Laboratory Exercise 9

Graphics and Animation

The purpose of this exercise is to learn how to display images and perform animation. We will use the DE2 Media Computer and the Video Graphics Array (VGA) Digital-to-Analog Converter (DAC) on an Altera DE2 Board.

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 a VGA Pixel Buffer and a VGA controller circuit, 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.

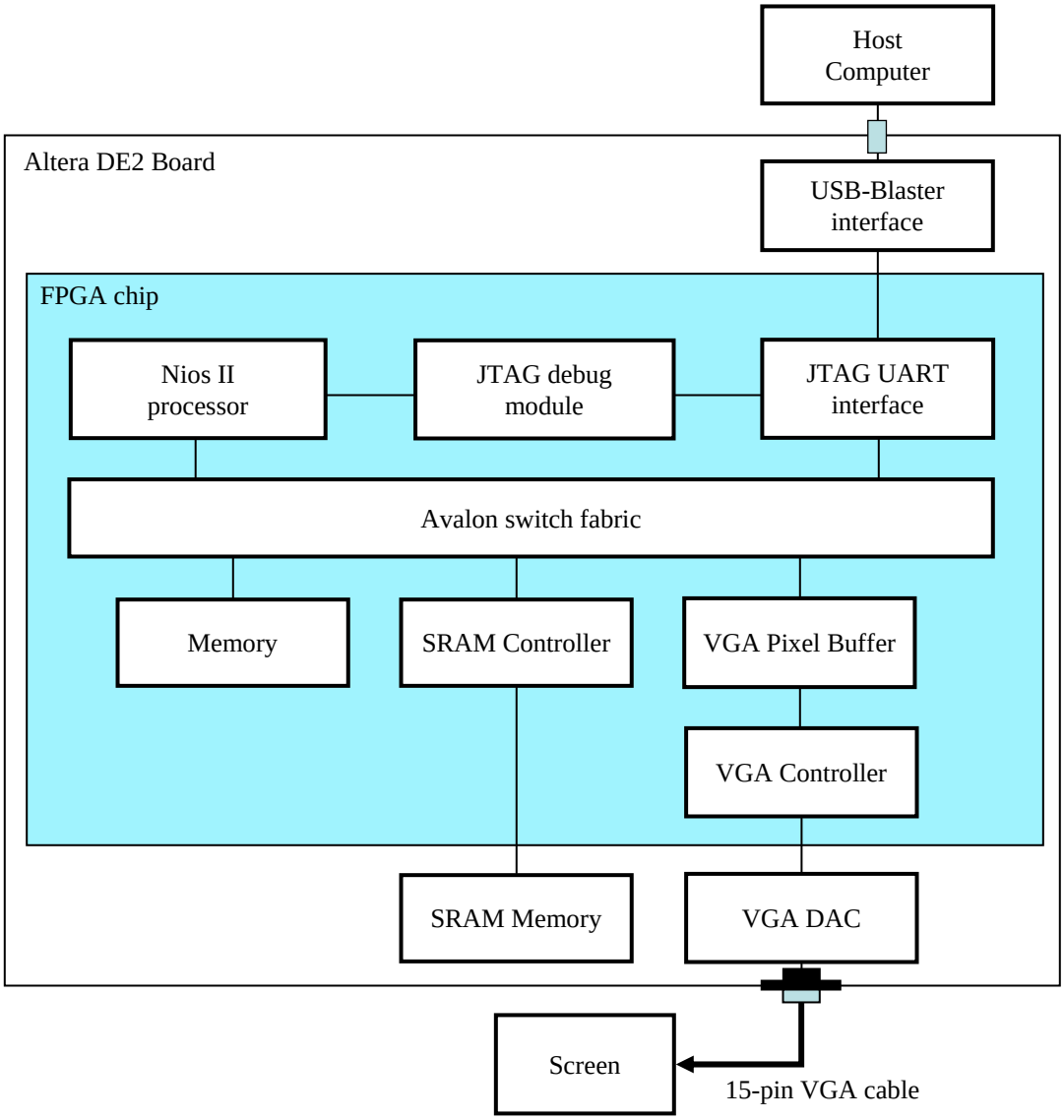


Figure 1. Portion of the DE2 Media Computer used in this exercise.

The VGA pixel buffer is an interface between programs executed by the Nios II processor and the VGA controller.

It gives the size of the screen and the location in the SRAM memory where an image to be displayed is stored. To display an image on the screen, the VGA pixel buffer retrieves it from 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.

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.

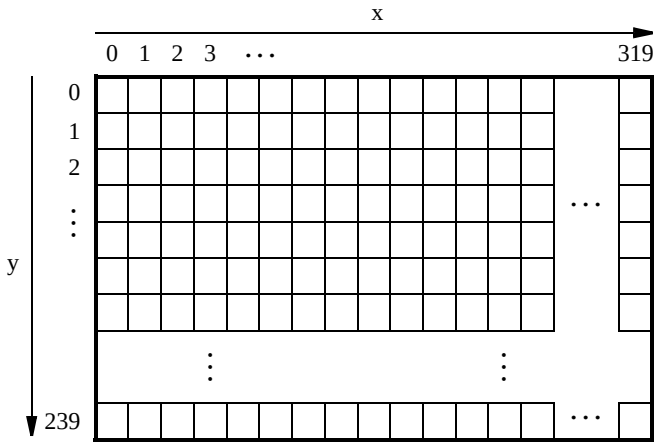
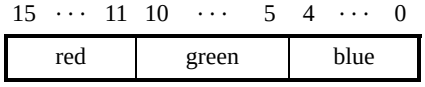
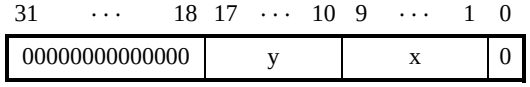


Figure 2. Pixel array.

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 $(8410)_{16}$.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 3. Pixel color and offset.

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)$_{16}$.

To display images from a program running on the DE2 Media Computer, the VGA pixel buffer module 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.

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 $(08000000)_{16}$. We will discuss how to use these buffers in Part VI of the 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.

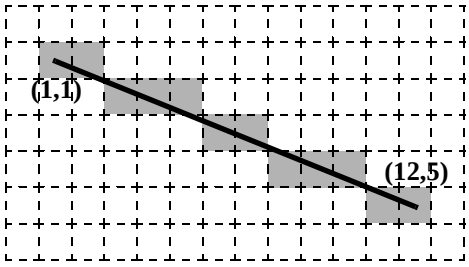


Figure 5. An example of drawing a line between points $(1, 1)$ and $(12, 5)$.

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      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      int deltax = x1 - x0
10     int deltay = abs(y1 - y0)
11     float error = 0
12     float slope = deltay / deltax
13     int y_step
14     int y = y0
15     if y0 < y1 then y_step = 1 else y_step = -1
16     for x from x0 to x1
17         if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
18         error = error + slope
19         if error >= 0.5 then
20             y = y + y_step
21             error = error - 1.0
```

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

This 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.

Write a C-language program that draws a few lines on the screen using this algorithm. Attempt to optimize the algorithm to use only integer operations. 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. Connect a 15-pin VGA cable to the VGA connector on the DE2 board and the monitor.

   The VGA cable can be connected to the screen in two ways. If the screen has multiple VGA input ports, you can connect the VGA cable to an unused port. Then, using the buttons on the screen change the video source to the corresponding port. If the screen has only a single VGA port, a KVM (Keyboard-Video-Mouse) switch is needed. Using the KVM switch, you can connect multiple video sources to a single screen. This device will allow you to choose which video source will be displayed.
5. 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.

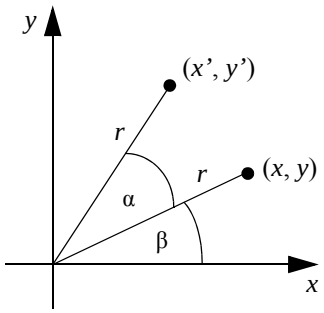


Figure 7. An example of rotation about the center of axes.

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:

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

For y' we have :

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

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 $(160, 120)$, consider each 20 pixels on the screen to be one unit, 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:

$$x_{pixel} = 160 + x_w \times 20$$
$$y_{pixel} = 120 - y_w \times 20$$

Write a C-language program that rotates a line of length five 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.

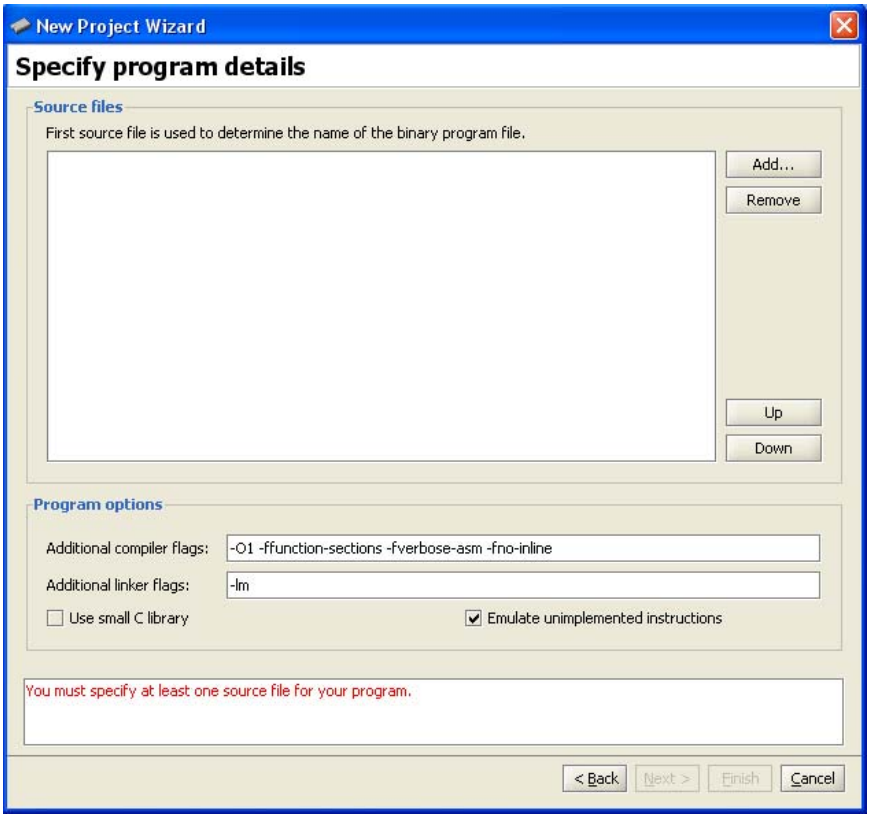


Figure 8. Adding -lm flag to instruct the linker to include the math library in your program.

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

Part IV

So far, we have discussed how to display and animate 2D objects. However, the world around us is inherently 3D, as in addition to width and height objects have depth. Also, when an object is close it appears larger than when it is farther away. We can simulate these effects on a computer screen using *perspective projection*.

Perspective projection is a method of representing a 3D object in a 2D image, as it appears from the point of view of an observer. The idea is illustrated in Figure 9. To project a 3D object onto a 2D plane, we draw a dotted line between each point on the object and the observer. The location at which the line crosses the projection plane is the position on the plane where the observer perceives the point to be.

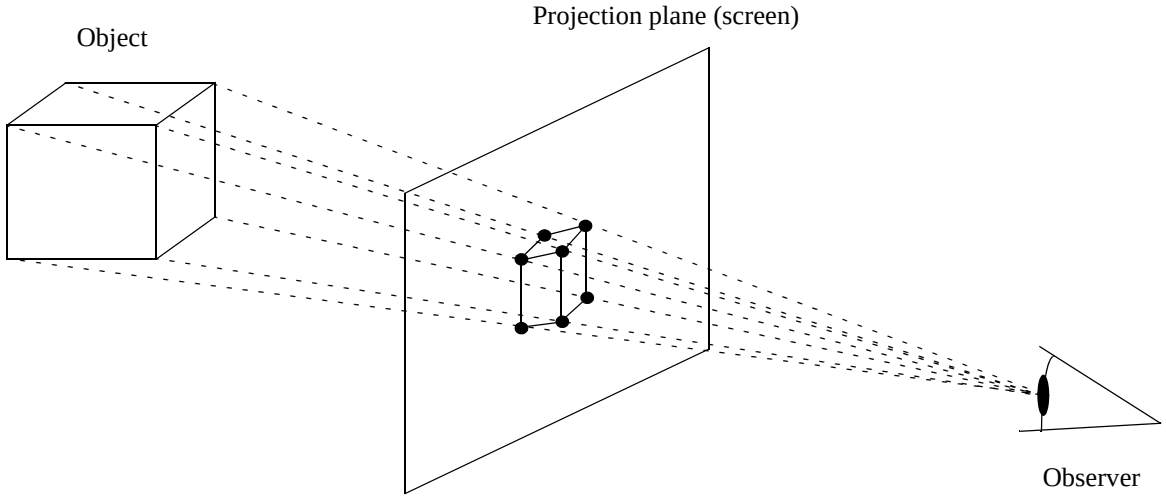


Figure 9. Visualization of the perspective projection.

To project a point (x, y, z) from a 3D world, where z axis points into the screen, we need to compute the location of point (x_s, y_s) on the screen where the point would be seen. The problem is illustrated in Figure 10.

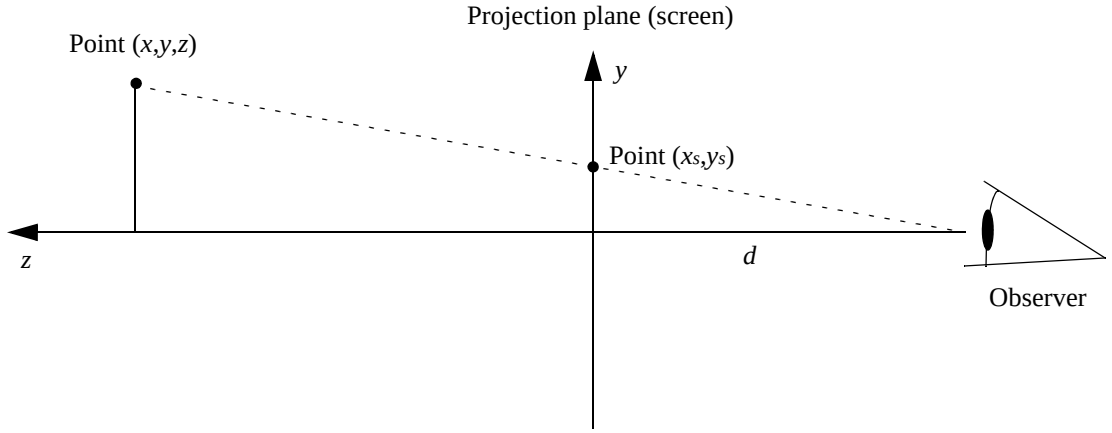


Figure 10. Visualization of the perspective projection.

Given the location of the point (x, y, z), and a distance d the observer is from the screen, we compute coordinates of the point (x_s, y_s) using the concept of similar triangles from geometry. We compute the y_s coordinate as:

$$\frac{y}{z+d} = \frac{y_s}{d}$$

$$y_s = \frac{y \times d}{z + d}$$

For x_s we have:

$$x_s = \frac{x \times d}{z + d}$$

Extend the C-language program from Part III to include perspective projection. In your program, you should rotate a line about the z axis (same rotation equations as in part III) by two degrees at every refresh cycle. In addition, your line should bounce between planes $z = 100$ and $z = 5$. Set the observer location to be 20 units away from the screen. Run and test your program.

Part V

Rotating objects around the origin of the 3D world is only marginally useful. In practical applications, where objects move and change their orientation, we need to allow objects to rotate on the spot wherever they may be. To do this we need to introduce a local coordinate system for each object. The local coordinate system is the same as the world coordinate system, except it is centered where the object is located. This idea is illustrated in Figure 11.

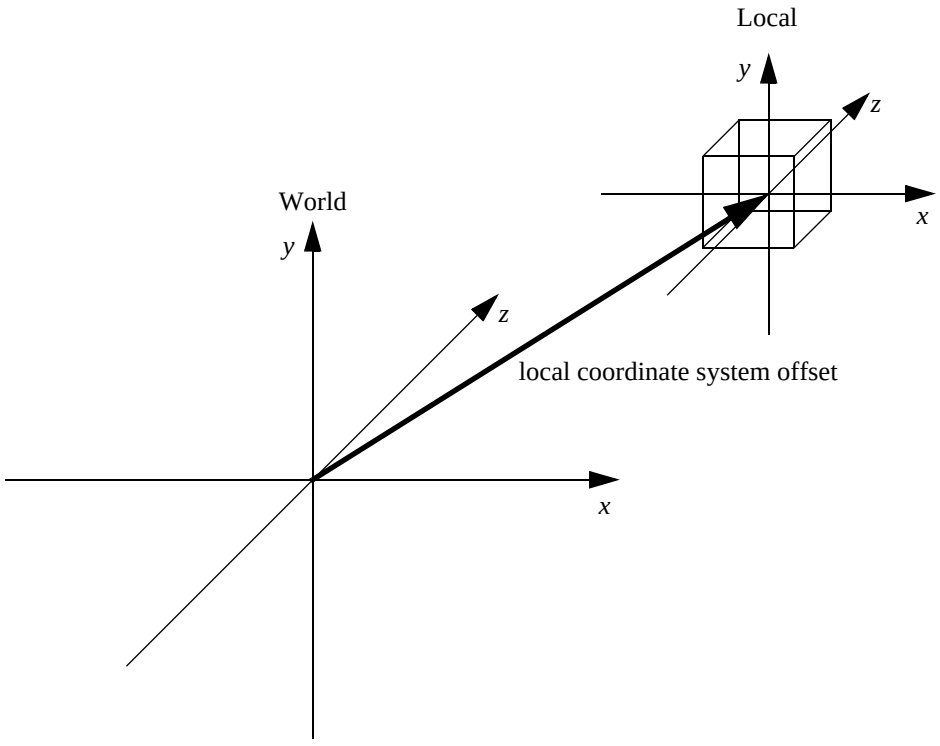


Figure 11. Relationship between the local and the world coordinate systems.

In the figure, the cube is centered in the local coordinate system, some distance away from the origin of the 3D world. Any change in the orientation of the cube can be accomplished by a rotation within the local coordinate system, whereas its movement in the world is accomplished by moving the local coordinate system around the 3D world. The location of each point of the object is therefore the sum of the displacement of the local coordinate system from the origin of the 3D world, and the location of the point in the local coordinate system.

In this part you have to write a C-language program to draw a wire-frame object (object composed of lines) and animate it on the screen. Use the t_3D_wireframe_object and t_3D_point data structures in Figure 12 to represent the object.

```
typedef struct s_3D_point {
    float x,y,z;
} t_3D_point;

typedef struct s_3D_wireframe_object {
    t_3D_point origin;
    t_3D_point *points;
    int num_points;
    int *line_segments;
    int num_line_segments;
} t_3D_wireframe_object;
```

Figure 12. Wire-frame object data structures.

The t_3D_wireframe_object structure consists of an *origin* field, that specifies the location of the local coordinate system with respect to the origin of the 3D world. The array of *points* defines the points that represent the object, and their number is specified by the *num_points* field. The array of *line_segments* holds pairs of integers that indicate which points should be connected by a line in this object. The array contains $2\times$*num_line_segments* elements, where *num_line_segments* is the number of line segments in the object.

Write a C-language program that rotates a wire-frame object in the shape of a cone with a hexagonal base. The object should rotate about the axes in its local coordinate system. The direction of rotation will be defined by switches SW_{2-0} on the DE2 board. When switch SW_2 is ON, the object should rotate around the x axis, when SW_1 is ON the object should rotate around the y axis, and when SW_0 is ON the object should rotate around the z axis. The rotation equations about each of the axes are given in Figure 13.

Rotation about the x axis:

$$x' = x$$
$$y' = y \times cos\alpha - z \times sin\alpha$$
$$z' = y \times sin\alpha + z \times cos\alpha$$

Rotation about the y axis:

$$x' = x \times cos\alpha + z \times sin\alpha$$
$$y' = y$$
$$z' = z \times cos\alpha - x \times sin\alpha$$

Rotation about the z axis:

$$x' = x \times cos\alpha - y \times sin\alpha$$
$$y' = x \times sin\alpha + y \times cos\alpha$$
$$z' = z$$

Figure 13. Rotation equations in 3D

Part VI

The program written for Part V may have appeared to flicker or draw an incomplete image of the cone. This happens when a computer takes too much time to render a new image on the screen. This looks bad and is undesirable in computer animation.

A commonly used technique called *double buffering* can remedy this problem. Double buffering uses two buffers, rather than just one, to render an image on the screen. One of the buffers is visible on the screen and the other is hidden. The visible buffer is called the *front buffer* and the hidden buffer is called the *back buffer*. To create an animation using two buffers we draw an image in the back buffer. When the image is ready and the VGA controller is about to draw a new image on the screen, we swap the front and the back buffers. This operation

makes a seamless transition from one image to another. Once the buffers are swapped, the back buffer becomes the front buffer and vice-versa.

The VGA pixel buffer in the DE2 Media Computer supports double buffering. The location of the buffers in memory is stored in registers *Buffer* and *Backbuffer*, showin in Figure 4. Initially, the location of both buffers is the same, thus only one buffer is used. To enable double buffering, we need to separate the front and the back buffers by setting the Backbuffer register to address $(08040000)_{16}$. This will cause half of the SRAM memory in the DE2 Media Computer to be used for the front buffer, and the other half for the back buffer.

Once we have two buffers, the question becomes how to display the image in the back buffer. This is accomplished by the function in Figure 14, which swaps the buffers.

```
volatile int *vga_pixel_buffer_buffer_reg = (int *) 0x10003020;
volatile int *vga_pixel_buffer_status_reg = (int *) 0x1000302C;

volatile int *vga_screen_front_buffer = (int *) 0x08000000;
volatile int *vga_screen_back_buffer = (int *) 0x08040000;

void swap_buffers() {
    register int status;

    // Display the back buffer
    *vga_pixel_buffer_buffer_reg = 1;

    // Swap the addresses for the back and front buffers
    register int *temp = vga_screen_front_buffer;
    vga_screen_front_buffer = vga_screen_back_buffer;
    vga_screen_back_buffer = temp;

    // Wait until the buffer swap is completed by the VGA pixel buffer.
    status = *vga_pixel_buffer_status_reg;
    while( (status & 0x01) != 0 ) {
        status = *vga_pixel_buffer_buffer_swap_status_reg;
    }
}
```

Figure 14. Swapping front and back buffers.

When the function is executed, the image in the back buffer will be displayed and the front and back buffers will be swapped. This allows us to draw another image in the back buffer without disturbing the image displayed on the screen.

Extend the C-language program from Part V to use both buffers. Your program should only draw objects in the back buffer and erase the back buffer after every buffer swap.

Preparation

The recommended preparation for this laboratory exercise includes C-language source code for parts I through VI.