0.1 Action Phase: Smooth scrolling

After the player is done setting up the game, it is time for the scrolling engine to shine. On bitmapped displays without hardware scrolling like the EGA card, the entire screen have to be erased and redrawn in the slightly shifted position whenever the player moved in any direction. This would kill the CPU as you need to update all pixels of all four planes on the EGA card.

So here John Carmack came with a smart solution. The scrolling engine is based on a simple yet powerful technology called *Adaptive Tile Refreshment*. The core idea is to refresh only those areas on the screen that needed to change.

The screen is divided into tiles of 16x16 pixels. On a screen with 320x200 pixels, it means a grid of 20x13 tiles (actually it is 12.5 tiles high, but we round to full tiles). Let's look at *Commander Keen 1: Marooned on Mars* in Figure 1. 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 moved 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 see which tiles needs to be changed upon screen refresh. The trick behind the scrolling in the first Commander Keen games was to only redraw tiles that actually changed after panning 16 pixels (one tile), since most maps had large swathes of constant background. In case of Figure 1 only 69 tiles of the total 260 tiles need to be refreshed, which is 27% of the screen!

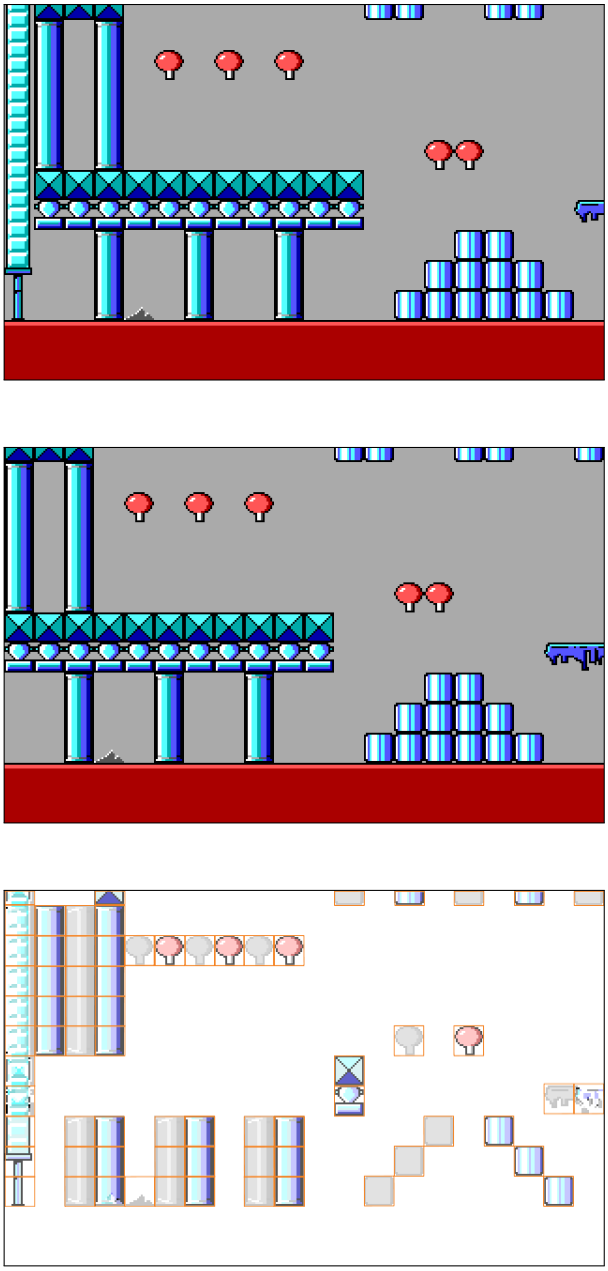


Figure 1: Start of the world, moved one tile to the right and difference.

So now we know how to scroll the screen in steps of 16 pixels, which is still pretty 'choppy'. For smooth scrolling we need to dive deeper into the EGA card, which is explained in the next section.

0.1.1 EGA Virtual Screen

The EGA adds a powerful twist to linear addressing: the logical width of the virtual screen in VRAM memory need not to be the same as the physical width of the screen display. The programmer is free to define a logical screen width of up to 4096 pixels and then use the physical screen as a window onto any part of the virtual screen. What's more, a virtual screen can have any logical height up to the capacity of the VRAM memory. The code below illustrates how to change the 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 area of the virtual screen displayed at any given time is selected by setting the display memory address at which to begin fetching video data. This is set by way of the CRTC Start Address register. The default address is `A000:0000h`, but the offset can be changed to any other number. In EGA's planar graphics modes, the eight bits in each byte of video RAM correspond to eight consecutive pixels on-screen. Panning down a scan line requires only that the start address is increased by the logical width in bytes. Horizontal panning is possible by increasing the start address by one byte, although in this case only relative coarse of 8 pixels (1 byte) adjustments are supported. See the code below how to set the CRTC Start Address register.

```
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
```

0.1.2 Horizontal Pel Panning

Smooth pixel scrolling of the screen is provided by the Horizontal Pel Panning register in the Attribute Controller (ATC). Up to 7 pixels' worth of single pixel panning of the displayed image to the left is performed by increasing the register from 0 to 7.

There is one annoying quirk about programming the Attribute Controller: when the ATC Index register is set, only the lower five bits (bits 0-4) are used as the 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 ouput of the palette RAM controls the displayed pixels; this is normal operation. When bit 5 is 0, video data doesn't come from the palette RAM, and the screen becomes a solid color. To ensure the ATC index register is restored to normal video, we must set bit 5 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
```

0.1.3 Smooth scrolling: Bring it all together

Now we know how to perform tile refresh and smooth scrolling, it is time to bring it all together. The game and all actors are defined in a global coordinate system, which is scaled to 16 times a pixel. The higher resolution enables more precision of movements and better simulation of movement acceleration. Conversion between global, pixel and tile coordinate systems can be easily performed by bit shift operations:

- From global to pixel is shifing 4 bits to right.
- From pixel to tile is shifting 4 bits to right.
- From global to tile is shifting 8 bits to right.

The idea is to first perform all actions and movements in the global coordinate system, and then translate back to pixel or tile coordinate system for video updates.

```
#define G_T_SHIFT 8   // global >> ?? = tile
#define G_P_SHIFT 4   // global >> ?? = pixels
#define SY_T_SHIFT 4  // screen y >> ?? = tile

void RFL_CalcOriginStuff (long x, long y)
{
  originxglobal = x;
  originyglobal = y;
  originxtile = originxglobal>>G_T_SHIFT;
  originytile = originyglobal>>G_T_SHIFT;
  originxscreen = originxtile<<SX_T_SHIFT;
  originyscreen = originytile<<SY_T_SHIFT;
  originmap = mapbwidthtable[originytile] + originxtile*2;

  //panning 0-15 pixels
  panx = (originxglobal>>G_P_SHIFT) & 15;
  //pan pixels 0-7 (0) or 8-15 (1)
  pansx = panx & 8;
  pany = pansy = (originyglobal>>G_P_SHIFT) & 15;
  //Start location in VRAM
  panadjust = panx/8 + ylookup[pany];
}
```

So the smooth horizontal and vertical panning should be viewed as a series 16-pixel tile refreshment and fine adjustments in the 8-pixel range. The scrolling is defined by the following steps, see also Figure 2:

- Calculate the panning in pixels for both x- and y-direction
- The y-panning is defined by adding `logical width * y` to the CRTC start address
- In case the panning in x-direction is more than 8 pixels, increase the CRTC start address by 1 byte. This is where we need `pansx`.
- the remaining pixels, ranging from 0-7, will be adjusted using horizontal pel panning

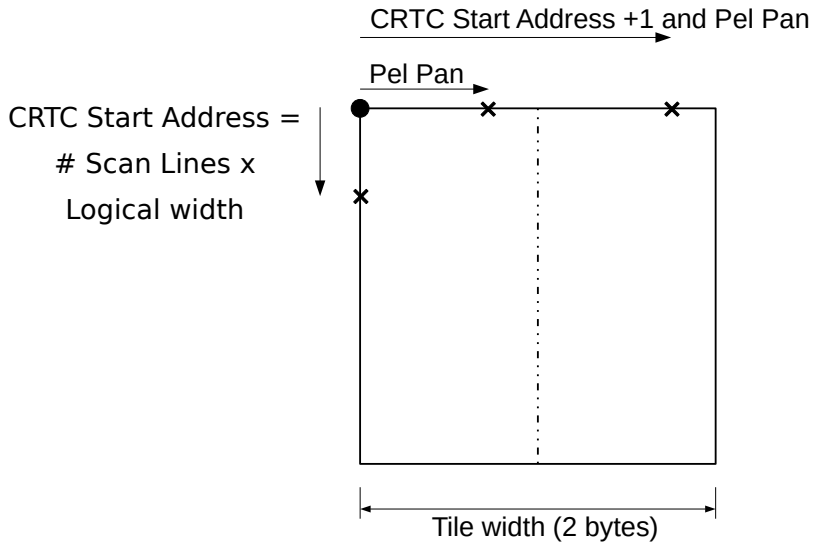


Figure 2: Smooth scrolling in EGA.

0.2 Virtual screen buffer

Even if the screen is not scrolling, tile refreshes are required to support sprite and tile animations. Since moving a sprite in this way involves first erasing it and then redrawing it, the image of the erased sprite may be visible briefly, causing flicker. This is where double buffering comes in: setting up a second buffer into which the code can draw 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.

Now, let's have a closer look at the EGA memory setup. In the file `id_vw.h` the virtual screen in VRAM is defined by SCREENSPACE, which is set to 512x240 pixels (64x240 = 15,360 bytes). This is more than sufficient since the visible screen in mode `0x0D` is 320x200 pixels.

Since one screen only uses 15,360 bytes of VRAM (which is 3,840 bytes per plane), there is more than enough space to store more than two full screens of video data. The video memory is organized into three virtual screens:

- Page 0 and 1, which are used to switch between buffer and visble screen
- A master page containing a static page, which is copied to the buffer memory when performing the screen refresh.

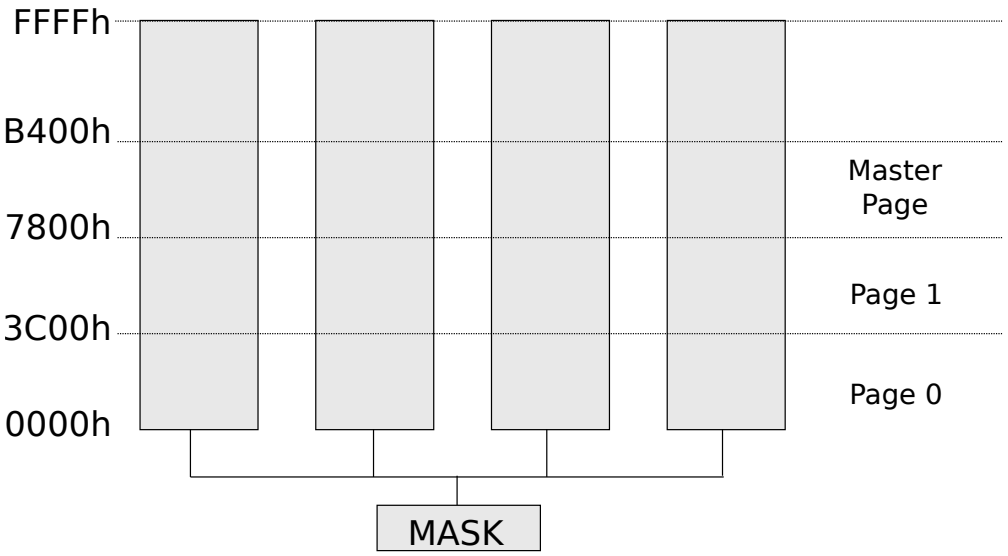


Figure 3: Virtual screen layout on EGA card.

The page that is actually displayed at any given time is selected by setting the CRTC Start Address register at which to begin fetching video data.

0.3 Adaptive Tile Refreshment

The approach of refreshing the screen is different between the first Commander Keen games, Commander Keen 1-3, and the ones after. In the first games the algoritm keeps the view and buffer screen at fixed VRAM locations, where it performs a check which tiles are changed after the scroll. In the later games, it makes use of the moving the VRAM location and add a full row or column at the beginning or end of the view port.

0.3.1 Tile buffer and tile view layout

Before explaining the scrolling algoritm, let's first explain how the view port and buffer layout are setup. The visible viewing screen on EGA has a resolution of 320x200 pixels. Translated in 16x16 pixel tiles, the screen view has a size of 20x13 tiles. By making the view port one tile higher and wider than the screen, the engine can scroll the screen up to 16 pixels to the right or bottom side of the screen without any tile refresh, by means of adjusting the CRTC Start Address and Pel Pan registers. Finally, the buffer must have enough space to float the view port up to two tiles in all direction. At the end of Section

0.3.4 it is explained why we need a spare buffer of 2 tiles.

So summarized, as illustrated in Figure 4, the following tile views are defined:

- Screen View size of 20x13 tiles and Port View size of 21x14 tiles.
- Buffer screen size of 22x14 tiles. This is one tile wider than the Port View, where the additional tile is used to mark a '0' at the end of each tile row.
- Total tile buffer is the buffer screen plus two times a spare buffer to support floating the buffer screen two tiles in any direction.

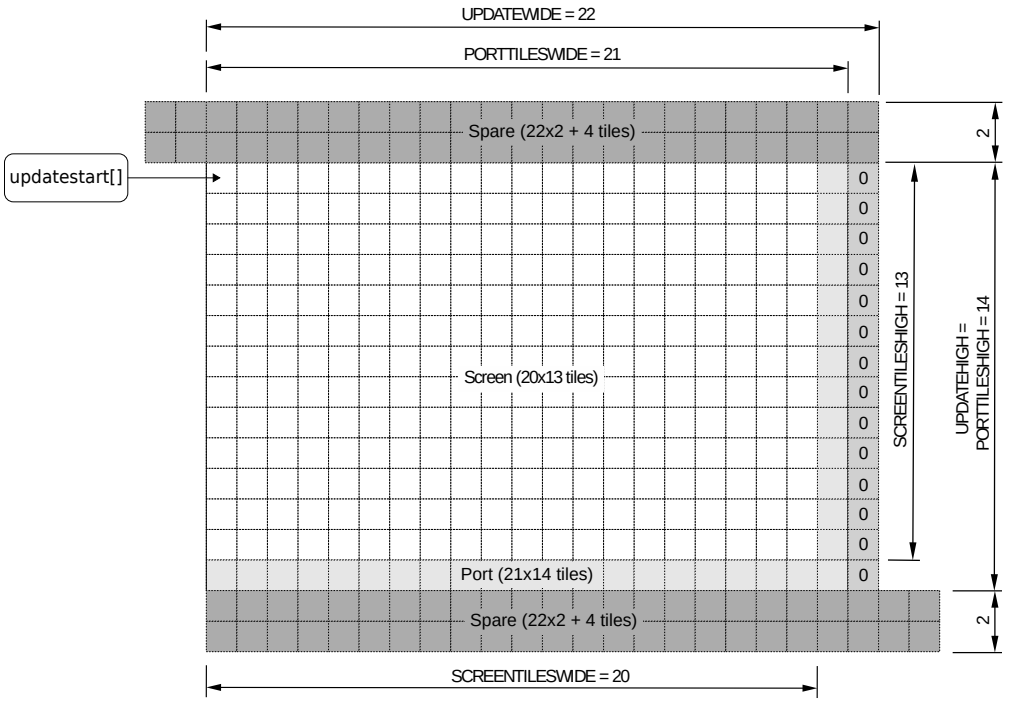


Figure 4: Tile view and tile buffer layout.

To keep track of the tile refresh process, the following variables are defined in the game:

- `screenstart[]` are pointers to the starting address (upper-left pixel) of the viewports in VRAM. As explained, we maintain three viewports in VRAM:
  - screenpage, the active displayed screen on the monitor. Note that the engine never works or updates the active screen.

  - otherpage, which is the buffer screen. This screen is updated and will be switched with the screenpage upon next refresh.
  - masterpage, which stores a complete static background, without sprites. This page is used to update the buffer screen.

- `updatestart[]` are pointers to the tile arrays. It maintains which tiles needs to be updated upon next refresh. There are two tile buffer arrays; one for the screenpage and one for otherpage.

0.3.2 Adaptive tile refreshment in Commander Keen 1-3

In the this section we explain how the first 3 versions of the game are working[1]. Six stages are involved in drawing a 2D scene:

1. Check if the player has moved one tile in any direction.
2. Validate which tiles have changed (both from scrolling and animated tiles), copy these respective tiles to the Master screen and mark the tiles in both tile arrays (view and buffer tile arrays).
3. Refresh the buffer screen by scanning all tiles. If a tile needs to be updated, copy the tile from the master screen to the buffer screen.
4. Iterate through the sprite removal list and copy corresponding image block from the master screen to buffer screen.
5. Iterate through the sprite list and copy corresponding sprite image block from asset location in RAM to buffer screen.
6. Switch the view and buffer screen by adjusting the CRTC Start Address and Pel Panning registers.

In the next six screenshots, we take you step-by-step through each of the stages. The player has moved and forces the screen to scroll one tile to the right.

[1] We can only explain how the algoritm is working without code examples, since the only released code is Keen Dreams which is using the improved algoritm.

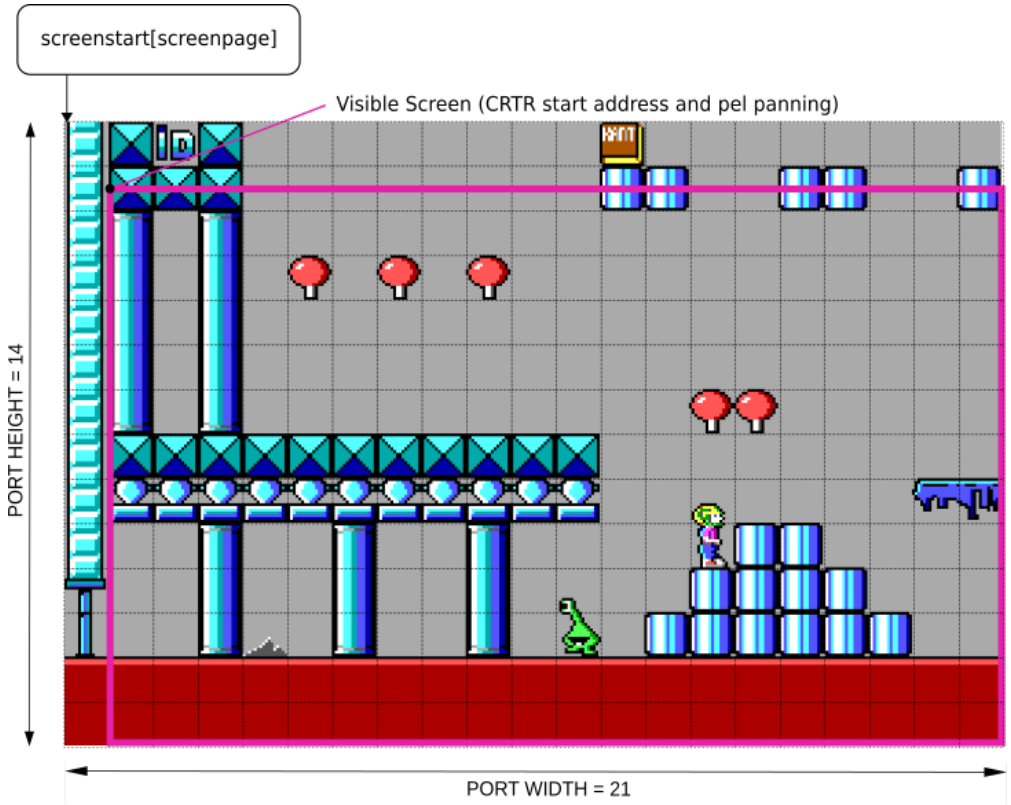


Figure 5: Step 1: Scroll screen to the right

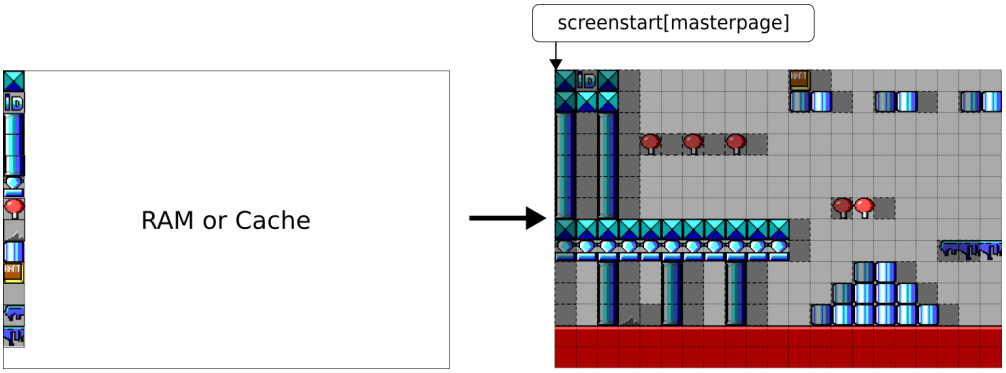


Figure 6: Step 2: Update changed tiles in masterscreen, marked in dark grey

Each tile of the buffer screen is compared with the corresponsing tile on the view screen. If the tile number has changed, the tile needs to be updated by copying tile data from the asset location into the corresponding location of the masterpage.

In parallel both the tile buffer and tile view array the changed tiles are marked with a '1', which means it needs to be updated upon next refresh.

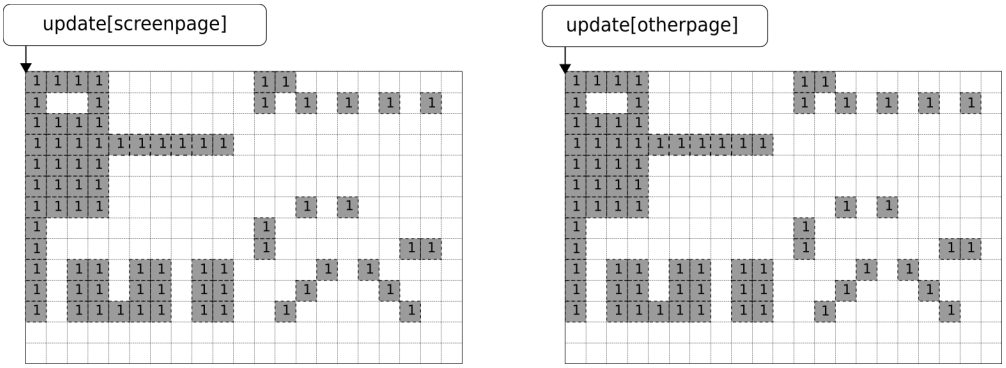


Figure 7: Mark all changed tiles with '1' in both tile buffer and tile view array.

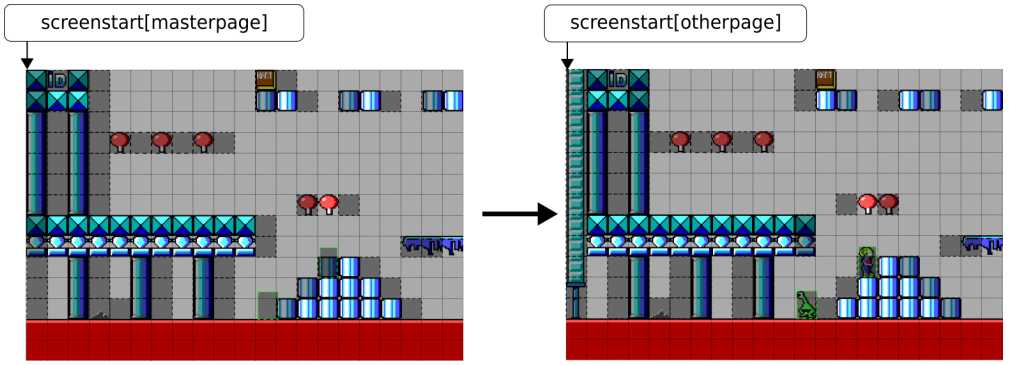


Figure 8: Step 3 and 4: Copy tiles from master to buffer screen and remove sprites

The next step is to scan all tiles in the tile buffer array and for each tile marked as '1', copy the tile from master to buffer screen.

If a sprite has moved, the previous sprite location is added to the block removal list. For each block in this removal list, erase the sprite by copying the width and height of the sprite block (marked in green in Figure 8) from the master screen to the buffer screen, and mark the corresponding tiles only in the tile buffer array with a '2'.

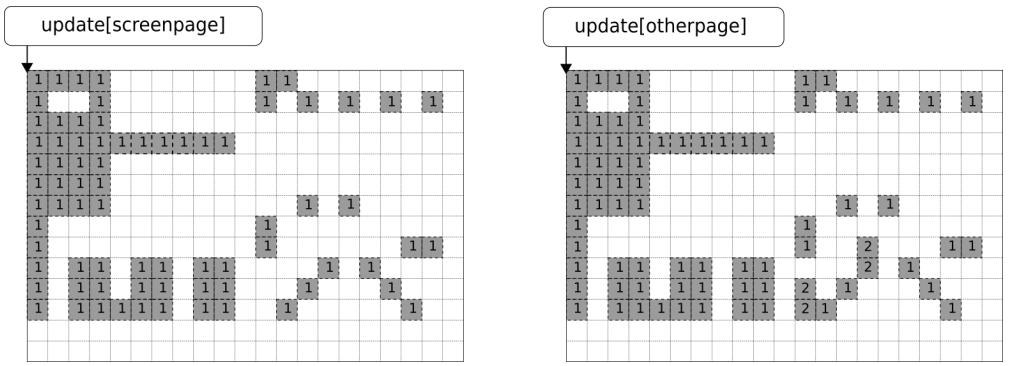


Figure 9: Mark removed sprites with '2' in tile buffer array only.

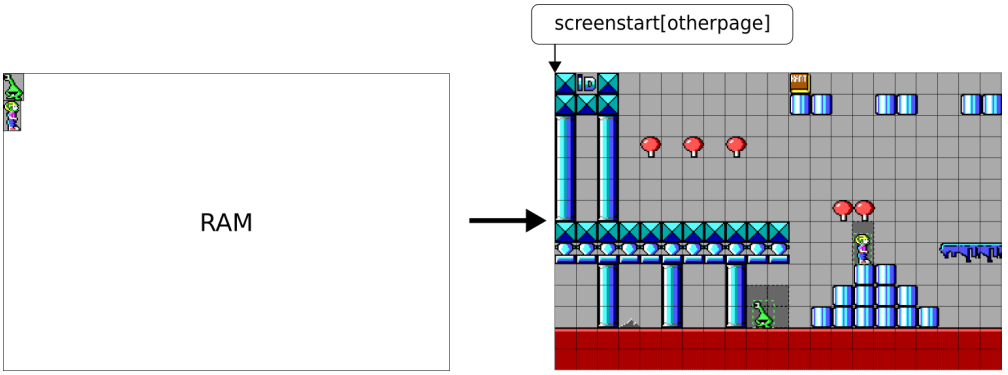


Figure 10: Step 5: Scan sprite list and copy sprite onto buffer screen

Next, the engine scans the sprite list. Validate if the sprite is in the visible part of the view port and copy the sprite image to the buffer screen. Mark the corresponding tiles in the tile buffer array with a '3'.

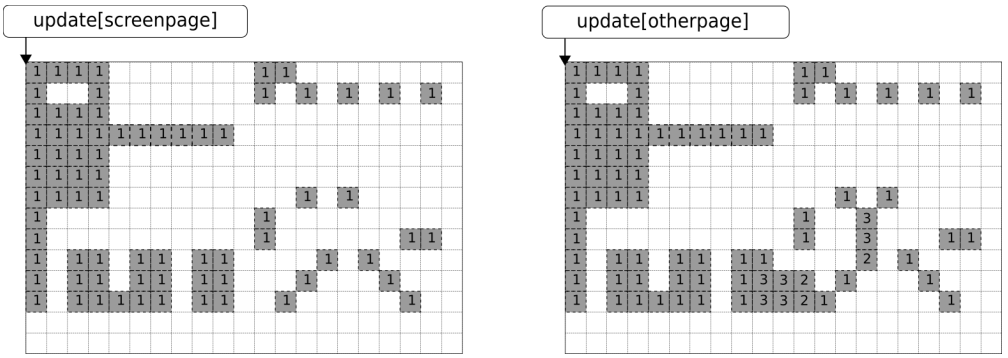


Figure 11: Mark new sprites locations with '3' in tile buffer array only.

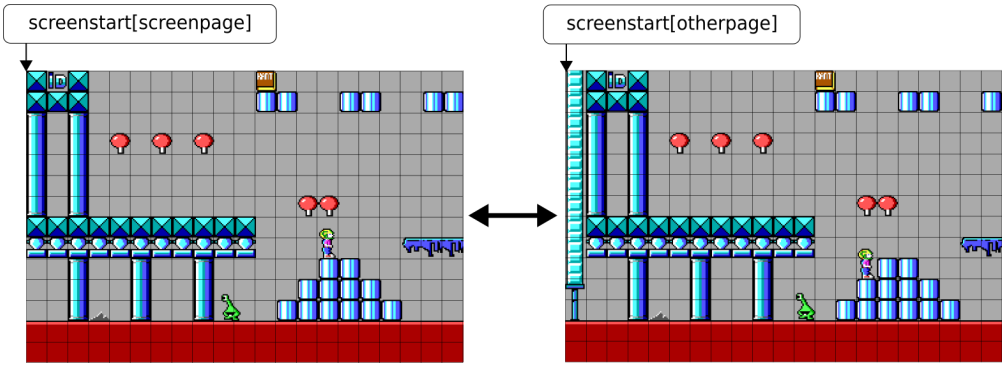


Figure 12: Step 6: Swap buffer and screen page

As the final step, point the visible screen to the buffer screen by updating the CRTR start address and horizontal Pel Panning register. The entire tile buffer array is then cleared to '0'. Finally the otherpage and screenpage of both the `screenstart[]` and `update[]` are swapped. Then step 1 is repeated.

Note that after swapping, the tile buffer array still has marked all tiles that have changed from scrolling the screen. This makes sense as the current buffer screen is not yet updated (it was displayed in the previous cycle, and we never update the view screen).

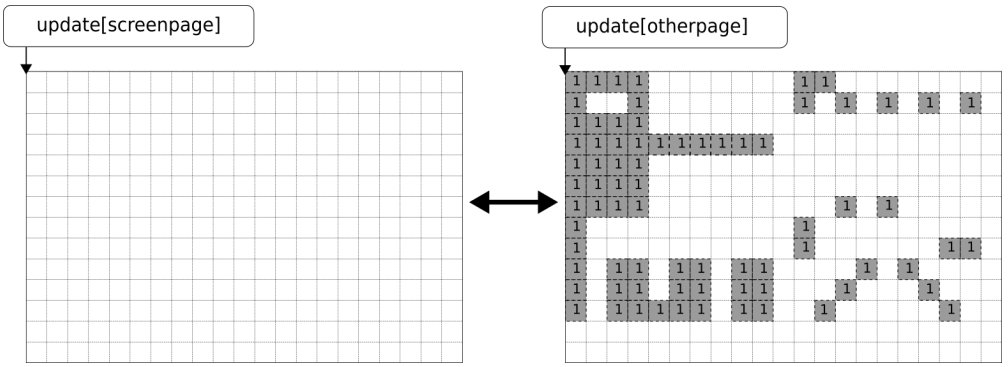


Figure 13: Clear tile update array and swap arrays.

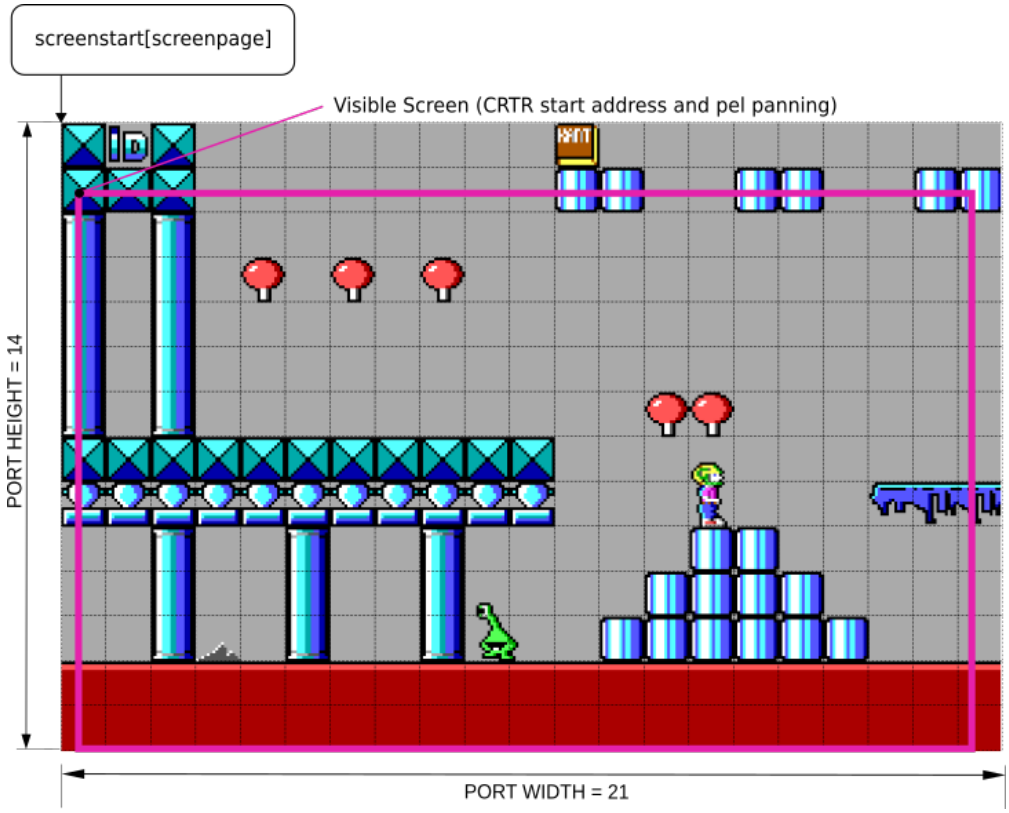


Figure 14: Step 6: Swap buffer and screen page

Step 2 and 3 (except for the animated tiles) only needs to happen if Commander Keen is moving more than 16 pixels, where step 4 and 5 normally needs to happen for each refresh. So the number of drawing operations required during each refresh is controllable by the level designer. If they choose to place large regions of identical tiles (the large swathes of constant background), less redrawing (meaning: less redrawing in step 2 and 3) is required.

0.3.3 Wrap around the EGA Memory

John Carmack explored what would happen if you push the virtual screen over de 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.

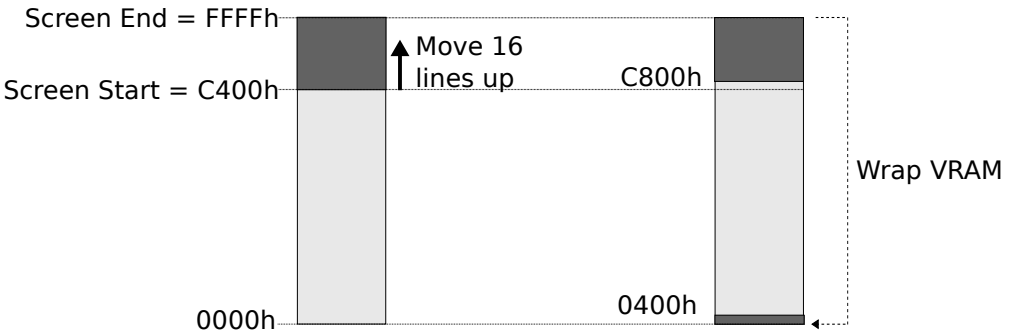


Figure 15: Wrap virtual screen around the EGA memory

There was however an issue with the introduction of Super VGA cards, which had typically more than 256kB RAM[2]. This resulted in crippled backwards compatibility and the wrapping around `0xFFFF` did not work anymore on these cards.

Luckily there was a way to resolve this issue. As you can see in Figure 3 on page 8, the space between `0xB400` and `0xFFFF` is not used and contains enough space for another virtual screen. Each screen buffer has a size of `0x3C00` in each memory bank. In case the start address is between `0xC400` and `0xFFFF` the corresponsing screen is copied to the opposite end of the buffer, as illustrated in Figure 16.

[2] 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.

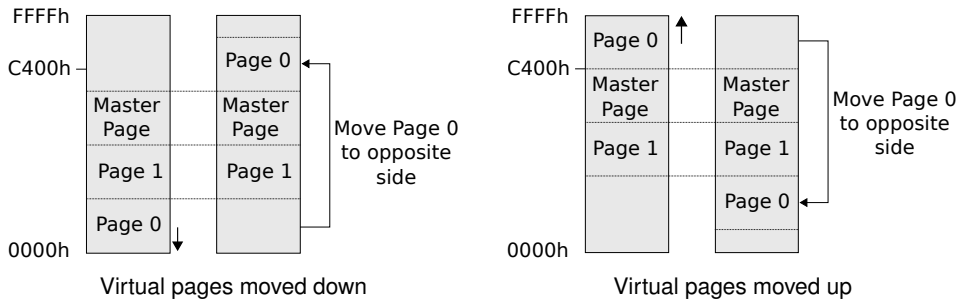


Figure 16: Move screen to opposite end of VRAM buffer

The code verifies if any of the virtual pages is between the addresses `0xC400` and `0xFFFF` and, if this is the case, copies the entire VRAM to the opposite side.

```
#define SCREENSPACE    (SCREENWIDTH*240)
#define FREEEGAMEM     (0x10000l-3l*SCREENSPACE)

screenmove = deltay*16*SCREENWIDTH + deltax*TILEWIDTH;
for (i=0;i<3;i++)
{
  screenstart[i]+= screenmove;
  if (compatability && screenstart[i] > (0x10000l-
   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);
  }
}
```

But how can we copy four VRAM planes fast enough, without noticing any performance hit?

As explained in Section **??** each pixel is encoded by four bits, which are spread across the four EGA banks. Since all write operations are one byte wide, it is not hard to imagine the difficulty in plotting a single pixel without changing the others stored in the same byte. One would have to do four read, four xor, and four writes.

Since the designers of the EGA were not complete sadists, they added some circuitry to simplify this operation. For each bank, they created a latch placed in front of a configurable ALU.

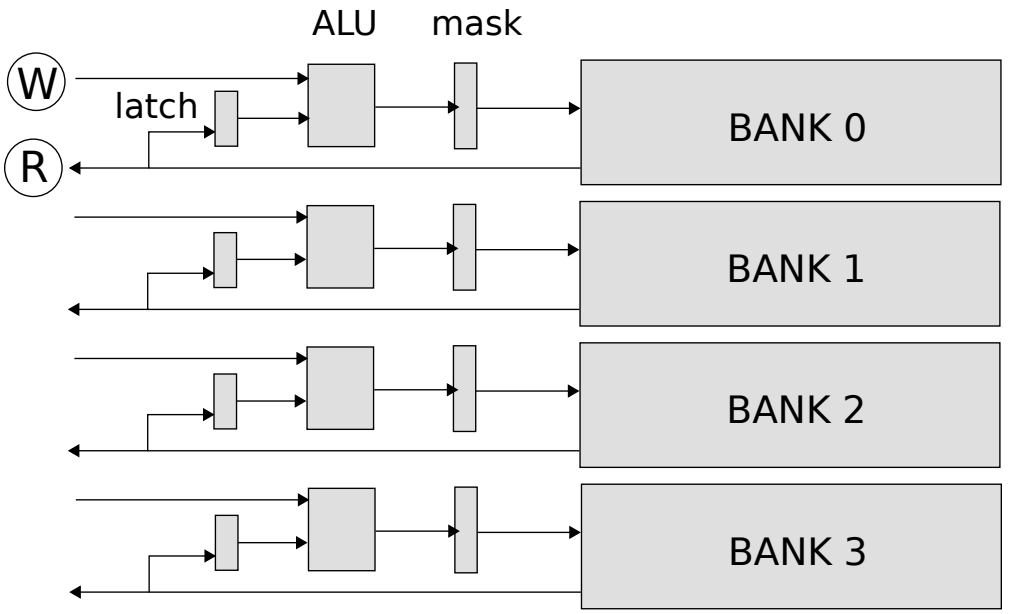


Figure 17: Latches memorize read operations from each bank. The memorized value can be used for later writes.

With this architecture, each time the VRAM is read (R), the latch from the corresponding bank is loaded with the read value. Each time a value is written to the VRAM (W), it can be composited by the ALU using the latched value and the written value. This design allowed mode `0Dh` programmers to plot a pixel easily with one read, one ALU setup, and one write instead of four reads, 4 xors, and 4 writes.

By getting a little creative, the circuitry can be re-purposed. The ALU in front of each bank can be setup to use only the latch for writing. 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. This system allows transfer from VRAM to VRAM 4 bytes at a time. Now

it is possible to copy the entire buffer fast enough, without notifying any peformance impact.

```
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
```

To take full advantage of this optimization, the refresh algoritm maintains a list of tiles that are already copied on the masterpage via `tilecache` variable. If a tile is already on the master screen the algoritm copies the tile from that location to its destination instead of the RAM location in memory, saving the four separated writes to each memory plane.

0.3.4 Scroll and screen refresh in Keen Dreams

The EGA memory wrapping results in the following improved algoritm to scroll and refresh the screen for Keen Dreams:

1. Check if the player has moved one tile in any direction.
2. In case the player moved one tile, move the `screenstart[]` pointers accordingly.
3. Copy the new introduced column or row of tiles to the Master screen and flag this new column/row of tiles to be updated in the next refresh for both pages.
4. Refresh the buffer screen by scanning all tiles in the tile buffer array. If a tile is flagged for update, copy the tile from the master screen to the buffer screen.

5. Iterate through the sprite removal list and copy corresponding image block from master screen to buffer screen.

6. Iterate through the sprite list and copy corresponding sprite image block from asset location in RAM to buffer screen.

7. Switch the view screen and buffer screen by adjusting the CRTC Start Address and Pel Panning register.

As you can see, step 2 to 4 are different from Commander Keen 1-3, the rest of the steps are the same. In the example below the screen is forced to scroll to the left.

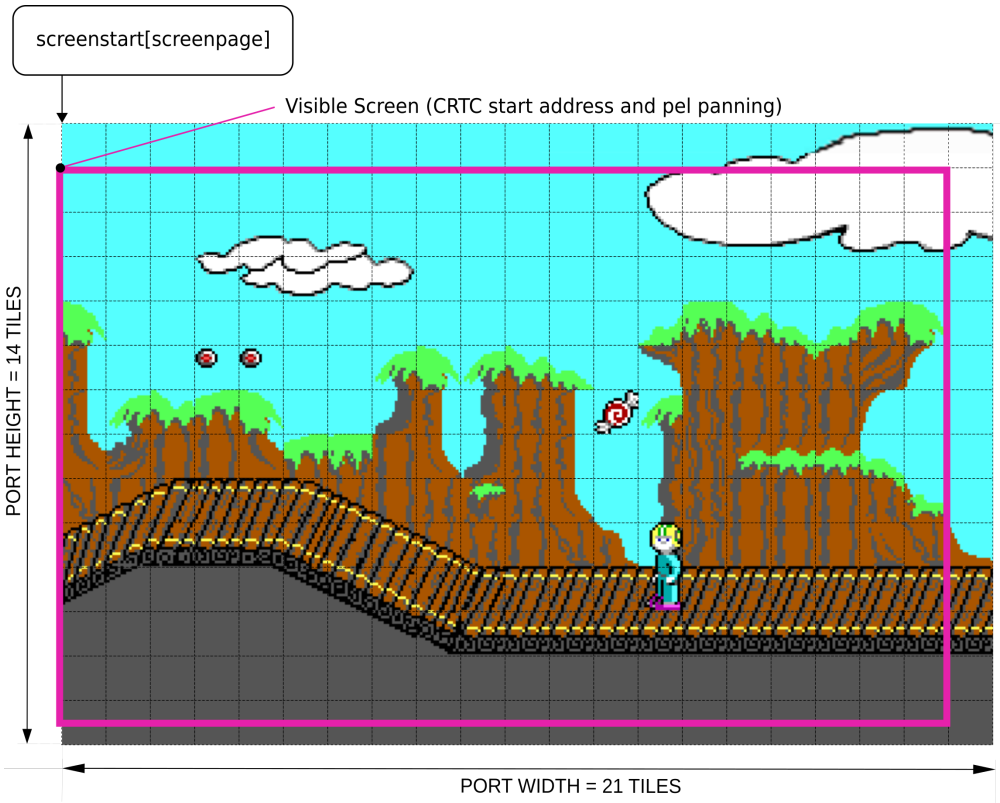


Figure 18: Step 1: Scroll screen to the left

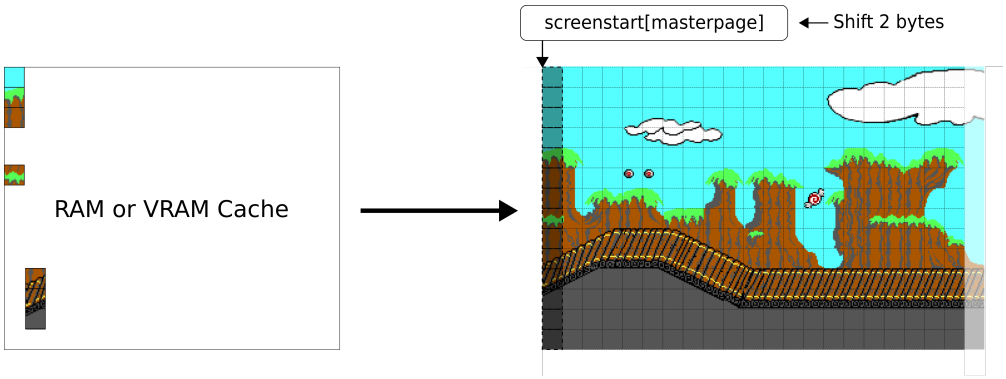


Figure 19: Step 2 and 3: Shift screen pointer and add column to VRAM

First decrease the `screenstart[]` of all three screen locations 2 bytes (1 tile). Then copy a left-column of tiles from the asset or cache location into the corresponding location of the masterscreen.

In parallel decrease both the tile buffer and tile display array pointers one byte and mark each tile on the left border in both buffer arrays with '1', so it is updated upon the next refresh. Finally, the most right column (which is now outside the view port) is marked with a '0'.

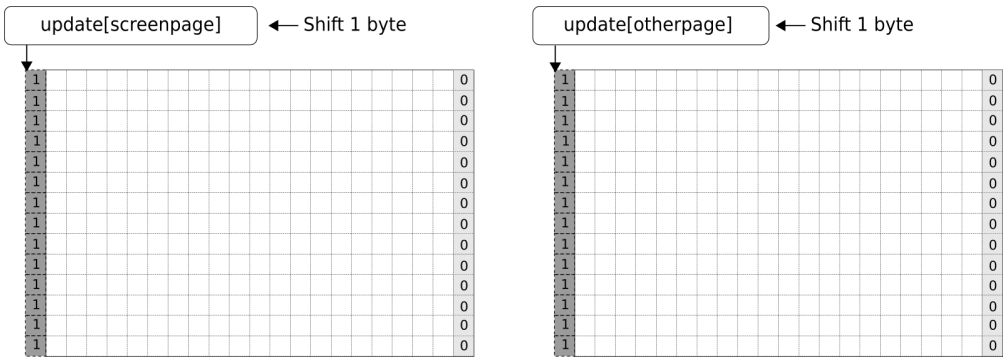


Figure 20: Mark new column in both tile buffer and display array.

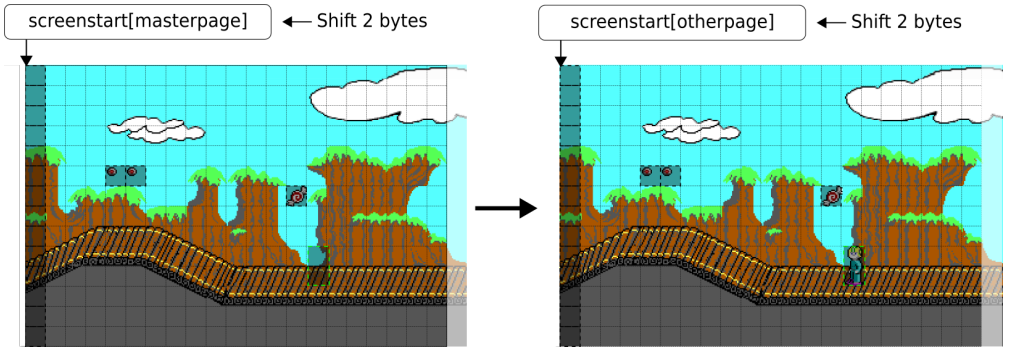


Figure 21: Step 4 and 5: Copy master to buffer screen and remove sprites

Just like before, we copy the animated tiles from asset or cache location to master screen and mark them as '1' in both tile arays. Then we scan all '1' and copy those tiles from master to buffer screen. Finally, we remove sprites by copying the removal block from master to buffer screen and mark the corresponding tiles with '2' in the tile buffer array. The result is shown in Figure 21 and Figure 22.

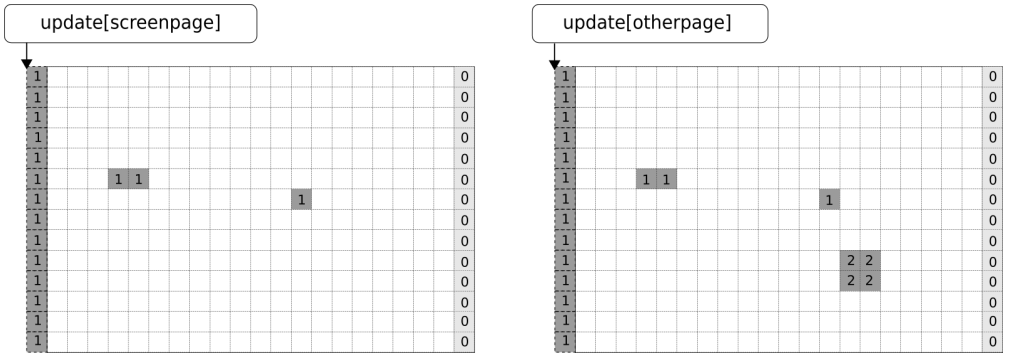


Figure 22: Tile buffer array with removed sprites marked with '2'.

Trivia : The removal blocks which are marked '2' in the tile buffer array are nowhere used in the engine.

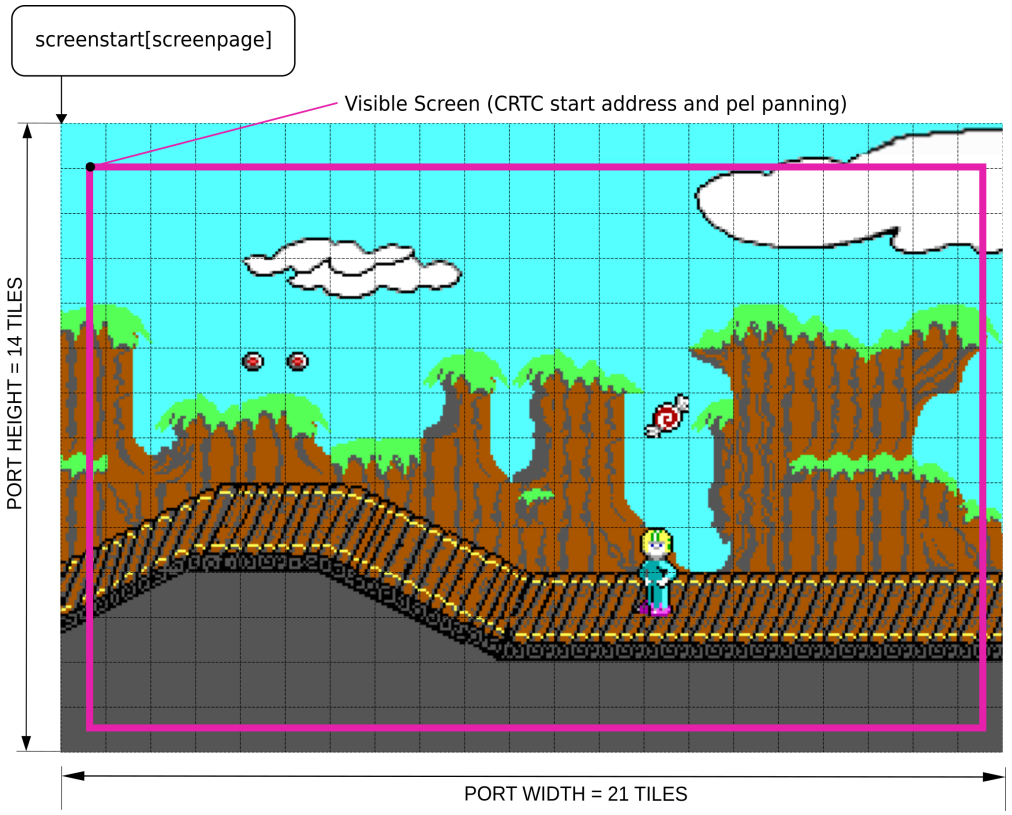


Figure 23: Step 7: Updated screen after swapping buffer and screen page

The remaining steps are the same as before, meaning putting the sprites on the buffer screen and finally swap both the buffer and screen page. Since we only need to update one border, the engine needed to update only 6% of the screen!

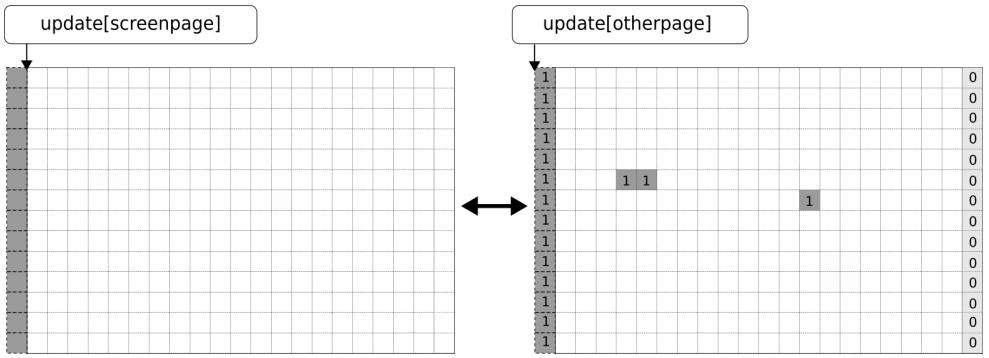


Figure 24: Clear and reset tile buffer array and swap arrays.

Now it is also clear why the tile buffer and view arrays are 2 tiles wider on all sides than the tile view port (see Figure 4). Let's take the situation where the screen scrolls to the top-left, meaning 1 tile left and 1 tile up as illustrated in Figure 25. Both `*updatestart` pointers are updated and tiles are marked as '1'. After completing all tile refresh steps, the buffer screen is updated on places where the tile buffer array is marked '1'.

After the visible screen is swapped with the buffer screen, the `*updatestart[otherpage]` pointer is cleared and the pointer is resetted. However, the `*updatestart[screenpage]` is not cleared nor resetted since we did not update the screenpage (we only updated the buffer screen).

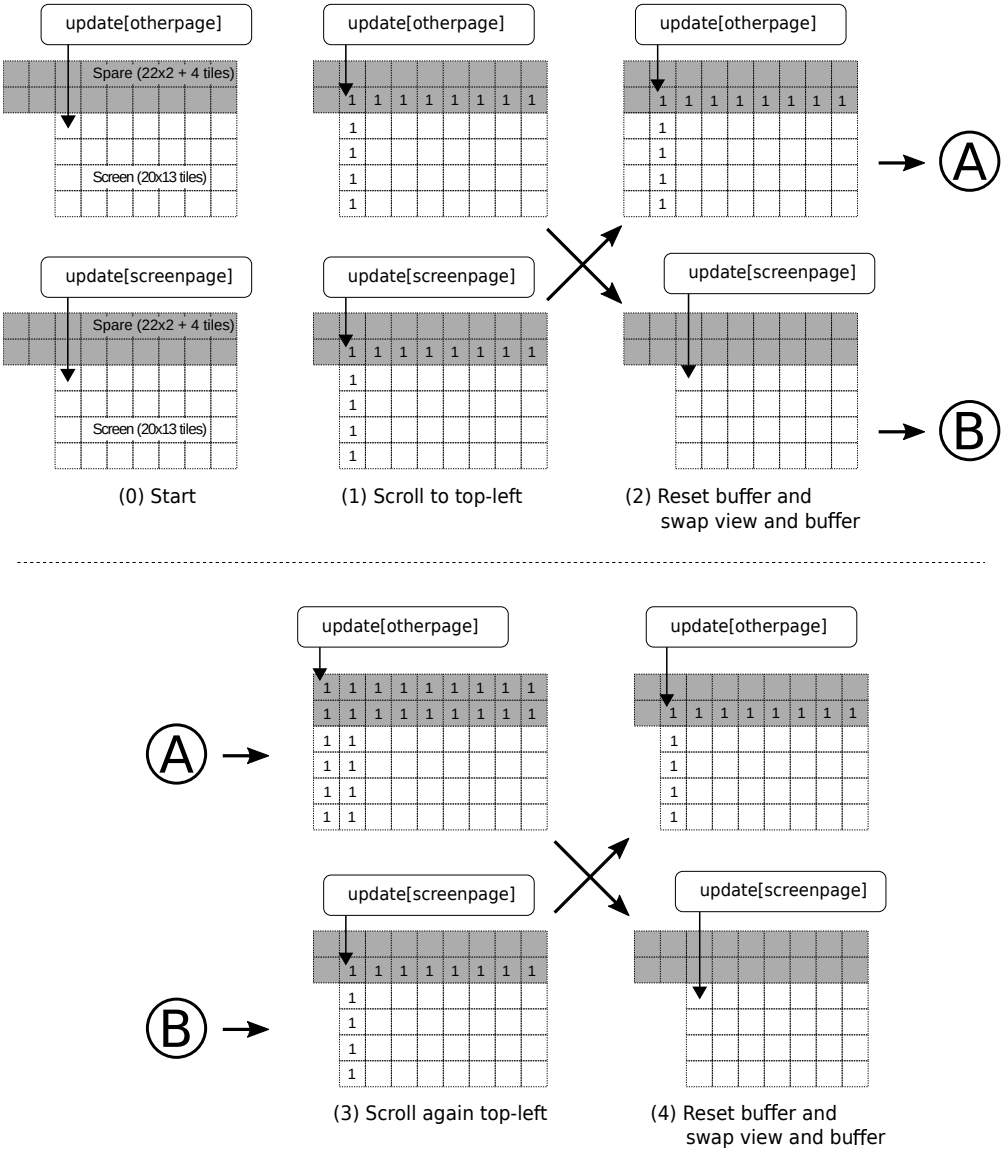
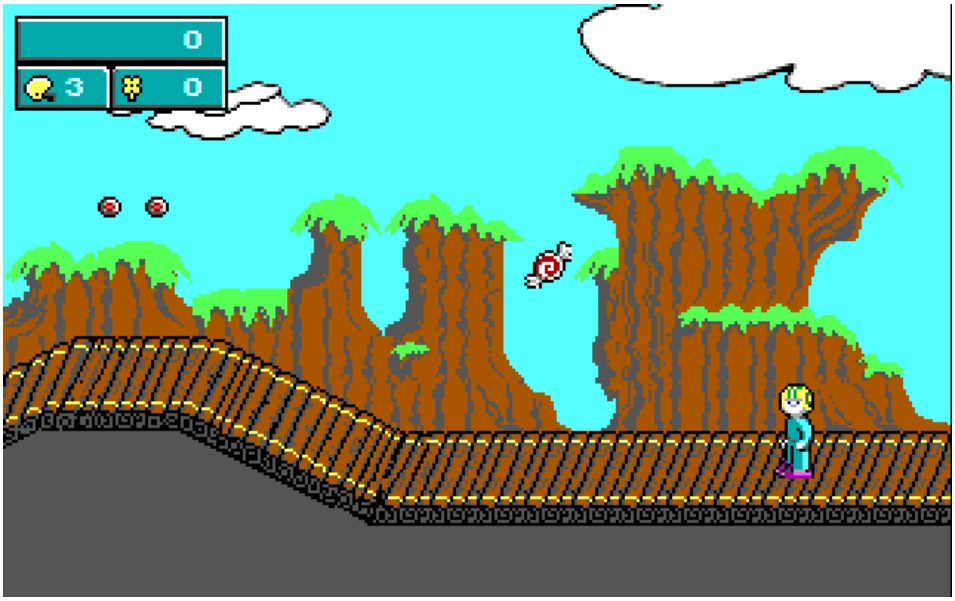


Figure 25: scroll to top-left tile

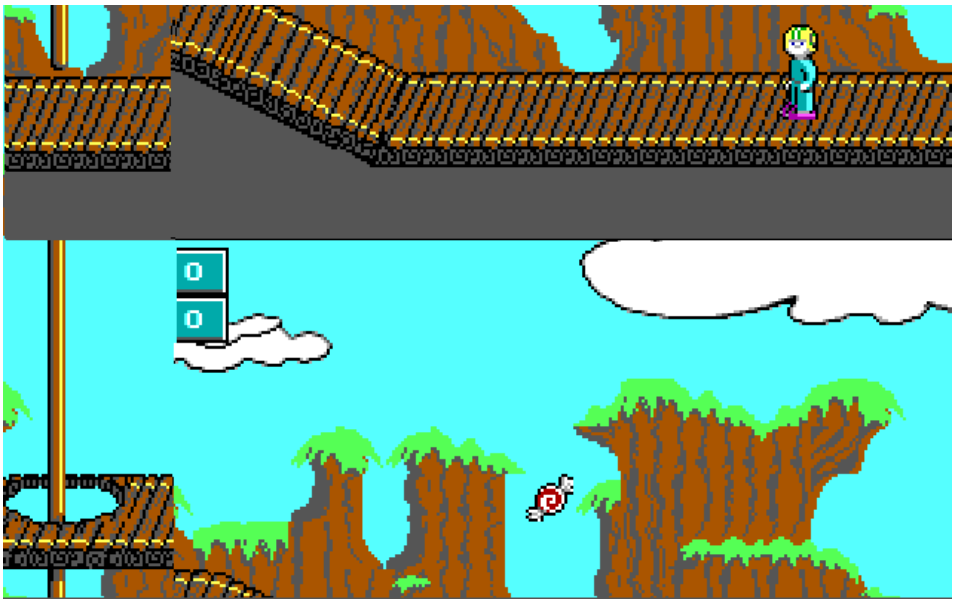
Now, if the screen is again scrolling to the top-left one can see why there is a need for the 2nd row in the buffer. After this cycle the `*update[otherpage]` pointer is cleared and resetted as shown in the last illustration of Figure 25.

0.3.5 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:



This glitch shows both misalignment and parts of two pages. This problem has to do with the timing between updating the CRTC starting address and screen refresh. The start address is latched by the EGA's internal circuitry exactly once per frame, typically at the start of the vertical retracement. The CRTC starting address is a 16-bit value but the `out` instruction can only write 8 bits at a time.

Now we have the following situation, where the current CRTC start address (Page 0) is pointing to `0x0000` and the buffer (Page 1) to `0x3C00`. We moved one tile to the left and now Page 0 is pointing at 0xFFFE in VRAM and Page 1 is at 0x3BFE. Page 1 is the updated buffer and will be displayed upon next refresh cycle. Poor timing of the vertical retracement and start address update results in the CRTC only picking up the first byte of the address, `0x3B`, and setting the new start address to `0x3B00` instead of `0x3BFE`.

```
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
```

The most obvious solution to this isue is to update the start address when we pick up the vertical retracement signal via the Input Status 1 Register (bit 3 of `0x3DA`). Unfortunately, by the time the vertical retrace status is observed by a program, the start address for the next frame has already been latched, having happened the instant the vertical retrace pulse began.

The trick is to update the start address sufficient far away from when the vertical retracement starts. So we're looking for a signal that tells us it just finished a horizontal or vertical retrace and started a scan line, far enough away from vertical retrace so we can be sure the new start address will get used at the next vertical sync. This signal is provided by the Display Enable status signal via the Input Status 1 Register, where a value of 1 indicates the display is in a horizontal or vertical retracement[3].

[3]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'.

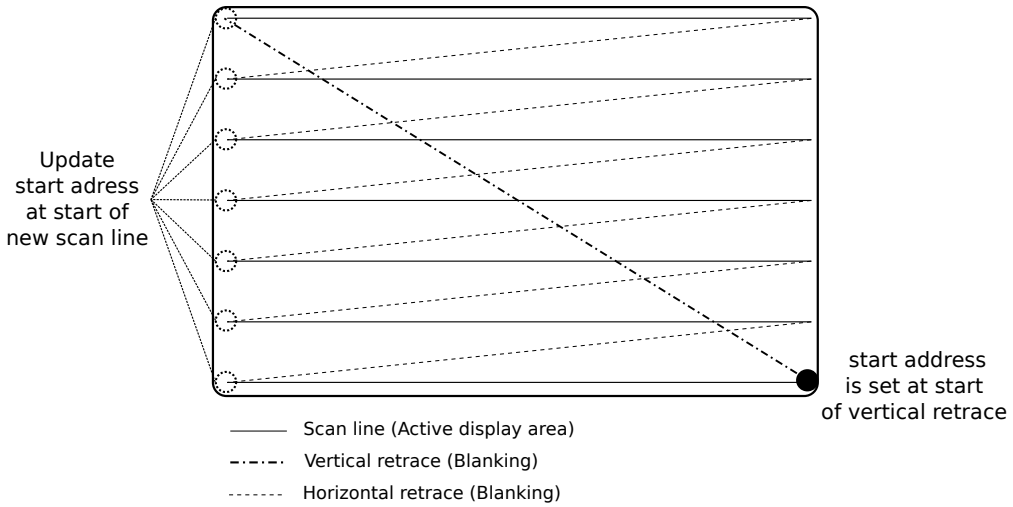


Figure 26: Update CRTC start address at beginning of new scan line.

Once the Display Enable status is observed, the program sets the new start address. Once the CRTC start address is updated, the game waits for vertical retrace to happen, sets the new pel panning state, and then continues drawing.

```
;==============
;
; 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 address

;
; wait for a vertical retrace to set pel panning
;
  mov dx,STATUS_REGISTER_1
@@waitvbl:
  sti                   ;service interrupts
  jmp $+2
  cli
  in  al,dx
  test  al,00001000b  ;look for vertical retrace
  jz  @@waitvbl

endif

; #### set horizontal panning
```

0.4 Actors and sprites

0.4.1 A.I.

To simulate enemies, some objects are allowed to "think" and take actions like walking, shooting or emitting sounds. These thinking objects are called "actors". Actors are programmed via a state machine. They can be aggressive (chase you), just running in any direction, or dump (throwing things at you). To model their behavior, all enemies have an associated state:

- Chase Keen
- Smash Keen
- Shoot projectile
- Climb and slide from pole
- Walking around
- Turn into flower
- Special Boss (Boobus)

Each state has associated think, reaction and contact method pointers. There is also a next pointer to indicate which state the actor should transition to when the current state is completed.

```
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;
```

All actors have a state chain, as example the tater trooper.

```
statetype s_taterwalk1 = {TATERTROOPWALKL1SPR,TATERTROOPWALKR1SPR,step,
    false, true,10, 128,0, TaterThink, NULL, WalkReact, &s_taterwalk2};
statetype s_taterwalk2 = {TATERTROOPWALKL2SPR,TATERTROOPWALKR2SPR,step,
    false, true,10, 128,0, TaterThink, NULL, WalkReact, &s_taterwalk3};
statetype s_taterwalk3 = {TATERTROOPWALKL3SPR,TATERTROOPWALKR3SPR,step,
    false, true,10, 128,0, TaterThink, NULL, WalkReact, &s_taterwalk4};
statetype s_taterwalk4 = {TATERTROOPWALKL4SPR,TATERTROOPWALKR4SPR,step,
    false, true,10, 128,0, TaterThink, NULL, WalkReact, &s_taterwalk1};

statetype s_taterattack1 = {TATERTROOPLUNGEL1SPR,TATERTROOPLUNGER1SPR,
    step,false, false,12, 0,0, NULL, NULL, BackupReact, &s_taterattack2};
statetype s_taterattack2 = {TATERTROOPLUNGEL2SPR,TATERTROOPLUNGER2SPR,
    step,false, false,20, 0,0, NULL, NULL, DrawReact, &s_taterattack3};
statetype s_taterattack3 = {TATERTROOPLUNGEL1SPR,TATERTROOPLUNGER1SPR,
    step,false, false,8, 0,0, NULL, NULL, DrawReact, &s_taterwalk1};
```

All types of enemies (including Boobus) have their own state machine. They often share the same reactions (e.g. WalkReact and ProjectileReact), but often have their own thinking state.

0.4.2 Drawing Sprites

Once the state of the actor is updated, it is time to render the actor on the screen. This is done using sprites and contains the following steps:

1. Update the state and move actors within the active region.
2. Determinate if a actor has changed or moved
3. Update the actor by removing and drawing sprites to it's new position

Unlike many game consoles such as Nintendo, the concept of sprites did not exists on the EGA card, so again the team needs to write their own solution. As explained in Section **??** (Page 34, table 1) each sprite asset contains additional information which is illustrated in Figure 27.

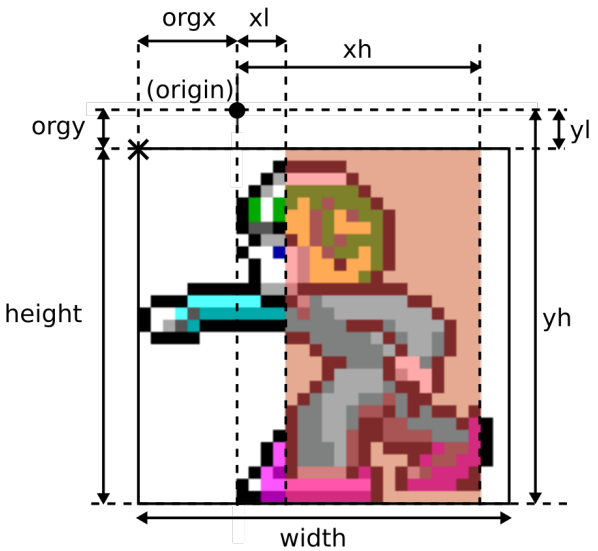


Figure 27: sprite structure

All global movement takes place from the origin. The origin is offset by `(orgx, orgy)` from the top-left location of the sprite. The parameters `(xl, xh, yl, yh)` define the hit box of the sprite, which is used to detect collisions.

index	width	height	orgx	orgy	xl	yl	xh	yh	shift
0	3	24	0	0	0	0	368	368	4
1	3	32	0	0	64	0	304	496	4
2	3	30	0	16	64	0	304	496	4
3	3	30	0	32	64	48	304	496	4
4	3	32	0	0	64	0	304	496	4
5	3	30	0	32	64	48	304	496	4
...	...	...	...	...	...	...	...	...	...
296	12	103	-128	0	256	128	752	1648	4

Table 1: content of spritetable[] in the `KDREAMS.EGA` asset file.

As each sprite can float freely over the screen, here also bitshifted sprites are used to position the sprite on a byte-aligned memory layout (as explained in section **??** on page **??**). The value in the shift column defines the amount of steps the sprite has to be shifted within 8 pixels. A shift value of 4 means the sprite is shifted in 4 steps with a 2 pixel interval.

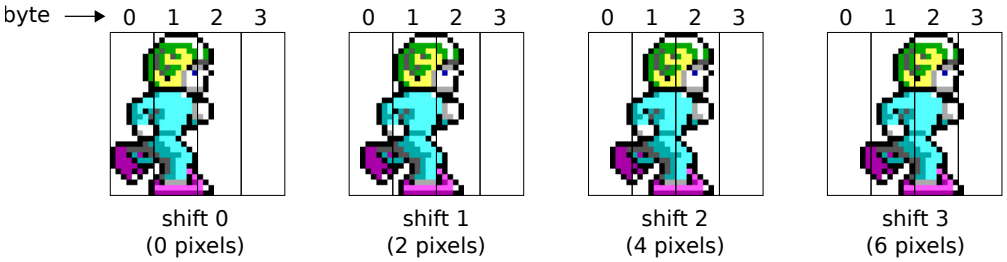


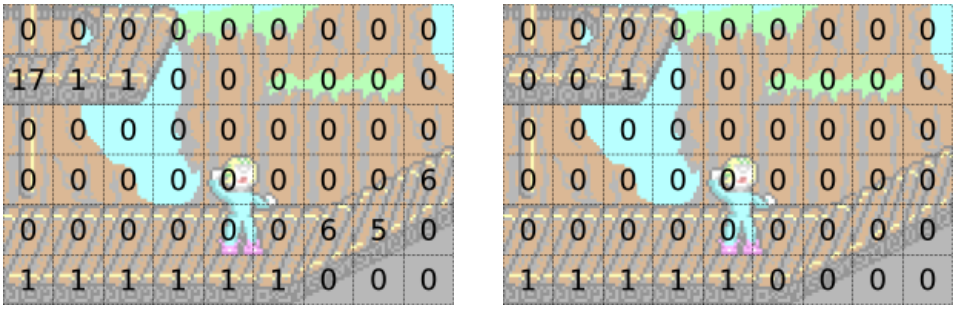
Figure 28: Sprite shifted in 4 steps.

Displaying the correct shifted sprite is as simple as below.

```
//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
```

0.4.3 Clipping

Before drawing a sprite on the screen, the engine determines if the boundaries of a sprite are hitting a wall or floor. This is called clipping and ensures an actor doesn't fall through a floor or walks through a vertical wall. To define whether a tile is a wall or floor, a tile is enriched with tile information, as explained in section **??** on page **??**. Each foreground tile contains a `NORTHWALL`, `SOUTHWALL`, `EASTWALL` and `WESTWALL`, as explained in section **??**. A number greater than 0 means the tile is a wall or floor when approaching from a given direction.



(a) Wall type map NORTHWALL

(b) Wall type map EASTWALL

Figure 29: Foreground tile clipping information.

As example, when a sprite, moving from right to left, is hitting a wall on the left side, it will update the sprite movement to ensure the sprite clips to the eastwall of the left tile as illustrated in Figure 30. The east/west wall clipping logic is covered by `ClipToEastWalls()` and `ClipToWestWalls()` functions.

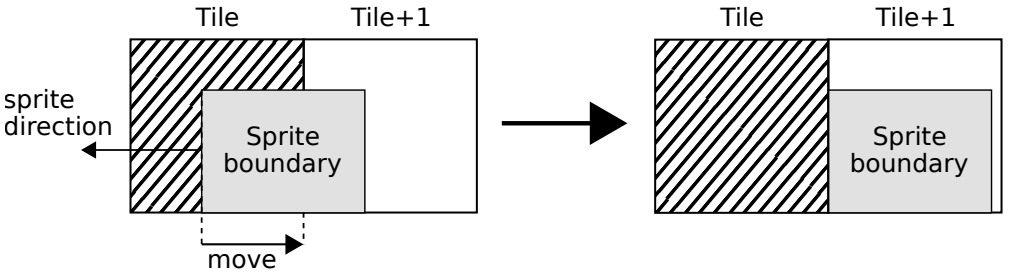


Figure 30: Clipping to east wall when moving from the west.

```
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 top and bottom the engine also needs to take clipping to slopes into account. After the sprite is clipped to the top or bottom of the wall tile, an offset can be applied to move a sprite up or down a slope. The offset is defined by a lookup table, where the midpoint of the sprite and the wall type from the foreground tile defines the offset.

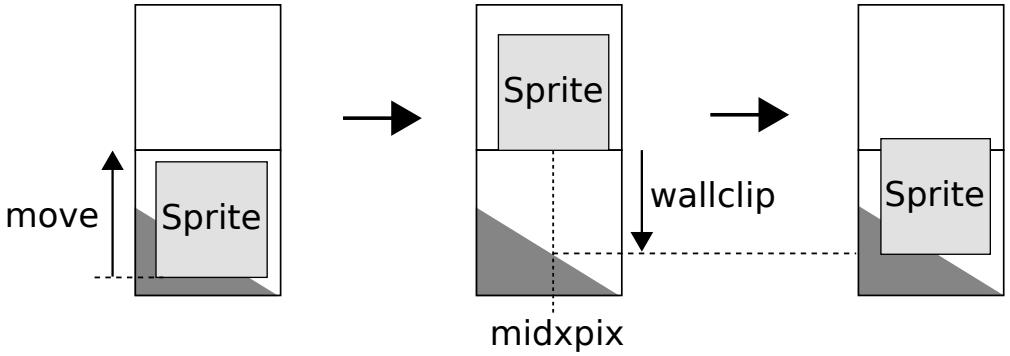


Figure 31: Clipping north wall with slope.

```
// walltype / x coordinate (0-15)

int wallclip[8][16] = {      // the height of a given point in a tile
{ 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256, 256},
{   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0},
{   0,0x08,0x10,0x18,0x20,0x28,0x30,0x38,0x40,0x48,0x50,0x58,0x60,0x68,0x70,0x78},
{0x80,0x88,0x90,0x98,0xa0,0xa8,0xb0,0xb8,0xc0,0xc8,0xd0,0xd8,0xe0,0xe8,0xf0,0xf8},
{   0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0},
{0x78,0x70,0x68,0x60,0x58,0x50,0x48,0x40,0x38,0x30,0x28,0x20,0x18,0x10,0x08,   0},
{0xf8,0xf0,0xe8,0xe0,0xd8,0xd0,0xc8,0xc0,0xb8,0xb0,0xa8,0xa0,0x98,0x90,0x88,0x80},
{0xf0,0xe0,0xd0,0xc0,0xb0,0xa0,0x90,0x80,0x70,0x60,0x50,0x40,0x30,0x20,0x10,   0}
};
```

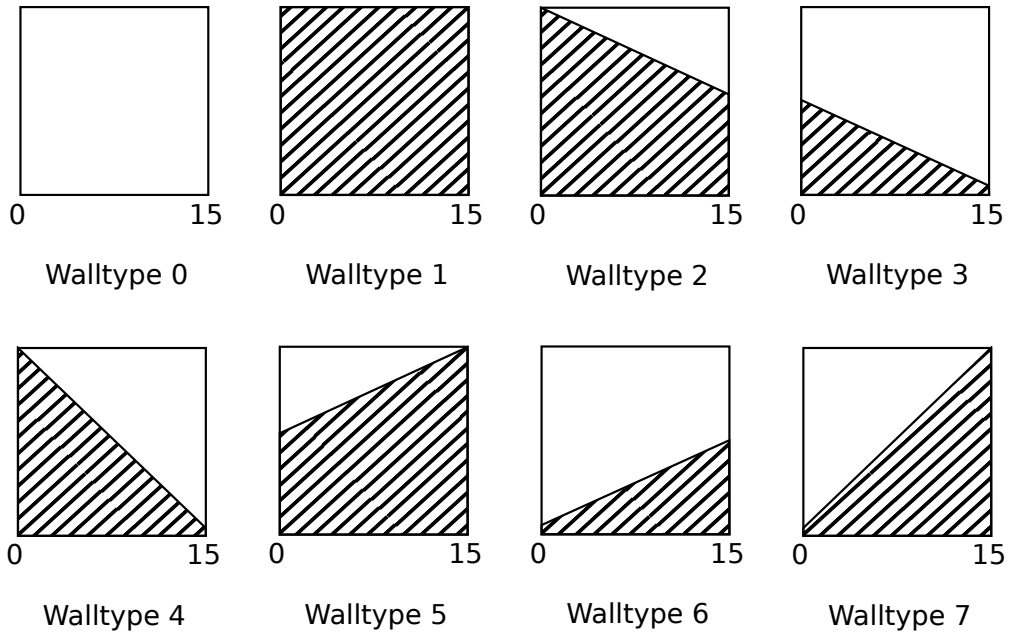


Figure 32: Eight different walltypes (slopes) defined.

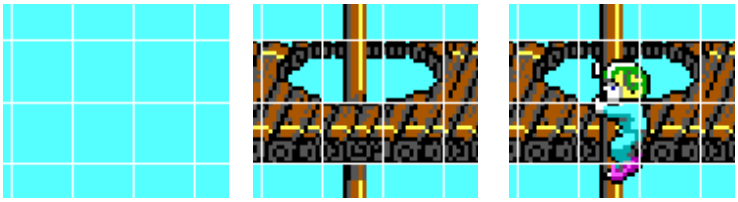
```
void ClipToEnds (objtype *ob)
{
  ...

  //Get midpoint of sprite [0-15]
  midxpix = (ob->midx&0xf0) >> 4;

  map = (unsigned far *)mapsegs[1] +
    mapbwidthtable[oldtilebottom-1]/2 + ob->tilemidx;
  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;
      }
    }
  }
```

0.4.4 Priority of tiles and sprites on screen

The normal screen build is as follows:

1. Draw the background tiles.
2. Draw the masked foreground tiles.
3. Draw the sprites on top of both the background and foreground tiles.

If multiple sprites are displayed on the same tile, each sprite is given a priority 0-3 to define the order of drawing. A sprite with a higher priority number is always displayed on top of lower priority sprites. As sprites are always displayed on top of tiles, this is causing unnatural situation when Commander Keen is climbing through a hole as illustrated in Figure 33.



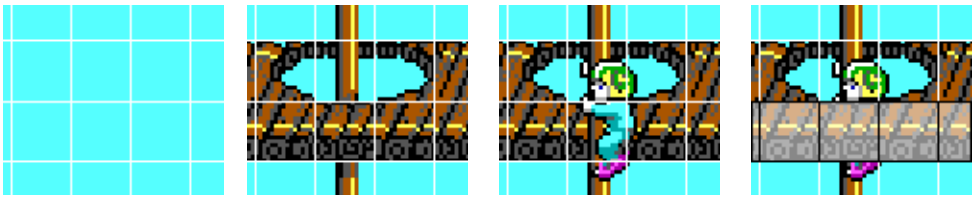
(a) Background tile. ***(b)*** Foreground tile. ***(c)*** Sprite on top.

Figure 33: Unnatural situation where Commander Keen is in front of a hole.

To draw sprites 'inside' a foreground tile, a small trick is used by introducing a priority foreground tile. As explained in section **??** each foreground tile is enriched with INTILE ('inside tile') information. If the highest bit (`80h`) of INTILE is set, this foreground tile has a higher priority than sprites with a priority 0, 1 or 2. So when drawing the tiles and sprites the following drawing order is applied:

1. Draw the background tile.
2. Draw the masked foreground tile.
3. Draw sprites with priority 0, 1 and 2 (in that order) and mark the corresponding tile in the tile buffer array with '3' as illustrated in Figure 11 on page 14.
4. Scan the tile buffer array for tiles marked with '3'. If the corresponsing foreground INTILE high bit (`80h`) is set, redraw the masked foreground tile.
5. Finally, draw sprites with priority 3. These sprites are always on top of everything.

The priority foreground tiles are updated in the `RFL_MaskForegroundTiles()` function.



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

Figure 34: Draw sprite inside a tile, by redrawing foreground tile.

```
  jmp SHORT @@realstart   ; start the scan
@@done:
;=============
; all tiles have been scanned
;=============
  ret

@@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
```

0.4.5 Manage refresh timing

After each screen refresh a certain amount of time, which we call ticks, has passed. The amount of ticks depends on several factors like amount of tiles refreshed and waiting time for a screen vertical retrace. Since all game actions and reactions rely on the amount of ticks between two refreshes, it is important to keep the tick interval consistent.

Without controlling the tick interval, the state and speed of actors could become unreliable, they could run faster and even can "warp" to an unpected location. To control refresh intervals, a minimum and maximum number of tics is defined in the play 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;
    }
}
```