0.1 Performance and other Tricks

Various smart techniques were employed to ensure every CPU cycle counted. This section explores some of the techniques used in Commander Keen.

0.1.1 Tile caching

When updating the master screen in VRAM, the engine copies each tile from RAM to VRAM. Each pixel requires a read and write operation to the four memory banks. As explained in section **??**, by reprogramming the latches on the EGA card, it was possible to copy four bytes at once from one VRAM location to another VRAM location. This trick could be applied to tiles if they contained only a background tile with no foreground layer.

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

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

The assembly function `RFL_NewTile` is responsible for drawing tiles to the master screen. It first checks if the background tile is already available in the tile cache array. If it is, the tile is copied with 32 bits per cycle from VRAM. If it is not, the tile is loaded from RAM, and the memory pointer in VRAM for that tile is stored in the tile cache array, allowing future requests to be served from VRAM.

```
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
```

0.1.2 Memory alignment

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. Even worse, writing a word at offset address `FFFFh` causes an exception.

Since a tile is word-aligned (16 bits wide) and the screen is refreshed in tile-size steps, it is always aligned with even memory addresses. However, this is not the case for sprites, which are byte-aligned. When a sprite is drawn starting from an odd memory address, the CPU can only read and write 8 bits at a time. To utilize the 16-bit data bus and avoiding the memory exception, the engine first checks whether the destination is at an even or odd memory address. f the address is odd, it first writes a byte to VRAM to align the next operation to an even address, after which it continues writing in 16-bit words. To optimize sprite writing to VRAM, the engine uses small function routines for each combination of sprite width and even/odd address alignment.

	Start address	
Sprite width (bytes)	**Even**	**Odd**
1	Byte	Byte
2	Word	Byte, Byte
3	Word, Byte	Byte, Word
4	Word, Word	Byte, Word, Byte
5	Word, Word, Byte	Byte, Word, Word
6	Word, Word, Word	Byte, Word, Word, Byte
7	Word, Word, Word, Byte	Byte, Word, Word, Word
8	Word, Word, Word, Word	Byte, Word, Word, Word, Byte
9	Word, Word, Word, Word, Byte	Byte, Word, Word, Word, Word
10	Word, Word, Word, Word, Word	Byte, Word, Word, Word, Word, Byte

```
EVEN
mask1E:
  MASKBYTE
  SPRITELOOP  mask1E

EVEN
mask2E:
  MASKWORD
  SPRITELOOP  mask2E

EVEN
mask2O:
  MASKBYTE
  MASKBYTE
  SPRITELOOP  mask2O
```

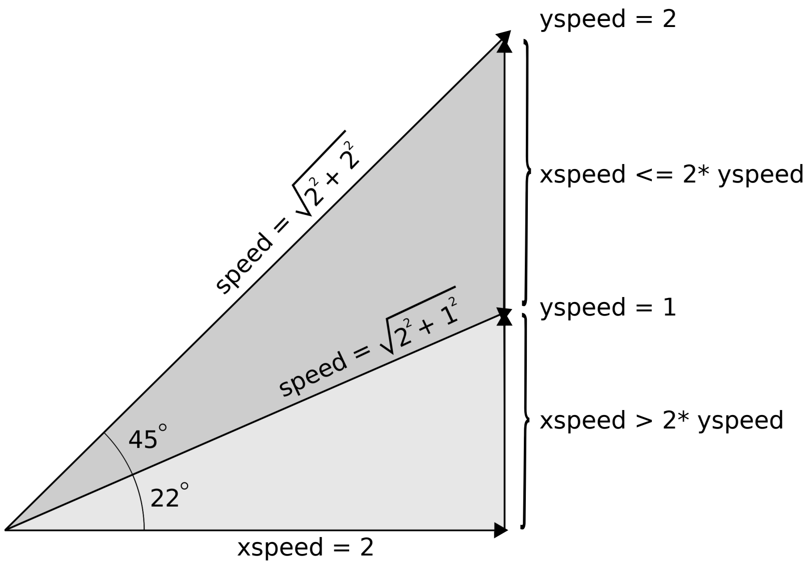
Each function pointer is stored in a `maskroutines` array. By cleverly using bit-shift operations and the Carry Flag (CF), the engine calls the appropriate function to write the sprite to VRAM.

```
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

routinetouse  dw  ?

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
```

0.1.3 Bouncing Physics

When Keen throws a flower it bounces of the walls. For flat walls and floors the bounce can be easily calculated by reversing either the x-speed (for vertical walls) or y-speed (for horizontal walls). It becomes more complicated for slopes. Making an accurate calculation of the bounce on a slope requires expensive `cos` and `sin` methods.

Instead, the game used a simple algorithm that approximates the angle to either 22°, 45°, 67° or 90°. Based on the ratio between the x- and y-speed it calculates the resulting speed and corresponding angle.



The speed is calculated as a factor of either the x- or y-speed, depending which of the two has the largest absolute value. Notice that for higher precision the speed is multiplied with 256.

```
void  PowerReact (objtype *ob)
{
    unsigned wall,absx,absy,angle,newangle;
    unsigned long speed;

    absx = abs(ob->xspeed);
    absy = ob->yspeed;

    wall = ob->hitnorth;
    if ( wall == 17)  // go through pole holes
    {
        [...]
  }
    else if (wall)
    {
        ob->obclass = bonusobj;
        if (ob->yspeed < 0)
            ob->yspeed = 0;

        absx = abs(ob->xspeed);
        absy = ob->yspeed;
        if (absx>absy)
        {
            if (absx>absy*2)  // 22 degrees
            {
                angle = 0;
                speed = absx*286; // x*sqrt(5)/2
            }
            else         // 45 degrees
            {
                angle = 1;
                speed = absx*362; // x*sqrt(2)
            }
        }
        [...]        // Handle 67 and 90 degrees
    }
}
```

For each combination of the eight type of slopes (Figure **??**) and incoming angle, the corresponding bounce angle is calculated using a simple lookup table.

```
// bounceangle[walltype][angle]

unsigned  bounceangle[8][8] =
{
{0,0,0,0,0,0,0,0},
{7,6,5,4,3,2,1,0},
{5,4,3,2,1,0,15,14},
{5,4,3,2,1,0,15,14},
{3,2,1,0,15,14,13,12},
{9,8,7,6,5,4,3,2},
{9,8,7,6,5,4,3,2},
{11,10,9,8,7,6,5,4}
};
```

The value in the table refers to the corresponding bounce angle calculation. As example, walltype 3 with incoming angle of 22°, results in bounce calculation case 5.

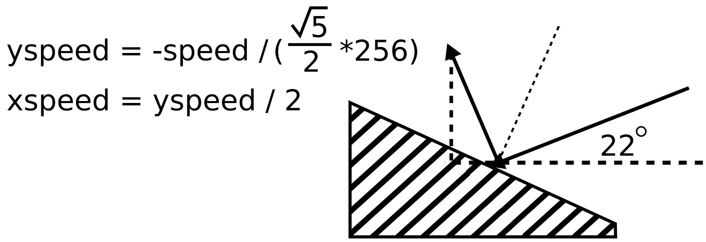
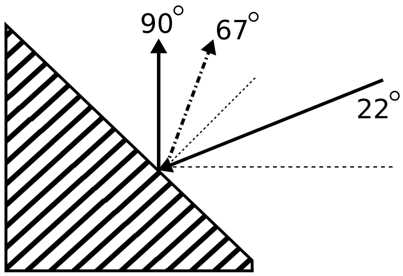


Figure 1: Walltype 3 with incoming angle of 22° (angle=0).

```
if (ob->xspeed > 0)
  angle = 7-angle;          // mirror angle

speed >>= 1;                // speed / 2 after bounce
newangle = bounceangle[ob->hitnorth][angle];
switch (newangle)
{
[...]

case 5:
  ob->yspeed = -(speed / 286);
  ob->xspeed = ob->yspeed / 2;
  break;

[...]
}
```

Notice that in several cases the bounce angle is not following the laws of physics. As example, for an incoming angle of 22° on a 45° slope the bounce angle is 90°, instead of 67°.



0.1.4 Pseudo Random Generator

Random numbers are necessary for many things during runtime, such as calculating whether an enemy is able to hit the player based on its accuracy. This is achieved with a precalculated pseudo-random series of 256 elements.

```
rndindex   dw   ?

rndtable

db    0,   8, 109, 220, 222, 241, 149, 107,  75, 248, 254, 140,  16,  66
db   74,  21, 211,  47,  80, 242, 154,  27, 205, 128, 161,  89,  77,  36
db   95, 110,  85,  48, 212, 140, 211, 249,  22,  79, 200,  50,  28, 188
db   52, 140, 202, 120,  68, 145,  62,  70, 184, 190,  91, 197, 152, 224
db  149, 104,  25, 178, 252, 182, 202, 182, 141, 197,   4,  81, 181, 242
db  145,  42,  39, 227, 156, 198, 225, 193, 219,  93, 122, 175, 249,   0
db  175, 143,  70, 239,  46, 246, 163,  53, 163, 109, 168, 135,   2, 235
db   25,  92,  20, 145, 138,  77,  69, 166,  78, 176, 173, 212, 166, 113
db   94, 161,  41,  50, 239,  49, 111, 164,  70,  60,   2,  37, 171,  75
db  136, 156,  11,  56,  42, 146, 138, 229,  73, 146,  77,  61,  98, 196
db  135, 106,  63, 197, 195,  86,  96, 203, 113, 101, 170, 247, 181, 113
db   80, 250, 108,   7, 255, 237, 129, 226,  79, 107, 112, 166, 103, 241
db   24, 223, 239, 120, 198,  58,  60,  82, 128,   3, 184,  66, 143, 224
db  145, 224,  81, 206, 163,  45,  63,  90, 168, 114,  59,  33, 159,  95
db   28, 139, 123,  98, 125, 196,  15,  70, 194, 253,  54,  14, 109, 226
db   71,  17, 161,  93, 186,  87, 244, 138,  20,  52, 123, 251,  26,  36
db   17,  46,  52, 231, 232,  76,  31, 221,  84,  37, 216, 165, 212, 106
db  197, 242,  98,  43,  39, 175, 254, 145, 190,  84, 118, 222, 187, 136
db  120, 163, 236, 249
```

Each entry in the array has a dual function. It is an integer within the range [0-255][1] and it is also the index of the next entry to fetch for next call. This works overall as a 255 entry chained list. The pseudo-random series is initialized using the current time modulo 256 when the engine starts up.

[1] Or at least it was intended to!

```
;================================================
;
;
; void US_InitRndT (boolean randomize)
; Init table based RND generator
; if randomize is false, the counter is set to 0
;
;
;================================================


PROC  US_InitRndT randomize:word

  uses  si,di
  public  US_InitRndT

  mov ax,[randomize]
  or  ax,ax
  jne @@timeit    ;if randomize is true, really random

  mov dx,0      ;set to a definite value
  jmp @@setit

@@timeit:
  mov ah,2ch
  int 21h     ;GetSystemTime
  and dx,0ffh

@@setit:
  mov [rndindex],dx
  ret

ENDP
```

The random number generator saves the last index in `rndindex`. Upon request for a new number, it simply looks up the new value and updates `rndindex`.

```
;==================================================
;
; int US_RndT (void)
; Return a random # between 0-255
; Exit : AX = value
;
;==================================================
PROC  US_RndT
  public  US_RndT

  mov bx,[rndindex]
  inc bx
  and bx,0ffh
  mov [rndindex],bx
  mov al,[rndtable+BX]
  xor ah,ah
  ret

ENDP
```

0.1.5 Screen fades

When a new level is loaded, the screen fades from black to the default colors. Here it makes use of reassigning the color palette. This can easily be done by calling BIOS software interrupt `10h`.

```
_AX = 0x1000 ; Set One Palette Register
_BL = 0      ; index color number to set
_BH = 0x5    ; 6-bit rgbRGB color to display for that index
geninterrupt (0x10) ; Generate Video BIOS interrupt
```

Earlier in the hardware chapter, Section **??**, it was explained that most EGA monitors did not support the extended 64-color rgbRGB palette, but kept the CGA pin assignment. That means applying "rgbRGB" results in wrong color mapping to the monitor. To better understand this, let's have a look at the pin signals.

Pin	EGA modes (rgbRGB)	CGA modes (RGBI)
1	Ground	Ground
2	Secondary Red (Intensity)	Ground
3	Primary Red	Red
4	Primary Green	Green
5	Primary Blue	Blue
6	Secondary Green (Intensity)	Intensity
7	Secondary Blue (Intensity)	Reserved
8	Horizontal Sync	Horizontal Sync
9	Vertical Sync	Vertical Sync

Figure 2: EGA and CGA DE-9 connector pin signals.

If one assigns the color brown (rgbRGB is `010100b`) to one of the color indexes, the resulting color on the CGA pin assignment is light red; The secondary green pin ("r" in rgbRGB) is mapped to the Intensity pin in CGA mode, which results color red with intensity and not the expected brown color. So mapping the color to one of the indexes is based on "RGBI", using the Secondary Green ("r") for the intensity. The "b" has no meaning and the "r" (Ground) is normally set to 0.

By calling `_AX=1002h` the entire palette can be reprogrammed. In this case `ES:BX` points to 17 bytes; an rgbRGB value for each of 16 palette index plus one for the border. The screen fading is defined by the `colors[7][17]` scheme. Note that the "b" bit is set for all intensity colors, but this had no effect on the results since the pin is unassigned for CGA.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	0
2	0	0	0	0	0	0	0	0	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
3	0	1	2	3	4	5	6	7	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
4	0	1	2	3	4	5	6	7	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0
5	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f

Figure 3: Color fading table.

Fading in the screen from black to color is rather straight forward.

```
void VW_FadeIn(void)
{
    int i;

    for (i=0;i<4;i++)
    {
        colors[i][16] = bordercolor;
        _ES=FP_SEG(&colors[i]);
        _DX=FP_OFF(&colors[i]);
        _AX=0x1002;
        geninterrupt(0x10);
        VW_WaitVBL(6);
    }
    screenfaded = false;
}
```