CMPSCI 201 – Fall 2006

Professor William T. Verts

Assignment #5 – Array and Graphics

EXTRA CREDIT

In this assignment you are to extend the program in assignment #4 to include the framework for a simple graphics system using text characters as pixels. The basic graphics screen will be 32 raster lines (rows) by 96 pixels per line (columns) so that it will fit on the screen of the ARM debugger. Each pixel will occupy a single bit of a word, so each memory word will contain 32 packed pixels. Thus, the entire memory footprint of the graphics "screen" will be a two-dimensional array containing 96 words, using three words per raster line.

Following this paragraph are new symbols to include at the beginning of your program. You ***must*** use these symbols in your code instead of special constants; I want to be able to change the `Rows` and `Columns` definitions and still have your program work after assembly with the new values. Note that `Cols` represents the number of 32-bit words per raster line, *not* the number of pixels. (You may use the explicit constant 32 when dealing with the number of bits per integer word, but not as the number of raster rows.)

```
Rows      EQU  32        ; Total rows of text
Cols      EQU  3         ; Memory words per row
MaxRow    EQU  Rows-1    ; Maximum allowed Row
MaxCol    EQU  Cols*32-1 ; Maximum allowed Col
Words     EQU  Rows*Cols ; Total words to reserve
Bytes     EQU  Words*4   ; Total bytes to reserve
```

Add new subroutine frames called `Set_Pixel`, `Clear_Screen`, and `Print_Screen` to the end of your program. Declare the `Screen` array representing the graphics display directly after these three subroutines (by declaring it there it will be within `LDR/STR` range of `Set_Pixel`) with:

```
Screen    %    Bytes
```

Neither the `Clear_Screen` nor the `Print_Screen` subroutines require any parameters. The `Clear_Screen` routine clears the `Screen` array to all zeroes, and the `Print_Screen` routine prints out the screen as 32 rows of 96 characters per row (using a modified version of `Print_Binary`, described later).

The `Set_Pixel` subroutine gets two input parameters through the registers: register `R0` contains the X value (column) and register `R1` contains the Y value (row) of the pixel bit to be set to 1. The upper-left pixel is at location `<0,0>`. X and Y values must be *clipped* to the screen; i.e., if the X and Y values are off-screen (X less than zero or greater than `MaxCol`, or Y less than zero or greater than `MaxRow`) then no pixel will be set.

You may not add other global symbols, code, or variables. All subroutines are required to be transparent with respect to register usage and must use the stack for temporary register storage.

Modify your existing `Print_Binary` routine to print "`.`" instead of "**`0`**" and "**`*`**" instead of "**`1`**". Modify `Plot_Line` to call `Set_Pixel` to draw a straight line on the grid between the two endpoints passed in (instead of just printing out the endpoints as numbers). The `Plot-Line` routine is fairly tricky to implement correctly, but there are several approaches that can be used.

Your main program must first call `Clear_Screen`, then `DeCasteljau` (to plot the Bézier curve with the values from the previous assignment), and finally `Print_Screen`.

When working, print out the `.ALI` assembly listing and a screenshot containing the source code after execution with **your name visible**, and the console window showing the resulting graphics screen. The graphics screen will occupy most of the window. Staple the screenshot on top of the listing. Here are the point penalties for this 20-point assignment (no assignment will score less than zero):

1. -5 for cosmetic errors: printouts not stapled, or screenshot not on top.
2. -20 for name not visible on `.ALI` listing (did you write the code?)
3. -5 for not printing the screen at all (does `Print_Screen` work?)
4. -5 for not correctly painting the Bézier curve (`Clear_Screen` & `Set_Pixel`)
5. -5 for non-transparent subroutines (are registers saved and restored?)
6. -5 for using explicit memory locations (did you use the stack?)

The final score will be treated as extra credit.

"EXTRA" EXTRA CREDIT

Each of these suggestions is optional and worth an additional +5 points when implemented correctly. You must test all new code to show that it works completely, showing appropriate screenshots in each case.

1. Implement Outlined Circles

Write a subroutine called `Outline_Circle` so that it will draw a circle centered at coordinate `<X,Y>` with radius `R`. The values are passed in through integer registers `R0`, `R1`, and `R2`, where `R0=X`, `R1=Y`, and `R2=R`. Here is a pseudocode template for creating outlined circles:

```
Procedure Outline_Circle (X,Y,R:Integer)
     Var XX, YY, SS : Integer { Local variables }
Begin
     XX := R
     YY := 0
     SS := -R
     While (XX >= YY) Do
          Begin
               Set_Pixel (X+XX, Y+YY)
               Set_Pixel (X-XX, Y+YY)
               Set_Pixel (X+XX, Y-YY)
               Set_Pixel (X-XX, Y-YY)
               Set_Pixel (X+YY, Y+XX)
               Set_Pixel (X-YY, Y+XX)
               Set_Pixel (X+YY, Y-XX)
               Set_Pixel (X-YY, Y-XX)
               SS := SS + YY + YY + 1
               YY := YY + 1
               If SS > 0 Then
                    Begin
                         SS := SS – XX – XX + 2
                         XX := XX - 1
                    End
          End
End
```

2. Implement Horizontal_Line

Write a subroutine called `Horizontal_Line` to plot horizontal lines from coordinate `<X1,Y>` to coordinate `<X2,Y>`. In this case the coordinates are passed in through integer registers `R0`, `R1`, and `R2`, where `R0=X1`, `R1=X2`, and `R2=Y`. As usual, all registers must be preserved (on the stack).

3. Implement Solid Circles

Write a subroutine called `Solid_Circle` so that it will draw a circle centered at coordinate `<X,Y>` with radius `R`. The values are passed in through integer registers `R0`, `R1`, and `R2`, where `R0=X`, `R1=Y`, and `R2=R`. Here is a pseudocode template for creating solid circles:

```
Procedure Solid_Circle (X,Y,R:Integer)
     Var XX, YY, SS : Integer { Local variables }
Begin
     XX := R
     YY := 0
     SS := -R
     While (XX >= YY) Do
          Begin
               Horizontal_Line (X-XX, X+XX, Y-YY)
               Horizontal_Line (X-XX, X+XX, Y+YY)
               Horizontal_Line (X-YY, X+YY, Y-XX)
               Horizontal_Line (X-YY, X+YY, Y+XX)
               SS := SS + YY + YY + 1
               YY := YY + 1
               If SS > 0 Then
                    Begin
                         SS := SS – XX – XX + 2
                         XX := XX - 1
                    End
          End
End
```