- Frame Buffer, Color Model and Pixels -

Goal:
During this exercise you will learn to write pixels according to the color model directly into the frame buffer device to create color bar patterns. You can draw simple shapes, even with alphblending.

Exercise 12.1: Our first pixels
Create a new C-project in Eclipse (either as Managed-Make project → then you will need a separate project for each little example; or you work with a Makefile project → then you can manage different code samples in the same project, but you have to create your own Makefile target rules, based on the template provided).
Then copy the example code **`fb6test_balken.c`** into your project. Build the executable both for the host and the target.

Note: Running the X11 window system, access to the frame buffer device is blocked. To avoid conflicts with X11, we switch to a virtual console to run our graphics programs. This can be done with Ctl+Alt+F1. Switching back to X11 is done by Ctl+Alt+F7.

First test your color model both on host and target using:

```
sudo fbset
```

(If needed, you first have to install the package **`fbset`**). What is the outcome? What does it mean for the **`put_pixel()`** routine?)
After switching to a virtual console, we run the executable (manually) and check the result:

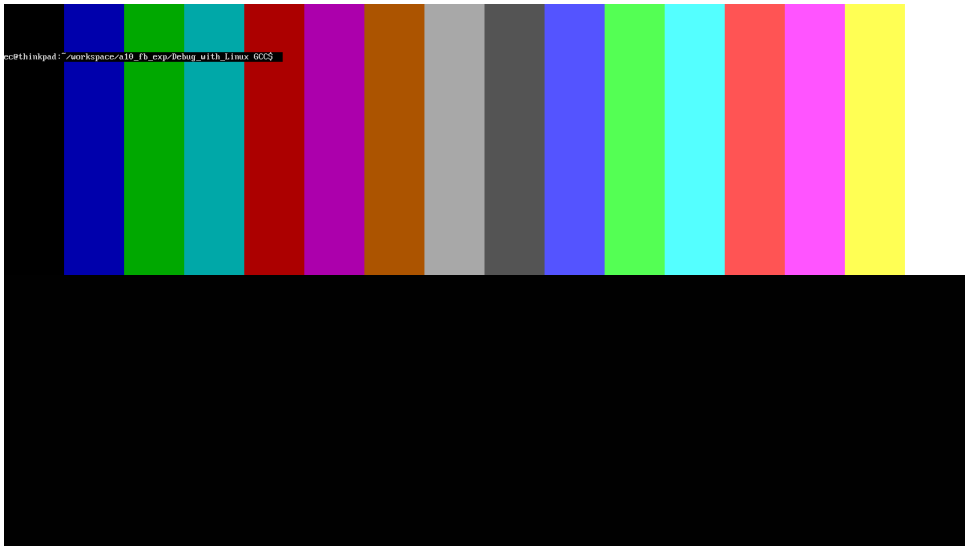

Fig.: Color bars

Exercise 12.2 (Bonus): Using Dithering to improve Graphics Quality
Copy your code into a new project (or *.c-file). Modify the **`draw()`** routine to render a pattern of horizontal, single-colored stripes. Start e.g. at y=0 with a full red color, which step-by-step get darker with increasing values of y to reach full black at the bottom of the screen. How many stripes do you notice in the RGB565 color model? What does it mean for high-quality graphics?
Now you apply dithering to your **`draw()`** routine avoid visible color transitions. First step is to render the complete image without dithering. In a second step, the contents of the frame buffer memory is dithered using a new routine **`draw_floydsteinberg()`**. (s. https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering , s. https://en.wikipedia.org/wiki/Dither). Create the routine and evaluate the quality.

Exercise 12.3: And now – our own Shapes
Copy the results from exercise 12.1 into a new project (or *.c-file). Extend the **`draw()`** routine into a routine **`draw_shape(x,y, shape, size, r,g,b)`**, which will render the shape <shape> (square, triangle, circle, etc.) at position x,y using color r,g,b. Build and then execute on host and target.

Exercise 12.4: There is more behind…
To get transparency effects (alphablending), we read the pixel value of each position (background color) before we overwrite it with a new value (foreground color). We get a 50% transparency of our shapes, if we mix the color of the shape with its background color using a 1:1 ratio (blending, linear interpolation).
Copy the result of 12.3 into a new project (or *.c-file). Then create a simple routine **`draw_background()`** to render a black/grey/white pattern into the frame buffer.
Now you extend your **`draw_shape()`** routine with a new parameter alpha (0.0 … 1.0). Extend your **`put_pixel()`** routines to first read the old pixel values and then blend it with the new color. Note: this has to be done independently for each of the color channels R,G,B.

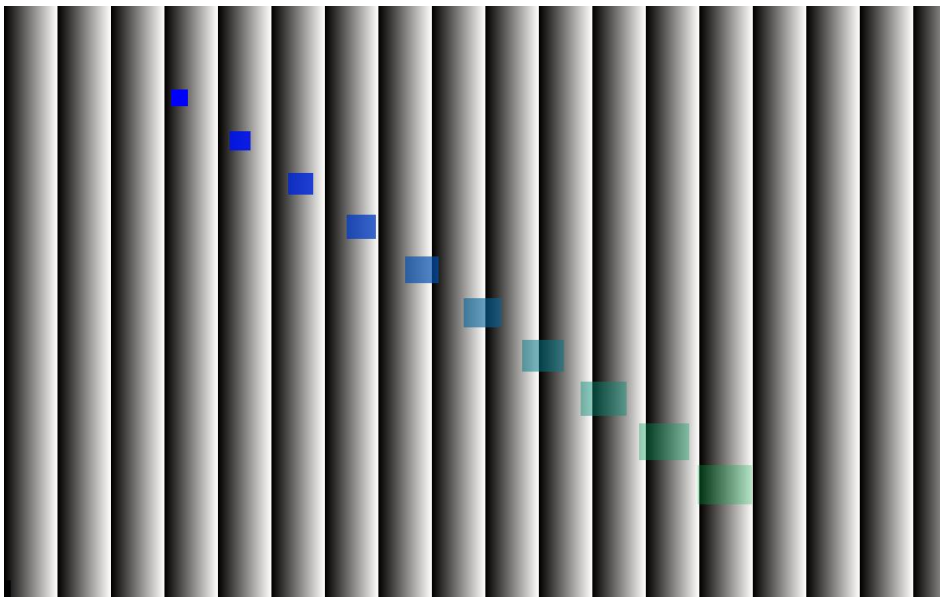

Fig.: Rectangular Shape with changing Colors and increasing Transparency on top of a Gray Scale