GAME ENGINE BLACK BOOK

COMMANDER KEEN

v2022.06.06 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 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 deep interest into programming and learned about sprites and side-scrolling techniques.

The same year I had my first computer experience, 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. Earlier side scrolling games, 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

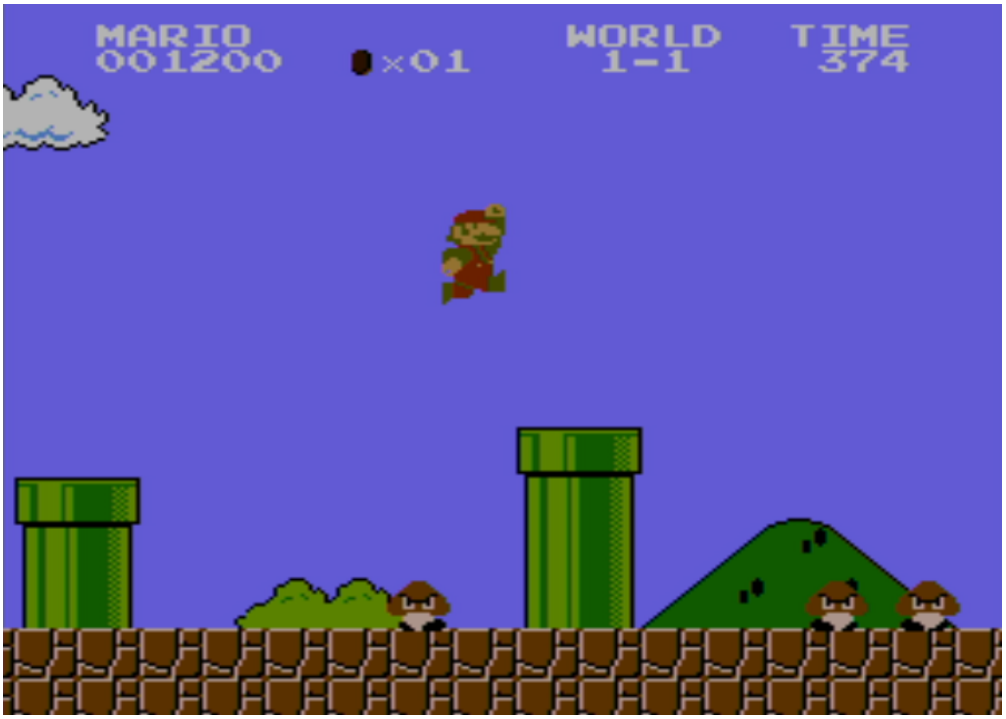


Figure 1.1: Super Mario Bros. on Nintendo Entertainment System

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 the only way is 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 is be redrawing the entire screen offset by the number of pixels you want to scroll, which is 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 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.

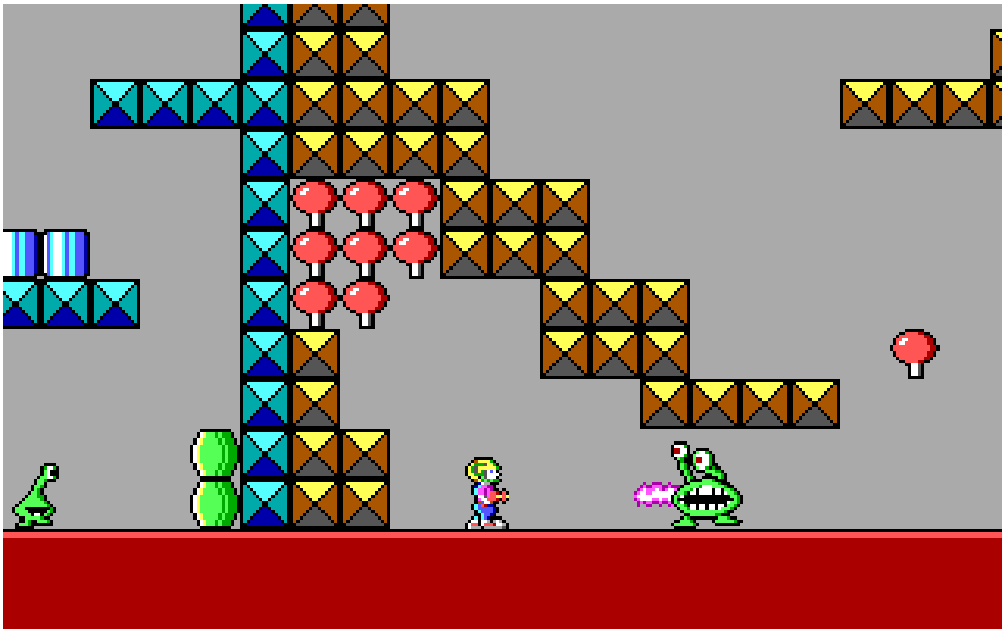


Figure 1.2: Commander Keen in Invasion of the Vorticons

How was this possible on the IBM PC? Yes, the 286 CPU of the PC outperformed any home computer on the market in terms of raw power.

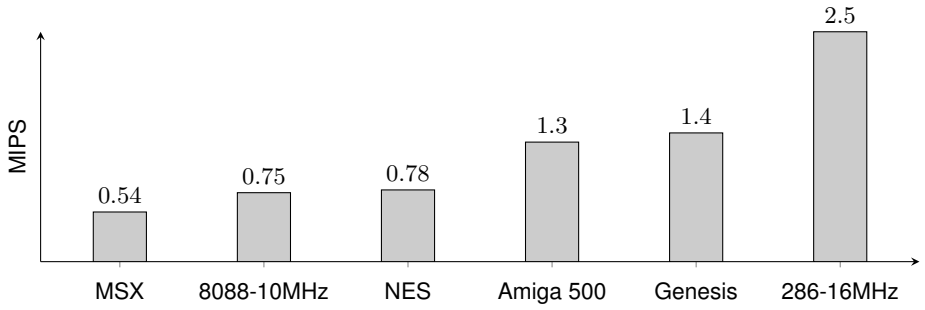


Figure 1.3: Consoles[2]vs PC, CPU comparison with MIPS[3][4].

But that was about everything the IBM PC excelled in. Many obstacles had to be overcome:

- 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

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 on a disk.

[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.

- 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.

The book is written around "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 public 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.

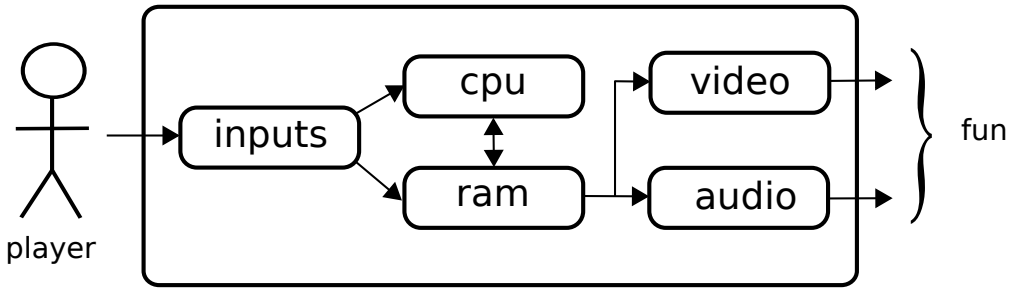


Figure 2.1: Hardware pipeline.

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	Impossible

Figure 2.2: Component quality for a game engine.

2.1 CPU: Central Processing Unit

In 1991 there were 54 million PCs in the USA[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 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] "Computers". Collier's Encyclopedia. Vol. 7, 1992: 114, 129.

[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.

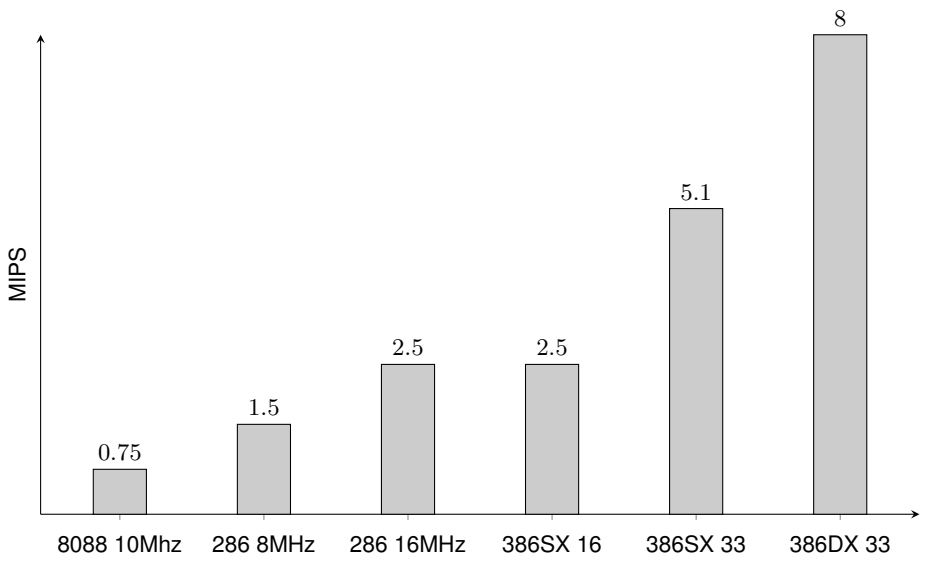


Figure 2.3: Comparison[3]of CPUs with MIPS

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.

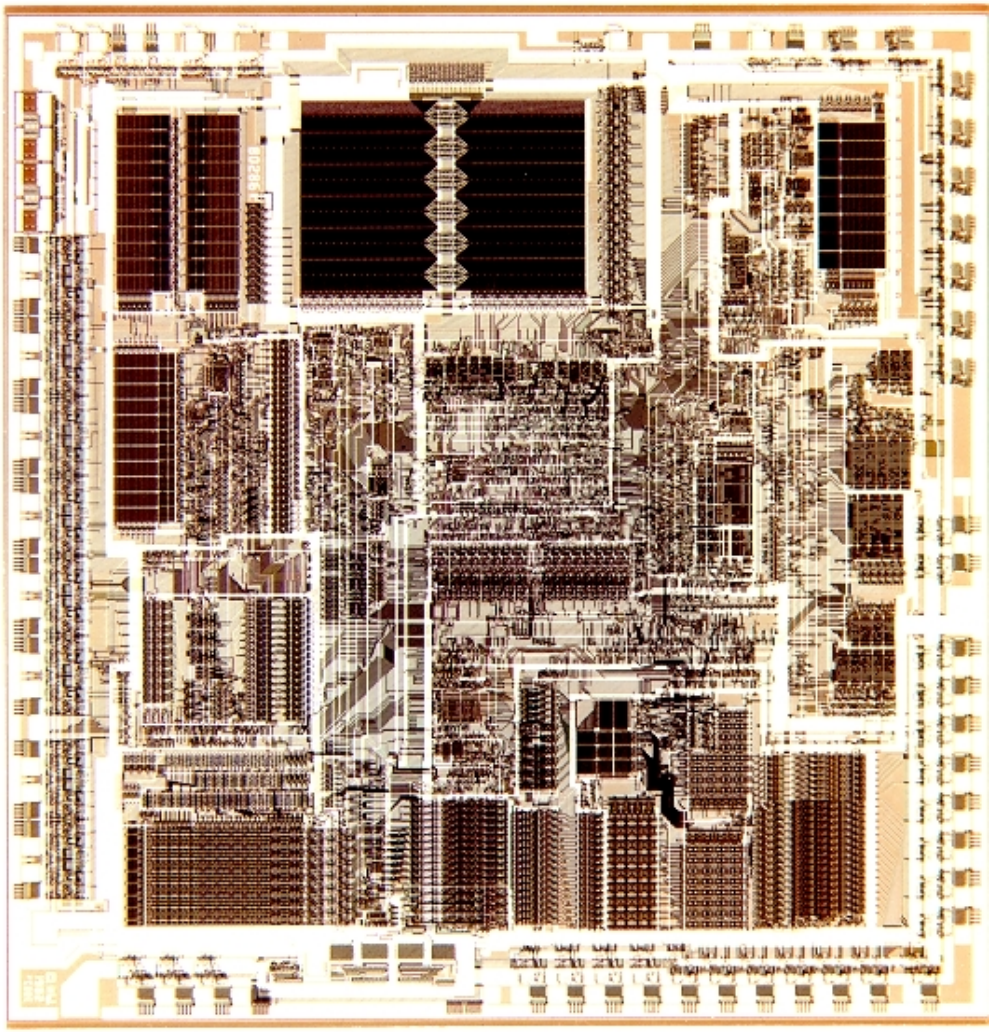
When IBM developed the AT, it selected the 286 as the basis for the new system because the chip provided compatibility with the 8088 used in the PC and the XT. Therefore, software written for those chips should run on the 286. The 286 chip is many times faster than the 8088 used in the XT, and at the 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.

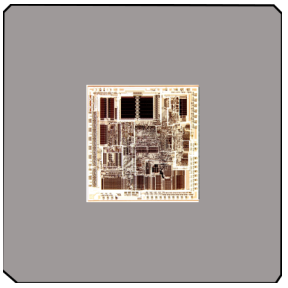
cycles on the 8086 or 8088, but takes an average of only 4.5 cycles on the 286 processor. Additionally, the 286 chip can handle up to 16 bits of data at a time through an external data bus twice the size of the 8088.

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 believes that it 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 instead of 29,000.




A80286-8
L9409047
©1982,1985



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

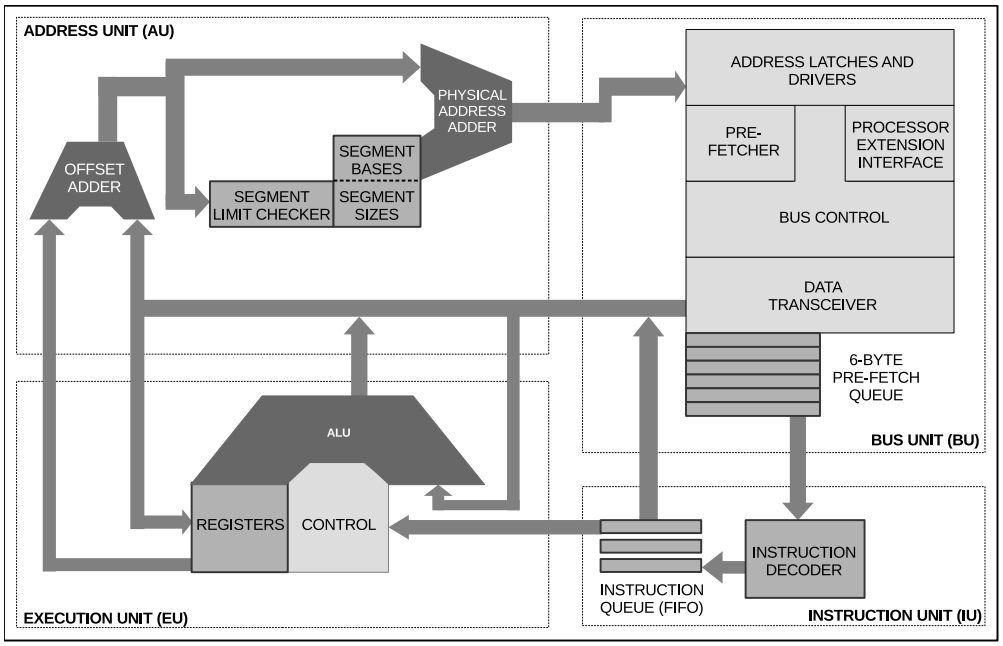
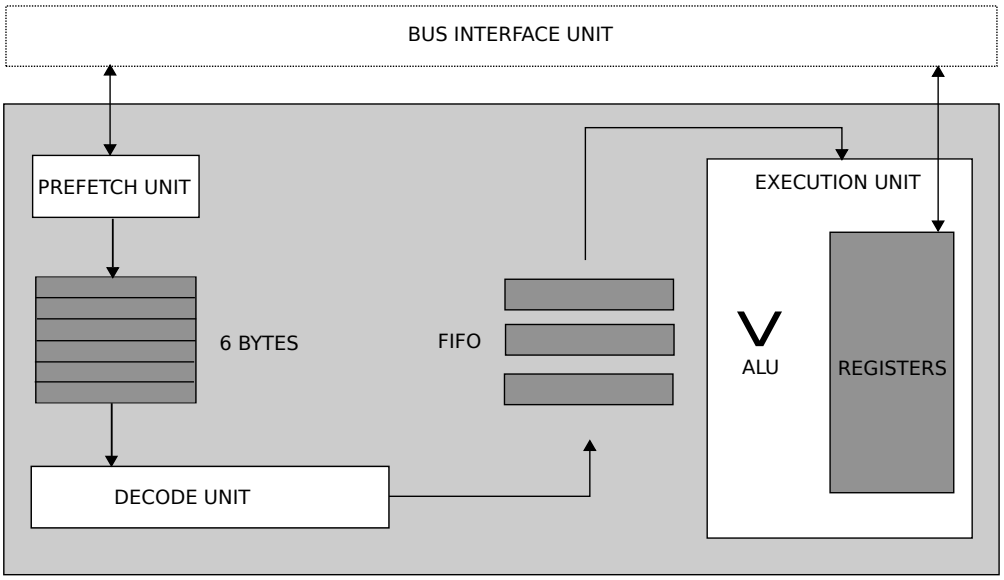


Figure 2.4: Internal block diagram of the 80286 processor

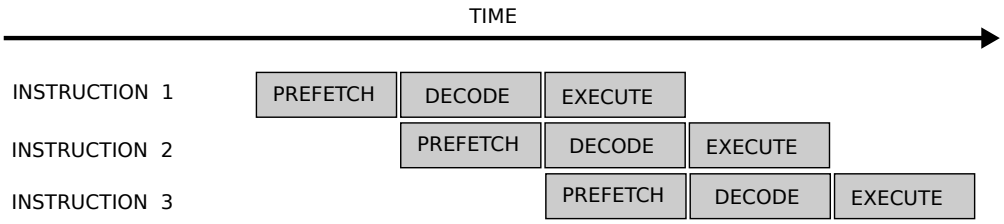
The four functional units can be described by

- **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.



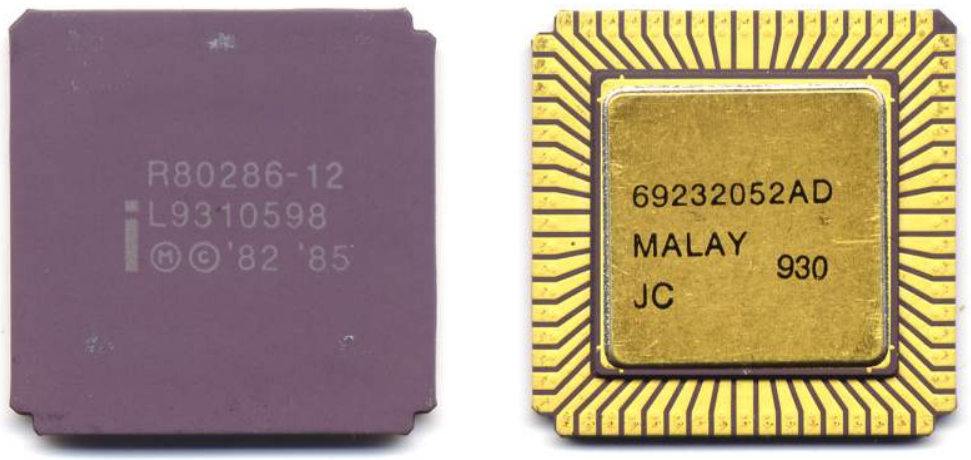


Figure 2.5: The Intel 286, 10mm by 10mm packing 134,000 transistors

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 4.01[9] and even the later release DOS 5.0[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'.

The memory layout is as follows:

- From 00000h to 003FFh : the Interrupt Vector Table.

[8] "Tales of Application Compatibility", Old New Thing by Raymond Chen.
[9] Released in July 1989.
[10] Released in June 1991

- 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.

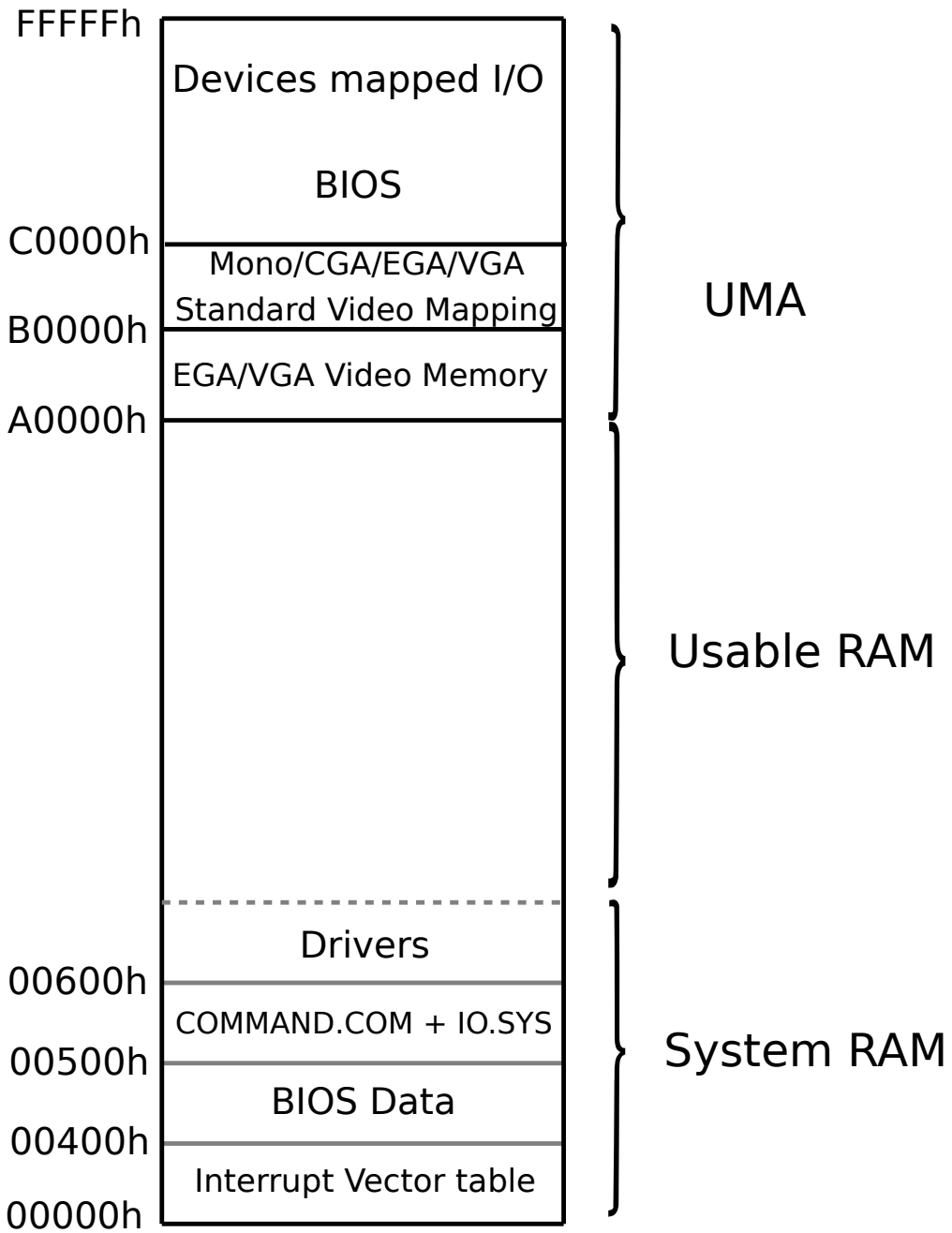


Figure 2.7: First 1MiB of RAM layout.

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.

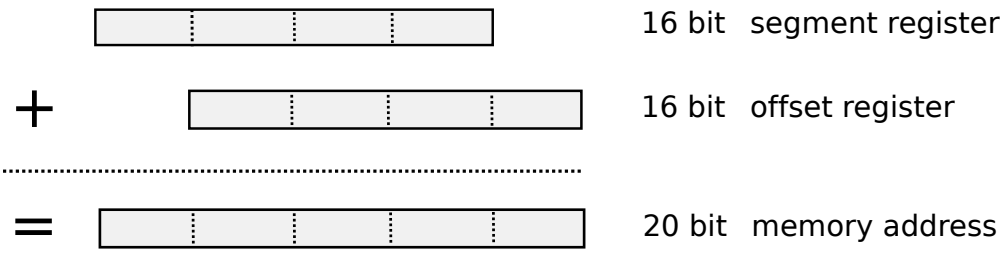


Figure 2.8: How registers are combined to address memory.

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.

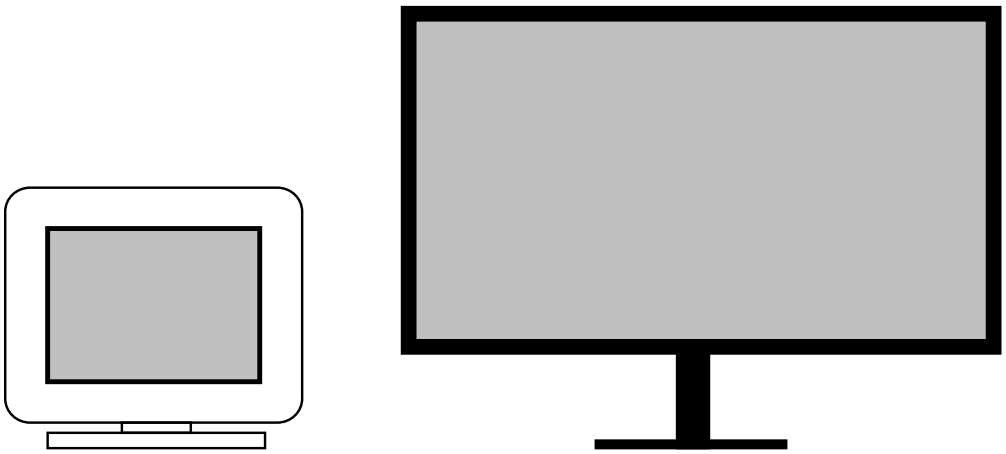


Figure 2.9: CRT (left) vs LCD (right)

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 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.

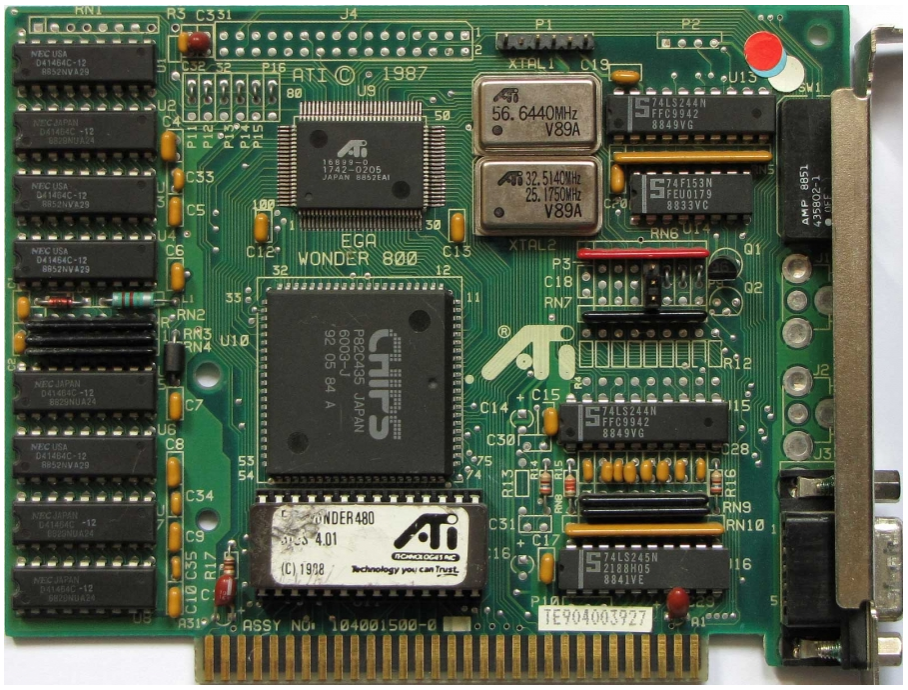
Name	Year Released	Memory	Max Resolution
`MDA` (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[11]
`Hercules`	1982	64KiB	720x348
`CGA` (**C**olor **G**raphics **A**dapter)	1981	16KiB	640x200
`EGA` (**E**nhanced **G**raphics **A**dapter)	1985	64KiB	640x350
`VGA` (**V**ideo **G**raphics **A**rray)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

Figure 2.10: 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].



2.3.2 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 plane, but almost all other EGA cards were equiped with the full 64KiB per plane. 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.

[12] Each VRAM chip from this ATI EGA cards can store 32KiB, accounting for a total of 256KiB VRAM.
[13] Transistor Transistor Logic
[14] Digital to Analog Converter
[15] Computer Graphics: Principles and Practice 2nd Edition, page 168.

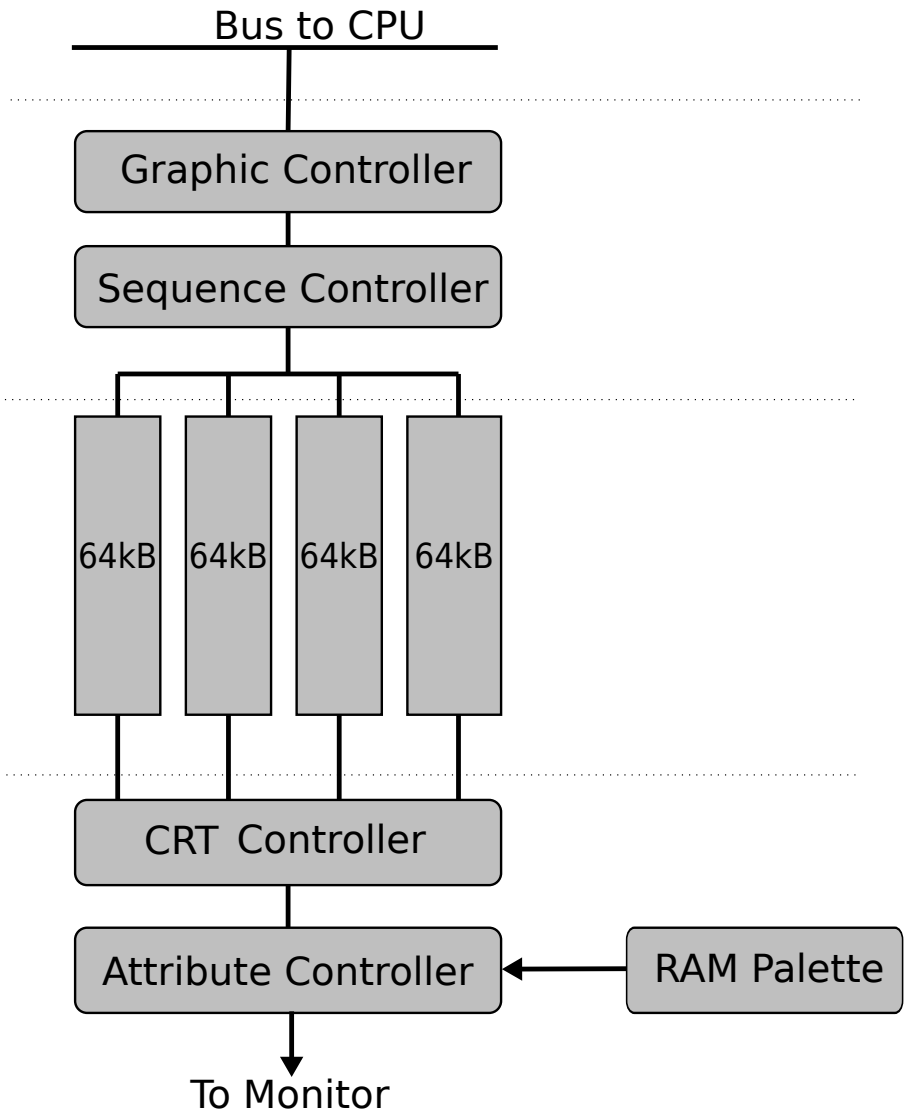


Figure 2.11: EGA Architecture.

2.3.3 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: Red, Green, Blue and Intensity (RGBI). For example, if a bit is set in the blue plane as well as the red plane, that pixel will appear purple on-screen. Each of these banks is mapped to the same UMA memory address. This layout is better explained with a drawing.

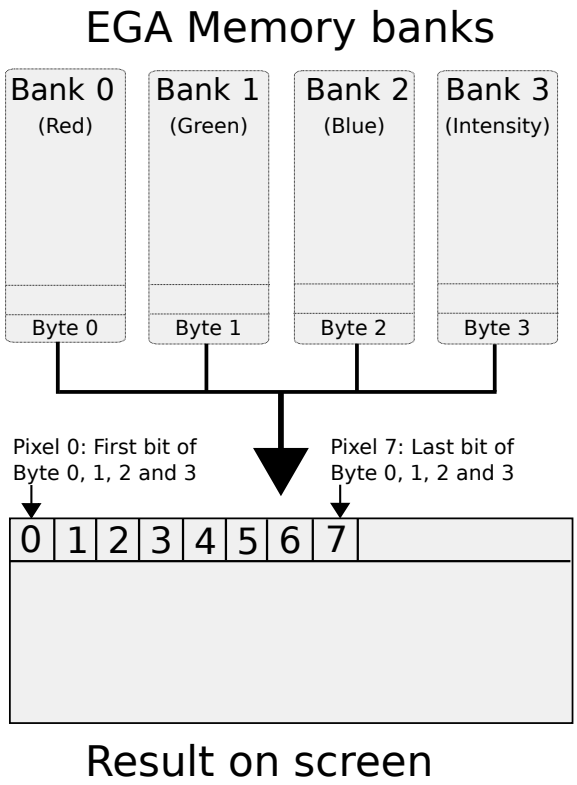


Figure 2.12: EGA mode 0Dh, How bank layout appears on screen.

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 2.11, which described the architecture, was actually deceptively simplified. Figure 2.13 shows how IBM's reference documentation explained the EGA. The maze of wire

showcases well the actual complexity of the system.

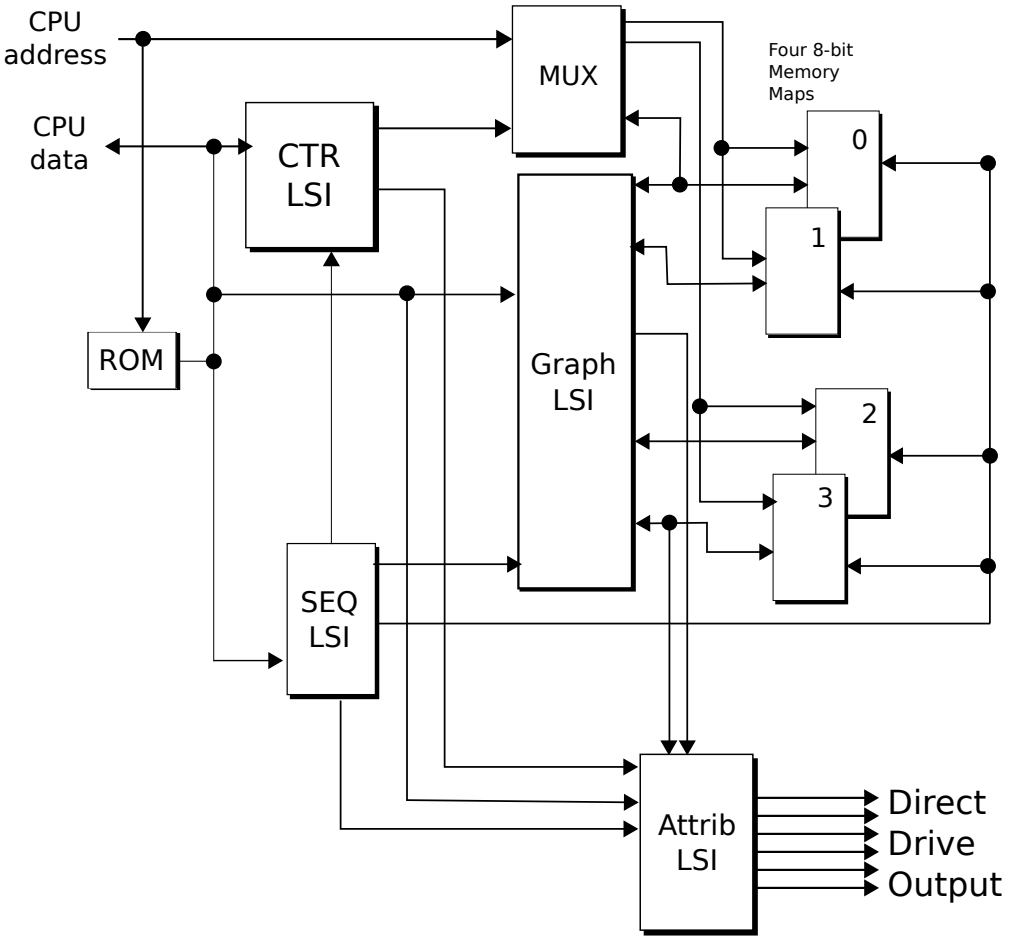


Figure 2.13: IBM's EGA Documentation.

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 11 available with an associated resolution, number of colors, and memory layout.

2.3.4 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 2.14: EGA Modes available.

2.3.5 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.

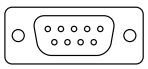


Figure 2.15: EGA Port

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 `10h`, using the 320x200 resolution with 16 colors.

2.3.6 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 (two bits each for red, green and blue).

When selecting a color from the EGA palette, two bits are used for the red, green and blue channels. This allows each channel a value of 0, 1, 2 or 3. To select the color magenta, the red and blue values would be medium intensity (2, or 10 in binary) and the green value would be off (0). 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 and an uppercase letter is the most significant bit. For magenta, the most significant bit in the red and blue values is a 1, so the uppercase R and B placeholders would become 1. All other digits are zeros, giving the binary number 000101 for the color magenta. This is 5 in decimal, so setting a palette entry to 5 would result in it being set to magenta. All the color values for the default colors are listed in the table below.

However, standard EGA monitors do 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 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
0	Black	000000	0
1	Blue	000001	1
2	Green	000010	2
3	Cyan	000011	3
4	Red	000100	4
5	Magenta	000101	5
6	Brown	010100	20
7	Ligh grey	000111	7
8	Dark grey	111000	56
9	Bright blue	111001	57
10	Bright green	111010	58
11	Bright cyan	111011	59
12	Bright red	111100	60
13	Bright magenta	111101	61
14	Bright yellow	111101	62
15	White	111101	63

Figure 2.16: Default EGA 16-color palette

2.3.7 EGA Programming: Memory Mapping

To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in table 2.14. 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 2.17. 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.

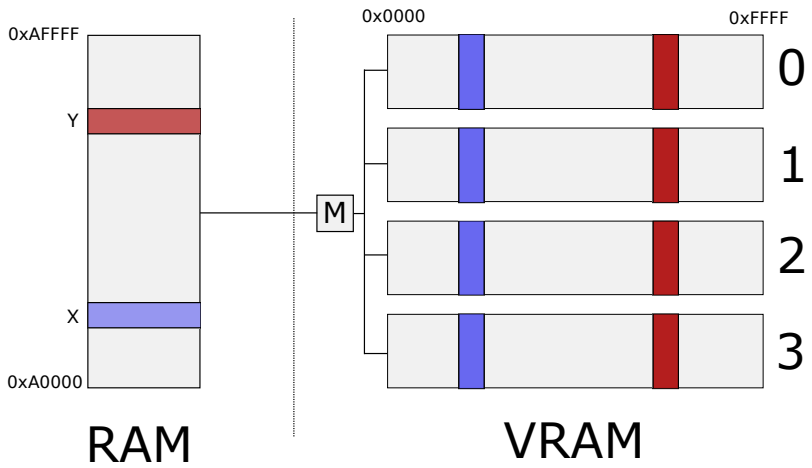


Figure 2.17: Mapping PC RAM to EGA VRAM banks.

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 2.18 the first pixel has color magenta(`05h`), second pixel dark grey (`08h`) and third pixel bright yellow(`0Eh`).

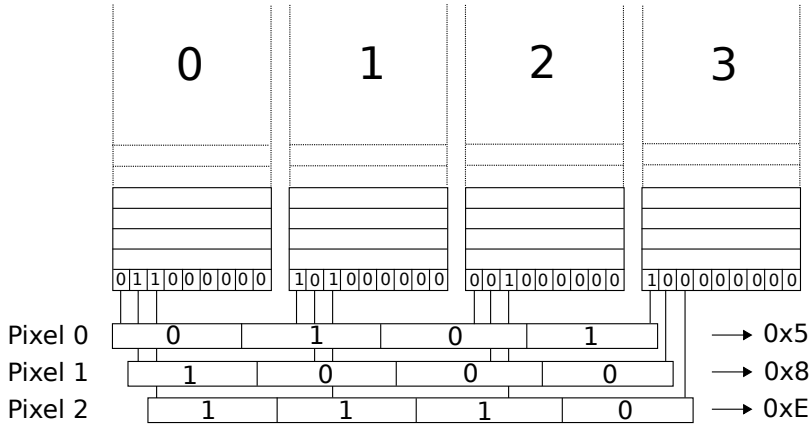


Figure 2.18: EGA bank layout

2.3.7.1 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
```

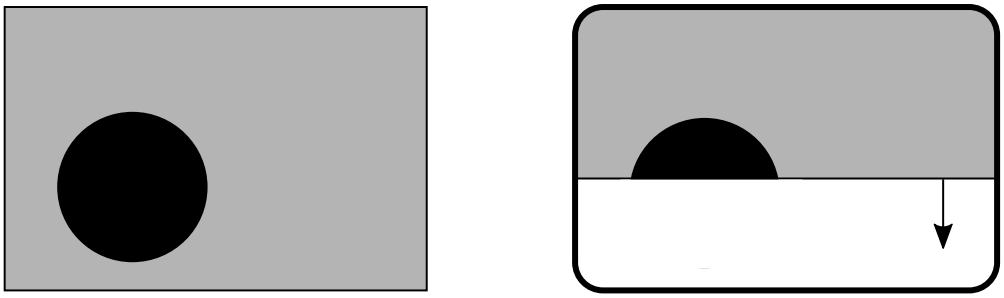
The `geninterrupt (0x10)` instruction is a software interrupt caught by the BIOS routine in charge of graphic setup. It looks up the `ax` register, which can be set in the Borland Compiler by `_AX`, to setup all EGA registers with the corresponding mode.

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.

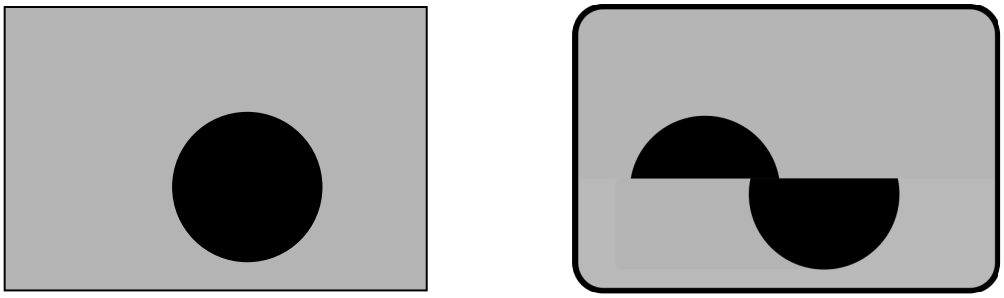
```
char far *EGA = (unsigned char far*)0xA0000000L;

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

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

2.3.8 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".

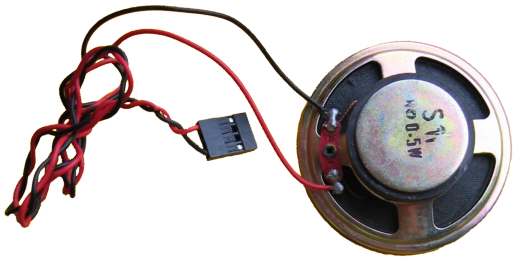
With two buffers (a.k.a double buffering) the CPU can start writing in the second framebuffer without messing with the framebuffer being scanned to the screen[16]. No more tearing!

Note that creating 320x200 picture 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.



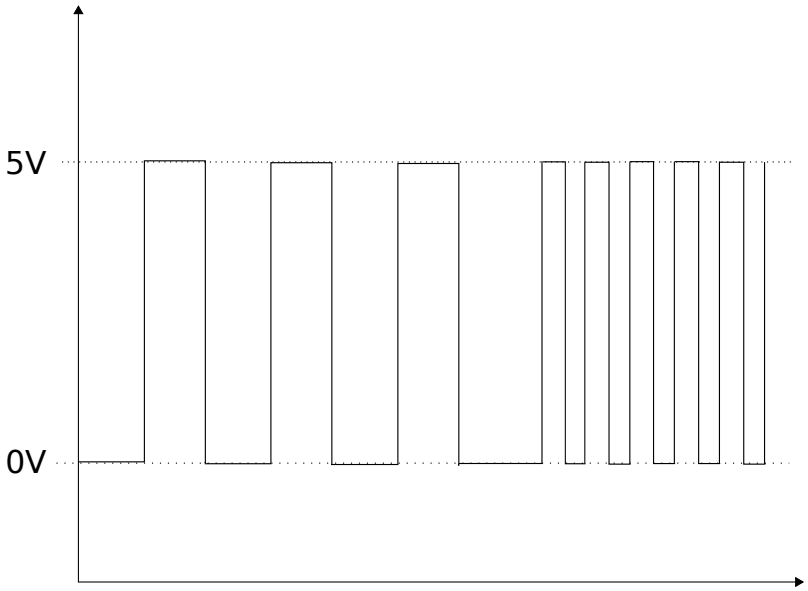


Figure 2.19: Two beeps of different frequencies generated via PC Speaker.

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".

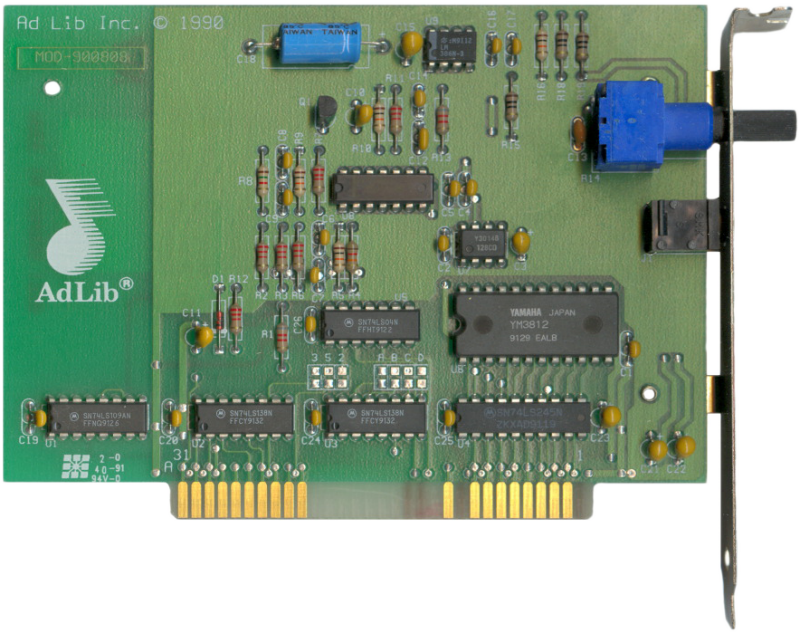


Figure 2.20: An AdLib sound card. Notice the big YM3812 chip and the 8-bit ISA connector.

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.

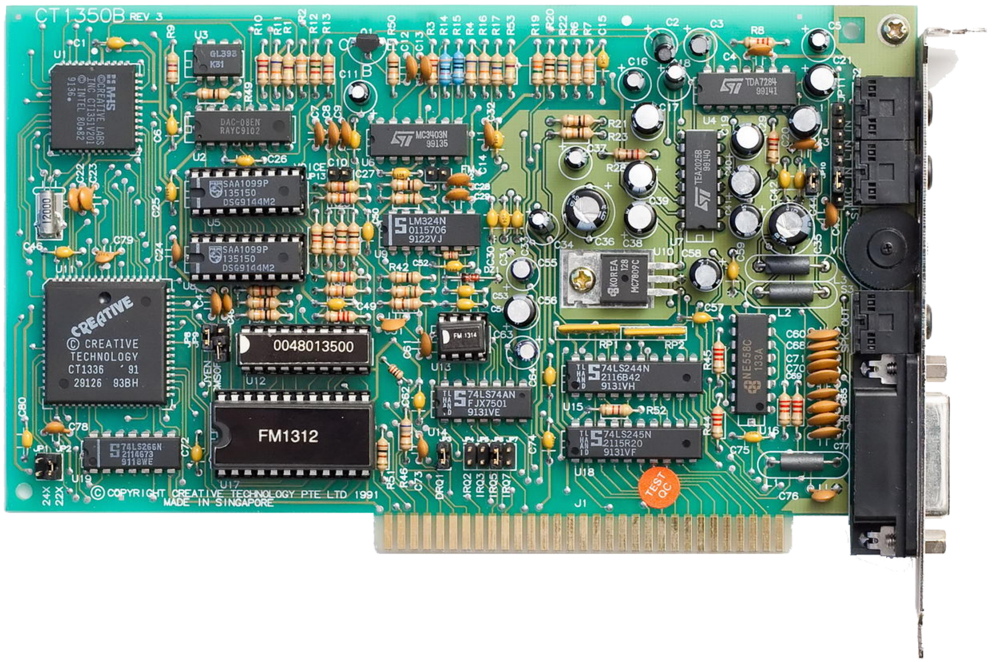


Figure 2.21: A SoundBlaster (v1.2).

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 2.21 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 2.22: The speaker box (DAC not shown).

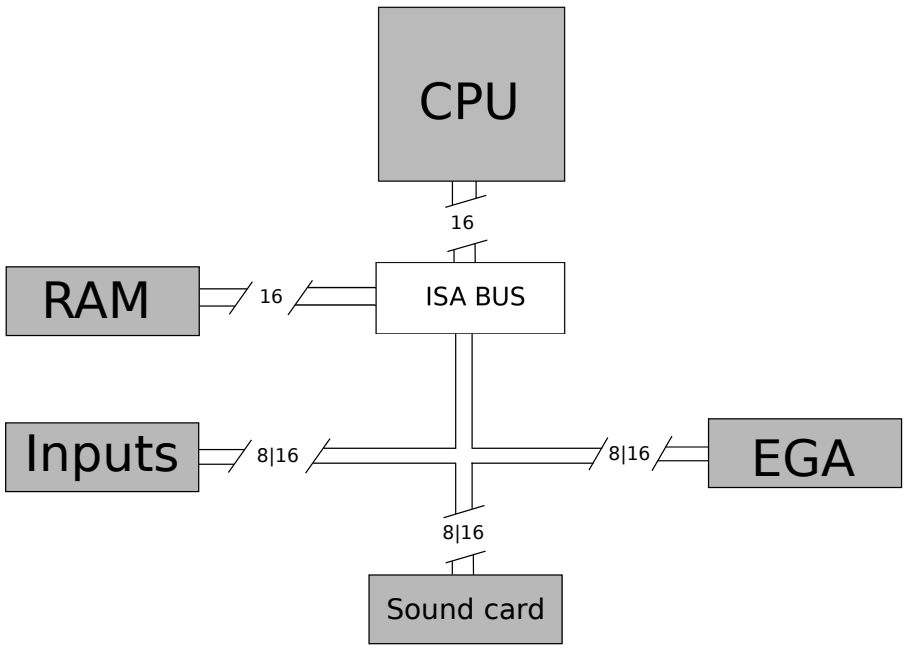
It was incredibly easy to set up, simple to program (it could only play one type of PCM and had no FM synthesizer), and very cheap compared to the other audio solutions ($14). It would have made programmers and customers happy if not for one serious limitation. The parallel port bandwidth[21] allowed 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 Bus

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

[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.

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.



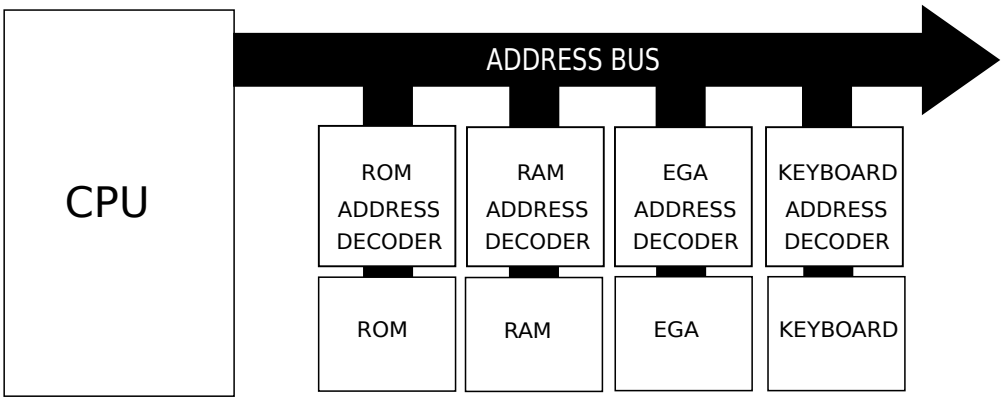
[22] Industry Standard Architecture.

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`].



In practice the effective bandwidth of the bus is divided by two due to packet overhead and interrupts. As a result, a PC equipped with an 8 bit ISA EGA card can push 19.1Mbit-s/s/2/8 = 1.1MB/s. In mode `0Dh`, since a frame is 320x200/2 = 32,000 bytes, the theoretical maximum framerate with a CPU taking 0ms to render a frame is 1,100,000 / 32,000 = 34 frames per second.

If you factor in other things which had to be transported by the bus such as palette, keyboard interrupt, mouse inputs, and music/sounds it is easy to understand how important it was to limit data transfer.

2.6 Inputs

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

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

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.

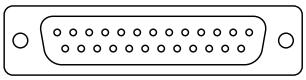


Figure 2.23: Parallel Port

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

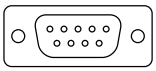


Figure 2.24: Serial Port

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

Figure 2.25: 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].

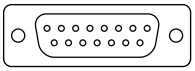


Figure 2.26: Game Port

2.7 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

[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).

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 id Software[25].

[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).

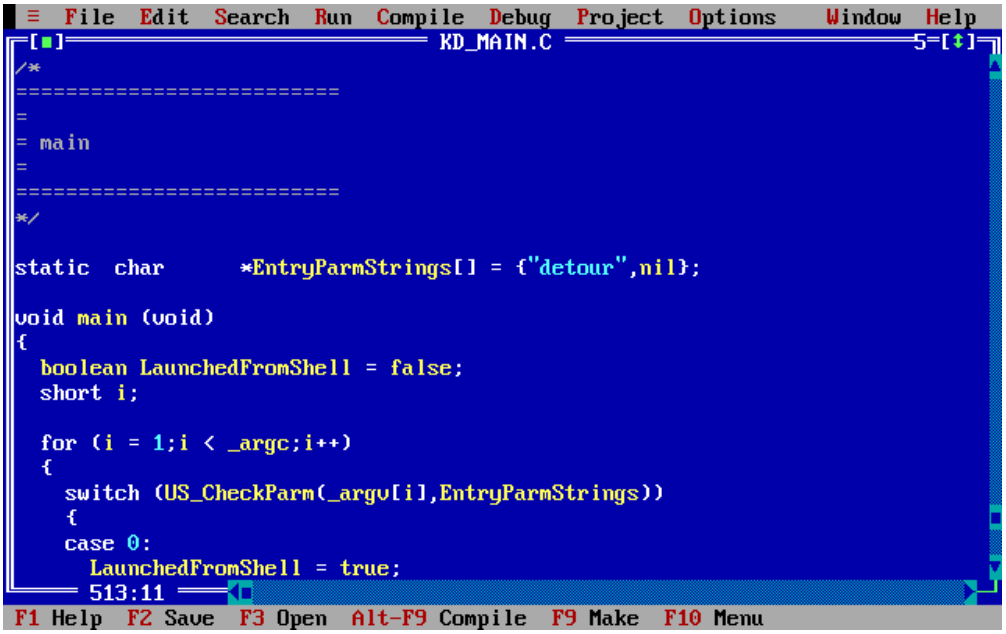


Figure 3.1: Borland C++ 3.1 editor

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.

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

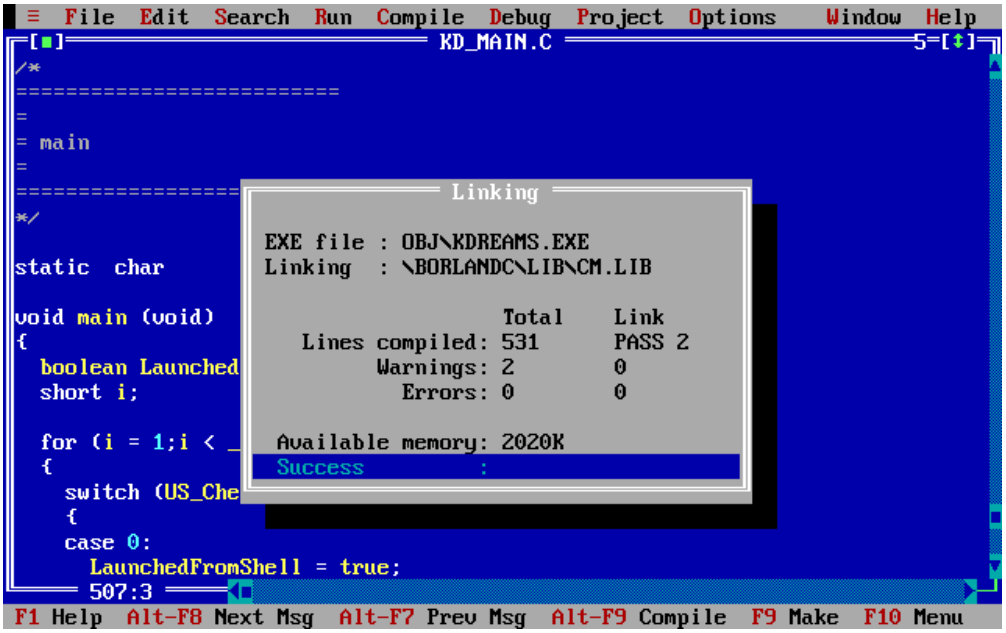
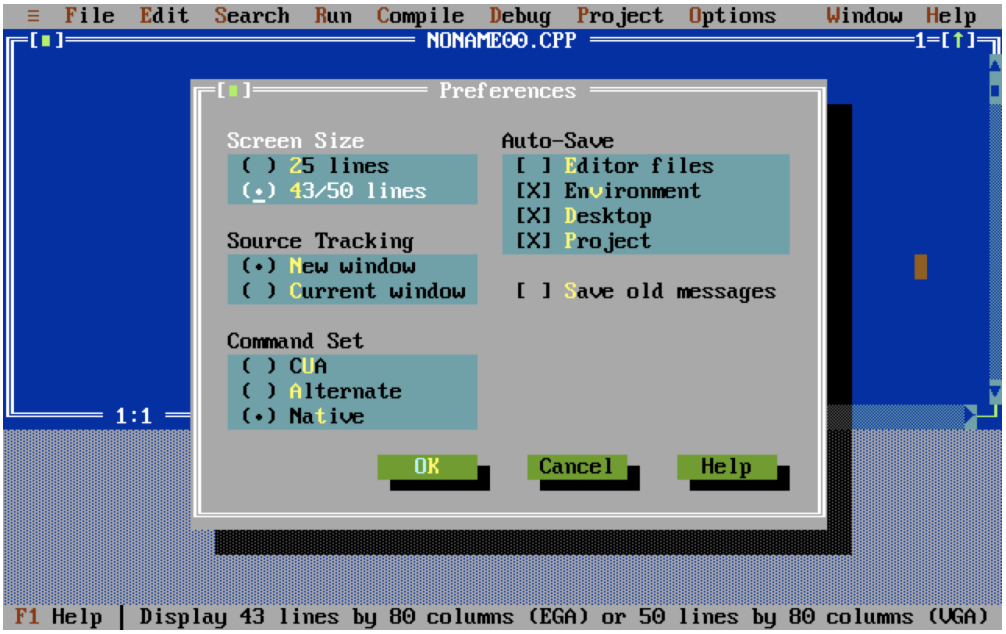
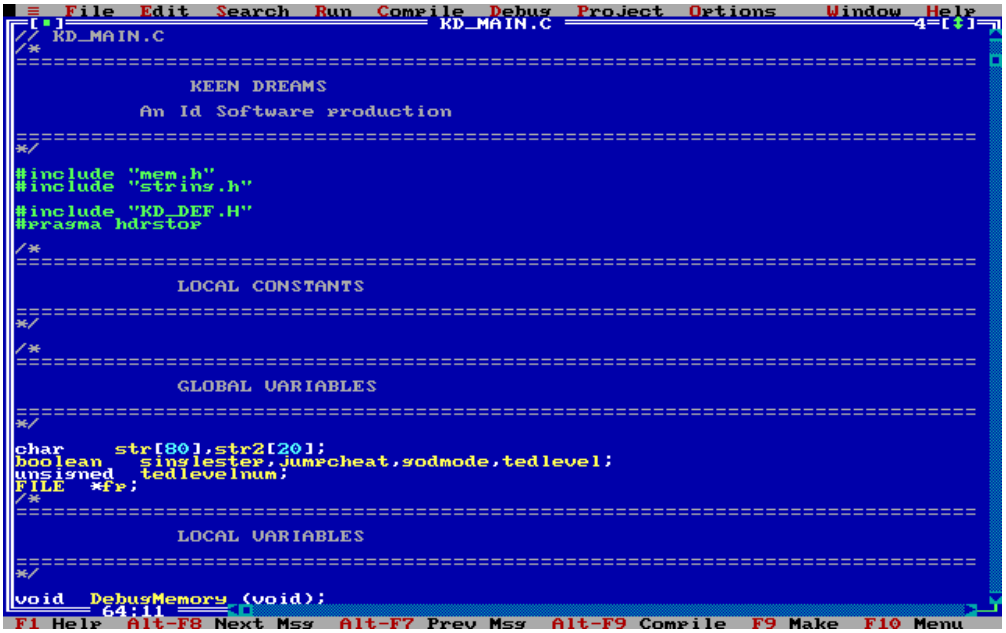


Figure 3.2: Compiling Keen Dreams with Borland C++ 3.1

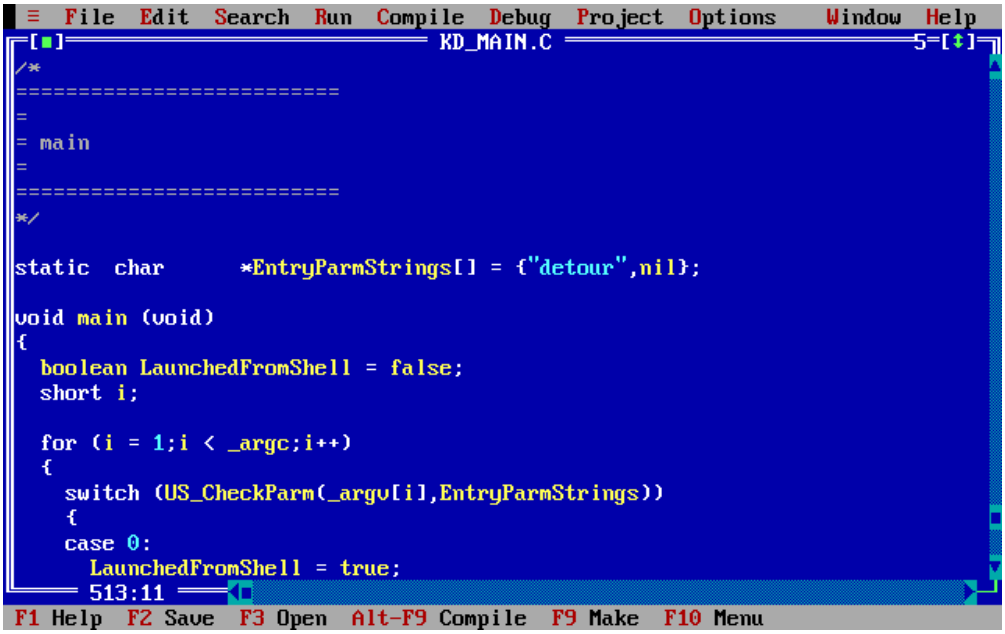
Another way to improve screen real estate was to use "high resolution" 50x80 text mode.

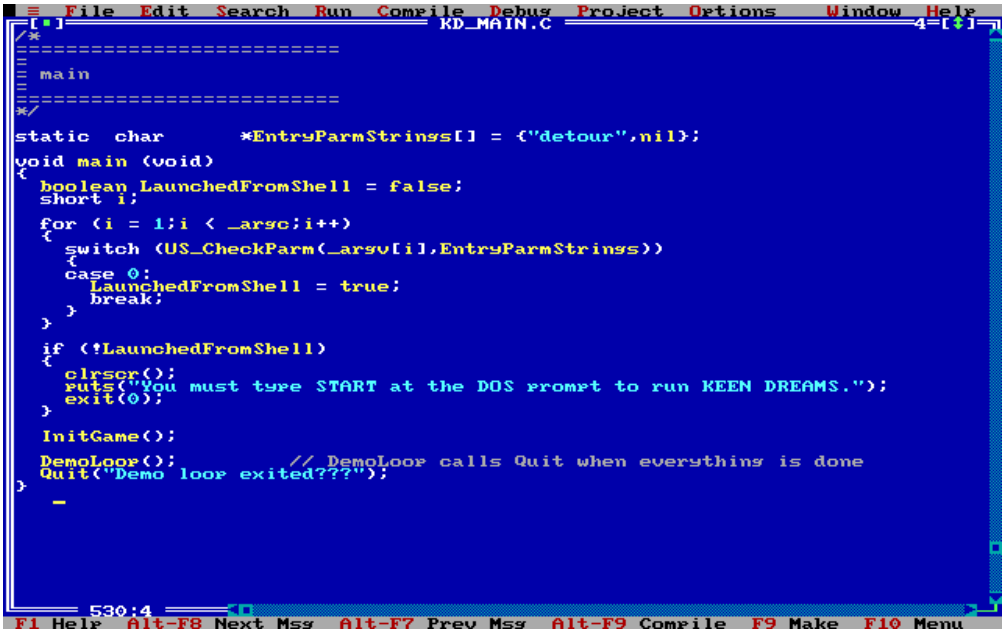


The comments still fit perfectly on screen since only the vertical resolution is doubled.



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


File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
5=[↕]
/*
==========
=
= main
=
==========
*/
static char *EntryParmStrings[] = {"detour",nil};
void main (void)
{
boolean LaunchedFromShell = false;
short i;
for (i = 1;i < _argc;i++)
{
switch (US_CheckParm(_argv[i],EntryParmStrings))
{
case 0:
LaunchedFromShell = true;
513:11
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu


File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
4=[↕]
/*
==========
=
= main
=
==========
*/
static char *EntryParmStringss[] = {"detour",nil};
void main (void)
{
boolean LaunchedFromShell = false;
short i;
for (i = 1;i < _argc;i++)
{
switch (US_CheckParm(_argv[i],EntryParmStringss))
{
case 0:
LaunchedFromShell = true;
break;
}
}
if (!LaunchedFromShell)
{
clrscr();
puts("You must type START at the DOS prompt to run KEEN DREAMS.");
exit(0);
}
InitGame();
DemoLoop(); // DemoLoop calls Quit when everything is done
Quit("Demo loop exited???");
}
530:4
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

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.

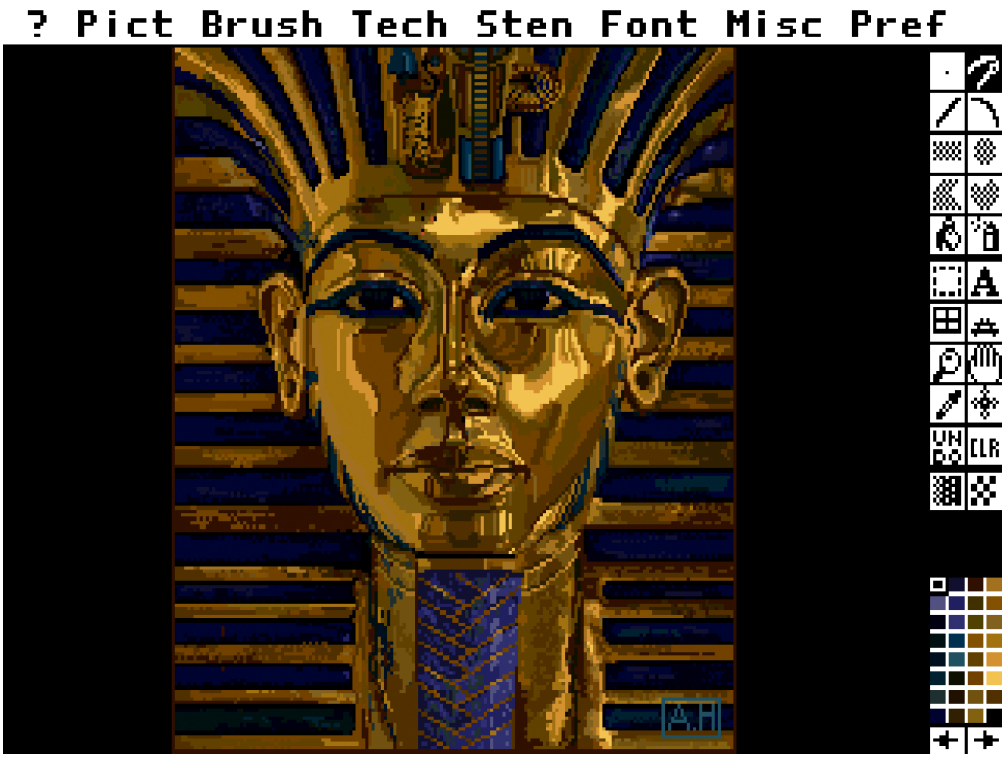


Figure 3.3: Deluxe Paint was used to draw all assets in the game.

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.

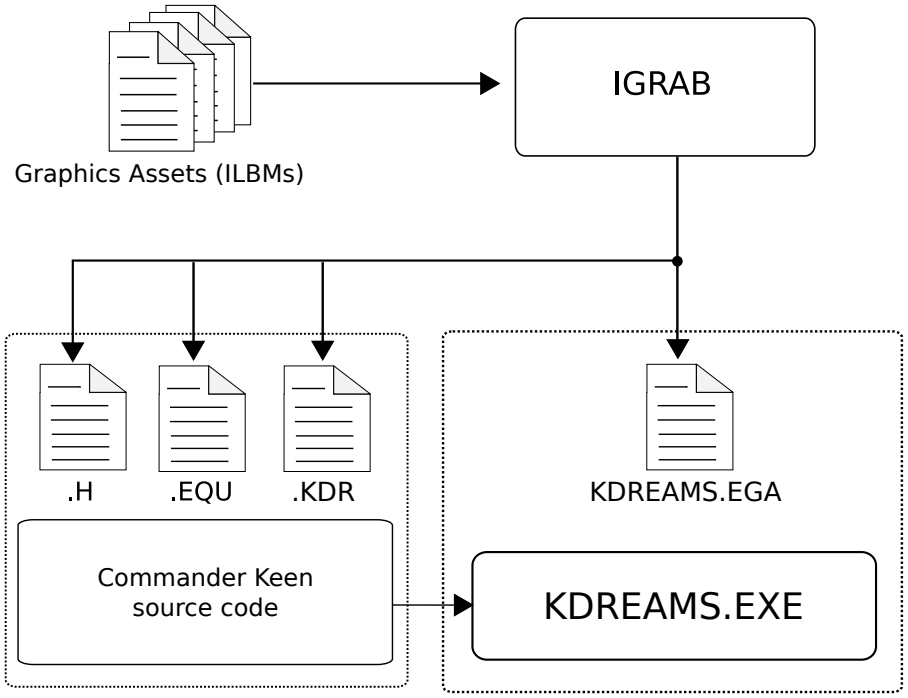


Figure 3.4: Asset creation pipeline for graphics items

```
//////////////////////////////////////
//
// 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 gives 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 **??** shows what is inside the `KDREAMS.EGA` asset file

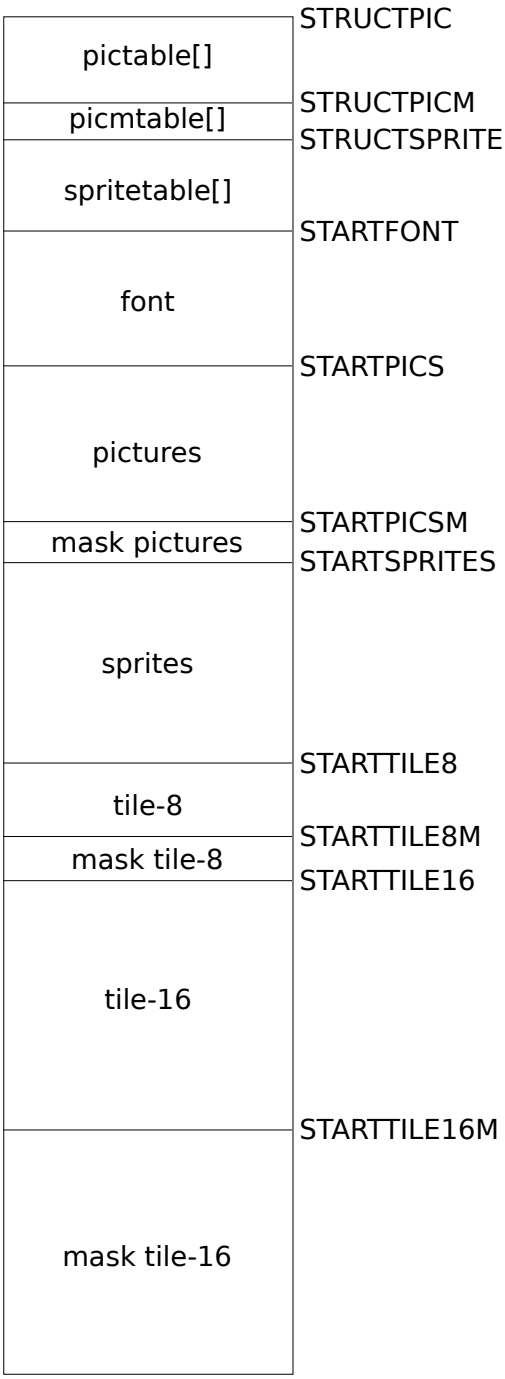
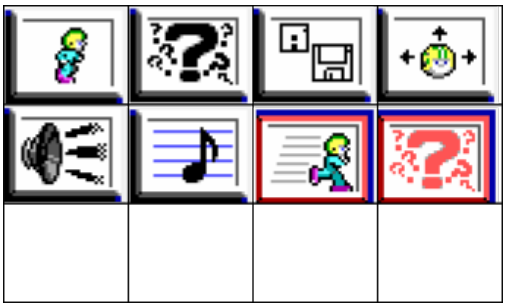


Figure 3.5: File structure of `KDREAMS.EGA` asset file.

The `pictable[]` contains the width and height in bytes for each picture in the asset file. The actual picture data is located in the `pictures` location of the file. The same table structure applies for the 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

(a) content of pictable[].



(b) Pictures data.

Figure 3.6: Picture table structure in `KDREAMS.EGA` asset file.

The `spritetable[]` contains beside width and height in bytes also information on the sprite center, boundaries and number of shifted sprites (this will be explained in section xxx). The sprites are stored in the `sprites` area.

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 3.1: content of spritetable[].

Figure 3.7: Sprite data.

Chapter 4

Software

4.1 About the Source Code

Commander Keen series 1-3 and 4-6 source code is not available as the current owner Zenimax **??** 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 for the 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
```

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[1] 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.

[1] 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`.

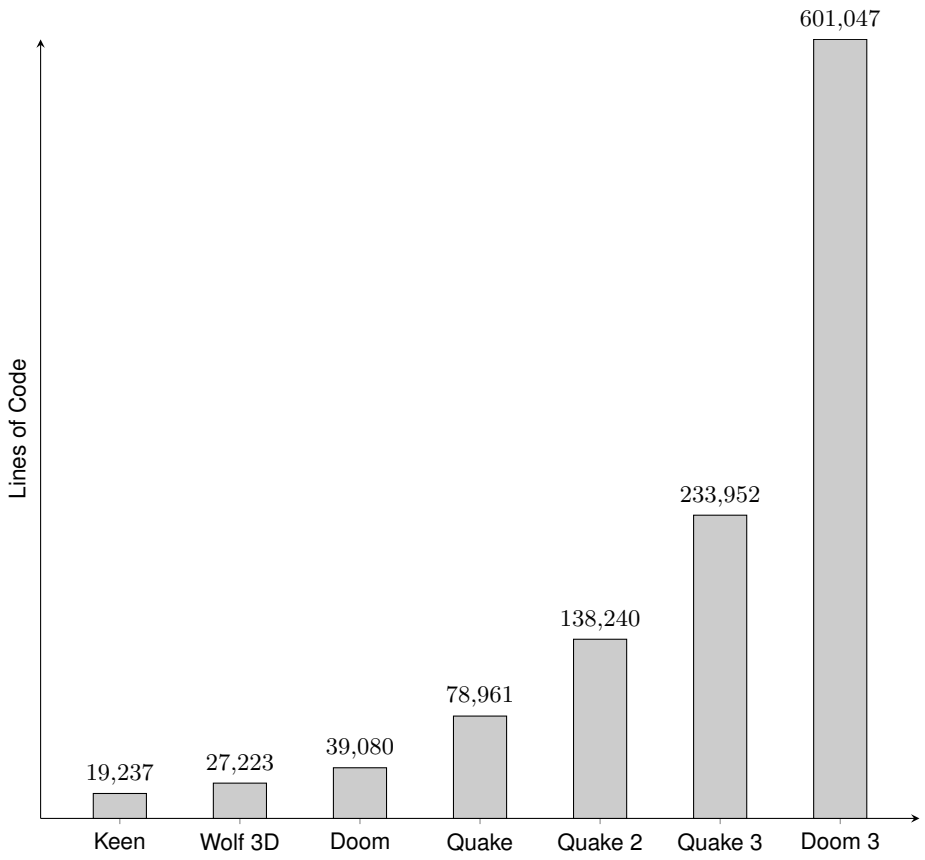


Figure 4.1: Lines of code from id Software game engines.

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

- `static` folder: Static header files for loading assets (will be explained later).
- `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` files.

```
chdir STATIC
make.bat
```

Once the static object 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:

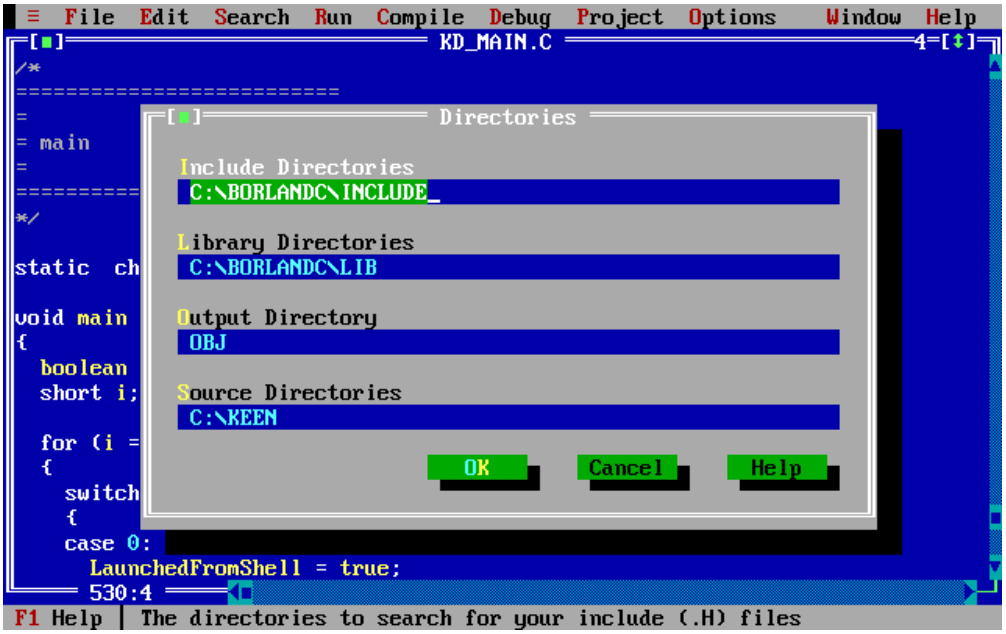


Figure 4.2: Borland C++ 3.1 directory settings

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.

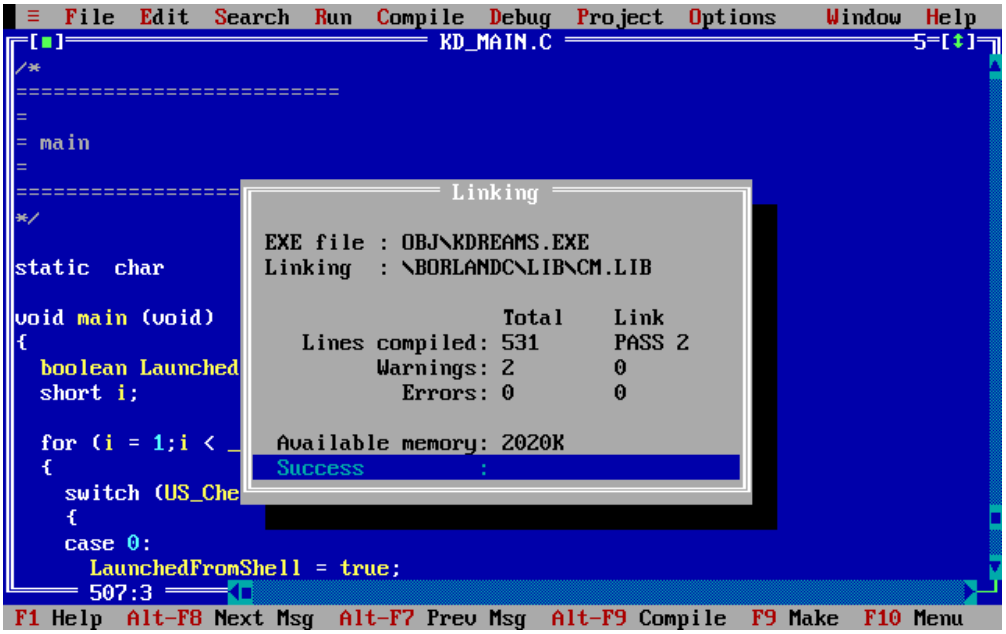


Figure 4.3: Commander Keen compiling

4.5 Big Picture

The game engine is divided in three blocks:

- Menu engine which lets users configure 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 music and 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 world according to the wall-time tracked by `TimeCount` variable.

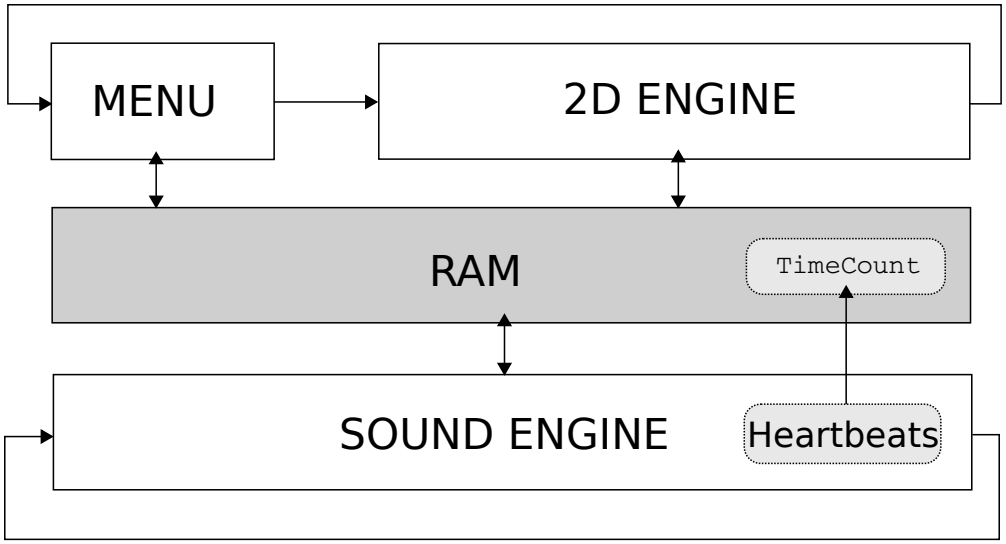


Figure 4.4: Game engine three main systems.

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 two renderers 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 32-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 ();     // Font 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 world, and render world 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 1991).

The reason for interrupts is extensively explained in Chapter 3.15 "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.

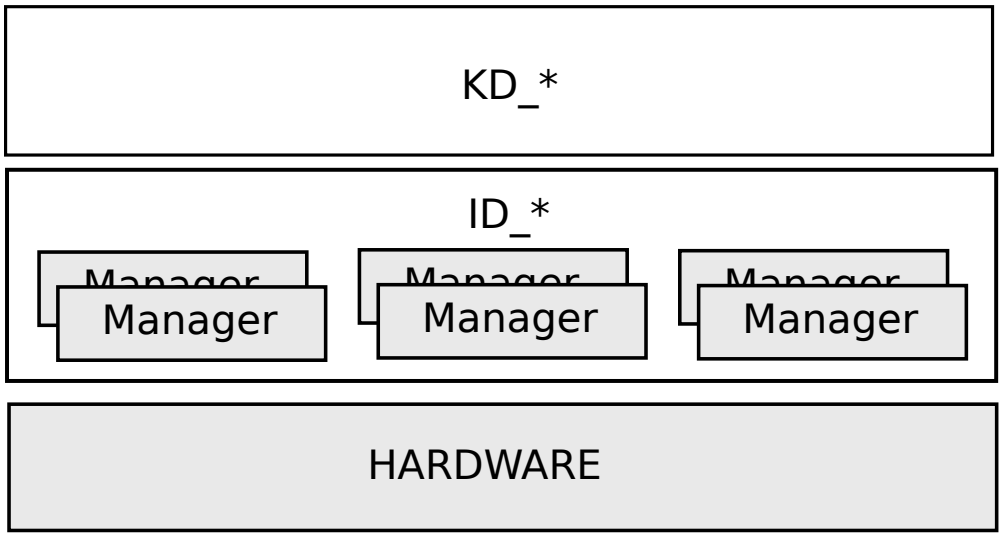


Figure 4.5: Commander Keen source code layers.

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).

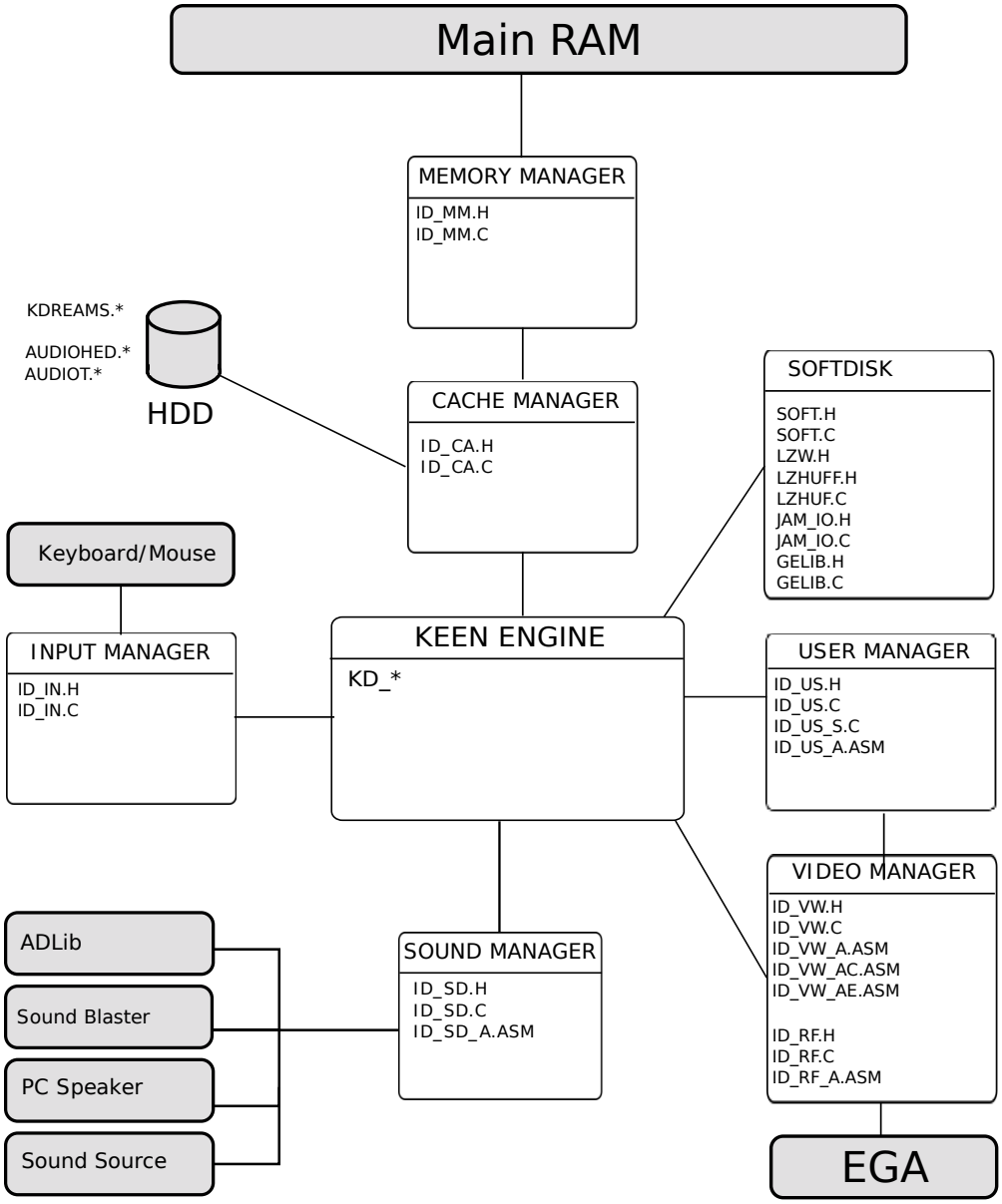


Figure 4.6: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

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

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.

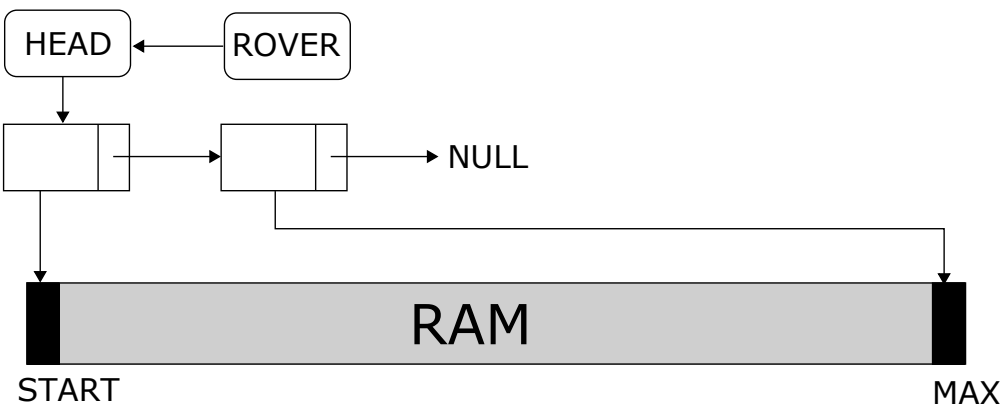


Figure 4.7: Initial memory manager state.

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`.

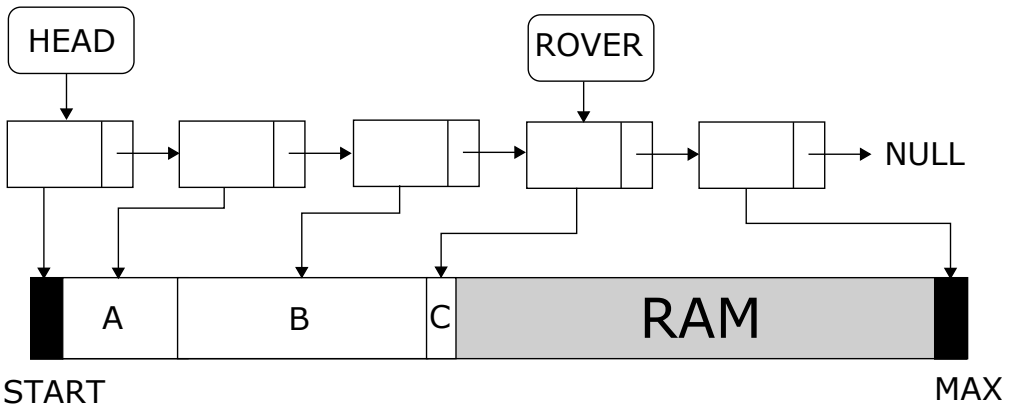


Figure 4.8: MM internal state after three pass 1 allocations.

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

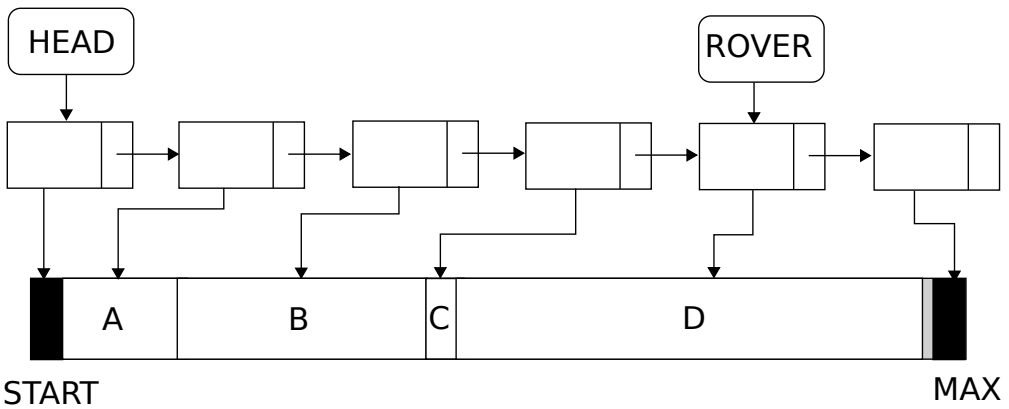


Figure 4.9: Pass 1 failure: Not enough RAM after the ROVER.

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).

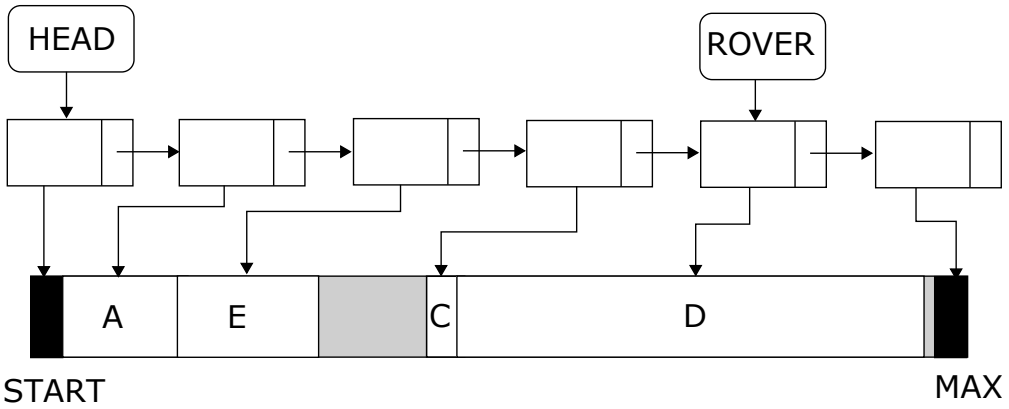


Figure 4.10: `B` was purged. `E` was allocated in pass 2.

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.

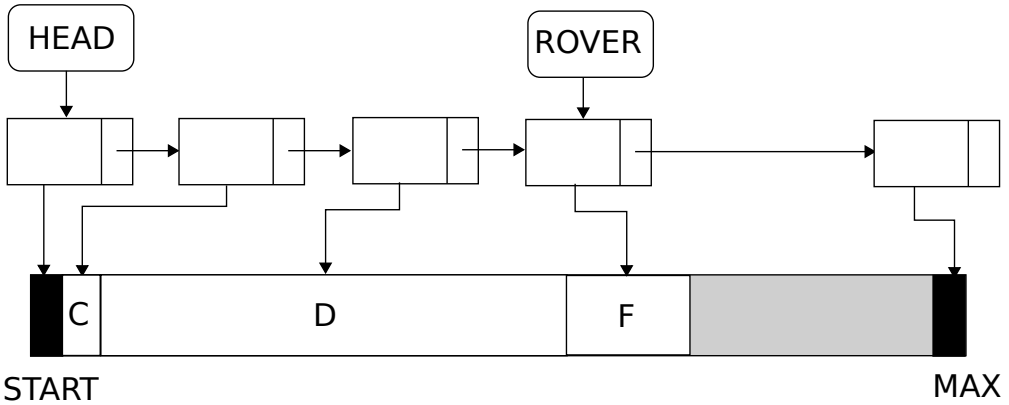


Figure 4.11: `A` and `E` were purged. `C` and `D` compacted. `F` allocated in pass3.

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.

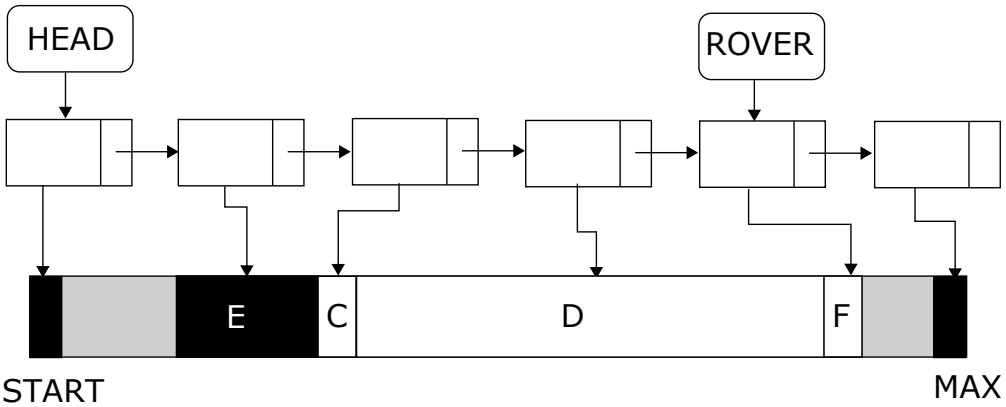


Figure 4.12: `E` is locked and cannot be compacted.

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 update tiles, and is also made both C and ASM code.

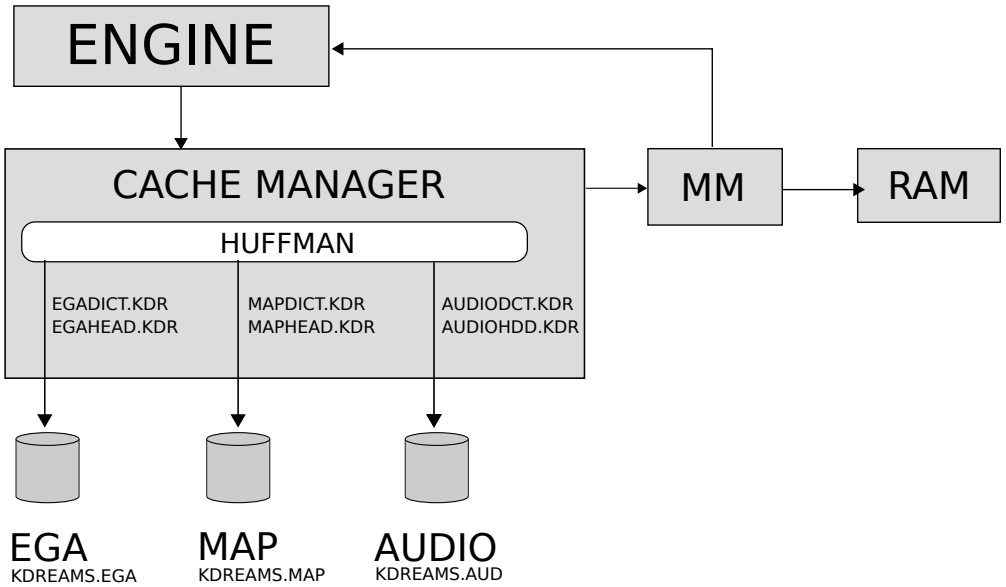
The video manager is described extensively in the "XXX" section.

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 XXX. 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 keep track of required assets, an array `gr_needed[]` is maintained to mark if an asset needs to be loaded from disk.

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

ca_levelbit = 1;

void InitGame (void)
{
  [...]   //
          // load in and lock down some basic chunks
          //

  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

}
```

The function `CA_CacheMarks()` loads and decompress all required graphical assets from disk to memory. Since access and loading from disk is a slow process, the engine will try to load as much as possible assets in one go from the disk.

```
#define NUMCHUNKS 3016  // Maximum number of graphic assets
int grhandle;           // handle to EGAGRAPH

void CA_CacheMarks (char *title, boolean cachedownlevel)
{
  int   i,next;
  long  pos,endpos,compressed;
  byte  far *source;

  //
  // go through and load in anything still needed
  //
  for (i=0;i<NUMCHUNKS;i++)
    // Asset needed, but not loaded in memory
    if ( (grneeded[i]&ca_levelbit) && !grsegs[i])
    {
      pos = grstarts[i];
      next = i +1;
      while (grstarts[next] == -1)    // skip past any
   sparse tiles
        next++;

      compressed = grstarts[next]-pos;
      endpos = pos+compressed;

      // load buffer with a new block from disk
      // try to get as many of the needed blocks in as
   possible
      {
        [...]
        lseek(grhandle,pos,SEEK_SET);
        CA_FarRead(grhandle,bufferseg,endpos-pos);
        source = bufferseg;
      }
      CAL_ExpandGrChunk (i,source);  // Decompress data
    }
  }
}
```

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 for a given 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 the complete font is stored with the following information of 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, where the width could vary as illustrated in Figure 3.13.

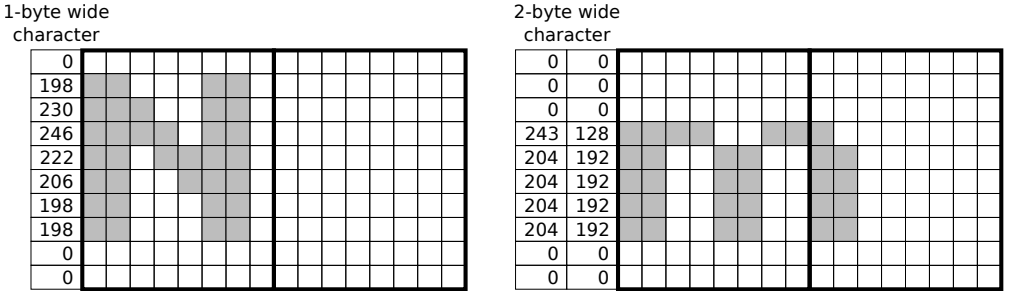


Figure 4.13: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

As explained before, in the video memory each 8 pixels are represented by 1 byte. So how do we print a character which is not perfectly aligned with the memory layout? Here a trick of bit shift 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 3.14 you see how the bitshift is working for the values '1' and '198'.

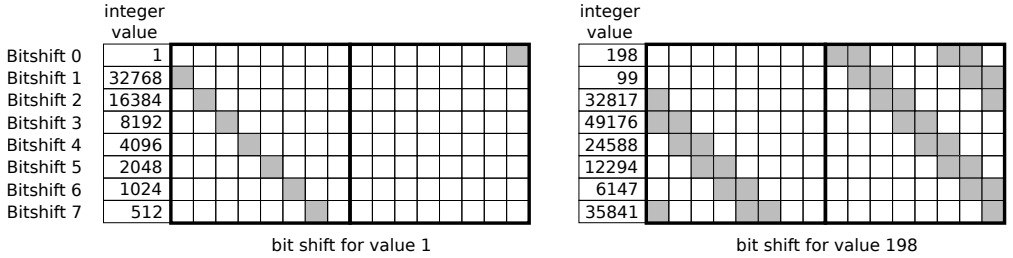


Figure 4.14: Right bitshift [0-7] for 1 and 198.

Each bitshift table is generated in `id_vw_a.asm`

```
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 `x` 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

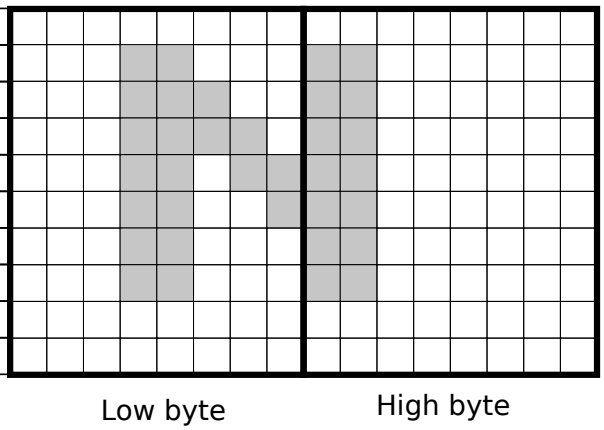


Figure 4.15: Bitshift 'N' over 3 bits using bit shift tables.

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. To avoid the next character overwrites the last high byte in the buffer, every low byte is added to the last high byte by applying a logical `OR`-operation.

```
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.

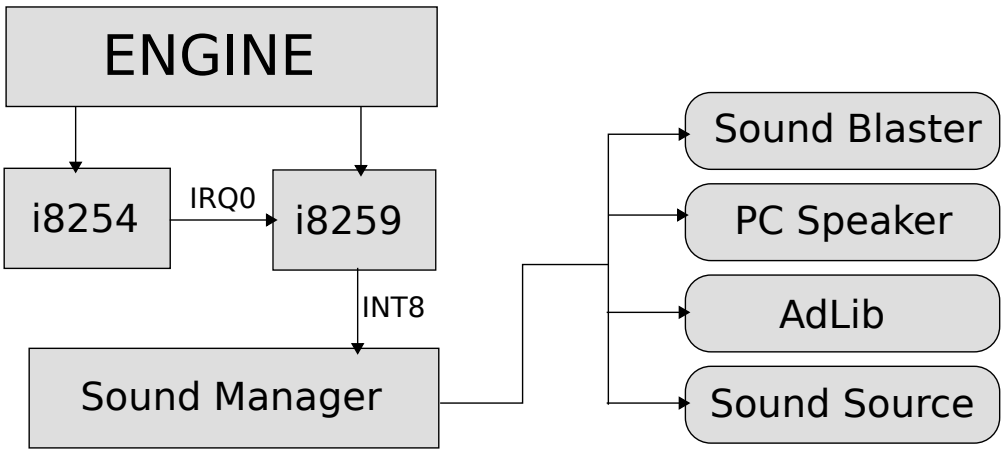


Figure 4.16: Sound system architecture.

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. However, 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 and "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 4.17: Keen Dreams intro screen

4.8 Action Phase: Adaptive Tile Refreshment

After the player is done setting up the game, it is time for the scrolling engine to shine. On bitmapped displays without hardware scrolling like the EGA card, the entire screen have to be erased and redrawn in the slightly shifted position whenever the player moved in any direction. This would kill the CPU as you need to update all pixels of all four planes on the EGA card (remember the planar mapping of section 2.3.3?).

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

The visible screen is divided into tiles of 16x16 pixels. On a screen with 320x200 pixels, it means a grid of 20x13 tiles (actually it is 12.5 tiles high, but we need to round to integer). Let's look at *Commander Keen 1: Marooned on Mars* in Figure 3.18. 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 3.18 only 69 tiles of the total 260 tiles need to be refreshed, which is 27% of the screen!

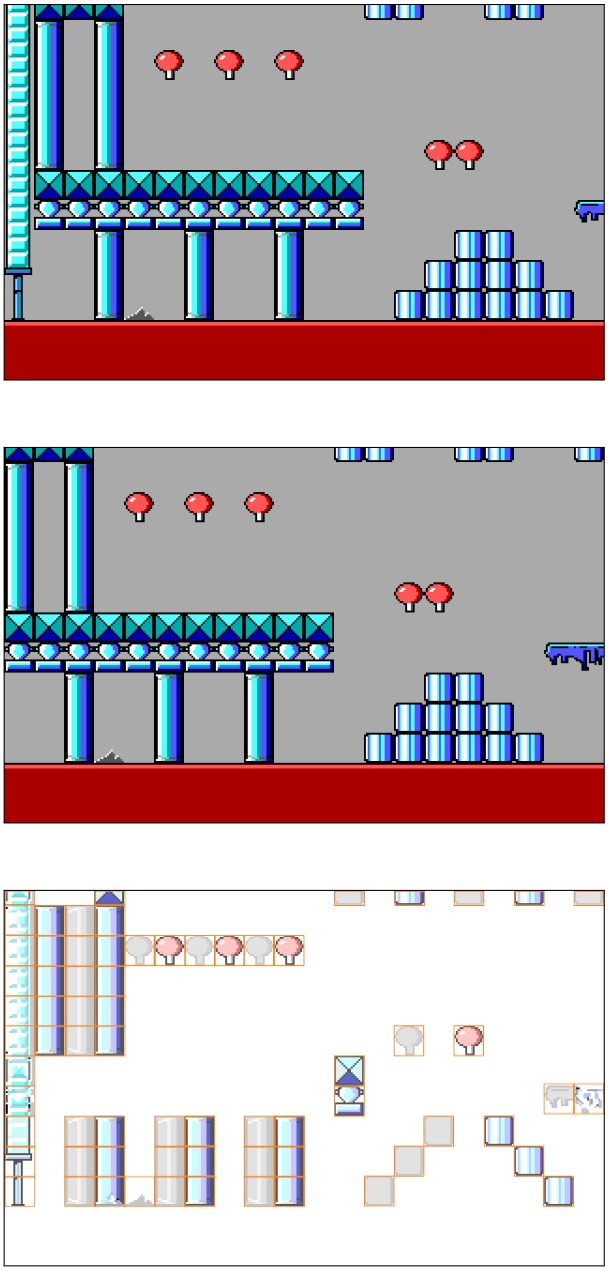


Figure 4.18: Start of the world, moved one tile to the right and difference.

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 logical width of the virtual screen is expressed in the number of `words` of display memory considered to make up one scan line. So 20 `words` of display memory is setting a scan line of 320 pixels. 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 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 Start Address register.

```
CRTC_INDEX   = 03D4h
CRTC_STARTHIGH   = 12

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

  cli                       ;disable interrupts

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

  sti                       ;enable interrupts

  ret
```

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 move back to pixels 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 3.19:

- 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

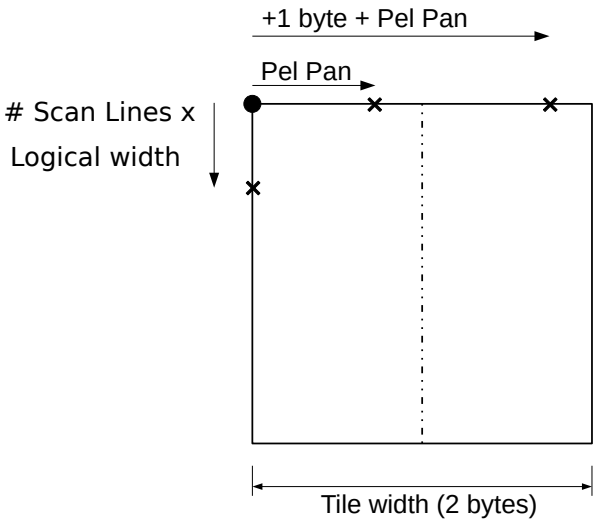


Figure 4.19: Smooth scrolling in EGA.

4.9 View Port and Buffer setup

Before explaining the scrolling algoritm, let's first explain how the view port and buffer layout are setup. The visible viewing screen on EGA has a resolution of 320x200 pixels. Translated in 16x16 pixel tiles, the screen view has a size of 20x13 tiles. By making the view port one tile higher and wider than the screen (21x14 tiles), 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 using the Start Address and Pel Pan registers. Finally, an update buffer is defined that has enough space to float the view port up to two tiles in all direction. At the end of Section 3.11.4 it is explained why we need a spare buffer of 2 tiles.

So summarized, as illustrated in Figure 3.20, the following tile views are defined:

- Screen View size of 20x13 tiles and Port View size of 21x14 tiles.
- Buffer screen size of 22x14 tiles. This is one tile wider than the Port View, where the additional tile is used to mark a '0' at the end of each tile row.
- Total buffer size is stored in UPDATESIZE, which contains the UPDATESCREENSIZE and 2 two times a spare buffer to support the floating of two tiles in any direction.

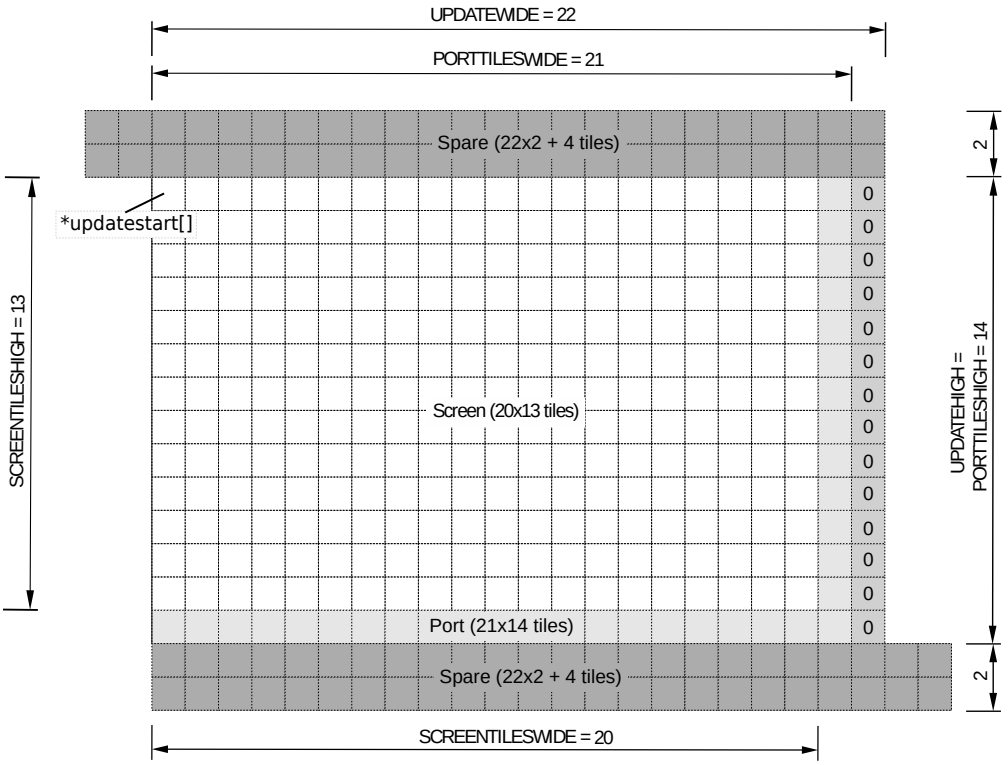


Figure 4.20: View and buffer port tile layout.

4.10 Screen buffer

Even if the screen is not scrolling, tile refreshes are required to support sprite animations. Since moving a sprite in this way involves first erasing it and then redrawing it, the image of the erased sprite may be visible briefly, causing flicker. This is where double buffering comes in: setting up a second buffer into which the code can draw while the first buffer is being shown on screen, which is then switched out during screen refresh. This ensures that no frame is ever displayed mid-drawing, which yields smooth, flicker-free animation.

Now, let's have a closer look at the EGA memory setup. As explained in the previous section, the view port has a size of 21x14 tiles, which is 336x224 pixels. That means the logical width in VRAM must be at least 336 pixels (42 bytes) wide. In the file `id_vw.h` the VRAM screen buffer is defined by SCREENSPACE, which is set to 64x240 bytes, or 512x240 pixels. This is more than sufficient to update one virtual screen in VRAM.

Since one screen only uses 15,360 bytes of VRAM (which is 3,840 bytes per plane), we

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

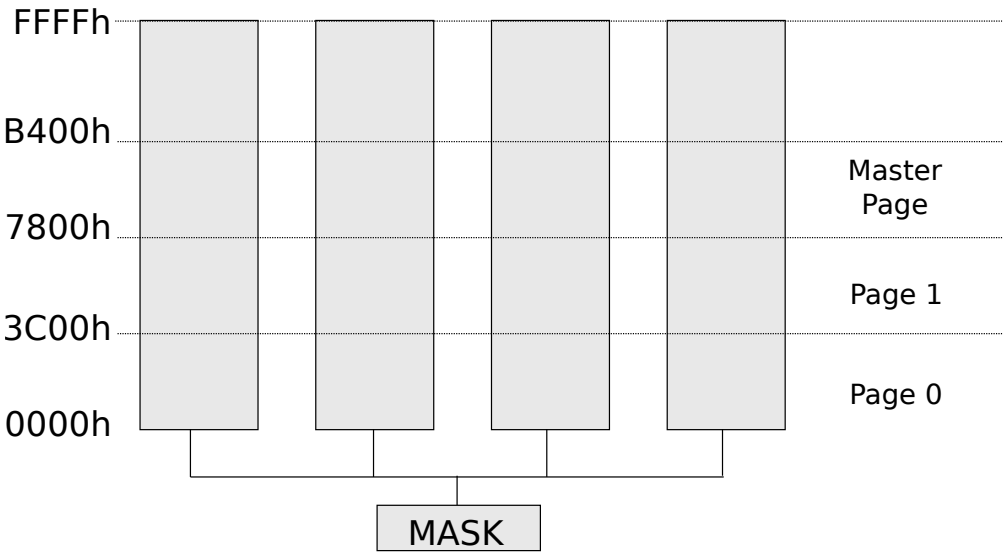


Figure 4.21: Virtual screen layout on EGA card.

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

4.11 Life of a 2D Frame

The approach of refreshing the screen is different between the first Commander Keen games, Commander Keen 1-3, and the ones after. In the first games the algoritm keeps the screen view and buffer 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.

4.11.1 Scroll and screen refresh in Commander Keen 1-3

In the this section we explain how the first games are working[2]. 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 view in VRAM and mark the tiles to be updated in the next refresh.
3. Refresh the buffer view by scanning all tiles. If a tile needs to be updated, copy the tile from the master view to the buffer view in VRAM.
4. Iterate through the sprite removal list and copy corresponding image block from master view to buffer view in VRAM.
5. Iterate through the sprite list and copy corresponding sprite image block from asset location in RAM to buffer view in VRAM
6. Switch the screen and buffer view by adjusting the Start Address and Pel Panning registers.

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

- `screenstart[]` points to the starting address (upper-left pixel) of the viewport in VRAM. As explained before we maintain three viewports in VRAM:
  - screenpage, the active displayed screen on the monitor. Note that the engine never works or updates the active screen.
  - otherpage, which is the buffer screen. This screen is updated and will be switched with the screenpage upon next refresh.
  - masterpage, which stores a complete static background, without sprites. This page is used to update the buffer page.
- `updatestart[]` points to the tile buffer array. It maintains which tiles needs to be updated upon next refresh. There are two tile buffer arrays; one for the screenpage and one for otherpage.
- `Visible screen`, which refers to the starting position of the visible screen on the monitor. This is done by setting both the `CRTR` address and `Pel Panning`. As explained above, the visible screen is within `screenstart[screenpage]`.

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

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

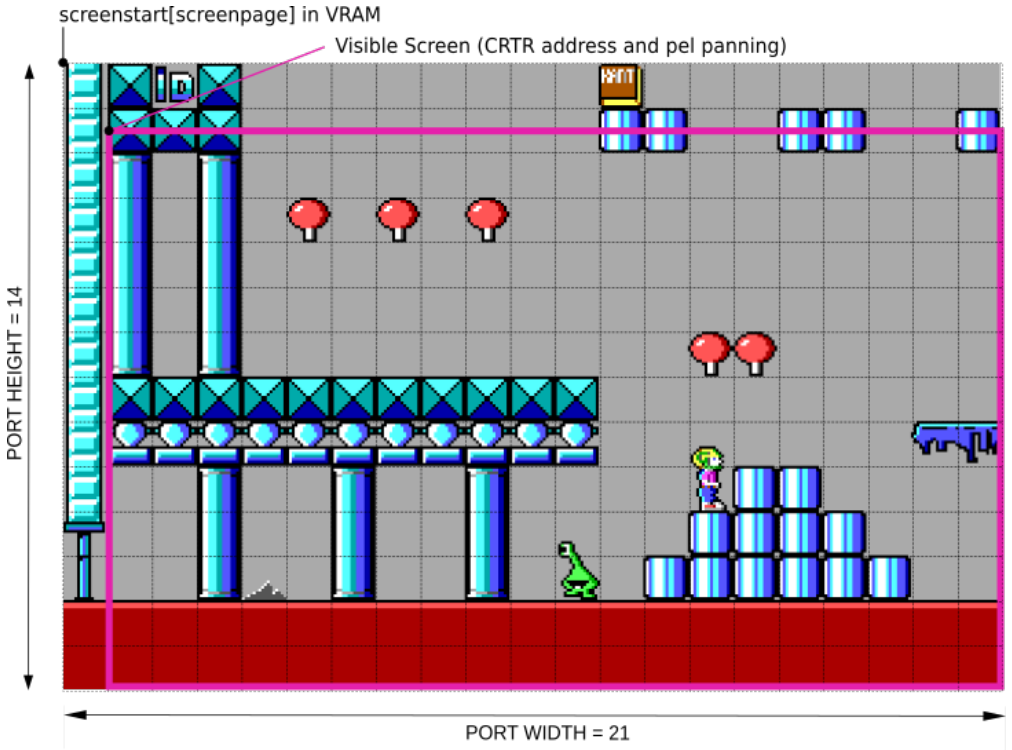


Figure 4.22: Step 1: Player moved to the right and forces the screen to scroll

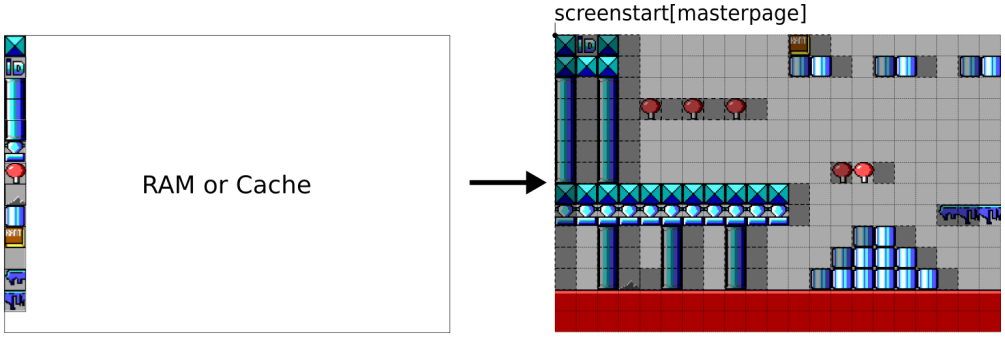


Figure 4.23: Step 2: Update changed tiles in masterscreen, marked in dark grey

Each tile of the buffer screen is compared with the tile on the corresponding level location. If the tile number has changed, the tile is updated by copying tile data from the asset location into the corresponding location in the VRAM of the masterpage.

In parallel each tile in both tile buffer and display array will be marked with a ’1’, which means it needs to be updated upon next refresh.

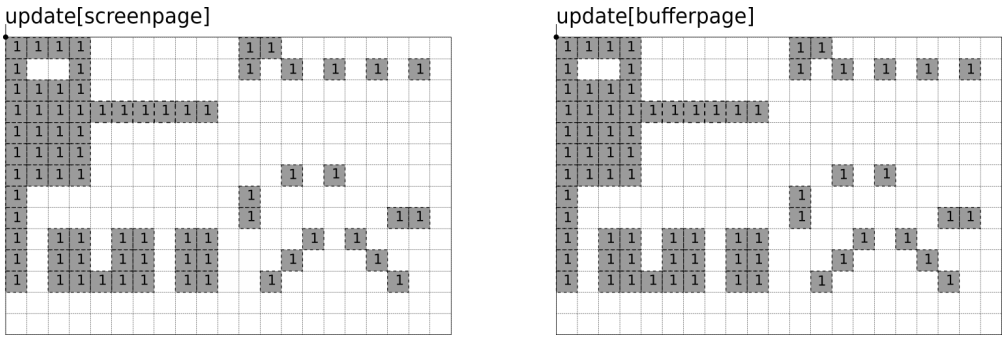


Figure 4.24: Mark all changed tiles with ’1’ in both tile buffer and display arrays.

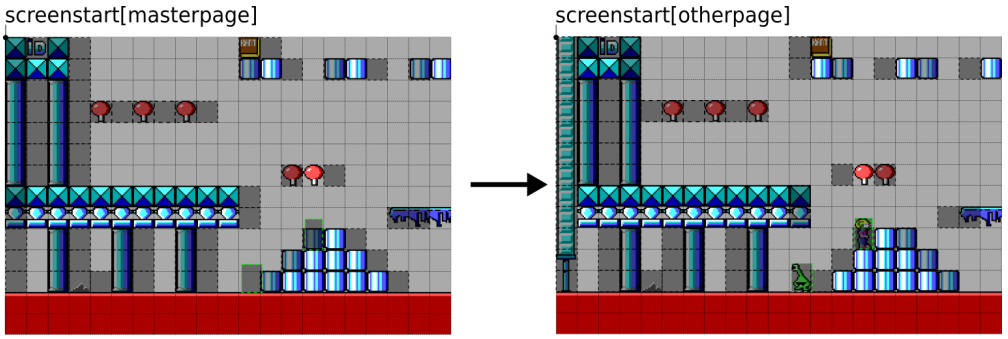


Figure 4.25: Step 3 and 4: Copy tiles from master to buffer screen and remove sprites

Scan all tiles in the tile buffer array and for each tile marked as '1', copy the tile from master to buffer page in VRAM.

If a sprite has moved, the previous sprite location is added to the block removal list. For each block in this list, erase the sprite by copying the width and height size of the sprite block (marked in green in Figure 3.25) from the master screen to the update screen, and mark the corresponding tiles only in the tile buffer array with a '2'.

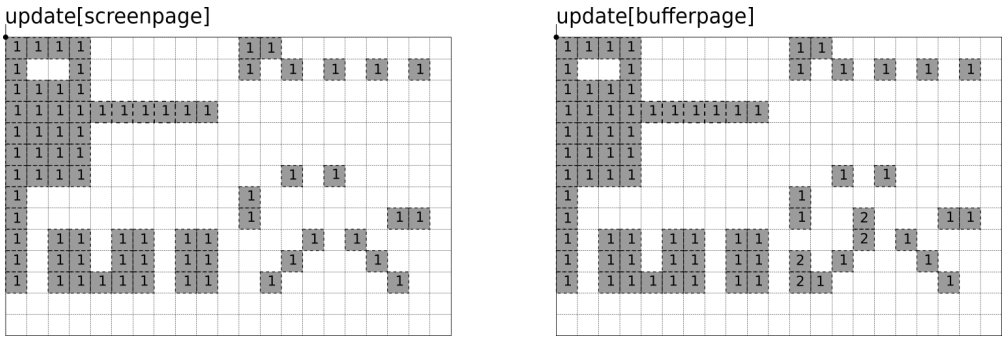


Figure 4.26: Mark removed sprites with '2' in tile buffer array only.

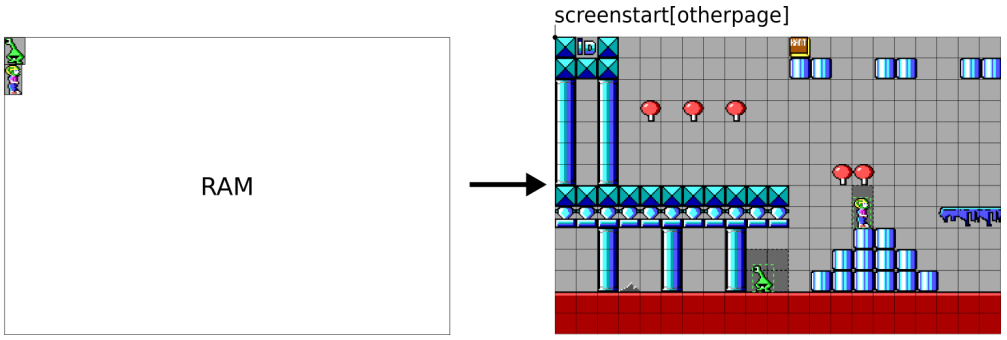


Figure 4.27: Step 5: Scan sprite list and copy sprite onto buffer screen

Scan the sprite list. Validate if the sprite is in the visible part of the view port and copy the sprite image into the VRAM buffer screen. Mark the corresponding tiles in the tile buffer array with a '3'.

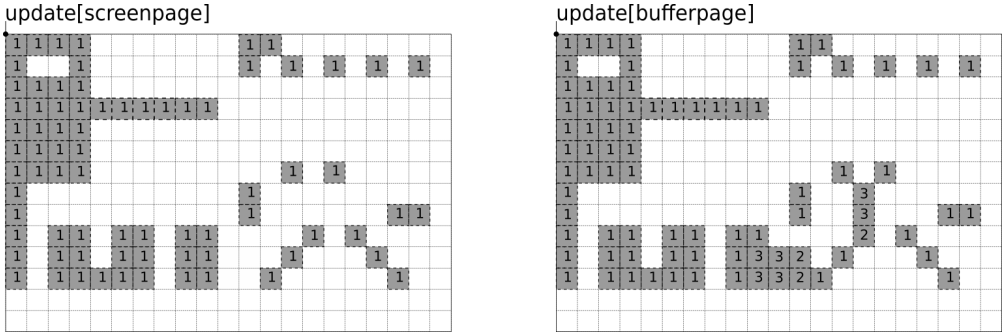


Figure 4.28: Mark new sprites locations with '3' in tile buffer array only.

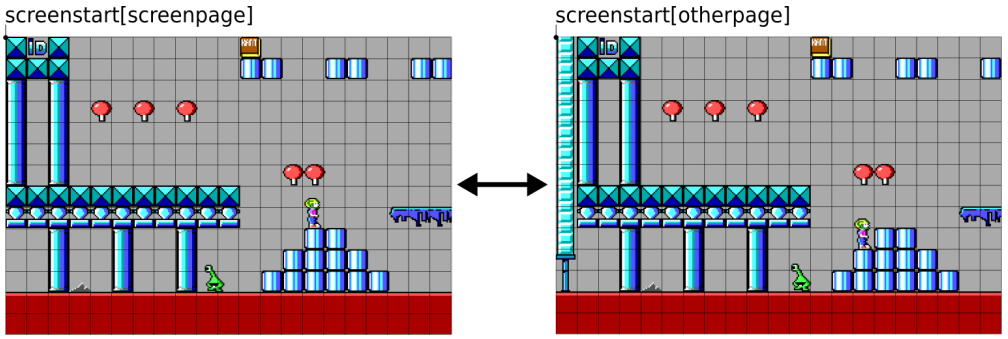


Figure 4.29: Step 6: Swap buffer and screen page

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

Note that after swapping, the tile buffer now has marked all tiles that have changed from scrolling the screen. This makes sense as the current buffer video screen is not yet updated (it was displayed in the previous cycle, remember?).

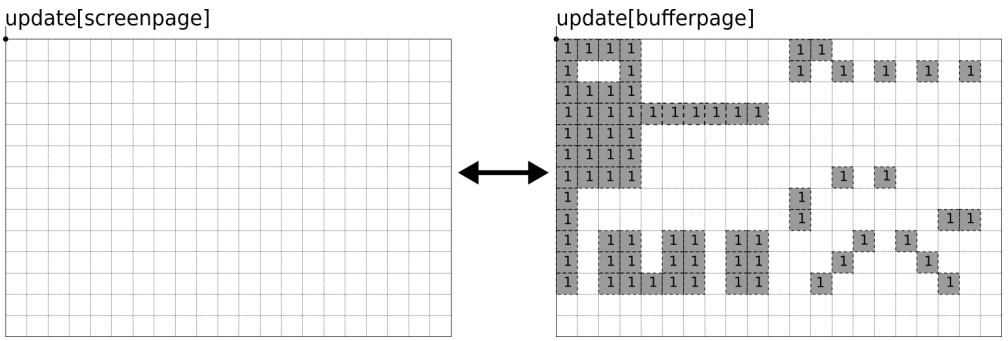


Figure 4.30: Clear tile update array and swap arrays.

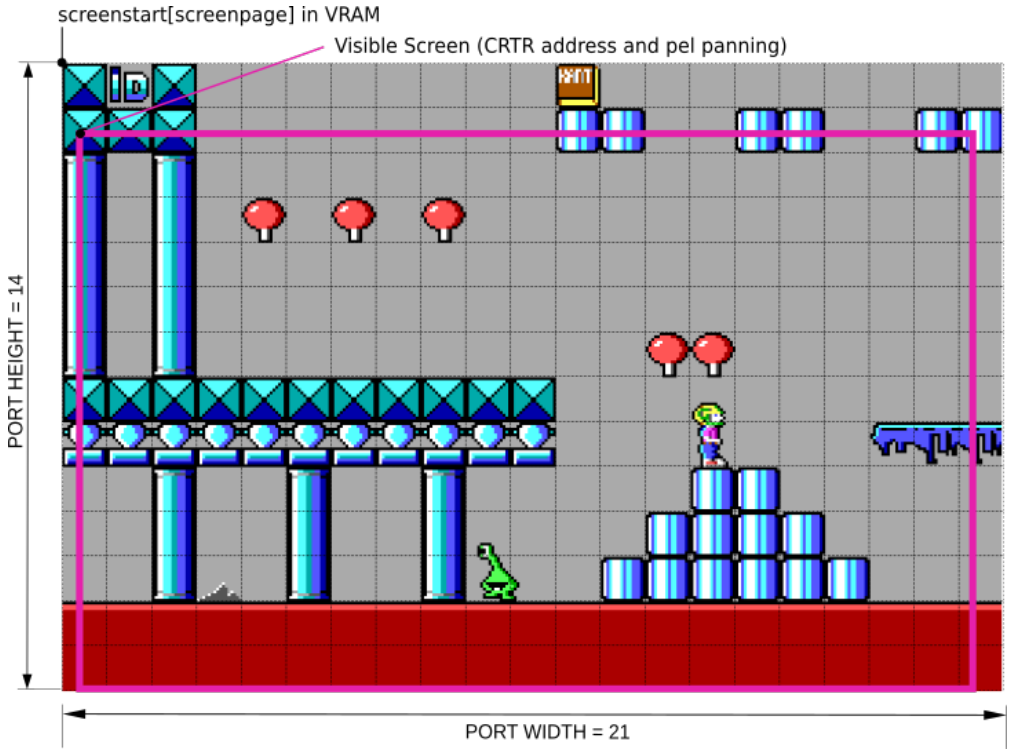


Figure 4.31: Step 6: Swap buffer and screen page

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.

4.11.2 Optimize tile updates

The hardware chapter described Mode 0Dh, which despite being unfit for games, still has an interesting characteristic. As explained in Section 2.3.7 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.

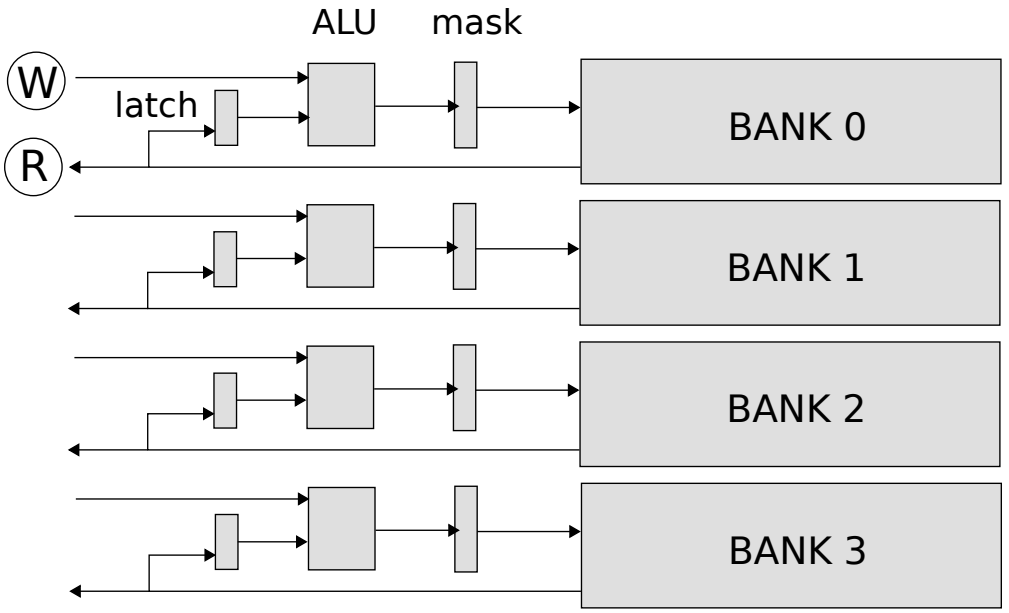


Figure 4.32: Latches memorize read operations from each bank. The memorized value can be used for later writes.

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

By getting a little creative, the circuitry can be re-purposed. The ALU in front of each bank can be setup to use only the latch for writing. With such a setup, upon doing one read, four

latches are populated at once and four bytes in the bank are written with only one write to the RAM. This system allows transfer from VRAM to VRAM 4 bytes at a time.

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

;==============
;
; Set EGA mode to read/write from latch
;
;==============

cli                         ;interrupts disabled
mov dx,GC_INDEX             ;mode 1, each memory plane is
mov ax,GC_MODE+256*1        ;written with the content of
out dx,ax                   ;the latches only

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

sti                         ;interrupts enabled
```

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

4.11.3 Wrap around the EGA Memory

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

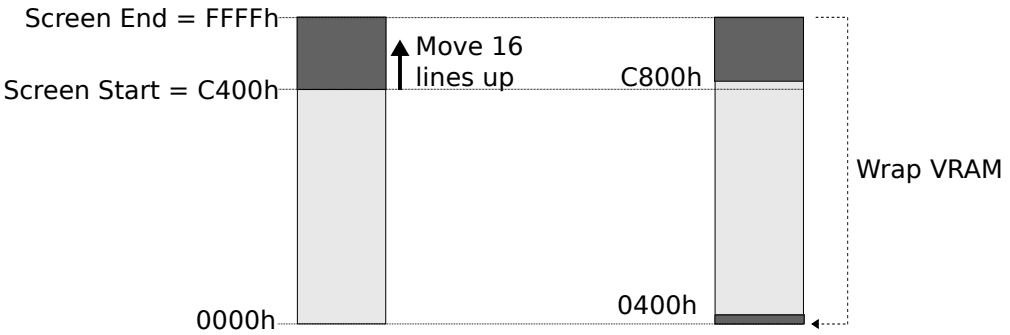
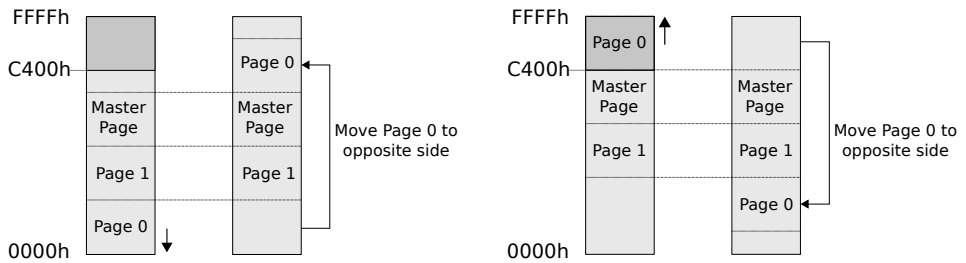


Figure 4.33: Wrap virtual screen around the EGA memory

There was however an issue with the introduction of Super VGA cards, which had typically more than 256kB RAM[3]. This resulted in crippled backwards compatibility and the wrapping around `0xFFFF` did not work anymore on these cards.

Luckily there was a way to resolve this issue. As you can see in Figure 3.21 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 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 3.34.



(a) Virtual pages moved down ***(b)*** Virtual pages moved up

Figure 4.34: Move screen to opposite end of VRAM buffer

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

[3]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 >256kB RAM, which from a hardware constraint resulted in 512kB.

latches as explain in section 3.11.2.

```
#define SCREENSPACE   (SCREENWIDTH*240)
#define FREEEGAMEM    (0x100001-3l*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);
  }
}
```

4.11.4 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, add the respective column or row in the Master view in VRAM and flag the new tiles of column/row to be updated in the next refresh for both the screenpage and otherpage update list. Update the screenstart and tile buffer array pointers as well.
3. Refresh the buffer view by scanning all tiles. If a tile needs to be updated, copy the tile from the master view to the buffer view in VRAM.

4. Remove the opposite row/column from the Master view.
5. Iterate through the sprite removal list and copy corresponding image block from master view to buffer view in VRAM.
6. Iterate through the sprite list and copy corresponding sprite image block from asset location in RAM to buffer view in VRAM
7. Switch the screen and buffer view by adjusting the Start Address and Pel Panning registers.

As you can see, step 2 until 4 are different and the rest of the steps are the same as Commander Keen 1-3. In the next screenshots we explain only steps that are different compared to Commander Keen 1-3. In the case below the screen is forced to scroll to the left.

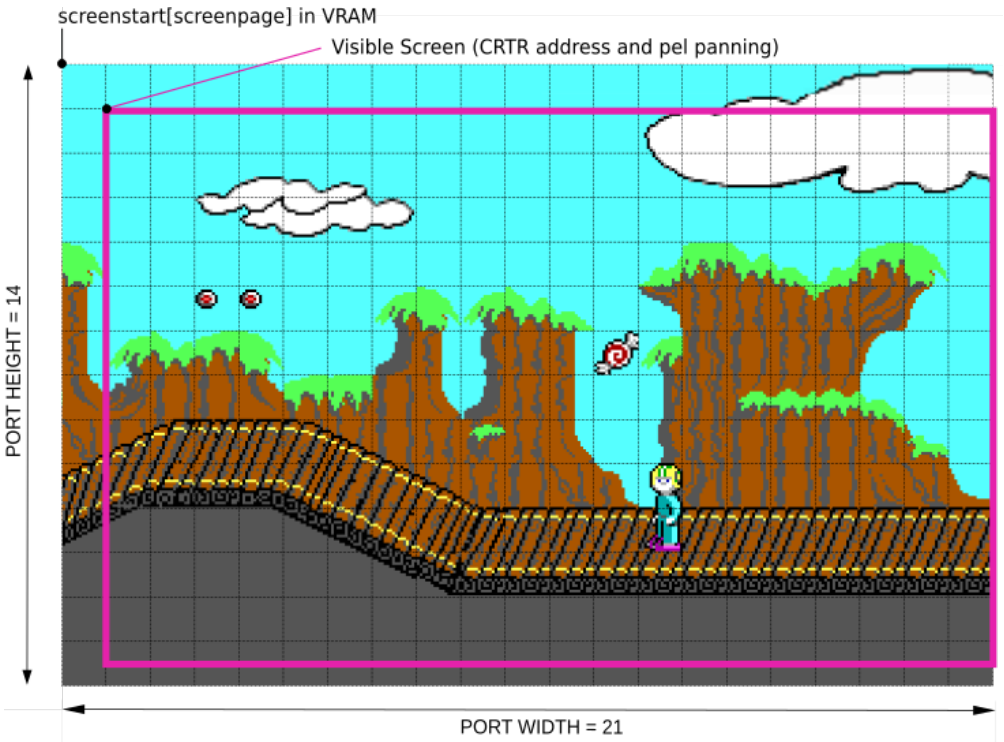


Figure 4.35: Step 1: Player moved to the left and forces the screen to scroll

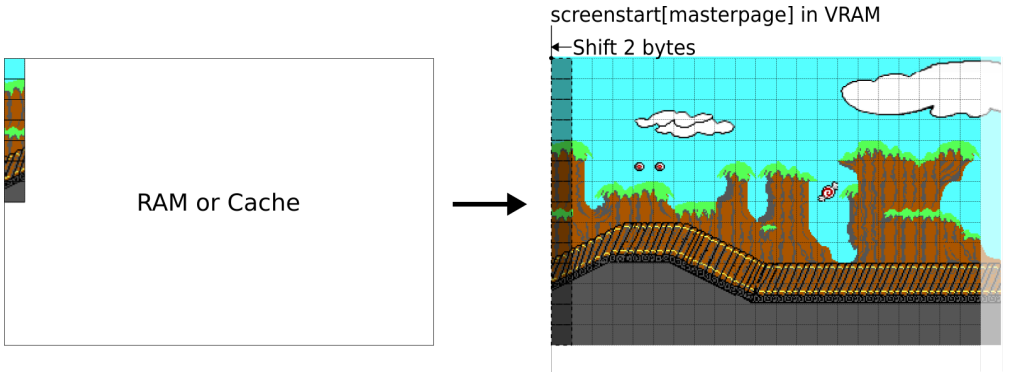


Figure 4.36: Step 2: Move screen start and add column to VRAM

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

In parallel decrease both tile buffer and 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'.

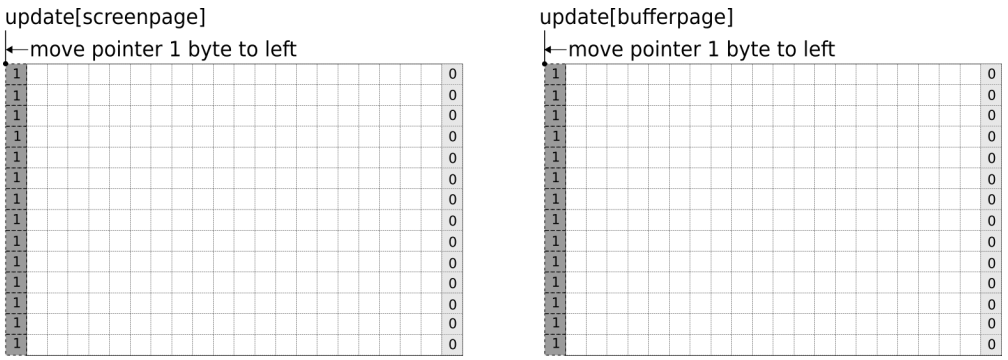


Figure 4.37: Mark new column in both tile buffer and display array.

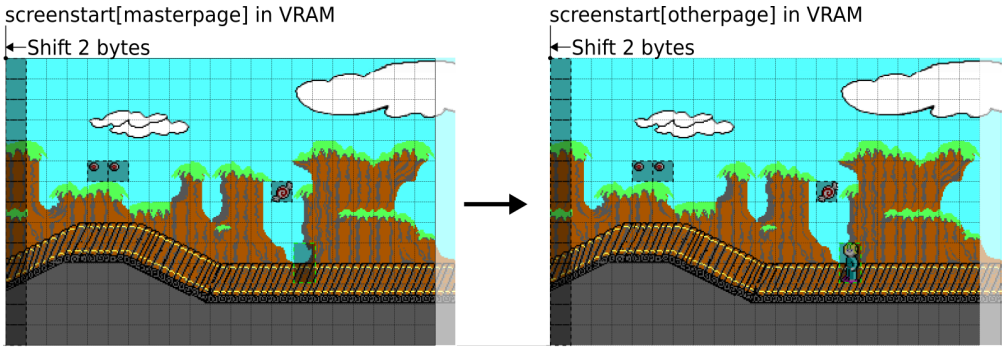


Figure 4.38: Step 2: Copy master to buffer screen and remove sprites

Just like before, we copy the animated tiles from asset 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 3.38 and Figure 3.39.

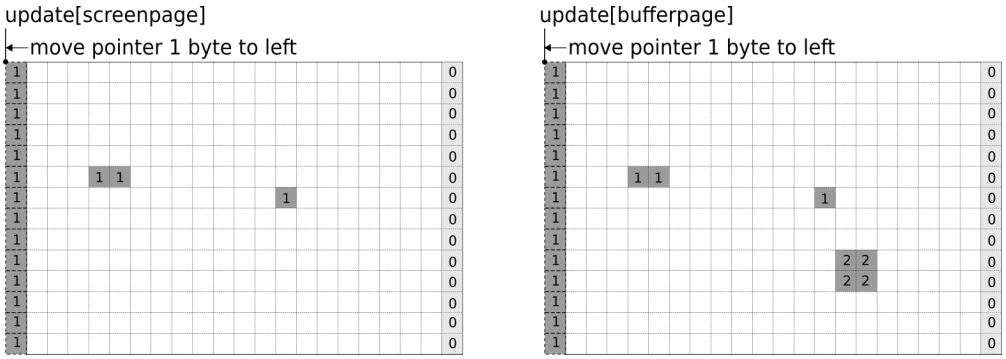


Figure 4.39: Tile buffer array with removed sprites marked with '2'.

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

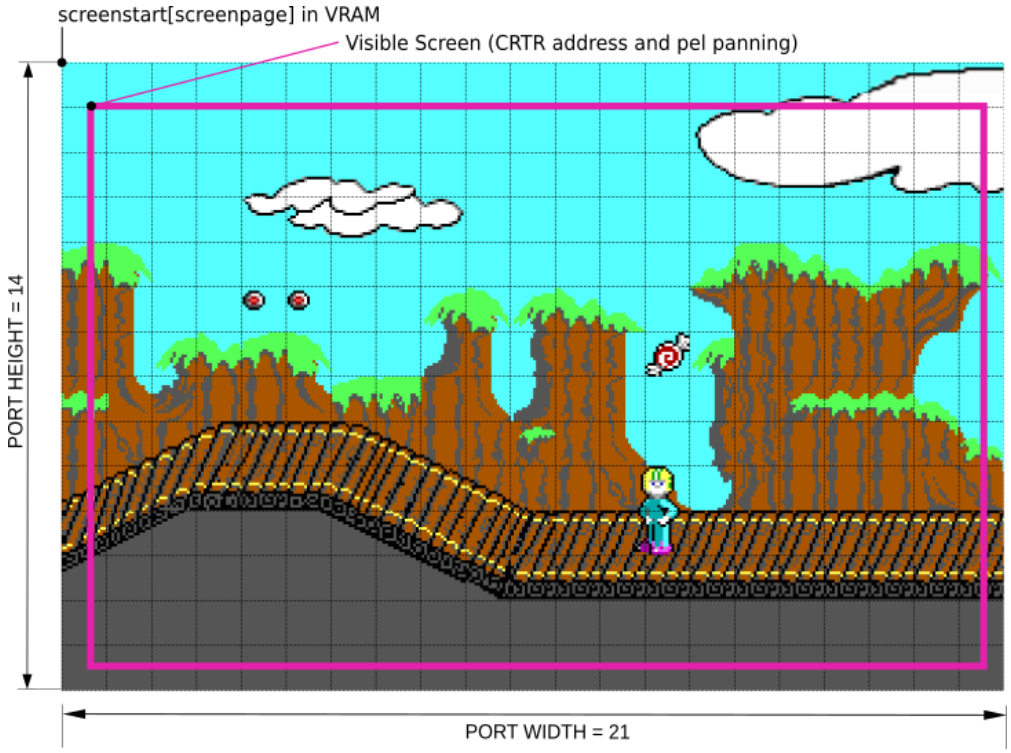


Figure 4.40: Step 6: Updated screen after swapping buffer and screen page

The remaining steps are also 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 replace only 6% of the screen!

We now can also see why the tile array buffer is 2 tiles wider on all sides than the view port. Let's take the situation where the screen needs to scroll to the top-left, meaning 1 tile left and 1 tile up as illustrated in Figure 3.41. 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).

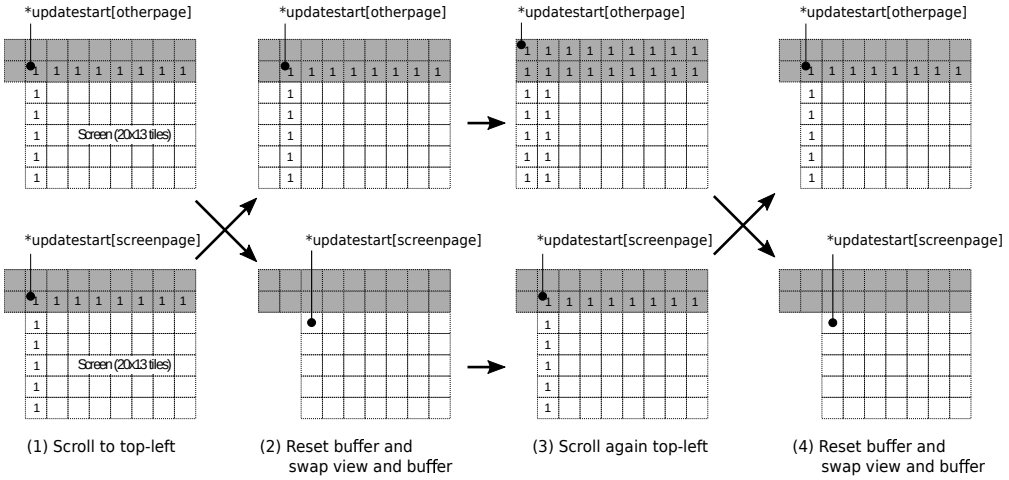


Figure 4.41: scroll to top-left tile

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

4.12 Refresh video screen

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

4.13 A.I.

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

4.14 Drawing Sprites

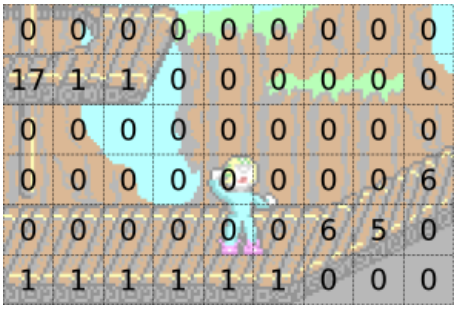
Explain how animation are performed. Should we somehow also explain sprites on other systems, like Nintendo, etc.?
Once the state of the actor is updated, it is time to render the actor on the screen. This is a two step operation.

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

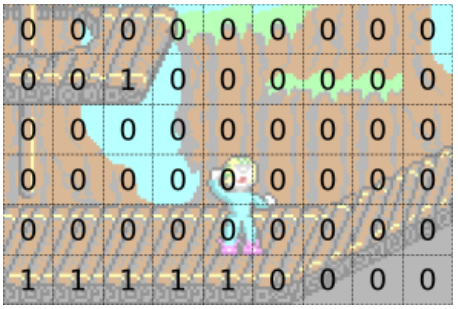
4.14.1 Visible actor determination

4.14.2 Clipping

Before drawing a sprite on the screen, the engine determines if the boundaries of a sprite are hitting a wall or floor. This is called clipping and ensures an actor doesn't fall through a floor or walks through a vertical wall. To define whether a tile is a wall or floor, a tile can be enriched with wall information. For each level map the tile info variable `tinf` contains a `NORTHWALL`, `SOUTHWALL`, `EASTWALL` and `WESTWALL` map as illustrated in Fig 3.42.



(a) Wall type map NORTHWALL



(b) Wall type map EASTWALL

Figure 4.42: Tile clipping map

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

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

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

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

The clipping for top and bottom is a bit more complex, as the engine also needs to take walking on slopes into account. After the sprite is clipped to the top or bottom of the wall tile, an offset can be applied to move a sprite up or down a slope.

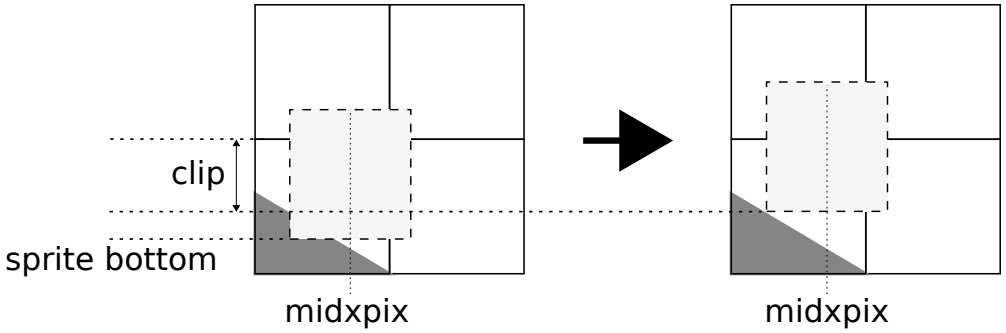


Figure 4.43: Clipping NORTHWALL

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

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

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

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

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

  map = (unsigned far *)mapsegs[1] +
    mapbwidthtable[oldtilebottom-1]/2 + ob->tilemidx;
  for (y=oldtilebottom-1 ; y<=ob->tilebottom ; y++,map+=
   mapwidth)
  {
    //Do we hit a NORTH wall
    if (wall = tinf[NORTHWALL+*map])
    {
      //offset from tile border clip
      clip = wallclip[wall&7][midxpix];
      //Clip bottom side actor to top side tile + offset-1
      move = ( (y<<G_T_SHIFT)+clip - 1) - ob->bottom;
      if (move<0 && move>=maxmove)
      {
        ob->hitnorth = wall;
        MoveObjVert (ob,move);
        return;
      }
    }
  }
```

4.15 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[4]. We now have two systems running in parallel.

[4] Interrupt Service Routine

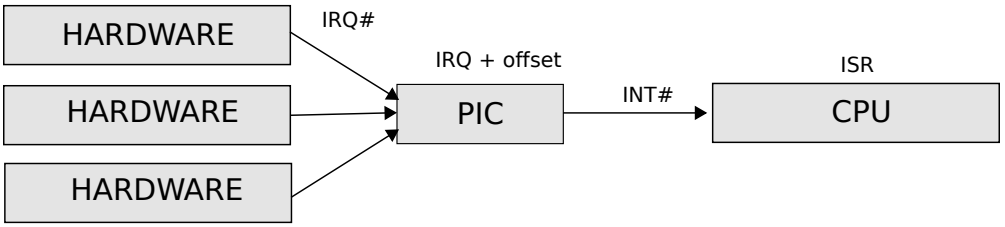


Figure 4.44: Hardware interrupts are translated to software interrupt via the PIC.

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 [5]. 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.

4.15.1 IRQs and ISRs

The IRQ and ISR system relies on two chips: the Intel 8254 which is a PIT[6] and the Intel 8259 which is a PIC[7]. The PIT features a crystal oscillating in square waves. 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"[8]. Counter #0 is connected to the PIC. When counter #0 hits zero it generates an IRQ[9] and sends it to the PIC.

[5] Interrupt Mask Register
[6] Programmable Interval Timer
[7] Programmable Interrupt Controller
[8] 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.
[9] Interrupt Request Line: Hardware lines over which devices can send interrupt signals to the CPU.

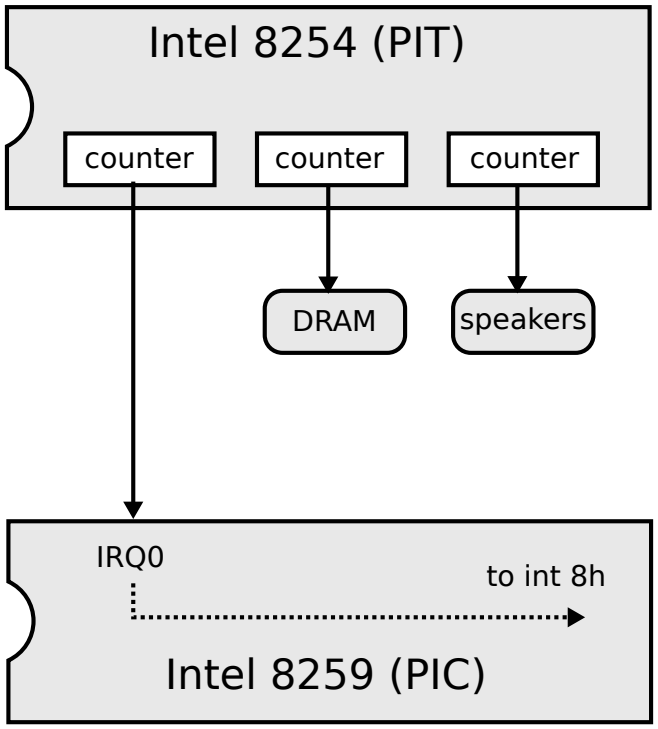


Figure 4.45: Interactions between PIT and PIC.

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).

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 4.46: 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.

4.15.2 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. 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 1991, oscillators were much cheaper and could have used any frequency but backward compatibility prevented this.

4.15.3 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
}
```

Note that `SDL_SetTimer0` is using a frequency of 1.192755MHz, instead of the PIT documented 1.193182MHz. Most likely at the time Keen Dreams was develop, the actual frequency of the PIT chip was not known by Jason Blochowiak and he derived the value based on the standard 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);
}
```

4.15.4 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[10] 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`.

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