A SuperBASIC example using the 'mouse' command and a bunch of pokes and peeks

A low-budget stand-alone SuperBASIC example

This program was developed to suit the same use case as the assembly language project discussed above. It does not accomplish as much, but reads the mouse, updates registers, and reacts to button presses.

SuperBASIC's mouse support is confusing, or at least, the manual is. There are two commands referenced, '**mouse**' and '**mdelta**'. We will use the former; it is lightweight and has handful of features.

As functional as this little program is, there is nothing fancy going. The intent was to squeeze as much functionality into about 40 lines of setup and logic.

Examining the general structure, you'll notice that none of SuperBASIC's graphic features are leveraged. Instead, direct memory pokes are used to both plug the shape data into memory, and to configure sprite attributes. You might consider this approach a bridge to assembly language as it will acclimate you to the process of defining and manipulating sprites.

With an Ernesto Contreras donated sprite (the hand from the Foenix Sprite Editor), the program fills the screen with 16 x 16 pixel (3 color) sprites and then enables the hardware mouse pointer. Data for the sprites is loaded into memory at $7900 (all 64 sprites point to the same image); and a block of code spaces them 20 pixels apart, in 8 rows of 8.

The sample program had three design goals; a) fit on a single screen (when listed); b) be clear enough to understand. e.g. inserted ':' chars between routines and nesting where space allowed; c) leverage mouse features and do so using simple coding such that it can be easily repurposed. I hope that you will find that this program delivers on all three objectives.

Code review in detail

Once initialized, an endless loop is entered from lines 680-910 (you'll need to ctrl-C or 'BREAK' to exit*).

The SuperBASIC manual explains that the **mouse** command returns 6 arguments, x, y, and z positions, and then left, right, and center mouse button status.

Importantly, the values returned do not resemble the registers discussed above or even MicroKernel event data. Rather, a 320 x 240 pixel scale is used for mouse position (but does not move the pointer). A range of 0..255 is used for the z-axis (scroll wheel); and finally, a value of -1 is returned when buttons are depressed.

At first click, a single line of text will appear on the first line of the screen, detailing sprite RGB color values. Pressing the left, right, and center button (or pressing the scroll wheel if you have one) will rapidly advance the byte value of the respective color.

The scroll wheel shifts the array of sprites up or down; as a row gets close to the bottom or top of the screen, you'll notice some unusual north/south wrapping. This, due to low budget code on lines 810-860.

Lines 690, 710-790 convert from the *x* and *y* positions reported by the SuperBASIC **mouse** command to the VICKY mouse *x* and *y* register pairs. We need to scale and then convert from 8-bit to 16-bit values.

Block from 500-530:

- binds the address of the bitmap to sprite # 0 .. 63 with the low byte set to 0, bits 8..15 set to $79, and bits 16..18 (the highest byte) set to 0. This points each sprite to the shape data located at $7900
- Within the loop, value of 65 is written to each sprite's attribute byte to enable (bit 0), and to select the 16 x 16 size
- 63 is stored to the VICKY master control register ($D000), to enable graphics modes and text and the cursor is turned off. Just like the old days, sprite data is poked into memory from data statements

Lines 550-610 set the *x* and *y* coordinate for each of the 64 sprites, through a for/next loop. With a starting position of 80, 80 sprites are laid out 8 on a horizontal line, 20 pixels apart. Similar logic is used within the scroll wheel sprite positioning code

The code block from 630-660 establishes a color palette after selecting the character set and graphics LUT bank of `mmu_io_ctrl`

* Upon BREAK, you'll want to restore order by issuing a **cursor on** and **sprites off**. If you plan on modifying this code, be sure to save your work frequently. An incorrect `mmu_io_ctrl` or errant **poke** can lock up your machine (just like in the *good ole* days)

In the Commodore 64 days, well before BASIC 3.5 added graphics commands to the Commodore Ted series, all "we kids" had was the Programmer Reference Guide which contained a few sample programs (the famous "Commodore Balloon", was one).

There were ways to add commands to the Commodore platform, but I never much liked them; also, I could not afford them. In the context of the F256 platform, I urge you to study (and experiment) with Peter Weingartner's expertly written reference manual and some good old fashioned poke and peek commands. Doing so will broaden your understanding of the platform and help to reach beyond that which even SuperBASIC can achieve. The least of these (otherwise unreachable) features include alternate text modes (40 x 30 or 60 x 30) and redefined characters, but also, the Commodore SID synthesizer ICs, the serial port, and real time clock. The original F256 platform is loaded with features and as a retro-computer, it really is *all you need*.

- Three colors are set, based on `spreditjr.bas`'s sprite data: color #1 (black @ 0,0,0); color #4 (grey @ 64, 64, 64) and color #7 (white @ 255,255,255). Color #7 will altered based on mouse button press

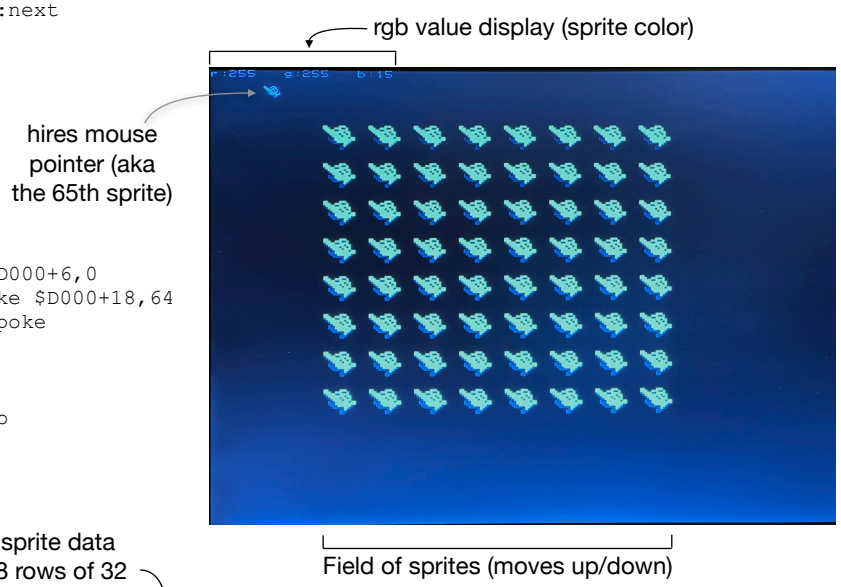
Main loop (680-910) - the core of the program

- Begins by setting 'lc' (last c) to that of 'c', the scroll wheel axis. The sprite move routine is expensive since it iterates for 64 counts. It incorporates math and a decision statement; by checking for a change in *z*-axis from the prior loop, we can avoid lines 820-860 if no change to scroll wheel has taken place
- The **mouse** command returns a, b, and c for the *x*, *y*, and *z* axis and a1, b1, c1 representing buttons
- Since the *x* and *y* axis are scaled to 0..319 and 0..239 respectively, we must **multiply each by 2** in order to scale to match the mouse geometry. In doing so, we unconditionally **poke** potentially two-byte values into low-byte registers. SuperBASIC does not do any checking. Of course, this is not good software programming, but we are on a budget. We will deal with the high-byte of this coordinate pair in the next code block.
- Block 710-750 compares the scaled a (x-axis) register to 511; if greater, pokes a 2 into the high-byte x register. If not 511, but greater than 255, a 1 is poked in, else 0. (block 770-790 is similar but for the y-axis and must only check whether > 255)
- Block 810-860 manages the *z*-axis mentioned above. A test compare against the prior value. If it has changed, all sprites are adjusted with a new *y* value. This routine is expensive so we employ the test to avoid running it on every iteration
- Block 880-900 checks for button press and if clicked, increases the current color value of the RGB set by a value of 1. Calls proc `upd_rgb` to print the new value to the top of the screen

Lines 2000-2070 simply contain byte values for each of the 256 pixels of the 16 x 16 sprite

To quote Paul Robson, "here *endeth* the lesson" :)

```
500   cls :poke 1,0:for x=0 to 63:poke $D900+(x*8)+1,0:poke $D900+(x*8)+2,$79
510   poke $D900+(x*8)+3,0:poke $D900+(x*8),65:next :poke $D000,63:cursor off
520   poke $D6E0,1:poke $D6E2,70:poke $D6E4,70
530   for x=0 to 255:read y:poke $7900+x,y:next
540   :
550   x=80:y=80:for n=0 to 63
560   poke $D900+(n*8)+6,y
570   poke $D900+(n*8)+7,0
580   poke $D900+(n*8)+4,x
590   poke $D900+(n*8)+5,0
600   x=x+20:if x>230:x=80:y=y+20:endif
610   next
620   :
630   poke 1,1
640   poke $D000+4,0:poke $D000+5,0:poke $D000+6,0
650   poke $D000+16,64:poke $D000+17,64:poke $D000+18,64
660   poke $D000+28,255:poke $D000+29,255:poke
        $D000+30,255:poke 1,0
670   :
680   lc=c:mouse a,b,c,a1,b1,c1
690   a=a*2:poke $D6E2,a:b=b*2:poke $D6E4,b
700   :
710   if a>511 poke $D6E3,2
720      else if a>255 poke $D6E3,1
730               else poke $D6E3,0
740           endif
750   endif
760   :
770   if b>255 poke $D6E5,1
780      else poke $D6E5,0
790   endif
800   :
810   if lc<>c
820      x=0:y=c:for n=0 to 63
830         poke $D900+(n*8)+6,y
840         x=x+1:if x>7:x=0:y=y+20:endif
850      next
860   endif
870   :
880   if a1=-1:poke 1,1:poke $D01E,(peek($D01E)+1):upd_rgb():endif
890   if b1=-1:poke 1,1:poke $D01C,(peek($D01C)+1):upd_rgb():endif
900   if c1=-1:poke 1,1:poke $D01D,(peek($D01D)+1):upd_rgb():endif
910   goto 680
920   :
930   proc upd_rgb()
940   print "                    ";
950   for x=1 to 21:print chr$(2);:next
960   print "r:";peek($D01E),"g:";peek($D01D),"b:";peek($D01C);
970   for x=1 to 21:print chr$(2);:next :poke 1,0
980   endproc
```



sprite data (8 rows of 32 = 256 bytes)

```
2000  data 0,1,1,0,0,0,0,0,1,1,1,0,0,0,0,0,1,7,7,1,0,0,1,1,7,7,7,1,0,0,0,0
2010  data 1,7,7,7,1,1,7,7,7,1,4,7,1,0,0,0,1,4,7,7,7,7,1,4,7,7,7,7,1,0,0,0
2020  data 0,1,4,7,7,7,7,7,7,7,1,7,7,1,0,0,0,0,1,4,7,7,7,7,1,7,7,1,7,1,0,0
2030  data 0,0,1,1,4,7,7,7,7,1,7,7,7,1,1,0,0,0,0,1,4,1,4,7,7,7,7,7,7,1,7,7,1
2040  data 0,0,1,4,4,4,7,7,7,7,7,7,7,7,1,1,0,0,0,0,1,4,4,4,7,7,7,7,7,7,4,1,0
2050  data 0,0,0,0,0,1,4,4,4,4,4,1,7,7,4,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,4,7,4,1,0,0,0
2060  data 0,0,0,0,0,0,0,0,1,4,4,4,4,1,0,0,0,0,0,0,0,0,0,0,0,0,1,4,4,1,0,0,0,0,0,0
2070  data 0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Pro tip: double-clicking the 'outer' button will swap the *handedness* of the mouse, aka change a right hand mouse to a lefty. This is a feature, not a bug! When you read the full article, this will all be clear :)