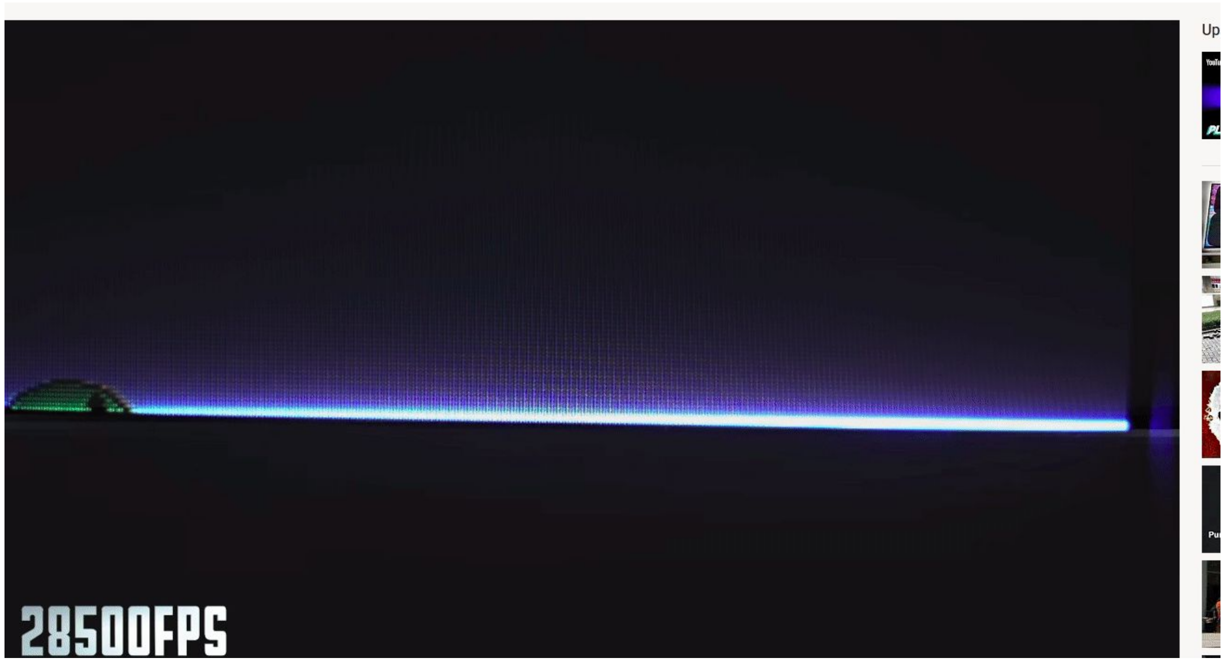
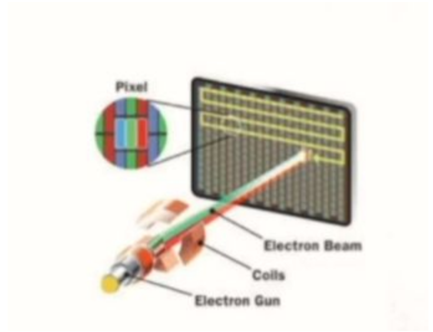
Atari 2600 (6502 processor) Assembly Programming

Displaying images on a CRT TV - Painting the screen



A CRT TV paints an image by shooting an electron gun at the screen in horizontal 'scanlines'. A 1280x720 CRT (they exist and I want one) would have 720 visible scanlines.



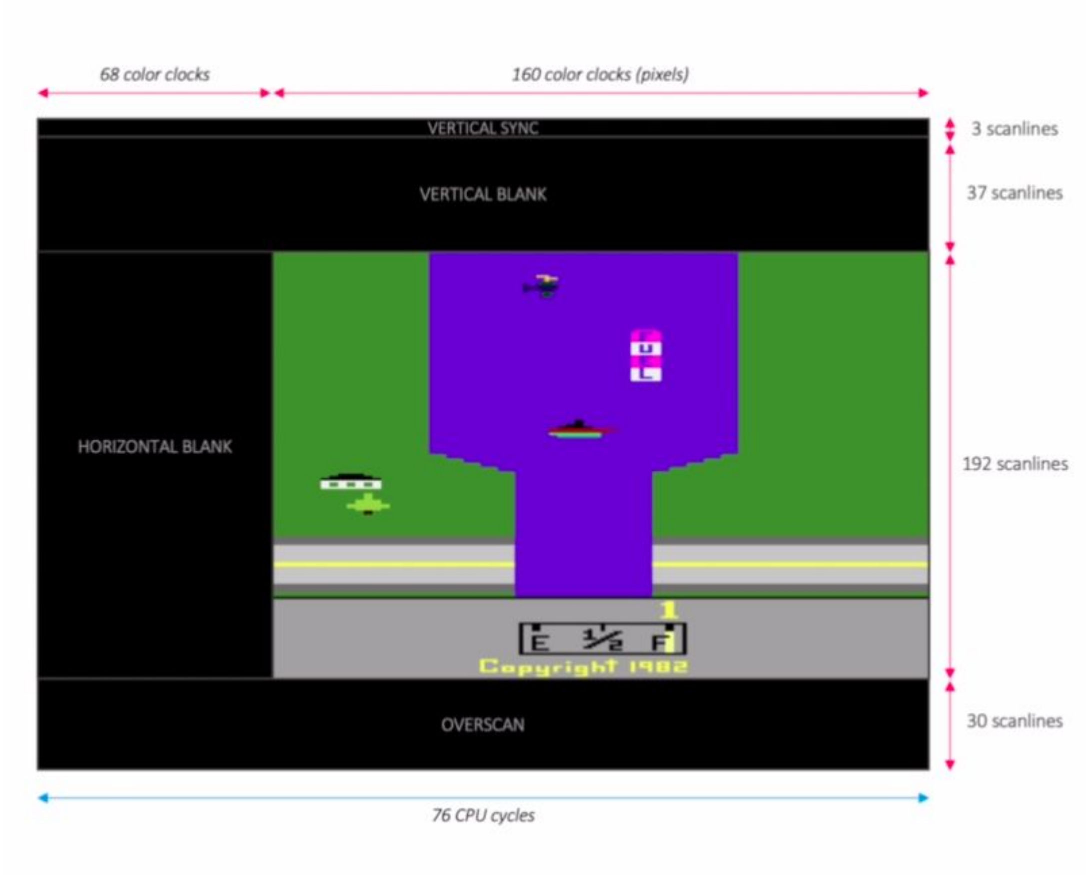
Racing the beam

The Atari 2600 will always paint 192 visible scanlines, with 160 pixels (color clocks) per scanline as it moves left to right.

However, it also has processing time available while the beam re-positions itself from right to left, or from bottom to top.

This time equates to:

- After each scanline, the amount of time it would otherwise take to draw 68 pixels (the Horizontal Blank).
- 70 full scanline-times after all scanlines have been painted (i.e. after one whole frame). Programmers call these 70 the Vertical Blank and Overscan, and the Atari hardware abstracts them as happening 'before' and 'after' a frame.

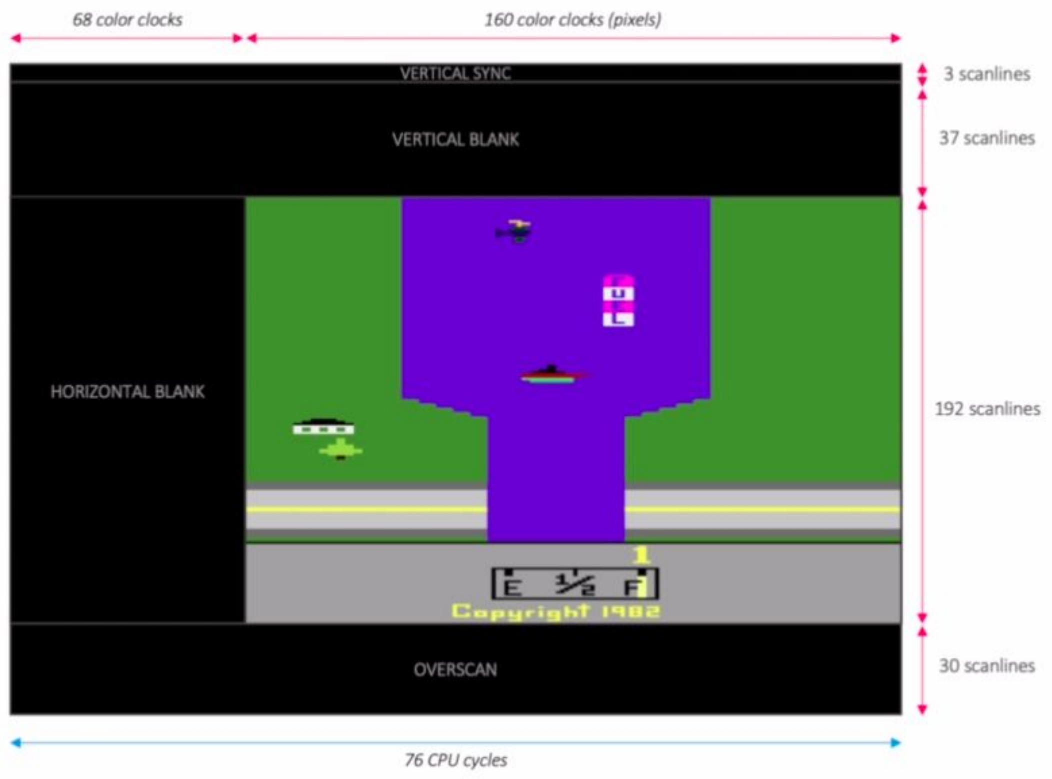


Racing the beam

The Atari does not have enough memory to render frame-by-frame (i.e. it can't hold the values of 192x160 = 30720 pixels in memory all at once), so it renders scanline-by-scanline.

This means the main game 'loop' runs for each scanline. The programmer's main concern is their central 'gameVisibleScanline()' method, which needs to know when and where it should paint what object.

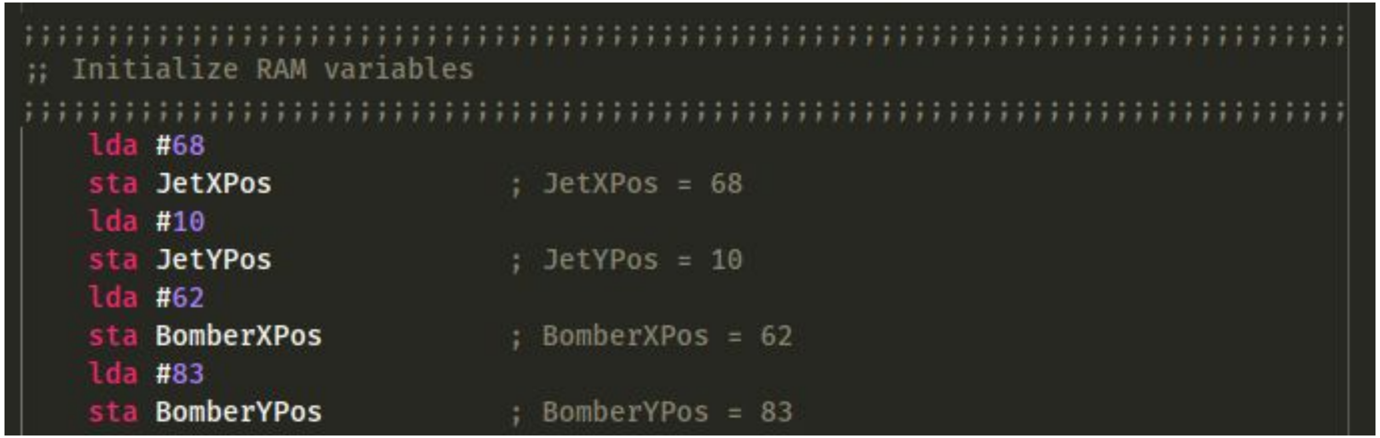
The VBLANK, HBLANK and OVERSCAN are important because they give the programmer time to run their commands and get a scanline ready **before** the beam begins firing.



The assembly code

For anyone totally unfamiliar with assembly like I was, it looks scary but is obvious in hindsight. It’s just a set of instructions like any other language - the instructions are just limited to three characters each, so you have no hope of knowing what they do until you look at a reference sheet or someone tells you, unlike something like Python which is much more human-readable.

Assigning some variables initial values:



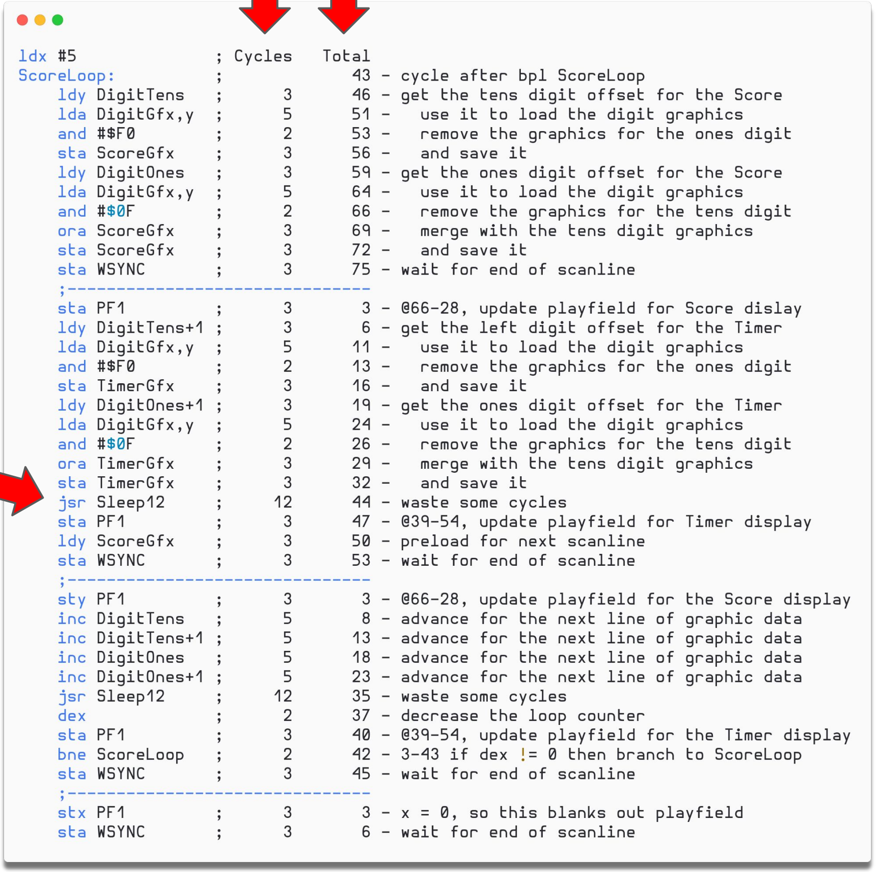
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initialize RAM variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    lda #68
    sta JetXPos              ; JetXPos = 68
    lda #10
    sta JetYPos              ; JetYPos = 10
    lda #62
    sta BomberXPos           ; BomberXPos = 62
    lda #83
    sta BomberYPos           ; BomberYPos = 83
```

Racing the beam

Of course, clock cycles are limited, so we can’t perform a bunch of calculations before a scanline and expect the beam to wait for us. If you do perform too many instructions, the beam carries on with whatever instruction the Atari has gotten up to - you’ve ‘lost the race’, and you’ll have no idea where your code execution is up to at any given time.

As seen here, Atari programmers often leave comments on every single line of code for how many clock cycles a command takes, plus a running total, so they don’t run out of render time for each scanline.

For example, the given subroutine (function) Sleep12 wastes 12 clock cycles when called but has no code inside it, because a `jsr` command (jump to subroutine) takes 6 cycles, and an `rts` (return from subroutine) takes 6 as well.

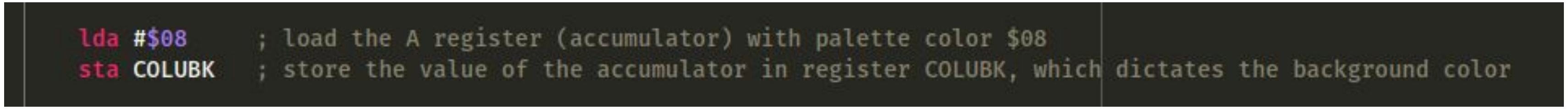


```
ldx #5                ; Cycles   Total
ScoreLoop:            ;             43 - cycle after bpl ScoreLoop
    ldy DigitTens     ;       3     46 - get the tens digit offset for the Score
    lda DigitGfx,y    ;       5     51 -   use it to load the digit graphics
    and #$F0          ;       2     53 -   remove the graphics for the ones digit
    sta ScoreGfx      ;       3     56 -   and save it
    ldy DigitOnes     ;       3     59 - get the ones digit offset for the Score
    lda DigitGfx,y    ;       5     64 -   use it to load the digit graphics
    and #$0F          ;       2     66 -   remove the graphics for the tens digit
    ora ScoreGfx      ;       3     69 -   merge with the tens digit graphics
    sta ScoreGfx      ;       3     72 -   and save it
    sta WSYNC         ;       3     75 - wait for end of scanline
    ;------------------------------
    sta PF1           ;       3      3 - @66-28, update playfield for Score dislay
    ldy DigitTens+1   ;       3      6 - get the left digit offset for the Timer
    lda DigitGfx,y    ;       5     11 -   use it to load the digit graphics
    and #$F0          ;       2     13 -   remove the graphics for the ones digit
    sta TimerGfx      ;       3     16 -   and save it
    ldy DigitOnes+1   ;       3     19 - get the ones digit offset for the Timer
    lda DigitGfx,y    ;       5     24 -   use it to load the digit graphics
    and #$0F          ;       2     26 -   remove the graphics for the tens digit
    ora TimerGfx      ;       3     29 -   merge with the tens digit graphics
    sta TimerGfx      ;       3     32 -   and save it
    jsr Sleep12       ;      12     44 - waste some cycles
    sta PF1           ;       3     47 - @39-54, update playfield for Timer display
    ldy ScoreGfx      ;       3     50 - preload for next scanline
    sta WSYNC         ;       3     53 - wait for end of scanline
    ;------------------------------
    sty PF1           ;       3      3 - @66-28, update playfield for the Score display
    inc DigitTens     ;       5      8 - advance for the next line of graphic data
    inc DigitTens+1   ;       5     13 - advance for the next line of graphic data
    inc DigitOnes     ;       5     18 - advance for the next line of graphic data
    inc DigitOnes+1   ;       5     23 - advance for the next line of graphic data
    jsr Sleep12       ;      12     35 - waste some cycles
    dex               ;       2     37 - decrease the loop counter
    sta PF1           ;       3     40 - @39-54, update playfield for the Timer display
    bne ScoreLoop     ;       2     42 - 3-43 if dex != 0 then branch to ScoreLoop
    sta WSYNC         ;       3     45 - wait for end of scanline
    ;------------------------------
    stx PF1           ;       3      3 - x = 0, so this blanks out playfield
    sta WSYNC         ;       3      6 - wait for end of scanline
```

The assembly code

Because Atari games are so simple, there are dedicated registers for things like background color and player display.

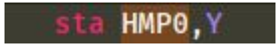
This means, to set the background color, all that needs to be done is



```
lda #$08     ; load the A register (accumulator) with palette color $08
sta COLUBK   ; store the value of the accumulator in register COLUBK, which dictates the background color
```

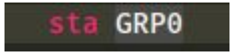
There are other dedicated registers that are set to indicate things like:

- “Player 1 should be displayed (somewhere) on this scanline”.



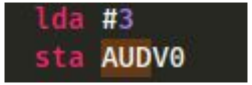
```
sta HMP0,Y
```

- “Player 1’’s color is (accumulator value)”.



```
sta GRP0
```

- “A sound of volume 3 should be played until another sound command is executed.”

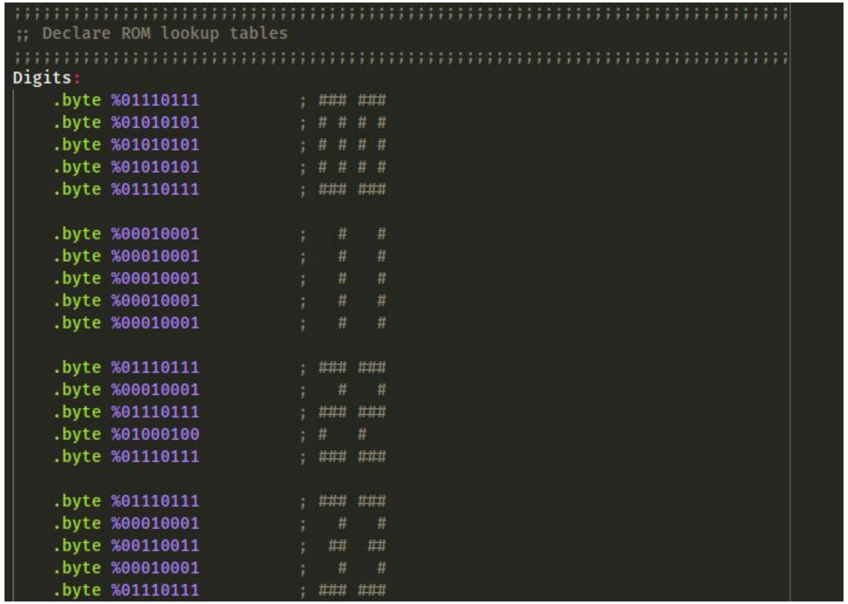


```
lda #3
sta AUDV0
```

This makes Atari programming incredibly forgiving in a lot of ways compared to modern games (even though you have to learn 6502 assembly), because the Atari already has the abstractions of ‘player 1’ or ‘background’ or ‘audio’ built into its **hardware,** and has some idea of what the software is expecting to be able to do with them.

The assembly code

Every frame of animation has to be manually declared, stored and retrieved from memory, as binary pixel patterns, even numbers for a scoreboard! Usually done right at the end of the ROM (bottom of the code). No procedural generation here.

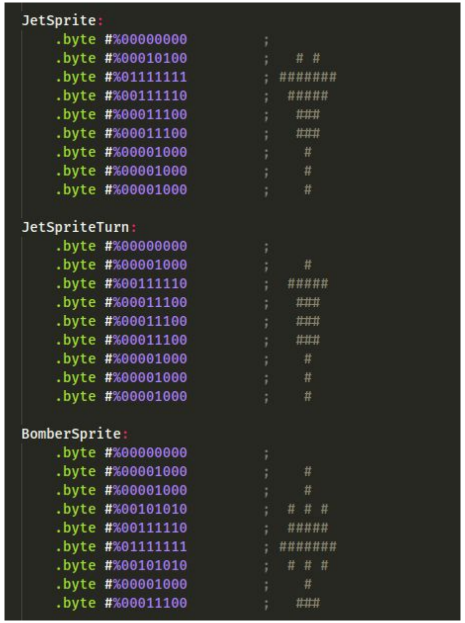


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Declare ROM lookup tables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
Digits:
    .byte %01110111          ; ### ###
    .byte %01010101          ; # # # #
    .byte %01010101          ; # # # #
    .byte %01010101          ; # # # #
    .byte %01110111          ; ### ###

    .byte %00010001          ;   #   #
    .byte %00010001          ;   #   #
    .byte %00010001          ;   #   #
    .byte %00010001          ;   #   #
    .byte %00010001          ;   #   #

    .byte %01110111          ; ### ###
    .byte %00010001          ;   #   #
    .byte %01110111          ; ### ###
    .byte %01000100          ; #   #
    .byte %01110111          ; ### ###

    .byte %01110111          ; ### ###
    .byte %00010001          ;   #   #
    .byte %00110011          ;  ##  ##
    .byte %00010001          ;   #   #
    .byte %01110111          ; ### ###
```



```
JetSprite:
    .byte #%00000000         ;
    .byte #%00010100         ;   # #
    .byte #%01111111         ; #######
    .byte #%00111110         ;  #####
    .byte #%00011100         ;   ###
    .byte #%00011100         ;   ###
    .byte #%00001000         ;    #
    .byte #%00001000         ;    #
    .byte #%00001000         ;    #

JetSpriteTurn:
    .byte #%00000000         ;
    .byte #%00001000         ;    #
    .byte #%00111110         ;  #####
    .byte #%00011100         ;   ###
    .byte #%00011100         ;   ###
    .byte #%00011100         ;   ###
    .byte #%00001000         ;    #
    .byte #%00001000         ;    #
    .byte #%00001000         ;    #

BomberSprite:
    .byte #%00000000         ;
    .byte #%00001000         ;    #
    .byte #%00001000         ;    #
    .byte #%00101010         ;  # # #
    .byte #%00111110         ;  #####
    .byte #%01111111         ; #######
    .byte #%00101010         ;  # # #
    .byte #%00001000         ;    #
    .byte #%00011100         ;   ###
```

The final game!

https://drive.google.com/file/d/1dqYiU0lNIYODVJDQ8-Yk6RjZomaBA2HN/view?usp=sharing

https:/ /javatari.org/