Work In Progress

Commander X16

VERA
FX Update

Introduction

This is a guide for the VERA FX Update. It is intended to explain the features of this update to VERA, how they work and how to use them. The FX update and this document is developed by the (discord) community of the Commander X16.

The goal of the VERA FX update is to enable certain “special effects” features that would otherwise be very slow or practically impossible without the update. It does not require any new hardware and is backwards compatible with the current (bitstream) version of VERA.

It is inspired and is in some ways similar to the “Super FX”-chip extension found on some SNES cartridges: it helps to create certain (3D-like) shapes or effects on screen.

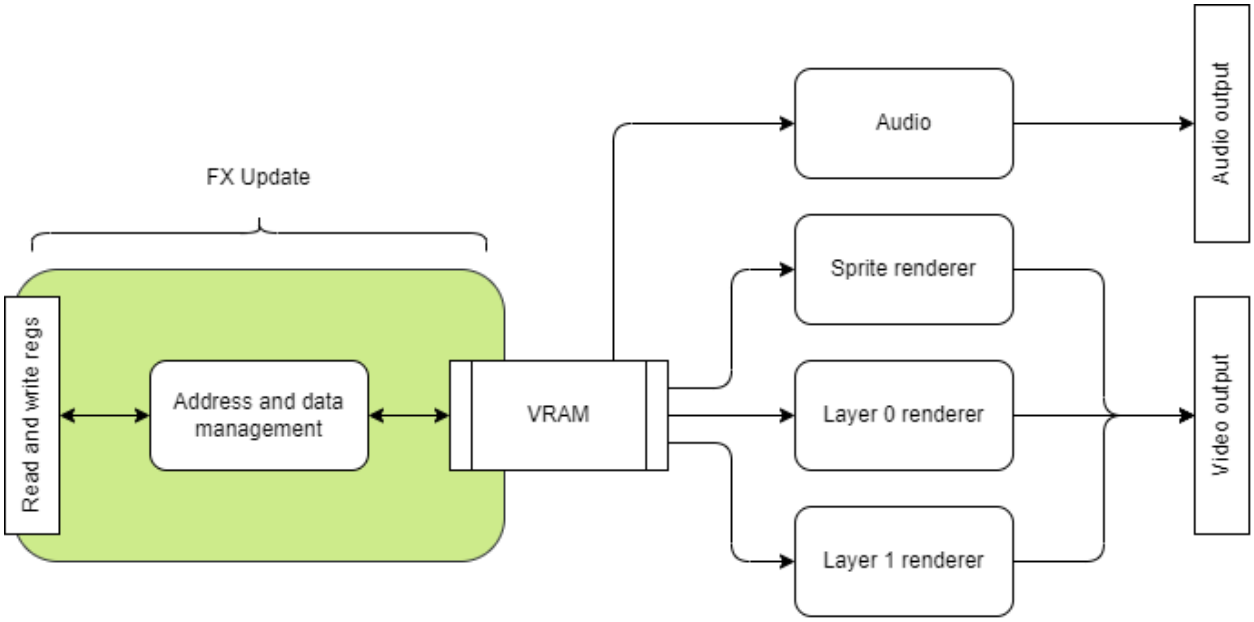
Philosophy

The FX Update mainly adds “helpers” inside of VERA that can be used by the CPU. There is no “magic button” that allows you to do 3D graphics for example. It mainly *helps* at certain CPU time-consuming tasks, most notably the ones that are present in the (deep) inner-loop of a game/graphics engine. The FX Update does therefore not fundamentally change the architecture or nature of VERA, it extends and improves it.

In other words: the CPU is still the orchestrator of all that is done, but it is alleviated from certain operations where it is not (very) good at or does not have direct access to.

FX Update extends *addressing* modes, it does not add or extend *renderers*.

The scope of the FX update is limited to the part that does the address and data management/logic as can be seen in this (very simplified) block diagram of VERA:



So even though the FX update allows one to create various graphical effects, these effects are not *implemented* in the renderers themselves. The helpers can produce these effects by accessing (that is: reading or writing) VRAM in a clever way. This allows for more versatility. It also means there is no “magic button” (and therefore no “black box”) to turn on an effect.

Impressions

Below are a few screenshots of test programs and demos that use the new features of the FX update. It gives you an impression of what kind of results can be achieved.

TODO

Features

The FX Update adds the following (high level) features:

- A **32-bit cache** that gets filled when reading from VRAM (using an `lda DATA0` or `lda DATA1`) one byte (or nibble) at the time but can be *written* to VRAM 4 bytes at the time (half of a "micro blitter**"** if you will) when doing an `sta DATA0` or `sta DATA1`. During this write it is also possible to mask parts of the 32-bits (on a nibble level) when writing to VRAM. Furthermore: which part of the cache is filled can be tightly controlled.
    - Use cases:
        - Filling the screen with one color (aka "clear screen") really fast
        - Copying from VRAM to VRAM
        - Storing pixels that were retrieved using the affine helper and writing them to a bitmap screen really fast
        - Fill a horizontal line (of a polygon) really fast
        - Draw (32-bit aligned) dithering patterns on a horizontal line
        - Acts as input for the 16-bit x 16-bit multiplier and 32-bit accumulator
- The ability to **write transparent pixels**: when writing to VRAM (using DATA0 or DATA1) the value 0 can be treated differently: it can be ignored and no VRAM bytes will be changed. This is also true when used in combination with the 32-bit cache.
    - Use cases:
        - Copying bitmap/sprite data that contains transparent pixels
        - Drawing pixels of rotated/sheared/scaled images -so in combination with the affine helper- in clipping mode.
- A **4-bit** addressing mode that allows you to manipulate the address on a nibble-level. Also added is a nibble incrementer (or decrementer).
    - Use cases:
        - Writing individual 4-bit color pixels incrementally (one at a time)
        - Starting to write 4-bit pixels at a precise pixel location, without disturbing nearby pixels
        - Reading 4-bits from VRAM into the 32-bit cache (this is especially useful when using the affine helper)

- An **affine helper** that allows you to sample (tiled) pixels in such a way that rotation, scaling and shearing is possible. Also (with extra work on the CPU side) a perspective transformation can be achieved. This mode uses a tilemap of (8x8 pixel) tiles. This way tiles can be reused. A map can either be clipped (when combined with transparent writes) or repeated.
    - Use cases:
        - Scaling and rotating of sprites or (large) bitmaps on the screen
        - Creating a perspective/mode7-style view (like “Mario Kart”)
- A **line-draw** helper which helps you draw lines on a screen by incrementing the address given a certain (settable) slope. Effectively implementing a Bresenham's line algorithm.
    - Use cases:
        - Fast wireframe 3D drawing
        - Any moment you want to draw many diagonal lines quickly
- A **polygon filler** helper which helps you fill polygons/triangles really fast. It allows you to set *two* slopes: one for each side of a polygon part that has to be filled (see for example here). The helper will tell you the length of the current line to fill and -after filling it yourself- will let you trigger it to increment the x-position of both sides of the polygon part and also sets the address to the starting pixel. Also, the way the helper gives back the fill line length is packed in such a way to make the use of jump tables (a 65**C**02 feature) *very* efficient.
    - Use cases:
        - Polygon 3D drawing
        - Any moment you want to draw a triangle/polygon quickly
- A **16-bit x 16-bit multiplier and 32-bit accumulator** which allows for signed 16-bit multiplication and 32-bit accumulation. This works from 32-bit cache to VRAM. Special cache-index and address-incrementors are added to facilitate bulk math operations.
    - Use cases:
        - Doing (really fast) *bulk* 3D math → needed for doing polygon 3D
            - Projection, Rotation, Culling, Light, etc
        - Doing single math operations (still very fast despite VRAM “roundtrip”)

Usage of the features

In this section the use of the FX update features is explained.

The use of DCSEL

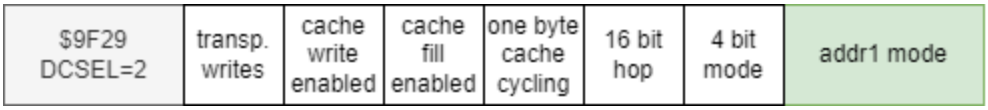
VERA is mapped as 32 8-bit registers in the memory space of the Commander X16, starting at address $9F20 and ending at $9F3F. Many of these are (fully) used, but some bits remain unused. The DCSEL bits in register $9F25 (also called CTRL) has been extended to 6-bits to allow for the 4 registers $9F29-$9F2C to have multiple meanings.

For the FX update the values 2, 3, 4, 5 and 6 are used for DCSEL. This effectively allows for 20 8-bit registers to be used for the FX update. Note that 17 of these registers are *write-only*, 2 of them are *read-only* and 1 is both *readable and writable*,

***Important**: when DCSEL is kept 0 or 1 the behavior of VERA is exactly the same as it was before the FX update. Since the higher bits of DCSEL were unused before, this assures that the FX update is backwards compatible with the current bitstream of VERA.*

Setting the addr1 mode

When DCSEL=2 the main FX configuration register becomes available ($9F29), which is both readable and writable. The 2 lower bits are the addr1 mode bits, which will change the behavior of how and when ADDR1 is updated. This puts the FX helpers in a certain “role”.

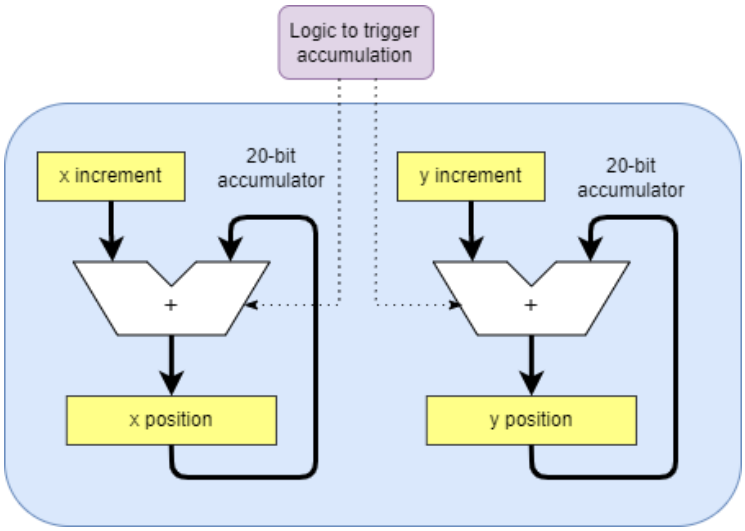


In the following sections each of the possible addr1 modes is explained. For each of these modes certain other registers become relevant, which will be explained as well. By default the addr1 is set to 0 (=00b), which is the **normal** and already known behavior of VERA when it comes to ADDR1.

The two 20-bit accumulators

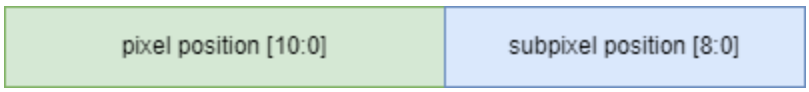
Before we go deeper into each addr1 mode this section gives a little bit of technical background information about what all the addr1-modes use “under the hood”: the two 20-bit accumulators.

There are two 20-bit accumulators used in the FX update. They basically contain the coordinates of positions on the screen or on a tile map. In this (simplified) diagram you can see they essentially consists for 2 increment values, 2 position values and 2 adders:



In many cases, when a read or write to DATA1 (or sometimes DATA0) occurs the accumulators will be triggered and will add another increment value.

The 20-bits represent a fixed-point value that is split in the following way (11 and 9 bits):



This means that the smallest value can be 1/512th of a pixel and the largest value can be (almost) 2048 pixels.

As an example: the number 621.668 (decimal) would be stored as 01001101101.101010110 Since 621 = 01001101101 (binary) and 0.668 is approx. 342/512 and 342 = 101010110 (binary).

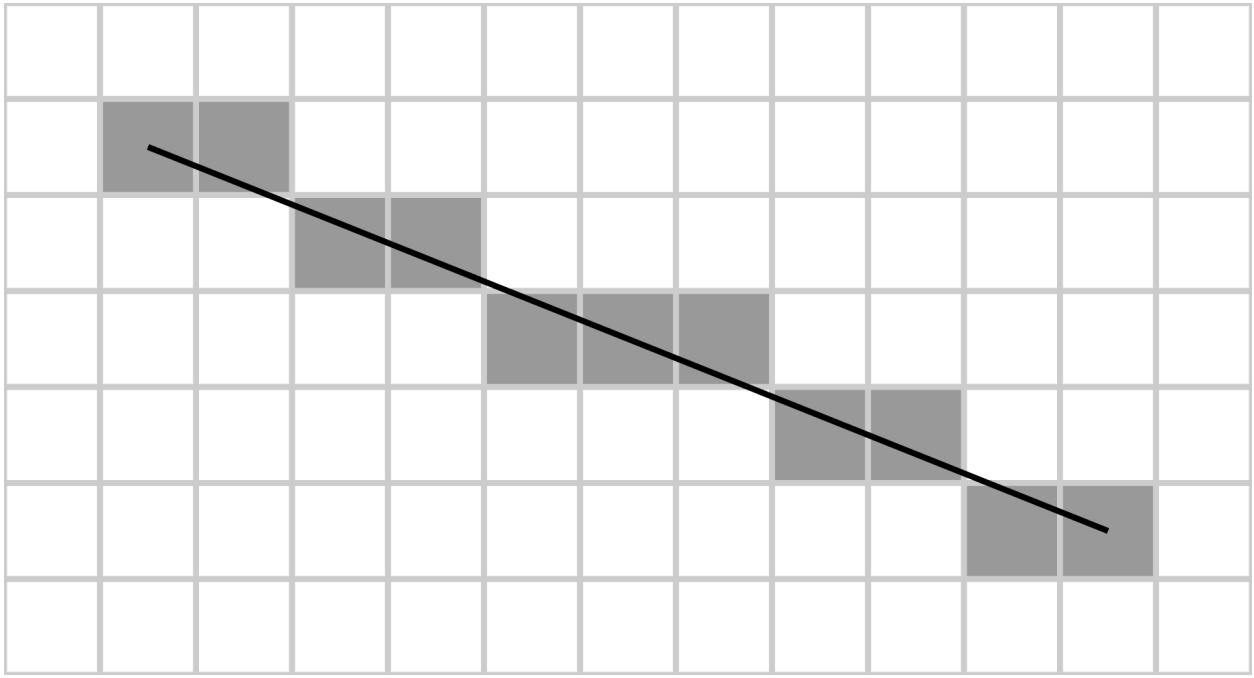
Here is how and where they are used:

- **Line draw mode**: only the X accumulator is used for keeping track of either the X or Y subpixel position (also called the “error value”) in the Bresenham’s Algorithm. Essentially only the lower 10 bits (not all 20) are used of the X accumulator: the 9 subpixel position bits and the lowest bit of the pixel position (which is used as an “overflow” bit).
- **Polygon filler mode**: both the X and Y accumulator are used but as X1 and X2 positions: the start-X and end-X position of a horizontal fill line. For each line to fill, they are updated with their increments.
- **Affine helper mode**: both the X and Y accumulator are used as is. They keep track of the positions in a tiled map that is being “sampled”. By setting the X and Y increments in various ways, affine transformations can be accomplished this way.

Line draw helper

When addr1 mode is set to 1 (=01b) the **line draw helper** is enabled. In order to draw a line several other registers have to be set correctly. Before explaining those a bit of information about the Bresenham's line draw algorithm is needed.

Below is a depiction of a line draw of pixels on a screen (from wikipedia):



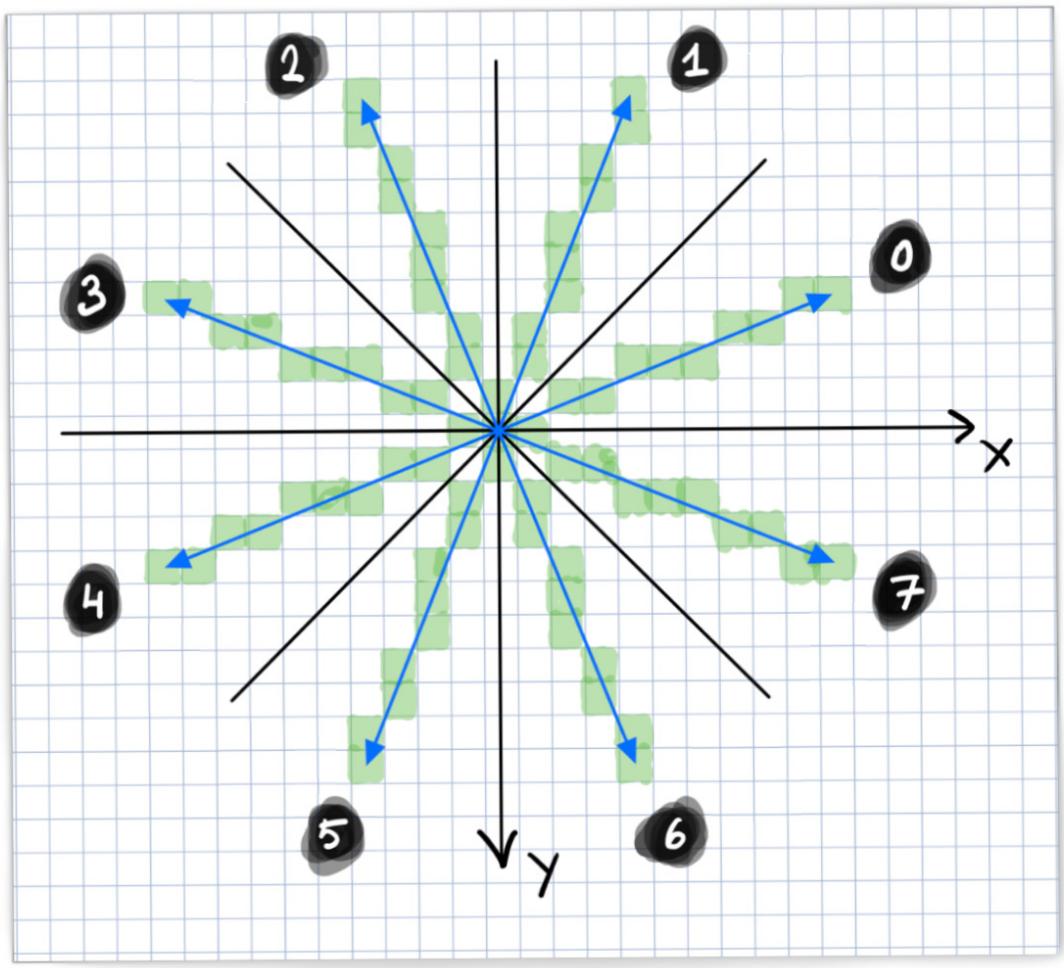
As can be seen by the black line (in the top left): this is the intended "virtual line" that starts in the *middle* of the starting pixel and ends in the *middle* of the ending pixel. This line has a certain slope (=dy/dx).

The line drawing algorithm starts by drawing a single pixel at the starting position of the virtual line. At each step the pixel-cursor will always move one to the *right*. But if during a step the black line also crosses a horizontal pixel border, the pixel-cursor is *also* moved *down* by one. After moving the pixel-cursor a new pixel can be drawn at that position.

In order to implement this in the *current* VERA a lot of inconvenient address-handling is needed. To alleviate this the line draw helper can do this address-handling for you.

The Octants

The above illustration shows what to do when a line is drawn at a certain slope/angle: from the top-left to the bottom-right and more to the right than down. But this method can be re-used for all 8 possible directions, also called “octants”. Below is an illustration of these octants: (taken from here, which contains a good explanation of the algorithm)

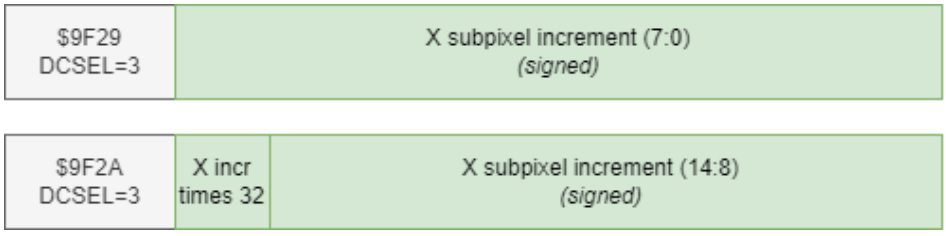


Note that in our example a line was drawn in octant 7: always move to the right, sometimes move down. The same algorithm can be applied to any of these octants, we just need to adjust the directions of our steps and whether we cross a vertical or horizontal border.

Setting up the line draw helper

In order to use the line draw helper, there are a few steps that have to be taken to setup using the line draw helper (assuming 320 pixel wide screen here):

- Set ADDR1 to the address of the starting pixel
    - *Tip*: use a 240-entry lookup table to find the address of the left-most pixel of each line of the screen and add your starting x-coordinate to that.
- Determine the octant you are going to draw in (read up on the Bresenham algo)
- Set your ADDR1 increment in the direction you will *always* increment each step (+1, -1, -320 or +320)
- Set your ADDR0 increment in the direction you will only increment when a horizontal or vertical pixel-line is crossed (+1, -1, -320 or +320)
- Calculate your slope (dy/dx or dx/dy) depending on which octant you are drawing in. In "line draw"-mode a slope should always be between (and including) 0 and 1.0.
    - *Tip*: generate a slope table to speed up this process. This can be memory expensive though.
- Set your slope into the two "X subpixel increment"[1] registers (DCSEL=3, see below). Note that increment registers are 15-bit signed fixed-point number:
    - 6 bits for the pixel increment
    - 9 bits for the subpixel increment
    - 1 additional bit that indicates the actual value should be multiplied by 32



[1] The line draw helper uses one of the two available incrementers, namely the X incrementer. However depending on the octant you are drawing in, this incrementer will be used to depict either x or y pixel increments. So the "X" should not be taken literally here, it just means the "first of the two" incrementors.

- Due to the fact that we are in line draw mode, by setting the high bits of the "X subpixel increment" ($9F2A, DCSEL=3), the "X *subpixel* position" (the lower 9 bits of the position) are automatically set to half a pixel. Furthermore the lowest bit of the *pixel* position (which acts as an overflow bit) is set to 0 as well.[2] This effectively sets the starting x-position to 0.5 of a pixel.

Example code

Below is example code that does the setup of the line draw helper. Note that (for simplicity) we assume layer 0 to be in bitmap mode (320 wide, 8bpp) with a starting VRAM address at $00000.

```
lda #%00000100          ; DCSEL=2, ADDRSEL=0
sta VERA_CTRL

lda #%11100000          ; ADDR0 increment: +320 bytes
sta VERA_ADDR_BANK

lda #%00000001          ; Entering *line draw mode*
sta $9F29

lda #%00000101          ; DCSEL=2, ADDRSEL=1
sta VERA_CTRL

lda #%00010000          ; ADDR1 increment: +1 byte
sta VERA_ADDR_BANK
lda #0                  ; Setting start to $00000
sta VERA_ADDR_HIGH
lda #0                  ; Setting start to $00000
sta VERA_ADDR_LOW

lda #%00000110          ; DCSEL=3, ADDRSEL=0
sta VERA_CTRL

; Note: 73 is just a nice looking slope ;)
; 73 means: for each x pixel-step there is 73/256th y pixel-step
lda #<(73<<1)           ; X increment low
sta $9F29
lda #>(73<<1)           ; X increment high
sta $9F2A
```

[2] There is no need for setting the higher bits of the X pixel position, since only the lower 9 subpixel bits of the position and 1 pixel position bit (=overflow bit) is used in line draw mode.

For each octant you draw in, you have to set the increments to ADDR0 and ADDR1 accordingly. The example above **always** moves one pixel to the *right* (so increment of ADDR1 is +1) and **sometimes** moves one pixel *down* (so increment of ADDR0 is +320).

There are 8 combinations of settings ADDR0/ADDR1 to +1/-1/+320/-320 that correspond to all octants. When creating a general line draw routine one should determine which octant is the case and also determine/calculate the slope needed within that octant.

Drawing the line

When the line draw helper is properly set up you can start drawing the line. It is assumed that the CPU has determined how many pixels have to be drawn.

Drawing the line is simple: you simply do `sta DATA1` as many times as needed, where `a` contains the color you want to draw. Using a different CPU register also works of course. Note that by reading from DATA1 the addr1 is also updated. If you use read and write to it intermittently a "dotted line" effect can be created.

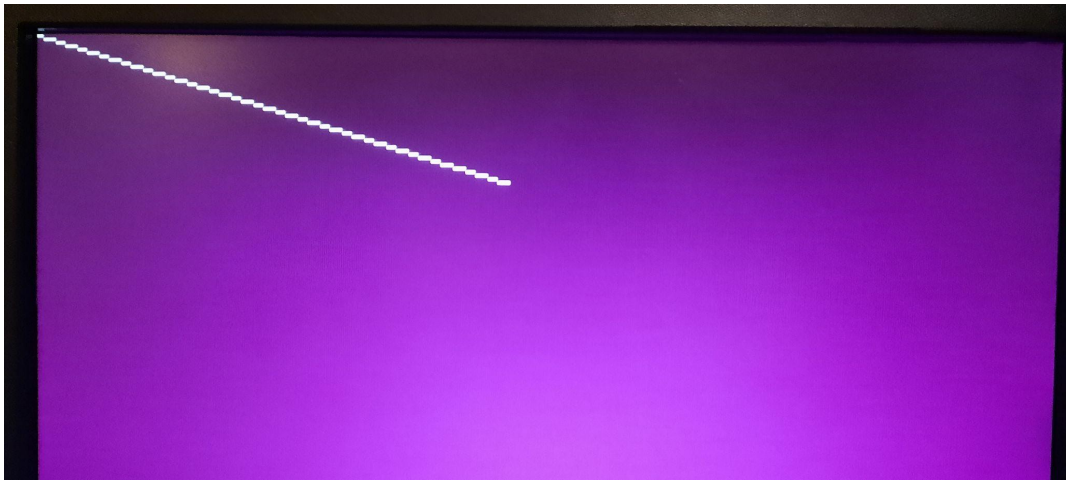
Example code

Below is some very simple code that (after setup of the line draw helper) will draw a diagonal line on the screen. Note that both color and the (horizontal) length of the line are hardcoded here for simplicity.

```
    ldx #150 ; Drawing 150 pixels to the right
    lda #1   ; White color

draw_line_next_pixel:
    sta VERA_DATA1
    dex
    bne draw_line_next_pixel
```

Note that no attempt is being made to make this code performant here, just readable. You could for example unroll this loop to speed it up a lot.

Below is a screenshot of what the above code will produce on the screen (screen has been cleared with a purple color beforehand).



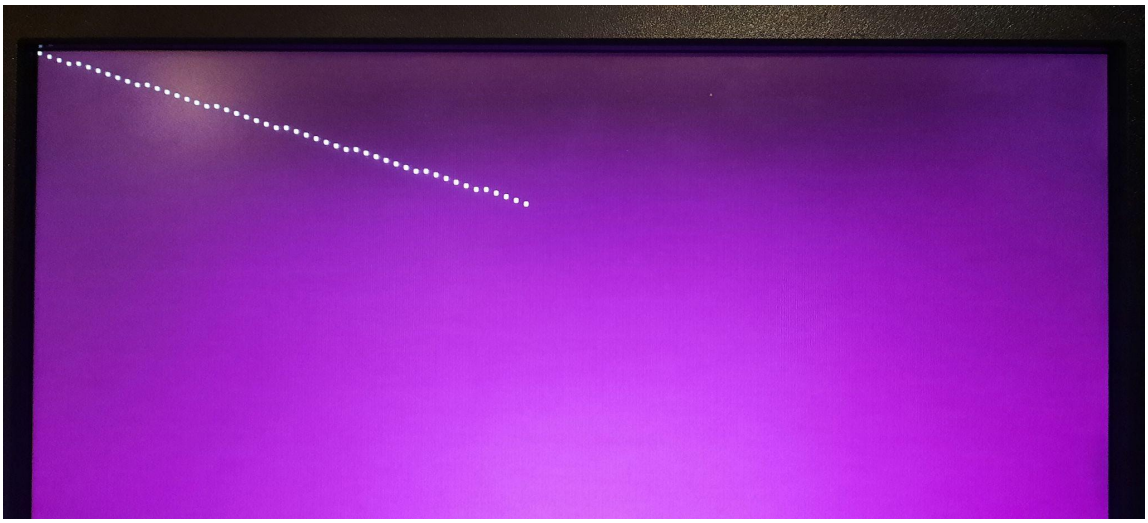
Here is an example of drawing a dotted line:

```
    ldx #150/3 ; Drawing 150 pixels to the right
    lda #1     ; White color

draw_dotted_line_next_pixel:
    sta VERA_DATA1  ; write pixel
    ldy VERA_DATA1  ; read (and ignore)[3]
    ldy VERA_DATA1  ; read (and ignore)
    dex
    bne draw_dotted_line_next_pixel
```

Note that we are *reading* from DATA1 (two thirds of the time), which does not write a pixel, but does increment our position. This creates the effect of *skipping* pixels on a line.

[3] We are using `ldy` to read from the DATA1 register. This will of course overwrite the y-register. Using the `bit` opcode would most likely be the better choice here. But since not everybody knows about the `bit` opcode, the example contains the more explicit load-opcode (`ldy`) and store-opcode (`sta`).

As you can see in this screenshot this will produce a dotted line on the screen:



4-bit line drawing

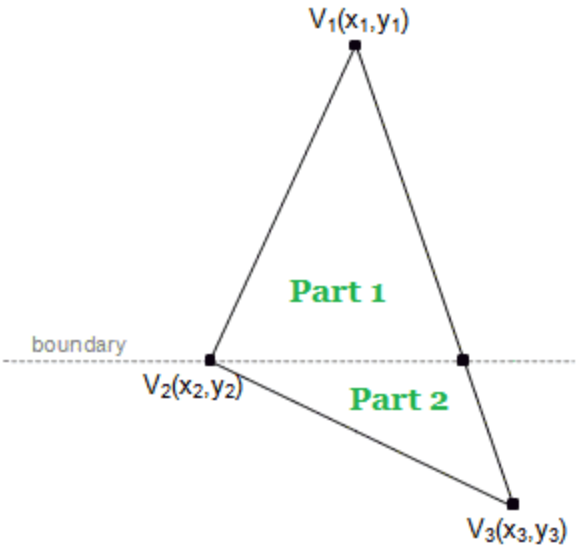
TODO: explain (and maybe example of) 4-bit line drawing

Polygon filler helper

When addr1 mode is set to 2 (=10b) the **polygon filler helper** is enabled. In order to draw a polygon or triangle several other registers have to be set correctly. Before explaining those a bit of information about the polygon filling is needed. In this documentation we are going to restrict ourselves to *triangle* filling. But the same mechanism can be used for more complex polygons.

The illustrations below are taken from [this webpage], which contains explanations of triangle rasterization techniques.

In essence the triangle is drawn in two parts. So the first step is to split the triangle into a top and bottom part:



Usually there is exactly *one* top point (V1), a left point (V2) and a right point (V3)[4]. The first part of the triangle is drawn from the top to either the y-position of V2 or V3 (shown as the "boundary" line in the diagram above). When the left point is higher than the right point then the y-position of the left point becomes the "boundary" line to partition the triangle. If it's the right point its y-position becomes the "boundary" line.

[4] Note: "left" and "right" is *not* determined by the x-*position* of the points, but by the *angle* with the top point. So the *slope/angle* towards the left point is more "clockwise" than towards the right one.

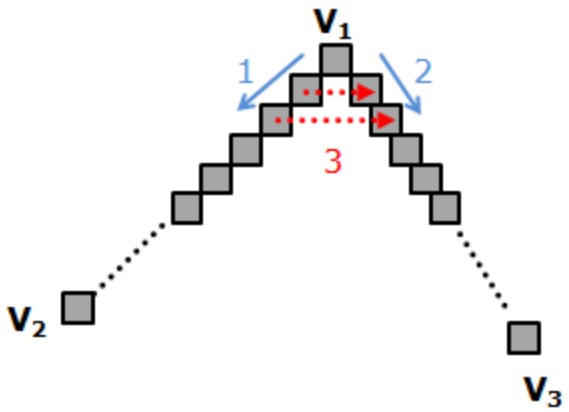
If the triangle has a “flat top” there are effectively *two top* points (the two points have the same y-position). In that case there is effectively no “Part 1” to draw, so it is skipped. This does require you to set up the two starting x-positions to the appropriate x-positions of the left and right top points (which you don’t have to do when drawing the top part first).

If the triangle has a “flat bottom” there are effectively *two bottom* points (with the same y-position). In that case there is effectively no “Part 2” to draw, so it is skipped.

Filling horizontal lines

Triangle filling/rasterization can be realized by essentially using the Bresenham’s Algorithm *twice*: one for the *left* slope of a triangle-part and one for the *right* slope of a triangle-part.

Below is an illustration,:



It shows a top point V1 and two lower points V2 (left) and V3 (right). The procedure goes as follows:

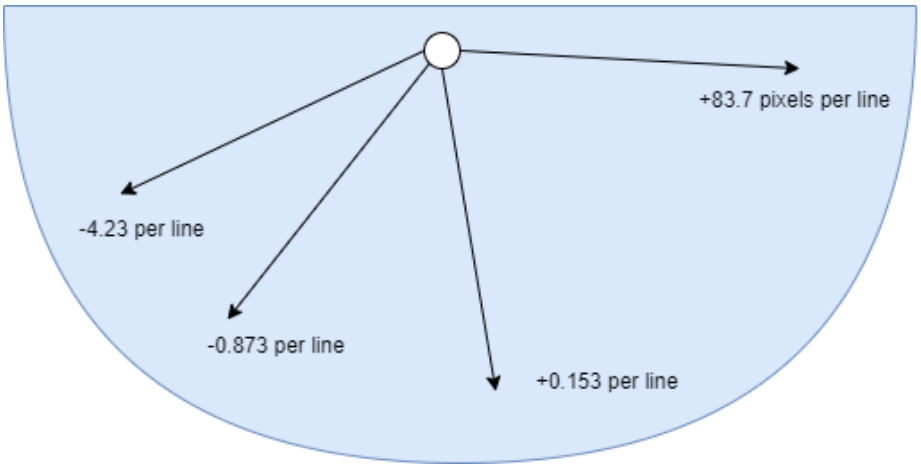
- Two accumulators are used. They contain the X-position of the gray squares (seen left and right for each horizontal line).
- Both of these are initially set to the X-position of V1 (the top gray square)
- Both slopes (the blue “1” and “2” in the illustration) are calculated using the X/Y coordinates of the V1,V2 and V1,V3 points respectively.
- For each scanline:
    - The slopes are used to increment or decrement the horizontal pixel position on the left and right side for each scan line (the gray squares).

    - By knowing these two (start and end) pixel positions for each horizontal line you can fill in the correct pixels for each line (red arrows).
- When the boundary is reached of the top part, one of the two slopes is changed.
- Each scanline is filled as before but now each fill line will become smaller and smaller until you reach the bottom point.

Setting up the polygon filler helper

In order to use the polygon filler helper, there are a few steps that have to be taken to set it up (assuming 320 pixel wide screen here):

- Set ADDR0 to the address of the y-position of the top point of the triangle and x=0 (so on the left of the screen). Set its increment to +320.
    - Note: ADDR0 is used as "base address" for calculating ADDR1 for each horizontal line of the triangle. ADDR0 should therefore start at the top of the triangle and increment exactly one line each time.
    - *Tip*: use a 240-entry lookup table to find the address of the left-most pixel of each line of the screen.
    - There is no need to set ADDR1. This is done by VERA.
- Calculate your slopes (dx/dy) for both the left and right point. Unlike the line draw helper, these slopes can be negative and can exceed 1.0. They are not dependent on octant, but cover the whole 180 degrees downwards. Below is an illustration of some (not so accurate) examples of increments:

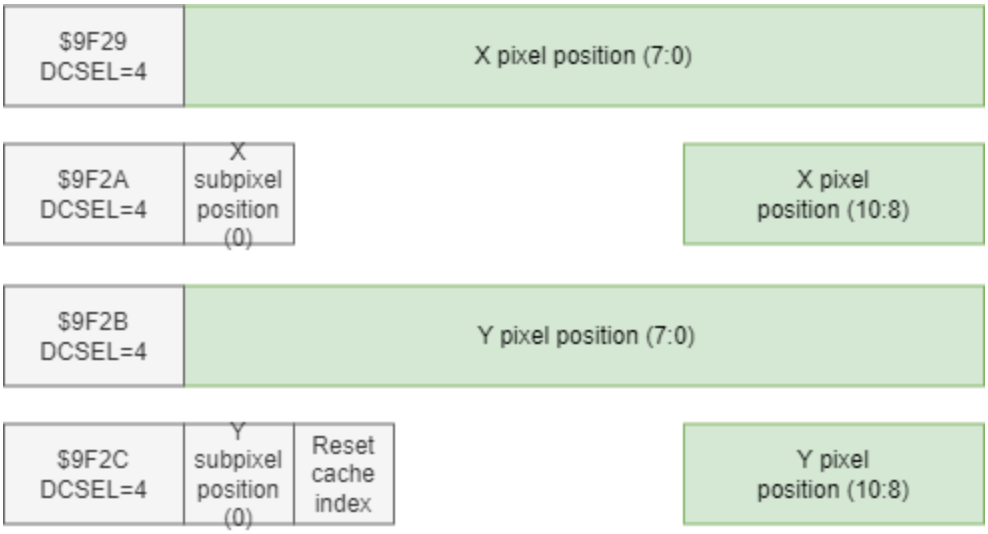


    - *Tip*: generate a slope table to speed up this process. This can be memory expensive though.
- Set your ADDR1 increment to +1 (or to +4 if you use 32-bit cache writes)
- Set your **left** slope into the two "**X** subpixel increment" registers and your **right** slope into the two "**Y** subpixel increment" registers (DCSEL=3, see below).
    - **Important**: They should be set to **half** the increment (or decrement) per horizontal line! This is because the polygon filler increments in *two steps per line*.
- Note that increment registers are 15-bit signed fixed-point number:
    - 6 bits for the pixel increment
    - 9 bits for the subpixel increment
    - 1 additional bit that indicates the actual value should be multiplied by 32

$9F29 DCSEL=3		X subpixel increment (7:0) *(signed)*
$9F2A DCSEL=3	X incr times 32	X subpixel increment (14:8) *(signed)*
$9F2B DCSEL=3		Y subpixel increment (7:0) *(signed)*
$9F2C DCSEL=3	Y incr times 32	Y subpixel increment (14:8) *(signed)*

- Due to the fact that we are in "polygon fill"-mode, by setting the high bits of the "X subpixel increment" ($9F2A, DCSEL=3), the "X *subpixel* position" (the lower 9 bits of the position) are automatically set to half a pixel. The same goes for the high bits of the Y increment ($9F2C, DCSEL=3) and Y subpixel position.

- Set the "X pixel position" *and* "Y pixel position" to the x-pixel-position of the top triangle point.



Example code

Below is example code that does the setup of the polygon filler helper. Note that (for simplicity) we assume layer 0 to be in bitmap mode (320 wide, 8bpp) with a starting VRAM address at $00000.

Some constants are assumed: `TRIANGLE_TOP_POINT_X=90` and `TRIANGLE_TOP_POINT_Y=20`.

```
    lda #%00000100          ; DCSEL=2, ADDRSEL=0
    sta VERA_CTRL

    lda #%11100000          ; ADDR0 increment: +320 bytes
    sta VERA_ADDR_BANK

    ; Note: we are setting ADDR0 to the leftmost pixel of a pixel row.
    lda #>(TRIANGLE_TOP_POINT_Y*320)
    sta VERA_ADDR_HIGH
    lda #<(TRIANGLE_TOP_POINT_Y*320)
    sta VERA_ADDR_LOW

    lda #%00000010          ; Entering *polygon filler mode*
```

```
sta $9F29

lda #%00000110          ; DCSEL=3, ADDRSEL=0
sta VERA_CTRL

; IMPORTANT: these increments are *HALF* steps!
lda #<(-110)            ; X1 increment low (signed)
sta $9F29
lda #>(-110)            ; X1 increment high (signed)
and #%01111111          ; increment is only 15-bits long
sta $9F2A
lda #<(380)             ; X2 increment low (signed)
sta $9F2B
lda #>(380)             ; X2 increment high (signed)
and #%01111111          ; increment is only 15-bits long
sta $9F2C

; Setting x1 and x2 pixel position

lda #%00001001          ; DCSEL=4, ADDRSEL=1
sta VERA_CTRL

lda #<TRIANGLE_TOP_POINT_X
sta $9F29               ; X (=X1) pixel position low [7:0]
sta $9F2B               ; Y (=X2) pixel position low [7:0]

lda #>TRIANGLE_TOP_POINT_X
sta $9F2A               ; X (=X1) pixel position high [10:8]
ora #%00100000          ; Reset subpixel position
sta $9F2C               ; Y (=X2) pixel position high [10:8]

lda #%00010000          ; ADDR1 increment: +1 byte
sta VERA_ADDR_BANK

ldy #1                ; White color
lda #150              ; Hardcoded amount of lines to draw
sta NUMBER_OF_ROWS

jsr draw_polygon_part_using_polygon_filler


; omitted code for the bottom part (see section further down)
```

Filling the triangle part(s)

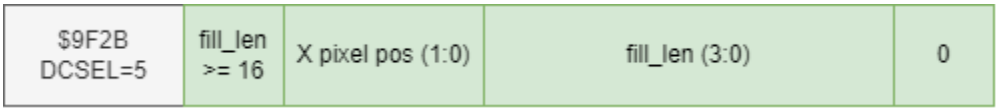
When the polygon filler helper is properly set up you can start filling the triangle(part). It is assumed that the CPU has determined how many fill lines have to be drawn.

Each line the polygon filler helper can do the following:

- Increment the X1 and X2 accumulators by their (half) increments
- Set the ADDR1 to the start address of a given fill line
- Give the CPU the number of pixels to fill in a line

Here are the steps that are needed for filling a triangle part with lines:[5]

- Read from DATA1
    - This will not return any useful data but will do two things in the background:
        - Increment/decrement the X1 and X2 positions[6] by their corresponding increment values.
        - Set ADDR1 to ADDR0 + X1
- Then read the “Fill length (low)”-register:



- If fill_len >= 16 then also read the “Fill length (high)”-register:

Important: when the two highest bits of fill_len (bits 8 and 9) are both 1, it means there is a *negative* fill length. The line should *not* be drawn!

[5] The instructions given here are not at all focused on performance, but on simply getting it to work. Much can be done to speed things up a *lot*. The fill-length registers in particular are meant to used as input to a jump-table (65C02 feature), but this is not done in the given instructions.
[6] In the context of the polygon filler it makes more sense to use “X1” and “X2” as the names for the positions the accumulators represent. In the registers they are called “X” and “Y”, since they are used in that sense in the “affine helper”-mode.

- Together they give you 10-bits of fill length (ignore the other bits for now). Since ADDR1 is already set properly you can immediately start drawing this number of pixels (given by fill_len).
- Then read from DATA0: this will (also) increment X1 and X2
- Check if all lines of this triangle part have been drawn, if not go to the first step.

To draw the second/bottom part of the triangle, you have to set either the X1 or the X2 increment registers to the correct slope (note: this also resets its subpixel position). When that is done you can use the same (above) loop to draw the bottom part of the triangle.

Example code

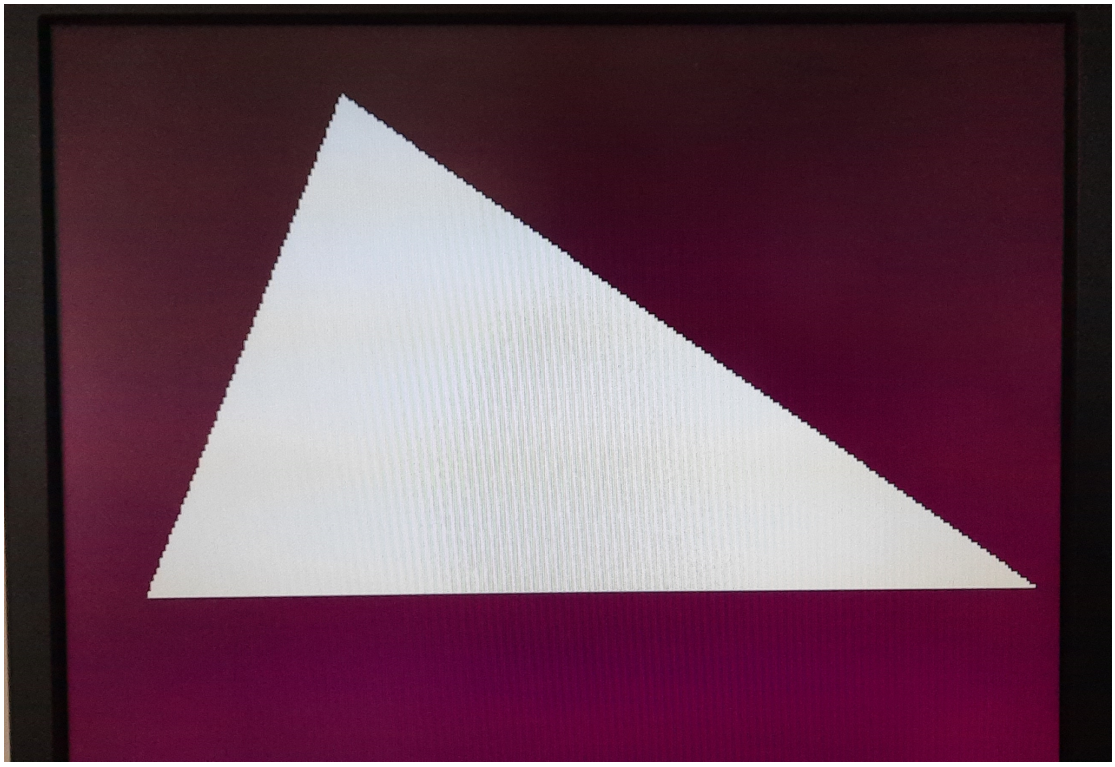
Below is example code that draws the top part of a triangle. It is very simplified and slow, but should give you a good idea of how the polygon filling helper actually works in practice.

There are three ZP-registers being used here: `FILL_LENGTH_LOW`, `FILL_LENGTH_HIGH` and `NUMBER_OF_ROWS`. This is

```
; Routine to draw a triangle part
draw_polygon_part_using_polygon_filler:

    lda #%00001010        ; DCSEL=5, ADDRSEL=0
    sta VERA_CTRL

polygon_fill_triangle_row_next:

    lda VERA_DATA1    ; This will do three things (inside of VERA):
                      ;   1) Increment the X1 and X2 positions.
                      ;   2) Calculate the fill_length value (= x2 -
x1)
                      ;   3) Set ADDR1 to ADDR0 + X1

    ; What we do below is SLOW: we are not using all the information
    ; we get here and are *only* reconstructing the 10-bit value.

    lda $9F2B             ; This contains: FILL_LENGTH >= 16,
X1[1:0],
                          ;                FILL_LENGTH[3:0], 0
    lsr
    and #%00000111        ; We keep the 3 lower bits (note that bit 3
                            is ALSO in the HIGH byte, so we discard
it)
```

```
    sta FILL_LENGTH_LOW   ; We now have 3 bits in FILL_LENGTH_LOW

    stz FILL_LENGTH_HIGH
    lda $9F2C             ; This contains: FILL_LENGTH[9:3], 0
    asl
    rol FILL_LENGTH_HIGH
    asl
    rol FILL_LENGTH_HIGH  ; FILL_LENGTH_HIGH now contains the two
                          highest bits: 8 and 9
    ora FILL_LENGTH_LOW
    sta FILL_LENGTH_LOW   ; FILL_LENGTH_LOW now contains all lower 8
                          bits

    tax
    beq done_fill_triangle_pixel  ; If x = 0, we don't have to draw
any
                                  pixels

polygon_fill_triangle_pixel_next:
    sty VERA_DATA1        ; This draws a single pixel
    dex
    bne polygon_fill_triangle_pixel_next

done_fill_triangle_pixel:

    ; We draw an additional FILL_LENGTH_HIGH * 256 pixels on this row
    lda FILL_LENGTH_HIGH
    beq polygon_fill_triangle_row_done

polygon_fill_triangle_pixel_next_256:
    ldx #0
polygon_fill_triangle_pixel_next_256_0:
    sty VERA_DATA1
    dex
    bne polygon_fill_triangle_pixel_next_256_0
    dec FILL_LENGTH_HIGH
    bne polygon_fill_triangle_pixel_next_256

polygon_fill_triangle_row_done:

    ; We always increment ADDR0
    lda VERA_DATA0   ; this will increment ADDR0 with 320 bytes
                     ; So +1 vertically

    ; We check if we have reached the end, and if so we stop
    dec NUMBER_OF_ROWS
    bne polygon_fill_triangle_row_next
```

Below is a screenshot of what the above code will produce:

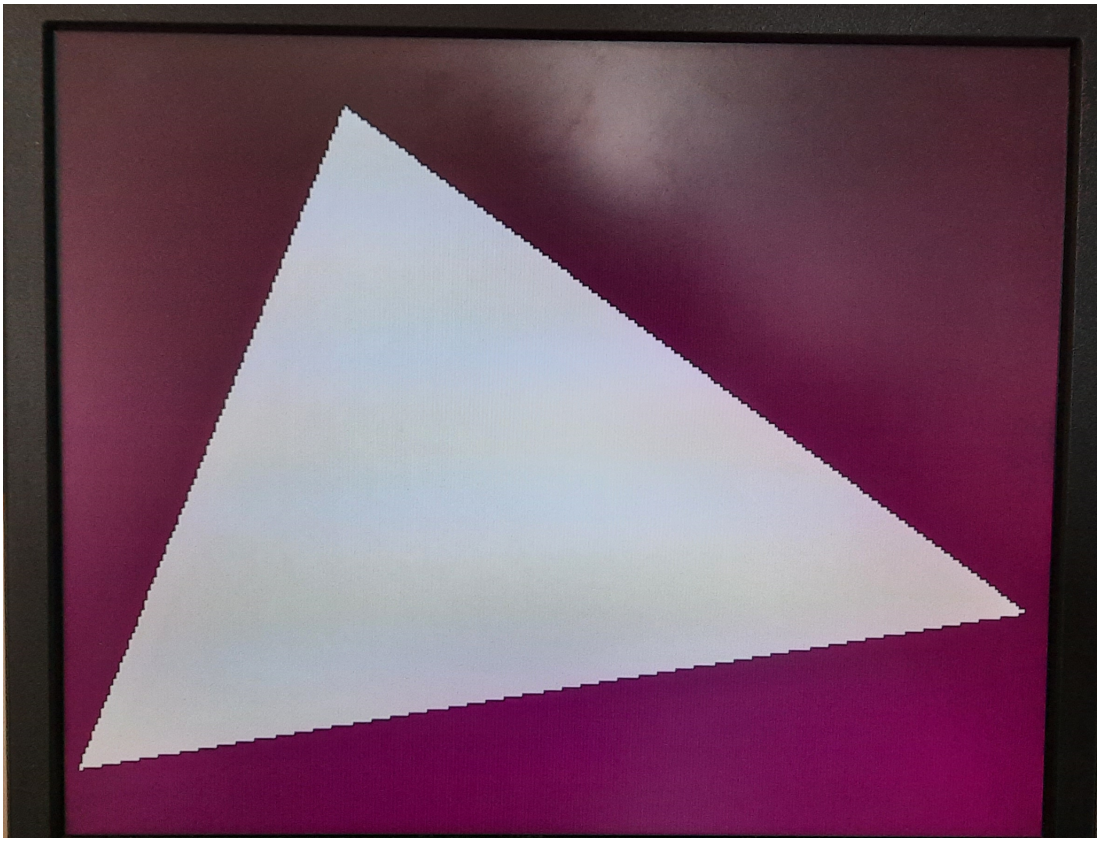


Bottom part

To draw the bottom part, you set up the increment of (in this case X2) and draw it this way:

```
lda #%00000110           ; DCSEL=3, ADDRSEL=0
sta VERA_CTRL

; NOTE that these increments are *HALF* steps!!
lda #<(-1590)             ; X2 increment low
sta $9F2B
lda #>(-1590)             ; X2 increment high
and #%01111111            ; increment is only 15-bits long
sta $9F2C

lda #50
sta NUMBER_OF_ROWS
jsr draw_polygon_part_using_polygon_filler
```

Below is a screenshot of what all of the code above will produce:



Note that in the above code examples there is no check on negative fill length (bit 8 and 9 being both 1). This should be added when it is not certain that X1 can become larger than X2.

4-bit polygon filling

Filling a polygon in 4-bit mode essentially works the same as in 8-bit, with a few changes.

- Assuming the bitmap you write to is also in 4-bit you need to set the increment of ADDR0 to +160 instead of +320 (when dealing with a 320 pixel screen).
- The increment of ADDR1 (when not using cache writes) should be +1 nibble (+0.5 bytes). See the section about “4-bit mode” on how to set this.

- When ADDR1 is calculated by VERA (by reading from DATA1) the calculation takes into account the size of the pixels. So instead of ADDR1 = ADDR0 + X1, it now becomes ADDR1 = ADDR0 + X1 / 2
- The fill length (low) register will have a slightly different layout, as shown here:

$9F2B DCSEL=5	fill_len >= 8	X pixel pos (1:0)	X pixel pos (2)	fill_len (2:0)	0

  Note that *two* bits have changed compared to the 8-bit version:

  - There is one less bit available of fill_len: only 3 bits are present. The fill_len[3] bit is replaced by X pixel pos[2].
    - Background info: since 32-bits contain 8 nibbles, more precision is needed for determining where in the 32-bit aligned address the first pixel starts.
  - The highest bit will contain a 1 if "fill_len >= 8" (instead of 16). This is because one less bit of fill_len is available, so the need for reading the fill_len (high) register ($9F2C, DCSEL=5) is changed.

TODO: give an example of 4-bit polygon filling?

2-bit polygon filling

Even though the hardware of VERA is not well suited to do 2-bit writes to VRAM it is possible to draw 2-bit polygons, essentially using a workaround. An explanation of this technique/workaround is given here.

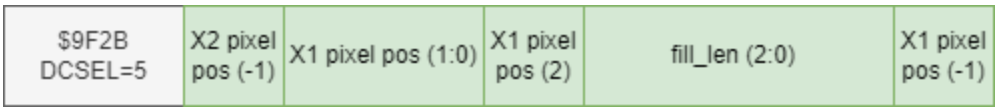
It should first be noted that this technique actually uses 4-bit filling, by simply **halving** the slopes (the X1 and X2 increment values). But in order for the first and last pixel of each fill line to be correct, they may have to be "2-bit patched". In essence: when the starting pixel (of a horizontal fill line) is at an *odd* pixel you skip the first 4-bit draw but instead do a special "2-bit poke" into VRAM at that pixel location. You do the same at the end of the fill line, but only if the end-pixel is at an *even* pixel.

Below is the location of the “2-bit polygon pixels”-bit:

It should be noted that this bit is only active when polygon mode is active *and* when 4-bit mode is 1 and the bit itself is 1. Otherwise it has no effect.

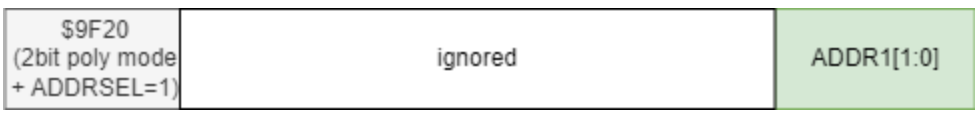
These are the things you would differently (from 4-bit polygon filling):

- Assuming the bitmap you write to is also in 2-bit you need to set the increment of ADDR0 to +80 instead of +160 (when dealing with a 320 pixel screen).
- The fill length (low) register will have a slightly different layout, as shown here:



  Note that another *two* bits have changed compared to the 4-bit version:

  - The highest bit will not contain “fill_len >= 8 or 16” anymore. Instead it will contain whether X2 is at an odd or even pixel location. In essence the highest bit the X2 *sub* pixel location will be given.
  - The lowest bit will not contain a 0 anymore. Instead it will contain whether X1 is at an odd or even pixel location. In essence the highest bit the X1 *sub* pixel location will be given.
- After reading from DATA1 (which sets ADDR1 to ADDR0 + X1 / 2 , like in 4-bit mode) you would not start drawing yet. Instead you determine (based on “X1 pixel pos [-1]” whether the first pixel should be “2-bit poked” or not.
- When a “2-bit poke” is needed you perform the following action:
  - You write the 2 lowest bits of the *byte*-position of the pixel into the ADDR_LOW (assuming ADDRSEL=1 here). All other bits are ignored by VERA. Basically you set the 2 lower bits of ADDR1 to the correct value.



This sets the 2 lowest bits of ADDR1 (not counting the nibble bit as the lowest bit here). Note that this will automatically make sure that DATA1 will be filled with the byte from VRAM corresponding to the new ADDR1 address.

Important: this puts VERA in a special “2-bit poke mode”, which will change its behavior when writing to DATA1, but only *once*.

- You write 2 lowest bits of the *pixel*-position to DATA1, like so:

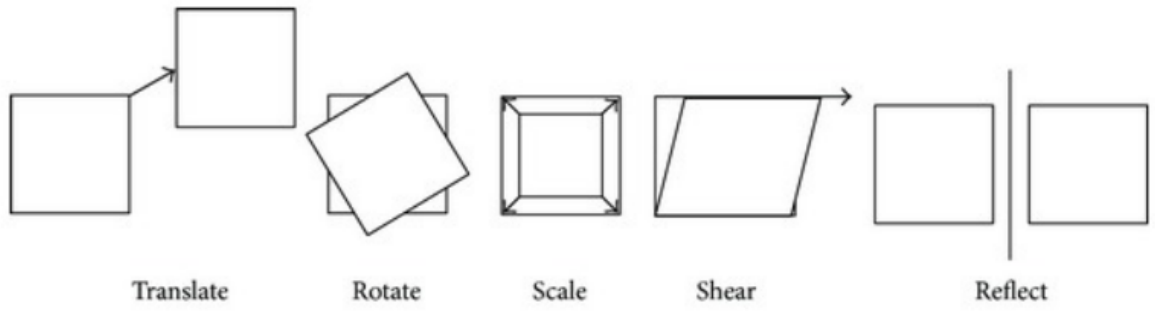
ADDR1[-1] here means the nibble bit of the pixel. ADDR1[-2] here means whether it's the odd or even 2-bit pixel within that nibble. VERA is in “2-bit poke mode” and will temporarily (only during this write to DATA1) turn off all increments, transparency, cache writes or 4-bit mode settings. It will instead merge the DATA1 value it has just loaded (which contains the 8-bits of VRAM at address ADDR1) with 8-bits from the 32-bit cache (using the cache byte index). This “merge” is in such a way that 2-bits from the 8-bits of cache are placed over the 8-bits from VRAM at the location determined by the value just written to DATA1. This will be written to ADDR1 as one byte.

- After doing this procedure you need to write the fill length as nibbles (you are in 4-bit mode). But in case you just “2-bit poked” you need to skip a nibble. This is very awkward to do in this “2-bit polygon pixel”-mode. It is however highly recommended to do this with cache write turned, since you can then write the correct mask and skip the appropriate nibble that way.
- When reaching the end of the fill line you should determine whether you should 2-bit poke at the end of the line (by using the “X2 pixel pos [-1]” bit value. And if needed do the same 2-bit poking as was done with the start of the fill line. But in this case it should be an even pixel.

Affine helper

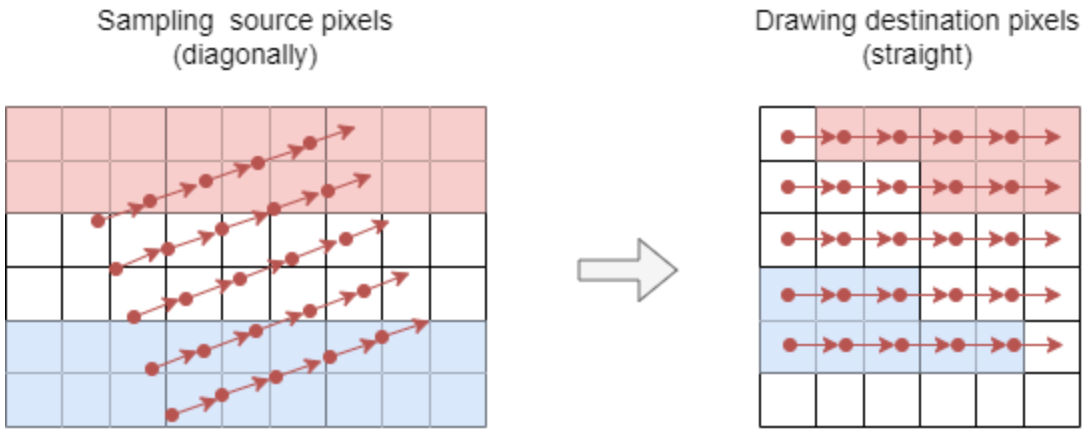
When addr1 mode is set to 3 (=11b) the **affine (transformation) helper** is enabled. In order to do affine transformations on an (bitmap)image several other registers have to be set correctly. Before explaining those a bit of information about the affine transformation is needed.

The illustration below (taken from here) shows most affine transformations that can be performed on an original 2D image:



Often these transformations are described using a mathematical function or matrix. While this is interesting it does not tell you (much) on how to implement it into code and pixels.

One the approach to realize these affine transformations in practice is to **sample** the source pixel rows **diagonally** and **draw** the pixels in a **straight** way:



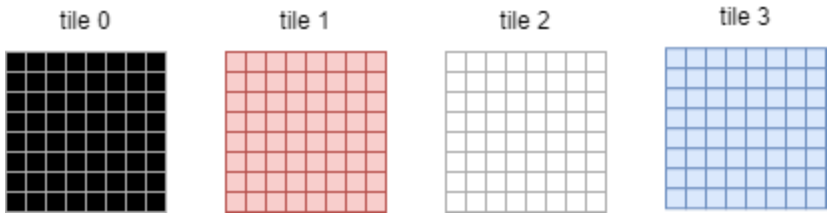
As can be seen in the illustration this simple method can cause a “rotational effect”.

This (rather simple) method is used by the affine helper: it can keep track of an x/y position in a source image[7], read from it pixel by pixel (diagonally) after which you can draw it straight as pixels into a sprite or bitmap. This way all of the above affine transformations can be achieved.

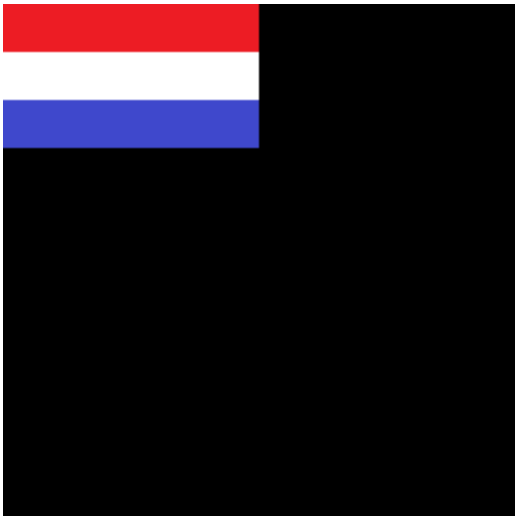
In the following sections an example of shearing a source image is provided. The source image is very simple (just some horizontal bars) but any source image can be used.

Setting up the source image

We set up a *very* simple source image first. We have to put it into 8x8 pixel tiles. We use these 4 different 8x8 pixel tiles:



We create a tile map of 32x32 tiles so that our total image/map looks looks like this[8]:



So the map is effectively a 256x256 pixel image. Black will be used as the transparent color.

[7] To be more precise: the position is a x/y coordinate on a *tile map*. So a source image is not directly accessed. Basically a source image needs to consist of 8x8 pixel tiles and these tiles should be placed in a tile map.
[8] Can you guess the nationality of the author? ;)

Example code

In order to create the tiles and the tile map we have to do a little bit of work on the cpu side. This can be achieved in many ways, but here we choose a very simple programmatic approach. First the tiles themselves:

```
    ; -- Setting up the VRAM address for the tile data --
    lda #%00010000          ; increment 1 byte
    ora #VRAM_ADDR_TILE_DATA >> 16
    sta VERA_ADDR_BANK
    lda #>VRAM_ADDR_TILE_DATA
    sta VERA_ADDR_HIGH
    lda #<VRAM_ADDR_TILE_DATA
    sta VERA_ADDR_LOW

    lda #0          ; tile 0 is black
    ldx #64
next_black_pixel:
    sta VERA_DATA0
    dex
    bne next_black_pixel

    lda #2          ; tile 1 is red
    ldx #64
next_red_pixel:
    sta VERA_DATA0
    dex
    bne next_red_pixel

    lda #1          ; tile 2 is white
    ldx #64
next_white_pixel:
    sta VERA_DATA0
    dex
    bne next_white_pixel

    lda #6          ; tile 3 is blue
    ldx #64
next_blue_pixel:
    sta VERA_DATA0
    dex
    bne next_blue_pixel
```

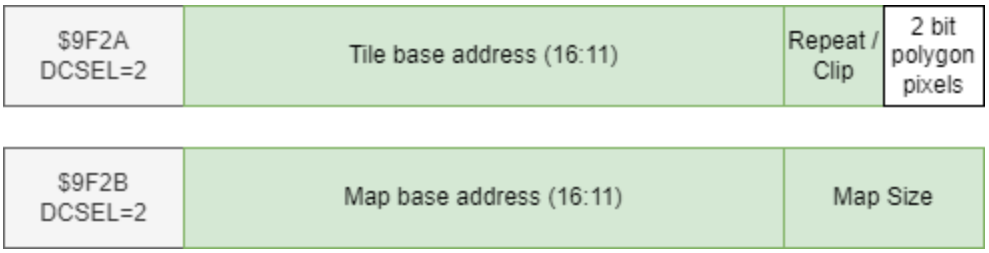
Then setting up the tile map. Here is the map data (in code):

```
tile_map_data:

    .byte 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

And here we load the map data into VRAM:

```
    ; -- Setting up the VRAM address for the map data --
    lda #%00010000          ; increment 1 byte
    ora #(VRAM_ADDR_MAP_DATA >> 16)
    sta VERA_ADDR_BANK
    lda #>VRAM_ADDR_MAP_DATA
    sta VERA_ADDR_HIGH
    lda #<VRAM_ADDR_MAP_DATA
    sta VERA_ADDR_LOW

    ; -- Load tile indexes into VRAM (32x32 map) --

    lda #<tile_map_data
    sta LOAD_ADDRESS
    lda #>tile_map_data
    sta LOAD_ADDRESS+1

    ldx #4
next_eight_rows:
    ldy #0
next_tile_index:
    lda (LOAD_ADDRESS), y
    sta VERA_DATA0
    iny
    bne next_tile_index
    inc LOAD_ADDRESS+1      ; we increment the address by 256
    dex
    bne next_eight_rows
```

After this we have to tell the affine helper where the tile data is, where the map data is, how big the map is, and whether we want to clip or repeat the map. There are the two registers that we have to be set:



Note that both addresses are limited to 6 bits, so they have to be chosen accordingly.

And here it is in code to do it:

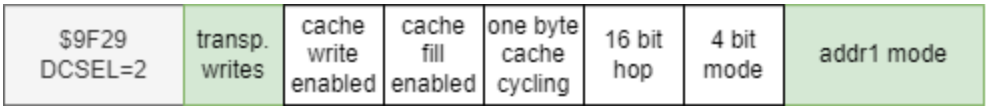
```
    ; -- Set up tile data and map data addresses, map size and
clipping

    lda #%00000100           ; DCSEL=2, ADDRSEL=0
    sta VERA_CTRL

    lda #(VRAM_ADDR_TILE_DATA >> 9)
    and #%11111100   ; the 6 highest bits of the tile address are set
    ora #%00000010   ; clip = 1
    sta $9F2A

    lda #(VRAM_ADDR_MAP_DATA >> 9)
    and #%11111100   ; the 6 highest bits of the map address are set
    ora #%00000010   ; Map size = 32x32 tiles
    sta $9F2B
```

Now the source image data has been set up and we can start setting up the affine helper for affine transformation of the source data.

Setting up and using the affine helper

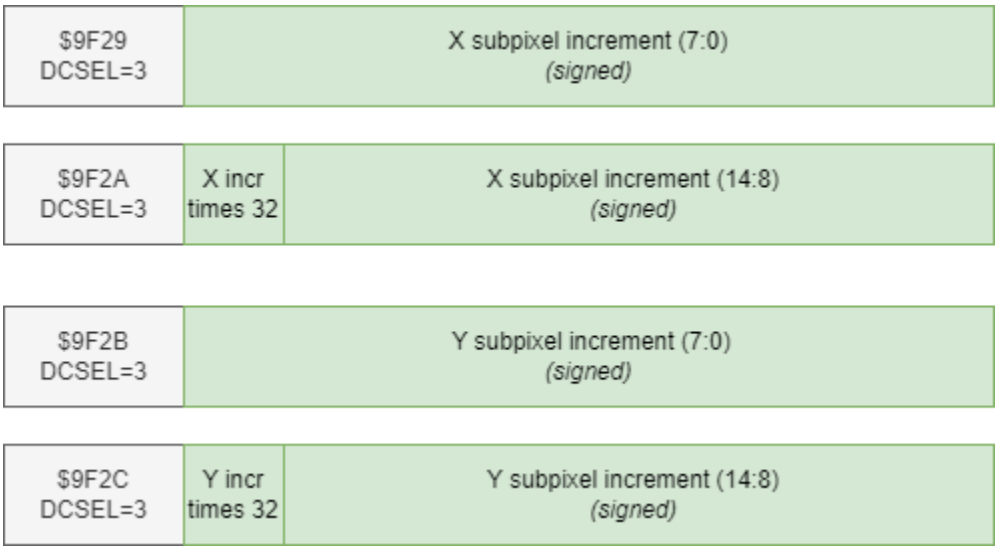
In order to use the affine helper, there are a few steps that have to be taken to set it up (assuming 320 pixel wide screen here):

- Turning on affine helper and (in this case) turning on transparent writes. When addr1 mode is set to 3 (=11b) the affine helper is enabled. The highest bit in $9F29 controls whether transparent writes are enabled.



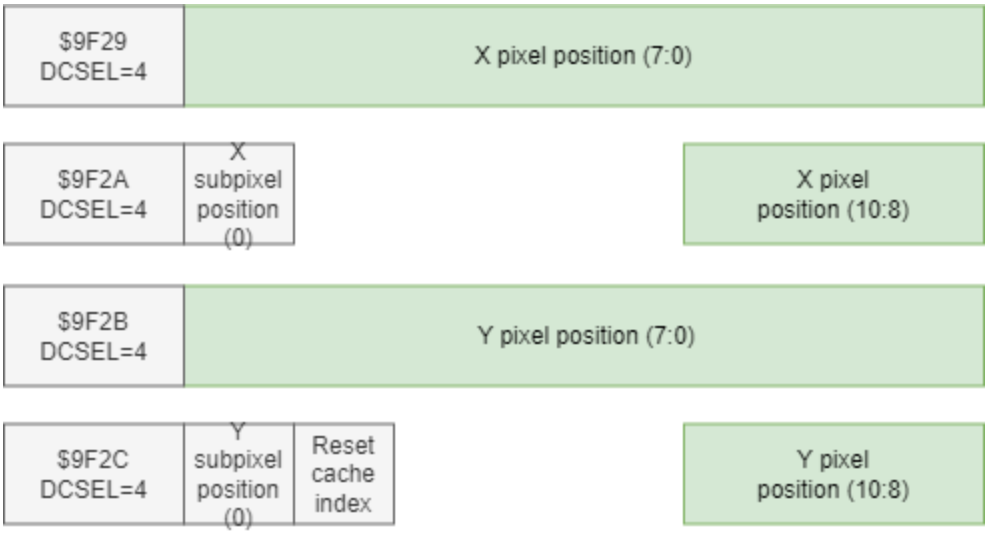
  Transparent writes means that a 0 written to DATA0 or DATA1 does not change VRAM (but it does increment the address). Since our tile 0 contains only zeros it will act as a transparent tile. Also: when the edge of the map is reached (and clipping is enabled) the pixels of tile 0 will be returned when reading from DATA1. So this will effectively make the edges of the map transparent as well. This ensures that nothing but our non-transparent tiles get used as our source and we do not overwrite unwanted parts of the screen.

- After each pixel read from the source the X and Y position in that source is incremented by the X and Y increment values. So set the "**X** subpixel increment" and the "**Y** subpixel increment" registers (DCSEL=3, see below) to the values you want the affine helper to step in x and y direction for each pixel you read. Here are some guidelines:
    - If the step is larger, you will scale down the resulting image (and vice versa)
    - If you set x or y to a negative value, the image will be reversed/flipped on that axis.
    - If you set your x and y to the same value, you will sample your source image at 45 degrees.
    - By making x-increment different from your y-increment you can change the angle at which you sample your source image.
- Note that increment registers are 15-bit signed fixed-point number:
    - 6 bits for the pixel increment
    - 9 bits for the subpixel increment
    - 1 additional bit that indicates the actual value should be multiplied by 32



- Set your starting ADDR0 (to where you want to start drawing on the screen/sprite)

- For each row you draw you do the following:
    - Set your ADDR0 to the starting pixel of the row you want to draw. For shearing and scaling (so: non-rotation) this is always at the same pixel column on your screen
    - Set your x and y position of this row in your source image:



      For the type of shearing we are doing here this position is always on the same pixel column of your source image. For other types of affine transforms more work/calculations on the CPU are needed. More on that later.

    - Iterate for each pixel (as many as your destination row is)
        - Read from DATA1: this will read the value at the current x/y position of the source image. It will also increment the x and y value afterwards.
        - Write to DATA0: this will write the value to the pixel on the screen/sprite.

Example code

Here is example code to set up and use the affine helper to do a *shearing* transform.

First the set up:

```
    lda #%10000011  ; transparent writes = 1, affine helper mode
    sta $9F29

    ; -- Set up x and y increments --

    lda #%00000110    ; DCSEL=3, ADDRSEL=0
    sta VERA_CTRL

    lda #0
    sta $9F29
    lda #%00000010    ; X increment = 1.0 pixel to the right each
step
    sta $9F2A
    lda #<(-40<<1)
    sta $9F2B
    lda #>(-40<<1)    ; Y increment = -40/256th of a pixel each step
    and #%01111111    ; increment is only 15 bits long
    sta $9F2C

    ; We start to draw at the top-left position on screen
    stz VRAM_ADDR_DESTINATION
    stz VRAM_ADDR_DESTINATION+1
    stz VRAM_ADDR_DESTINATION+2
```

Then drawing of each row:

```
    ldx #0

draw_next_row:


    lda #%00000110           ; DCSEL=3, ADDRSEL=0
    sta VERA_CTRL

    lda #%00010000           ; ADDR0 increment is +1 byte
    ora VRAM_ADDR_DESTINATION+2
    sta VERA_ADDR_BANK
    lda VRAM_ADDR_DESTINATION+1
    sta VERA_ADDR_HIGH
    lda VRAM_ADDR_DESTINATION
    sta VERA_ADDR_LOW
```

```
    ; Setting the source x/y position

    lda #%00001001          ; DCSEL=4, ADDRSEL=1
    sta VERA_CTRL

    lda #0                  ; X pixel position low [7:0] = 0
    sta $9F29
    lda #0                  ; X pixel position high [10:8] = 0
    sta $9F2A

    txa                     ; Lazy and simple: we use register x (=
destination y) as our y-position in the source
    sta $9F2B
    lda #%00000000          ; Y pixel position high [10:8] = 0
    sta $9F2C

    ldy #0
draw_next_pixel:
    lda VERA_DATA1
    sta VERA_DATA0

    iny
    bne draw_next_pixel
```

At the end of each row:

```
    ; We increment our destination address with +320
    clc
    lda VRAM_ADDR_DESTINATION
    adc #<320
    sta VRAM_ADDR_DESTINATION
    lda VRAM_ADDR_DESTINATION+1
    adc #>320
    sta VRAM_ADDR_DESTINATION+1
    lda VRAM_ADDR_DESTINATION+2
    adc #0
    sta VRAM_ADDR_DESTINATION+2

    inx
    cpx #180                ; we do 180 rows
    bne draw_next_row
```

Below is a screenshot of what all of the code above will produce:

4-bit affine helper

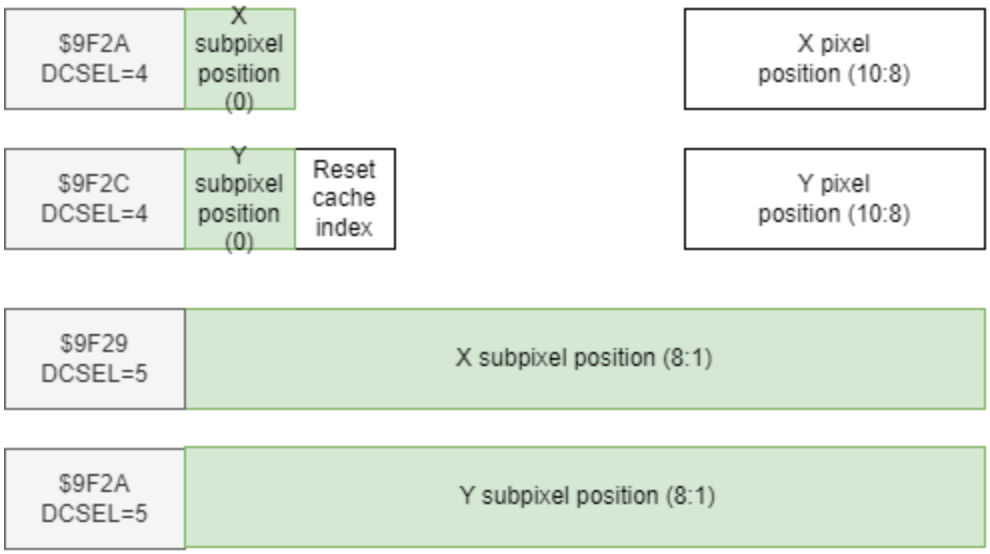
When using the affine helper in 4-bit mode things work the same as in 8-bit, with a few changes.

- You probably want to set the target bitmap (you write to) to 4-bit color depth
- The increment of ADDR0 (when not using cache writes) should be +1 nibble (+0.5 bytes). See the section about “4-bit mode” on how to set this.
- When ADDR1 is (re-)calculated by VERA (by reading from DATA1) the calculation takes into account the size of the pixels. Tiles are 8x8 pixels, but only take 32 bytes in memory in 4-bit mode. Nothing has to be done for this, except to make sure your tiles actually contain 4-bit pixels and are therefore half the size of 8-bit pixel tiles.

TODO: explain more about scaling and rotation.

TODO: MAYBE USE THIS FOR ROTATION? X16 logo (32x32 pixels)

TODO: explain when you want/need to set the subpixel positions directly:



4-bit mode

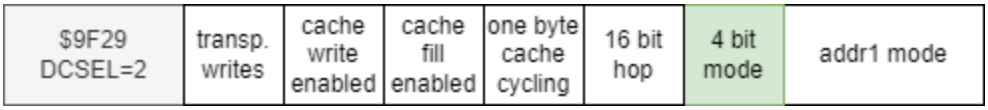
The FX update allows for a 4-bit addressing mode. This means you can set both ADDR0 and ADDR1 in a nibble level and set an increment of +0.5 bytes (+1 nibble) or -0.5 bytes (-1 nibble). When writing to DATA0 or DATA1 in 4-bit mode only one nibble of the data that is sent to VERA is written to VRAM, namely the nibble that the address was set to. Reading from DATA0 or DATA1 is not changed: the full byte is retrieved from VERA.

Both the address nibble bit (the 18th bit of the address if you will) and the bit that turns on the nibble incrementer are located in the $9F22 register:

Keep in mind that the $9F22 is dependent on ADDRSEL (part of the CTRL register of VERA). So this works the same way as the other bits in the $9F22 register. Note that these two bits were unused before the FX update.

While these two bits can be set (and read) without 4-bit mode being turned on, they are only *active* when 4-bit is turned on.

To turn on 4-bit mode the following bit has to be set to 1:



Important to note is that the nibble incrementer/decrementer only works if the byte increment is set to 0. So if you want to (for example) turn on nibble incrementing for ADDR0 you set its “Address increment” to 0000b and its “Address nibble increment” to 1b. If you want to decrement by one nibble you also set the “DECR” to 1b.

4-bit behavior per addr1-mode or cache filling

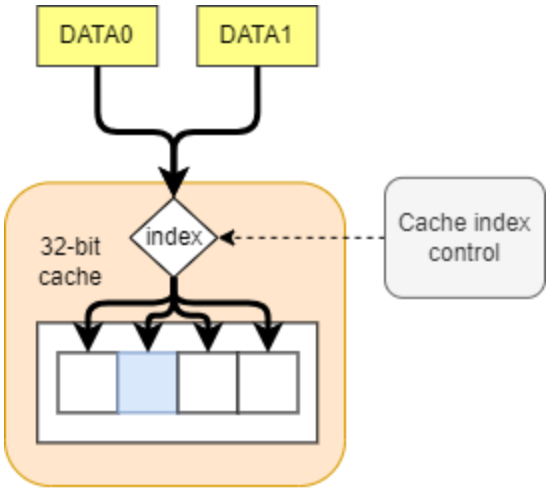
Some addr1-modes (and the cache filling) show a slightly different behavior when 4-bit mode is turned on. This is mentioned in their corresponding sections.

32-bit cache

The 32-bit cache is -in a way- is the heart of FX update: it can be used to speed up operations quite a bit. Below is an explanation of how it works and how/when to use it.

Filling the cache

Whenever the CPU reads from registers $9F23 or $9F24 (aka DATA0 or DATA1) and cache filling is enabled, a byte (or nibble, in the case 4-bit mode is enabled) is copied from the DATA0 or DATA1 register into the 32-bit cache. The position where the data is put into the cache is determined by the cache index. This is depicted in the diagram below:



The blue square is an example destination of the data. By default the cache byte (or nibble) index is incremented each time a read from DATA0 or DATA1 happens, allowing for convenient filling of the cache. It loops around when it reaches its max index. Keep in mind that when in 4-bit mode there are not four possible destinations for the data to be filled, but eight.

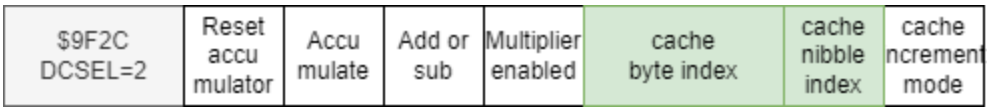
Below is the control bit to turn on cache filling:

$9F29 DCSEL=2	transp. writes	blit write enabled	cache fill enabled	one byte cache cycling	16 bit hop	4 bit mode	addr1 mode

Cache index control

You can also (re)set the cache byte index (or nibble index) directly by setting it to a specific value: 0-3 in 8-bit mode, 0-7 in 4-bit mode. In 8-bit mode a 0 means the byte “32-bit cache [7:0]” is selected, while 3 means the byte “32-bit cache [31:24]” is selected. In 4-bit mode a 0 means the nibble “32-bit cache [3:0]” is selected, while 3 means the nibble “32-bit cache [31:28]” is selected.

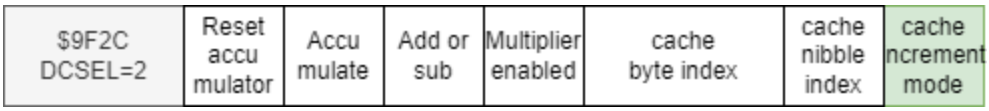
Here are the bits that can be used to set the cache byte/nibble index:



Cache index incrementing

Normally the cache byte index counts from 0 to 3 and then goes back to 0 again. When the alternative cache increment mode is turned on, it counts either from 0 to 1 (and back to 0 again) or from 2 to 3 (and back to 2 again). Which of the two is the case depends on the current cache byte index: when it's 0 or 1 its 0 → 1 (and back to 0), when it's 2 or 3 its 2 → 3 (and back to 2). This incrementing mode is only available in 8-bit mode (not in 4-bit mode). It is most useful in combination with 16-bit hop mode (more on that mode in the section about the multiplier and accumulator).

Below is the control bit to turn on the alternative cache increment mode:



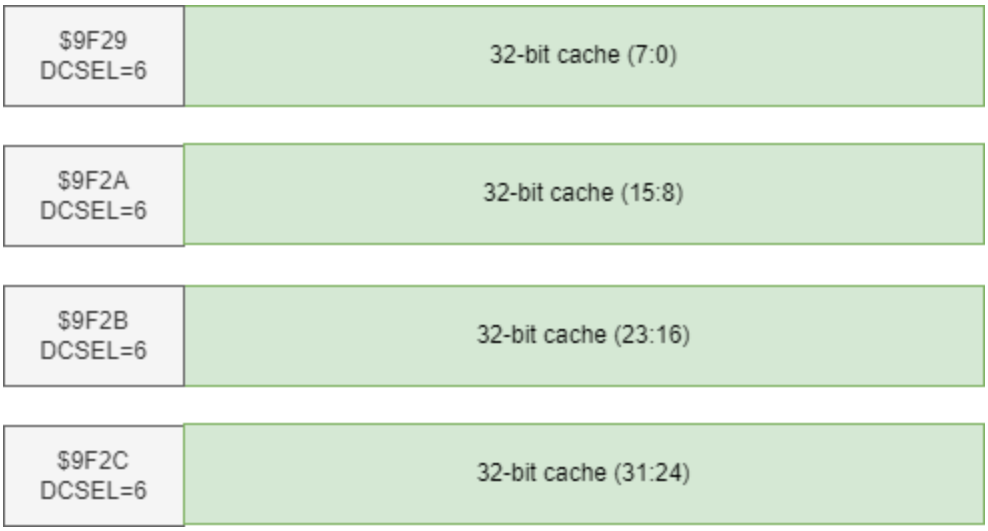
Cache index reset and triggering

Usually the incrementing of the cache index is only triggered by reading from DATA0 or DATA1 when cache filling is enabled. However it can also be triggered by reading from DATA0 in polygon mode when cache filling is *not* enabled and “cache byte cycling” is enabled. This is useful for ordered dithering of polygons. More on that in the section about “cache byte cycling”.

Setting the cache data directly

Instead of filling the cache by reading from DATA0 or DATA1 the cache data can also be set directly.

Below are the registers that allow you to set the cache directly:



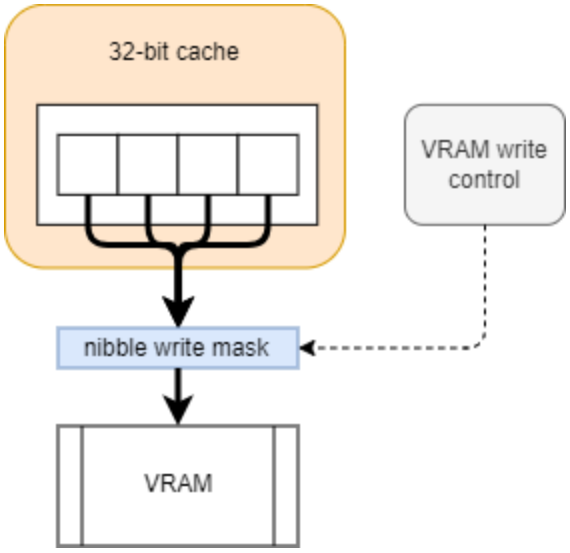
Setting the cache data directly does not affect the cache byte/nibble index.

Writing the cache to VRAM

This entire (4 byte) cache can be written to VRAM at once, but this write will always be a memory address that is *aligned to 32-bits*. Because of this, tight control of the cache index and control over what bytes (or nibbles) are written to VRAM is needed and made possible.

The cache gets written to VRAM whenever there is a write to DATA0 or DATA1 and cache writes are enabled.

Below is a depiction of a cache write to VRAM[9]:



Below is the control bit to turn on cache writing:

$9F29 DCSEL=2	transp. writes	cache write enabled	cache fill enabled	one byte cache cycling	16 bit hop	4 bit mode	addr1 mode

Write masks

When writing the cache to VRAM the value you write to DATA0 or DATA1 gets (by default) interpreted as a **nibble mask**: each of the 8 bits you write represents a nibble in the 32-bit cache. If a bit is 0 the corresponding nibble (4 bits) gets written to VRAM. If a bit is 1 the corresponding nibble is not written to VRAM.

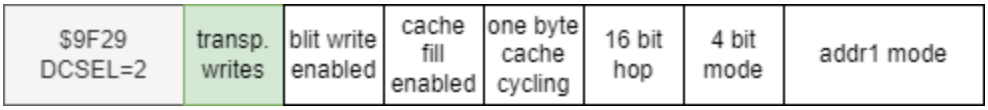
Transparency writes

It is also possible to write with transparency. This means that when a byte (or a nibble in case of 4-bit mode) is 0 in the cache, the byte (or nibble) is masked and does not get written

[9] Technically speaking the output of the 32-cache is also connected to the multiplier and accumulator. More on that in the next section.

to VRAM. This is useful when you (quickly) want to copy something with transparency over something else without disturbing it.

Here is the control bit to turn on transparent writes:



Note that transparent writes also work without the cache: if cache writes is disabled and transparent writes are enabled, the value you write to DATA0 or DATA1 is written to VRAM unless its a 0 (the 0 will not be written to VRAM).

Cache byte cycling

When cache byte cycling is turned on the cache-byte that the “cache byte index” points to is used to write to VRAM. When cache writes are turned on as well the byte is duplicated 4 times when writing to VRAM.

When in polygon mode reading from ADDR0 triggers an increment in the cache byte index. Reading from ADDR0 is done *exactly once* for each fill line. This effectively means that each line gets a different color each time.

Since there are 4 bytes in the cache this will repeat every 4 fill lines. In 4-bit mode this means that each line can have 4 vertically alternating colors and 2 horizontally alternating colors (there are 2 pixels in a byte for 4bpp): a 2x4 *dithering pattern* if you will. For 2-bit colors this becomes a 4x4 dithering pattern.

Common usage

One common pattern for the use of the cache is to turn on **cache filling** and then read 4 bytes from VRAM: by reading from (for example) DATA0. By doing so the cache gets filled with those 4 bytes. Then you **write** the whole 4-byte **cache** at once to VRAM by simply writing to (for example) DATA1. Usually you set your increment of the latter address to +4 so you can keep going.

This is useful for:

- copying bytes or nibbles from VRAM to VRAM, using the normal addr1 mode
- copying bytes or nibbles from VRAM to VRAM, using the affine helper

Another pattern is to fill the cache once beforehand, turn off cache filling and write it several/many times to VRAM.

This is useful for:

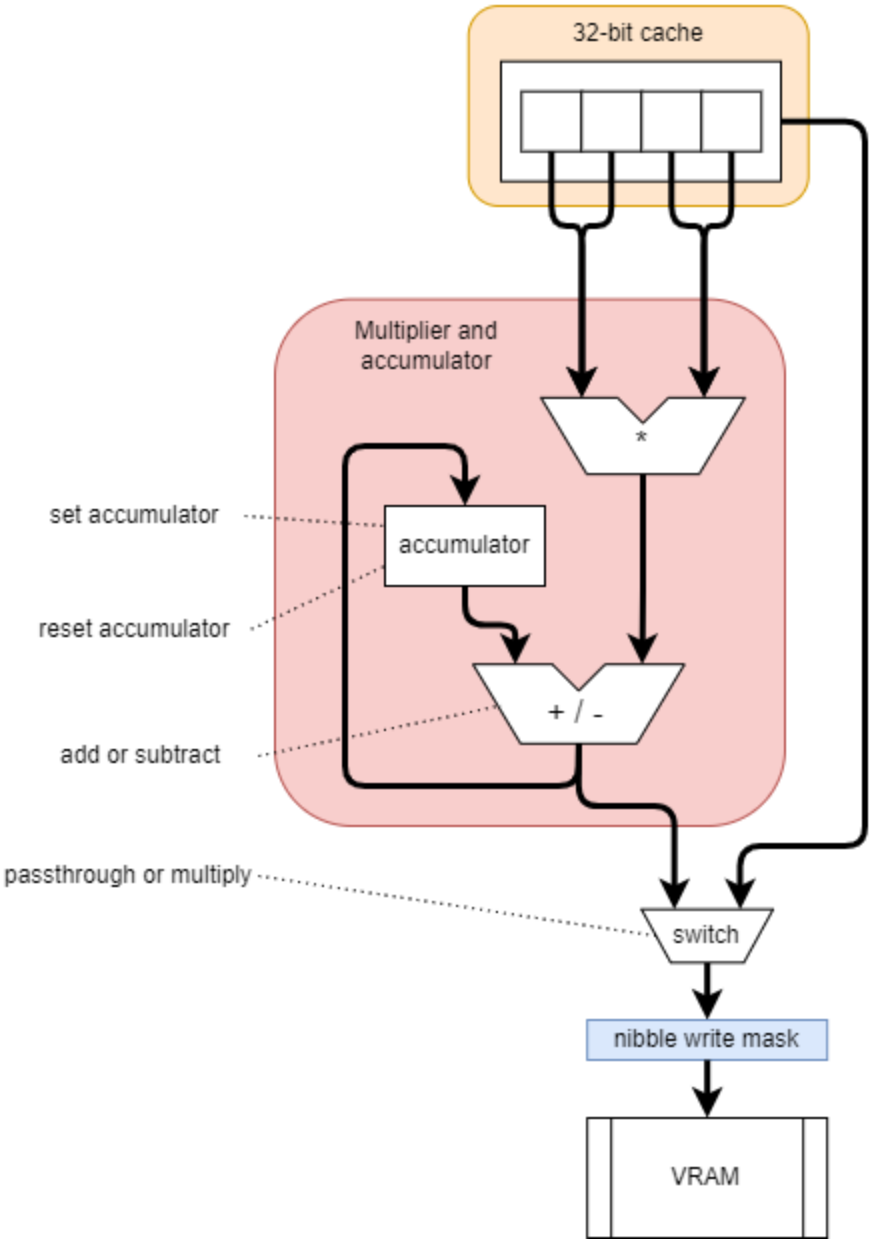
- Filling (large parts of) the screen with a color (aka “clear screen”).
- Filling a horizontal line of a polygon. Masking is needed for the starting and ending parts of a horizontal line.
    - When combined with cache byte cycling a dithering pattern can be applied

TODO: add some example code of cache usage!

Multiplier and accumulator

It should be noted that the cache output is (also) connected to the input of the multiplier and accumulator. By controlling the index of the 32-bit cache carefully and a special address incrementer bulk math operations can be performed.

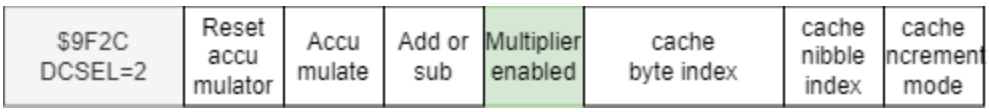
Below is a block diagram showing how the 32-bit cache is connected to the multiplier and accumulator:



How to do math

Some explanation is needed to clarify how this works in practice. When the “Multiplier enabled” bit is 0 the output of the 32-bit cache is connected to nibble write mask/VRAM. This way it works as a “pass through” and everything works as mentioned in the previous sections.

When “Multiplier enabled” bit is set to 1 the output of the multiplier / adder / accumulator is connected to the nibble write mask/VRAM. Below shows this bit:



Since the (32-bit) accumulator is by default 0, the output towards VRAM is going to be the 16x16 signed multiplication of the two 16-bit numbers in the 32-bit cache.

Single multiplication example

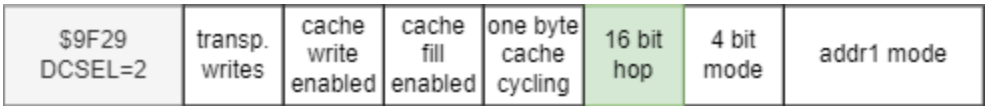
Here is an example how you could use this mode to do single multiplications:

- Set to addr1-mode to normal, turn on cache writes
- Set the increment of ADDR0 to +1
- Set the increment of ADDR1 to +4
- Set both ADDR0 and ADDR1 to any (but the same) VRAM address that is free
- Set the cache data (2x16 bits) directly using DCSEL=6 and $9F29,$9F2A and $9F2B,$9F2C (see section above). This is the input of your multiplication
- Write to DATA1: this will write the 4 byte multiplication result to VRAM
- Read 4 times from DATA0 giving you the 4 bytes of the result

Several multiplications example

It's quite likely you want to do multiple multiplications. It's also very likely you don't want all 4 bytes of the result but only 2 bytes (this is a very common pattern).

For this use case you can use the 16-bit hop mode which is located here:



What the 16-bit hop mode does is change the way ADDR1 is incrementing when set to +4 or +320. Instead of adding 4 (or 320) each increment it adds 1 and 3 (or 1 and 319) each increment. The 16-bit hop mode does *not* change anything to the way ADDR1 is incremented when DECR = 1.

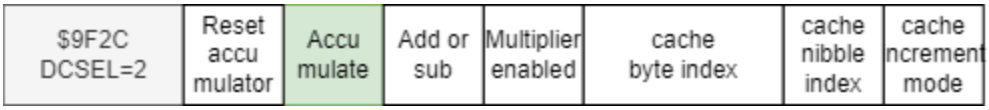
You can now do the following:

- Set to addr1-mode to normal and turn on 16-bit hop mode, turn on cache writes
- Set the increment of ADDR0 to +4
- Set the increment of ADDR1 to +4 (this will lead to +1, +3 increments)
- Set both ADDR0 and ADDR1 to any VRAM address that is free[10], **but** you set ADDR1 0,1 or 2 bytes higher than ADDR0, depending on which part of the 4 resulting bytes you are interested in.
- You keep doing the following as many multiplication as you need to do:
    - Set the cache data (2x16 bits) directly using DCSEL=6 and $9F29,$9F2A and $9F2B,$9F2C (see section above). This is the input of your multiplication.
    - Write to DATA1: this will write the 4 byte multiplication result to VRAM
    - Read **2** times from DATA0 giving you the 2 bytes of the result you want

[10] You need a range of free VRAM addresses: 4 * the number of multiplications you want to do in a row. The reason why 16-bit hop mode also works for +320 (not just +4) is that you can output a *column* of math-results on the side of the screen (and hide it from the view).

Accumulation

By default the accumulator is 0. But -as can be seen in the diagram- the output of the adder (or subtractor) is fed back into the accumulator. When writing a 1 to the “accumulate”-bit the output of the adder/subtractor is put into the accumulator. Here is this bit:



Now the output towards VRAM will contain the **addition** (when “Add or sub” is set to 0) of the content of the accumulator and the result of the multiplication. If the input of the cache is now changed you effectively allow for calculation like this:

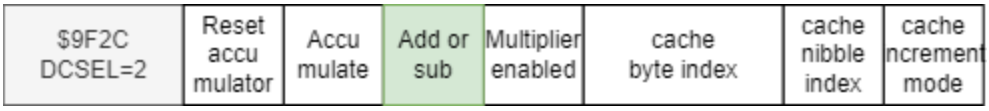
```
Result = A*B + C*D
```

Where A*B was your initial multiplication (A and B were set as 16-bit inputs) that was stored in the accumulator. After setting there C and D into the cache their multiplication is now outputted towards VRAM. A cache write will lead to this result being written to VRAM.

The “formula” is a very common pattern when doing linear algebra (“matrix multiplication”).

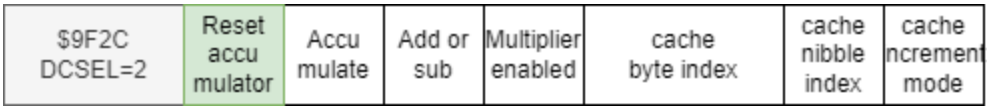
Adding or subtracting

When you want to subtract when doing accumulation you can do that by setting this bit:



Resetting the accumulator

When you want to restart you (accumulated) calculation you can reset the accumulator using this bit:



TODO: add some example code of the multiplier and accumulator usage!

Doing bulk chained math

It's quite possible you want to do bulk (chained) math. This is especially true when doing math for 3D polygons. Rotation, translation, perspective, culling, dot-products etc. There is a lot involved.

Often these calculations are in *bulk* and are *chained*: there are dozens (if not hundreds) of vertices and each of them require several sequential operations on their values. In such a case you want the input and output values to be readily available and quickly to be loaded into and extracted from the multiplier/accumulator.

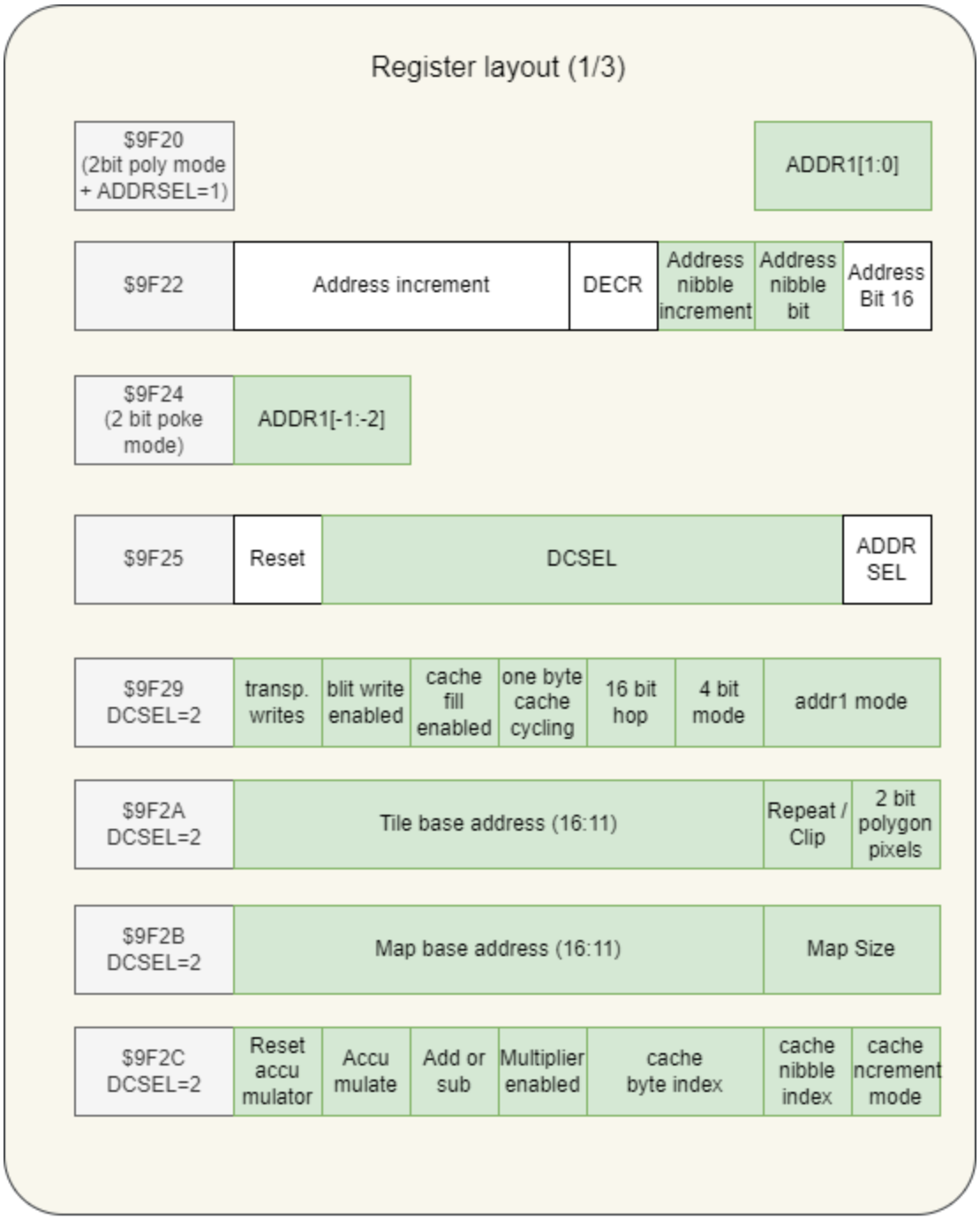
It's also possible/likely that some of these operations require one constant (16-bit) input value and one variable 16-bit input value. In such a case you probably want to fill the cache by reading from VRAM (using 16-bit hop mode) and write back to VRAM using the cache write function. You do that many times: one operation for all the vertices.

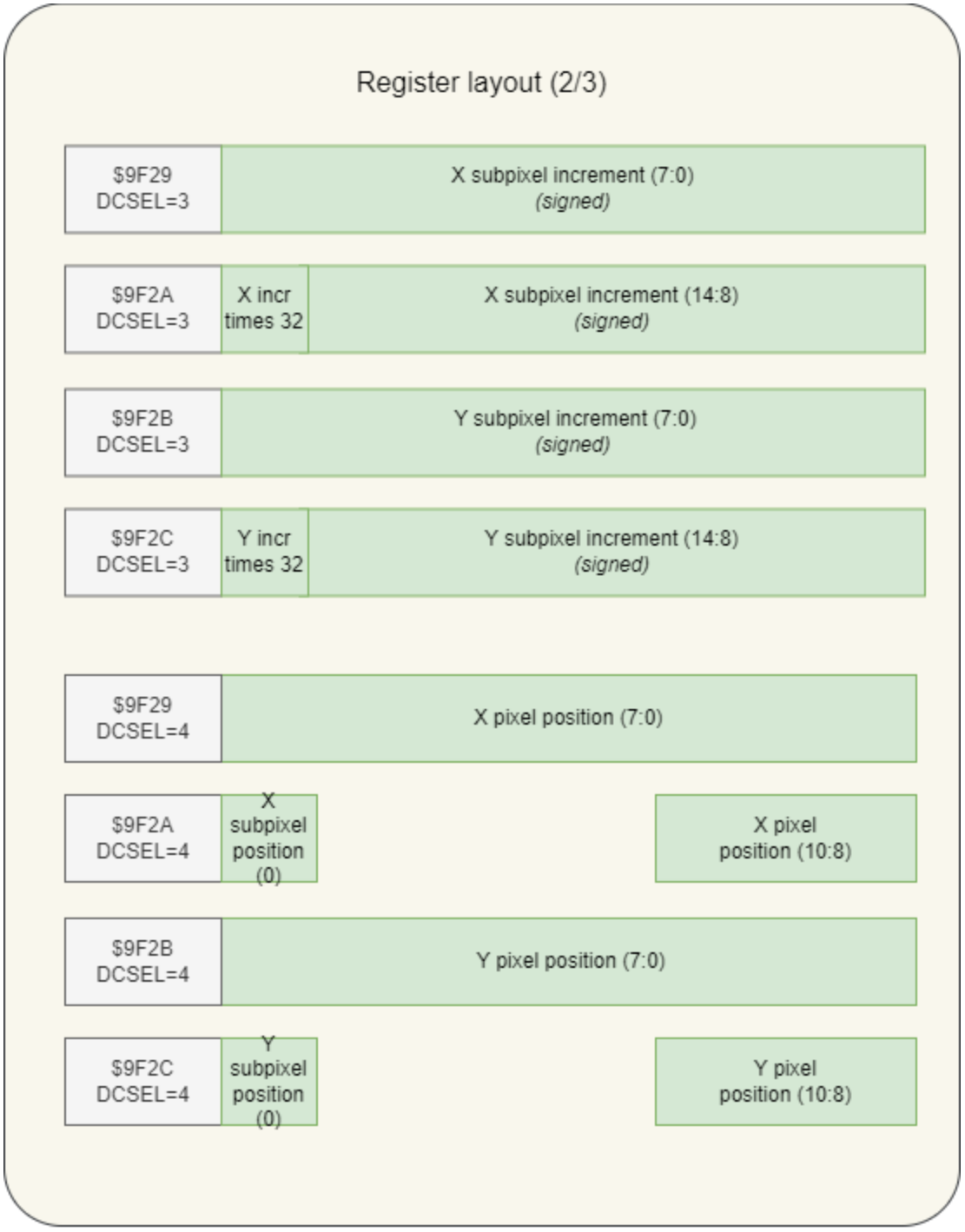
For example multiplying with a cosine of an angle (for rotating all vertices around the player). You would -in such a cache- set 16-bit of the cache to the cosine value and set you cache byte index to the start of the other 16-bit (let's say this is index 2). If you then set your “cache increment mode”-bit to 1, then the filling of the cache will only be done at index 2 and 3, which is exactly what you want. This can of course be combined with 16-bit hop mode so the data you read into slots 2 and 3 are 16-bits of the 32-bits that resulted from previous calculations.

It's not possible to enumerate all the possible ways in which this can be used, but suffice to say that this can be a very powerful tool for doing bulk chained math.

Appendices

Register map




Register layout (2/3)
$9F29 DCSEL=3
X subpixel increment (7:0) (signed)
$9F2A DCSEL=3
X incr times 32
X subpixel increment (14:8) (signed)
$9F2B DCSEL=3
Y subpixel increment (7:0) (signed)
$9F2C DCSEL=3
Y incr times 32
Y subpixel increment (14:8) (signed)
$9F29 DCSEL=4
X pixel position (7:0)
$9F2A DCSEL=4
X subpixel position (0)
X pixel position (10:8)
$9F2B DCSEL=4
Y pixel position (7:0)
$9F2C DCSEL=4
Y subpixel position (0)
Y pixel position (10:8)

Register layout (3/3)

$9F29 DCSEL=5	X subpixel position (8:1)				
$9F2A DCSEL=5	Y subpixel position (8:1)				
$9F2B DCSEL=5	fill_len >= 16	X pixel pos (1:0)	fill_len (3:0)		0
$9F2B DCSEL=5	fill_len >= 8	X pixel pos (1:0)	X pixel pos (2)	fill_len (2:0)	0
$9F2B DCSEL=5	X2 pixel pos (-1)	X1 pixel pos (1:0)	X1 pixel pos (2)	fill_len (2:0)	X1 pixel pos (-1)
$9F2C DCSEL=5	fill_len (9:3) (read-only)				0
$9F29 DCSEL=6	32-bit cache (7:0)				
$9F2A DCSEL=6	32-bit cache (15:8)				
$9F2B DCSEL=6	32-bit cache (23:16)				
$9F2C DCSEL=6	32-bit cache (31:24)				

Polygon filling details

TODO: this section goes deeper into some nuances of triangle drawing. Its probably best put into an appendix.

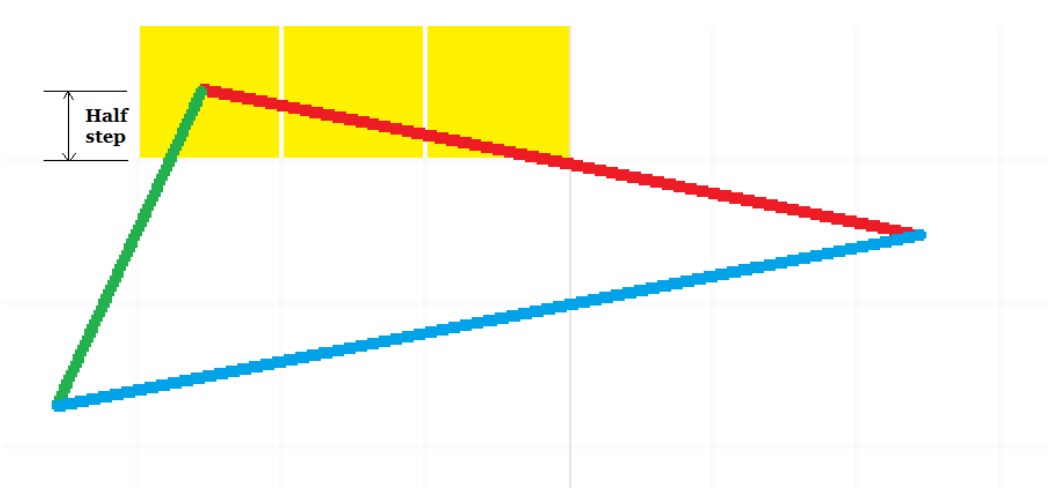
Jump table and 32-bit cache integration

TODO: The fill length returned by the polygon filler helper is crafted in such a way that it can be used for jump tables (a 65**C**02 feature). Probably a large section in this appendix can be dedicated to this subject: how it works in principe, how the 32-cache plays a role, how to generate these jump tables (and the code that belongs to it). And how much faster it is.

Starting and stopping with filling lines

TODO: this part is still incomplete/unedited/rough:

It is important to understand that *first line* and the *last line of a* triangle part should work a little different from the others as illustrated by this drawing (most notable the red line):

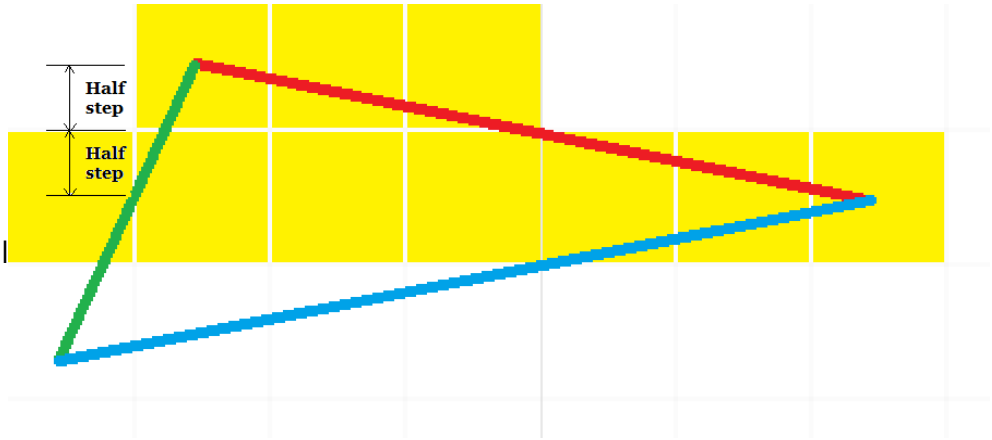


We start a half the pixel in y. At that point both X1 and X2 are the same so if at that moment the difference between the two was calculated (which is the length of the horizontal fill line) a 0 would come out of that. But that should clearly not be the case!

As can be seen: the slope of the red line is 5 pixels per line. Since the right point of the triangle is only 1 pixel lower than the top pixel, the first fill line should be around 3 pixels long (yellow squares). So to start first *half* of the increment should be added and *then* the polygon helper should be asked how long the fill line should be.

Now for the last line of a triangle part:

When the l



TODO: add a sections that explains what the effect is of:

- Filling a line with the given number of pixels (means: no overdraw, but requires the same same slope calculations to be re-used, to prevent “holes” between neighboring triangles)
- Filling with one extra pixel (1 pixel overdraw, less accuracy needed)