APS 105F – Computer Fundamentals

Assignment #6: OOT Graphics

Fall 1997

To be completed by Oct. 15/16.

Objective: To learn how to use OOT graphics.

Background Notes On Computer Displays

The video display tube on a computer screen is composed of a regular pattern of small dots called pixels. The pixels are arranged in the fashion of a rectangular grid. Images that appear on the screen are created by controlling the intensity (and hue, if the display is color) of each pixel. For example, text characters are created in a character space by making a specific pattern of the pixels in that space black, and the other pixels white. When you type out text, you naturally don't have to specify this pixel pattern for every character, as characters are formed from standardized pixel arrangements that the monitor knows about. When you work with graphics, you usually do not have to specify intensities and hues for individual pixels (although, you can), but you will have to supply pixel coordinates to indicate where to place a graphics object.

On the NCD19 Xterminals in SF1012, the resolution is 1280x1024, meaning that there are over 1.3 million individually controllable pixels on the screen, arranged in a rectangular pattern of 1280 columns and 1024 rows. Therefore, in OOT, you can create graphics windows that are up to 1280x1024 pixels in size. The pitch of the display, which is the spacing between pixels, is 0.01 inches (also described as 100 dots per inch). By contrast, a TV image (NTSC) consists of 525 lines of 440 dots each (relatively poor quality, at about 30 dots per inch on a 21" set), while many laser printers offer 600 dots per inch (near photographic quality).

1 Simple Graphics

As mentioned in the notes on computer graphics, you will be able to draw simple graphics objects without manipulating individual pixels. This is done using some of the built-in graphics procedures such as **drawline, drawbox, drawoval,** and others that are listed in the OOT reference manual. In addition, you can create customized graphics procedures that will perform specific graphics tasks if you supply some information. Here is a simple program that will illustrate the use of custom-built graphics procedures. Type it in and run it:

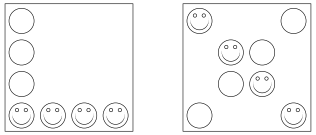

Figure 1: Face pattern 1 and pattern 2

```
include "/usr/copy/aps105/images.t"
setscreen ("graphics:300;400,nocursor")
put "Hello World"
DrawCircle (100,100)
DrawHappyFace (200,300)
DrawSadFace (200,100)
```

This example makes use of three custom-built graphics procedures:

`DrawCircle`, `DrawHappyFace`, and `DrawSadFace`.

The code for these procedures is located in the file

`/usr/copy/aps105/images.t`

hence the `include` statement at the beginning.

The setscreen statement is necessary to inform OOT to set up a run window capable of displaying graphics. The `300;400` in the argument of `setscreen` indicates that the graphics window is to be 300 pixels wide and 400 pixels high. When locating objects in the graphics window, you must use a pixel address, which is simply an x,y co-ordinate. The bottom left corner is always 0,0. For this example, the lower right corner is 299,0, the upper left corner is 0,399, and the upper right corner is 299,399.

As can be expected from their names, the procedures `DrawCircle`, `DrawHappyFace`, and `DrawSadFace` draw a circle, a happy face, and a sad face respectively. The arguments for these procedure calls are the pixel addresses at which the images are to be centered. The circle that is drawn is centered at 100,100; the happy face is centered at 200,300 and the sad face is centered at 200,100.

1.1 Repetitive Circle And Happy Face Drawing

Use repetition statements to create the pattern illustrated on the left hand side of Figure 1. For the column of circles use a for loop, and for the row of happy faces, use a loop loop. You may want to make use of the `maxx` and `maxy` symbols, which return the maximum x co-ordinate and the maximum y co-ordinate respectively. See your Turing reference manual for details.

Adjust the program to produce the pattern shown on the right hand side of Figure 1. Two additional statements should be all that is required. When you are finished, each of your loops will be controlling both x and y co-ordinate variables.

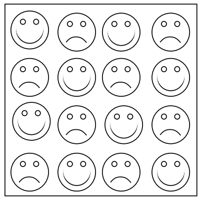

Figure 2: Happy and sad checkerboard.

1.2 Creating A Checkerboard Pattern

Design and test a Turing program to produce a "checkerboard" pattern of happy and sad faces, as shown in Figure 2.

1.3 Creating The Illusion Of Movement (Optional)

The programs in this assignment have illustrated how to create patterns of circles and faces using repetition. Objects can be made to appear to move across the screen using repetition if, when a new object is drawn, the old one is erased. Experiment with the for loops and loop loops to create a "movie" of a happy face moving across the window. Delays may have to be added in between iterations of the loop to slow down the image. You may want to look up the **cls** and **delay** statements in the Turing reference manual.

Once you have figured out how to get the happy face moving across the screen, modify your program so that the happy face will continue to move around the window indefinitely. Have the happy face bounce into another direction when it reaches the edge of the window.

You may want to try this late at night or early in the morning when the system is not very loaded so that movements are smooth; if the system is heavily loaded then the movement will tend to be jerky.

2 Scaling and Plotting Functions

In this exercise the only new graphics statement you will have to know is **drawline**, which you can look up in the Turing reference manual. You are to design, implement and test a Turing program which can plot *any* function as a graph. The function will appear as code in your program as follows:

```
function y (x: real): real
  result code to calculate function here
end y
```


For example, the function f(x) = sin(x)cos(2x) would be coded as

```
function y (x: real): real
  result sin(x)*cos(2*x)
end y
```

Your functions may be considerably more complex if you wish.

The general steps your program will follow are:

1. Prompt the user for initial and final values of x to determine the domain of the graph.
2. Prompt the user for the number of equally spaced points in the domain that are to construct the plot.
3. Evaluate f(x) for each of the equally spaced points in the specified domain to determine the maximum absolute value in the range of the function.
4. Create a graphics window, and draw in a horizontal axis halfway up the window. From the domain of the function to be plotted and the horizontal size of the graphics window, determine the function that will map a horizontal point of your graph to an x co-ordinate of the graphics window.
5. From the maximum absolute value in the range of the function and the vertical size of the graphics window, determine the function that will map a vertical point of your graph to a y co-ordinate of the graphics window.
6. For each point on the graph, locate its pixel address on the graphics window from the mapping functions determined above. Draw a line from this point to the previous point on the graph.
7. Finish off the graph by indicating the domain and range of the plot and by drawing a vertical axis if x=0 is within the domain of the plot.

The key to understanding this assignment is that you have to map an image area from real coordinates to an image area on the screen in pixel coordinates. For example, if you have a window X pixels wide and a real domain of $[-10, 20]$ then it is necessary to map the $[-10, 20]$ onto $[0, X]$. Similarly, if the minimum and maximum y value is $miny$ and $maxy$ respectively, then you need to map $[miny, maxy]$ onto $[0, Y]$ assuming the window is Y pixels high. Remember that this has to work even if $miny$ and $maxy$ have opposite signs.

3 Implementing Drawline

In this section you will implement a version of the **drawline** procedure using the **drawdot** builtin procedure, i.e., your program must compute where to draw dots to construct the line. To avoid confusion with the existing **drawline** procedure, call your procedure **lineto**. It should be used just like **drawline**.

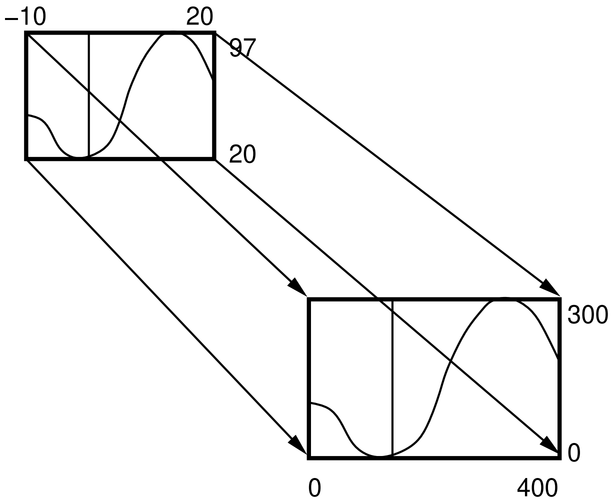

Figure 3: Mapping real coordinates onto pixel coordinates

Show that your procedure works by using it in a program. In your program, you should demonstrate that your procedure works for under all conditions for the endpoints, including endpoints that are off the screen. In the latter case, you can implement what is called *clipping*, where you draw the visible portion of the line.