, compile with `make fbtest8` and execute the following command sequence:

```
$ fbset -g 1920 1080 1920 1080 16
$ ./fbtest8
The framebuffer device was opened successfully.
Original 1920x1080, 16bpp
$ fbset -g 960 540 960 540 16
$ ./fbtest8
The framebuffer device was opened successfully.
Original 960x540, 16bpp
$ fbset -g 640 360 640 360 16
$ ./fbtest8
The framebuffer device was opened successfully.
Original 640x360, 16bpp
$ fbset -g 448 256 448 256 16
$ ./fbtest8
The framebuffer device was opened successfully.
Original 448x256, 16bpp
$
```

...which should produce images like the ones here (the first one at 1920x1080 and the second at 448x256):





Of course the resolution can be changed programmatically too (no need to leave it to the end-user) - change the variable info setting lines in the `main` to:

```
// Change variable info - force 8 bit and resolution
vinfo.bits_per_pixel = 8;
vinfo.xres = 320;
vinfo.yres = 240;
vinfo.xres_virtual = vinfo.xres;
vinfo.yres_virtual = vinfo.yres;
if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo)) {
  printf("Error setting variable information.\n");
}
```

Save as `fbtest8b.c`, compile with `make fbtest8b` and execute with `./fbtest8b` ...and you should get the same image with even bigger pixels ...centered horizontally at the middle of the display with black borders at sides (if you have the typical widescreen display in landscape mode).

So it seems that VC GPU scales to 'best fit' keeping pixels square - if the specified resolution conforms to the display aspect ratio, the image will fill the entire display - if it does not, there will be black borders at either left&right or top&bottom. You might want to experiment with different `xres` and `yres` values (for example 512x256, more ideas maybe here) to find a suitable resolution for your use-case - bearing in mind that the GPU seems (based on the hello_pi examples and experimenting) to expect the xres to be a multiple of 32 and yres of 16 - if not, there may be some side effects... For example, for the 'fire effect' in the screenshot in part two, I had to go as low as 320x240 to get a bearable frame-rate.

Code available also in GitHub

[Continued in part X]

Posted by Unknown at 11:26

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

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)