0.1 Smooth scrolling on EGA

Performing a full-screen redraw per frame would kill the CPU, as it would require updating every pixel on all four EGA planes. Achieving a 60Hz frame rate while refreshing the entire screen is impossible under these constraints. If we were to run the following code, which simply fills all memory banks, it would run at 5 frames per seconds.

```
# define SC_INDEX   0x3C4
# define SC_DATA    0x3C5
# define SC_MAPMASK 0x02

char far *EGA = (unsigned char far*)0xA0000000L;

void selectPlane (int plane) {
  outp ( SC_INDEX , SC_MAPMASK );
  outp ( SC_DATA , 1 << plane );
}

void WriteScreen(void){
  int i, bank_id;

  for (bank_id = 0 ; bank_id < 4 ; bank_id++) {
    selectPlane(bank_id);
    for (i = 0; i < 40 * 200; i++) {
      EGA[i] = 0x00;
  }
}
```

> Clearly, there is a whole world of awesome things there that we just couldn't do on the PC...
>
> You can just redraw the whole screen, but then it turns out...
>
> Well, you're going five frames a second.
>
> **John Carmack**[1]

[1] Interview with Lex Fridman in 2022, this quote is around 1h:41m.

So, how did they create a smooth scrolling game with these limitations? The solution was twofold:

- Utilizing specific EGA hardware tricks
- Updating only the portion of the screen that has changed

Each method is described in detail in the following sections.

0.1.1 EGA Virtual Screen and Pel Panning

The EGA card adds a powerful approach to linear addressing: the logical width of the virtual screen in VRAM does not need to match the physical screen display width. A programmer can define a virtual screen width of up to 4096 pixels and use the physical screen as a window onto any part of the virtual screen. What's more, the virtual screen's logical height can extend up to the total VRAM capacity. The code below demonstrates how to set a custom logical width.

```
CRTC_INDEX  = 03D4h
CRTC_OFFSET = 19

;==============
;
; set wide virtual screen
;
;==============

mov dx,CRTC_INDEX
mov al,CRTC_OFFSET
mov ah,[BYTE PTR width]  ;screen width in bytes
shr ah,1                 ;register expresses width
                         ;in word instead of byte
out dx,ax
```

The displayed area of the virtual screen at any time is determined by setting the display memory address for fetching video data, configured via the CRTC Start Address register. The default address is `A000:0000h`, though the offset can be adjusted to any other address.

```
CRTC_INDEX   = 03D4h
CRTC_STARTHIGH   = 12

;==============
;
; VW_SetScreen
;
;==============

  cli                       ;disable interrupts

  mov   cx,[crtc]           ;[crtc] is start address
  mov   dx,CRTC_INDEX       ;set CRTR register
  mov   al,CRTC_STARTHIGH   ;start address high register
  out   dx,al
  inc   dx                  ;port 03D5h
  mov   al,ch
  out   dx,al               ;set address high
  dec   dx                  ;set CRTR register
  mov   al,0dh              ;start address low register
  out   dx,al
  mov   al,cl
  inc   dx                  ;port 03D5h
  out   dx,al               ;set address low

  sti                       ;enable interrupts

  ret
```

To pan down a scan line, it requires only to increase the CRTC start address by the logical width. Horizontal panning is achieved by incrementing the start address by one byte. In EGA's planar graphics modes, the eight bits in each video RAM byte correspond to eight consecutive on-screen pixels, meaning the screen can move horizontally in steps of 8 pixels. This is rather coarse and not really smooth horizontal scrolling. Fortunately, another EGA register addresses this limitation.

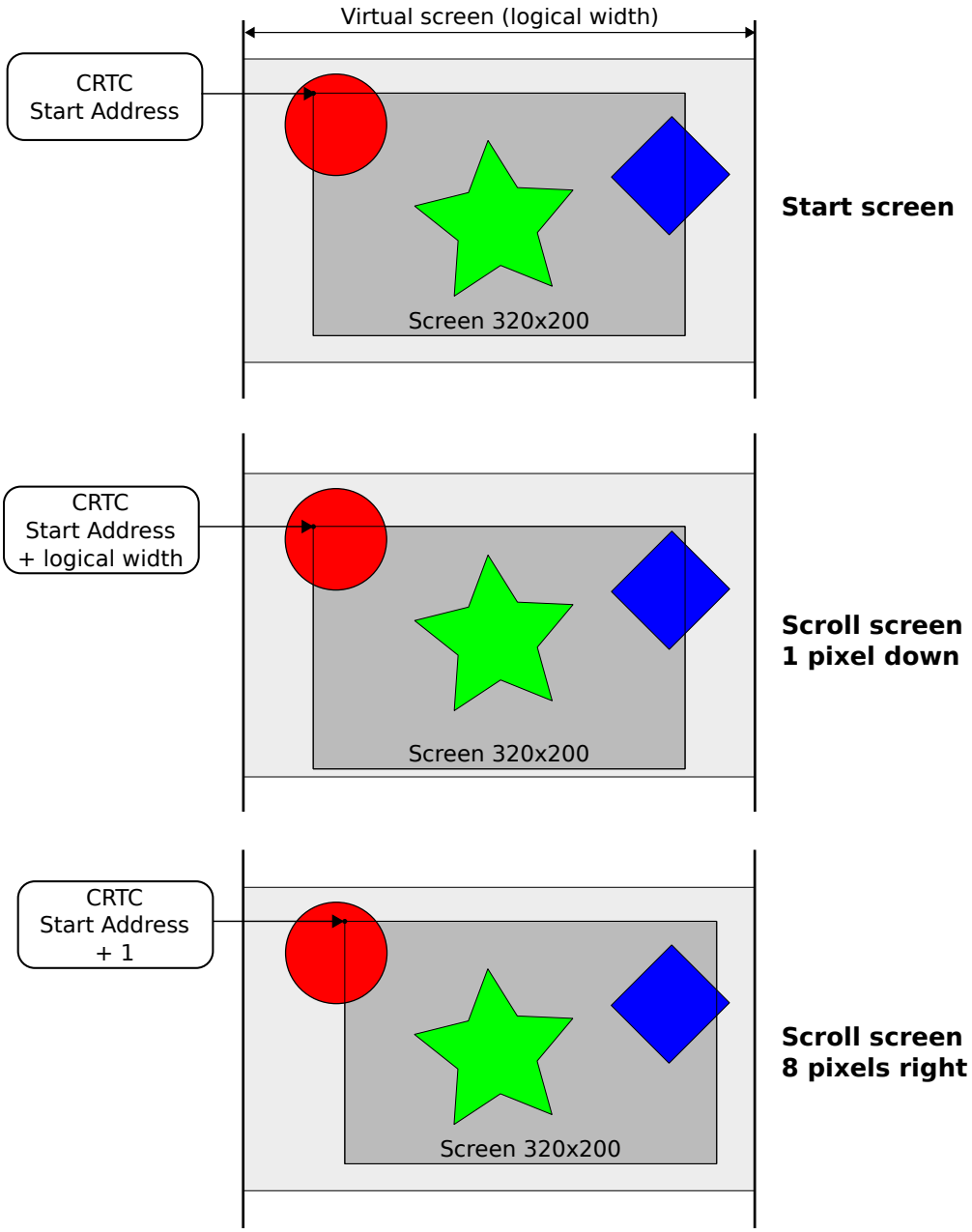


Figure 1: EGA screen scrolling by updating the CRTC Start Address.

0.1.2 Horizontal Pel Panning

Smooth horizontal pixel scrolling is managed by the "Horizontal Pel Panning" register in the Attribute Controller (ATC), allowing fine adjustment in 1-pixel increments up to 7 pixels to the left.

However, programming the ATC requires attention: the ATC Index register only uses the lower five bits (bits 0-4) as its internal index. The next most significant bit, bit 5, controls the source of the video data send to the monitor by the EGA card. When bit 5 is set to 1, the output of the color palette controls the displayed pixels; this is normal operation. When bit 5 is 0, video data doesn't come from the color palette, and the screen becomes a solid color. To maintain normal video operation, bit 5 must be set to 1 by writing `20h` to the register.

```
ATR_INDEX = 03C0h
ATR_PELPAN  = 19

;==============
;
; set horizontal panning
;
;==============

  mov dx,ATR_INDEX
  mov al,ATR_PELPAN or 20h ;horizontal pel panning register
                           ;(bit 5 is high to keep palette
                           ;RAM addressing on)
  out dx,al
  mov al,[BYTE pel]        ;pel pan value [0 to 8]
  out dx,al
```

Smooth horizontal scrolling on EGA involves adjusting the CRTC Start Address alongside fine-tuning within an 8-pixel range using horizontal pel panning. The following steps accomplish this:

- Calculate the panning offset in pixels for both `x`- and `y`-direction.
- Smooth y-panning is achieved by adding the `(logical width *) y` to the CRTC start address.
- For coarse horizontal scrolling, increase the CRTC start address by `x /8` bytes.
- Fine horizontal adjustment is applied using `(x % 8)` to the horizontal pel panning register.

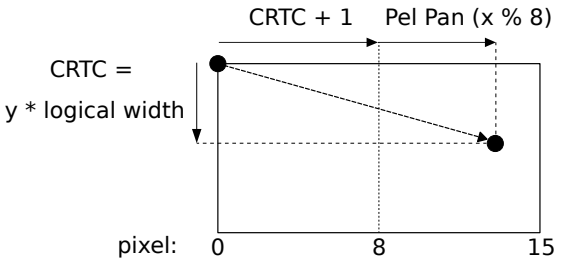


Figure 2: Smooth scrolling in EGA.

0.2 Adaptive Tile Refreshment

So far, we have built a virtual screen in VRAM allowing smooth one pixel moves in both axes using only EGA registers. However, once the screen edge is reached, a full-screen redraw would be too slow, causing a drop to 5 fps.

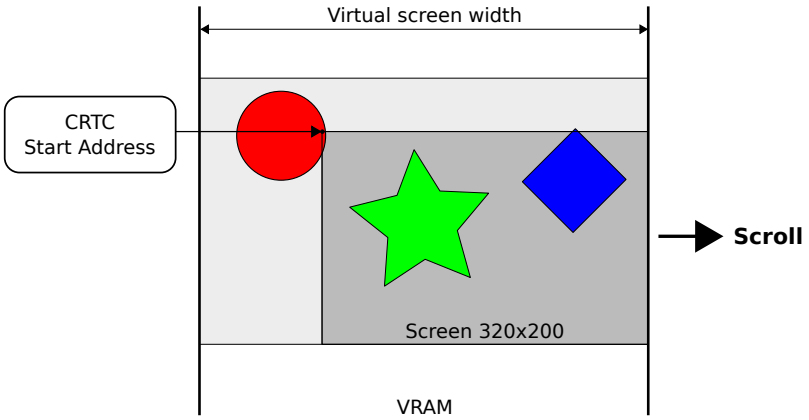


Figure 3: Screen reaching the edge of virtual screen.

One option for screen refresh is to load the entire level into VRAM, setting the logical width to match the level width. By adjusting the CRTC Start Address and Horizontal Pel Panning registers, smooth scrolling can be achieved. However, loading the entire level requires far more VRAM than is available. The first level, *Horsh Radish Hill*, is 136 tiles wide and 37 tiles high. Each tile is 128 bytes, which means a total of 136 x 37 x 128 bytes = 644KB is required to store the entire level in VRAM. Since we only have 256KB available, this is not a feasible option.

Trivia : At the time of writing this book, most video cards contain more than 1GB VRAM, sufficient to store all Commander Keen levels at once in VRAM.

To address this, John Carmack invented "Adaptive Tile Refresh" (ATR). The core idea is to refresh only those areas of the screen that need to change.

Let's look at *Commander Keen 1: Marooned on Mars* in Figure 4. This is the first level of Marooned, immediately to the right of the crashed Bean-with-Bacon Megarocket. The first figure is the start of the level, the second figure is after Keen has scrolled one tile (16 pixels) to the right through the world. They look almost identical to the naked eye, don't they?

Now, if we perform a difference on both images, you can see which tiles need to be changed upon screen refresh. The trick behind the scrolling is to only redraw tiles that actually changed after panning 16 pixels (one tile). For matching tiles there is nothing to do and they are skipped entirely. In Figure 4.21, there are large swathes of constant background tiles. In total, only 69 tiles out of the 260 tiles need to be refreshed, which is 27% of the screen!

This is also the part where the game designers Tom Hall and Jogn Romero had to help the engine. Smooth scrolling is inversely proportional to the number of tiles to redraw. A checkerboard tile pattern basically means a full-screen redraw and would kill the CPU. So to avoid costly "jolts", the game designers built tile maps with a lot of reperating tiles.

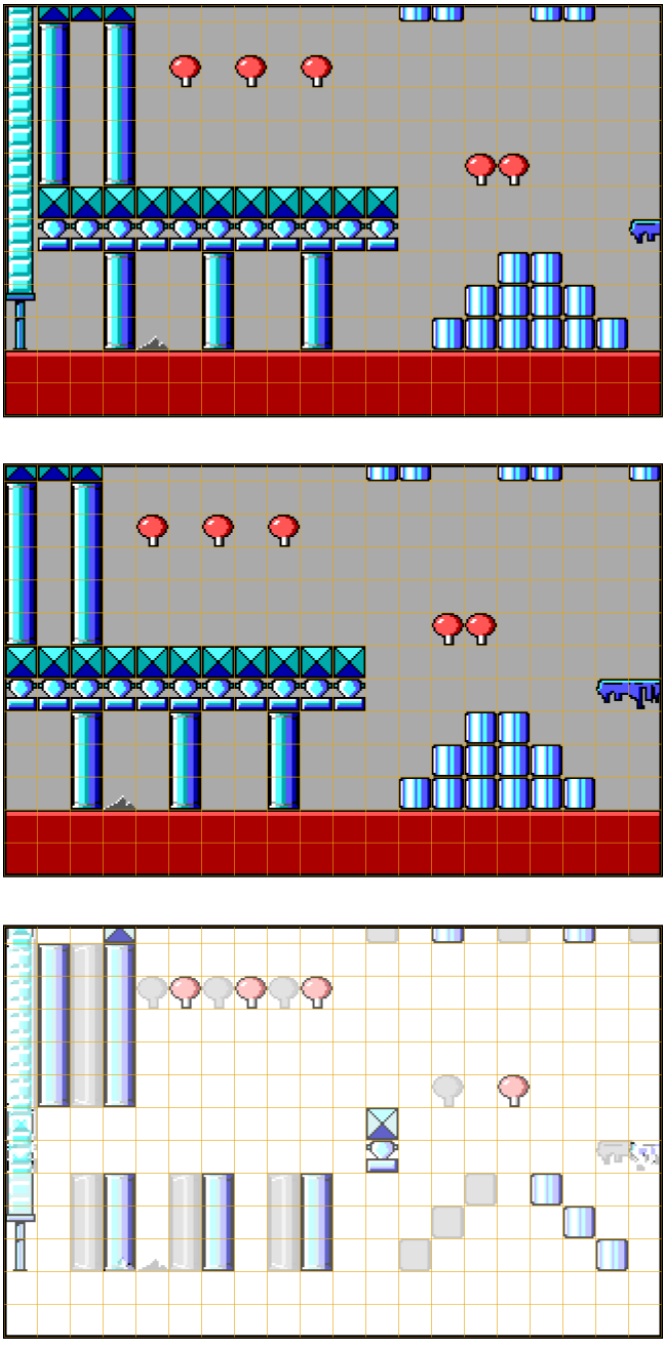
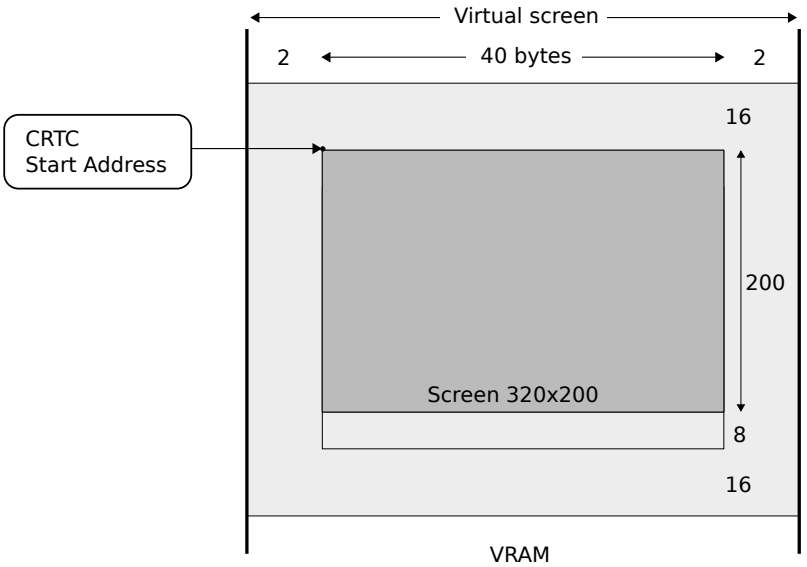


Figure 4: Start of the world, moved one tile to the right and difference.

0.2.1 Adaptive tile refreshment in Commander Keen 1-3

This section explains how ATR is working for the first three episodes of Commander Keen. Since there is no source code released for these episodes, ATR will be explained without code examples. The later versions of Commander Keen, including Keen Dreams, used a different, improved engine which will be explained in the next section[2].

The EGA screen in mode `0xD` has a resolution of 320x200 pixels, or 40x200 bytes. Let's extend the height by 8 bytes to have a height of 208 pixels, so the screen fits nicely in 20x13 tiles. By making the virtual screen one tile higher and one wider on each side of the screen, the engine can scroll up to 16 pixels to any direction of the screen without any tile refresh, by simply adjusting the CRTC Start Address and Pel Pan registers.



Now, let's have a closer look at the EGA VRAM setup. The video memory is organized into three virtual screens:

- Page 0 and 1, which are used to switch between buffer and visible screen. The idea between two pages (double buffer) is that the code can draw in the second buffer while the first buffer is being shown on screen, which is then switched out during screen refresh. This ensures that no frame is ever displayed mid-drawing, which yields smooth, flicker-free animation.

[2] https://retrocomputing.stackexchange.com/questions/22175/what-is-adaptive-tile-refresh-in-the-context-of-commander-keen

- A master page containing a static page, which is copied to the buffer screen when performing the screen refresh.

Each virtual screen has a size of 44*240*4=42,420 bytes. So within a 256KB EGA card there is enough VRAM available to keep all three virtual screens in memory. The page that is displayed on screen is selected by setting the CRTC Start Address register at which to begin fetching video data.

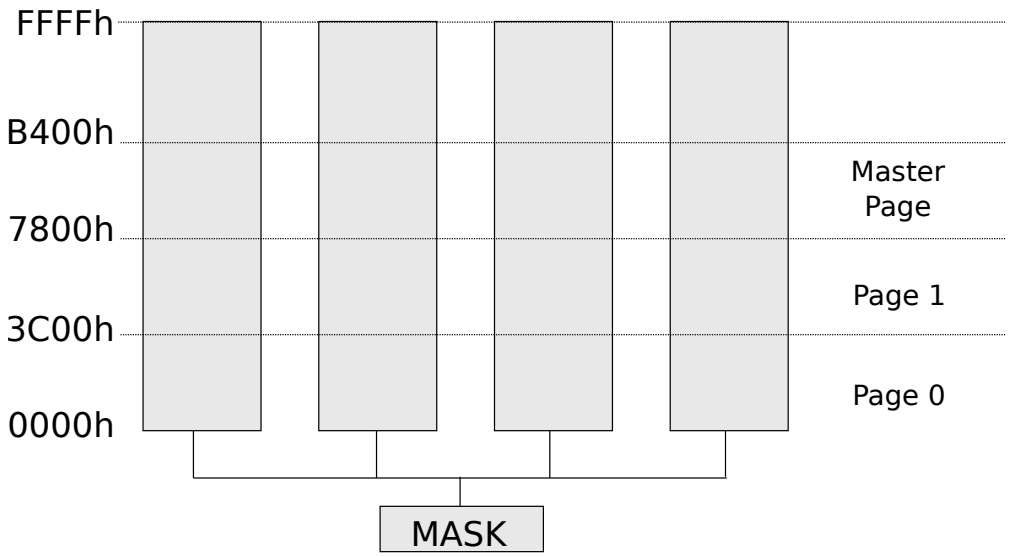


Figure 5: Virtual screen layout on EGA card.

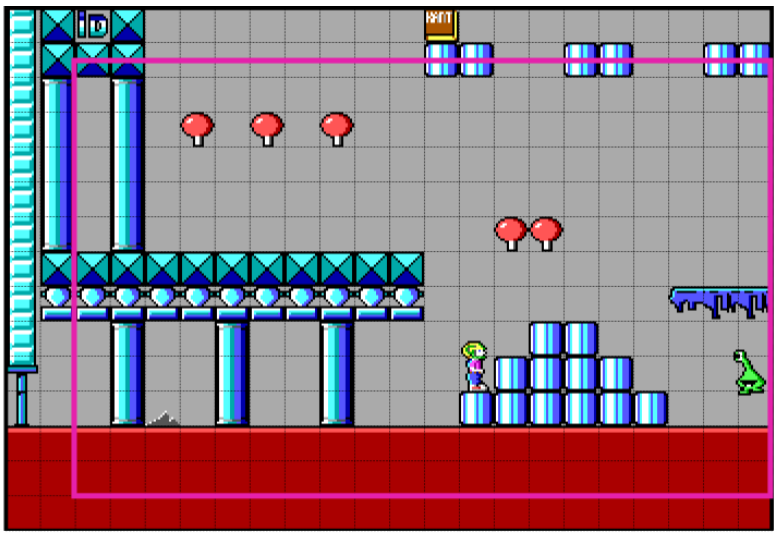
For both the buffer and visible screen a tile array is created to maintain which tiles are changed since last refresh.

```
byte    update[2][UPDATESIZE];
```

Steps to refresh the screen are as follows:

1. Check if the player has moved one tile in any direction.
2. Identify changed tiles and copy these to the master page, marking them in both tile arrays.
3. Refresh the buffer page by copying marked tiles from the master page.
4. Switch the view and buffer pages by adjusting the CRTC Start Address and Pel Panning registers.

Ignoring sprites for now, the following steps illustrate these stages. In the next four screenshots, we take you step-by-step through each of the stages. The player has reached the edge of the virtual screen and moves further to the right.



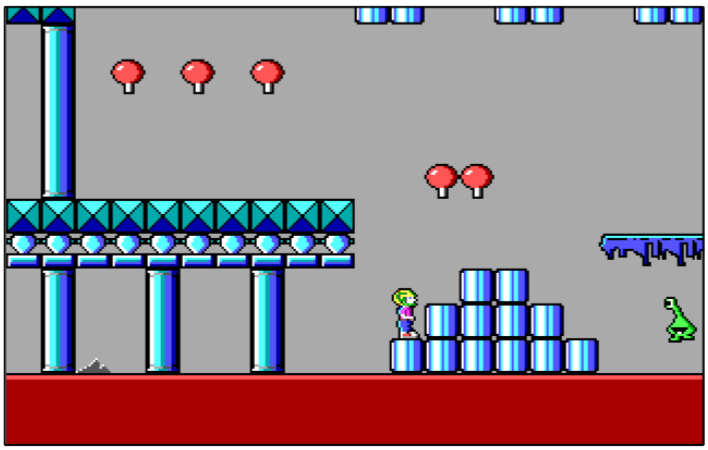


Figure 6: Start: Reach the end of the virtual screen.

The engine keeps track of which tile numbers are part of the virtual screen. Since the engine only refresh the screen at tile size granularity, it can determine extremely fast what has changed on the screen by comparing tile numbers. If the tile number has changed, the tile is updated by copying tiles from RAM into the VRAM master page. The changed tiles are marked with a '1' in both tile arrays, meaning it needs to be updated upon next refresh.

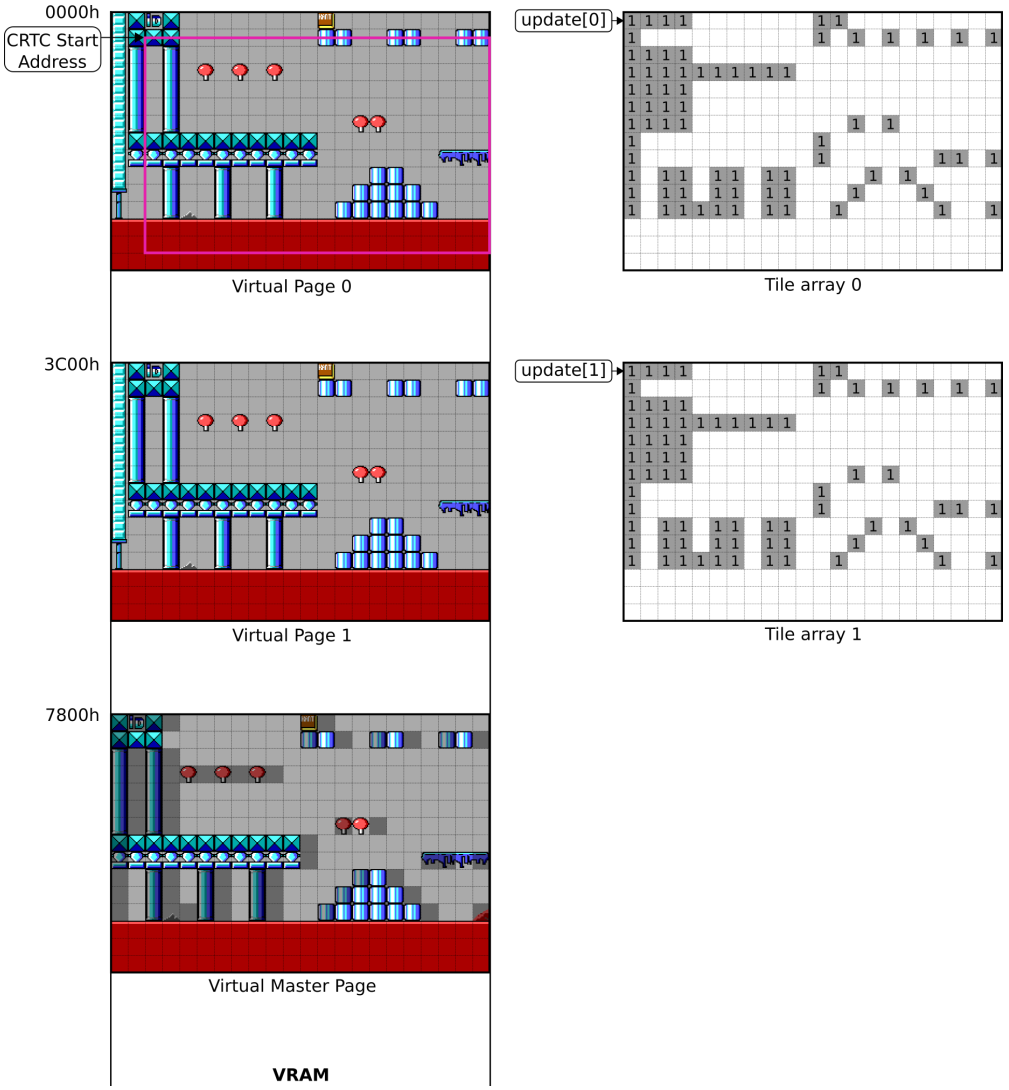


Figure 7: Update tiles in master page and update both tile arrays.

The next step is to scan all tiles in buffer tile array (array 1) and for each tile marked '1', copy the corresponding tile from master to virtual page 1 page.

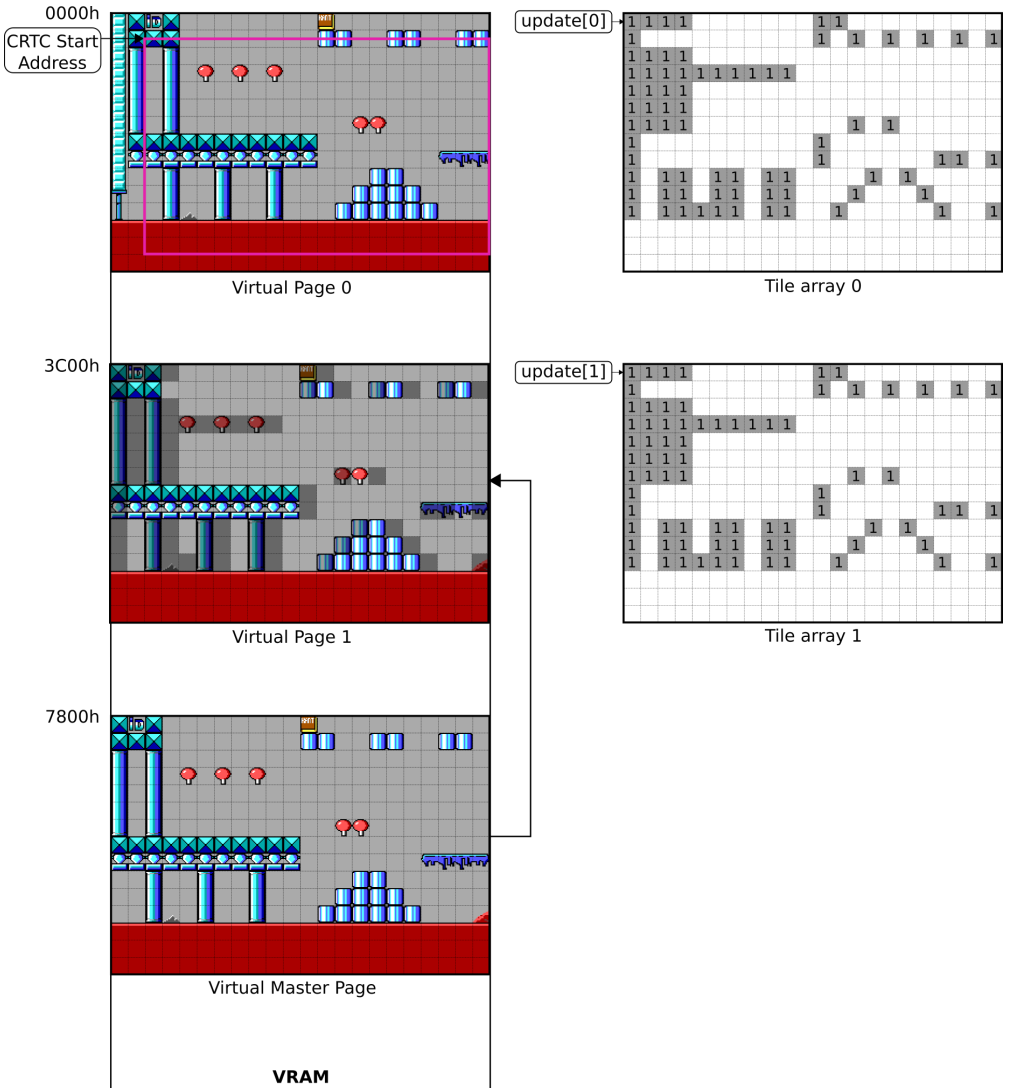


Figure 8: Copy changes tiles from virtual master page to page 1.

In the final step, point the visible screen to virtual page 1 by updating the CRTC start address. Finally, tile array 1 is cleared to '0'. Now the first step is repeated, but this time virtual page 0 acts as the buffer screen. Note that after swapping, tile array 0 keeps marked tiles from last update. This makes sense, as the current buffer page is not yet refreshed since it was displayed in the previous refresh cycle.

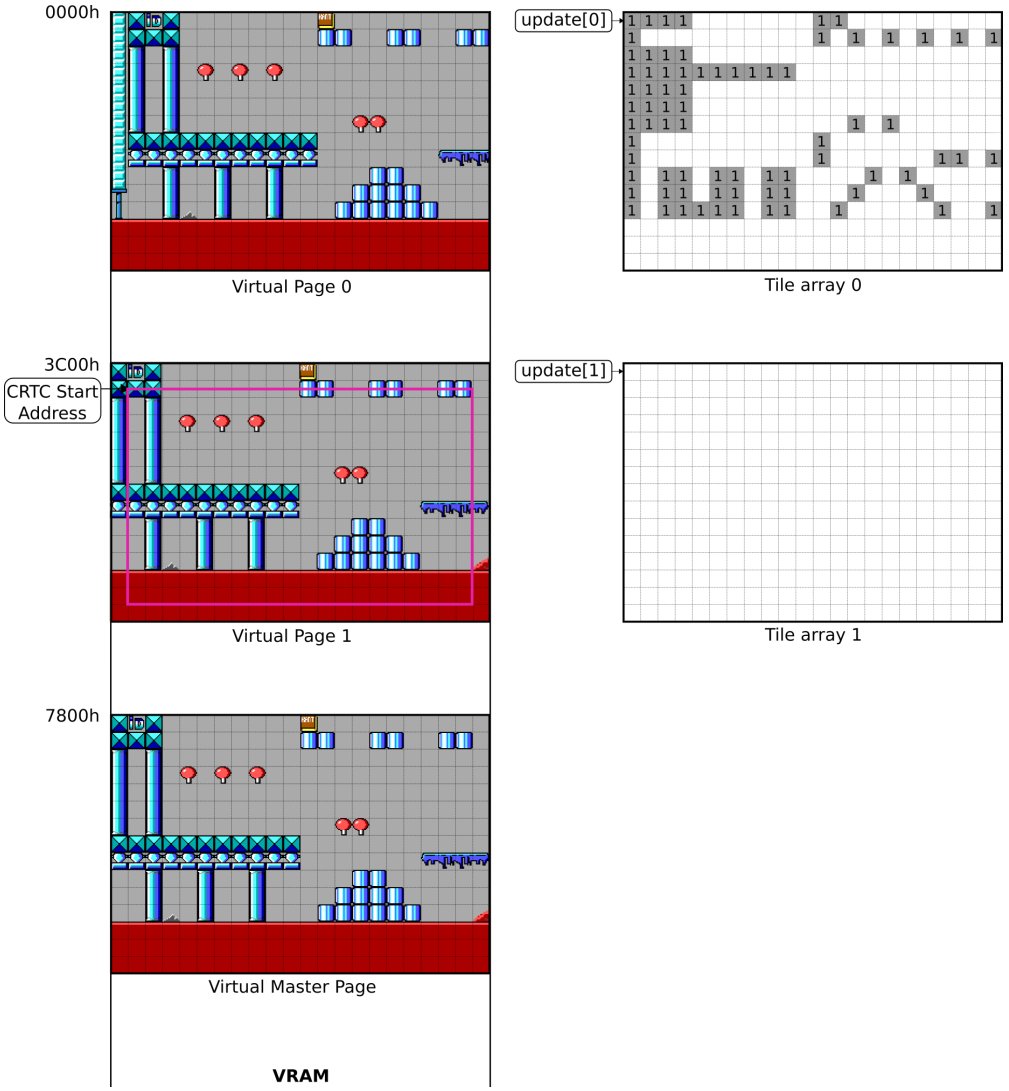
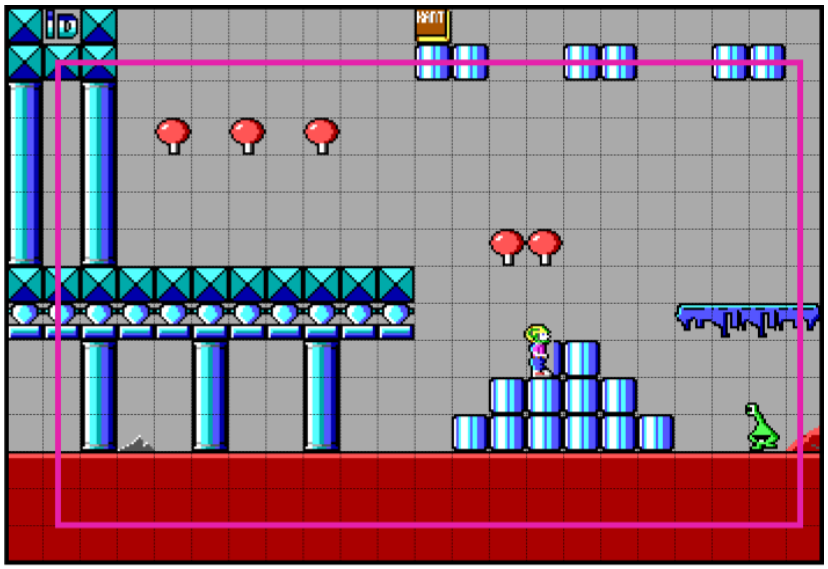


Figure 9: Update CRTC Start Address to virtual page 1 and empty tile array 1.

Now the buffer is refreshed and the CRTC address is updated, the final step is fine adjustment using the Pel Panning register.



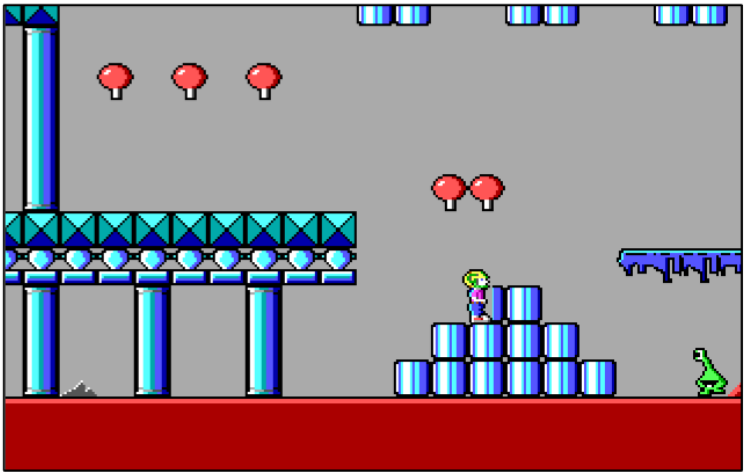


Figure 10: Screen is refreshed and scrolled to the right.

0.2.2 Wrap around the EGA Memory

John Carmack explored what would happen if you push the virtual screen over the 64kB border (address `0xFFFF`) in video memory. It turned out that the EGA continues the virtual screen at `0x0000`. This means you could wrap the virtual screen around the EGA memory and only need to add a stroke of tiles on one of the edges when Commander Keen moves more than 16 pixels.

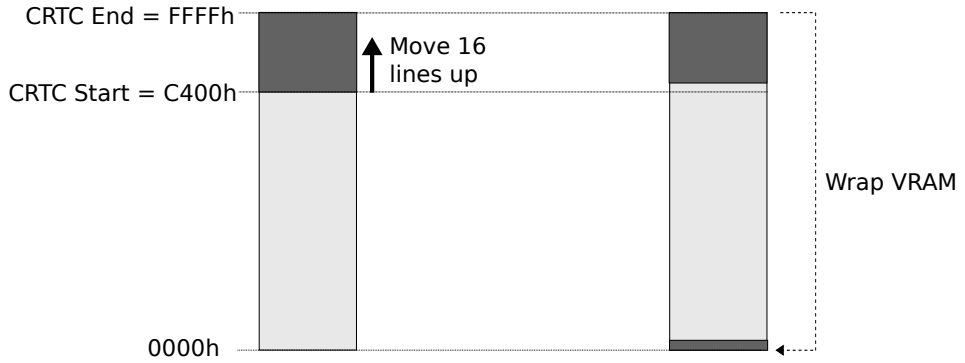


Figure 11: Wrap virtual screen around the EGA memory

> I finally asked what actually happens if you just go off the edge [OF THE VRAM]?
>
> If you take your [CRTC] start and you say OK, I can move over and I get to what should be the bottom of the memory window. [...] What happens if I start at 0xFFFE at the very end of the 64k block? It turns out it just wraps back around to the top of the block.
>
> I'm like oh well this makes everything easy. You can just scroll the screen everywhere and all you have to draw is just one new line of tiles.
>
> It just works. We no longer had the problem of having fields of similar colors. It doesn't matter what you're doing, you could be having a completely unique world and you're just drawing the new strip.
>
> **John Carmack**[3]

[3] An explanation further elaborated during the same interview with Lex Fridman in 2022.

There was however an issue with the introduction of Super VGA cards, which had typically more than 256kB RAM[4]. This resulted in crippled backwards compatibility and the wrapping around `0xFFFF` did not work anymore on these cards.

There is an easy solution to resolve this issue. As you can see in Figure 5 on page 10, the space between `0xB400` and `0xFFFF` is not used and contains enough space for another virtual screen. In case the start address is between `0xC400` and `0xFFFF` the corresponding screen is copied to the opposite end of the buffer, as illustrated in Figure 12.

> “ I was in a tough position. Do I have to track every single one of these [SUPER VGA CARDS] and it was a madhouse back then with 20 different video card vendors with all slightly different implementations of their non-standard functionality. Either I needed to natively program all of the cards or I kind of punt. I took the easy solution of when you finally did run to the edge of the screen I accepted a hitch and just copied the whole screen up.
>
> **John Carmack[5]** ”

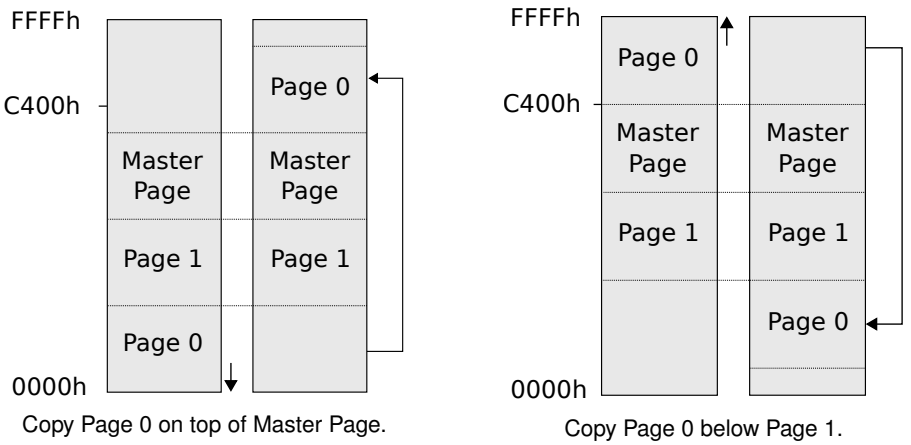


Figure 12: Move screen to opposite end of VRAM buffer

[4] In 1989 the VESA consortium standardized an API to use Super VGA modes in a generic way. One of the first modes was 640x480 at 256 colors requiring at least 256kB RAM, which from a hardware constraint resulted in 512kB.

[5] And again the same interview with Lex Fridman in 2022.

```
#define SCREENSPACE   (SCREENWIDTH*240)
#define FREEEGAMEM    (0x100001-31*SCREENSPACE)

screenmove = deltay*16*SCREENWIDTH + deltax*TILEWIDTH;
for (i=0;i<3;i++)
{
  screenstart[i]+= screenmove;
  if (compatability && screenstart[i] > (0x100001-
   SCREENSPACE) )
  {
    // move the screen to the opposite end of the buffer
    screencopy = screenmove>0 ? FREEEGAMEM : -FREEEGAMEM;
    oldscreen = screenstart[i] - screenmove;
    newscreen = oldscreen + screencopy;
    screenstart[i] = newscreen + screenmove;
    // Copy the screen to new location
    VW_ScreenToScreen (oldscreen,newscreen,
       PORTTILESWIDE*2,PORTTILESHIGH*16);

    if (i==screenpage)
      VW_SetScreen(newscreen+oldpanadjust,oldpanx &
   xpanmask);
  }
}
```

As explained in Section **??** each pixel is encoded by four bits, which are spread across the four EGA banks. So how can we copy four VRAM planes fast enough, without noticing any performance hit? To copy eight pixels, byte-aligned, one would have to do four read and four writes.

But how can we copy four VRAM planes fast enough, without noticing any performance hit? As explained in Section 3.3.7 each pixel is encoded by four bits, which are spread across the four EGA banks. To copy eight pixels, byte-aligned, one would have to do four read and four writes.

Having a closer look at the EGA card one notice there is a latch placed in front of the ALU, which can be re-purposed for the greater good. With a little creativity the ALU in front of each bank can be setup to use only the latch for writing[6]. With such a setup, upon doing one read, four latches are populated at once and four bytes in the bank are written with only one write to the RAM. Now it is possible to copy the entire buffer fast enough, without notifying any performance impact.

[6]The mask trick is discussed in *Game Engine Black Book: Wolfenstein 3D* for the VGA card. The trick is the same for the EGA card.

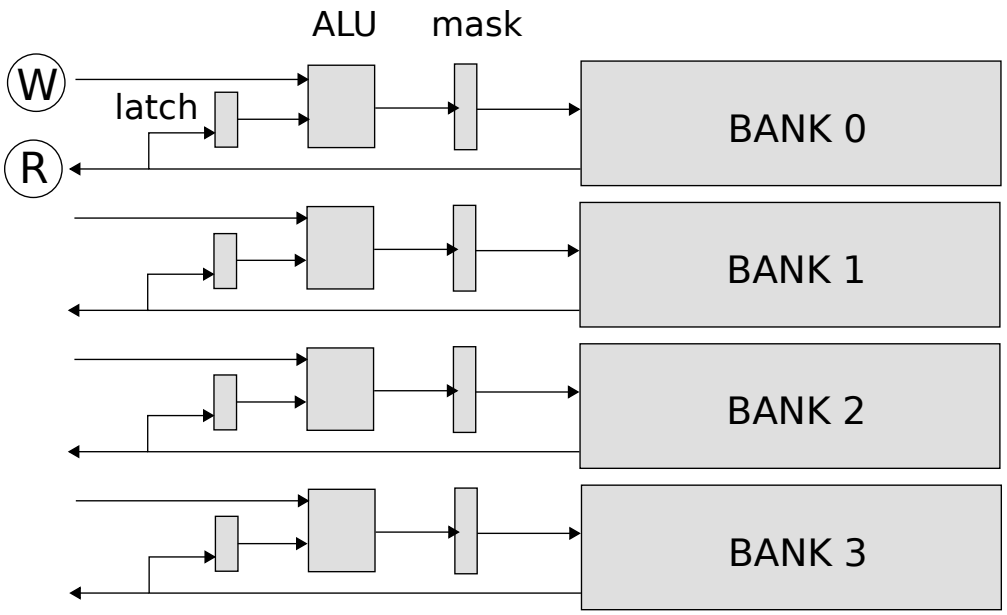


Figure 13: Latches memorize read operations from each bank. The memorized value can be used for later writes.

```
GC_INDEX     =    0x3CE     ;Graphics Controller register
GC_MODE      =    5         ;mode register
SC_INDEX     =    0x3C4     ;Sequence register
SC_MAPMASK   =    2         ;map mask register

;==============
; Set EGA mode to read/write from latch
;==============
cli                         ;interrupts disabled
mov dx,GC_INDEX             ;mode 1, each memory plane is
mov ax,GC_MODE+256*1        ;written with the content of
out dx,ax                   ;the latches only

mov dx,SC_INDEX             ;enable writing to all 4 planes
mov ax,SC_MAPMASK+15*256    ;at once
out dx,ax
sti                         ;interrupts enabled
```

0.2.3 Adaptive tile refreshment in Keen Dreams

The EGA memory wrapping results in an improved, more simplified Adaptive Tile Refreshment algorithm. First, a virtual screen and tile array is defined by making the display one tile taller and wider, creating what is known as the viewport. This allows the engine to scroll the display up to 16 pixels to the right and bottom without refreshing tiles, by simply adjusting the CRTC Start Address and Pel Pan register.

An additional column is then added to the viewport to support horizontal scrolling. Finally, the tile array is further expanded to allow the entire viewport to move up to two tiles in any direction.

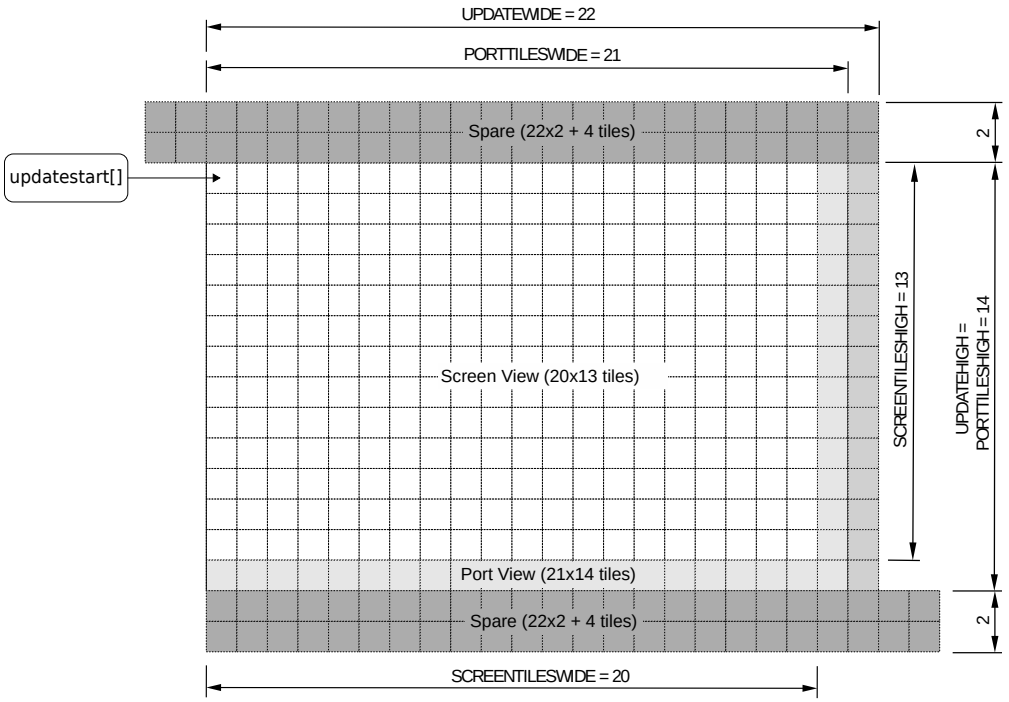


Figure 14: Tile array layout.

The full refresh cycle, including sprite updates, proceeds as follows:

1. Verify if the player has moved one tile in any direction.
2. Copy the new column or row of tiles to the master page, update both the tile array pointers, and mark the tiles in both arrays.

3. Refresh the buffer page by scanning the tile index array. If a tile is marked, copy it from the master page to the buffer page.
4. Iterate through the sprite removal list, copying the corresponding image block from the master page to the buffer page to clear the sprite.
5. Iterate through the sprite list, copying each sprite image block to the buffer page.
6. Switch the view and buffer pages by adjusting the CRTC Start Address and Pel Panning registers.

```
void RF_Refresh (void)
{
    updateptr = updatestart[otherpage];

    RFL_AnimateTiles ();// update animated tiles

    // copy newly scrolled and animated tiles
    // from the master to buffer screen
    EGAWRITEMODE(1);
    EGAMAPMASK(15);      // write 4 bytes of VRAM at once
    RFL_UpdateTiles (); // copy from master to buffer page
    RFL_EraseBlocks (); // remove sprites

    // update sprites
    EGAWRITEMODE(0);
    RFL_UpdateSprites ();

    // display the changed screen (swap view and buffer)
    VW_SetScreen(bufferofs+panadjust,panx & xpanmask);
}
```

Let's go over the refresh cycle step by step. At the start, all three virtual pages display the same viewport, with virtual page 0 currently shown on-screen. Both tile arrays are empty, meaning no tile updates are required in the next refresh cycle. The player has reached the edge of the virtual screen and moves further to the left.

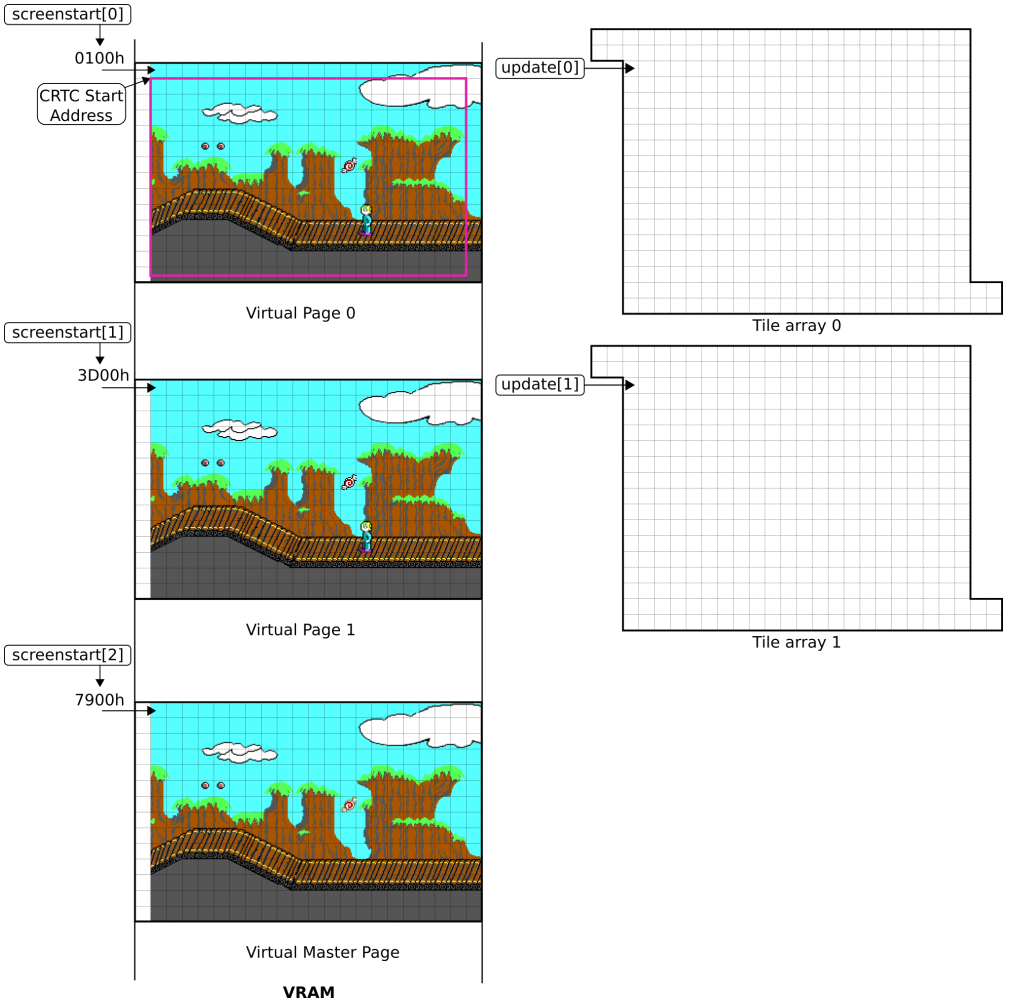


Figure 15: VRAM and tile arrays before scrolling to the left.

Both VRAM and tile array pointers shift left to introduce a new column of tiles. The VRAM pointer is lowered by 2 bytes (1 tile), and the tile array pointer is decreased by 1 byte. A new column of tiles is copied from RAM to the master page, while the leftmost column in both tile arrays is marked with '1' to ensure it updates in the next refresh cycle. Animated tiles are then copied to the master page, and the corresponding tiles in the array are also marked with '1'.

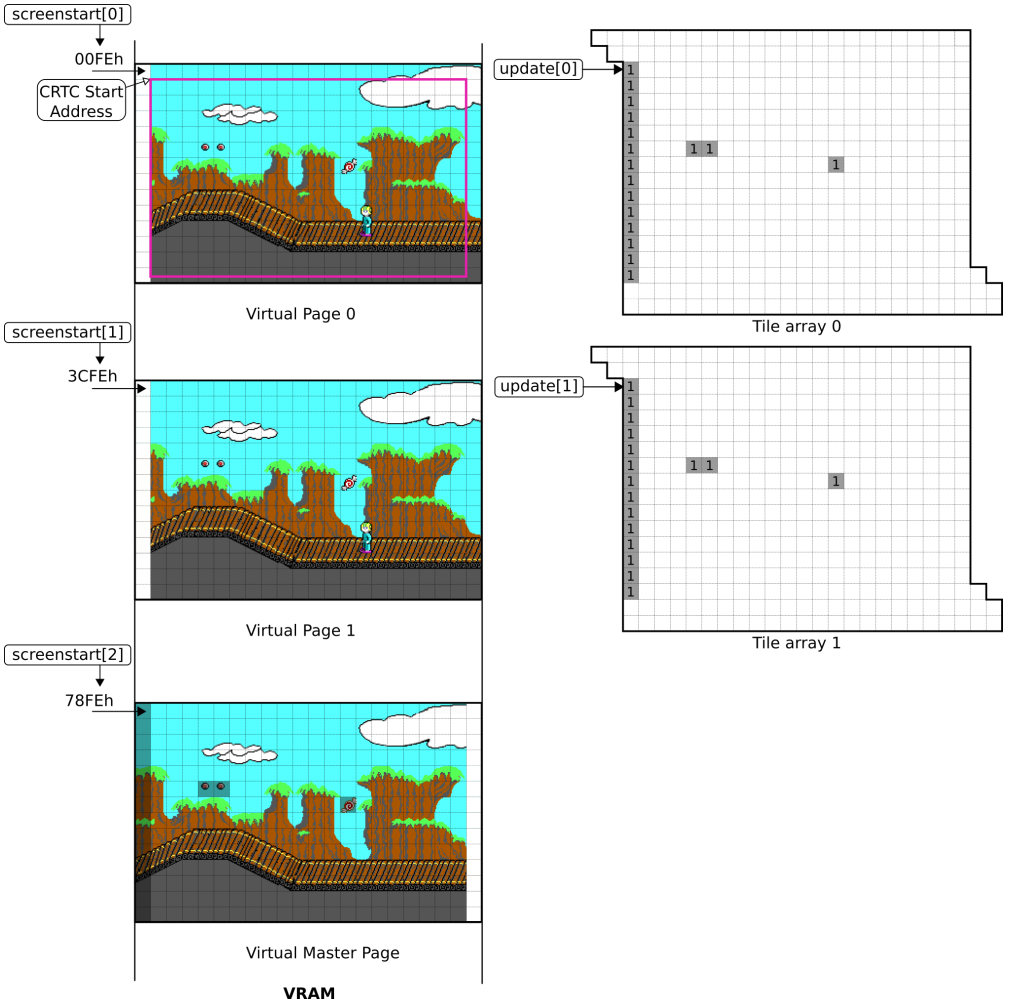


Figure 16: Update Virtual master with new left column and animated tiles.

Next, the engine scans all ’1’s and copies the corresponding tiles from the master to the buffer page.

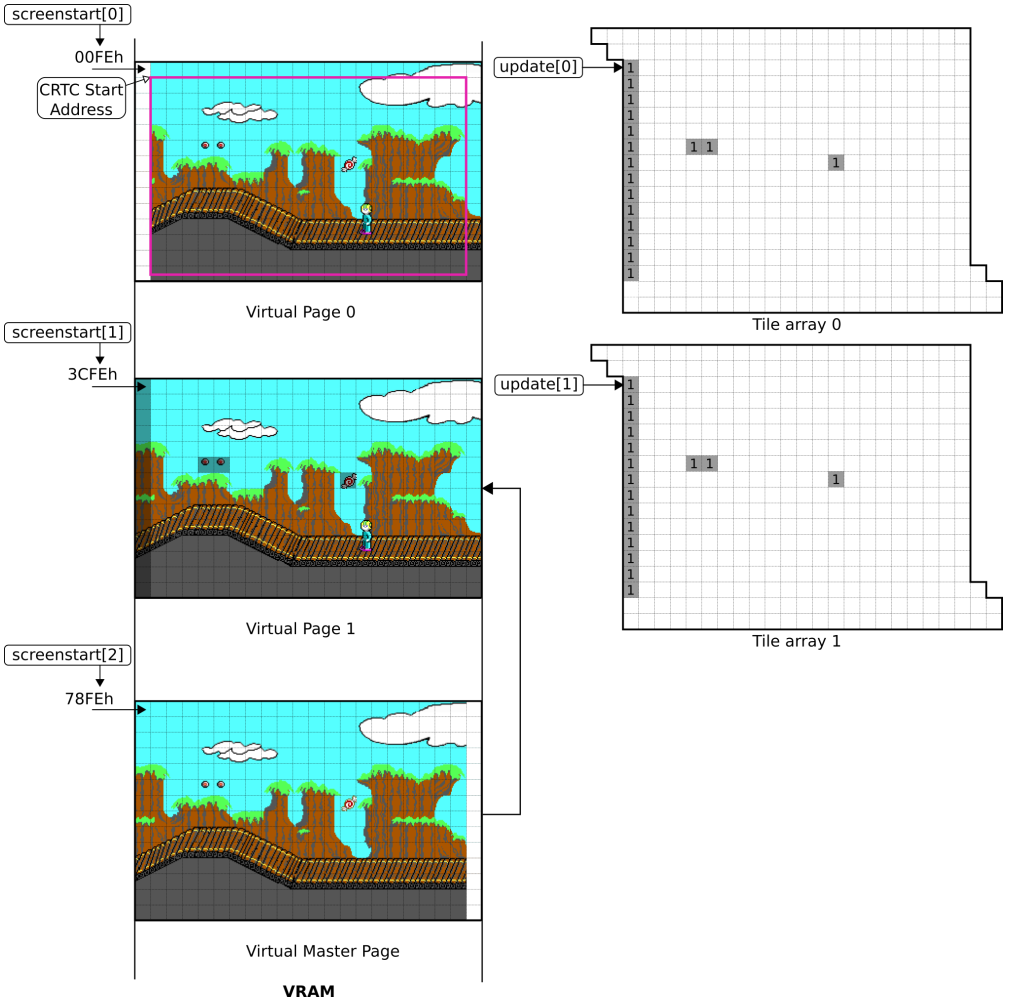


Figure 17: Copy changed tiles from master to buffer page.

Each sprite removed from the screen has its location and size stored in the sprite removal list, which removes the sprite by copying a specific section from the master screen to the virtual page. Tiles overlapping with the removal block are marked with a '2'.

```
typedef struct
{
    int     screenx,screeny;
    int     width,height;
} eraseblocktype;

//sprite removal list for Page 0 and Page 1
eraseblocktype  eraselist[2][MAXSPRITES],*eraselistptr[2];
```

```
void RFL_EraseBlocks (void)
{
  eraseblocktype  *block,*done;
  unsigned  pos;

  block = &eraselist[0][0];
  done = eraselistptr[0];

  while (block != done)
  {
    [...]

    //
    // erase the block by copying from the master screen
    //
    pos = ylookup[block->screeny]+block->screenx;
    block->width = (block->width + (pos&1) + 1)& ~1;
    pos &= ~1;          // make sure a word copy gets used
    VW_ScreenToScreen (masterofs+pos,bufferofs+pos,
      block->width,block->height);

    [...]
    block++;
  }
}
```

Trivia : The '2' marking is nowhere used in the engine, most likely it is used in the original ATR algorithm.

The final step in updating the buffer is copying sprites to their new locations, with each corresponding tile marked with a '3' in the tile array.

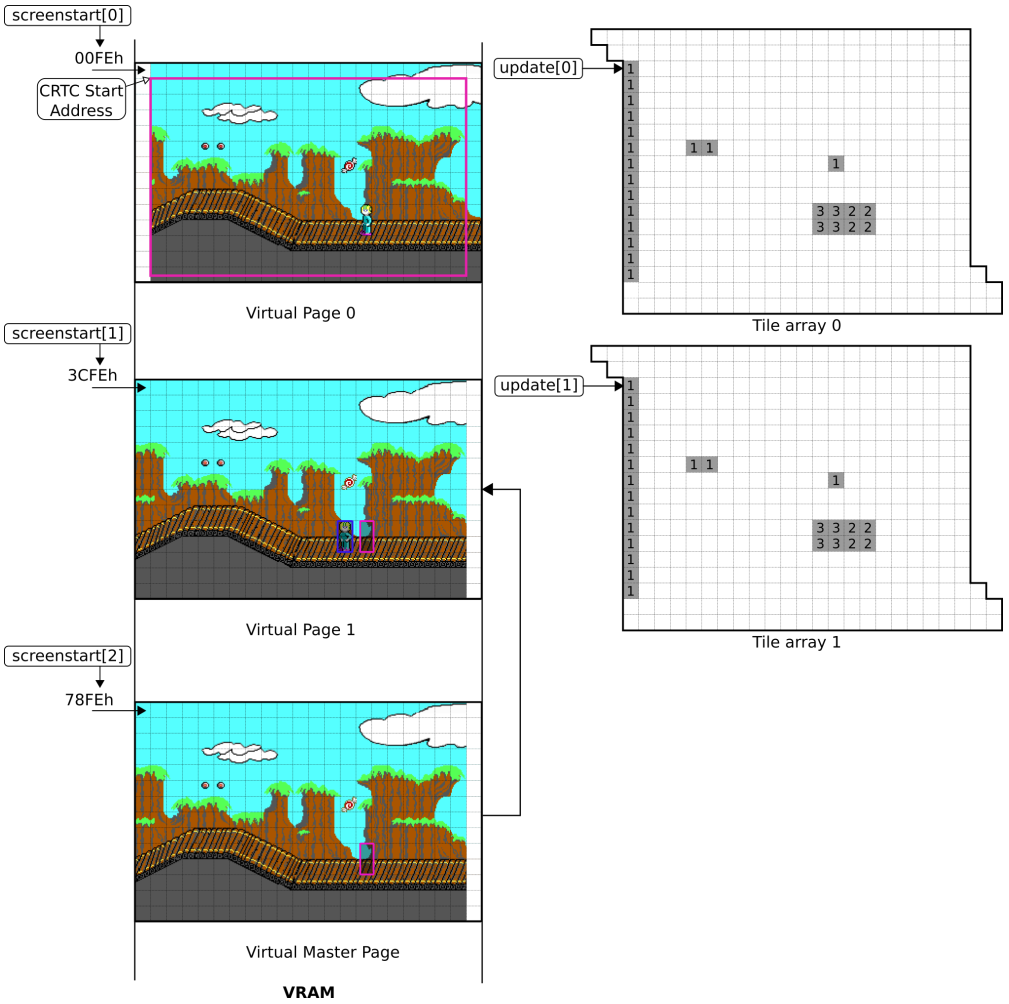


Figure 18: Removing and updating sprites to the buffer screen.

Once the buffer refresh is completed, the engine needs only to update the CRTC start address to virtual page 1. At this point, the visible tile array is emptied, and the pointer resets to its starting location. The new buffer array (virtual page 0) retains the marked tiles since this screen has not yet refreshed. At this point, the refresh cycle restarts, with virtual page 0 now functioning as the buffer.

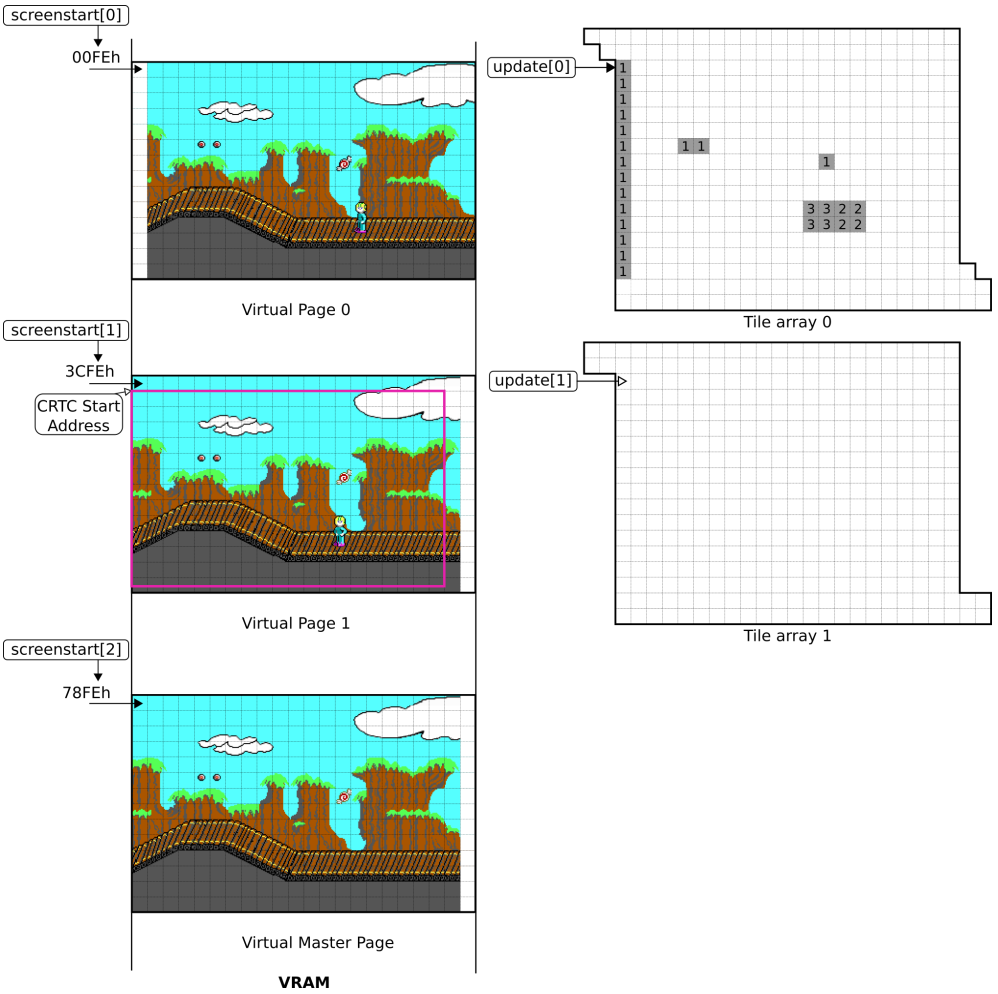


Figure 19: Swap buffer and visible screen by updating CRTC start address.

The final step in scrolling is to adjust the horizontal fine-pixel alignment by setting the Pel Pan register. *Et voilà*, the screen scrolls smoothly to the left. This scrolling process is significantly more efficient than the original ATR engine, resulting in only 8% of tiles to be refreshed!

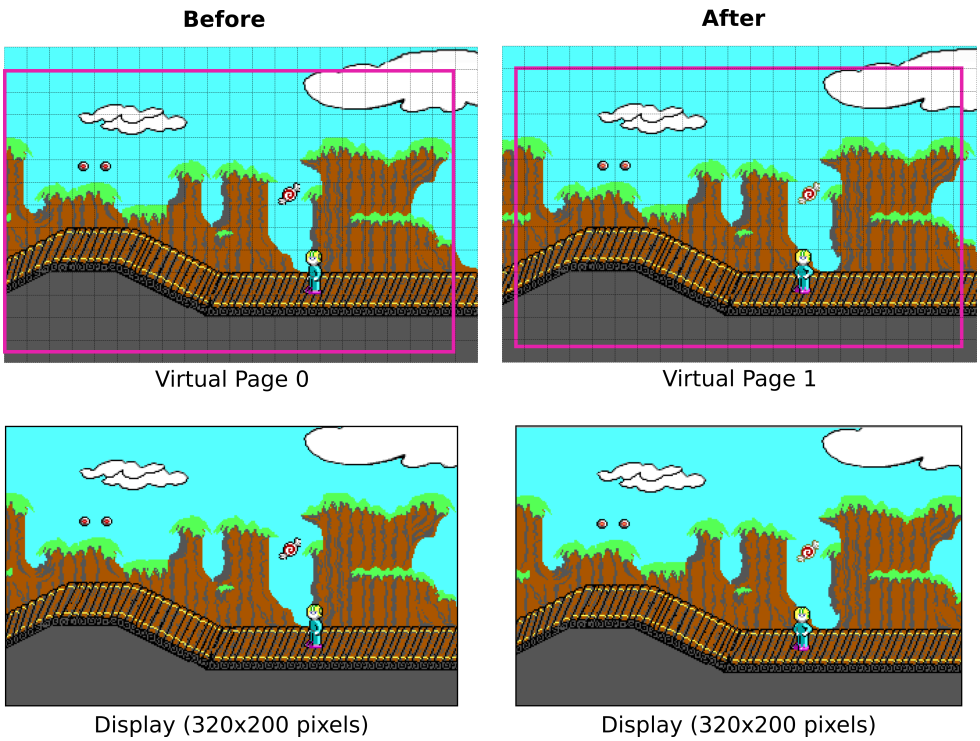
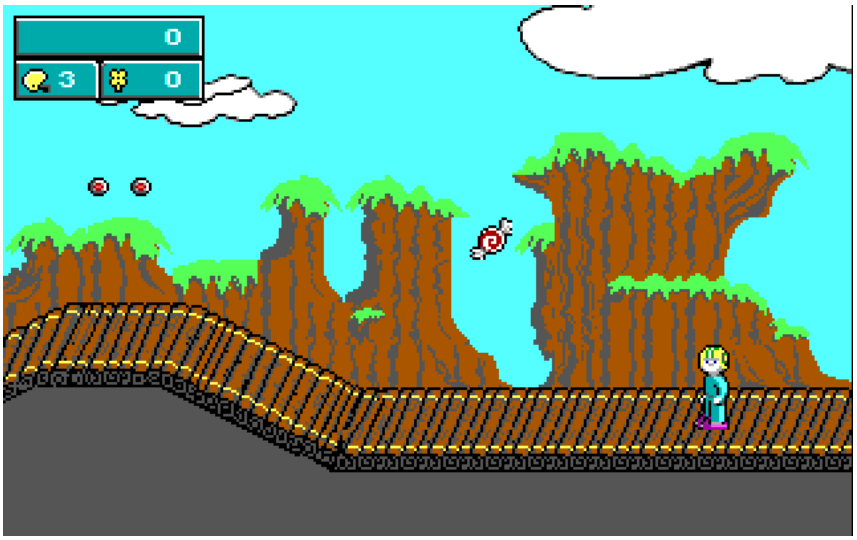


Figure 20: Left screen is before scrolling, right screen after scrolling.

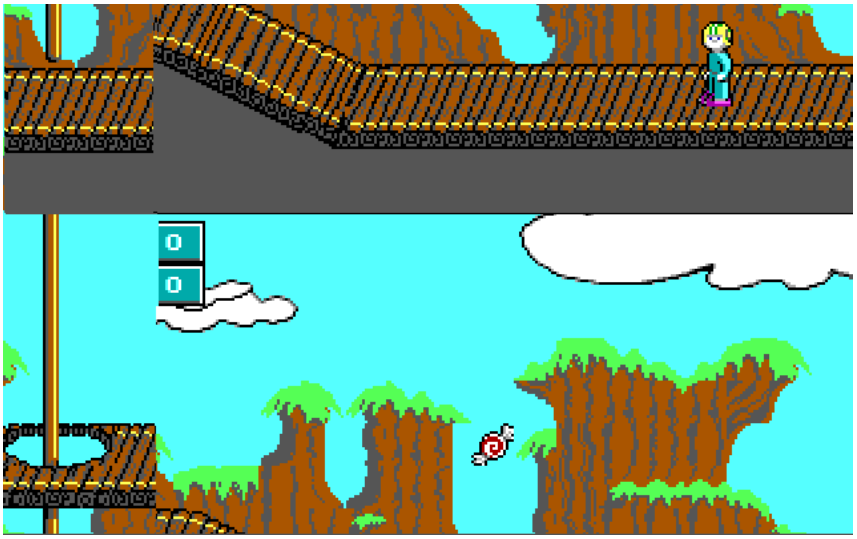
If the player continues to move to the left, the tile array 0 pointer lowers a second time, underscoring why additional space is needed to allow the entire viewport to float up to two tiles in all directions.

0.2.4 Screen refresh

Flipping between the pages is as simple as setting the CRTC start address registers to page 0 or page 1 starting point. However, there is one issue to solve. If you were to run it, every once in a while the expected screen shown below...



...would instead appear distorted, showing both misalignment and parts of two pages:



This problem results from the timing between updating the CRTC start address and the screen refresh. The start address is latched by the EGA's internal circuitry exactly once per frame, usually at the beginning of the vertical retrace. Although the CRTC start address is a 16-bit value, the `out` instruction can only write 8 bits at a time.

This issue can be illustrated with the following setup: the current CRTC start address (Page 0) points to 0000h, while the buffer (Page 1) points to 3C00h. After moving one tile to the left, Page 0 now points to FFFEh in VRAM, and Page 1 points to 3BFEh. Since Page 1 is the updated buffer, it will be displayed in the next refresh cycle. However, due to poor timing in updating the vertical retrace and start address, the CRTC only picks up the first byte of the address, 3Bh, setting the start address to 3B00h instead of 3BFEh.

```
CRTC_INDEX   = 03D4h
CRTC_STARTHIGH   = 12

  cli                       ;disable interrupts
  mov   cx,[crtc]           ;[crtc] is start address
  mov   dx,CRTC_INDEX       ;set CRTR register
  mov   al,CRTC_STARTHIGH   ;start address high register
  out   dx,al
  inc   dx                  ;port 03D5h
  mov   al,ch
  out   dx,al               ;set address high

  ;******** VERTICAL RETRACE STARTS HERE  !!!!!!!  ****
  ;******** AND SHOWS 2 PARTIAL FRAMEBUFFERS **********

  dec   dx                  ;set CRTR register
  mov   al,0dh              ;start address low register
  out   dx,al
  mov   al,cl
  inc   dx                  ;port 03D5h
  out   dx,al               ;set address low
  sti                       ;enable interrupts

  ret
```

One solution to this issue is to update the start address when the vertical retrace signal is detected through the Input Status 1 Register, specifically bit 3 of 3DAh. Unfortunately, by the time this status is observed by the program, the start address for the next frame has already been latched, occurring immediately when the vertical retrace pulse begins.

To address this, the start address must be updated at a point sufficiently distant from the start of the vertical retrace. This requires identifying a signal that confirms the completion of a horizontal or vertical retrace and the beginning of a new scan line, far enough from the vertical retrace to ensure the new start address is latched during the next vertical sync. The Display Enable Status signal, accessible via the Input Status 1 Register, provides this information; a value of 1 indicates that the display is within a horizontal or vertical retracement[7].

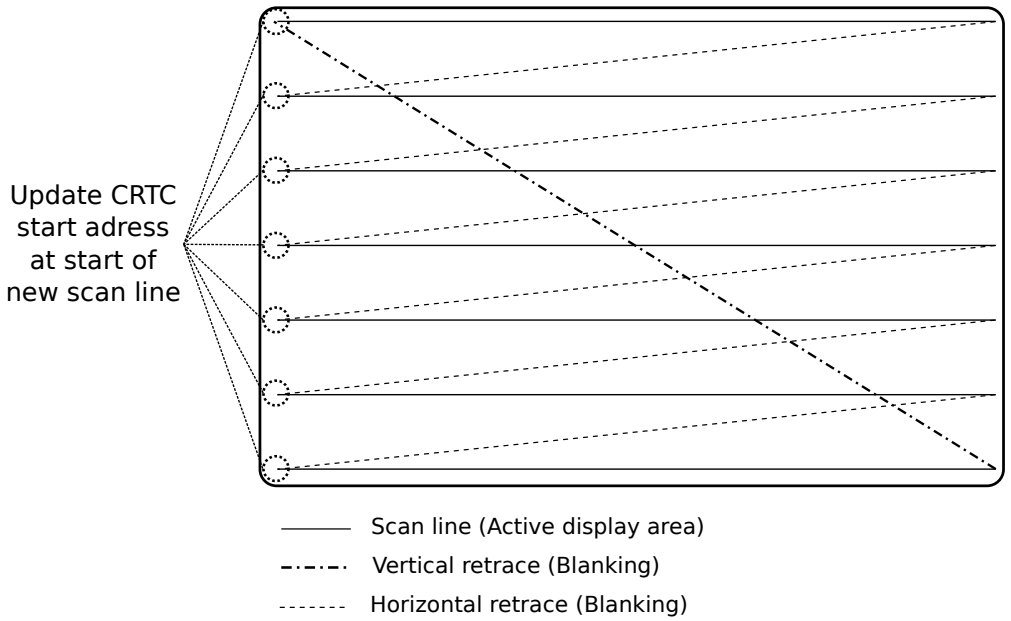


Figure 21: Update CRTC start address at beginning of new scan line.

To guarantee that the start address updates at the very beginning of a new scan line, the process first waits until the current scan line completes. Then, it waits for a full retrace, checking that the Display Enable Status returns to '0'. At this point, the CRTC address is updated.

Trivia : Regardless of CPU speed, the game's speed is limited by the CRTC sync to the monitor's refresh rate, which is 60Hz for EGA and 70Hz for VGA.

[7] Documentation is a bit unclear here. The IBM technical documentation for VGA explains retrace takes place when bit 0 of the Input Status Register 1 is set to high ('1'). The IBM technical EGA documentation explains the opposite, saying when bit 0 is set low ('0') a retrace is taking place. For now, we assume source code and VGA documentation is correct, retrace takes place on a '1'.

```
;==============
;
; VW_SetScreen
;
;==============

  mov dx,03DAh          ;Status Register 1
;
; wait util the CRTC just starts scaning a diplayed line
; to set the CRTC start
;
  cli

@@waitnodisplay:        ;wait until scan line is finished
  in  al,dx
  test  al,01b
  jz  @@waitnodisplay

@@waitdisplay:          ;wait until retrace is finished
  in  al,dx
  test  al,01b
  jnz @@waitdisplay

endif

;
; set CRTC start
;
  mov cx,[crtc]
  mov dx,CRTC_INDEX
  mov al,0ch    ;start address high register
  out dx,al
  inc dx
  mov al,ch
  out dx,al
  dec dx
  mov al,0dh    ;start address low register
  out dx,al
  mov al,cl
  inc dx
  out dx,al
```

0.2.5 Manage Refresh Timing

After each screen refresh, a certain amount of time, referred to as "ticks", has passed. The tick count depends on several factors, such as the number of tiles refreshed and the waiting period for a screen's vertical retrace. Since all game actions and reactions rely on the tick interval between two refreshes, it is important to keep this interval consistent.

Without controlling the tick interval, the state and speed of actors can become unpredictable, potentially causing them to move too quickly or even "warp" to an unexpected location. To manage refresh intervals effectively, a minimum and maximum number of ticks are defined within the refresh loop.

```
#define MINTICS         2
#define MAXTICS         6

void RF_Refresh (void)
{
    [...]

//
// calculate tics since last refresh for adaptive timing
//
    do
    {
        newtime = TimeCount;
        tics = newtime-lasttimecount;
    } while (tics<MINTICS);
    lasttimecount = newtime;

    if (tics>MAXTICS)
    {
        TimeCount -= (tics-MAXTICS);
        tics = MAXTICS;
    }
}
```

0.3 Actors and AI

0.3.1 State Machine

All objects in the game, such as Commander Keen, enemies, bonus points, doors, and even the scoreboard, are called "actors". Each actor can "think" and perform actions like walking, shooting, or emitting sounds. Actors are controlled via a state machine, enabling them to take actions such as chasing the player, running in various directions, throwing objects, or doing nothing at all. To model their behavior, all enemies have an associated state, which can include:

- Chasing Keen
- Hitting or smashing Keen
- Shooting projectiles
- Climbing and sliding on poles
- Turning into a flower
- Special Boss (Boobus)

Each state has associated think, reaction, and contact method pointers. Additionally, there is a `tictime` and `*nextstate` pointer, which indicate when the actor should transition to another state after a specific number of tics have passed in the current state.

```
typedef struct
{
  int     leftshapenum,rightshapenum; // Sprite to render
                                      // on screen
  enum    {step,slide,think,stepthink,slidethink} progress;
  boolean skippable;
  boolean pushtofloor;  // Make sure sprites stays
                        // connected with ground
  int tictime;          // How long stay in that state
  int xmove;
  int ymove;
  void (*think) ();
  void (*contact) ();
  void (*react) ();
  void *nextstate;
} statetype;
```

Each actor has a defined state chain, as example the pea pod.

```
statetype s_peapodwalk1 = {PEAPODRUNL1SPR,PEAPODRUNR1SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk2};
statetype s_peapodwalk2 = {PEAPODRUNL2SPR,PEAPODRUNR2SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk3};
statetype s_peapodwalk3 = {PEAPODRUNL3SPR,PEAPODRUNR3SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk4};
statetype s_peapodwalk4 = {PEAPODRUNL4SPR,PEAPODRUNR4SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk1};

statetype s_peapodspit1 = {PEAPODSPITLSPR,PEAPODSPITRSPR,step,false, true
    ,30, 0,0, SpitPeaBrain, NULL, DrawReact, &s_peapodspit2};
statetype s_peapodspit2 = {PEAPODSPITLSPR,PEAPODSPITRSPR,step,false, true
    ,30, 0,0, NULL, NULL, DrawReact, &s_peapodwalk1};
```

Different types of enemies have their own state machines, often sharing reaction function (e.g., WalkReact and ProjectileReact) but usually possessing their own unique "thinking" states.

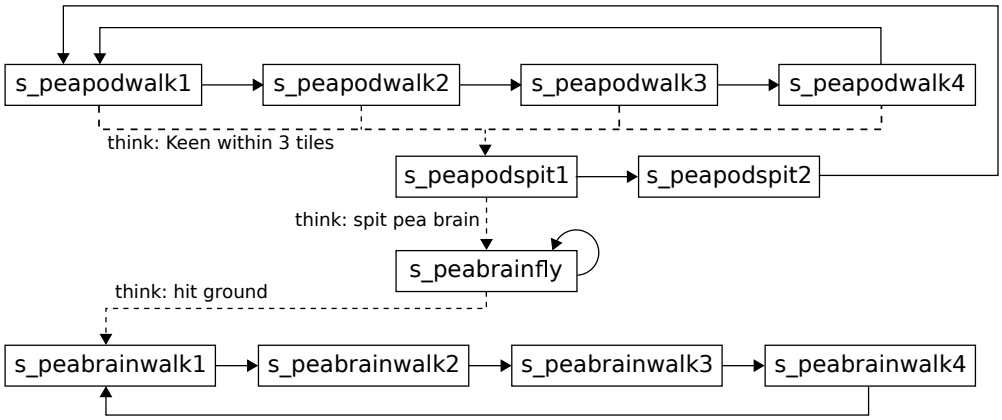


Figure 22: State machine for Pea pod and Pea brain.

The `*react` function is responsible for managing how an enemy reacts to the level, like turning around upon hitting a wall or the edge of a platform. The `*think` function defines how an enemy behaves when Commander Keen is nearby (e.g., attacking or firing a projectile) or when it reaches an edge (e.g. jumping). In some cases it introduces randomness, like when a pea pod might decide to spit a pea brain.

```
void PeaPodThink (objtype *ob)
{
  if ( abs(ob->y - player->y) > 3*TILEGLOBAL )
    return;

  if (player->x < ob->x && ob->xdir == 1)
    return;

  if (player->x > ob->x && ob->xdir == -1)
    return;

  // Randomness to spit pea brain
  if (US_RndT()<8 && ob->temp1 < MAXPEASPIT)
  {
    ob->temp1 ++;
    ob->state = &s_peapodspit1;
    ob->xmove = 0;
  }
}
```

The `*contact` function checks if an object has come into contact with another object and defines the resulting interaction, such as Commander Keen taking damage or losing a life.

0.3.2 Clipping

Whether an actor can move or fall through a tile is determined by the `tinf[]` parameters. Each foreground tile includes four directional parameters: north, south, east, and west. If, for example, the east parameter has a value greater than 0, the tile is considered solid, and the actor cannot enter it from the east side. When a sprite moves to the left and encounters a solid tile from the east, the engine adjusts the actor's movement to prevent it from entering the tile.

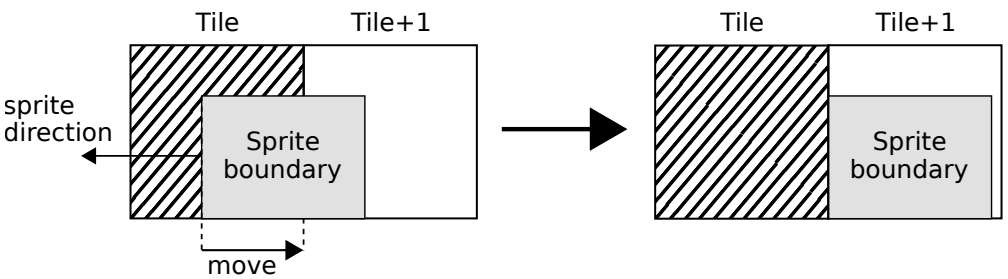


Figure 23: Clipping to east wall when actor moves left.

```
void ClipToEastWalls (objtype *ob)
{
    ...

  for (y=top;y<=bottom;y++)
  {
    map = (unsigned far *)mapsegs[1] +
      mapbwidthtable[y]/2 + ob->tileleft;

    //Check if we hit EAST wall
    if (ob->hiteast = tinf[EASTWALL+*map])
    {
      //Clip left side actor to left side
      //of next right tile
      move = ( (ob->tileleft+1)<<G_T_SHIFT ) - ob->left;
      MoveObjHoriz (ob,move);
      return;
    }
  }
}
```

For clipping along the top and bottom of tiles, the engine also accounts for standing on slopes. After the actor is clipped to the top or bottom of a slope tile, an offset is applied to move the actor up or down along the slope.

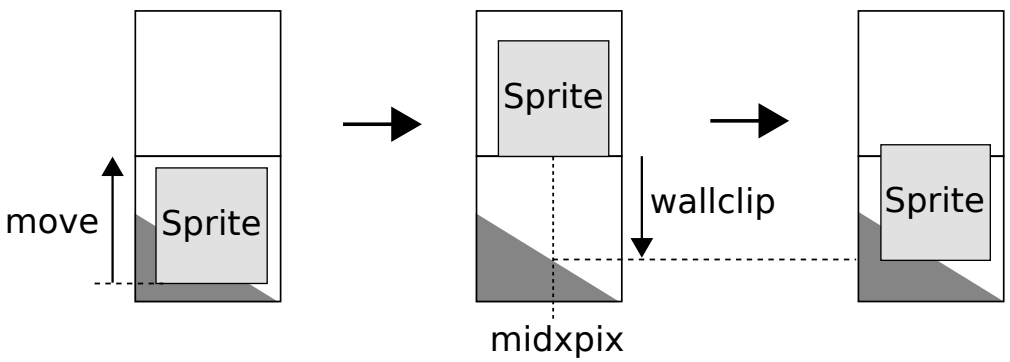


Figure 24: Clipping north wall with slope.

This offset is defined by the lookup table `wallclip[8][16]`, which uses the actor's midpoint and the wall type to determine the slope type.

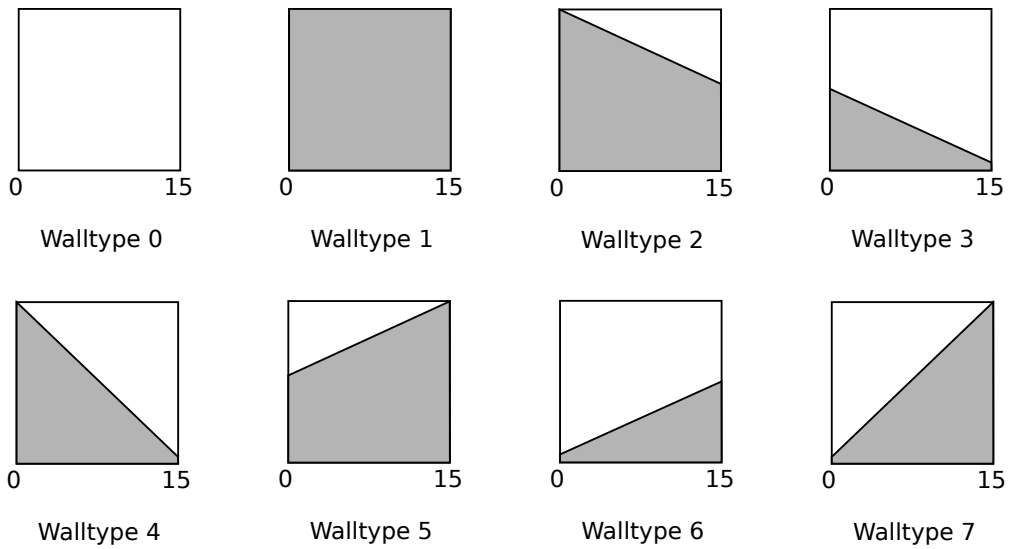


Figure 25: Fall through, solid and 6 slope types.

```
void ClipToEnds (objtype *ob)
{
  [...]
  //Get midpoint of sprite [0-15]
  midxpix = (ob->midx&0xf0) >> 4;
  for (y=oldtilebottom-1 ; y<=ob->tilebottom ; y++,map+=
   mapwidth)
  {
    //Do we hit a NORTH wall
    if (wall = tinf[NORTHWALL+*map])
    {
      //offset from tile border clip
      clip = wallclip[wall&7][midxpix];
      //Clip bottom side actor to top side tile + offset-1
      move = ( (y<<G_T_SHIFT)+clip - 1) - ob->bottom;
      if (move<0 && move>=maxmove)
      {
        ob->hitnorth = wall;
        MoveObjVert (ob,move);
        return;
      }
    }
  }
```

0.4 Drawing layer for layer

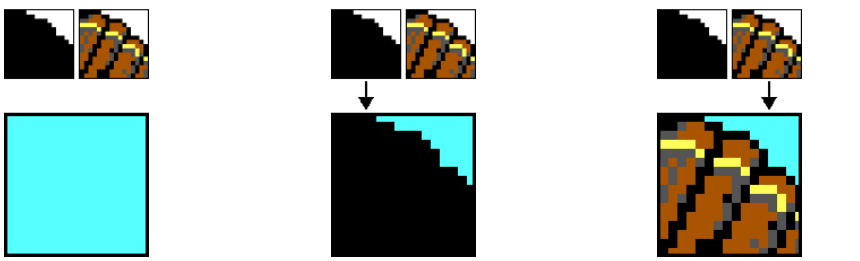
Each tile on the screen can contain up to three layers: the background layer, the foreground layer and a sprite layer. Rendering the final screen requires multiple redraws on the same tile to achieve the correct layering:

1. Draw the background layer or a combined background and foreground tile.
2. Draw the sprites.
3. Re-draw the foreground layer if a sprite should not appear on top.

It must run quickly and therefore most of the code is written in assembly.

0.4.1 Draw background and foreground tiles

Drawing the background layer is straightforward, as it only requires copying 128 bytes to VRAM. Drawing foreground tiles on top of the background layer requires an additional mask, which defines which pixels are overwritten by the foreground tile.



(a) Background tile. ***(b)*** `AND` foreground mask. ***(c)*** `OR` foreground bitmap.

Figure 26: Draw masked foreground tile.

Combining the background and foreground layers involves an `AND`-bitwise operator to clear the background and the `OR`-bitwise operator to write the foreground layer.

```
PUBLIC  RFL_NewTile

[...]
  es,[mapsegs+2]          ;foreground plane
  mov bx,[es:si]
  mov es,[mapsegs]        ;background plane
  mov si,[es:si]

[...]
  or  bx,bx               ;do we have foreground tile?
  jz  @@singletile        ;draw background tile only
  jmp @@maskeddraw        ;draw both together

[...]
@@maskeddraw:
  shl bx,1
  mov ss,[grsegs+STARTTILE16M*2+bx]
  shl si,1
  mov ds,[grsegs+STARTTILE16*2+si]

  xor si,si               ;first word of tile data

  mov ax,SC_MAPMASK+0001b*256 ;map mask for plane 0

  mov di,[cs:screenstartcs]

@@planeloopm:
  WORDOUT
tileofs = 0
lineoffset  = 0
REPT  16
  mov bx,[si+tileofs]     ;background tile
  and bx,[ss:tileofs]     ;mask
  or  bx,[ss:si+tileofs+32] ;masked data
  mov [es:di+lineoffset],bx
tileofs   = tileofs + 2
lineoffset  = lineoffset + SCREENWIDTH
ENDM
```

0.4.2 Drawing sprites

The next step is to render sprites on the screen. Most home computers of that era had built-in sprite functionality on the video card. For example, on a MSX computer, one could simply enter

```
PUT SPRITE <SpriteNumber>, <X>, <Y>, Color
```

to display a sprite on the screen. Updating the `(X, Y)` coordinates would move the sprite, with the display adapter handling everything else. Unfortunately, the concept of sprites did not exist on EGA cards, so game developers had to implement their own solution.

A challenge arises from the fact that sprites can move freely across the screen and are not byte-aligned. To address this, the bit-shifting technique described in section **??** is used. When caching a sprite into memory, each sprite is copied four times, with each copy shifted by two or more pixels using this technique. The property `*spr->shifts` determines the number of bit shifts applied to each of the four copies.

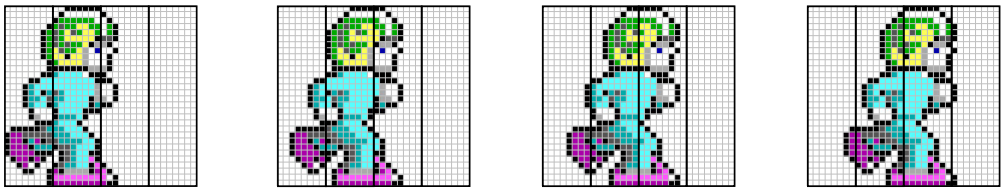


Figure 27: Sprite shifted in 4 steps.

Displaying the correct shifted sprite is as simple as

```
#define G_P_SHIFT   4 // global >> ?? = pixels

//Set x,y to top-left corner of sprite
y+=spr->orgy>>G_P_SHIFT;
x+=spr->orgx>>G_P_SHIFT;

shift = (x&7)/2;  // Set sprite shift
```

```
void CAL_CacheSprite (int chunk, char far *compressed)
{
[...]
//
// make the shifts!
//
  switch (spr->shifts)
  {
  case  1:  // no shifts
    [...]

  case  2:  // one shift of 4 pixels
    for (i=0;i<2;i++)
    {
      dest->sourceoffset[i] = shiftstarts[0];
      dest->planesize[i] = smallplane;
      dest->width[i] = spr->width;
    }
    for (i=2;i<4;i++)
    {
      dest->sourceoffset[i] = shiftstarts[1];
      dest->planesize[i] = bigplane;
      dest->width[i] = spr->width+1;
    }
    CAL_ShiftSprite ((unsigned)grsegs[chunk],dest->
   sourceoffset[0],
      dest->sourceoffset[2],spr->width,spr->height,4);
    break;

  case  4:  // four shifts of 2 pixels
    [...]
  default:
    Quit ("CAL_CacheSprite: Bad shifts number!");
  }
}
```

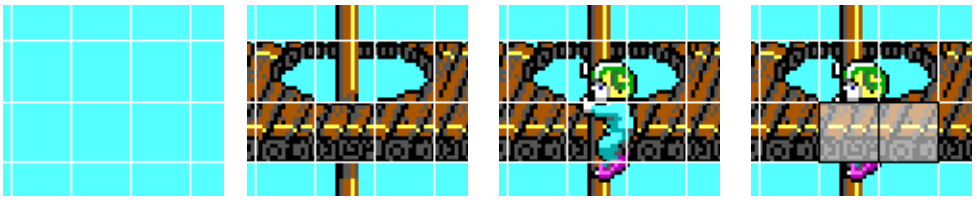
If multiple sprites are displayed on the same tile, each sprite is assigned a priority from 0 to 3 to determine the drawing order. A sprite with a higher priority number is always drawn on top of sprites with a lower priority. Since sprites are always drawn on top of tiles, this can create unnatural situations, such as when Commander Keen is climbing through a hole, as illustrated in Figure 28.

(a) Background tile. ***(b)*** Foreground tile. ***(c)*** Sprite on top.

Figure 28: Unnatural situation where Commander Keen is in front of a hole.

To draw sprites 'inside' a foreground tile, a trick is used by introducing a priority foreground tile in the `tinf` info table. The attribute is named INTILE ("in-front"). If a foreground tile has its INTILE attribute high bit set (`80h`), sprites with priority 0-2 will not be drawn over it. Only sprites with priority 3 are drawn over the foreground tile. This results in the following drawing order:

1. Draw the background tile and the masked foreground tile.
2. Draw sprites with priority 0, 1, and 2 (in that order), and mark the corresponding tile in the tile buffer array with a '3', as illustrated in Figure **??** on page **??**.
3. Scan the tile buffer array for tiles marked with '3'. If the corresponding foreground tile's INTILE attribute high bit (`80h`) is set, re-draw the foreground tile.
4. Finally, draw sprites with priority 3. These sprites are always drawn on top of everything.



(a) Background tile. ***(b)*** Foreground tile. ***(c)*** Sprite on top. ***(d)*** Redraw foreground tile.

Figure 29: Draw sprite inside a tile, by redrawing foreground tile.

```
PROC  RFL_MaskForegroundTiles
PUBLIC  RFL_MaskForegroundTiles

[...]

@@realstart:
  mov di,[updateptr]
  mov bp,(TILESWIDE+1)*TILESHIGH+2
  add bp,di          ; when di = bx,
  push di            ; all tiles have been scanned
  mov cx,-1          ; definately scan the entire thing
;=============
; scan for a 3 in the update list
;=============
@@findtile:
  mov ax,ss
  mov es,ax          ; scan in the data segment
  mov al,3           ; check for tiles marked as '3's
  pop di             ; place to continue scanning from
  repne scasb
  cmp di,bp
  je  @@done
;=============
; found a tile, see if it needs to be masked on
;=============
  push  di
  sub di,[updateptr]
  shl di,1
  mov si,[updatemapofs-2+di]  ; offset from originmap
  add si,[originmap]
  mov es,[mapsegs+2]      ; foreground map plane segment
  mov si,[es:si]          ; foreground tile number
  or  si,si
  jz  @@findtile          ; 0 = no foreground tile
  mov bx,si
  add bx,INTILE           ; INTILE tile info table
  mov es,[tinf]
  test [BYTE PTR es:bx],80h ; high bit = masked tile
  jz  @@findtile

; mask the tile
```

0.4.3 Tile Draw Performance Tricks

To draw one background tile to VRAM, the engine must read and write 128 bytes (2 x 16 bytes x 4 memory banks), which is the minimum number of read/write operations required. In the worst case, up to 512 bytes of read/write operations are required (background, foreground, sprite, and foreground again), with multiple bitwise operations involved. The engine employs several tricks to squeeze the maximum out of each CPU cycle: background tile caching and word-aligned memory writing.

Background Tile Caching

When updating the master screen in VRAM, the engine copies each tile from RAM to VRAM. Copying each pixel involves one read and one write operation across the four memory banks. As explained in section 0.2.2, by reprogramming the latches on the EGA card, it is possible to copy four bytes at once from one VRAM location to another. The same technique can be applied to tiles containing only a background layer.

Once a background tile is loaded into the master screen, subsequent requests for the same tile can be handled by copying it directly from VRAM using the reprogrammed latches, rather than copying from RAM. The engine only needs to track which background tiles are already loaded into VRAM using an array.

```
unsigned  tilecache[NUMTILE16];
```

The assembly function `RFL_NewTile` is responsible for drawing tiles to the master screen. It first checks whether the background tile is already stored in the tile cache array. If it is, the tile is copied at 32 bits per cycle from VRAM to VRAM. Otherwise, the tile is loaded from RAM, and the VRAM pointer for that tile is stored in the tile cache array for future use.

```
PROC  RFL_NewTile updateoffset:WORD
  [...]
  mov ax,[tilecache+si]
  or  ax,ax
  jz  @@singlemain    ; if 0, tile not in cache
;=============
; Draw single tile from cache
;=============
  [...]
  ret
;=============
; Draw single tile from main memory
;=============
@@singlemain:
  mov ax,[cs:screenstartcs]
  mov [tilecache+si],ax    ;next time it can be drawn from
   here with latch
```

Word-aligned memory writing

The 286 CPU can read and write 16 bits in a single cycle, but there is a caveat: this only works when accessing even memory addresses. Additionally, writing a word at the offset address `FFFFh` causes an exception.

Because tiles are word-aligned (16 bits wide) and the screen refreshes in tile-sized steps, they are always aligned with even memory addresses. However, sprites are byte-aligned. When a sprite is drawn from an odd memory address, the CPU is limited to reading and writing 8 bits at a time. To fully utilize the 16-bit data bus, the engine uses small function routines for each combination of sprite width and even/odd address alignment.

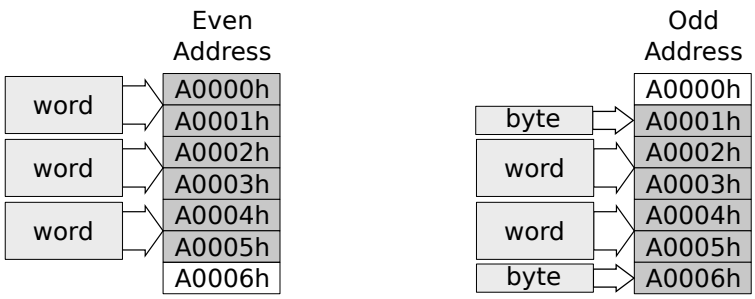


Figure 30: Word-optimized 6 byte wide sprite to even (`MASK6E`) and odd (`MASK6O`) address.

In total 23 functions are written to optimize `word`-size writing up to 10 bytes wide sprites. Each function pointer reference is stored in an array.

```
maskroutines
  dw   mask0,mask0,mask1E,mask1E,mask2E,mask20,mask3E,mask30
  dw   mask4E,mask40,mask5E,mask50,mask6E,mask60
  dw   mask7E,mask70,mask8E,mask80,mask9E,mask90
  dw   mask10E,mask100
```

The engine first determines whether the destination is at an even or odd memory address. If the address is odd, it first writes a single byte to align the subsequent operations to an even address, after which it continues writing in 16-bit words. By cleverly using bit-shift operations and the Carry Flag (CF), the engine calls the appropriate function to write the sprite to VRAM.

```
PROC  VW_MaskBlock  segm:WORD, ofs:WORD, dest:WORD, wide:
   WORD, height:WORD, planesize:WORD

[...]

@@unwoundroutine:
  mov cx,[dest]
  shr cx,1
  rcl di,1          ;shift a 1 in if destination is odd
  shl di,1          ;to index into a word width table
  mov ax,[maskroutines+di]  ;call the right routine
  mov [routinetouse],ax   ;and store the function pointer

@@startloop:
  mov ds,[segm]

@@drawplane:
  [...]
  mov si,[ofs]        ;start back at the top of the mask
  mov di,[dest]       ;start at same place in all planes
  mov cx,[height]       ;scan lines to draw
  mov dx,[ss:linedelta]

  jmp [ss:routinetouse]   ;draw one plane
```