Let’s Make an Atari 2600 (VCS) Game!

By Darrell Spice, Jr. (adapted by Duane Alan Hahn, a.k.a. Random Terrain)

# Introduction

For my Atari 2600 Homebrew presentation I've been giving a rundown of the challenges involved in writing an Atari game, namely the limited resources and capabilities of the hardware, as well as the tools (DASM, bB, Stella, Harmony, etc.) and resources (AtariAge, Mini Dig, etc.) that are available to help you.

I've been updating the presentation for each time I give it. On the most recent update, for the 2013 Houston Arcade Expo, I added code for a very simple 2600 program. You can download the source and ROM from my website—colorful.zip.

The code addition went over very well so I've decided to expand upon it for my next presentation, which will be given the weekend of August 16th at the Classic Game Fest in Austin. I decided a very simple game would be the way to go and have worked up a mockup of what it might look like:

A screenshot of a video game

Description automatically generated

The game's going to be called Collect. It's a 1 player game and your objective is to collect as many boxes as you can before the timer runs out. My goals for code are to show:

* How to use TV Type, Select and Reset console switches
* How to use joystick to move player
* How to use the hardware collision detection
* How to use a 2 Line Kernel to draw a reflected playfield with 2 players and 2 missiles
* Sound Effects (timer tick, collected box, end-of-game)

# Step 1: Generate a Stable Display

First things first—head over to MiniDig - Best of Stella and download the [Stella Programmer's Guide](https://www.qotile.net/minidig/docs/stella.pdf) from the docs page. I've also attached it to my blog entry, but you should still check out what's available over at MiniDig.

The heart of the Atari is the Television Interface Adaptor (TIA). It's the video chip, sound generator, and also handles some of the controller input. As a video chip, the TIA is very unusual. Most video game systems have memory that holds the current state of the display. Their video chip reads that memory and uses that information to generate the display. But not the TIA—memory was very expensive at the time, so the TIA was designed with a handful of registers that contain just the information needed to draw a single scanline. It's up to our program to change those registers in real-time so that each scanline shows what its supposed to. It's also up to our program to generate a "sync signal" that tells the TV when its time to start generating a new image.

Turn to page 2 of the programmer's guide. You'll find the following diagram, which I've slightly modified:

A screen shot of a computer

Description automatically generated

The Horizontal Blank is part of each scanline, so we don't need to worry about generating it. Everything else though is up to us! We need to generate a sync signal over 3 scanlines, after which we need to wait 37 scanlines before we tell TIA to "turn on" the image output. After that we need up update TIA so each of the 192 scanlines that comprise visible portion of the display show what they're supposed to. Once that's done, we "turn off" the image output and wait 30 scanlines before we start all over again.

In the source code, available below, you can see the Main Program Loop which follows the diagram:

Main:

        jsr VerticalSync    ; Jump to SubRoutine VerticalSync

        jsr VerticalBlank   ; Jump to SubRoutine VerticalBlank

        jsr Kernel          ; Jump to SubRoutine Kernel

        jsr OverScan        ; Jump to SubRoutine OverScan

        jmp Main            ; JuMP to Main

Each of the subroutines handles what's needed, such as this section which generates the sync signal:

VerticalSync:

        lda #2      ; LoaD Accumulator with 2 so D1=1

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        sta VSYNC   ; Accumulator D1=1, turns on Vertical Sync signal

        sta WSYNC   ; Wait for Sync - halts CPU until end of 1st scanline of VSYNC

        sta WSYNC   ; wait until end of 2nd scanline of VSYNC

        lda #0      ; LoaD Accumulator with 0 so D1=0

        sta WSYNC   ; wait until end of 3rd scanline of VSYNC

        sta VSYNC   ; Accumulator D1=0, turns off Vertical Sync signal

        rts         ; ReTurn from Subroutine

Currently there's no game logic, so the VerticalBlank just waits for the 37 scanlines to pass:

VerticalBlank:

        ldx #37         ; LoaD X with 37

vbLoop:

        sta WSYNC       ; Wait for SYNC (halts CPU until end of scanline)

        dex             ; DEcrement X by 1

        bne vbLoop      ; Branch if Not Equal to 0

        rts             ; ReTurn from Subroutine

The Kernel is the section of code that draws the screen:

Kernel:

    ; turn on the display

        sta WSYNC       ; Wait for SYNC (halts CPU until end of scanline)

        lda #0          ; LoaD Accumulator with 0 so D1=0

        sta VBLANK      ; Accumulator D1=1, turns off Vertical Blank signal (image output on)

    ; draw the screen

        ldx #192        ; Load X with 192

KernelLoop:

        sta WSYNC       ; Wait for SYNC (halts CPU until end of scanline)

        stx COLUBK      ; STore X into TIA's background color register

        dex             ; DEcrement X by 1

        bne KernelLoop  ; Branch if Not Equal to 0

        rts             ; ReTurn from Subroutine

For this initial build it just changes the background color so we can see that we're generating a stable picture:

A colorful bars of paint

Description automatically generated with medium confidence

Like Vertical Blank, OverScan doesn't have anything to do besides turning off the image output, so it just waits for enough scanlines to pass so that the total scanline count is 262.

OverScan:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        lda #2      ; LoaD Accumulator with 2 so D1=1

        sta VBLANK  ; STore Accumulator to VBLANK, D1=1 turns image output off

        ldx #27     ; LoaD X with 27

osLoop:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        dex         ; DEcrement X by 1

        bne osLoop  ; Branch if Not Equal to 0

        rts         ; ReTurn from Subroutine

# Step 2: Timers

In Step 1, I used loops of sta WSYNC commands to delay the program so that Vertical Blank and OverScan would last for the proper duration. That method works fine when all we want to do is generate a static display, but as soon as we start to add game logic that won't work out so well.

The problem with the game logic is there will be so many different paths the code can take that it is nearly impossible for us to know how long the code ran, and thus we won't know how many scanlines we need to delay before the next section of code can run. As an example, if the player isn't moving the joystick then none of the "move player" logic will run. If the player is moving the joystick left and up then the "move horizontal" and "move vertical" logic will run. If the player is only holding the joystick left then only the "move horizontal" logic will run.

Fortunately for us, the Atari 2600 contains a RIOT chip. That acronym stands for RAM, Input/Output and Timer. We're interested in the Timer for this update to Collect, we'll look at RAM and I/O in a later update.

First thing I changed was OverScan. The original routine looked like this:

OverScan:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        lda #2      ; LoaD Accumulator with 2 so D1=1

        sta VBLANK  ; STore Accumulator to VBLANK, D1=1 turns image output off

        ldx #27     ; LoaD X with 27

osLoop:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        dex         ; DEcrement X by 1

        bne osLoop  ; Branch if Not Equal to 0

        rts         ; ReTurn from Subroutine

So what we want to do is set a timer that will go off after 27 scanlines to pass. There's 76 cycles of time per scanline, so we need the timer to go off after 2052 cycles have passed. When we set the timer, we also select how frequently the timer will decrement in value. RIOT has options to decrement the timer every 1, 8, 64 or 1024 cycles.

The timer is set using a single byte, so it can only be set to any value from 0 to 255. As such, we know we can't use decrement every 1 cycle as 2052 is too large. So let's check if decrement every 8 cycles will work:

2052/8 = 256.5

Almost, but 256 won't fit so we're going to have to use the decrement every 64 cycles option. To figure out the initial value to set the timer to, use this equation:

(scanlines \* 76) / 64

The new OverScan routine that uses the timer looks like this:

OverScan:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        lda #2      ; LoaD Accumulator with 2 so D1=1

        sta VBLANK  ; STore Accumulator to VBLANK, D1=1 turns image output off

        lda #32     ; set timer for 27 scanlines, 32 = ((27 \* 76) / 64)

        sta TIM64T  ; set timer to go off in 27 scanlines

    ; game logic will go here

OSwait:

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        lda INTIM   ; Check the timer

        bne OSwait  ; Branch if its Not Equal to 0

        rts         ; ReTurn from Subroutine

For Vertical Blank we're going to set up the timer a little different. There's time in the Vertical Sync we can utilize, so we'll set the timer there—look for the code using ldx and stx:

VerticalSync:

        lda #2      ; LoaD Accumulator with 2 so D1=1

        ldx #49     ; LoaD X with 49

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        sta VSYNC   ; Accumulator D1=1, turns on Vertical Sync signal

        stx TIM64T  ; set timer to go off in 41 scanlines (49 \* 64) / 76

        sta WSYNC   ; Wait for Sync - halts CPU until end of 1st scanline of VSYNC

        sta WSYNC   ; wait until end of 2nd scanline of VSYNC

        lda #0      ; LoaD Accumulator with 0 so D1=0

        sta WSYNC   ; wait until end of 3rd scanline of VSYNC

        sta VSYNC   ; Accumulator D1=0, turns off Vertical Sync signal

        rts         ; ReTurn from Subroutine

We're also going to check the timer in the Kernel section so we can start drawing the screen as soon as it goes off:

Kernel:

        sta WSYNC       ; Wait for SYNC (halts CPU until end of scanline)

        lda INTIM       ; check the timer

        bne Kernel      ; Branch if its Not Equal to 0

    ; turn on the display

        sta VBLANK      ; Accumulator D1=0, turns off Vertical Blank signal (image output on)

    ; draw the screen

        ldx #192        ; Load X with 192

KernelLoop:

        sta WSYNC       ; Wait for SYNC (halts CPU until end of scanline)

        stx COLUBK      ; STore X into TIA's background color register

        dex             ; DEcrement X by 1

        bne KernelLoop  ; Branch if Not Equal to 0

        rts             ; ReTurn from Subroutine

For the moment, these changes leave Vertical Blank with nothing to do:

VerticalBlank:

        rts             ; ReTurn from Subroutine

# Step 3: Score and Timer Display

After getting a stable display, I like to implement the routines for displaying the score. You can see that in the first builds of Frantic, Medieval Mayhem, and Space Rocks. Even though we're not ready to show the player's score, the display is very useful for showing diagnostic information—such as this build of Frantic which uses it to show which sprites are colliding with the player.

To draw the score we're going to use the playfield graphics. The playfield pattern is comprised of 20 bits stored in 3 bytes of the TIA.

* PF0 - only the upper 4 bits are used, and they're output in reverse order
* PF1 - all 8 bits are used, output as you'd expect
* PF2 - all 8 bits are used, output in reverse order

Andrew Davie posted this rather nice diagram in his 2600 Programming For Newbies tutorial which shows how that works:

A diagram of a color scheme

Description automatically generated

The 20 bits are either repeated or reflected to draw the 20 bits on the right half of the screen.

For our game, a two digit score and a two digit timer will meet our requirements. We could show a single digit each in PF1 and PF2, but due to the reversed output of PF2 that would mean we'd need to create both normal and mirrored digit graphics. Instead, we're going to create digit graphics that are 3 bits across, which will allow us to show two digits using PF1. You may have seen the graphics in one of my blog entries with the revised mode file for jEdit. Each digit appears twice as we can use a simple mask to get the tens (AND #$F0) and ones (AND #$0F) digits. If we only saved the image as the ones position we'd need to apply 4 shift instructions to create a tens position image.

A screenshot of a computer

Description automatically generated

PF1 is displayed twice on the screen and if we time it correctly we can change the contents of PF1 so that both sides of the screen are different. Andrew Davie posted another handy diagram that shows the timing:

A screen shot of a screen

Description automatically generated

In looking at the diagram we can figure out the update times required for each instance of PF1.

* Left PF1 - need to update on or after cycle 66 of the prior scanline, or on or before cycle 28 of the current scanline
* Right PF1 - need to update on or after cycle 39 and on or before cycle 54 of the current scanline

As mentioned before, the RIOT chip in the Atari 2600 stands for RAM, Input/Output and Timer. For this update we're going to look at the RAM and Input.

To show the score we need to keep track of a few things, so let's allocate some space in RAM:

        ORG $80

    ; Holds 2 digit score, stored as BCD (Binary Coded Decimal)

Score:          ds 1    ; stored in $80

    ; Holds 2 digit timer, stored as BCD

Timer:          ds 1    ; stored in $81

    ; Offsets into digit graphic data

DigitOnes:      ds 2    ; stored in $82-$83, DigitOnes = Score, DigitOnes+1 = Timer

DigitTens:      ds 2    ; stored in $84-$85, DigitTens = Score, DigitTens+1 = Timer

    ; graphic data ready to put into PF1

ScoreGfx:       ds 1    ; stored in $86

TimerGfx:       ds 1    ; stored in $87

    ; scratch variable

Temp:           ds 1    ; stored in $88

Vertical Blank is now doing a little bit of work:

VerticalBlank:

        jsr SetObjectColors

        jsr PrepScoreForDisplay

        rts             ; ReTurn from Subroutine

When the macro CLEAN\_START initialized the Atari's hardware, it set all the colors to black. So we need to set the object colors if we want to see anything. This routine also reads the state of the console switches (via one of RIOT's Input registers) in order to determine if the player selected Color or Black & White:

SetObjectColors:

        ldx #3          ; we're going to set 4 colors (0-3)

        ldy #3          ; default to the color entries in the table (0-3)

        lda SWCHB       ; read the state of the console switches

        and #%00001000  ; test state of D3, the TV Type switch

        bne SOCloop     ; if D3=1 then use color

        ldy #7          ; else use the b&w entries in the table (4-7)

SOCloop:

        lda Colors,y    ; get the color or b&w value

        sta COLUP0,x    ; and set it

        dey             ; decrease Y

        dex             ; decrease X

        bpl SOCloop     ; Branch PLus (positive)

        rts             ; ReTurn from Subroutine

Colors:

        .byte $86   ; blue       - goes into COLUP0, color for player0 and missile0

        .byte $C6   ; green      - goes into COLUP1, color for player1 and missile1

        .byte $46   ; red        - goes into COLUPF, color for playfield and ball

        .byte $00   ; black      - goes into COLUBK, color for background

        .byte $0E   ; white      - goes into COLUP0, color for player0 and missile0

        .byte $06   ; dark grey  - goes into COLUP1, color for player1 and missile1

        .byte $0A   ; light grey - goes into COLUPF, color for playfield and ball

        .byte $00   ; black      - goes into COLUBK, color for background

The score and timer will be stored using BCD (Binary Coded Decimal) but for now we'll treat and display them as hexadecimal values (that's why the digit graphics are 0-9 then A-F). This routine takes the upper and lower nybble (4 bits) of the Score and Timer and multiplies them by 5 in order to get the offset into the digit graphic data. The 6507 does not have a multiply command, though it does have a shift feature which is equivalent to \*2. If a nybble is value X then X \* 2 \* 2 + X is the same as X \* 5:

PrepScoreForDisplay:

    ; for testing purposes, change the values in Timer and Score

        inc Timer       ; INCrement Timer by 1

        bne PSFDskip    ; Branch Not Equal to 0

        inc Score       ; INCrement Score by 1 if Timer just rolled to 0

PSFDskip

        ldx #1          ; use X as the loop counter for PSFDloop

PSFDloop:

        lda Score,x     ; LoaD A with Timer(first pass) or Score(second pass)

        and #$0F        ; remove the tens digit

        sta Temp        ; Store A into Temp

        asl             ; Accumulator Shift Left (# \* 2)

        asl             ; Accumulator Shift Left (# \* 4)

        adc Temp        ; ADd with Carry value in Temp (# \* 5)

        sta DigitOnes,x  ; STore A in DigitOnes+1(first pass) or DigitOnes(second pass)

        lda Score,x     ; LoaD A with Timer(first pass) or Score(second pass)

        and #$F0        ; remove the ones digit

        lsr             ; Logical Shift Right (# / 2)

        lsr             ; Logical Shift Right (# / 4)

        sta Temp        ; Store A into Temp

        lsr             ; Logical Shift Right (# / 8)

        lsr             ; Logical Shift Right (# / 16)

        adc Temp        ; ADd with Carry value in Temp ((# / 16) \* 5)

        sta DigitTens,x ; STore A in DigitTens+1(first pass) or DigitTens(second pass)

        dex             ; DEcrement X by 1

        bpl PSFDloop    ; Branch PLus (positive) to PSFDloop

        rts             ; ReTurn from Subroutine

The Kernel's been modified so it now uses the data in DigitOnes and DigitTens to update PF1. Each line of graphic data is output twice so the digits are drawn over 10 scanlines which gives them a better appearance than if they had been drawn over 5 scanlines.

ldx #5

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

Score and Timer:

A screenshot of a video game

Description automatically generated

When TV Type switched to B&W:

A screenshot of a video game

Description automatically generated

You'll notice that the score and timer are different colors even though they're both drawn using the playfield. This is because I've modified Vertical Sync to turn on SCORE mode. SCORE mode tells the TIA to use the color of player0 for the left half of the playfield and the color of player1 for the right half.

VerticalSync:

        lda #2      ; LoaD Accumulator with 2 so D1=1

        ldx #49     ; LoaD X with 49

        sta WSYNC   ; Wait for SYNC (halts CPU until end of scanline)

        sta VSYNC   ; Accumulator D1=1, turns on Vertical Sync signal

        stx TIM64T  ; set timer to go off in 41 scanlines (49 \* 64) / 76

        sta CTRLPF  ; D1=1, playfield now in SCORE mode

...

        rts         ; ReTurn from Subroutine

# Step 4: 2 Line Kernel

Let's review the TIA Timing diagram from last time:

A screen shot of a diagram

Description automatically generated

We used that to determine when we could safely update the playfield data in order to draw the score and timer. For moveable objects (player0, player1, missile0, missile1 and ball) if you update their graphics during the Visible Screen (cycles 23-76) you run the risk of shearing. For something that's moving fast, like the snowball in Stay Frosty 2, shearing may be an acceptable design compromise:

A screenshot of a video game

Description automatically generated

That snowball should be square, but the left edge has sheared due to the ball object being updated mid-scanline.

To prevent shearing we need to update the objects on cycles 0-22. There's a lot of calculations to be done in the kernel to draw just one player. For Collect I'm using DoDraw, which looks like this for drawing player0:

DoDraw0:

        lda #HUMAN\_HEIGHT-1 ; 2  2 - height of the humanoid graphics, subtract 1 due to starting with 0

        dcp HumanDraw       ; 5  7 - Decrement HumanDraw and compare with height

        bcs DoDrawGrp0      ; 2  9 - (3 10) if Carry is Set, then humanoid is on current scanline

        lda #0              ; 2 11 - otherwise use 0 to turn off player0

        .byte $2C           ; 4 15 - $2C = BIT with absolute addressing, trick that

                            ;        causes the lda (HumanPtr),y to be skipped

DoDrawGrp0:                 ;   10 - from bcs DoDrawGrp0

        lda (HumanPtr),y    ; 5 15 - load the shape for player0

        sta GRP0            ; 3 18 - update player0 to draw Human

That's 18 cycles to draw a single player. One way to make it easier to fit all the code in is to use a 2 Line Kernel (2LK). In a 2LK we update the TIA's registers over 2 scanlines in order to build the display. For Collect, the current routines are updating them like this:

1. player0, playfield
2. player1, playfield

The actual code looks like this:

ldy #ARENA\_HEIGHT   ; 2  7 - the arena will be 180 scanlines (from 0-89)\*2

ArenaLoop:                  ;   13 - from bpl ArenaLoop

    ; continuation of line 2 of the 2LK

    ; this precalculates data that's used on line 1 of the 2LK

        lda #HUMAN\_HEIGHT-1 ; 2 15 - height of the humanoid graphics, subtract 1 due to starting with 0

        dcp HumanDraw       ; 5 20 - Decrement HumanDraw and compare with height

        bcs DoDrawGrp0      ; 2 22 - (3 23) if Carry is Set, then humanoid is on current scanline

        lda #0              ; 2 24 - otherwise use 0 to turn off player0

        .byte $2C           ; 4 28 - $2C = BIT with absolute addressing, trick that

                            ;        causes the lda (HumanPtr),y to be skipped

DoDrawGrp0:                 ;   23 - from bcs DoDrawGrp0

        lda (HumanPtr),y    ; 5 28 - load the shape for player0

        sta WSYNC           ; 3 31

;---------------------------------------

    ; start of line 1 of the 2LK

        sta GRP0            ; 3  3 - @ 0-22, update player0 to draw Human

        ldx #%11111111      ; 2  5 - playfield pattern for vertical alignment testing

        stx PF0             ; 3  8 - @ 0-22

    ; precalculate data that's needed for line 2 of the 2LK

        lda #HUMAN\_HEIGHT-1 ; 2 10 - height of the humanoid graphics,

        dcp BoxDraw         ; 5 15 - Decrement BoxDraw and compare with height

        bcs DoDrawGrp1      ; 2 17 - (3 18) if Carry is Set, then box is on current scanline

        lda #0              ; 2 19 - otherwise use 0 to turn off player1

        .byte $2C           ; 4 23 - $2C = BIT with absolute addressing, trick that

                            ;        causes the lda (BoxPtr),y to be skipped

DoDrawGrp1:                 ;   18 - from bcs DoDrawGRP1

        lda (BoxPtr),y      ; 5 23 - load the shape for the box

        sta WSYNC           ; 3 26

;---------------------------------------

    ; start of line 2 of the 2LK

        sta GRP1            ; 3  3 - @0-22, update player1 to draw box

        ldx #0              ; 2  5 - PF pattern for alignment testing

        stx PF0             ; 3  8 - @0-22

        dey                 ; 2 10 - decrease the 2LK loop counter

        bpl ArenaLoop       ; 2 12 - (3 13) branch if there's more Arena to draw

If you look at that closely, you'll see I'm splitting DoDraw a bit so that this is how the 2LK works:

1. update player0, update playfield, precalc player1 data for next scanline
2. update player1, update playfield, precalc player0 data for next scanline

By pre-calculating data during the visible portion of the scanline, we'll have more time during the critical 0-22 cycles for when we add the other objects.

Since we're updating the players on every other scanline, each byte of graphic data is displayed twice (compare the thickness of the humanoid pixels with the red lines drawn with the playfield). Also, the players never line up as they're never updated on the same scanlines:

A black screen with green and blue text

Description automatically generated

Closeup:

A red and blue text with black stripes

Description automatically generated with medium confidence

The designers of the TIA planned for this by adding a Vertical Delay feature to the players and ball (though sadly not the missiles). The TIA registers for this are VDELP0, VDELP1 and VDELBL. For this update to Collect, I've tied the Vertical Delay to the difficulty switches, putting the switch in position A will turn on the delay for that player so we can experiment with how that works. For the next update I'll set the Vertical Delay based on the Y position of the player (this also means the maximum Y value will be double that of this build).

Left Difficulty A, Right Difficulty B so VDELP0 = 1 and VDELP1 = 0. Sprites line up with the same Y:

A black screen with green and blue text

Description automatically generated

Closeup:

A red and black striped background with blue and green letters

Description automatically generated

Left Difficulty B, Right Difficulty A so VDELP0 = 0 and VDELP1 = 1. Sprites line up when player1's Y = player0's Y + 1:

Sprites line up when player1's Y = player0's Y + 1.

A screen shot of a video game

Description automatically generated

Closeup:

A blue and green letters on a red and black striped background

Description automatically generated

The code that preps the data used by DoDraw looks like this:

; HumanDraw = ARENA\_HEIGHT + HUMAN\_HEIGHT - Y position

        lda #(ARENA\_HEIGHT + HUMAN\_HEIGHT)

        sec

        sbc ObjectY

        sta HumanDraw

        ; HumanPtr = HumanGfx + HUMAN\_HEIGHT - 1 - Y position

        lda #<(HumanGfx + HUMAN\_HEIGHT - 1)

        sec

        sbc ObjectY

        sta HumanPtr

        lda #>(HumanGfx + HUMAN\_HEIGHT - 1)

        sbc #0

        sta HumanPtr+1

        ; BoxDraw = ARENA\_HEIGHT + HUMAN\_HEIGHT - Y position

        lda #(ARENA\_HEIGHT + HUMAN\_HEIGHT)

        sec

        sbc ObjectY+1

        sta BoxDraw

        ; BoxPtr = HumanGfx + HUMAN\_HEIGHT - 1 - Y position

        lda #<(HumanGfx + HUMAN\_HEIGHT - 1)

        sec

        sbc ObjectY+1

        sta BoxPtr

        lda #>(HumanGfx + HUMAN\_HEIGHT - 1)

        sbc #0

        sta BoxPtr+1

...

HumanGfx:

        .byte %00011100

        .byte %00011000

        .byte %00011000

        .byte %00011000

        .byte %01011010

        .byte %01011010

        .byte %00111100

        .byte %00000000

        .byte %00011000

        .byte %00011000

HUMAN\_HEIGHT = \* - HumanGfx

The graphics are much easier to see using my mode file for jEdit:

A screenshot of a computer

Description automatically generated

I'm sure some of you are wondering why the human graphics are upside down. If you wanted to loop through something 10 times, you'd normally think to write the code like this:

ldy #0

Loop:

  ; do some work

  iny

  cpy #10

  bne Loop

But the 6507 does an automatic check for 0 (as well as positive/negative) which lets you save 2 cycles of processing time by eliminating the CPY command:

ldy #10

Loop:

  ; do some work

  dey

  bne Loop

Alternatively, if your initial value is less than 128, you can use this:

ldy #(10-1)

Loop:

  ; do some work

  dey

  bpl Loop

Making the loop count down instead of up saves 2 cycles, but doing so requires the graphics to be upside down. 2 cycles doesn't sound like much, but in a scanline that's 2.6% of your processing time and saving it might be what allows you to update everything you want. In Kernels I've written, I often use every cycle—and that includes eliminating the sta WSYNC to buy back 3 cycles of processing time. See the reposition kernels in this post about Draconian.

I've also added joystick support that will let you move around the players. Pressing FIRE will slow down the movement, making it easier to line things up. The score (on the left) is used to display player0's Y position, and the timer is used for player1. As an added bonus, I'm showing how you can save ROM space by creating graphics that only face in one direction by using REFP0 and REFP1 (REFlect Player) to make the graphics face the other way. The routine's fairly sizable, so I'm not posting it here. Download the source code and check it out!

# Step 5: Automate Vertical Delay

For this update, we're going to double the Y range of the player objects. To use the new Y value for the 2LK we just need to divide it in half using the LSR command. The remainder of the divide, which ends up in the Carry flag, will conveniently tell us if we need to turn on Vertical Delay.

This routine preps the 2LK data for player0 and turns on VDELP0 if required (if you're wondering, VDELP0 is turned off in VerticalSync):

    ; prep Humanoid's Y position for 2LK

        ldx #1              ; preload X for setting VDELPx

        lda ObjectY         ; get the human's Y position

        lsr                 ; divide by 2 for the 2LK position

        sta Temp            ; save for position calculations

        bcs NoDelay0        ; if carry is set we don't need Vertical Delay

        stx VDELP0          ; carry was clear, so set Vertical Delay

NoDelay0:

    ; HumanDraw = ARENA\_HEIGHT + HUMAN\_HEIGHT - Y position

        lda #(ARENA\_HEIGHT + HUMAN\_HEIGHT)

        sec

        sbc Temp

        sta HumanDraw

    ; HumanPtr = HumanGfx + HUMAN\_HEIGHT - 1 - Y position

        lda #<(HumanGfx + HUMAN\_HEIGHT - 1)

        sec

        sbc Temp

        sta HumanPtr

        lda #>(HumanGfx + HUMAN\_HEIGHT - 1)

        sbc #0

        sta HumanPtr+1

One minor problem with the prior 2LK was that player1 could not show up on the topmost scanline of the Arena:

A black screen with green and blue numbers

Description automatically generated

Closeup:

A colorful figure with black background

Description automatically generated with medium confidence

To fix this, we'll modify the kernel to prime GRP1 before it enters the loop that draws the Arena:

        ldy #ARENA\_HEIGHT+1 ; 2  7 - the arena will be 180 scanlines (from 0-89)\*2

    ; prime GRP1 so player1 can appear on topmost scanline of the Arena

        lda #BOX\_HEIGHT-1   ; 2  9 - height of the box graphics,

        dcp BoxDraw         ; 5 14 - Decrement BoxDraw and compare with height

        bcs DoDrawGrp1pre   ; 2 16 - (3 17) if Carry is Set, then box is on current scanline

        lda #0              ; 2 18 - otherwise use 0 to turn off player1

        .byte $2C           ; 4 22 - $2C = BIT with absolute addressing, trick that

                            ;        causes the lda (BoxPtr),y to be skipped

DoDrawGrp1pre:              ;   17 - from bcs DoDrawGRP1pre

        lda (BoxPtr),y      ; 5 22 - load the shape for the box

        sta GRP1            ; 3 25 - @0-22, update player1 to draw box

        dey                 ; 2 27

ArenaLoop:                  ;   13 - from bpl ArenaLoop

The 2LK calculations for player1 used to be the same as for player0, but now must be modified to compensate for the priming of GRP1:

    ; prep box's Y position for 2LK

        lda ObjectY+1       ; get the box's Y position

        clc

        adc #1              ; add 1 to compensate for priming of GRP1

        lsr                 ; divide by 2 for the 2LK position

        sta Temp            ; save for position calculations

        bcs NoDelay1        ; if carry is set we don't need Vertical Delay

        stx VDELP1          ; carry was clear, so set Vertical Delay

NoDelay1:

    ; BoxDraw = ARENA\_HEIGHT + BOX\_HEIGHT - Y position + 1

    ; the + 1 compensates for priming of GRP1

        lda #(ARENA\_HEIGHT + BOX\_HEIGHT +1)

        sec

        sbc Temp

        sta BoxDraw

    ; BoxPtr = BoxGfx + BOX\_HEIGHT - 1 - Y position

        lda #<(BoxGfx + BOX\_HEIGHT - 1)

        sec

        sbc Temp

        sta BoxPtr

        lda #>(BoxGfx + BOX\_HEIGHT - 1)

        sbc #0

        sta BoxPtr+1

Added GRP1 priming which allows player1 to cover full Arena:

A black screen with green and blue squares

Description automatically generated

Closeup:

A colorful lines and a black background

Description automatically generated with medium confidence

Lastly, I added a new Box graphic for player1:

A screenshot of a video game

Description automatically generated

# Step 6: Spec Change

As anybody involved in writing software can tell you, project specifications will often change when new information becomes available.

When I started working on Collect, my plan was to use it for my presentation at Classic Game Fest. As I've progressed I've come to the realization that a full blown game is going to be just too much information for a one hour presentation. I decided that I'm going to leave the existing example in place and just add a few slides about Collect with a link to these blog entries for anybody who is interested.

Since I'm no longer planning to fit this project into a presentation, I've decided on a few changes:

* Make Collect a 1-2 player game
* Add the Ball object to draw an additional box, especially useful for 2 player games
* Change Timer display to the right player's score
* Add a Timer Bar to indicate time remaining

Mockup of two player game with new timer bar:

A screenshot of a video game

Description automatically generated

Timer has decreased:

A screenshot of a video game

Description automatically generated

One player variation will hide right player's score and use player1 as an additional box:

A screenshot of a video game

Description automatically generated

The Ball Object has a vertical delay feature. When used, the ball should be updated on the same scanline as player0. Due to this, I've revised the 2LK to be like this:

1. update player1, update playfield, precalc player0 data for next scanline
2. update player0, update playfield, precalc player1 data for next scanline

This was done to plan ahead for when the playfield is no longer updated on every scanline. Updating the playfield and ball will make the 2LK look something like this:

1. update player1, update playfield, precalc player0 and ball data for next scanline
2. update player0, update ball, precalc player 1 data for next scanline

I'll also need to add in updates for the missiles. Ideally we want to update them on every scanline like this:

1. update player1, update missile0, update missile1, update playfield, precalc player0 and ball and missile data for next scanline
2. update player0, update missile0, update missile1, update ball, precalc player 1 and missile data for next scanline

It's possible the timing won't work out for that. If it doesn't, then a change like this should work:

1. update player1, update missile1, update playfield, precalc player0 and ball and missile0 data for next scanline
2. update player0, update missile0, update ball, precalc player1 and missile1 data for next scanline

That would make it so that the missile objects can only start on every-other-scanline, but that's an OK compromise for our game.

In this build I've revised the Arena to be a little bit shorter to make room for the new timer display. The timer currently "ticks" once every 64 frames. Whenever it ticks, a bunch of byte rotations are done to shorten the length of the timer bar.

DecrementTimer:

        lsr TimerPF+5   ; PF2 right side, reversed bits so shift right

        rol TimerPF+4   ; PF1 right side, normal bits so shift left

        ror TimerPF+3   ; PF0 right side, reversed bits so shift right

        lda TimerPF+3   ; only upper nybble used, so we need to put bit 3 into C

        lsr

        lsr

        lsr

        lsr

        ror TimerPF+2   ; PF2 left side, reversed bits so shift right

        rol TimerPF+1   ; PF1 left side, normal bits so shift left

        ror TimerPF     ; PF0 left side, reversed bits so shift right

        rts

Since there are 40 playfield pixels, the total playtime would be 40\*64/60 = 42.7 seconds. We might decide that's too short of a play time. If so, we'll just change the tick to occur every 128 frames for 40\*128/60 = 85.3 seconds of game time, or maybe even once very 256 frames for 40\*256/60 = 170.7 seconds.

SetObjectColors has been modified to add a color for the timer bar. The Timer Bar and the Arena are both drawn using the playfield, so to make the Arena a different color than the Timer Bar I store the current Arena color in a RAM location.

SetObjectColors:

        ldx #4          ; we're going to set 5 colors (0-4)

        ldy #4          ; default to the color entries in the table (0-4)

        lda SWCHB       ; read the state of the console switches

        and #%00001000  ; test state of D3, the TV Type switch

        bne SOCloop     ; if D3=1 then use color

        ldy #9          ; else use the b&w entries in the table (5-9)

SOCloop:

        lda Colors,y    ; get the color or b&w value

        sta COLUP0-1,x  ; and set it

        dey             ; decrease Y

        dex             ; decrease X

        bne SOCloop     ; Branch Not Equal to Zero

        lda Colors,y    ; get the Arena color

        sta ArenaColor  ; save in RAM for Kernal Usage

        rts             ; ReTurn from Subroutine

Colors:

        .byte $46   ; red        - goes into COLUPF, color for Arena (after Timer is drawn)

        .byte $86   ; blue       - goes into COLUP0, color for player0 and missile0

        .byte $C6   ; green      - goes into COLUP1, color for player1 and missile1

        .byte $64   ; purple     - goes into COLUPF, color for Timer

        .byte $00   ; black      - goes into COLUBK, color for background

        .byte $0A   ; light grey - goes into COLUPF, color for Arena (after Timer is drawn)

        .byte $0E   ; white      - goes into COLUP0, color for player0 and missile0

        .byte $06   ; dark grey  - goes into COLUP1, color for player1 and missile1

        .byte $04   ; dark grey  - goes into COLUPF, color for Timer

        .byte $00   ; black      - goes into COLUBK, color for background

For testing, I've set it up so the Right Difficulty switch is used to determine if the game is a one or two player game for which graphics to use for player1:

ldx #0

        bit SWCHB

        bpl TwoPlayer

        ldx #1

TwoPlayer:

    ; Player1Ptr = BoxGfx + HUMAN\_HEIGHT - 1 - Y position

        lda ShapePtrLow,x

        sec

        sbc Temp

        sta Player1Ptr

        lda ShapePtrHi,x

        sbc #0

        sta Player1Ptr+1

        rts

ShapePtrLow:

        .byte <(HumanGfx + HUMAN\_HEIGHT - 1)

        .byte <(BoxGfx + HUMAN\_HEIGHT - 1)

ShapePtrHi:

        .byte >(HumanGfx + HUMAN\_HEIGHT - 1)

        .byte >(BoxGfx + HUMAN\_HEIGHT - 1)

Right Difficulty = B:

A screenshot of a video game

Description automatically generated

Right Difficulty = A:

A screenshot of a video game

Description automatically generated

# Step 7: Draw the Playfield

For this update, we're going to modify the Arena Loop to draw the Arena using the playfield. The new Arena loop has these new changes:

ArenaLoop:                  ;   27 - (currently 7 from bpl ArenaLoop)

        tya                 ; 2 29 - 2LK loop counter in A for testing

        and #%11            ; 2 31 - test for every 4th time through the loop,

        bne SkipX           ; 2 33 (3 34) branch if not 4th time

        inx                 ; 2 35 - if 4th time, increase X so new playfield data is used

SkipX:                      ;   35 - use 35 as it's the longest path here

...

    ; start of line 1 of the 2LK

        sta GRP1            ; 3  3 - @0-22, update player1 graphics

        lda ArenaPF0,x      ; 4  7 - get current scanline's playfield pattern

        sta PF0             ; 3 10 - @0-22 and update it

        lda ArenaPF1,x      ; 4 14 - get current scanline's playfield pattern

        sta PF1             ; 3 17 - @71-28 and update it

        lda ArenaPF2,x      ; 4 21 - get current scanline's playfield pattern

        sta PF2             ; 3 24 - @60-39

...

    ; start of line 2 of the 2LK

        sta GRP0            ; 3  3 - @0-22, update player0 graphics

        dey                 ; 2  5 - decrease the 2LK loop counter

        bne ArenaLoop       ; 2  7 - (3  8) branch if there's more Arena to draw

        sty PF0             ; 3 10 - Y is 0, blank out playfield

        sty PF1             ; 3 13 - Y is 0, blank out playfield

        sty PF2             ; 3 16 - Y is 0, blank out playfield

        rts                 ; 6 22 - ReTurn from Subroutine

The first change is we're using X as an index into the playfield graphic data. We're changing X every fourth time thru the 2LK, so each byte of playfield data will be used over 8 scanlines. This saves a bit of ROM.

Second change is all 3 playfield registers (PF0, PF1 and PF2) are now updated, and they're only updated on line 1 of our 2LK.

Third change is on line 2, the bpl ArenaLoop is now a bne ArenaLoop else the bottom row of playfield data was only used for 2 scanlines instead of 8. We also blank out the playfield registers when we are done drawing the playfield. The bne change also impacted Overscan—TIM64T was originally set to 32, it's now set to 35.

The playfield data looks like this in jEdit:

A screenshot of a computer

Description automatically generated

And this onscreen:

A screen shot of a game

Description automatically generated

Lastly we added some collision detection code. Some space was allocated in RAM:

   ;save player locations for playfield collision logic

SavedX:         ds 2    ; stored in $A1-A2

SavedY:         ds 2    ; stored in $A3-A4

Then the Process Joystick routines save the current X and Y values before processing the joystick:

PJloop:

        ldy ObjectX,x   ; save original X location so the player can be

        sty SavedX,x    ;   bounced back upon colliding with the playfield

        ldy ObjectY,x   ; save original Y location so the player can be

        sty SavedY,x    ;   bounced back upon colliding with the playfield

Finally OverScan was modified to move the players back to their previous X and Y location if a collision was detected:

; Test if player collided with playfield

        bit CXP0FB      ; N = player0/playfield, V=player0/ball

        bpl notP0PF     ; if N is off, then player0 did not collide with playfield

        lda SavedX      ; recall saved X

        sta ObjectX     ; and move player back to it

        lda SavedY      ; recall saved Y

        sta ObjectY     ; and move player back to it

notP0PF:

        bit CXP1FB      ; N = player1/playfield, V=player1/ball

        bpl notP1PF     ; if N is off, then player1 did not collide with playfield

        lda SavedX+1    ; recall saved X

        sta ObjectX+1   ; and move player back to it

        lda SavedY+1    ; recall saved Y

        sta ObjectY+1   ; and move player back to it

notP1PF:

# Step 8: Select and Reset Support

For this update we're adding initial support for the Select and Reset buttons. For this we're adding a new RAM variable called GameState to keep track of "Game Active" vs "Game Over".

    ; D7, 1=Game Active, 0=Game Over

GameState:      ds 1    ; stored in $A7

We're going to use D7 to denote the state as we can easily test D7 (as well as D6) by using the BIT command. You can see this in the revised Vertical Blank routine were we test GameState to determine if UpdateTimer and ProcessJoystick should be skipped over:

VerticalBlank:

        jsr ProcessSwitches

        bit GameState

        bpl NotActive

        jsr UpdateTimer

        jsr ProcessJoystick

NotActive:

        jsr PositionObjects

        jsr SetObjectColors

        jsr PrepScoreForDisplay

        rts             ; ReTurn from Subroutine

ProcessSwitches will check SWCHB to see if RESET is pressed. If so, it'll start up a new game. If not, it'll check if SELECT is pressed, and if so cancel an active game.

ProcessSwitches:

        lda SWCHB       ; load in the state of the switches

        lsr             ; D0 is now in C

        bcs NotReset    ; if D0 was on, the RESET switch was not held

        jsr InitPos     ; Prep for new game

        lda #%10000000

        sta GameState   ; set D7 on to signify Game Active

        rts

NotReset:

        lsr             ; D1 is now in C

        bcs NotSelect

        lda #0

        sta GameState   ; clear D7 to signify Game Over

NotSelect:

        rts

In the next update ProcessSwitches will be expanded upon so that the Select routine will let you select a game variation (and if you check the source you'll see a new Arena layout is already in place for that).

In order to visually show you that the game is over, I've revised the Color routines to color cycle if the game is not active.

SetObjectColors:

        lda #$FF

        sta Temp2       ; default to color mask

        and ColorCycle  ; color cycle

        bit GameState

        bpl SOCgameover

        lda #0          ; if game is active, no color cycle

SOCgameover:

        sta Temp

        ldx #4          ; we're going to set 5 colors (0-4)

        ldy #4          ; default to the color entries in the table (0-4)

        lda SWCHB       ; read the state of the console switches

        and #%00001000  ; test state of D3, the TV Type switch

        bne SOCloop     ; if D3=1 then use color

        ldy #$0f

        sty Temp2       ; set B&W mask

        ldy #9          ; else use the b&w entries in the table (5-9)

SOCloop:

        lda Colors,y    ; get the color or b&w value

        eor Temp        ; color cycle

        and Temp2       ; B&W mask

        sta COLUP0-1,x  ; and set it

        dey             ; decrease Y

        dex             ; decrease X

        bne SOCloop     ; Branch Not Equal to Zero

        lda Colors,y    ; get the Arena color

        eor Temp        ; color cycle

        and Temp2       ; B&W mask

        sta ArenaColor  ; save in RAM for Kernal Usage

        rts             ; ReTurn from Subroutine

Color cycle example:

A yellow and blue rectangular object with numbers

Description automatically generated

B&W Color Cycle example:

A screen shot of a video game

Description automatically generated

# Step 9: Game Variations

It's common for Atari games to have a number of game variations. To simplify the logic, the variations are usually driven by individual bits and/or groups of bits within a single byte that holds the game variation. A good example of that would be Space Invaders—check out the game matrix from the manual:

A white card with multicolored squares

Description automatically generated with medium confidence

A byte is comprised of 8 bits, usually numbered 0-7 where 7 is the leftmost bit as in 76543210.

For Space Invaders, the bits in the game variation are used in this fashion:

* 7 - not used
* 654 - selects 1 of 7 player variations
* Invisible Invaders
* Fast Bombs
* ZigZagging Bombs
* Moving Shields

Humans start counting at 1, but computers start at 0, so if you select game variation 12, internally it's really 11. 11 in binary is %00001011, which means bits 0, 1 and 3 are all turned on so the game variation has Invisible Invaders, ZigZagging Bombs and Moving Shields. You can confirm that by looking at column 12 of the Game Matrix above.

With this update to Collect, we're using bits 1 and 0 to give us 4 game variations:

* 765432 - not used
* 1 - Arena 1 or 2
* 0 - # of players

If we have space at the end of the project, I plan to add some additional arenas. If we can add 2 more we'd just start using bit 2 and let the game Variation go from 1-8:

* 76543 - not used
* 21 - Arena 1, 2, 3 or 4
* 0 - # of players

If we can add 6 more we'd add bit 3 and let the game variation go from 1-16

* 76543 - not used
* 321 - Arena 1, 2, 3, 4, 5, 6, 7 or 8
* 0 - # of players

The ProcessSwitches routine has been modified so that hitting Select will increment the new variable Variation. It will also limit Variation to only the values 0-3. After changing Variation, the left score will be set to show Variation+1 as Humans prefer to see 1-4 instead of 0-3. The Right Score will be used to show the # of players, either 1 or 2.

NotReset:

        lsr             ; D1 is now in C

        bcs NotSelect   ; if D1 was on, the SELECT switch was not held

        lda #0

        sta GameState   ; clear D7 to signify Game Over

        ldx Variation   ; Get the Game Variation

        inx             ; and increase it

        txa             ; transfer it to A

        and #%00000011  ; limit Variation to 0-3

        sta Variation   ; save it

        tax             ; transfer it to X

        inx             ; and increase it by 1 for the human readable varation 1-4

        stx Score       ; save in Score so it shows on left side

        ldy #1          ; default to showing 1 player variation

        lsr             ; D0 of Variation, # of players, now in Carry flag

        bcc Not2        ; if Carry is clear, then show 1 player

        iny             ; else set Y to 2 to show 2 players

Not2:

        ror Players     ; put Carry into D7 for BIT testing of # of players

        sty Score+1     ; show the human readable # of players on right side

NotSelect:

        rts

The routine works, but when Select is pressed the game variation will rapidly change making it difficult to select a specific game variation. You can see that in the build in my blog entry.

To fix that, we'll add a SelectDelay variable so that holding down SELECT will only result in Variation changing at the rate of once per second. However, if the user rapidly presses/releases SELECT then Variation will also rapidly change.

ProcessSwitches:

        lda SWCHB       ; load in the state of the switches

        lsr             ; D0 is now in C

        bcs NotReset    ; if D0 was on, the RESET switch was not held

        jsr InitPos     ; Prep for new game

        lda #%10000000

        sta GameState   ; set D7 on to signify Game Active

        bne NotSelect   ; clear SelectDelay

NotReset:

        lsr             ; D1 is now in C

        bcs NotSelect   ; if D1 was on, the SELECT switch was not held

        lda #0

        sta GameState   ; clear D7 to signify Game Over

        lda SelectDelay ; do we need to delay the Select switch?

        beq SelectOK    ; if delay is 0 then no

        dec SelectDelay ; else decrement the delay

        rts             ; and exit the subroutine

SelectOK:

        lda #60         ; Set the Select Delay to 1 second

        sta SelectDelay ;

        ldx Variation   ; Get the Game Variation

        inx             ; and increase it

        txa             ; transfer it to A

        and #%00000011  ; limit Variation to 0-3

        sta Variation   ; save it

        tax             ; transfer it to X

        inx             ; and increase it by 1 for the human readable varation 1-4

        stx Score       ; save in Score so it shows on left side

        ldy #1          ; default to showing 1 player variation

        lsr             ; D0 of Variation, # of players, now in Carry flag

        bcc Not2        ; if Carry is clear, then show 1 player

        iny             ; else set Y to 2 to show 2 players

Not2:

        ror Players     ; put Carry into D7 for BIT testing of # of players

        sty Score+1     ; show the human readable # of players on right side

        rts

NotSelect:

        lda #0          ; clears SelectDelay if SELECT not held

        sta SelectDelay

        rts

The routine PositionObjects has been modified to use a Box Graphic for player1 if a 1 player game has been selected:

PositionObjects:

...

        lda Variation       ; get the game variation

        and #1              ; and find out if we're 1 or 2 player

        tax

    ; Player1Ptr = BoxGfx + HUMAN\_HEIGHT - 1 - Y position

        lda ShapePtrLow,x

        sec

        sbc Temp

        sta Player1Ptr

        lda ShapePtrHi,x

        sbc #0

        sta Player1Ptr+1

        rts

ShapePtrLow:

        .byte <(BoxGfx + HUMAN\_HEIGHT - 1)

        .byte <(HumanGfx + HUMAN\_HEIGHT - 1)

ShapePtrHi:

        .byte >(BoxGfx + HUMAN\_HEIGHT - 1)

        .byte >(HumanGfx + HUMAN\_HEIGHT - 1)

The Kernel has also been modified so that the correct Arena will be drawn. A little bit after TimerBar: you'll find this:

TimerBar:

...

        lda Variation       ; 3 20

        lsr                 ; 2 22 - which Arena to show

        tay                 ; 2 24 - set for index

        ldx ArenaOffset,y   ; 4 28 - set X for which arena to draw

        lda ArenaPF0,x      ; 4 32 - reflect and priority for playfield

        and #%00000111      ; 2 34 - get the lower 3 bits for CTRLPF

        ora #%00110000      ; 2 36 - set ball to display as 8x pixel

        sta CTRLPF          ; 3 39

...

ArenaOffset:

        .byte 0         ; Arena 1

        .byte 22        ; Arena 2

The lsr command shifts bit 1 down to bit 0 so that we end up with 0 or 1 for the Arena number. That's used to set X to either 0 or 22 via the command ldx ArenaOffset,y.

I also added code to update CTRLPF based on the first PF0 data byte for the selected Arena. CTRLPF uses its bits like this:

* 76 - not used
* 54 - set width of BALL object
* 3 - not used
* 2 - Playfield Priority
* 1 - Score Mode
* 0 - Reflected Playfield

Since PF0 only uses bits 7654, also known as the upper nybble of the byte, we can use the lower nybble to hold extra information to specify whether or not the selected Arena uses Playfield Priority (as opposed to Player Priority) or has a Reflected Playfield(as opposed to Repeated Playfield). We could even specify Score Mode which would just color the two sides of the playfield to match the colors of the players (like in the score display).

ArenaPF0:   ; PF0 is drawn in reverse order, and only the upper nybble

        .byte %11110001 ; Arena 1   lower nybble controls playfield, set for REFLECT

        .byte %00010000

        .byte %00010000

        .byte %00010000

...

        .byte %11110100 ; Arena 2 - lower nybble controls playfield, set for PLAYFIELD PRIORITY

        .byte %00010000

        .byte %00010000

        .byte %00010000

Game Variation 2, Arena 1 with 2 players. Arena 1 features Reflected Playfield and Player Priority:

A screen shot of a video game

Description automatically generatedGame

Variation 3, Arena 2 with 1 player. Arena 2 features Repeated Playfield and Playfield Priority:

A screen shot of a game

Description automatically generated

Look at the left Humanoid's head in each screenshot to see the difference that setting Playfield Priority makes. You might remember this being used in some games like Combat where the planes go behind the clouds.

A screenshot of a video game

Description automatically generatedJust for fun, here's Arena 2 with SCORE mode set (I've moved the players to the side of the screen they didn't start on):

A screenshot of a video game

Description automatically generatedThe code change for that is:

.byte %11110010 ; Arena 2 - lower nybble controls playfield, set for SCORE

# Step 10: “Random Numbers”