0.1   Action Phase: Adaptive Tile Refreshment

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 (remember the planar mapping of section **??**?).

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 visible 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 need to round to integer). 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!
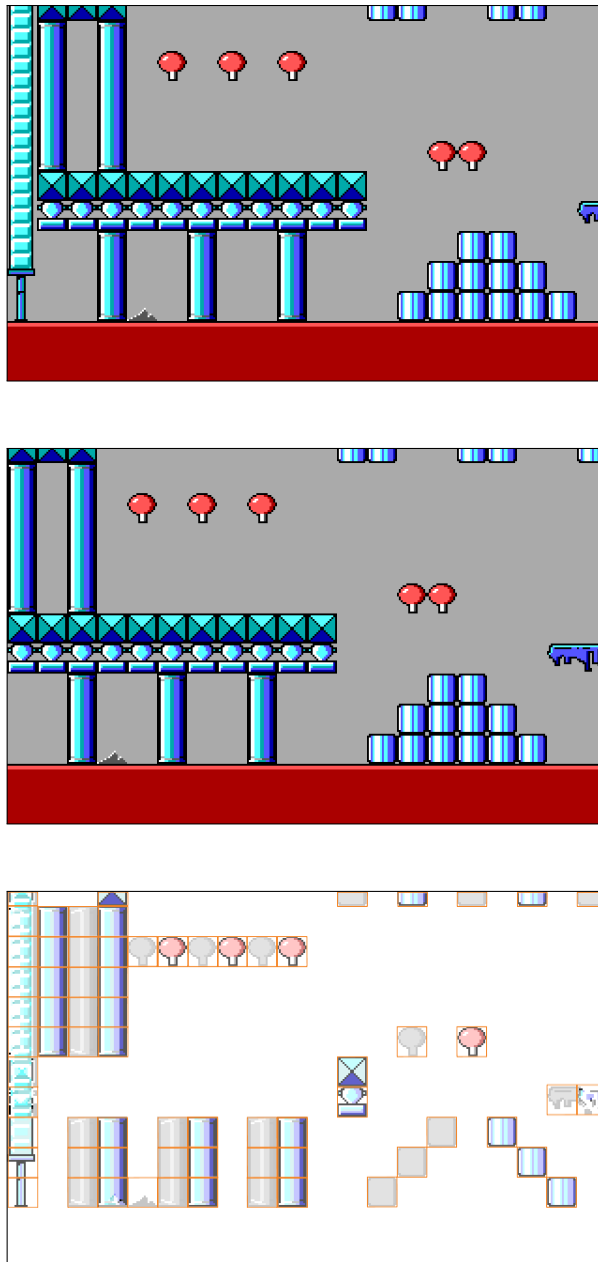
Figure 1: Start of the world, moved one tile to the right and difference.

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 high

  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.

```c
#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
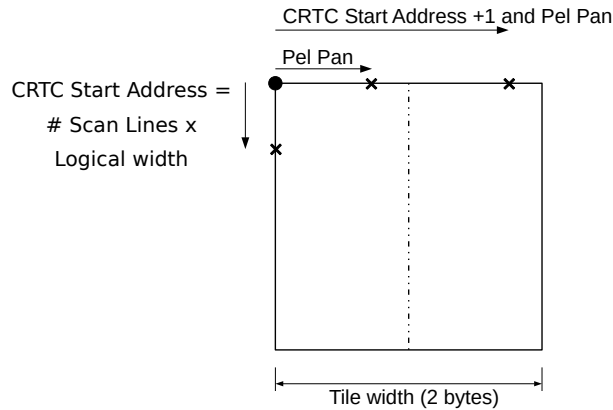
Figure 2: Smooth scrolling in EGA.

0.2   View Port and Buffer setup

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

So summarized, as illustrated in Figure 3, 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 buffer is the buffer screen plus two times a spare buffer to support floating the buffer screen two tiles in any direction.
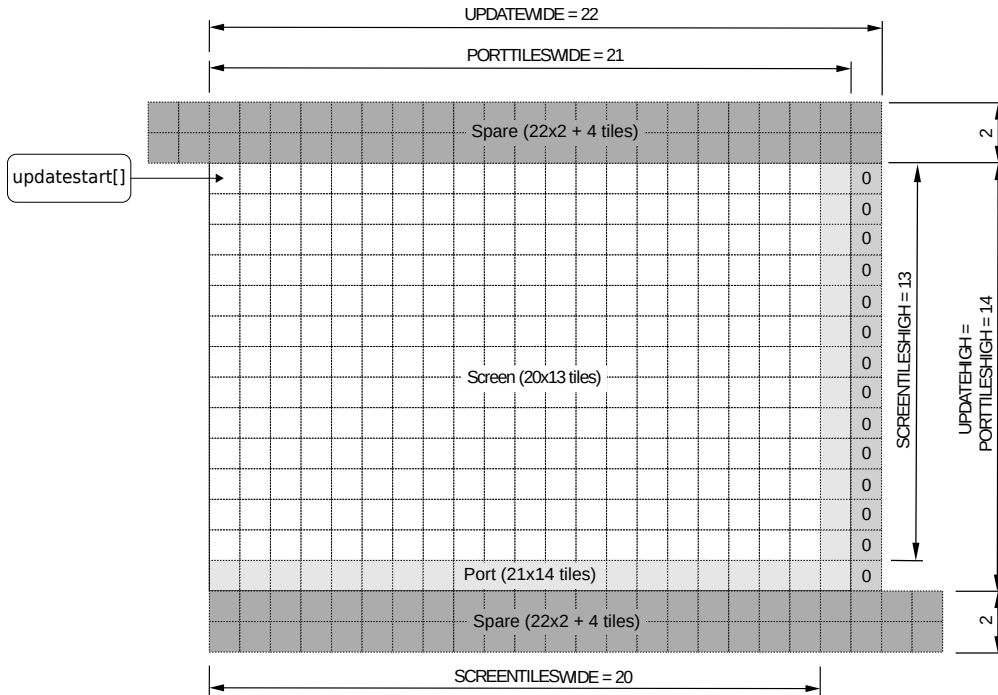
Figure 3: View and buffer tile layout.

0.3   Virtual screen buffer

Even if the screen is not scrolling, tile refreshes are required to support sprite 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. As explained in the previous section, the view port has a size of 21x14 tiles, which is 336x224 pixels. That means the logical width in VRAM must be at least 336 pixels (42 bytes) wide. In the file `id_vw.h` the VRAM screen buffer is defined by SCREENSPACE, which is set to 64x240 bytes, or 512x240 pixels. This is more than sufficient to update one virtual screen in VRAM.

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:

8

- Page 0 and 1, which are used to switch between buffer and view screen

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

Figure 4: 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.

0.4   Life of a 2D Frame

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.

9

0.4.1 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 to be updated upon the next refresh for both pages (view and buffer screen).

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.

Before we start the explanation, let's first describe the most important variables and definitions used for the refresh process:

- `screenstart[]` is a pointer to the starting address (upper-left pixel) of the viewport in VRAM. As explained before 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[]` is a pointer to the tile buffer array. 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.

- `Visible screen`, which refers to the starting position of the visible screen on the monitor. This is done by setting both the `CRTR` start address and `Pel Panning`.

---

[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.

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.
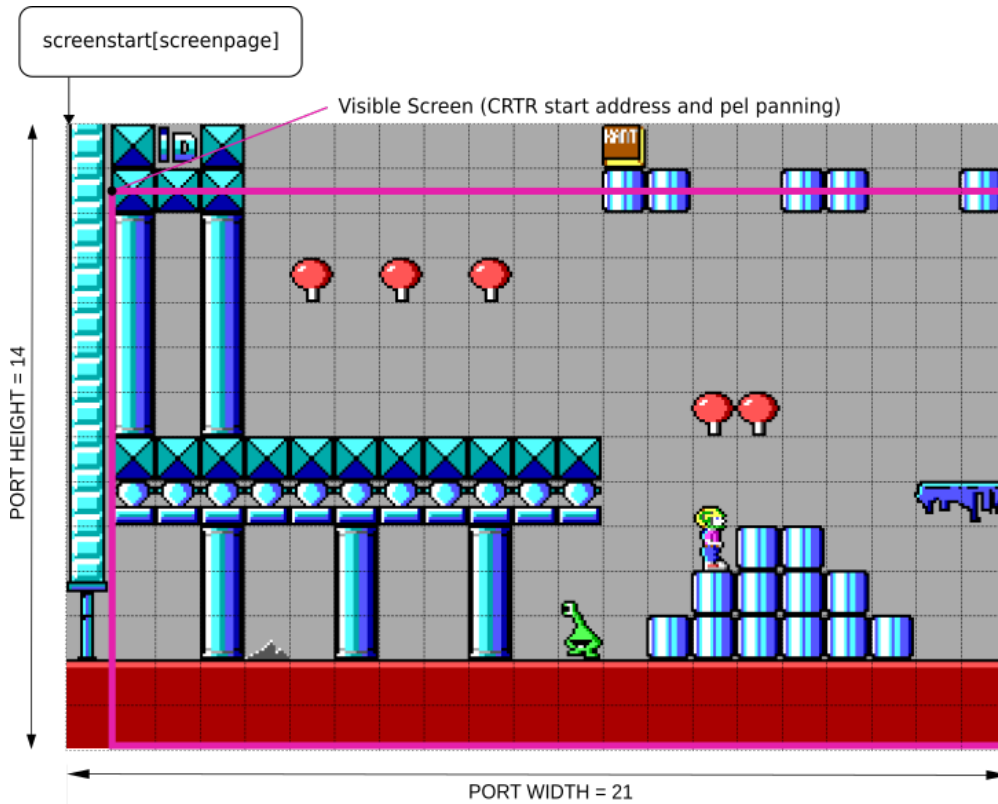

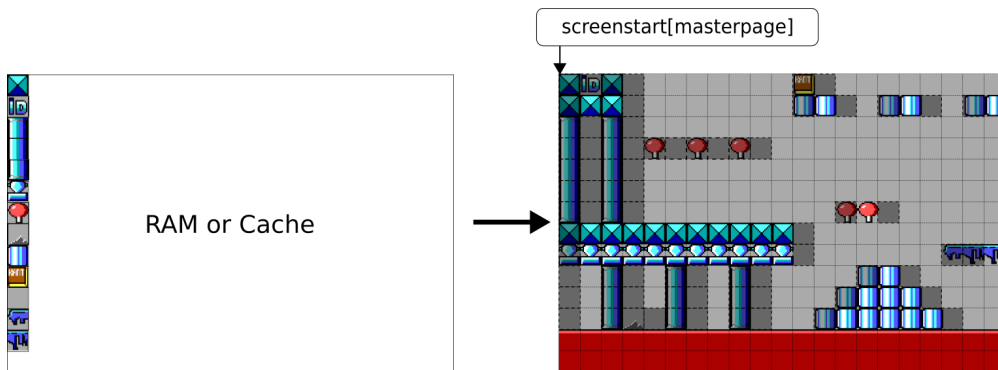*Figure 5:* Step 1: Scroll screen to the right

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.
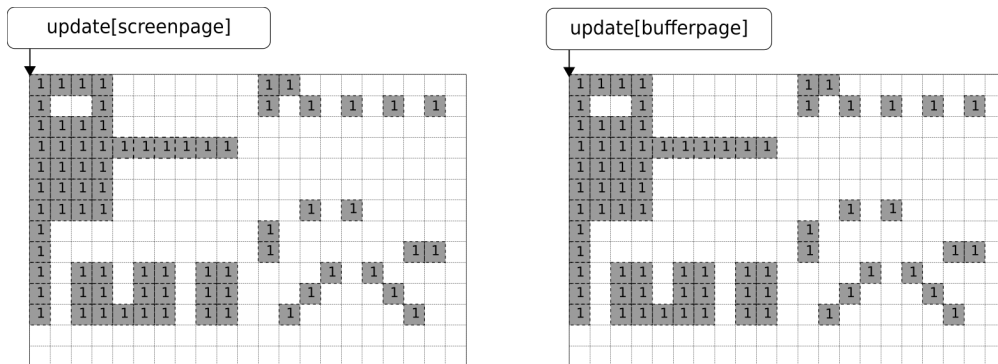
Figure 7: Mark all changed tiles with '1' in both tile buffer and tile view array.
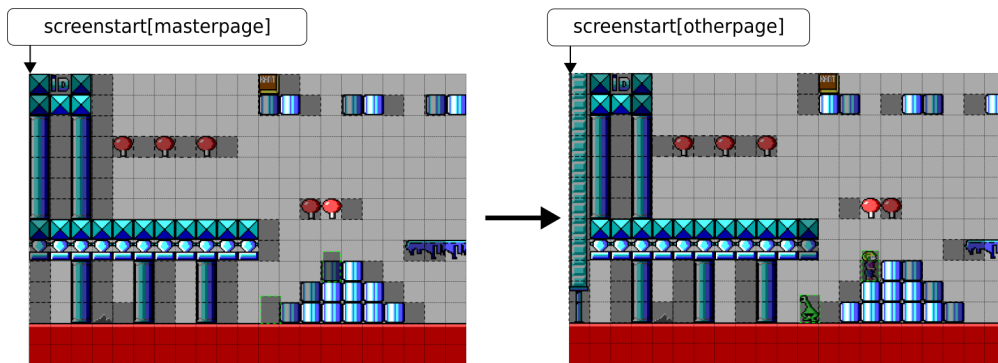
Figure 8: Step 3 and 4: Copy tiles from master to buffer screen and remove sprites

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'.
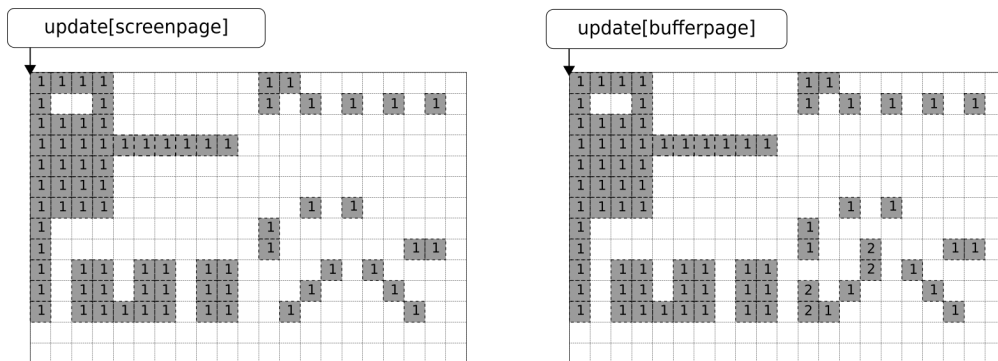

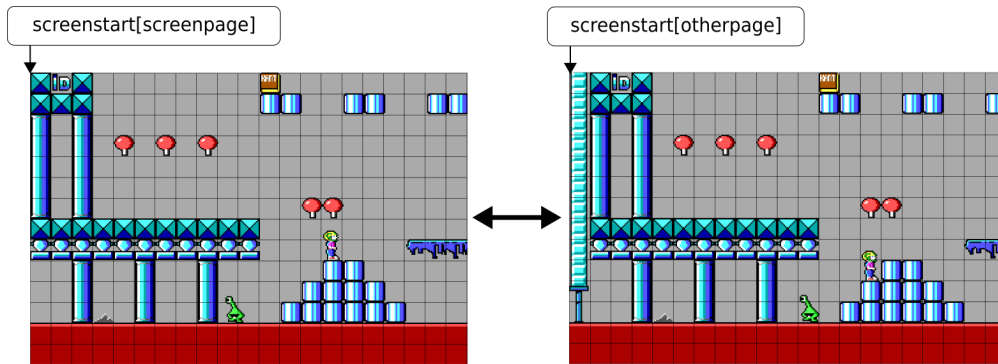**Figure 9:** Mark removed sprites with '2' in tile buffer array only.

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).
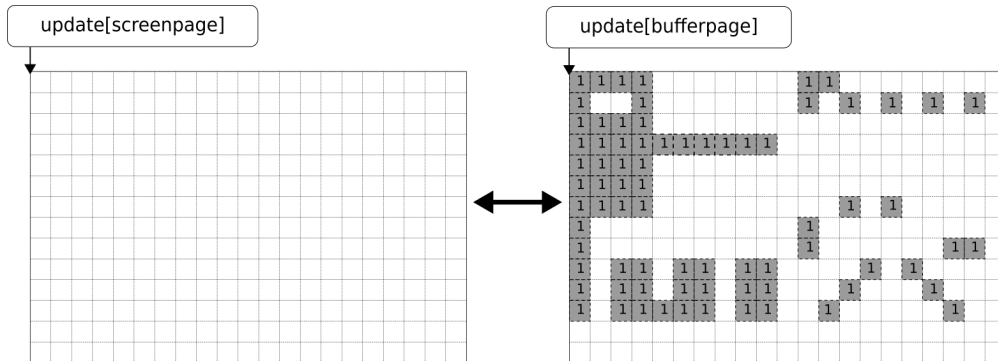
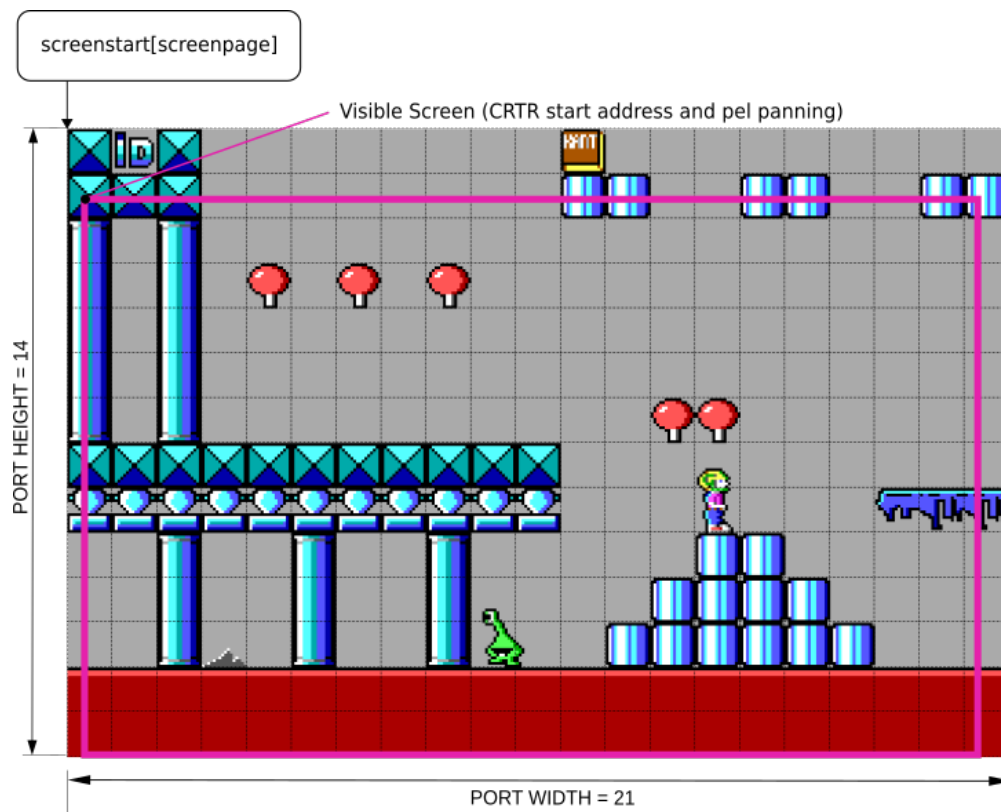**Figure 13:** Clear tile update array and swap arrays.

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.

0.4.2 Optimize tile updates

The hardware chapter described Mode `0Dh`, which despite being unfit for games, still has an interesting characteristic. 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.
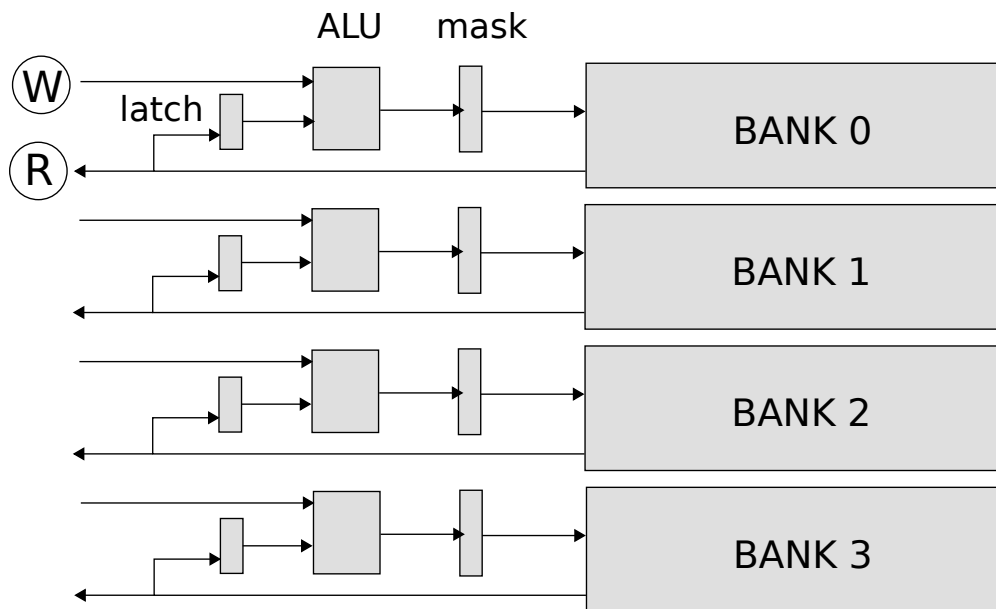

Figure 15: Latches memorize read operations from each bank. The memorized value can be used for later writes.

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.

```
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.

0.4.3   Wrap around the EGA Memory

In the later versions of Commander Keen, John Carmack explored what would happen if you push the virtual screen over de 64kB, or `0xFFFF`, border 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.
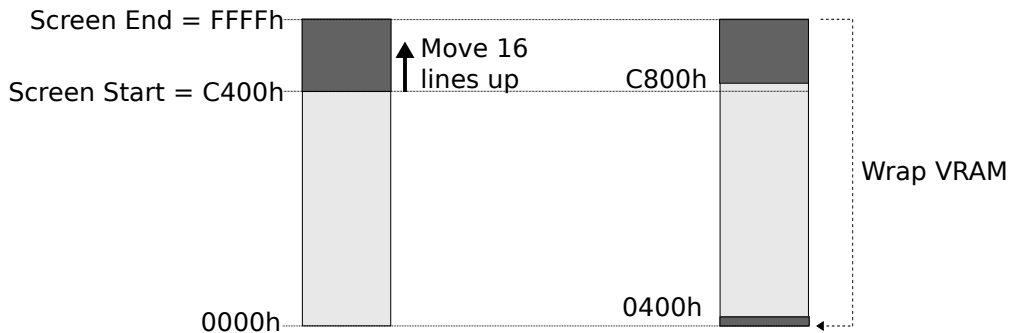
18

Figure 16: 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 4 on page 9, 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 corresponding screen is copied to the opposite end of the buffer, as illustrated in Figure 17.

(a) Virtual pages moved down   *(b)* Virtual pages moved up
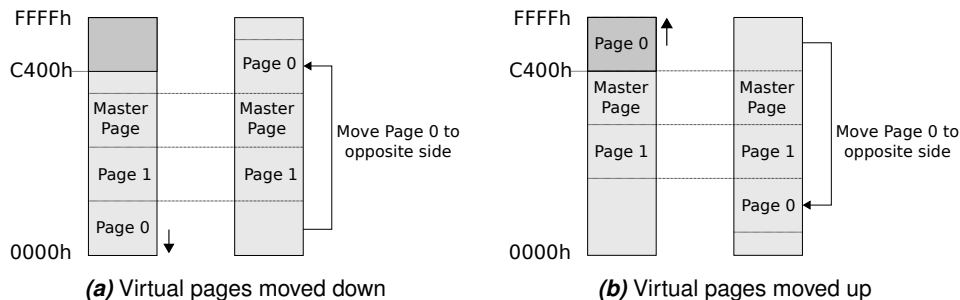
Figure 17: Move screen to opposite end of VRAM buffer

The screen copy comes at the cost of some performance, but is not noticed during game play, also due to the fact that we can make use of copying 4 bytes in one cycle using the

---

[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.

latches as explain in section 0.4.2.

```
#define SCREENSPACE    (SCREENWIDTH*240)
#define FREEEGAMEM     (0x10000l-31*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);
  }
}
```

0.4.4   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.

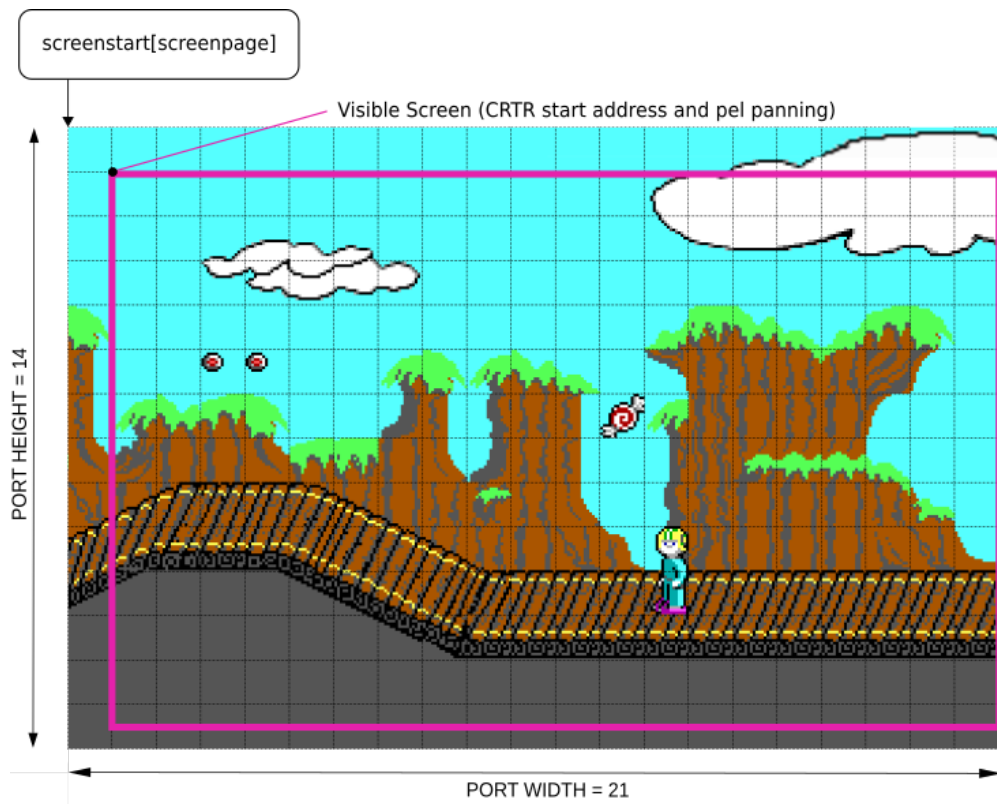As you can see, step 2 to 4 are different from Commander Keen 1-3, the rest of the steps are the same. In the example below the screen is forced to scroll to the left.

Figure 18: Step 1: Scroll screen to the left
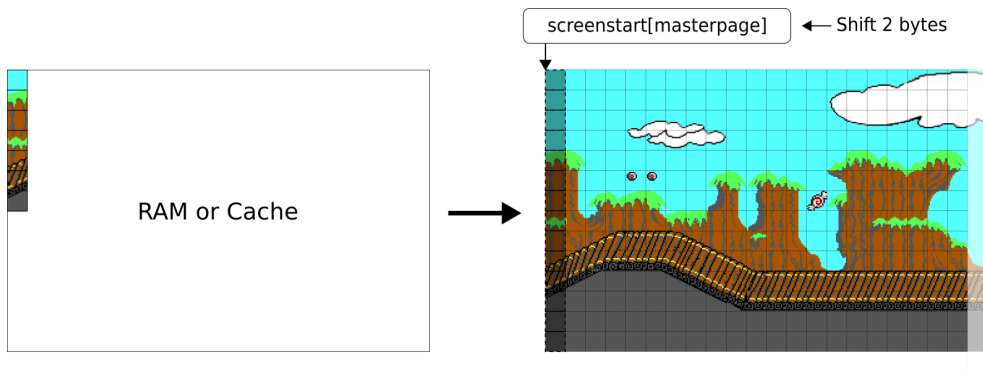
Figure 19: Step 2 and 3: Shift screen pointer and add column to VRAM

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'.
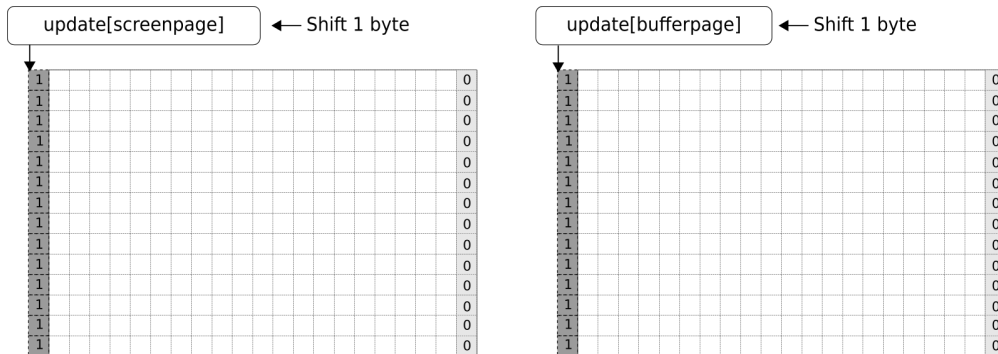

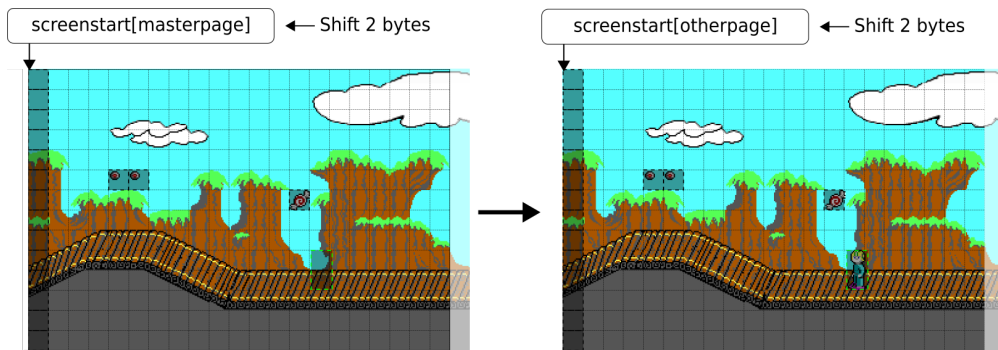**Figure 20:** Mark new column in both tile buffer and display array.

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.
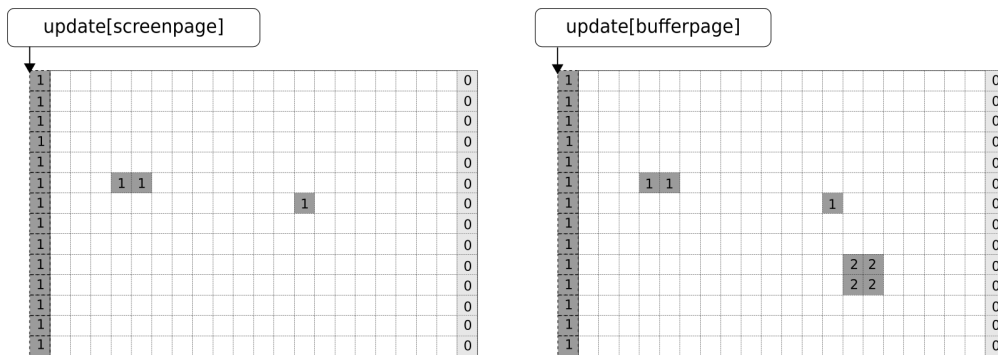

Figure 22: Tile buffer array with removed sprites marked with '2'.

<u>Trivia :</u> The removal blocks which are marked '2' in the tile buffer array are nowhere used in the engine.
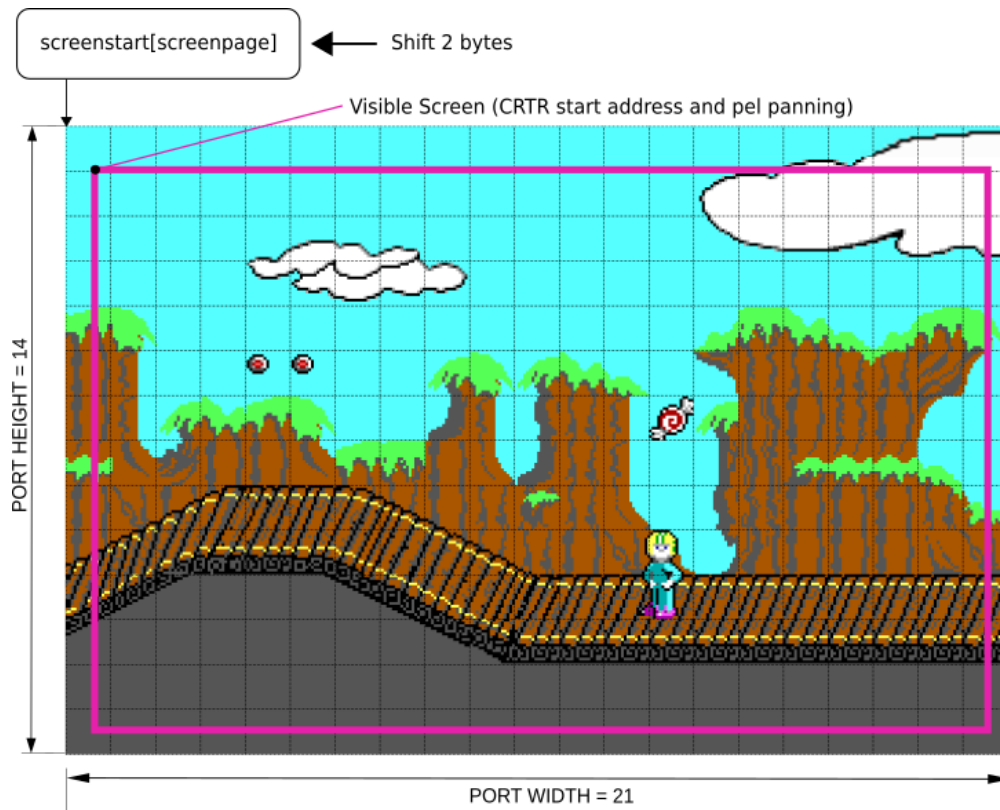
Figure 23: Step 7: Updated screen after swapping buffer and screen page

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!

Now we can also see why the tile array buffer is 2 tiles wider on all sides than the view port (see Figure 3). Let's take the situation where the screen needs to scroll to the top-left, meaning 1 tile left and 1 tile up as illustrated in Figure 24. 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).
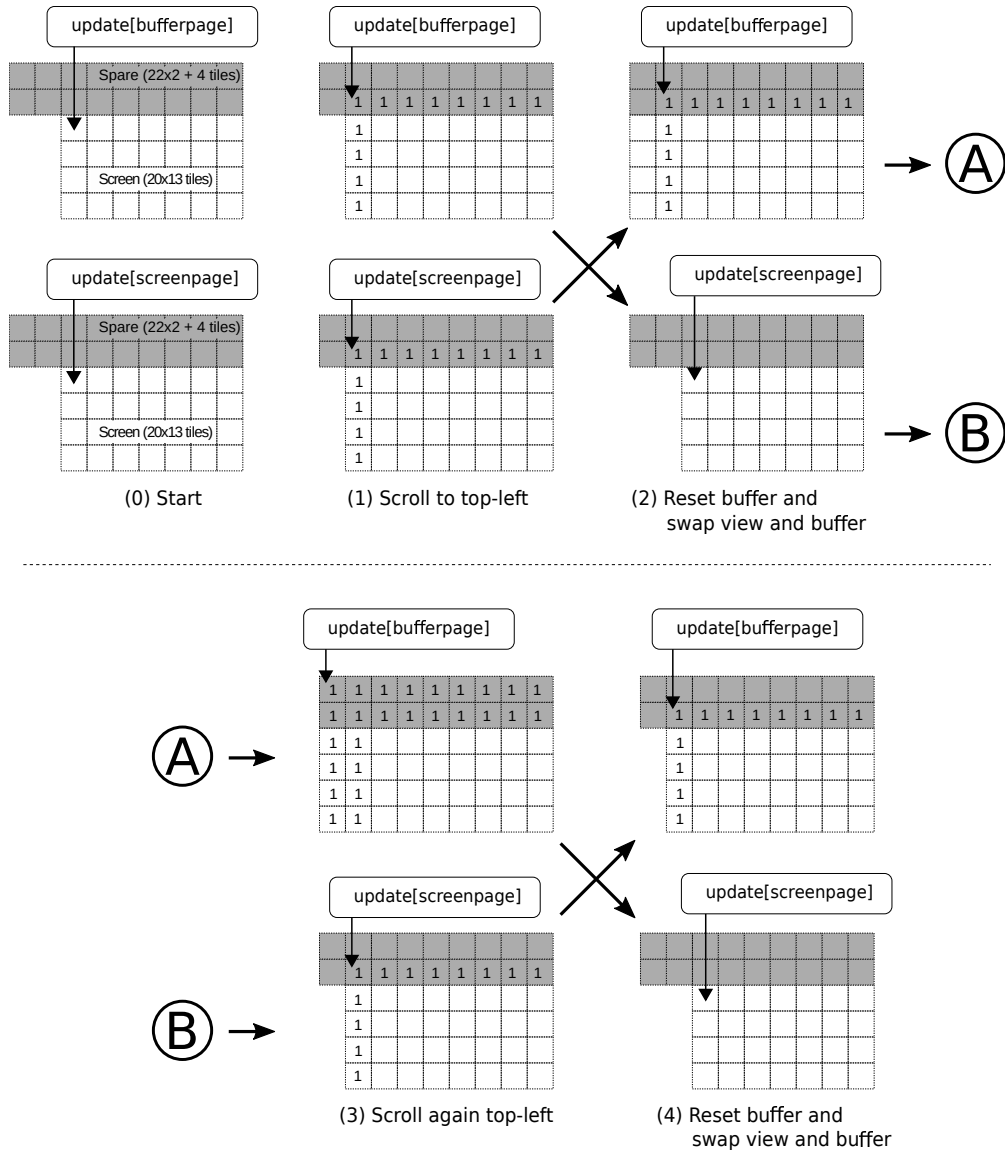
Figure 24: scroll to top-left tile

Now, if the screen needs again to scroll to the top-left you can see why we need the 2nd row for the buffer. The process will repeat, but after this cycle the *update[otherpage] is cleared and resetted.

0.5 Refresh video screen

Here explain with code. Focus on when to perform screen refresh (waiting for vertical retracement)

0.6 A.I.

To simulate enemies, some objects are allowed to "think" and take actions like firing, walking, or emitting sounds. These thinking objects are called "actors". Actors are programmed via a state machine. They can be aggressive, sneaky, or dumb (XXX for instance). To model their behavior, all enemies have an associated state:

0.7 Drawing Sprites

Once the state of the actor is updated, it is time to render the actor on the screen. This is a three step operation.

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

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 **??**, table **??**) each sprite asset contains additional information which is explained in Figure 25.
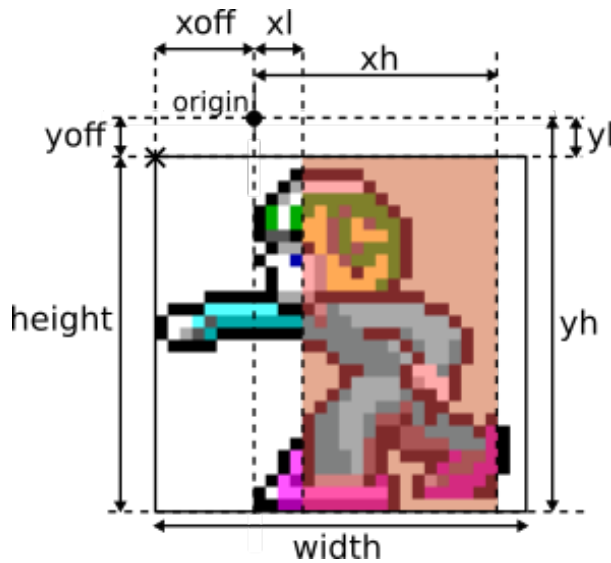
Figure 25: sprite structure

All global movement takes place from the origin. The offset (`xoff`,`yoff`) defines the top-left position of the sprite. Together with the width and height it defines the dimensions of the sprite. The parameters xl, xh, yl and yh define the hit box of the sprite, which is used to detect collisions.

As each sprite can float freely over the screen, also here bit shift sprites are used to position the sprite (as explained in section **??** on page **??**). The value in the shift table defines the amount of steps the sprite has to be shifted within 8 pixels. So a value of 4 means the sprite is shifted in steps of 2 pixels (8 / 4), where a shift value of 1 means the sprite is not shifted at all (so this sprite is always perfectly 8-pixel aligned).

0.7.1 Visible actor determination

0.7.2 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 can be enriched with wall information. For each level map the tile info variable `tinf` contains a `NORTHWALL`, `SOUTHWALL`, `EASTWALL` and `WESTWALL` map as illustrated in Fig 26.

(a) Wall type map NORTHWALL

(b) Wall type map EASTWALL
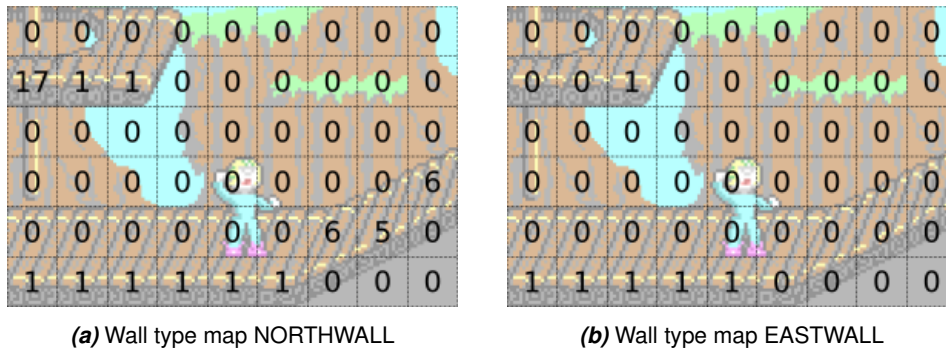
Figure 26: Tile clipping map

When the sprite boundary is hitting a wall on the right (east), it will update the sprite movement to ensure the sprite right boundary is equal to the right side of the tile as illustrated in Fig. XX. The east/west wall clipping logic is covered by `ClipToEastWalls()` and `ClipToWestWalls()`.

```
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;
    }
  }
}
```

The clipping for top and bottom is a bit more complex, as the engine also needs to take walking on 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.
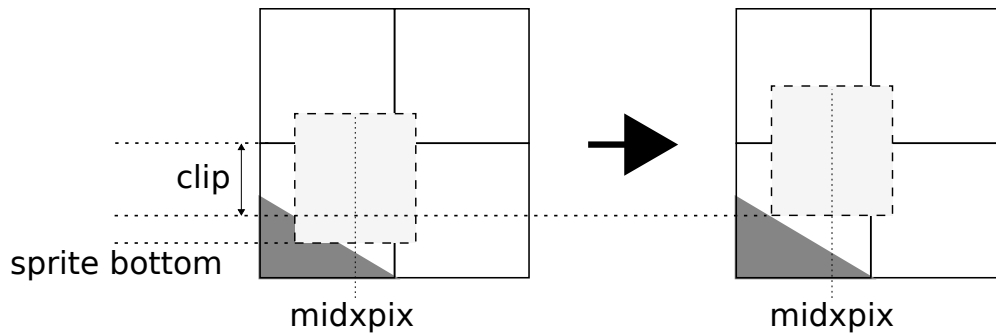
Figure 27: Clipping NORTHWALL

```
// 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}
};
```

When a sprite is clipped to a top or bottom tile, the corresponding midpoint pixels (0-15) and tile info map defines the offset from this top or bottom tile.

```
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;
      }
    }
  }
}
```