XGameStation Micro Edition
Floormapped Planes
By Michael Ollanketo

In this lesson you will learn how to create an infinte plane effect on the XGameStation Micro Edition (XGSME). Differences between NTSC and PAL will be pointed out in this document.

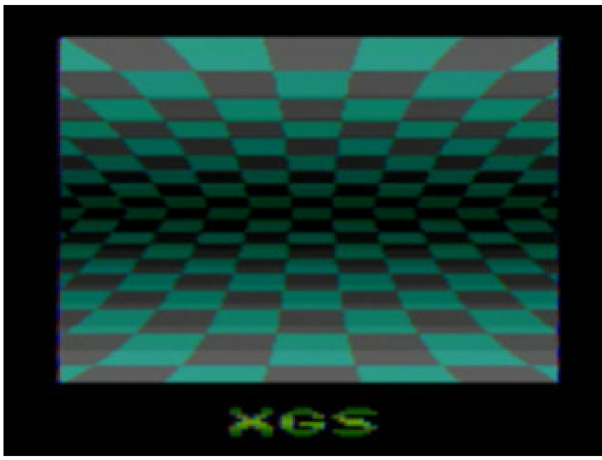

Figure 1: NTSC version

Figure 2: PAL version

These images were grabbed through a TV-card, and the noise you see is because of the card and will not be present on an actual TV. The reason why the NTSC and PAL versions have different colors is that color is handled differently on NTSC and PAL devices, so getting the same results regardless of the format would require some extra work.

What this demo does is the following: An XOR pattern is mapped onto a surface according to a simple scaling algorithm to create the illusion of an infinite plane. The plane is shaded and mirrored across the center of the screen to make it look like the inside of a cylinder or some other curved shape. An XGS logo is drawn at the bottom of the screen, in the overscan area. The NTSC version of the demo also slowly fades the hue to give a wider range of colors. All of this will be explained throughout this tutorial.

As in the previous lesson, we begin with some assembler directives to set things up for the XGSME.

Legend

`; n`	Instruction takes n cycles
`; m (n)`	Instruction takes m cycles, and the entire group it belongs to takes n cycles
`foobar`	NTSC-specific code
`foobar`	PAL-specific code

```
      TITLE "Plane demo v02"
; Textured/shaded planes for the XGS ME
; Works only at 80 MHz


; This file needs 5 include files:
;     general_define.src:     Defines constants (EQU) and some system variables.
```

```
;     general_macro.src:      Useful general purpose macro definitions and
;                             system functions.
;     ninja.mus               Music data.
;     xgslogo.src             XGS logo bitmap.
;     xgsmp.src               XGS music player.

      DEVICE      SX52, OSCHS3, XTLBUFD, IFBD   ; Set everything for the XGS ME
      RESET Start
      FREQ  80_000_000  ; this is a directive to the ide only
                        ; if you want to put the XGS ME into RUN mode
                        ; you must make sure you go into the
                        ; device settings and make sure that
                        ; HS3 is enabled, and crystal drive and
                        ; feedback are disabled and then re-program
                        ; the chip in PGM mode and then switch it to RUN

      IRC_CAL IRC_FAST  ; Prevent a warning
      ID    "micplane"  ; ID string
```

Each register bank can hold up to 16 words. Normally I choose to put the variables declared in general_define.src in bank $20, and put my program-specific variables in banks $30 and up. The music player (xgsmp.src) uses banks $A0-$C0 for its variables by default. If you need to move several variables from one bank to another you can use the global registers `10-15` in bank 0. That way you can save yourself some bankswitches.

```
; Global variables, starting at bank #2
            org   $20
include           "general_define.src"

      org $40
pixel       ds    1
xCnt        ds    1
yCnt        ds    1
temp        ds    1
cnt1        ds    1
cnt2        ds    1
u_w         ds    1     ; u whole
u_f         ds    1     ; u fractional
v_w         ds    1     ; v whole
v_f         ds    1     ; v fractional
u2_w        ds    1
u2_f        ds    1
du_w        ds    1     ; delta u whole
du_f        ds    1     ; delta u fractional
dv_w        ds    1
dv_f        ds    1
      org $50
temp2       ds    1
vertPtr2    ds    1
chroma_cnt  ds    1
chroma_delta ds   1
temp3       ds    1
temp4       ds    1
u_w2        ds    1
u_f2        ds    1
v_w2        ds    1
v_f2        ds    1
one_u2      ds    1
one_v2      ds    1
```

```
du_w2       ds    1
du_f2       ds    1
dv_w2       ds    1
dv_f2       ds    1


include     "general_macro.src"
include     "ninja.mus"       ; Include song data
include     "xgsmp.src"       ; Include music player

            org   $0          ; Set the start of the program code
            org   $+2         ; leave 2 free word here for the debugger


; Jump table
InitMusic   jmp @XgsMpInit
UpdateMusic jmp @XgsMpUpdate

Start       ; Our real code starts here
```

To write to a variable you need to select the appropriate register bank. The `BANK` instruction copies bits 4..6 of the operand into *`FSR`* (the File Select Register). If you want to select one of the upper 8 banks (bank $80-$F0) you have to set bit 7 of `FSR` yourself, or you can use the `_bank` macro in general_macro.src which will do this for you. The `CLR` instruction is a simpler way of saying `MOV fr,#0`. Using `CLR` will save you one cycle and one instruction word, so there's a simple 50% optimization for you!

```
            ; Initialize variables
            bank $50
            mov chroma_cnt,#0
            mov chroma_delta,#1
            clr v_w2
            clr v_f2

            bank #$40
            clr yCnt
            clr v_w
            clr v_f

            ; Initialize music player. Song data starts at $800
            mov 10,#$00
            mov 11,#$08
            call InitMusic
            _bank $20

            INITIALIZE_VIDEO       ; Initialize I/O controller for video
```

This is the starting point of our frame (or rather, field) rasterizer. Each field is composed of 262.5 lines in NTSC mode and 312.5 lines in PAL mode. I have chosen to ignore the fractional part and settle for 262 and 312 lines per field, respectively. Out of those 262 (312) lines, 184 are made up of the actual effect, 4 (3) are used for vertical sync, 42 (65) are used for the bottom overscan and the last 32 (60) lines are used for the top overscan.

```
Begin_Raster

            ; Initialize texture coordinates and deltas
            bank $40
            clr u_w                ; u (horizontal), whole part
            clr u_f                ; u, fractional part
```

```
            clr du_w                 ; delta u, whole part
            mov du_f,#64             ; delta u, fractional part (64 <=> 0.25)
            clr dv_w                 ; delta v, whole part
            mov dv_f,#64             ; delta v, fractional part

            bank $20


            ; Upper half
```

The texture coordinates and deltas are all 8.8 fixed point numbers. If you're unfamiliar with fixed point, this basically means that you can represent real numbers instead of just integers, and that you can do so with 8 bits of precision for the whole (integer) part and 8 bits of precision for the fractional ("decimal") part. As the name implies, the decimal point is at a fixed position unlike the floating point numbers which traditionally are used to represent real numbers on computers. This feature makes fixed point numbers very straightforward and efficient to work with, at the expense of lower precision. The fixed point variables are made up of pairs of variables, with the whole part having a `_w` suffix and the fractional part having an `_f` suffix.

As you may have learned in your SX52 studies, the `OPTION` register controls interrupts and the prescaler setting of *RTCC* (the RealTime Cycle Counter). Bit 7 of `OPTION` should always be set, and bits 4 and 5 should be clear. Bit 6 is a master interrupt flag and is active low, so we set the bit to disable interrupts. Bits 0..2 allows for specification of 8 different prescaler settings ranging from 2 (000) to 256 (111). The prescaler is used when bit 4 is clear, otherwise a prescaler of 1 is implied.

```
            mov scanline, #92              ; Render 92 scanlines
            mov !OPTION, #%11000100        ; Sets prescaler to 1:32
Raster_Loop1      ; Loop here for each scanline
            PREPARE_VIDEO_HORIZ burst_phase ; Prepare required video signal for
                                           ; a scanline
            PREPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC                 ; 1
            mov RE,#BLACK            ; 2 (3)
```

After using the `PREPARE_VIDEO_HORIZ` macro to set up the current scanline, we reset the `RTCC` and output black for a while. The SX52 has a number of ports that you can use to interface with other devices. For example, port `RA` is used both for sound output and joystick input, while port `RE` is used to send video data. Before you start using a port you need to set it up for reading or writing – this is what the `INITIALIZE_VIDEO` macro does for the video port.

What I was looking for here was to create the illusion of perspective. What's an easy way to do that? Make things that are closer look bigger! So we need a low-complexity algorithm to scale things up as they come "closer" to the viewer. My solution to the problem was this: use a coordinate `C={u,v}` and two deltas `D={u',v'}` and `E={u",v"}`. For each pixel we do `C := C + {u',0}`. For each scanline we do `C := C + {0,v'}` and `D := D + E`. The horizontal origin (`u`) is also moved back every scanline. This creates a shape that narrows down every scanline. If we draw it bottom-up, we get this:

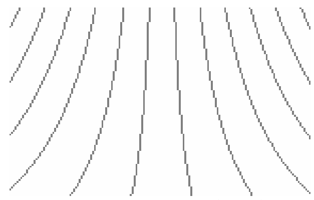

Figure 3: Texture bending

Our texture is just a simple XOR pattern, which means that we can generate it on-the-fly rather than storing it in memory. To create the effect of having a bunch of large tiles I use bit 3 of the texture coordinates in the texel generation. This way we get a virtual texture with 8*8 dot blocks laid out in a chessboard manner. And when we apply this texture on our virtual surface we get the following result:

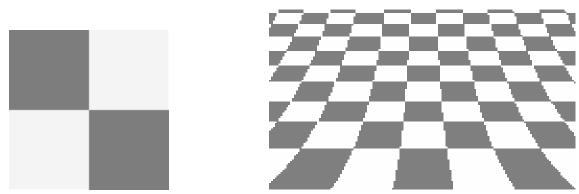

Figure 4: Texture applied to plane

This is all good, but it looks a bit bland. To give you and idea of what I mean I'm going to let you look at a picture.

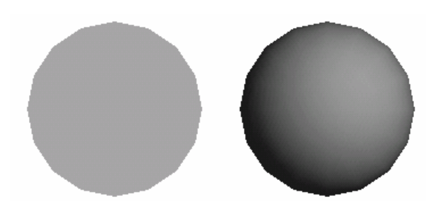

Figure 5: Shading

Which of these objects looks more like a sphere? Probably the one on the right. This is because the object on the right provides our brain with important cues about dimension and depth, whereas the one on the left looks completely flat. To create the shading effect on the planes I use a table with 92 entries – one for each scanline. The table contains raw luma values, ranging from 7 to 15. After applying shading to our initial effort we end up with something like this:

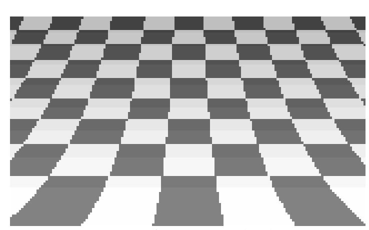

Figure 6: Shaded plane

Big improvement, don't you think? So let's write the code to do all this.

```
            bank $20                ; 1
            mov M,#$0C        ; Page C holds the shading table (92 entries)
            mov W,scanline          ; Select table offset
            iread                   ; Read one word
            bank $40
            mov pixel,W             ; Save the lower 8 bits of the word read

            bank $20
            DELAY(500)

            bank $40                ; 1
            mov u2_w,u_w            ; Make a copy of u_w
            mov u2_f,u_f            ; ...
            mov xCnt,#160           ; Number of pixels per row
```

I have written a macro called `ADD_8_8` to add two 8.8 fixed point numbers – i.e. 8 bits for the fractional part and 8 bits for the whole part. The two first operands contains the first numbers, and the last two operands contains the second number. The result of the addition will be written to the first number. The macro itself looks like this:

```
add \2,\4
snc
inc \1
add \1,\3
```

What this does is the following: the 8 fractional bits of the second number (operand 4) are added to the fractional part of the first number (operand 2). If this addition generated carry we need to increment the whole part. Then we add the whole parts together and we are done. An example of how this works:

Add $04C3 (4.76171875) and $0250 (2.3125)

1. $C3 + $50 = $113
2. Carry was set, so increment first number => $05
3. $05 + $02 = $07

The result is $0713 = 7.07421875 = 4.76171875 + 2.3125 OK!

```
draw_scanline
            mov W,u2_w              ; 1. u (whole)
            xor W,v_w               ; 1. v (whole)
            and W,#8                ; 1. We're only interested in bit 3

            mov temp,W              ; 1. Save W
            mov W,#(15*16)          ; 1. Use color15 if ((u ^ v) & 8) != 0
            sb temp.3
            mov W,#(5*16)           ; 1. Use color5 if ((u ^ v) & 8) == 0
            nop
            or W,pixel              ; 1. OR in some luma

            mov RE,W                ; 1

            ADD_8_8 u2_w,u2_f,du_w,du_f   ; 6. 8.8 fixed point addition
            djnz xCnt,draw_scanline

scanline_done
            mov RE, #BLACK

            ADD_8_8 v_w,v_f,dv_w,dv_f     ; v += delta v
```

```
            add dv_f,#3             ; delta v += 0.01171875
            snc
            inc dv_w                ; increase whole part if necessary

            add du_f,#1             ; delta u += 0.00390625
            snc
            inc du_w

            mov u2_w,#$FF
            mov u2_f,#$B0
            ADD_8_8 u_w,u_f,u2_w,u2_f     ; u -= 0.3125

            bank $20
            cjb RTCC,#131,$               ; Wait for this scanline to end
            DELAY(8)
            cjb RTCC,#130,$
            djnz scanline, Raster_Loop1   ; Loop for the next scanline
```

The `RTCC` is used to time the length of each scanline in a very simple manner. We just use the `CJB` instruction to loop until x*prescaler cycles have passed.

After we have finished drawing the upper plane we draw another plane in the same way, but flip it across the x-axis. Since the planes move in the same direction, it will look like they are part of one solid object. And with the shading we apply you won't be able to see where one plane ends and the other begins.

```
            ; Bottom half

            mov scanline, #92             ; Render 92 scanlines
Raster_Loop2      ; Loop here for each scanline
            PREPARE_VIDEO_HORIZ burst_phase     ; Prepare required video signal
                                                ; for a scanline
            PREPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC                ; 1
            mov RE,#BLACK                 ; 2 (3)

            bank $20
            mov M,#$0C
            mov count1,#92
            sub count1,scanline
            mov W,count1
            iread                         ; Read from shading table
            bank $40
            mov pixel,W

            bank $20                      ; 1
            DELAY(496)
            bank $40                      ; 1
            mov u2_w,u_w
            mov u2_f,u_f
            mov xCnt,#160

draw_scanline2
            mov W,u2_w                    ;1
            xor W,v_w                     ;1
            and W,#8                      ;1

            mov temp,W
            mov W,#(15*16)
            sb temp.3
```

```
            mov W,#(5*16)
            nop
            or W,pixel
            mov RE,W                      ;1

            ADD_8_8 u2_w,u2_f,du_w,du_f   ;6
            djnz xCnt,draw_scanline2
```

This is where we do the per-scanline increments of delta u and delta v. Note that all increments are inverted compared to those used for the upper plane. This is because we are rendering the bottom plane in the opposite direction.

After doing the increments we wait for this scanline to end and then loop back to draw the next line. After all 92 lines of the plane has been drawn we jump to the next ROM page and start drawing the bottom overscan.

```
scanline_done2
            mov RE, #BLACK

            ; v += delta v
            add v_f,dv_f
            snc
            inc v_w
            add v_w,dv_w

            ; delta v -= 0.01171875
            add dv_w,#$FF
            add dv_f,#$FD
            snc
            inc dv_w

            ; delta u -= 0.00390625
            add du_w,#$FF
            add du_f,#$FF
            snc
            inc du_w

            mov u2_w,#$00
            mov u2_f,#80
            ADD_8_8 u_w,u_f,u2_w,u2_f     ; u += 0.3125

            bank $20
            cjb RTCC,#131,$
            DELAY(8)
            cjb RTCC,#130,$
            djnz scanline, Raster_Loop2   ; Loop for the next scanline


            ; page_200 trick: this clunky trick is to prevent involuntary jumps
            ; between a page boundary.
            jmp @page_200
IF $ > $200
ERROR "Page Spillage!"
ENDIF

;###############################################################################


org $200
page_200
```

```
            page $                   ; Set the new page

            bank $20

            bank $40
            mov cnt2,#(34+8)         ; Number of scanlines
            mov cnt2,#(34+31)
:Vblank_Loop1
            bank $20
            PREPARE_VIDEO_HORIZ burst_phase
            PREPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC
            bank $40
            cjae cnt2,#30, :Black_line1
            cjb  cnt2,#14, :Black_line2
```

As in the previous lesson we have an XGS logo stored as an 24*8 pixel bitmap with 1 bit per pixel, and the least significant bit of each byte holding the leftmost pixel. The logo is drawn in three loops, each of which will draw 8 pixels. Every iteration of the inner loops one bit is shifted out using the `RR` (Rotate Right) instruction. This bit will end up in the `C` flag, which we can use to determine if we the pixel should transparent (black) or not.
The logo is scaled up to 24*16 by only incrementing the `yCnt` variable which selects the line within the logo on even scanlines. Refer to this figure:

Figure 7: Scaled logo

This line-doubling serves the purpose of making it seem as if we have a very large bitmap, even though it only uses 24 words of ROM space. Obviously we end up with a very blocky result since we're not using any kind of filter, but with regular TVs being relatively unsharp we can get away with this. A gradient that is calculated at runtime is also applied to the logo to further improve its look.

```
            ; Draw XGS overscan logo

            mov RE,#BLACK      ; 2. Output black
            mov cnt1,#8        ; 2. Number of pixels
            mov M,#LOGO_PAGE   ; 1
            mov W,yCnt         ; 1
            iread              ; 4. Read first byte
            mov pixel,W        ; 1

            ; Calculate the logo gradient
            mov temp,cnt2      ; 2
            sub temp,#22       ; 2. 22 = first_line-char_height = 30-8
            sb temp.7          ; 1. Skip if the result is negative
            not temp           ; 1. Invert bits
            mov W,#7           ; 1. Keep lower 3 bits
            and temp,W         ; 1
            add temp,#(COLOR11+2)   ; 2. Add base color

            bank $20           ; 1
            DELAY(1614 - 2 - 14 - 8 - 1 - 10)
            bank $40           ; 1
```

```
:Draw_bits_0_7
            mov W,temp        ; 1
            rr pixel          ; 1. Place lsb in C
            sc                ; 2 / 1. Skip if bit is set
            mov W,#BLACK      ; 1. Bit was clear, should be black
            mov RE,W          ; 1. Output color
            DELAY(31)
            djnz cnt1,:Draw_bits_0_7 ; 4 / 2
            mov RE,#BLACK     ; 2

            ; Read the next byte
            mov cnt1,#8       ; 2
            mov M,#LOGO_PAGE  ; 1
            inc yCnt
            mov W,yCnt        ; 1
            iread             ; 4
            mov pixel,W       ; 1 (10)
:Draw_bits_8_15
            mov W,temp        ; 1
            rr pixel          ; 1
            sc                ; 2 / 1
            mov W,#BLACK      ; 1
            mov RE,W          ; 1
            DELAY(31)
            djnz cnt1,:Draw_bits_8_15 ; 4 / 2
            mov RE,#BLACK           ; 2

            ; ..and finally the last one
            mov RE,#BLACK           ; 2
            mov cnt1,#8             ; 2
            mov M,#LOGO_PAGE        ; 1
            inc yCnt
            mov W,yCnt              ; 1
            iread                   ; 4
            mov pixel,W             ; 1 (12)
:Draw_bits_16_23
            mov W,temp ;#(COLOR14+5) ; 1
            rr pixel                ; 1
            sc                      ; 2 / 1
            mov W,#BLACK            ; 1
            mov RE,W                ; 1
            DELAY(31)
            djnz cnt1,:Draw_bits_16_23 ; 4 / 2
            nop               ; 1
            nop               ; 1

            mov RE,#BLACK     ; 2
            sub yCnt,#2       ; 2. Set yCnt back to its prior value

            ; yCnt is increased by 3 on even scanlines
            ; (logo is 24 pixels wide = 3 bytes/row)
            mov W,cnt2              ; 1
            not W
            and W,#1                ; 1
            add yCnt,W              ; 1
            add yCnt,W              ; 1
            add yCnt,W              ; 1

            bank $20
            jmp :Next_line          ; 3

:Black_line1
            bank $20
            mov RE, #BLACK          ; ( 2 cycles ) sync
```

```
            jmp :Next_line
:Black_line2
            bank $20
            mov RE, #BLACK          ; ( 2 cycles ) sync
            jmp :Next_line

:Next_line
            bank $40
            cjb RTCC,#131,$
            DELAY(8)
            cjb RTCC,#131,$
            djnz cnt2, :Vblank_Loop1
            ; END BOTTOM SCREEN OVERSCAN
```

After drawing the bottom overscan we send out a sync signal to do the required vertical syncing of the video signal. Again, we use the `RTCC` to time the length of this period, and within the vertical sync period we can do whatever we want since we don't have to worry about generating video.

```
            ; VERTICAL SYNC PULSE
:Begin_Blank
            mov RE, #SYNC           ; Send sync signal
            mov !OPTION, #%11000111 ; Turns off interrupts, sets prescaler to
                                    ; 1:256
            clr RTCC                ; Start RTCC counter

;##############################################################################


            ; The idea here is to change the color burst phase every
            ; 128th frame (when chroma_cnt hits zero). The burst phase
            ; is either incremented or decremented depending on the value
            ; in chroma_delta. The delta is inverted (negated) when the
            ; color burst is 0 or 13.
            bank $50
            mov W,chroma_delta
            dec chroma_cnt
            decsz chroma_cnt
            jmp no_chroma_change
            bank $20
            swap burst_phase              ; Swap nibbles
            add burst_phase,W             ; Add delta
            and burst_phase,#15           ; Only four bits of chroma
            cje burst_phase,#13,invert_delta
            cje burst_phase,#0,invert_delta
            jmp dont_invert
invert_delta
            bank $50
            not chroma_delta
            inc chroma_delta              ; not+inc == neg
            bank $20
dont_invert
            swap burst_phase              ; Swap nibbles back to original order
            or burst_phase,#BLACK_LEVEL   ; OR in some base luma
no_chroma_change

            jmp @page_400
IF $ > $400
ERROR "Page Spillage!"
ENDIF

;##############################################################################
```

```
org $400
page_400
        page $
```

At the end of each field we increment the vertical starting position within the texture. This will make it appear as if the planes are moving. Take a look at this figure to see the effect of this:

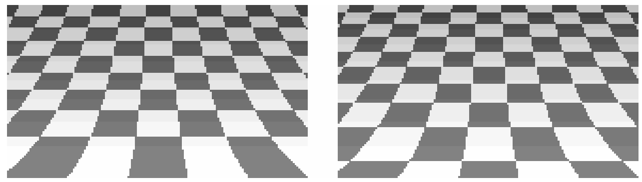

Figure 8: Scrolling plane

After that we call the music player to let it do any necessary updates before we wait for the vertical sync period to end. The vertical sync period is exactly 60*256 cycles = 192 µs = 3 scanlines in PAL mode, while it's closer to 4 scanlines in NTSC mode. You are free to experiment with other settings.

```
        ; Change the vertical texture coordinate every frame to create
        ; the illusion of motion.
        bank $50
        add v_f2,#$30     ; v += 0.1875
        snc
        inc v_w2
        mov W,v_w2
        bank $40
        mov v_w,W
        bank $50
        mov W,v_f2
        bank $40
        mov v_f,W

        bank $40
        clr yCnt

        mov 10,#$00
        mov 11,#$08
        call @UpdateMusic
        _bank $20

        ; We are done, now wait for the remaining time to finish this video
        ; frame and prepare overscan (bottom and top) for the next!
        cjb RTCC, #79, $  ; Wait for remaining vsync (79*256 cycles = 20224,
                          ; close enough)
        cjb RTCC, #60, $
        ; END VERTICAL SYNC PULSE


        mov !OPTION, #%11000100
```

Now we set the prescaler back to 1:32 and draw the top overscan. Once we've drawn the top overscan we have completed one field and can start rasterizing the next field.

This also concludes this lesson which has been the second in this series. The key point in this lesson was the use of fixed point numbers, which are of great use on processors that lack floating point hardware. This lesson also indirectly covers the basics of affine texturemapping, which will be put to use in coming lessons. As you may have noticed, there is a lot of code which is shared between the different lessons. It is a good idea to make up a skeleton of code that fits your style of programming, and then you can "fill in the blanks" for each program you write, rather than starting from scratch every time.

If you are familiar with the concepts of fixed point math and affine texture mapping, this lesson should have been easy for you to grasp, and you can focus on the SX52-specific things. If this is all new to you then you could try sitting down with a scientific calculator or a pen and paper and try out some examples of your own until you get the hang of it. In case your binary arithmetic skills are a bit rusty now would also be a good time to refresh them, you'll thank yourself later.

```
            ; TOP SCREEN OVERSCAN
            mov scanline, #32
            mov scanline, #60
:Vblank_Loop3
            PREPARE_VIDEO_HORIZ burst_phase
            PREPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC
            mov RE, #BLACK
            cjb RTCC,#131,$
            DELAY(8)
            cjb RTCC,#130,$
            djnz scanline, :Vblank_Loop3

            jmp @Begin_Raster       ; Loop back for the next frame

IF $ > $600
ERROR "Page Spillage!"
ENDIF

;##############################################################################



; XGS overscan logo
LOGO_PAGE   EQU   $0B
org         LOGO_PAGE*$100
include           "xgslogo.src"

org $C00
dw    7
dw    7
dw    7
dw    7
dw    7
dw    7
dw    7
dw    8
dw    8
dw    8
dw    8
dw    8
dw    8
```

```
dw    9
dw    9
dw    9
dw    9
dw    9
dw    9
dw    9
dw    9
dw    10
dw    10
dw    10
dw    10
dw    10
dw    10
dw    10
dw    10
dw    10
dw    11
dw    11
dw    11
dw    11
dw    11
dw    11
dw    11
dw    11
dw    11
dw    11
dw    12
dw    12
dw    12
dw    12
dw    12
dw    12
dw    12
dw    12
dw    12
dw    12
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    13
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    14
dw    15
dw    15
```

```
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
dw    15
```