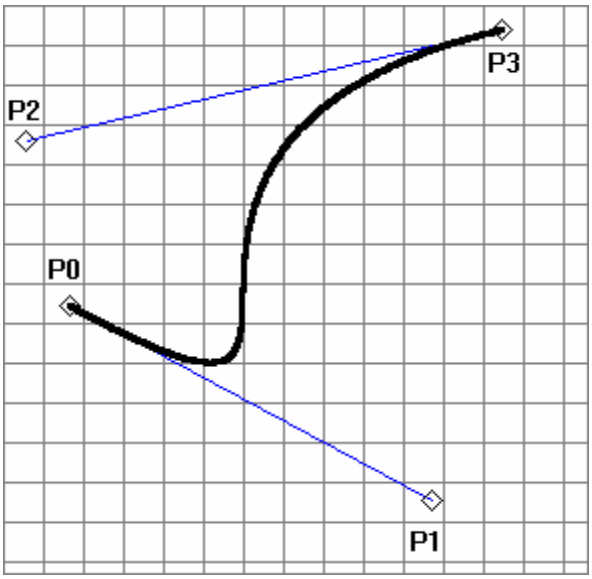
CMPSCI 201 – Fall 2006

Professor William T. Verts

Assignment #4 – Recursive Bézier Curves

The purpose of this exercise is to demonstrate your understanding of (nearly) everything we have covered throughout the semester. You will be graded on formatting and style, as well as on the correct operation of your program. We will be looking closely at efficiency of register usage, parameter passing mechanisms, use of symbolic labels as stack-offset constants, selection of appropriate variable names, thoroughness of commentary, etc., etc. We expect people to spend a considerable amount of time getting the appearance of their programs correct.

The problem we will solve is the computation of the points along a Bézier curve (a piecewise parametric cubic polynomial), using a technique called "DeCasteljau's Algorithm." Four points in space define a Bézier curve; `P0` and `P3` are the end-points; `P1` is `P0`'s control point, and `P2` is `P3`'s control point. The curve starts at `P0`, approaches but does not pass through `P1` and `P2`, and finally ends at `P3`. At each end-point, the curve is tangent to a line between that end-point and its corresponding control point, as you see in the image.



Bézier curves are used in graphics design because several of them can be stacked end-to-end; two curves will flow smoothly into each other only so long as there is a single line that passes through the common end-points and both of the corresponding control points. The equations (one for each dimension) are cubic polynomials of the form $p(t) = at^3 + bt^2 + ct + d$, where the coefficients a, b, c, and d for each equation are computed from the appropriate coordinates of the four points as follows:

$$a = p_3 - 3p_2 + 3p_1 - p_0$$
$$b = 3p_2 - 6p_1 + 3p_0$$
$$c = 3p_1 - 3p_0$$
$$d = p_0$$

Pseudo-Code

The DeCasteljau approach is computationally much simpler than polynomial evaluation (although it does use recursion). A pseudo-code procedure to show the general algorithm for two dimensions is as follows (each `Point` variable contains an *x* and a *y* coordinate value):

```
Procedure DeCasteljau (P0,P1,P2,P3:Point)

 Var P01  : Point   { Local mid-point variable }
     P12  : Point   { Local mid-point variable }
     P23  : Point   { Local mid-point variable }
     P012 : Point   { Local mid-point variable }
     P123 : Point   { Local mid-point variable }
     P0123: Point   { Local mid-point variable }

Begin
     If Distance(P0,P3) < Threshold Then
          Plot_Line (P0, P3)
     Else
          Begin
               P01   := (P0   + P1  ) / 2
               P12   := (P1   + P2  ) / 2
               P23   := (P2   + P3  ) / 2
               P012  := (P01  + P12 ) / 2
               P123  := (P12  + P23 ) / 2
               P0123 := (P012 + P123) / 2
               DeCasteljau (P0, P01, P012, P0123)
               DeCasteljau (P0123, P123, P23, P3)
          End
End
```

The advantage to this algorithm for graphics is that it can be written to use nothing but integers (and shifts for the divisions by two), since pixel positions on screen always use integer coordinates. For this assignment, however, you are to treat the point coordinates and the threshold value as ***single-precision floating-point numbers***. Each `Point` in the pseudo-code above is two coordinate values *x* and *y*, so point `P0` really represents two values `P0X` and `P0Y`, for example. Thus, the DeCasteljau procedure is called with eight floating-point numbers as parameters (`P0X`, `P0Y`, `P1X`, `P1Y`, `P2X`, `P2Y`, `P3X`, and `P3Y`).

Program Structure

This section describes both the basic layout of your program and the expected contents of each of the routines you must write. Follow these guidelines carefully; they will not only guide your design, they will also make debugging and grading much easier than otherwise possible.

Main Program

Your main program will call `DeCasteljau` with the coordinates of four points (eight single precision floating-point numbers) passed on the stack as call-by-value. Upon exit from the subroutine the parameters are discarded, and then the program ends.

Variables

Due to distance limits on the ARM between any `LDR` instruction and the variable it references, all global memory variables should appear between the main driver program and the `DeCasteljau` subroutine. The only global variables allowed are the initial values of the eight floating-point numbers loaded onto the stack before the initial call to `DeCasteljau` (and never used in any other context), and the `Threshold` value used by the `DeCasteljau` routine itself. The numbers are defined by the `DCFS` pseudo-instruction with initial values as follows:

```
P0X       DCFS  0.0
P0Y       DCFS  0.0
P1X       DCFS 15.0
P1Y       DCFS  0.0
P2X       DCFS  0.0
P2Y       DCFS 15.0
P3X       DCFS 15.0
P3Y       DCFS 15.0
Threshold DCFS  1.0
```

DeCasteljau

Convert the pseudo-code procedure into ARM assembly code and get it running. The DeCasteljau routine is recursive, has local variables, and uses the stack heavily. All memory references, save one, are to and from the stack. The only reference to a global variable allowed here is to `Threshold`; you *may not ever* reference directly the original point values in memory in this routine. A (strongly suggested) stack frame layout appears at the end of this document.

Distance

The `Distance` function has four arguments passed (by value) to it via floating-point registers `F0`, `F1`, `F2`, and `F3`. These arguments represent arbitrary points in the plane $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$, respectively. The routine computes and returns the Euclidean distance between the points as $\texttt{Sqrt}((x_2-x_1)^2+(y_2-y_1)^2)$ in floating register `F0` (register `F0` has dual use as a call-by-value input parameter and as a call-by-return output result).

Plot_Line

The `Plot_Line` procedure is passed (by value) two points $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$, via floating-point registers `F0`, `F1`, `F2`, and `F3`, respectively. It then truncates the four arguments to integer form, calls `Print_Signed` four times, along with some extra characters printed for clarity, and finally calls `Print_LF` to end the current line. The printed result for a call such as

`Plot_Line(2.7, 3.6, 5.2, 7.1)` must be **`<+2,+3> -- <+5,+7>`**, for example. The two printed points might be to the same "pixel" (or to adjacent pixels).

Print_Number, Print_LF, etc.

Copy your code for the `Print_Signed`, `Print_Unsigned`, `Print_Blank`, and `Print_LF` routines from the previous assignment, along with helper routine `UDiv10`. You will not need `Print_Binary`, `Print_Word`, or `Print_Nybble`. You may not change the definitions of any of these routines. You may improve the code of these routines as necessary, but their functionality must not be altered from the previous assignment. To support `Plot_Line`, you must create new subroutines here called `Print_Dash`, `Print_Less`, `Print_Greater`, and `Print_Comma` to print the appropriate characters.

General Layout and Rules

The main program will be followed by the variables, the `DeCasteljau` procedure, the Euclidean `Distance` function, the `Plot_Line` procedure, the printing routines you developed in the previous assignment, and all new printing routines. Items must appear in this order:

1.	main driver	(program)
2.	global variables	(nine floating-point numbers)
3.	`DeCasteljau`	(new subroutine)
4.	`Distance`	(new subroutine)
5.	`Plot_Line`	(new subroutine)
6.	`Print_Signed`	(old subroutine)
7.	`Print_Unsigned`	(old subroutine)
8.	`Print_LF`	(old subroutine)
9.	`Print_Blank`	(old subroutine)
10.	`Print_Dash`	(new subroutine)
11.	`Print_Less`	(new subroutine)
12.	`Print_Greater`	(new subroutine)
13.	`Print_Comma`	(new subroutine)
14.	`UDiv10`	(old subroutine)

No subroutine may reference any values on the stack outside of their own local stack frame. For example, when `DeCasteljau` calls `Distance`, the `Distance` subroutine may not reach deep into the stack to get at `DeCasteljau`'s variables. All subroutines must be transparent in all registers used, as usual.

You must submit a screenshot showing that the program has been run and belongs to you. It must contain enough of the source code to show your name and with the execution bar on the `SWI &11` instruction. The console window must show the output lines printed by `Plot_Line`. You must also show the sixteen integer registers and the eight floating-point registers. Staple your complete `.ALI` listing file to the screenshot.

Stack Frame for DeCasteljau

In order to help you design your program, and get the parameter passing mechanism set up correctly, I recommend the following stack layout for the calls to `DeCasteljau`. Remember that in the ARMulator the `IP` register is specified in lower case as the `ip` register.