GAME ENGINE BLACK BOOK

COMMANDER KEEN

v2023.08.04 by Bas Smits

Contents

Chapter 1

Introduction

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

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

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

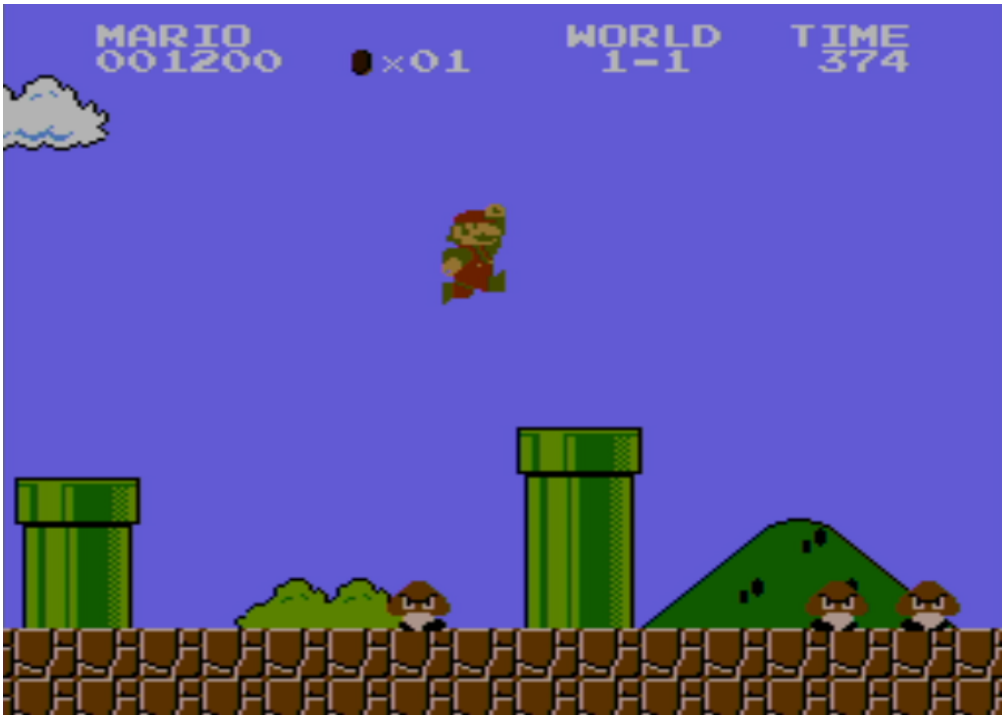


Figure 1.1: Super Mario Bros. on Nintendo Entertainment System

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

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

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

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

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

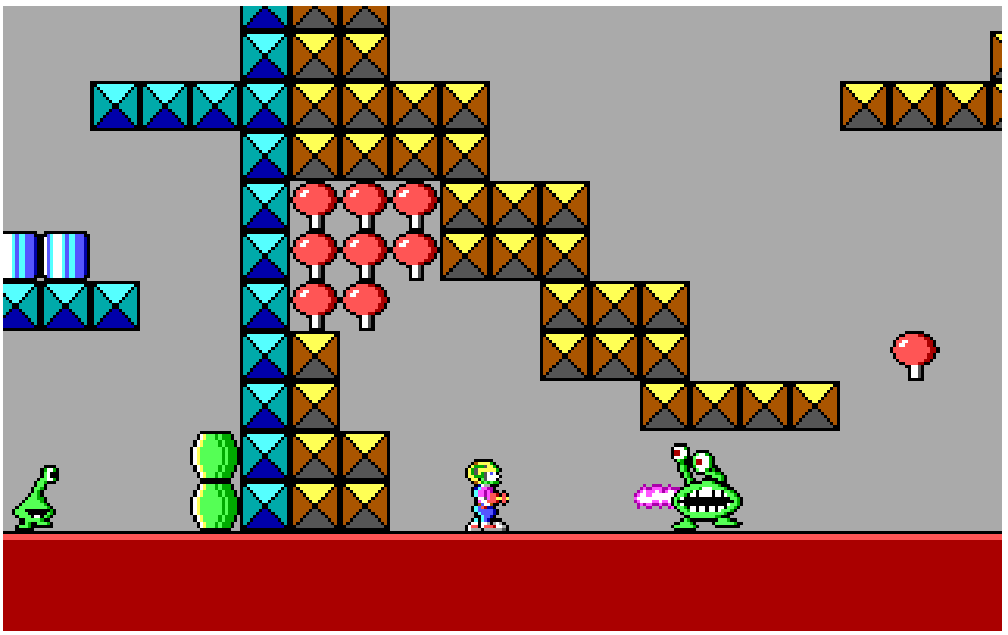


Figure 1.2: Commander Keen in Invasion of the Vorticons

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

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

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

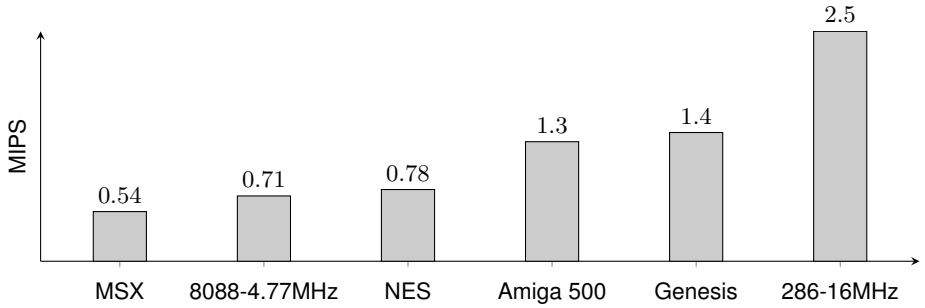


Figure 1.3: Consoles[2]vs PC, CPU comparison with MIPS[3][4].

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

- Chapter 2: The Hardware. The five components of a PC from 1990.
- Chapter 3: The tools and assets. Which tools are used for game development and how are assets created and structured.
- Chapter 4: The Software. The Commander Keen game engine.
- Chapter 5: Commander Keen game engine in CGA graphics.

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

[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.

advantages.

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

Figure 1.4: Commander Keen in Keen Dreams

Chapter 2

Hardware

Chapter 3

Hardware

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

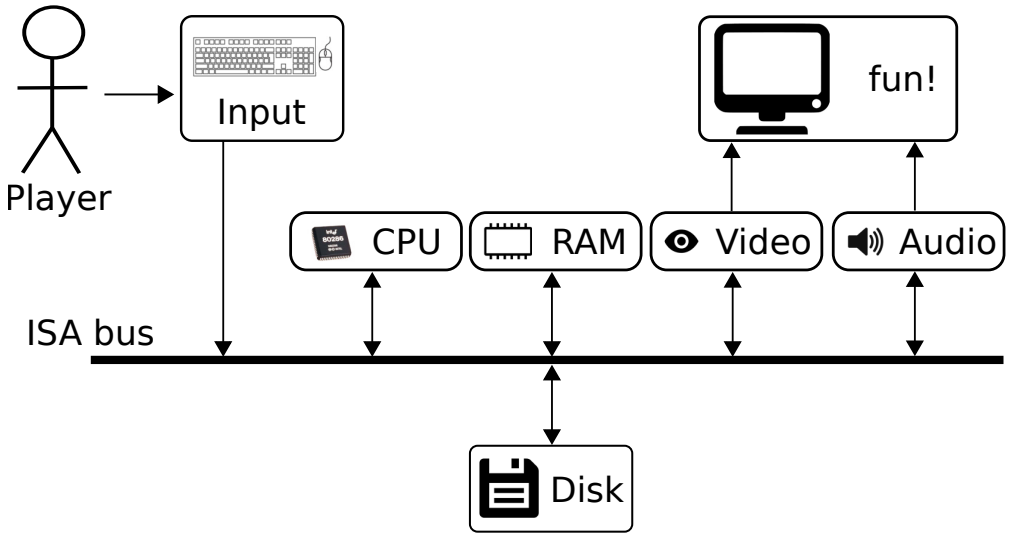


Figure 3.1: Hardware pipeline.

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

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

Figure 3.2: Component quality for a game engine.

3.1 CPU: Central Processing Unit

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

3.1.1 Overview

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

In 1982 Intel released the 80286 microchip. A typical 8088 chip was running at 4.77Mhz, where the 80286 was running at 8Mhz and later versions at 12-16Mhz. The 80286 was employed for the IBM PC/AT, introduced in 1984, and then widely used in most PC/AT compatible computers until the early 1990s. Commander Keen could run on a 8088, but an Intel 286 was recommended.

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

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

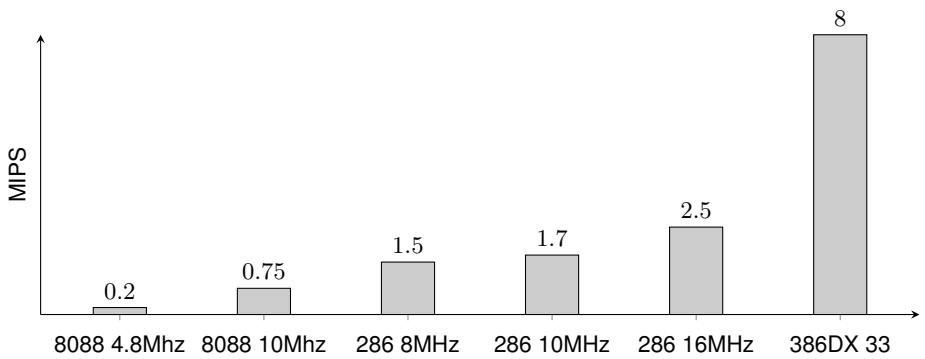


Figure 3.3: Comparison[3]of CPUs with MIPS

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

3.1.2 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.

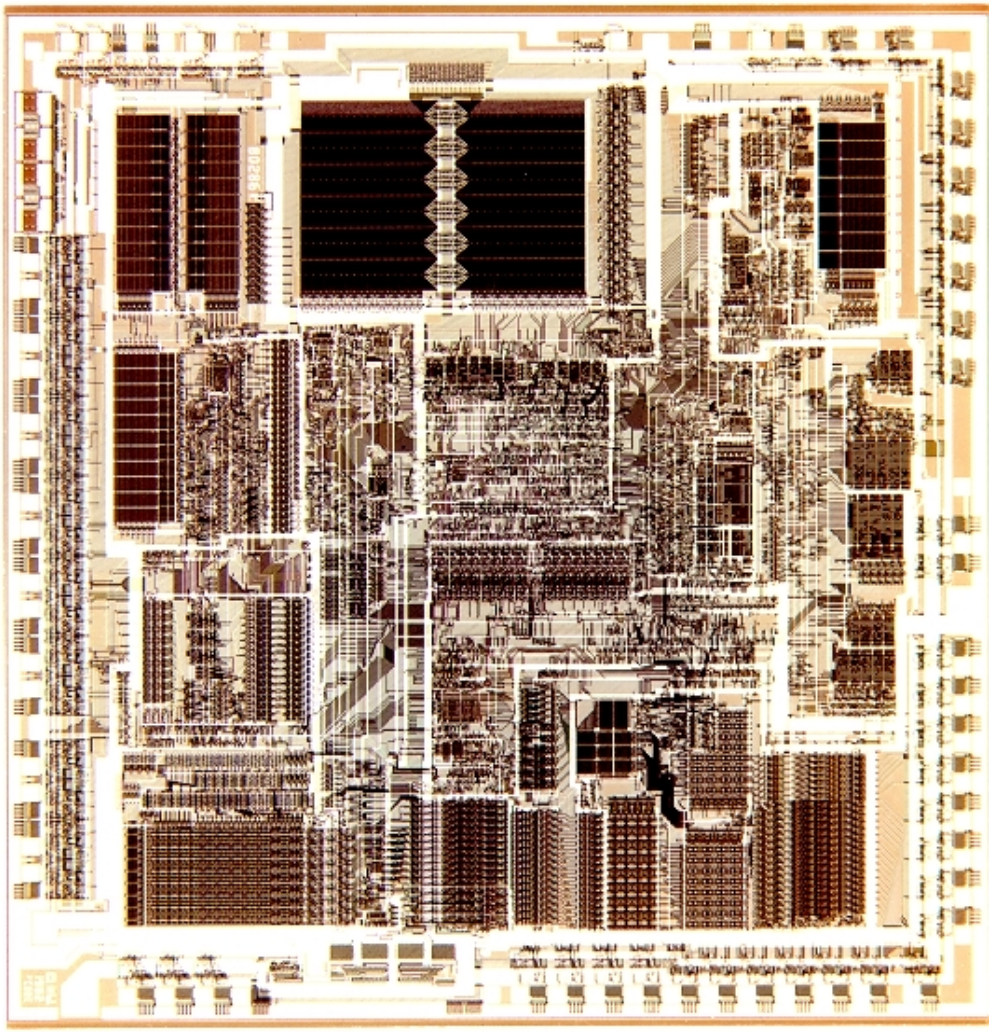
When IBM developed the AT, it selected the 286 as the basis for the new system because the chip provided compatibility with the 8088 used in the PC and the XT. Therefore, software written for those chips should run on the 286. The 286 chip is many times faster than the 8088 used in the XT, and at the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

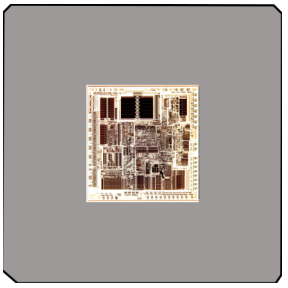
[3]Roy Longbottom's PC Benchmark Collection: http://www.roylongbottom.org.uk/mips.htm.

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

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




A80286-8
L9409047
©1982,1985



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

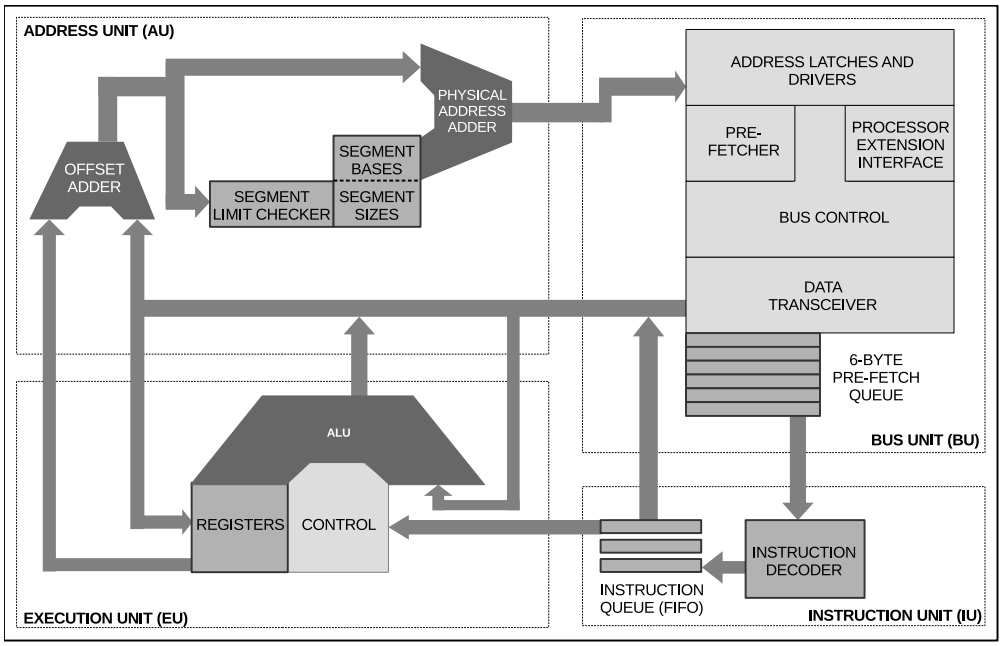
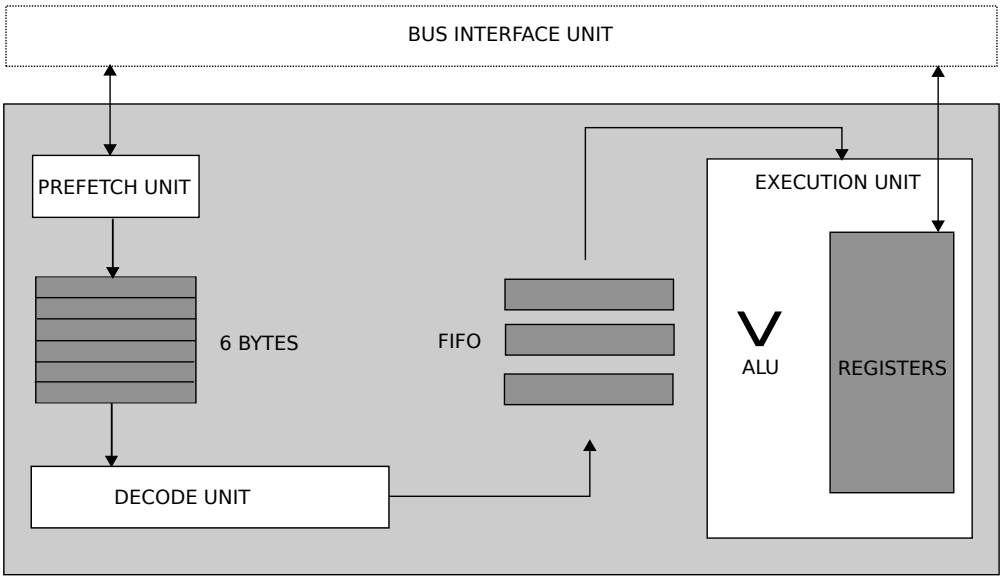


Figure 3.4: Internal block diagram of the 80286 processor

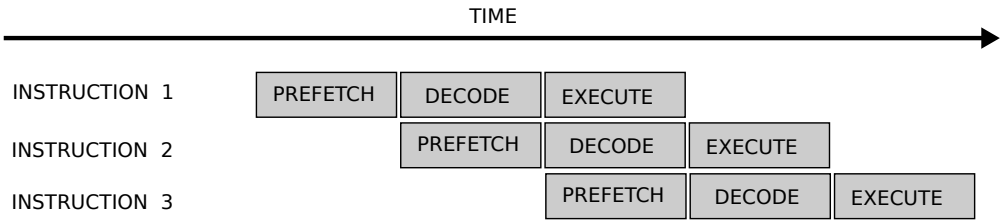
The four functional units can be described by

- **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.



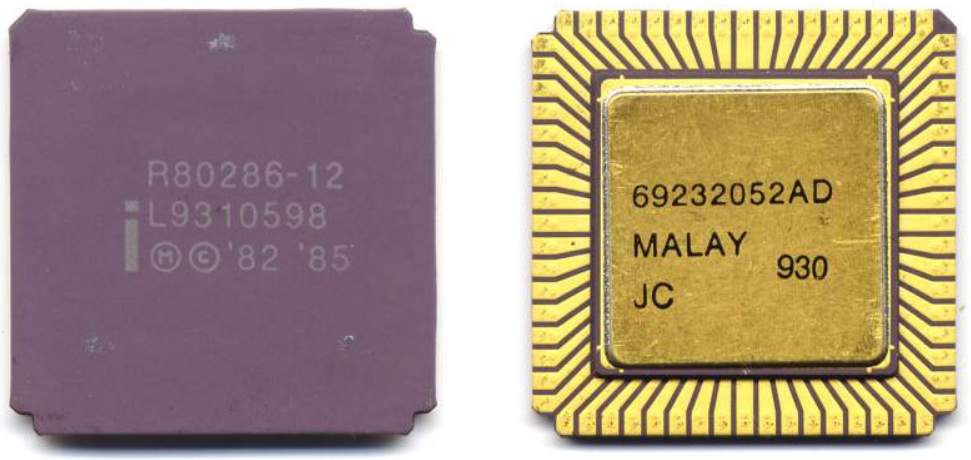


Figure 3.5: The Intel 286, 10mm by 10mm packing 134,000 transistors

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

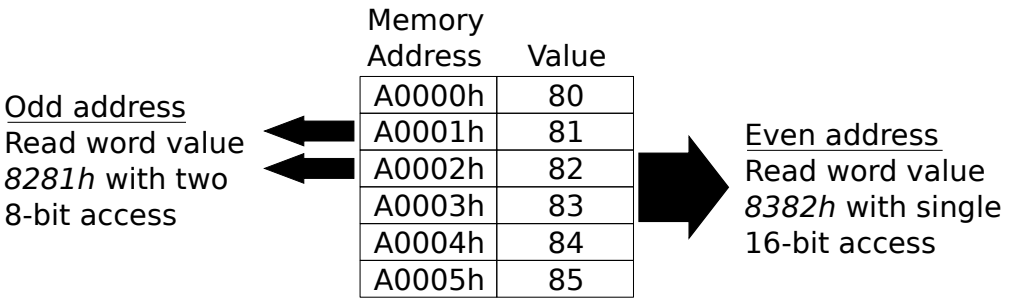
Despite its pipeline design, the 286 cannot do an operation in less than two cycles. Even a simple `ADD reg, reg` or `INC reg` takes two clocks. This is due to the absence of a SRAM on-chip cache and a slow decoding unit. Also have a look at multiplications which cost 24 cycles. So as a game developer 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 3.6: 286 instruction costs[4]

3.1.3 16-bit Data alignment

Compared to the Intel 8088, the 286 CPU contained a 16-bit external data bus where the 8088 only had a 8-bit bus. Thanks to its 16-bit bus, the 286 can access and write word-sized memory variables just as fast as byte-sized variables. There's a catch, however: That's only true for word-sized variables that start at even memory addresses. When the 286 is asked to perform a word-sized access starting at an odd memory address, it actually performs two separate accesses, each of which fetches 1 byte, just as the 8088 does for all word-sized accesses. In other words, the effective capacity of the 286's external data bus is halved when a word-sized access to an odd address is performed[5].



The way to deal with the data alignment cycle-eater is straightforward: Don't perform word-sized accesses to odd addresses on the 286 if you can help it. This is not an issue for small memory operations, but it will harm performance when copying large memory blocks.

[4] Intel 80286 programmer's reference manual - 1987.
[5] See Michael Abrash's Graphics Programming Black Book Special Edition, chapter 11.

3.2 RAM

The first CPUs in the Intel x86 family were designed in 1976. It was introduced at a time when the largest register in a CPU was only 16-bits long which meant it could address only 65,536 bytes (64 KiB[6]) of memory, directly. As example both the Apple II and the Commodore 64 both shipped with 64KiB which was enough to write and run amazing things.

But everyone was hungry for a way to run much larger programs. Rather than create a CPU with larger register sizes, the designers at Intel decided to keep the 16-bit registers for their new 8086 and 8088 CPU and added a different way to access more memory: They expanded the instruction set, so programs could tell the CPU to group two 16-bit registers together whenever they needed to refer to an absolute memory location beyond 64 KiB.

3.2.1 Memory addressing

If the designers had allowed the CPU to combine two registers into a high and low pair of 32-bits, it could have referenced up to 4 GiB of memory in a linear fashion. Keep in mind, however, this was at a time when many never dreamed we'd need a PC with more than 640 KiB of memory for user applications and data[7]. So, instead of dealing with whatever problems a linear addressing scheme of 32-bits would have produced, they created the Segment:Offset scheme which allows a CPU to effectively address about 1 MiB[8] of memory. The Segment:Offset schema combines two 16-bit registers, one designating a segment and the other an offset within that segment.

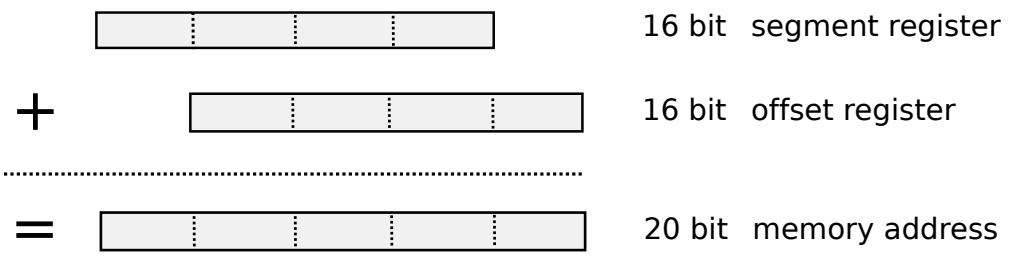


Figure 3.7: How registers are combined to address memory.

[6]This book uses IEC notation where KiB is 2^{10} and KB is 10^3.

[7]We've often heard that Bill Gates said something to the effect: "640K of memory should be enough for anyone.". Though many of us no longer believe he ever said those exact words, he did, however, during a video interview with David Allison in 1993 for the National Museum of American History, Smithsonian Institution, say: "I laid out memory so the bottom 640KiB was general purpose RAM and the upper 384KiB I reserved for video and ROM, and things like that."

[8]This book uses IEC notation where MiB is 2^{20} and MB is 10^6.

To support this architecture, the CPU keeps track of four different segments. Each of these segments has its own purpose and segment register:

- CS segment register, where the machine instructions resides.
- DS segment register, where the data resides.
- SS segment register, where the stack resides.
- ES segment register, which is used as extra data segment.

In C-language a memory address can be accessed directly using pointers. A pointer is a variable that stores the memory address of another variable as its value. There are two kinds of pointers: `near` and `far`.

A `near` pointer refers to a function or data object that is within the default segment. It is 16 bits long and contains an offset into the current DS data segment if it's a data pointer, or into the current CS code segment if it's a function pointer. A `far` pointer could refer to a function or data that is in a different segment than the current, default segment. It is 32 bits long and contains a segment and offset, which identifies the location where the code or data is stored.

Accessing code or data with a `near` reference is much quicker than accessing it with a `far` pointer. When you use a `far` reference, your program must first find the segment and then find the code or data within that segment. When you use a `near` reference, your program only needs to find the code or data via the offset. For a faster program, one should make as many `near` references as possible. The downside of using `near` pointers only is that it cannot handle more than 64KiB of memory for the program or data.

Note that a `far` pointer only increments the offset, not the segment. If you iterate on a data array larger than 64KiB there will be no overflow handling, so you will be able to handle only 64KiB of memory.

```
char far *p = (char far *)0xA000FFFF;
p++;
printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
```

Will output:

```
A000:0000
```

You could use yet another type of pointer named `huge` to make pointer arithmetic work beyond 64KiB.

```
char huge *p = (char huge *)0xA000FFFF;
p++;
printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
```

Will output the address:

```
B000:0000
```

The `huge` pointer is based on the absolute (or lineair) 20-bit memory location and Segment:Offset normalized address. The absolute memory address can be calculated by

```
Absolute memory address = (Segment * 0x10h) + Offset
```

For example the absolute address of `A000:002F` is `A0000h` + `002Fh` = `A002Fh`. By confining the offset to just the hexadecimal values `0h` through `Fh`, we have a unique way to reference all Segment:Offset memory pair locations. To convert an arbitrary Segment:Offset pair into a normalized address is a two-step process that's quite easy for an assembly programmer:

1. Convert the Segment:Offset pair into a single absolute address.
2. Then simply insert the colon (:) between the last two hex digits and add three zeros before the last hex digit.

For example, the normalized address for `A000:002F` is `A002:000F`. A `huge` pointer is normalized when pointer arithmetic is performed on it.

```
# include <stdio .h>
# include <dos.h>
int main (void){
  char huge *p = MK_FP (0xA000 , 0xFFFF );
  p--;
  p++;
  printf("%04X:%04X\n", FP_SEG(p), FP_OFF(p));
}
```

Will output:

```
AFFF:000F
```

Trivia : Since the normalized form will always have three leading zero bytes in its offset, programmers often write it with just the digit that counts: `AFFF:F`

A `huge` reference is much slower than the `far` reference as it comes with additional overhead to update the segment and address normalization after every arithmetic manipulation. So, most programmers avoided the `huge` pointer, unless really needed.

3.2.2 80286 Real and Protected mode

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[9]. 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. It would have worked out if the operating system had been able to run in protected mode. However, in the name of backward compatibility, Microsoft's DOS could only handle real mode which effectively locked developers into 16-bit programming[10].

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. The memory layout for the real mode is as follows:

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

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

[9] Memory Management Unit

[10] 16-bit programming and memory managers were covered in Game Engine Black Book: Wolfenstein 3D.

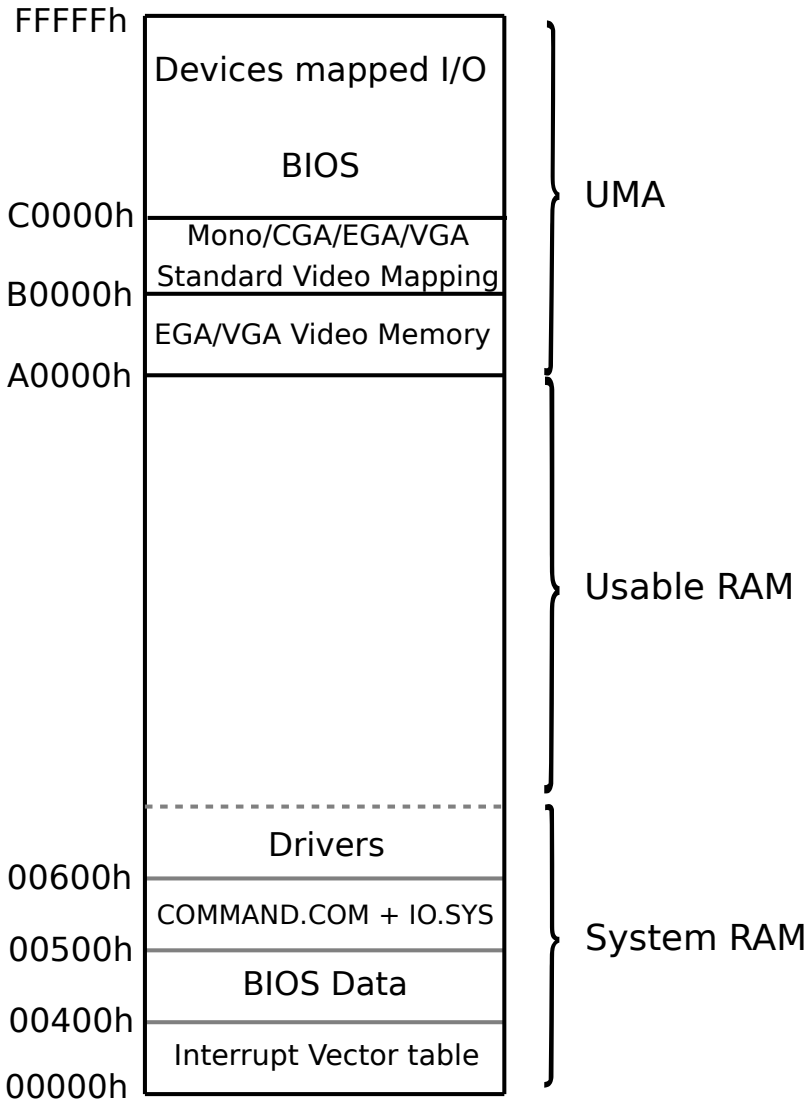


Figure 3.8: First 1MiB of RAM layout.

3.2.3 Real mode: Memory models

When a program is compiled and executed, the operating system allocates a chunk of memory to the program. This memory is divided into different segments as shown below

- The code section stores the program executable. When you compile a C program, the compiler converts your code to assembly instructions that the CPU will execute.
- The data section stores (un)initialized global and static variables.
- The heap section is memory that you can dynamically reserve from calling `malloc()` function. The heap typically grows upwards, which means it grows toward larger memory addresses.
- The stack section is memory for local variables and data inside your functions. The stack grows downwards (i.e. grow toward lower memory addresses).

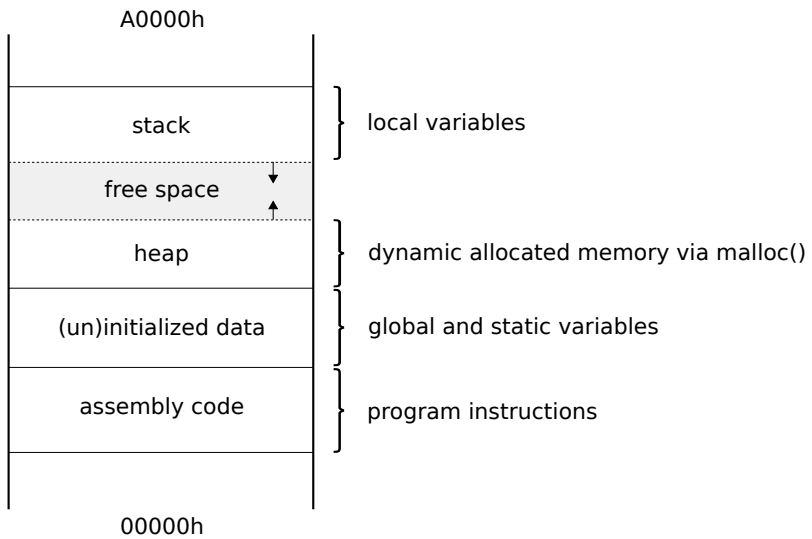


Figure 3.9: Memory layout in x86 real mode.

The x86 real mode provided different memory segment layouts which are called *memory models*. Each memory model controls how the segments are organized, as well as defining the default pointer type for functions and data to `near` or `far`. A near pointer is automatically associated with either CS segment register for assembly code or DS segment register for data. In total six different memory models[11] are defined, making trade-offs between minimum system requirements, maximizing code efficiency, and gaining access to every available memory location.

[11] See Borland C++ 3.1 Programmer's guide, section DOS Memory management.

Model	Default pointer type		Size		Definition
	Code	Data	Code	Data	
Tiny	near	near	<64KiB		CS=DS=SS
Small	near	near	<64KiB	<64KiB	DS=SS
Medium	far	near	>64KiB	<64KiB	DS=SS, multiple code segments
Compact	near	far	<64KiB	>64KiB	single code segment, multiple data segments
Large	far	far	>64KiB	>64KiB	multiple code and data segments
Huge	far	far	>64KiB	>64KiB	multiple code and data segments, global data >64KiB

In the tiny memory model, all three segment registers, CS, DS, and SS, start at the same memory location. The first part of the memory is used for all code instructions, followed by data section. The heap is starting directly after the data section. Finally, the stack is starting at the opposite side of the segment and is growing downwards. Both the heap and stack can grow or shrink dynamically during program execution. If both the heap and stack continue to grow, they may eventually collide, resulting in a system or application crash.

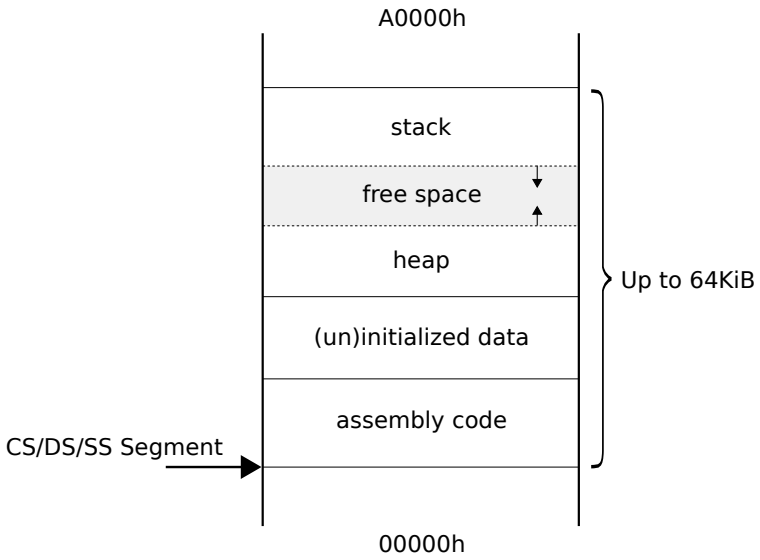


Figure 3.10: Tiny memory model.

All function and data pointers are accessed via near offset references, using the CS and DS segment registers respectively. Since only near references are used, program execution is very efficient. However, there is a downside: everything resides in one segment of a maximum of 64KiB, similar to programming on a 16-bit computer.

Trivia : The tiny memory model was required by programs that ended with the .COM extension, and it existed for backward compatibility with CP/M. CP/M ran on the 8080 processor which supported a maximum of 64KB of memory.

The small memory model separates instructions from data, each having their own 64KiB segment. The code segment has a maximum size of 64KiB for instructions, while the global data, stack, and heap share a separate 64KiB segment. Code execution remains efficient as all functions and data pointers use near references.

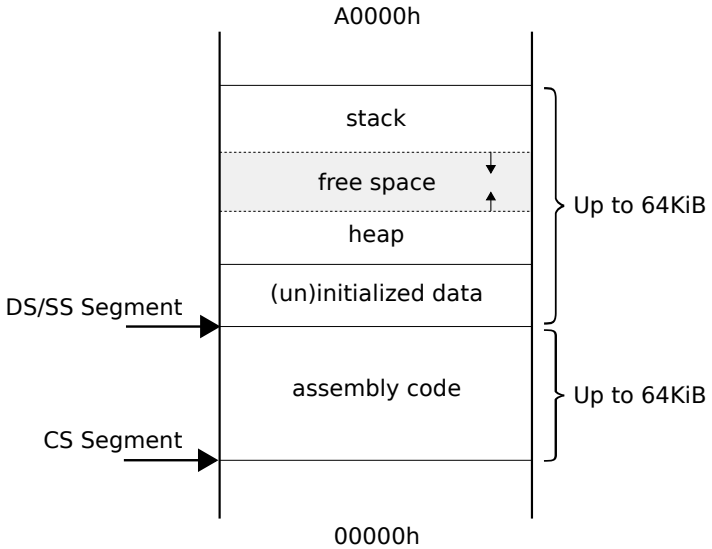


Figure 3.11: Small memory model, code and data each have 64KiB.

The medium model is useful for programs that have a lot of code but not a lot of data. Many computer games fell into this category, because they had a lot of game logic, but not a lot of game state data. Commander Keen was no exception. In the medium memory model each C-source file has its own code segment, allowing up to 64 KiB for each file. A table with all code segment references is maintained, and the CS register points to only one segment at a time. There is no default CS segment, and functions are reached using a far reference pointer.

Trivia : When you compile a source file, the resulting code for that file cannot be greater

than 64KiB, since it must all fit inside one code segment. If the source file is too big to fit into one 64KiB code segment, the programmer must break it up into different source files and compile each file separately.

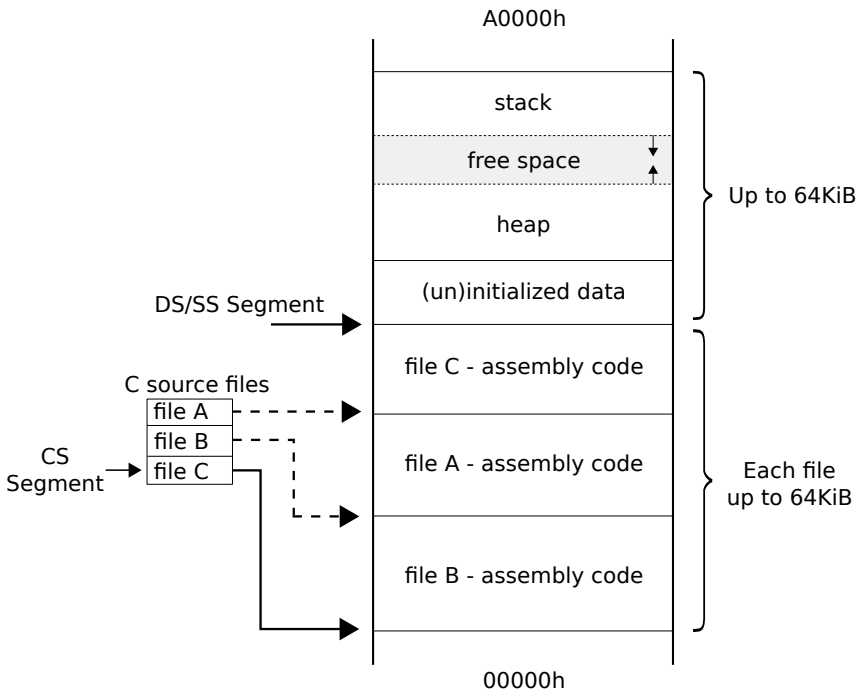


Figure 3.12: Medium memory model, code can be larger than 64KiB.

The total data segment size remains 64KiB. For most games, this is enough for global variables and the stack. However, there is an issue with the size of the heap, which is used to load game data such as graphic assets and levels. One option to increase the data segment size is to use the large memory model, but this comes at the cost of slower data access as it is using far reference pointers for all data access.

There is, luckily, a way to access additional memory using the medium model. A large amount of unallocated memory is available between the heap and the high address `A0000h`. This part of the memory is called the *far heap* and can be allocated using the `farmalloc()` function with far or huge pointer reference. This kind of programming, where you combine a standard memory model with your own `near` and `far` pointer allocation, is called *mixed model programming*. You keep the benefits from the medium memory model, using near pointer references for global data and the stack, and have sufficient memory for dynamic allocation.

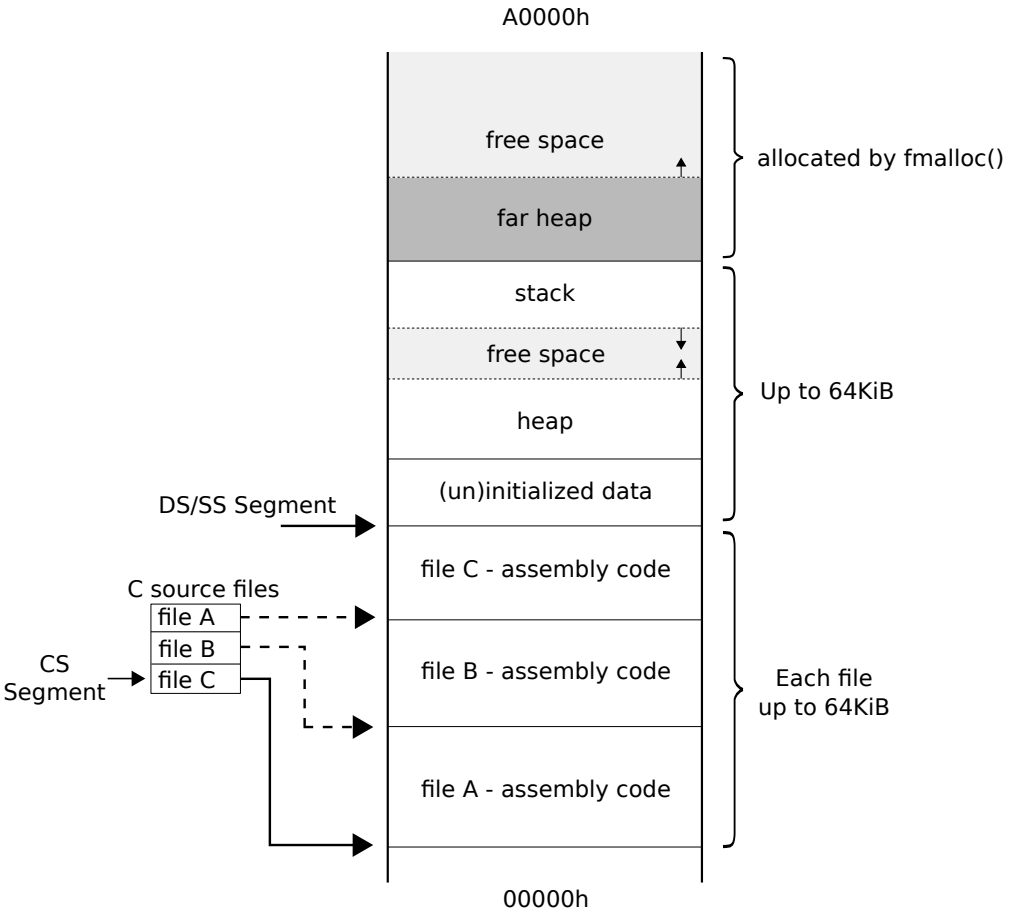


Figure 3.13: Far heap memory.

The opposite of the medium model is the compact model. Here the total function code cannot exceed 64KiB, but there is more space for global data, stack and heap. The large model can deal with function code and data larger than 64KiB, but is slow in execution as only far pointers are used for both code and data. Borland C++ limits the size of all global data to 64KiB. The huge memory model sets aside that limit, allowing global data to occupy more than 64KiB[12].

[12]Also here the same limitation as before: If the source file is too big to fit into one 64KiB data segment, the programmer must break it up into different source files and compile each file separately.

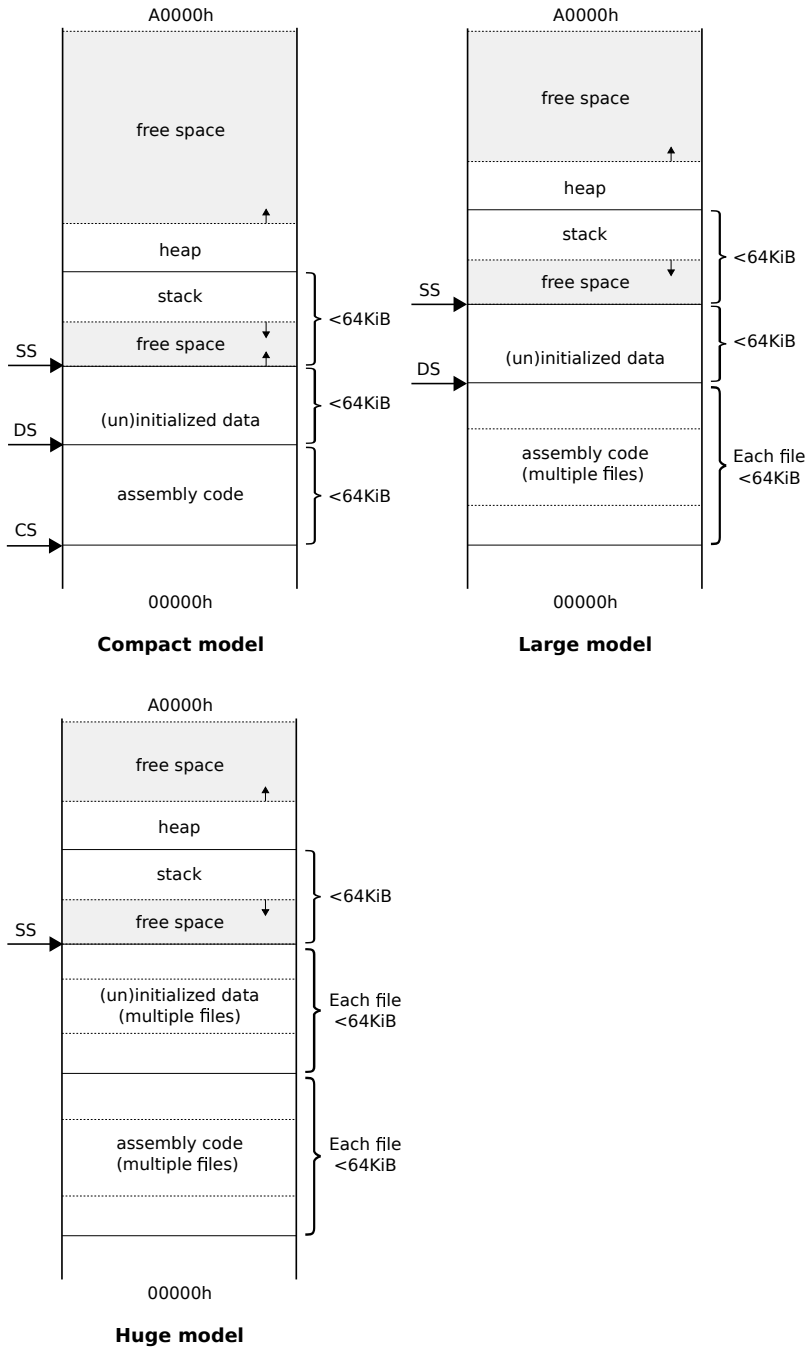


Figure 3.14: Compact, Large and Huge memory model.

Trivia : Although the name implies differently, the huge memory model in Borland C++ is still limited to segments up to 64KiB using far pointers, and not using huge pointers. The Watcom compiler, which gained popularity in the 90's, was able to break the 64KiB segment barrier using the huge pointer reference for the huge memory model[13].

[13] See https://open-watcom.github.io/open-watcom-v2-wikidocs/clr.html

3.3 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.

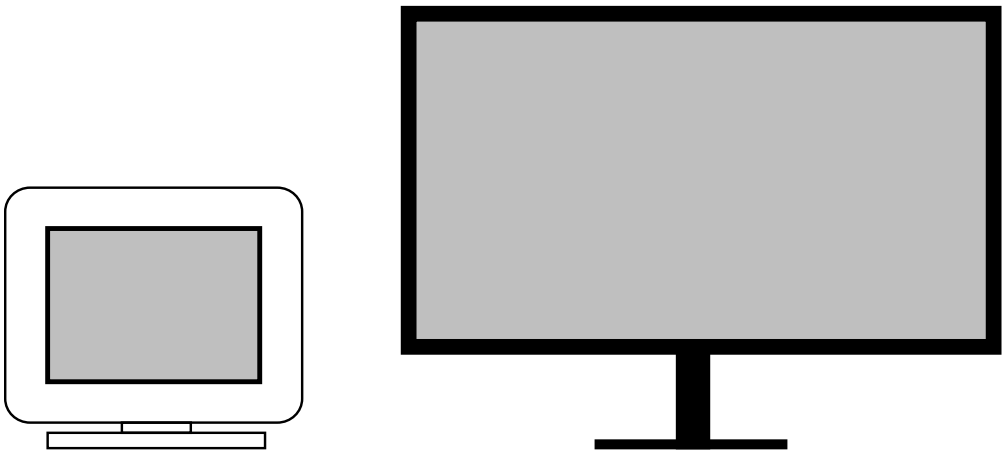


Figure 3.15: CRT (left) vs LCD (right)

To give you an idea of the size and resolution, figure 3.15 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).

3.3.1 CRT Monitor

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

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

known as horizontal blanking.

The area immediately surrounding the display area, in which the beam may be turned on during the retrace interval, is called the overscan (or border). The active display area is the portion of the screen that contains characters and/or graphics. These components of the scan are shown in simplified form in Figure 3.16.

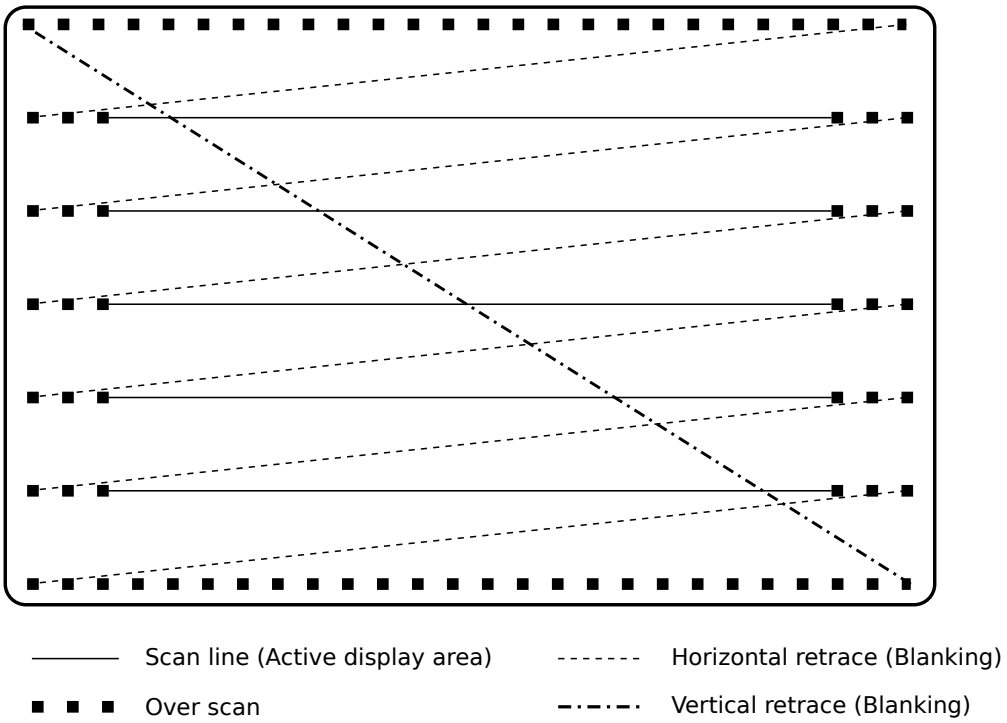


Figure 3.16: Simplified CRT monitor scan.

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

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

3.3.2 History of Video Adapters

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

Name	Year Released	Memory	Max Resolution
MDA (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[14]
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 3.17: 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.

3.3.3 Introduction of EGA Video Card

IBM introduced the Enhanced Graphics Adapter (EGA) in 1985 as the successor of CGA. The standard card was shipped with only 64KiB video memory, but it had the option to expand the memory using the onboard graphics memory expansion card. Figure 3.19 shows the original IBM EGA card, a clunky beast full of discrete components. The memory consists out of TMS4416 RAM, a common memory chip for (home) computers around that period. Each chip contains 16KiB of 4-bits memory, so one needs two chips to end up having 16KiB of 8-bit memory and eight chips for 64KiB of 8-bit memory.

Trivia : Texas Instruments introduced the first 16KiB by 4-bits as TMS4416 in 1980[15]. Still it took until 1983-84 until they became widely available and lower priced than four TMS4116 chips (16KiB by 1-bit). However, at that time 64 KiB RAM was the way to go for new designs. Computers with only 16 KiB as base memory - and that's where TMS4416 would have been a cost saver - were already on the way out.

[14] Text mode only.
[15] https://pdf1.alldatasheet.com/datasheet-pdf/view/103706/TI/TMS4416.html

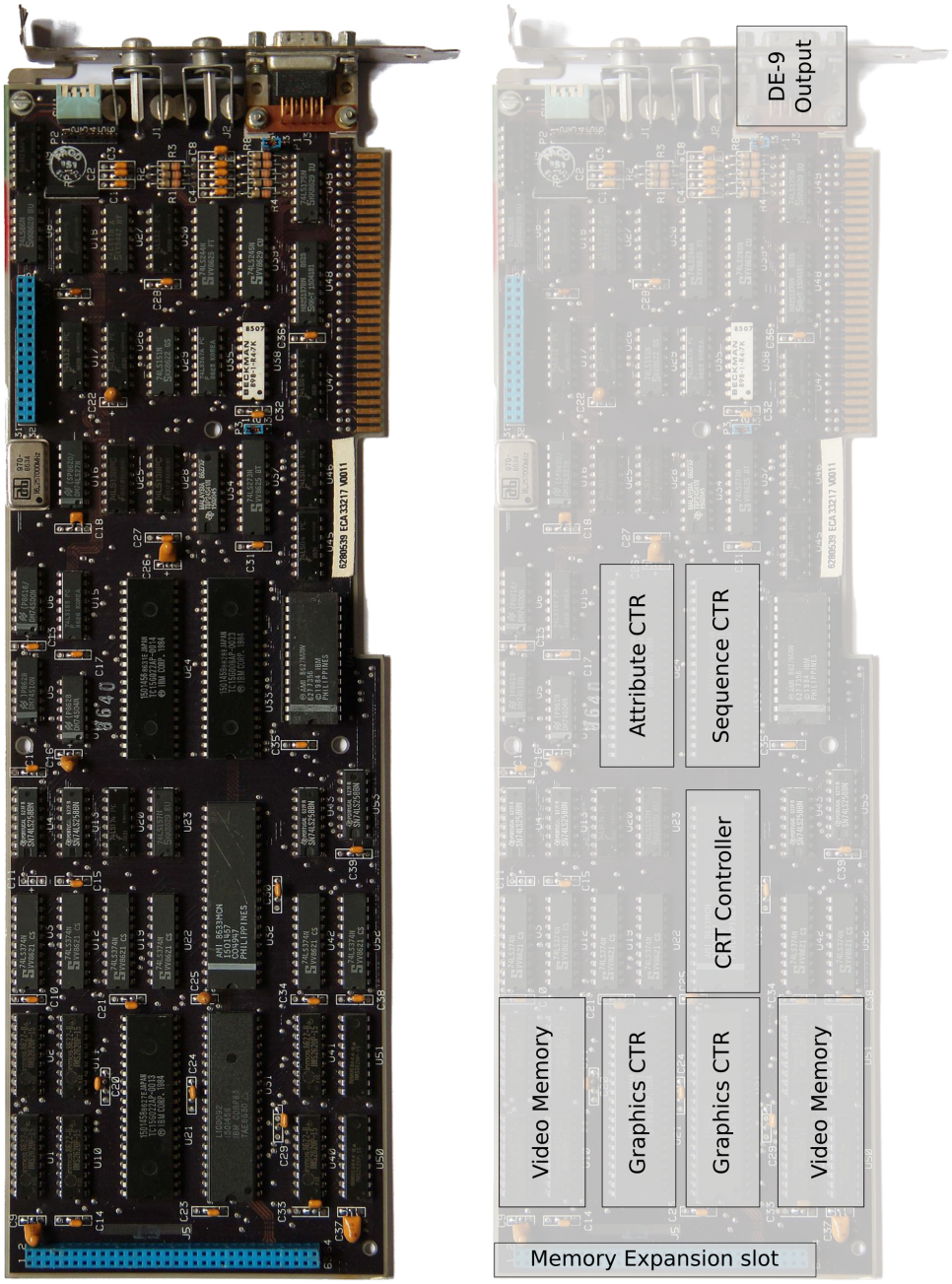


Figure 3.18: Original IBM EGA card

To add additional video memory to the IBM EGA card a Graphics Memory Expansion Card could be purchased. By default only the bottom row of memory was populated with chips, expanding the total EGA video memory to 128KiB. The expansion card provided DIP (dual in-line package) sockets for further memory expansion. Populating the DIP sockets with a Graphics Memory Module Kit adds two additional rows of 64KiB, bringing the EGA memory to its maximum of 256KiB.

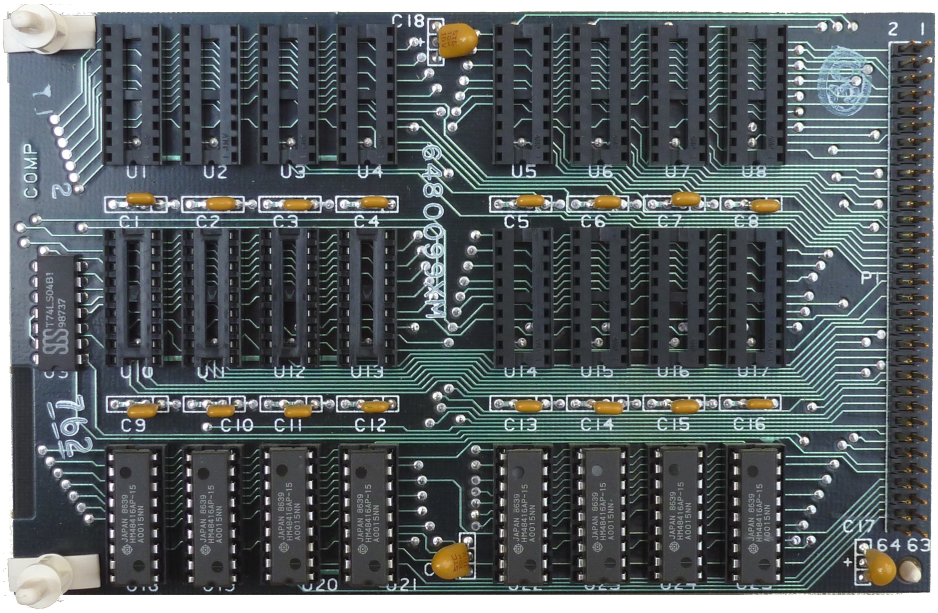
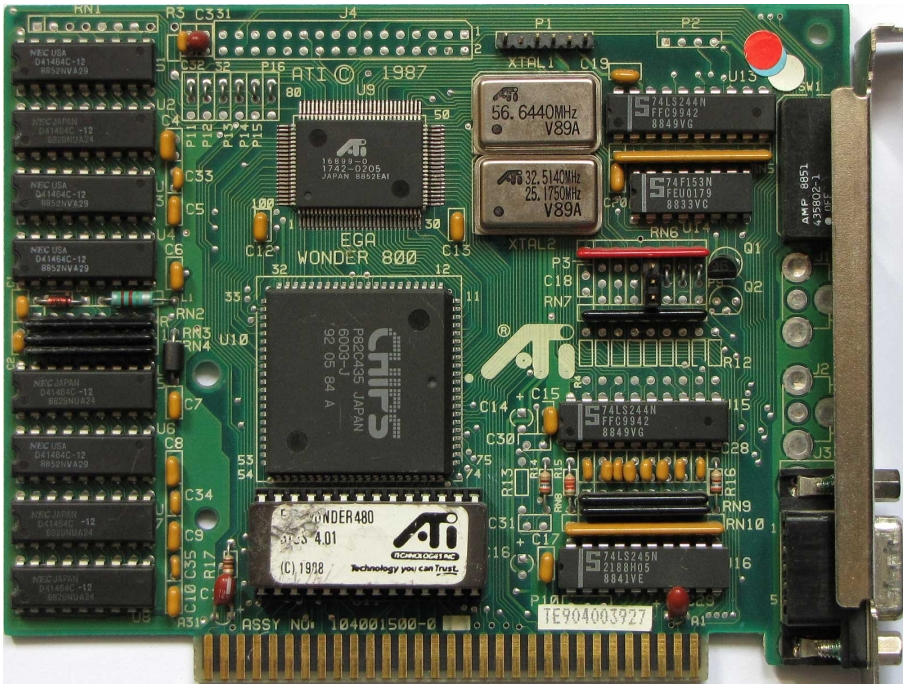


Figure 3.19: EGA Graphics Memory Expansion Card, bottom row populated with chips.

The EGA clones that started coming along in 1986-87 were based on integrated chipsets, and the vast majority of them came with the maximum of 256KiB on board. When Commander Keen came out, the headcount of EGA cards with less than 256KiB would've been practically negligible[16].

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.

[16]PC Tech Journal Oct 1986 (page 82-83) and PC Tech Journal Nov 1986 (page 148-149)


ATI © 1987
56.6440MHz
V89A
32.5140MHz
25.1750MHz
V89A
EGA
WONDER 800
(C) 1988
Technology you can Trust.
ASSY NO: 104001500-0

3.3.4 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 64KiB (rather than one bank of 256KiB).
- The CRT Controller and the Attribute Controller taking care of converting the palette-indexed framebuffer to RGB and then to digital TTL[17] signal for display

Trivia : In the 1980's integrated video DACs[18] 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[19] 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.

[17] Transistor Transistor Logic
[18] Digital to Analog Converter
[19] Computer Graphics: Principles and Practice 2nd Edition, page 168.

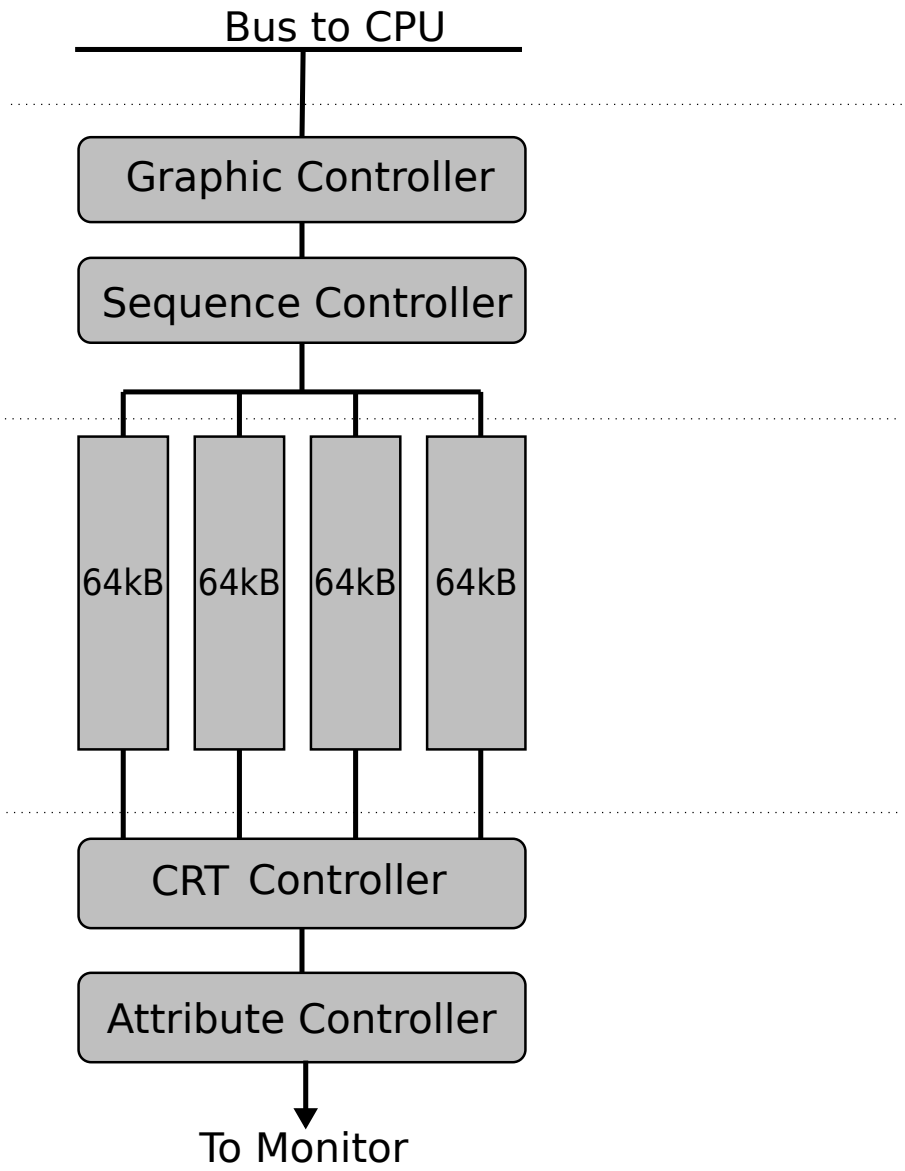


Figure 3.20: EGA Architecture.

3.3.5 EGA Planar Madness

Four memory banks grant enough throughput to reach high resolutions at 60Hz. This type of architecture is called "planar". Each plane is like a black-and-white image that stores

information about a single color. For EGA there are 4 planes, where combining one bit from each plane results in a color index. This color index is then via the attribute controller translated into a color to the screen. This layout is better explained with a drawing.

To write the color of the first pixel, a developer has to write the first bit of the first byte in plane 0, the second in plane 1, the third in plane 2 and the fourth in plane 3. The CRT Controller then reads 4 bytes at a time (one from each plane) and converts them via the Attribute Controller into 8 pixels on screen.

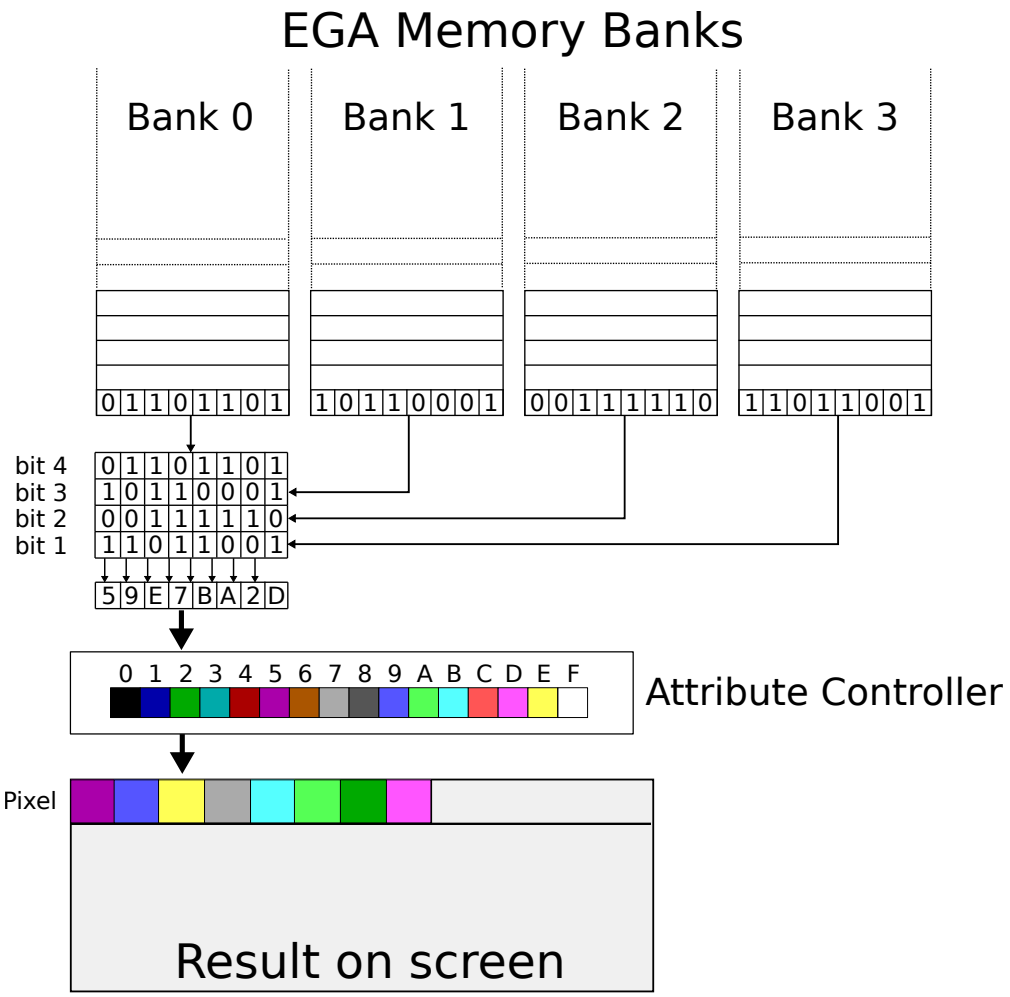


Figure 3.21: EGA mode 0Dh, how bank layout appears on screen.

3.3.6 EGA Modes

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 3.20, which described the architecture, was actually deceptively simplified. Figure 3.22 shows how IBM's reference documentation explained the EGA. The maze of wire showcases well the actual complexity of the system.

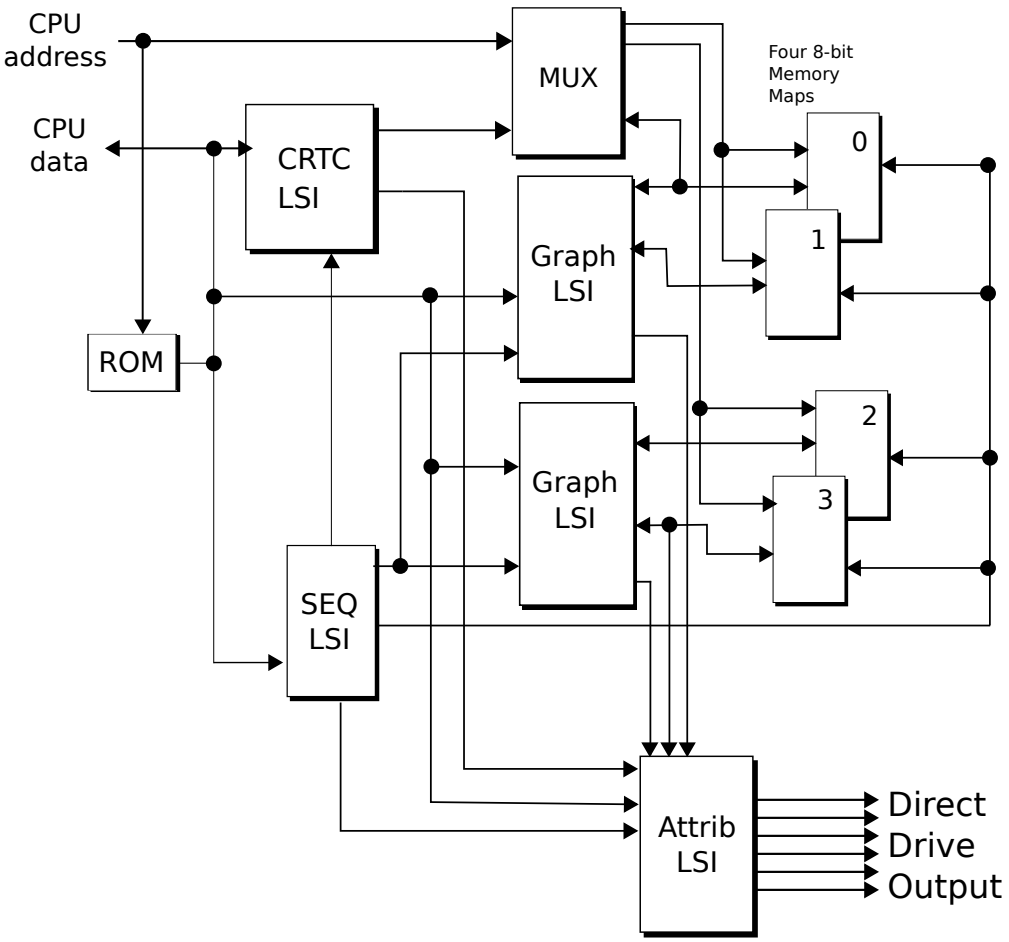


Figure 3.22: IBM's EGA Documentation.

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

Mode	Type	Format	Colors	RAM Mapping	Hz
00h	text	40x25	16 (monochrome)	B8000h	60
01h	text	40x25	16	B8000h	60
02h	text	80x25	16 (monochrome)	B8000h	60
03h	text	80x25	16	B8000h	60
04h	CGA Graphics	320x200	4	B8000h	60
05h	CGA Graphics	320x200	4 (monochrome)	B8000h	60
06h	CGA Graphics	640x200	2	B8000h	60
07h	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 3.23: EGA Modes available.

Trivia : The Modes `08h-0Ah` are reserved for PCjr (or Tandy Graphics Adapter) graphics modes, which offered 160x200 with 16 colors, 320x200 with 16 colors and 640x200 with 4 colors. Modes `0Bh` and `0Ch` are reserved for internal EGA BIOS.

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.

3.3.7 EGA Programming: Memory Mapping

To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in figure 6.3. In mode `0Dh` for example, the VRAM is mapped from 0xA0000 to 0xAFFFF. One of the first questions to come to mind is "How can I access 256KiB of RAM with only 64KiB

of address space?" The answer is "bank switching" as summarized in figure 3.24. Write and Read operations are routed based on a mask register indicating which bank should be read or written to.

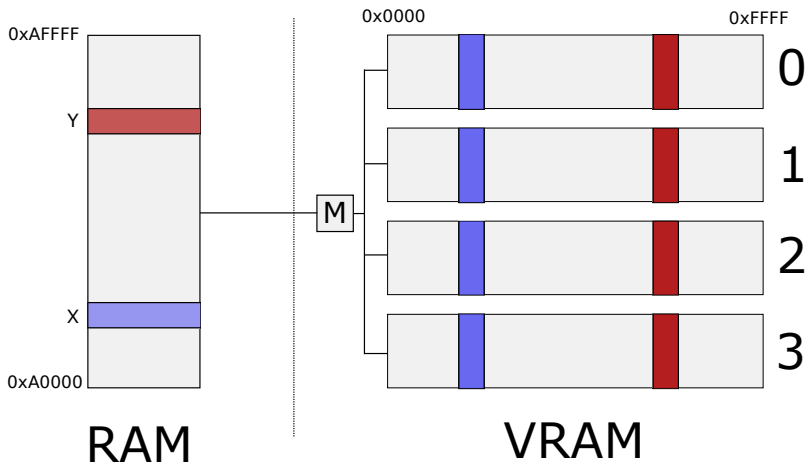


Figure 3.24: Mapping PC RAM to EGA VRAM banks.

3.3.8 EGA Color Palette

The EGA CRTC does not expect RGB values to generate pixels. Instead it is based on an index-based color palette system. Each pixel is a 4-bit index number, assigned to a color from the Attribute Controller. The default color palette are all 16 CGA colors, but it allows substitution of each of these colors with any one from a total of 64 colors.

When calculating the intended value in the 64-color EGA palette, the 6-bit number of the intended entry is of the form "rgbRGB" where a lowercase letter is the least significant bit of the channel intensity ($\frac{1}{3}$ color intensity) and an uppercase letter is the most significant bit of intensity ($\frac{2}{3}$ color intensity). The more intensity, the brighter the color is. For example, `02h` will produce green, `10h` will produce dim green and `12h` will produce bright green. The color magenta is created by setting both "R" and "B", which is color code `05h`. Each of the 16 color indexes could be reassigned to one color from the "rgbRGB" palette.

	00h	01h	02h	03h	04h	05h	06h	07h	08h	09h	0Ah	0Bh	0Ch	0Dh	0Eh	0Fh
00h																
10h																
20h																
30h																

Figure 3.25: EGA "rgbRGB" color palette (64 values from 00h to 3Fh)

However, standard EGA monitors did not support use of the extended color palette in 200-line modes. Both CGA and EGA cards use a female nine-pin D-subminiature (DE-9) connector for output.

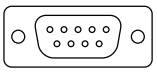


Figure 3.26: DE-9 video output connector.

The EGA monitor could only distinct EGA and CGA cards based on the Vertical Sync signal, which is either 200- or 350-line mode. If the Vertical Sync is 350-line mode, the monitor switched to Mode 2 operations which supported the extended rgbRGB-color information[20]. But in the 200-line mode, the monitor cannot distinguish between being connected to a CGA or an EGA card.

The CGA color output is based on the form "RGBI", where the 'I' stands for Intensity and adds brightness to the RGB color. Compared to CGA, EGA redefines some pins of the DE-9 connector to carry the extended rgbRGB-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.

[20] IBM Enhanced Color Display documentation.

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

Figure 3.27: EGA and CGA DE-9 connector pin signals.

Suppose one assigns the color brown (rgbRGB is `010100b`) to one of the color indexes, the resulting color on the CGA pin assignment is light red; The secondary green pin ("r" in rgbRGB) is mapped to the Intensity pin in CGA mode, which results to the color red with intensity and not the expected brown color.

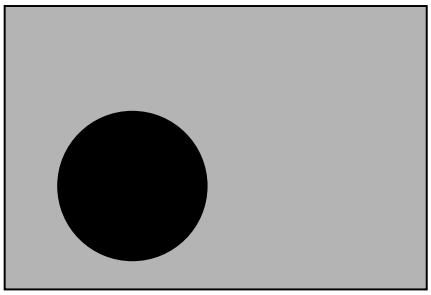
For this reason, EGA monitors will use the CGA pin assignment (mode 1) in 200-line modes so the monitor can also be used with a CGA card and vice versa. Therefore, the EGA card is fully backwards compatible with a standard CGA monitor. Thereby it is able to show all 16 CGA (RGBI-)colors simultaneously, instead of only 4 colors when using a CGA card.

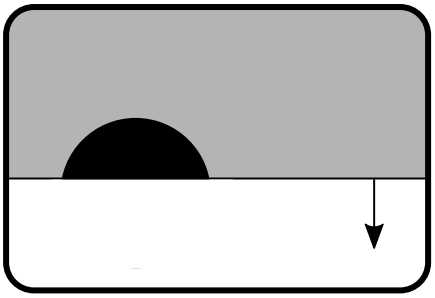
Index Number	Color	rgbRGB	RGBI
00h	Black	000000b	0000b
01h	Blue	000001b	0010b
02h	Green	000010b	0100b
03h	Cyan	000011b	0110b
04h	Red	000100b	1000b
05h	Magenta	000101b	1010b
06h	Brown	010100b	1100b
07h	Light grey	000111b	1110b
08h	Dark grey	111000b	0001b
09h	Bright blue	111001b	0011b
0Ah	Bright green	111010b	0101b
0Bh	Bright cyan	111011b	0111b
0Ch	Bright red	111100b	1001b
0Dh	Bright magenta	111101b	1011b
0Eh	Yellow	111110b	1101b
0Fh	White	111111b	1111b

Figure 3.28: Default EGA 16-color palette

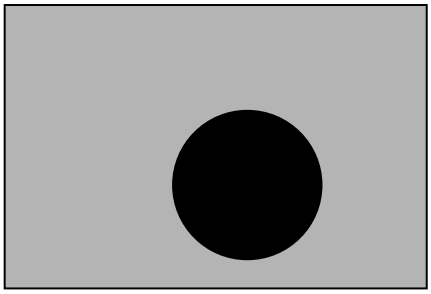
3.3.9 The Importance of Double-Buffering

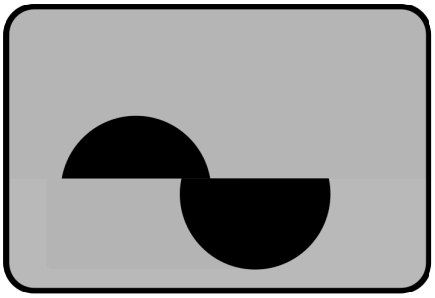
Double buffering has been mentioned often while describing the hardware, but so far we have not reviewed why it is paramount to achieving smooth animation. With only one buffer the software has to work at exactly the frequency of the CRT (60Hz). Otherwise a phenomenon known as "tearing" appears. Let's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





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





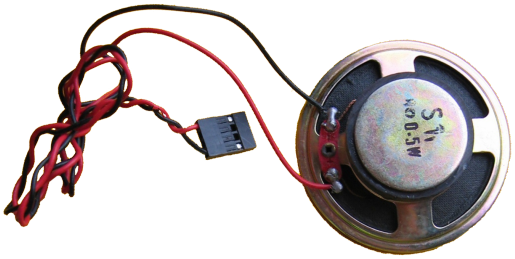
If the CPU is faster than the frequency of the CRT (60Hz), it can write the framebuffer again, before the scan is completed. This is what happened here. The next frame was drawn with the circle moved to the right. The electron beam did not know that and kept on scanning the framebuffer. The result on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

With two buffers (a.k.a double buffering) the CPU can start writing in the second framebuffer without messing with the framebuffer being scanned to the screen[21]. No more tearing! A full screen of 320x200 pixels with 16 colors requires 320x200x 4-bits = 32KB of VRAM (spread over 4 planes, each 8KB), so there is plenty of video memory left to keep multiple screens in an EGA card with 256KiB memory.

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

3.4 Audio

For the first 5-6 years of the IBM PC and its compatibles, their audio output came from nothing more than a simple loudspeaker with a tone generator. For business, this was acceptable - even preferable, since a PC in an office environment really shouldn't be a distraction to others! The loudspeaker, commonly known as a "PC Speaker", was capable of generating a square wave via 2 levels of output.



3.4.1 History of Sound Cards

The introduction of real game music and sounds on the PC started with Sierra back in 1988. They prepared to change all this by creating games that contained serious, high quality musical compositions drawing on add-on hardware. *King's Quest IV* was the first commercially released game for IBM PC compatibles to support sound cards. In addition to the familiar PC speaker and Tandy 1000, it could utilize

- Roland MT-32
- IBM Music Feature Card
- AdLib

Sierra struck a deal with two companies, Roland and AdLib, where Sierra would also become a reseller for these soundcards.

The Roland MT-32 was the higher end of these music devices. In today's terminology, it would be labeled a "Wavetable Synthesizer". A wavetable synthesizer usually implies that real instrument sounds are recorded into the hardware of the device. This device can then manipulate them to play them back at the various notes you need. The MT-32 had the ability to manipulate parts of its built in sounds using something called "Linear Arithmetic (LA)" synthesis. It was a very good device that can rival even today's sound cards. To connect the MT-32 to a PC required, what Roland called an MPU-401[22], in one of the PC's

[22]Midi Processing Unit-401

expansion slots. Sierra sold The MT-32 with a necessary MPU-401 interface for $550. The high price prevented it from dominating the end-user market of players.

Figure 3.29: Roland MT-32 synthesizer box.

The IBM Music Feature Card was launched in March 1987 as a collaboration between IBM and Yamaha. Essentially the Music Feature Card was a synthesizer installed on an 8-bit expansion card[23]. The Music Feature Card had 8 FM voices, controllable via 4 frequency operators. It came with over 300 high-quality synthesized instruments on-board, and it was actually possible to have two Music Feature Cards in a single PC to get 16 voices! With a tag price of $495 it was just like the Roland MT-32 an expensive card, and its audience was primarily business users.

[23]Roland released the LAPC-I in 1989, which basically was similar to the Music Feature Card: a MT-32-compatible Roland sythesizer with a MPU-401 unit, integrated onto one single full-length 8-bit ISA card.

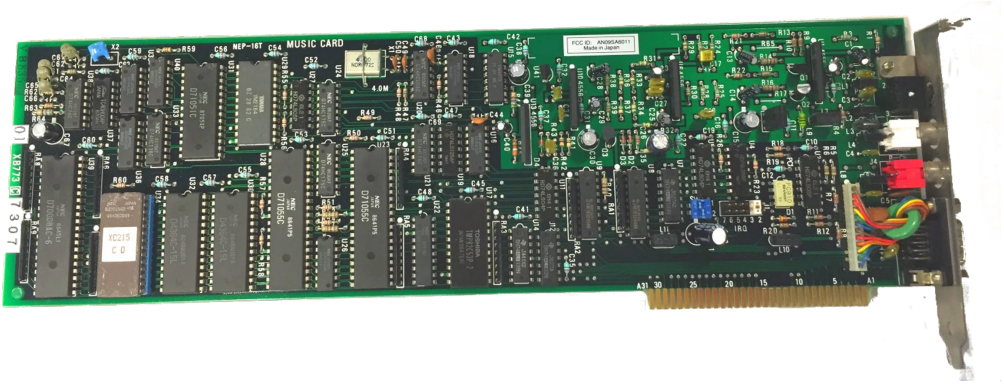


Figure 3.30: IBM Music Feature Card.

Trivia : The IBM Music Feature Card was the very first general purpose "sound card" for the PC, beating AdLib by several months.

3.4.2 AdLib

AdLib was the other company, beside Roland, that struck a deal with Sierra as a reseller. The company was founded in 1987 by Martin Prevel, a former professor of music from Quebec. The AdLib soundcard used a technology called FM synthesis. The technology is based on the idea of generating superimposing waveforms to create a sound. This technology was much less expensive than Roland's Wavetable technology.

The AdLib card was built around the Yamaha YM3812, also known as the OPL2 chip, and could produce either 9 sound channels or 6 sound channels plus 5 hit instruments simultaneously using Frequency Modulation (FM). Ideally, if you have enough generators and can fine tune the waveforms well enough, you can create a realistic sound. However, to reach this ideal, you need lots of skilled people, lots of money for equipment, and lots of time to develop. The reality is, this is a cheap PC soundcard offering a better alternative than the PC Speaker. Thus, FM synthesis sounded very artificial. Still, this was a great improvement over the PC Speaker. With a price tag of $219.99, it was much cheaper than the Roland MT-32 and IBM Music Feature Card, and soon ruled at the top of the early PC sound card market.

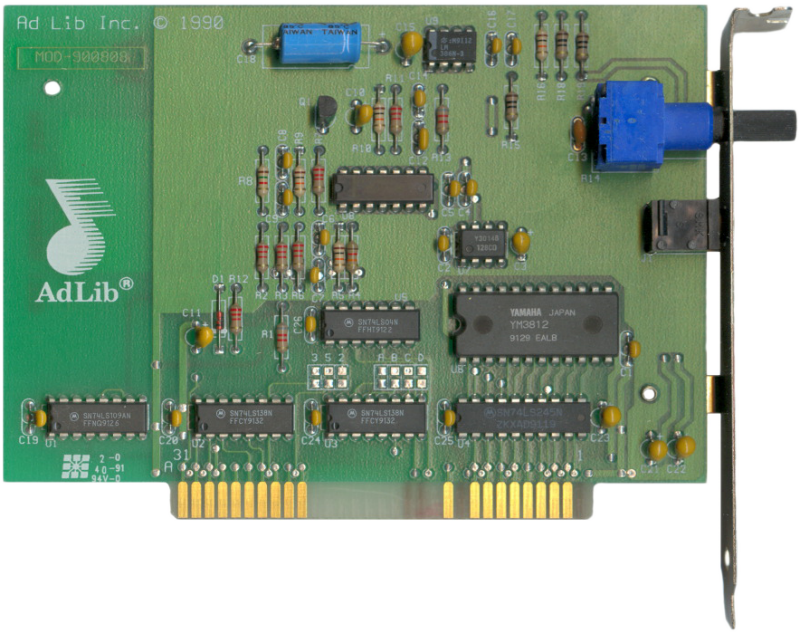


Figure 3.31: An AdLib sound card. Notice the big YM3812 chip and the 8-bit ISA connector.

The AdLib card dominated the PC market for almost three years. In 1989, Creative Labs released the competing SoundBlaster, which quickly dominated over AdLib. To compete with SoundBlaster, AdLib planned a new 12-bit stereo sound card called AdLib Gold. Sadly, due to AdLib's dependence on Yamaha who suffered long delays introducing their latest multimedia chipset, their new product, AdLib Gold PC-1000, was never to see the light of day under AdLib's management. Unable to remain solvent, AdLib closed its doors on 1st May 1992.

3.4.3 SoundBlaster

Creative Labs, the company behind the SoundBlaster, didn't enter the sound card industry until 1987 with the introduction of their Creative Music System (C/MS). From inception in 1981, they were a computer repair shop in Singapore. Creative Music System, or "Game Blaster" as it was renamed a year later, was an FM synthesizer card similar to AdLib. Based on the Philips SAA1099 chip which was essentially a square-wave generator, it sounded much like twelve simultaneous PC speakers would have, except for each channel having amplitude control. The card did not sell well.

The original Sound Blaster v1.0 and v1.5 were released in 1989 as the successor to their Game Blaster. Not only was it equipped with the OPL2 chip, providing 100% compatibility with AdLib music playback, but it was also technologically superior with a DSP[24] 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.

Creative Labs boasts in their own advert AdLib compatibility and even uses an image of AdLib Inc's product[25]! On the other hand they sued every company that tried to market a 'Sound Blaster compatible' product for using the name of their product.

Figure 3.32: A SoundBlaster 1.5, model CT1320B

Figure 3.32 is the SoundBlaster model CT1320B. Notice the OPL2 chip (labeled FM1312) and the CT1321 DSP on the middle top. In the middle of the card are the two CMS-301 chips, to ensure backwards compatbility with the Creative Music System.

Trivia : The CT1320B model (Sound Blaster 1.5) was a cost-cutting measure. Having recognised that C/MS was unpopular, they replaced the two C/MS chips with sockets. You could still purchase the C/MS chips for $29.95 if you wished and install them into these sockets.

[24] An Intel MCS-51 "Digital Sound Processor", not "Digital Signal Processor".
[25] Advert p20 in Compute!, April 1990.

Figure 3.33: Sound Blaster advertisement with Adlib compatability.

3.4.4 Disney Sound Source

The parallel port is nothing more than a way to pass out 8 bits simultaneously. Each PC had this port, and the conversion of this kind of data to an analogue signal is pretty simple and can either be achieved by using a DAC[26] or simply a set of resistors. Hence the only thing missing was the interface, which a company named Covox delivered with their "Speech Thing" for only $70.

The difficulty about this sound device was to get exact timing. All bytes are converted the moment they reach the parallel port. In many ways it works pretty similar to the PC-speaker, only its connected to the parallel port. The CPU was in the center of all this, and it required lots of CPU resources to timely move information from RAM to the parallel port. This left limited CPU time for the game or program, in a time period where the CPU already was the bottleneck.

Figure 3.34: The Disney Sound Source speaker box (DAC not shown).

The Disney Sound Source came out around 1990 and fixed multiple issues: It contained a small buffer, so even an unsteady data stream resulted in a good audio signal. It also allowed to pass through the printer port, so there's no need to switch cables to print. And at $14 it was dirt cheap! It came with its own amplifier, which ran with a 9 Volt battery.

[26] Digital to Analogue Converter.

It would have made programmers and customers happy if not for one serious limitation: The buffer was fixed at a 7,000Hz, which is about the quality you get from an old fashioned phone. This was still enough to produce pleasant music, but fell short for sound effects when compared to AdLib and Sound Blaster.

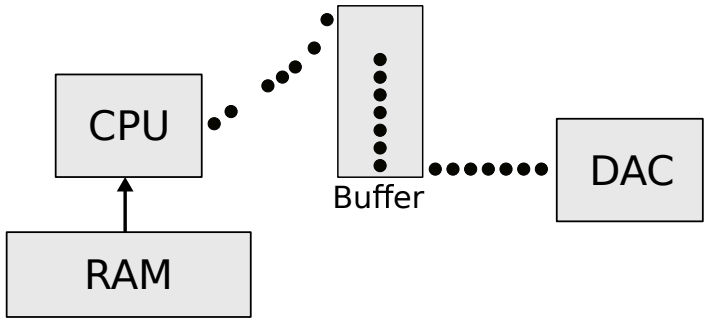


Figure 3.35: Disney Sound Source with buffer, running at fixed 7,000Hz.

3.5 Floppy Disk Drive

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

Trivia : An USB stick of 128GB contains more than 91K high-density 3½-inch (1.44MB) floppy disks.

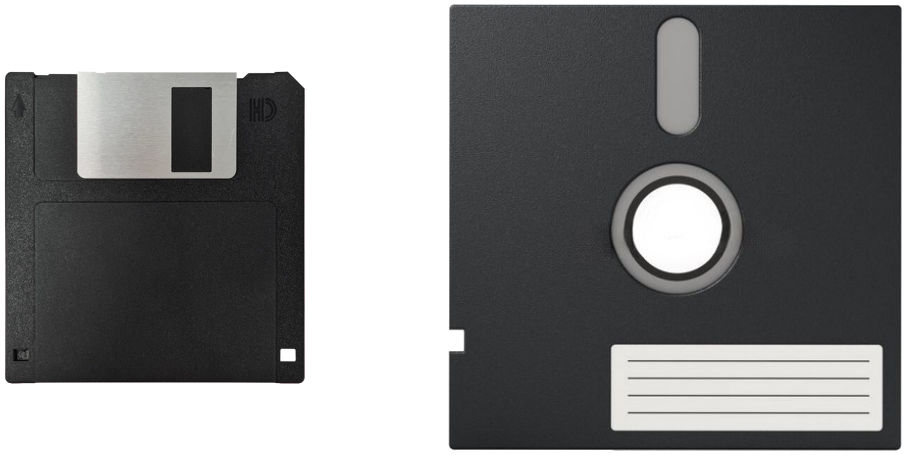


Figure 3.36: 3½-inch and 5¼-inch floppy disk.

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

The data on a floppy disk is stored in concentric circular tracks divided into arc-shaped sectors. The amount of data a disk can store is determined by the number of tracks and sectors and the density of the recorded information (single and double density). Near the center, there is a small hole called the index hole, which marks the start of a sector.

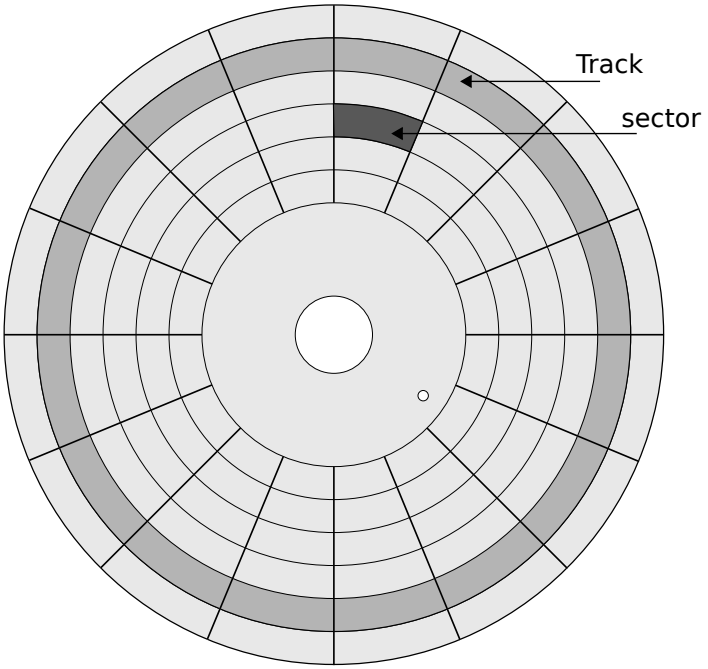


Figure 3.37: Floppy disk with tracks, sectors and the index hole.

When the floppy disk drive is powered up, the read/write heads move to the target track, starting from track 0 (the starting track on a floppy disk). Once the sensor reaches the target track, the computer is ready to retrieve or write data onto the floppy disk. To select a specific sector, the drive must wait for that sector to pass under the head. The drive determines which sector it is on by waiting for the index hole to be under the head. Since the rotation speed is kept constant, the time when the next sector is under the head is known. Now, the head can read or write to the specific sector/track combination on the disk.

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

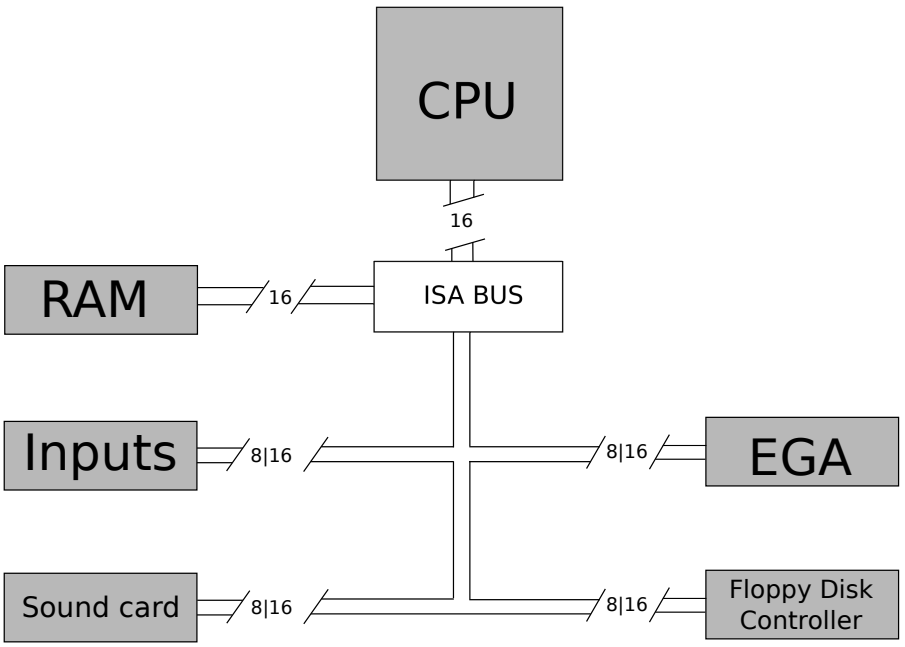
- Turn the disk motor on. When you turn a floppy drive motor on, it takes quite a few milliseconds to "spin up", to reach the (stabilized) speed needed for data transfer.
- Perform seek operation, which moves the head to the correct location for reading the data.
- Read the data from the floppy disk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

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

3.6 Bus

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

The ISA[27] 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.



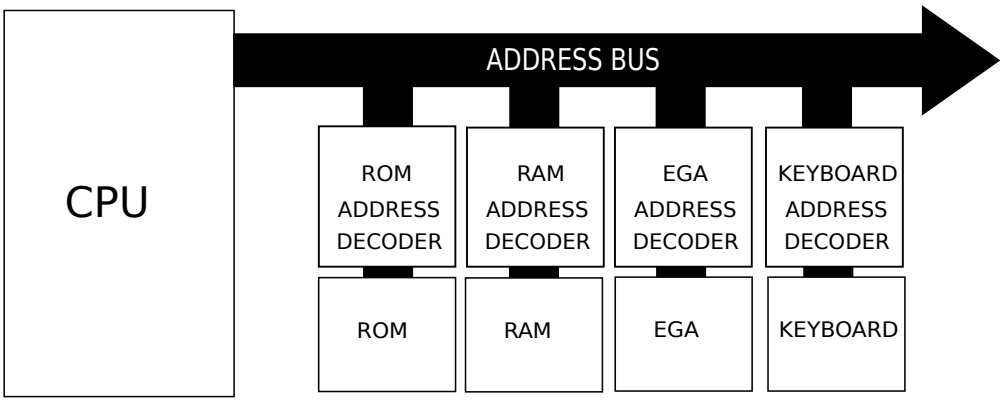
[27] Industry Standard Architecture.

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[28].

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`].



3.7 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, and the memory model only allowed 1 MiB with an address composed of two separate 16-bit registers. Last, but not least, the default sound system could only produce square waves.

Yet despite all these unfavorable conditions, teams of developers gathered to tame the beast and unleash its power to gamers. One of these called themselves *Ideas From the Deep*[29].

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

[29] 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 4

Assets

4.1 Programming

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

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

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

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

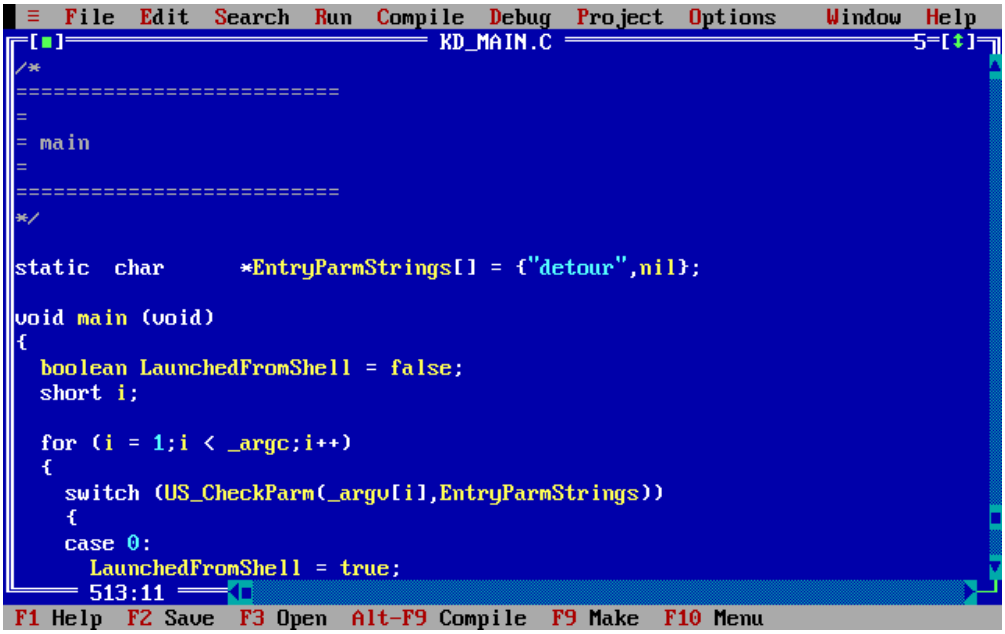


Figure 4.1: Borland C++ 3.1 editor

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

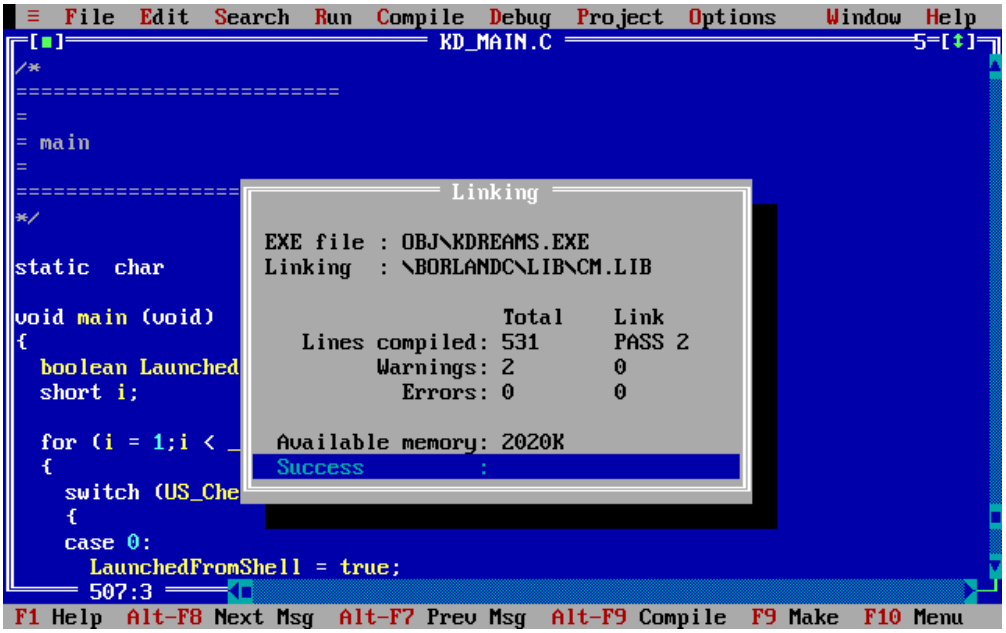
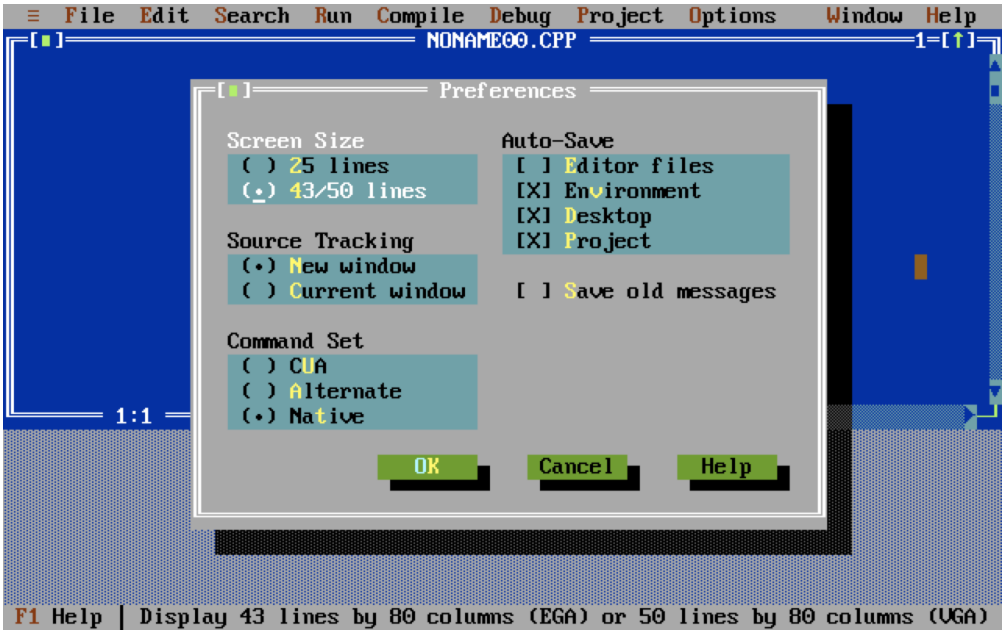
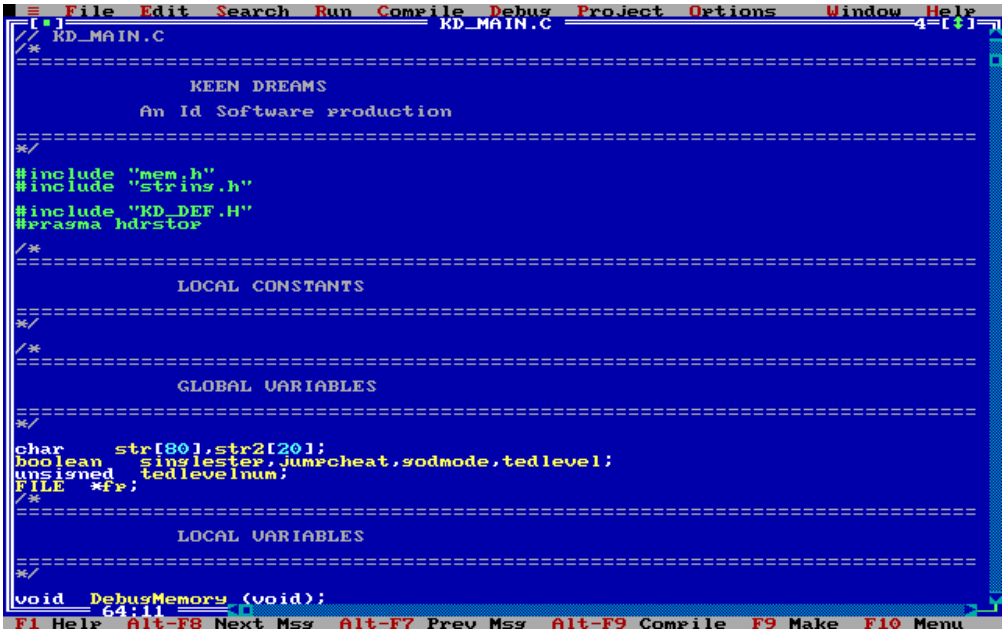


Figure 4.2: Compiling Keen Dreams with Borland C++ 3.1

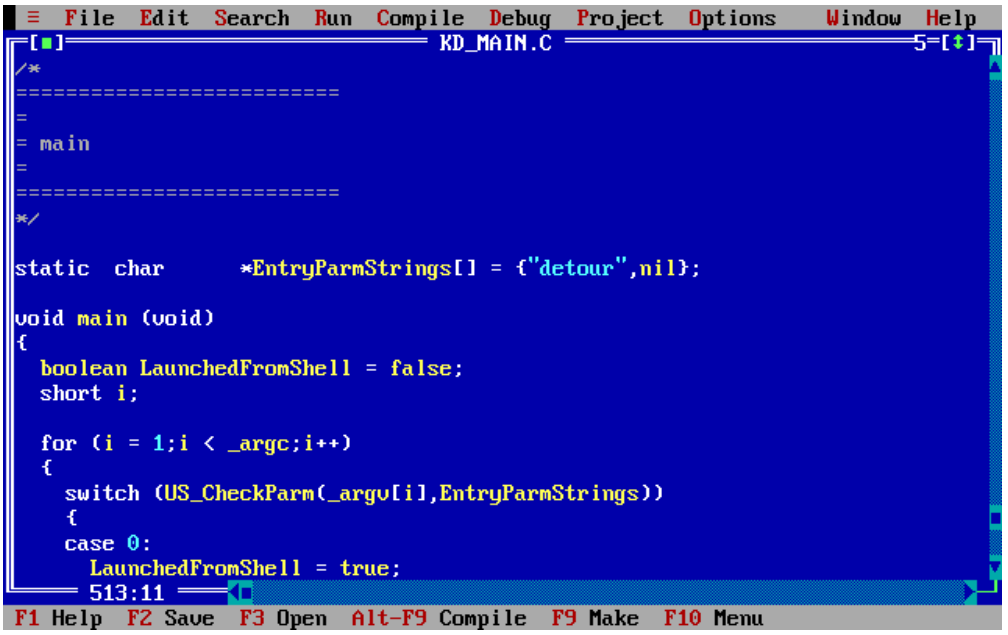
Another way to improve screen real estate was to use "high resolution" 50x80 text mode.

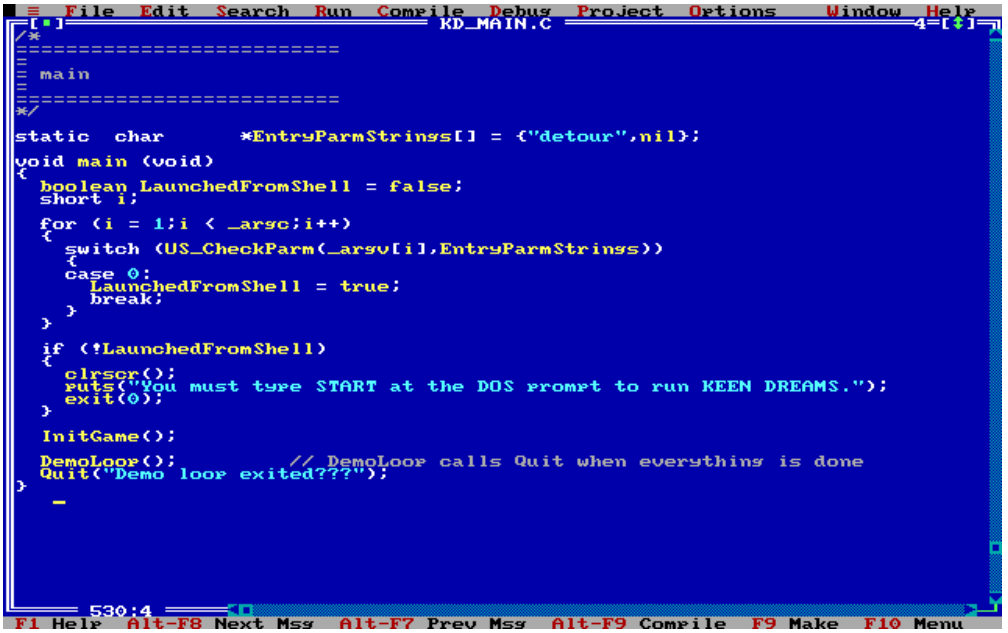


The comments still fit perfectly on screen since only the vertical resolution is doubled.



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





4.2 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.

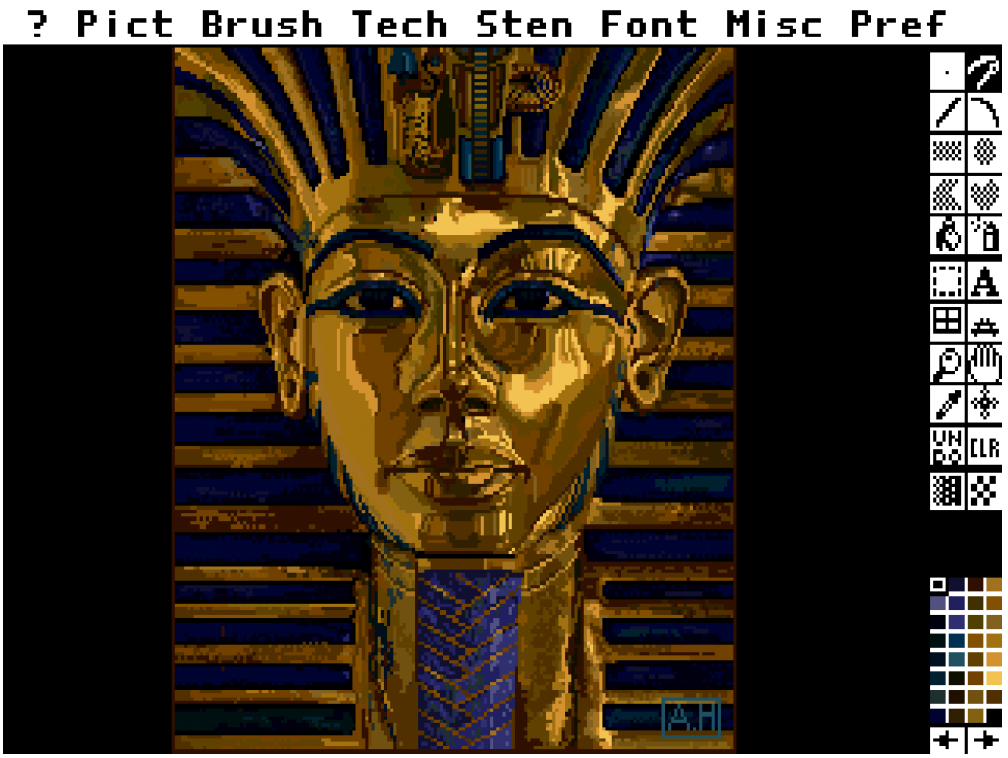


Figure 4.3: Deluxe Paint was used to draw all assets in the game.

[2] InterLeaved BitMap.

4.2.1 Assets Workflow

After the graphic assets were generated, a tool (IGRAB) packed all ILBMs together in a compressed DATA archive (EGA-file) and generated a HEAD table, DICT file (KDR-files[3]) and C header file with asset IDs. The engine references an asset directly by using these IDs.

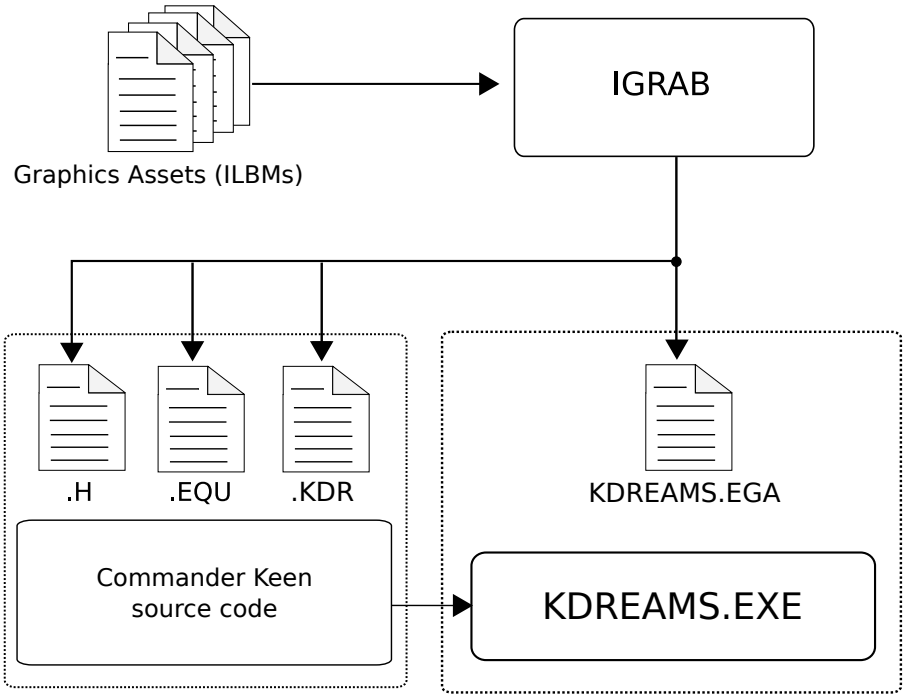


Figure 4.4: Asset creation pipeline for graphics items

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. With this indirection layer, assets could be regenerated and reordered at will with no modification in the source code.

Trivia : The HEAD and DICT files in the source code must match the DATA file from the shareware version of Keen Dreams. The latest release of the source code (v1.93) doesn't match with the final shareware version 1.13. You need to retrieve a specific git commit to ensure a HEAD, DICT and DATA archive are matching, which will be explained in the next chapter.

[3]both KDR-files are located in the static folder of the source code.

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


#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 CURSORARROWSPR         71
#define KEENSTANDRSPR          72
#define KEENRUNR1SPR           73
#define KEENRUNR2SPR           74
#define KEENRUNR3SPR           75
#define KEENRUNR4SPR           76
...

//
// Data LUMPs
//
#define CONTROLS_LUMP_START         4
#define CONTROLS_LUMP_END          68
#define KEEN_LUMP_START            72
#define KEEN_LUMP_END             212
#define WORLDKEEN_LUMP_START      213
#define WORLDKEEN_LUMP_END        240
#define BROCCOLASH_LUMP_START     241
#define BROCCOLASH_LUMP_END       256
#define TOMATO_LUMP_START         257
#define TOMATO_LUMP_END           260
...
```

4.2.2 Assets archive file structure

Figure 4.5 shows the structure of the `KDREAMS.EGA` asset archive file. The first sections contains data tables for pictures and sprites, followed by the font and all sprites and tiles.

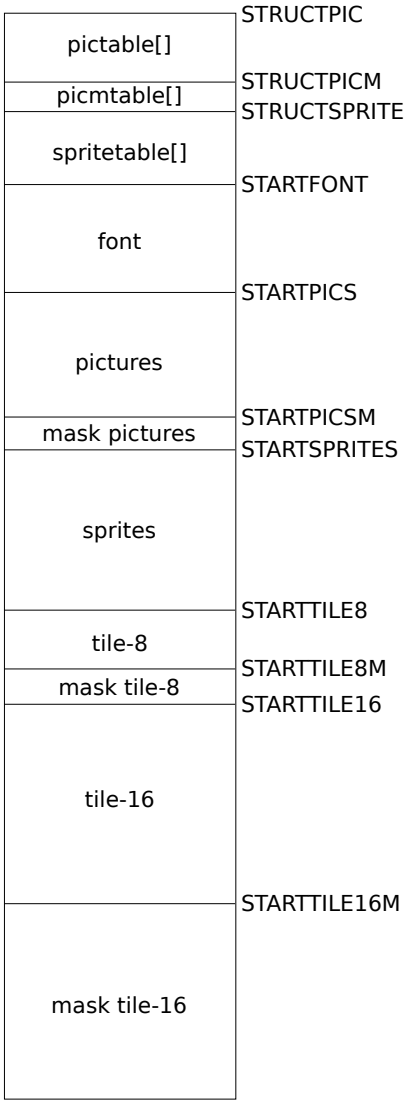


Figure 4.5: File structure of `KDREAMS.EGA` archive file.

Boh the **pictable[]** and **picmtable[]** contain the width and height for each (masked) picture in the asset file.

picture index	width (bytes)	height (bytes)
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

The **spritetable[]** contains, beside width and height, also information on the sprite center, hit boundaries and number of bitshifts. Sprites will be explained in detail in section 5.12.2 on page 155.

```
// ID_VW.H

typedef struct
{
  int width,
  height,
  orgx,orgy,
  xl,yl,xh,yh,
  shifts;
} spritetabletype;

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

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

Figure 4.6: Font asset data.

The remainder of the archive file contains all graphic assets, including pictures, sprites and tiles. Each asset contains four planes, aligned with the EGA architecture. All masked tiles and sprites include an additional mask plane as well.

Figure 4.7: Picture assets.

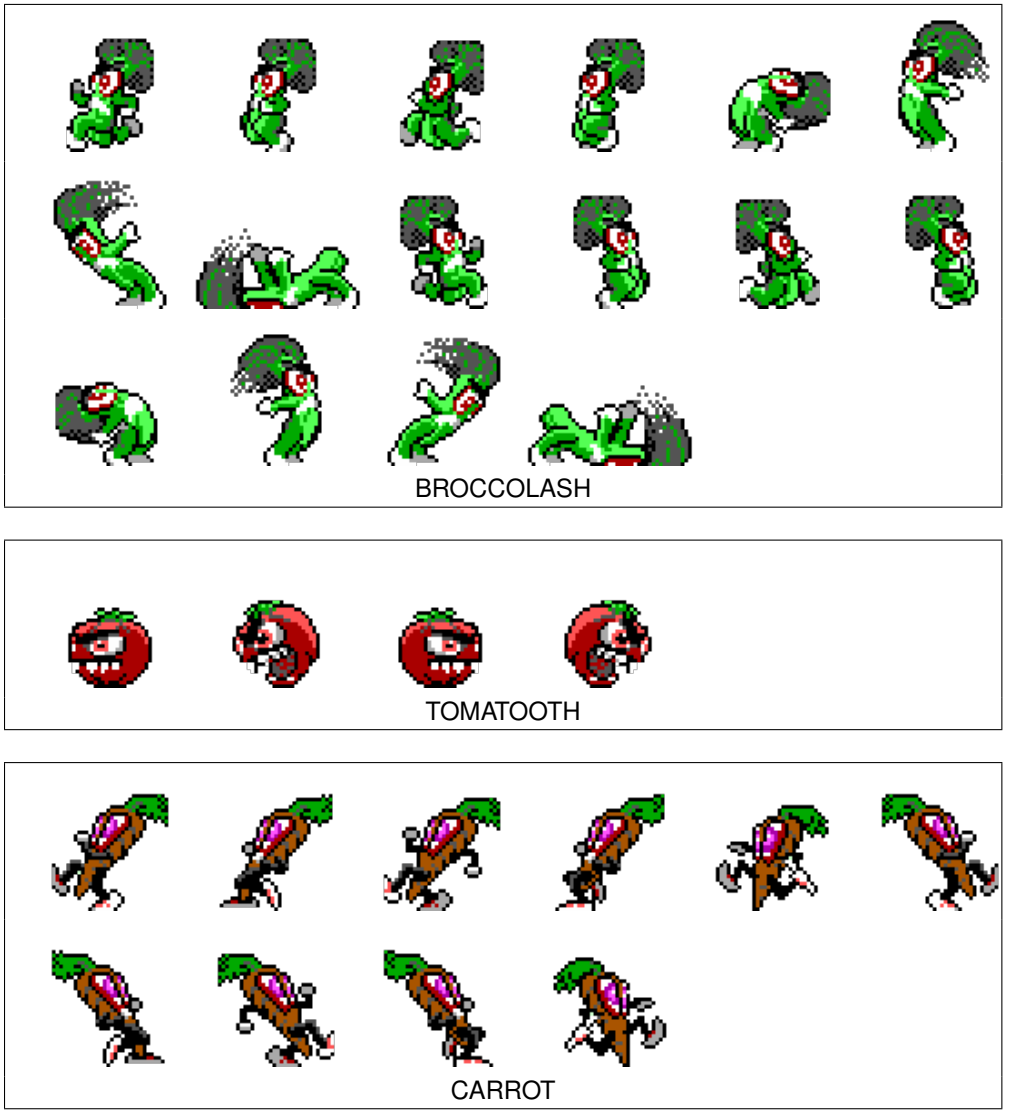


Figure 4.8: Sprite assets.

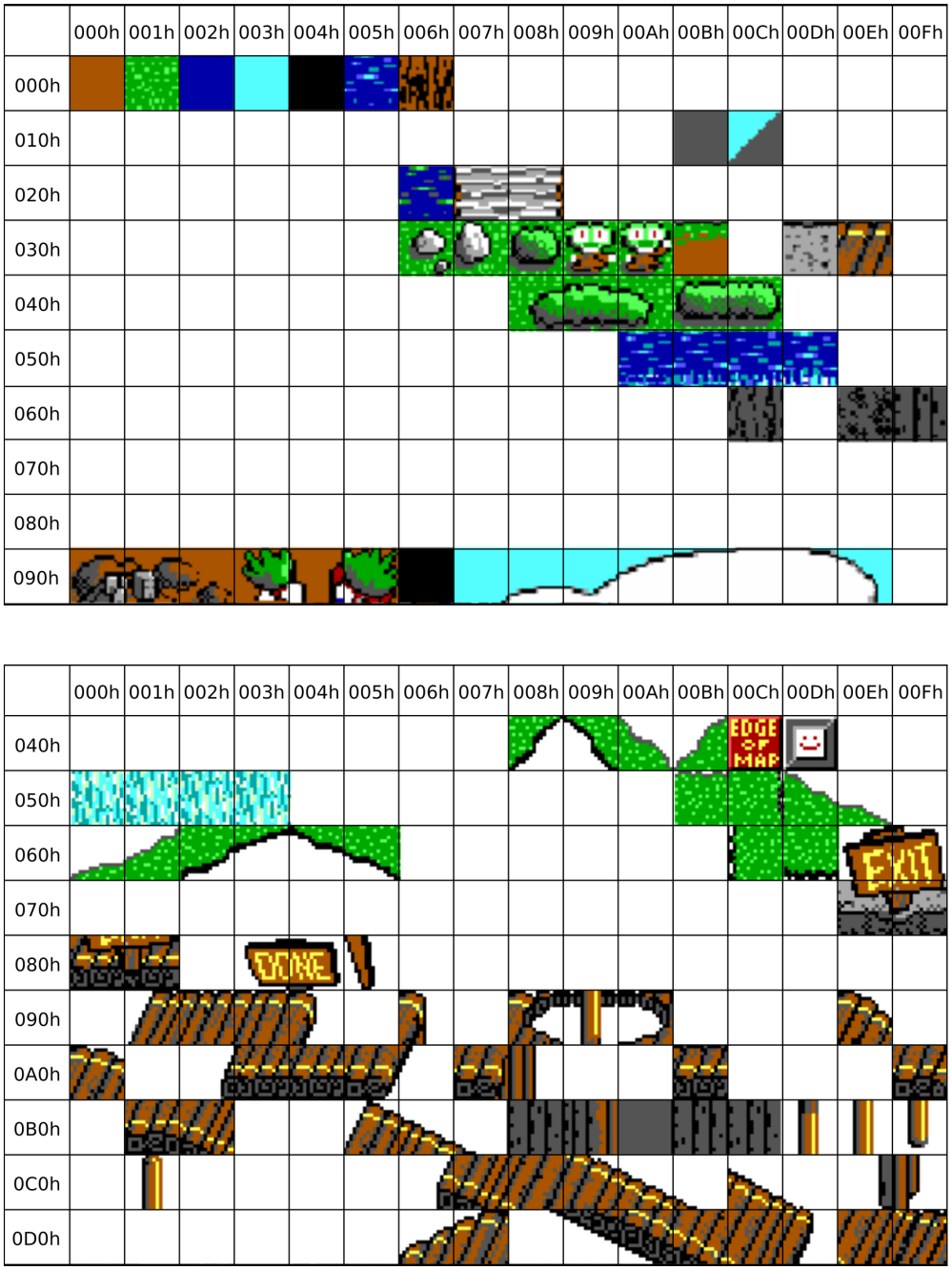
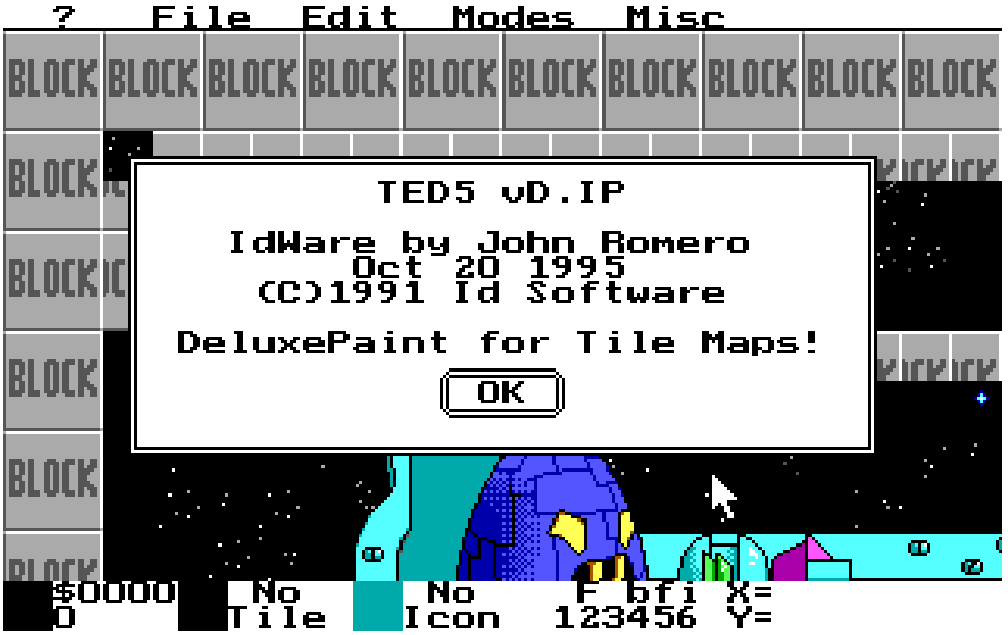


Figure 4.9: Background (Tile-16) and foreground (masked Tile-16) assets.

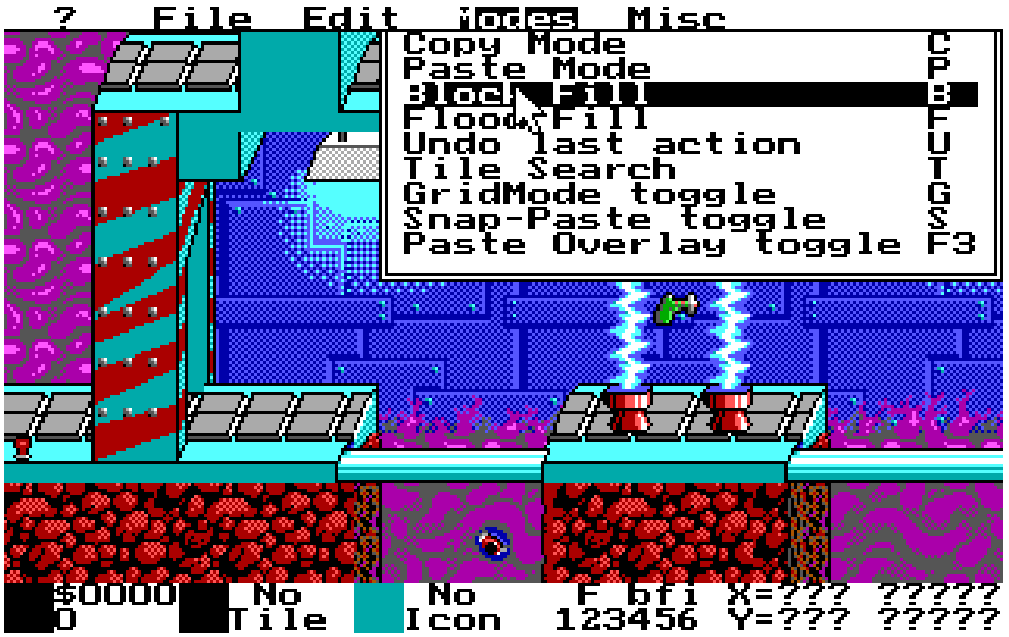
4.3 Maps

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

TED5 is not stand-alone; in order to start, it needs an asset archive and the associated header (as described in the graphic asset workflow Figure 4.4 on page 71). This way, tile IDs are directly encoded in the map.



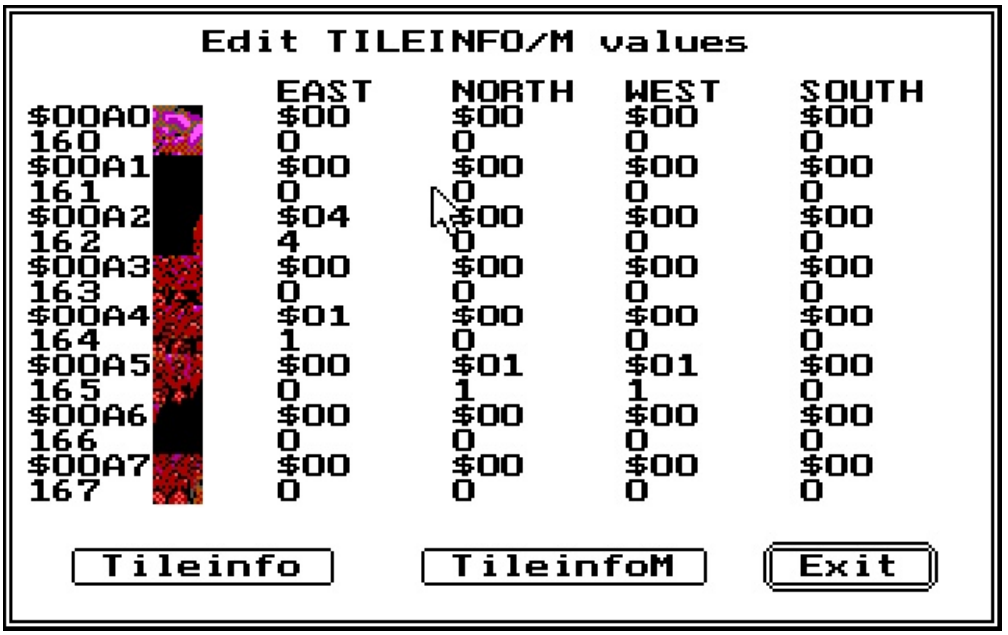
Trivia : The suffix, "vD.IP", was put in by the Rise of the Triad team in 1994. It stood for "Developers of Incredible Power".



TED5 allows placement of tiles on layers called "planes". In Commander Keen, there are three type of planes:

- Background plane tiles
- Foreground plane tiles, which is a mask over the background plane
- Information plane tiles, containing location of actors and special places

A level is created by placing tiles on each of these three layers. Each foreground and background tile could be enriched with additional tile information. It controls how everything interacts with tiles, like tile clipping, "deadly" tiles and animated tiles.



Just like IGRAB, the TED5 tool packed all levels together in a compressed `MAP` archive and generated a `HEAD` and `DICT` file (`KDR`-extension).

4.3.1 Map header structure

The map header structure is part of the engine code and contains a header offset and size, which refers to the location and size in the `KDREAMS.MAP` archive file.

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

A maximum of 100 maps is supported in the game. The `tileinfo[]` data contains information for tile interaction.

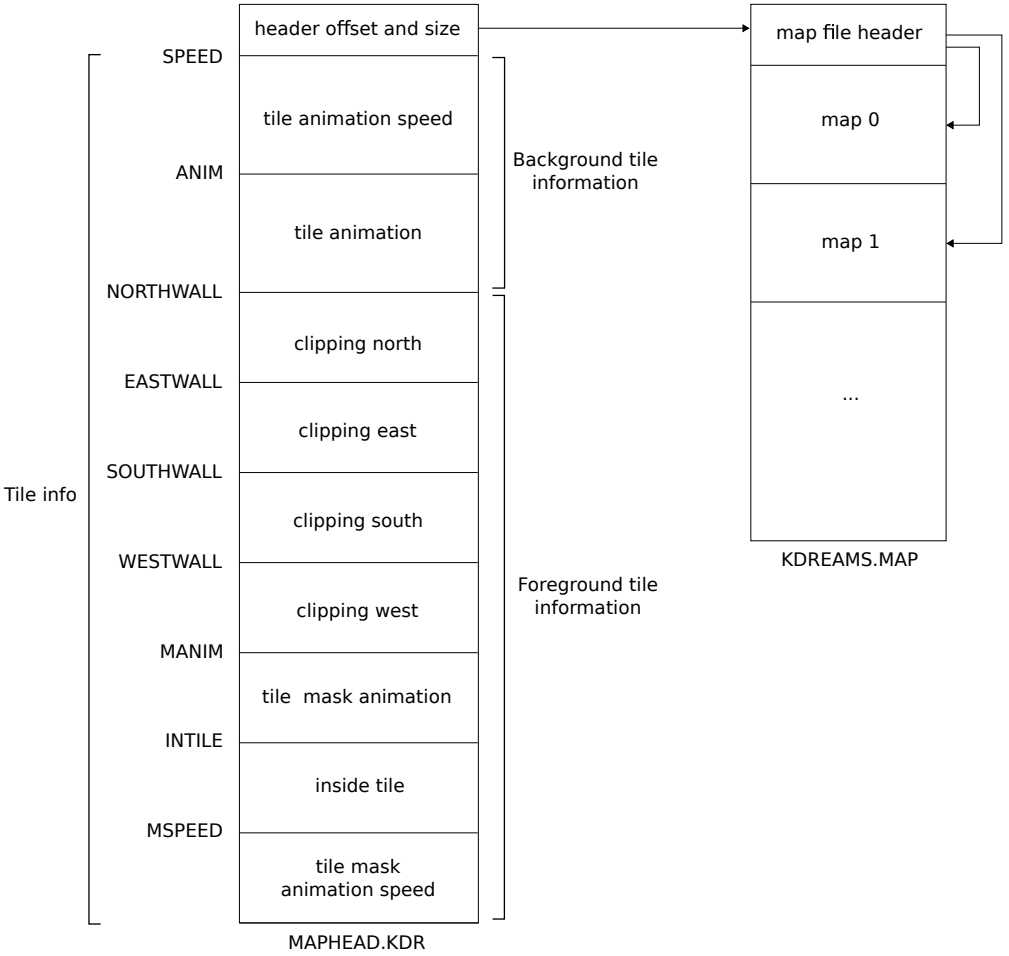


Figure 4.10: Structure of `MAPHEAD.KDR` header file.

For background tile animation two information tables are defined: **tile animation** and **tile animation speed**. The tile animation refers which tile, relative to this tile, the tile will animate to. In Table 4.1 the next animation after tile #90 is #91 (+1), followed by #92 (+1) and #93 (+1). After tile #93 (-3) the sequence is going back to tile #90. The animation speed is expressed in number of ticks before the next tile is displayed.

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

Table 4.1: Background tile animation.

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

The clipping tables contain how Commander Keen is clipped against foreground tiles. 'Inside' tile information is used to climb on a pole and mimics Commander Keen going through a floor opening ('inside' the tile). In section 5.12.3 (see page 158) both the clipping and 'inside' logic is further explained.

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

Table 4.2: Foreground tile clipping and 'inside' tile information.

4.3.2 Map archive structure

The structure of KDREAMS.MAP archive is explained in Figure 4.11. For each map there is a small header containing the width, height and name of the map as well as a reference pointer to each of the three planes. Each plane exists out of a map of tile numbers for background, foreground and info.

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

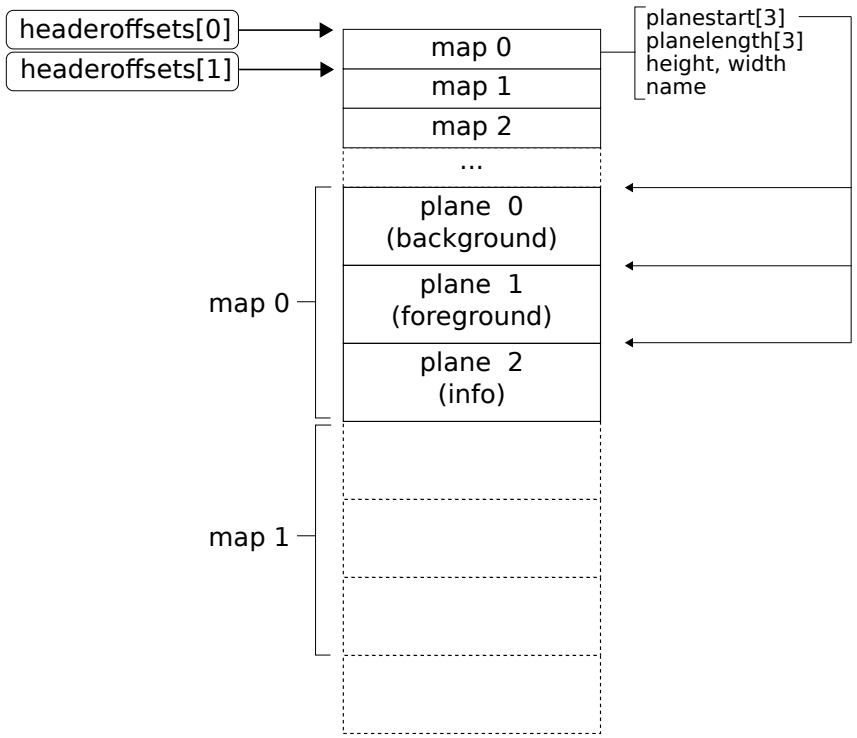


Figure 4.11: Structure of KDREAMS.MAP file.

4.4 Audio

4.4.1 Sounds

The original Commander Keen Trilogy did only support the default PC Speaker. With the introduction of Commander Keen in Keen Dreams the team decided to support sound cards as well.

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

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

Figure 4.12: Setup music menu in Keen Dreams, although there is no music in the game.

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

As a consequence, only two sets of each audio effect are shipped with the game:

1. For PC Speaker
2. For AdLib

Game music was not introduced until Commander Keen IV.

All sound effects are done by Robert Prince. Sound effects are created using Cakewalk and inhouse MUSE tool, which is extensively described in *Wolfenstein 3D*[4] Just like described before, the MUSE tool packed all sound effects together in a compressed `SOUND` archive and generated a `HEAD`, `DICT` file and a C header file with asset IDs.

4.5 Distribution

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

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

[4] See Wolfenstein 3D Blackbook, section 3.6 on page 94.
[5] The settlement with Softdisk is explained in Appendix B

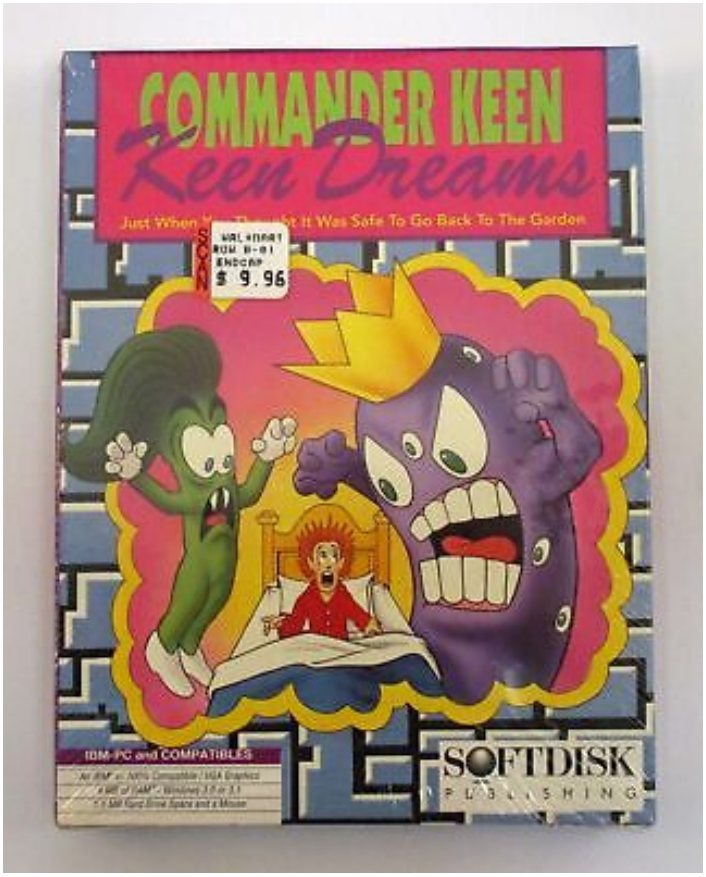


Figure 4.13: Retail version of Commander Keen in Keen Dreams by Softdisk.

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

The files can be divided in seven parts:

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

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

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

C:\KDREAMS>
```

Figure 4.14: All Keen Dreams files as they appear in DOS command prompt.

Chapter 5

Software

5.1 About the Source Code

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

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

5.2 Getting the Source Code

The source code is made available via `github.com`. It is important to take the the source code from shareware version 1.13, otherwise you run into issues due to incompatible map headers[2]. To get the correct source code

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

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

[2] See issue #7 on https://github.com/keendreams/keen.

5.3 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[3] 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.

[3] 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`.

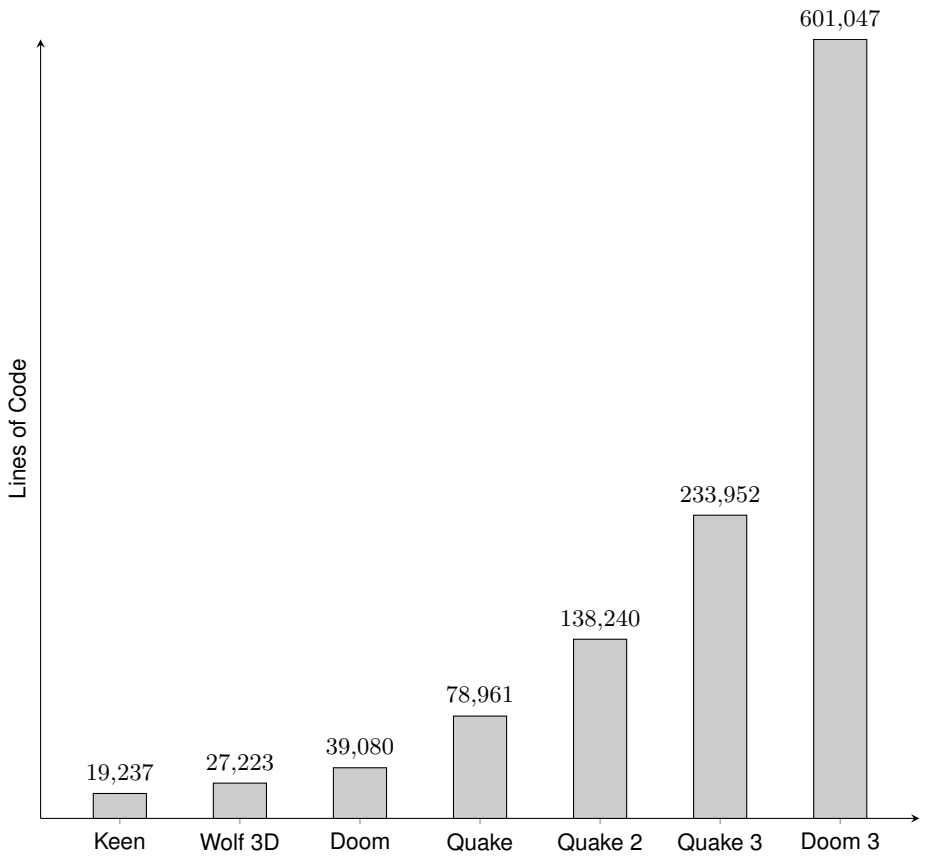


Figure 5.1: Lines of code from id Software game engines.

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

- `static` folder: Static header files for loading assets (as explained in Section 4.2).
- `lscr` folder: Load and decompress Softdisk data files.
- `README`: How to build the executable.

5.4 Compile source code

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

- Commander Keen source code.

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

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

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

```
mkdir OBJ
```

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

```
chdir STATIC
make.bat
```

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

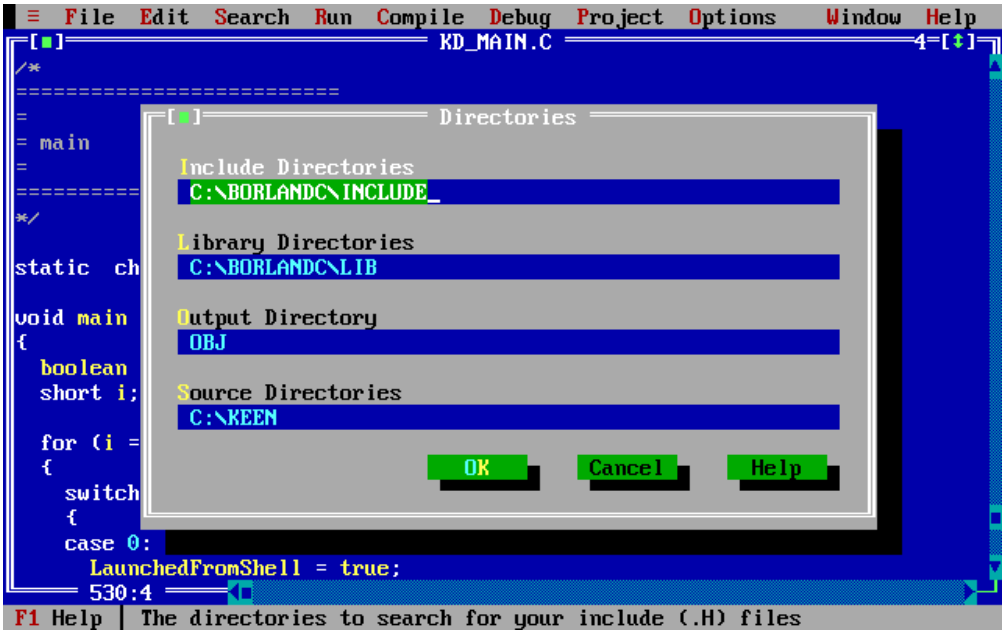


Figure 5.2: Borland C++ 3.1 directory settings

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.

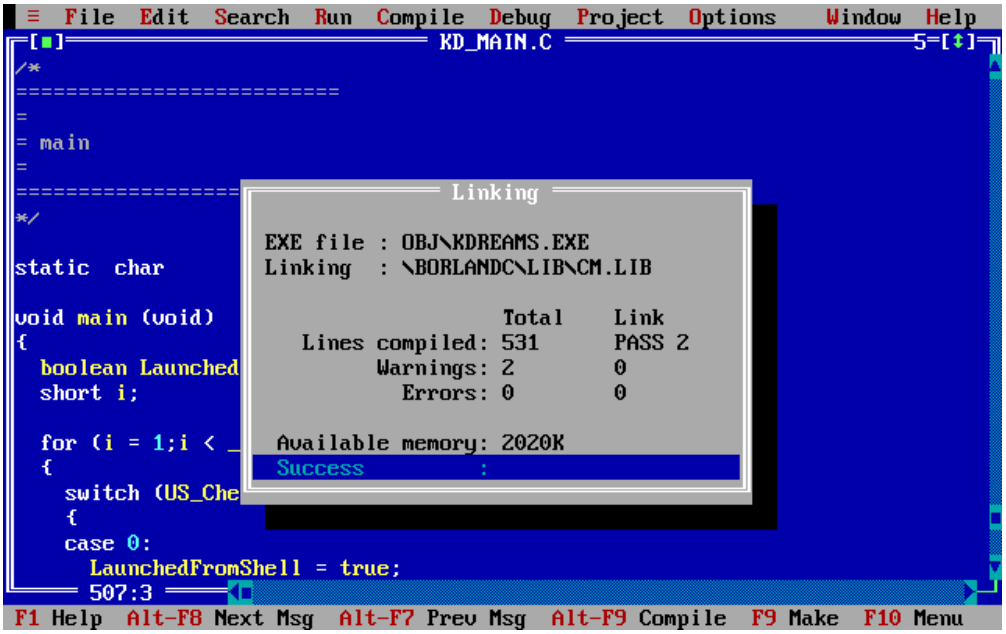


Figure 5.3: Commander Keen compiling

5.5 Big Picture

The game engine is divided in three blocks:

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

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

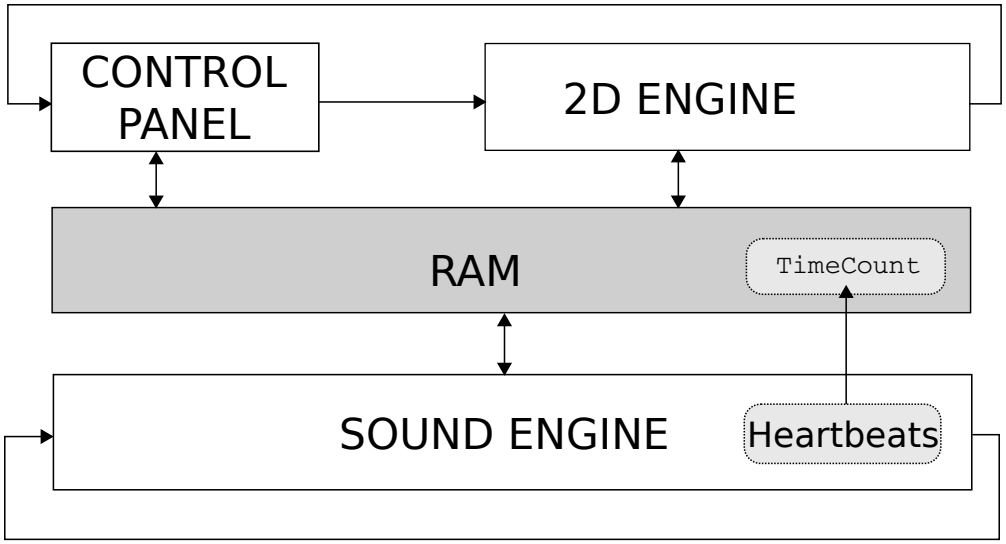


Figure 5.4: Game engine three main systems.

5.5.1 Unrolled Loop

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

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

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

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

  InitGame();

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

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

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

  MM_Startup ();     // Memory Manager

  US_TextScreen();  // Show intro screen

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

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

  CA_ClearMarks ();  // Clears out all the marks

  CA_LoadAllSounds (); // Load all sounds

}
```

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

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

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

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

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

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

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

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

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

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

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

The reason for interrupts is extensively explained in Chapter 5.15 "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;

}
```

5.6 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.

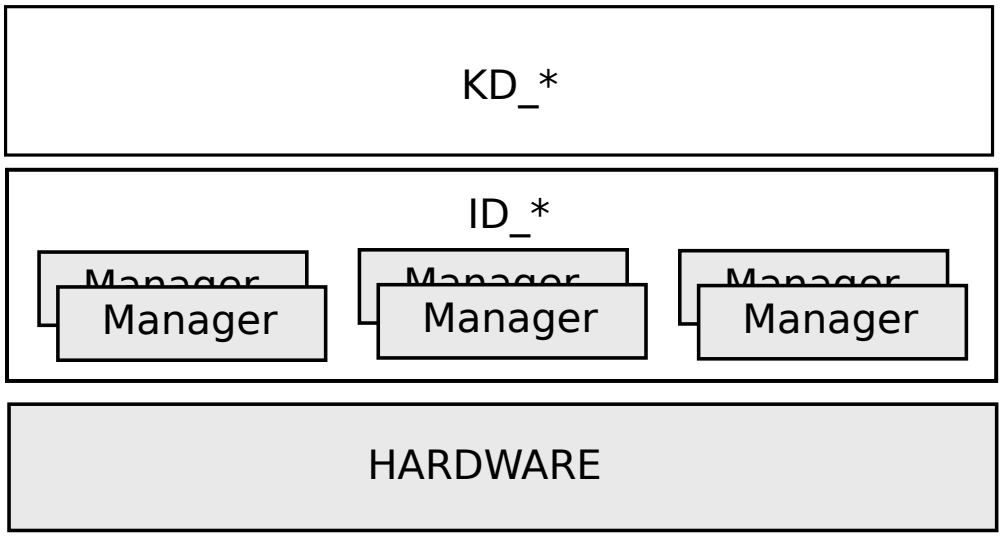


Figure 5.5: Commander Keen source code layers.

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).

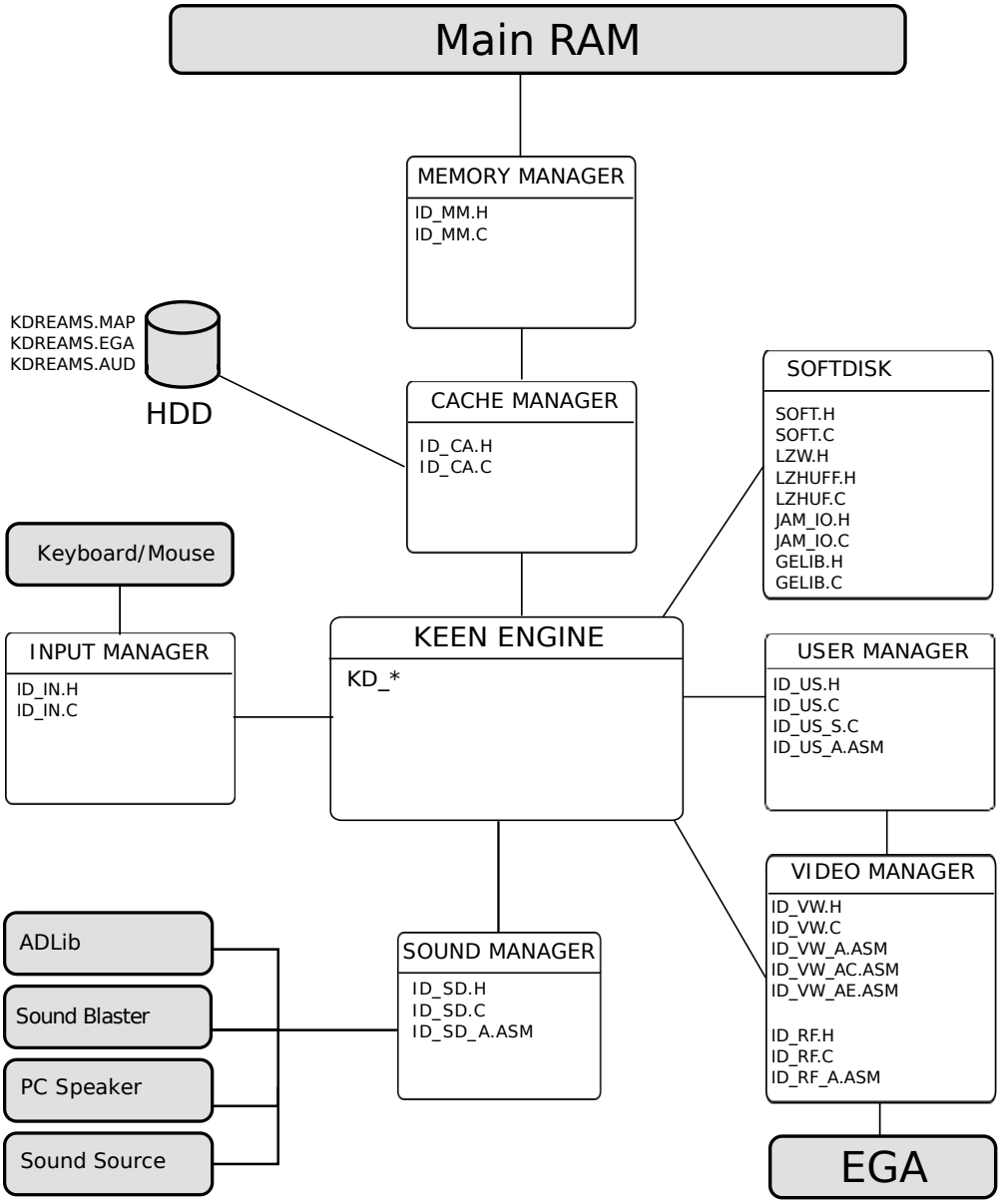


Figure 5.6: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

Next to the hard drives (HDD) you can see the assets packed as described in Chapter 4.2.

5.6.1 Memory Manager (MM)

The engine has its own memory manager, to have more control over memory fragmentation and optimization. The memory manager is made of a linked list of "blocks" keeping track of the RAM. A block points to a starting point in RAM, has a size and can be marked with attributes:

- LOCKBIT : This block of RAM cannot be moved during compaction.
- PURGEBITS : Two levels available, 0= unpurgeable, 3=purge first.

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

The memory manager starts by allocating all RAM, starting from 00000h, and assigning each block to the linked list. Since the engine is compiled using the medium memory model, there are two heaps available: the near heap between the global variables and stack, and the far heap starting right behind the stack. The total available free memory space for the near heap can be obtained by

```
length=coreleft();
start = (void far *)(nearheap = malloc(length));
```

The memory manager allocates all memory blocks in size of segments, meaning each block is provided in step size of 16 bytes. Therefore, the start and length of the memory block must be segment aligned.

```
length -= 16-(FP_OFF(start)&15); // round to 16 bytes
seglength = length / 16;         // now in segments
segstart = FP_SEG(start)+(FP_OFF(start)+15)/16;
```

The first allocated block is the unusable block of memory from segment 0000h to the start of the near heap.

```
GETNEWBLOCK;
mmhead = mmnew;     // this will allways be the first node
mmnew->start = 0;
mmnew->length = segstart;
mmnew->attributes = LOCKBIT;
```

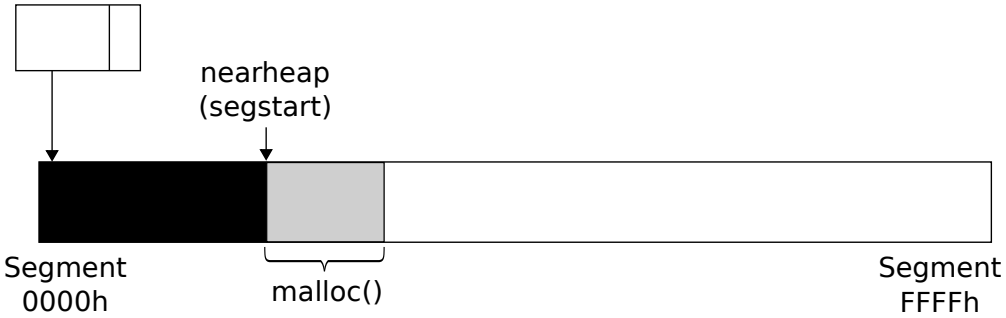


Figure 5.7: First locked block of unusable memory.

The stack is the next unusable block of memory. As the stack will grow/shrink during game execution, an additional part of the near heap free memory must be reserved for the stack.

```
#define SAVENEARHEAP 0x400 //space to leave in data segment
length -= SAVENEARHEAP;    //Reduce length of near heap
```

The next step is allocate the far heap, which can be obtained by

```
length=farcoreleft();
start = farheap = farmalloc(length);
```

The second block of unusable memory is the start of the stack memory, including the reserved memory block, until the start of the far heap.

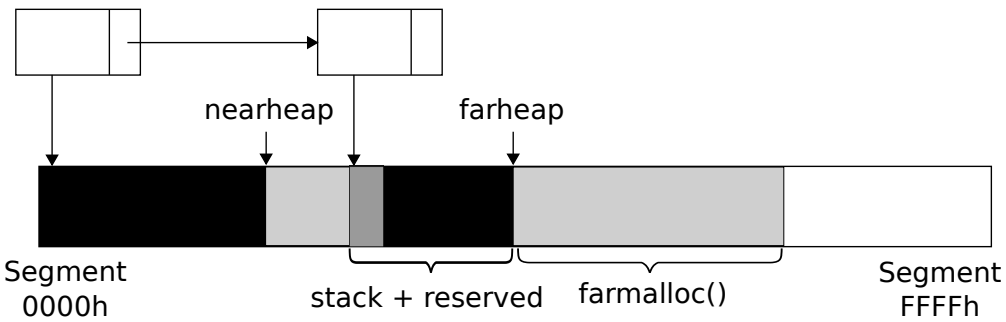


Figure 5.8: Second locked block: stack.

Finally, the last block of unusable memory is between the end of the far heap and the end of the 1MiB memory, the area that is typically for VRAM, ROM, etc. Now the entire memory starting at segment `0000h` up to `FFFFh` is allocated and chained together, and under full

control of the memory manager.

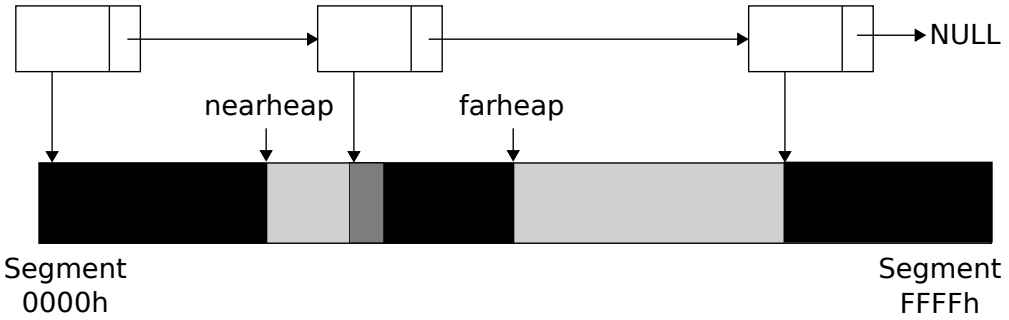


Figure 5.9: Final initial memory manager state.

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[4].

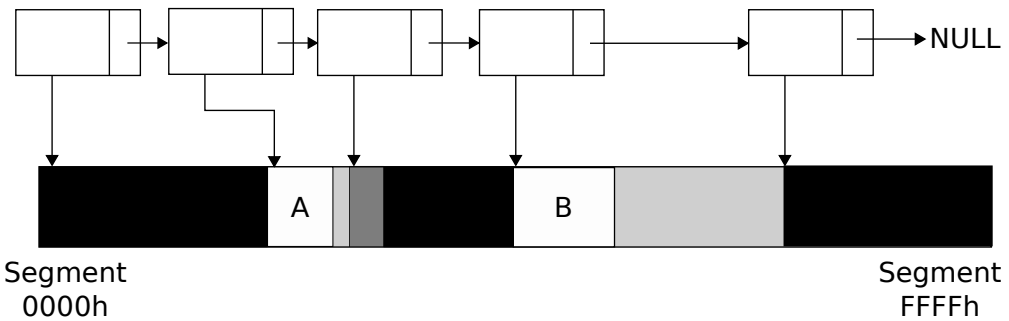


Figure 5.10: Allocation of memory.

By calling `MM_ShowMemory` you can inspect at any moment during game play the current memory usage. Each pixel represents 16 bytes (one segment increment). The red area represents blocked memory, black is available memory, blue is unpurgeable and purple is purgeable memory. The small white pixels are the start of a new memory block from the linked list. Notice how small the size of the near heap is (first black line) compared with the large heap!

[4]See the "Game Engine Black book Wolfenstein 3D" for a detailed description how the memory manager is allocating memory

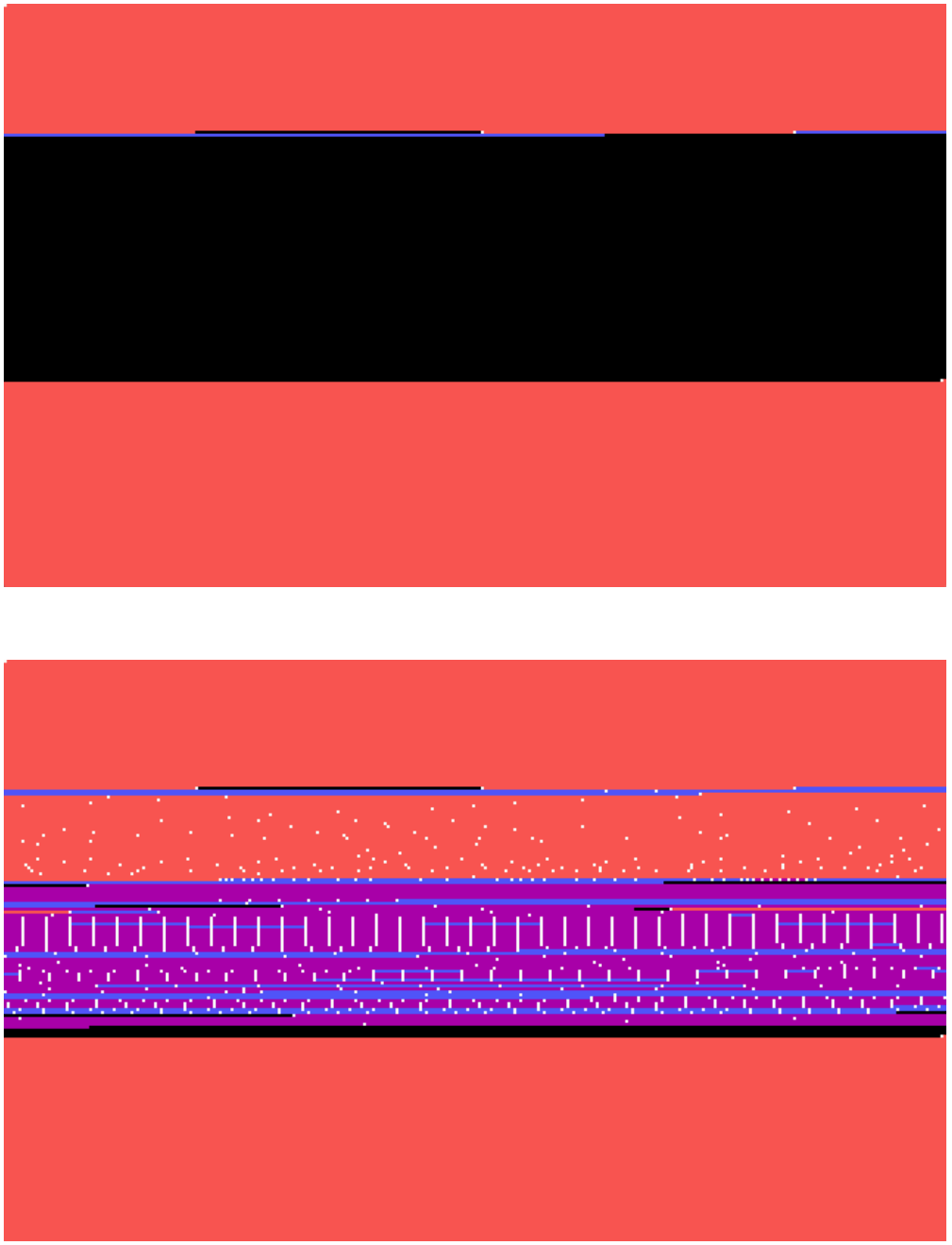


Figure 5.11: Memory dump after (i) initializing memory manager and (ii) running the game.

5.6.2 Video Manager (VW & RF)

The video manager features two parts:

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

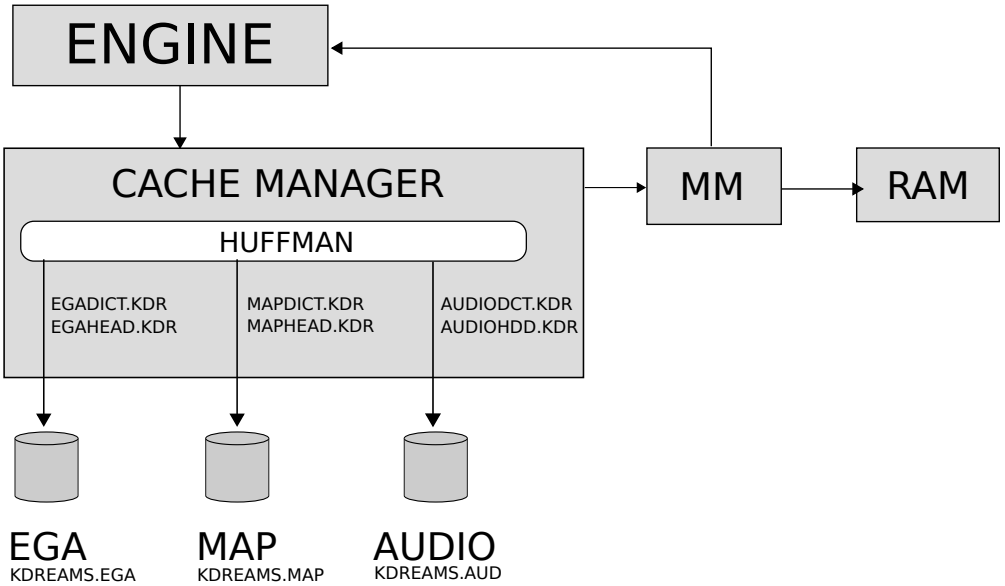
The video manager is described extensively in section 5.9 on page 122.

5.6.3 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 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 4.1. The header and dictionary files are provided with the source code in the `static` folder and contain `*.KDR` extension. Both file types are integrated in the engine code 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, and the maps have an additional RLEW compression.



5.6.4 User Manager (US)

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

Once we start the game, we move the display to EGA graphic mode `0x0D`. Here we can't directly print characters on the screen anymore. So a key function of the User Manager is to print text on a given pixel location. In the graphic assets file the complete font is stored with the following information for each character:

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

Each character has the same height of 10 pixels, but the character width could vary as illustrated in Figure 5.12.

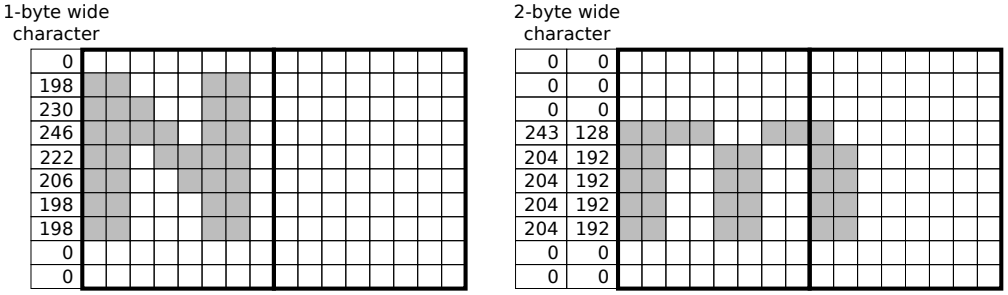


Figure 5.12: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

As explained in section 3.3.7 on page 46, each 8 pixels are represented by 1 byte per memory bank. So how do we print a character which is not perfectly aligned with the memory layout? Here a trick of bitshift tables 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 8 shift tables to fully cycle through 8-bits. In Figure 5.13 the bitshift for the values '198' is illustrated.

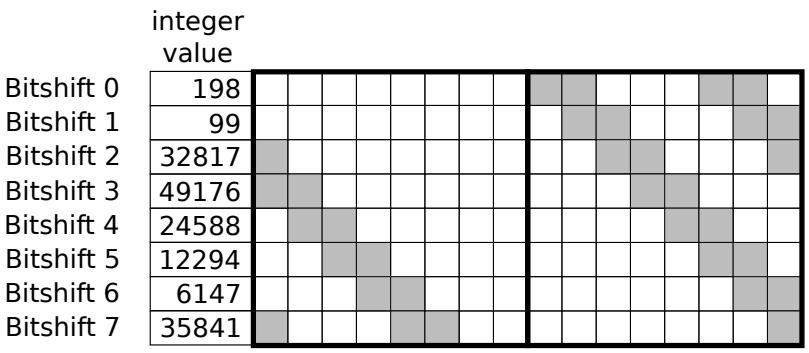


Figure 5.13: Right bitshift [0-7] for 198.

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

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

Now let's take the example of printing 'N', with an offset of 3 pixels. A simple lookup in `shiftdata3` results in the 3-bit shifted 'N'. Note that you first display the low byte and then the high byte value.

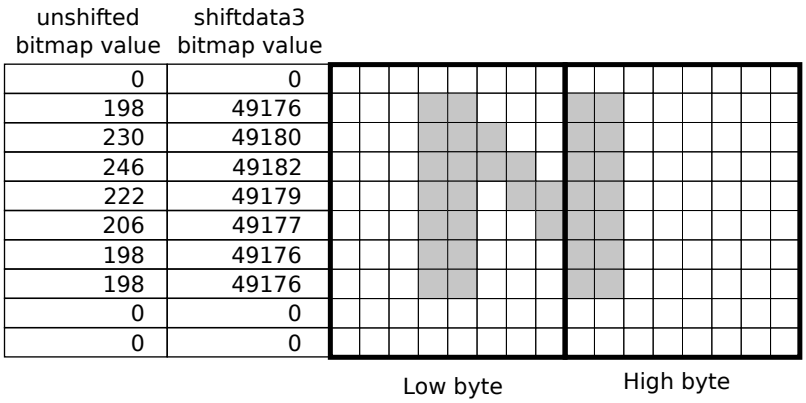


Figure 5.14: Bitshift 'N' over 3 bits using bit shift tables.

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. Finally, the data buffer is copied to VRAM, so it gets displayed on the screen.

```
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
```

5.6.5 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.

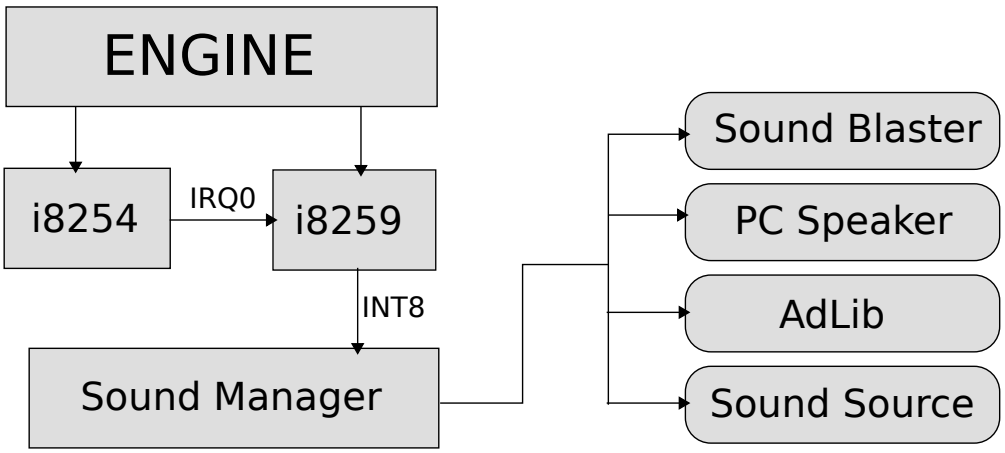


Figure 5.15: Sound system architecture.

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

5.6.6 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.

5.6.7 Softdisk files

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

5.7 Startup

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

After successfully starting the game the intro image is displayed, 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 5.16: Keen Dreams intro screen

5.8 Caching and Compression

The floppy disk is, beside the slowest component of the PC, also constraint in terms of amount of storage sppace. File asset compression ensures significantly less usage of storage capacity than uncompressed files, but also less time for loading the game.

The other challenge is that the total amount of assets did not fit in RAM memory. That is where a cache manager becomes important. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the slower floppy disk. By storing frequently accessed data, games can dramatically reduce their load times.

5.8.1 Asset caching

To manage and keep track of the assets to be loaded into memory, the engine keeps track of caching levels. An array with the size of the total number of assets is maintained to mark if an asset needs to be loaded from disk.

```
byte    grneeded[NUMCHUNKS];
byte    ca_levelbit,ca_levelnum;
```

The index of this array refers to the graphic asset IDs. By using the eight bits the array can maintain the required assets for different levels. The cache manager starts with an empty `grneeded[]` array.

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

Figure 5.17: Initiating `grneeded[]` array.

When new resources needs to be cached in memory, all required assets are marked by setting the current level bit (bit 1).The function `CA_CacheMarks()` loads and decompresses all required graphical assets from disk to memory for the current cache level.

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

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

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

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

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

Figure 5.18: Mark all assets required for the new map in level bit 1.

If, during playing the game, the user opens the control panel (e.g. to pause the game), the assets for the control panel needs to be cached. By increasing the bit level we keep the assets for control panel separated from the map.

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

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

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

The `gr_needed[]` array looks as follows

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

Figure 5.19: Mark all assets required for the control panel in level bit 2.

If the control panel is closed again, it lowers the bit level of `gr_needed[]` to 1, where all required assets for playing the level are memorized. It simply calls the function `CA_CacheMarks()` to reload any map assets removed from memory.

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

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

To avoid often used assets, like fonts and Commander Keen sprites, are reloaded every time from the disk they are are loaded permanent into memory using a non-moveable, unpurgeable block of memory.

```
MM_SetLock (&grsegs[STARTFONT],true);
MM_SetLock (&grsegs[STARTFONTM],true);
MM_SetLock (&grsegs[STARTTILE8],true);
MM_SetLock (&grsegs[STARTTILE8M],true);
for (i=KEEN_LUMP_START;i<=KEEN_LUMP_END;i++)
  MM_SetLock (&grsegs[i],true);
```

5.8.2 Asset compression

As the floppy disk is both limited in speed (100-250 kbps[5]) and storage (3½-inch disk size is either 720KB or 1.44MB), it was important to use file compression. This ensures that the game takes up less space and loads faster. id Software used Huffman compression for all asset files and an additional RLE compression to further shrink map files.

Huffman compression involves changing how various characters are stored. Normally, all characters in a given segment of data are equal and take an equal amount of space to store. However, by making more common characters take up less space while allowing less commonly used characters to take up more, the overall size of a segment of data can be reduced.

To illustrate the various aspects of Huffman compression, the following text, where each character is one byte, will be Huffman compressed.

```
Commander Keen in Keen Dreams
```

The first step is to make a character frequency table.

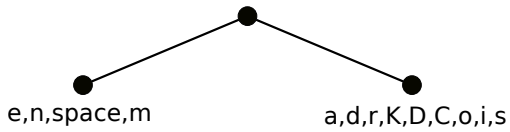
Character	Frequency
e	6
n	4
space	4
m	3
a	2
r	2
K	2

Character	Frequency
d	1
D	1
C	1
o	1
i	1
s	1

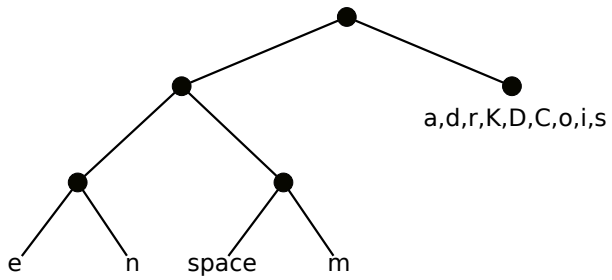
Figure 5.20: Character frequency table.

[5]Kilobit per second is a unit of data transfer rate equal to: 1,000 bits per second or 125 bytes per second.

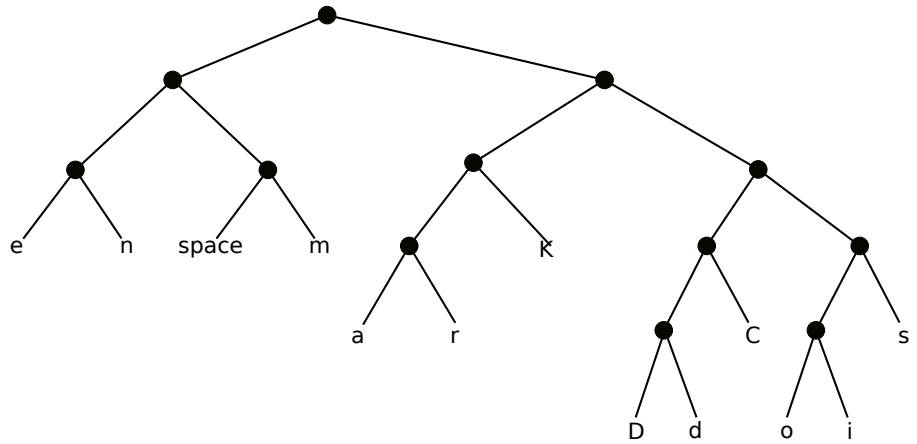
The next step is to create an optimal binary tree, also called a dictionary. This is done by starting with the most common character and checking if it occurs more frequently than all the other characters combined. If not, then the second most common character is added, and so on. In our example, four characters make up more than half of the total number of characters: 'e', 'n', space, and 'm'. All of these characters are placed on the left side of the root node, while the remaining characters are placed on the right side.



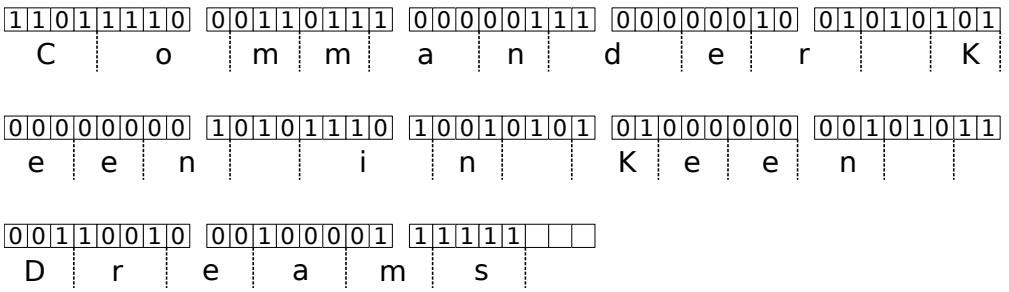
The next step involves creating a new node with left and right branches and repeating the process. The two most common characters in the left node, 'e' and 'n', are assigned to the left branch of this new node. A third node is created, and since there are only two options, each is given a branch. The same process applies to the remaining characters of the left node, resulting in:



Then we do the same for the right node, where 'a', 'd', 'r' and 'K' make up more than half of the total number of characters in the right node. After following the same steps, the final binary tree looks as follows



Now we can encode the entire text moving left (bit 0) and right (bit 1) in the tree, starting from the top node. For example, the character 'e' is `000`, 'm' is `011` and 'o' is `11100`. The complete compressed message in bit form is now



This is a total of 13 bytes used for a 31 byte message, more than 50% reduction. In Commander Keen, the dictionary is part of the source code. Therefore, it is important that the asset files from the shareware version correspond with the dictionary files of the source code[6]. Reading the Huffman-compressed asset file is straightforward. You just read bit by bit from the asset file and follow the dictionary starting from the top node until you hit an end-node. Write the byte to memory and start at the top node in the dictionary again for the next bit in the file.

[6]That's why a specific source code version is mention in section xxx.

```
typedef struct
{
  unsigned bit0,bit1; // 0-255 is a character,
                      // >255 is a pointer to a node
} huffnode;
```

The tile map asset files use a second compression, on top of Huffman. Upon closer inspection of a level, you'll notice large chunks of the same tile. For example, consider the first level you enter (Horse Radish Hill), which contains extensive areas of blue sky. Here, RLE compression comes in useful.

The essence of this compression method is to compress data by saving the "run-length" of the encountered values, essentially storing the data as a collection of length/value pairs. To illustrate, the string 'aaaaaaaabbb' could be compressed to '8a3b' (8 bytes of 'a', 3 bytes of 'b'). The trick with RLE is to ensure that the data does not end up becoming larger after processing. For instance, using pure length/value pairs, 'abracadabra' would become '1a1b1r1a1c1a1d1a1b1r1a'.

In Commander Keen this is solved by using a 'flag'. This special value instructs the program to take specific action upon encountering it. Every value is passed through unchanged until an RLE flag value is encountered. When this flag is read, instead of directly outputting it like other values, two further values are read. The first indicates the number of times to repeat, and the second represents the value to be repeated. The algorithm used in Commander Keen is based on 16-bits (word), and therefore is named RLEW, where the 'W' stands for word. The RLEW flag is defined as `0xABCD`.

The resulting compression is reducing the file size by almost 50%!

Asset	Uncompressed file	Compressed file
Graphics (KDREAMS.EGA)	360KB	213KB
Game maps (KDREAMS.MAP)	210KB	66KB
Total	570KB	279KB

Figure 5.21: Compressed versus uncompressed[7] asset files.

[7] The uncompressed size is an estimation based on the expanded size of each asset.

```
void CA_RLEWexpand (unsigned huge *source, unsigned huge *
    dest,long length, unsigned rlewtag)
{
  unsigned value,count,i;
  unsigned huge *end;
  unsigned sourceseg,sourceoff,destseg,destoff,endseg,
    endoff;

  end = dest + (length)/2;    // length is COMPRESSED length
  sourceseg = FP_SEG(source);
  sourceoff = FP_OFF(source);
  destseg = FP_SEG(dest);
  destoff = FP_OFF(dest);
  endseg = FP_SEG(end);
  endoff = FP_OFF(end);

  asm mov bx,rlewtag          // RLEW tag: 0xABCD
  asm mov si,sourceoff
  asm mov di,destoff
  asm mov es,destseg
  asm mov ds,sourceseg

  expand:
  asm lodsw
  asm cmp ax,bx               // value is RLEW tag?
  asm je  repeat
  asm stosw
  asm jmp next

  repeat:
  asm lodsw
  asm mov cx,ax               // repeat count
  asm lodsw                   // repeat value
  asm rep stosw

  next:                       // Next word value
  [...]
}
```

5.9 Smooth scrolling

Performing a full-screen redraw per frame would kill the CPU, as it would need to update all pixels of all four EGA planes. There is no way to maintain a 60Hz framerate while refreshing the whole screen. If we were to run the following code, which simply fills all memory banks, it would run at 5 frames per seconds.

```
# define SC_INDEX   0x3C4
# define SC_DATA    0x3C5
# define SC_MAPMASK 0x02

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

void selectPlane (int plane) {
  outp ( SC_INDEX , SC_MAPMASK );
  outp ( SC_DATA , 1 << plane );
}

void WriteScreen(void){
  int i, bank_id;

  for (bank_id = 0 ; bank_id < 4 ; bank_id++) {
    selectPlane(bank_id);
    for (i = 0; i < 40 * 200; i++) {
      EGA[i] = 0x00;
  }
}
```

So how did they create a smooth scrolling game with these limitations? By using some EGA hardware tricks and by changing only that part of the screen that has changed.

5.9.1 EGA Virtual Screen and Pel Panning

The EGA card 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 virtual screen width of up to 4096 pixels and then use the physical screen as a window onto any part of the virtual screen. What's more, a virtual screen can have any logical height up to the capacity of the VRAM memory. The code below illustrates how to change the logical width.

```
CRTC_INDEX  = 03D4h
CRTC_OFFSET = 19

;==============
;
; set wide virtual screen
;
;==============

mov dx,CRTC_INDEX
mov al,CRTC_OFFSET
mov ah,[BYTE PTR width]  ;screen width in bytes
shr ah,1                 ;register expresses width
                         ;in word instead of byte
out dx,ax
```

The area of the virtual screen displayed at any given time is selected by setting the display memory address at which to begin fetching video data. This is set by way of the CRTC Start Address register. The default address is `A000:0000h`, but the offset can be changed to any other number.

```
CRTC_INDEX   = 03D4h
CRTC_STARTHIGH   = 12

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

  cli                       ;disable interrupts

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

  sti                       ;enable interrupts

  ret
```

Panning down a scan line requires only that the CRTC start address is increased by the logical width. Horizontal panning is possible by increasing the start address by one byte. In EGA's planar graphics modes, the eight bits in each byte of video RAM correspond to eight consecutive pixels on-screen. That means the screen moves horizontally in steps of 8 pixels, which is coarse, not smooth scrolling. Luckily there is another EGA register to resolve this.

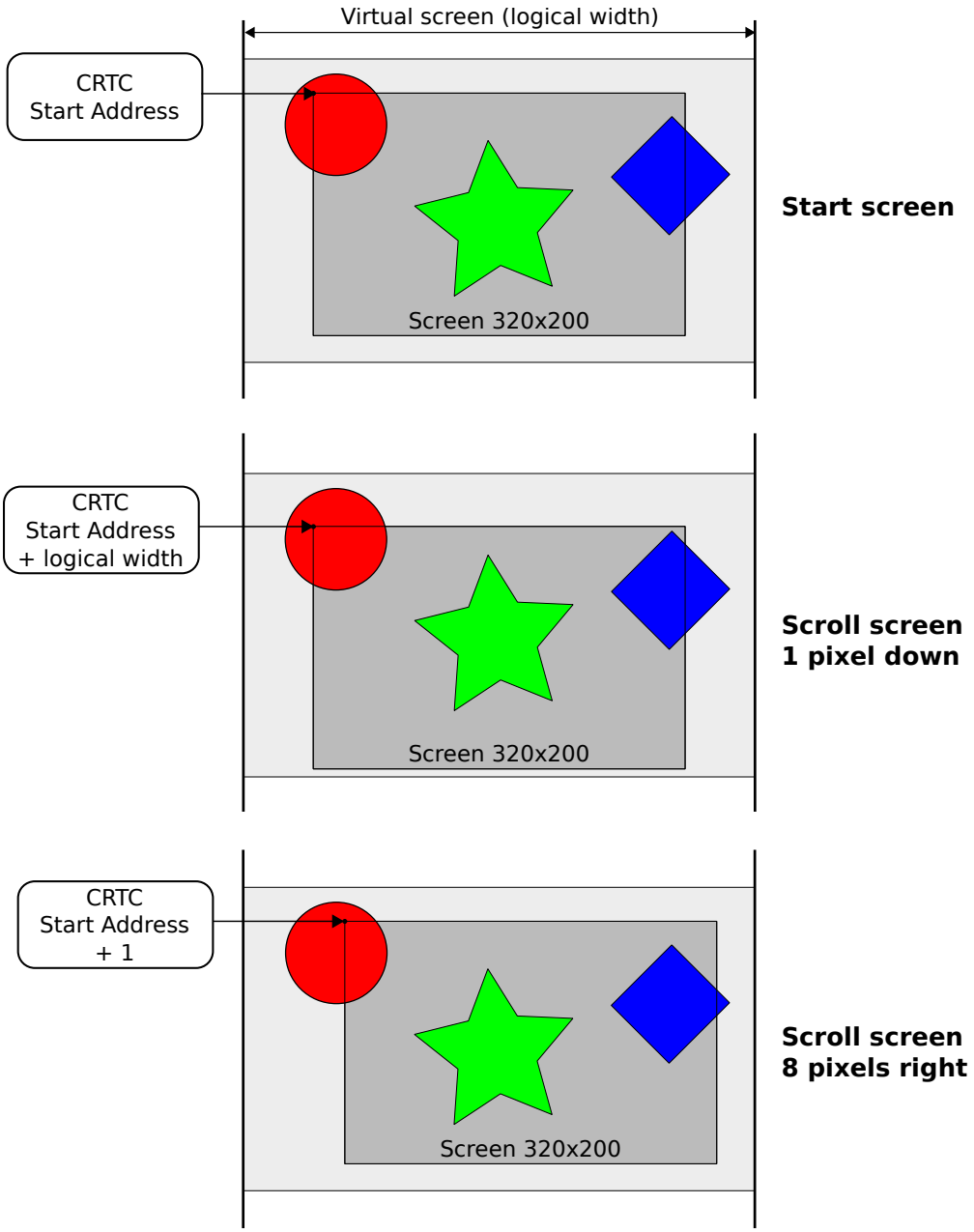


Figure 5.22: EGA screen scrolling by updating the CRTC Start Address.

5.9.2 Horizontal Pel Panning

Smooth horizontal 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 output of the color palette controls the displayed pixels; this is normal operation. When bit 5 is 0, video data doesn't come from the color palette, 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
```

Smooth horizontal scrolling on EGA should be viewed as adjusting the CRTC Start Address and fine horizontal pel panning adjustments in the 8-pixel range. The scrolling is achieved by the following steps:

- calculate the panning in pixels for both x- and y-direction
- the smooth y-panning is defined by adding `logical width * y` to the CRTC start address
- increase the CRTC start address width `x/8` bytes for coarse horizontal scrolling.
- the horizontal fine tuning is then adjusted by `(x % 8)` using horizontal pel panning

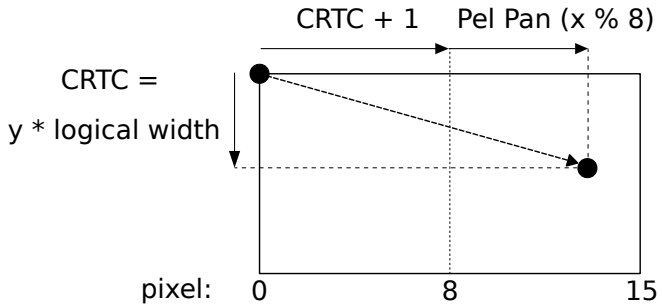


Figure 5.23: Smooth scrolling in EGA.

5.10 Adaptive Tile Refreshment

So far, we have built a virtual screen in VRAM allowing smooth one pixel moves in both axes using only EGA registers. But once it reaches an edge, the virtual screen must be reset. And it cannot be a full screen redraw because it would drop the framerate to 5fps.

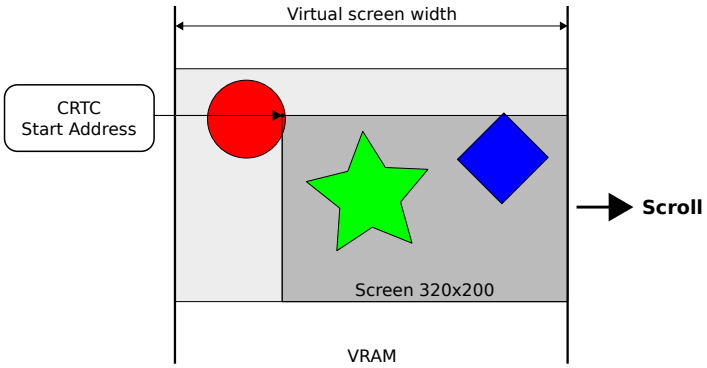


Figure 5.24: Screen reaching the edge of virtual screen.

An option to resolve screen refresh is loading the entire level into VRAM memory and setting the logical width to the level width. By simply updating the CRTC Start Address and Horizontal Pel Panning registers, the game scrolls smoothly through the virtual screen. Unfortunately, a level won't fit into the VRAM. The first level, *Horsh Radish Hill*, is 2176 pixels wide and 592 pixels high, which means a total of 2176*592/2 = 644KB is required to store the entire level in VRAM. Since we only have 256KB available, this is not a feasible option.

Trivia : At the time of writing this book, most video cards contain more than 1GB VRAM, sufficient to store all Commander Keen levels at once in VRAM.

This is where John Carmack's invention starts. The scrolling engine is based on a simple yet powerful technology called *Adaptive Tile Refreshment (ATR)*. The core idea is to refresh only those areas of the screen that need to change. The screen is divided into tiles of 16x16 pixels. On a screen with 320x200 pixels, it means a grid of 20x13 tiles (actually it is 12.5 tiles high, but we round to full tiles).

Let's look at *Commander Keen 1: Marooned on Mars* in Figure 5.25. 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 scrolled one tile (16 pixels) to the right through the world. They look almost identical to the naked eye, don't they?

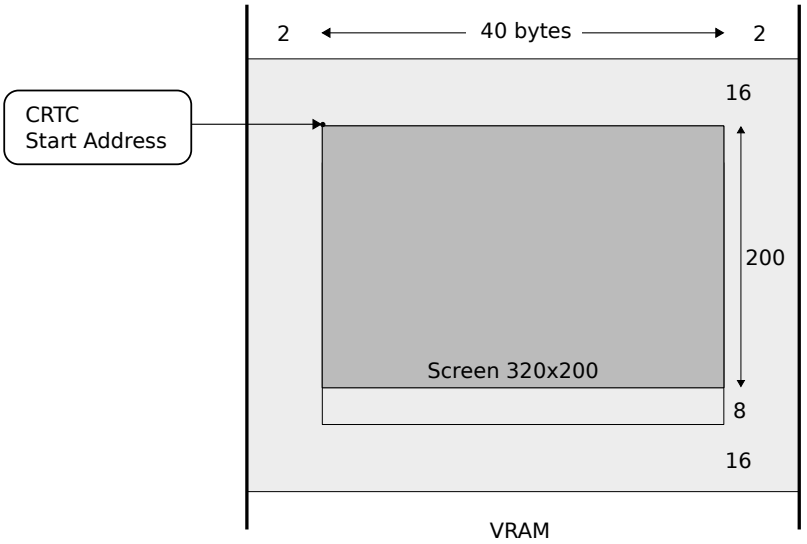
Now, if we perform a difference on both images, you can see which tiles need to be changed upon screen refresh. The trick behind the scrolling is to only redraw tiles that actually changed after panning 16 pixels (one tile). For matching tiles there is nothing to do and they are skipped entirely. In Figure 4.21, there are large swathes of constant background tiles. In total, only 69 tiles out of the 260 tiles need to be refreshed, which is 27% of the screen!

Figure 5.25: Start of the world, moved one tile to the right and difference.

5.11 ATR in action: Commander Keen 1-3

This section explains how ATR is working for the first three episodes of Commander Keen. Since there is no source code released for the first three episodes of Commander Keen, ATR will be explained based without code examples. The later versions of Commander Keen, including Keen Dreams, used a different, improved engine which will be explained in the next section[8].

The EGA screen in mode `0xD` has a resolution of 320x200 pixels, which is 40x200 bytes. Let's extend the height by 8 bytes to have a height of 208 pixels, so the screen fits nicely in 20x13 tiles of 16x16 pixels. By making the virtual screen one tile higher (16 bytes) and one wider (2 bytes) on each side of the screen, the engine can scroll up to 16 pixels to any direction of the screen without any tile refresh, by simply adjusting the CRTC Start Address and Pel Pan registers.



Now, let's have a closer look at the EGA VRAM setup. The video memory is organized into three virtual screens:

- Page 0 and 1, which are used to switch between buffer and visible screen. The idea between two pages (double buffer) is that the code can draw in the second buffer 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.

[8] https://retrocomputing.stackexchange.com/questions/22175/what-is-adaptive-tile-refresh-in-the-context-of-commander-keen

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

Each virtual screen has a size of 44*240*4=42,420 bytes, or 10,560 bytes per memory bank. So within a 256KB EGA card there is enough VRAM available to keep all three virtual screens in memory. The page that is actually displayed at any given time is selected by setting the CRTC Start Address register at which to begin fetching video data.

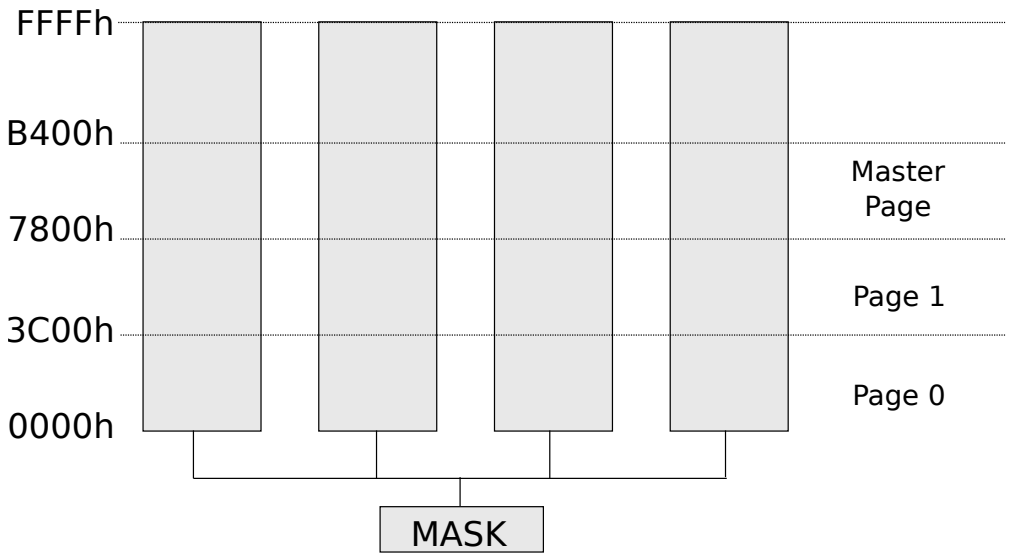


Figure 5.26: Virtual screen layout on EGA card.

For both the buffer and visible screen a tile index array is created to maintain which tiles are changed since last refresh.

```
byte    update[2][UPDATESIZE];
```

The steps to refresh the screen are as follows:

1. Check if the player has moved one tile in any direction.
2. Validate which tiles have changed and copy these respective tiles to the master page and mark the tiles in both tile index arrays.
3. Refresh the buffer page by scanning the tile index array. If a tile is marked, copy the tile from the master page to the buffer page.
4. Switch the view and buffer page by adjusting the CRTC Start Address and Pel Panning registers.

For now let's ignore sprites on the screen, that will be explained in the next section. In the next four screenshots, we take you step-by-step through each of the stages. The player has reached the edge of the virtual screen and moves further to the right.

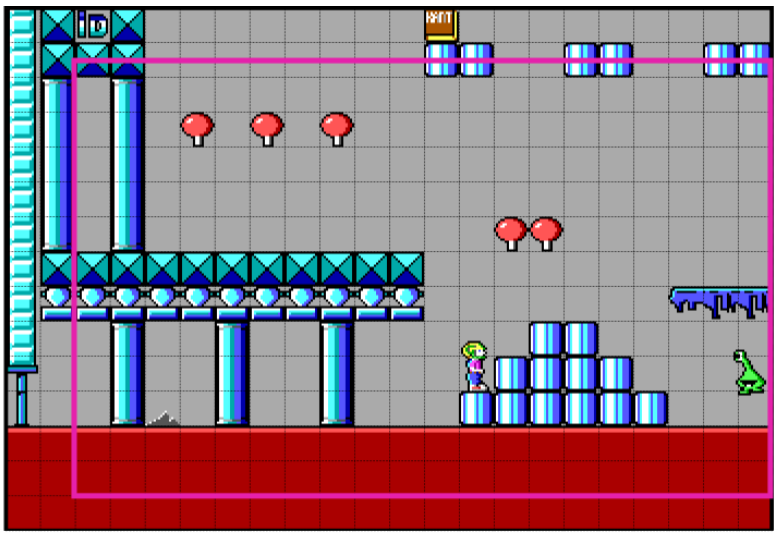


Figure 5.27: Start: Reach the end of the virtual screen.

The engine keeps track of which tile numbers are in the virtual screen. Since the engine only refresh at tile size granularity, it can determine extremely fast what has changed on the screen by comparing tile numbers. If the tile number has changed, the tile is updated by copying tiles from RAM into the VRAM master page. The changed tiles are marked with a '1' in both tile arrays, which means it needs to be updated upon next refresh.

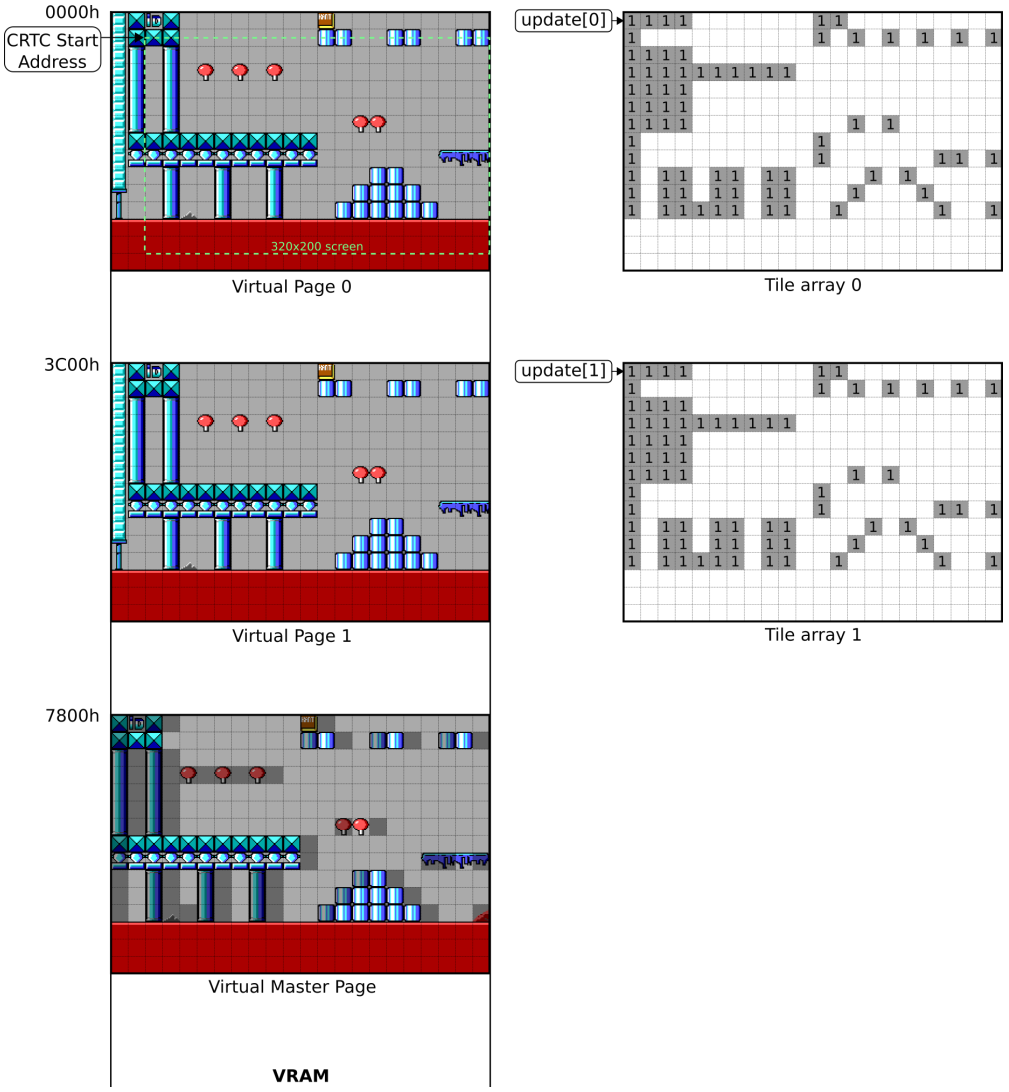


Figure 5.28: Update tiles in master page and update both tile arrays.

The next step is to scan all tiles in buffer tile array (array 1) and for each tile marked '1', copy the corresponsing tile from master to virtual page 1 page.

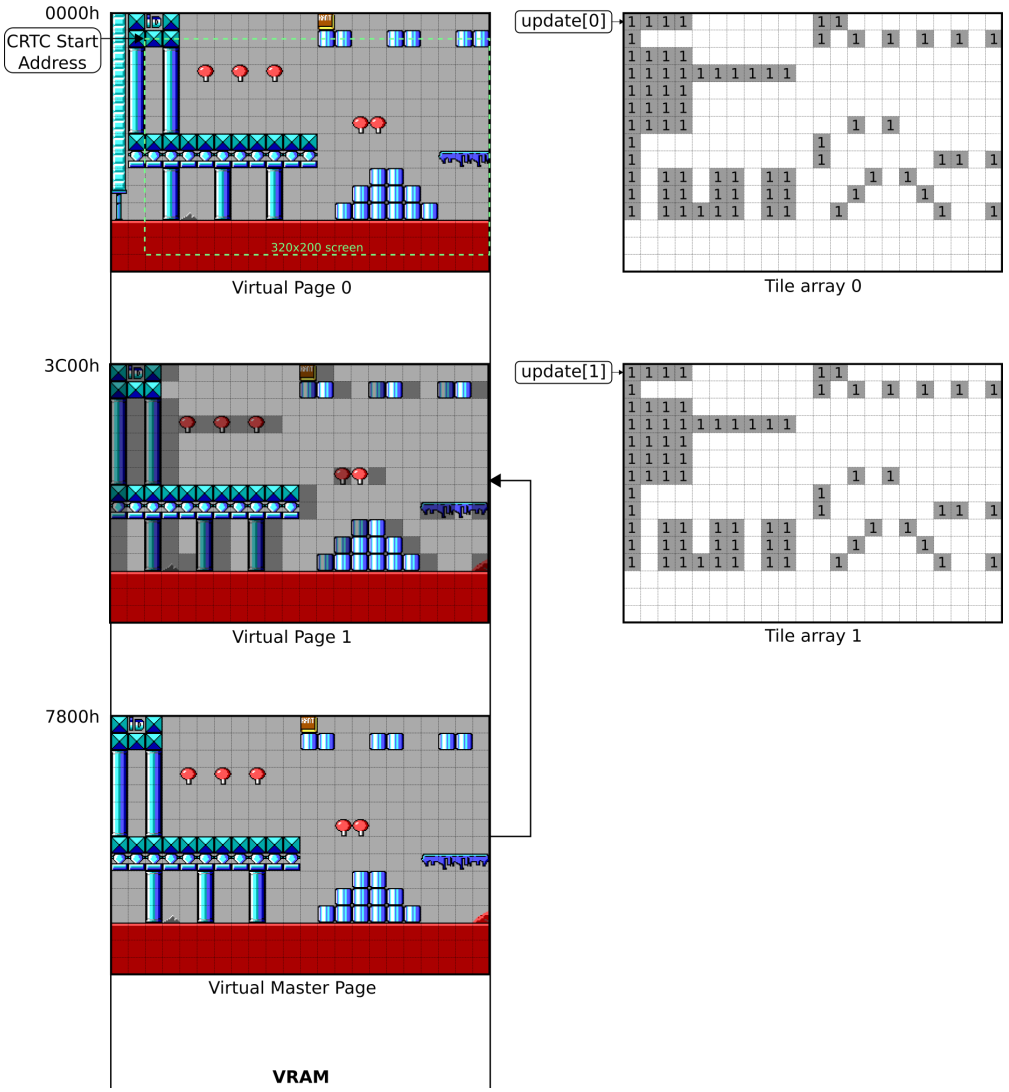


Figure 5.29: Copy changes tiles from virtual master page to page 1.

In the last step, point the visible screen to virtual page 1 by updating the CRTC start address. Finally, tile array 1 is cleared to '0'. Now the first step is repeated, but this time virtual page 0 acts as the buffer screen. Note that after swapping, tile array 0 keeps marked tiles from last update. This makes sense, as the current buffer page is not yet refreshed since it was displayed in the previous refresh cycle.

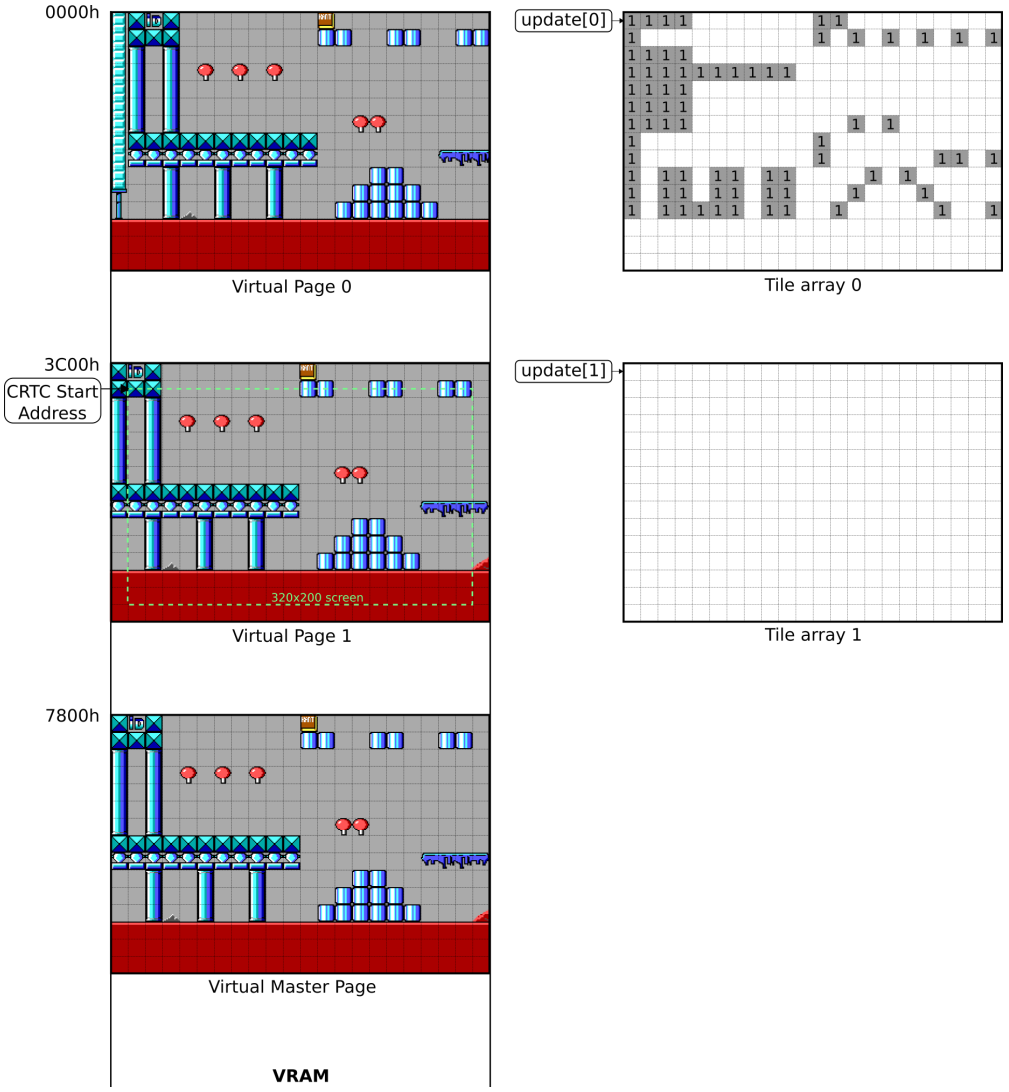


Figure 5.30: Update CRTC Start Address to virtual page 1 and empty tile array 1.

Now the buffer is refreshed and the CRTC address is updated, the final step is fine adjustment using the Pel Panning register.

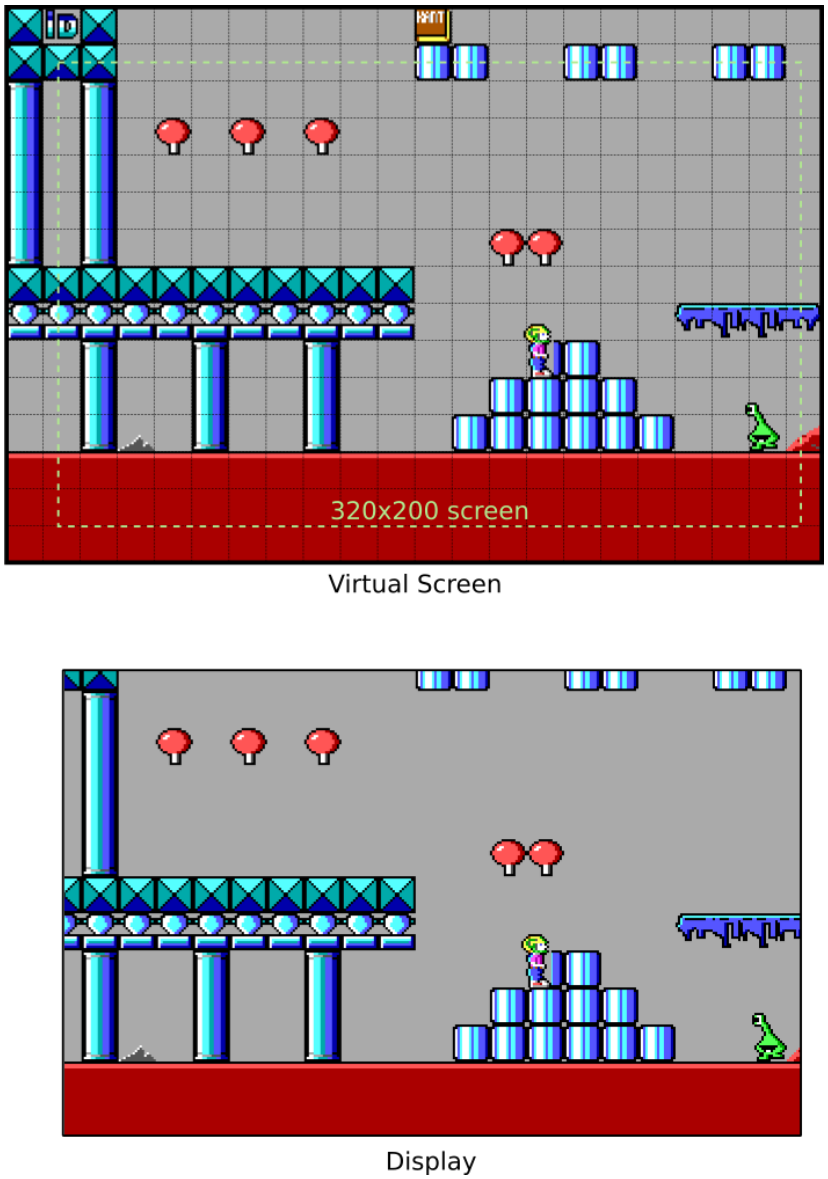


Figure 5.31: End: Screen is refreshed and scrolled to the right.

5.11.1 Wrap around the EGA Memory

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

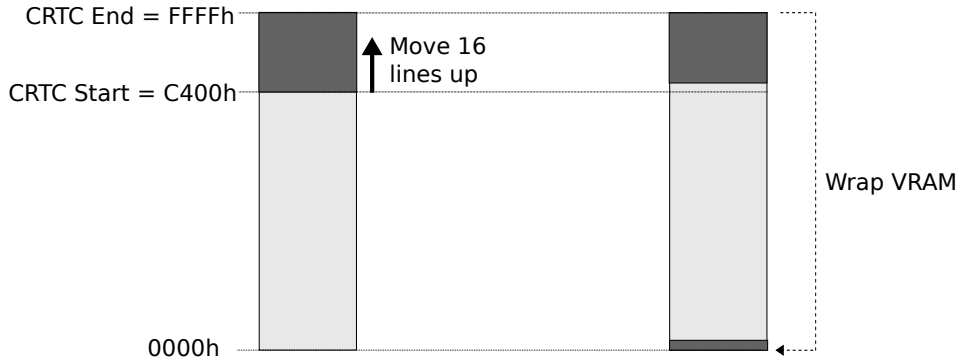


Figure 5.32: Wrap virtual screen around the EGA memory

> I finally asked what actually happens if you just go off the edge [OF THE VRAM]?
>
> If you take your [CRTC] start and you say OK, I can move over and I get to what should be the bottom of the memory window. [...] What happens if I start at 0xFFFE at the very end of the 64k block? It turns out it just wraps back around to the top of the block.
>
> I'm like oh well this makes everything easy. You can just scroll the screen everywhere and all you have to draw is just one new line of tiles.
>
> It just works. We no longer had the problem of having fields of similar colors. It doesn't matter what you're doing, you could be having a completely unique world and you're just drawing the new strip.
>
> **John Carmack**[9]

[9] An explanation further elaborated during an interview with Lex Fridman in 2022.

There was however an issue with the introduction of Super VGA cards, which had typically more than 256kB RAM[10]. This resulted in crippled backwards compatibility and the wrapping around `0xFFFF` did not work anymore on these cards.

There is an easy solution to resolve this issue. As you can see in Figure 5.26 on page 131, the space between `0xB400` and `0xFFFF` is not used and contains enough space for another virtual screen. In case the start address is between `0xC400` and `0xFFFF` the corresponding screen is copied to the opposite end of the buffer, as illustrated in Figure 5.33.

> “ I was in a tough position. Do I have to track every single one of these [SUPER VGA CARDS] and it was a madhouse back then with 20 different video card vendors with all slightly different implementations of their non-standard functionality. Either I needed to natively program all of the cards or I kind of punt. I took the easy solution of when you finally did run to the edge of the screen I accepted a hitch and just copied the whole screen up.
>
> **John Carmack[11]** ”

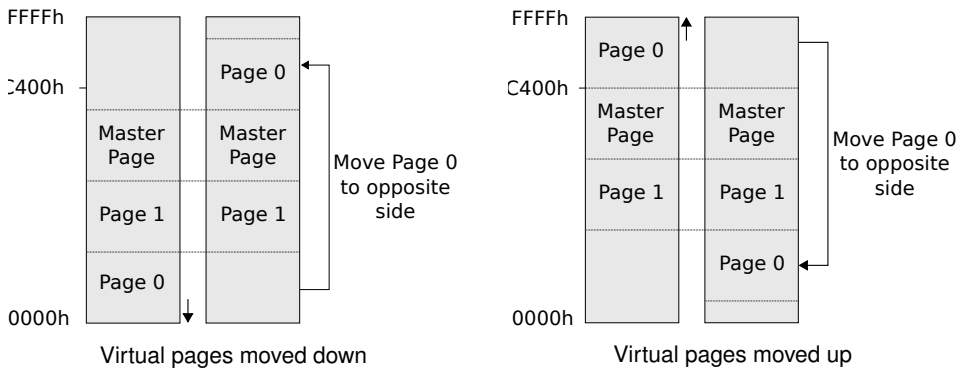


Figure 5.33: Move screen to opposite end of VRAM buffer

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

[11] Same interview with Lex Fridman in 2022.

```
#define SCREENSPACE   (SCREENWIDTH*240)
#define FREEEGAMEM    (0x100001-31*SCREENSPACE)

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

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

But how can we copy four VRAM planes fast enough, without noticing any performance hit? As explained in Section 3.3.7 each pixel is encoded by four bits, which are spread across the four EGA banks. To copy eight pixels, byte-aligned, one would have to do four read and four writes.

Having a closer look at the EGA card one notice there is a latch placed in front of the ALU, which can be re-purposed for the greater good. With a little creativity the ALU in front of each bank can be setup to use only the latch for writing[12]. With such a setup, upon doing one read, four latches are populated at once and four bytes in the bank are written with only one write to the RAM. This system allows transfer from VRAM to VRAM 4 bytes at a time. Now it is possible to copy the entire buffer fast enough, without notifying any performance impact.

[12]The mask trick is discussed in *Game Engine Black Book: Wolfenstein 3D* for the VGA card. The trick is the same for the EGA card.

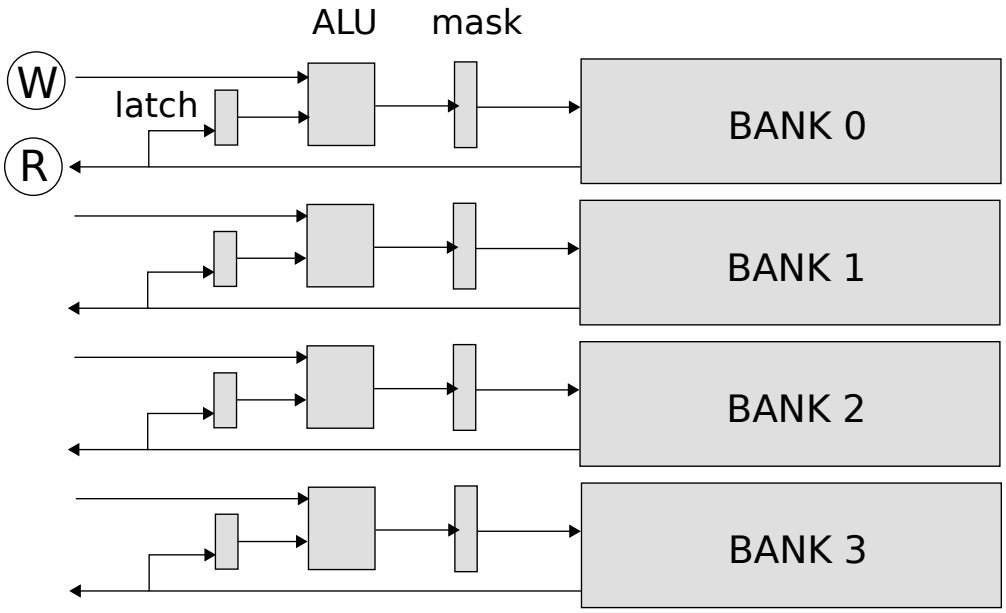


Figure 5.34: Latches memorize read operations from each bank. The memorized value can be used for later writes.

```
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
```

5.11.2 Adaptive tile refreshment in Commander Keen in Keen Dreams

The EGA memory wrapping results in an improved, more simplified Adaptive Tile Refreshment algorithm. First, a tile array is defined by making the display one tile higher and wider, which is named the view port. Now the engine can scroll the display up to 16 pixels to the right and bottom without any tile refreshment, simply by adjusting the CRTC Start Address and the Pel Pan register. Next, an additional column is attached to the port view to support scrolling to left and right. Finally, the tile array is further increased to have enough space to float the entire view port up to two tiles in all directions.

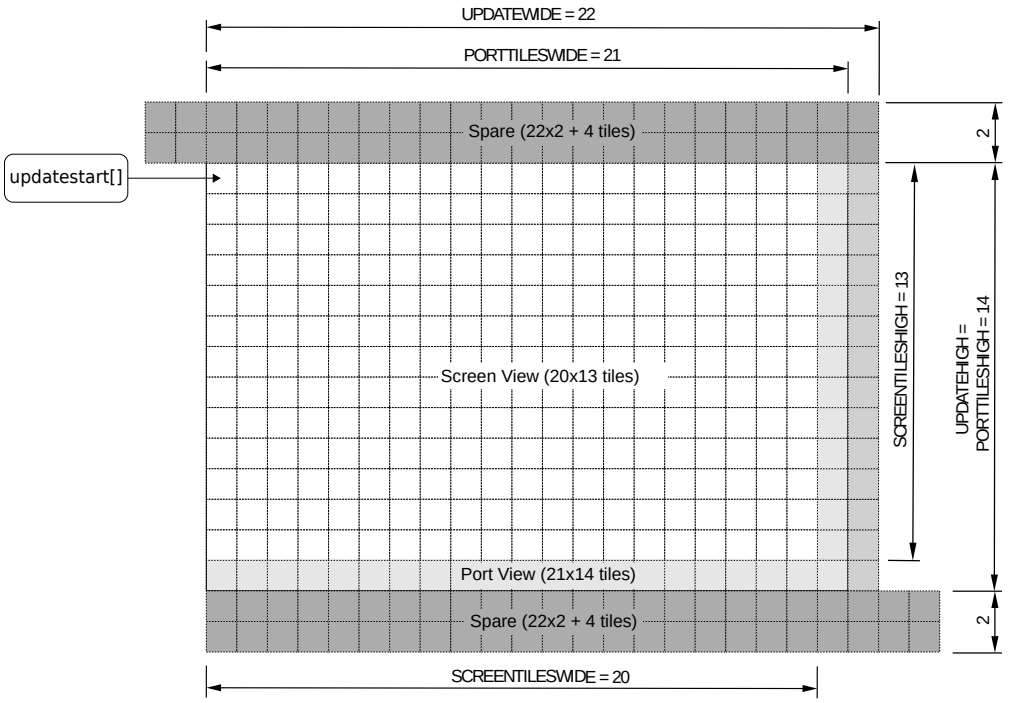


Figure 5.35: Tile array layout.

A full refresh cycle, including sprite updates, take place as follows:

1. Check if the player has moved one tile in any direction.
2. Copy the new column or row of tiles to the master page, update the tile array pointers and mark the tiles in both tile arrays.
3. Refresh the buffer page by scanning the tile index array. If a tile is marked, copy the tile from the master page to the buffer page.

4. Iterate through the sprite removal list and copy corresponding image block from master to buffer page.
5. Iterate through the sprite list and copy corresponding sprite image block to the buffer page.
6. Switch the view and buffer page by adjusting the CRTC Start Address and Pel Panning registers.

```
void RF_Refresh (void)
{
    updateptr = updatestart[otherpage];

    RFL_AnimateTiles ();// update animated tiles

    // copy newly scrolled and animated tiles
    // from the master to buffer screen
    EGAWRITEMODE(1);
    EGAMAPMASK(15);     // write 4 bytes of VRAM at once
    RFL_UpdateTiles (); // copy from master to buffer page
    RFL_EraseBlocks (); // remove sprites

    // update sprites
    EGAWRITEMODE(0);
    RFL_UpdateSprites ();

    // display the changed screen (swap view and buffer)
    VW_SetScreen(bufferofs+panadjust,panx & xpanmask);
}
```

The starting situation is that all three virtual pages have exactly the same view port, where virtual page 0 is currently shown on the screen. Both tile arrays are empty, meaning no tile updates upon next refresh cycle. The player is reaching the edge of the virtual screen and moves further to the left. Notice the master page only contains a static view, without any sprites.

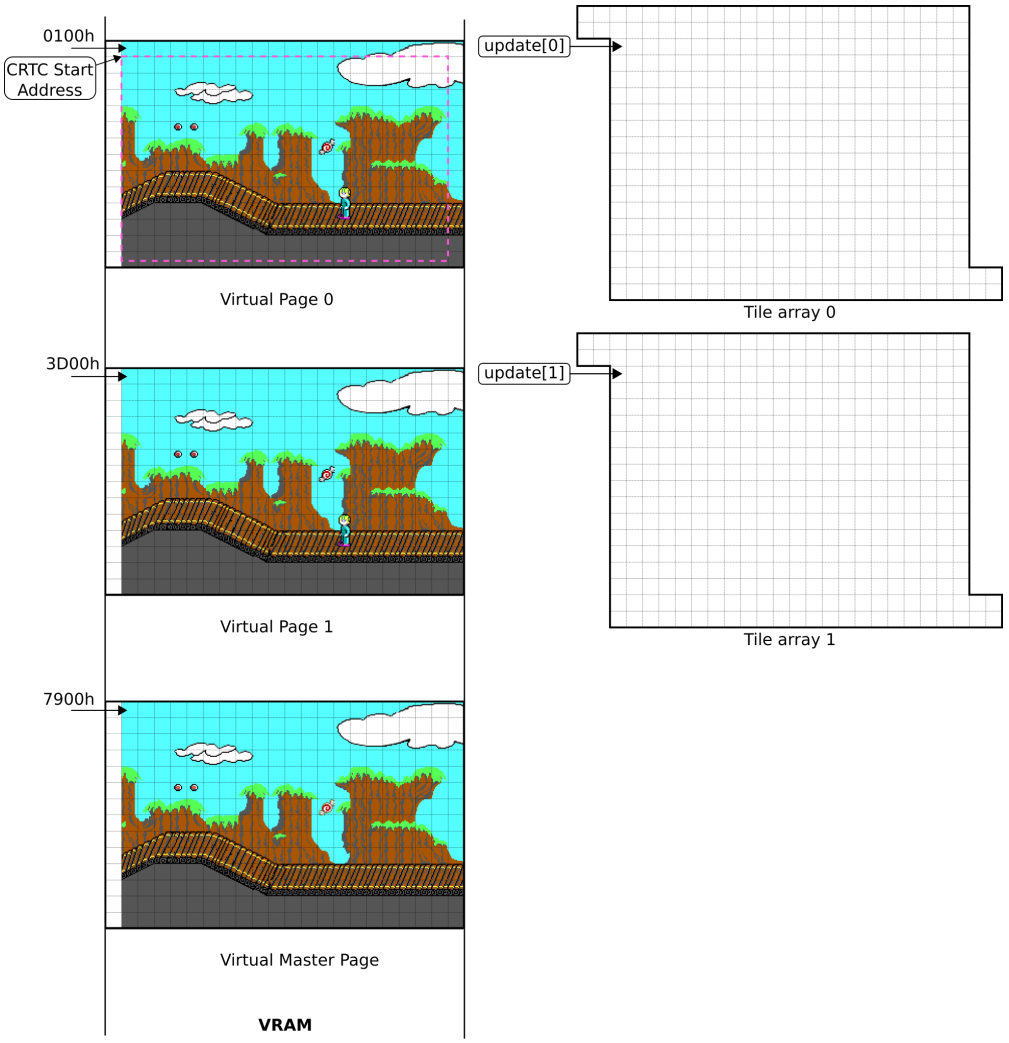


Figure 5.36: VRAM and tile arrays before scrolling to the left.

Both VRAM and tile arrays pointers are moved to the left to introduce a new column of tiles. The VRAM pointer is lowered by 2 bytes (1 tile width) and the tile array pointer is lowered 1 byte. Next, a new column of tiles is copied to the virtual master page, and the most left column in both tile arrays is marked with '1', so it is updated upon next refresh cycle. Finally, animated tiles are copied to the virtual master page and the corresponding tiles in the tile array are marked with '1' as well.

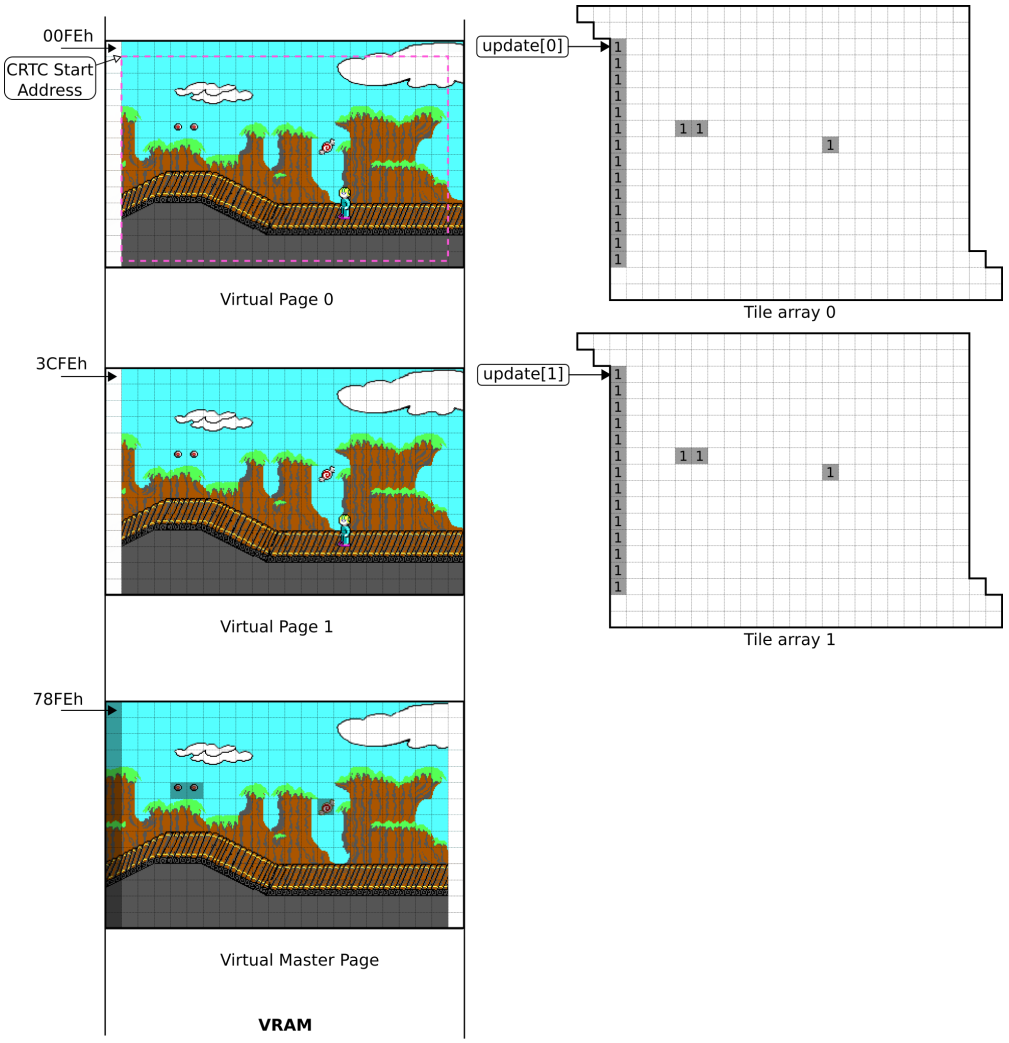


Figure 5.37: Update Virtual master with new left column and animated tiles.

In the next step the engine scans all ’1’ and copy the corresponsing tiles from master to the buffer page.

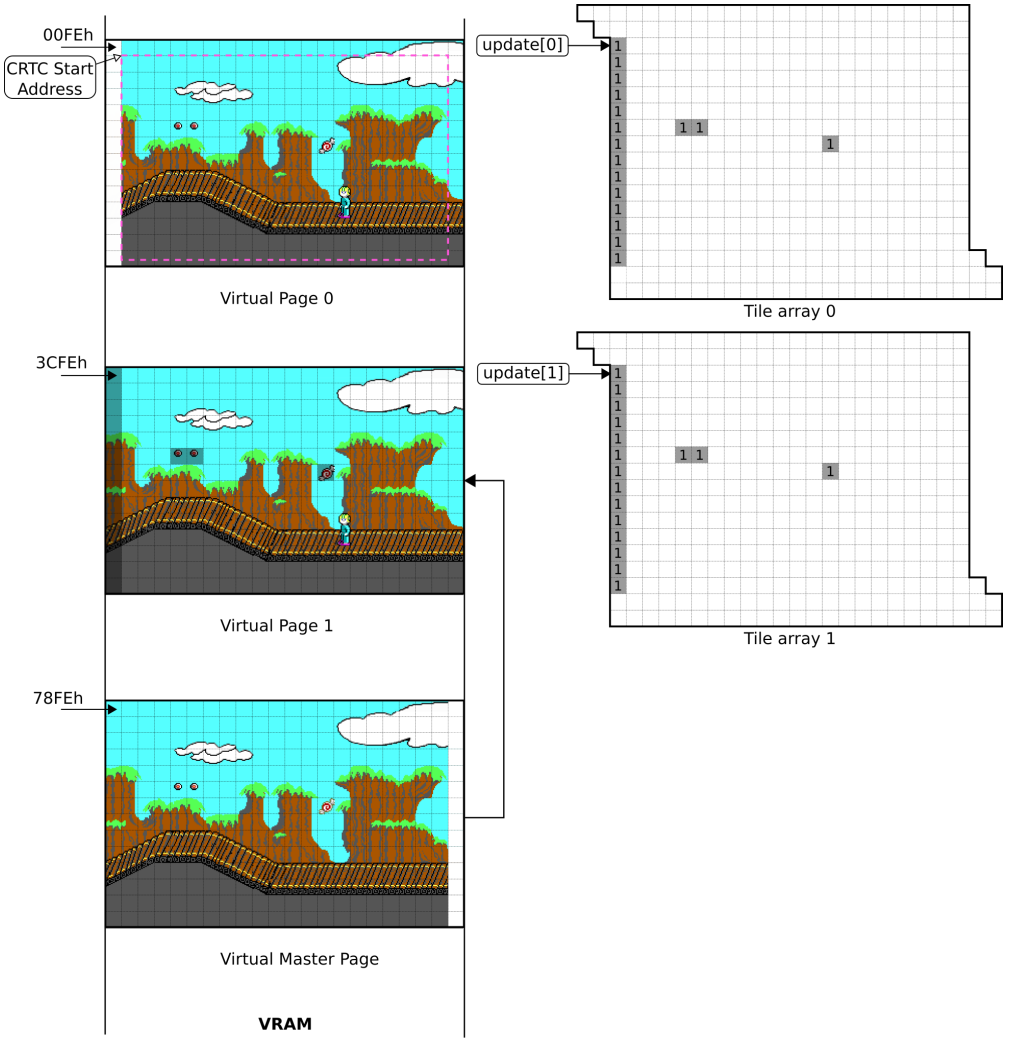


Figure 5.38: Copy changed tiles from master to buffer page.

A list of removed sprites is maintained by the sprite removal block list.

```
typedef struct
{
    int     screenx,screeny;
    int     width,height;
} eraseblocktype;

eraseblocktype  eraselist[2][MAXSPRITES],*eraselistptr[2];

void RFL_EraseBlocks (void)
{
  eraseblocktype  *block,*done;
  unsigned  pos;

  block = &eraselist[0][0];
  done = eraselistptr[0];

  while (block != done)
  {
    [...]

  //
  // erase the block by copying from the master screen
  //
    pos = ylookup[block->screeny]+block->screenx;
    block->width = (block->width + (pos&1) + 1)& ~1;
    pos &= ~1;          // make sure a word copy gets used
    VW_ScreenToScreen (masterofs+pos,bufferofs+pos,
      block->width,block->height);

    [...]
    block++;
  }
}
```

Each removal block contains the location and size to be removed, which is done by copying the specific section from the master screen to the virtual page. The corresponding tiles which overlap with the removal block are marked with a '2'.

Trivia : The '2' marking is nowhere used in the engine, most likely it is a used in the original ATR engine.

The final step in updating the buffer is to copy sprites to the new location. Each corresponding tile is marked with a '3' in the tile array.

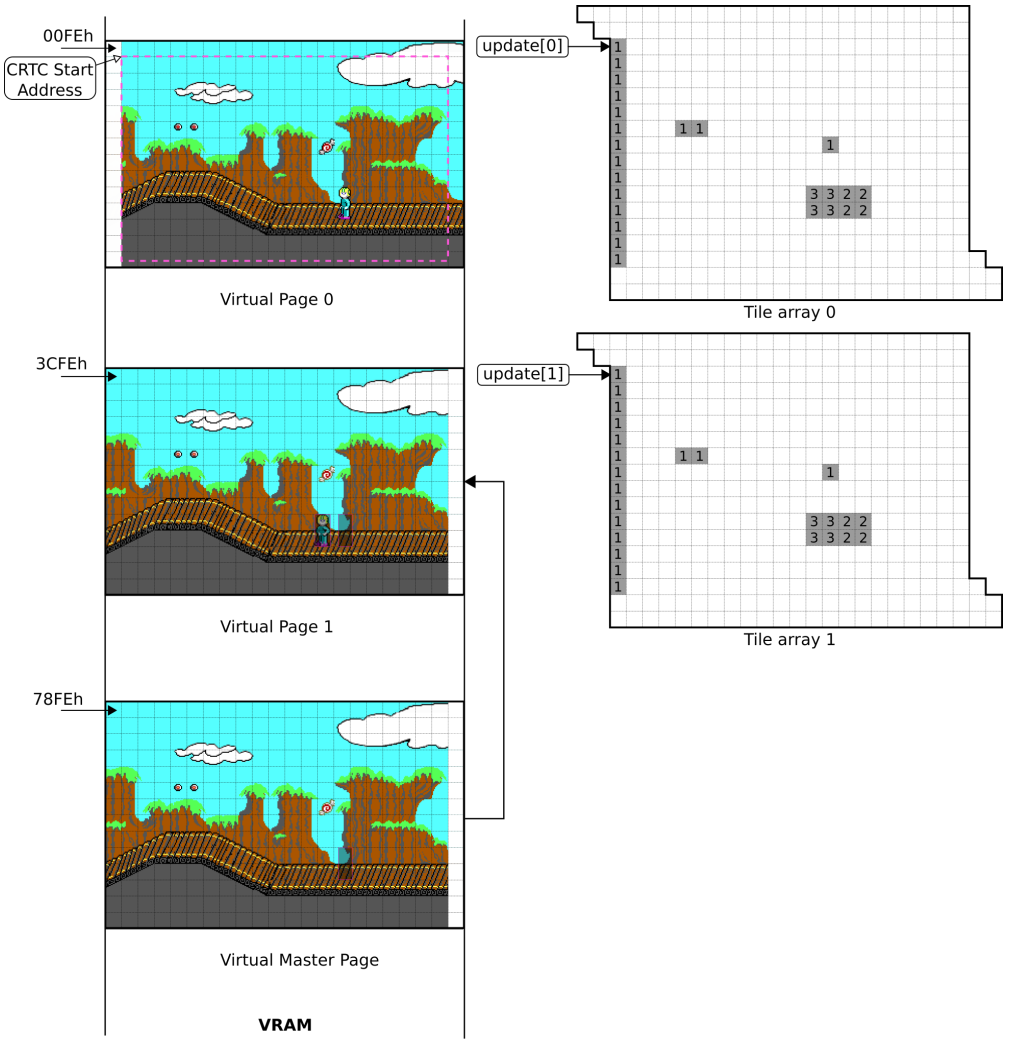


Figure 5.39: Add updated sprites to the buffer screen.

After the buffer has been refreshed, the engine only has to update the CTCR start address to virtual page 1. The visible tile array is emptied and the pointer is reset to the starting location, while the new buffer array (virtual page 0) keeps the tiles marked, since the corresponding screen has not been refreshed yet. Now, the entire refresh cycle starts over again, where virtual page 0 is now the buffer.

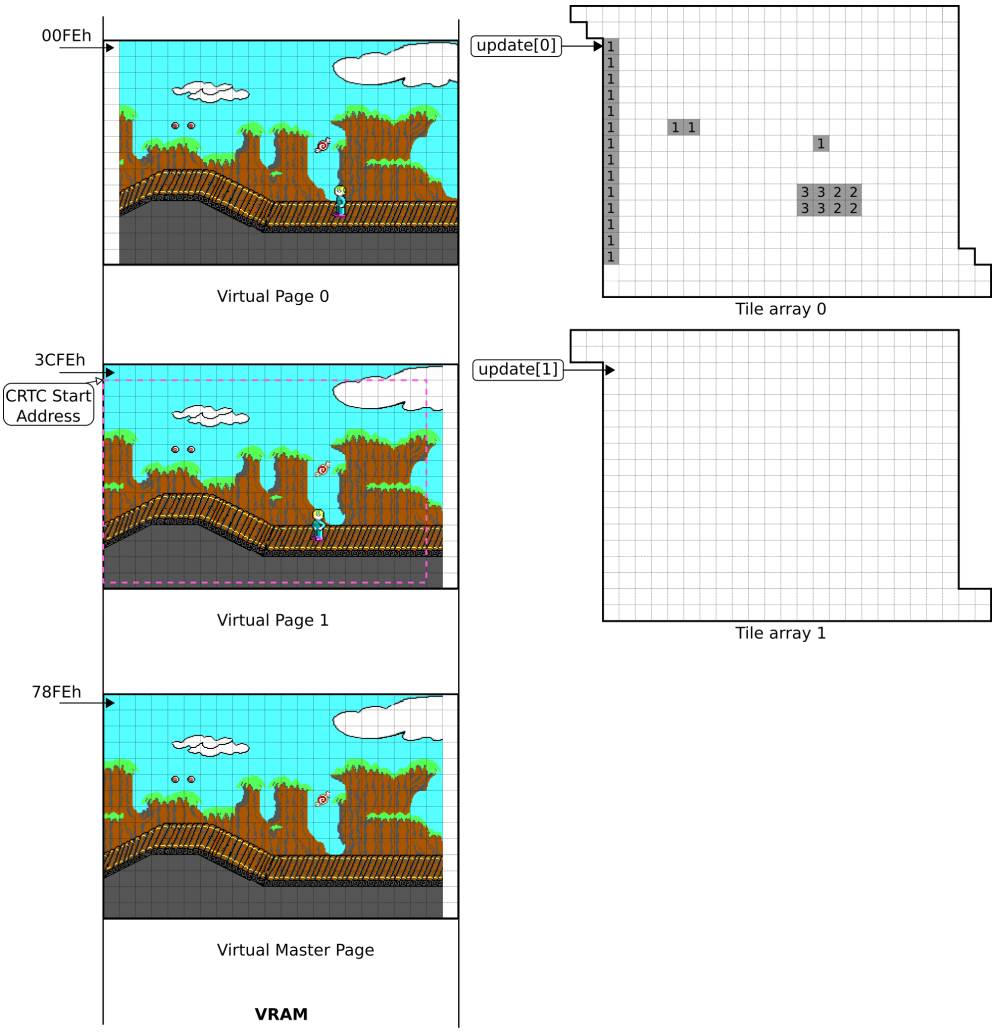


Figure 5.40: Swap buffer and visible screen by updating CRTC start address.

The final step is to update the horizontal fine-pixel alignment by setting the pel pan register. *Et voila,* the screen has scrolled to the left. Notice how much more efficient scrolling takes place compared to the orginal ATR engine in Figure 5.30. Only 8% of the screen requires a tile refresh!

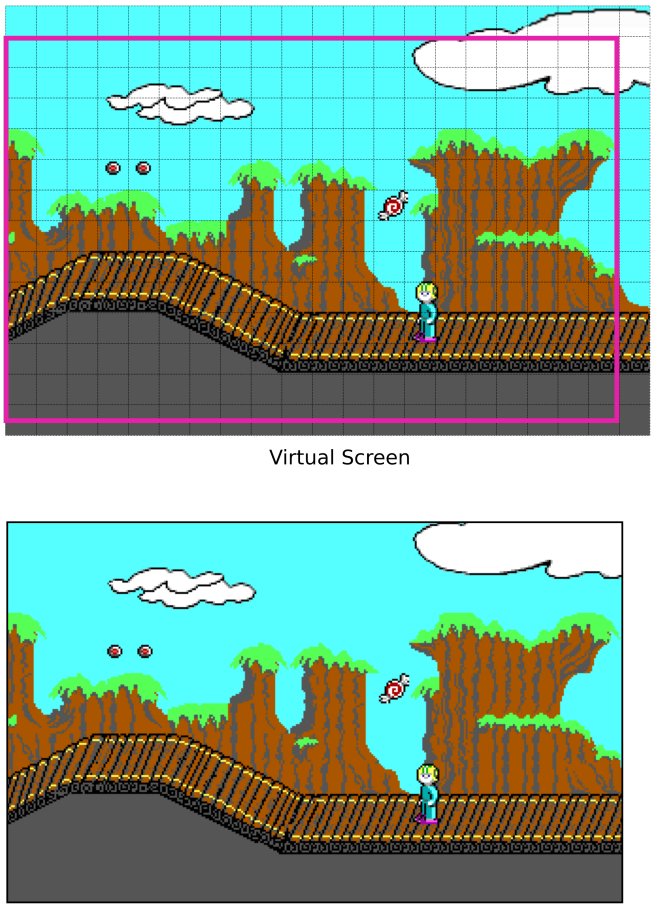
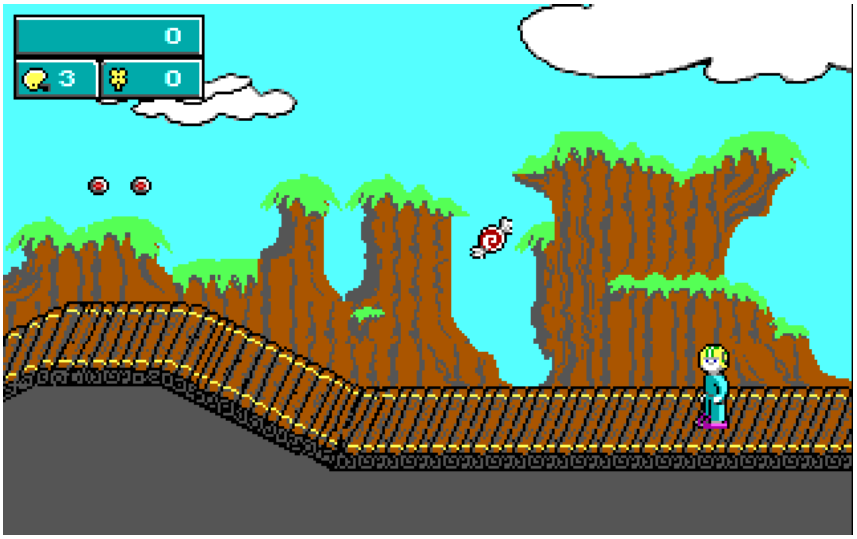


Figure 5.41: Start: Scroll screen to the left

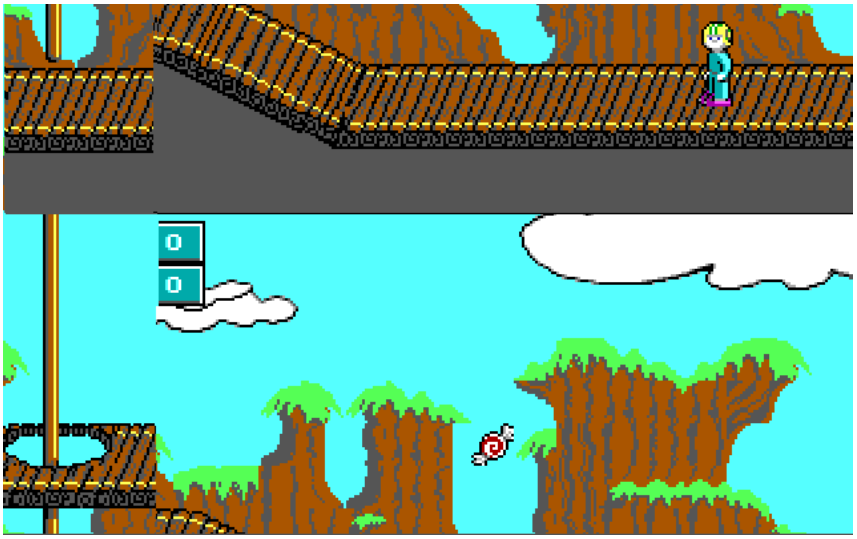
If the player has moved again to the left, tile array 0 pointer will be lowered for the second time, which explains why we need enough space to float the entire viewport up to two tiles in all directions.

5.11.3 Screen refresh

Flipping between the pages is as simple as setting the CRTC start address registers to page 0 or page 1 starting point. However, there is one issue to solve. If you were to run it, every once in a while the expected screen shown below...



...would instead appear distorted:



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

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

```
CRTC_INDEX   = 03D4h
CRTC_STARTHIGH   = 12

  cli                       ;disable interrupts
  mov   cx,[crtc]           ;[crtc] is start address
  mov   dx,CRTC_INDEX       ;set CRTR register
  mov   al,CRTC_STARTHIGH   ;start address high register
  out   dx,al
  inc   dx                  ;port 03D5h
  mov   al,ch
  out   dx,al               ;set address high

  ;******** VERTICAL RETRACE STARTS HERE  !!!!!!!  ****
  ;******** AND SHOWS 2 PARTIAL FRAMEBUFFERS **********

  dec   dx                  ;set CRTR register
  mov   al,0dh              ;start address low register
  out   dx,al
  mov   al,cl
  inc   dx                  ;port 03D5h
  out   dx,al               ;set address low
  sti                       ;enable interrupts

  ret
```

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

The trick is to update the start address sufficient far away from when the vertical retracement starts. So we're looking for a signal that tells us it just finished a horizontal or vertical retrace and started a scan line, far enough away from vertical retrace so we can be sure the new start address will get used at the next vertical sync. This signal is provided by the Display Enable status signal via the Input Status 1 Register, where a value of 1 indicates the display is in a horizontal or vertical retracement[13].

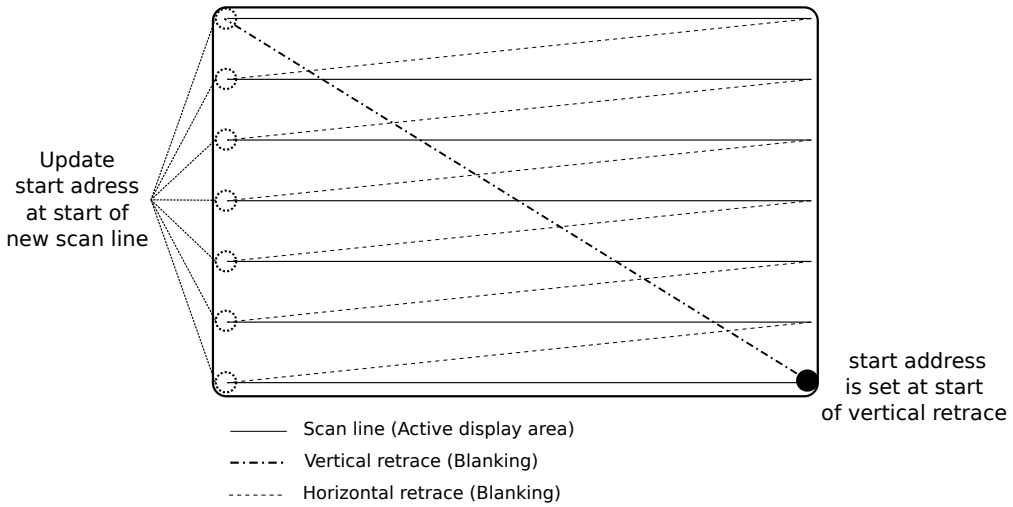


Figure 5.42: Update CRTC start address at beginning of new scan line.

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

[13]Documentation is a bit unclear here. The IBM technical documentation for VGA explains retrace takes place when bit 0 of the Input Status Register 1 is set to high ('1'). The IBM technical EGA documentation explains the opposite, saying when bit 0 is set low ('0') a retrace is taking place. For now, we assume source code and VGA documentation is correct, retrace takes place on a '1'.

```
;==============
;
; VW_SetScreen
;
;==============

  mov dx,03DAh          ;Status Register 1
;
; wait util the CRTC just starts scaning a diplayed line
; to set the CRTC start
;
  cli

@@waitnodisplay:        ;wait until scan line is finished
  in  al,dx
  test  al,01b
  jz  @@waitnodisplay

@@waitdisplay:          ;wait until retrace is finished
  in  al,dx
  test  al,01b
  jnz @@waitdisplay

endif

; #### set CRTC start address

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

endif

; #### set horizontal panning
```

5.12 Actors and sprites

5.12.1 A.I.

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

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

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

```
typedef struct
{
  int     leftshapenum,rightshapenum; // Sprite to render
                                      // on screen
  enum    {step,slide,think,stepthink,slidethink} progress;
  boolean skippable;
  boolean pushtofloor;  // Make sure sprites stays
                        // connected with ground
  int tictime;          // How long stay in that state
  int xmove;
  int ymove;
  void (*think) ();
  void (*contact) ();
  void (*react) ();
  void *nextstate;
} statetype;
```

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

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

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

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

5.12.2 Drawing Sprites

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

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

Unlike many game consoles such as Nintendo, the concept of sprites did not exists on the EGA card, so again the team needs to write their own solution. As explained in Section 4.2.2 (Page 156, table 5.1) each sprite asset contains additional information which is illustrated in Figure 5.43.

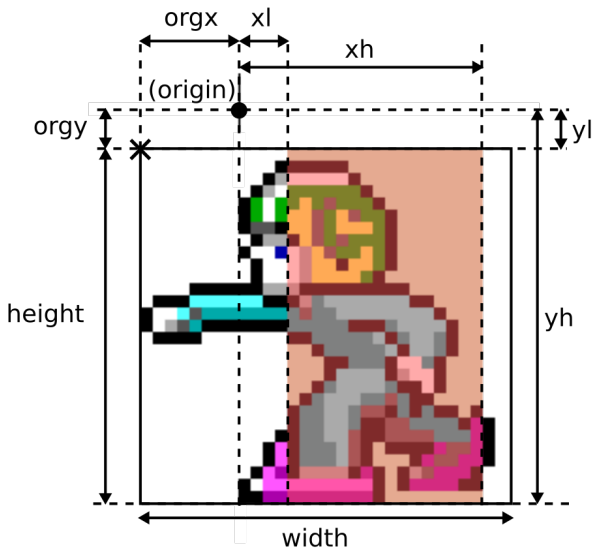


Figure 5.43: sprite structure

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

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

Table 5.1: content of spritetable[] in the `KDREAMS.EGA` asset file.

Once the new state and movement of the actor is calculated from the state machine, the engine needs to first remove the current sprite from the buffer and then redraw the sprite on the new location.

Erasing sprites from the buffer page is done by maintaining a list of erase blocks. Each erase block contains the screen (x,y) location, width and height of the sprite to be erased.

Instead of copying each variable one-by-one, it makes smartly use of `memcpy` function.

```
typedef struct spritelststruct
{
    int     screenx,screeny;
    int     width,height;
    [...]
} spritelisttype;

typedef struct
{
    int     screenx,screeny;
    int     width,height;
} eraseblocktype;


//
// post an erase block to both pages by copying
// screenx,screeny,width,height
// both pages may not need to be erased if the sprite
// just changed last frame (updatecount = 2)
//
if (sprite->updatecount<2)
{
    if (!sprite->updatecount)
        memcpy (eraselistptr[otherpage]++,sprite,
            sizeof(eraseblocktype));
    memcpy (eraselistptr[screenpage]++,sprite,
        sizeof(eraseblocktype));
}
```

The engine is maintaining two list of erase blocks; one for the view page and one for the buffer page. As explained before, each erase block is copying from the (static) master page to the buffer page.

As each sprite can float freely over the screen, also here bitshifted sprites are used to position the sprite on a byte-aligned memory layout (as explained in section 5.6.4 on page 107). The value in the shift column defines the amount of steps the sprite has to be shifted within 8 pixels. A shift value of 4 means the sprite is shifted in 4 steps with a 2 pixel interval.

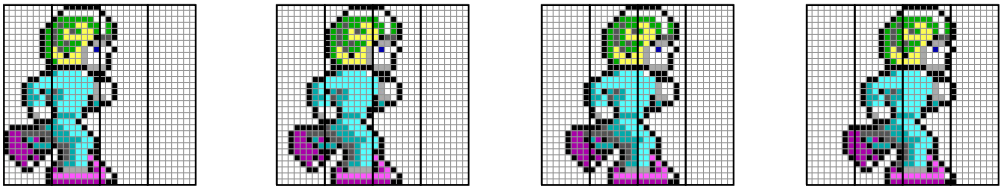


Figure 5.44: Sprite shifted in 4 steps.

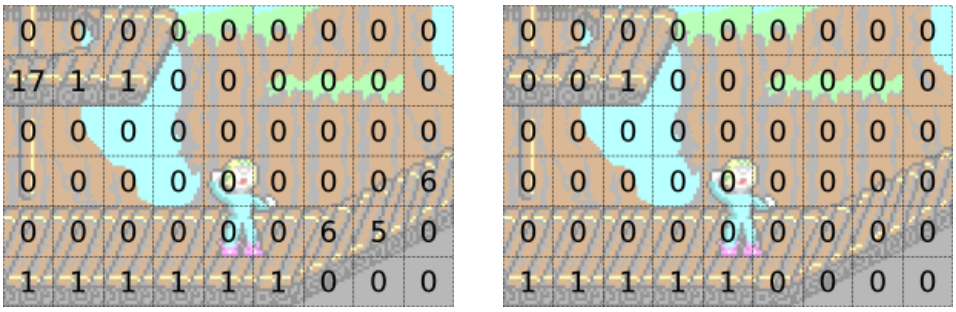
Displaying the correct shifted sprite is as simple as below.

```
//Set x,y to top-left corner of sprite
y+=spr->orgy>>G_P_SHIFT;
x+=spr->orgx>>G_P_SHIFT;

shift = (x&7)/2;  // Set sprite shift
```

5.12.3 Clipping

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



(a) Wall type map NORTHWALL ***(b)*** Wall type map EASTWALL

Figure 5.45: Foreground tile clipping information.

As example, when a sprite, moving from right to left, is hitting a wall on the left side, it will update the sprite movement to ensure the sprite clips to the east wall of the left tile as illustrated in Figure 5.46. The east/west wall clipping logic is covered by `ClipToEastWalls()` and `ClipToWestWalls()` functions.

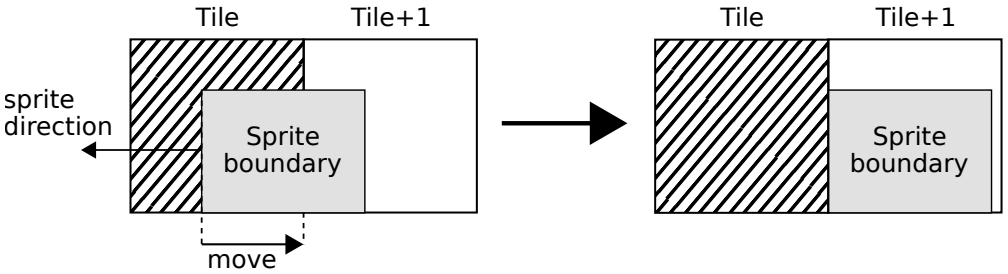


Figure 5.46: Clipping to east wall when moving from the west.

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

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

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

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

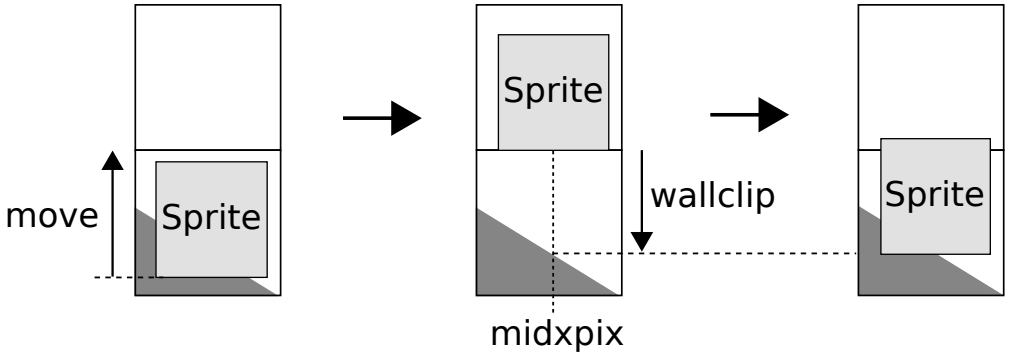


Figure 5.47: Clipping north wall with slope.

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

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

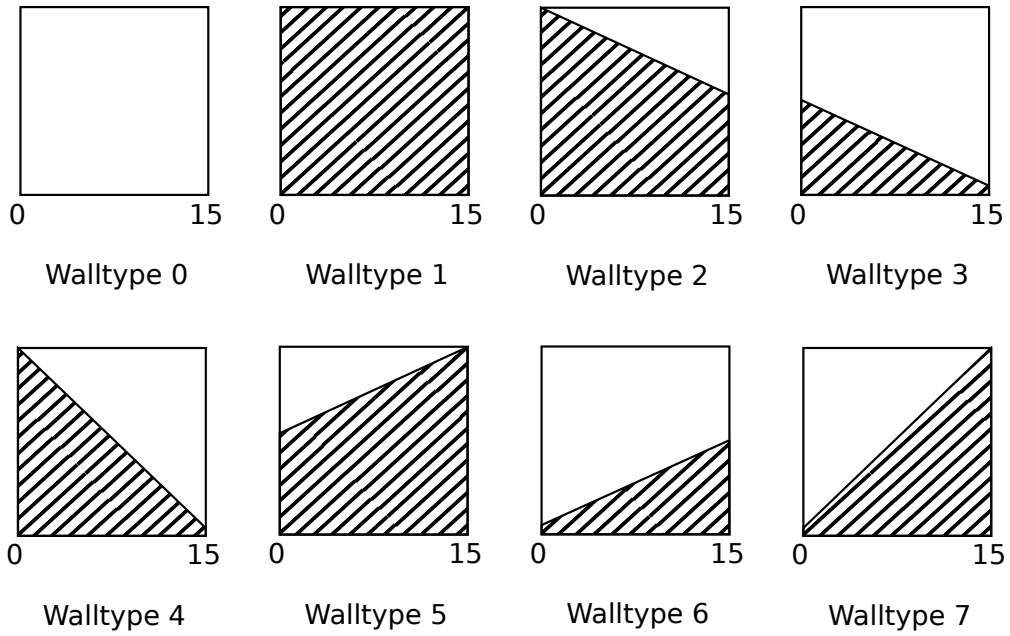


Figure 5.48: Eight different walltypes (slopes) defined.

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

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

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

5.12.4 Priority of tiles and sprites on screen

The normal screen build is as follows:

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

If multiple sprites are displayed on the same tile, each sprite is given a priority 0-3 to define the order of drawing. A sprite with a higher priority number is always displayed on top of lower priority sprites. As sprites are always displayed on top of tiles, this is causing unnatural situation when Commander Keen is climbing through a hole as illustrated in Figure 5.49.

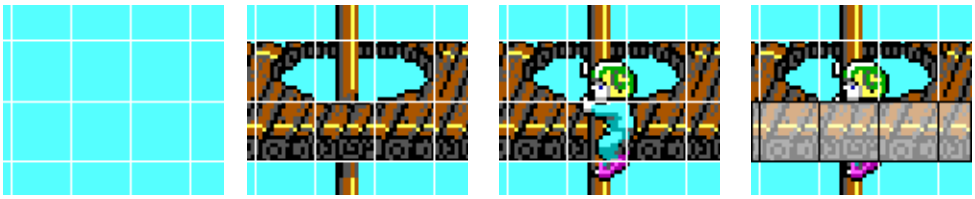
(a) Background tile. ***(b)*** Foreground tile. ***(c)*** Sprite on top.

Figure 5.49: Unnatural situation where Commander Keen is in front of a hole.

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

1. Draw the background tile.
2. Draw the masked foreground tile.
3. Draw sprites with priority 0, 1 and 2 (in that order) and mark the corresponding tile in the tile buffer array with '3' as illustrated in Figure 5.54 on page 169.
4. Scan the tile buffer array for tiles marked with '3'. If the corresponding foreground INTILE high bit (`80h`) is set, redraw the masked foreground tile.
5. Finally, draw sprites with priority 3. These sprites are always on top of everything.

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



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

Figure 5.50: Draw sprite inside a tile, by redrawing foreground tile.

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

@@realstart:
  mov di,[updateptr]
  mov bp,(TILESWIDE+1)*TILESHIGH+2
  add bp,di           ; when di = bx,
  push di             ; all tiles have been scanned
  mov cx,-1           ; definately scan the entire thing
;=============
; scan for a 3 in the update list
;=============
@@findtile:
  mov ax,ss
  mov es,ax           ; scan in the data segment
  mov al,3            ; check for tiles marked as '3's
  pop di              ; place to continue scanning from
  repne scasb
  cmp di,bp
  je  @@done
;=============
; found a tile, see if it needs to be masked on
;=============
  push  di
  sub di,[updateptr]
  shl di,1
  mov si,[updatemapofs-2+di]  ; offset from originmap
  add si,[originmap]
  mov es,[mapsegs+2]      ; foreground map plane segment
  mov si,[es:si]          ; foreground tile number
  or  si,si
  jz  @@findtile          ; 0 = no foreground tile
  mov bx,si
  add bx,INTILE           ; INTILE tile info table
  mov es,[tinf]
  test [BYTE PTR es:bx],80h ; high bit = masked tile
  jz  @@findtile

; mask the tile
```

5.12.5 Manage refresh timing

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

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

```
#define MINTICS         2
#define MAXTICS         6

void RF_Refresh (void)
{
    [...]

//
// calculate tics since last refresh for adaptive timing
//
    do
    {
        newtime = TimeCount;
        tics = newtime-lasttimecount;
    } while (tics<MINTICS);
    lasttimecount = newtime;

    if (tics>MAXTICS)
    {
        TimeCount -= (tics-MAXTICS);
        tics = MAXTICS;
    }
}
```

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 and pixel coordinatess can be easily performed by bit shift operations.

5.13 use later

```
#define G_P_SHIFT 4   // global >> ?? = pixels

void RFL_CalcOriginStuff (long x, long y)
{
  originxglobal = x;
  originyglobal = y;
  [...]

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

5.14 ATR

In the file `id_vw.h` the virtual screen in VRAM is defined by SCREENSPACE, which is set to 512x240 pixels (64x240 = 15,360 bytes). This is more than sufficient since the visible screen in mode `0x0D` is 320x200 pixels.

Since one screen only uses 15,360 bytes of VRAM (which is 3,840 bytes per plane), there is more than enough space to store more than two full screens of video data.
Before explaining the scrolling algorithm, let's first explain how the tile layout is organized. The EGA screen in mode `0xD` has a resolution of 320x200 pixels. Translated in 16x16 pixel tiles, the screen view has a size of 20x13 tiles (actually 12.5 tiles high, but we round up to 13 tiles). By making the port view one tile higher and wider than the screen, the engine can scroll the screen up to 16 pixels to the right or bottom side of the screen without any tile refresh, by means of adjusting the CRTC Start Address and Pel Pan registers. Finally, the buffer must have enough space to float the view port up to two tiles in all directions (This is

required for later versions of Commander Keen, which will be explained in Section 5.11.2). In total two tile arrays are maintained; one for the view screen and one for the buffer screen.

5.14.1 Adaptive tile refreshment in Commander Keen 1-3

In the this section we explain how the first 3 versions of the game are working[14]. 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 page and mark the tiles in both tile arrays (view and buffer tile arrays).
3. Refresh the buffer page by scanning all tiles. If a tile needs to be updated, copy the tile from the master page to the buffer page.
4. Iterate through the sprite removal list and copy corresponding image block from the master page to buffer page.
5. Iterate through the sprite list and copy corresponding sprite image block from RAM to buffer page.
6. Switch the view and buffer page by adjusting the CRTC Start Address and Pel Panning registers.

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

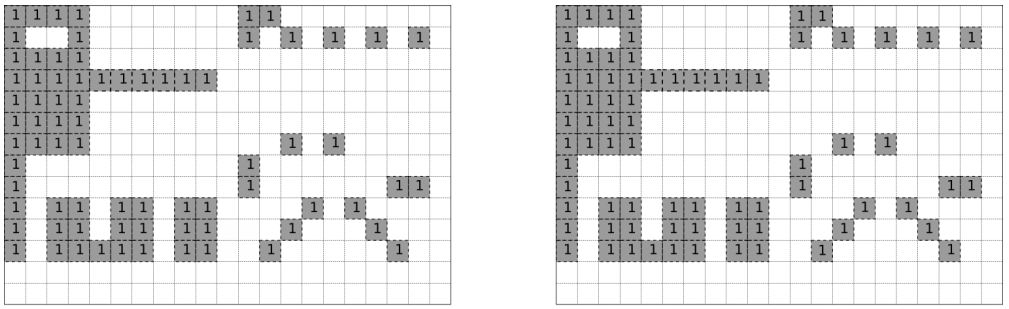


Figure 5.51: Mark all changed tiles with '1' in both tile buffer and tile view array.

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

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

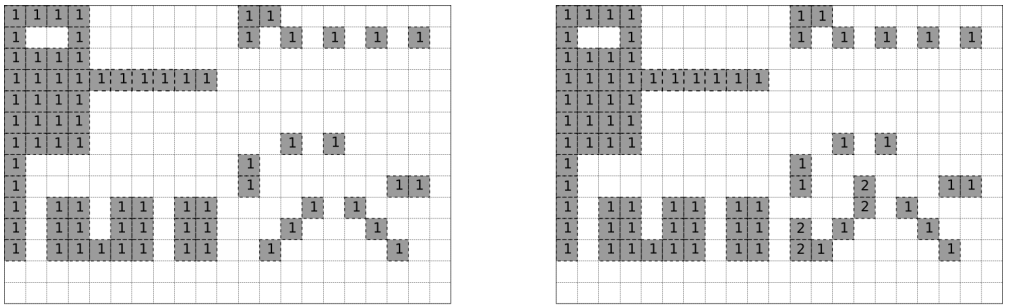


Figure 5.52: Mark removed sprites with '2' in tile buffer array only.

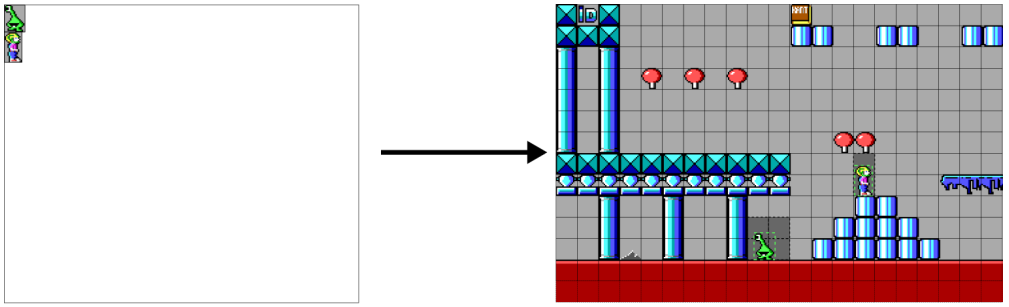


Figure 5.53: Step 5: Scan sprite list and copy sprite onto buffer page

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

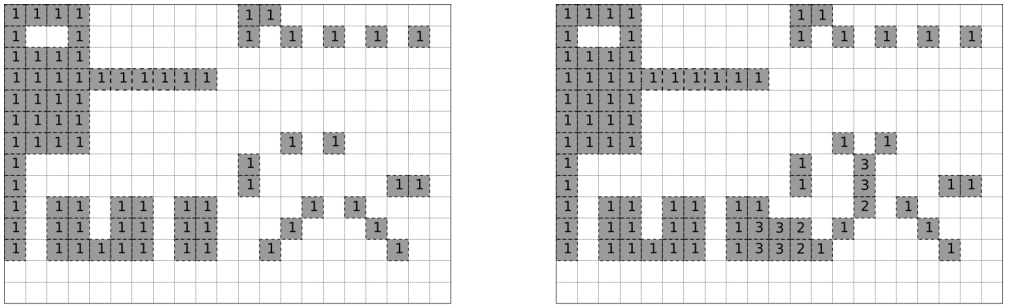


Figure 5.54: Mark new sprites locations with '3' in tile buffer array only.

In the last step, point the visible screen to the buffer page by updating the CRTC start address and horizontal Pel Panning register. Finally, the tile buffer array is cleared to '0' and both arrays are swapped (so tile buffer becomes the tile view). Then step 1 is repeated.

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

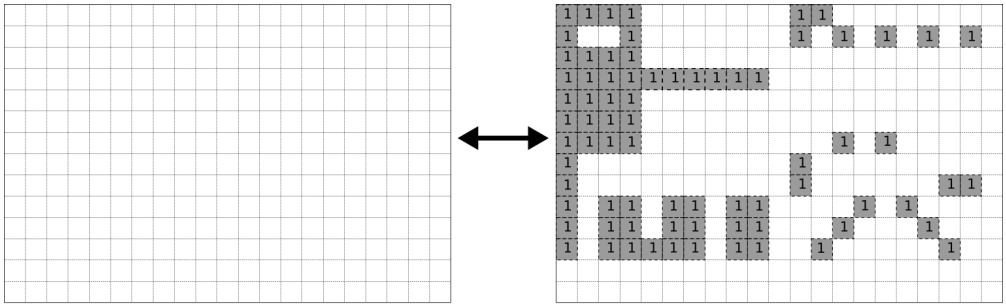


Figure 5.55: Clear tile buffer array and swap tile arrays.

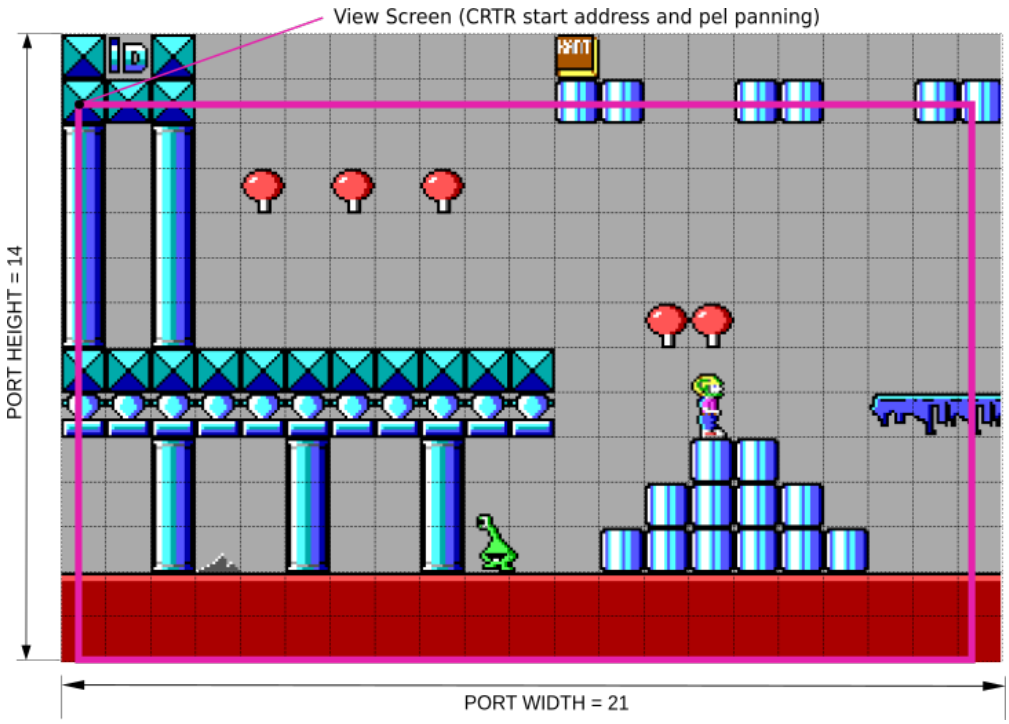


Figure 5.56: Step 6: Swap buffer and screen page

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.

To take full advantage of this optimization, the refresh algorithm maintains a list of tiles that are already copied on the master page via `tilecache` variable. If a tile is already on the master page the algorithm 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.

5.15 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[15]. We now have two systems running in parallel.

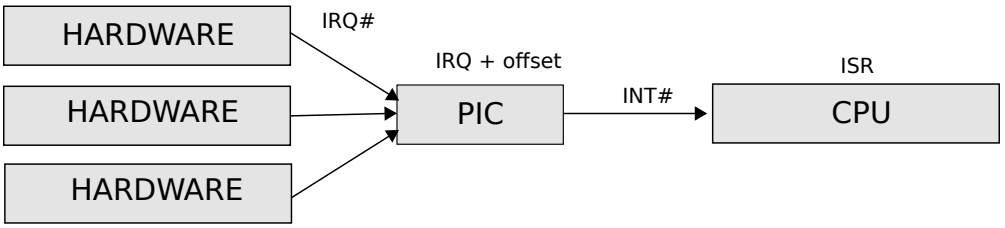


Figure 5.57: Hardware interrupts are translated to software interrupt via the PIC.

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 [16]. 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.

[15] Interrupt Service Routine
[16] Interrupt Mask Register

5.15.1 IRQs and ISRs

The IRQ and ISR system relies on two chips: the Intel 8254 which is a PIT[17] and the Intel 8259 which is a PIC[18]. The PIT features a crystal oscillating in square waves. The PIT contains three channels, each connected with a counter. On each period, it decrements its three counters. Counter #2 is connected to the buzzer and generates sounds. Counter #1 is connected to the RAM in order to automatically perform something called "memory refresh"[19]. Counter #0 is connected to the PIC. When counter #0 hits zero it generates an IRQ[20] and sends it to the PIC.

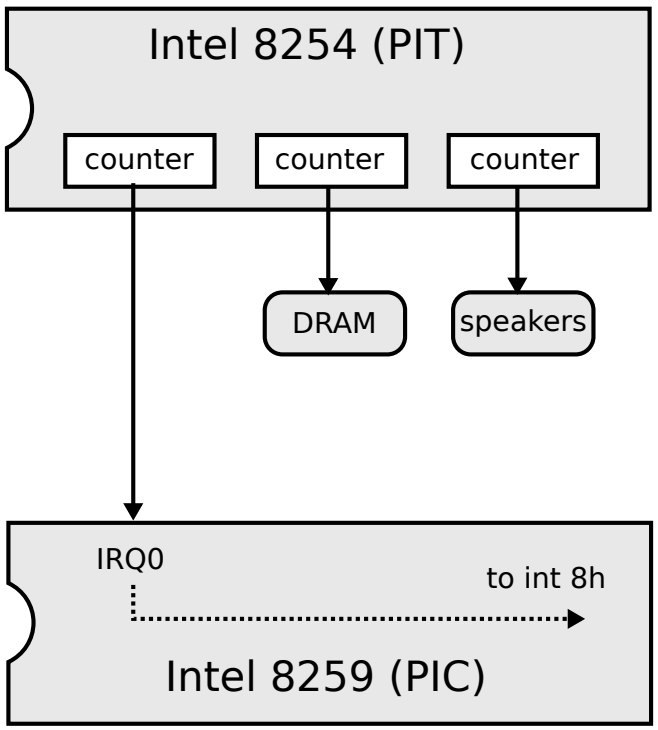


Figure 5.58: Interactions between PIT and PIC.

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).

[17] Programmable Interval Timer

[18] Programmable Interrupt Controller

[19] 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.

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

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

Figure 5.59: The Interrupt Vector Table (entries 0 to 18).

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

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

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

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

variables indicating information about the state of the computer[21]:

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

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

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

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

5.15.2 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.

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

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

5.15.3 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
}
```

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

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

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

```
#define TickBase  70

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

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

boolean SD_SetSoundMode(SDMode mode)
{
  word  rate;

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

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

5.15.4 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[22] 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`.

5.15.5 Audio System

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

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

[...]
```

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

```
[...]

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

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

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

Sounds effects are stored in three formats.

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

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

Despite being part of the source code, support for digital effects for the SoundBlaster & Sound Source devices was cut prior to release of Commander Keen Dreams. Therefore I won't explain digital effects (PCM) in this book[23].

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

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


#define NUMSOUNDS           28
#define NUMSNDCHUNKS        84

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

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

5.15.5.1 FM Synthesizer: OPL2/YM3812 Programming

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

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

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

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

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

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

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

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

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

  asm popf

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

The engine does not know about any of the details of the OPL2. There is zero abstraction layer of transformation here. An IMF sound is made of a series of messages containing the values to write to the register and data ports of the OPL2.

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

5.15.6 PC Speaker: Square Waves

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

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

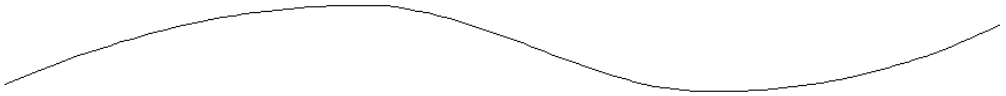


Figure 5.60: The original sound.

Figure 5.61: The same sound approximated with square wave and frequency changes.

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

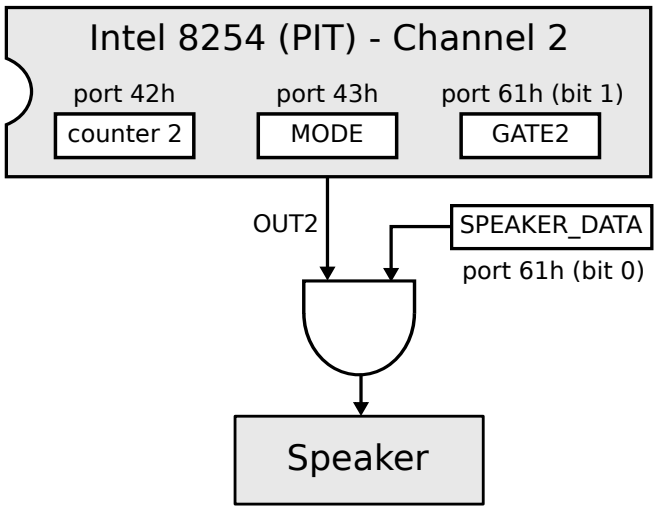


Figure 5.62: Built-in speaker hardware diagram.

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

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

Figure 5.63: Available modes of a PIT counter.

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

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

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

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

  [...]

  s = *pcSound++;

  asm pushf
  asm cli

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

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

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

  asm popf

}
```

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

```
word       pcSoundLookup[255];

void
SD_Startup(void)
{
    [...]

    for (i = 0;i < 255;i++)
        pcSoundLookup[i] = i * 60;

}
```

Notice how `0xb6` (`10110110`) is sent to the PIC Command register[24]

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

5.16 User Inputs

In an era before Microsoft harnessed all inputs under DirectInput API with Windows 95, developers had to write drivers for each input type they wanted to support. This involved talking directly to the hardware in the vendor's protocol on a physical port. The keyboard is plugged into a PS/2 or AT port, the mouse to a serial port (DE-9), and the joystick to a game port (DA-15).

5.16.1 Keyboard

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

[24] See https://wiki.osdev.org/Programmable_Interval_Timer for details.

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

static void INL_StartKbd(void) {

  IN_ClearKeysDown();

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

  INL_KeyHook = 0;  // Clear key hook
}

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

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

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

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

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

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

5.16.2 Mouse

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

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

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

software interrupt `0x33`. The interface works with requests issued in register AX[25] and responses issued in registers CX, BX and DX. With Borland compiler syntactic sugar it is easy to write with almost no boilerplate (notice direct access to registers thanks to _AX and co special keywords).

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

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

Request	Type	Response
AX=0	Get Status	AX = FFFFh : available. AX Value = 0 : not available
AX=1	Show Pointer	
AX=2	Hide Pointer	
AX=3	Mouse Position	CX = X Coordinate, DX = Y Coordinate
AX=3	Mouse Buttons	BX = 1 Left Pressed, BX = 2 Right Pressed, BX = 3 Center Button Pressed
AX=7	Set Horizontal Limit	CX=MaxX1 DX=MaxX2
AX=8	Set Vertical Limit	CX=MaxY1 DX=MaxY2
AX=11	Read Mouse Motion Counters	CX = horizontal mickey count[26], DX = vertical mickey count

Figure 5.64: Mouse request/response.

5.16.3 Joystick

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

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

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

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

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

Figure 5.65: Joystick sampling bits and their meaning.

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

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

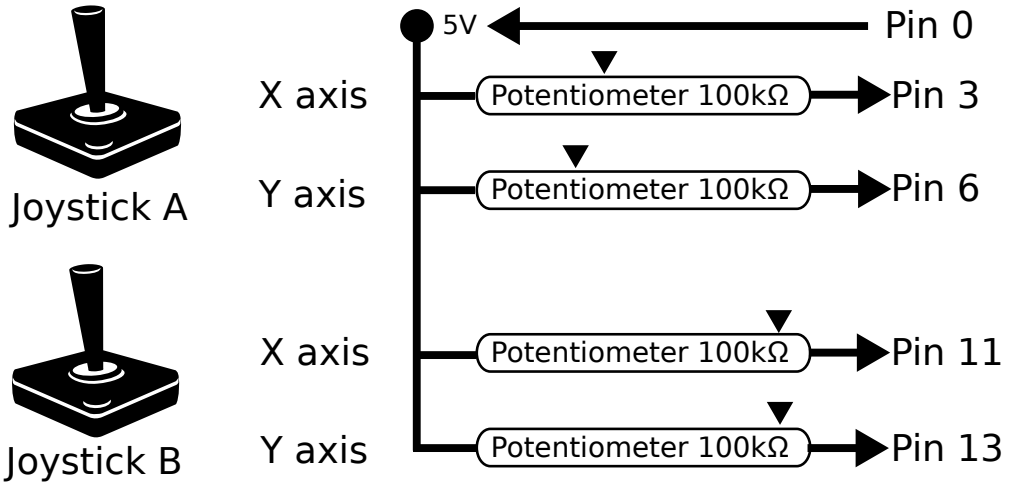


Figure 5.66: Two joysticks and the four potentiometers connected to the game port pins.

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

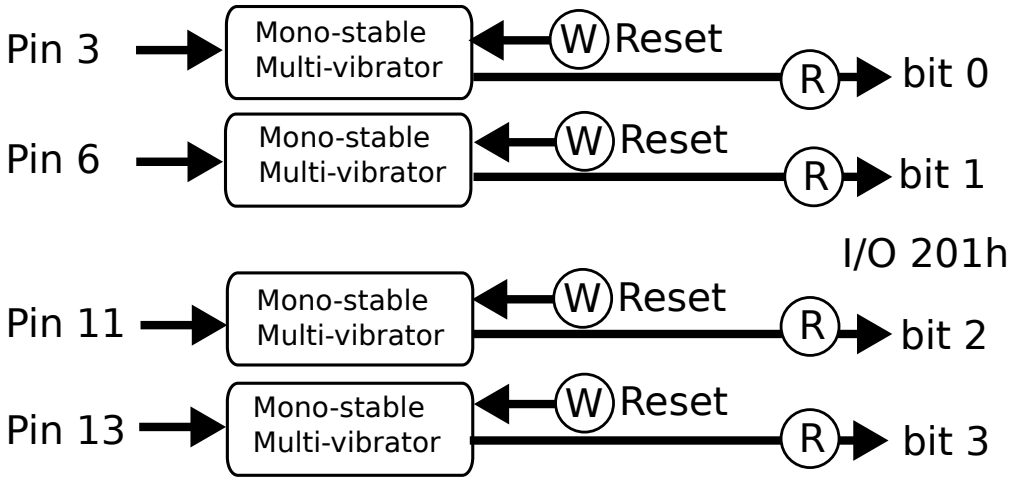


Figure 5.67: Each potentiometer is connected to a capacitor able to output either 0 or 1 depending on its charging state.

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

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

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

This explains why joysticks have to be calibrated. For the flight simulators of the 90s where accurate position was needed, the player would be asked to put the joystick in upper-left position (to set the potentiometers on both axis to minimum resistance) and press a button to read the "loop count". The player would then repeat the operation at the lower right position so that the system would know the min and max "loop count" for this joystick/CPU combination.[27]

Figure 5.68: Strike Commander startup screen makes you calibrate your joystick.

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

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

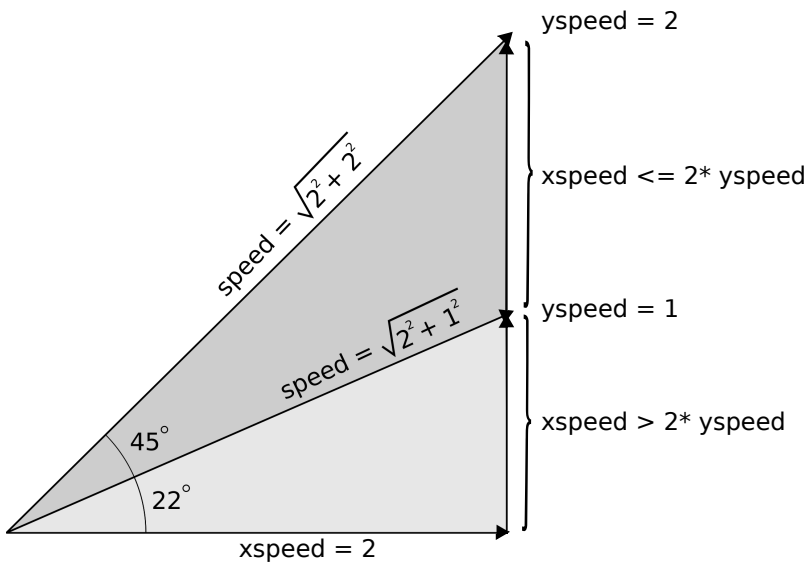
5.17 Tricks

This section describes random tricks used to keep a smooth game play.

5.17.1 Bouncing Physics

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

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



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

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

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

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

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

For each combination of the eight type of slopes (Figure 5.48) and incoming angle, the corresponding bounce angle is calculated using a simple lookup table.

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

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

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

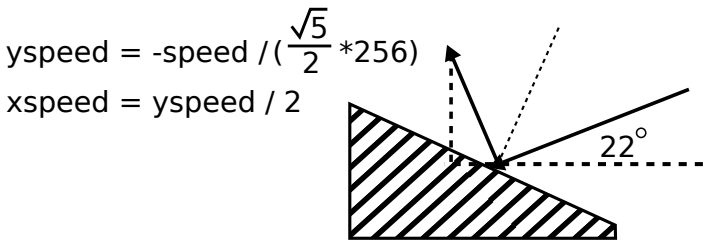


Figure 5.69: Walltype 3 with incoming angle of 22° (angle=0).

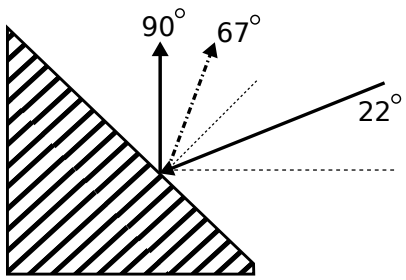
```
if (ob->xspeed > 0)
  angle = 7-angle;          // mirror angle

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

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

[...]
}
```

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



5.17.2 Pseudo Random Generator

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

```
rndindex  dw  ?

rndtable

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

Each entry in the array has a dual function. It is an integer within the range [0-255][28] and it is also the index of the next entry to fetch for next call. This works overall as a 255 entry chained list. The pseudo-random series is initialized using the current time modulo 256 when the engine starts up.

[28] Or at least it was intended to!

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


PROC  US_InitRndT randomize:word

  uses  si,di
  public  US_InitRndT

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

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

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

@@setit:
  mov [rndindex],dx
  ret

ENDP
```

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

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

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

ENDP
```

5.17.3 Screen fades

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

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

Earlier in the hardware chapter, Section 3.3.8, it was explained that most EGA monitors did not support the extended 64-color rgbRGB palette, but kept the CGA pin assignment. That means applying "rgbRGB" results in wrong color mapping to the monitor. To better understand this, let's have a look at the pin signals.

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

Figure 5.70: EGA and CGA DE-9 connector pin signals.

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

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

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

Figure 5.71: Color fading table.

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

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

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

Chapter 6

Keen Dreams in CGA

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

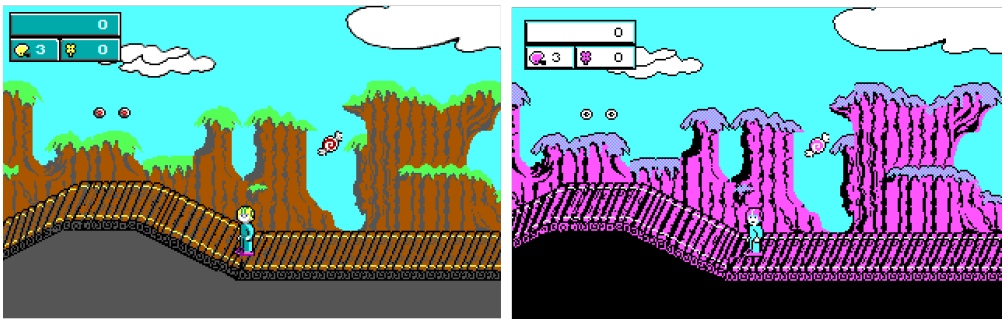


Figure 6.1: Keen Dreams EGA and CGA version.

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

6.1 CGA Video card

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

The CGA card can be summarized by the following hardware:

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

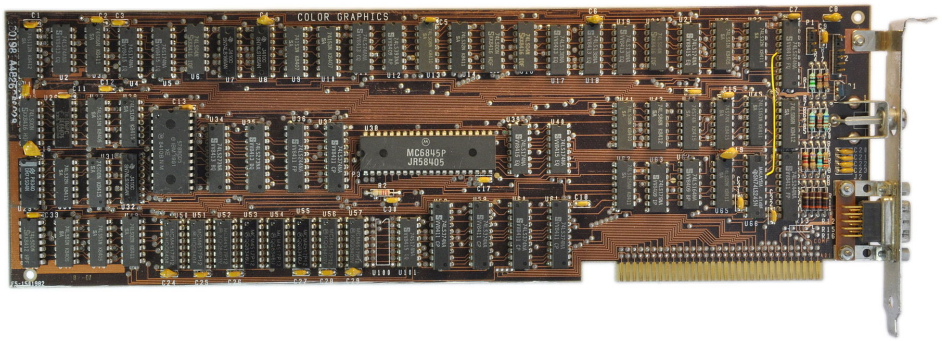


Figure 6.2: The CGA is a full-length 8-bit ISA card.

The CGA card has the following text and graphics modes:

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

Figure 6.3: CGA Modes available.

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

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

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

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

Figure 6.4: CGA color palettes.

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

6.2 Memory architecture and Interlacing

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

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

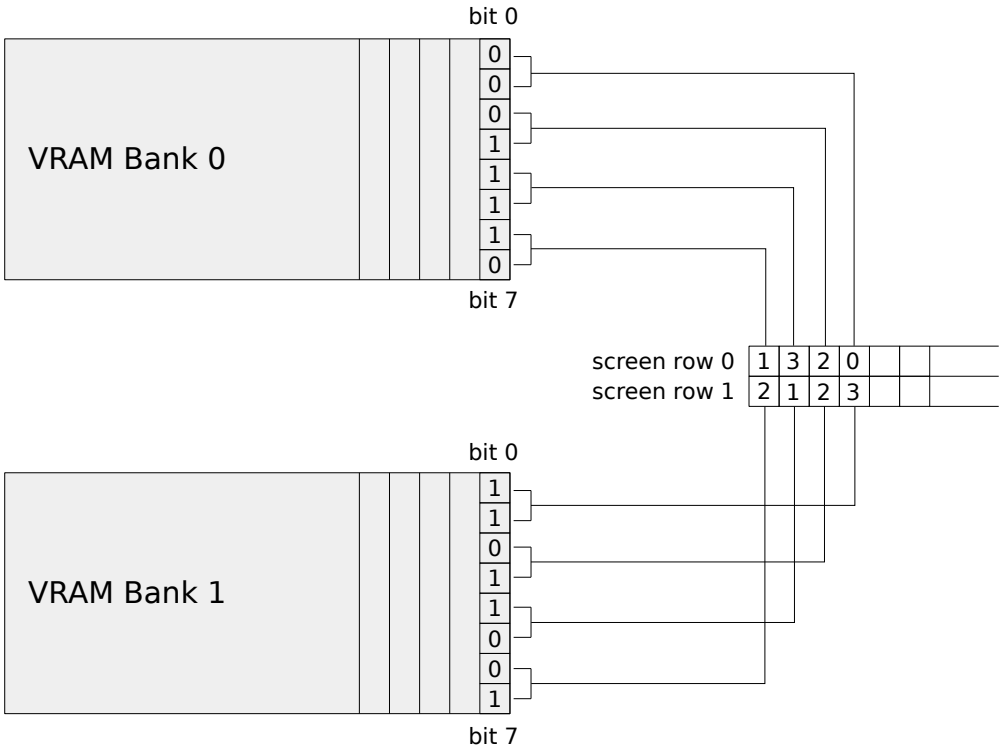


Figure 6.5: CGA interlaced memory.

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

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

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

6.3 Double buffering

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

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

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

The memory buffer contains both the buffer page and the static master page. The buffer page starts at offset `0x0000h` and the master page start at `0x8000h`. Both pages float around in the 64KiB memory segment, making use of the same memory wrapping as explained in section 5.11.1.

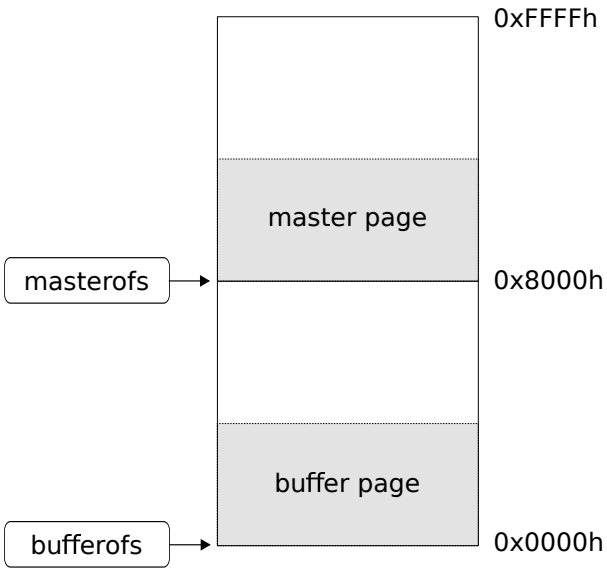


Figure 6.6: CGA double buffering memory layout.

6.4 Screen refresh

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

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

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

    originxglobal = x;
    originyglobal = y;

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

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

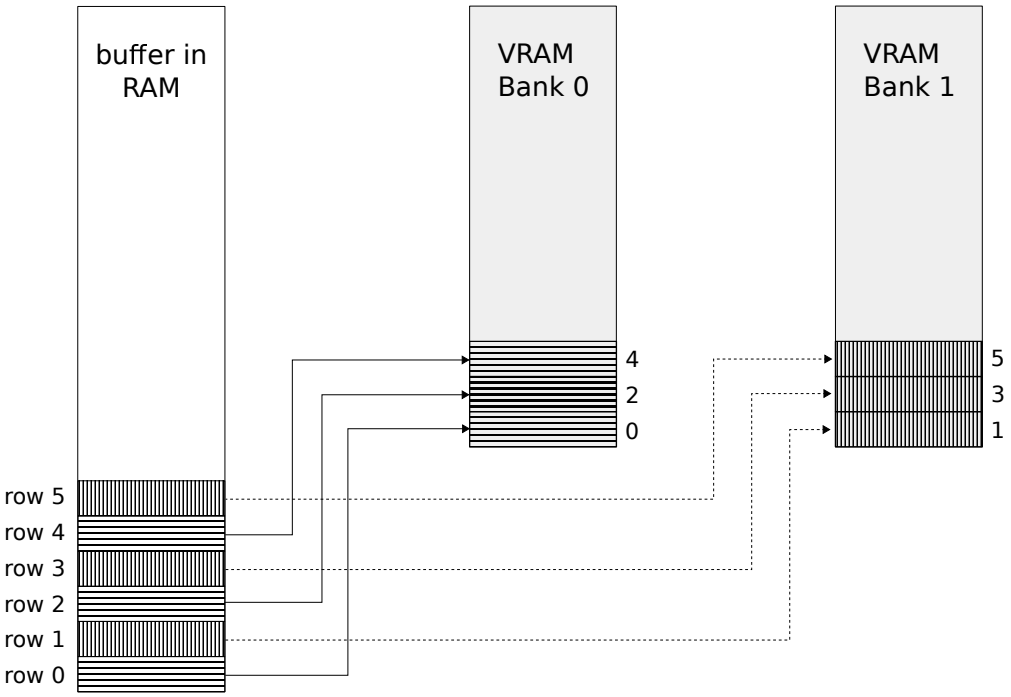


Figure 6.7: CGA memory to VRAM copy.

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

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

    asm mov ax,0xb800
    asm mov es,ax

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

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

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

    [...]

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

    asm dec bx
    asm jnz copytwolines

    [...]
}
```

Trivia : The original IBM CGA card could not handle writing and reading VRAM at the same time. Because video memory on the IBM CGA isn't dual-ported, when the CPU and the video card need access to the same byte of video RAM, the CPU wins; the card ends up reading a random value, causing "snow" on the screen. The only way to avoid the snow was, also in this case, to wait for the vertical retrace. Most CGA clones resolved the issue by enabling read/write VRAM at the same time, but if one would play Commander Keen on the original IBM CGA card you experience both screen tearing and snow on the screen.

Appendices

Appendix A

Dangerous Dave in Copyright Infringement

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

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

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

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

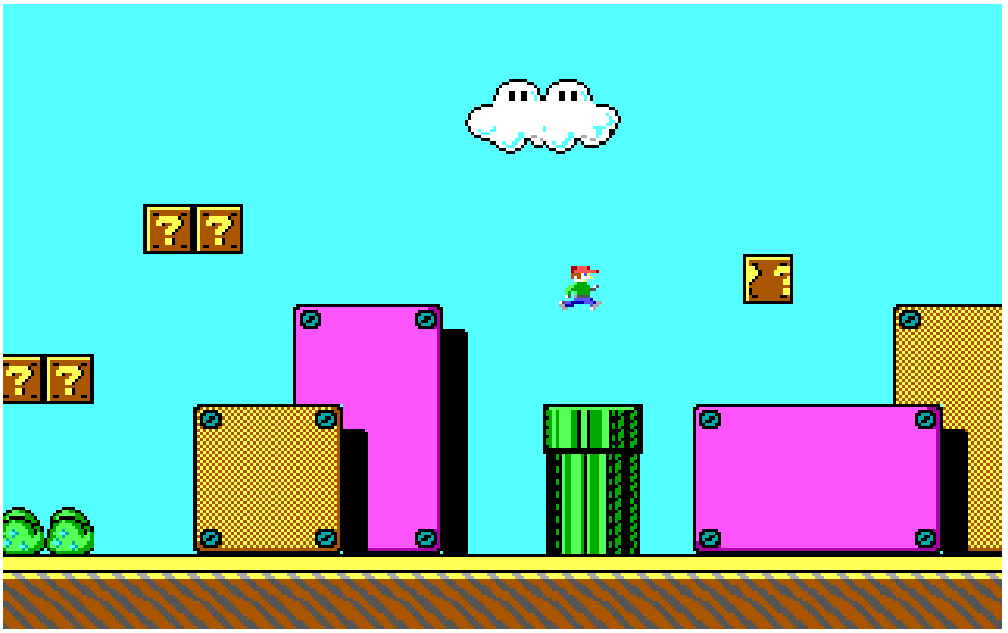


Figure A.1: Dangerous Dave in Copyright Infringement demo

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

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

Appendix B

Founding of id Software

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

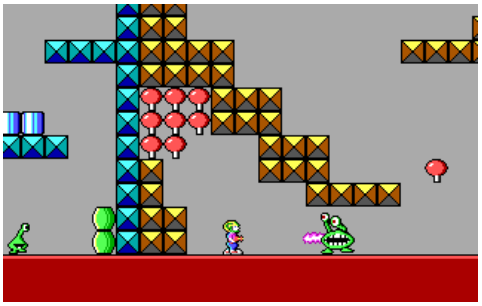
After the arrival of the first royalty check from Apogee, the team planned to quit Softdisk and start their own company. On February 1, 1991, the team founded *id Software* having four owners: John Carmack, John Romero, Tom Hall and artist Adrian Carmack[1].

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

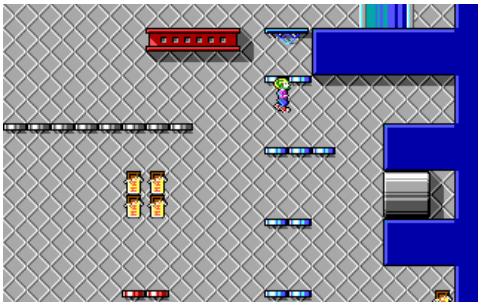
When their boss and owner of Softdisk, Al Vekovius, confronted them on their plans, as well as their use of company resources to develop the game, the team made no secret of their intentions. Vekovius initially proposed a joint venture between the team and Softdisk, which fell apart when the other employees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

[1] See Masters of Doom, chapter 4

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



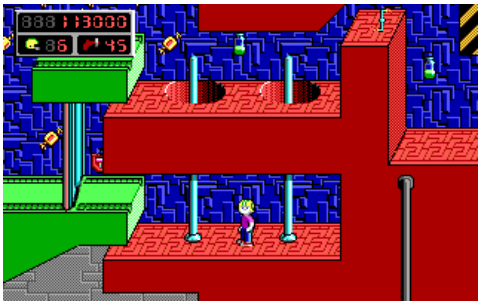
Keen I - Marooned on Mars.



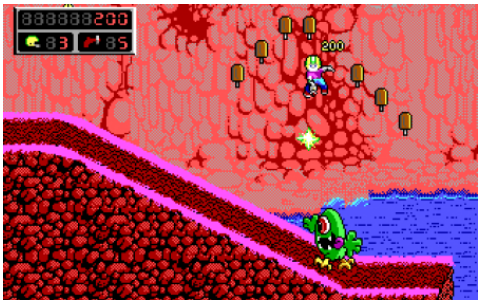
Keen II - The Earth Explodes.

Keen III - The Earth Explodes.

Keen IV - Secret of the Oracle.



Keen V - The Armageddon Machine.



Keen VI - Aliens Ate My Babysitter.

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

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

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