ECE 243S - Computer Organization
March 2024
Lab 7

Introduction to Graphics and Animation

Due date/time: During your scheduled lab period in the week of March 11, 2024.

Learning Objectives

The goal of this lab is to learn how to display images and perform animation, written in the C language for the DE1-SoC Computer. Graphics can be displayed by connecting a VGA monitor to the *video-out* port on the board. You can also use CPUlator to develop and debug graphics code - graphics that would normally appear on a VGA display are instead rendered inside a sub-window, labeled `VGA pixel buffer`, within CPUlator. This subwindow can be set to various sizes and it can also be "popped out" of the main browser window, if desired.

What To Submit

You should hand in the following files prior to the end of your lab period: The C code for Parts I,II, III and IV in the files `part1.c`, `part2.c`, and `part3.c`.

Make sure you've read the companion document "Introduction the VGA Controller - The Frame Buffer and Character Buffer"

Part I

In this part you will learn how to implement a simple line-drawing algorithm, in the C programming language. Drawing a line on a screen requires coloring pixels between two points (x_1, y_1) and (x_2, y_2), such that the pixels represent the desired line as closely as possible. Consider the example in Figure 1, where we want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares in the figure represent the location and size of pixels on the screen. As indicated in the figure, we cannot draw the line precisely—we can only draw a shape that is similar to the line by coloring the pixels that fall closest to the line's ideal location on the screen.

We can use algebra to determine which pixels to color. This is done by using the end points and the slope of the line. The slope of our example line is $slope = (y_2 - y_1)/(x_2 - x_1) = 4/11$. Starting at point $(1, 1)$ we move along the x axis and compute the y coordinate for the line as follows:

$$y = y_1 + slope \times (x - x_1)$$

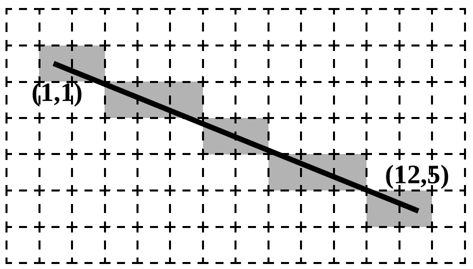

Figure 1: Drawing a line between points $(1, 1)$ and $(12, 5)$.

Thus, for column $x = 2$, the y location of the pixel is $1 + \frac{4}{11} \times (2 - 1) = 1\frac{4}{11}$. Since pixel locations are defined by integer values we round the y coordinate to the nearest integer, and determine that in column $x = 2$ we should color the pixel at $y = 1$. For column $x = 3$ we perform the calculation $y = 1 + \frac{4}{11} \times (3 - 1) = 1\frac{8}{11}$, and round the result to $y = 2$. Similarly, we perform such computations for each column between x_1 and x_2.

The approach of moving along the x axis has drawbacks when a line is steep. A steep line spans more rows than it does columns, and hence has a slope with absolute value greater than 1. In this case our calculations will not produce a smooth-looking line. Also, in the case of a vertical line we cannot use the slope to make a calculation. To address this problem, we can alter the algorithm to move along the y axis when a line is steep. With this change, we can implement a line-drawing algorithm known as *Bresenham's algorithm*. Pseudo-code for this algorithm is given in Figure 2. The first 15 lines of the algorithm make the needed adjustments depending on whether or not the line is steep, and on its vertical (down or up) and horizontal (left or right) directions. Then, in lines 17 to 25 the algorithm increments the *x* variable 1 step at a time and computes the *y* value. The *y* value is incremented when needed to stay as close to the ideal location of the line as possible. Bresenham's algorithm calculates an *error* variable to decide whether or not to increment each *y* value. The *error* variable takes into account the relative difference between the width (*deltax*) and height of the line (*deltay*) in deciding how often *y* should be incremented. The version of the algorithm shown in Figure 2 uses only integers to perform all calculations.

```
 1   draw_line(x0, y0, x1, y1)
 2       boolean is_steep = abs(y1 - y0) > abs(x1 - x0)
 3       if is_steep then
 4          swap(x0, y0)
 5          swap(x1, y1)
 6       if x0 > x1 then
 7          swap(x0, x1)
 8          swap(y0, y1)
 9
10       int deltax = x1 - x0
11       int deltay = abs(y1 - y0)
12       int error = -(deltax / 2)
13       int y = y0
14       if y0 < y1 then y_step = 1 else y_step = -1
15
16       for x from x0 to x1
17          if is_steep then
18             draw_pixel(y, x)
19          else
20             draw_pixel(x, y)
21          error = error + deltay
22          if error > 0 then
23             y = y + y_step
24             error = error - deltax
```

Figure 2: Pseudo-code for a line-drawing algorithm.

Do the following: Write a C-language program that implements Bresenham's line-drawing algorithm, and uses this algorithm to draw a few lines on the screen. An example of a suitable main program is given in Figure 3, and is included in the design files provided with this lab, in a file named *part1.c* (Note: this file also includes constants for various addresses in the DE1-SoC Computer. Such constants would normally be stored in a '*.h*' *header* file, but we have placed everything inside the file *part1.c* so that the code is compatible with CPUlator, since it supports only one file at a time). You are to write the *draw_line* function, which takes in the end points of the line and the colour to be drawn, as illustrated in Figure 3.

The code first determines the address of the frame buffer by reading from the frame buffer controller, and stores this address into the global variable *frame_buffer_start*. The main program clears the screen, and then draws four lines. An example of a function that uses the global variable *frame_buffer_start* is shown at the end of Figure 3. The function *plot_pixel ()* sets the pixel at location *x*, *y* to the color *line_color*. This function implements the pixel addressing scheme shown in the document "Introduction to the VGA Controller," Figure 2.

Create a new folder to hold your solution for this part and put your code into a file called `part1.c`. You will need to make a Monitor Program project for running your code on a DE1-SoC board; but you can also debug your program at home using CPUlator. Show it working to your TA for grading in the lab. Submit `part1.c` to Quercus before the end of your lab period.

Note: Defining pointers as volatile tells the compiler that this address is not a memory (SDRAM) location, so bypass the cache when accessing it. This is necessary to force the compiler to use stwio and ldwio instructions. However, when you access the SDRAM, using non-io versions of the instructions cause CPUlator warnings. For example, when you define your buffers as global arrays.

```c
int pixel_buffer_start; // global variable

int main(void)
{
    volatile int * pixel_ctrl_ptr = (int *)0xFF203020;
    /* Read location of the pixel buffer from the pixel buffer controller */
    pixel_buffer_start = *pixel_ctrl_ptr;

    clear_screen();
    draw_line(0, 0, 150, 150, 0x001F);   // this line is blue
    draw_line(150, 150, 319, 0, 0x07E0); // this line is green
    draw_line(0, 239, 319, 239, 0xF800); // this line is red
    draw_line(319, 0, 0, 239, 0xF81F);   // this line is a pink color
}

// code not shown for clear_screen() and draw_line() subroutines


void plot_pixel(int x, int y, short int line_color)
{
    volatile short int *one_pixel_address;

        one_pixel_address = pixel_buffer_start + (y << 10) + (x << 1);

        *one_pixel_address = line_color;
}
```

Figure 3: Part of the program for Part I.

Part II

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing this same object at different locations on the screen. A simple way to "move" an object is to first draw the object at one position, and then after a short time erase the object and draw it again at another nearby position. To realize animation it is necessary to move objects at regular time intervals. The VGA controller in your DE1-SoC board's computer system redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, it is reasonable to control an animation using this unit of time.

To ensure that the VGA image is changed only once every $1/60^{th}$ of a second, you can use the frame buffer controller to synchronize with the vertical synchronization cycle of the VGA controller. As described in the background section of this exercise, synchronizing with the VGA controller can be accomplished by writing the value 1 into the *Buffer* register in the frame buffer controller, and then waiting until bit S of the *Status* register becomes equal to 0. For this part of the exercise you do not need to use a back buffer, so ensure that the *Buffer* and *Backbuffer* addresses in the frame buffer controller are the same. In this approach, a frame buffer "swap" can be used as a way of synchronizing with the VGA controller via the *S* bit in the *Status* register.

Do the following: Write a C-language program that moves a horizontal line up and down on the screen and "bounces" the line off the top and bottom edges of the display. Your program should first clear the screen and draw the line at a starting row on the screen. Then, in an endless loop you should erase the line (by drawing the line using black), and redraw it one row above or below the last one. When the line reaches the top, or bottom, of the screen it should start moving in the opposite direction.

Create a new folder to hold your solution for this part and put your code into a file called `part2.c`. You will need to make a Monitor Program project for running your code on a DE1-SoC board; but you can also debug your program at home using CPUlator. Show it working to your TA for grading in the lab. Submit `part2.c` to Quercus before the end of your lab period.

Note: When your code is running on a DE1-SoC board, it will take a fixed amount of time for the horizontal line to move through the 240 lines of the VGA display, which is $240 \times 1/60 = 4$ seconds. However, if your code is running on CPUlator, which performs a software *simulation* of this process, it is slower—measure how long it takes on your computer.

Part III

Having gained the basic knowledge about displaying images and animations, you can now create a more interesting animation.

You are to create an animation of eight small filled boxes on the screen. These boxes should appear to be moving continuously and "bouncing" off the edges of the screen. The boxes should be connected with lines to form a chain. An illustration of the animation is given in Figure 4. Part a of the figure shows one position of the boxes with arrows that indicate the directions of movement, and Figure 4b shows a subsequent position of the boxes. In each step of your animation each of the boxes should appear to "move" on a diagonal line: up/left, up/right, down/left, or down/right. Move the boxes one row and one column at a time on the VGA screen.

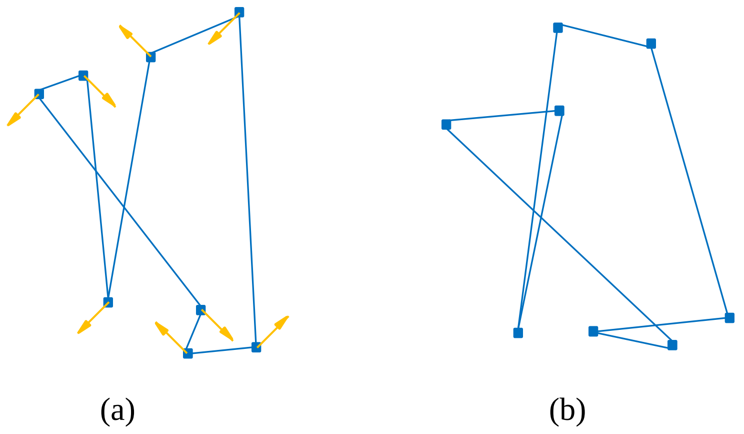

Figure 4: Two instants of the animation.

To make the animation look slightly different each time you run it, use the C library function *rand ()* to help calculate initial positions for each of the boxes, and to determine their directions of movement.

Do the following: Write a C-language program to implement your animation. Define two global arrays as a front and a back buffer in your program, so that you can avoid making changes to the image while it is being displayed by the frame buffer controller. An example of a suitable main program is given in Figure 5 and provided in the file *part3.c*. The code sets the location in memory of *both* the front and back frame buffers—the front buffer is set to the start of Buffer1, and the back buffer to the starting address of Buffer2.

In each iteration of the while loop, the code removes from the screen any boxes and lines that have been drawn in the previous loop-iteration (note: if you clear the entire screen, rather than just "erasing" the boxes and lines that are currently visible, you may find that your animation runs more slowly than expected), then draws new boxes and lines, and then updates the locations of boxes. At the bottom of the while loop the code calls the function *wait_for_vsync ()*, which synchronizes with the VGA controller and swaps the front and back frame buffer pointers.

Experiment with your code by modifying it to use just a single frame buffer (simply change the address of the back buffer to be the same as the front buffer). Explain what you see in the `VGA frame buffer` subwindow as a result of this change.

Create a new folder to hold your solution for this part and put your code into a file called `part3.c`. You will need to make a Monitor Program project for running your code on a DE1-SoC board; but you can also debug your program at home using CPUlator. Show it working to your TA for grading in the lab. Submit `part3.c` to Quercus before the end of your lab period.

```
volatile int pixel_buffer_start; // global variable
short int Buffer1[240][512]; // 240 rows, 512 (320 + padding) columns
short int Buffer2[240][512];

int main(void)
{
    volatile int * pixel_ctrl_ptr = (int *)0xFF203020;
    // declare other variables(not shown)
    // initialize location and direction of rectangles(not shown)

    /* set front pixel buffer to Buffer 1 */
    *(pixel_ctrl_ptr + 1) = (int) &Buffer1; // first store the address in the
        back buffer
    /* now, swap the front/back buffers, to set the front buffer location */
    wait_for_vsync();
    /* initialize a pointer to the pixel buffer, used by drawing functions */
    pixel_buffer_start = *pixel_ctrl_ptr;
    clear_screen(); // pixel_buffer_start points to the pixel buffer

    /* set back pixel buffer to Buffer 2 */
    *(pixel_ctrl_ptr + 1) = (int) &Buffer2;
    pixel_buffer_start = *(pixel_ctrl_ptr + 1); // we draw on the back buffer
    clear_screen(); // pixel_buffer_start points to the pixel buffer

    while (1)
    {
        /* Erase any boxes and lines that were drawn in the last iteration */
        ...

        // code for drawing the boxes and lines (not shown)
        // code for updating the locations of boxes (not shown)

        wait_for_vsync(); // swap front and back buffers on VGA vertical sync
        pixel_buffer_start = *(pixel_ctrl_ptr + 1); // new back buffer
    }
}

// code for subroutines (not shown)

void plot_pixel(int x, int y, short int line_color)
{
    volatile short int *one_pixel_address;

        one_pixel_address = pixel_buffer_start + (y << 10) + (x << 1);

        *one_pixel_address = line_color;^M
}
```

Figure 5: Main program for Part III.