ECE342 - Computer Hardware

University of Toronto

Lab 3: Graphics and Animation

Introduction

The purpose of this exercise is to learn how to display images and perform animation using C programming language. We will use the *DE2 Media Computer* and the Video Graphics Array (VGA) Digital-to-Analog Converter (DAC) on an Altera DE2 Board. This laboratory exercise contains three parts.

Background

The DE2 Media Computer uses a number of circuits, called cores, to control the VGA DAC and display images on a screen. These include the *VGA Pixel Buffer* and *VGA Controller* circuits, which are used together with the SRAM Memory and the SRAM Controller to allow programs executed by the Nios II processor to generate images for display on the screen. The required portion of the DE2 Media Computer is shown in Figure 1.

The *VGA Pixel Buffer* is a controller circuit which serves as an interface between programs executed by the Nios II processor and the VGA Controller. The image data that is to be displayed on the screen is initially stored in a *buffer* in the SRAM Memory. This is achieved by running a program on Nios II and writing the image data to the SRAM memory. The VGA Pixel Buffer continuously retrieves the image data from the SRAM memory and sends it to the VGA Controller. The VGA Controller then uses the VGA DAC to send the image data across the VGA cable to the screen. The VGA Pixel Buffer controller also provides the size of the screen and the address in the SRAM Memory where an image to be displayed is stored.

An image consists of a rectangular array of picture elements, called *pixels*. Each pixel appears as a dot on the screen, and the entire screen consists of 320 columns by 240 rows of pixels, as illustrated in Figure 2. Pixels are arranged in a rectangular grid, with the coordinate $(0, 0)$ at the top-left corner of the screen.

The color of a pixel is a combination of three primary colors: red, green and blue. By varying the intensity of each primary color, any other color can be created. We use a 16-bit halfword to represent the color of a pixel. The five most-significant and least-significant bits in this halfword represent the intensity of the red and blue components, respectively, while the remaining six bits represent the intensity of the green color component, as shown in Figure 3a. For example, a red color would be represented by a value $(F800)_{16}$, a purple color by a value $(F81F)_{16}$, white by $(FFFF)_{16}$, and gray by $(8410)_{16}$.

The color of each pixel in an image is stored at a corresponding address in a buffer in the SRAM memory. The address of a pixel is a combination of a *base* address and an (x, y) offset. In the DE2 Media Computer, the buffer is located at address $(08000000)_{16}$, which is the starting address of the SRAM memory. The (x, y) offset is computed by concatenating the 9-bit x coordinate starting at the 1^{st} bit and the 8-bit y coordinate starting at the 10^{th} bit, as shown in Figure 3b. This computation is accomplished in C programming language by using the left-shift operator:

$$\text{offset} = (x << 1) + (y << 10)$$

To determine the location of each pixel in memory, we add the (x, y) offset to the base address. Using this scheme, the pixel at location $(0, 0)$ has the address $(08000000)_{16}$, the pixel at $(1, 0)$ has the address *base* + $(00000002)_{16}$ = $(08000002)_{16}$, the pixel at $(0, 1)$ has the address *base* + $(00000400)_{16}$ = $(08000400)_{16}$, and the pixel at location $(319, 239)$ has the address *base* + $(0003BE7E)_{16}$ = $(0803BE7E)_{16}$.

The VGA Pixel Buffer controller contains memory-mapped registers that are used to access the VGA Pixel Buffer information and control its operation. These registers, located at starting address $(10003020)_{16}$, are listed in Figure 4.

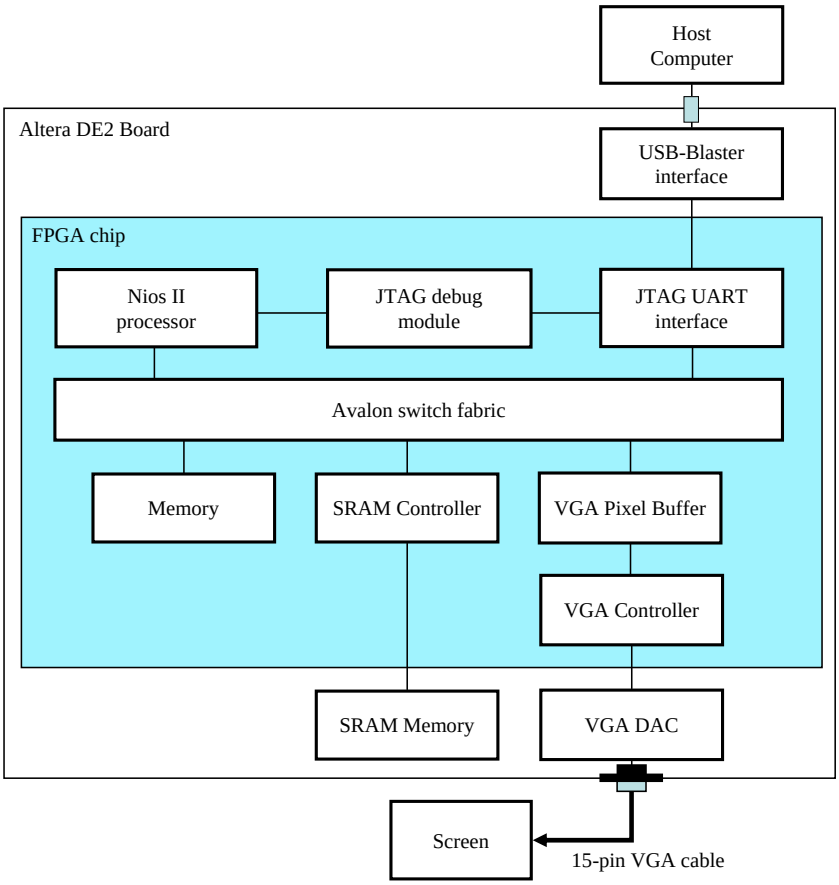


Figure 1: Portion of the DE2 Media Computer used in this exercise.

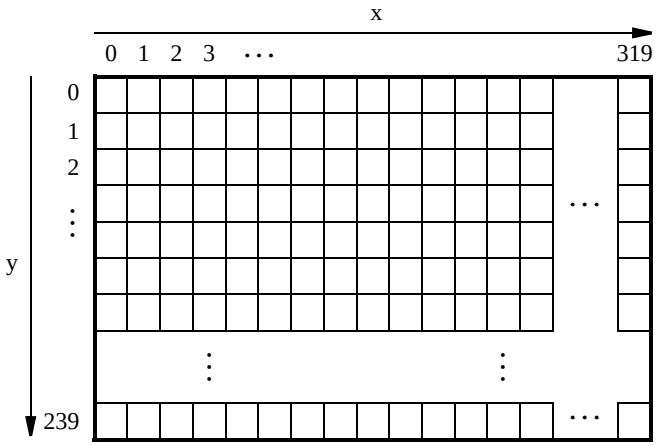
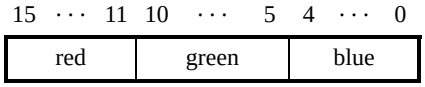
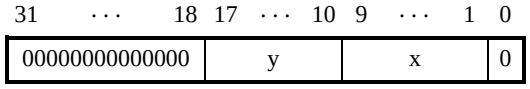


Figure 2: Pixel array.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 3: Pixel color and offset.

Address	31 . . . 24	23 . . . 16	15 . . . 8	7 . . . 4	3	2	1	0	
0x10003020	front buffer address								Buffer register
0x10003024	back buffer address								Backbuffer register
0x10003028	Y		X						Resolution register
0x1000302C	m	n	Unused	B	Unused	A	S		Status register

Figure 4: VGA pixel buffer memory-mapped registers.

The *Buffer* and *Backbuffer* registers store the location in the memory where two image buffers are located. The first buffer, called the *front buffer*, is the memory where the image currently visible on the screen is stored. The second buffer, called the *back buffer*, is used to draw the next image to be displayed. Initially, both registers store the value $(08000000)_{16}$. The *Backbuffer* functionality is not used in this laboratory exercise.

The *Resolution* register holds the width and height of the screen in terms of pixels. The 16 most-significant bits give the vertical resolution, while the 16 least-significant bits give the horizontal resolution, of the screen. The *Status* register holds information about the VGA Pixel Buffer. We will discuss the use of these registers as they are needed in the exercise.

Part I

In this part you will learn how to implement a simple line-drawing algorithm. Drawing a line on a screen requires coloring pixels between two points, (x_1, y_1) and (x_2, y_2), such that they resemble a line as closely as possible. Consider the example in Figure 5.

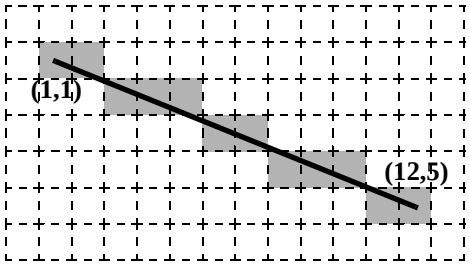


Figure 5: An example of drawing a line between points $(1, 1)$ and $(12, 5)$.

We want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares represent pixels that can be colored. To draw a line using pixels, we have to follow the line and for each column color the pixel closest to the line. To form a line between points $(1, 1)$ and $(12, 5)$ we color the shaded pixels in the figure.

We can use algebra to determine which pixels to color. This is done using the end points and the slope of the line. The slope of the 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:

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

Thus, for column $x = 2$, the y location of the pixel is $\frac{4}{11} + 1 = 1\frac{4}{11}$. Because 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 $y = 1$. We perform this computation for each column between x_1 and x_2.

The approach of moving along the x axis has a drawback when a line is steep. A steep line spans more rows than columns, so if the line-drawing algorithm moves along the x axis to compute the y coordinate for each column there will be gaps in the line. For example, a vertical line has all points in a single column, so the algorithm would fail to draw it properly. To remedy the 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. The pseudo-code for the algorithm is shown in Figure 6.

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

Figure 6: Pseudo-code for a line-drawing algorithm.

The original algorithm uses floating point operations to compute the location of each pixel in the line. Since floating point operations are usually much slower to perform than integer operations, most implementations of this algorithm are altered to use integer operations only. For example, the code shown is Figure 6 is optimized to use only integer operations.

Write a C-language program that draws a few lines on the screen using this algorithm. Do the following:

1. Write a C-language program that implements the line algorithm.
2. Create a new project for the *DE2 Media Computer* using the Altera Monitor Program.
3. Download the DE2 Media Computer onto the DE2 board.
4. Compile and run your program.

Part II

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing the same object at different locations on the screen. To move an object on the screen we must display it at one position first, and then at another later on. A simple way to achieve this is to draw an object at one position, and then erase it and draw it at another position.

The key to animation is timing, because to realize animation it is necessary to move objects at regular time intervals. The time intervals depend on the graphics controller. This is because the controller draws images onto a screen at regular time intervals. The VGA controller in the DE2 Media Computer redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, this will be the unit of time.

To ensure that we draw on the screen only once every $1/60^{th}$ of a second, we use the VGA Pixel Buffer to synchronize a program executing on the DE2 Media Computer with the redraw cycle of the VGA Controller. This is accomplished by writing 1 to the *Buffer* register and waiting until bit 0 of the *Status* register in the VGA Pixel Buffer becomes 0. This signifies that a $1/60^{th}$ of a second has passed since the last time an image was drawn on the screen.

Write a C-language program that moves a horizontal line vertically across the screen and bounces it off the top and bottom edges of the screen. Your program should first clear the screen, by setting all pixels to black color, and then repeatedly draw and erase (draw the same line using the black color) the line during every redraw cycle. When the line reaches the top, or the bottom, of the screen it should start moving in the opposite direction.

Part III

Rotating objects is another interesting part of animation. One way to rotate an object is to pre-compute and store images of a rotated object and then display them on the screen. This method allows an object to be rendered quickly at display time, but requires a lot of memory and is therefore not practical for modern applications. Another way to rotate an object is to compute in real time a new location for each of its points.

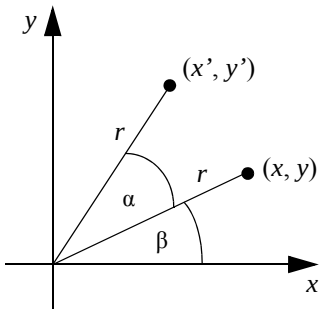


Figure 7: An example of rotation about the center of axes.

To rotate a point (x, y) around an axis by an angle α, we compute the location (x', y') where the point will be after it is rotated. This is illustrated in Figure 7. The point is a distance r away from the origin, at an angle β counterclockwise from the x axis. After the rotation, the point will remain a distance r from the origin and be at an angle $\alpha + \beta$. Using trigonometry we can compute the location (x', y') as:

$$\begin{aligned} r &= \frac{y}{sin\beta} = \frac{x}{cos\beta}, \\ x' &= r \times cos(\alpha + \beta), \text{and} \\ y' &= r \times sin(\alpha + \beta) \end{aligned}$$

We can simplify the equation for x' as:

$$\begin{aligned} x' &= r \times (cos\alpha cos\beta - sin\alpha sin\beta) \\ &= \frac{x}{cos\beta} \times (cos\alpha cos\beta) - \frac{y}{sin\beta} \times (sin\alpha sin\beta) \end{aligned}$$

$$= \quad x \times cos\alpha - y \times sin\alpha$$

For y' we have :

$$y' \quad = \quad x \times sin\alpha + y \times cos\alpha$$

These equations rotate a point around the origin. Since in the screen coordinate system the origin is in the top-left corner, we want to create a Cartesian coordinate system with the origin in the center of the screen to represent the location of each point. Then, to draw the point on the screen we will map the location of the point from the Cartesian coordinate system onto the screen.

To do so, we place the origin at pixel $(160, 120)$, and have the y axis point upwards. To map any point (x_w, y_w) onto the screen at location (x_{pixel}, y_{pixel}), we use equations:

$$\begin{aligned} x_{pixel} &= 160 + x_w \\ y_{pixel} &= 120 - y_w \end{aligned}$$

Write a C-language program that rotates a line of 20 pixels in length around the center of the screen. During each redraw cycle, your program should rotate the line by two degrees counterclockwise. Do the following:

1. Write a C-language program to rotate a line. Make sure to include *math.h* library in your code to be able to use *sin* and *cos* functions.

2. Create a new project in the Altera Monitor Program. Add the *-lm* flag to the *Additional Linker Flags* field as shown in Figure 8 to include the math library when compiling your program.

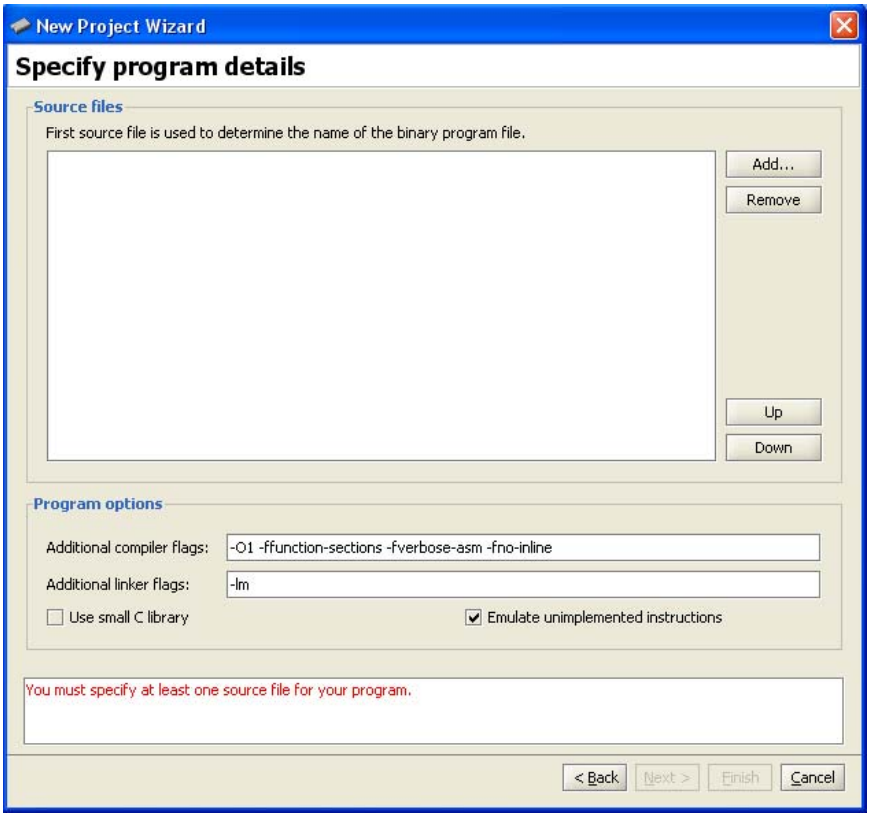


Figure 8: Adding -lm flag to instruct the linker to include the math library in your program.

3. Compile, download and run your program on the DE2 Media Computer.

Preparation and In-lab

The preparation for this laboratory exercise must include C-language source code for Parts I through III. In the lab you will demonstrate the operation of your programs on the DE2 board and the VGA screen.

Notes

The manual for the DE2 Media Computer can be found on Altera's University Program website. Go to *http://university.altera.com* and then *Educational Materials → Computer Organization → Design Examples*.

In the lab, copy your files to the local hard drive (`C:`). The tools run much faster when they access a local disk. The network drive and USB stick are much slower devices. Make sure to backup files once you are done with the lab.

Marking scheme

Preparation

- Part I (2 marks)
- Part II (2 marks)
- Part III (2 marks)

In-lab

- Part I (2 marks)
- Part II (2 marks)
- Part III (2 marks)