Laboratory Exercise 5

Using ASCII Graphics for Animation

The purpose of this exercise is to learn how to perform simple animations under Linux*. Graphics will be "drawn" by sending ASCII *escape commands* to the Linux Terminal window.

Part I

Although a Linux Terminal window normally displays ASCII text, we can use the Terminal's *escape command* sequences to make simple drawings, often called *ASCII graphics*. Animations can be created on the Terminal by using commands to clear the screen, move the Terminal cursor to specific locations, show/hide the cursor, change the color of characters, and so on. An example of a program that uses ASCII graphics is given in Figure 1.

```
1  /* This program draws a few characters on the screen. */
2  #include <stdio.h>
3  #define YELLOW 33
4  #define CYAN 36
5  #define WHITE 37
6
7  void plot_pixel(int, int, char, char);
8
9  int main(void) {
10    char c;
11    int i;
12    printf ("\e[2J");              // clear the screen
13    printf ("\e[?25l");            // hide the cursor
14
15    plot_pixel (1, 1, CYAN, 'X');
16    plot_pixel (80, 24, CYAN, 'X');
17    for (i = 8; i < 18; ++i)
18       plot_pixel (40, i + 12, YELLOW, '*');
19
20    c = getchar ();                // wait for user to press return
21    printf ("\e[2J");              // clear the screen
22    printf ("\e[%2dm", WHITE);     // reset foreground color
23    printf ("\e[%d;%dH", 1, 1);    // move cursor to upper left
24    printf ("\e[?25h");            // show the cursor
25    fflush (stdout);
26 }
27
28 void plot_pixel(int x, int y, char color, char c)
29 {
30    printf ("\e[%2dm\e[%d;%dH%c", color, y, x, c);
31    fflush (stdout);
32 }
```

Figure 1: An example of code that uses ASCII graphics.

The first line of code in Figure 1 *includes* the *stdio.h* library, which is required to use the *printf* function. The next three lines define constants, described later, for text colors which can be used in the Terminal window. A command is sent to the Terminal in line 12 by using *printf*. All Terminal window commands begin with the ASCII

`ESC` (escape) character, which is specified in the *printf* string using the syntax \e. The command in line 12, which is `[2J`, instructs the Terminal to clear the screen. Another command, `[?25l`, given in line 13, causes the Terminal to *hide* the cursor so that it is not visible to the user. Next, the function `plot_pixel` is called to draw some characters at specific locations on the screen. In this example the size of the Terminal window is assumed to be 80 columns by 24 rows. Coordinates $(1, 1)$ are at the top-left corner of the screen and $(80, 24)$ are at the bottom-right corner. The calls to `plot_pixel` in lines 15 to 18 draw cyan-colored `X` characters at coordinates $(1, 1)$ and $(80, 24)$, and a vertical yellow line of ten `*` characters down the middle of the screen.

The `plot_pixel` function, shown in lines 28 to 32 uses two commands to draw a character. The first command is `[ccm`, where *cc* is called an *attribute*. The attribute can be used to set the color of text characters, by using different values of *cc*. Examples of color attributes are $cc = 31$ (red), 32 (green), 33 (yellow), 34 (blue), 35 (magenta), 36 (cyan), and 37 (white). The second command in `plot_pixel` is `[yy;xxH`, where yy and xx specify a row and column on the screen, respectively. This command moves the Terminal cursor to (x, y) coordinates (xx, yy). In line 20 the program waits, using the `getchar` function, for the user to press a key. Finally, commands are sent to the Terminal to *clear* the screen, *set* the color to white, *set* the cursor to coordinates $(1, 1)$, and *show* the cursor.

The use of ASCII graphics described above became popular around the year 1980 when they were available in computer video terminals called the *VT100*, manufactured by Digital Equipment Corporation. The Linux Terminal provides ASCII graphics by *emulating* the capabilites of the VT100 video terminal. A listing of some escape commands is given in a table at the end of this document. More information about ASCII graphics commands can be found by searching on the Internet for a topic such as "VT100 graphics".

Perform the following

1. Create a C source-code file for Figure 1. The code is provided along with this laboratory exercise.
2. Compile your program and execute it in a Linux Terminal window to display the ASCII graphics.

Part II

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 coordinates (x_1, y_1) and (x_2, y_2), such that the pixels represent the desired line as closely as possible. In the case of ASCII graphics, a *pixel* represents a character coordinate on the screen. Consider the example in Figure 2, where we want to draw a line between coordinates $(1, 1)$ 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.

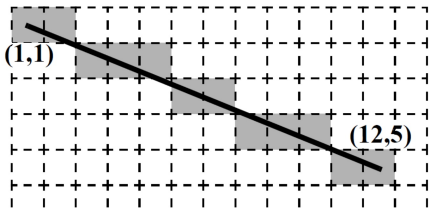

Figure 2: Drawing a line between coordinates $(1, 1)$ and $(12, 5)$.

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:

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

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*. A key property of this algorithm is that all variables are *integers*. Pseudo-code for this algorithm is given in Figure 3. The first 15 lines of the algorithm make the needed adjustments depending on whether or not the line is steep, and to account for the horizontal and vertical directions of the line. Then, in lines 17 to 22 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 of the line (*deltax*) and height of the line (*deltay*) in deciding how often *y* should be incremented. The version of the algorithm shown in Figure 3 uses only integers to perform all calculations.

```
1  draw_line(x0, x1, y0, y1)
2
3     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    deltax = x1 - x0
12    deltay = abs(y1 - y0)
13    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 3: Pseudo-code for a line-drawing algorithm.

Perform the following:

1. Write a C program that uses Bresenham's line-drawing algorithm to draw a few lines on the screen. An example of output that your program might produce is illustrated in Figure 4. In this example the lines are drawn using the '*' character, but you can use any ASCII character to draw your lines.
2. Compile and test your program using a Linux Terminal.

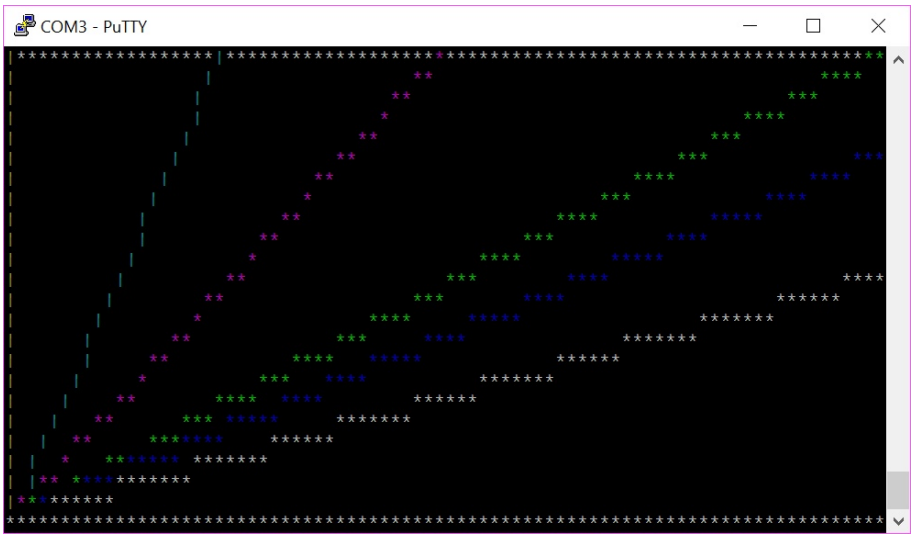

Figure 4: Drawing a few lines on the screen.

Part III

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. For this exercise we can use a timer controlled by the Linux kernel, such as the *nanosleep* function. A reasonable delay might be about 0.1 seconds, but you should experiment with different animation delays and observe the results. Documentation for *nanosleep* can be obtained by searching on the Internet.

Perform the following:

1. 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 then draw the line at a starting row. Then, in an endless loop you should erase the line, 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. Note that the Terminal window may be somewhat *slow* to receive ASCII characters, and this may limit the speed of your animation. When erasing a line, it may be more efficient to clear the whole screen, using the escape command `[2J`, as compared to redrawing the line by printing black characters.

2. Compile your code and test it. Experiment with different animation delay times.

Part IV

Having gained some basic knowledge about drawing lines and animations, you can now create a more interesting animation.

You are to create an animation of several objects on the screen. These objects should appear to be moving continuously and "bouncing" off the edges of the screen. The objects should be connected with lines to form a chain. An illustration of the animation is given in Figure 5. Part a of the figure shows one position of the objects with arrows that indicate the directions of movement, and Figure 5b shows a subsequent position of the objects. In each step

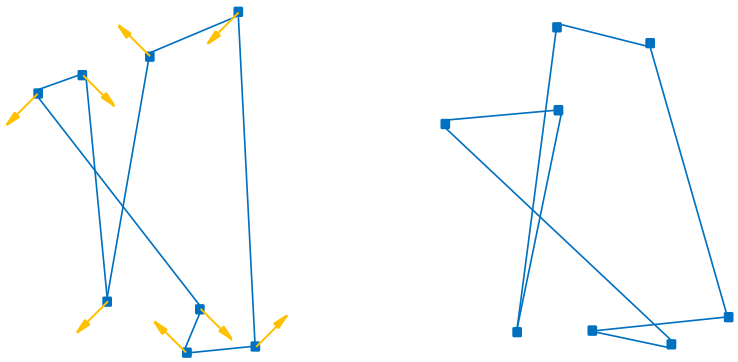

Figure 5: Two instants of the animation.

of your animation each of the objects should appear to "move" on a diagonal line: up/left, up/right, down/left, or down/right. Move the objects one row and one column at a time on the Terminal screen.

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 objects, and to determine their directions of movement.

Perform the following:

1. Write a C-language program to implement your animation. Use some ASCII character to represent each object—for example, the 'X' character. Connect the objects together by drawing lines using other characters—for example '*', '-', or 'l'. Use Bresenham's algorithm, described in Part II, to draw the lines. An example of a suitable main program is given in Figure 6. The code first initializes some variables, sets the locations of the objects to be drawn, and sets the animation time. The code then enters an infinite loop, which can be interrupted by pressing `^C` on the keyboard.

   In each iteration of the loop the code calls a function to draw the animation. The `draw` function first clears the screen, draws the objects and lines, and then updates the locations of objects. At the bottom of the loop the code calls the *nanosleep* function to wait for the animation time to expire.

2. Compile your program and test the animation. Note that, depending on the quantity of text being shown on the screen, your animation may look "smoother" if the connection between your host computer and the DE-series board has a high bandwidth; for example, a network connection may provide better results in comparison to a serial-port connection.

Part V

For this part of the exercise you are to enhance the animation from Part IV so that during the animation the following changes can take place:

1. The speed of movement of the objects can be increased or decreased
2. The number of objects in the animation can be increased or decreased
3. The lines between objects can be drawn or not drawn

```
... include files (not shown)

volatile sig_atomic_t stop;    // used to exit the program cleanly
void catchSIGINT(int);
struct timespec animation_time;  // used to control the animation timing

... declare function prototypes and variables (not shown)

/******************************************************************************
 * This program draws Xs and lines on the screen, in an animated fashion.
 ******************************************************************************/
int main(void)
{
   ... declare variables (not shown)

   // catch SIGINT from ^C, instead of having it abruptly close this program
   signal(SIGINT, catchSIGINT);

   // set random initial position, "delta", and color for all Xs
   ... code not shown

   printf ("\033[?25l");                              // hide the cursor
   fflush (stdout);

   // initialize the animation speed
   animation_time.tv_sec = 0;
   animation_time.tv_nsec = 200000000;                // 0.2 seconds
   while (!stop)
   {
      draw ( );                                       // draw the animation
      nanosleep (&animation_time, NULL);              // wait for timer
   }
   printf ("\033[2J");                                // clear the screen
   printf ("\e[%2dm\e[%d;%dH", WHITE, min_x, min_y);  // reset color and cursor
   printf ("\e[?25h");                                // show the cursor
   return 0;
}

/* Function to allow clean exit of the program */
void catchSIGINT(int signum)
{
   stop = 1;
}
```

Figure 6: Main program for Part IV.

Perform the following:

1. Implement the speed control discussed above for the animation, as indicated in Table 1.

   When any of the switches *SW* is set to the "on" position the lines between objects should not be drawn; only when all *SW* switches are set to the "off" position should the lines appear. You can read the KEY and SW switches using either memory-mapped I/O, or by reading from their character device drivers.

2. Compile your code and test the new features for changing the animation.

3. Add any other animation features that you may find interesting.

Table 1: Animation control using the DE1-SoC board.

KEY	Action
KEY_0	The speed of animation should be increased by a noticeable amount
KEY_1	The speed of animation should be decreased by a noticeable amount
KEY_2	The number of displayed objects should be increased by some quantity
KEY_3	The number of displayed objects should be decreased by some quantity

Table of ASCII escape commands.

Command	Result
`\e7`	save cursor position and attributes
`\e8`	restore cursor position and attributes
`\e[H`	move the cursor to the home position
`\e[?25l`	hide the cursor
`\e[?25h`	show the cursor
`\e[2J`	clear window
`\e[ccm`	set foreground color to `cc`
`\e[yy;xxH`	set cursor location to row `yy`, column `xx`