GAME ENGINE BLACK BOOK

COMMANDER KEEN

v2023.07.15 by Bas Smits

Contents

Chapter 1

Introduction

My personal introduction to computer gaming started in 1985, when my parents bought a MSX-1 Home Computer. I was fascinated by games such as Konami's *Knightmare* and *Nemesis 2*. It was not only the gameplay that interested me, but also how such games are developed. That's how I started my interest into programming and game development.

The same year I had my first home computer, Nintendo released a game called *Super Mario Bros.* on the Nintendo Entertainment System (NES). It was an instant blockbuster; it combined great graphics with smooth side scrolling. Side scrolling games from that time period, like Knightmare and Nemesis 2, moved at constant and "choppy" speed. Super Mario Bros. was different, as the player dictates the scrolling speed. You could smoothly accelerate from walk to run or jump, and the screen would smoothly follow your actions. Super Mario Bros. was immensely successful, both commercially and critically. It helped popularize the side-scrolling platform game genre, and served as a killer app for the NES[1].

[1] Upon release in Japan, 1.2 million copies were sold during its September 1985 release month. Within four months, about 3 million copies were sold in Japan

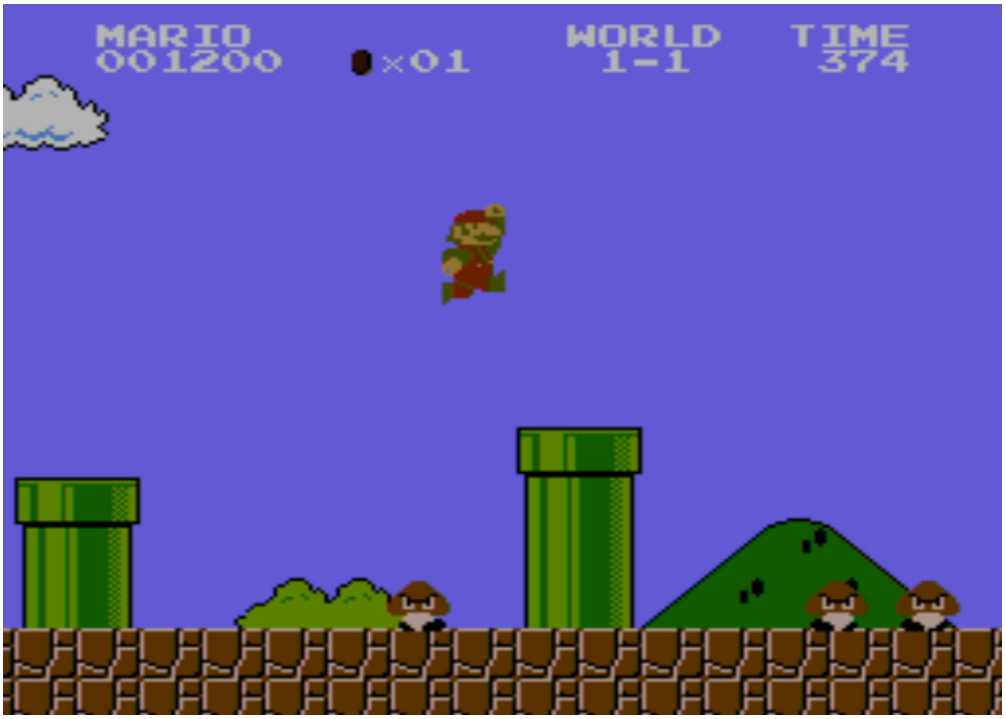


Figure 1.1: Super Mario Bros. on Nintendo Entertainment System

Super Mario Bros. showed the real power of the NES, which was hardware supported scrolling. Most computers around that time, like MSX and Commodore-64 computer systems, only had hardware support for sprites. To perform side scrolling on these platforms one need to move all the background "characters" (typically represented by 8x8 pixel tiles), which is why you get that super choppy "scrolling". The only way to actually get smooth pixel scrolling was by redrawing the entire screen, offset by the number of pixels you want to scroll. This was incredibly performance intensive, and not even possible with most hardware of that time.

The NES was one of the very first home computers that supported smooth scrolling. Essentially, the hardware had a register you could just write to set the fine (pixel) scroll. If you want your background to be displayed scrolled 120 pixels in from the right, and 22 pixels from the top, you just write "120" and then "22" in order, to the same register. Done deal! The video chip takes care of the rest, running at the same constant speed as it always done.

The IBM PC was by late 80s far behind the gaming power of the NES. It was designed for office work rather than gaming. It was meant to crunch integers and display static im-

ages for word processing and spreadsheet applications. Most PC games around that time are graphic adventure games (King's Quest), static platform games (Prince of Persia) and simulation games (Sim City). Basically, the PC lacked all hardware support for sprites and smooth scrolling.

Then, on December 14th, 1990, a small unknown software company called "Ideas from the Deep" released *Commander Keen in Invasion of the Vorticons* for the IBM PC. It was the first smooth side-scrolling game on a PC, similar like Super Mario Bros on the NES.

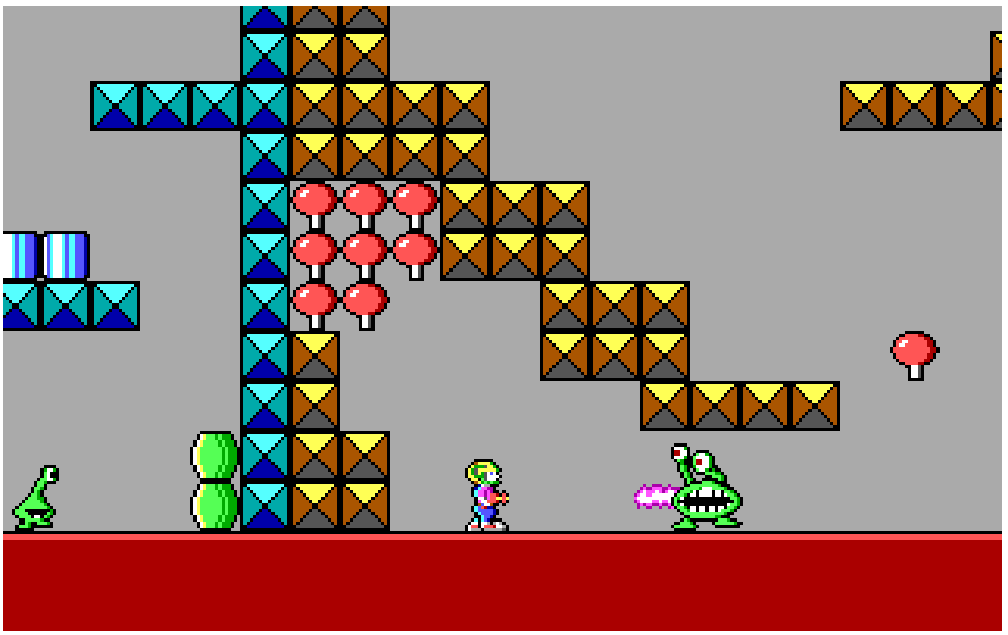


Figure 1.2: Commander Keen in Invasion of the Vorticons

How was this possible on the IBM PC? Many obstacles had to be overcome:

- The first 8086 CPU did not outperform the average home computer in terms of raw power. Only with the release of the 286 CPU the PC started to outperform the market in terms of raw power.
- As stated before, the video system (called EGA) did not support any form of scrolling. It did not even support any form of sprites, which allowed movement of something on the screen by simply updating its `(x,y)` coordinates.
- The video system could not double buffer. It was not possible to have smooth scrolling without ugly artifacts called "tears" on the screen.

- The PC Speaker, the default sound device, could only produce square waves resulting in a bunch of "beeps" which were more annoying than anything else.
- The audio ecosystem was fragmented. Each of the various sound systems had different capabilities and expectations
- The RAM addressing mode was not flat but segmented, resulting in complex and error prone pointer arithmetic.
- The bus was slow and I/O with the VRAM was a bottleneck. It was next to impossible to write a full framebuffer at 70 frames per second

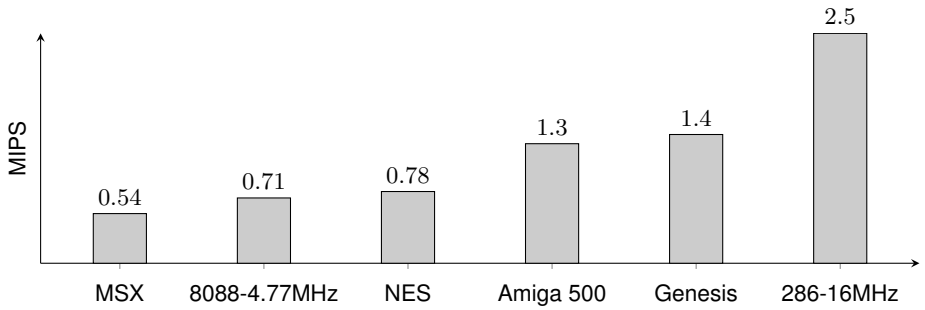


Figure 1.3: Consoles[2]vs PC, CPU comparison with MIPS[3][4].

Overall, it seemed impossible to create any reasonable side-scrolling game on the PC platform. But many around the world did not accept that and tinkered with the hardware to achieve unexpected results. How they did it is the *raison d'être* of this book. I've chosen to divide this book into three chapters:

- Chapter 2: The Hardware. The five components of a PC from 1990.
- Chapter 3: The tools and assets. Which tools are used for game development and how are assets created and structured.
- Chapter 4: The Software. The Commander Keen game engine.

By first showing the hardware constraints, I hope programmers will develop an appreciation for the software and how it navigated obstacles, sometimes turning limitations into advantages.

[2]The MSX uses a Zilog Z80 running at 3.6MHz. The Amiga 500 and Genesis have a Motorola 68000 CPU respectively running at 7.16 MHz and 7.6 MHz. The NES uses a Ricoh 2A03 CPU running at 1.8 MHz.

[3]Million Instructions Per Second.

[4]Gamicus Fandom: https://gamicus.fandom.com/wiki/Instructions_per_second.

The book is based on *Commander Keen in Keen Dreams*, which is developed after the first 3 releases of the game. The reason is that this is the only version where the source code is publicly released. Where needed, I will also explain how the technology changed between the different versions of Commander Keen, but it will be without code examples unfortunately.

Figure 1.4: Commander Keen in Keen Dreams

Chapter 2

Hardware

To study the IBM PC, it is easiest to first break it down to small parts. Five sub-systems form a pipeline: Inputs, CPU, RAM, Video, and Audio.

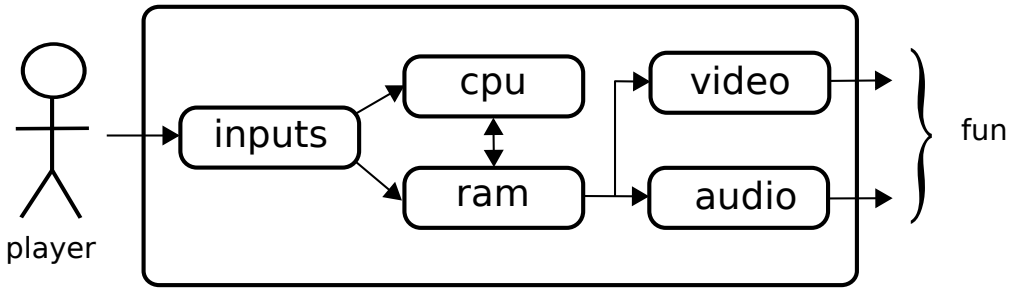


Figure 2.1: Hardware pipeline.

A lot of friction was present since manufacturers had not embraced the gaming industry yet. Parts quality varied from bad, terrible, to downright impossible to deal with.

Stage	Quality
RAM	Bearable
Video	Impossible
Audio	Very Poor
Inputs	Ok
CPU	Very Poor

Figure 2.2: Component quality for a game engine.

2.1 CPU: Central Processing Unit

In 1989 around 15% of the households owned a computer[1]. The performance of these machines was so overwhelmingly determined by the CPU that a PC was referred to not by its brand or GPU[2], but by the main chip inside. If a PC had an Intel 8088 or equivalent, it was called a "XT" . If it had an Intel 80286, it was a "286" or "AT".

2.1.1 Overview

Intel released the 8086 in 1979, which was the first microchip of the succesfull x86 family line. One year later, in 1979, it released the 8088 which was a variant of the 8086. The main difference between the two is that there are only eight data lines for the external data bus in the 8088 instead of the 8086's 16 lines. However, because it retained the full 16-bit internal registers and the 20-bit address bus, the 8088 ran 16-bit software and was capable of addressing a full 1MB of RAM. IBM chose the 8088 over the 8086 for its original PC/XT, because Intel offered a better price for the former and could supply more units.

In 1982 Intel released the 80286 microchip. A typical 8088 chip was running at 4.77Mhz, where the 80286 was running at 8Mhz and later versions at 12.5-16Mhz. The 80286 was employed for the IBM PC/AT, introduced in 1984, and then widely used in most PC/AT compatible computers until the early 1990s. Commander Keen could run on a 8088, but an Intel 286 was recommended.

[1] https://www.statista.com/statistics/184685/percentage-of-households-with-computer-in-the-united-states-since-1984/

[2] There was no GPU yet. The term was coined by Nvidia in 1999, who marketed the GeForce 256 as "the world's first GPU", or Graphics Processing Unit.

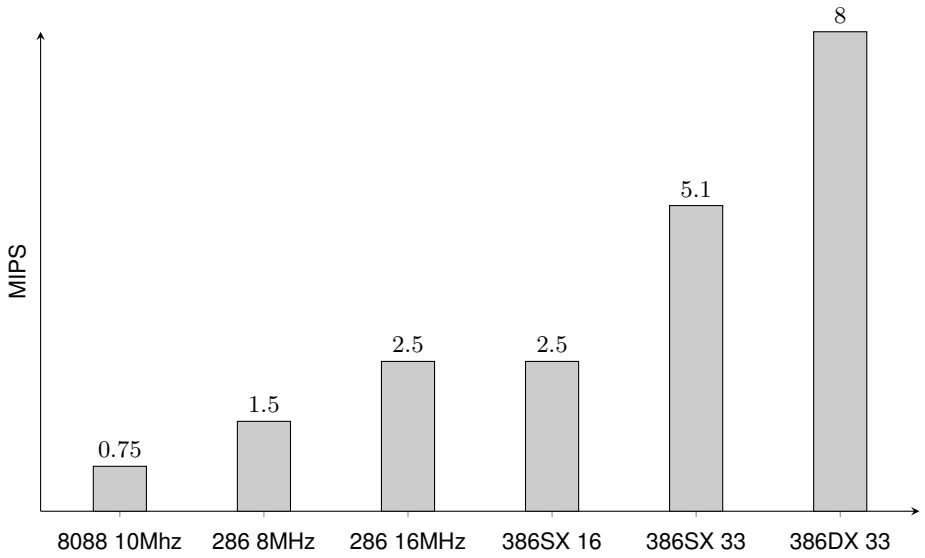


Figure 2.3: Comparison[3]of CPUs with MIPS

Trivia : A modern processor such as the Intel Core i7 3.33 GHz operates at close to 180,000 MIPS.

2.1.2 The Intel 80286

The Intel 80286 chip, first introduced in 1982, is the CPU behind the original IBM PC AT (Advanced Technology). Other computer makers manufactured what came to be known as IBM clones, with many of these manufacturers calling their systems AT-compatible or AT-class computers.

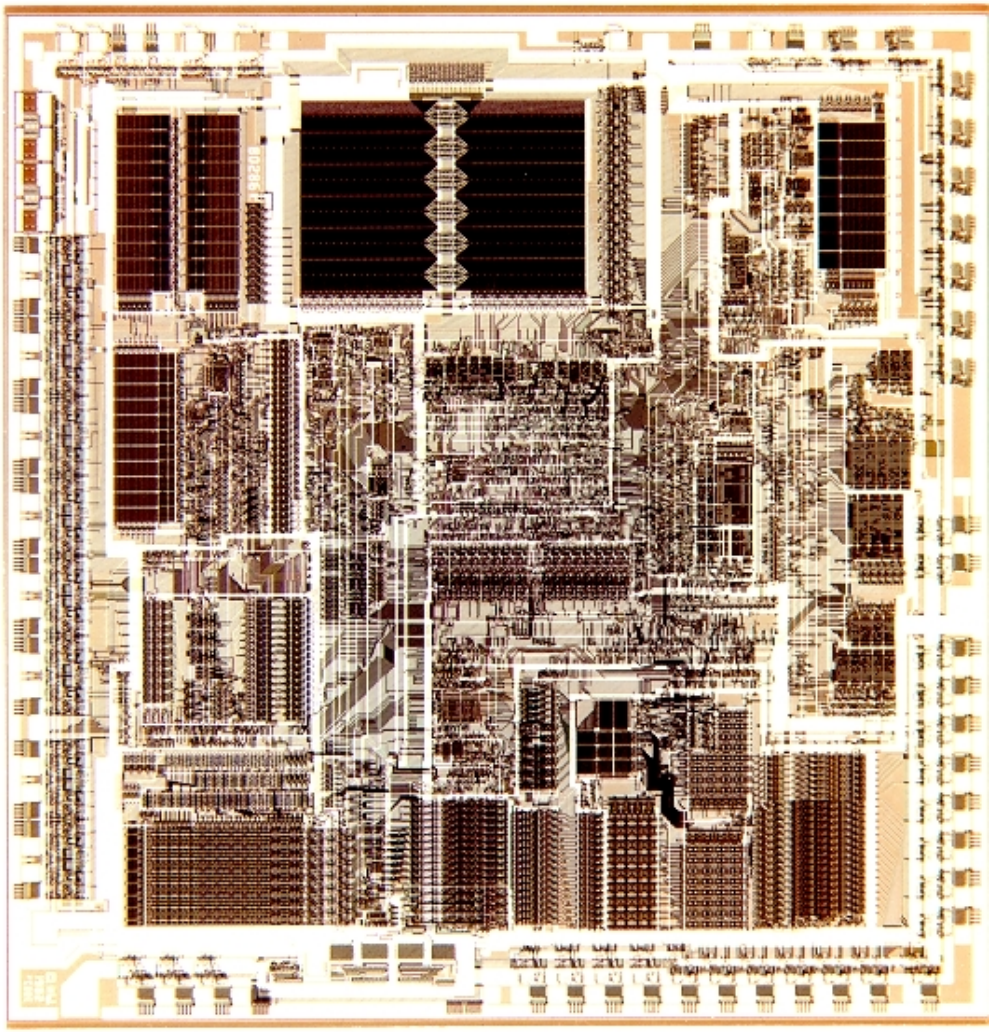
When IBM developed the AT, it selected the 286 as the basis for the new system because the chip provided compatibility with the 8088 used in the PC and the XT. Therefore, software written for those chips should run on the 286. The 286 chip is many times faster than the 8088 used in the XT, and at the time it offered a major performance boost to PCs used in businesses. The processing speed, or throughput, of the original AT (which ran at 6MHz) is five times greater than that of the PC running at 4.77MHz. 286 systems are faster than their predecessors for several reasons. The main reason is that 286 processors are much more efficient in executing instructions. An average instruction takes 12 clock

[3]Roy Longbottom's PC Benchmark Collection: http://www.roylongbottom.org.uk/mips.htm#anchorIntel2.

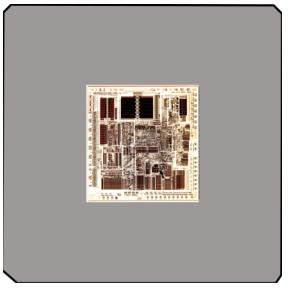
cycles on the 8086 or 8088, but takes an average of only 4.5 cycles on the 286 processor. Additionally, the 286 chip can handle up to 16 bits of data at a time through an external data bus twice the size of the 8088.

The 286 chip has two modes of operation: real mode and protected mode. The two modes are distinct enough to make the 286 resemble two chips in one. In real mode, a 286 acts essentially the same as an 8086 chip and is fully compatible with the 8086 and 8088. In the protected mode of operation, the 286 was truly something new. In this mode, a program designed to take advantage of the chip's capabilities has access to 1GB of memory (including virtual memory). The 286 chip, however, can address only 16MB of hardware memory. A significant failing of the 286 chip is that it cannot switch from protected mode to real mode without a hardware reset (a warm reboot) of the system (It can, however, switch from real mode to protected mode without a reset).

While the 8088 used a 3.0μm process, the 20286 used a 1.5μm process. The smaller process and increased surface (from 33mm^2 to 49mm^2) allowed Intel to pack 134,000 transistors on a 286 chip versus 29,000 on a 8088 chip.




A80286-8
L9409047
©1982,1985



Despite the apparent complexity, the 80286 can be summarized by functional units and a three-stage instruction pipeline.

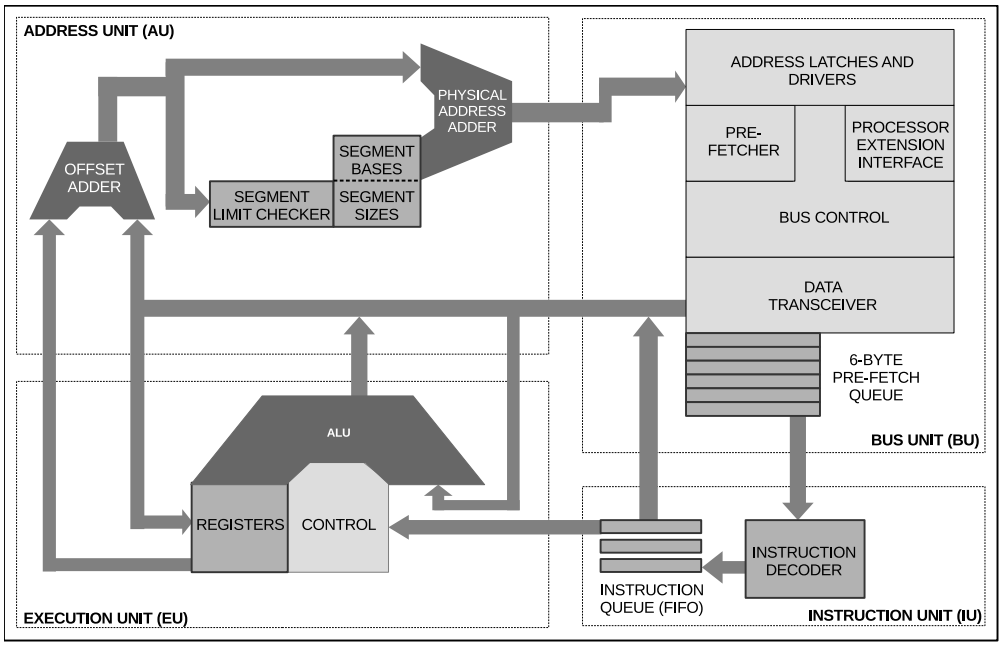
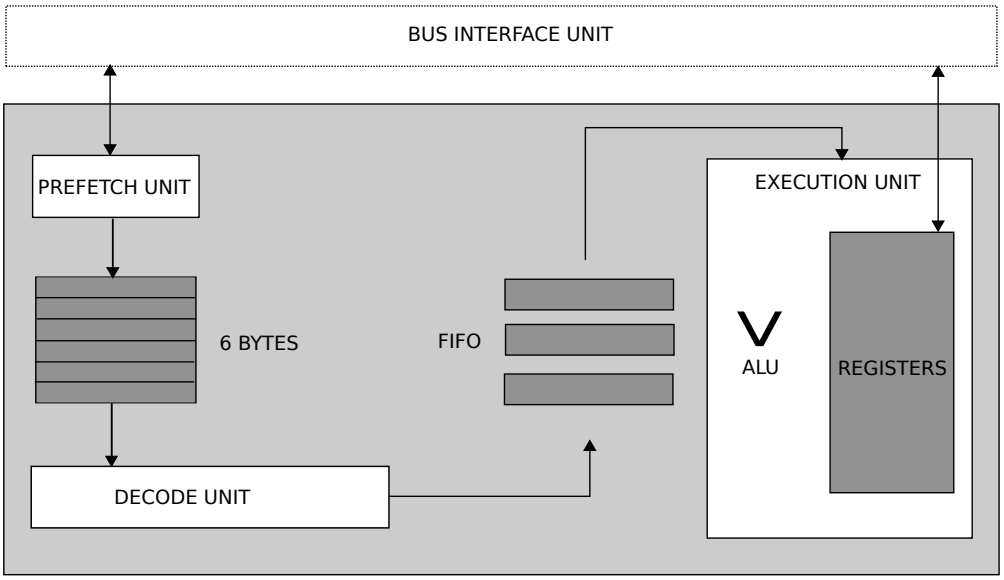


Figure 2.4: Internal block diagram of the 80286 processor

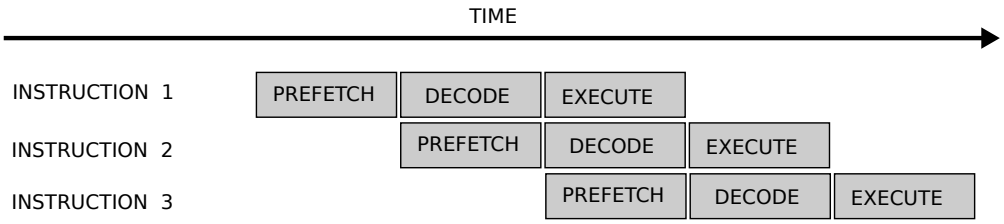
The four functional units can be described by

- **address unit (AU)** is used to determine the physical addresses of instructions and operands which are stored in memory. The address lines derived by AU can be used to address different peripheral devices such as memory and I/O devices.
- **bus unit (BU)** interfaces the 80286 with memory and I/O devices. The bus unit is used to fetch instruction bytes from the memory and stores them in the prefetch queue.
- **instruction unit (UI)** receives instructions from the prefetch queue and an instruction decoder decodes them one by one. The decoded instructions are latched onto a decoded instruction queue.
- **execution unit (EU)** is responsible for executing the instructions received from the decoded instruction queue. The execution unit consists of the register bank, arith-

metic and logic unit (ALU) and control block. The ALU is the core of the EU and perform all the arithmetic and logical operations.



The three units in the execution group form a three stage pipeline: Prefetch, Decode, and Execute. The Prefetch Unit wakes up when the Execution unit is performing but not using the bus and fetches instructions in a 6-byte queue. The prefetcher is linear and cannot predict the result of a branch. As a result, a jump (`JMP`) instruction triggers a flush of the entire pipeline. Instructions go down the pipeline and are decoded by the Decode Unit: the result of the decode operation is stored in a three-element FIFO where it is picked up by the Execution Unit.



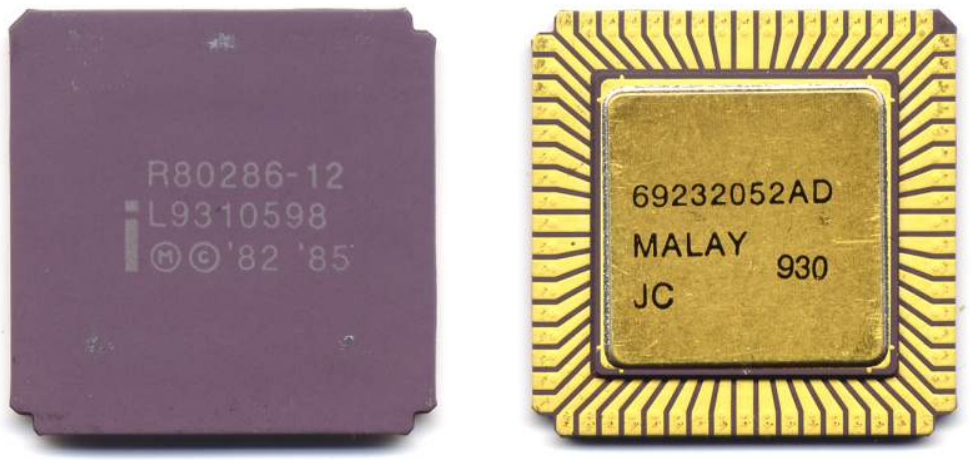


Figure 2.5: The Intel 286, 10mm by 10mm packing 134,000 transistors

From a programming perspective, a 286 CPU can be summarized by the following elements:

- Arithmetic Logic Unit performing `add`, `sub`, `mul` et cetera.
- 14 registers:
  - 16-bit General Purpose Registers: AX, BX, CX, DX
  - 16-bit Index Registers: SI, DI, BP, SP
  - 16-bit Segment Registers: CS, DS, ES, SS
  - 16-bit Status and Control Register
  - 16-bit Program Counter: IP
- A 24-bit address bus for up to 16MB of flat addressable RAM
- Memory Management Unit

Despite its pipeline design, the 286 cannot do an operation in less than two cycles. Even a simple `ADD reg, reg` or `INC reg` takes two clocks. This is due to the absence of a SRAM on-chip cache and a slow decoding unit. Also have a look at multiplications which cost 24 cycles. So as a game developer you really want to avoid many multiplications during game runtime.

Instruction type	Clocks
`ADD reg8, reg8`	2
`INC reg8`	2
`IMUL reg16, reg16`	24
`IDIV reg16, reg16`	28
`MOV [reg16], reg16`	5
`OUT [reg16], reg16`	3
`IN [reg16], reg16`	5

Figure 2.6: 286 instruction costs[4]

2.2 RAM

The first CPUs in the Intel x86 family were designed in 1976. At a time when RAM was very expensive, the 8086 and 8088 had 16-bit registers with a 20-bit-wide address bus capable of addressing 1MiB[5] of RAM. It is difficult to stress how big 1MiB of RAM was in the 70's but as an example the Apple II and the Commodore 64 both shipped with 64KiB[6] which was enough to write and run amazing things. Sixteen-bit registers and a 20-bit address bus were plenty even though programming was difficult and required combining two registers to build a pointer.

By 1986, hardware had gotten cheaper and Intel made a departure from the old architecture with its 286. This new CPU could be put in what is called "protected mode" featuring a 24-bit-wide address bus for up to 16 MiB of flat RAM protectable with a MMU[7]. To make sure old programs could still run, the 286 processor could be put in "real mode" which replicates how the Intel 8086 and 8088 operated: 16-bit registers, 20-bit address bus giving 1MiB addressable RAM with segmented addressing.

For compatibility reasons all PCs have to start in real mode. You may assume that programmers of the late 80s promptly switched the CPU to protected mode to unleash the full potential of the machines and ditch the 20-year-old real mode. Unfortunately, there was a major obstacle: the operating system MS-DOS by Microsoft Corporation.

[4] Intel 80286 programmer's reference manual - 1987.
[5] This book uses IEC notation where MiB is 2^{20} and MB is 10^6.
[6] This book uses IEC notation where KiB is 2^{10} and KB is 10^3.
[7] Memory Management Unit

2.2.1 DOS Limitations

Microsoft Corporation highly valued the applications running on their operating systems. As a business priority, they were adamant to never break anything with a new system[8]. Since many applications were written during the 80s on machines having only real mode, DOS 3.3[9] and even the later release DOS 4.01[10] kept running that way and as a result its routines and system calls were incompatible with protected mode. This created an awkward situation where the de-facto operating system delivered with every machine sold prevented programmers from using the machine at its full potential. Developers were forced to ignore all the features of a 1984 CPU and instead use it like a very fast Intel 8086 CPU from 1976. They were thus limited to the following characteristics:

- ALU
- 14 registers:
  - 16-bit General Purpose Registers: AX, BX, CX, DX
  - 16-bit Index Registers: SI, DI, BP, SP
  - 16-bit Program Counter: IP
  - 16-bit Segment Registers: CS, DS, ES, SS
  - 16-bit Status Register
- Up to 1MiB of RAM

Trivia : Only a small amount of software that took advantage of the 286 chip was sold until Windows 3.0 offered standard mode for 286 compatibility; by that time, the hottest-selling chip was the 386. Still, the 286 was Intel's first attempt to produce a CPU chip that supported multitasking, in which multiple programs run at the same time.

2.2.2 The Infamous Real Mode: 1MiB RAM limit

With protected mode unavailable, 1990 developers programmed like it was 1976: with a 20-bit-wide address bus offering only 1MiB of addressable RAM. Regardless how much memory was installed on the machine, only 1MiB could be addressed. To top it all off, addressing had to be done by combining two 16-bit registers. One was the segment, the other an offset within that segment. Hence the name: '16-bit segmented programming'.

[8] "Tales of Application Compatibility", Old New Thing by Raymond Chen.
[9] Released in April 1987.
[10] Released in July 1989

The memory layout is as follows:

- From 00000h to 003FFh : the Interrupt Vector Table.
- From 00400h to 004FFh : BIOS data.
- From 00500h to 005FFh : command.com+io.sys.
- From 00600h to 9FFFFh : Usable by a program (about 620KiB in the best case).
- From A0000h to FFFFFh : UMA (Upper Memory Area): Reserved to BIOS ROM, video card and sound card mapped I/O.

Out of the original 1024KiB, only 640KiB (called Conventional Memory) was accessible to a program. 384KiB was reserved for the UMA and every single driver installed (`.SYS` and `.COM`) took away from the remaining 640KiB.

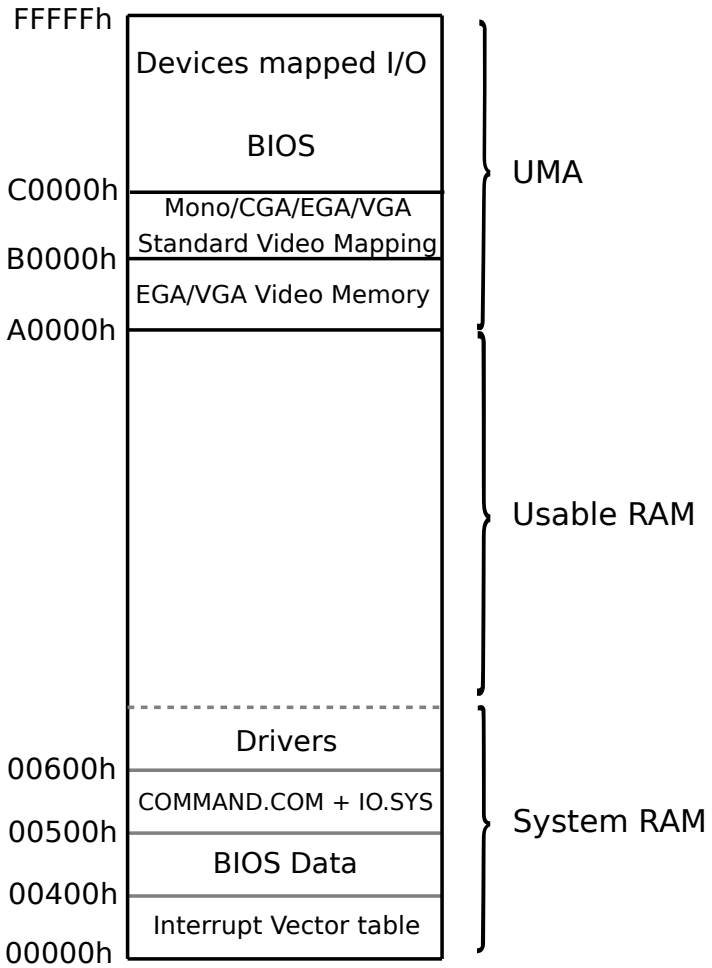


Figure 2.7: First 1MiB of RAM layout.

2.2.3 The Infamous Real Mode: 16-bit Segmented addressing

With a 20-bit address bus and registers too small to contain a whole address (16-bit wide), Intel had to come up with an addressing system. Their solution was to combine two 16-bit registers, one designating a segment and the other an offset within that segment.

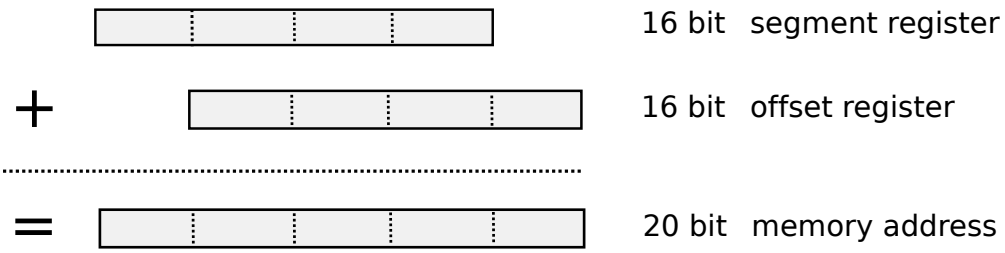


Figure 2.8: How registers are combined to address memory.

There are two kinds of pointers: `near` and `far`. A `near` pointer is 16 bits and considered *fast* because it can be used as is (but it only allows a `jmp` in the current code segment). A `far` pointer can access anything and allows a `jmp` anywhere but is slower since a 16-bit segment register has to be shifted left 4 bits and combined with the other 16-bit-offset register to form a 20-bit address.

That may not sound too bad, but in practice this segmented addressing leads to many issues. The least problematic is about the language. Since C was invented on a flat memory machine, it had to be augmented by PC compiler manufacturers. That is how the `near` and `far` keywords came into existence. Macro `MK_FP` built them and `FP_SEG/FP_OFF` accessed individual components. `libc` is also "different": `malloc` returns a `near` pointer and therefore can only allocate up to 64KiB. To get more than 64KiB, `farmalloc` is needed.

The larger issue is that two pointers referring to the same address can fail an equality test. In this model, the 1MiB of RAM is divided in 65536 paragraphs by the segment pointer. A paragraph is 16 bytes but an offset can be up to 65536 bytes which results in many overlaps. This can be explained with the following examples.

Pointer A defined as:

```
    0000 0000 0000 0000        Segment   16 bits
  +      0000 0001 0010 0000   Offset    16 bits
============================
    0000 0000 0001 0010 0000   Address   20 bits
```

Pointer B defined as:

```
    0000 0000 0001 0000        Segment   16 bits
  +      0000 0000 0010 0000   Offset    16 bits
============================
    0000 0000 0001 0010 0000   Address   20 bits
```

Pointer C defined as:

```
    0000 0000 0001 0010        Segment   16 bits
  +      0000 0000 0000 0000   Offset    16 bits
============================
    0000 0000 0001 0010 0000   Address   20 bits
```

As defined, A, B, and C all point to the same memory location however they will fail a comparison test.

```
#include <stdio.h>
#include <dos.h>

int main(int argc, char** argv){

  void far *a = MK_FP(0x0000, 0x0120);
  void far *b = MK_FP(0x0010, 0x0020);
  void far *c = MK_FP(0x0012, 0x0000);

  printf("%d\n",a==b);
  printf("%d\n",a==c);
  printf("%d\n",b==c);
}
```

Will output:

```
0
0
0
```

With this system, pointer arithmetic must also receive careful consideration. A `far` pointer increment only increments the offset, not the segment. If you iterate on an array larger than 64KiB you will end up wrapping around. You could use yet another type of pointer `int huge*` to make pointer arithmetic work beyond 64KiB but really, nobody wants to go there.

Trivia : As of 2017, more than thirty five years after the introduction of the 8086, in the name of backward compatibility, all PCs in the world still start in real mode. A bootloader

switches them to protected mode, loads the kernel, and then actual startup can begin.

2.3 Video

PCs were connected to CRT monitors: big, heavy, small diagonal, cathode ray-based, curved-surface screens. Most had a 14" diagonal with a 4:3 aspect ratio.

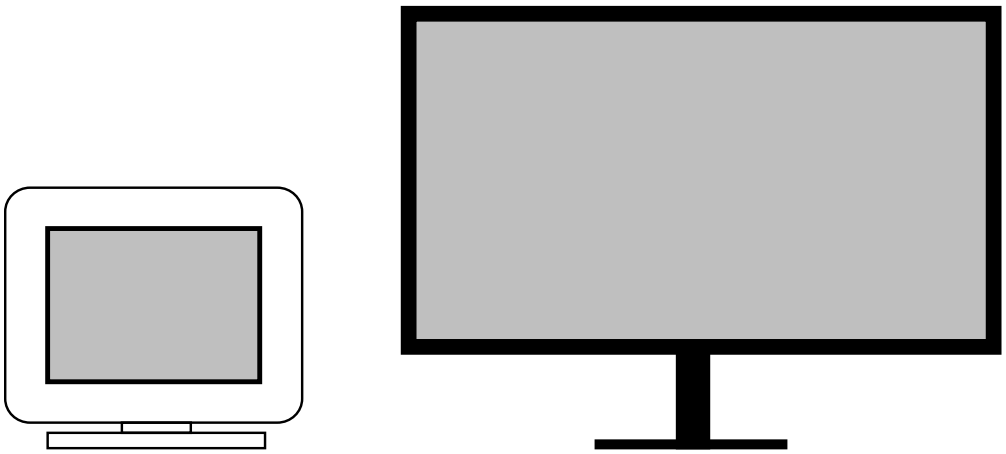


Figure 2.9: CRT (left) vs LCD (right)

To give you an idea of the size and resolution, figure 2.9 shows a comparison between a 14" CRT from 1990 (capable of a resolution of 640x200) and a 30" Apple Cinema Display from 2014 (capable of a resolution of 2560x1600).

Trivia : Despite their difference of capabilities, both monitors are the same weight: 27.5 pounds (12 kg).

2.3.1 CRT Monitor

All standard PC monitors use a raster-scan display to create the image. In a raster-scan display, the position of the electron beam is continually sweeping across the surface of the tube. The tube's surface is coated with phosphors that glow when struck by electrons (and for a short time thereafter), and, of course, the beam may be turned on in order to light a phosphor or off to leave it black.

The electron beam scans the phosphor-coated screen from left to right and top to bottom. The period during which the beams return to the left is known as the horizontal retrace. During most of the retrace, the guns must be turned off to prevent writing in the active display area (the area which contains the actual character and/or graphics data); this is known as horizontal blanking.

The area immediately surrounding the display area, in which the beam may be turned on during the retrace interval, is called the overscan (or border). The active display area is the portion of the screen that contains characters and/or graphics. These components of the scan are shown in simplified form in Figure 2.10.

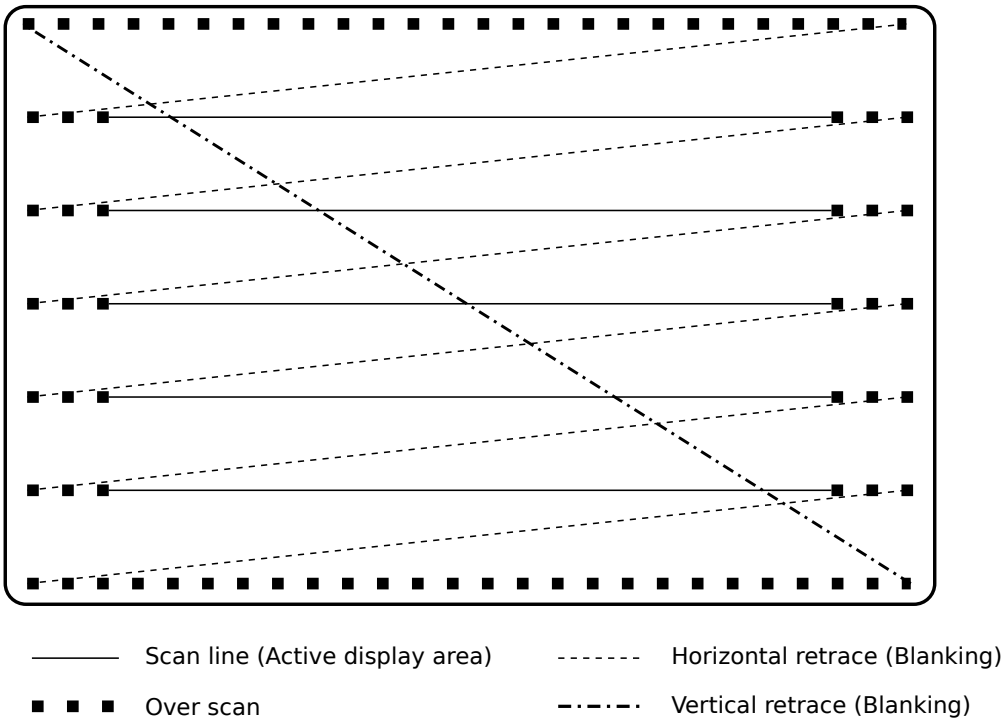


Figure 2.10: Simplified CRT monitor scan.

After a horizontal scan has been completed, the beam is moved to the next line during the horizontal retrace. This sequence continues until the last line, at which point the vertical retrace begins. The vertical retrace is similar to the horizontal retrace; the electron beam may be enabled through a small overscan area and then turned off (vertical blanking) as the beam returns to the top left comer of the screen.

If the vertical refresh is too slow, the display will flicker. Most people can detect flicker when the refresh rate drops below 60 Hz, and thus most displays use vertical refresh frequencies of about 60Hz (EGA) to 70Hz (VGA).

2.3.2 History of Video Adapters

The Monochrome Display Adapter (MDA) was released in 1981 with the IBM PC 5150. It offered two colors, allowing 80 columns by 25 lines of text. While not great, it was standard on every PC. Many other systems followed over the years, each of them preserving backward compatibility.

Name	Year Released	Memory	Max Resolution
MDA (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[11]
Hercules	1982	64KiB	720x348
CGA (**C**olor **G**raphics **A**dapter)	1981	16KiB	640x200
EGA (**E**nhanced **G**raphics **A**dapter)	1985	64KiB	640x350
VGA (**V**ideo **G**raphics **A**rray)	1987	256KiB	640x480

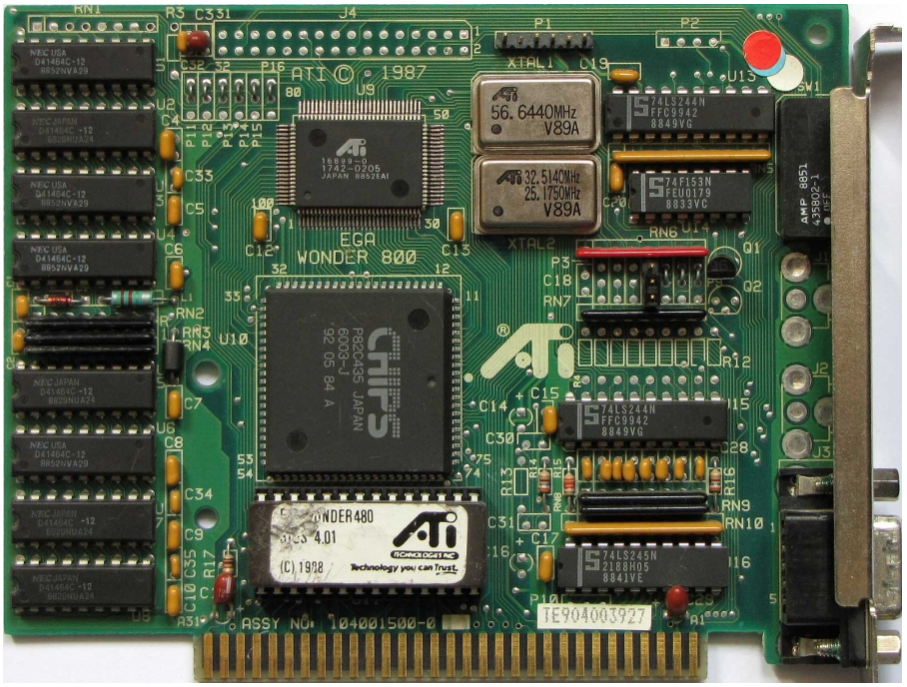
Figure 2.11: Video interface history.

Each iteration added new features and by 1990 the predominant graphic system was EGA, although the VGA system was rapidly becoming the new standard. All video cards installed on PCs had to follow the standard set by IBM. The universality of that system was a double-edged sword. While developers had to program for only one graphic system, there was no escaping its shortcomings.

The EGA palette allows 16 colors to be used simultaneously, and it allows substitution of each of these colors with any one from a total of 64 colors, at a resolution of 640 x 350.

Below an ATI EGA Wonder 800 (8-bit ISA). The eight chips on the left of the card form the VRAM where the framebuffers are stored[12].

[11] Text mode only.



[12] Each VRAM chip from this ATI EGA cards can store 32KiB, accounting for a total of 256KiB VRAM.

2.3.3 EGA Architecture

EGA can be summarized as three major systems made of input, storage, and output:

- The Graphic Controller and Sequence Controller controlling how EGA RAM is accessed (the CPU-VRAM interface)
- The framebuffer (the VRAM) made of four memory banks with a minimum of 16KiB (rather than one bank of 64KiB). Via memory expansion each memory bank could be upgraded to 32KiB or 64KiB (resulting in 128KiB or 256KiB total VRAM). The original model from IBM came with 16KiB per memory bank, but almost all other EGA cards were equiped with the full 64KiB per memory bank. For the remainder of this book we will discuss only 256KiB EGA operations.
- The CRT Controller and the Attribute Controller taking care of converting the palette-indexed framebuffer to RGB and then to digital TTL[13] signal for display

Trivia : In the 1980's integrated video DACs[14] were expensive and difficult to embed into custom chips. Most home computers with RGB output used TTL for digital output. With the introduction of VGA the DAC became the standard.

The most surprising part of the architecture is obviously the framebuffer. Why have four small fragmented banks instead of one big linear one?

The main reason was RAM latency and the need for minimum bandwidth. A CRT running at 60Hz and displaying 640x350 in 16 colors needs a pixel every $\frac{1}{640*350*60} = 74$ nanosecond. At this resolution, one pixel is encoded with 4 bits. Each nibble is translated to a RGB color via the TTL. So that means it requires one byte every 148 nano-seconds.

Unfortunately, RAM access latency was 200ns - not nearly fast enough[15] to refresh the screen at 60hz, so the TTL would starve. If latency could not be reduced, the throughput could still be improved by reading from four banks at a time. Reading in parallel gave an amortized RAM latency of 200/4 = 50ns, which was fast enough.

Keep in mind that this architecture reduced the penalty of read operations, but plotting a pixel in the framebuffer with a write operation was still slow. Writing to the VRAM as little as possible was crucial to maintaining a decent framerate.

[13] Transistor Transistor Logic
[14] Digital to Analog Converter
[15] Computer Graphics: Principles and Practice 2nd Edition, page 168.

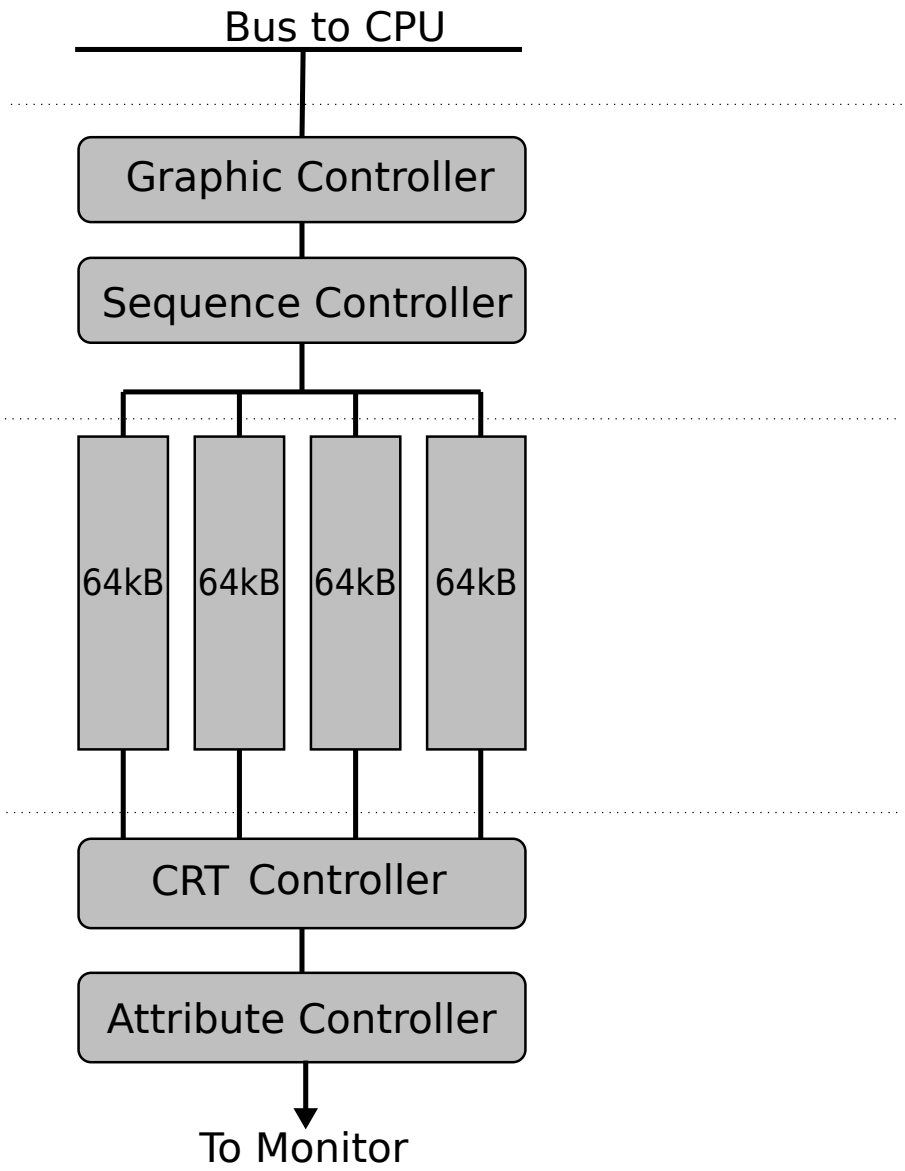


Figure 2.12: EGA Architecture.

2.3.4 EGA Planar Madness

Four memory banks grant enough throughput to reach high resolutions at 60Hz. However, the price for this solution is complexity of programming.

The first problem with this design is that it is unintuitive. There is no linear framebuffer and figuring out which byte corresponds to which pixel on screen is difficult.

This type of architecture is called "planar". Each plane is like a black-and-white image that stores information about a single colour. For EGA there are 4 planes. Each pixel contains 1 bit per plane, in total this results in 2^4=16 colors. Each of these banks is mapped to the same UMA memory address. This layout is better explained with a drawing.

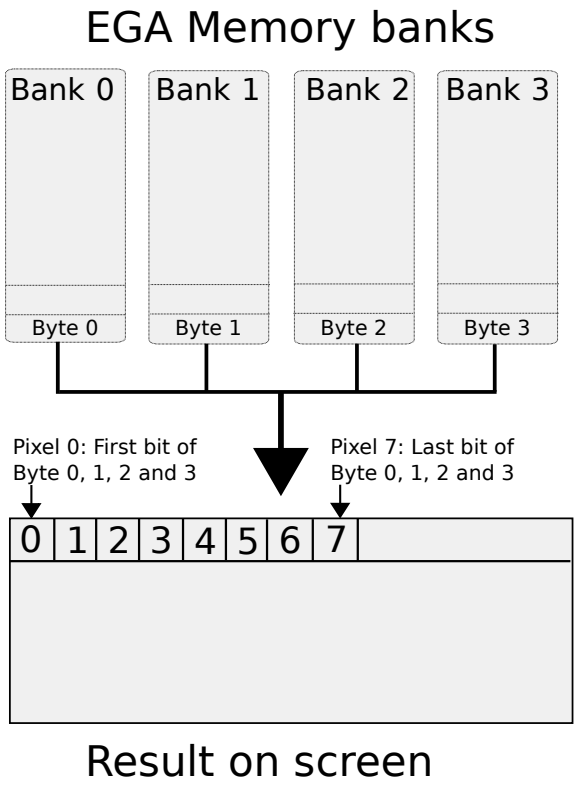


Figure 2.13: EGA mode 0Dh, How bank layout appears on screen.

In order to configure this mess of planes and the controllers, 50 poorly documented internal registers must be set. Needless to say few programmers dove into the internals of the EGA.

Figure 2.12, which described the architecture, was actually deceptively simplified. Figure 2.14 shows how IBM's reference documentation explained the EGA. The maze of wire

showcases well the actual complexity of the system.

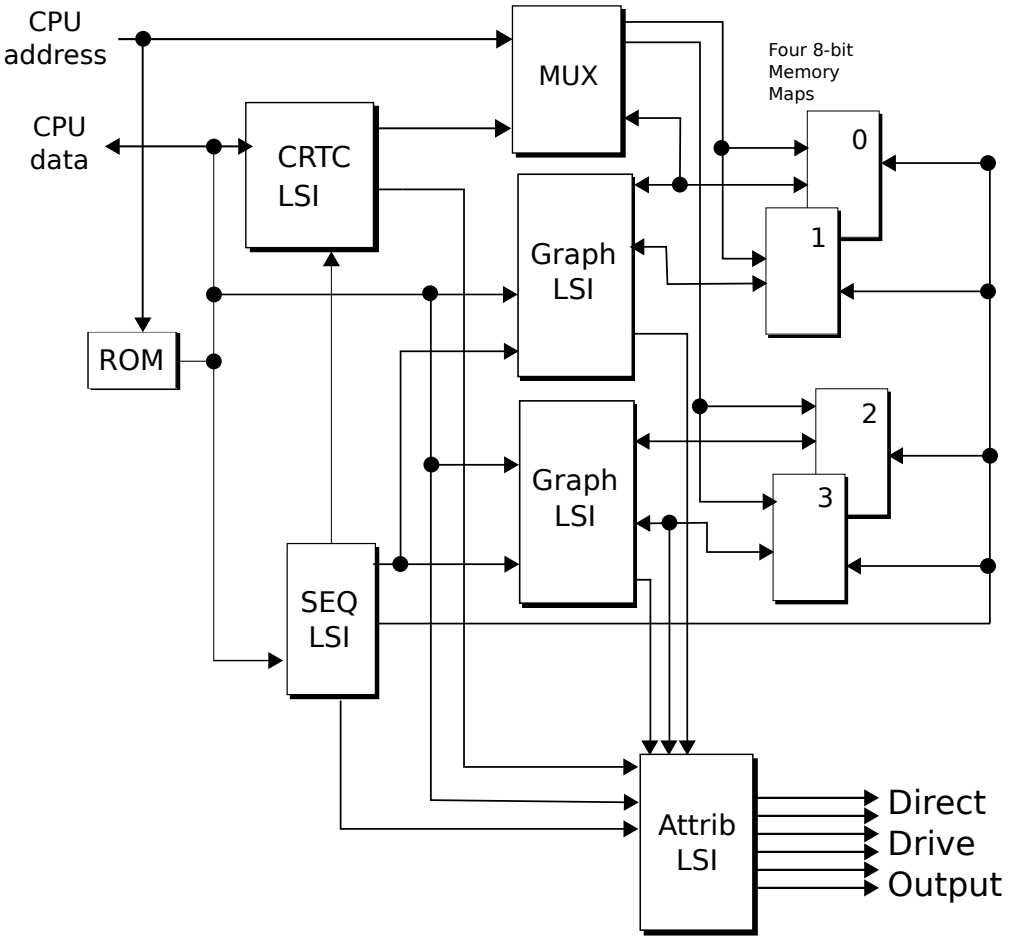


Figure 2.14: IBM's EGA Documentation.

To compensate for the complexity, IBM provided a routine to initialize all the registers via one BIOS call. One mode can be selected out of 12 available with an associated resolution, number of colors, and memory layout.

2.3.5 EGA Modes

The BIOS can be called to configure the EGA as follows.

Mode	Type	Format	Colors	RAM Mapping	Hz
0	text	40x25	16 (monochrome)	B8000h	60
1	text	40x25	16	B8000h	60
2	text	80x25	16 (monochrome)	B8000h	60
3	text	80x25	16	B8000h	60
4	CGA Graphics	320x200	4	B8000h	60
5	CGA Graphics	320x200	4 (monochrome)	B8000h	60
6	CGA Graphics	640x200	2	B8000h	60
7	MDA text	9x14	3 (monochrome)	B0000h	60
0Dh	EGA graphic	320x200	16	A0000h	60
0Eh	EGA graphic	640x200	16	A0000h	60
0Fh	EGA graphic	640x350	3	A0000h	60
10h	EGA graphic	640x350	16	A0000h	60

Figure 2.15: EGA Modes available.

2.3.6 EGA compatibility with 200-line CGA modes

The EGA uses a female nine-pin D-subminiature (DE-9) connector for output, identical to the CGA connector, and the signal standard and pinout is backwards-compatible with CGA, allowing EGA monitors to be used on CGA cards and vice versa. When operating in 200-line CGA modes, the EGA card is fully backwards compatible with a standard CGA monitor. Thereby it was able to show all 16 CGA colors simultaneously, instead of only 4 colors when using a CGA card.

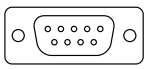


Figure 2.16: EGA Port

Although EGA supported high resolutions like 640x350 pixels, it required an expensive high resolution EGA monitor. For reasons of the compatibility with CGA and avoid acquiring an expensive EGA monitor most game programmers used mode `0Dh`, using the 320x200 resolution with 16 colors.

2.3.7 EGA Color Palette

For each pixel a number index is derived from the 4 planes, representing a color number. The default color palette are all 16 CGA colors, but it allows substitution of each of these

colors with any one from a total of 64 colors.

When calculating the intended value in the 64-color EGA palette, the binary number of the intended entry is of the form "rgbRGB" where a lowercase letter is the least significant bit of the channel intensity ($\frac{1}{3}$ color intensity) and an uppercase letter is the most significant bit of intensity ($\frac{2}{3}$ color intensity). The more intensity, the brighter the color is. For example, `02h` will produce green, `10h` will produce dim green and `12h` will produce bright green. The color magenta is created by setting both "R" and "B", which is color code `05h`. Each of the 16 color indexes could be reassigned to one color from the "rgbRGB" palette.

However, standard EGA monitors did not support use of the extended color palette in 200-line modes. The monitor cannot distinguish between being connected to a CGA card or being connected to an EGA card in a 200-line mode. Compared to CGA, EGA redefines some pins of the DE-9 connector to carry the extended color information. If the monitor were connected to a CGA card, these pins would not carry valid color information, and the screen might be garbled if the monitor were to interpret them as such. For this reason, standard EGA monitors will use the CGA pin assignment in 200-line modes so the monitor can also be used with a CGA card. To keep CGA compatibility most video games did not take advantage of the color palette and kept the 16 standard CGA colors.

Index Number	Color	rgbRGB	Decimal
00h	Black	000000	0
01h	Blue	000001	1
02h	Green	000010	2
03h	Cyan	000011	3
04h	Red	000100	4
05h	Magenta	000101	5
06h	Brown	010100	20
07h	Ligh grey	000111	7
08h	Dark grey	111000	56
09h	Bright blue	111001	57
0Ah	Bright green	111010	58
0Bh	Bright cyan	111011	59
0Ch	Bright red	111100	60
0Dh	Bright magenta	111101	61
0Eh	Yellow	111110	62
0Fh	White	111111	63

Figure 2.17: Default EGA 16-color palette

2.3.8 EGA Programming: Memory Mapping

To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in figure 5.3. In mode `0Dh` for example, the VRAM is mapped from 0xA0000 to 0xAFFFF. One of the first questions to come to mind is "How can I access 256KiB of RAM with only 64KiB of address space?" The answer is "bank switching" as summarized in figure 2.18. Write and Read operations are routed based on a mask register indicating which bank should be read or written to.

The most commonly considered mode for game programming is mode `0Dh`. It offers a resolution of 320x200 at 60hz with 16 colors. Each pixel is encoded in 4 bits (a nibble) spread across the four banks.

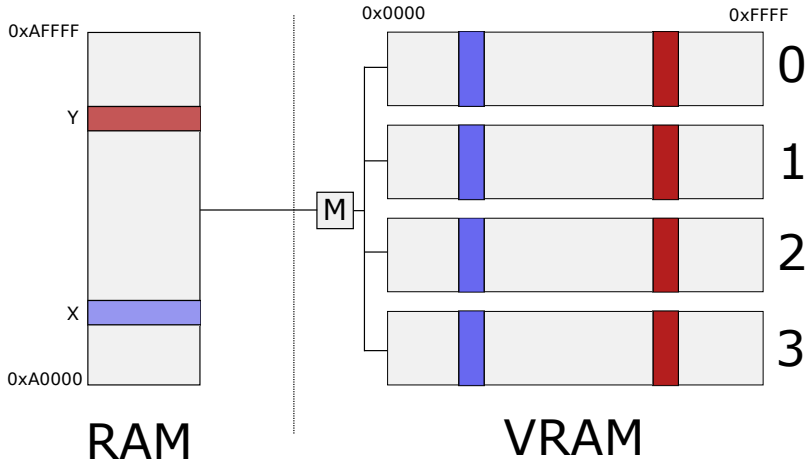


Figure 2.18: Mapping PC RAM to EGA VRAM banks.

To write the color of the first pixel, a developer has to write the first bit of the nibble in plane 0 (R), the second in plane 1 (G), the third in plane 2 (B) and the fourth in plane 3 (I). The CRT Controller then reads 4 bytes at a time (one from each plane) resulting in 8 pixels on screen. So in figure 2.19 the first pixel has color magenta(`05h`), second pixel dark grey (`08h`) and third pixel bright yellow(`0Eh`).

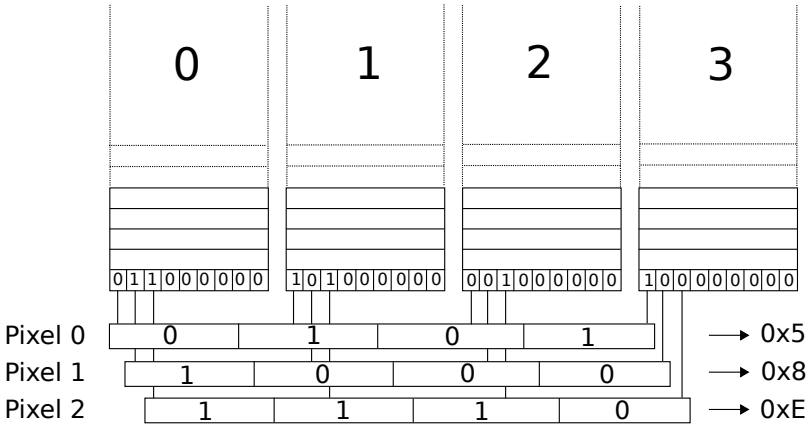


Figure 2.19: EGA bank layout

2.3.8.1 Setup

To setup the EGA in Mode `0Dh` using the BIOS is incredibly easy. It can be done with only two instructions:

```
_AX = 0xd ; AH=0 (Change video mode), AL=0Dh (Mode)
geninterrupt (0x10) ; Generate Video BIOS interrupt
```

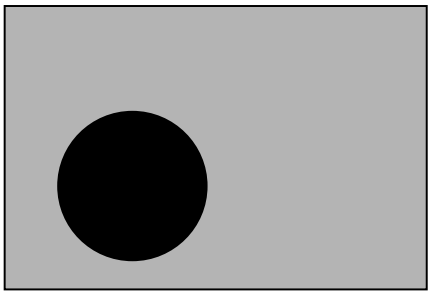
The `geninterrupt (0x10)` instruction is a software interrupt caught by the BIOS routine in charge of graphic setup. It looks up the `ax` register, which can be set in the Borland Compiler by `_AX`, to setup all EGA registers with the corresponding mode.

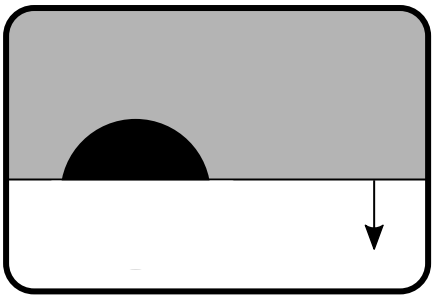
After the EGA is initialized one can write to the mapped memory at `0xA0000`. This can be demonstrated with a code sample; here is some code to clear the screen to black.

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

void ClearScreen(void){
    int i;
    _AX = 0xd;
    geninterrupt (0x10);

    for (i=0 ; i < 320*200 ; i++)
         EGA[i] = 0x00;
}
```

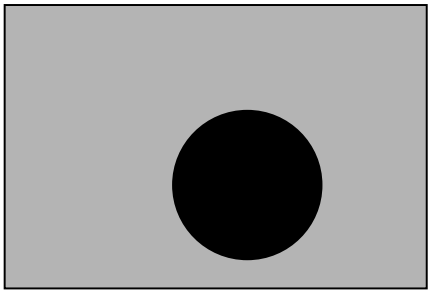
2.3.9 The Importance of Double-Buffering

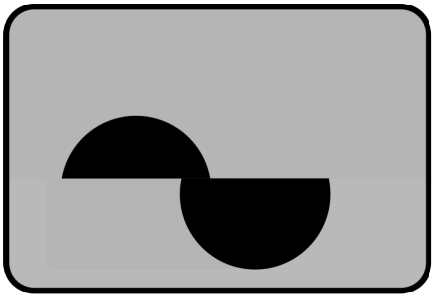
Double buffering has been mentioned often while describing the hardware, but so far we have not reviewed why it is paramount to achieving smooth animation. With only one buffer the software has to work at exactly the frequency of the CRT (60Hz). Otherwise a phenomenon known as "tearing" appears. Let's take the example of an animation rendering a circle moving from the left to the right:





In this example the CPU has finished writing the framebuffer (on the left) and the CRT's (on the right) electron beam has started to scan it onto the screen. At this point in time the electron beam has scanned half the framebuffer, so the circle has been partially drawn on the screen.





If the CPU is faster than the frequency of the CRT (60Hz), it can write the framebuffer again, before the scan is completed. This is what happened here. The next frame was drawn with the circle moved to the right. The electron beam did not know that and kept on scanning the framebuffer. The result on screen is now a composite of two frames. It looks like two frames were torn and taped back together. Hence the name "tearing".

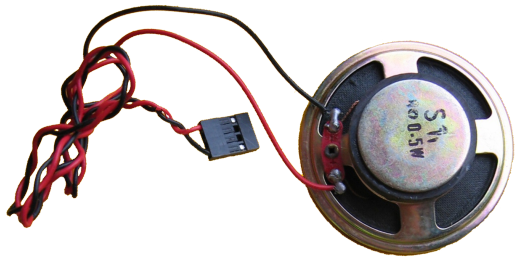
With two buffers (a.k.a double buffering) the CPU can start writing in the second framebuffer without messing with the framebuffer being scanned to the screen[16]. No more tearing!

Note that creating 320x200 picture with 16 colors on the screen requires 8KiB of VRAM (4 planes, each 2KiB).

[16]Now the CPU speed is capped by the CRT refresh rate. Triple buffering can solve this at the price of frame latency.

2.4 Audio

PCs came equipped with a silver-dollar-sized beeper commonly known as a "PC Speaker", capable of generating a square wave via 2 levels of output.



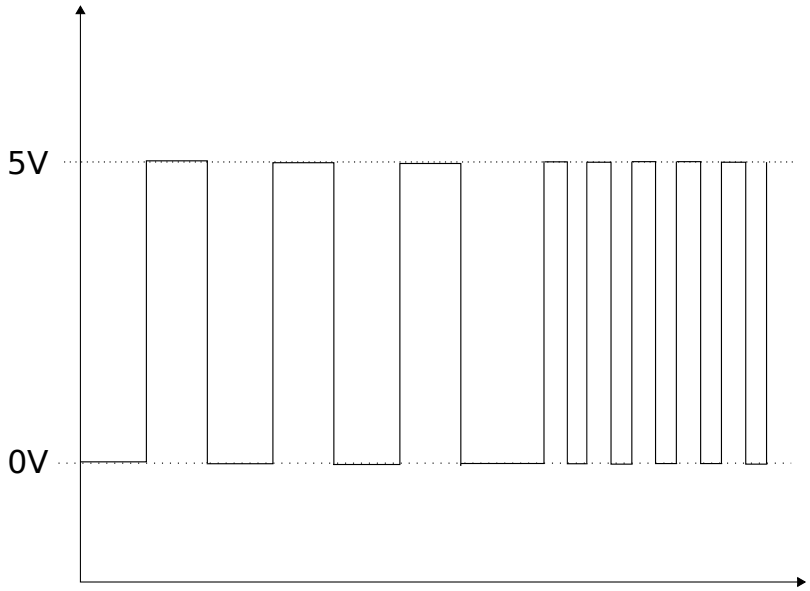


Figure 2.20: Two beeps of different frequencies generated via PC Speaker.

To this day, the PC speaker is the first output device to be activated during the boot process. The purpose of this primitive loudspeaker is to signal hardware problems with beep codes. It was intended to remain silent after a successful boot.

Beep Code	Meaning
No Beeps	Short, Bad CPU/MB, Loose Peripherals
One Beep	Everything is normal
Two Beeps	POST/CMOS Error
One Long Beep, One Short Beep	Motherboard Problem
One Long Beep, Two Short Beeps	Video Problem
One Long Beep, Three Short Beeps	Video Problem
Three Long Beeps	Keyboard Error
Repeated Long Beeps	Memory Error
Continuous Hi-Lo Beeps	CPU Overheating

However, square waves are not useful for producing anything pleasant. Some people saw a potential market and companies began manufacturing what were known as "sound cards". Users could buy these separately and insert them into one of the machine's ISA slots. These cards could be connected to real audio speakers via 3.5mm jacks and tremendously improved sound capabilities. In 1990, there were three cards on the market:

- AdLib music card
- SoundBlaster 1.0
- Disney Sound Source

Although adoption was growing (Creative would go on to sell one million SoundBlaster cards in 1991), the majority of PCs had no sound card which once again presented a huge problem for game developers. Commander Keen 1-3 did only support the PC speaker, only after introduction of Keen Dreams soundcards were supported.

2.4.1 AdLib

AdLib's music card was first on the market. The company was founded in 1988 by Martin Prevel, a former professor of music from Quebec. After an initial struggle to get game developers to use their card (the SDK was $300), AdLib managed to convince Taito, Velocity, and Sierra On-Line to support their hardware. Sierra in particular did much to increase adoption with King's Quest IV selling close to 3 million copies. Soon after, all games supported the "music card".

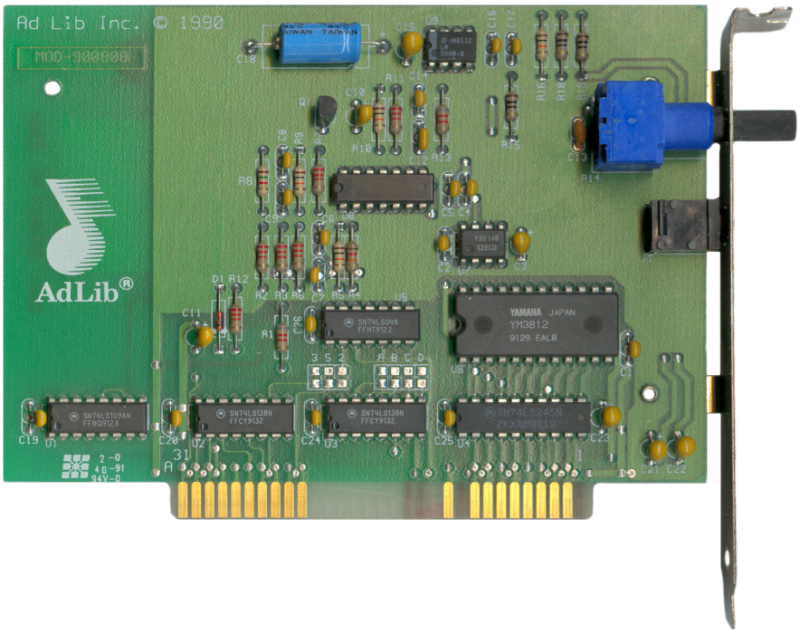


Figure 2.21: An AdLib sound card. Notice the big YM3812 chip and the 8-bit ISA connector.

Equipped with a Yamaha YM3812, also known as the OPL2, the card can produce 9 channels of sound, each capable of simulating an instrument. Based on FM synthesis, the channels were limited but allowed for pleasant music.

Trivia : Canadian companies, and especially those from Quebec, were prevalent in the early 90s due to their technological prowess. AdLib manufactured Sound Cards, Matrox made a killing with its Millenium Graphics Card, and Watcom sold the best DOS C compiler[17]. ATI[18] would later emerge as a major GPU innovator in the 2000s.

2.4.2 Sound Blaster

The Sound Blaster 1.0 (code named "Killer Kard"), was released in 1989 by Creative. It was a smart product which was clearly targeting AdLib's dominant position.

[17] Watcom's compiler was so good id would use it to compile Doom.
[18] History would repeat itself in the late 90s in the field of graphic cards: Nvidia vs ATI.

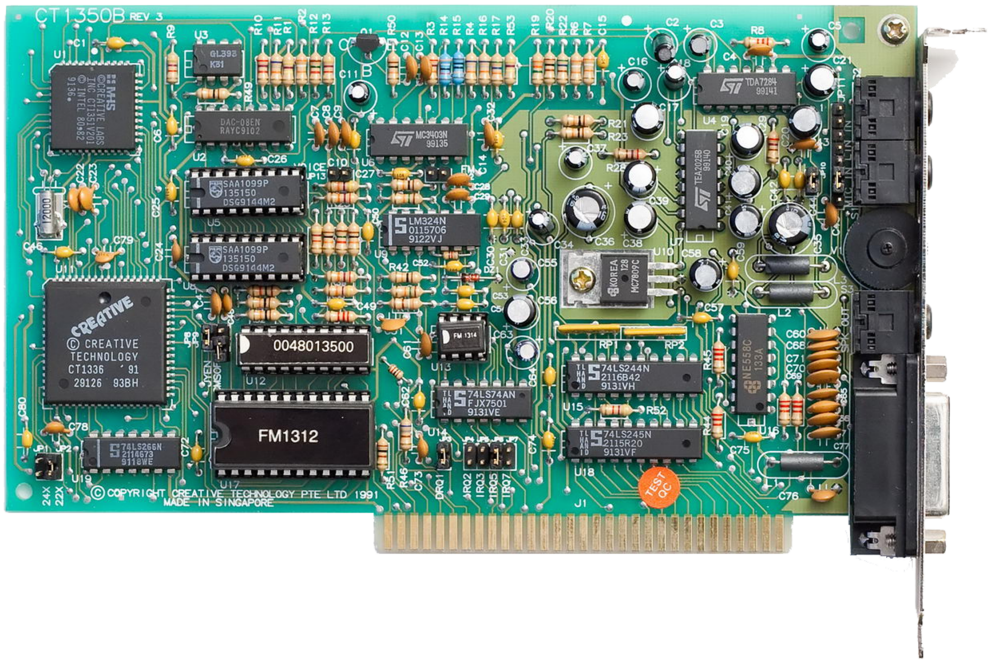


Figure 2.22: A SoundBlaster (v1.2).

Not only was it equipped with the same OPL2 chip, providing 100% compatibility with AdLib music playback, but it was also technologically superior with a DSP[19] allowing PCM playback (digitized sounds) at 8 bits per sample and up to 22.05kHz sampling rate. The card also came with a DA-15 port allowing joystick connection. Most importantly, the SoundBlaster was $90 cheaper than the AdLib.

Figure 2.22 is the Sound Blaster model CT1350B. Notice the OPL2 chip (labeled FM1312), the big CT1336 bus interface (labeled "CREATIVE") on the center left, the CT1351 DSP on the upper left, and the 8-bit ISA bus connector.

Trivia : The numerous advantages of the Sound Blaster card over the AdLib made it the de-facto standard shortly after its release and eventually brought AdLib to bankruptcy[20].

[19] An Intel MCS-51 "Digital Sound Processor", not "Digital Signal Processor".

[20] The reign of the Sound Blaster came to an end with Windows 95, which standardized the programming interface at application level and eliminated the importance of compatibility with Sound Blaster

2.4.3 Disney Sound Source

In 1990, Disney began selling the Disney Sound Source (DSS). Plugged into the printer port (parallel port) of the PC, an 8-bit DAC similar to the "Covox Speech Thing" was connected to a speaker box.

Figure 2.23: The speaker box (DAC not shown).

It was incredibly easy to set up, simple to program (it could only play one type of PCM and had no FM synthesizer), and very cheap compared to the other audio solutions ($14). It would have made programmers and customers happy if not for one serious limitation. The parallel port bandwidth[21] allowed a sampling rate up to 18,750 Hz but the design of the DSS limited the PCM sampling rate to 7,000Hz. This was still enough to produce pleasant sounds, but fell short when compared to the 22kHz of a Sound Blaster.

2.5 Floppy Disk Drive

In the time before the internet, a floppy disk was the best medium to share and distribute software and data. The orginal XT systems were equiped with 5¼-inch floppy disk with

[21] The parallel port maximum bandwidth was 150 kbits/s at the time. Enhanced Parallel Port and later Enhanced Capability Port significantly increased the transfer rate necessary to scanner and laser printers.

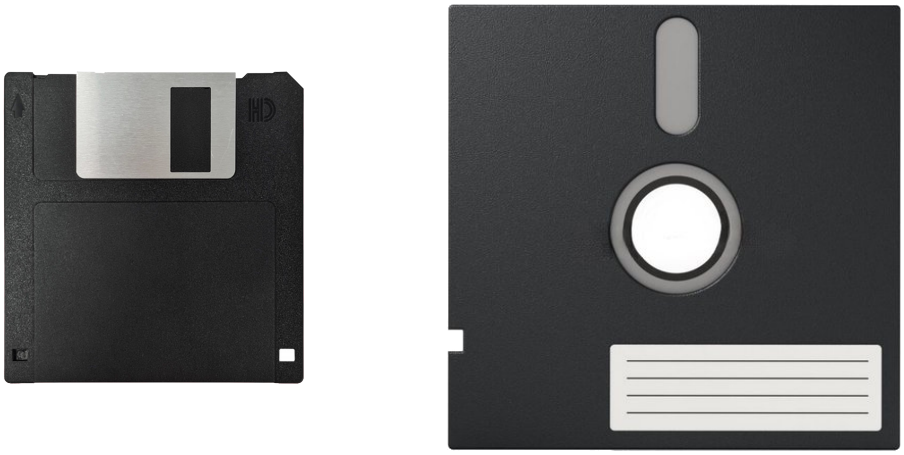


Figure 2.24: 3½-inch and 5¼-inch floppy disk.

a capacity of 360Kb. In 1984, IBM introduced with its PC AT the 1.2 MB dual-sided 5¼-inch floppy disk, but it never became very popular. IBM started using the 720 KB double density 3½-inch floppy disk in 1986 and the 1.44 MB high-density version in 1987. The advantages of the 3½-inch disk were its higher capacity, its smaller physical size, and its rigid case which provided better protection from dirt and other environmental risks. By the mid-1990s, 5¼-inch drives had virtually disappeared, as the 3½-inch disk became the predominant floppy disk.

Trivia : An USB stick of 128GB contains 91.000 high-density 3½-inch (1.44MB) floppy disks.

A floppy disk is essentially a very flexible piece (hence the term floppy disk) of plastic coated on both sides in a magnetic material. This 'disk' of plastic is contained within a protective envelope or hard plastic case, which is then inserted into the drive and automatically locked onto a spindle. It is then rotated at a constant speed, 360 rpm for standard PC floppy drives. A head assembly consisting of two magnetic read/write heads, one in contact with the upper surface and one in contact with the lower surface of the disk, may be moved in discrete steps across the disk and read the data from the disk.

The floppy disk is controlled via the Floppy Disk Controller (FDC), a typical read operation from the floppy disk contains the following steps:

- Turn the disk motor on. When you turn a floppy drive motor on, it takes quite a few milliseconds to "spin up", to reach the (stabilized) speed needed for data transfer.
- Perform seek operation, which moves the head to the correct location for reading the

data.

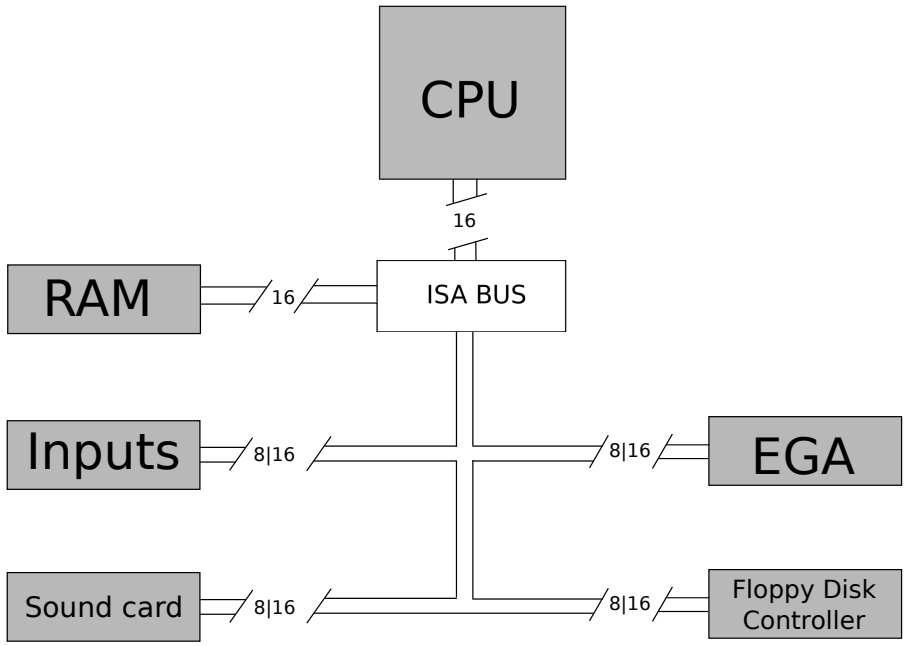
- Read the data from the floppy disk and store the data via the FDC to RAM memory.
- Turn the disk motor off.

You should wait a few seconds before turning the motor off. The reason to leave the motor on for a few seconds is that your driver may not know if there is a queue of sector reads or writes that are going to be executed next. If there are going to be more drive accesses immediately, they won't need to wait for the motor to spin up again.

2.6 Bus

Although developers had no control over them, it is still worth mentioning how these components were connected to each other.

The ISA[22] bus connects the CPU to all devices, including RAM. It was almost 10 years old in 1990 but still used universally in PCs. The data path to the RAM is 16 bits wide for 286 machines. It runs at the same frequency as the CPU.



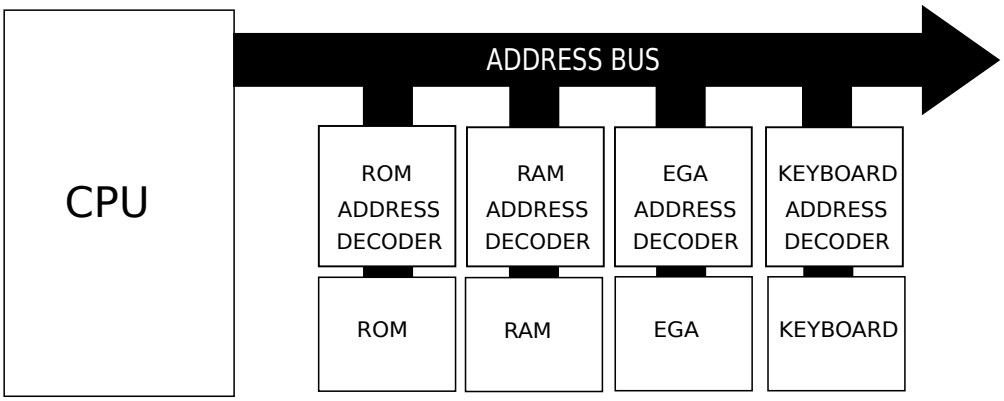
[22] Industry Standard Architecture.

The rest of the bus connecting to everything that is not the RAM can be either:

- 8 bits wide at 4.77 MHz for 19.1 Mbit/s
- 16 bits wide at 8.33MHz for 66.7 Mbit/s[23].

It is also backward compatible and an 8-bit ISA card can be plugged into a 16-bit ISA bus.

Trivia : On ISA all devices are connected to the bus at all times and listen on the bus address lane. Each device features an "address decoder" to detect if it should reply to a bus request. This is how the EGA RAM is "mapped" in RAM. The EGA card "address decoder" filters out everything that is not within `A0000h` and `AFFFFh`. Accordingly, the RAM disregards any request that is within the range [`A0000h - AFFFFh`].



2.7 Inputs

At a time before the ubiquitous USB, inputs were a mess with no less than four ports, all programmed differently.

The parallel port (DB-25) was on every computer and usually used to connect dot-matrix printers (loud things that printed with needles). The parallel port was multi-purpose and the Disney Sound Source could be plugged into it.

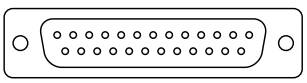


Figure 2.25: Parallel Port

[23]https://en.wikipedia.org/wiki/List_of_device_bit_rates .

The serial port (DE9) was used to connect the mouse.

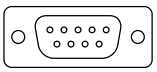


Figure 2.26: Serial Port

The PS/2 port was used to connect a keyboard.

Figure 2.27: PS/2 Port

Finally, a SoundBlaster sound card connected via the ISA bus provided a Game Port (DA-15) allowing for connection to a joystick[24].

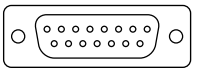


Figure 2.28: Game Port

2.8 Summary

To say a PC was difficult to program for games would be an understatement. It was a nightmare. The CPU was good at doing the wrong thing, the best graphic interface didn't allow double buffering, the memory model only allowed 1 standard MiB with an address composed of two separate 16-bit registers, and the `near`/`far` pointers forbade using standard C. Last, but not least, the default sound system could only produce square waves.

Yet despite all these unfavorable conditions, teams of developers gathered to tame the beast and unleash its power to gamers. One of these called themselves *Ideas From the Deep*[25].

[24] In 1981, the very first IBM PC could be purchased with a DA-15 "Game Port" extension card at the cost of $55 ($159 in 2018).

[25] They originally called themselves Ideas From the Deep but then decided to shorten it to simply id, which stands for "in demand", and is pronounced as in "did" or "kid." The name also refers to id, the part of the brain that behaves by the pleasure principle in Freudian psychology.

Chapter 3

Assets

3.1 Programming

Development was done with Borland C++ 3.1 (but the language used was C) which by default ran in EGA mode 3 offering a screen 80 characters wide and 25 characters tall.

John Carmack took care of the runtime code. John Romero programmed many of the tools (TED5 map editor, IGRAB asset packer, MUSE sound packer). Jason Blochowiak wrote important subsystems of the game (Input manager, Sound manager, User manager).

Borland's solution was an all-in-one package. The IDE, `BC.EXE`, despite some instabilities allowed crude multi-windows code editing with pleasant syntax highlights. The compiler and linker were also part of the package under `BCC.EXE` and `TLINK.EXE`[1].

[1] Source: Borland C++ 3.1 User Guide.

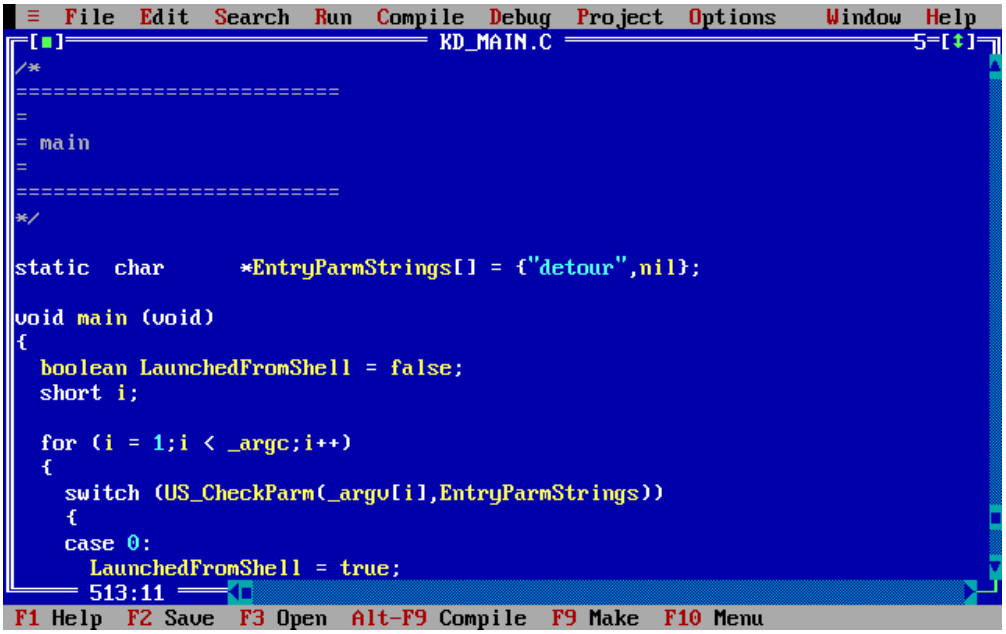


Figure 3.1: Borland C++ 3.1 editor

There was no need to enter command-line mode however. The IDE allowed to create a project, build, run and debug.

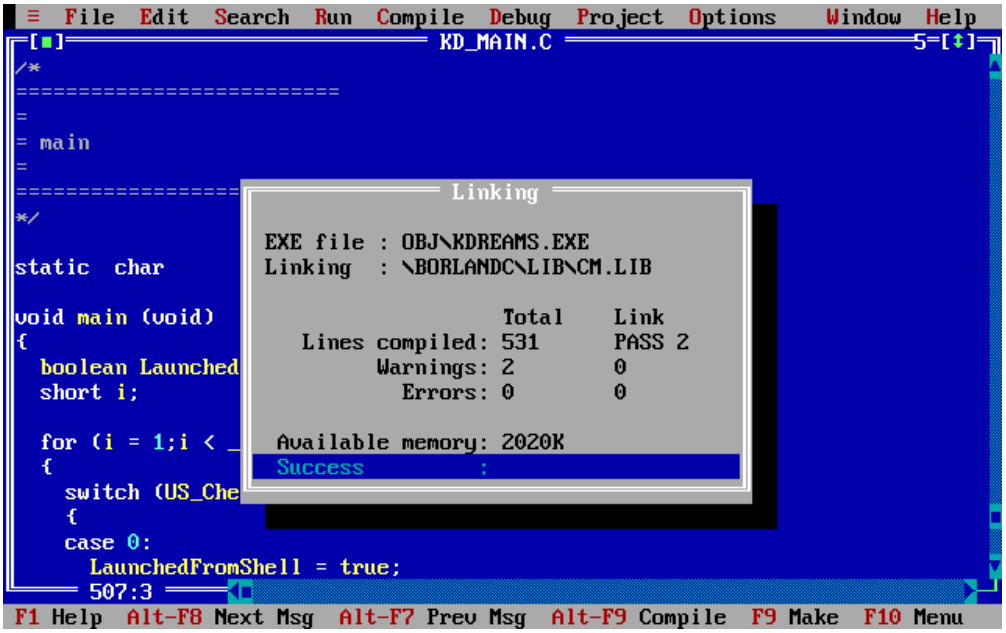
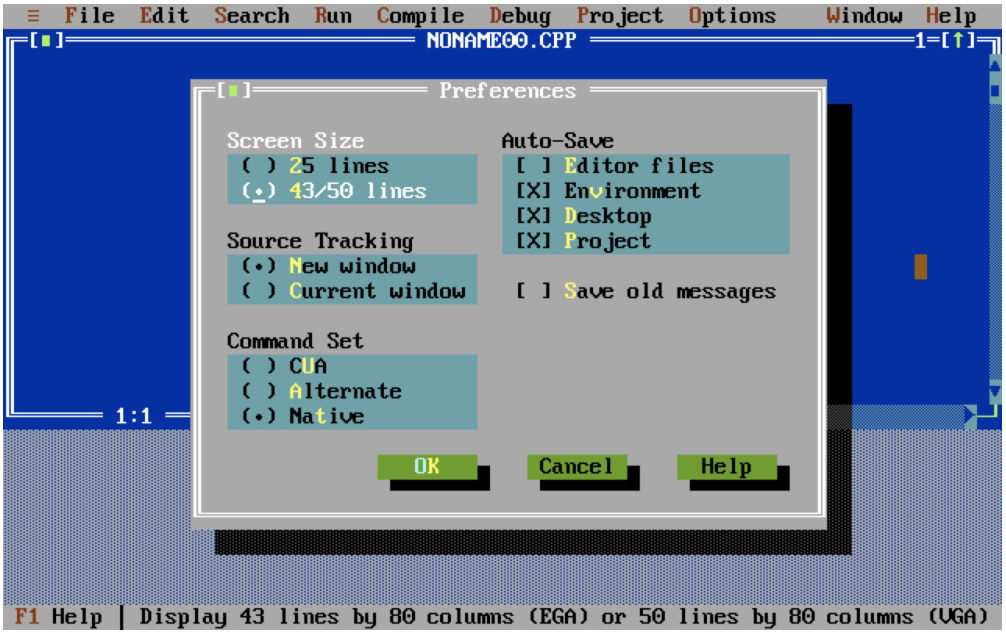
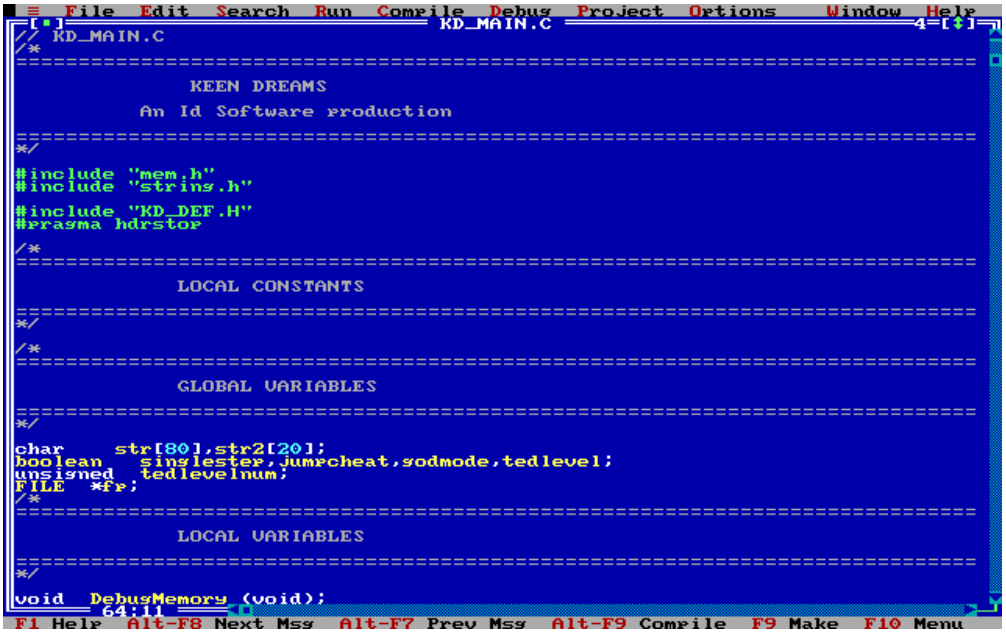


Figure 3.2: Compiling Keen Dreams with Borland C++ 3.1

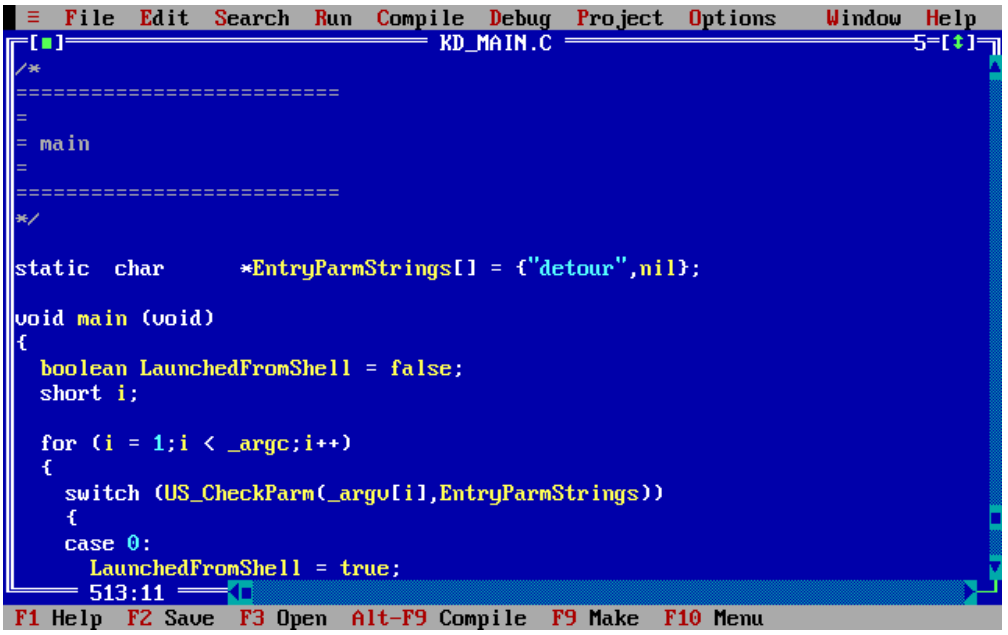
Another way to improve screen real estate was to use "high resolution" 50x80 text mode.

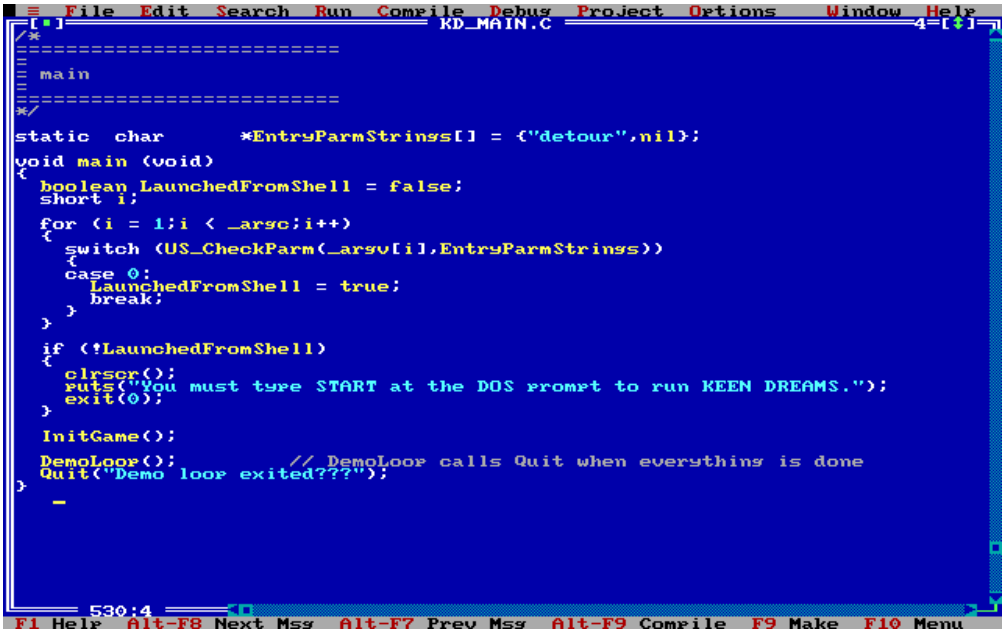


The comments still fit perfectly on screen since only the vertical resolution is doubled.



The file `KD_MAIN.C` opened in both modes demonstrates the readability/visibility trade-off.





3.2 Graphic Assets

All graphic assets were produced by Adrian Carmack. All of the work was done with Deluxe Paint (by Brent Iverson, Electronic Arts) and saved in ILBM[2] files (Deluxe Paint proprietary format). All assets were hand drawn with a mouse.

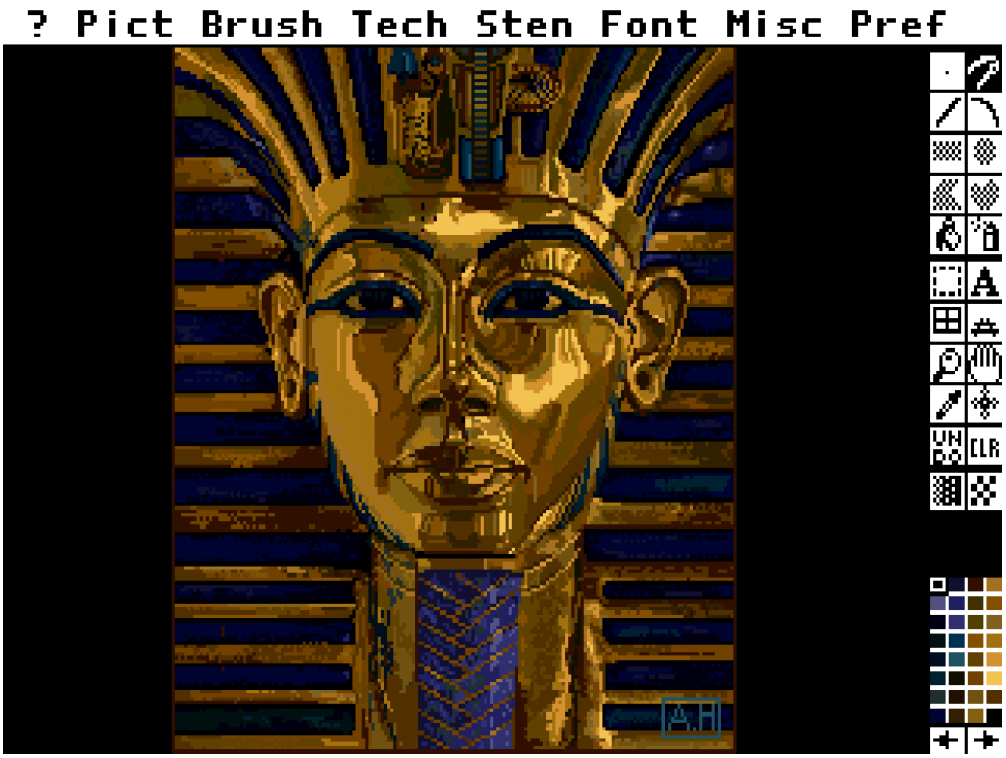


Figure 3.3: Deluxe Paint was used to draw all assets in the game.

3.2.1 Assets Workflow

After the graphic assets were generated, a tool (IGRAB) packed all ILBMs together in an archive and generated a header table file (`KDR`-format) and C header file with asset IDs. The engine references an asset directly by using these IDs.

[2] InterLeaved BitMap.

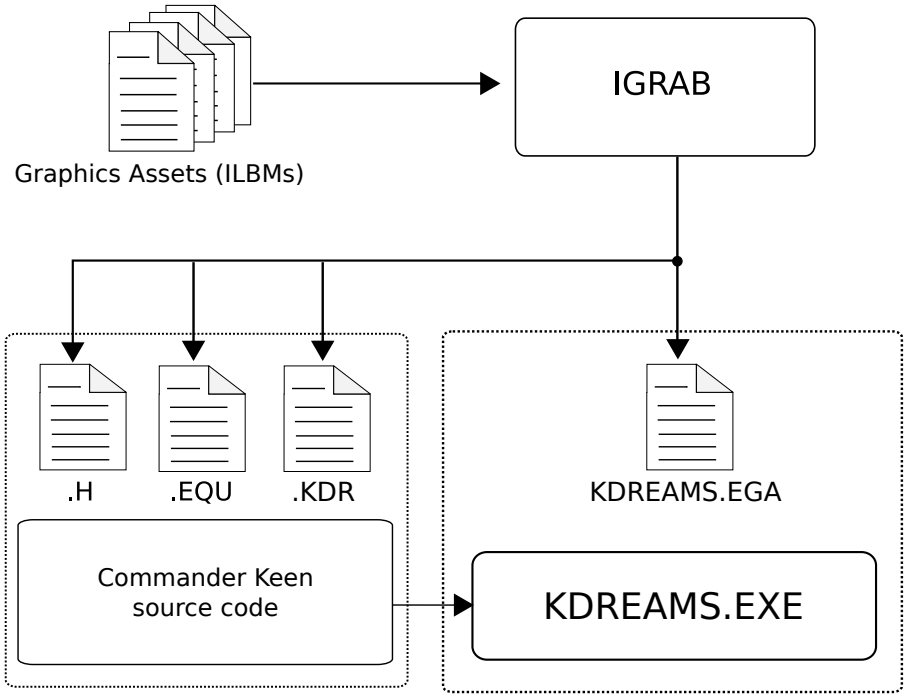


Figure 3.4: Asset creation pipeline for graphics items

```
//////////////////////////////////////
//
// Graphics .H file for .KDR
// IGRAB-ed on Fri Sep 10 11:18:07 1993
//
//////////////////////////////////////

typedef enum {
    #define CTL_STARTUPPIC        4
    #define CTL_HELPUPPIC         5
    #define CTL_DISKUPPIC         6
    #define CTL_CONTROLSUPPIC     7
    #define CTL_SOUNDUPPIC        8
    #define CTL_MUSICUPPIC        9
    #define CTL_STARTDNPIC       10
    #define CTL_HELPDNPIC        11
    #define CTL_DISKDNPIC        12
    #define CTL_CONTROLSDNPIC    13
    ...
    #define BOOBUSWALKR4SPR     366
    #define BOOBUSJUMPSPR       367
```

In the engine code, asset usage is hardcoded via an enum. This enum is an offset into the `HEAD` table which contains an offset in the `DATA` archive. The `HEAD` table files are stored in the `\static` folder as `*.KDR` files.

3.2.2 Assets file structure

Figure 3.5 shows the structure of the `KDREAMS.EGA` asset file.

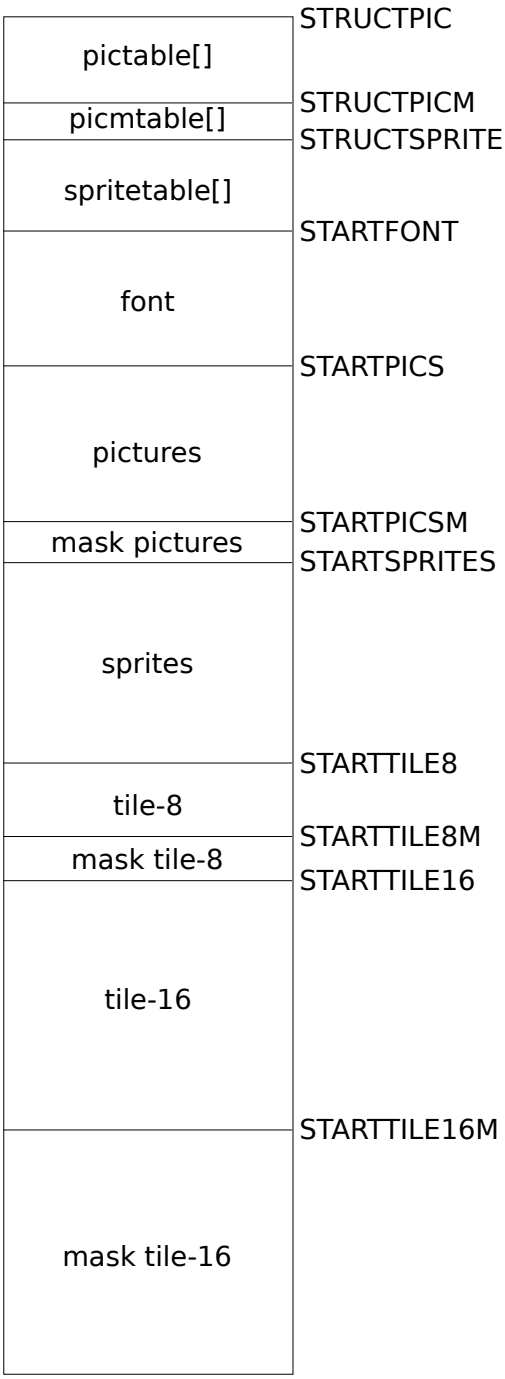


Figure 3.5: File structure of `KDREAMS.EGA` asset file.

The `pictable[]` contains the width and height in bytes for each picture in the asset file. Note that a width of 5 bytes means a width of 40 pixels on the screen. The same size structure is applied for mask pictures.

index	width	height
0	5	32
1	5	32
2	5	32
3	5	32
4	5	32
5	5	32
6	5	32
7	5	32
...	...	...
64	5	24

Table 3.1: content of `pictable[]`.

The `spritetable[]` contains, beside width and height, also information on the sprite center, hit boundaries and number of shifted sprites, which will be explained later (section 4.12.2 on page 137).

The font segment contains a table for the height (same for all characters) and width of the font, as well as a reference where the character data is located.

```
// ID_VW.H

typedef struct
{
  int height;
  int location[256];
  char width[256];
} fontstruct;
```

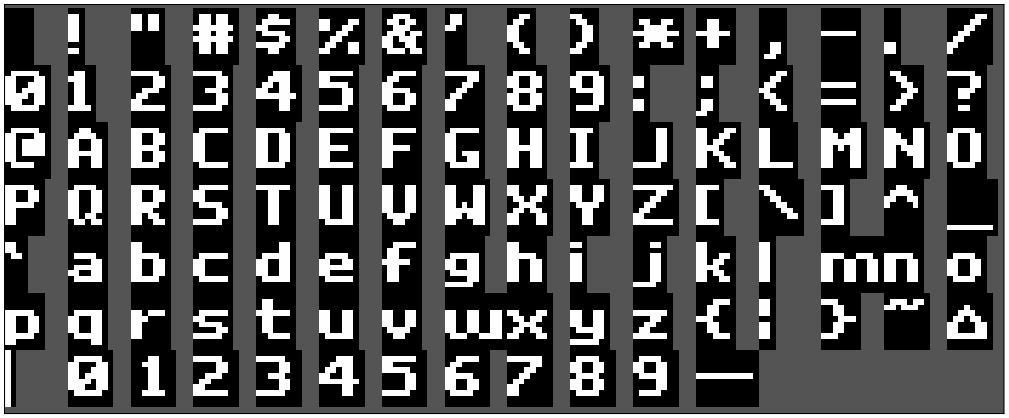


Figure 3.6: Font asset data.

Since all tiles have fixed dimension (either 8 or 16 pixels), there is no need to store any tile size table structure.

From `STARTPICS` location onwards all graphical assets are stored. Each asset contains four planes, aligned with the EGA architecture. Foreground tiles and sprites include a mask plane as well.

Figure 3.7: Picture asset data.

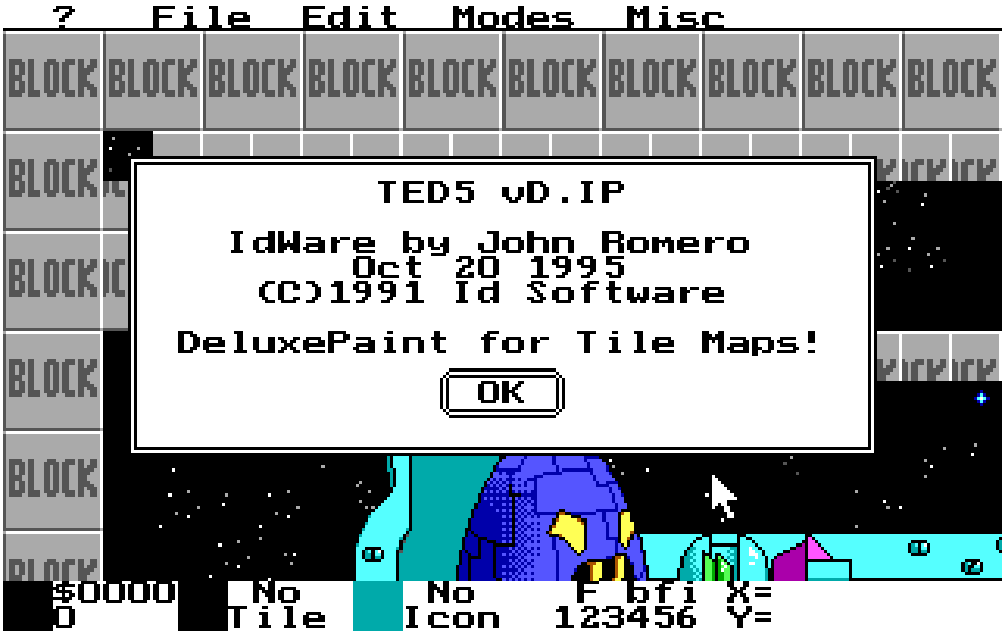
Figure 3.8: Sprite asset data.

Figure 3.9: Background (Tile16) and foreground (masked Tile16) assets data.

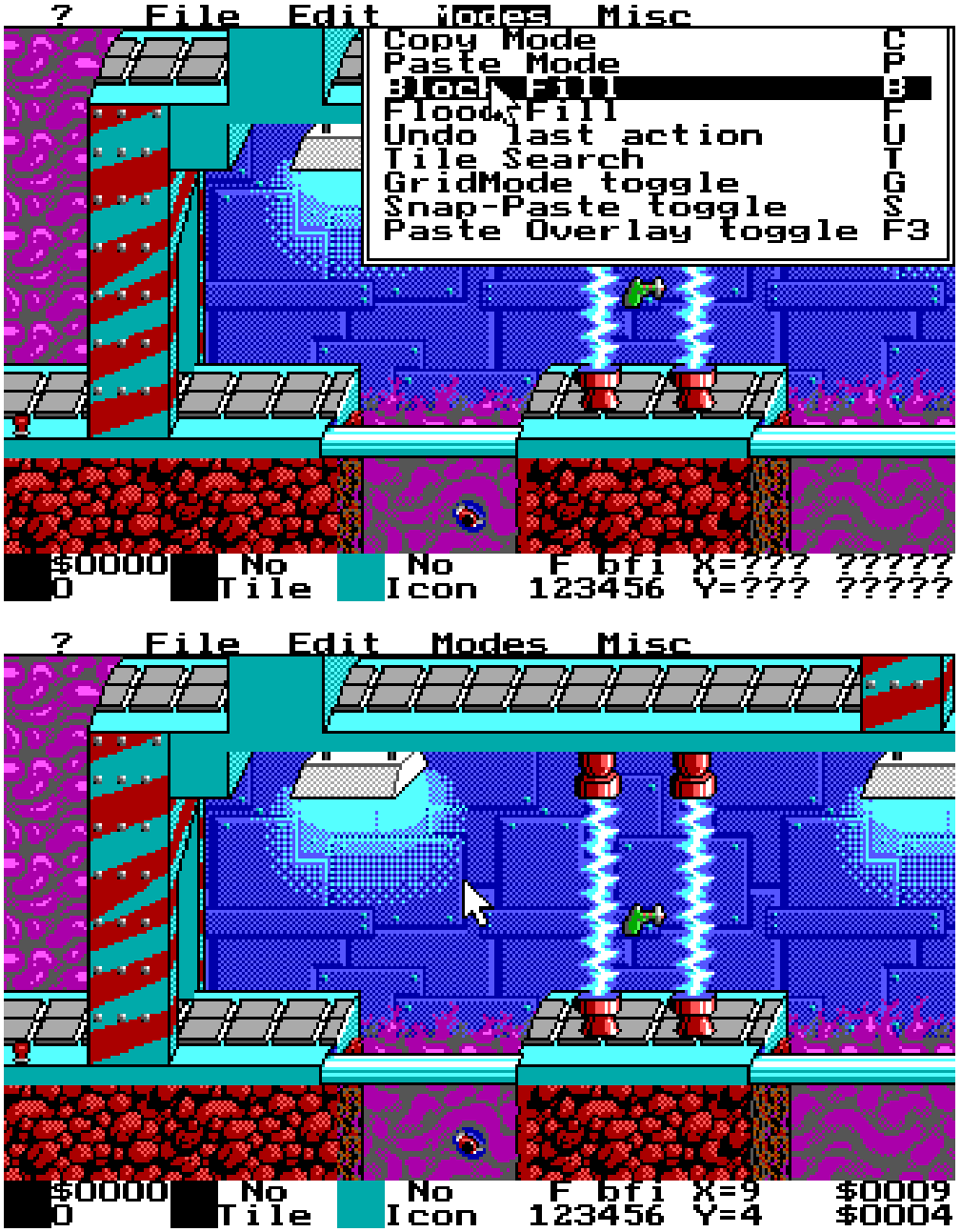
3.3 Maps

Maps were created using an in-house editor called TED5, short for Tile EDitor. Over the years TED5 had improvements and the same tool is later used for creating maps of both side-scrolling games and top-down games like *Wolfenstein 3D*.

TED5 is not stand-alone; in order to start, it needs an asset archive and the associated header (as described in the graphic asset workflow Figure 3.4 on page 57). This way, texture IDs are directly encoded in the map.



Trivia : The suffix, "vD.IP", was put in by the Rise of the Triad team in 1994. It stood for "Developers of Incredible Power".



TED5 allows placement of tiles on layers called "planes". In Commander Keen, layers are used for background, foreground and information planes. Note that foreground tiles are

always using a mask as they are overlayed with the background. The info plane contains the location of actors and special places. Each foreground and background tile could also be enriched with additional tile information such as tile clipping and animated tiles. Just like IGRAB, the TED5 tool generated a header table file (KDR-format) and a *.MAP file containing the levels. Figure 3.10 shows the map header table structure, which is hard-coded in the source code.

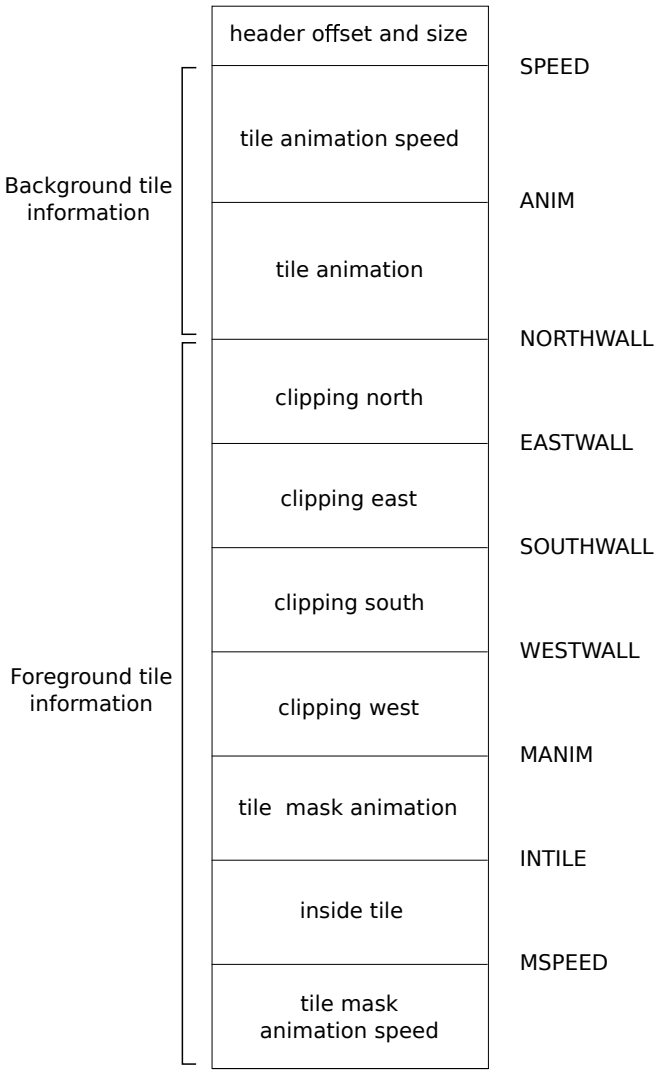


Figure 3.10: File structure of MAPHEAD.KDR header file.

3.3.1 Map header structure

The header offset and header size refer to the location and size in the `KDREAMS.MAP` file. A maximum of 100 maps is supported in the game.

```
/*
=================================================

              LOCAL CONSTANTS

=================================================
*/


typedef struct
{
  unsigned  RLEWtag;
  long    headeroffsets[100];
  byte    headersize[100];    // headers are very small
  byte    tileinfo[];
} mapfiletype;
```

3.3.2 Background tile information

For background tile animation two information tables are required: tile animation and tile animation speed. The tile animation refers to the next tile in the animation sequence. So in case of tile #90 (see Table 3.2), the next animation tile is #91 (+1), followed by #92 (+1) and #93 (+1). After tile #93 (-3) the sequence is going back to tile #90. The animation speed is expressed in TimeCount, which is the number of ticks before the next tile is displayed.

index	tile animation	tile animation speed
0	0	0
1	0	0
...	...	...
57	1	32
58	-1	24
...	...	...
90	1	8
91	1	8
92	1	8
93	-3	8
...	...	...

Table 3.2: Background tile animation.

3.3.3 Foreground tile information

The foreground tiles contain, beside tile animation (similar like background tiles), also clipping and 'inside' tile information.

The clipping tables contain how Commander Keen is clipped against foreground tiles. 'Inside' tile information is used to climb on a pole and mimics Commander Keen going through a floor opening ('inside' the tile). In section 4.12.3 (see page 139) both the clipping and 'inside' logic is further explained.

index	clip north	clip east	clip south	clip west	inside tile
...	...	...	...	...	...
238	0	1	5	0	0
239	0	0	0	0	0
240	0	0	5	0	0
241	0	0	0	0	128
242	1	1	1	1	128
243	1	0	1	0	0
244	0	0	2	0	0
...	...	...	...	...	...

Table 3.3: Foreground tile clipping and 'inside' tile information.

3.3.4 Map file structure

The structure of KDREAMS.MAP is explained in Figure 3.11. For each map there is a small header containing the width, height and name of the map as well as a reference pointer to each of the three planes. Each plane exists out of a map of tile numbers for foreground, background and info.

```
typedef struct
{
  long          planestart[3];
  unsigned      planelength[3];
  unsigned      width,height;
  char          name[16];
} maptype;
```

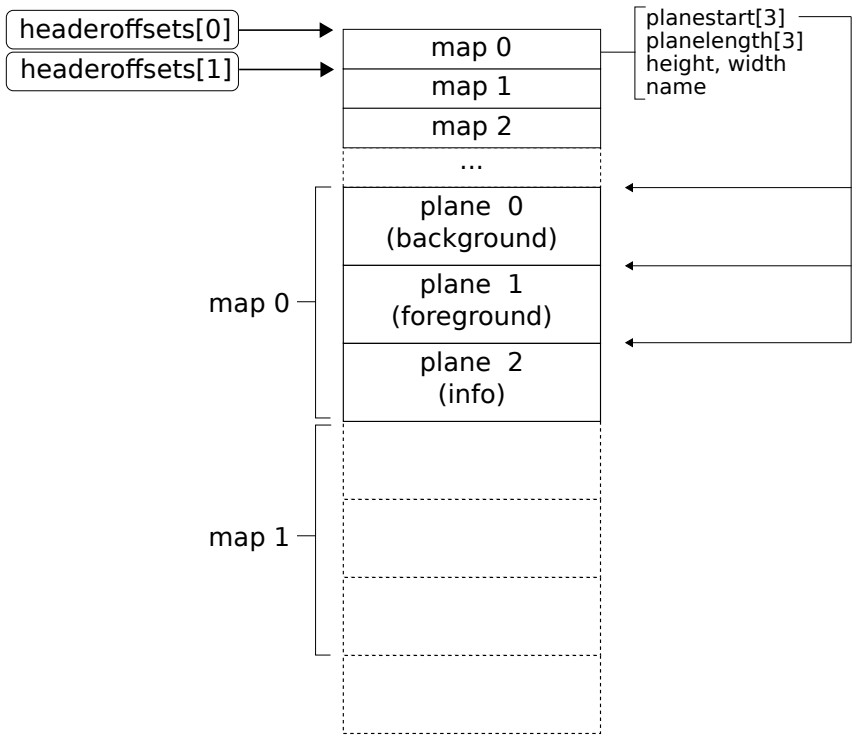


Figure 3.11: File structure of KDREAMS.MAP file.

3.4 Audio

3.4.1 Sounds

The original Commander Keen Triology did only support the default PC Speaker. Only with the introduction of Commander Keen Dreams the team decided to support sound cards.

Trivia : Apogee, the publisher of Ideas from the Deep, didn't publish any game with Adlib support until *Dark Ages* in 1991. And even then it still had pc speaker music.

As mentioned in Chapter 2.4, audio hardware was highly fragmented. Commander Keen supported three sound cards and the default PC speaker, which meant generating assets multiple times for each and packing them together with an in-house tool called MUSE into an AUDIOT archive (an id software proprietary format):

Figure 3.12: MUSE splash screen.

id Software intended for Keen Dreams to have music and digital effects support for the Sound Blaster & Sound Source devices. In fact, Bobby Prince composed the song "You've Got to Eat Your Vegetables!!" for the game's introduction. However, Softdisk Publishing wanted Keen Dreams to fit on a single 360K floppy disk, and in order to do this, id Software had to scrap the game's music at the last minute. The team didn't even have time to remove the music setup menu.

Figure 3.13: Setup music menu in Keen Dreams, although there is no music in the game.

Trivia : The song "You've Got to Eat Your Vegetables!!" would finally make its debut in Commander Keen IV: Secret of the Oracle.

Two sets of each audio effect shipped with the game:

1. For PC Speaker
2. For AdLib

3.4.2 Sound effects

All sound effects are done by Robert Prince. In the early days of the OPL soundcards, the "gold standard" sequencing software was Sequencer Plus Gold ("SPG") by Voyetra.

The reason for this was it had an OPL instrument/instrument bank editor. To rough out compositions, Robert used Cakewalk ("CW").

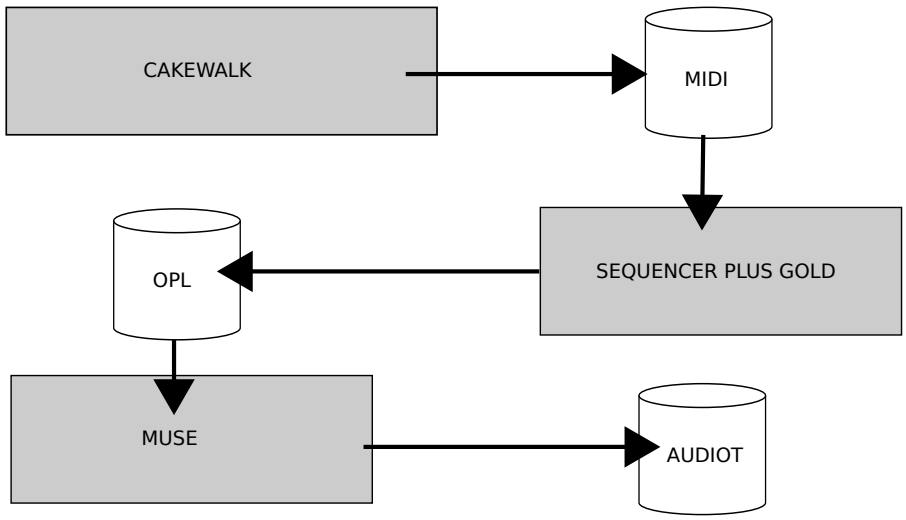


Figure 3.14: Music and sound effect pipeline as used by Bobby Prince.

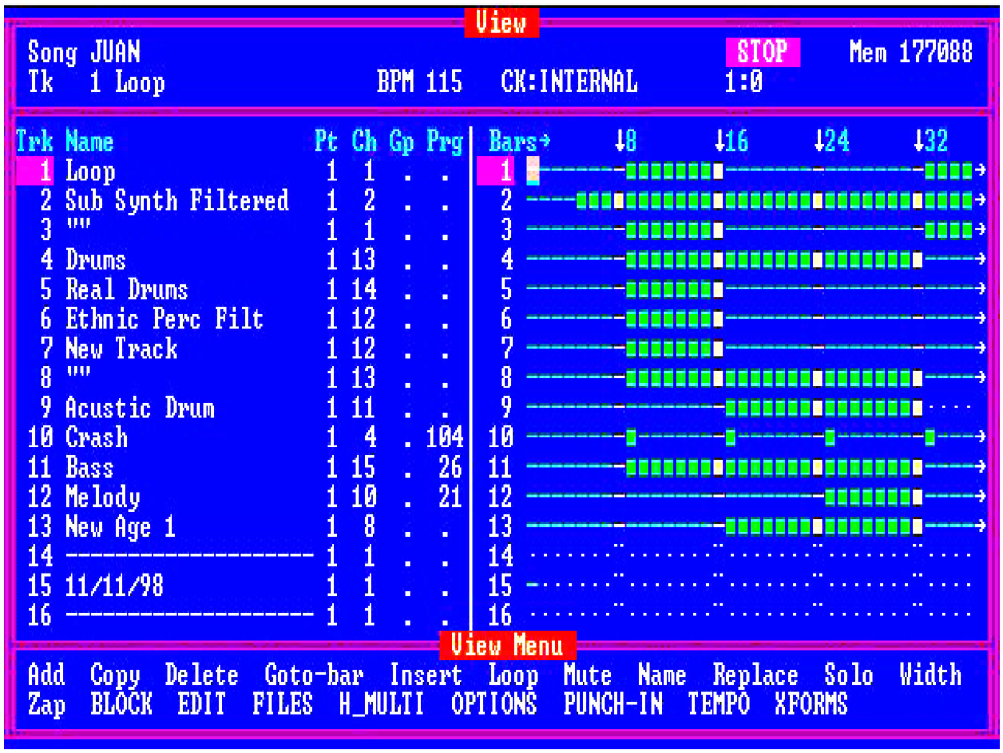


Figure 3.15: Sequencer Plus Gold ("SPG") by Voyetra.

What goes in the `AUDIOT` archive is a music and sound effect format called IMF[3]. As it supports only the YM3812, it is tailored for the chip with zero abstraction layers. It consists of a stream of machine language commands with associated delays[4].

The stream pilots the nine channels in the OPL2. A channel is able to simulate an instrument and play notes thanks to two oscillators, one playing the role of a modulator and the other the role of a carrier. There are many other ways to control a channel such as envelope, frequency or octave.

The way a channel is programmed is described in detail in Section 4.13.5.1, "FM Synthesizer: OPL2/YM3812 Programming" on page 155.

[3] Id software Music File.
[4] IMF format is explained in detail on page 155

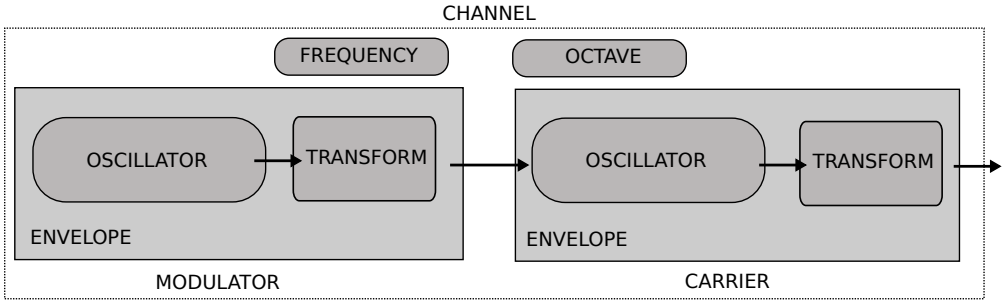


Figure 3.16: Architecture of a YM3812 channel.

Trivia : The YM3812's unmistakable sonority is due to its peculiar set of waveform transformers (they are right after the output of each oscillator in the drawing). Four waveforms are available on the OPL2: Sin ①, Abs-sin②, Pulse-sin ③, and Half-sin ④.

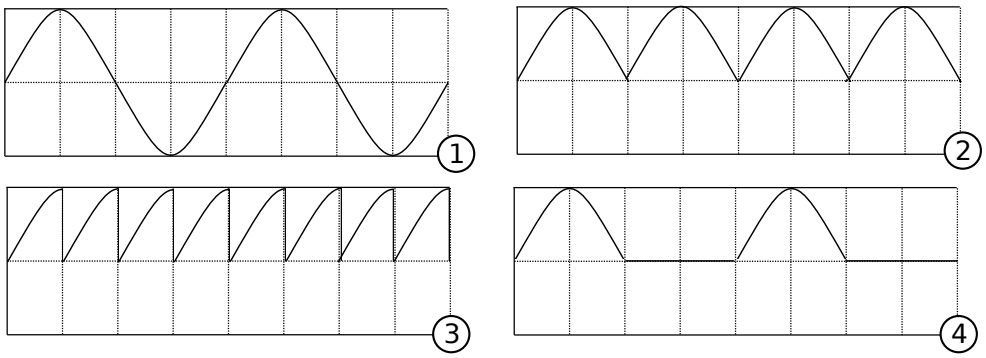


Figure 3.17: The four waveform transforms available.

3.5 Distribution

On December 14th, 1990 the first episode was released via Apogee. Episodes 1-5 are all published by Apogee Software. The game engine and first episode were given for free and encourages to be copied and distributed to a maximum number of people. To reveive the oher episodes, each player had to pay $30 (for Episide 1-3) to *ideas from the Deep*.

Commander Keen in Keen Dreams was published as a retail title by Softdisk, as part of a settlement for using Softdisk resources to make their own game[5].

[5]The settlement with Softdisk is explained in Appendix B

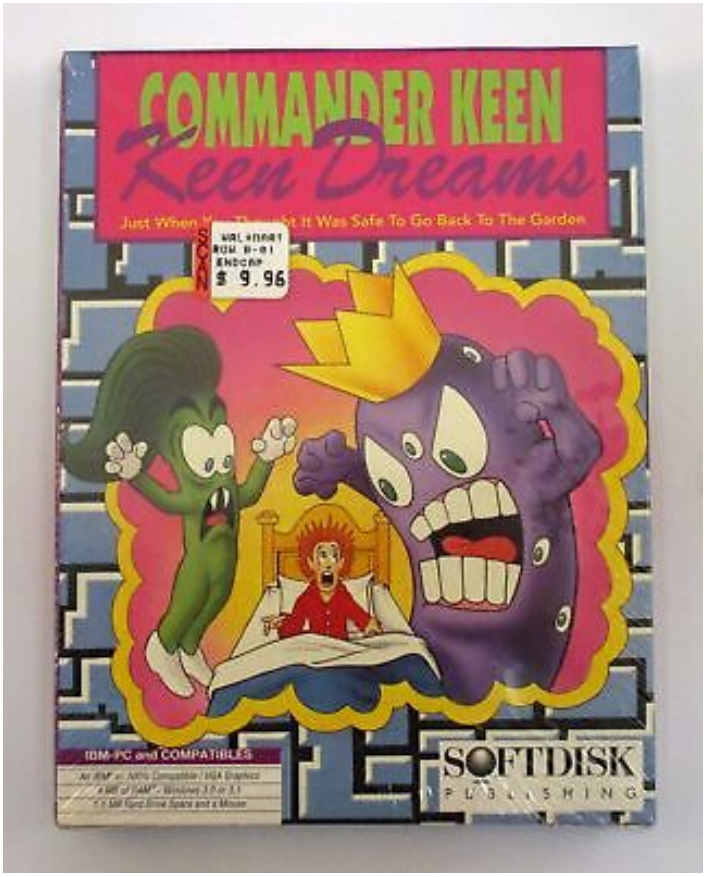


Figure 3.18: Retail version of Commander Keen in Keen Dreams by Softdisk.

In 1990, the Internet was still in its infancy and the best medium was the 3½-inch floppy disk. The game shipped as follows:

The files can be divided in seven parts:

- KDREAMS.EXE: Game engine.
- KDREAMS.EGA: Contains all the assets (sprites, tiles) needed during the game.
- KDREAMS.AUD: Sound effect files.
- KDREAMS.MAP: Contains all levels layouts.
- KDREAMS.CMP: Introduction picture of the game, which is a compressed LBM image file.

- Softdisk Help Library files, which are text screens, shown when starting the game
  - START.EXE: Decompress and show user guide, and then start the game.
  - LOADSCN.EXE: Shows the closing screen when quiting the game. Decompresses the ENDSCN.SCN chunk in LAST.SHL
  - *.SHL: Text user guide assets.
- Several *.TXT files which can be read in DOS by typing the corresponsing *.BAT file.

```
Directory of C:\KDREAMS\.
.            <DIR>           16-05-2023 21:04
..           <DIR>           16-05-2023 20:52
BKGND    SHL               285 16-05-2023 20:59
FILE_ID  DIZ               508 16-05-2023 20:59
HELP     BAT                34 16-05-2023 20:59
HELPINFO TXT             1,038 16-05-2023 20:59
INSTRUCT SHL             2,763 16-05-2023 20:59
KDREAMS  AUD             3,498 16-05-2023 20:59
KDREAMS  CFG               656 16-05-2023 21:00
KDREAMS  CMP            14,189 16-05-2023 20:59
KDREAMS  EGA           213,045 16-05-2023 20:59
KDREAMS  EXE           354,691 04-05-2023 19:40
KDREAMS  MAP            65,673 16-05-2023 20:59
LAST     SHL             1,634 16-05-2023 20:59
LICENSE  DOC             8,347 16-05-2023 20:59
LOADSCN  EXE             9,959 16-05-2023 20:59
MENU     SHL               447 16-05-2023 20:59
NAME     SHL                21 16-05-2023 20:59
ORDER    SHL             1,407 16-05-2023 20:59
PRODUCTS SHL             4,629 16-05-2023 20:59
QUICK    SHL             3,211 16-05-2023 20:59
README   TXT             1,714 16-05-2023 20:59
START    EXE            17,446 16-05-2023 20:59
VENDOR   BAT                32 16-05-2023 20:59
VENDOR   DOC            11,593 16-05-2023 20:59
VENDOR   TXT               810 16-05-2023 20:59
   24 File(s)          717,630 Bytes.
    2 Dir(s)       262,111,744 Bytes free.

C:\KDREAMS>
```

Figure 3.19: All Keen Dreams files as they appear in DOS command prompt.

Chapter 4

Software

4.1 About the Source Code

Commander Keen episodes 1-5 source code is not available as the current owner Zenimax[1] has, as of writing this book, no interest in selling intellectual properties. Luckily the ownership of Commander Keen: Keen Dreams was in the hands of Softdisk. In June 2013, developer Super Fighter Team licensed the game from Flat Rock Software, the then-owners of Softdisk, and released a version for Android devices.

The following September, an Indiegogo crowdfunding campaign was started to attempt to buy the rights from Flat Rock for US$1500 in order to release the source code to the game and start publishing it on multiple platforms. The campaign did not reach the goal, but it's creator Javier Chavez made up the difference, and the source code was released under GNU GPL-2.0-or-later soon after.

4.2 Getting the Source Code

The source code is made available via `github.com`. It is important to take the the source code from shareware version 1.13, otherwise you run into issues due to incompatible map headers. To get the correct source code

```
$ git clone https://github.com/keendreams/keen.git
$ cd keen
$ git checkout a7591c4af15c479d8d1c0be5ce1d49940554157c
```

[1] June 24, 2009, it was announced that id Software had been acquired by ZeniMax Media (owner of Bethesda Softworks).

4.3 First Contact

Once downloaded via `github` a folder 'keen' is created with all source files inside. `cloc.pl` is a tool which looks at every file in a folder and gathers statistics about source code. It helps for getting an idea of what to expect.

```
$ cloc keen

52 text files.
52 unique files.
7 files ignored.

-------------------------------------------------------------
Language         files        blank      comment         code
-------------------------------------------------------------
C                   20         4008         5361        14893
Assembly             5          992         1114         2688
C/C++ Header        19          508          665         1603
Markdown             1           18            0           40
DOS Batch            1            0            0           13
-------------------------------------------------------------
SUM:                46         5526         7140        19237
-------------------------------------------------------------
```

The code is 85% in C with assembly[2] for bottleneck optimizations and low-level I/O such as video or audio.

Source lines of code (SLOC) is not a meaningful metric against a single codebase but excels when it comes to extracting proportions. Commander Keen with its 19,237 SLOC is very small compared to most software. `curl` (a command-line tool to download url content) is 154,134 SLOC. Google's Chrome browser is 1,700,000 SLOC. Linux kernel is 15,000,000 SLOC.

[2] All the assembly in Keen is done with TASM (a.k.a Turbo Assembler by Borland). It uses Intel notation where the destination is before the source: `instr dest source`.

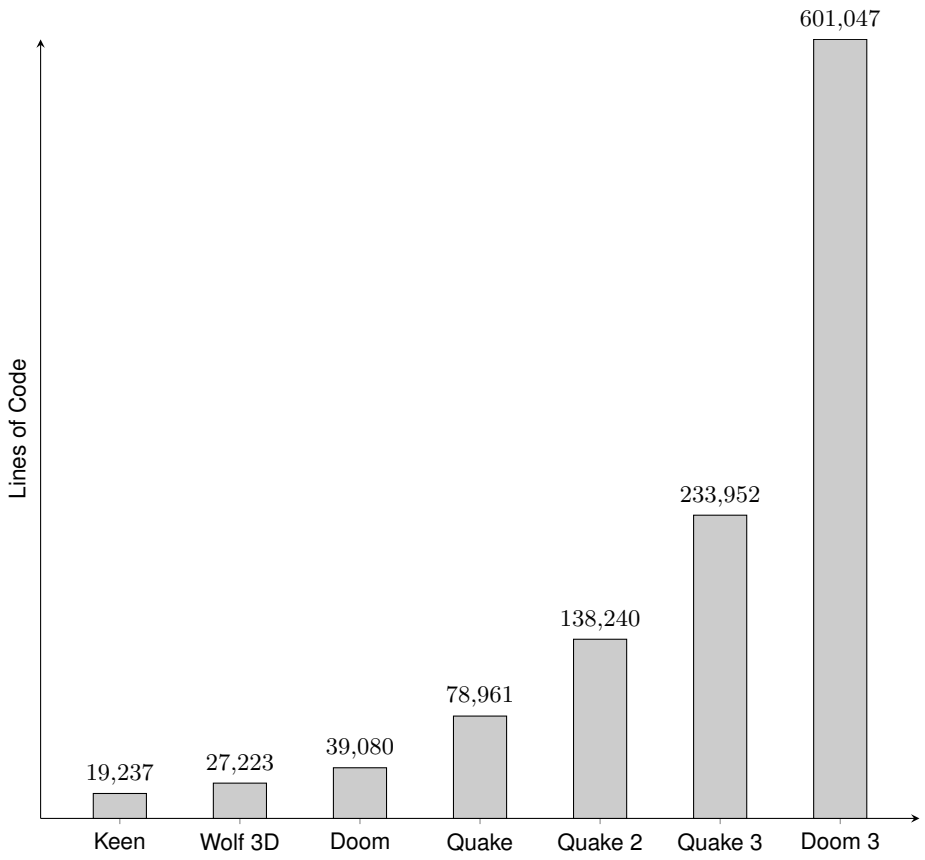


Figure 4.1: Lines of code from id Software game engines.

The archive contains more than just source code; it also features:

- `static` folder: Static header files for loading assets (as explained in Section 3.2).
- `lscr` folder: Load and decompress Softdisk data files.
- `README`: How to build the executable.

4.4 Compile source code

Now let's start to compile the source code. To compile the code like it's 1990 you need the following software:

- Commander Keen source code.

- DosBox.
- The Compiler Borland C++ 3.1.
- Commander Keen: Keen Dreams 1.13 shareware (for the assets).

After setting up the DosBox environment, with Borland C++ 3.1 installed (You can find a complete tutorial in "Let's compile like it's 1992" on `fabiensanglard.net`) download the source code via github.

Once you start DosBox and change directory to the `keen` folder, first create the folder where we create our compiled object files.

```
mkdir OBJ
```

Then we need to create the static `OBJ` header files.

```
chdir STATIC
make.bat
```

Once the static object header files are created, move back to the `keen` folder and open Borland C++. Open the `kdreams.prj` project file. Before we can start compiling we need to set the correct directories. Select Options -> Directories and change the values as follow:

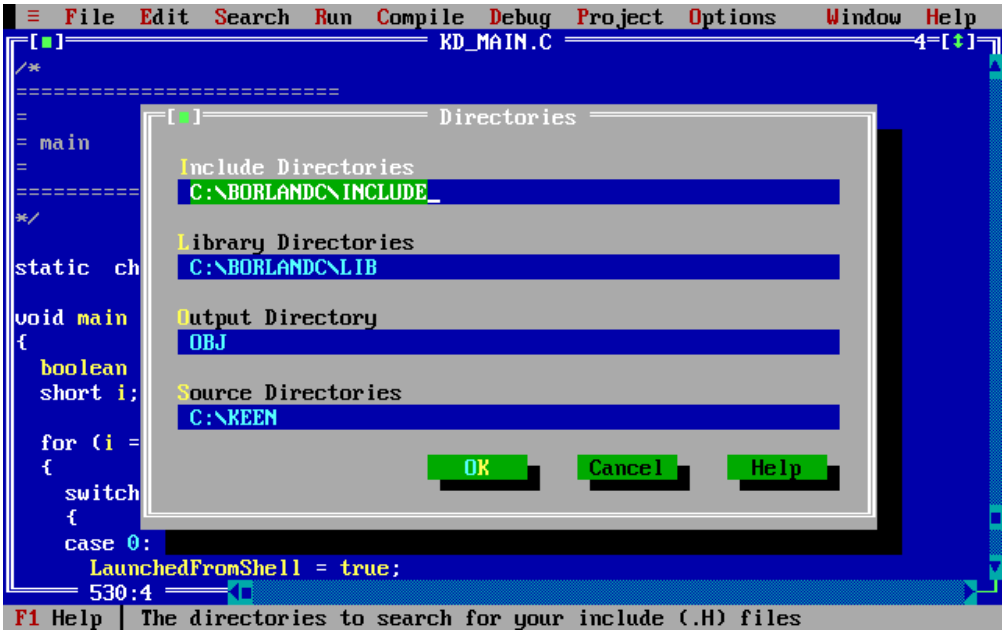


Figure 4.2: Borland C++ 3.1 directory settings

Now it's time to compile. Go to Compile -> Build all, and voila! The final step is to copy `kdreams.exe` to the Keen shareware folder. Now you can play your compiled version of Commander Keen.

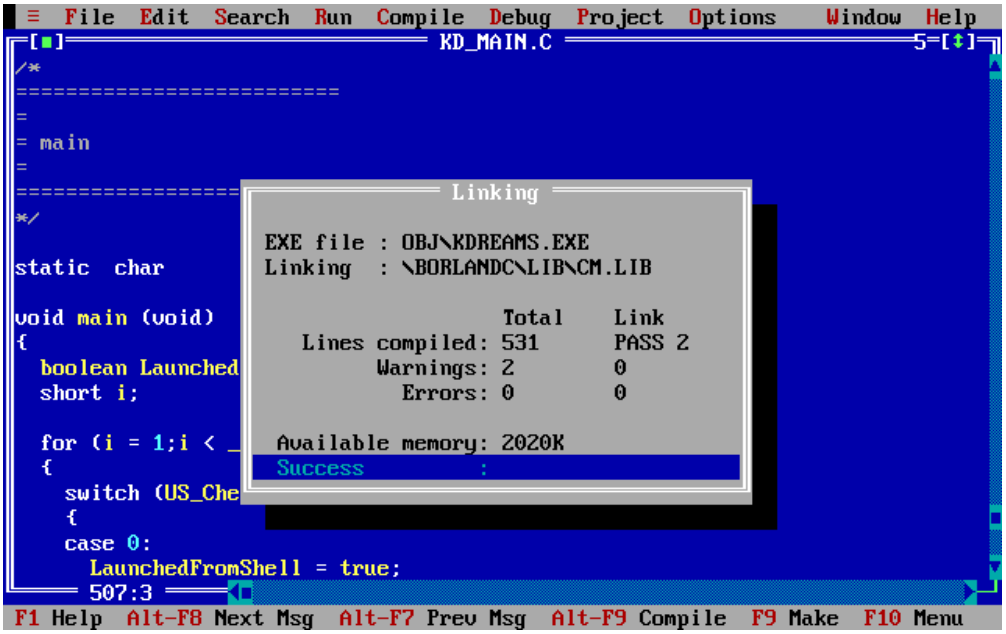


Figure 4.3: Commander Keen compiling

4.5 Big Picture

The game engine is divided in three blocks:

- Control panel which lets users configure and start the game.
- 2D game renderer where the users spend most of their time.
- Sound system which runs concurrently with either the Menu or 2D renderer.

The three systems communicate via shared memory. The renderer writes sound requests to the RAM (also making sure the assets are ready). These requests are read by the sound "loop". The sound system also writes to the RAM for the renderers since it is in charge of the heartbeat of the whole engine. The renderers update the screen according to the wall-time tracked by `TimeCount` variable.

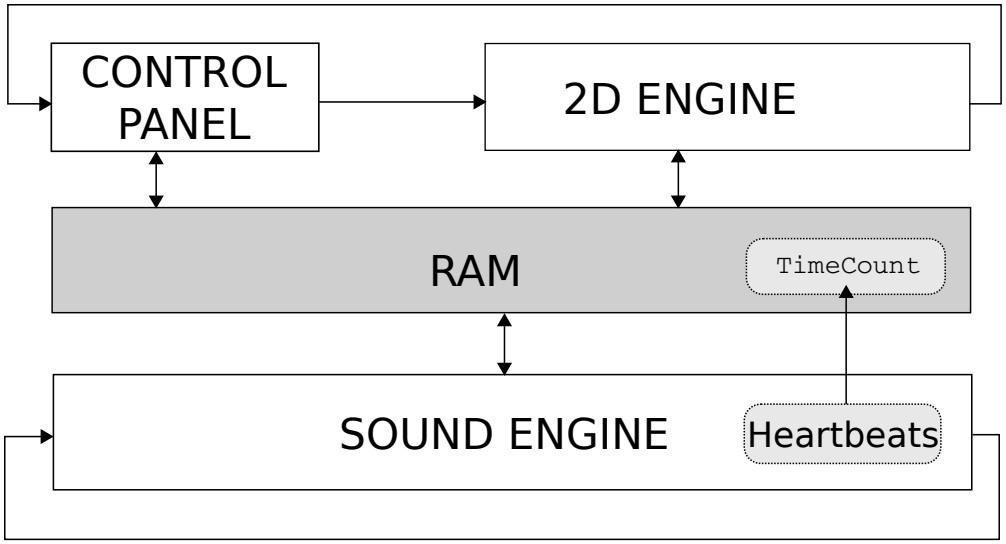


Figure 4.4: Game engine three main systems.

4.5.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop starting in `void main()`. The control panel and 2D renderer are regular loops but due to limitations explained later, the sound system is interrupt-driven and therefore out of `main`. Because of real mode, C types don't mean what people would expect from a 16-bit architecture.

- `int` and `word` are 16 bits.
- `long` and `dword` are 32 bits.

The first thing the program does is set the text color to light grey and background color to black.

```
void main (void)
{
  textcolor(7);
  textbackground(0);

  InitGame();

  DemoLoop();          // DemoLoop calls Quit when
   everything is done
  Quit("Demo loop exited???");
}
```

In `InitGame`, a valdation is performed to check if suficient memory is available and brings up all the managers.

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

  MM_Startup ();     // Memory Manager

  US_TextScreen();  // Show intro screen

  VW_Startup ();     // Video Manager
  RF_Startup ();     // Refresh Manager
  IN_Startup ();     // Input Manager
  SD_Startup ();     // Sound Manager
  US_Startup ();     // User Manager

  CA_Startup ();     // Cache Manager
  US_Setup ();

  CA_ClearMarks ();  // Clears out all the marks

  CA_LoadAllSounds (); // Load all sounds

}
```

Then comes the core loop, where the menu and 2D renderer are called forever.

```
void DemoLoop() {
    US_SetLoadSaveHooks();
    while (1) {
        VW_InitDoubleBuffer ();
        IN_ClearKeysDown ();
        VW_FixRefreshBuffer ();
        US_ControlPanel ();  // Menu
        GameLoop ();
        SetupGameLevel ();
        PlayLoop () ; // 2D renderer (action)
    }
    Quit("Demo loop exited???");
}
```

`PlayLoop` contains the 2D renderer. It is pretty standard with getting inputs, update screen, and render screen approach.

```
void PlayLoop (void)
{
  FixScoreBox ();     // draw bomb/flower
  do
  {
    CalcSingleGravity ();    // Calculate gravity
    IN_ReadControl(0,&c);    // get player input

    // go through state changes and propose movements
    obj = player;
    do
    {
      if (obj->active)
        StateMachine(obj);  // Enemies think
      obj = (objtype *)obj->next;
    } while (obj);

    [...]             // Check for and handle collisions
                      // between objects

    ScrollScreen(); // Scroll if Keen is nearing an edge.
                      // Draw new tiles to master screen in
                      // VRAM, and mark them in tile arrays

    [...]             // React to whatever happened, and post
                      // sprites to the refresh manager

    RF_Refresh();     // Copy marked tiles from master to
                      // buffer screen, and update sprites
                      // in buffer screen.
                      // Finally, switch buffer and view
                      // screen

    CheckKeys();      // Check special keys
  } while (!loadedgame && !playstate);
}
```

The interrupt system is started via the Sound Manager in `SDL_SetIntsPerSec(rate)`. While there is a famous game development library called Simple DirectMedia Layer (SDL), the prefix `SDL_` has nothing to do with it. It stands for SounD Low level (Simple DirectMedia Layer did not even exist in 1990).

The reason for interrupts is extensively explained in Chapter 4.13 "Audio and Heartbeat".

In short, with an OS supporting neither processes nor threads, it was the only way to have something execute concurrently with the rest of the engine.

An ISR (Interrupt Service Routine) is installed in the Interrupt Vector Table to respond to interrupts triggered by the engine.

```
void SD_Startup(void)
{
  if (SD_Started)
    return;

  t0OldService = getvect(8);  // Get old timer 0 ISR

  SDL_InitDelay();  // SDL_InitDelay() uses t0OldService

  setvect(8,SDL_t0Service);   // Set to my timer 0 ISR

  SD_Started = true;

}
```

4.6 Architecture

The source code is structured in two layers. KD_* files are high-level layers relying on low-level ID_* sub-systems called Managers interacting with the hardware.

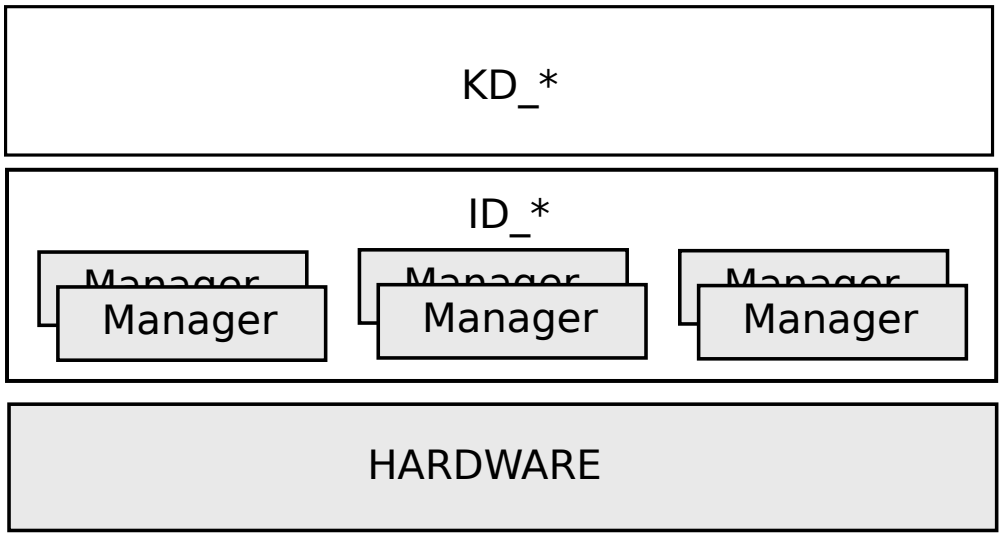


Figure 4.5: Commander Keen source code layers.

There are six managers in total:

- Memory
- Video
- Cache
- Sound
- User
- Input

The KD_ stuff was written specifically for Commander Keen while the ID_ managers are generic and later re-used (with improvements) for newer ID games (Hovertank One, Catacomb 3-D and Wolf3D).

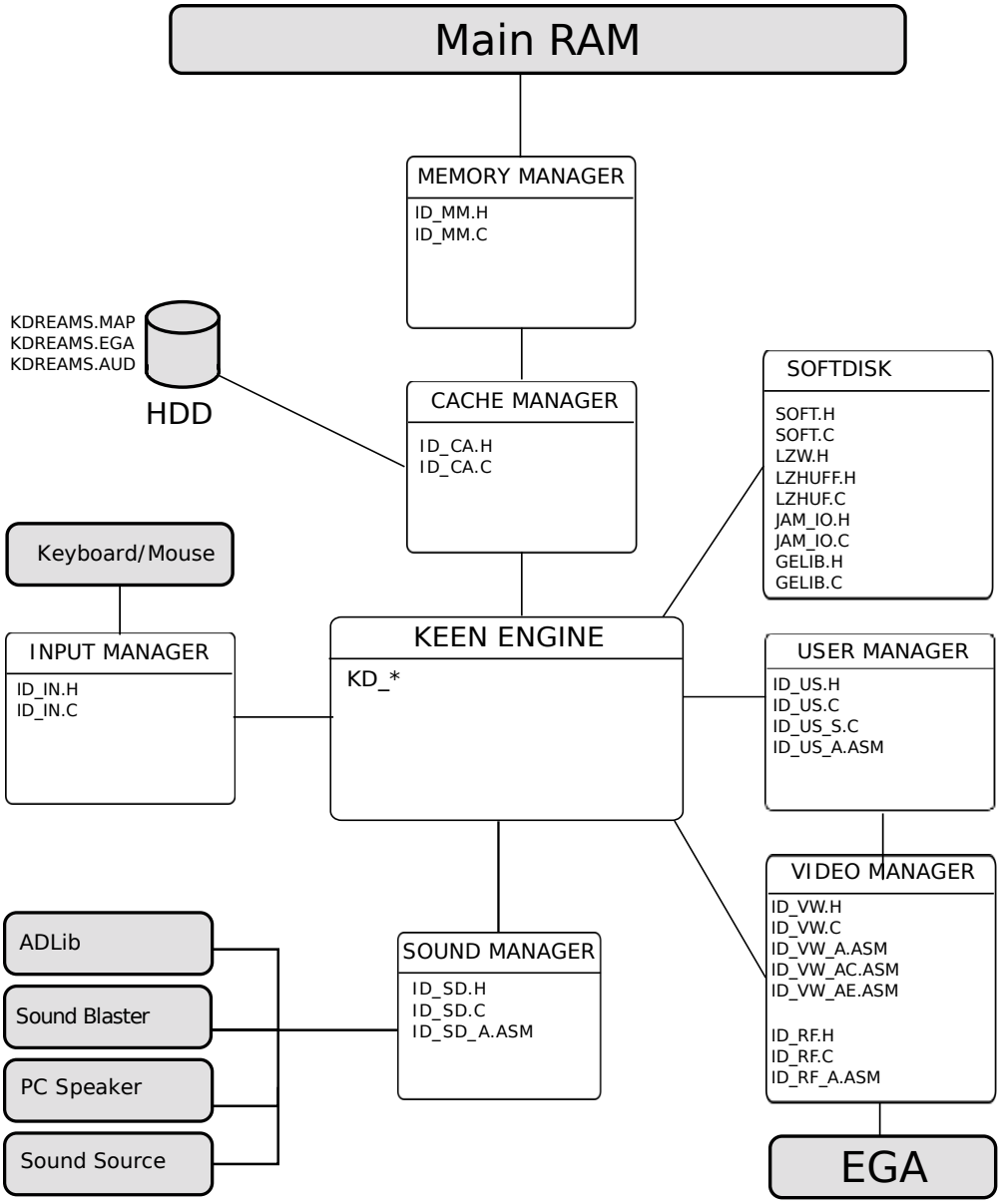


Figure 4.6: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

Next to the hard drives (HDD) you can see the assets packed as described in Chapter 3.2.

4.6.1 Memory Manager (MM)

The engine does not rely on `malloc` to manage conventional memory, as this can lead to fragmented memory and no way to compact free space. It has its own memory manager made of a linked list of "blocks" keeping track of the RAM. A block points to a starting point in RAM and has a size.

```
typedef struct mmblockstruct
{
  unsigned  start,length;
  unsigned  attributes;
  memptr    *useptr;
  struct mmblockstruct far *next;
} mmblocktype;
```

A block can be marked with attributes:

- `LOCKBIT` : This block of RAM cannot be moved during compaction.
- `PURGEBITS` : Four levels available, 0= unpurgeable, 1= purgeable, 2= not used, 3= purge first.

The memory manager starts by allocating all available RAM via `malloc`/`farmalloc` and creates a `LOCKED` block of size 1KiB at the end. The linked list uses two pointers: `HEAD` and `ROVER` which point to the second to last block.

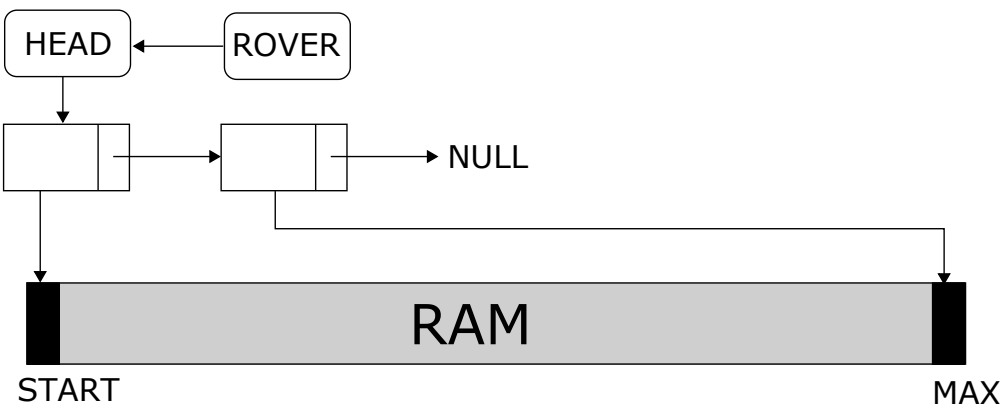


Figure 4.7: Initial memory manager state.

The engine interacts with the Memory Manager by requesting RAM (`MM_GetPtr`) and freeing RAM (`MM_FreePtr`). To allocate memory, the manager searches for "holes" between blocks. This can take up to three passes of increasing complexity:

1. After rover.
2. After head.
3. Compacting and then after rover.

The easiest case is when there is enough space after the rover. A new node is simply added to the linked list and the rover moves forward. In the next drawing, three allocation requests have succeeded: `A`, `B` and `C`.

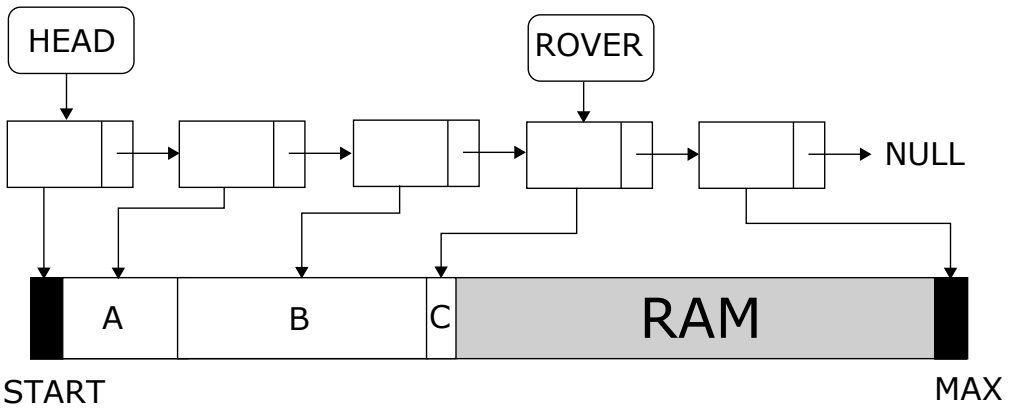


Figure 4.8: MM internal state after three pass 1 allocations.

Eventually the free RAM will be exhausted and the first pass will fail.

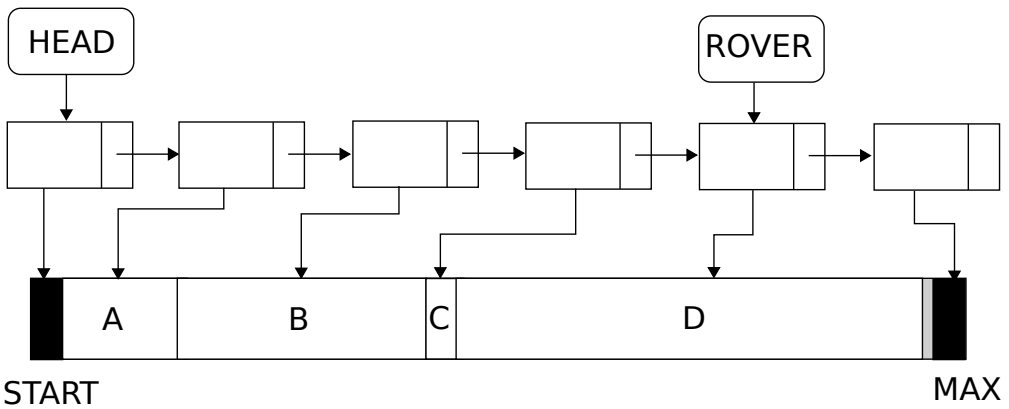


Figure 4.9: Pass 1 failure: Not enough RAM after the ROVER.

If the first pass fails, the second pass looks for a "hole" between the head and the rover. This pass will also purge unused blocks. If for example block B was marked as `PURGEABLE`, it will be deleted and replaced with the new block E. At this point fragmentation starts to appear (like if `malloc` was used).

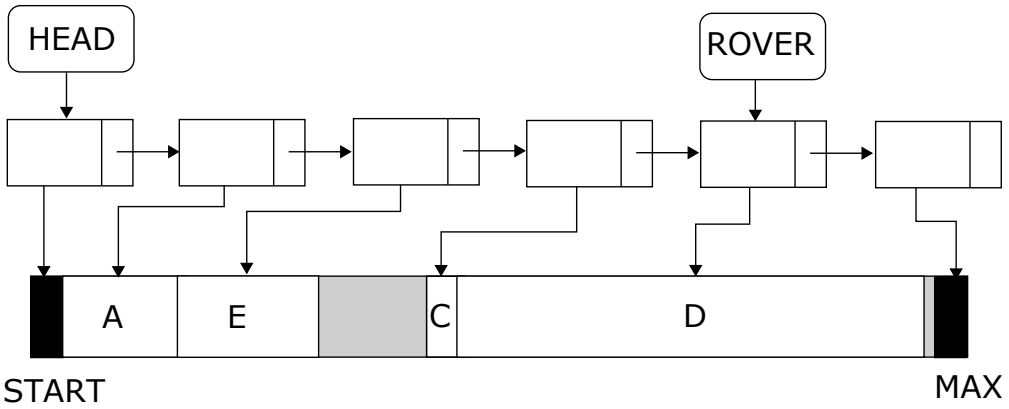


Figure 4.10: `B` was purged. `E` was allocated in pass 2.

If the first and second pass fail, there is no continuous block of memory large enough to satisfy the request. The manager will then iterate through the entire linked list and do two things: delete blocks marked as purgeable, and compact the RAM by moving blocks.

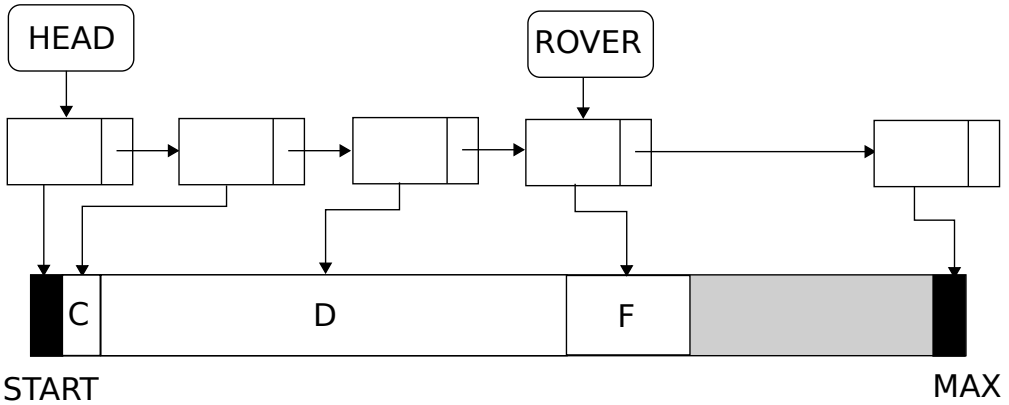


Figure 4.11: `A` and `E` were purged. `C` and `D` compacted. `F` allocated in pass3.

But if memory is moved around, how do previous allocations still point to what they did before the compaction phase? Notice that a `mmblockstruct` has a `useptr` pointer which

points to the owner of a block. When memory is moved, the owner of the block is also updated.

As some blocks are marked as `LOCKED`, compacting can be disturbed. Upon encountering a locked block, compacting stops and the next block will be moved immediately after the locked block, even if there was space available between the last block and the locked block.

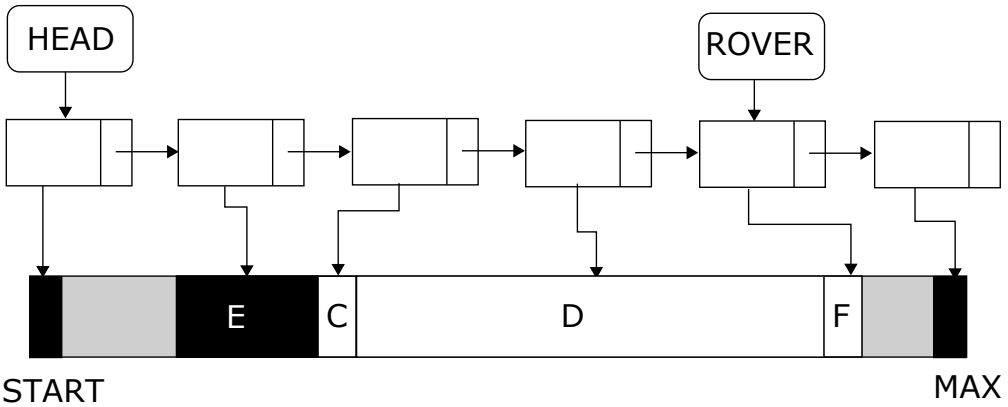


Figure 4.12: `E` is locked and cannot be compacted.

In the above drawing, C was moved after E, even though it could have been moved before. Avoiding this waste would have made the memory manager more complicated, so the waste was deemed acceptable. Often in designing a component you have to be practical and establish a certain trade off between accuracy and complexity.

4.6.2 Video Manager (VW & RF)

The video manager features two parts:

- The `VW_*` layer is made of both C and ASM, where the C functions abstract away EGA register manipulation via assembly routines.
- The `RF_*` layer is used to refresh tiles, and is also made of both C and ASM code.

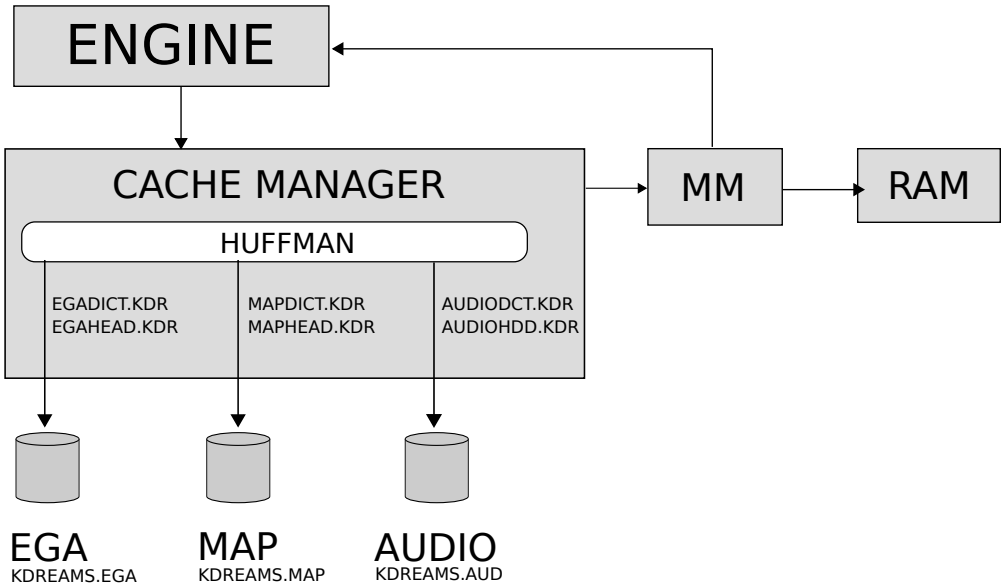
The video manager is described extensively in section 4.8 on page 104.

4.6.3 Cache Manager (CA)

The cache manager is a small but critical component. It loads and decompresses maps, graphics and audio resources stored on the filesystem and makes them available in RAM. Assets of each kind are stored into three files:

- A header file containing the offset to allow translation from asset ID to byte offset in the data file.
- A compression dictionary to decompress each asset
- The data file containing the assets

Details of each asset file are explained in chapter 3.1. The header and dictionary files are provided with the source code in the `static` folder and contain `*.KDR` extension. Both file types are hardcoded and required during compilation (they are converted into an `OBJ` file using `makeobj.c`). The data file containing the assets is not part of the source code and must be acquired via downloading the shareware version. All resources are compressed using a traditional huffman method for (de-)compression.



To manage and keep track of the assets to be loaded into memory, an array `gr_needed[]` is maintained to mark if an asset needs to be loaded from disk. The index of this array refers to the enum of the graphic assets. By using the eight bits the array can maintain the required assets for different levels. The cache manager starts with an empty `gr_needed[]` array.

level bit:	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
STARTFONT								
CTL_STARTUPPIC								
CTL_HELPUPPIC								
...								
KEENSTANDRSPR								
KEENRUNR1SPR								
...								
SCOREBOXSPR								
...								
TILE8								
TILE8M								
TILE16 #1								
TILE16 #2								
...								
TILE16M #1								
TILE16M #2								
...								

Figure 4.13: Initiating `gr_needed[]` array.

When new resources needs to be cached in memory, all required assets are marked by setting the current level bit (bit 1) to 1.

```
#define CA_MarkGrChunk(chunk) grneeded[chunk]|=ca_levelbit

void InitGame (void)
{
  CA_ClearMarks ();  // Clears out all the marks at the
   current level

  // Mark assets to be cached in memory
  CA_MarkGrChunk(STARTFONT);
  CA_MarkGrChunk(STARTFONTM);
  CA_MarkGrChunk(STARTTILE8);
  CA_MarkGrChunk(STARTTILE8M);
  for (i=KEEN_LUMP_START;i<=KEEN_LUMP_END;i++)
    CA_MarkGrChunk(i);

  CA_CacheMarks (NULL, 0); // Cache marked assets into
   memory
}
```

level bit:	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
STARTFONT								■
CTL_STARTUPPIC								
CTL_HELPUPPIC								
...								
KEENSTANDRSPR								■
KEENRUNR1SPR								■
...								
SCOREBOXSPR								■
...								
TILE8								■
TILE8M								■
TILE16 #1								■
TILE16 #2								
...								
TILE16M #1								
TILE16M #2								■
...								

Figure 4.14: Mark all assets required for the new map in level bit 1.

The function `CA_CacheMarks()` loads and decompress all required graphical assets from disk to memory for the current bit level. Now if during playing the game we open the control panel (e.g. to pause the game), the assets for the control panel needs to be cached. By increasing the bit level we keep the assets for control panel separated from the map.

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

  if (ca_levelnum==7)
    Quit ("CA_UpLevel: Up past level 7!");

  ca_levelbit<<=1;
  ca_levelnum++;
}
```

The `gr_needed[]` array looks as follows

level bit:	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*
STARTFONT							■	■
CTL_STARTUPPIC							■	
CTL_HELPUPPIC							■	
...								
KEENSTANDRSPR								■
KEENRUNR1SPR								■
...								
SCOREBOXSPR								■
...								
TILE8								■
TILE8M								■
TILE16 #1								■
TILE16 #2								
...								
TILE16M #1								
TILE16M #2								■
...								

Figure 4.15: Mark all assets required for the control panel in level bit 2.

If the control panel is closed again, it simply lowers the bit level and calls the function `CA_CacheMarks()` to reload any map assets removed from memory.

```
void CA_DownLevel (void)
{
  if (!ca_levelnum)
    Quit ("CA_DownLevel: Down past level 0!");
  ca_levelbit>>=1;
  ca_levelnum--;

  //recaches everyting from the previous level
  CA_CacheMarks(titleptr[ca_levelnum], 1);
}
```

4.6.4 User Manager (US)

The user manager is responsible for text layout and control panels like loading and saving games, configure controls and setting sound device.

Once we start the game, we move the display to EGA graphic mode `0x0D`. Here we can't directly print characters on the screen anymore. So a key function of the User Manager is to print text on a given pixel location. When a high-level routine needs to draw a string, it is first passed to `USL_MeasureString` which does all measurement (e.g. height and total width of string) and then to `USL_DrawString` which passes this information to the Video Manager (`VW_DrawPropString`), which takes care of rendition. In the graphic assets file the complete font is stored with the following information for each character:

- The width of the character
- The location in memory where each character is stored as a bitmap

Each character has the same height of 10 pixels, but the character width could vary as illustrated in Figure 4.16.

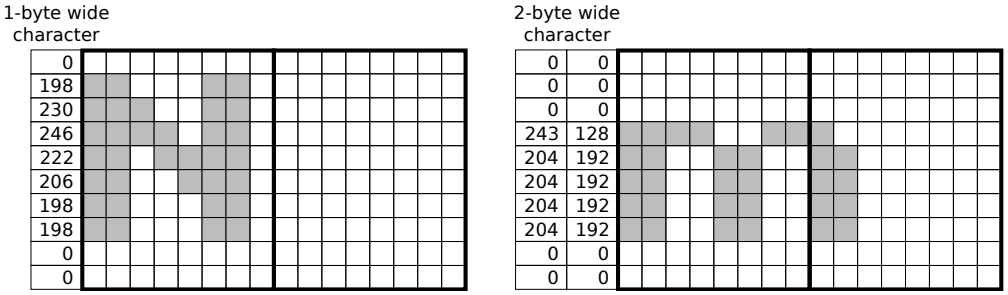


Figure 4.16: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

As explained in section 2.3.8 on page 37, each 8 pixels are represented by 1 byte per memory bank. So how do we print a character which is not perfectly aligned with the memory layout? Here a trick of bitshift tabels is being used.

The default table (`shiftdata0`) is defined as integer (16 bits) and contains all values from 0-255. Now, we shift this entire table 1 bit to the right. We can translate the bit shift back into an integer and store these values again in a table (`shiftdata1`). We can do this again when we shift another bit, until we cycled through the 8-bits. So at the end we have created 7 shift tables to fully cycle through 8-bits. In Figure 4.17 the bitshift for the values '198' is illustrated.

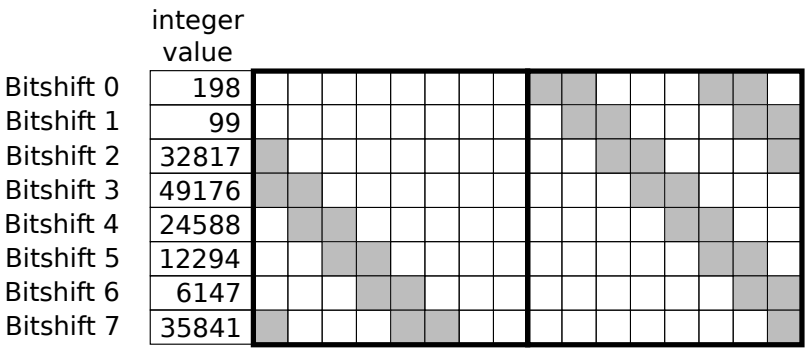


Figure 4.17: Right bitshift [0-7] for 198.

Each bitshift table is generated in `id_vw_a.asm`, see below bitshift 5.

```
LABEL shiftdata5 WORD
  dw     0, 2048, 4096, 6144, 8192,10240,12288,14336,16384,18432,20480,22528,24576,26624
  dw 28672,30720,32768,34816,36864,38912,40960,43008,45056,47104,49152,51200,53248,55296
  dw 57344,59392,61440,63488,    1, 2049, 4097, 6145, 8193,10241,12289,14337,16385,18433
  dw 20481,22529,24577,26625,28673,30721,32769,34817,36865,38913,40961,43009,45057,47105
  dw 49153,51201,53249,55297,57345,59393,61441,63489,    2, 2050, 4098, 6146, 8194,10242
  dw 12290,14338,16386,18434,20482,22530,24578,26626,28674,30722,32770,34818,36866,38914
  dw 40962,43010,45058,47106,49154,51202,53250,55298,57346,59394,61442,63490,    3, 2051
  dw  4099, 6147, 8195,10243,12291,14339,16387,18435,20483,22531,24579,26627,28675,30723
  dw 32771,34819,36867,38915,40963,43011,45059,47107,49155,51203,53251,55299,57347,59395
  dw 61443,63491,    4, 2052, 4100, 6148, 8196,10244,12292,14340,16388,18436,20484,22532
  dw 24580,26628,28676,30724,32772,34820,36868,38916,40964,43012,45060,47108,49156,51204
  dw 53252,55300,57348,59396,61444,63492,    5, 2053, 4101, 6149, 8197,10245,12293,14341
  dw 16389,18437,20485,22533,24581,26629,28677,30725,32773,34821,36869,38917,40965,43013
  dw 45061,47109,49157,51205,53253,55301,57349,59397,61445,63493,    6, 2054, 4102, 6150
  dw  8198,10246,12294,14342,16390,18438,20486,22534,24582,26630,28678,30726,32774,34822
  dw 36870,38918,40966,43014,45062,47110,49158,51206,53254,55302,57350,59398,61446,63494
  dw     7, 2055, 4103, 6151, 8199,10247,12295,14343,16391,18439,20487,22535,24583,26631
  dw 28679,30727,32775,34823,36871,38919,40967,43015,45063,47111,49159,51207,53255,55303
  dw 57351,59399,61447,63495
```

Now let's take the example of printing 'N', with an offset of 3 pixels. A simple lookup in

`shiftdata3` results in the 3-bit shifted 'N'. Note that you first display the low byte and then the high byte value.

unshifted bitmap value	shiftdata3 bitmap value
0	0
198	49176
230	49180
246	49182
222	49179
206	49177
198	49176
198	49176
0	0
0	0

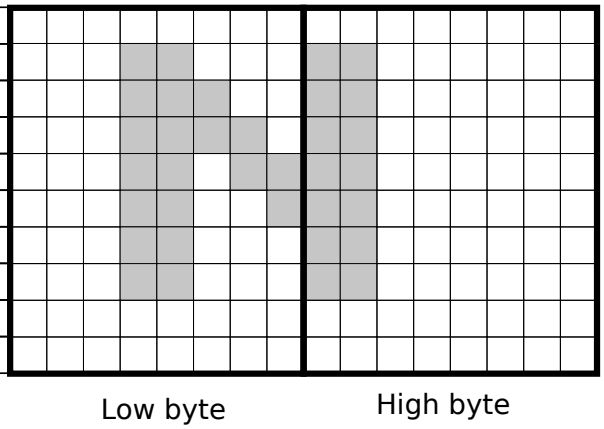


Figure 4.18: Bitshift 'N' over 3 bits using bit shift tables.

Once both bytes are copied to the data buffer, the buffer pointer is increased with the character width and then the next character is copied.

```
charloc   = 2      ;pointers to every character
BUFFWIDTH = 50     ;buffer width is 50 characters

PROC  ShiftPropChar NEAR

  mov es,[grsegs+STARTFONT*2] ;segment of font to use
  mov bx,[es:charloc+bx]      ;BX holds pointer to
   character data

; look up which shift table to use, based on bufferbit
  mov di,[bufferbit]      ;pixel offset within byte [0-7]
  shl di,1
  mov bp,[shifttabletable+di] ;BP holds pointer to shift
   table

  mov di,OFFSET databuffer
  add di,[bufferbyte]     ;DI holds pointer to buffer
  mov cx,[es:pcharheight] ;CX contains character height
  mov dx,BUFFWIDTH

; write one byte character
shift1wide:
  dec dx
EVEN
@@loop1:
  SHIFTNOXOR
  add di,dx         ; next line in buffer
  loop  @@loop1
  ret
ENDP

; Macros to table shift a byte of font
MACRO SHIFTNOXOR
  mov al,[es:bx]    ; source of font data
  xor ah,ah
  shl ax,1
  mov si,ax
  mov ax,[bp+si]    ; table shift into two bytes
  or  [di],al       ; OR with first byte
  inc di
  mov [di],ah       ; replace next byte
  inc bx            ; next source byte
ENDM
```

4.6.5 Sound Manager (SD)

The Sound Manager abstracts interaction with all four sound systems supported: PC Speaker, AdLib, Sound Blaster, and Disney Sound Source. It is a beast of its own since it doesn't run inside the engine. Instead it is called via IRQ at a much higher frequency than the engine (the engine runs at a maximum 70Hz, while the sound manager ranges from 140Hz to 700Hz). It must run quickly and is therefore written in small and fast routines.

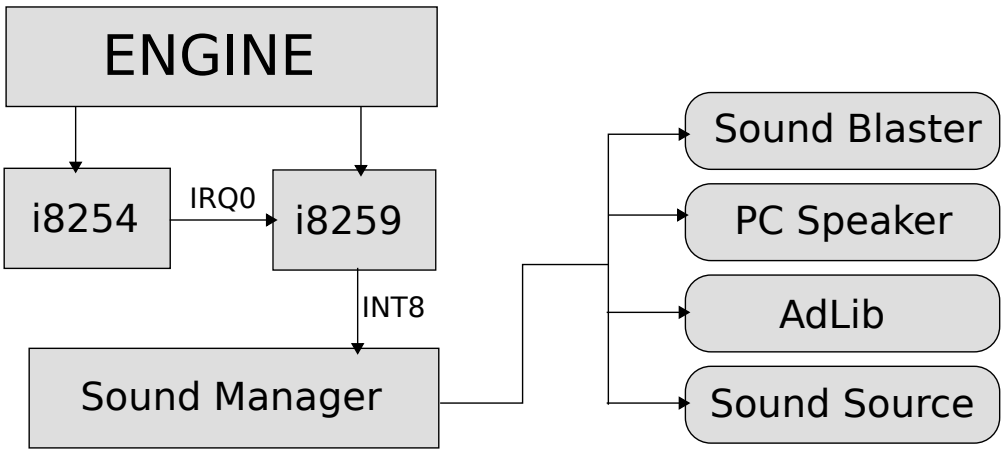


Figure 4.19: Sound system architecture.

The sound manager is described extensively in the "Sound and Music" section.

4.6.6 Input Manager (IN)

The input manager abstracts interactions with joystick, keyboard, and mouse. It features the boring boilerplate code to deal with PS/2, Serial, and DA-15 ports, with each using their own I/O addresses.

4.6.7 Softdisk files

The only function for the softdisk files is to load and show the intro screen bitmap, using `LoadLIBShape` from soft.c. Most of the functions in these files are actually not used and therefor not further discussed in this book.

4.7 Startup

As the game engine starts, it will first load the memory manager. Then it will check if there is at least 335KiB of RAM available. If not, it gives a warning, but you can continue with the game. But most likely somewhere soon the game will either crash or receives an "Out of memory" error.

After succesfully starting the game the intro image is diplayed, which is a Deluxe Paint-Bitmap image (*.LBM). After the user has hit any key, the intro image is unloaded from RAM to make more room for runtime and the control panel is shown.

Figure 4.20: Keen Dreams intro screen

4.8 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 4.21. 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 4.21 only 69 tiles of the total 260 tiles need to be refreshed, which is 27% of the screen!

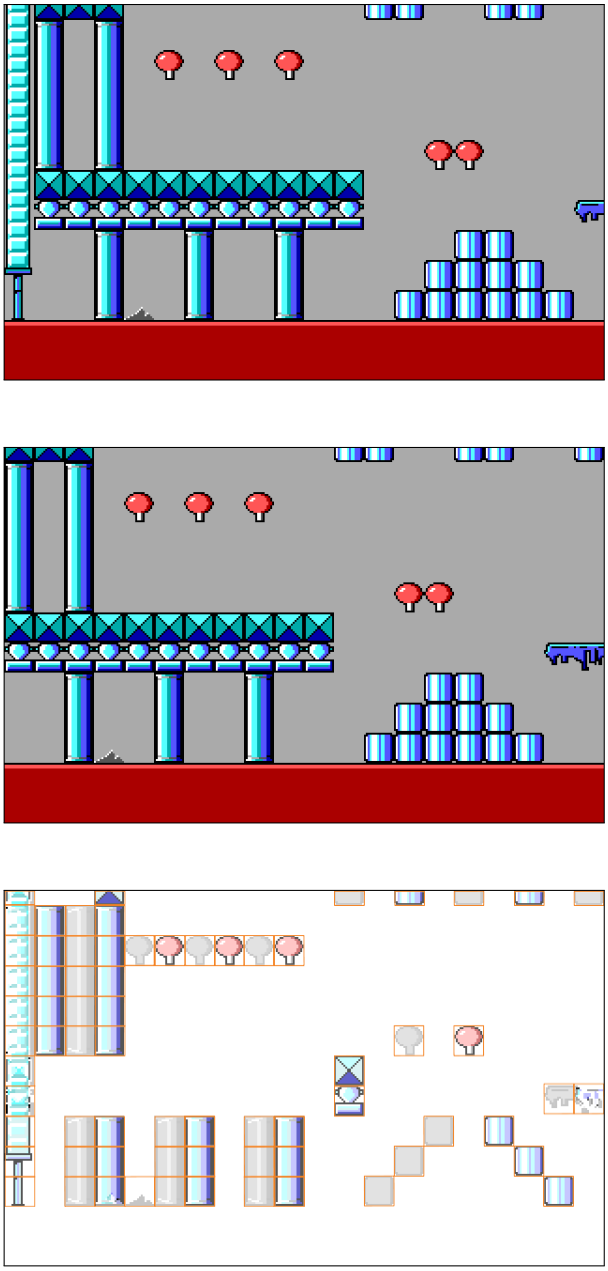


Figure 4.21: Start of the world, moved one tile to the right and difference.

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.

4.8.1 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
```

4.8.2 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
```

4.8.3 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 4.22:

- 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

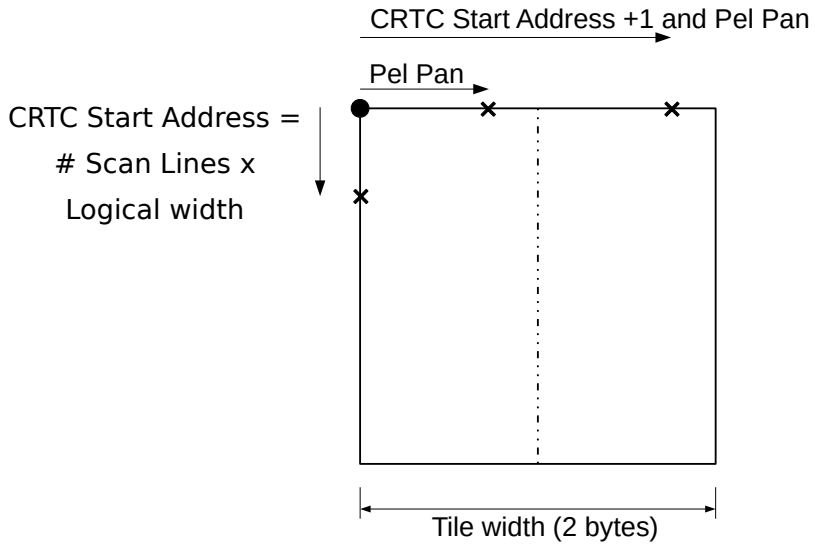


Figure 4.22: Smooth scrolling in EGA.

4.9 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 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.

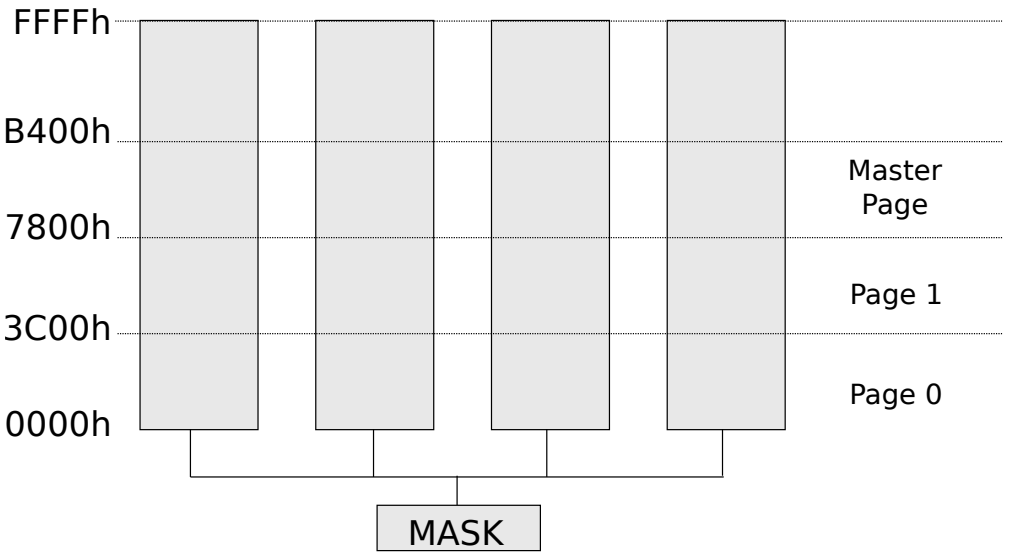


Figure 4.23: Virtual screen layout on EGA card.

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.

4.10 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.

4.10.1 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

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

So summarized, as illustrated in Figure 4.23, 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.

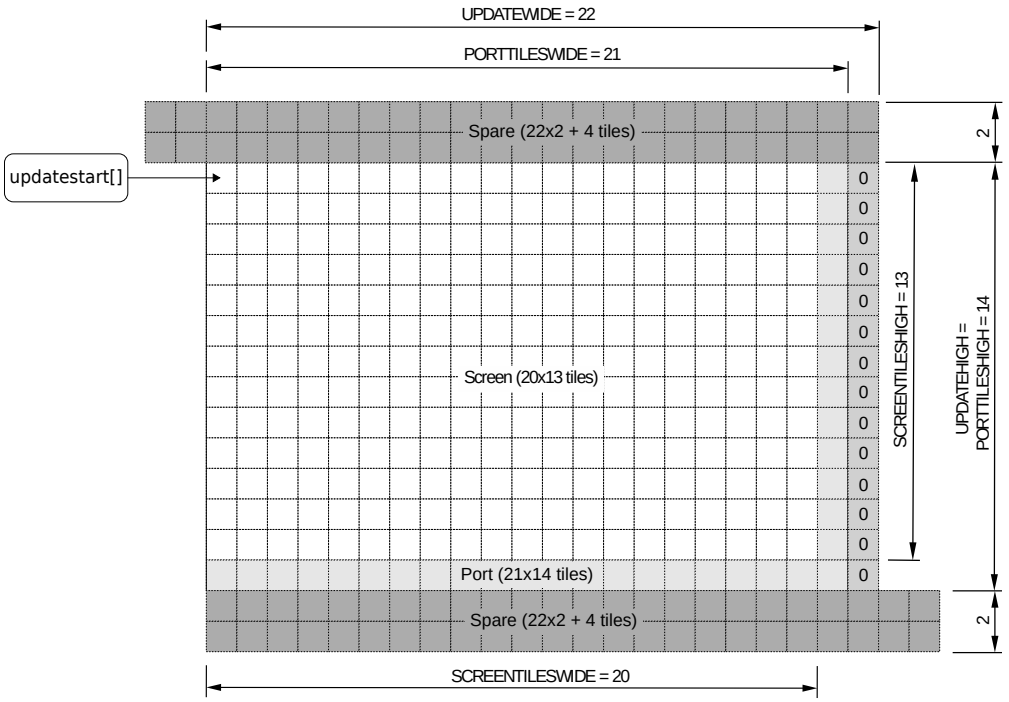


Figure 4.24: Tile view and tile buffer layout.

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.

4.10.2 Adaptive tile refreshment in Commander Keen 1-3

In the this section we explain how the first 3 versions of the game are working[3]. 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.

[3]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.

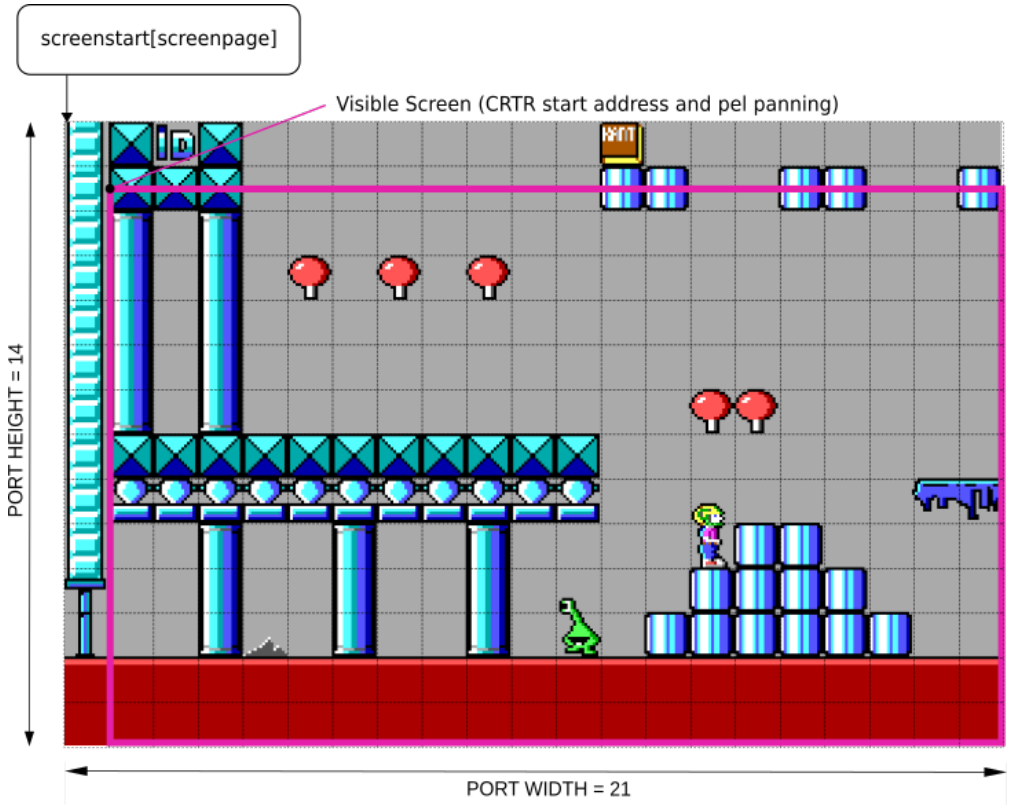


Figure 4.25: Step 1: Scroll screen to the right

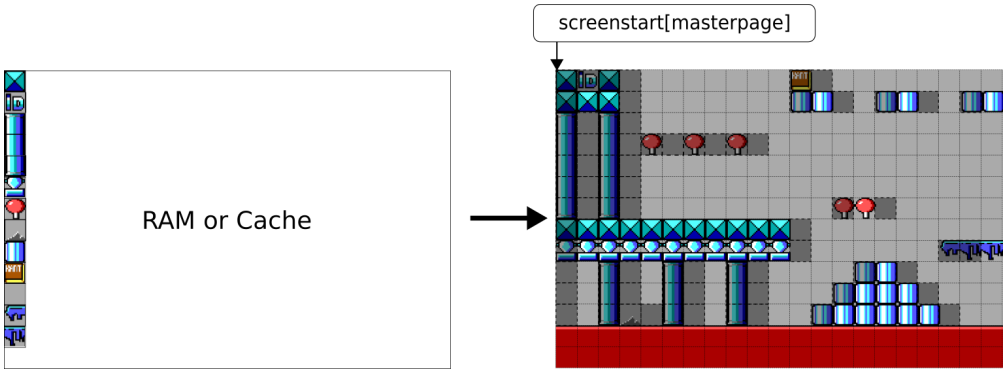


Figure 4.26: Step 2: Update changed tiles in masterscreen, marked in dark grey

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.

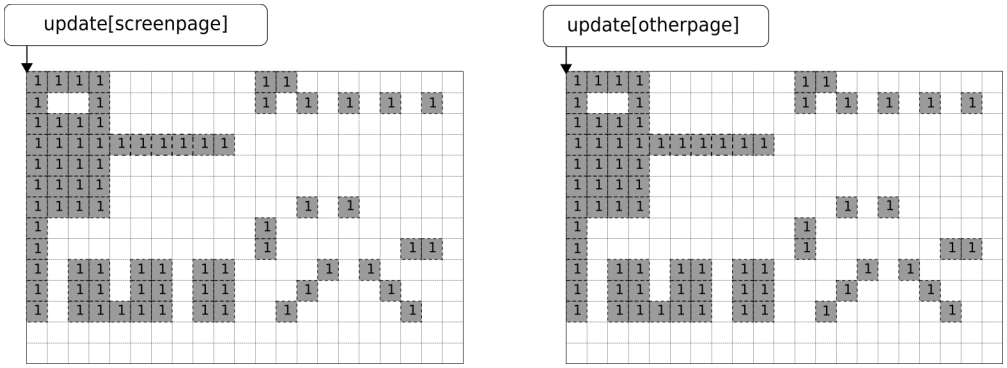


Figure 4.27: Mark all changed tiles with '1' in both tile buffer and tile view array.

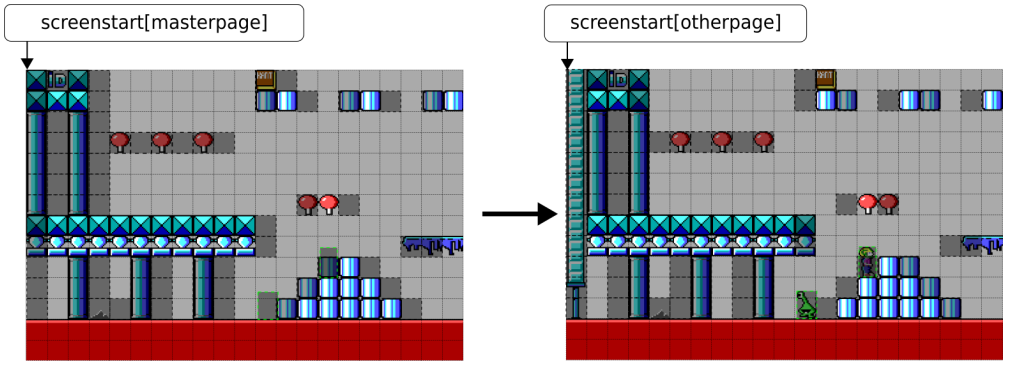


Figure 4.28: Step 3 and 4: Copy tiles from master to buffer screen and remove sprites

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 4.28) from the master screen to the buffer screen, and mark the corresponding tiles only in the tile buffer array with a '2'.

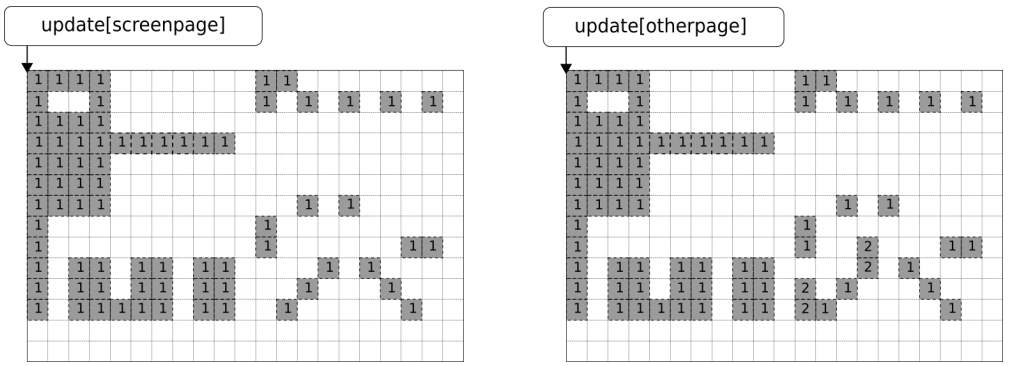


Figure 4.29: Mark removed sprites with '2' in tile buffer array only.

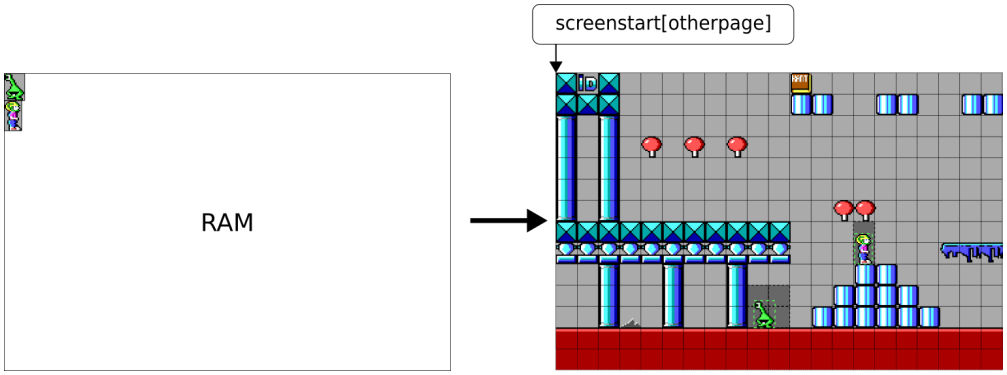


Figure 4.30: Step 5: Scan sprite list and copy sprite onto buffer screen

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’.

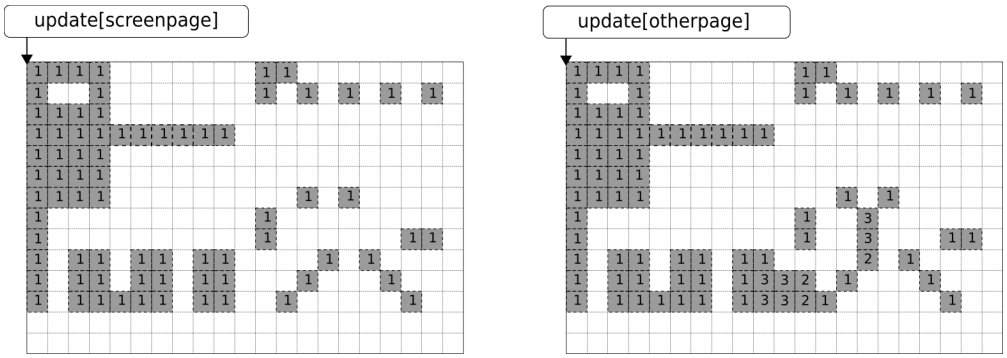


Figure 4.31: Mark new sprites locations with ’3’ in tile buffer array only.

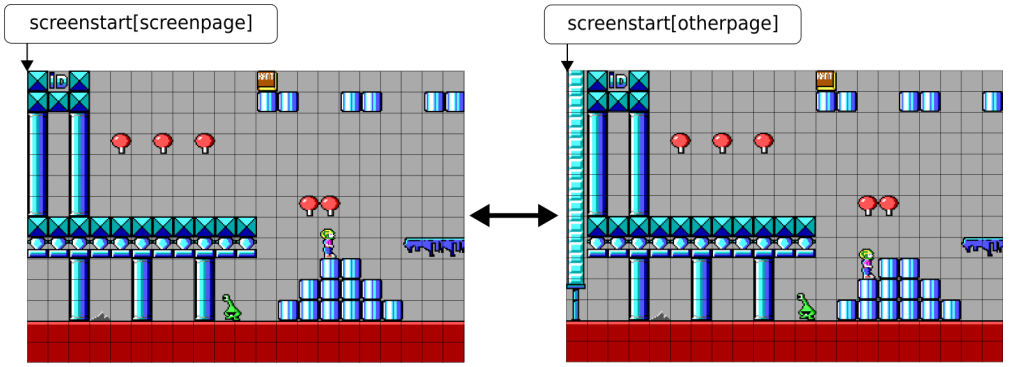


Figure 4.32: Step 6: Swap buffer and screen page

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).

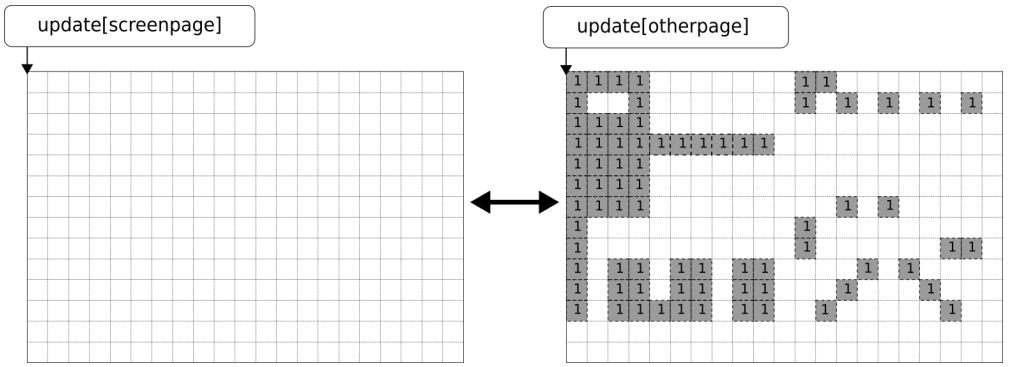


Figure 4.33: Clear tile update array and swap arrays.

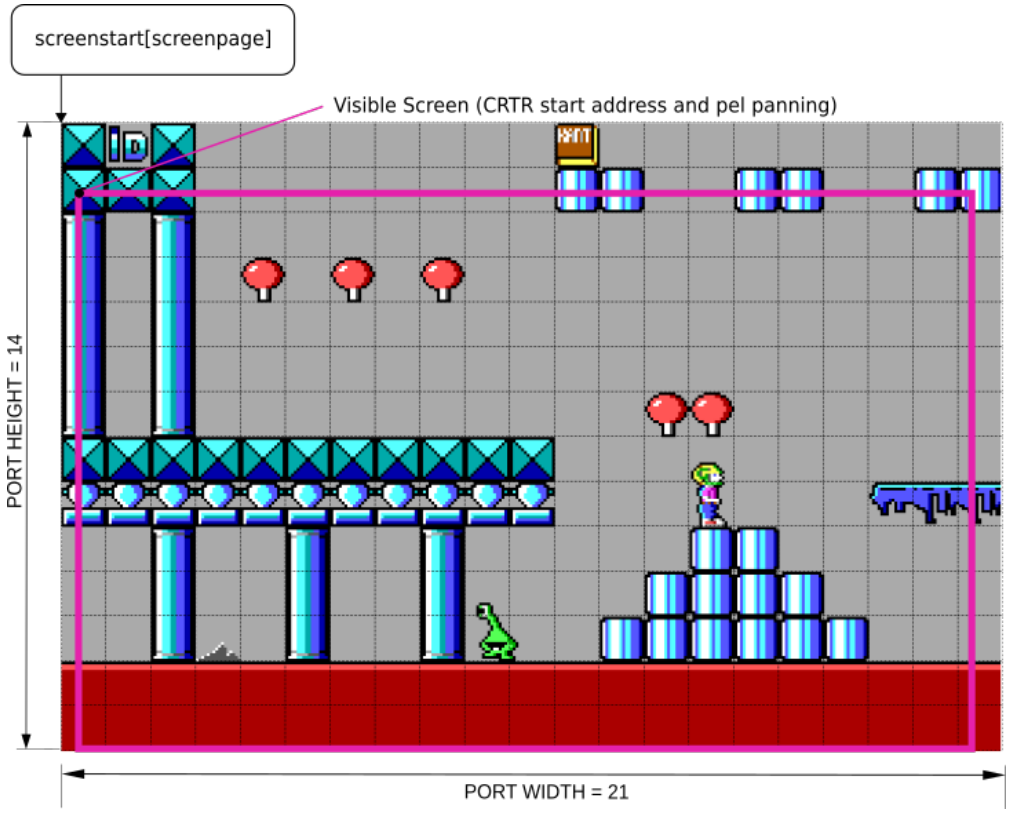


Figure 4.34: Step 6: Swap buffer and screen page

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.

4.10.3 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.

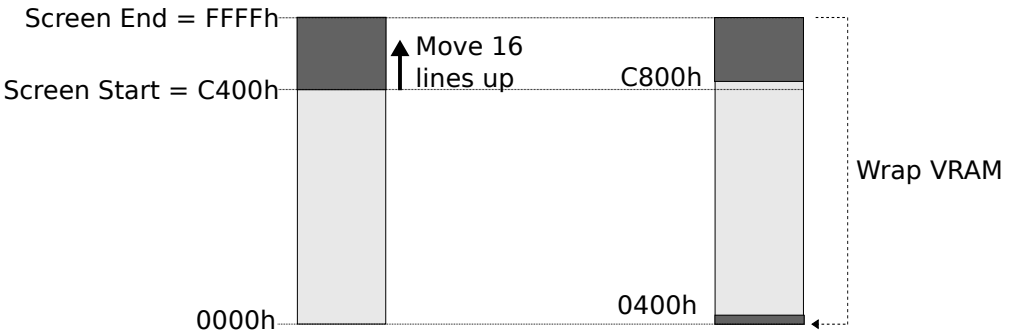


Figure 4.35: Wrap virtual screen around the EGA memory

There was however an issue with the introduction of Super VGA cards, which had typically more than 256kB RAM[4]. 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 4.24 on page 112, 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 4.37.

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

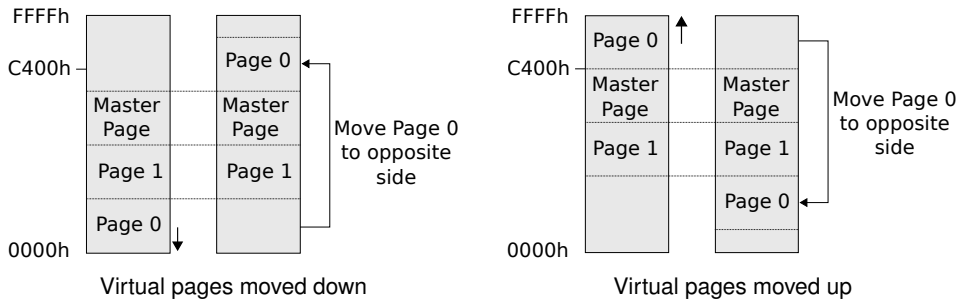


Figure 4.36: Move screen to opposite end of VRAM buffer

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     (0x100001-31*SCREENSPACE)

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

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

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

hit? As explained in Section 2.3.8 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.

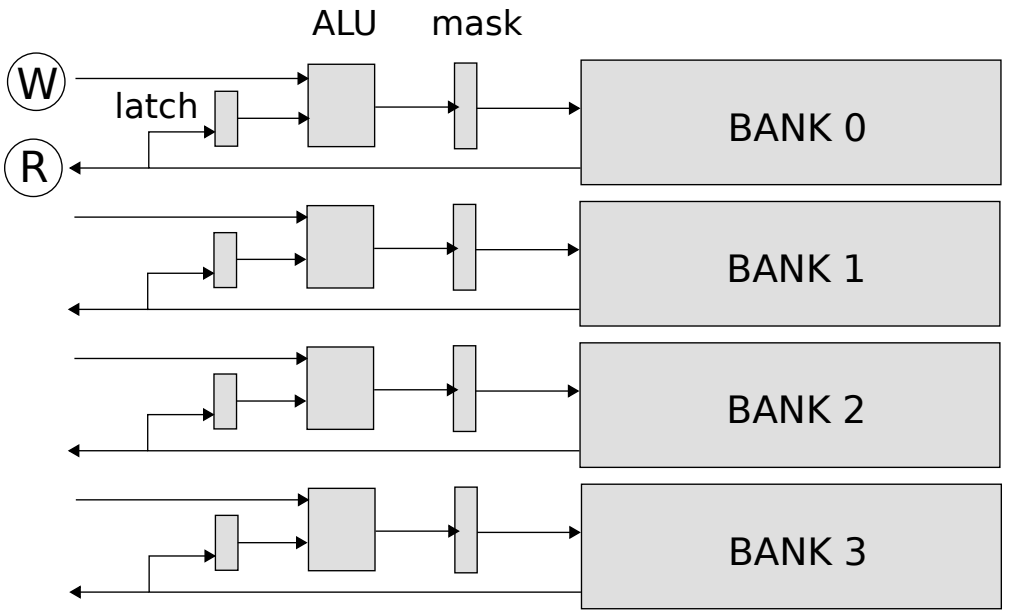


Figure 4.37: Latches memorize read operations from each bank. The memorized value can be used for later writes.

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.

4.10.4 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.

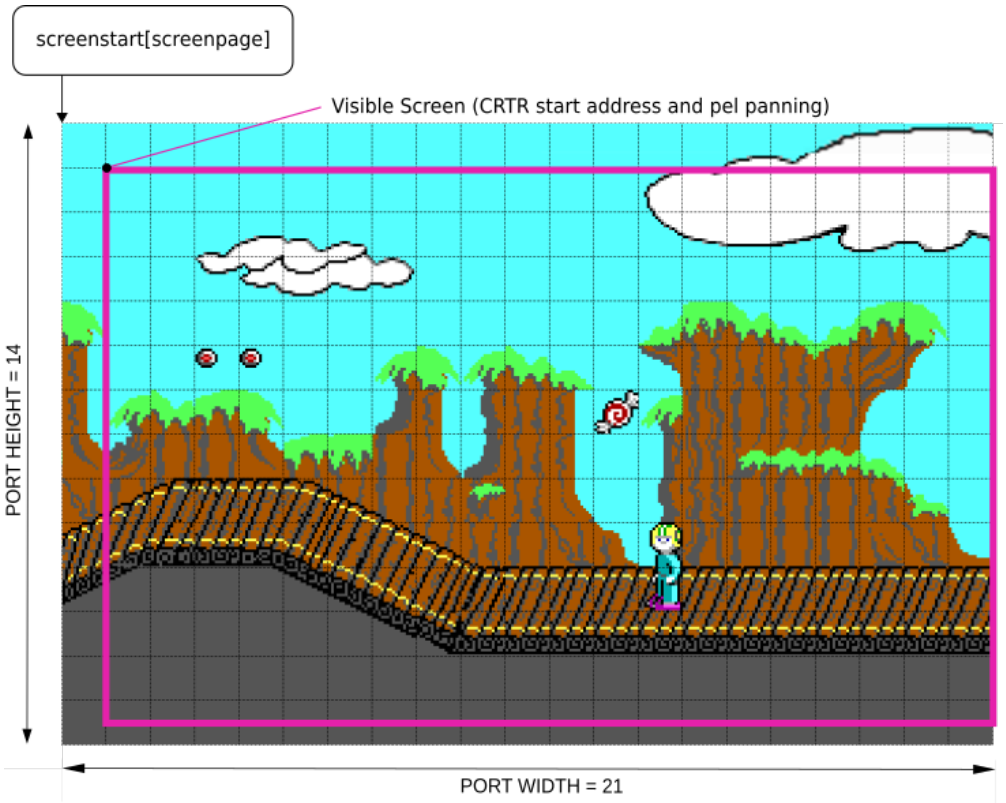


Figure 4.38: Step 1: Scroll screen to the left

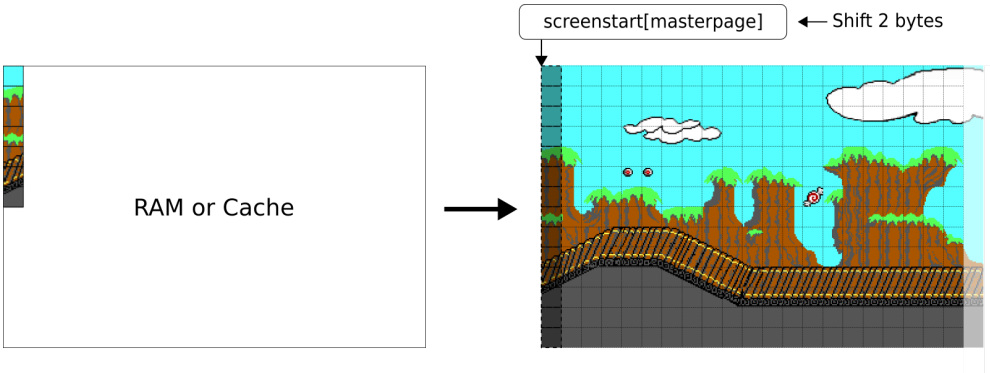


Figure 4.39: Step 2 and 3: Shift screen pointer and add column to VRAM

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'.

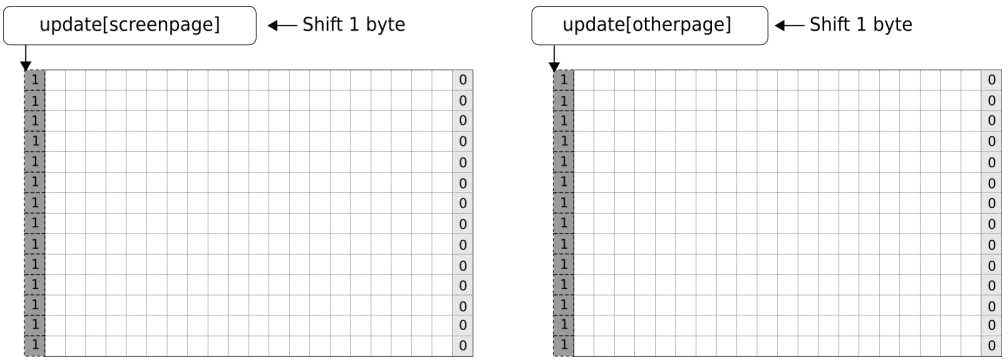


Figure 4.40: Mark new column in both tile buffer and display array.

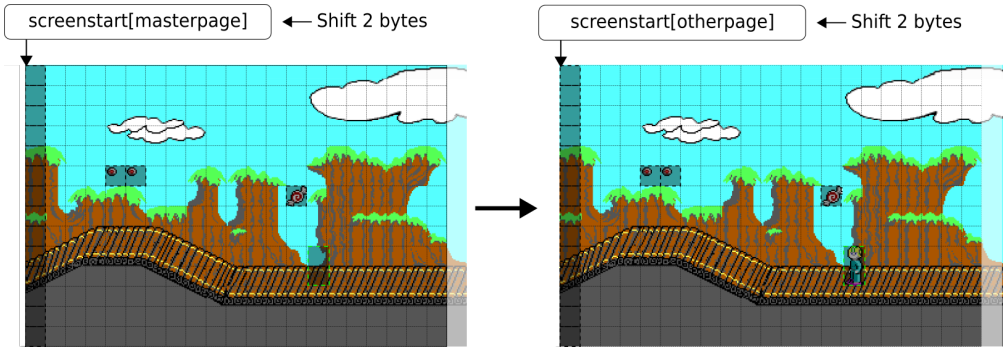


Figure 4.41: Step 4 and 5: Copy master to buffer screen and remove sprites

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 4.42 and Figure 4.43.

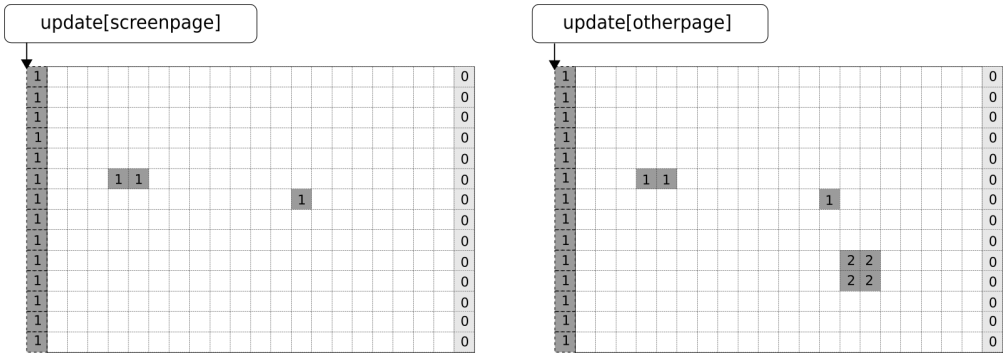


Figure 4.42: Tile buffer array with removed sprites marked with '2'.

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

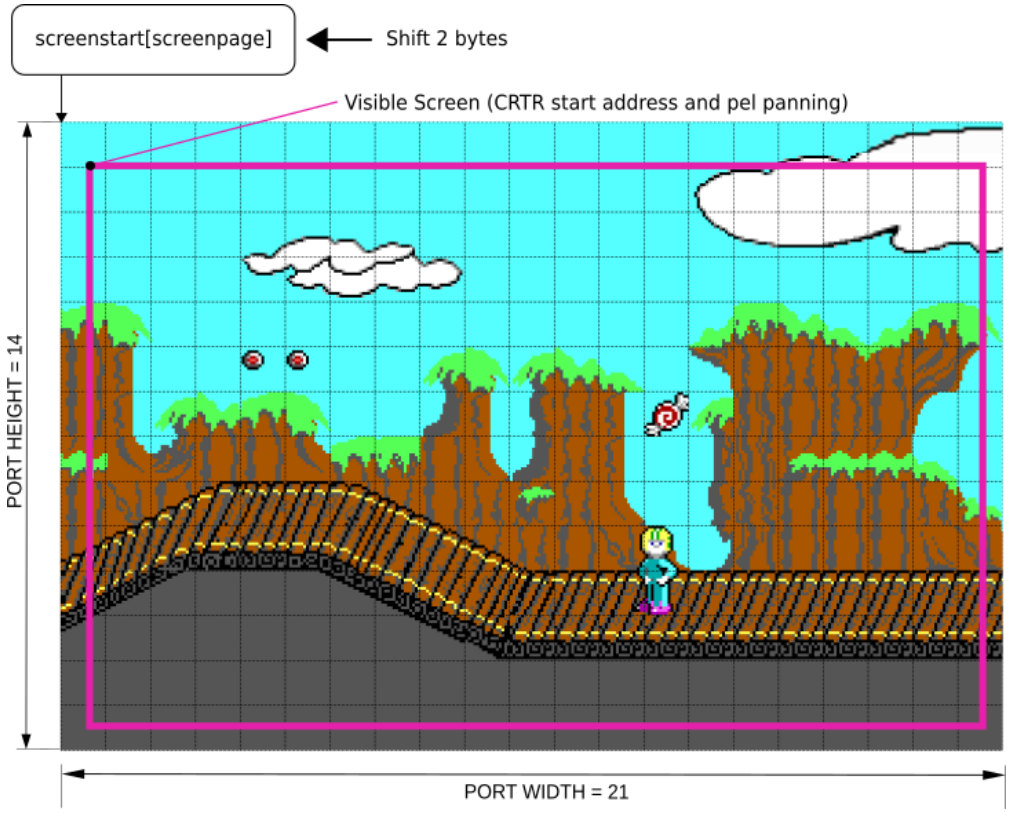


Figure 4.43: Step 7: Updated screen after swapping buffer and screen page

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!

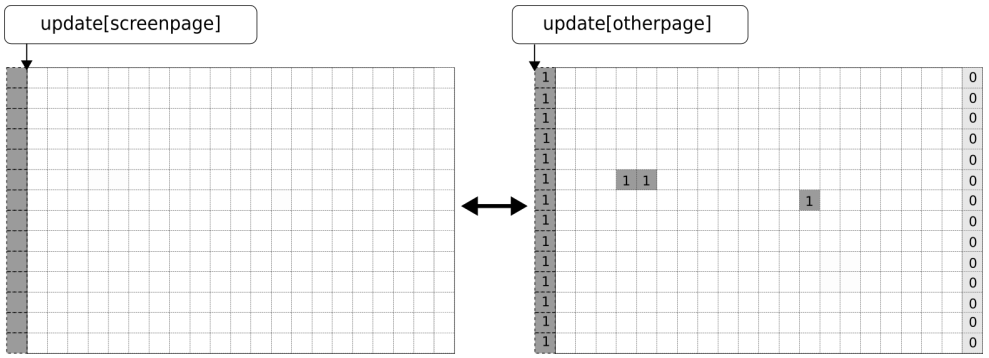


Figure 4.44: Clear and reset tile buffer array and swap arrays.

Now we can also explain why the tile buffer and view arrays are 2 tiles wider on all sides than the tile view port (see Figure 4.23). 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 4.46. 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).

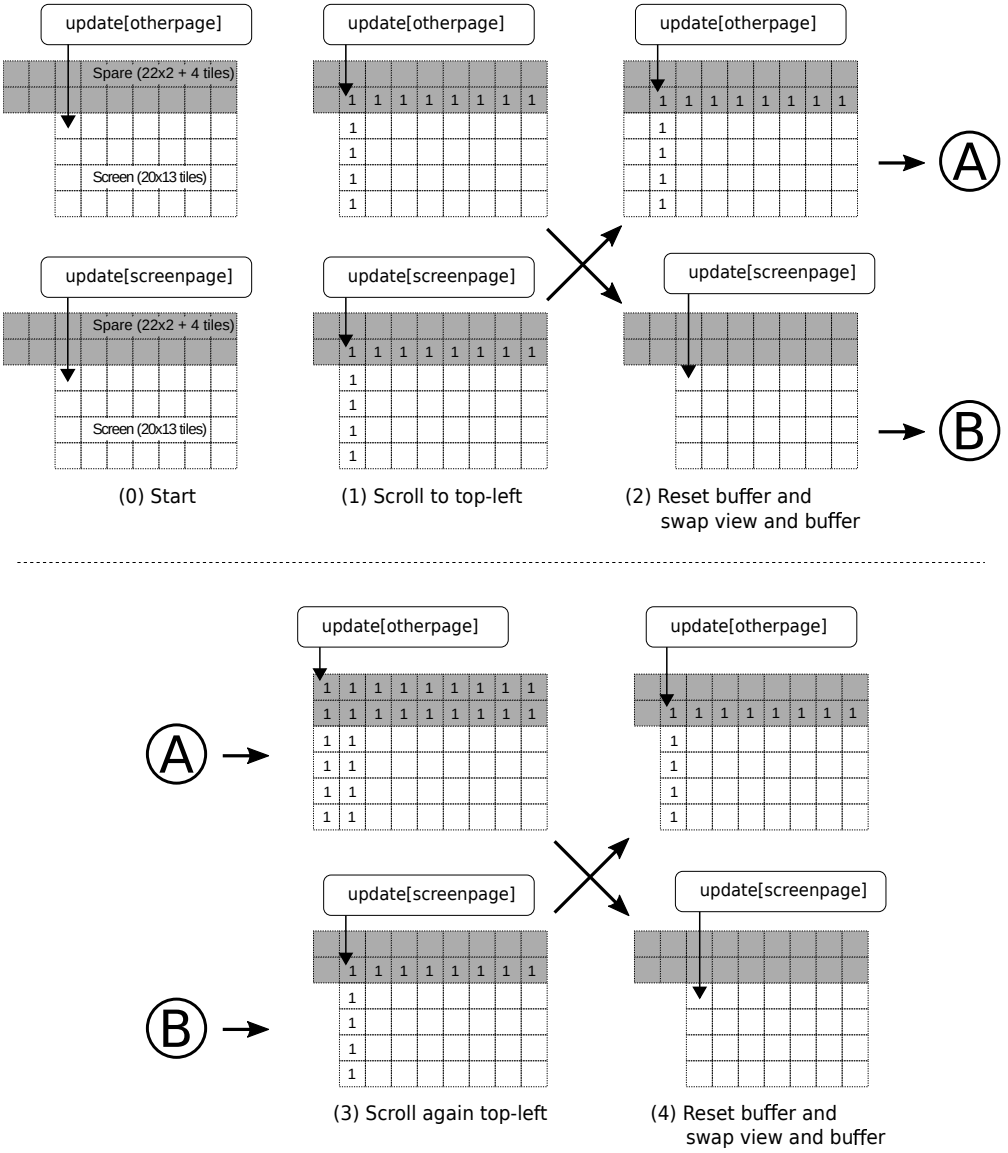
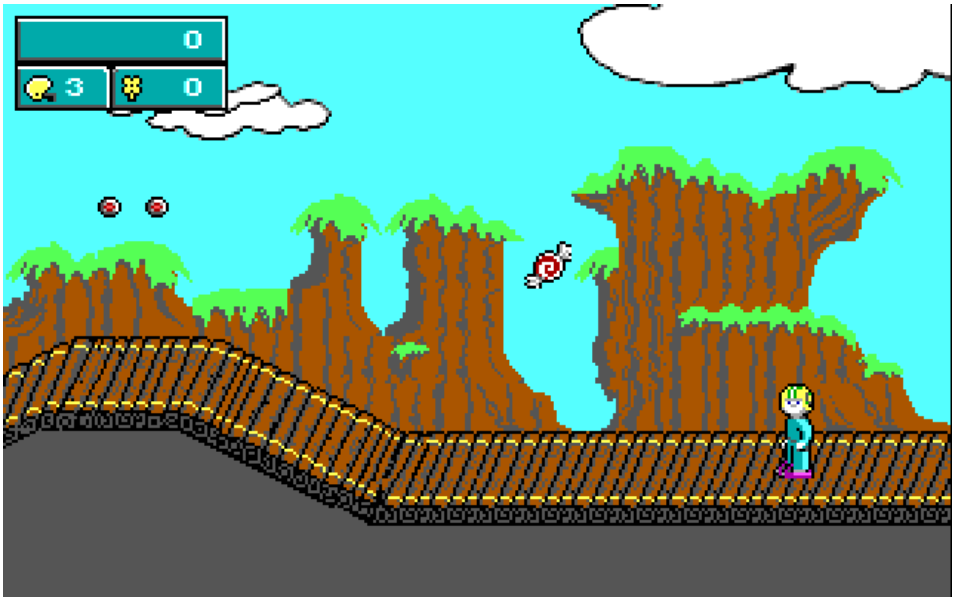


Figure 4.45: scroll to top-left tile

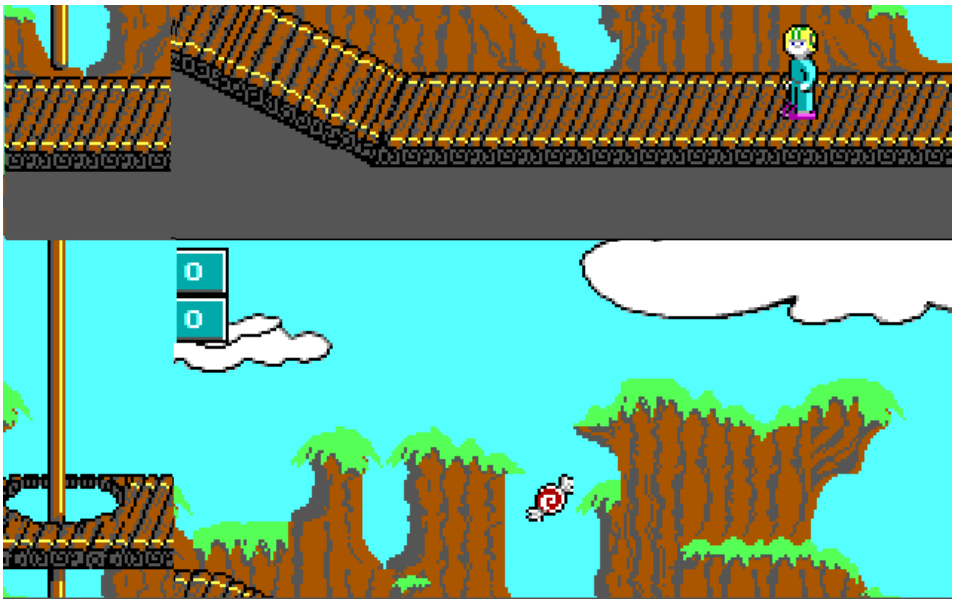
Now, if the screen is again scrolling to the top-left one can see why there is a need for the 2nd row in the buffer. After this cycle the `*update[otherpage]` pointer is cleared and resetted as shown in the last illustration of Figure 4.46.

4.10.5 Screen refresh

Flipping between the pages is as simple as setting the CRTC start address registers to page 0 or page 1 starting point, as explained in Section 4.8.1. 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 is pointing to 0x0000. 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 picking up a value of 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 option 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 via the Input Status 1 Register, where a value of 1 indicates the display is in a horizontal or vertical retracement[5].

[5]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'.

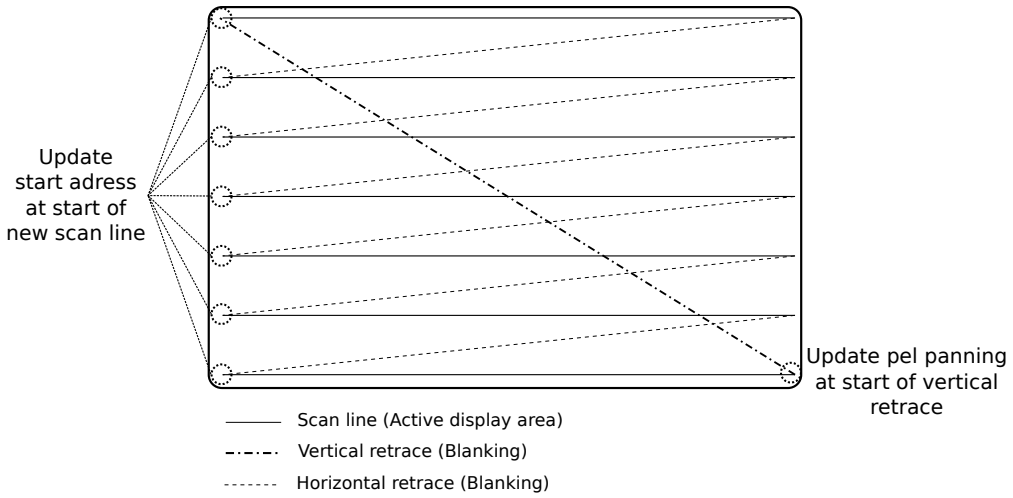


Figure 4.46: Update CRTC start address at beginning of new scan line.

Once the Display Enable status is observed, the program sets the new start address, 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
```

4.10.6 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 actions and reactions rely on the amount ticks between two refreshes, it is important to keep the tick interval consistent between two refreshes. 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;
    }
}
```

4.11 Actors and sprites

4.11.1 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.

4.11.2 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 3.2.2 (Page 138, table 4.1) each sprite asset contains additional information which is illustrated in Figure 4.47.

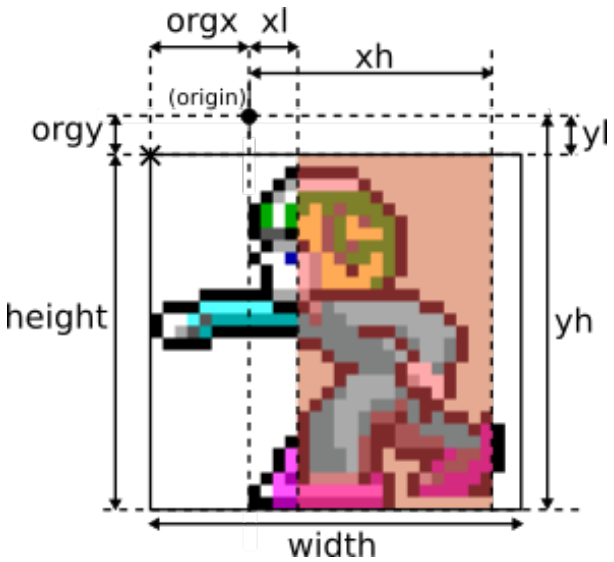


Figure 4.47: sprite structure

All global movement takes place from the origin. The origin `(orgx,orgy)` defines the top-left position of the sprite. Together with the width and height it defines the boundaries of the sprite. The parameters `xl, xh, yl and 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 4.1: 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 4.6.4 on page 98). The value in the shift column defines the amount of steps the sprite has to be shifted within 8 pixels. A value of 4 means the sprite is shifted in 4 steps with a 2 pixel interval.

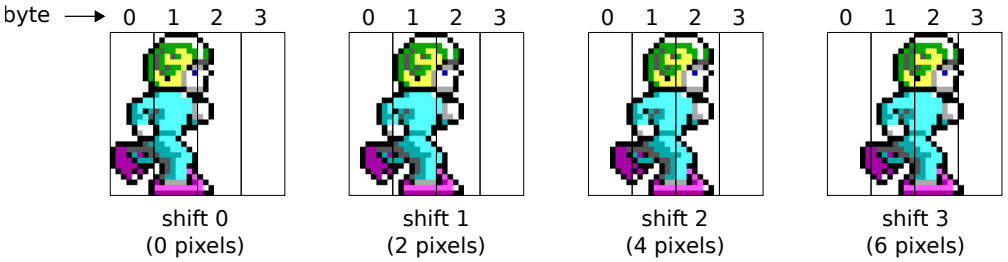


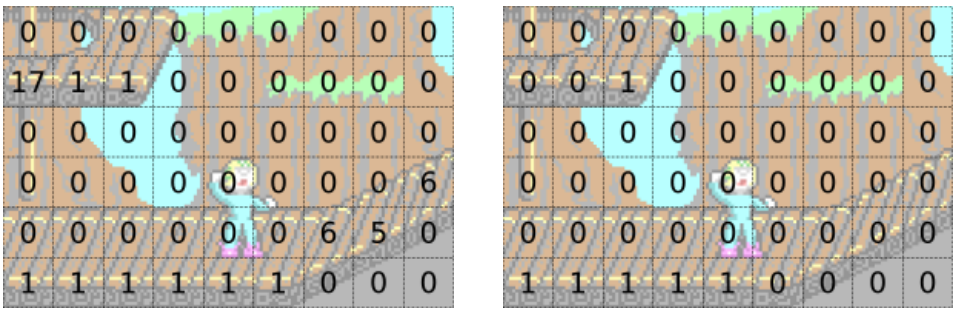
Figure 4.48: Sprite shifted in 4 steps.

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
```

4.11.3 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 3.3.3 on page 68. Each foreground tile contains a `NORTHWALL`, `SOUTHWALL`, `EASTWALL` and `WESTWALL`, as explained in section 3.3.3. 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 4.49: Foreground tile clipping information.

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 4.50. The east/west wall clipping logic is covered by `ClipToEastWalls()` and `ClipToWestWalls()` functions.

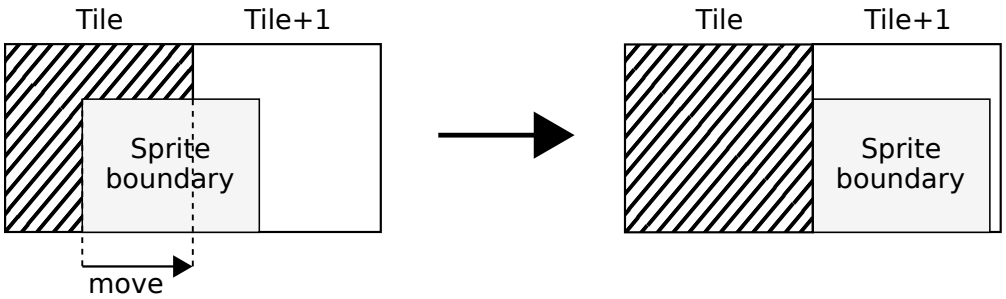


Figure 4.50: Clipping to east wall when moving from the west.

```
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 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. The offset is defined by a lookup table, where the midpoint pixel of the sprite (0-15) and the wall type from the map defines the offset.

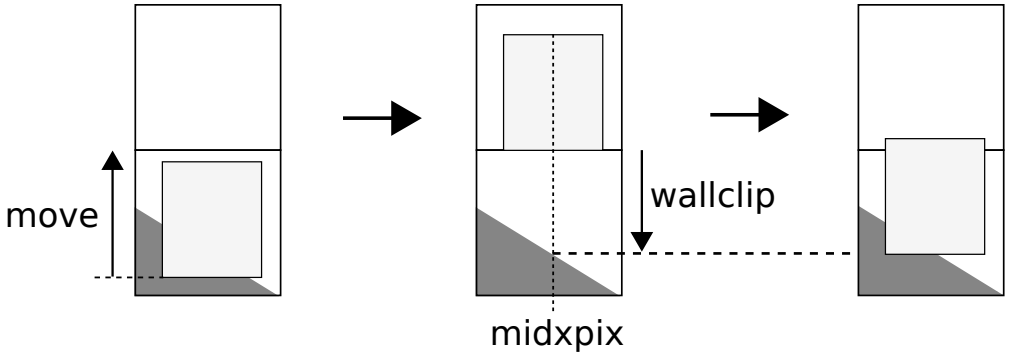


Figure 4.51: Clipping north wall with slope.

```
// 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}
};
```

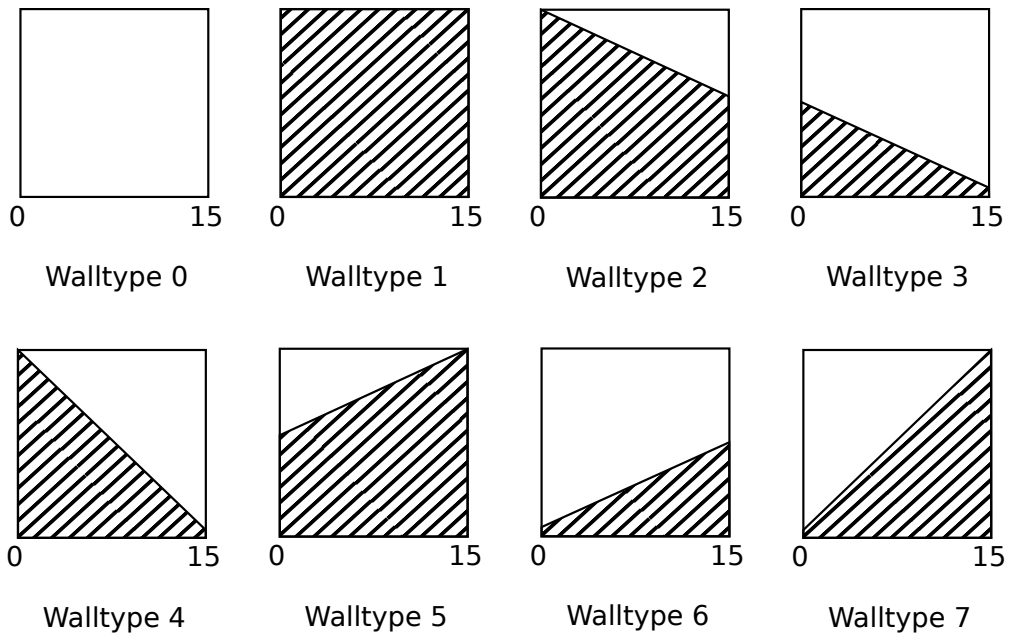


Figure 4.52: Eight different walltypes (slopes) defined.

```
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;
      }
    }
  }
```

4.11.4 Priority of tiles and sprites on screen

The normal screen build is as follows:

1. Draw the background tile.
2. Draw the masked foreground tile.
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 B.1.

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

Figure 4.53: 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 3.3.3 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 4.31 on page 117.
4. Scan the tile buffer array for tiles marked with '3'. If the corresponsing foreground INTILE high bit 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 4.54: 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
```

4.12 Audio and Heartbeat

The audio and heartbeat system runs concurrently with the rest of the program. On an operating system supporting neither multi-processes nor threads this means using interrupts to stop normal execution and perform tasks on the side.

The idea is to configure the hardware to trigger a hardware interrupt at a regular interval. This interrupt is caught by a system called PIC which transforms it into a software interrupt, or IRQ. The software interrupt ID is used as an offset in a vector to look up a function belonging to the engine. At this point, the CPU is stopped (a.k.a: interrupted) from doing whatever it was doing (likely running the 2D renderer), and it starts running the interrupt handler which is called an ISR[6]. We now have two systems running in parallel.

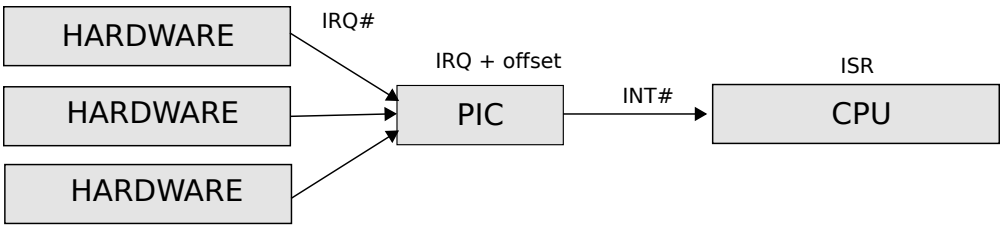


Figure 4.55: Hardware interrupts are translated to software interrupt via the PIC.

Since interrupts keep triggering constantly from various sources, an ISR must choose what should happen if an IRQ is raised while it is still running. There are two options. The ISR can decide it needs a "long" time to run and disable other IRQs via the IMR [7]. This path introduces the problem of discarding important information such as keyboard or mouse inputs.

Alternately, the ISR can decide not to mask other IRQs and do what it is supposed to do as fast as possible so as to not delay the firing of other important interrupts that may lose data if they aren't serviced quickly enough. Keen Dreams uses the latter approach and keeps tasks in its ISR very small and short.

[6] Interrupt Service Routine
[7] Interrupt Mask Register

4.12.1 IRQs and ISRs

The IRQ and ISR system relies on two chips: the Intel 8254 which is a PIT[8] and the Intel 8259 which is a PIC[9]. The PIT features a crystal oscillating in square waves. The PIT contains three channels, each connected with a counter. On each period, it decrements its three counters. Counter #2 is connected to the buzzer and generates sounds. Counter #1 is connected to the RAM in order to automatically perform something called "memory refresh"[10]. Counter #0 is connected to the PIC. When counter #0 hits zero it generates an IRQ[11] and sends it to the PIC.

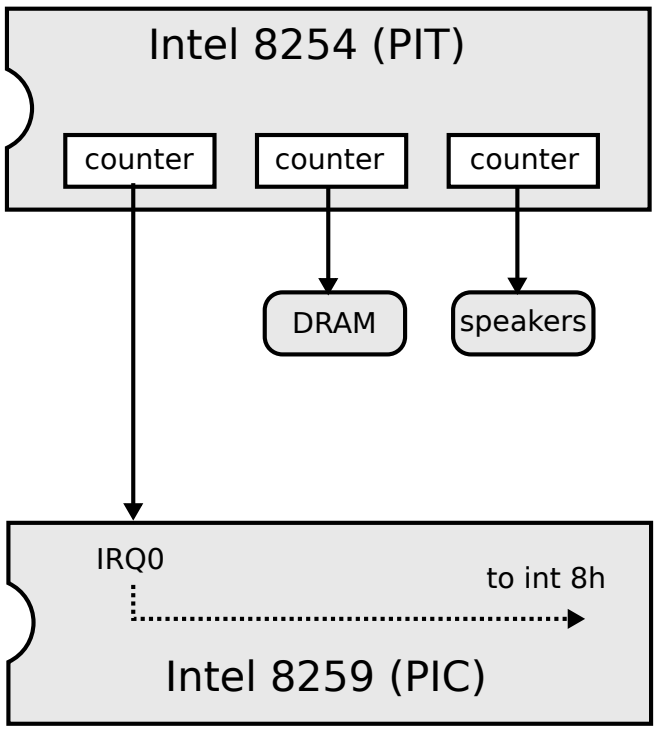


Figure 4.56: Interactions between PIT and PIC.

The PIC's hardware IRQ-0 to IRQ-8 are mapped to the Interrupt Vector starting at Offset 8 (resulting in mapping to software interrupts INT08 to INT0F).

[8] Programmable Interval Timer

[9] Programmable Interrupt Controller

[10] Without frequent refresh, DRAM will lose its content. This is one of the reasons it is slower and SRAM is preferred in the caching system.

[11] Interrupt Request Line: Hardware lines over which devices can send interrupt signals to the CPU.

I.V.T Entry #	Type
00h	CPU divide by zero
01h	Debug single step
02h	Non Maskable Interrupt
03h	Debug breakpoints
04h	Arithmetic overflow
05h	BIOS provided Print Screen routine
06h	Invalid opcode
07h	No math chip
08h	IRQ0, System timer
09h	IRQ1, Keyboard controller
0Ah	IRQ2, Bus cascade services for second 8259
0Bh	IRQ3, Serial port COM2
0Ch	IRQ4, Serial port COM1
0Dh	IRQ5, LPT2, Parallel port (HDD on XT)
0Eh	IRQ6, Floppy Disk Controller
0Fh	IRQ7, LPT1, Parallel port
10h	Video services (VGA)
11h	Equipment check
12h	Memory size determination

Figure 4.57: The Interrupt Vector Table (entries 0 to 18).

Notice #8 which is associated with the System timer and usually updates the operating system clock at 18.2 ticks per second. Because IVT #8 was hijacked, the operating system clock is not updated while Commander Keen runs. Upon exiting the game, DOS will run late by the amount of time played.

Using these two chips and placing its own function at Interrupt Vector Table (IVT) #8, the engine can stop its runtime at a regular interval, effectively implementing a subsystem running concurrently with everything else.

IVT #8 is also responsible for turning off the floppy disk motor after a disk read or write operation. The timer interrupt maintains a disk motor shutoff counter which is decreased every time the timer interupt is called. When the counter reaches 0, the interrupt timer shuts off the disk motor. But since IVT #8 is hijacked, this function is never called and the floppy disk motor keeps running forever. Although this is not an issue, it might give the user the idea that the floppy is still transferring data.

So solve the problem, the timer interrupt subsytem performs a check if any of the disk motors is still running. Checking the status of the disk motors can be done via the BIOS Data Area, which is a section of memory located at segment `0040h` and stores many

variables indicating information about the state of the computer[12]:

- BIOS data address `40h:3Fh` contains the motor status, where bit 0 flags if the disk 1 motor is on and bit 1 if disk 2 motor is on.
- BIOS data address `40h:40h` holds the disk motor shutoff counter, used by the original timer interrupt. The counter is lowered by the timer interrupt vector and once the counter reaches 0, it will turn off the disk motor.

The interrupt subsystem is taking over this functionality and validates if any of the two disk motors is runnning and then decrements the disk motor shutoff counter. In case the shutoff counter is 0 or 1 the original timer interrupt is being called, to shut down the disk motor.

```
// If one of the drives is on,
// and we're not told to leave it on...
if ((peekb(0x40,0x3f) & 3) && !LeaveDriveOn)
{
  if (!(--drivecount))
  {
    drivecount = 5;

    sdcount = peekb(0x40,0x40); // Get system drive count
    if (sdcount < 2)            // Time to turn it off
    {
      // Wait until it's off
      while ((peekb(0x40,0x3f) & 3))
      {
        asm pushf
        t0OldService(); // Call original timer interrupt
      }
    }
    else  // Not time yet, just decrement counter
      pokeb(0x40,0x40,--sdcount);
  }
}
```

4.12.2 PIT and PIC

The PIT chip runs at 1.193182 MHz. This initially seems like an odd choice from the hardware designers, but has a logical origin. In 1980 when the first IBM PC 5150 was designed, the common oscillator used in television circuitry was running at 14.31818 MHz. As it was mass produced, the TV oscillator was very cheap so utilizing it in the PC drove down cost.

[12]For a full overview of BIOS Data Area see https://www.stanislavs.org/helppc/bios_data_area.html.

Engineers built the PC timer around it, dividing the frequency by 3 for the CPU (which is why the Intel ran at 4.7MHz), and dividing by 4 to 3.57MHz for the CGA video card. By logically ANDing these signals together, a frequency equivalent to the base frequency divided by 12 was created. This frequency is 1.1931816666 MHz. By 1990, oscillators were much cheaper and could have used any frequency but backward compatibility prevented this.

4.12.3 Interrupt Frequency

Each counter on the PIT chip is 16-bit, which is decremented after each period. An IRQ is generated and send to the PIC whenever the counter wraps around after 2^{16} = 65,536 decrements. So at default, the interrupts are generated at a frequency of 1.19318MHz / 65,536 = 18.2Hz. Some programs require a faster period than the 18.2 interrupts/second standard rate (for example, execution profilers). So they reprogram the timer by changing the counter value.

```
// Set the number of interrupts generated
// by system timer 0 per second
static void SDL_SetIntsPerSec(word ints)
{
  SDL_SetTimer0(1192755 / ints);
}

// Sets system timer 0 to the specified speed
static void SDL_SetTimer0(word speed)
{
  outportb(0x43,0x36);       // Change PIT counter 0
  outportb(0x40,speed);      // Speed is counter decrements
  outportb(0x40,speed >> 8); // to send interrupt
}
```

Trivia : Note that `SDL_SetTimer0` is using a frequency of 1.192755MHz, instead of the PIT documented 1.193182MHz. Most likely the value is based on 18.2 interrupts per second * 65,536 = 1192755Hz.

So the engine can decide at what frequency to be interrupted, depending on the type of sound/music it needs to play and what devices will be used. As a result, two frequencies are defined:

1. Running at 140Hz to play sound effects and music on the PC beeper, AdLib and SoundBlaster.
2. Running at 700Hz to play sound effects and music on Disney Sound Source.

```
#define TickBase  70

typedef enum {
  sdm_Off,
  sdm_PC,
  sdm_AdLib,
  sdm_SoundBlaster
  sdm_SoundSource
} SDMode;

static  word  t0CountTable[] = {2,2,2,2,10,10};

boolean SD_SetSoundMode(SDMode mode)
{
  word  rate;

  if (result && (mode != SoundMode))
  {
    SDL_ShutDevice();
    SoundMode = mode;
    SDL_StartDevice();
  }

  // Interrupt refresh to either 140Hz or 700Hz
  rate = TickBase * t0CountTable[SoundMode];
  SDL_SetIntsPerSec(rate);
}
```

4.12.4 Heartbeats

Each time the interrupt system triggers, it runs another small (yet paramount) system before taking care of audio requests. The sole goal of this heartbeat system is to maintain a 32-bit variable: `TimeCount`.

```
longword TimeCount;

static void interrupt SDL_t0Service(void)
{
  static word count = 1,

  if (!(--count))
  {
    // Set count to match 70Hz update
    count = t0CountTable[SoundMode];
    TimeCount++;
  }

  outportb(0x20,0x20);  // Acknowledge the interrupt
}
```

It is updated at a rate of 70 units per seconds, to match the VGA update[13] rate of 70Hz. These units are called "ticks". Depending on how fast the audio system runs (from 140Hz to 700Hz), it adjusts how frequent it should increase `TimeCount` to keep the game rate at 70Hz.

Every system in the engine uses this variable to pace itself. The renderer will not start rendering a frame until at least one tick has passed. The AI system expresses action duration in tick units. The input sampler checks for how long a key was pressed, and the list goes on. Everything interacting with human players uses `TimeCount`.

4.12.5 Audio System

The audio system is complex because of the fragmentation of audio devices it can deal with. The early 90's was a time before Windows 95 harnessed all audio cards under the DirectSound common API. Each development studio had to write their own abstraction layer and id Software was no exception. At a high level, the Sound Manager offers a lean API divided in two categories: one for sounds and one for music.

```
void    SD_Startup(void);
void    SD_Shutdown(void);

[...]
```

[13] EGA was updated at a rate of 60Hz. Some games, like Keen Dreams, are developed with VGA already in mind.

```
[...]

void    SD_Default(boolean gotit,SDMode sd,SMMode sm);
void    SD_PlaySound(word sound);
void    SD_StopSound(void);
void    SD_WaitSoundDone(void);

void    SD_StartMusic(Ptr music);
void    SD_FadeOutMusic(void);
boolean SD_MusicPlaying(void);
boolean SD_SetSoundMode(SDMode mode);
boolean SD_SetMusicMode(SMMode mode);
word    SD_SoundPlaying(void);
```

But in the implementation lies a maze of functions directly accessing the I/O port of four sound outputs: AdLib, SoundBlaster, Buzzer, and Disney Sound Source. All belong to one of the three supported families of sound generators: FM Synthesizer (Frequency Modulation), PCM (Pulse Code Modulation) or Square Waves (PC speaker).

Sounds effects are stored in three formats.

1. PC Speaker.
2. AdLib.
3. SoundBlaster/Disney Sound Source.

They are all packaged in the `AudioT` archive created by Muse. Sounds are segregated by format but always stored in the same order. This way a sound can be accessed in three formats by using `STARTPCSOUNDS` + `sound_ID` or `STARTADLIBSOUNDS` + `sound_ID`.

Despite being part of the source code, support for digital effects for the Sound Blaster & Sound Source devices was cut prior to release of Commander Keen Dreams. Therefore I won't explain digital effects (PCM) in this book[14].

[14] For further details on digital sound and PCM, there is an excellent read in the book "Game engine blackbook - Wolfenstein 3D" by Fabien Sanglard.

```
//////////////////////////////////////////////////
//
// MUSE Header for .KDR
// Created Mon Jul 01 18:21:23 1991
//
//////////////////////////////////////////////////


#define NUMSOUNDS            28
#define NUMSNDCHUNKS         84

//
// Sound names & indexes
//
#define KEENWALK1SND         0
#define KEENWALK2SND         1
#define JUMPSND              2
#define LANDSND              3
#define THROWSND             4
#define DIVESND              5
#define GETPOWERSND          6
#define GETPOINTSSND         7
#define GETBOMBSND           8
#define FLOWERPOWERSND       9
#define UNFLOWERPOWERSND     10
[...]
#define OPENDOORSND          19
#define THROWBOMBSND         20
#define BOMBBOOMSND          21
#define BOOBUSGONESND        22
#define GETKEYSND            23
#define GRAPESCREAMSND       24
#define PLUMMETSND           25
#define CLICKSND         26
#define TICKSND          27

//
// Base offsets
//
#define STARTPCSOUNDS    0
#define STARTADLIBSOUNDS  28
#define STARTDIGISOUNDS   56
#define STARTMUSIC        84
```

4.12.5.1 FM Synthesizer: OPL2/YM3812 Programming

Programming the OPL2 output is esoteric to say the least. AdLib and Creative did publish SDKs but they were expensive. Documentation was sparse and often cryptic. Today, they are very difficult to find.

The OPL2 is made of 9 channels capable of emulating instruments. Each channel is made of two oscillators: a Modulator whose outputs are fed into a Carrier's input. Each channel has individual settings including frequency and envelope (composed of attack rate, decay rate, sustain level, release rate, and vibrato). Each oscillator can also pick a waveform (these characteristic forms are what gave the YM3812 its recognizable sound).

To control all of these channels, a developer must configure the OPL2's 244 internal registers. These are all accessed via two external I/O ports. One port is for selecting the card's internal register and the other is to read/write data to it.

```
0x388 - Address/Status port (R/W)
0x389 - Data port (W/O)
```

When the Adlib was first conceived in 1986, it was tested on IBM XTs and ATs, none of which exceeded a speed of 6 MHz. They wrote their specification based on this, writing that while the Adlib required a certain amount of "wait time" between commands, it was okay to send them as fast as possible because no PC was faster than the minimum wait time. They later found out that a Intel 386 was fast enough to send commands faster than the Adlib was expecting them, and they changed their specification to mention a minimum 35 microseconds wait time between commands.

The Programming Guide was amended with reliable specs to wait 3.3 microseconds after a register select write, and 23 microseconds after a data write. Within the source code it is implemented as a 10 microseconds and 25 microseconds respectively.

```
//////////////////////////////////////////////////
//
//  alOut(n,b) - Puts b in AdLib card register n
//
//////////////////////////////////////////////////
void
alOut(byte n,byte b)
{
  asm pushf
  asm cli

  asm mov   dx,0x388
  asm mov   al,[n]
  asm out   dx,al
  SDL_Delay(TimerDelay10);    //wait 10ms

  asm mov   dx,0x389
  asm mov   al,[b]
  asm out   dx,al

  asm popf

  SDL_Delay(TimerDelay25);    //wait 25ms
}
```

Every time the audio system wakes up via the timer interrupt, it checks if a sound effects should be sent, and plays the next sample out through the AdLib card.

4.12.6 PC Speaker: Square Waves

The hardware chapter described a problem for sound effects: the default PC speaker could only generate square waves, resulting in long beeps which are not acceptable for gaming.

The solution was to approximate a tune by placing the PC Speaker in repeat mode and make it change frequency every 1/140th of a second. It is simpler to understand when the signal is a simple sinusoid:

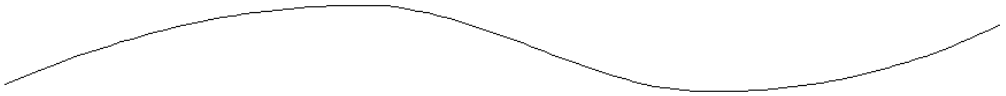


Figure 4.58: The original sound.

Figure 4.59: The same sound approximated with square wave and frequency changes.

To do this, the audio system once again relies on the PIT chipset. Channel 0 is used to trigger the audio system. Channel 1 is used to refresh the RAM periodically. Channel 2, however, is directly connected to the PC Speaker.

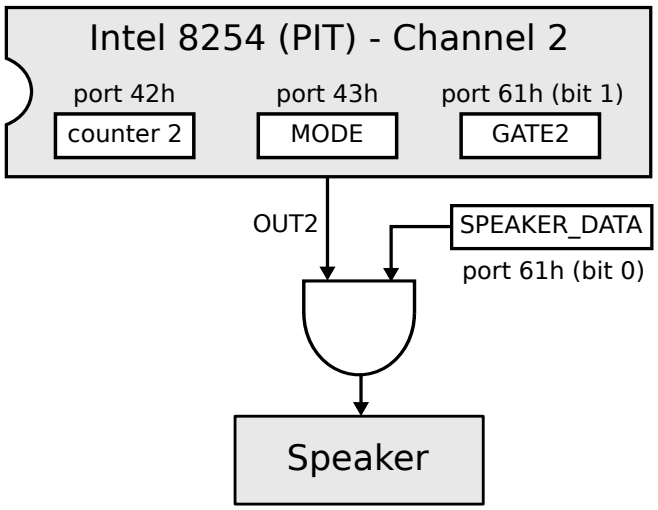


Figure 4.60: Built-in speaker hardware diagram.

OUT2 is the output of Channel 2 of the PIT, GATE2 is the enable/trigger control for the Channel 2 counter, and SPEAKER_DATA to control the speaker volume. The trick is to set OUT2 to square wave mode so it will repeat after it triggers and program the desired square wave frequency. This can be done by setting MODE in the PIT Command register to Mode 3.

Mode	Type
0	Interrupt on Terminal Count
1	Hardware Re-triggerable One-shot
2	Rate Generator
3	Square Wave Generator
4	Software Triggered Strobe
5	Hardware Triggered Strobe

Figure 4.61: Available modes of a PIT counter.

When instructed to play a PC Speaker sound effect, the audio system sets itself to run at 140Hz via PIT Counter 0. Every times it wakes up, it reads the frequency to maintain for the next 1/140th of a second and writes it to Counter 2. The frequencies to use are encoded as a stream of bytes, the value of which is decoded as follows:

```
frequency = 1193181 / (value * 60)
```

While the end result was not great, it was better than a beep.

```
static void SDL_PCService(void)
{
  byte  s;
  word  t;

  [...]

  s = *pcSound++;

  asm pushf
  asm cli

  if (s)              // We have a frequency!
  {
    t = pcSoundLookup[s];
    asm mov bx,[t]

    asm mov al,0xb6   // Write to channel 2 (speaker) timer
    asm out 43h,al
    asm mov al,bl
    asm out 42h,al    // Low byte
    asm mov al,bh
    asm out 42h,al    // High byte

    asm in  al,0x61   // Turn the speaker & gate on
    asm or  al,3
    asm out 0x61,al
  }
  else                // Time for some silence
  {
    asm in  al,0x61   // Turn the speaker & gate off
    asm and al,0xfc   // ~3
    asm out 0x61,al
  }

  asm popf

}
```

Notice how the `* 60` is not calculated but looked up. Once again the engine tries to save as much CPU time as possible by using a bit of RAM. The frequency is read from a lookup table `pcSoundLookup`.

```
word        pcSoundLookup[255];
```

Notice how 0xb6 (10110110) is sent to the PIC Command register:

- 10 = Target Counter 2.
- 11 = High & low byte of counter updated.
- 011 = (MODE) Square Wave Generator.
- 0 = 16-bit mode.

4.13 User Inputs

In an era before Microsoft harnessed all inputs under DirectInput API with Windows 95, developers had to write drivers for each input type they wanted to support. This involved talking directly to the hardware in the vendor's protocol on a physical port. The keyboard is plugged into a PS/2 or AT port, the mouse to a serial port (DE9), and the joystick to a game port (DA-15).

4.13.1 Keyboard

As the keyboard is the standard and oldest input medium, it is fairly easy to access. When a key is pressed, the interrupt is routed to an ISR in the Vector Interrupt Table. The engine installs its own ISR there.

```
#define KeyInt    9 // The keyboard ISR number

static void INL_StartKbd(void) {

  IN_ClearKeysDown();

  OldKeyVect = getvect(KeyInt);
  setvect(KeyInt,INL_KeyService);

  INL_KeyHook = 0;  // Clear key hook
}

static void interrupt INL_KeyService(void) {
  byte  k;
  k = inportb(0x60);  // Get the scan code

  // Tell the XT keyboard controller to clear the key
  outportb(0x61,(temp = inportb(0x61)) | 0x80);
  outportb(0x61,temp);

  [...] // Process scan code.
  Keyboard[k] = XXX;

  outportb(0x20,0x20); // ACK interrupt to interrupt system
    .
}
```

The state of the keyboard is maintained in a global array `Keyboard`, available for the entire engine to lookup.

```
#define NumCodes  128
boolean   Keyboard[NumCodes];
```

4.13.2 Mouse

A driver has to be loaded at startup for the mouse to be accessible. DOS did not come with one. It was usually on a vendor provided floppy disk. `MOUSE.COM` (or `MOUSE.SYS`) had to be added to `config.sys` so it would reside in RAM. It was usually stored in `DOS` folder.

```
C:\DOS\MOUSE.COM
```

The driver takes almost 5KiB of RAM. With the driver loaded all interactions happen with

software interrupt `0x33`. The interface works with requests issued in register AX[15] and responses issued in registers CX, BX and DX. With Borland compiler syntactic sugar it is easy to write with almost no boilerplate (notice direct access to registers thanks to _AX and co special keywords).

```
#define MouseInt  0x33
#define Mouse(x)  _AX = x,geninterrupt(MouseInt)

static void INL_GetMouseDelta(int *x,int *y) {
  Mouse(MDelta);
  *x = _CX;
  *y = _DX;
}
```

Request	Type	Response
AX=0	Get Status	AX = FFFFh : available. AX Value = 0 : not available
AX=1	Show Pointer	
AX=2	Hide Pointer	
AX=3	Mouse Position	CX = X Coordinate, DX = Y Coordinate
AX=3	Mouse Buttons	BX = 1 Left Pressed, BX = 2 Right Pressed, BX = 3 Center Button Pressed
AX=7	Set Horizontal Limit	CX=MaxX1 DX=MaxX2
AX=8	Set Vertical Limit	CX=MaxY1 DX=MaxY2
AX=11	Read Mouse Motion Counters	CX = horizontal mickey count[16], DX = vertical mickey count

Figure 4.62: Mouse request/response.

4.13.3 Joystick

All interactions with the joystick happen over I/O port `0x201`. Two joysticks can be chained together and the state of both of them fits in a byte.

[15] For a full overview of all mouse interrupt function, see https://www.stanislavs.org/helppc/int_33.html.
[16] values are 1/200 inch intervals (1 mickey = 1/200 in.).

```
word INL_GetJoyButtons(word joy){
  register  word  result;

  result = inportb(0x201); // Get all the joystick buttons
  result >>= joy? 6 : 4;   // Shift into bits 0-1
  result &= 3;             // Mask off the useless bits
  result ^= 3;
  return(result);
}
```

Bit Number	Meaning
0	Joystick A, X Axis
1	Joystick A, Y Axis
2	Joystick B, X Axis
3	Joystick B, Y Axis
4	Joystick A, Button 1
5	Joystick A, Button 2
6	Joystick B, Button 1
7	Joystick B, Button 2

Figure 4.63: Joystick sampling bits and their meaning.

The API looks clean at first, with each button associated with a bit indicating whether it is pressed or not. But if you take a closer look you will notice there is only one bit of information per axis, which is not enough to encode the position of a stick. This bit is actually a flag allowing an analog input to be converted into a digital value. To better understand, let's dive into details.

On the joystick side, each axis is connected to a 100kΩ potentiometer. An applied 5V voltage generates a variable current based on the stick position (from Ohm's law where $I = \frac{V}{R}$).

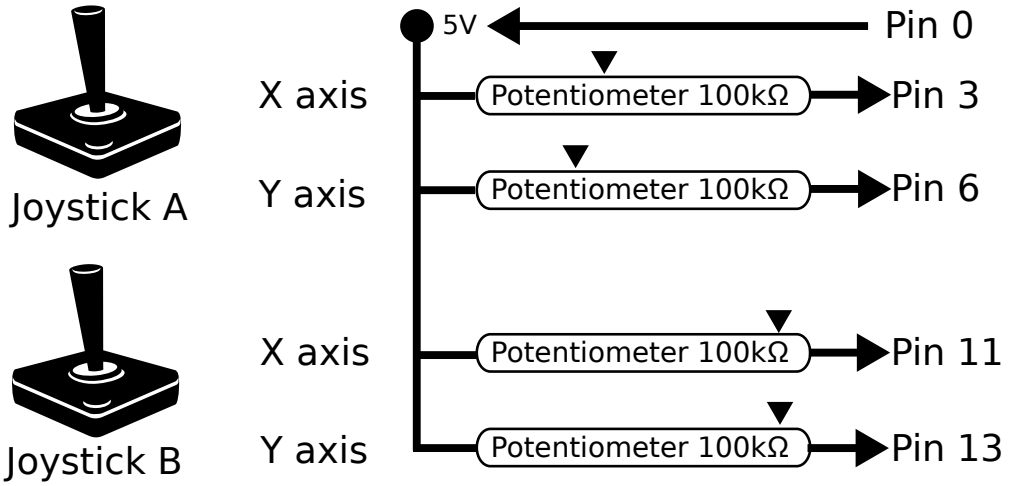


Figure 4.64: Two joysticks and the four potentiometers connected to the game port pins.

On the joystick side each pin carrying the current is connected to monostable multivibrators (which is a complicated name for a capacitor able to output 1 when it is charged and 0 when it is charging). The idea is to infer the position of the stick by measuring how long the vibrator takes to charge (a strong current will charge the capacitor faster than a weak current).

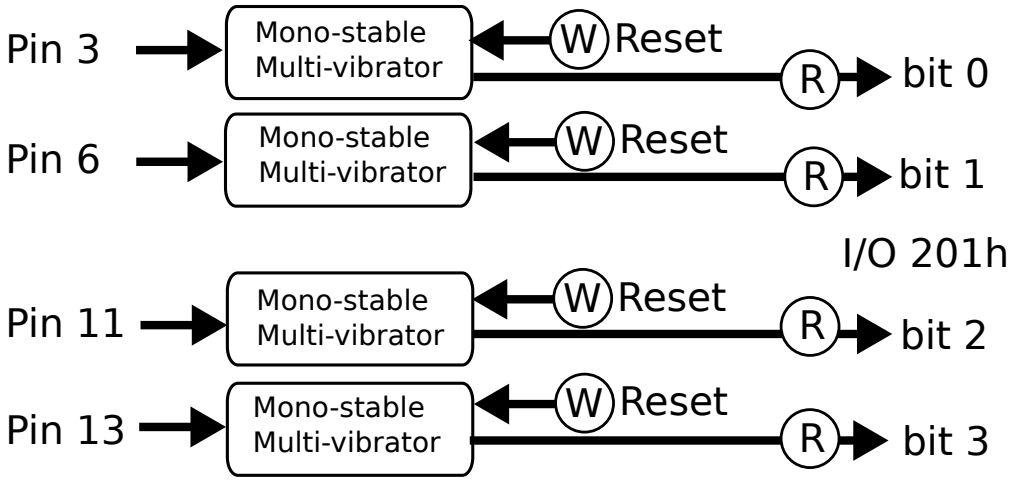


Figure 4.65: Each potentiometer is connected to a capacitor able to output either 0 or 1 depending on its charging state.

On the CPU side, retrieving the stick position is a three-step process:

1. Write (W) any value to I/O port 201h. This will discharge all capacitors.
2. Initialize a counter to zero and read (R) from 201h. At first all bits 0-4 will be equal to zero.
3. Loop forever (or until counter == 0xFFFF as a safety measure) increase counter on each iteration. Save the counter value for each bit when it is flipped to 1.

On a 286 CPU the counter value can range from 7 to 900 depending on the stick/capacitor position. On a 386 CPU, which will run loops faster, these values would be higher. Hence the values measured can only be translated to a stick position if they are compared to a min and a max.

This explains why joysticks have to be calibrated. For the flight simulators of the 90s where accurate position was needed, the player would be asked to put the joystick in upper-left position (to set the potentiometers on both axis to minimum resistance) and press a button to read the "loop count". The player would then repeat the operation at the lower right position so that the system would know the min and max "loop count" for this joystick/CPU combination.[17]

Figure 4.66: Strike Commander startup screen makes you calibrate your joystick.

There is no calibration process in Commander Keen because when the engine starts up it samples the loop count and assumes the joystick is in neutral position. When the game runs and joystick position is needed, the engine samples loop count and compares the count to what was measured with neutral. It is not enough to calculate the exact stick position on each axis but it is enough to determine up/down and left/right using >, == (with epsilon) and < comparison operations.

[17] The Mark-1 FCS by Thrustmaster and Flightstick Pro by CH were the best flight controllers of the 90s. They used all bits for one controller, offering a device with four buttons with the extra two axes serving as a four-way view hat.

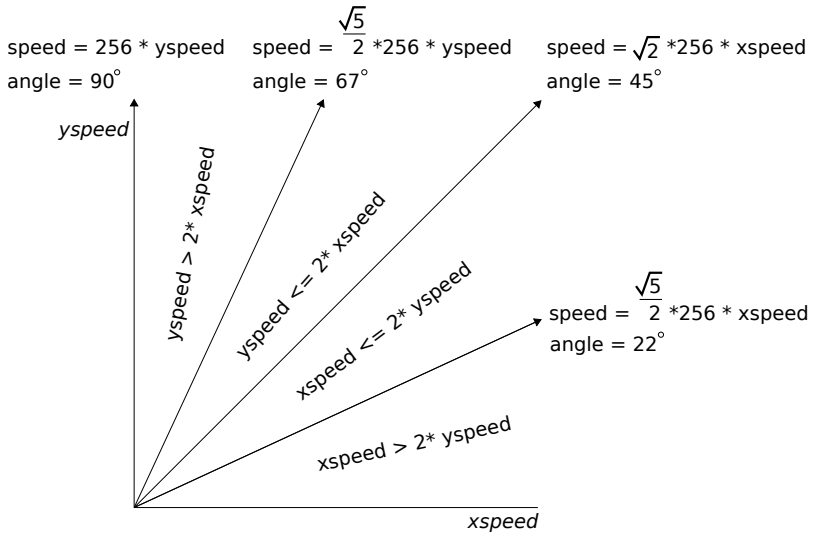
4.14 Tricks

This section describes random tricks used to speed up rendering.

4.14.1 Bouncing Flower

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° or 90°. Based on the ratio between the x- and y-speed it calculates the resulting speed and corresponding angle. Notice that for higher precision the speed is multiplied with 256.



For each combination of the eight type of slopes (Figure 4.52) 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, waltype 3 with incoming angle of 22°, results in bounce calculation case 5.

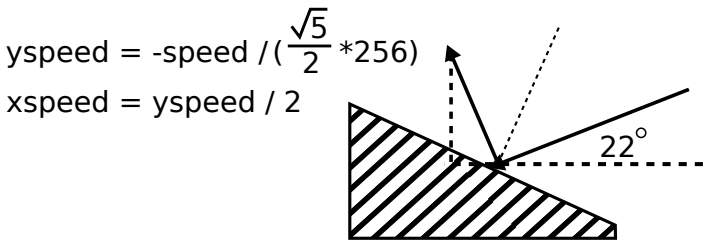
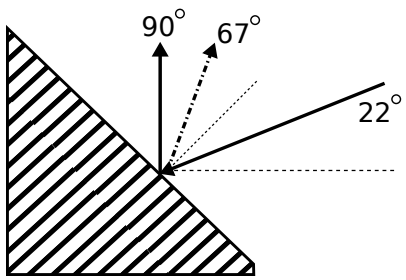


Figure 4.67: Walltype 3 with incoming angle of 22° (angle=0).

```
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 *256
  }
  else
  [...]              // Check for 45, 67 and 90 degrees
}

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°.



4.14.2 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][18] 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.

[18] 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
```

4.14.3 Screen fades

When a new level is loaded, the screen fades from black to the level.CHECK WHEN SCREEN FADES TO WHITE. Here it makes use of reassigning the color palette colors. 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 RGB color to display for that index
geninterrupt (0x10) ; Generate Video BIOS interrupt
```

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.

Earlier in the hardware chapter, Section 2.3.7, it was explained that most EGA monitors did not support the extended 64-color palette and uses 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	CGA modes
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 4.68: 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 "RGB" and setting the Secondary Green ("r") for the intensity color version. The "b" has no meaning and the "r" (Ground) is normally set to 0.

The screen fading is defined with the `colors[7][17]` scheme, where the first 16 columns refer to the color indexes, the last column is the border color. Note that the "b" bit is set for the intensity colors, but this had no effect on the results since the pin is unassigned for CGA.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	0
0	0	0	0	0	0	0	0	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
0	1	2	3	4	5	6	7	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
0	1	2	3	4	5	6	7	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0
0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f

Figure 4.69: Color fading table.

Fading the screen 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;
}
```

Chapter 5

Keen Dreams in CGA

The original Commander Keen, Commander Keen in Invasion of the Vorticons, was only released for the EGA videocard. Keen Dreams and later versions included a CGA version as well. The game play was exactly the same, sounds were the same, it was just that the graphics were CGA. Before diving into the source code, let's first get a better understanding of the CGA video hardware.

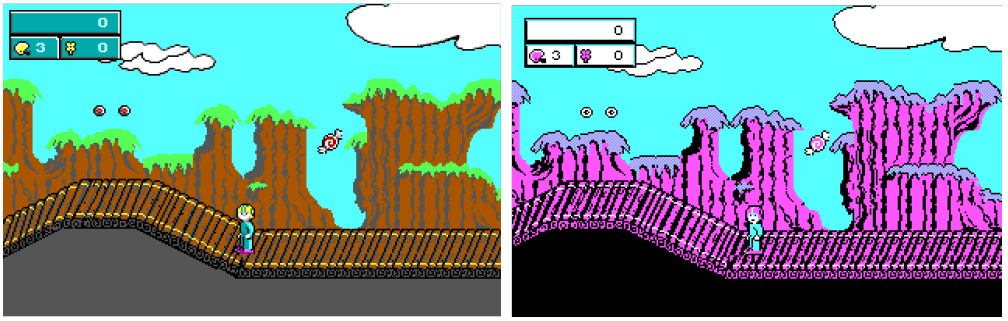


Figure 5.1: Keen Dreams EGA and CGA version.

Trivia : It's an ironic twist that Softdisk did not use the original Keen's engine, as the code violated the company policy by depending on 16-color EGA hardware without supporting older 4-color CGA cards!

5.1 CGA Videocard

The Color Graphics Adapter (CGA), originally also called the Color/Graphics Adapter or IBM Color/Graphics Monitor Adapter, introduced in 1981, was IBM's first color graphics card for the IBM XT.

The CGA card can be summarized by the following hardware:

- It was built around the Motorola 6845 display controller.
- The framebuffer (the VRAM) contained two memory banks of 8 kilobytes each, resulting in 16 kilobytes total.
- Character generator ROM, containing a 14-row font and two 8x8 fonts. This is the same ROM as used on the MDA videocard.

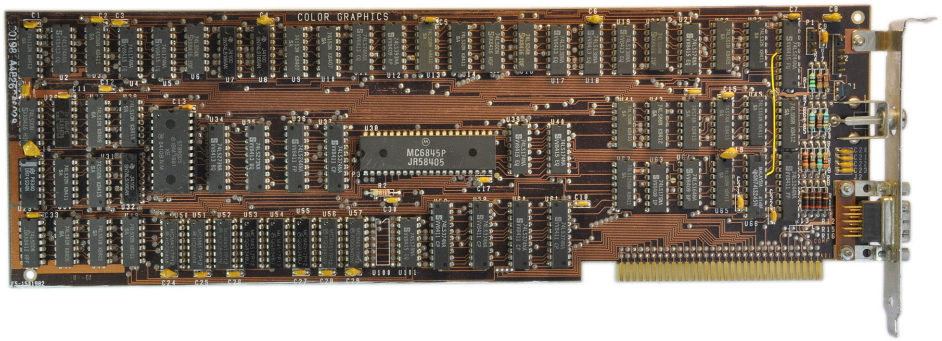


Figure 5.2: The CGA is a full-length 8-bit ISA card.

The CGA card has the following text and graphics modes:

Mode	Type	Format	Colors	RAM Mapping	Hz
0	text	40x25	16 (monochrome)	B8000h	60
1	text	40x25	16	B8000h	60
2	text	80x25	16 (monochrome)	B8000h	60
3	text	80x25	16	B8000h	60
4	CGA Graphics	320x200	4	B8000h	60
5	CGA Graphics	320x200	4 (monochrome)	B8000h	60
6	CGA Graphics	640x200	2	B8000h	60

Figure 5.3: EGA Modes available.

In the graphics mode 4, which is used by Commander Keen, each pixel is using 2 bits for color, resulting in only four colors being displayed at a time. These four colors could not be freely chosen from the 16 CGA colors, there were only two official palettes for this mode:

1. Magenta, cyan, white and background colour (black by default).
2. Red, green, brown/yellow and background colour (black by default).

The background color could be any of the 16 colors, but often it was kept black. For each mode there is a high- and low-intensity version of the palette.

Palette 1		Palette 2	
low intensity	**high intensity**	**low intensity**	**high intensity**
0 - Background	0 - Background	0 - Background	0 - Background
2 - Green	10 - Bright Green	3 - Cyan	11 - Bright Cyan
4 - Red	12 - Bright Red	5 - Magenta	13 - Bright Magenta
6 - Brown	14 - Yellow	7 - Bright Grey	15 - White

Figure 5.4: CGA color palettes.

The default palette when switching to Mode `04h` is palette 2 with high intensity, which is used by Commander Keen.

5.2 Memory architecture and Interlacing

The CGA memory layout in graphics mode is different compared to EGA, as it is based on interlaced architecture. Normally, video memory is strictly linear: the next row of display data corresponds to the next row of pixels. But with CGA, the next row of display data corresponded to the row of pixels two rows down. This continued until the end of the screen and only with the second half of display data were the in-between rows addressed. So VRAM bank 0 was for even rows 0, 2, 4, etc., until the end of the screen and VRAM bank 1 was for odd rows 1, 3, 5, etc. This added calculation steps to most CGA graphics operations if the programmer wanted to avoid visual artifacts when updating the screen.

Each pair of 2 bits is one pixel with a color value of 0-3, referring to the CGA color palette. The 2 most left bits represent pixel 0, the next 2 bits pixel 1, etc. So each byte in VRAM represents 4 pixels on screen.

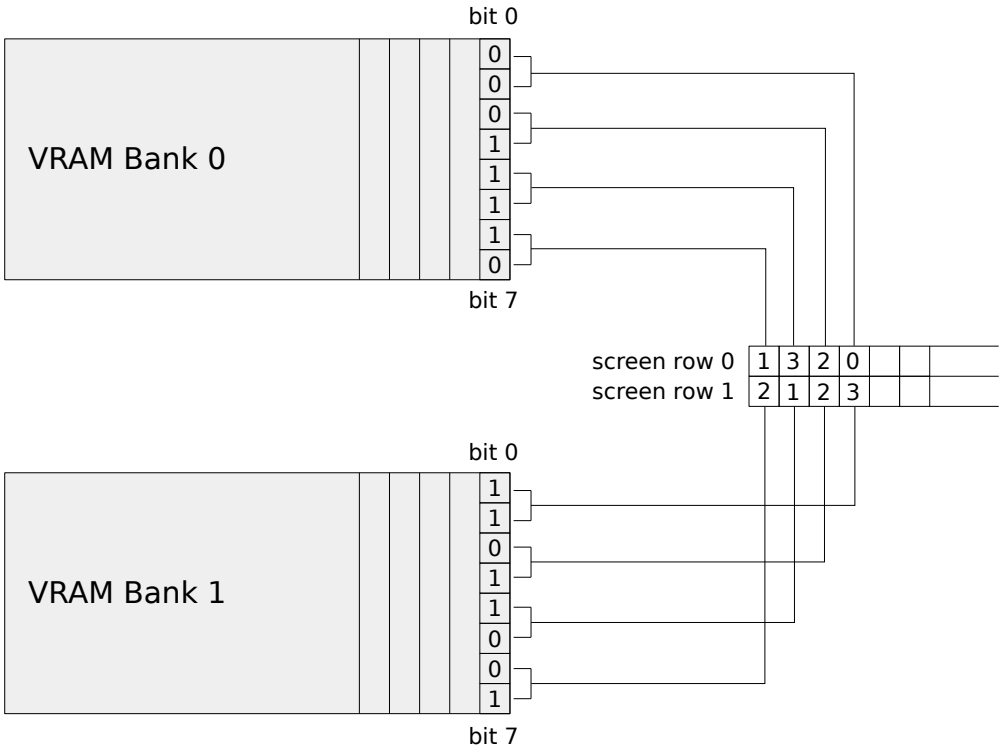


Figure 5.5: CGA interlaced memory.

The CGA card is making use of memory mapping, just like EGA. In mode 4, the VRAM

bank 0 is mapped from 0xB0000 to 0xB1FFF and VRAM bank 1 is mapped from 0xB2000 to 0xB3FFF. Unlike EGA, the CGA memory model doesn't require masking as the total 16KiB VRAM fits easily in a 64KiB memory segment.

Trivia : Interesting enough, interlacing is never really implemented in CGA. When displaying the VRAM to screen it does a progressive (lineair) scan, where it alternately reads from bank 0 and bank 1.

5.3 Double buffering

A full picture in mode 4 requires 320 pixels * 2 bits per pixel * 200 lines = 16,000 bytes of memory. This means the display screen requires all 16KiB memory and there is no capacity in VRAM for extra screens. The only way to introduce double buffering on CGA is by creating a 64 KiB buffer in conventional memory.

```
#if GRMODE == CGAGR
    grmode = CGAGR;

    // grab 64k for floating screen
    MM_GetPtr (&(memptr)screenseg,0x10000l);
#endif
```

The memory buffer contains both the buffer page and the static master page. The buffer page starts at offset `0x0000h` and the master page start at `0x8000h`. Both pages float around in the 64KiB memory segment, making use of the same memory wrapping as explained in section 4.11.3.

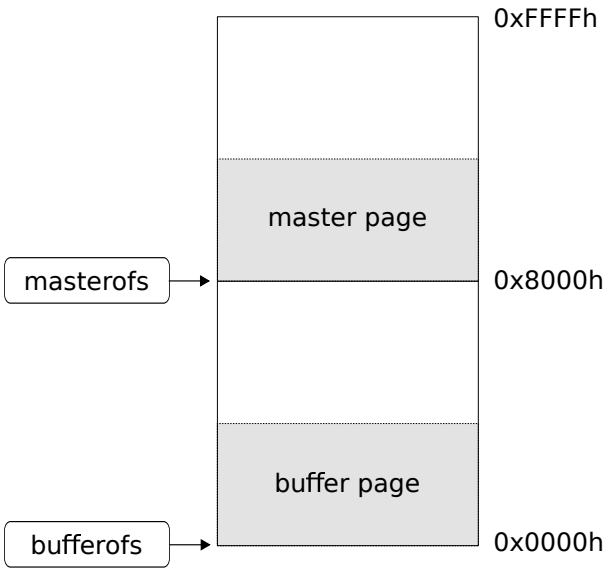
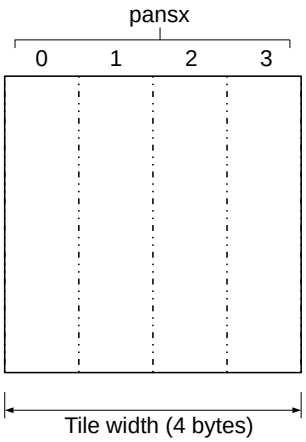


Figure 5.6: CGA double buffering memory layout.

5.4 Screen refresh

With the double buffering in place, the same algoritm as implemented for EGA can be used. The final step of the algoritm is updating the screen display by copying the buffer page to the VRAM. However, there are two complications with CGA.

The first complication is that the CGA card does not support pixel panning. So the smoothest pixel scroll is equal to scroll the screen with one byte. Since one byte represents 4 pixels, it means scrolling to left or right is in steps of 4 pixels.




```
void RFL_CalcOriginStuff (long x, long y)
{
    [...]

    originxglobal = x;
    originyglobal = y;

    panx = (originxglobal>>G_P_SHIFT) & 15;
    pansx = panx & 12;  //pansx is 0, 4, 8 or 12 pixels
    pany = pansy = (originyglobal>>G_P_SHIFT) & 15;
    panadjust = pansx/4 + ylookup[pansy];
}
```

The second complication involves copying the RAM buffer to the interlaced VRAM. This requires to split the lineair memory buffer into copying all even rows to VRAM bank 0 and odd rows to VRAM bank 1.

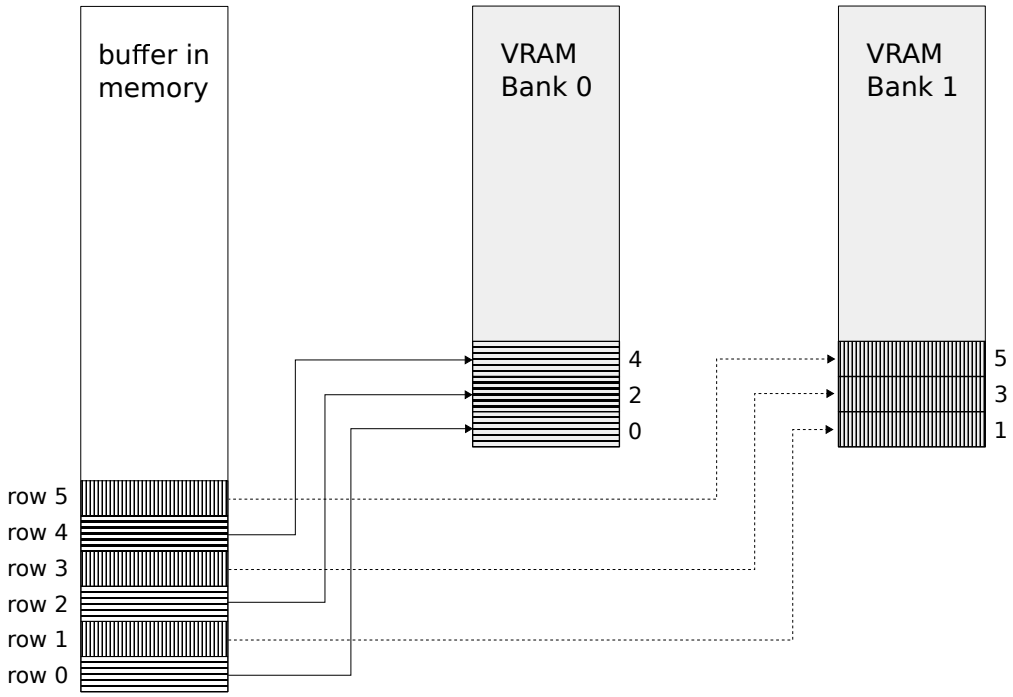


Figure 5.7: CGA memory to VRAM copy.

To avoid screen tearing the system should wait for a vertical retrace, like it was done with EGA. The problem is that computers weren't fast enough to copy all bytes from RAM buffer to VRAM during the vertical retrace period. So in the CGA version of Commander Keen, it was not possible to avoid screen tearing.

```
void VW_CGAFullUpdate (void)
{
    displayofs = bufferofs+panadjust;

    asm mov ax,0xb800
    asm mov es,ax

    asm mov si,[displayofs]
    asm xor di,di

    asm mov bx,100          // pairs of scan lines to copy
    asm mov dx,[linewidth]
    asm sub dx,80

    asm mov ds,[screenseg]  // buffer segment in memory
    asm test  si,1
    asm jz  evenblock

    [...]

    evenblock:
    asm mov ax,40           // words accross screen
    copytwolines:
    asm mov cx,ax
    asm rep movsw           // copy row to VRAM bank 0
    asm add si,dx
    asm add di,0x2000-80    // go to the interlaced bank 1
    asm mov cx,ax
    asm rep movsw           // copy row to VRAM bank 1
    asm add si,dx
    asm sub di,0x2000   // go to the non interlaced bank 0

    asm dec bx
    asm jnz copytwolines

[...]

}
```

The original IBM CGA card could not handle writing and reading VRAM at the same time. Because video memory on the IBM CGA isn't dual-ported, when the CPU and the video card need access to the same byte of video RAM, the CPU wins; the card ends up reading a random value, causing "snow" on the screen. The only way to avoid the snow was,

also in this case, to wait for the vertical retrace. Most CGA clones resolved the issue by enabling read/write VRAM at the same time, but if one would play Commander Keen on the original IBM CGA card you experience both screen tearing and snow on the screen.

Appendices

Appendix A

Dangerous Dave in Copyright Infringement

In September 1990, John Carmack, developed his first version of *Adaptive Tile Refreshment*. He discussed the idea with coworker Tom Hall, who encouraged him to demonstrate it by recreating the first level of the recent Super Mario Bros. 3 on a computer. The pair did so in a single overnight session, with Hall recreating the graphics of the game. They replaced the player character of Mario with Dangerous Dave, a character from an eponymous previous Gamer's Edge game, while Carmack optimized the code. The next morning on September 20, the resulting game, Dangerous Dave in Copyright Infringement, was shown to their other coworker John Romero.

> "As soon as the demo started running, I pressed the right arrow key to see if magic had indeed been made. As soon as little Dave walked a short way to the right...
>
> THE SCREEN SCROLLED.
>
> SMOOTHLY.
>
> Time stopped.
>
> I was speechless...
>
> **John Romero - founder of id Software.**"

Romero recognized Carmack's idea as a major accomplishment: Nintendo was one of the

most successful companies in Japan, largely due to the success of their Mario franchise, and the ability to replicate the gameplay of the series on a computer could have large implications.

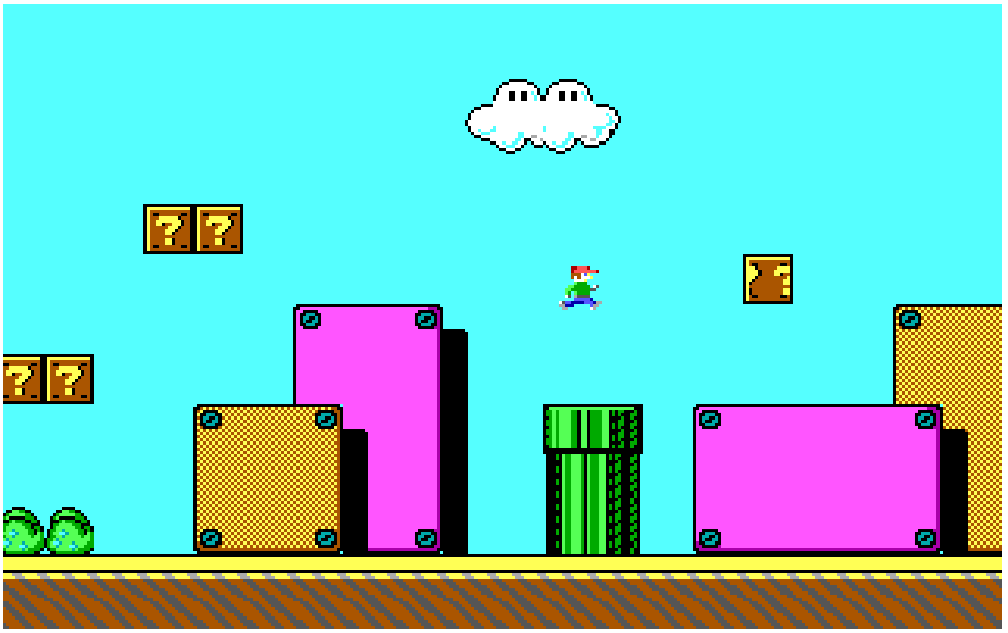


Figure A.1: Dangerous Dave in Copyright Infringement demo

The manager of the team and fellow programmer, Jay Wilbur, recommended that they take the demo to Nintendo itself, to position themselves as capable of building a PC version of Super Mario Bros. for the company. The team (composed of Carmack, Romero, Hall, and Wilbur, along with Lane Roathe, the editor for Gamer's Edge) decided to build a full demo game for their idea to send to Nintendo. As they lacked the computers to build the project at home, and could not work on it at Softdisk, they "borrowed" their work computers over the weekend, taking them in their cars to a house shared by Carmack, Wilbur, and Roathe, and made a copy of the first level of the game over the next 72 hours. The team send the demo to Nintendo Of America to see if they could do the PC port of the game.

The demo made it to Nintendo of Japan and Shigeru Miyamoto specifically. They were very impressed with the demo, but their corporate plan was to never release their IP on a platform other than their own.

Appendix B

Founding of id Software

Around the same time as the group was rejected by Nintendo, Romero was approached by Scott Miller of Apogee Software. They agreed to make *Commander Keen in Invasion of the Vorticons*, to be published by Apogee Software. The team could not afford to leave their jobs to work on the game full-time, so they continued to work at Softdisk, spending their time on the Gamer's Edge games during the day and on Commander Keen at night and weekends using Softdisk computers. The game was completed in early December 1990.

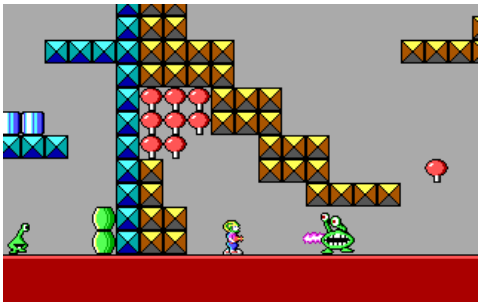
After the arrival of the first royalty check from Apogee, the team planned to quit Softdisk and start their own company. On February 1, 1991, the team founded *id Software* having four owners: John Carmack, John Romero, Tom Hall and artist Adrian Carmack[1].

> "I told them we need to start a company, do our own game and publish it, outside of Softdisk. Jay Wilbur happened by the office and I told him that after what had been done by John and Tom the night before, we were outta there. He kinda laughed and said, "Heheh, yeah..." and I said, "No. I'm serious - we're gone." Jay quickly closed the door and wanted to know what we were thinking of doing.
>
> **John Romero - founder of id Software.**

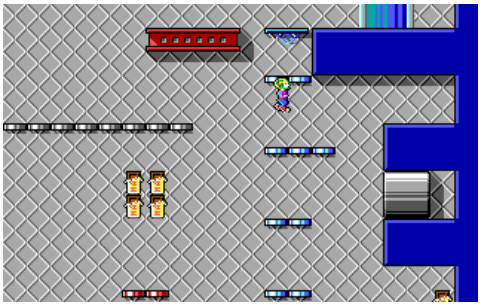
When their boss and owner of Softdisk, Al Vekovius, confronted them on their plans, as well as their use of company resources to develop the game, the team made no secret of their intentions. Vekovius initially proposed a joint venture between the team and Softdisk, which fell apart when the other employees of the firm threatened to quit in response, and after a few weeks of negotiation the team agreed to produce a series of games for Gamer's

[1] See Masters of Doom, chapter 4

Edge, one every two months. One of the games they developed to fulfill their obligation was Commander Keen in Keen Dreams.



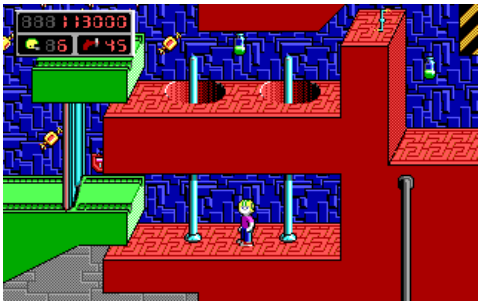
Keen 1 - Marooned on Mars.



Keen 2 - The Earth Explodes.

Keen 3 - The Earth Explodes.

Keen 4 - Secret of the Oracle.



Keen 5 - The Armageddon Machine.

Keen 6 - Aliens Ate My Babysitter.

Figure B.1: Commander Keen Episode 1-6.

Between 1990 and 1991 the team published *Commander Keen in Invasion of the Vorticons* and *Commander Keen in Goodbye, Galaxy*, and the stand-alone games *Commander Keen*

in Keen Dreams and *Commander Keen in Aliens Ate My Babysitter*. Another trilogy of episodes, titled *The Universe Is Toast*, was planned for December 1992; id worked on it for a couple of weeks, but then shifted the work to another game. The name of that new game was **Wolfenstein 3D**...