GAME ENGINE BLACK BOOK

COMMANDER KEEN

v2025.01.23 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].

Super Mario Bros. demonstrated the real power of the NES, which was hardware-supported scrolling. Most computers of that era, such as the MSX and Commodore 64 systems, lacked hardware support for scrolling. To achieve side-scrolling on these platforms, one needed to move all the characters (typically represented by 8x8 pixel), resulting in the super choppy scrolling effect often seen. The only way to produce smooth pixel scrolling was to redraw the entire screen, offset by the desired number of pixels. This approach was incredibly performance intensive and beyond the capabilities of most hardware at the time.

[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

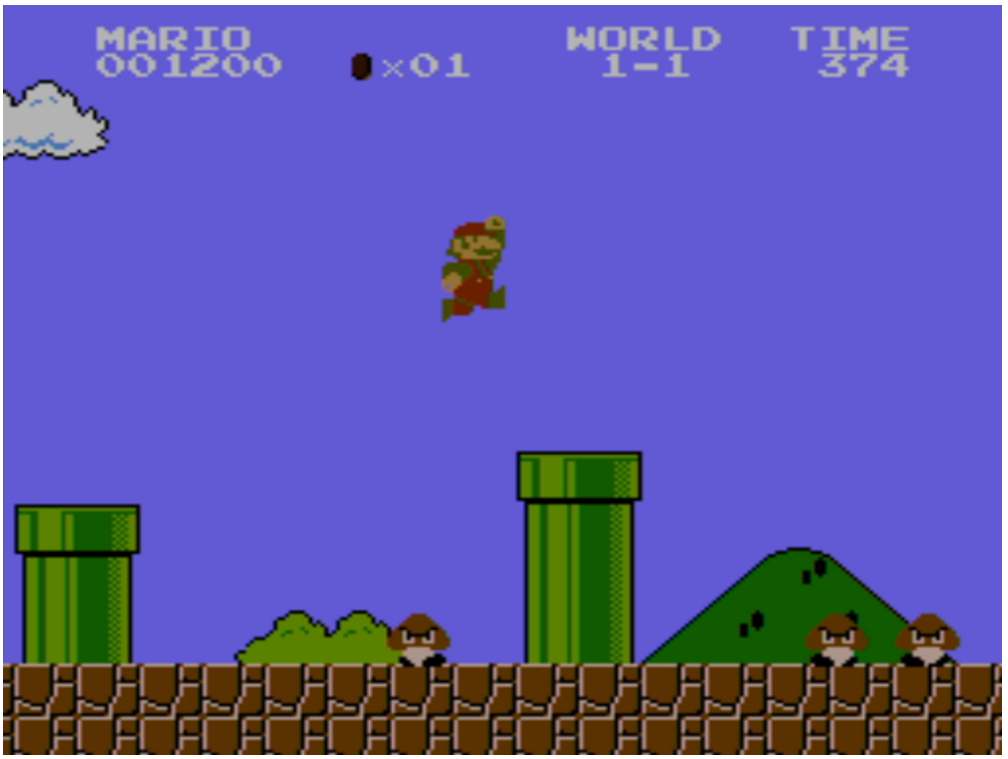


Figure 1.1: Super Mario Bros. on Nintendo Entertainment System

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 images 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.

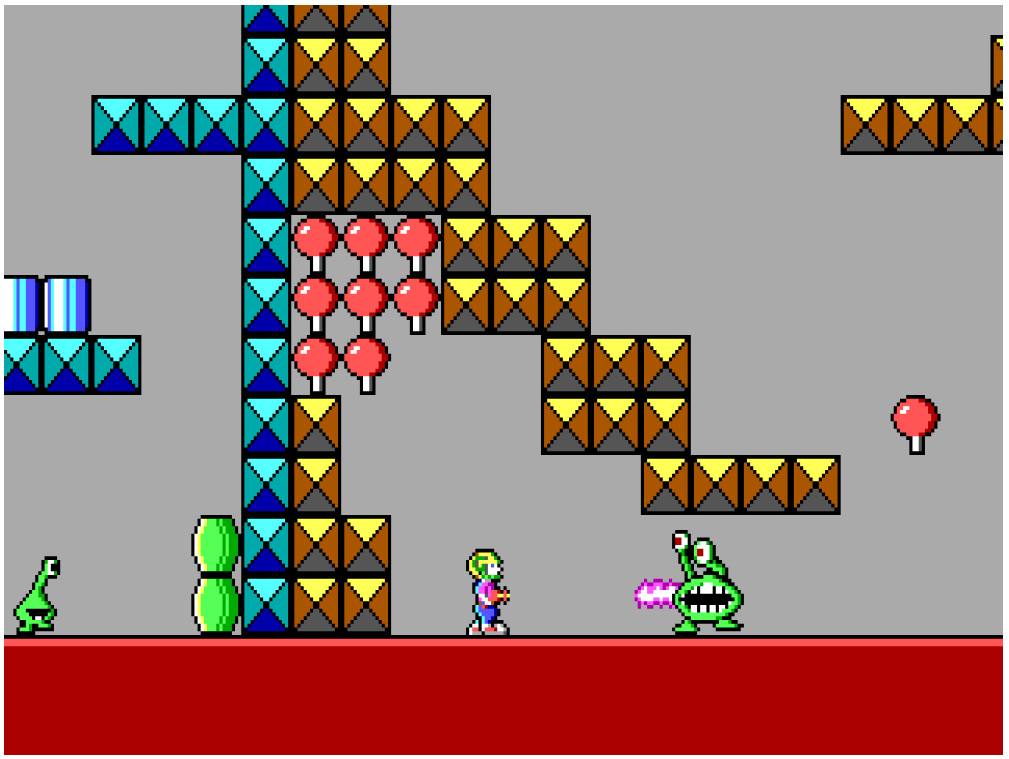


Figure 1.2: Commander Keen in Invasion of the Vorticons

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

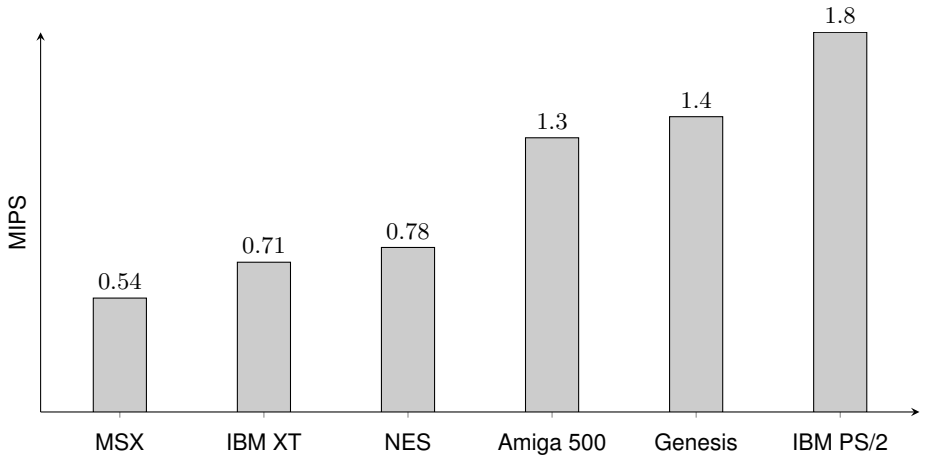


Figure 1.3: Consoles[2]vs PC, CPU comparison with MIPS[3,4].

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. Their ingenuity is what inspired me to write this book.

I've chosen to divide this book into three chapters:

- Chapter 2: The Hardware. An exploration of PC components 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. A deep dive into the Commander Keen game engine.

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

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

[3]Million Instructions Per Second.

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

This book focuses on *Commander Keen in Keen Dreams*, which is developed after the first three releases in the series. The reason for this choice is straightforward: it is the only version with publicly released source code. Where relevant, I will also highlight technological differences across the versions of Commander Keen. However, code examples will be limited to Keen Dreams, due to the availability of the source material.

Figure 1.4: Commander Keen in Keen Dreams

Chapter 2

Hardware

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

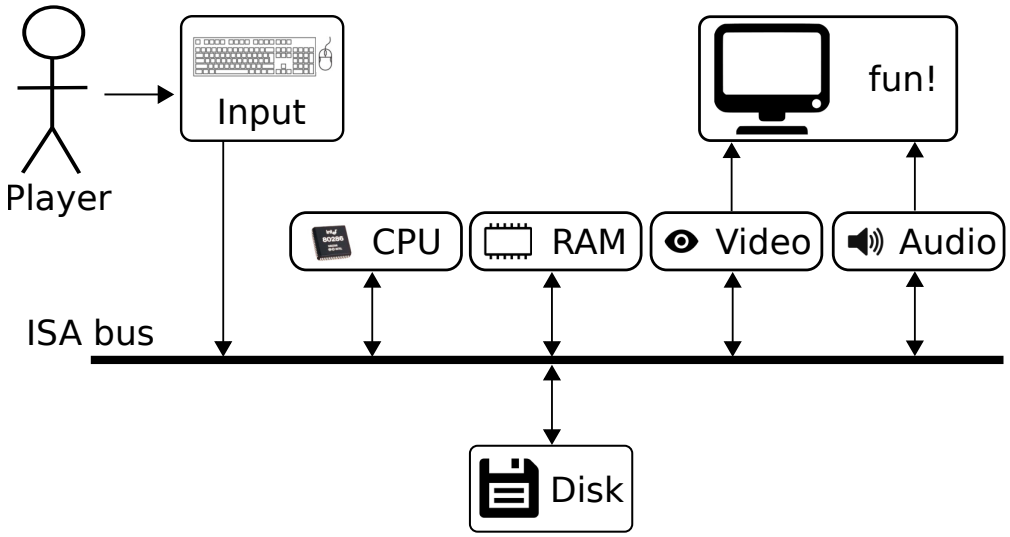


Figure 2.1: Hardware pipeline.

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

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

Figure 2.2: Component quality for a game engine.

2.1 CPU: Central Processing Unit

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

2.1.1 Overview

Intel released the 8086 in 1979, which was the first microchip of the 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. Although Commander Keen could run a 8088, it was developed with a 80286 in mind.

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

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

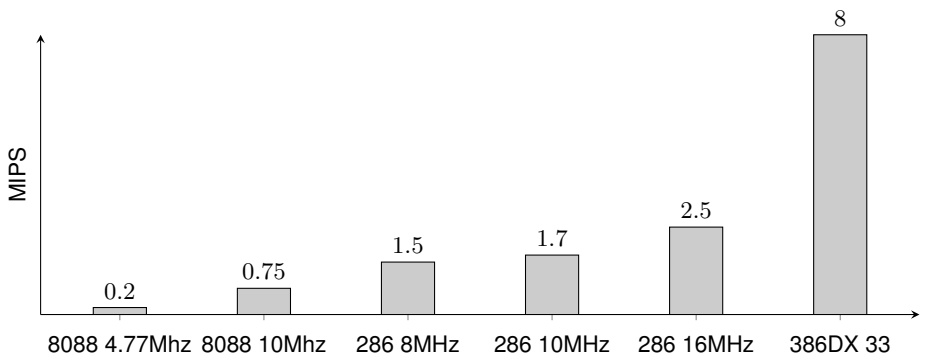


Figure 2.3: Comparison[3]of CPUs with MIPS

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

2.1.2 The Intel 80286

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

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

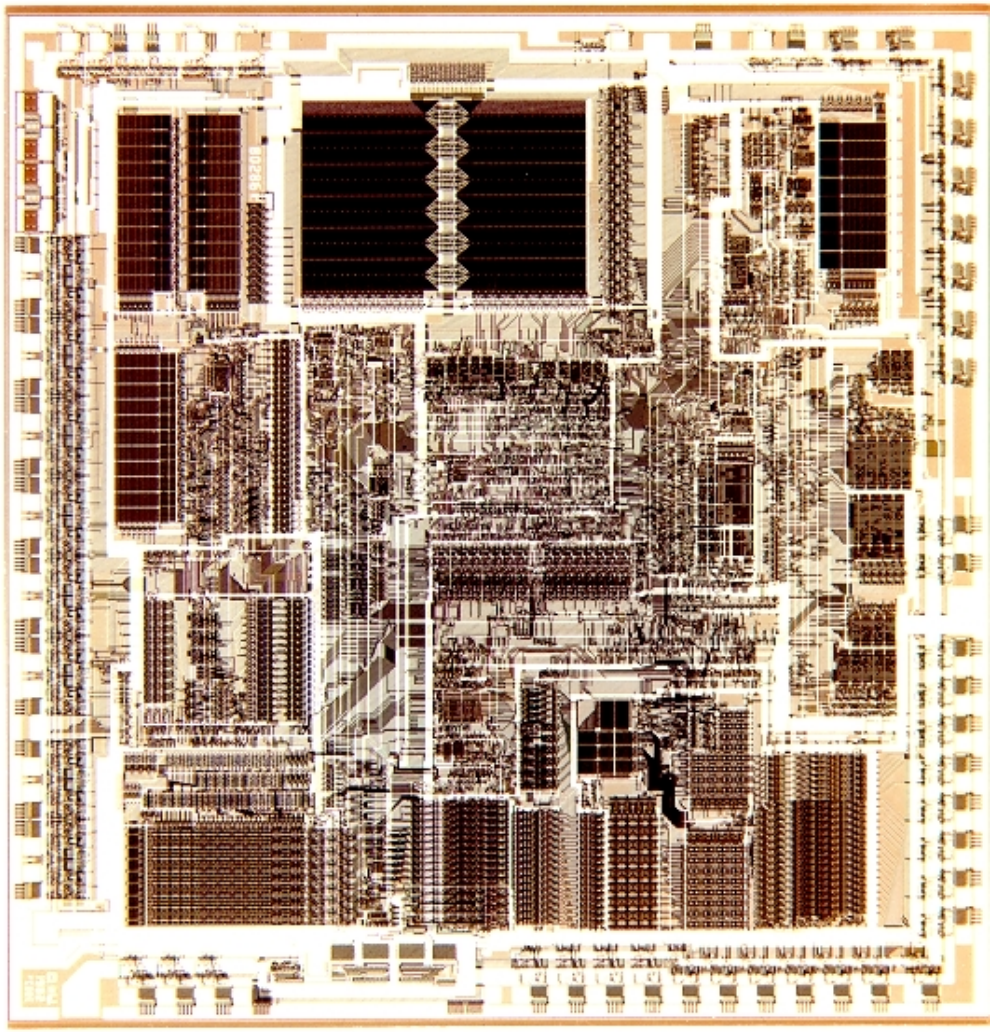
essentially the same as an 8086 chip and is fully compatible with the 8086 and 8088. In the protected mode of operation, the 286 was truly something new. In this mode, a program designed to take advantage of the chip's capabilities has access to 1GB of memory (including virtual memory). The 286 chip, 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).

Trivia : Gordon Letwin of Microsoft found a way to switch back from protected to real mode, using a "triple fault"[4] to soft reset the 286 CPU into real mode. However, it could take nearly 1 second, making switching from protected to real not feasible to be done often.

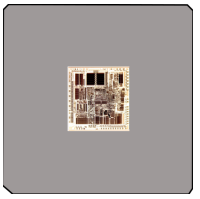
While the 8088 used a 3.0μm process, the 20286 used a 1.5μm process. The smaller process and increased surface (from 33mm^2 to 49mm^2) allowed Intel to pack 134,000 transistors on a 286 chip versus 29,000 on a 8088 chip.

If you are holding a physical 9.25"x7.5" copy of this book, the CPU packaging is 25x25 mm square and the die is 7x7 mm, at 1:1 scale.

[4] https://en.wikipedia.org/wiki/Triple_fault.




A80286-8
L9409047
©1982,1985



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

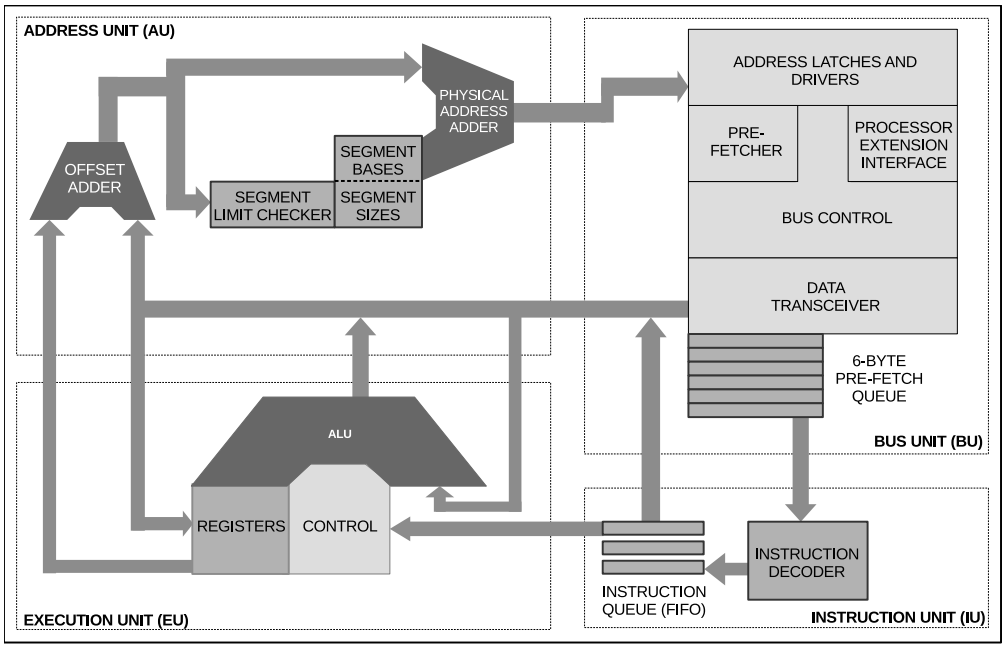
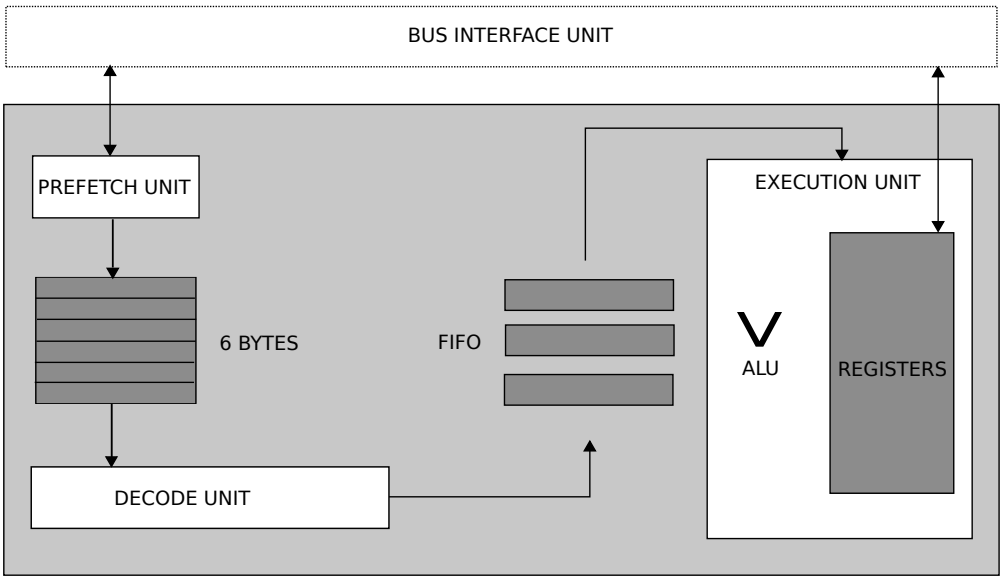


Figure 2.4: Internal block diagram of the 80286 processor

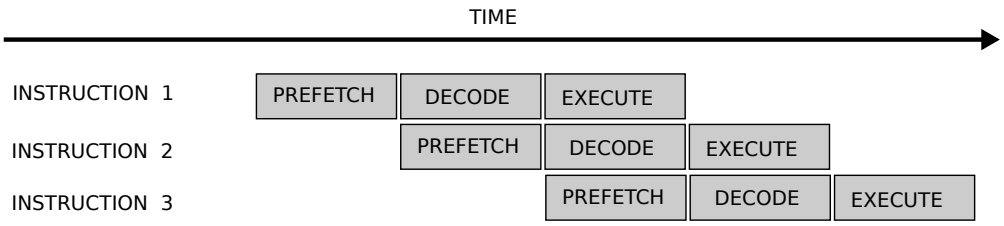
The four functional units can be described by

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

The performance increase of the 80286 over the 8088 was mainly to the fact that address calculations (such as segment+offset) were less expensive. They were performed by the dedicated address unit in the 80286, while the older 8088 had to do effective address computation using its ALU, consuming several extra clock cycles in many cases.



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



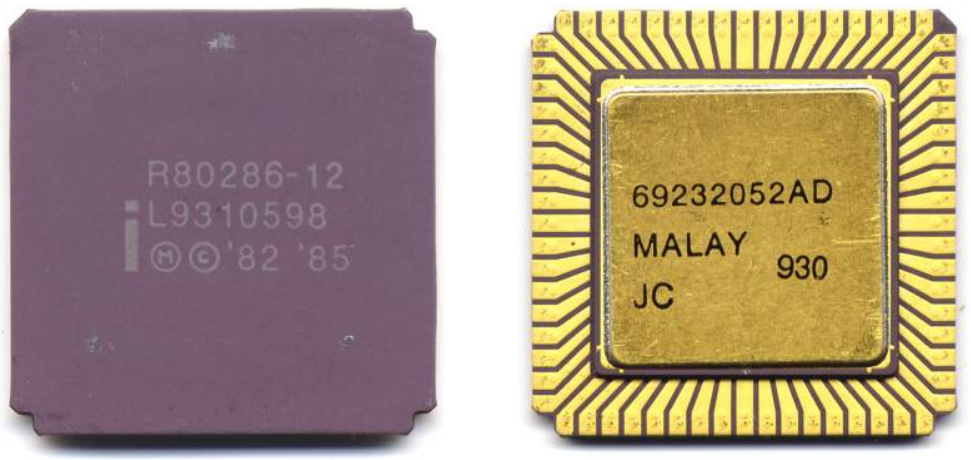


Figure 2.5: The Intel 286, 7mm by 7mm packing 134,000 transistors

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 16MiB of flat addressable RAM
- Memory Management Unit

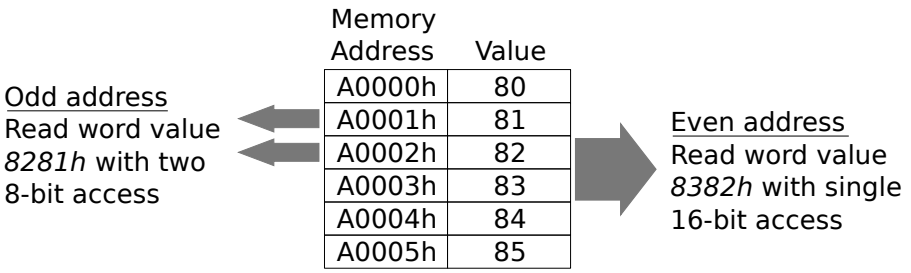
Despite its pipeline design, the 286 cannot do an operation in less than two cycles. Even a simple `ADD reg, reg` or `INC reg` takes two clocks. This is due to the absence of a SRAM on-chip cache and a slow decoding unit. Also have a look at multiplications which cost 24 cycles. So as a game developer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

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

Figure 2.6: 286 instruction costs[5].

2.1.3 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[6].



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.

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

2.2 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[7]) of memory, directly. For example, 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.

2.2.1 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[8]. 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[9] of memory. The Segment:Offset schema combines two 16-bit registers, one designating a segment and the other an offset within that segment.

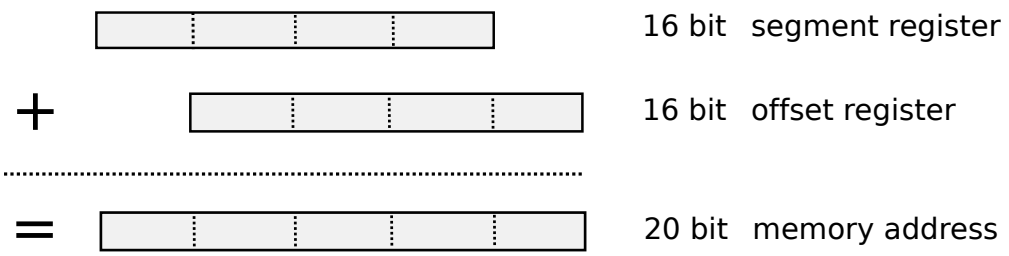


Figure 2.7: How registers are combined to address memory.

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

[8] 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."

[9] 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 pointer is much faster than using a far pointer. When you use a near pointer, the program only needs to locate the code or data through the offset (or index) register. However, when using a far pointer, the program must first find the segment and then locate the code or data within that segment. For faster execution, one should use as many near pointers as possible. The drawback of using only near pointers is that they limit the program or data to 64KiB of memory.

It's important to note that a far pointer increments only the offset, not the segment. If you iterate over a data array larger than 64KiB, there will be no automatic overflow handling, meaning you can only address up to 64KiB of memory.

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

Will output:

```
A000:0000
```

To work with memory beyond this limit, you can use another type of pointer called a "huge" pointer, which allows pointer arithmetic to function correctly beyond the 64KiB boundary.

```
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. This results in the normalized address `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.

2.2.2 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[10]. 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.

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.

[10] Memory Management Unit

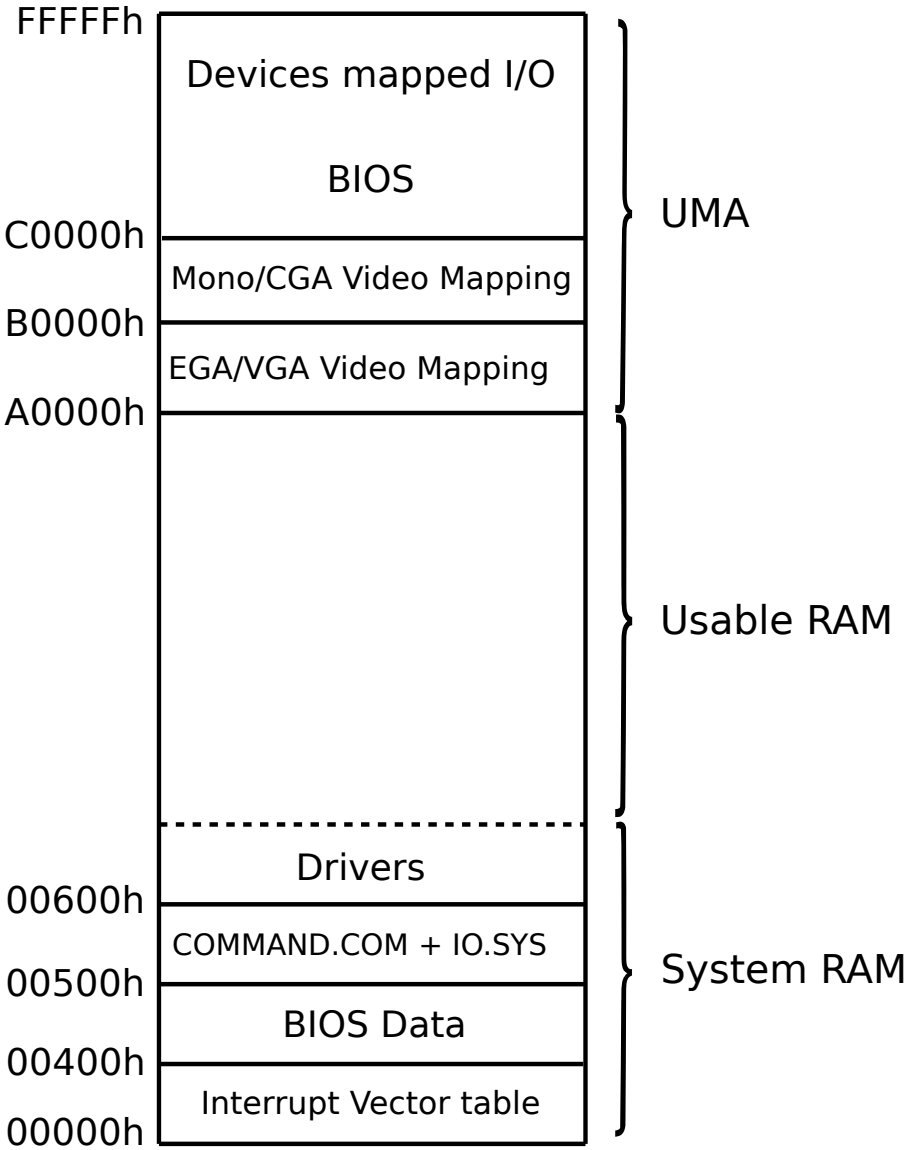


Figure 2.8: First 1MiB of RAM layout.

2.2.3 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:

- **Code section**: Stores the program executable. When you compile a C program, the compiler converts your code into assembly instructions that the CPU executes.
- **Data section**: Stores initialized and uninitialized global and static variables.
- **Stack section**: Memory used for local variables and data inside functions. The stack grows downwards, towards lower memory addresses.
- **Heap section**: Memory that is dynamically allocated using the malloc() function. The heap typically grows upwards, meaning it expands towards larger memory addresses.

The x86 real mode provides different memory segment layouts, called "memory models". Each memory model determines how segments are organized and defines whether the default pointer type for functions and data is near or far. A near pointer automatically associates with one of the segment registers. Six memory models[11] exist, each offering trade-offs between minimum system requirements, maximizing code efficiency, and accessing available memory.

	Default pointer type		**Size**		
Model	**Code**	**Data**	**Code**	**Data**	**Definition**
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 (global) data segments

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

The smallest is the "tiny memory model", where all three segment registers (CS, DS, SS) start at the same memory location. The first part of memory is used for code instructions, followed by the data section. The heap begins directly after the data section, and the stack starts at the opposite end of the segment, growing downward. Both the heap and stack can dynamically grow or shrink during program execution. If they continue to grow, they may eventually collide, causing a system or application crash.

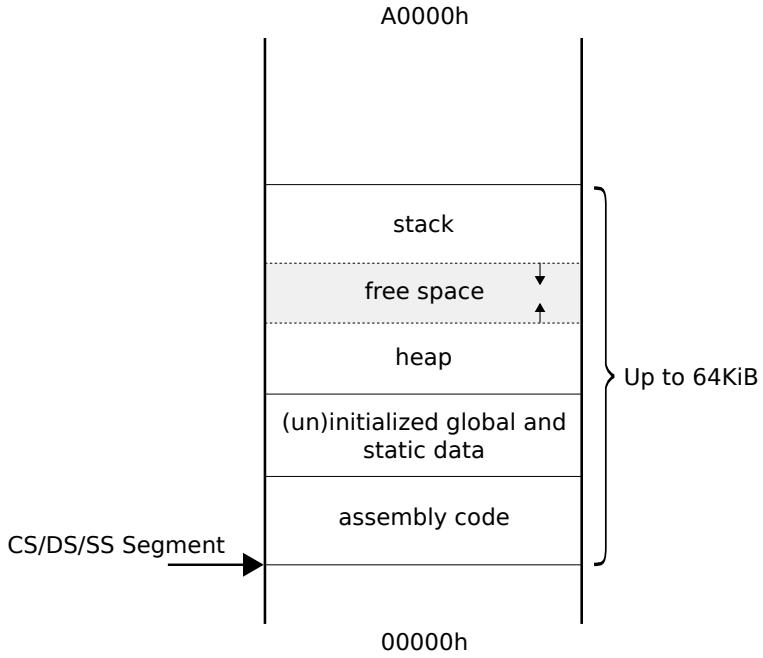


Figure 2.9: Tiny memory model.

All functions and data are accessed using near pointers, meaning the segment registers are set once and never changed during execution. However, this limits the program to a maximum of 64KiB, similar to programming on a 16-bit system.

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

In the "small memory model", instructions and data are separated, each having its own 64KiB segment. The code segment can store up to 64KiB of instructions, while the global data, stack, and heap share a separate 64KiB segment. Code execution is efficient since both functions and data are accessed using offset registers (near pointers) only.

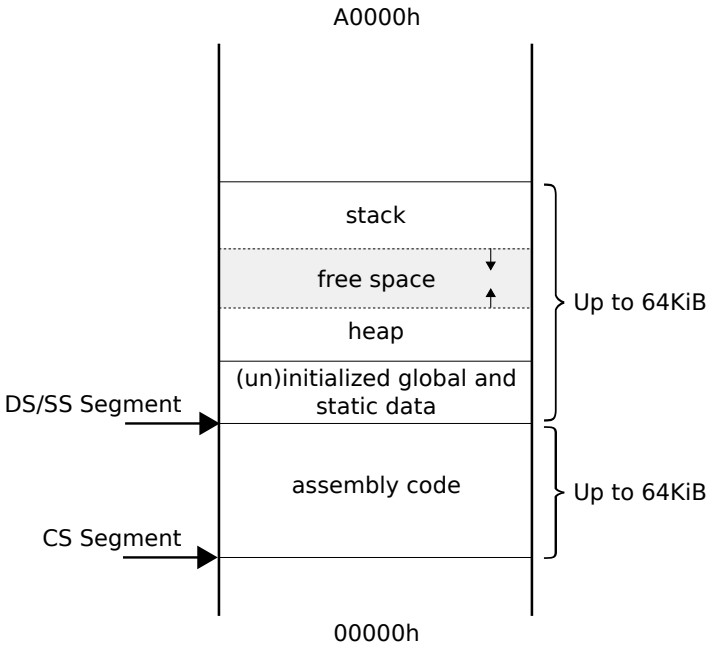


Figure 2.10: Small memory model, code and data each have 64KiB.

The "medium memory model" is ideal for programs with a large amount of code but minimal data. Many computer games fell into this category, since they had a lot of game logic, but not a lot of (global) game state data. In this model, each C source file has its own code segment, allowing up to 64KiB per file. A table manages all code segment references, with the CS register pointing to one segment at a time. Each function is a far pointer by default, requiring both the CS segment and IP offset register to be updated. While this model supports larger codebases, it comes at the cost of slower execution due to the far pointers.

Trivia : When compiling a source file, its code cannot exceed 64KiB, as it must fit inside one code segment. If the file is too large, the program must be broken into smaller source files and compiled separately.

The "compact memory model" is the opposite of the medium memory model, allowing more than 64KiB of data while restricting function code to 64KiB. The "large memory model" supports both code and data larger than 64KiB but requires far pointers for both, leading to slower execution. The "huge memory model" removes the 64KiB limit on global data, allowing more flexibility, but also incurs a performance penalty. A detailed layout and explanation of each memory model is described in Appendix B.

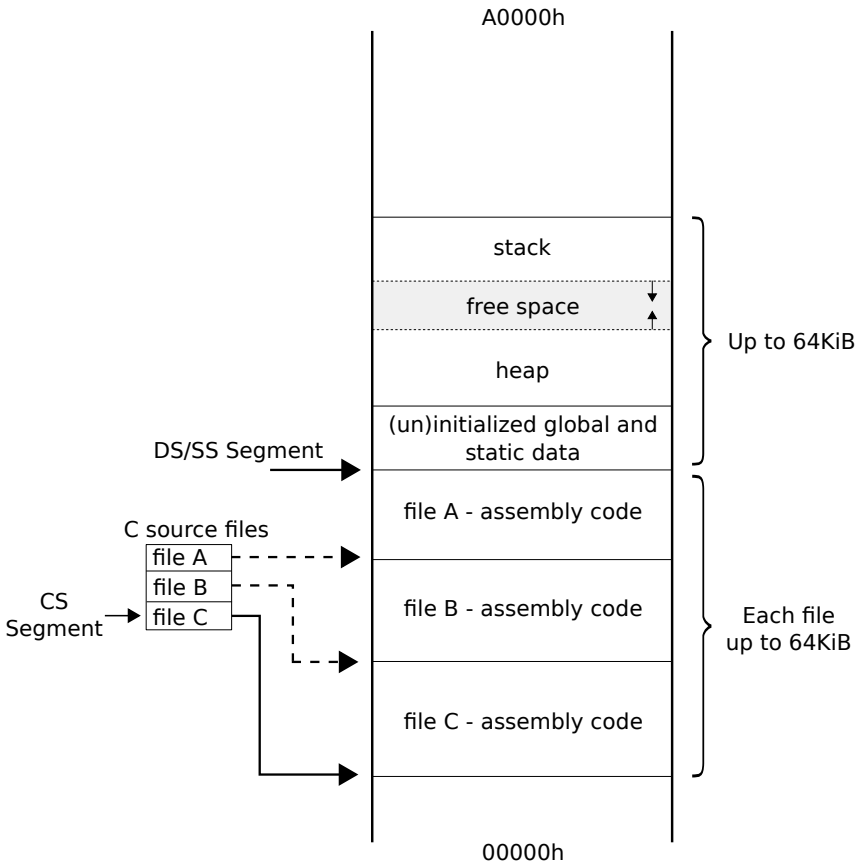


Figure 2.11: Medium memory model, code can be larger than 64KiB.

2.2.4 Mixed Model Programming

In the medium memory model, the data segment remains limited to 64KiB. For most games, this is enough to store game state variables. However, the size of the heap is far too small to store game data such as graphics and game levels. One option is to use the large or huge memory models, but these come with the downside of slower data access due to the use of far pointers for all data.

Fortunately, there is a way to access additional memory using the medium model. A large portion of unallocated memory is available between the stack and the high address

A0000h. This memory, known as the "far heap", can be allocated using the `farmalloc()` function. This technique, called mixed model programming, combines the advantages of one of the six standard memory models with custom near and far pointer allocation. In the case of the medium memory model, it allows the program to benefit from near pointer efficiency for global data and the stack, while also providing sufficient memory for dynamically allocated assets like graphics and levels.

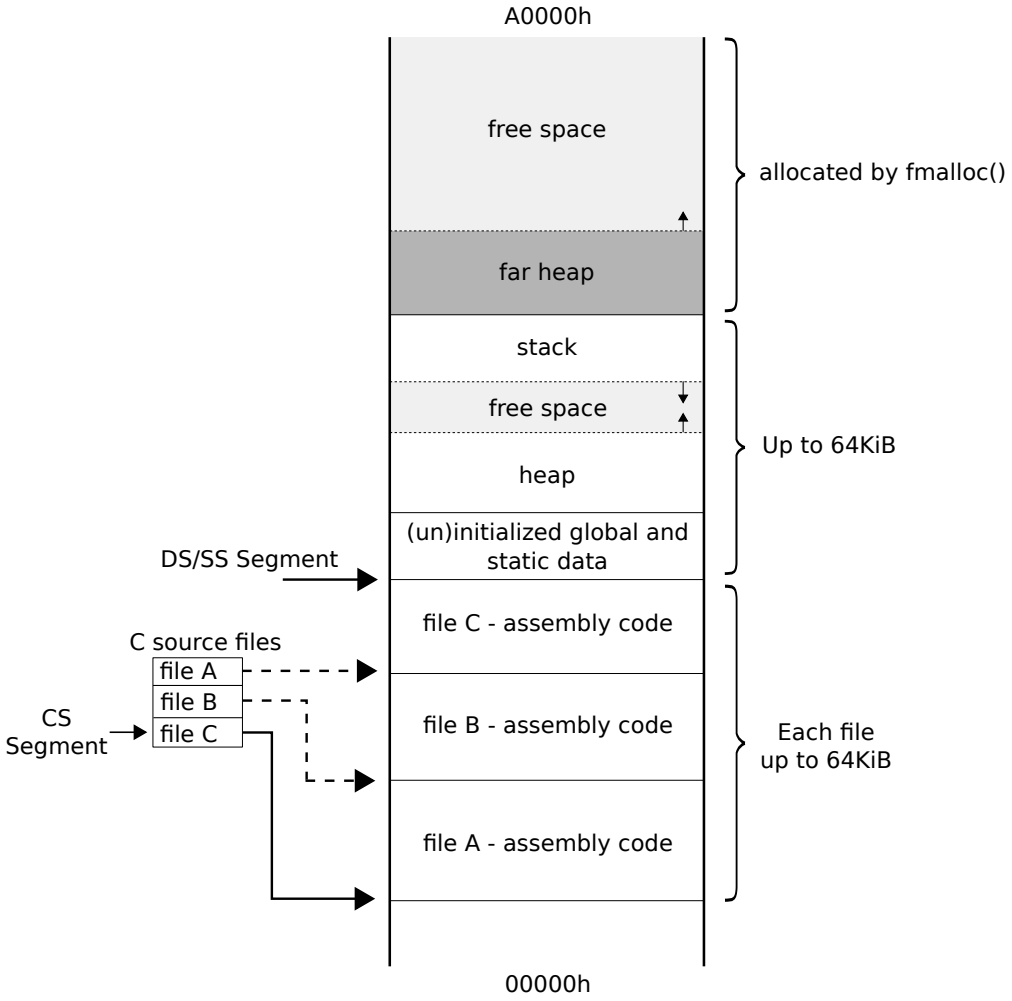


Figure 2.12: Medium memory model with far heap memory.

2.3 Video

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

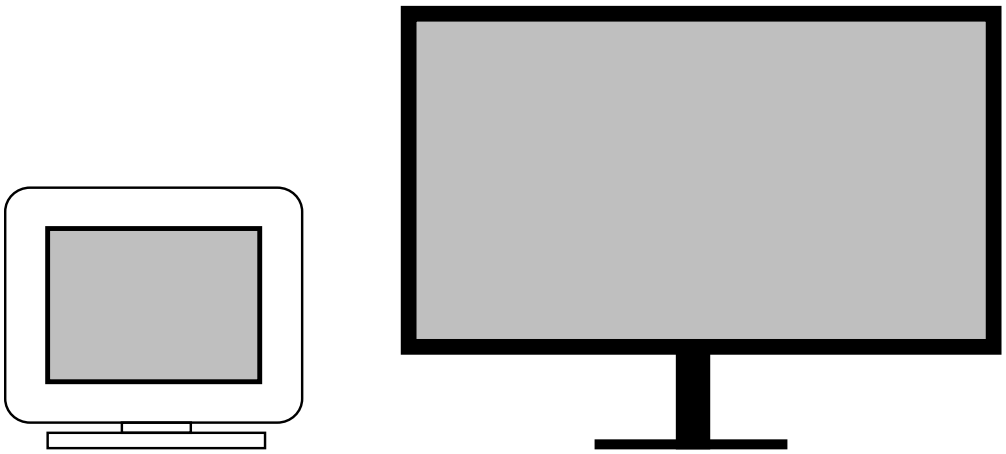


Figure 2.13: CRT (left) vs LCD (right)

To give you an idea of the size and resolution, figure 2.13 shows a comparison between a 14" CRT from 1990 (capable of a resolution of 640x200) and a 30" Apple Cinema Display from 2014 (capable of a resolution of 2560x1600).

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

2.3.1 CRT Monitor

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

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

known as horizontal blanking.

The area immediately surrounding the display area, in which the beam may be turned on during the retrace interval, is called the overscan or border. The active display area is the portion of the screen that contains characters and/or graphics. These components of the scan are shown in simplified form in Figure 2.14.

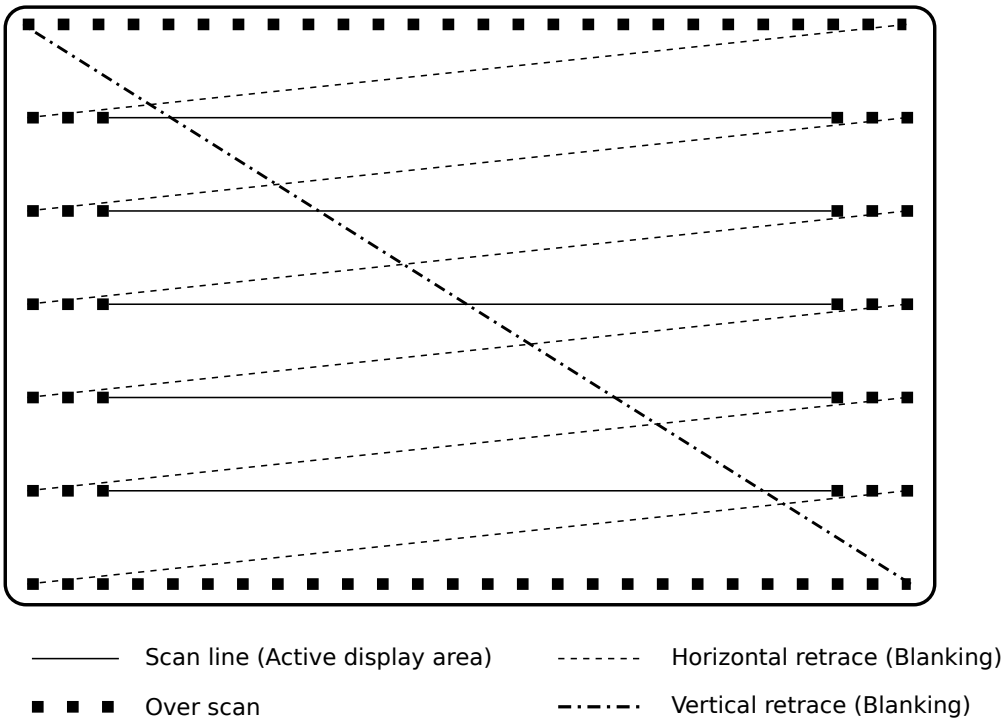


Figure 2.14: Simplified CRT monitor scan.

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

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

2.3.2 History of Video Adapters

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

Name	Year Released	Memory	Max Resolution
MDA (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[12]
Hercules	1982	64KiB	720x348
CGA (**C**olor **G**raphics **A**dapter)	1981	16KiB	640x200
EGA (**E**nhanced **G**raphics **A**dapter)	1985	64KiB	640x350
VGA (**V**ideo **G**raphics **A**rray)	1987	256KiB	640x480

Figure 2.15: 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.

2.3.3 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 2.17 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[13]. 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.

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

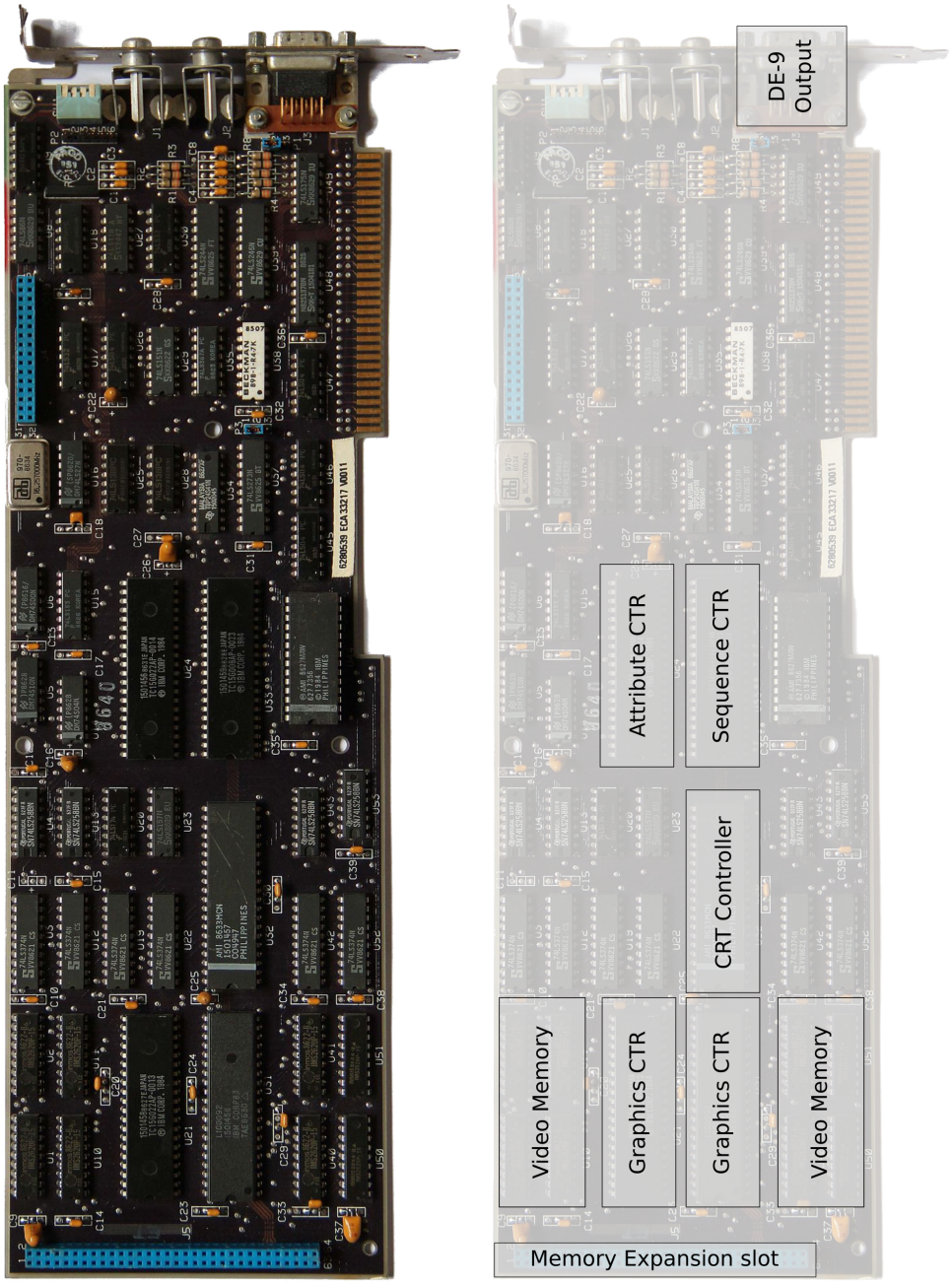


Figure 2.16: Original IBM EGA card

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.

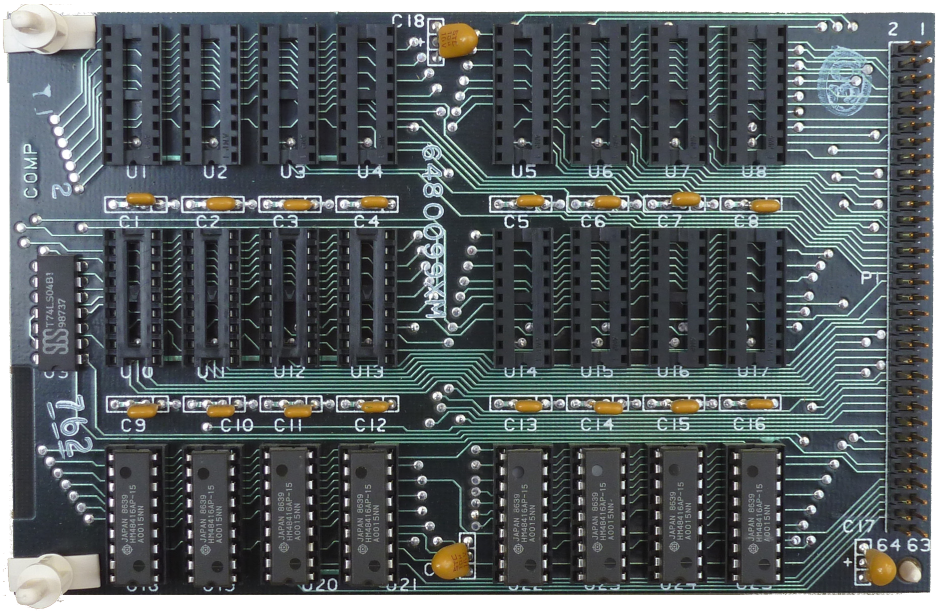
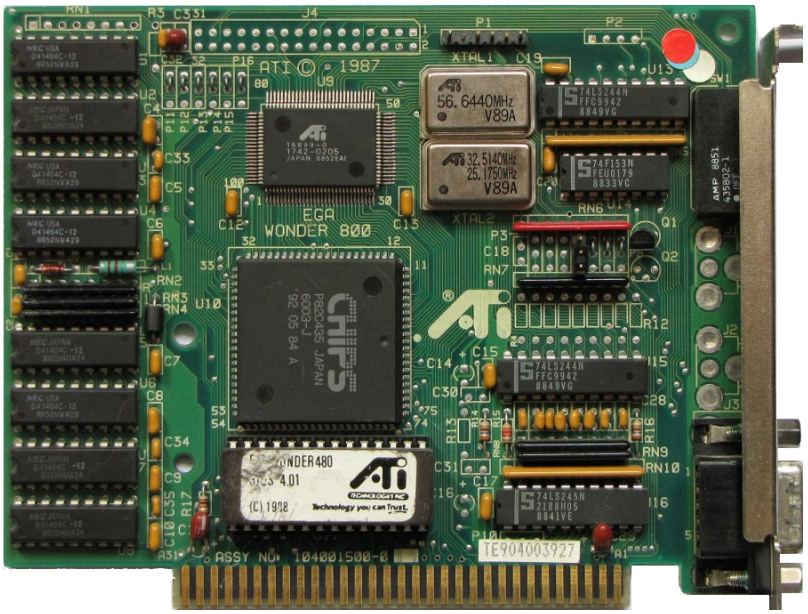


Figure 2.17: EGA Graphics Memory Expansion Card, bottom row populated with chips.

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[14].

The next page shows an ATI EGA Wonder 800 (top) and Paradise AutoSwitch EGA 350 (bottom), both are 8-bit ISA. The eight chips on the left of the card form the VRAM where the framebuffers are stored.

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


ATI © 1987
56.640MHz
V89A
32.514MHz
25.175MHz
V89A
EGA
WONDER 800
WONDER480
(C)1988
Technology you can Trust.
ASSY NO: 104001500-0


PARADISE
PEGA1A
38304 8841
015567190502
Paradise Basic EGA Card
ASSY 28250 REV 1
FCC ID: DBM8FL24LMYRP AEGA1A
MADE IN HONG KONG
R2TP 553645

2.3.4 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[15] signal for display

Trivia : In the 1980's integrated video DACs[16] 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[17] 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.

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

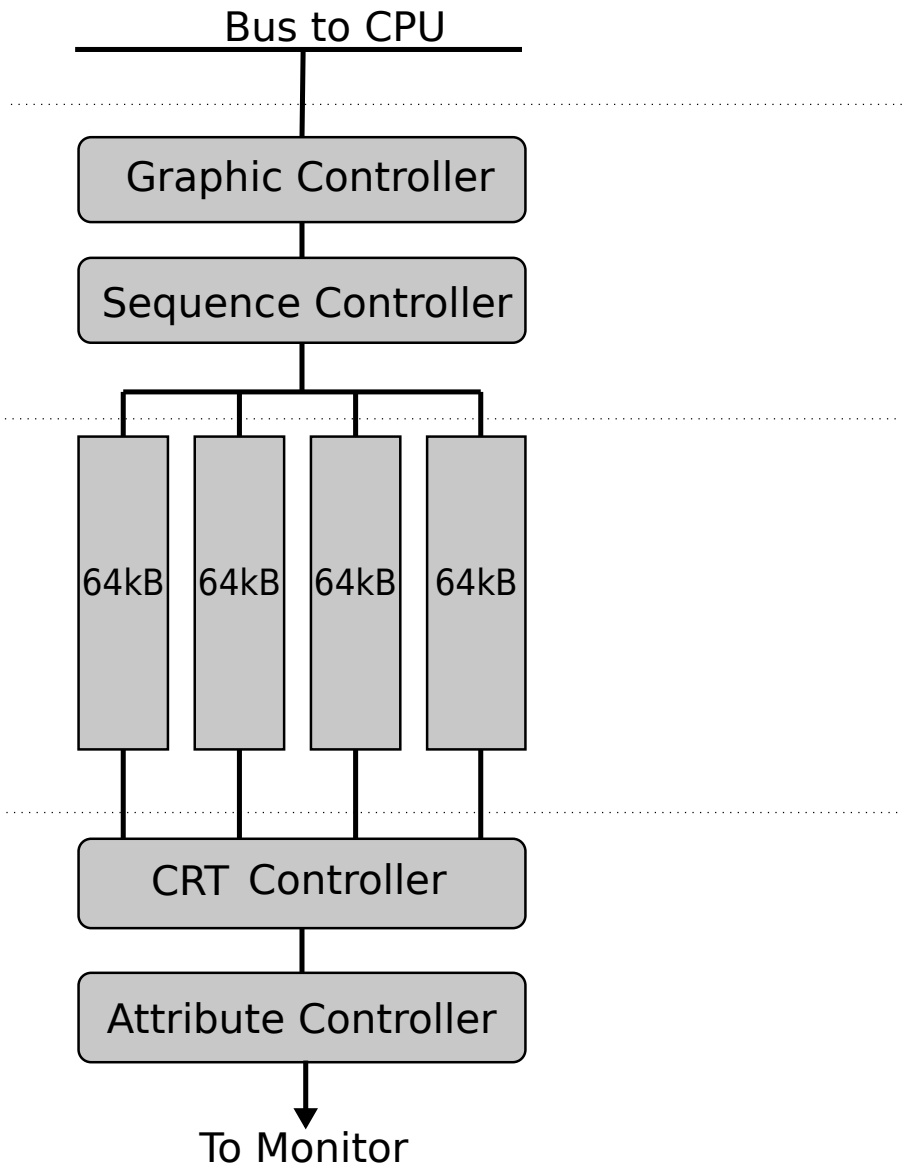


Figure 2.18: EGA Architecture.

2.3.5 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.

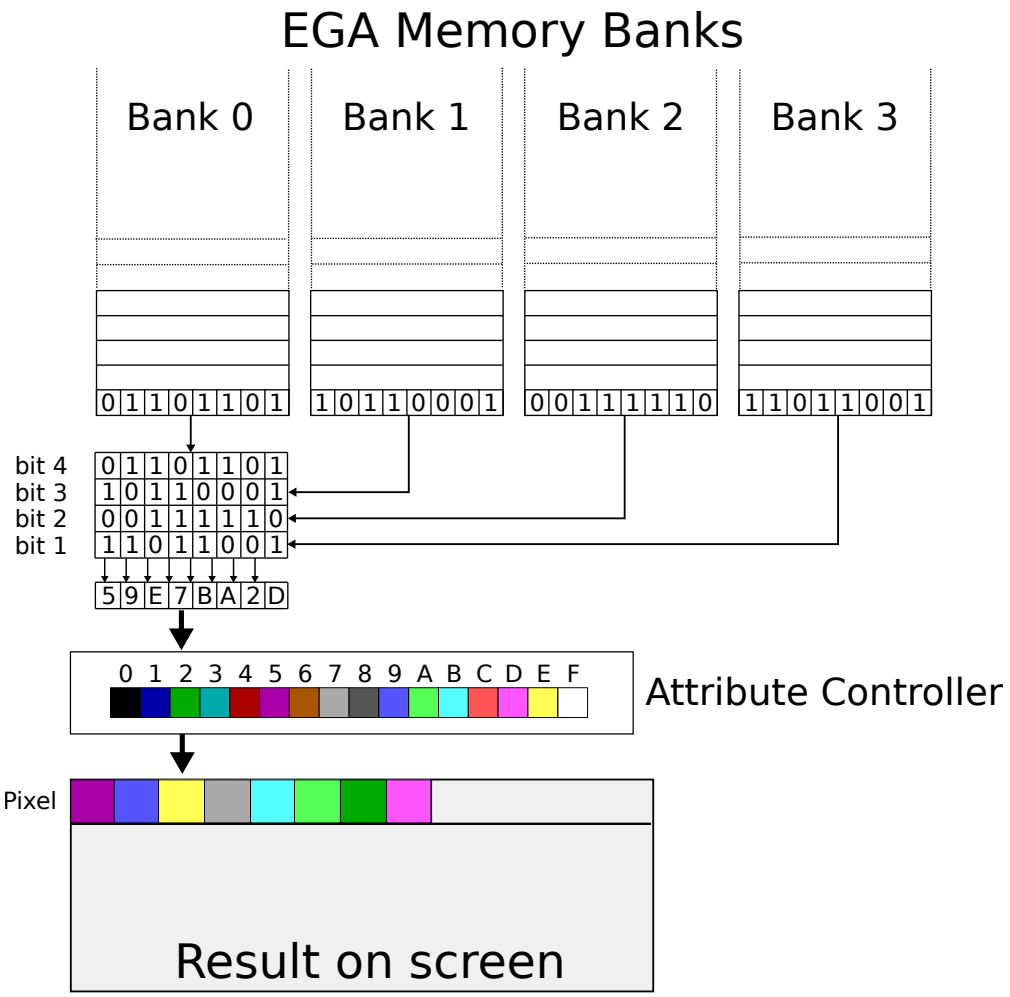


Figure 2.19: EGA mode 0Dh, how bank layout appears on screen.

2.3.6 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 2.18, which described the architecture, was actually deceptively simplified. Figure 2.20 shows how IBM's reference documentation explained the EGA. The maze of wire showcases well the actual complexity of the system.

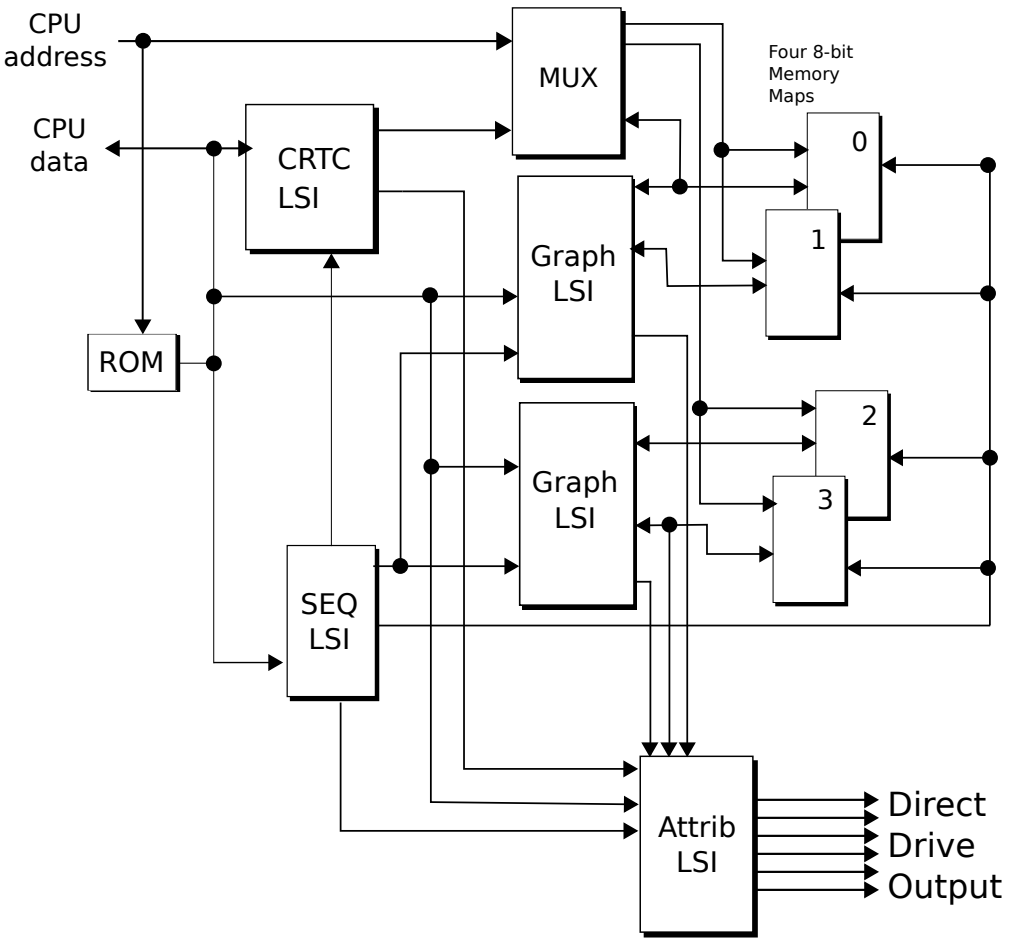


Figure 2.20: IBM's EGA Documentation.

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 2.21: 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.

2.3.7 EGA Programming: Memory Mapping

To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in figure 5.64. In mode `0Dh` for example, the VRAM is mapped from `A0000h` to `AFFFFh`. One of the first questions to come to mind is "How can I access 256KiB of RAM with only 64KiB

of address space?" The answer is "bank switching" as summarized in figure 2.22. Write and Read operations are routed based on a mask register indicating which bank should be read or written to.

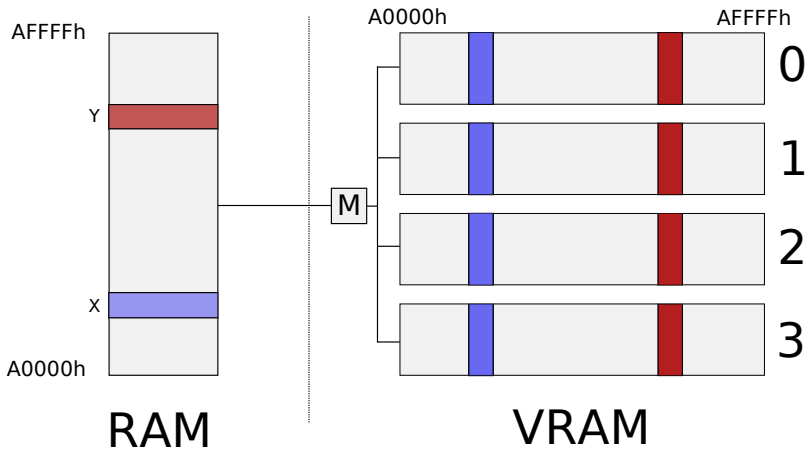


Figure 2.22: Mapping PC RAM to EGA VRAM banks.

2.3.8 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. 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 2.23: 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.

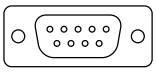


Figure 2.24: DE-9 video output connector.

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[18]. 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.

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 2.25: EGA and CGA DE-9 connector pin signals.

[18] IBM Enhanced Color Display documentation.

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 2.26: Default EGA 16-color palette

2.3.9 Double-Buffering in EGA

Fundamentally, a core goal in rendering is for each frame displayed on the monitor to present a single, coherent image.

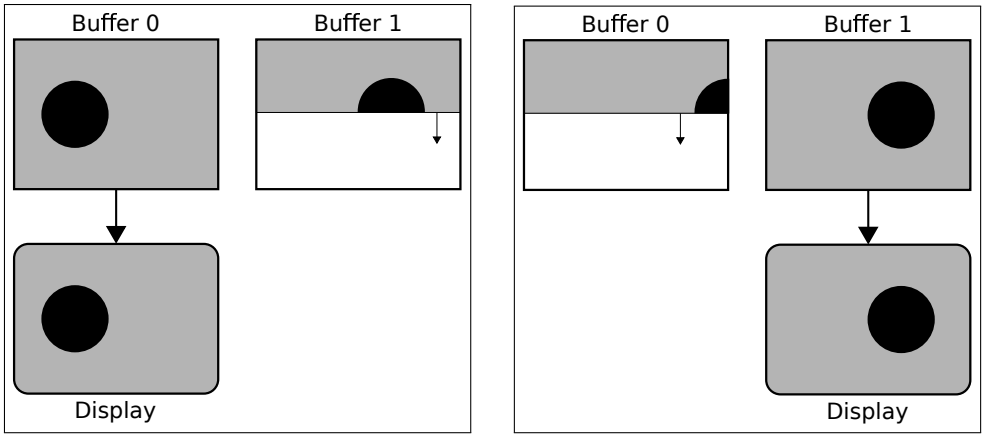
On the oldest hardware, there often wasn't enough memory to hold a full screen image, and so instead of drawing a screen image to VRAM, you needed to specify colors for each scanline individually, while the monitor was in the process of drawing that line. For example, on the Atari 2600, developers had just 76 machine instruction cycles to determine the color of each pixel in a scanline before the television started rendering it. This process repeated for every subsequent scanline until the entire frame was drawn.

Unlike early consoles, PC video cards contain dedicated VRAM, which the monitor reads from 60 times per second. Writing to VRAM, however, is controlled by the CPU. To avoid graphical artifacts, rendering must be timed carefully. If drawing a new frame takes too long, the scanline will "lap" the drawing process, resulting in a partially drawn frame. Conversely, if drawing gets ahead of the scanline, the new frame may overwrite part of the previous frame before it is fully displayed. This results in "tearing", where the top half of the screen displays one frame while the bottom half shows another.

Avoiding tearing with a single buffer is nearly impossible because writing to VRAM is significantly slower than reading from it. The scanline inevitably overtakes the rendering process, causing tearing.

Trivia : The original IBM CGA card could not simultaneously read from and write to VRAM. Because the video memory was not dual-ported, when both the CPU and the video card needed access to the same byte of VRAM, the CPU took priority. This resulted in the card reading a random value, causing a visual artifact known as "snow" on the screen. Later CGA clones could simultaneous read/write access to VRAM and did not have the same issue anymore.

A solution to tearing is double buffering, which introduces a second framebuffer in video memory. With two buffers, the CPU can draw the next frame in an off-screen framebuffer while the current one is being scanned to the display. Once drawing is complete, the buffers are swapped, ensuring a seamless transition between frames without tearing.



(a) Display buffer 0, rendering buffer 1.

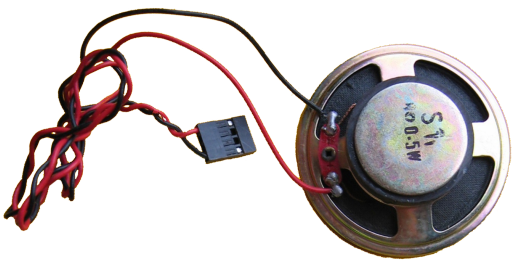
(b) Display buffer 1, rendering buffer 0.

Figure 2.27: Double-Buffering.

A full-screen image at 320 x 200 pixels with 16 colors requires 320 x 200 x 4 bits = 32 KB of VRAM (spread over four planes, each 8 KB). Given that EGA cards typically have 256 KB of VRAM, they have more than enough memory to support multiple buffers.

2.4 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.



2.4.1 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[19], in one of the PC's 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 gaming.

Figure 2.28: 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

[19]Midi Processing Unit-401

expansion card[20]. 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.

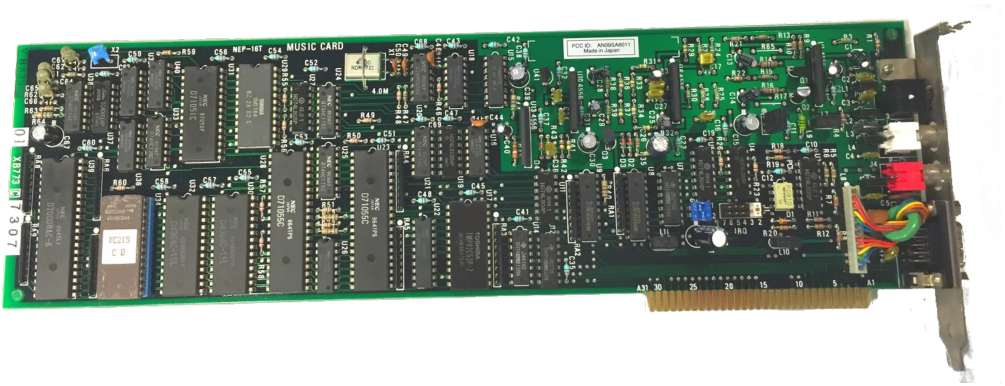


Figure 2.29: IBM Music Feature Card.

2.4.2 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. 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.

[20]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.

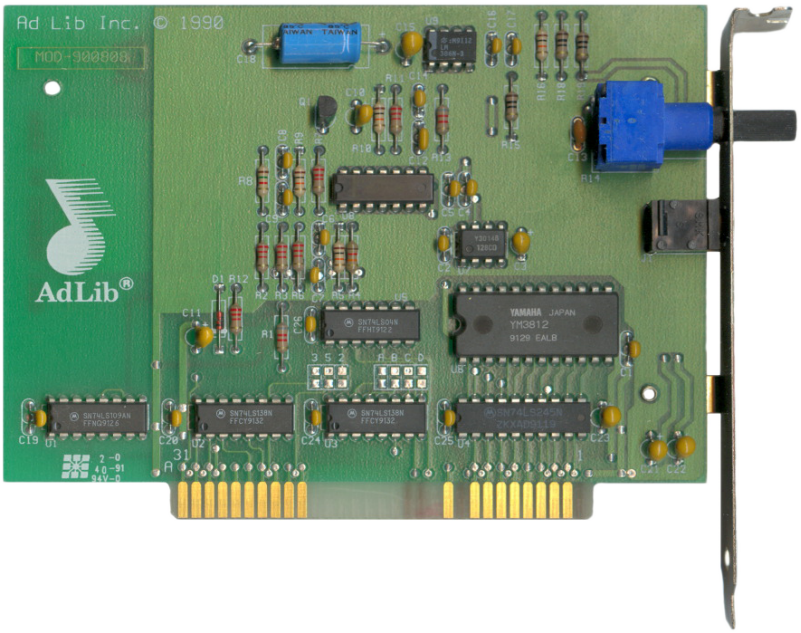


Figure 2.30: An AdLib sound card. Notice the big YM3812 chip.

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.

2.4.3 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 a 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[21] allowing PCM playback (digitized sounds) at 8 bits per sample and up to 22.05kHz sampling rate. The card also came with a DA-15 port allowing joystick connection. Most importantly, the SoundBlaster was $90 cheaper than the AdLib.

Figure 2.31: A SoundBlaster 1.5, model CT1320B

Figure 2.30 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. 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.

Trivia : Creative Labs boasts in their own advert AdLib compatibility and even uses an image of AdLib Inc's product[22]! On the other hand they sued every company that tried to market a 'Sound Blaster compatible' product for using the name of their product.

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

Figure 2.32: Sound Blaster advertisement with Adlib compatability.

2.5 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.

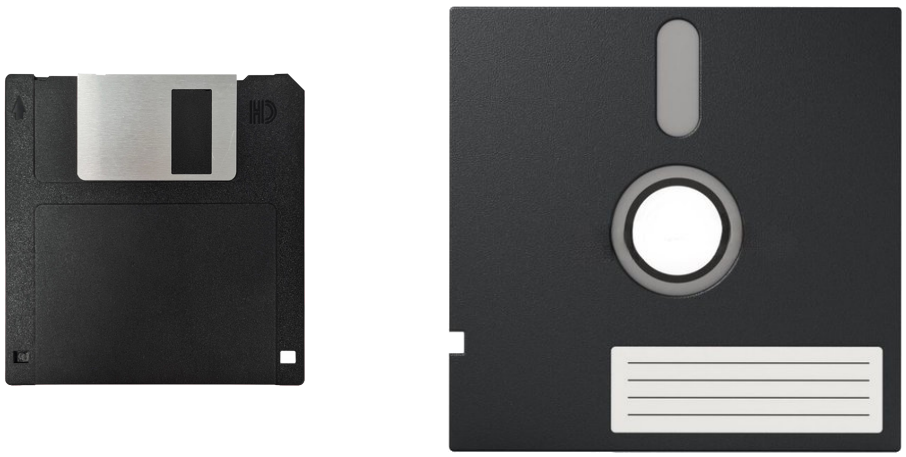


Figure 2.33: 3½-inch and 5¼-inch floppy disk.

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.

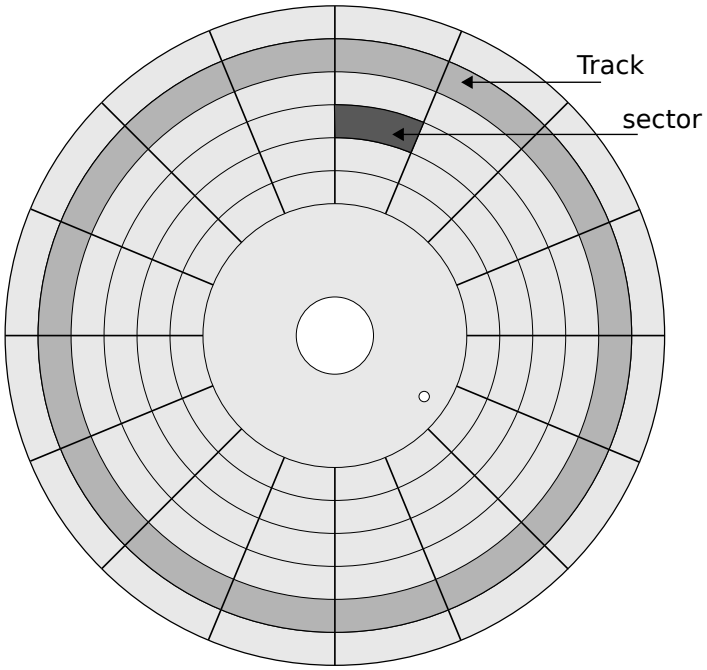


Figure 2.34: Floppy disk with tracks, sectors and the index hole.

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), and 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.

2.6 Keyboard

The original IBM PC and XT keyboards featured 83 keys. After receiving feedback from users frustrated by the layout, IBM introduced the 84-key PC AT keyboard, which included a rearranged layout and the addition of the "Sys Req" key. It also introduced 3 status LEDs for Caps Lock, Num Lock and Scroll Lock.

Figure 2.35: IBM AT Keyboard.

This design was further updated in 1986 with the release of the IBM Model M. It officially became the IBM PC standard in 1987 with the introduction of the IBM Personal System/2 (PS/2). The function keys were moved to the top, F11 and F12 were added, and the total number of keys increased to 101. The Model M was widely adopted and are still being used today.

Figure 2.36: IBM Keyboard model M (PS/2).

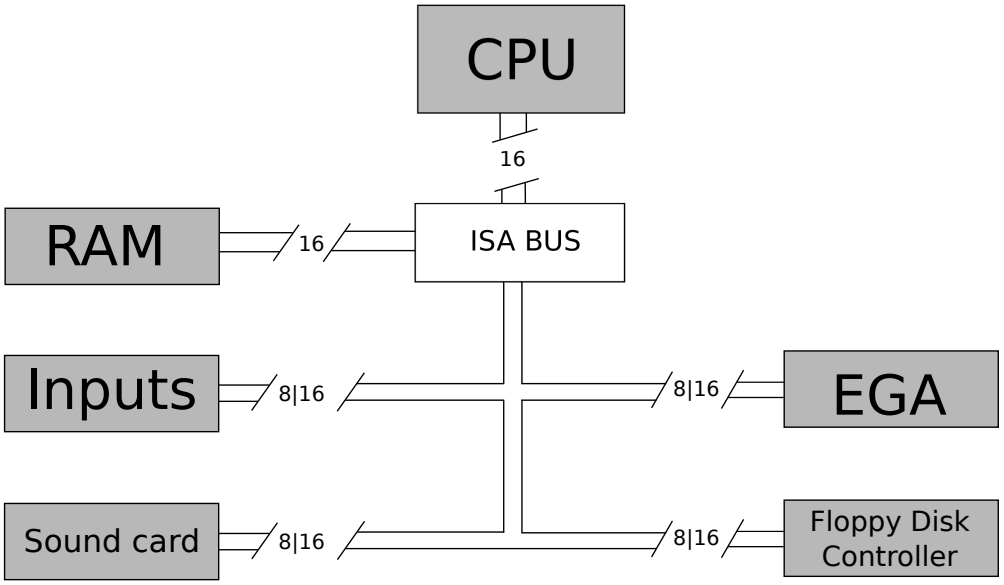
At the time, keyboards were connected either via PS/2 or AT ports.

Figure 2.37: PS/2 (left) and AT (right) keyboard connector.

2.7 Bus

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

The ISA[23] 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.



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

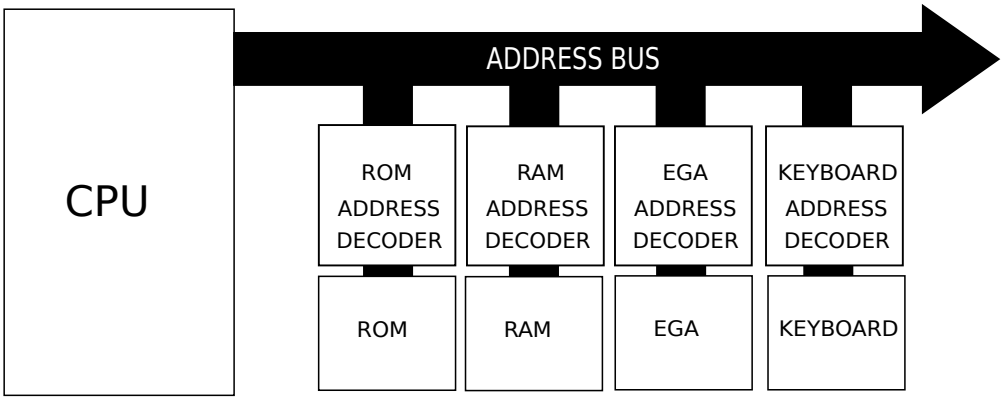
- 8 bits wide at 4.77 MHz for 19.1 Mbit/s
- 16 bits wide at 8.33MHz for 66.7 Mbit/s[24].

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

[23] Industry Standard Architecture.
[24] https://en.wikipedia.org/wiki/List_of_device_bit_rates .

disregards any request that is within the range [A0000h - AFFFFh].



2.8 Summary

To say a PC was difficult to program for games would be an understatement. It was a nightmare. The CPU was good at doing the wrong thing, the best graphic interface didn't allow double buffering, 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*.

Chapter 3

Assets

3.1 Programming

Development was done with Borland C++ 3.1 (but the language used was C). 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]. There was no need to enter command-line mode however. The IDE allowed to create a project, build, run and debug. The software came with two thick manuals, explaining everything regarding the IDE and programming in C++.

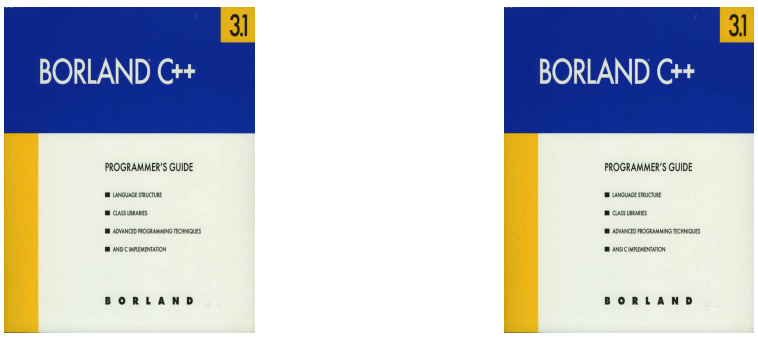
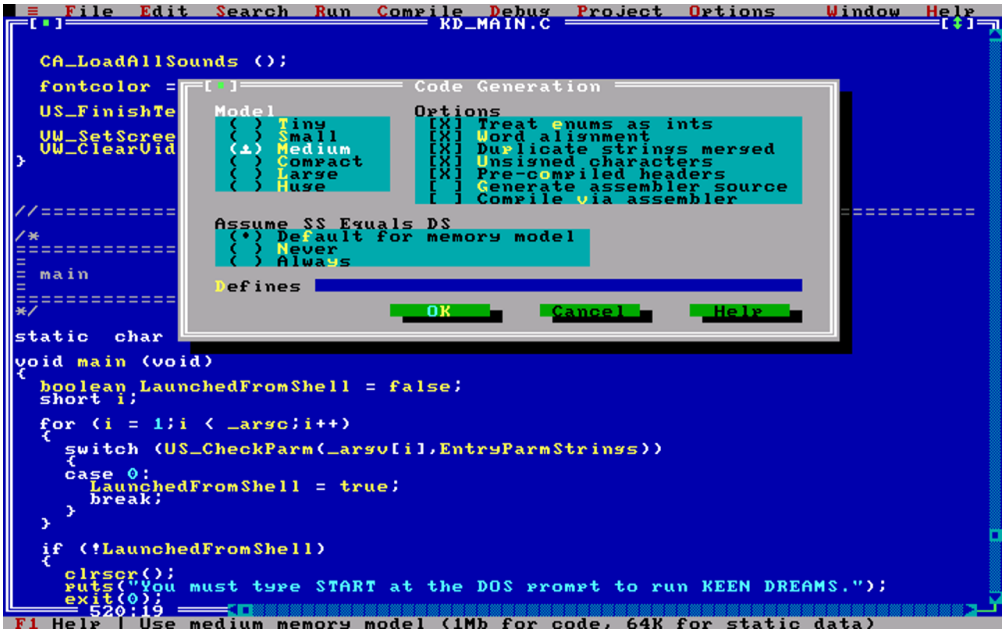
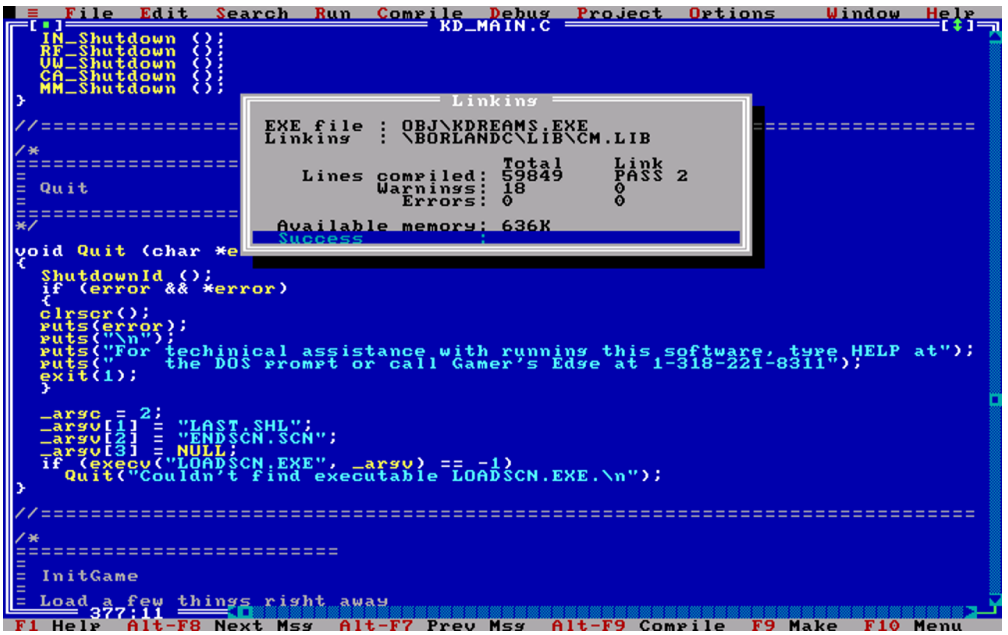


Figure 3.1: Borland C++ 3.1 User guide (238 pages) and Programmer guide (483 pages).

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

The IDE allowed to create a project, build, run and debug. It contained a "high resolution" 50x80 text mode, which doubled the standard vertical resolution.





3.2 Graphic Assets

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

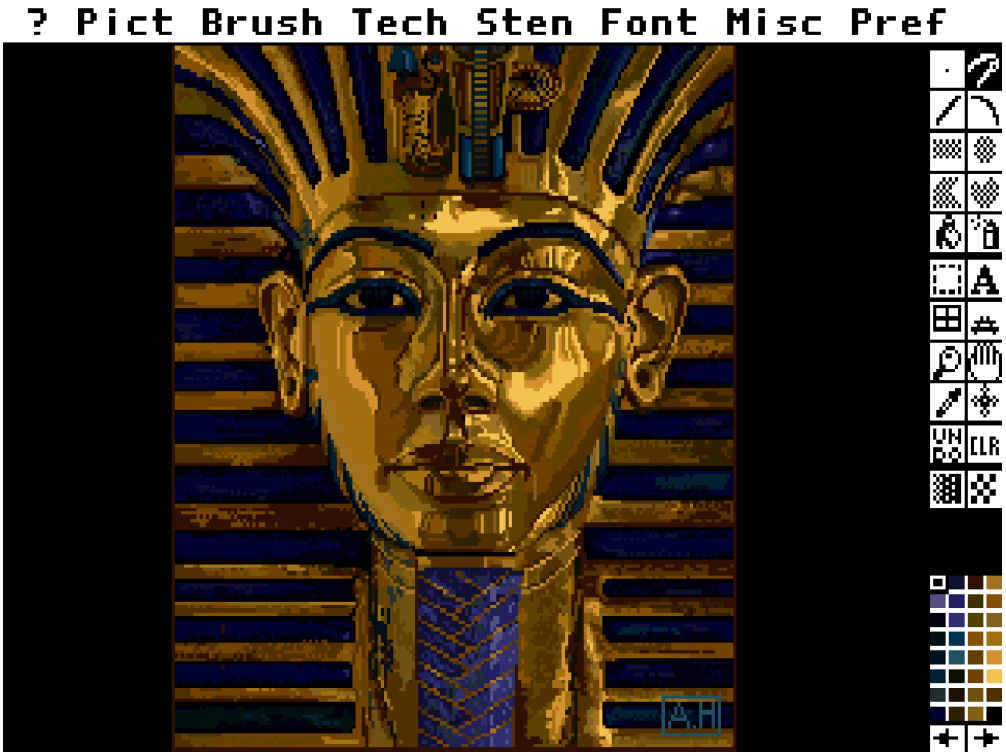


Figure 3.2: Deluxe Paint was used to draw all assets in the game.

3.2.1 Tile planar arrangement

Before diving into the details of game assets, let's first explore the basic principles behind platform games. Each level, or map, in a platform game is composed of "tiles". In Commander Keen, there are both background and foreground tiles. A background tile, or just "tile", is an image that measures 16x16 pixels. Each tile occupies 128 bytes (2 x 16 x 4 planes) of storage space in the graphics assets file.

[2] InterLeaved BitMap.

Individual tile images are stored in a planar format, with blue, green, red and intensity bits separated. This arrangement, called "graphic planar", stores complete planes sequentially, with each plane containing data for the entire tile.

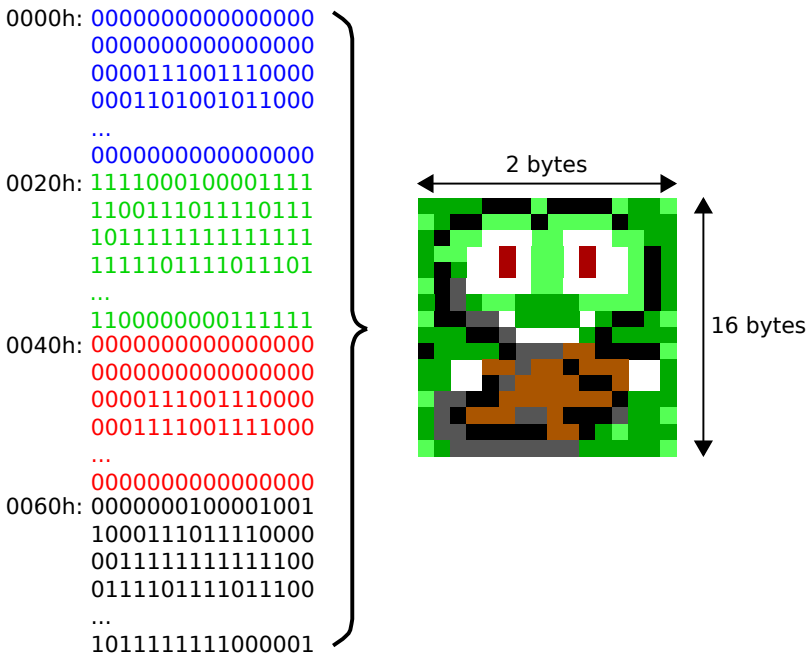


Figure 3.3: Graphic Planar data storage.

Foreground tiles, or "masked tiles", are similar to normal tiles but contain five planes instead of four. The extra plane stores the mask, and the order of planes is mask, blue, green, red, intensity. Consequently, a single masked tile requires 160 bytes (128 + 32) of storage. A mask bit of '0' means the background tile color is erased and replaced by the foreground tile color, while a mask bit of '1' blends the background color with the foreground color. If the foreground color is '0', the background color remains visible.

By combining tiles on the screen, a map is created. These maps define the entire game world in terms of background and foreground tiles. Maps also contain a list of all actors and their starting positions within the world. Essentially, everything the player encounters while progressing through the levels is specified in a map.

3.2.2 Assets Workflow

After the graphic assets were generated, a tool (IGRAB) packed all ILBM files into a compressed data asset file (EGA-file) and generated a HEAD table, DICT file (KDR-files[3]), and a C header file containing asset IDs. The engine references an asset directly using these IDs.

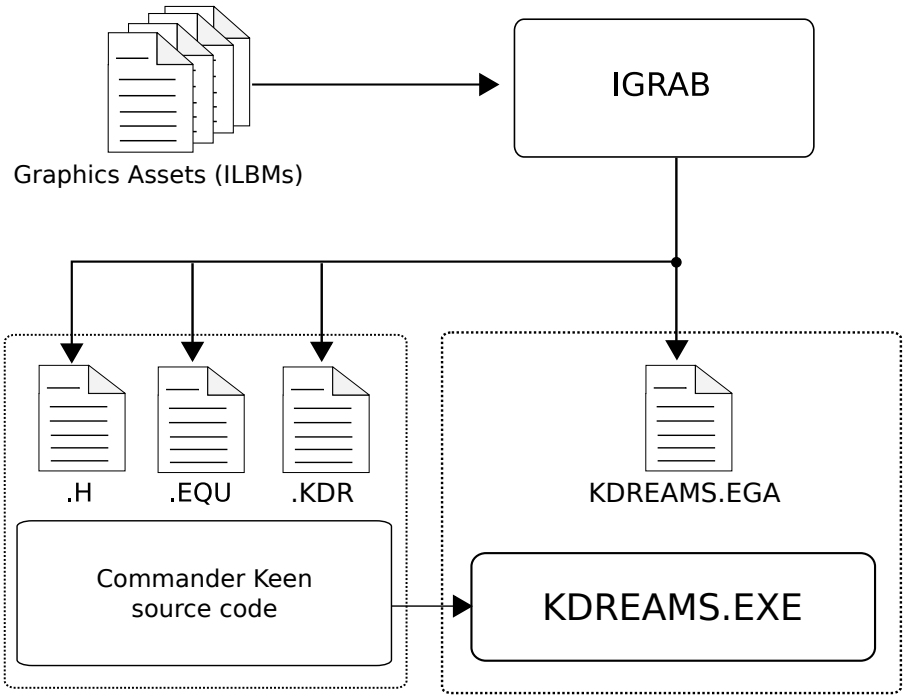


Figure 3.4: Asset creation pipeline for graphics items

In the engine's code, asset usage is hardcoded via an enum. This enum serves as an offset into the HEAD table, which then provides an offset within the data asset file. 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 asset file from the shareware version of *Keen Dreams*. However, the latest release of the source code (v1.93) does not match with shareware version v1.13. To ensure that the HEAD, DICT, and data asset file are compatible, you will need to retrieve a specific git commit, 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
...
```

3.2.3 Assets file structure

Figure 4.5 illustrates the structure of the `KDREAMS.EGA` asset file. The first section contains data tables for pictures and sprites, followed by the font, and all sprite and tile graphs.

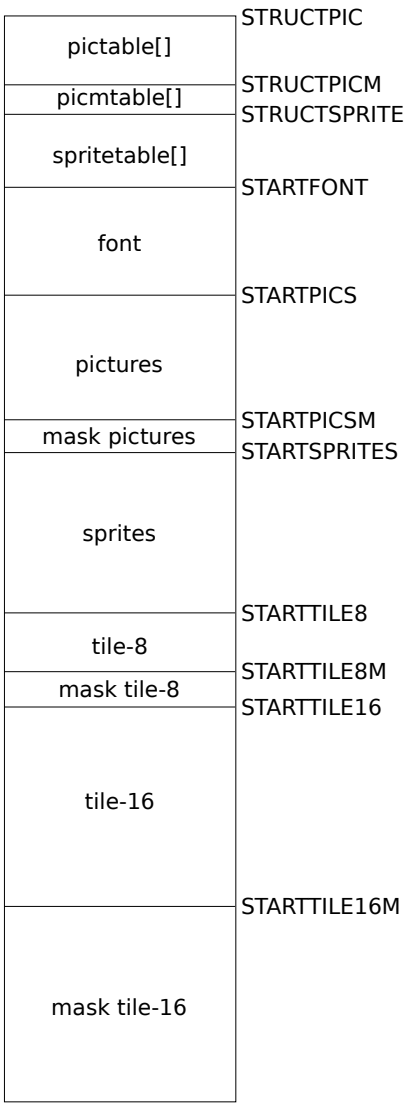


Figure 3.5: File structure of `KDREAMS.EGA` archive file.

Both 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
...	...	...
64	5	24

The `spritetable[]` includes not only width and height but also information on the sprite's center, hitbox, and number of bit shifts (explained section "Drawing sprites" on page 153).

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

All sprite placement occurs from the origin, which is offset by `(orgx, orgy)` from the sprite's top-left corner. The parameters `(xl, xh, yl, yh)` define the sprite's hitbox, used for collision detection. Sprites have an additional layer to indicate transparency, similar to masked tiles.

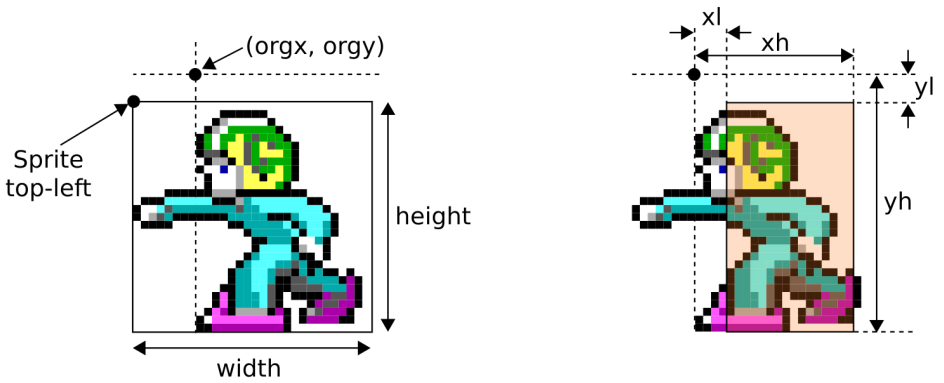


Figure 3.6: Sprite dimensions, origin and hitbox.

The font segment contains a table that stores the height and the width of the font, along with a reference to where the character data is located in the archive file.

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

Figure 3.7: Font asset data.

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

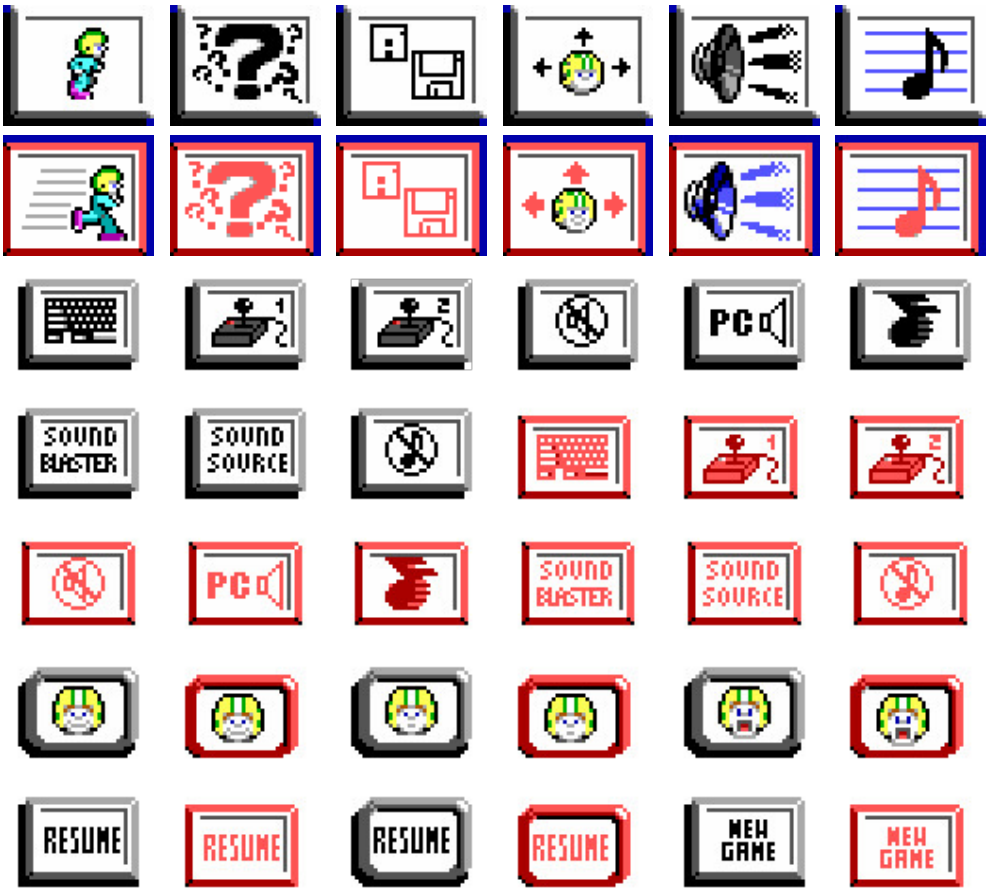


Figure 3.8: Picture assets.

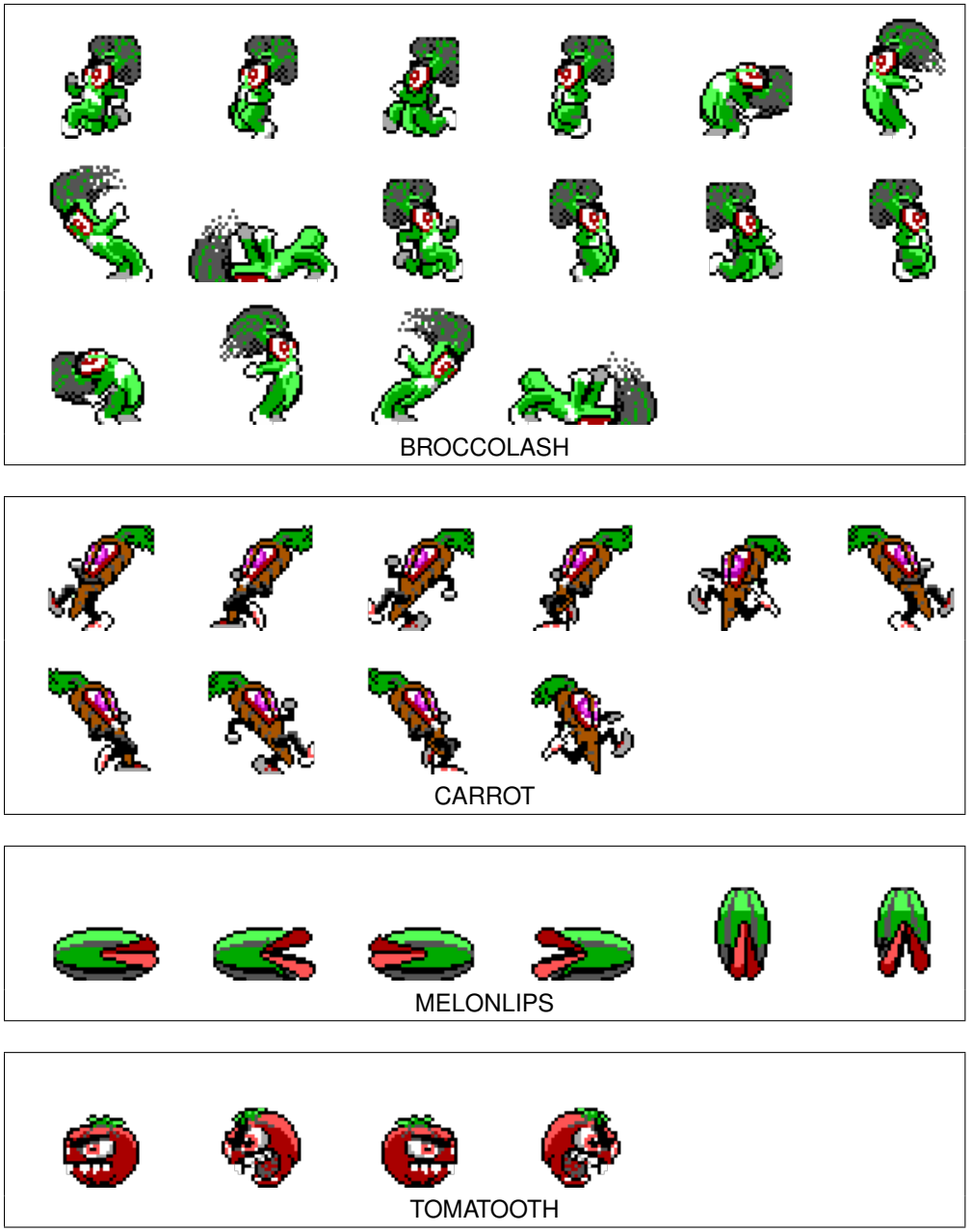


Figure 3.9: Sprite assets.

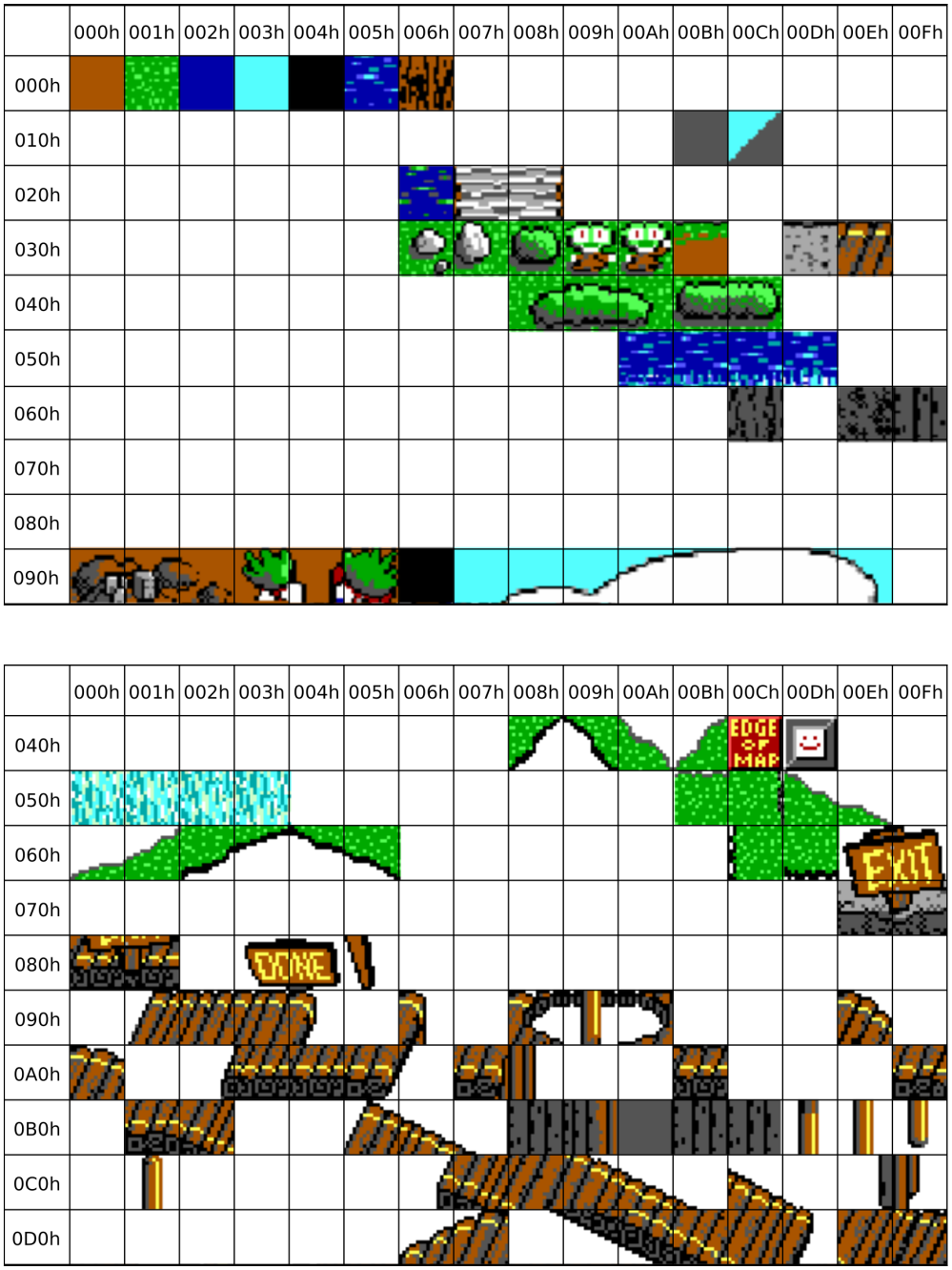
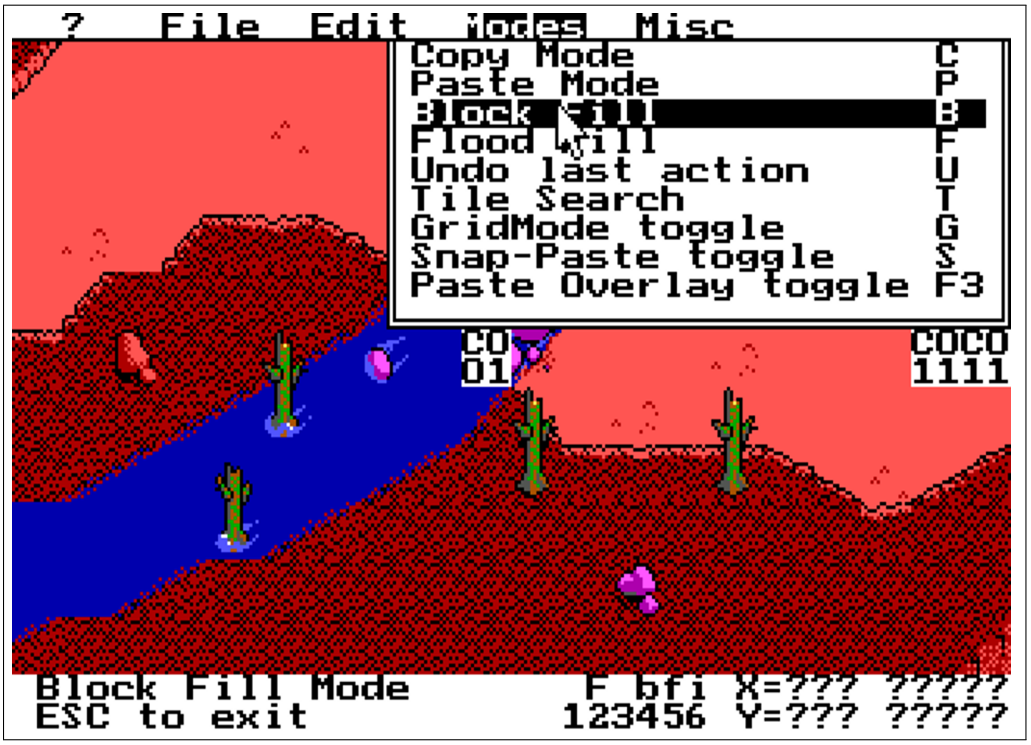


Figure 3.10: Background (Tile-16) and foreground (masked Tile-16) assets.

3.3 Maps

Maps were created using an in-house editor called TED5, short for Tile EDitor. Over the years TED5 had improvements and the same tool is later used for creating maps 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 Figure 4.4 on page 65). This way, tile IDs are directly encoded in the map.



TED5 allows the placement of tiles across multiple layers, referred to as "planes". In Commander Keen, there are three types of planes:

- Background plane tiles.
- Foreground plane tiles, which act as a mask over the background plane.
- Information plane tiles, which contain actor locations and special areas.

A level is created by placing tiles on each of these three layers. Foreground and background tiles can be enhanced with additional properties ("Tile info"), which controls interactions such as clipping, "deadly" tiles, and animated tiles.

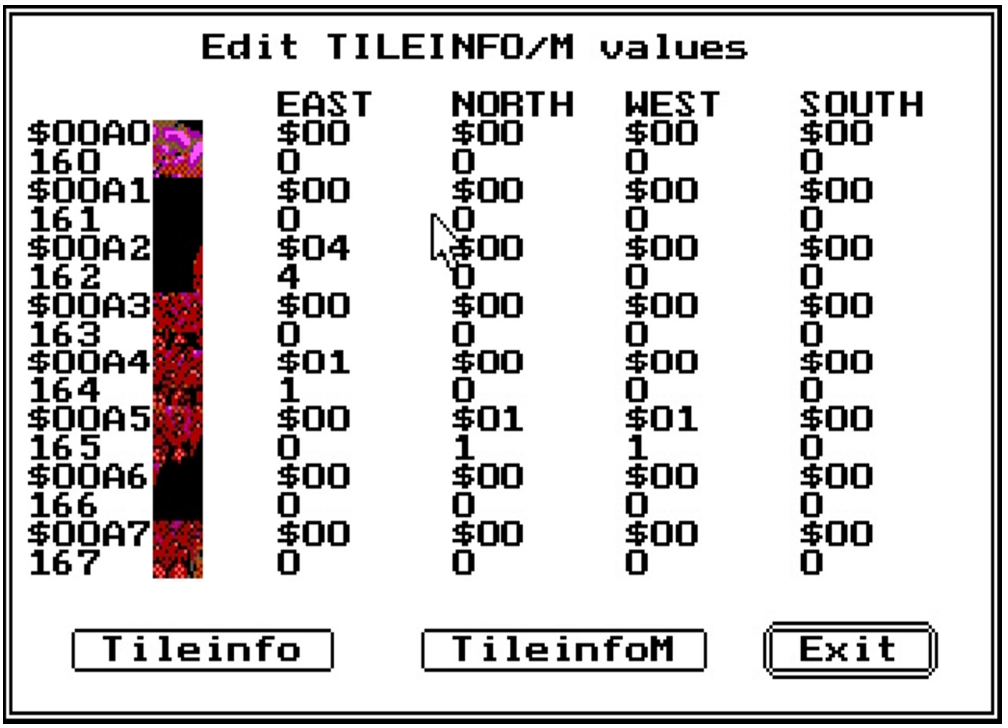


Figure 3.11: "Tile info" adding properties to tiles.

Similar to IGRAB, TED5 compresses all levels into a `MAP` archive and generates corresponding `HEAD` and `DICT` files.

3.3.1 Map header structure

The map header structure is embedded in the engine code and contains a header offset and size, which refer to the location and size within the `KDREAMS.MAP` archive file. The game supports a maximum of 100 maps. The `tileinfo[]` array contains properties data for each tile.

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

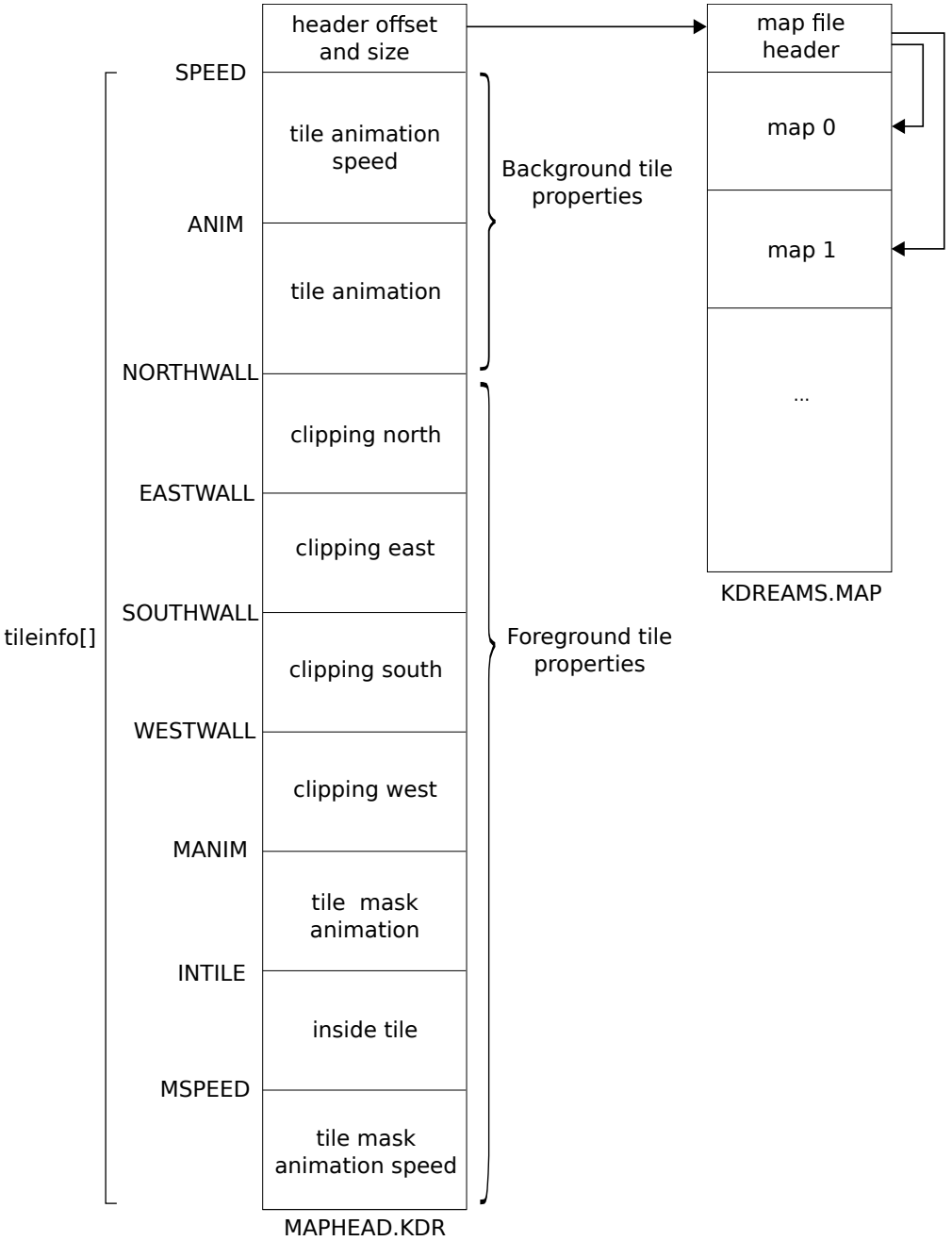


Figure 3.12: Structure of `MAPHEAD.KDR` header file.

For background tile animations, two information tables are defined: "tile animation" and "tile animation speed". The tile animation specifies the next tile in the animation sequence, relative to the current tile. For example, in Table 4.1, tile #90 animates to tile #91 (+1), followed by #92 (+1), and #93 (+1). After tile #93, the sequence returns to tile #90 (-3). The animation speed is expressed in number of ticks before the next tile is displayed.

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

Table 3.1: Background tile animation.

Tiles also contain clipping information and a column called `intile`. The `intile` column tells what the tile "does" to the player, like kill, climb, points, etc. Values between 128-255 are used for foreground tiles only. The `intile` column is customized for each version of *Commander Keen*. In *Keen Dreams* it is used for e.g. climbing poles.

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

Table 3.2: Foreground tile clipping and 'intile' tile information.

3.3.2 Map archive structure

The structure of the KDREAMS.MAP archive is illustrated in Figure 4.13. Each map contains a small header that includes the width, height, and name of the map, as well as a reference pointer to each of the three planes. Each plane consists of a map of tile numbers representing the background, foreground, and information planes.

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

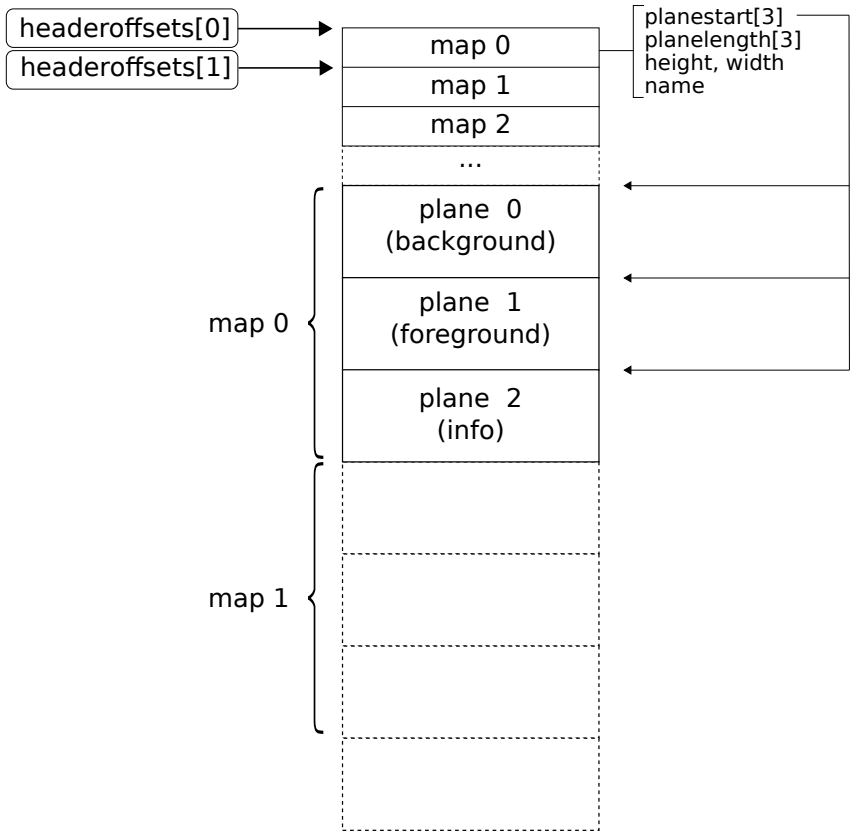


Figure 3.13: Structure of KDREAMS.MAP file.

3.4 Audio

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[4]. The team didn't even have time to remove the music setup menu.

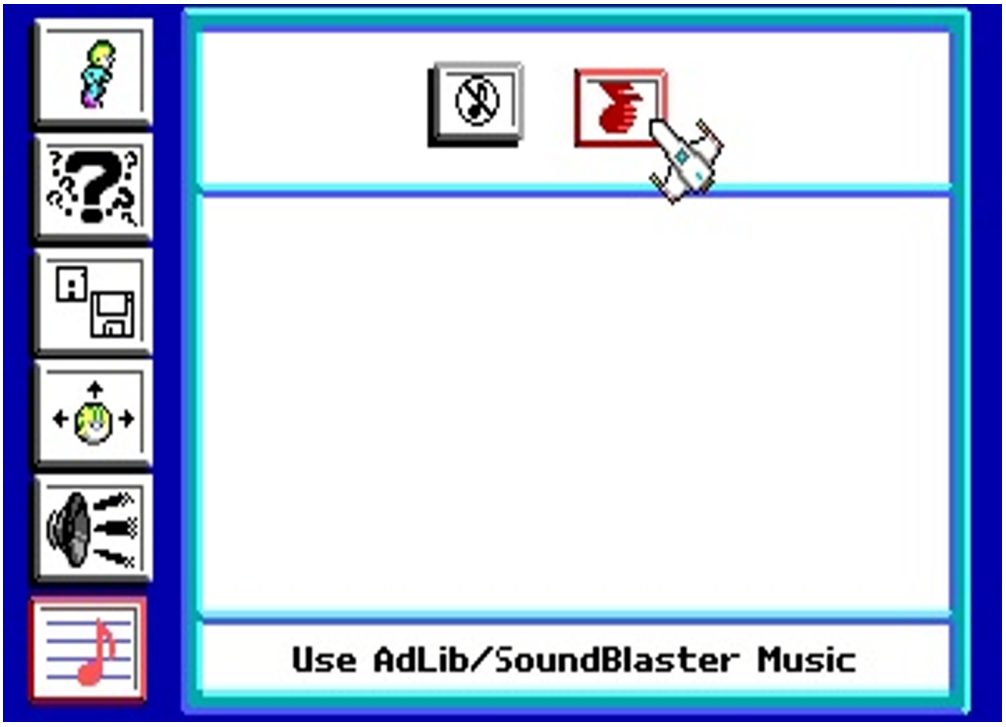


Figure 3.14: Setup music menu in Keen Dreams, although there is no music in the game.

[4] https://vimeo.com/4022128, at 2:00-4:55.

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

1. For PC Speaker
2. For AdLib

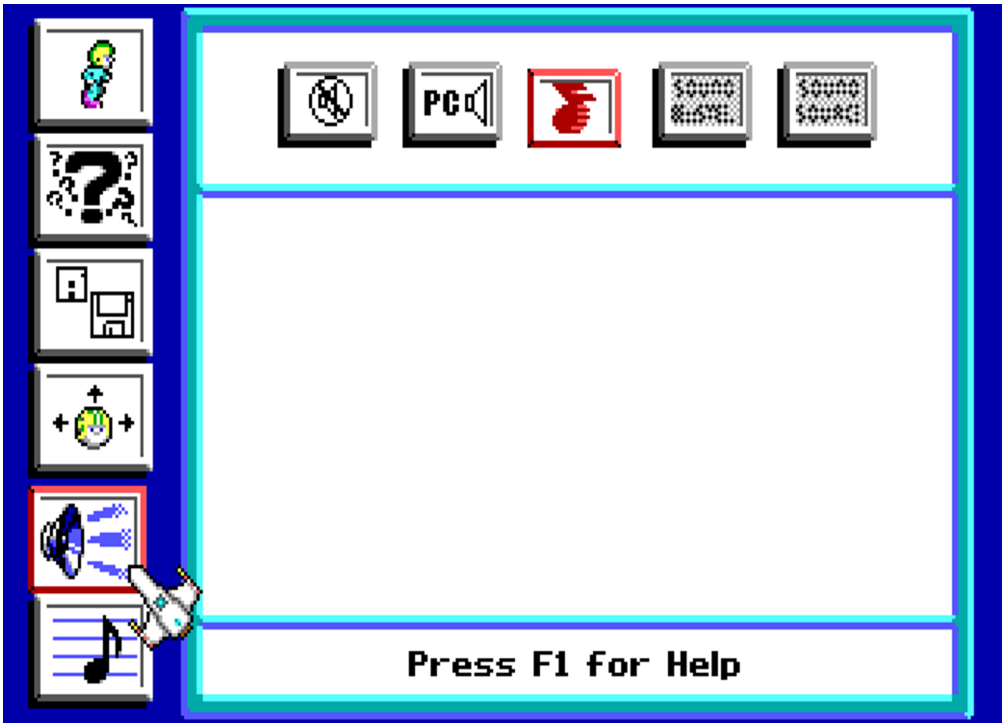


Figure 3.15: Both Sound Blaster and Sound System are disabled.

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

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 Blackbook*[5] 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.

[5]See Wolfenstein 3D Blackbook, section 3.6.

3.5 Distribution

On December 14th, 1990 the first episode was released via Apogee. Episodes 1-5 are all published by Apogee Software. The game engine and first episode were given for free and encourages to be copied and distributed to a maximum number of people. To 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[6].

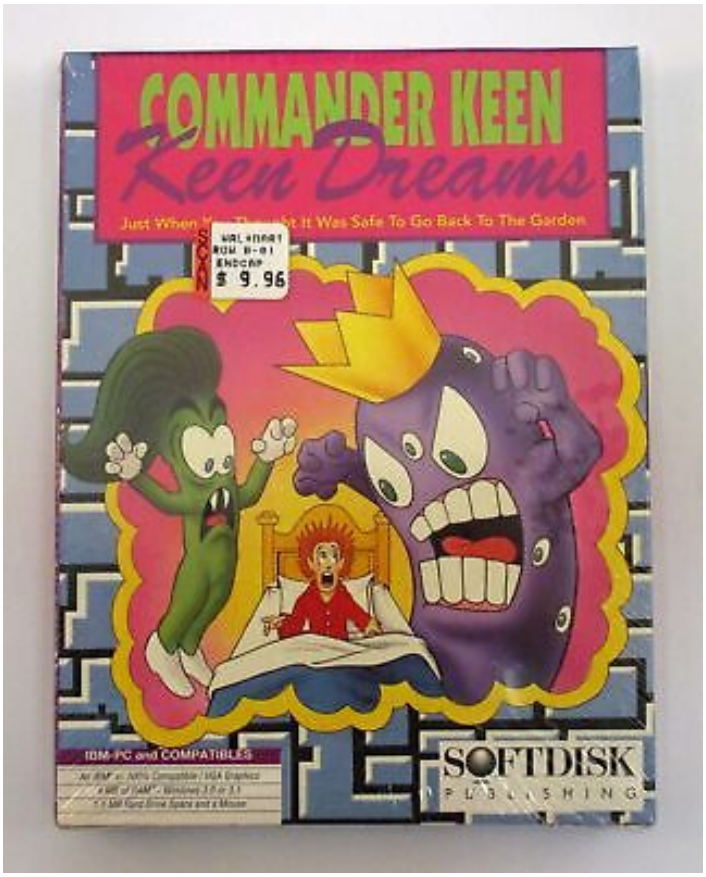


Figure 3.16: Retail version of Commander Keen in Keen Dreams by Softdisk.

[6]The settlement with Softdisk is explained in Appendix D

In 1990, the Internet was still in its infancy and the best medium was the 3½-inch floppy disk. The shipped game can be divided in sevent 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 level layouts.
- KDREAMS.CMP: Introduction picture of the game, a compressed LBM image file.
- Softdisk Help Library files, which are text screens, shown when starting and ending the game
- 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 3.17: All Keen Dreams files as they appear in DOS command prompt.

Chapter 4

Software

4.1 About the Source Code

Commander Keen episodes 1-5 source code is unavailable, as the current owner Zenimax[1] has, at the time of writing, shown no interest in selling the intellectual property. Luckily the ownership of Commander Keen in Keen Dreams was in the hands of Softdisk. In June 2013, developer Super Fighter Team licensed the game from Flat Rock Software, the owners of Softdisk at the time, and released a version for Android devices.

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

4.2 Getting the Source Code

The source code is available on `github`. It is essential to use the source code from the shareware version 1.13, or you may encounter issues due to incompatible map and graphs headers[2]. To get the correct source code, use the following commands in the console

```
$ 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.

4.3 First Contact

After downloading the repository from `github`, a folder named 'keen' is created, containing all the source files. The `cloc.pl` tool can be used to analyze the folder and generate statistics about the source code. This tool is useful 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
-------------------------------------------------------------
```

Approximately 85% of the code is in C, with assembly[3] used for bottleneck optimizations and low-level I/O operations such as video and audio handling.

Source lines of code (SLOC) are not particularly meaningful for comparing a single codebase but are helpful for determining the proportion of code in various components. Commander Keen has 19,237 SLOC, which is small compared to most modern software. For instance, `curl` (a command-line tool for downloading URLs) has 154,134 SLOC, Google Chrome has 1,700,000 SLOC, and the Linux kernel consists of 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`.

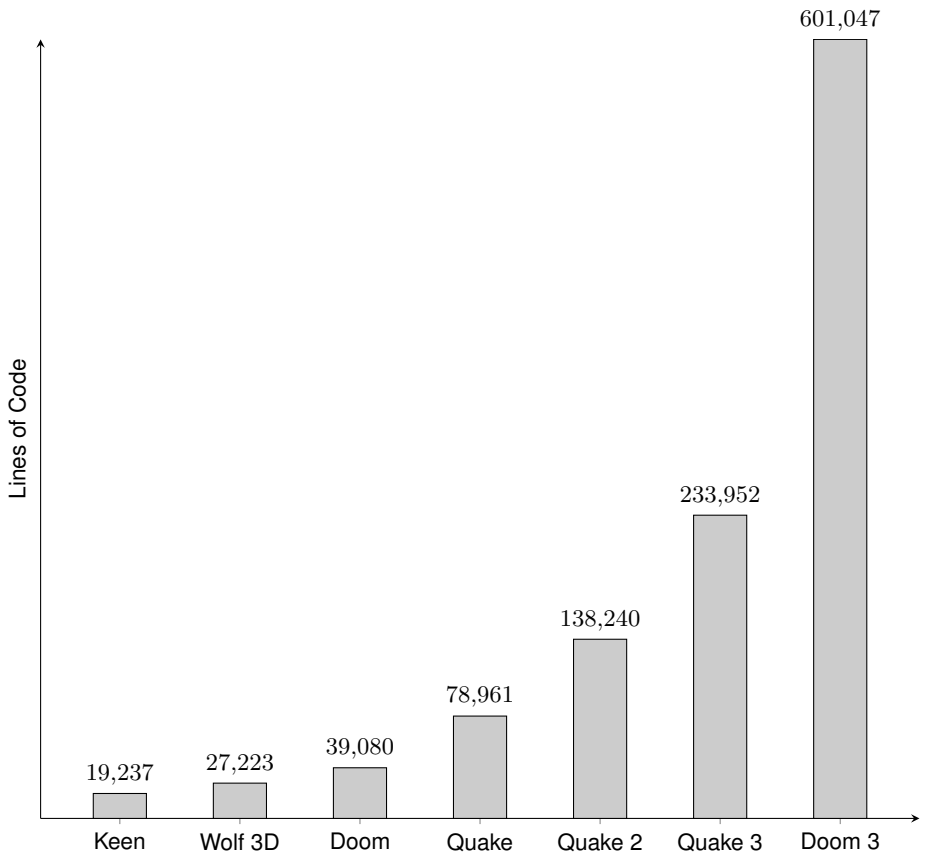


Figure 4.1: Lines of code from id Software game engines.

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

- A `static` folder with static header files for loading assets and maps (as explained in section "Graphic Assets" on page 63).
- An `lscr` folder for loading and decompressing Softdisk text files.
- A `README` file explaining how to build the executable.

4.4 Compile source code

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

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

After setting up DOSBox and installing Borland C++ 3.1[4], download the source code from github.

Once DOSBox is running and you've navigated to the `keen` folder, create a folder to store your compiled object files.

```
mkdir OBJ
```

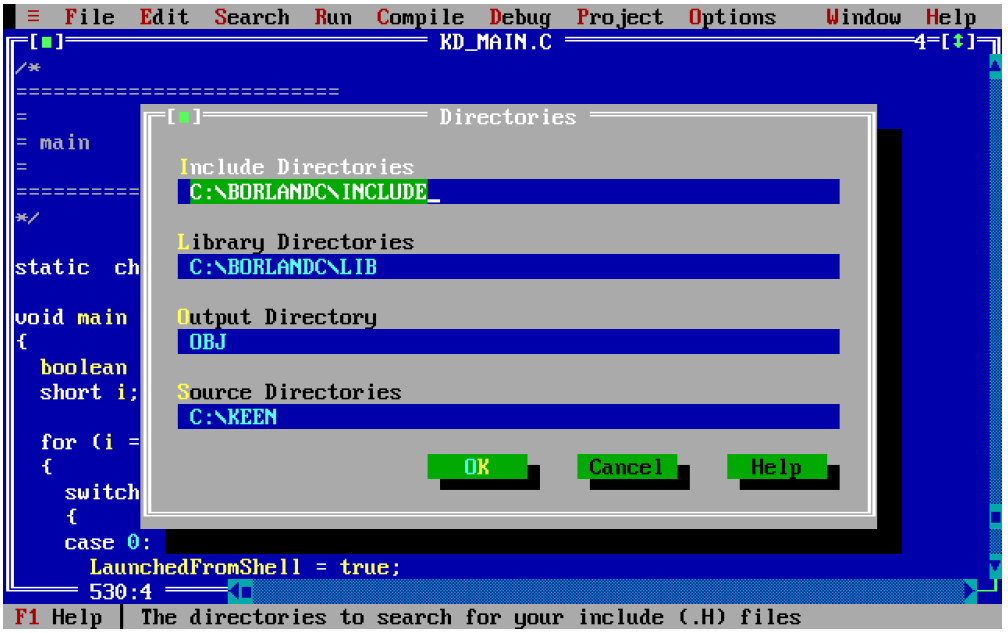
Then, generate the static `OBJ` header files.

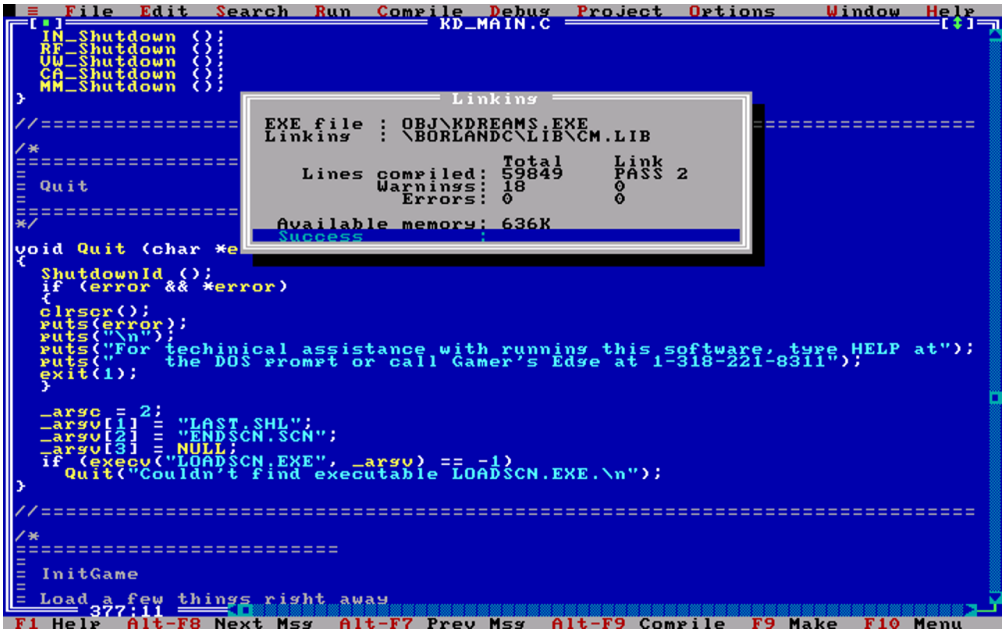
```
chdir STATIC
make.bat
```

After that, return to the `keen` folder and open Borland C++. Open the `kdreams.prj` project file. Before compiling, adjust the directory settings by selecting Options -> Directories and change the values as shown on the next page.

Now it's time to compile. Go to Compile -> Build All, and voilà! The final step is to copy `kdreams.exe` into the Keen shareware folder. You can now play your compiled version of *Commander Keen*.

[4] you can find a complete tutorial in "Let's compile like it's 1992" on fabiensanglard.net


File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
Directories
Include Directories
C:\BORLANDC\INCLUDE
Library Directories
C:\BORLANDC\LIB
Output Directory
OBJ
Source Directories
C:\KEEN
OK
Cancel
Help
F1 Help | The directories to search for your include (.H) files


File Edit Search Run Compile Debug Project Options Window Help
KD_MAIN.C
Linking
EXE file : OBJ\KDREAMS.EXE
Linking : \BORLANDC\LIB\CM.LIB
Total Link
Lines compiled: 59849 PASS 2
Warnings: 18 0
Errors: 0 0
Available memory: 636K
Success
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

4.5 Big Picture

The game engine is divided in three blocks:

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

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

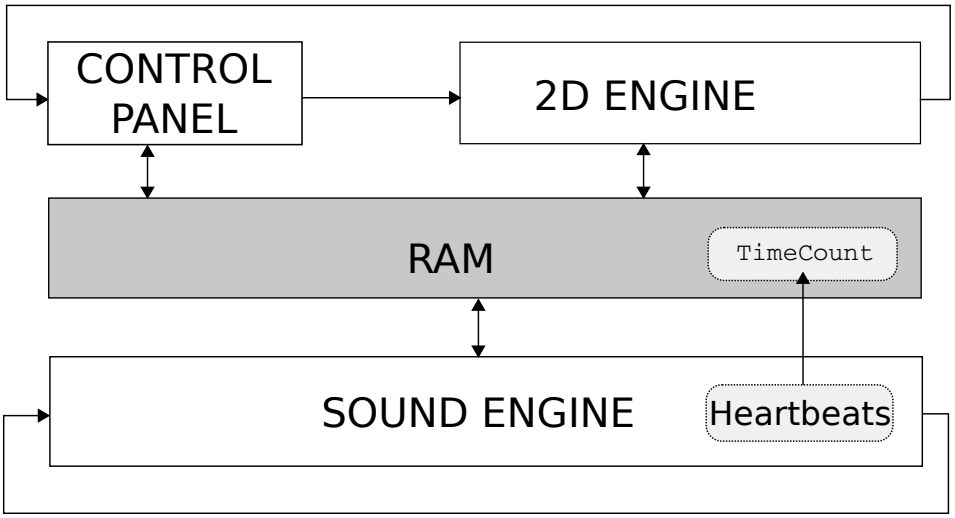


Figure 4.2: Game engine three main systems.

4.5.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop, starting with `void main()`. The control panel and 2D renderer follow regular loops, but due to limitations discussed later, the sound system is interrupt-driven and therefore operates outside the main loop. In real mode, C types do not behave as expected on a 16-bit architecture.

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

The first action taken by the program is to set the text color to light grey and the background to black.

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

  InitGame();

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

In `InitGame()`, the program checks whether there is enough memory 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);

    [...]             // Handle collisions between objects
    ScrollScreen(); // Scroll if Keen is nearing an edge.
    [...]             // React to whatever happened.
    RF_Refresh();     // Update buffer screen and 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 5.13 "Audio and Heartbeat". In short, with an OS supporting neither processes nor threads, it was the only way to have something execute concurrently with the rest of the engine.

4.6 Architecture

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

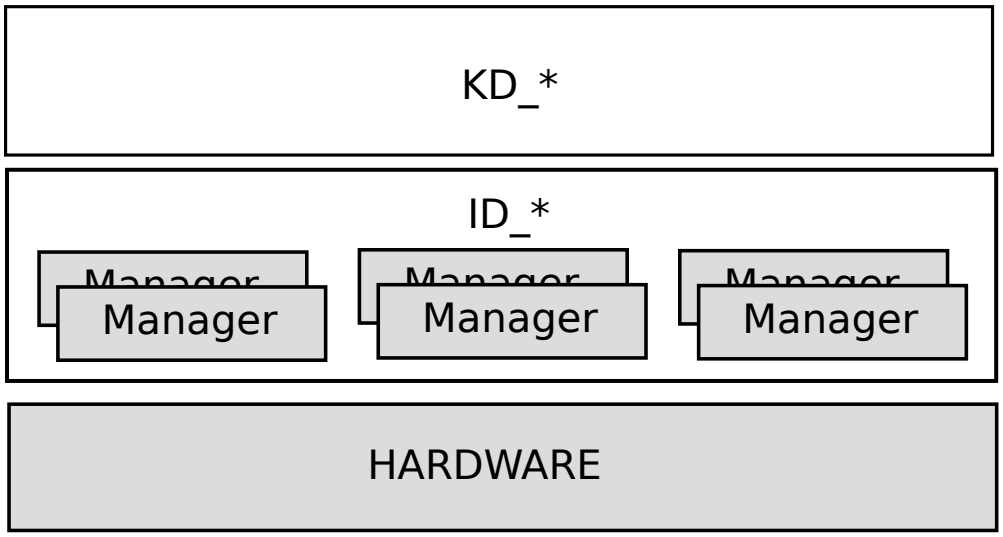


Figure 4.3: Commander Keen source code layers.

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 Software games (*Hovertank One*, *Catacomb 3-D* and *Wolf3D*). Next to the hard drives (HDD) you can see the assets packed as described in section "Graphic Assets" on page 63.

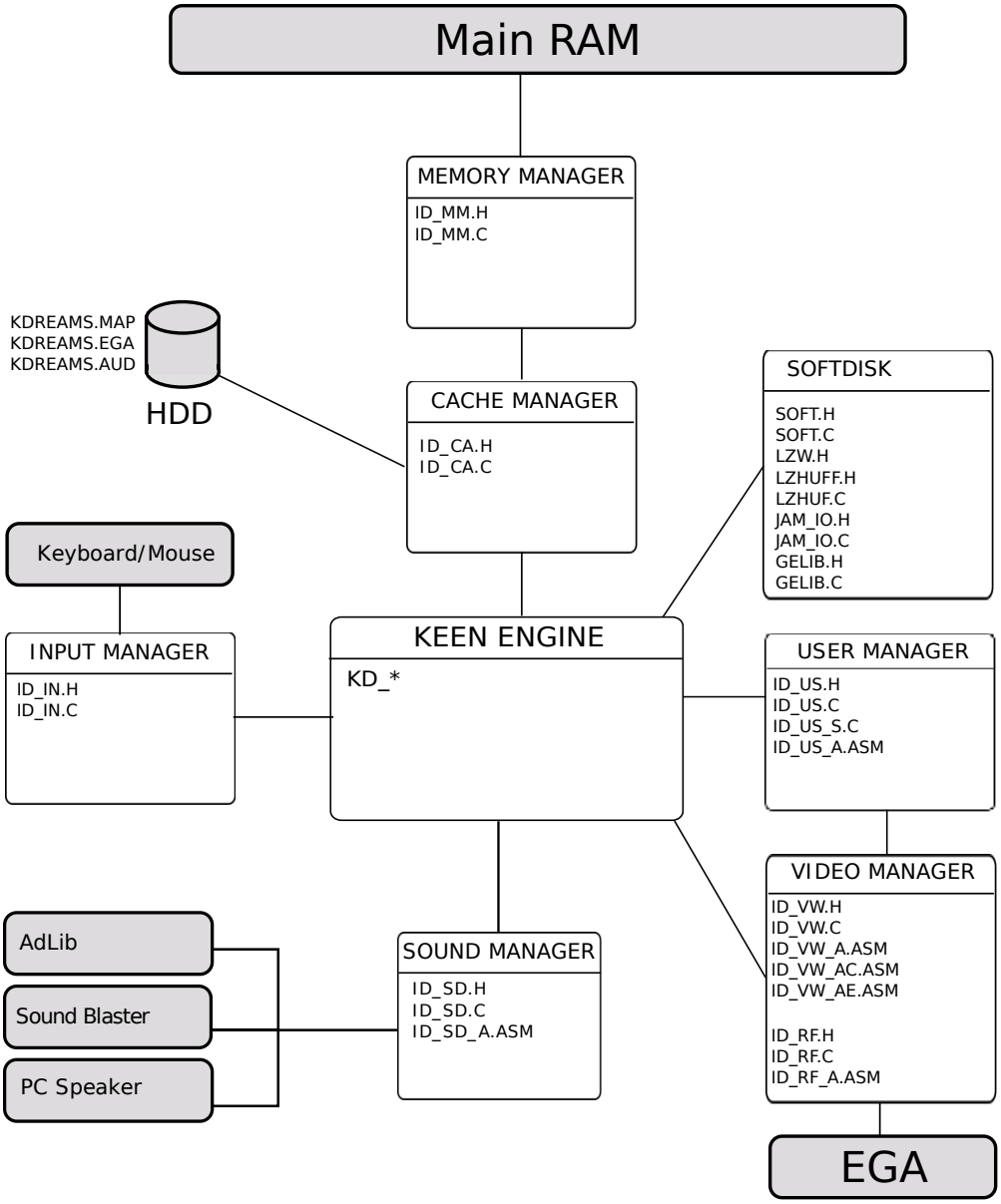


Figure 4.4: Architecture with engine and sub-systems (in white) connected to I/O (in gray).

4.6.1 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 allocates all RAM, starting from 0000:0000h, and assigns segments to the linked list. Since the engine is compiled using the medium memory model, there are two heaps: 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 is segment-aligned, meaning it allocates memory blocks in chunks of 16 bytes.

```
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 block allocated is the unusable 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;
```

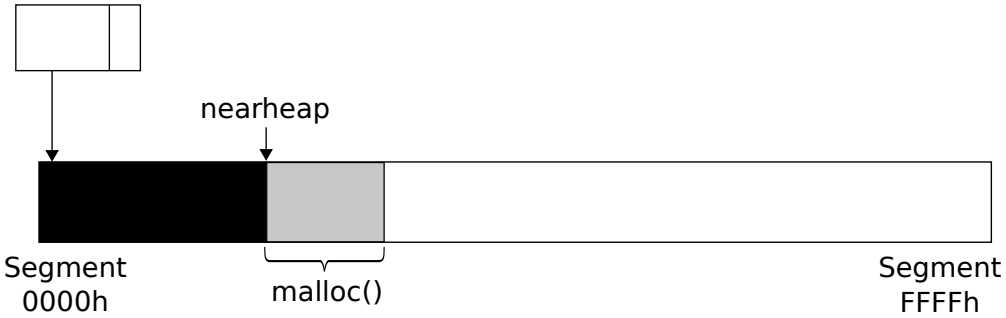


Figure 4.5: First locked block of unusable memory.

The stack is the next unusable memory block. Since the stack will grow or shrink during game execution, an additional part of the near heap's free memory must be reserved for it.

```
#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 block of unusable memory is between the start of the stack memory (including the reserved block) and the start of the far heap.

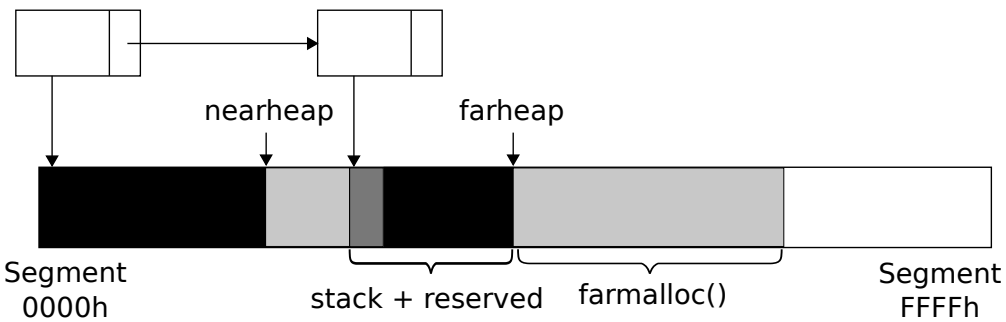


Figure 4.6: Second locked block: stack.

Finally, the last unusable block of memory lies 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 space from segment `0000h` to `FFFFh` is allocated, chained together, and fully controlled by the memory manager.

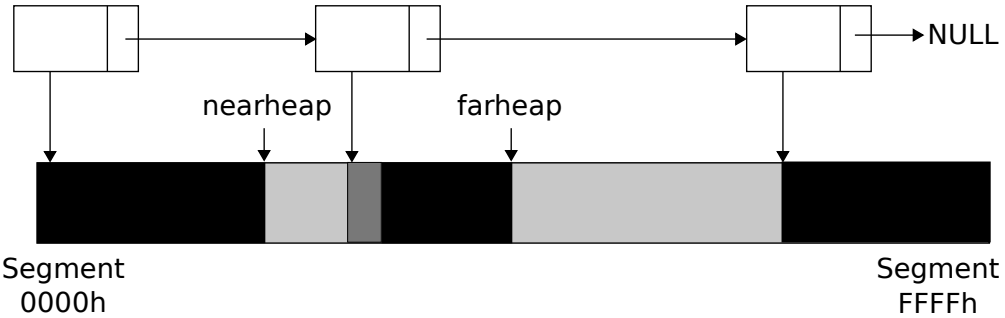


Figure 4.7: Final initial memory manager state.

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[5].

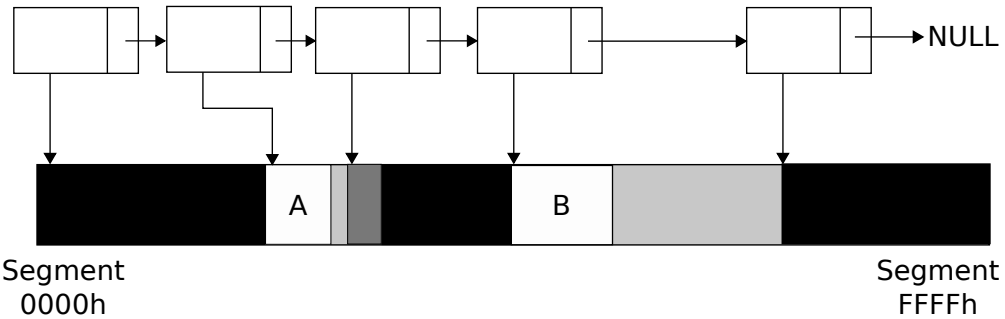


Figure 4.8: Allocation of memory.

By calling `MM_ShowMemory`, you can inspect the current memory usage during gameplay. Each pixel represents one memory segment of 16 bytes. Red indicates blocked memory, black indicates available memory, blue is unpurgeable memory, and purple is purgeable memory. Small white pixels indicate the start of new memory blocks in the linked list. Notice how small the near heap (the first black line) is compared to the far heap!

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

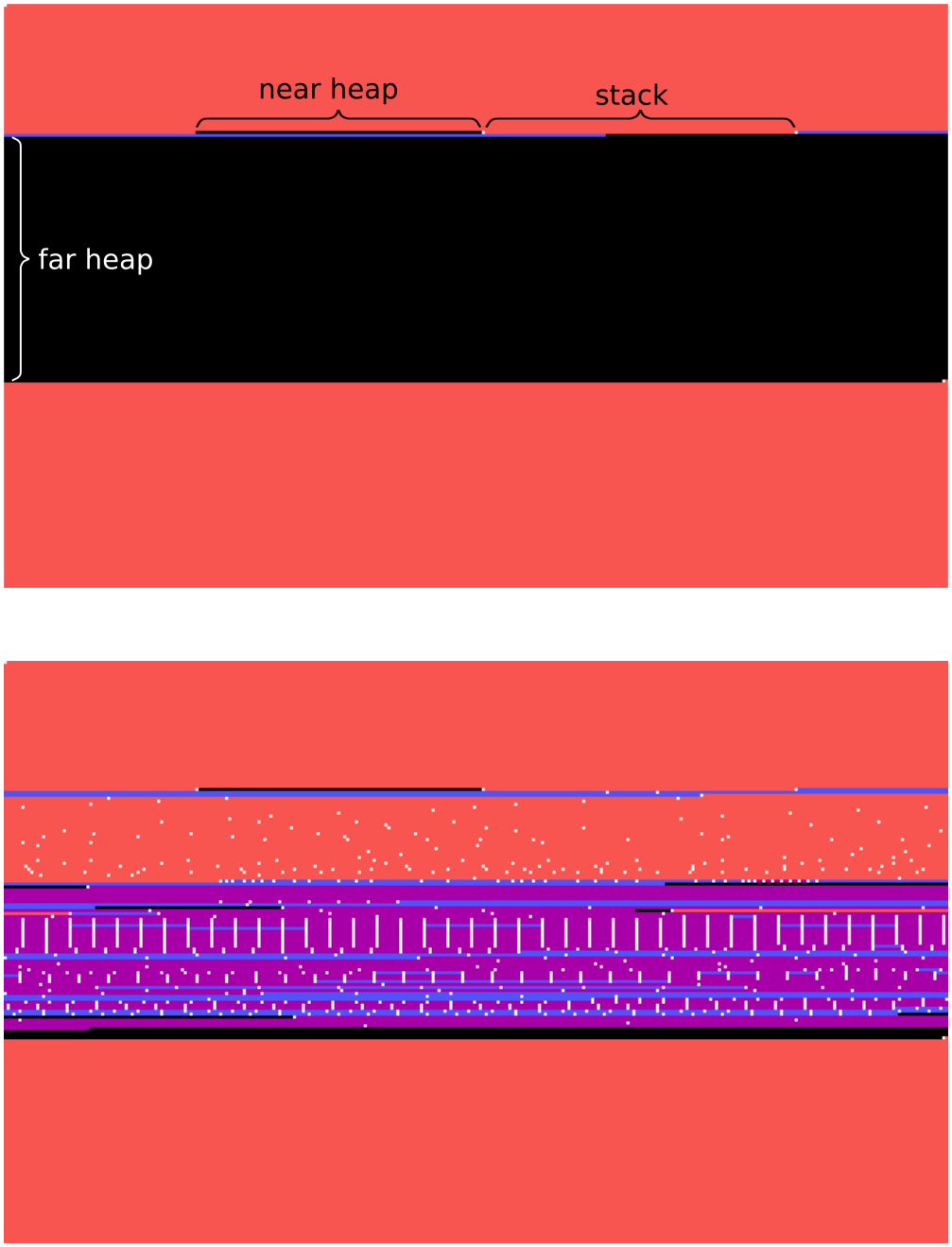


Figure 4.9: Memory dump after (i) initializing memory manager and (ii) running the game.

4.6.2 Video Manager (VW & RF)

The video manager features two parts:

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

The video manager is described extensively in section "Smooth scrolling on EGA" on page 114.

4.6.3 Cache Manager (CA)

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

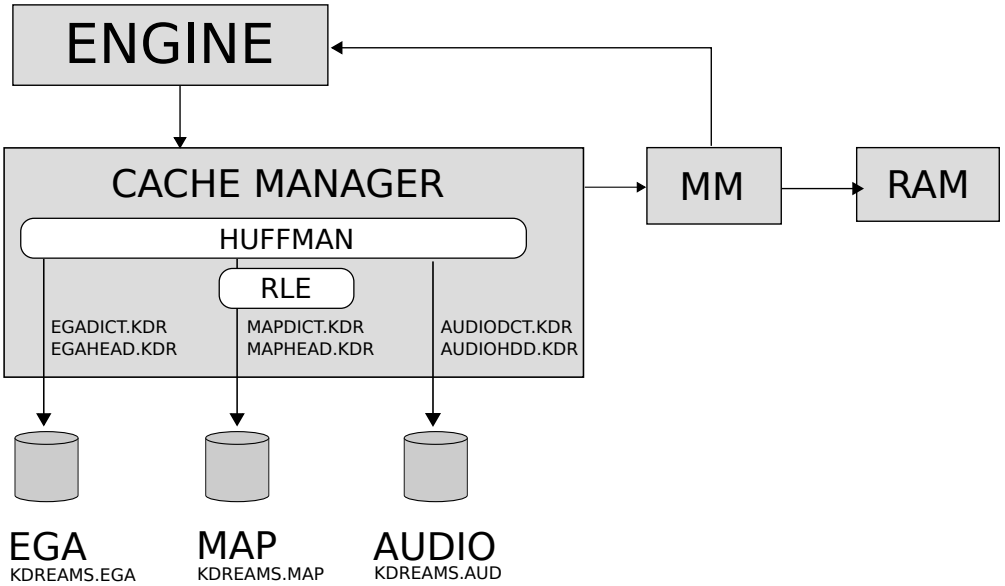
- A header file containing the offset to allow translation from asset ID to byte offset in the data file.
- A compression dictionary to decompress each asset.
- The data file containing the assets

Details of each asset file are explained in section "Programming" on page 61. The header and dictionary files are provided in the `static` folder with the `*.KDR` extension. These file types are integrated into the engine code and are required during compilation (converted into an `*.OBJ` file using `makeobj.c`).

> “ When I was trying to fit all the data for Keen 2 and Keen 3 on a 360K floppy and all the files wouldn't fit, I had to convert a bunch of files to .OBJ files (using a utility I developed), change the code to forgo the loading process for those files, then I had to link them into the KEEN EXE file, then finally I had to LZEXE the EXE file so it was much smaller
>
> – that was the only way that Keen 2 and 3 would fit on a 360K floppy.
>
> **John Romero**[6] ”

[6] Great article regarding history of Commander Keen on https://legacy.3drealms.com/keenhistory.

The data file containing the assets is not part of the source code and must be obtained by downloading the shareware version. All resources are compressed using Huffman compression, and maps have additional RLE compression. The cache manager is described extensively in the "Asset Caching and Compression" section.



4.6.4 User Manager (US)

The user manager is responsible for the layout of text and control panels, such as loading/saving games, configuring controls, and setting sound devices. Once we start the game, we move the display to EGA graphic mode `0x0D`. Here we cannot print characters on the screen using the `printf()` command.

A key function of the user manager is to render text at a specific pixel location. When the engine needs to draw a string, it is passed to `US_Print` which does all measurement (`VW_MeasurePropString`) and then passes this information to `VWB_DrawPropString`, which takes care of drawing the string on screen.

In the graphics asset file, each font character is stored with:

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

Each character is 10 pixels tall, but the width varies, as illustrated in Figure 5.10.

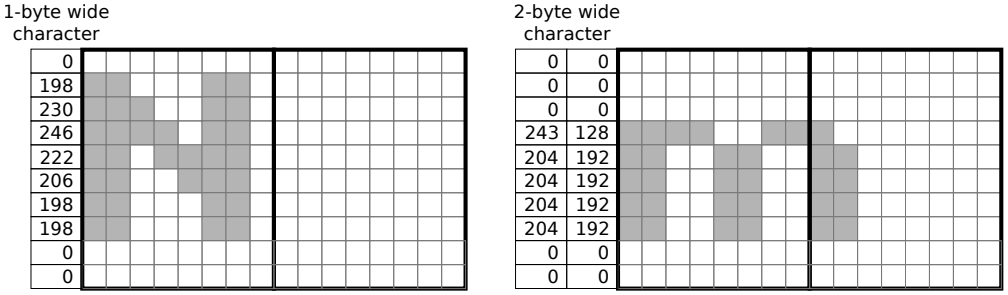


Figure 4.10: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

On the EGA videocard each 8 pixels take up 1 byte of VRAM (as explained in section "EGA Planar Madness" on page 39). When you need to print characters that aren't perfectly aligned with this 8-pixel grid, how do you manage the alignment? Let's say you want to display the letter 'N' on the screen. If the 'N' starts in the middle of a byte (say at pixel 3 instead of pixel 0), you need a clever way to shift the bits over to ensure it appears correctly on the screen.

Instead of manually shifting every pixel in your character bitmaps, the game engine uses pre-calculated bitshift tables. These tables are essentially lookup guides that help quickly adjust how characters are drawn based on their starting position. Here's how the process works:

1. Start with a base table (called `shiftdata0`) that holds all the possible values for a byte, from 0 to 255, and store this in an integer (16 bits).

2. Next, you generate seven more tables by shifting each value in `shiftdata0` one bit to the right for each table. So for `shiftdata1` the value 198 becomes 99, and for `shiftdata2` it becomes 32817, and so on. This process continues until you have eight tables, each representing a different bitshift.

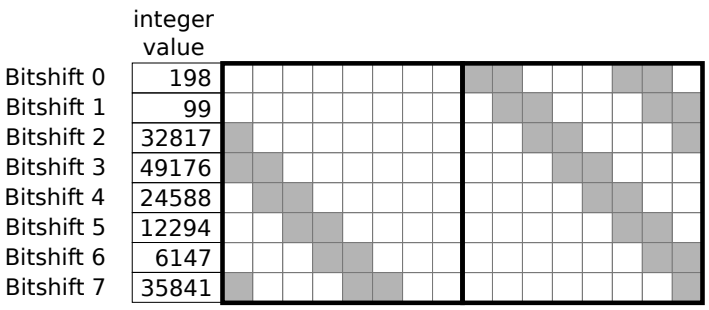


Figure 4.11: Right bitshift [0-7] for integer 198.

The pre-calculated bitshift tables are stored in `id_vw_a.asm`, below the `bitshift3` table.

```
LABEL shiftdata3 WORD
  dw     0, 8192,16384,24576,32768,40960,49152,57344,    1, 8193,16385,24577,32769,40961
  dw 49153,57345,    2, 8194,16386,24578,32770,40962,49154,57346,    3, 8195,16387,24579
  dw 32771,40963,49155,57347,    4, 8196,16388,24580,32772,40964,49156,57348,    5, 8197
  dw 16389,24581,32773,40965,49157,57349,    6, 8198,16390,24582,32774,40966,49158,57350
  dw     7, 8199,16391,24583,32775,40967,49159,57351,    8, 8200,16392,24584,32776,40968
  dw 49160,57352,    9, 8201,16393,24585,32777,40969,49161,57353,   10, 8202,16394,24586
  dw 32778,40970,49162,57354,   11, 8203,16395,24587,32779,40971,49163,57355,   12, 8204
  dw 16396,24588,32780,40972,49164,57356,   13, 8205,16397,24589,32781,40973,49165,57357
  dw    14, 8206,16398,24590,32782,40974,49166,57358,   15, 8207,16399,24591,32783,40975
  dw 49167,57359,   16, 8208,16400,24592,32784,40976,49168,57360,   17, 8209,16401,24593
  dw 32785,40977,49169,57361,   18, 8210,16402,24594,32786,40978,49170,57362,   19, 8211
  dw 16403,24595,32787,40979,49171,57363,   20, 8212,16404,24596,32788,40980,49172,57364
  dw    21, 8213,16405,24597,32789,40981,49173,57365,   22, 8214,16406,24598,32790,40982
  dw 49174,57366,   23, 8215,16407,24599,32791,40983,49175,57367,   24, 8216,16408,24600
  dw 32792,40984,49176,57368,   25, 8217,16409,24601,32793,40985,49177,57369,   26, 8218
  dw 16410,24602,32794,40986,49178,57370,   27, 8219,16411,24603,32795,40987,49179,57371
  dw    28, 8220,16412,24604,32796,40988,49180,57372,   29, 8221,16413,24605,32797,40989
  dw 49181,57373,   30, 8222,16414,24606,32798,40990,49182,57374,   31, 8223,16415,24607
  dw 32799,40991,49183,57375
```

Printing the letter "N" with an offset of 3 pixels involves a simple lookup in `shiftdata3`, resulting in a 3-bit shifted "N" on the screen. By sacrificing a bit of memory, a lot of CPU time is saved.

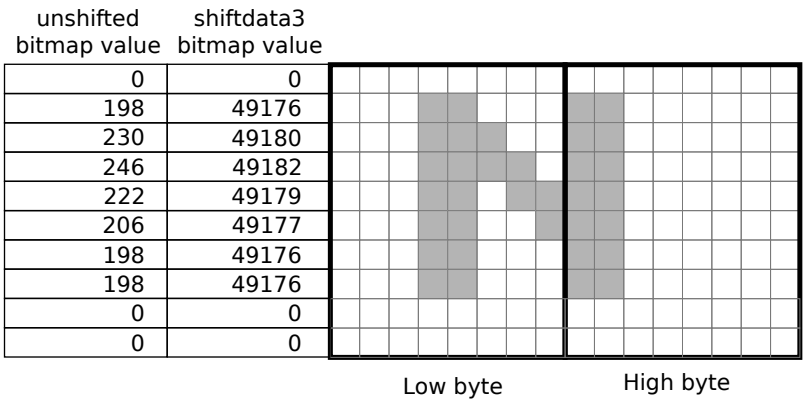


Figure 4.12: Bitshift 'N' over 3 bits using bit shift tables.

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

PROC  ShiftPropChar NEAR

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

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

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

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

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

4.6.5 Sound Manager (SD)

The Sound Manager abstracts interaction with all sound systems supported: PC Speaker, AdLib, and Sound Blaster. 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). The sound manager is described extensively in the "Audio and Heartbeat" section on page 160.

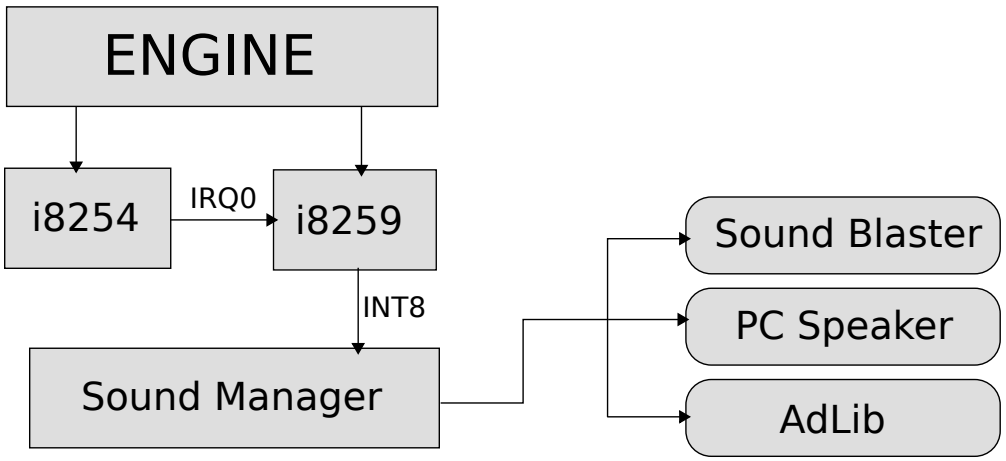


Figure 4.13: Sound system architecture.

4.6.6 Input Manager (IN)

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

4.6.7 Softdisk files

The primary function of the Softdisk files is to load and display the intro screen bitmap using the `LoadLIBShape` function from soft.c. Most of the functions in these files are not used and are therefore not discussed further in this book.

4.7 Startup

When the game engine starts, it first loads the Memory Manager. It then checks if at least 335KiB of RAM is available. If not, a warning is displayed, but the game can still continue. However, the game will likely crash or display an "Out of memory" error soon after.

Once the game has successfully started, the intro image, a Deluxe Paint bitmap image (`*.LBM`), is displayed. After the user presses any key, the intro image (using 320x200 = 64,000 bytes) is unloaded from RAM to free up memory for runtime, and the control panel is displayed.

Figure 4.14: Keen Dreams intro screen

4.8 Asset Caching and Compression

The floppy disk is not only the slowest component of the PC but also constrained in terms of storage space. Therefore, it is crucial to load and store game assets as efficiently as possible in memory, avoiding long and unnecessary loading times from the disk. To make matters worse, the total amount of assets and maps cannot fit into RAM all at once. This is where a cache manager becomes essential. Its primary purpose is to increase data retrieval performance by reducing the need to load data from the slow floppy disk.

4.8.1 Asset caching

To manage and track the assets to be loaded into memory, the game engine uses a caching level mechanism. An array, sized according to the total number of graphical assets, is maintained to mark whether an asset needs to be loaded into memory.

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

The index of this array corresponds to the graphic asset IDs. Using eight bits, the array can manage the required assets for different cache levels. The `ca_levelnum` variable points to the current cache level. Upon executing the engine, the cache manager starts with an empty array and `ca_levelnum` set to 1.

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 4.15: Initiating `grneeded[]` array, current cache level is 1.

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

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

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

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

The function `CA_CacheMarks` then iterates through the cache array, checking if any of the required assets are not yet available in memory. If not, it loads and decompresses the asset from disk 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								
...								

Figure 4.16: Mark and load assets required for the new map in cache level 1.

Now, during gameplay, the user opens the control panel (e.g. to pause the game), which requires to load assets for the control panel into memory. The cache level is increased to level two and for all required control panel assets the second bit is marked.

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

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

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

The `grneeded[]` cache array then appears as below.

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 4.17: Mark all assets required for the control panel in cache level 2.

The function `CA_CacheMarks()` is called again to load all cache level 2 assets into memory. It iterates over all assets, loading any missing ones into memory. Any asset that is not required for cache level 2, but is already in memory will be marked for purging, meaning the memory manager can remove it from memory in case of insufficient memory.

In this example, this applies to

- KEENSTANDRSPR
- KEENRUNR1SPR
- TILE16 #1

```
void CA_CacheMarks (char *title, boolean cachedownlevel)
{
  numcache = 0;
    //
    // go through and make everything not needed purgable
    //
  for (i=0;i<NUMCHUNKS;i++)
    if (grneeded[i]&ca_levelbit)
    {
      if (grsegs[i])          // its allready in memory,
   make
        MM_SetPurge(&grsegs[i],0);  // sure it stays there!
      else
        numcache++;
    }
    else
    {
      if (grsegs[i])          // not needed, so make it
   purgeable
        MM_SetPurge(&grsegs[i],3);
    }

  if (!numcache)      // nothing to cache!
    return;
  ...
}
```

When the user closes the control panel, the engine lowers the cache level back to 1, where all assets required for playing the level are memorized. Simply calling the function `CA_CacheMarks()` again reloads any assets that were 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 frequently used assets, such as fonts and Commander Keen sprites, being reloaded every time from the disk, they are loaded permanently into memory by flagging them as non-movable, unpurgeable blocks 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);
```

4.8.2 Asset compression

Given that the floppy disk is limited in both speed (100-250 kbps[7]) and storage capacity (3½-inch disk size is either 720KB or 1.44MB), file compression was essential. Compression ensures that the game occupies less space and loads more quickly. id Software used "Huffman compression" for all asset and map files, with additional "RLE compression" applied to further reduce the size of 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
```

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

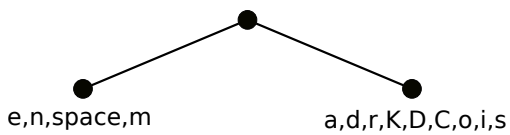
The first step is to make a character frequency table.

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

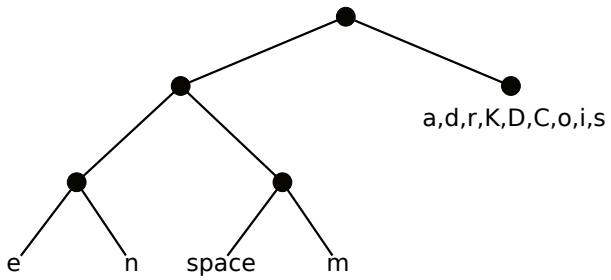
Character	Frequency
d	1
D	1
C	1
o	1
i	1
s	1

Figure 4.18: Character frequency table.

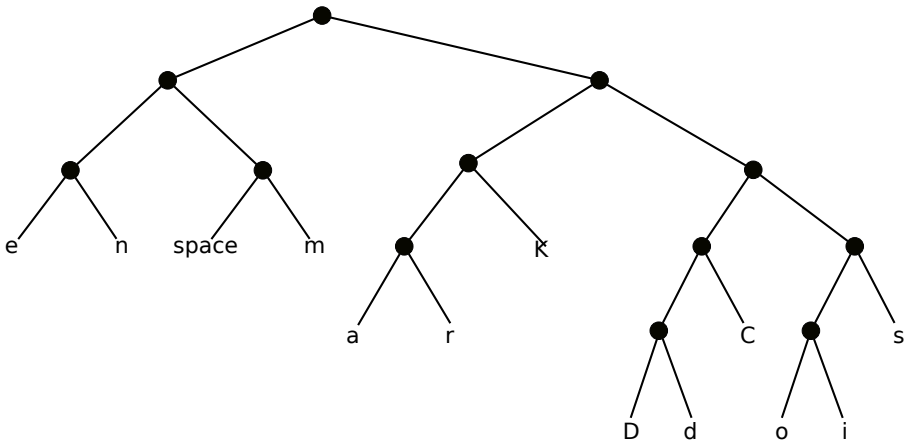
Next, an optimal binary tree, also known as a dictionary, is created. This is done by starting with the most common character and checking if it occurs more frequently than all the other characters combined. If not, then the second most common character is added, and so on. In our example, four characters 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.



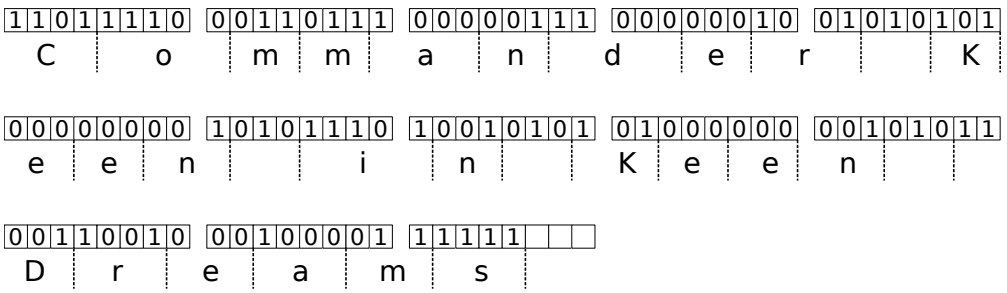
The process continues by creating a new node with left and right branches and repeating the steps. The two most common characters in the left node, 'e' and 'n', are assigned to the left branch of this new node. A third node is created, and since there are only two options, each is given a branch. This process is applied to the remaining characters of the left node, resulting in the structure shown below.



The same procedure is applied to the right node, where 'a', 'd', 'r', and 'K' constitute 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, the entire text can be encoded by moving left (bit 0) or right (bit 1) through 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 results in a total of 13 bytes used for a 31-byte message, more than a 50% reduction. In Commander Keen, the dictionary is part of the game engine. Therefore, it is important that the asset files from the shareware version correspond with the dictionary files in the source code[8]. Reading the Huffman-compressed asset file is straightforward: you read bit by bit from the file and follow the dictionary, starting from the top node, until you reach an end node. The engine then writes the byte to memory and returns to the top node in the dictionary for the next bit in the file.

[8]That's why a specific source code version is mentioned in section "Getting the Source Code" on page 83.

```
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 technique, 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, *Run-length encoding* (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 'tag'. This special tag value instructs the program to take specific action upon encountering it. Every value is passed through unchanged until an RLE tag value is encountered. When this tag 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, and therefore is named RLEW, where the 'W' stands for word. The RLEW tag is defined as `ABCDh`.

The overall result of applying Huffman and RLEW compression results in almost 65% file size reduction.

Asset type	**filename**	**Uncompressed size**[9]	**Compressed size**
Graphic assets	KDREAMS.EGA	354,075 bytes	213,045 bytes
Game levels	KDREAMS.MAP	417,714 bytes	65,673 bytes
Sound assets	KDREAMS.AUD	4,572 bytes	3,498 bytes
Total		776,361 bytes	282,216 bytes

[9] See Appendix A for details per asset file.

```
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: ABCDh
  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
  [...]
}
```

4.9 Smooth scrolling on EGA

Performing a full-screen redraw per frame would kill the CPU, as it would require updating every pixel on all four EGA planes. Achieving a 60Hz frame rate while refreshing the entire screen is impossible under these constraints. 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;
  }
}
```

> Clearly, there is a whole world of awesome things there that we just couldn't do on the PC...
>
> You can just redraw the whole screen, but then it turns out...
>
> Well, you're going five frames a second.
>
> **John Carmack**[10]

[10] Interview with Lex Fridman in 2022, this quote is around 1h:41m.

So, how did they create a smooth scrolling game with these limitations? The solution was twofold:

- Utilizing specific EGA hardware tricks
- Updating only the portion of the screen that has changed

Each method is described in detail in the following sections.

4.9.1 EGA Virtual Screen and Pel Panning

The EGA card adds a powerful approach to linear addressing: the logical width of the virtual screen in VRAM does not need to match the physical screen display width. A programmer can define a virtual screen width of up to 4096 pixels and use the physical screen as a window onto any part of the virtual screen. What's more, the virtual screen's logical height can extend up to the total VRAM capacity. The code below demonstrates how to set a custom 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 displayed area of the virtual screen at any time is determined by setting the display memory address for fetching video data, configured via the CRTC Start Address register. The default address is `A000:0000h`, though the offset can be adjusted to any other address.

```
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
```

To pan down a scan line, it requires only to increase the CRTC start address by the logical width. Horizontal panning is achieved by incrementing the start address by one byte. In EGA's planar graphics modes, the eight bits in each video RAM byte correspond to eight consecutive on-screen pixels, meaning the screen can move horizontally in steps of 8 pixels. This is rather coarse and not really smooth horizontal scrolling. Fortunately, another EGA register addresses this limitation.

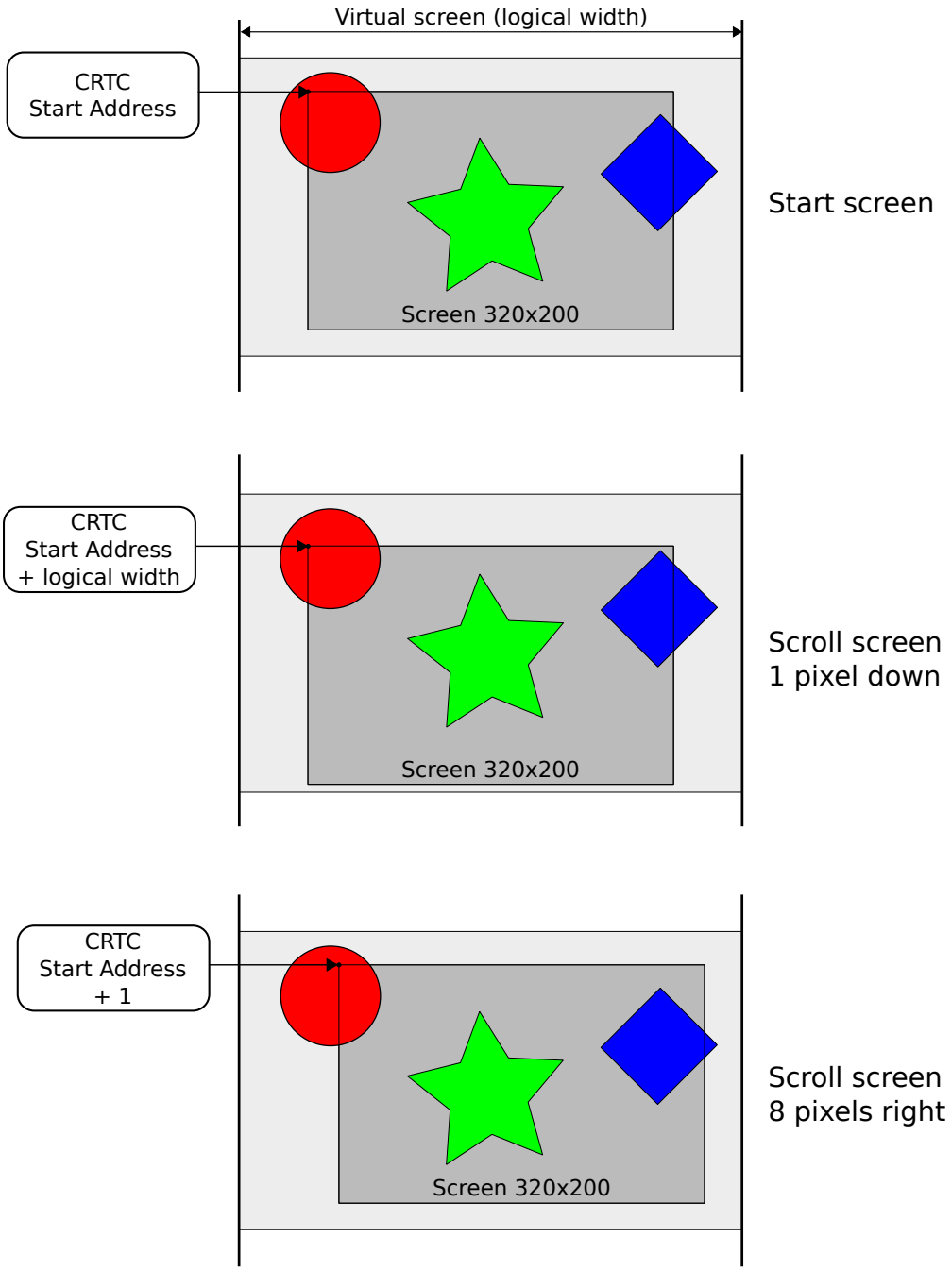


Figure 4.19: EGA screen scrolling by updating the CRTC Start Address.

4.9.2 Horizontal Pel Panning

Smooth horizontal pixel scrolling is managed by the "Horizontal Pel Panning" register in the Attribute Controller (ATC), allowing fine adjustment in 1-pixel increments up to 7 pixels to the left.

However, programming the ATC requires attention: the ATC Index register only uses the lower five bits (bits 0-4) as its 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 maintain normal video operation, bit 5 must be set 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 involves adjusting the CRTC Start Address alongside fine-tuning within an 8-pixel range using horizontal pel panning. The following steps accomplish this:

- Calculate the panning offset in pixels for both `x`- and `y`-direction.
- Smooth y-panning is achieved by adding the `(logical width * y)` to the CRTC start address.
- For coarse 8-pixel horizontal scrolling, increase the CRTC start address by `x /8` bytes.
- Fine horizontal pixel adjustment is applied using `(x % 8)` to the horizontal pel panning register.

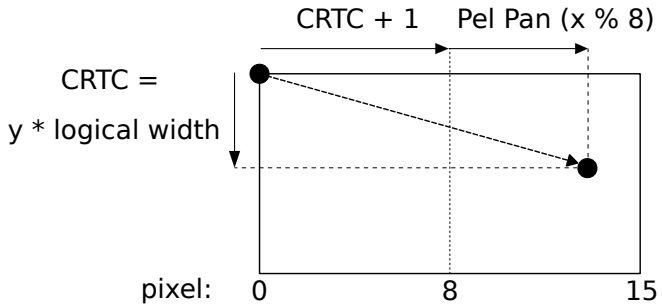


Figure 4.20: Smooth scrolling in EGA.

4.10 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. However, once the virtual screen edge is reached, a full-screen redraw would be too slow, causing a drop to 5 fps.

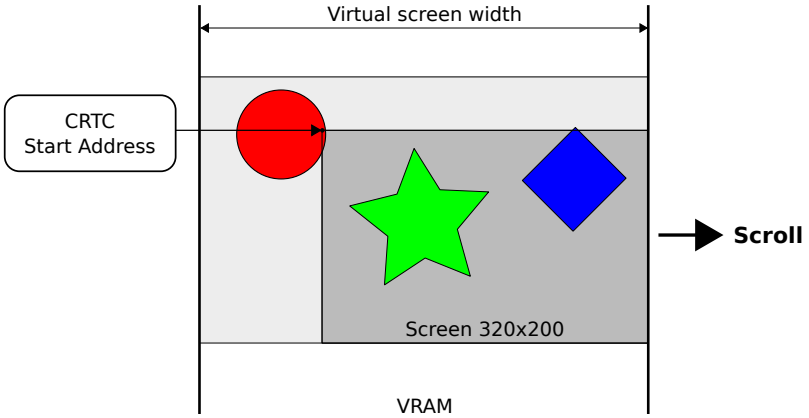


Figure 4.21: Screen reaching the edge of virtual screen, full refresh required.

A naive approach for screen refresh is to load the entire level into VRAM, setting the logical width to match the level width. By adjusting the CRTC Start Address and Horizontal Pel Panning registers, smooth scrolling can be achieved. However, loading the entire level requires far more VRAM than is available. The first level, *Horsh Radish Hill*, is 136 tiles wide and 37 tiles high. Each tile is 128 bytes, which means a total of 136 x 37 x 128 bytes = 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.

To solve the refresh issue, John Carmack invented "Adaptive Tile Refresh" (ATR). The core idea is to refresh only those areas of the screen that need to change.

Let's look at *Commander Keen 1: Marooned on Mars* in Figure 5.22. 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!

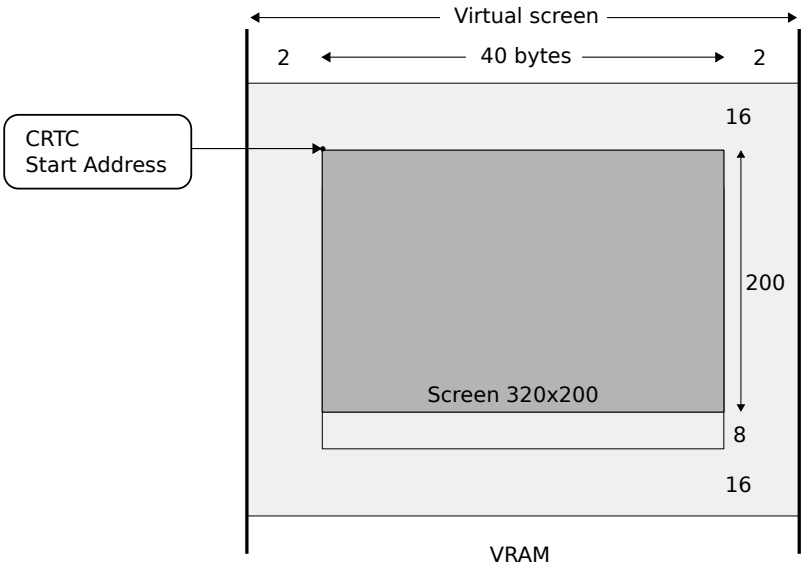
This is also the part where the game designers Tom Hall and Jogn Romero had to help the engine. Smooth scrolling is inversely proportional to the number of tiles to redraw. A checkerboard tile pattern basically means a full-screen redraw and would kill the CPU. So to avoid costly "jolts", the game designers built tile maps with a lot of reperating tiles.

Figure 4.22: Start of the world, moved one tile to the right and difference.

4.10.1 Adaptive tile refreshment in Commander Keen 1-3

This section explains how ATR is implemented for the first three episodes of Commander Keen. Since there is no source code released for these episodes, ATR will be explained 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[11].

The EGA screen in mode `0Dh` has a resolution of 320x200 pixels, or 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. By making the virtual screen one tile higher and one wider 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.

[11] 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. So within a 256KB EGA card there is enough VRAM available to keep all three virtual screens in memory. The page that is displayed on screen is selected by setting the CRTC Start Address register at which to begin fetching video data.

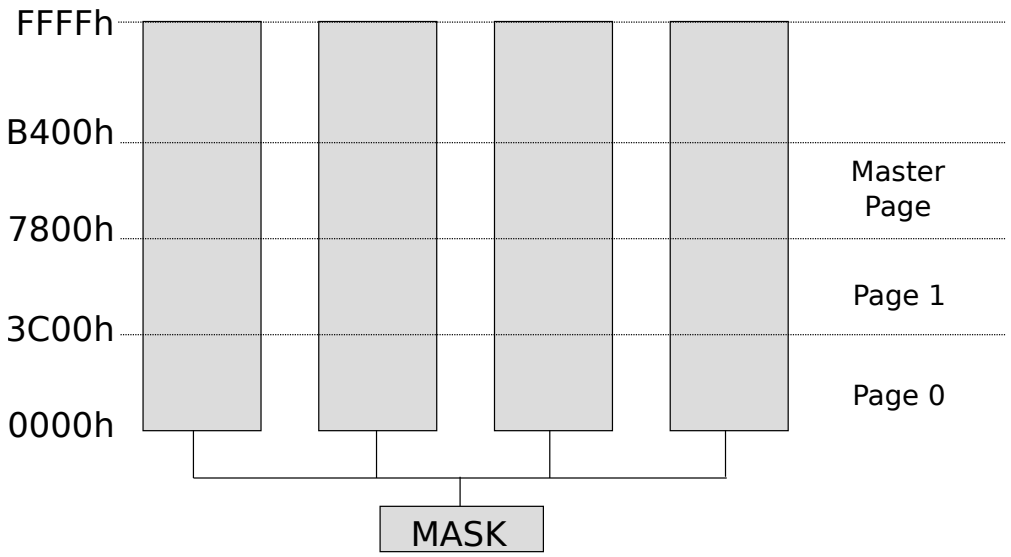


Figure 4.23: Virtual screen layout on EGA card.

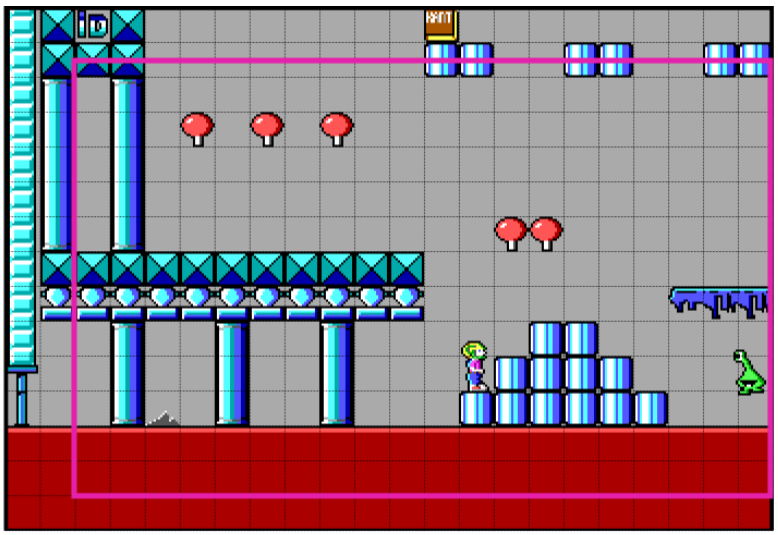
For both the buffer and visible screen a tile array is created to maintain which tiles are changed since last refresh.

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

Steps to refresh the screen are as follows:

1. Check if the player has moved one tile in any direction.
2. Identify changed tiles and copy these to the master page, marking them in both tile arrays.
3. Refresh the buffer page by copying marked tiles from the master page.
4. Switch the view and buffer pages by adjusting the CRTC Start Address and Pel Panning registers.

Ignoring sprites for now, the following steps illustrate these stages. 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.



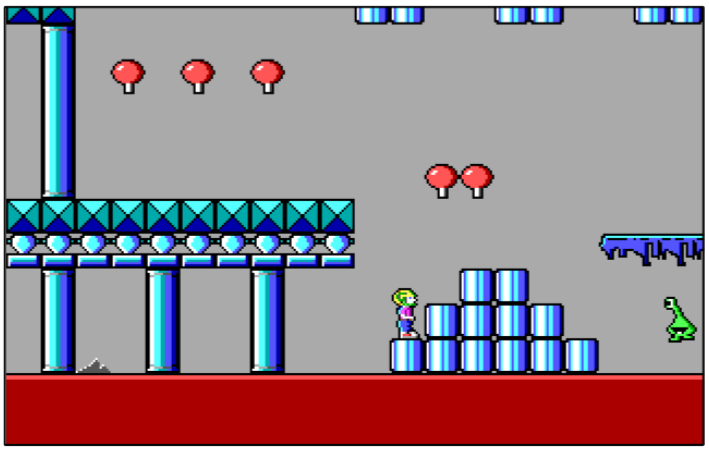


Figure 4.24: Start: Reach the end of the virtual screen.

The engine keeps track of which tile numbers are part of the virtual screen. Since the engine only refresh the screen 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, meaning it needs to be updated upon next refresh.

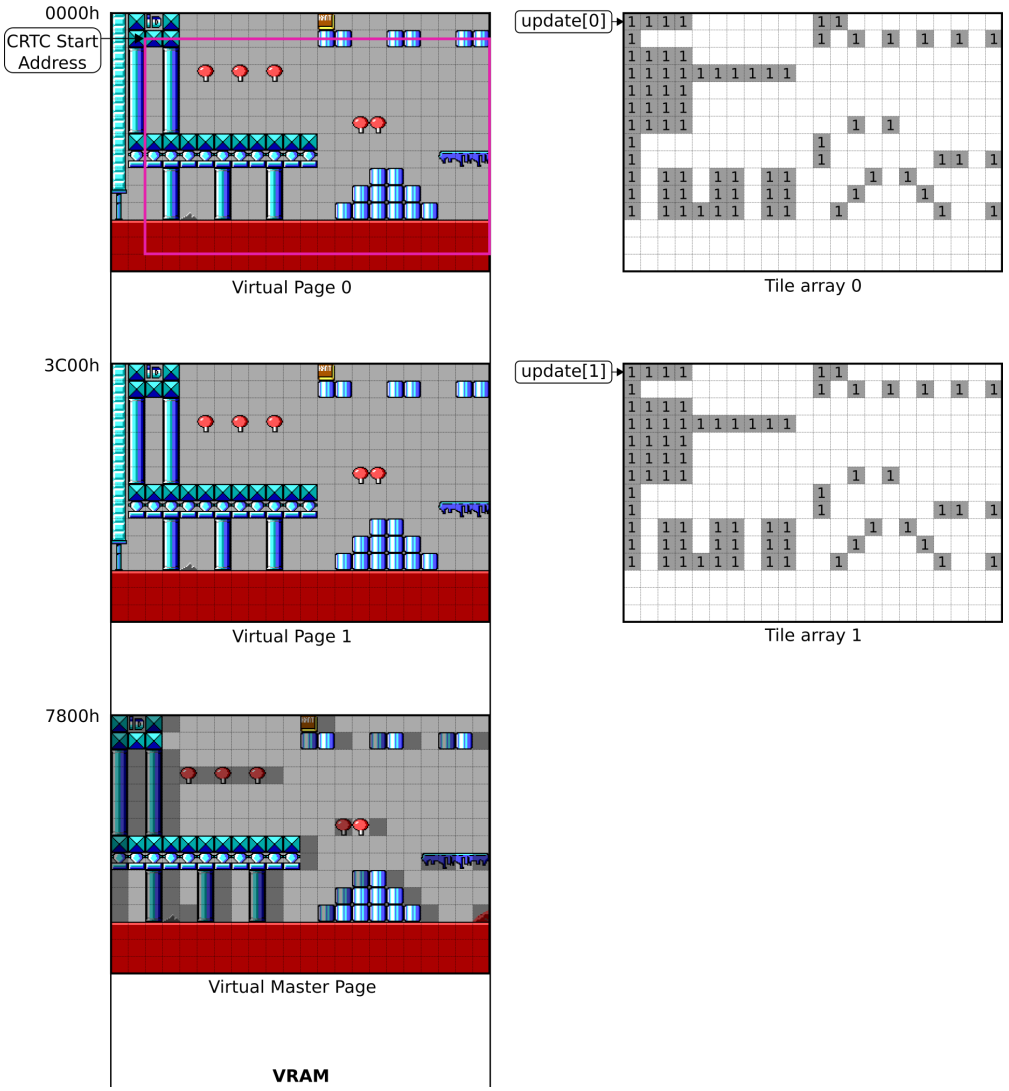


Figure 4.25: Update tiles in master page and update both tile arrays.

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

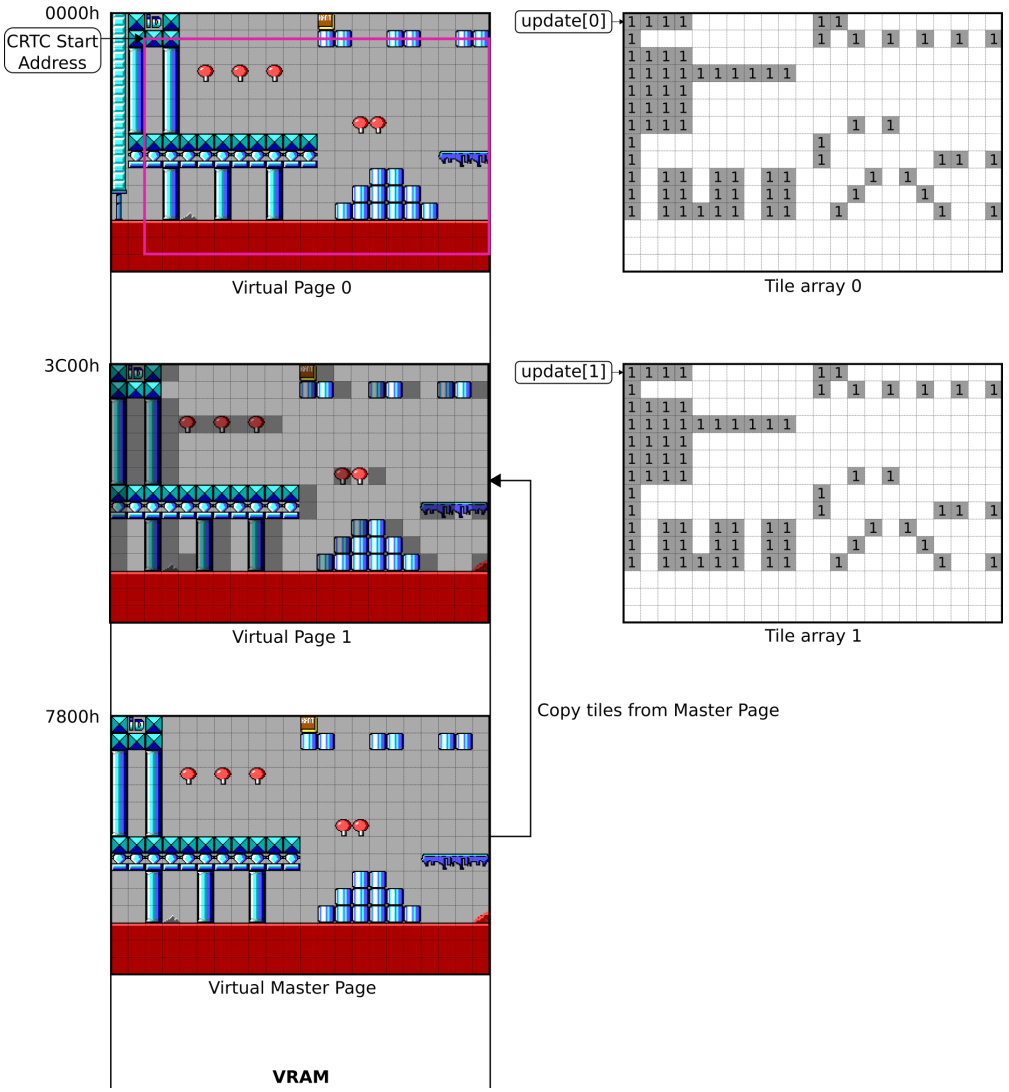


Figure 4.26: Copy changed tiles from virtual master page to page 1.

In the final 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.

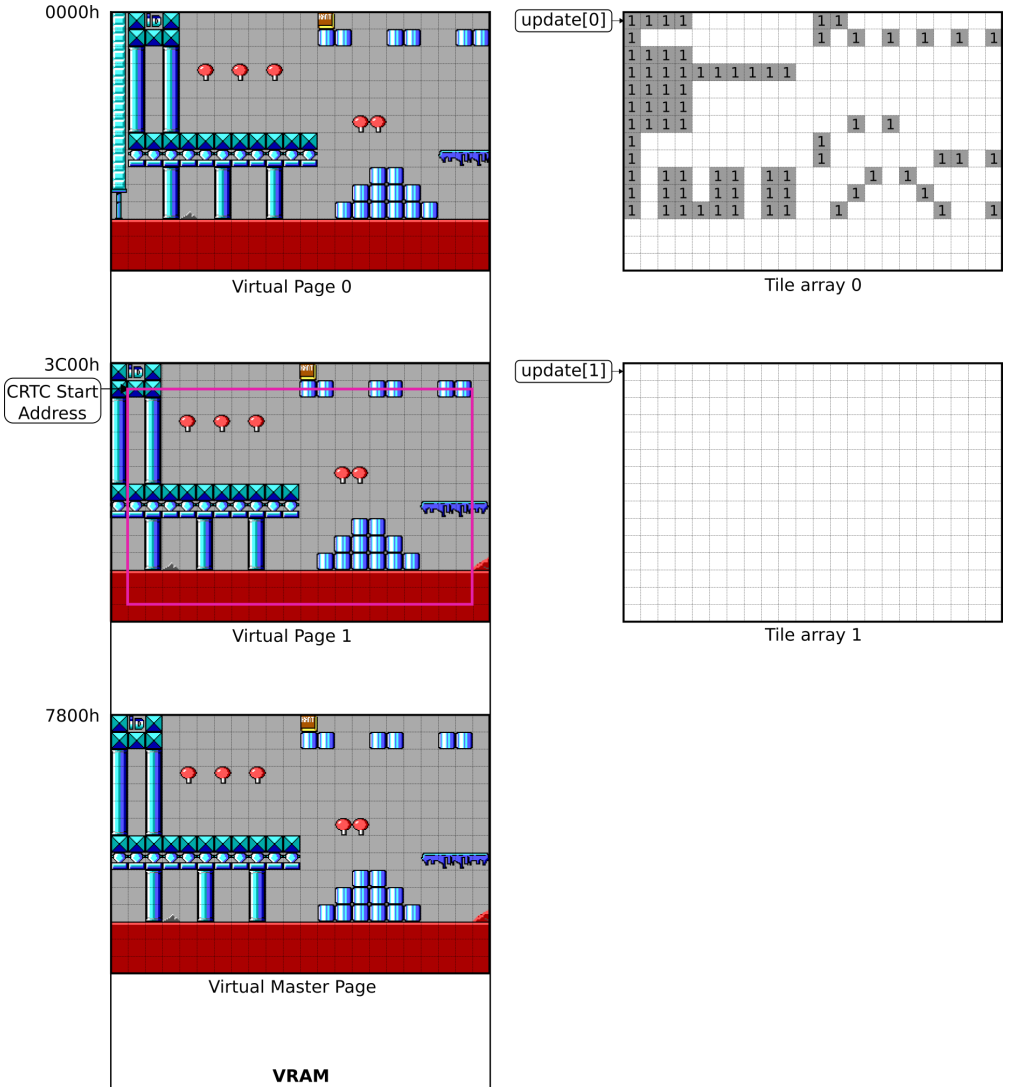


Figure 4.27: Update CRTC Start Address to virtual page 1 and empty tile array 1.

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

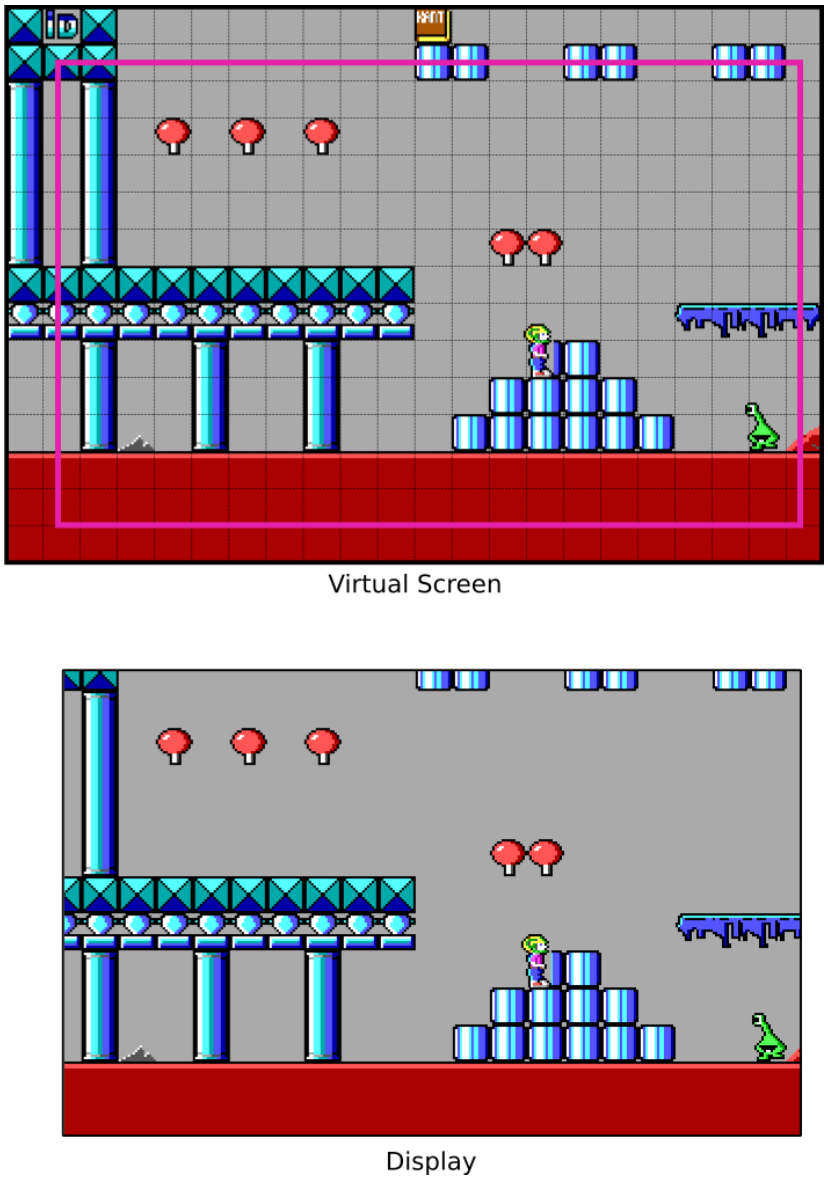


Figure 4.28: Screen is refreshed and scrolled to the right.

4.10.2 Wrap around the EGA Memory

John Carmack explored what would happen if you push the virtual screen over the 64kB border (address `FFFFh`) in video memory. It turned out that the EGA continues the virtual screen at `0000h`. This means you could wrap the virtual screen around the EGA memory and only need to add a stroke of tiles on one of the edges when Commander Keen moves more than 16 pixels.

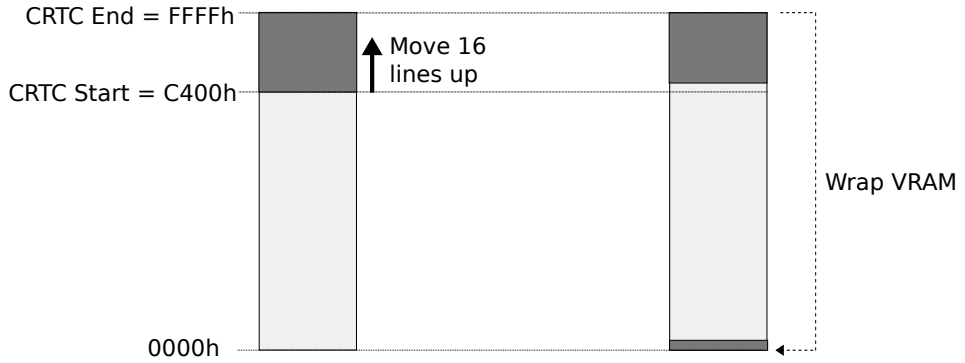


Figure 4.29: Wrap virtual screen around the EGA memory

> 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**[12]

[12] An explanation further elaborated during the same 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[13]. This resulted in crippled backwards compatibility and the wrapping around `FFFFh` did not work anymore on these cards.

There is an easy solution to resolve this issue. As you can see in Figure 5.23 on page 123, the space between `B400h` and `FFFFh` is not used and contains enough space for another virtual screen. In case the start address is between `C400h` and `FFFFh` the corresponding screen is copied to the opposite end of the buffer, as illustrated in Figure 5.30.

> “ 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[14]** ”

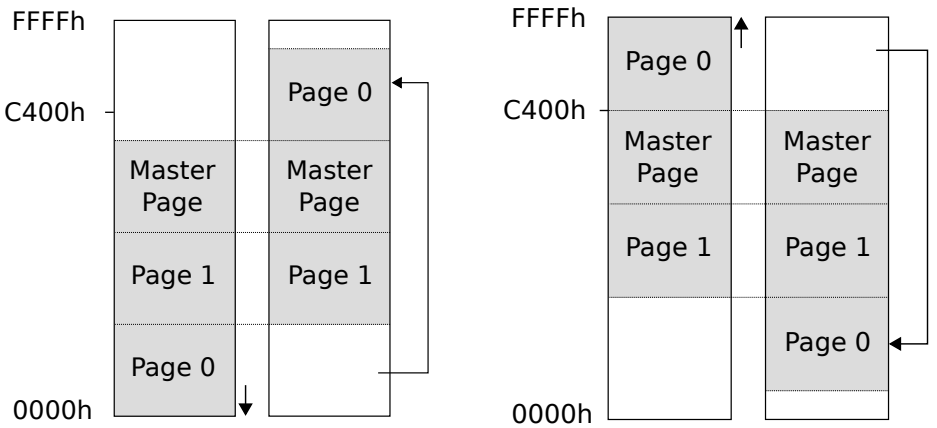


Figure 4.30: Move screen to opposite end of VRAM buffer

[13] 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.

[14] And again the 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);
  }
}
```

As explained in section "EGA Programming: Memory Mapping" on page 42 each pixel is encoded by four bits, which are spread across the four EGA banks. So how can we copy four VRAM planes fast enough, without noticing any performance hit? 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[15]. 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. Now it is possible to copy the entire buffer fast enough, without notifying any performance impact.

[15]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.

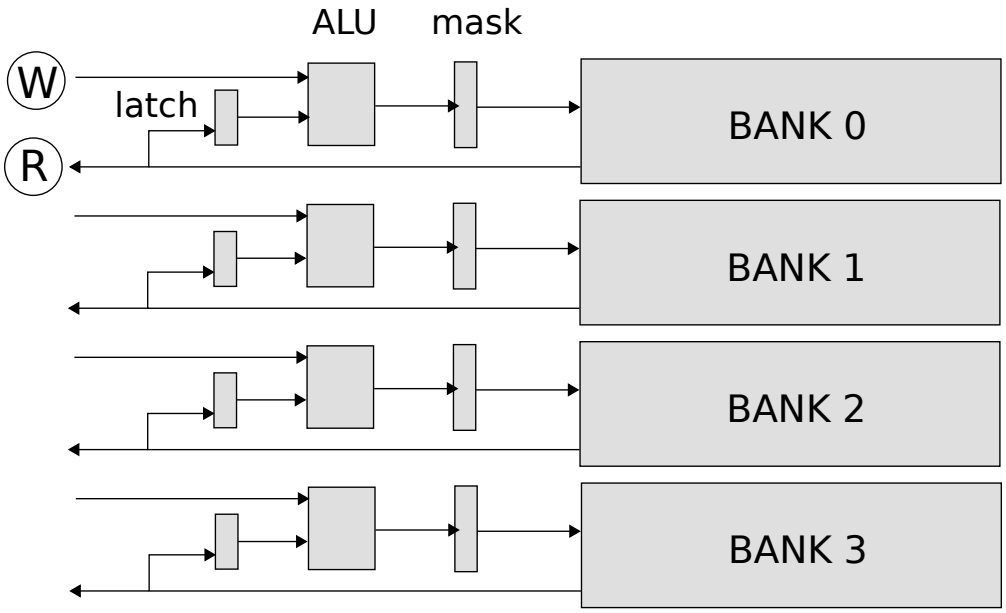


Figure 4.31: Latches memorize read operations from each bank. The memorized value can be used for later writes.

```
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
```

4.10.3 Adaptive tile refreshment in Keen Dreams

The EGA memory wrapping results in an improved, more simplified ATR algorithm. First, a virtual screen and tile array is defined by making the display one tile taller and wider, creating what is known as the viewport. This allows the engine to scroll the display up to 16 pixels to the right and bottom without refreshing tiles, by simply adjusting the CRTC Start Address and Pel Pan register.

An additional column is then added to the viewport to support horizontal scrolling. Finally, the tile array is further expanded to allow the entire viewport to move up to two tiles in any direction.

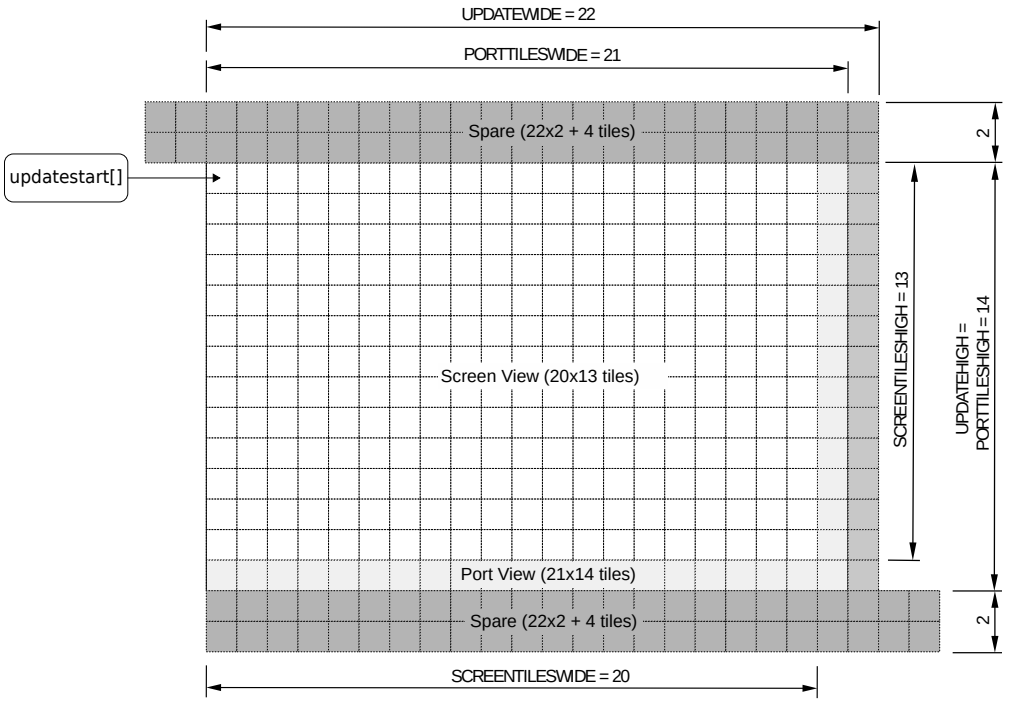


Figure 4.32: Tile array layout.

The full refresh cycle, including sprite updates, proceeds as follows:

1. Verify if the player has moved one tile in any direction.
2. Update both the tile array and VRAM pointers, copy the new column or row of tiles to the master page and mark the tiles in both arrays.

3. Refresh the buffer page by scanning the tile index array. If a tile is marked, copy it from the master page to the buffer page.
4. Iterate through the sprite removal list, copying the corresponding image block from the master page to the buffer page to clear the sprite.
5. Iterate through the sprite list, copying each sprite image block to the buffer page.
6. Switch the view and buffer pages 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);
}
```

> John's cool Adaptive Tile Refresh trick that made the scrolling even faster than redrawing the entire screen (this is not the Virtual Screen Tile Refresh trick he created for Keen 4-6).
>
> **John Romero**[16]

[16] Quotes from John Romero on https://legacy.3drealms.com/keenhistory/.

Let's go over the refresh cycle step by step. At the start, all three virtual pages display the same viewport, with virtual page 0 currently shown on-screen. Both tile arrays are empty, meaning no tile updates are required in the next refresh cycle. The player has reached the edge of the virtual screen and moves further to the left.

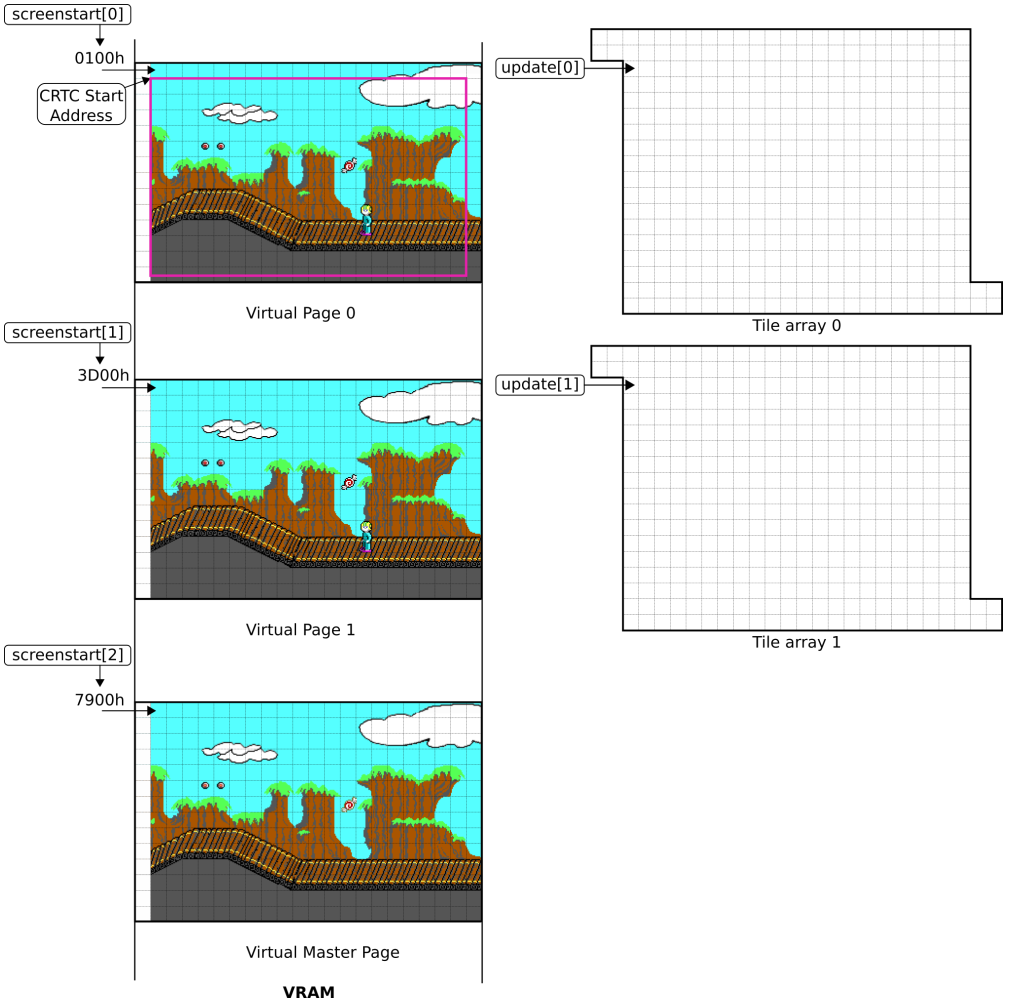


Figure 4.33: VRAM and tile arrays before scrolling to the left.

Both VRAM and tile array pointers shift left to introduce a new column of tiles. The VRAM pointer is lowered by 2 bytes (1 tile), and the tile array pointer is decreased by 1 byte. A new column of tiles is copied from RAM to the master page, while the leftmost column in both tile arrays is marked with '1' to ensure it updates in the next refresh cycle. Animated tiles are then copied to the master page, and the corresponding tiles in the array are also marked with '1'.

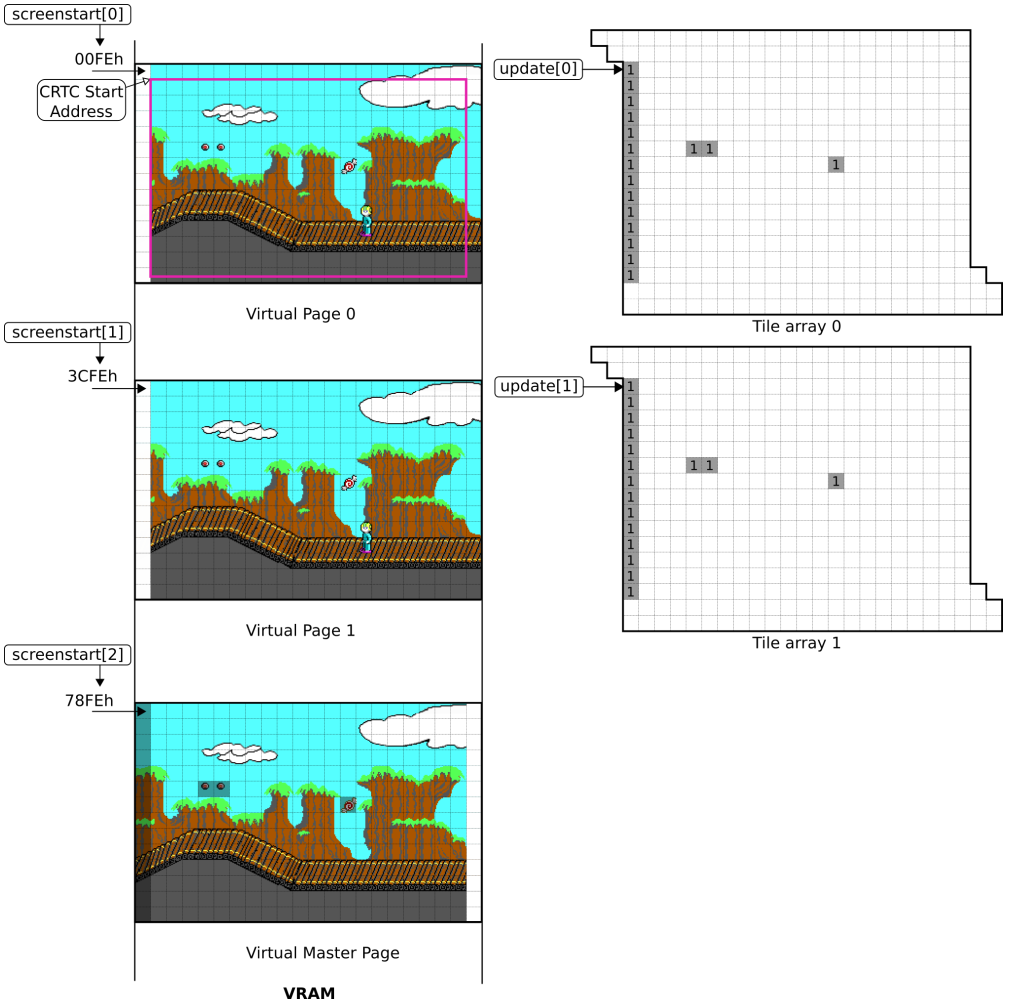


Figure 4.34: Update Virtual master with new left column and animated tiles.

Next, the engine scans all ’1’s and copies the corresponding tiles from the master to the buffer page.

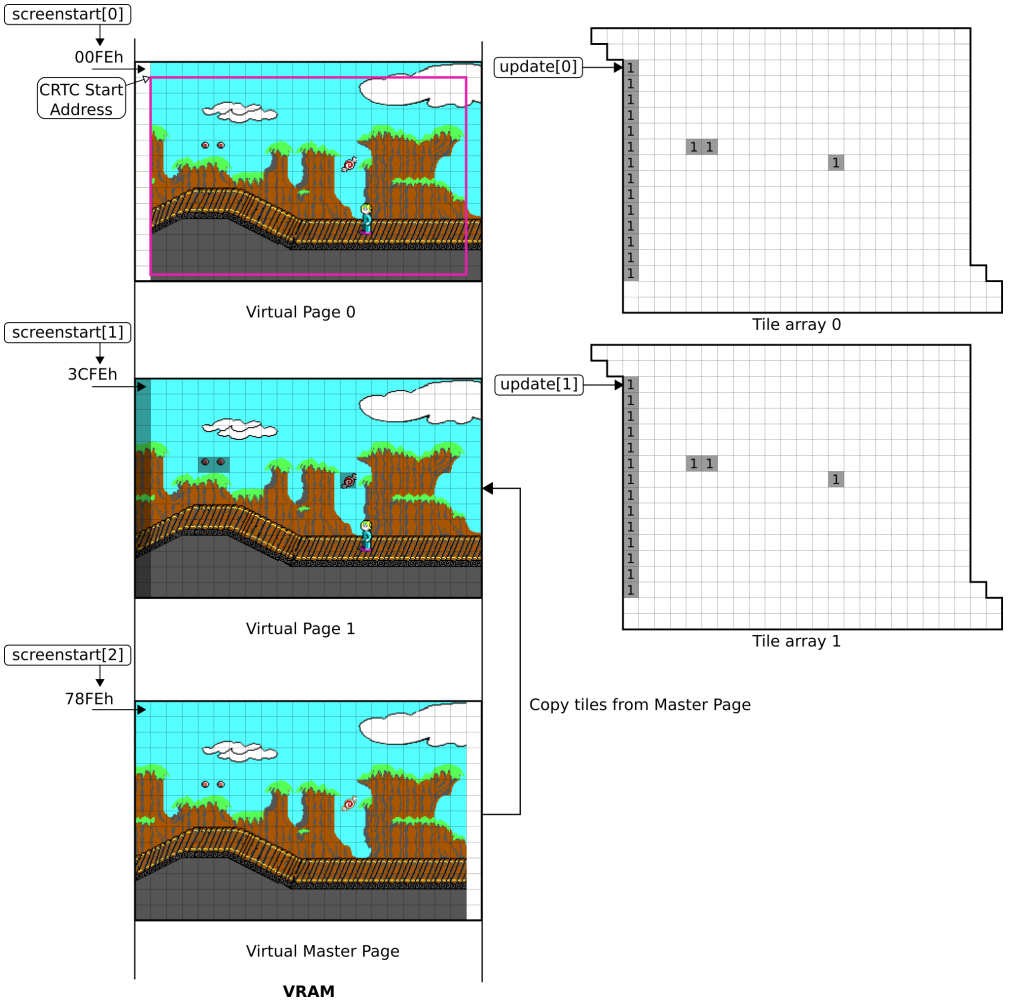


Figure 4.35: Copy changed tiles from master to buffer page.

Each sprite removed from the screen has its location and size stored in the sprite removal list, which removes the sprite by copying a specific section from the master screen to the virtual page. Tiles overlapping with the removal block are marked with a '2'.

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

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

//sprite removal list for Page 0 and Page 1
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++;
  }
}
```

The final step in updating the buffer is copying sprites to their new locations, with each corresponding tile marked with a '3' in the tile array.

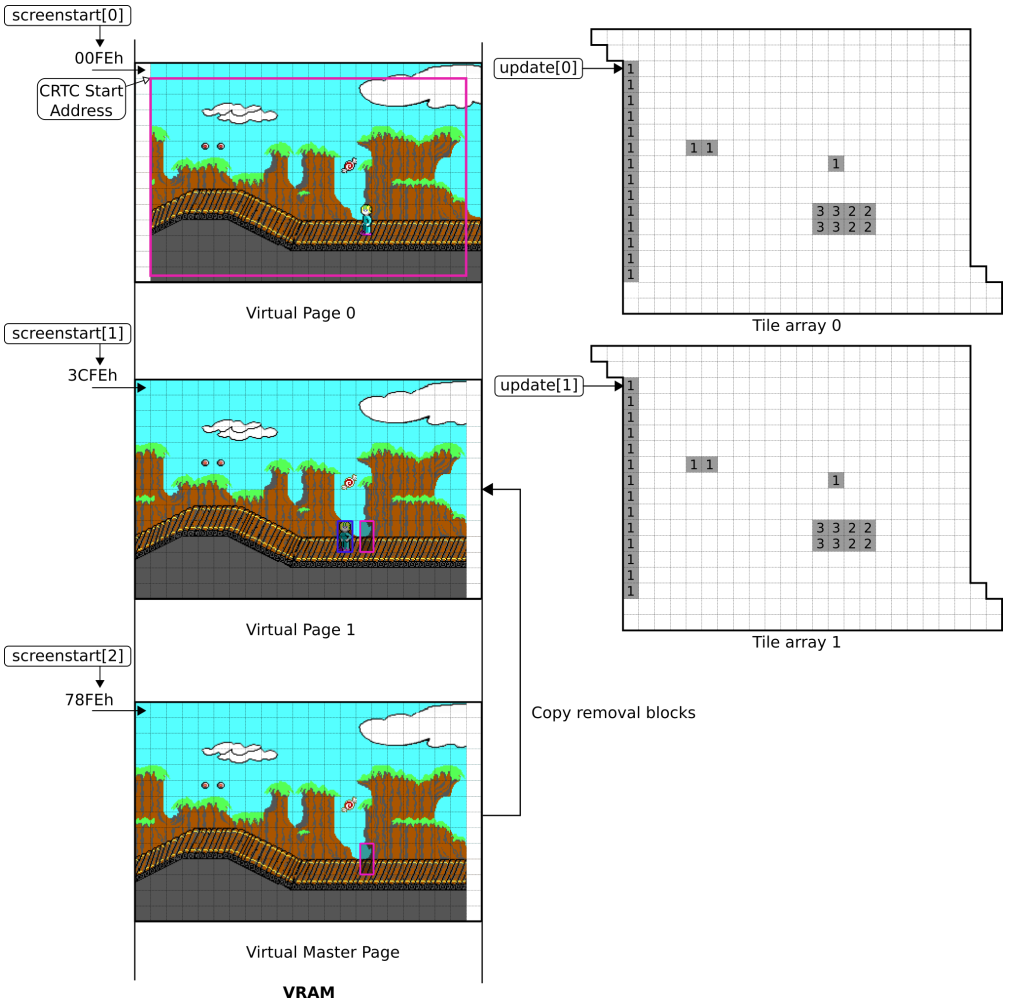


Figure 4.36: Removing and updating sprites to the buffer screen.

Once the buffer refresh is completed, the engine needs only to update the CRTC start address to virtual page 1. At this point, the visible tile array is emptied, and the pointer resets to its starting location. The new buffer array (virtual page 0) retains the marked tiles since this screen has not yet refreshed. Finally, the refresh cycle restarts, with virtual page 0 now functioning as the buffer.

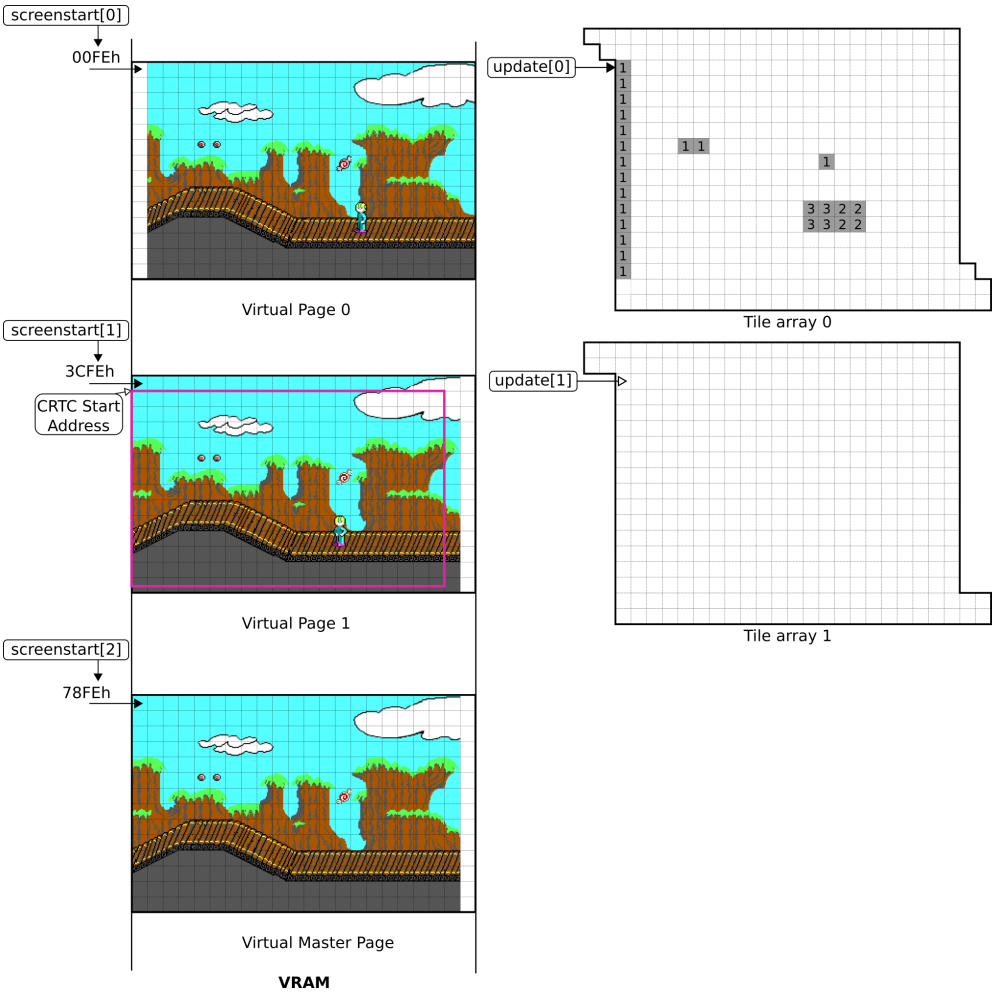


Figure 4.37: Swap buffer and visible screen by updating CRTC start address.

The final step in scrolling is to adjust the horizontal fine-pixel alignment by setting the Pel Pan register. *Et voilà*, the screen scrolls smoothly to the left. This scrolling process is significantly more efficient than the original ATR engine, resulting in only 8% of tiles to be refreshed!

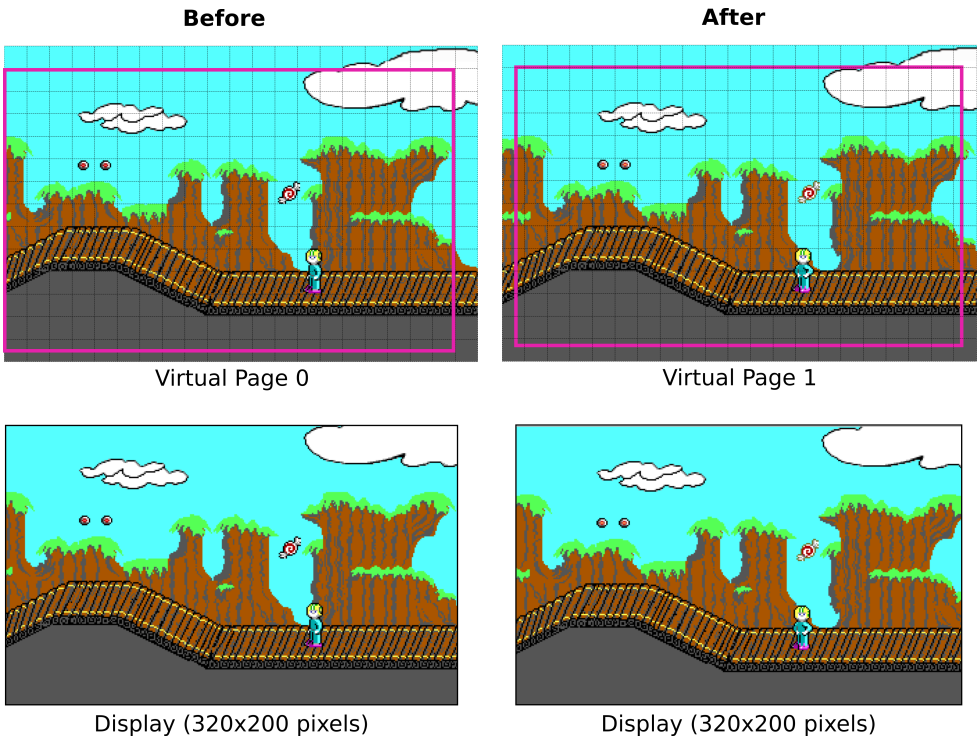
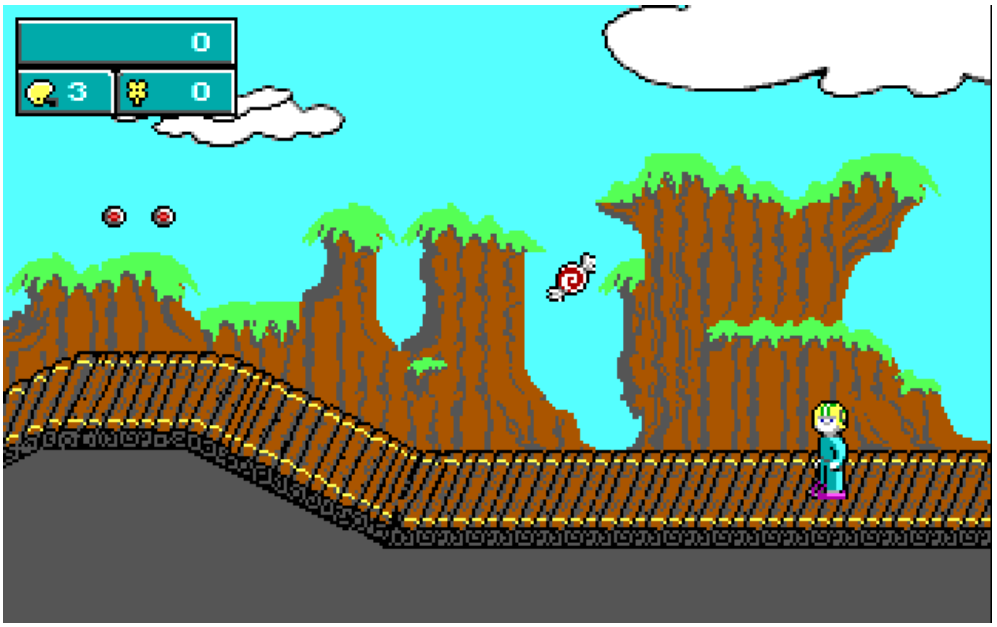


Figure 4.38: Left screen is before scrolling, right screen after scrolling.

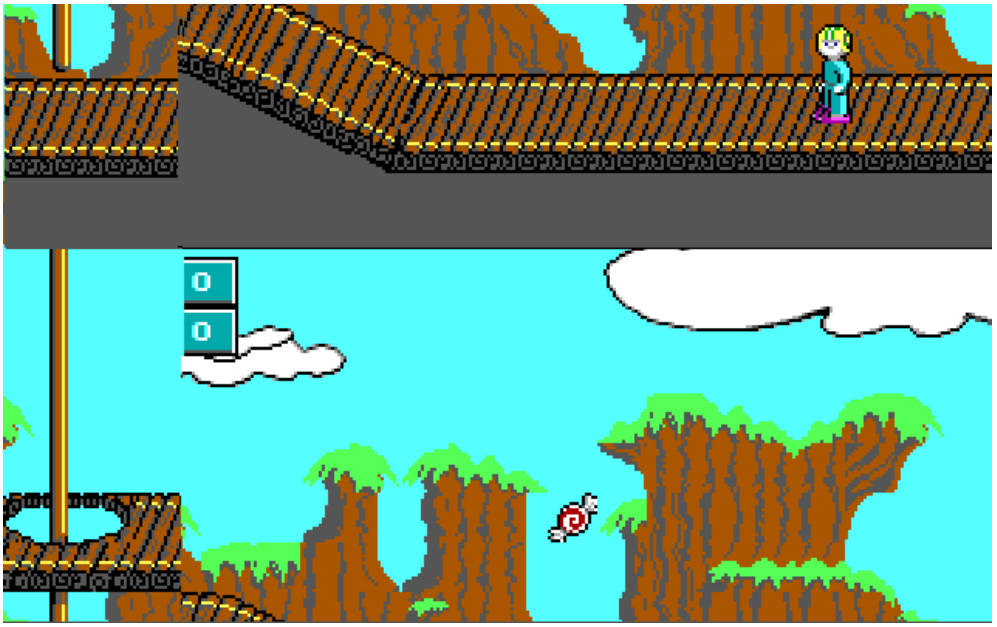
If the player continues to move to the left, the tile array 0 pointer lowers a second time, underscoring why additional space is needed to allow the entire viewport to float up to two tiles in all directions.

4.10.4 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, showing both misalignment and parts of two pages:



This problem results from the timing between updating the CRTC start address and the screen refresh. The start address is latched by the EGA's internal circuitry exactly once per frame, usually at the beginning of the vertical retrace. Although the CRTC start address is a 16-bit value, the `out` instruction can only write 8 bits at a time.

This issue can be illustrated with the following setup: the current CRTC start address (Page 0) points to `0000h`, while the buffer (Page 1) points to `3C00h`. After moving one tile to the left, Page 0 now points to `FFFEh` in VRAM, and Page 1 points to `3BFEh`. Since Page 1 is the updated buffer, it will be displayed in the next refresh cycle. However, due to poor timing in updating the vertical retrace and start address, the CRTC only picks up the first byte of the address, `3Bh`, setting the start address to `3B00h` instead of `3BFEh`.

```
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
```

One solution to this issue is to update the start address when the vertical retrace signal is detected through the Input Status 1 Register, specifically bit 3 of `3DAh`. Unfortunately, by the time this status is observed by the program, the start address for the next frame has already been latched, occurring immediately when the vertical retrace pulse begins.

To address this issue, the start address must be updated at a point sufficiently distant from the start of the vertical retrace. This requires identifying a signal that confirms the completion of a horizontal or vertical retrace and the beginning of a new scan line, far enough from the vertical retrace to ensure the new start address is latched during the next vertical sync. The Display Enable Status signal, accessible via the Input Status 1 Register, provides this information; a value of 1 indicates that the display is within a horizontal or vertical retracement[17].

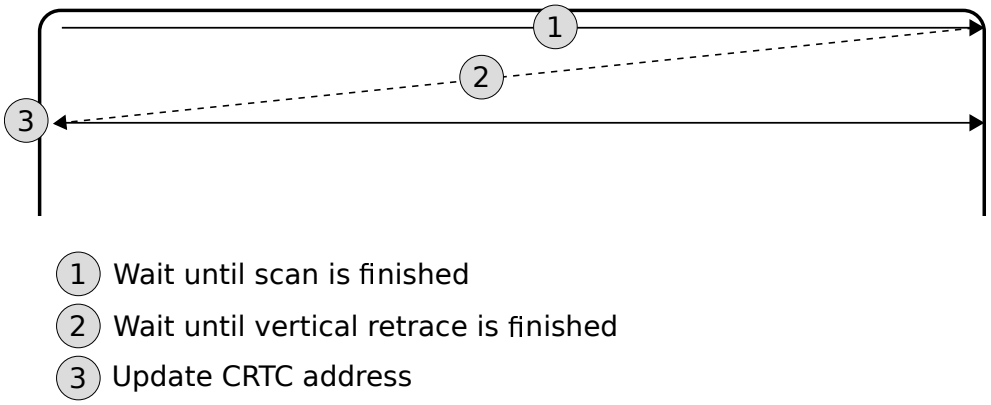


Figure 4.39: Update CRTC start address at beginning of new scan line.

To guarantee that the start address updates at the very beginning of a new scan line, the process first waits until the current scan line completes. Then, it waits for a full retrace, checking that the Display Enable Status returns to '0'. At this point, the CRTC address is updated.

Trivia : Regardless of CPU speed, the game's speed is limited by the CRTC sync to the monitor's refresh rate, which is 60Hz for EGA and 70Hz for VGA.

[17] 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. The IBM technical EGA documentation explains the opposite, saying when bit 0 is set low 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
;
  mov cx,[crtc]
  mov dx,CRTC_INDEX
  mov al,0ch    ;start address high register
  out dx,al
  inc dx
  mov al,ch
  out dx,al
  dec dx
  mov al,0dh    ;start address low register
  out dx,al
  mov al,cl
  inc dx
  out dx,al
```

4.11 Actors and AI

4.11.1 State Machine

All objects in the game, such as Commander Keen, enemies, bonus points, doors, and even the scoreboard, are called "actors". Each actor can "think" and perform actions like walking, shooting, or emitting sounds. Actors are controlled via a state machine, enabling them to take actions such as chasing the player, throwing objects, or doing nothing at all. To model their behavior, all enemies have an associated state, which can include:

- Chasing Keen
- Hitting or smashing Keen
- Shooting projectiles
- Climbing and sliding on poles
- Turning into a flower
- Special Boss (Boobus)

Each state has associated think, reaction, and contact method pointers. Additionally, there is a `tictime` and `*nextstate` pointer, which indicate when the actor should transition to another state after a specific number of tics have passed in the current state.

```
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;
```

Each actor has a defined state chain, as example the pea pod.

```
statetype s_peapodwalk1 = {PEAPODRUNL1SPR,PEAPODRUNR1SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk2};
statetype s_peapodwalk2 = {PEAPODRUNL2SPR,PEAPODRUNR2SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk3};
statetype s_peapodwalk3 = {PEAPODRUNL3SPR,PEAPODRUNR3SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk4};
statetype s_peapodwalk4 = {PEAPODRUNL4SPR,PEAPODRUNR4SPR,step,false, true
    ,10, 128,0, PeaPodThink, NULL, WalkReact, &s_peapodwalk1};

statetype s_peapodspit1 = {PEAPODSPITLSPR,PEAPODSPITRSPR,step,false, true
    ,30, 0,0, SpitPeaBrain, NULL, DrawReact, &s_peapodspit2};
statetype s_peapodspit2 = {PEAPODSPITLSPR,PEAPODSPITRSPR,step,false, true
    ,30, 0,0, NULL, NULL, DrawReact, &s_peapodwalk1};
```

Different types of enemies have their own state machines, often sharing reaction function (e.g., WalkReact and ProjectileReact) but usually possessing their own unique "thinking" states.

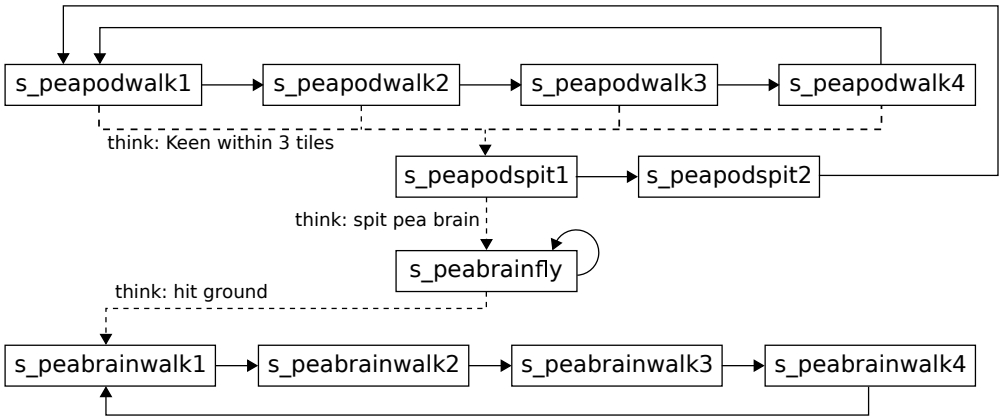


Figure 4.40: State machine for Pea pod and Pea brain.

The `*react` function is responsible for managing how an enemy reacts to the level, like turning around upon hitting a wall or the edge of a platform. The `*think` function defines how an enemy behaves when Commander Keen is nearby (e.g., attacking or firing a projectile) or when it reaches an edge (e.g. jumping). In some cases it introduces randomness, like when a pea pod might decide to spit a pea brain.

```
void PeaPodThink (objtype *ob)
{
  if ( abs(ob->y - player->y) > 3*TILEGLOBAL )
    return;

  if (player->x < ob->x && ob->xdir == 1)
    return;

  if (player->x > ob->x && ob->xdir == -1)
    return;

  // Randomness to spit pea brain
  if (US_RndT()<8 && ob->temp1 < MAXPEASPIT)
  {
    ob->temp1 ++;
    ob->state = &s_peapodspit1;
    ob->xmove = 0;
  }
}
```

The `*contact` function checks if an object has come into contact with another object and defines the resulting interaction, such as Commander Keen taking damage or losing a life.

4.11.2 Clipping

Whether an actor can move or fall through a tile is determined by the tile property `tinf[]`. Each foreground tile includes four directional parameters: north, south, east, and west. If, for example, the east parameter has a value greater than 0, the tile has a solid east wall, and the actor cannot enter it from the east side. When a sprite moves to the left and encounters a solid tile from the east, the engine adjusts the actor's movement to prevent it from entering the tile.

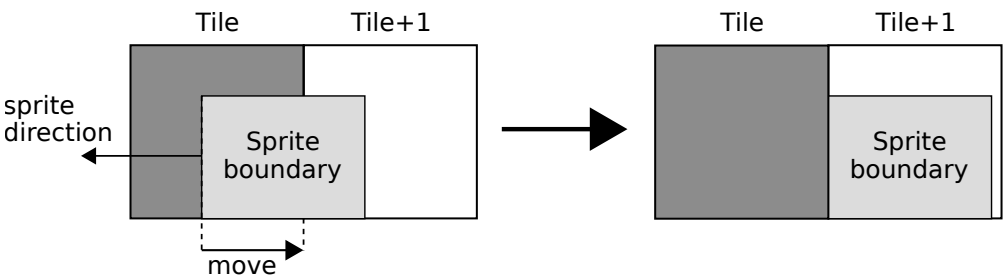


Figure 4.41: Clipping to east wall when actor moves left.

```
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 along the top and bottom of tiles, the engine also accounts for standing on slopes. After the actor is clipped to the top or bottom of a slope tile, an offset is applied to move the actor up or down along the slope.

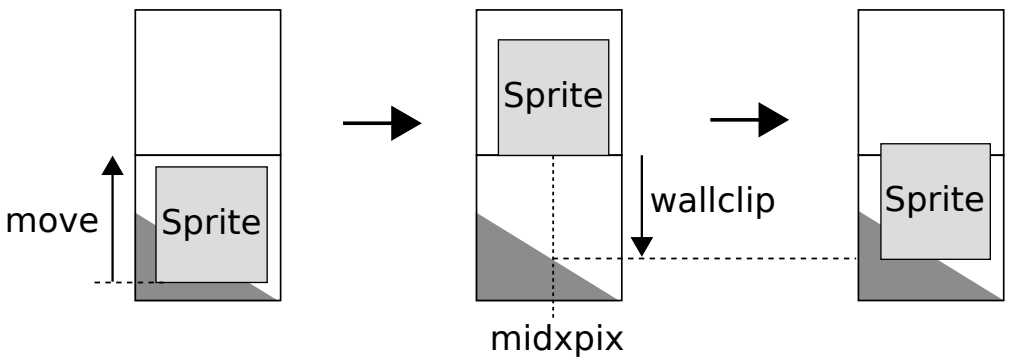


Figure 4.42: Clipping north wall with slope.

This offset is defined by the lookup table `wallclip[][]`, which uses the actor's midpoint and the wall type to determine the slope type.

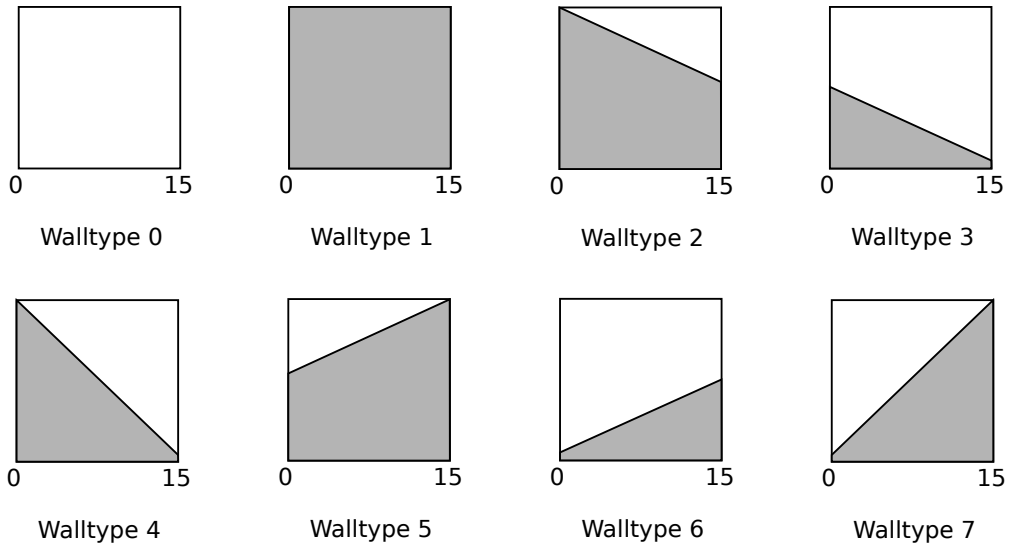


Figure 4.43: `wallclip[8][16]`: Fall through, solid and 6 slope types.

```
void ClipToEnds (objtype *ob)
{
  [...]
  //Get midpoint of sprite [0-15]
  midxpix = (ob->midx&0xf0) >> 4;
  for (y=oldtilebottom-1 ; y<=ob->tilebottom ; y++,map+=
   mapwidth)
  {
    //Do we hit a NORTH wall
    if (wall = tinf[NORTHWALL+*map])
    {
      //offset from tile border clip
      clip = wallclip[wall&7][midxpix];
      //Clip bottom side actor to top side tile + offset-1
      move = ( (y<<G_T_SHIFT)+clip - 1) - ob->bottom;
      if (move<0 && move>=maxmove)
      {
        ob->hitnorth = wall;
        MoveObjVert (ob,move);
        return;
      }
    }
  }
```

4.12 Drawing layer for layer

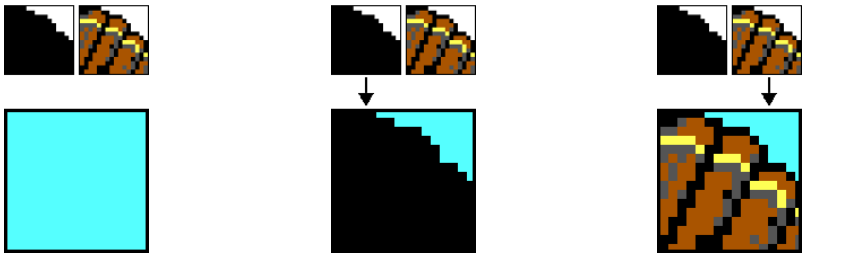
Each tile on the screen can contain up to three layers: the background layer, the foreground layer and a sprite layer. Rendering the final screen requires multiple redraws on the same tile to achieve the correct layering:

1. Draw the background layer or a combined background and foreground tile.
2. Draw the sprites.
3. Re-draw the foreground layer if a sprite should not appear on top.

It must run quickly and therefore most of the code is written in assembly.

4.12.1 Draw background and foreground tiles

Drawing the background layer is straightforward, as it only requires copying 128 bytes to VRAM. Drawing foreground tiles on top of the background layer requires an additional mask, which defines which pixels are overwritten by the foreground tile.



(a) Background tile. ***(b)*** `AND` foreground mask. ***(c)*** `OR` foreground bitmap.

Figure 4.44: Draw masked foreground tile.

Combining the background and foreground layers involves an `AND`-bitwise operator to clear the background and the `OR`-bitwise operator to write the foreground layer.

```
PUBLIC  RFL_NewTile

[...]
  es,[mapsegs+2]        ;foreground plane
  mov bx,[es:si]
  mov es,[mapsegs]      ;background plane
  mov si,[es:si]

[...]
  or  bx,bx             ;do we have foreground tile?
  jz  @@singletile      ;draw background tile only
  jmp @@maskeddraw      ;draw both together

[...]
@@maskeddraw:
  shl bx,1
  mov ss,[grsegs+STARTTILE16M*2+bx]
  shl si,1
  mov ds,[grsegs+STARTTILE16*2+si]

  xor si,si             ;first word of tile data

  mov ax,SC_MAPMASK+0001b*256 ;map mask for plane 0

  mov di,[cs:screenstartcs]

@@planeloopm:
  WORDOUT
tileofs = 0
lineoffset  = 0
REPT  16
  mov bx,[si+tileofs]   ;background tile
  and bx,[ss:tileofs]   ;mask
  or  bx,[ss:si+tileofs+32] ;masked data
  mov [es:di+lineoffset],bx
tileofs   = tileofs + 2
lineoffset  = lineoffset + SCREENWIDTH
ENDM
```

4.12.2 Drawing sprites

The next step is to render sprites on the screen. Most home computers of that era had built-in sprite functionality on the video card. For example, on a MSX computer, one could simply enter

```
PUT SPRITE <SpriteNumber>, <X>, <Y>, Color
```

to display a sprite on the screen. Updating the `(X, Y)` coordinates would move the sprite, with the display adapter handling everything else. Unfortunately, the concept of sprites did not exist on EGA cards, so game developers had to implement their own solution.

A challenge arises from the fact that sprites can move freely across the screen and are not byte-aligned. To address this, the bit-shifting technique described in section "User Manager (US)" on page 99 is used. When caching a sprite into memory, each sprite is copied four times, with each copy shifted by two or more pixels. The property `*spr->shifts` determines the number of bit shifts applied to each of the four copies.

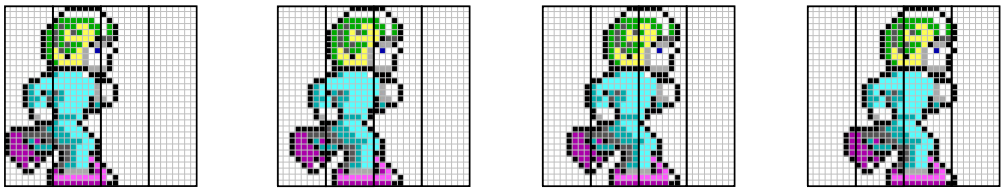


Figure 4.45: Sprite shifted in 4 steps.

Displaying the correct shifted sprite is as simple as

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

//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
```

```
void CAL_CacheSprite (int chunk, char far *compressed)
{
[...]
//
// make the shifts!
//
  switch (spr->shifts)
  {
  [...]
  case  4:
    dest->sourceoffset[0] = shiftstarts[0];
    dest->planesize[0] = smallplane;
    dest->width[0] = spr->width;

    dest->sourceoffset[1] = shiftstarts[1];
    dest->planesize[1] = bigplane;
    dest->width[1] = spr->width+1;
    CAL_ShiftSprite ((unsigned)grsegs[chunk],dest->
   sourceoffset[0],
      dest->sourceoffset[1],spr->width,spr->height,2);

    dest->sourceoffset[2] = shiftstarts[2];
    dest->planesize[2] = bigplane;
    dest->width[2] = spr->width+1;
    CAL_ShiftSprite ((unsigned)grsegs[chunk],dest->
   sourceoffset[0],
      dest->sourceoffset[2],spr->width,spr->height,4);

    dest->sourceoffset[3] = shiftstarts[3];
    dest->planesize[3] = bigplane;
    dest->width[3] = spr->width+1;
    CAL_ShiftSprite ((unsigned)grsegs[chunk],dest->
   sourceoffset[0],
      dest->sourceoffset[3],spr->width,spr->height,6);

    break;
  default:
    Quit ("CAL_CacheSprite: Bad shifts number!");
  }
}
```

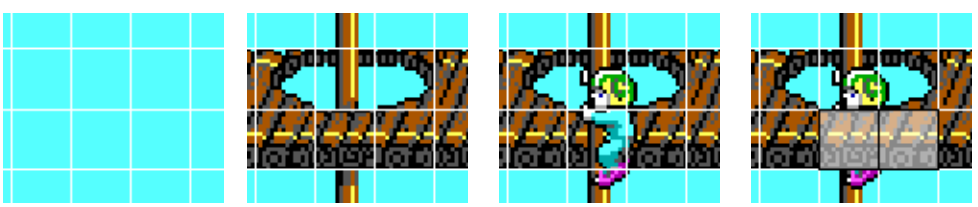
If multiple sprites are displayed on the same tile, each sprite is assigned a priority from 0 to 3 to determine the drawing order. A sprite with a higher priority number is always drawn on top of sprites with a lower priority. Since sprites are always drawn on top of tiles, this can create unnatural situations, such as when Commander Keen is climbing through a hole, as illustrated in Figure 5.46.

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

Figure 4.46: Unnatural situation where Commander Keen is in front of a hole.

To draw sprites 'inside' a foreground tile, a trick is used by introducing a priority foreground tile in the `tinf` info table. The attribute is named INTILE ("in-front"). If a foreground tile has its INTILE attribute high bit set (`80h`), sprites with priority 0-2 will not be drawn over it. This results in the following drawing order:

1. Draw the background tile and the masked foreground tile.
2. Draw sprites with priority 0, 1, and 2 (in that order), and mark the corresponding tiles in the tile buffer array with a '3', as illustrated in Figure 5.37 on page 140.
3. Scan the tile buffer array for tiles marked with '3'. If the corresponding foreground tile's INTILE attribute high bit (`80h`) is set, re-draw the foreground tile.
4. Finally, draw sprites with priority 3. These sprites are always drawn on top of everything.



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

Figure 4.47: Draw sprite inside a tile, by redrawing foreground tile.

```
PROC  RFL_MaskForegroundTiles
PUBLIC  RFL_MaskForegroundTiles

[...]

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

; mask the tile
```

4.12.3 Tile Draw Performance Tricks

To draw one background tile to VRAM, the engine must read and write 128 bytes (2 x 16 bytes x 4 memory banks), which is the minimum number of read/write operations required. In the worst case, up to 512 bytes of read/write operations are required (background, foreground, sprite, and foreground again), with multiple bitwise operations involved. The engine employs several tricks to squeeze the maximum out of each CPU cycle: background tile caching and word-aligned memory writing.

4.12.3.1 Background Tile Caching

When updating the master screen in VRAM, the engine copies each tile from RAM to VRAM. Copying each pixel involves one read and one write operation across the four memory banks. Section "Wrap around the EGA Memory" on page 129 explains how reprogramming EGA latches enables copying four bytes at once between VRAM locations, a technique that can be applied as well to tiles containing only a background layer.

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

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

The assembly function `RFL_NewTile` is responsible for drawing tiles to the master screen.

```
PROC  RFL_NewTile updateoffset:WORD
  [...]
  mov ax,[tilecache+si]
  or  ax,ax
  jz  @@singlemain   ; if 0, tile not in cache
;=============
; Draw single tile from cache
;=============
  [...]
  ret
;=============
; Draw single tile from main memory
;=============
@@singlemain:
  mov ax,[cs:screenstartcs]
  mov [tilecache+si],ax   ;next time it can be drawn from
                          ;here with latch
```

It first checks whether the background tile is already stored in the tile cache array. If it is, the tile is copied at 32 bits per cycle from VRAM to VRAM. Otherwise, the tile is loaded from RAM, and the VRAM pointer for that tile is stored in the tile cache array for future use.

4.12.3.2 Word-aligned memory writing

The 286 CPU can read and write 16 bits in a single cycle, but there is a caveat: this only works when accessing even memory addresses. Additionally, writing a word at the offset address `FFFFh` causes a CPU exception.

Because tiles are word-aligned (16 bits wide) and the screen refreshes in tile-sized steps, they are always aligned with even memory addresses. However, sprites are byte-aligned. When a sprite is drawn from an odd memory address, the CPU is limited to reading and writing 8 bits at a time. To fully utilize the 16-bit data bus, the engine uses small function routines for each combination of sprite width and even/odd address alignment.

Take for example drawing a 6-byte wide sprite. An optimal write depends whether the start address is even or odd. In case of of an even address, writing three times a `word` is optimal. In case of an odd address, the optimal is to first write a `byte`, followed by two `word` and then a `byte` again.

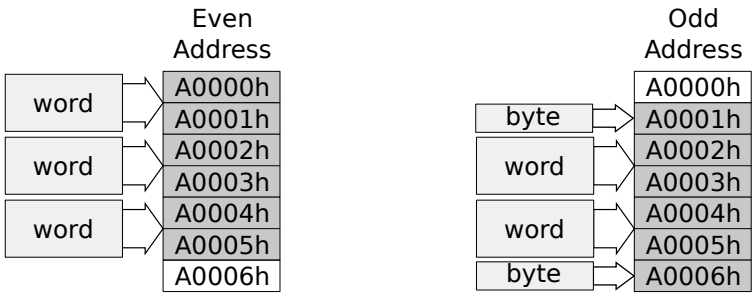


Figure 4.48: Word-optimized 6 byte wide sprite to even (`MASK6E`) and odd (`MASK60`) address.

In total 23 functions are defined to optimize `word`-size writing up to 10 bytes wide sprites. Each function pointer reference is stored in an array.

```
maskroutines
  dw   mask0,mask0,mask1E,mask1E,mask2E,mask20,mask3E,mask30
  dw   mask4E,mask40,mask5E,mask50,mask6E,mask60
  dw   mask7E,mask70,mask8E,mask80,mask9E,mask90
  dw   mask10E,mask100
```

The engine first determines whether the destination is at an even or odd memory address. If the address is odd, it first writes a single byte to align the subsequent operations to an even address, after which it continues writing in 16-bit words. By cleverly using bit-shift operations and the Carry Flag (CF), the engine calls the appropriate function to write the sprite to VRAM.

```
PROC  VW_MaskBlock  segm:WORD, ofs:WORD, dest:WORD, wide:
   WORD, height:WORD, planesize:WORD

[...]

@@unwoundroutine:
  mov cx,[dest]
  shr cx,1
  rcl di,1          ;shift a 1 in if destination is odd
  shl di,1          ;to index into a word width table
  mov ax,[maskroutines+di]  ;call the right routine
  mov [routinetouse],ax   ;and store the function pointer

@@startloop:
  mov ds,[segm]

@@drawplane:
  [...]
  mov si,[ofs]         ;start back at the top of the mask
  mov di,[dest]        ;start at same place in all planes
  mov cx,[height]        ;scan lines to draw
  mov dx,[ss:linedelta]

  jmp [ss:routinetouse]   ;draw one plane
```

4.13 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 requires using interrupts to pause normal execution and perform tasks in parallel.

4.13.1 Interrupts

When the user presses a key on the keyboard, how does the program know to respond to that event? We can take educated guesses about what happens in that situation. Perhaps the keyboard controller flips a byte in a memory-mapped area somewhere? Maybe it stores the keypress in a temporary buffer and makes it available on an I/O port whenever the program is ready to receive it?

Polling approaches require each program to actively monitor hardware events at regular intervals. No matter what a program is doing, it must remember to occasionally stop, communicate with the keyboard hardware to see if any keypresses have come in, and react to them if so. If the program gets busy or forgets to check, the keyboard goes un-serviced. And that's just one piece of hardware. Add to that the system timer, real time clock, mouse movement and disk drives... the list of things that need to be checked grows out of control. Moreover, polling can waste resources when no new hardware events have occurred.

The solution to this is "interrupts". At the processor level, an interrupt request (IRQ) is a special event caught by a system called PIC, that causes the flow of program execution (e.g. running the 2D renderer) to be suspended, followed by an unconditional jump to a specific section of code known as the interrupt service routine (ISR). The service routine does whatever it needs to do to adequately respond to the event, and then it signals a return. The return causes execution to jump right back to where it was before the interrupt was received, and the original program continues as if nothing had happened.

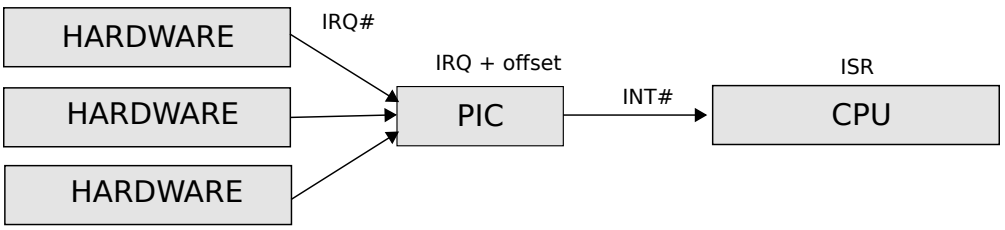


Figure 4.49: Hardware interrupts are translated to software interrupt via the PIC.

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[18]. This path introduces the problem of discarding important information such as keyboard or mouse inputs.

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

4.13.2 IRQs and ISRs

The audio and heartbeat system relies on two chips: the Intel 8254, which functions as a Programmable Interval Timer (PIT), and the Intel 8259, which acts as a Programmable Interrupt Controller (PIC). The PIT features a crystal oscillating at 1.193182 MHz. At its core, the PIT is a decrementing counter. The programmer loads a 16-bit value between 0 and 65,535 into a register in the PIT. With each clock pulse, the counter decrements toward zero. Once it reaches zero, it automatically resets to the original value stored in the register and starts over.

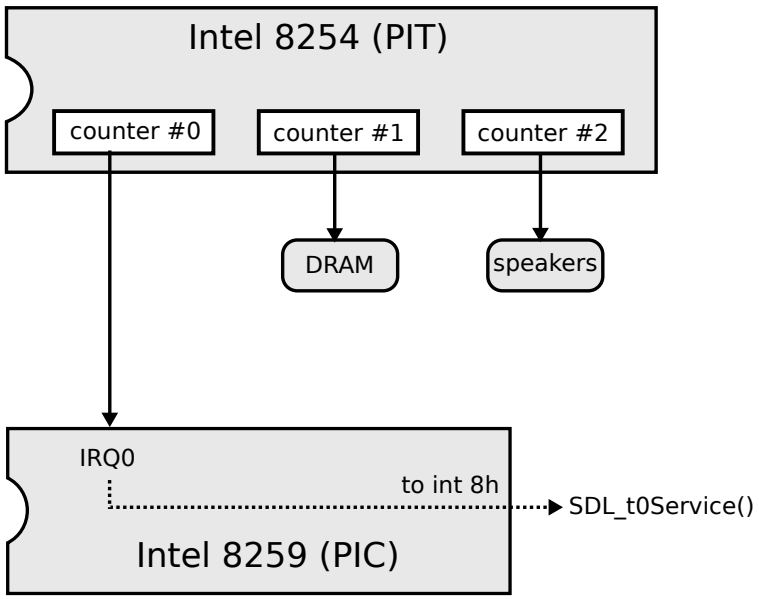


Figure 4.50: Interactions between PIT and PIC.

[18]Interrupt Mask Register

In fact, there are three counters in the PIT. Counter #1 is connected to the RAM in order to automatically perform something called "memory refresh" and was considered a "do not touch" part of the PIT[19]. Counter #2 is connected to the PC speaker and generates sounds, and will be explained in detail in the next section. Counter #0 is connected to the PIC and when it hits zero it triggers IRQ 0 and sends it to the PIC. The PIC manages hardware interrupts, mapping IRQ 0 - IRQ 8 to the Interrupt Vector Table (IVT), a list of pointers to the corresponding ISR addresses. Notice that IRQ 0 (mapped to IVT entry #8) is associated with the System timer and usually updates the operating system clock.

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

Figure 4.51: The Interrupt Vector Table (entries 0 to 18).

4.13.3 Hijacking the System Timer

By modifying the IVT #8 pointer, an application can hijack the interrupt to serve its own purposes. When this occurs, the engine halts its runtime at regular intervals and jumps to a custom interrupt function. We now have two systems running in parallel.

[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.

```
static void interrupt SDL_t0Service(void)
{ [...] }

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

  SDL_InitDelay();  // SDL_InitDelay() uses t0OldService

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

}
```

IVT #8, in its original configuration, not only operates the system clock but also manages the floppy disk motor. Specifically, it ensures the motor shuts off after a read or write operation. When IVT #8 is hijacked, this functionality is bypassed, causing the floppy disk motor, after loading data from the disk, to run indefinitely. Although this does not cause issues, the constant spinning of the disk can be both noisy and confusing, potentially giving users the impression that data loading is still in progress.

The current status of the disk motors is stored in the BIOS Data Area (BDA), which is a section of memory located at segment `0040h`. The BDA stores many variables indicating information about the state of the computer.

Address #	Description
40:00h	I/O ports for COM1-COM4 serial
40:08h	I/O ports for LPT1-LPT3 parallel
40:17h	Keyboard state flags
40:1Eh	Keyboard buffer
40:3Fh	Floppy disk drive motor status
40:40h	Floppy disk drive motor time-out counter
40:41h	Floppy disk drive status
40:49h	Display Mode
40:4Ah	Number of columns in text mode
40:75h	Number of hard disk drives detected

Figure 4.52: Partial list of BIOS Data Area variables[20].

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

BIOS data address 40:3Fh holds the motor status, where bit 0 indicates if the disk 1 motor is on and bit 1 if the disk 2 motor is on. BIOS data address 40:40h contains the disk motor shutoff counter. This counter is decremented by the timer interrupt vector. When the counter reaches 0, the disk motor is turned off.

The hijacked interrupt subsystem is taking over responsibility for this functionality. It checks whether either disk motor is running and decrements the shutoff counter as needed. When the counter drops below 2, the subsystem invokes the original timer interrupt to ensure the disk motor is properly shut down.

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

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

4.13.4 Heartbeats

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. By default, the interrupts are generated at a frequency of 1.19318MHz / 65,536 = 18.2Hz. To change the interrupt frequency, the timer can be reprogrammed by simply adjusting 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_SetIntsPerSec` is using a frequency of 1.192755MHz, instead of the PIT documented 1.193182MHz. This difference is likely derived from the calculation 18.2 Hz * 65,536 = 1.192755MHz.

The engine can decide at what frequency to be interrupted, depending on the type of sound 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);
}
```

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
}
```

Once the interrupt is completed, the interrupt handler must explicitly acknowledge the IRQ at the interrupt controller to return to the game engine loop. This is accomplished by writing an "end of interrupt" signal byte to I/O port `20h`.

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

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

4.13.5 Manage Refresh Timing

After each screen refresh, a certain amount of ticks has passed. The tick count depends on several factors, such as the number of tiles refreshed and the waiting period for a screen's vertical retrace. Since all game actions and reactions rely on the tick interval between two refreshes, it is important to keep this interval consistent.

Without controlling the tick interval, the state and speed of actors can become unpredictable, potentially causing them to move too quickly or even "warp" to an unexpected location. To manage refresh intervals effectively, a minimum and maximum number of ticks are defined within 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;
    }
}
```

4.13.6 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);

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 three sound outputs: AdLib, SoundBlaster and PC Speaker. All belong to one of the two supported families of sound generators: Square Waves (PC speaker) or FM Synthesizer (Frequency Modulation).

Sounds effects are stored in two formats.

1. PC Speaker.
2. AdLib.

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 two formats by using `STARTPCSOUNDS` + `sound_ID` or `STARTADLIBSOUNDS` + `sound_ID`.

Although the Sound Manager was designed to support music and digital sound playback on SoundBlaster and Disney Sound Source, this functionality was never implemented in Keen Dreams (as explained in section "Audio" on page 78) and will therefore not be further explained in this section.

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


#define NUMSOUNDS            28
#define NUMSNDCHUNKS         84

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

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

4.13.7 PC Speaker

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.

Earlier it was hinted that the PIT had three counter channels, of which channel 2 was used for PC speaker output. The counter output mode can be reprogrammed to "square wave generator" mode. If counter #2 is closer to its starting value than zero, the output of the PIT is a high electrical signal. If the counter is closer to zero, the output is low. The end result is a square wave, high half of the time and low the other half, with the frequency controlled by the value in the PIT register. This square wave signal is amplified and fed into the speaker.

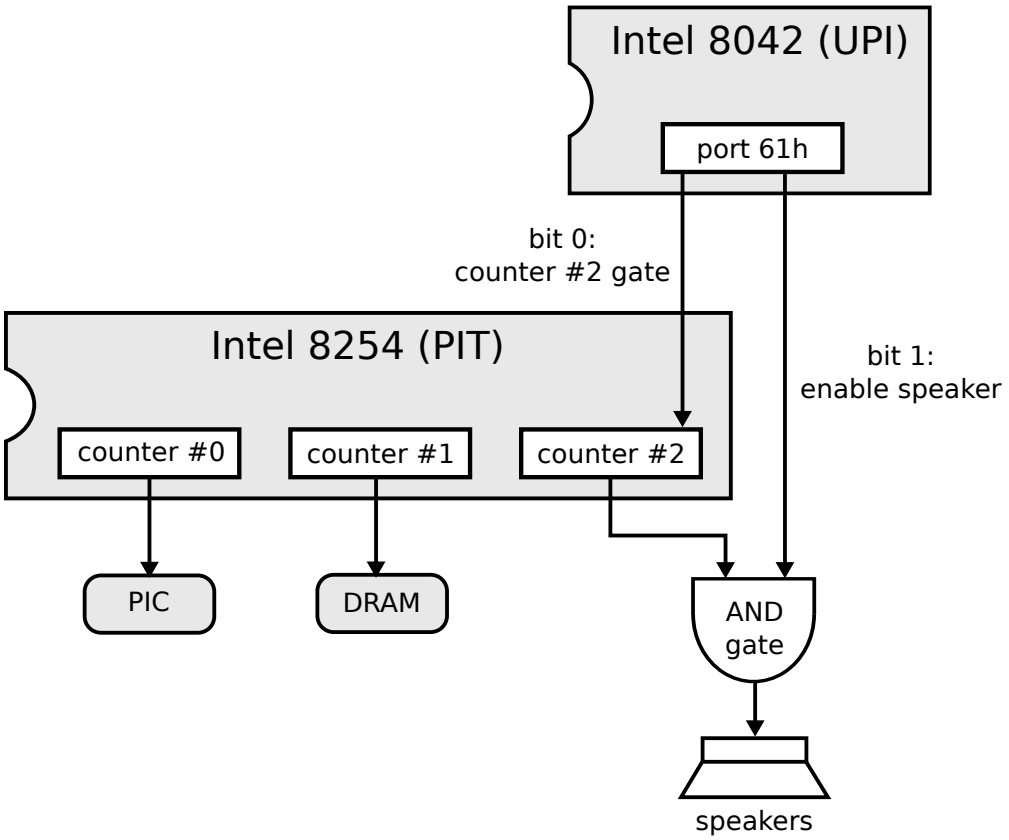


Figure 4.53: Built-in speaker hardware diagram.

A simple tone has only one frequency. As a practical example, say we want to play a middle C through the speaker. Middle C is 261.626 Hz[21], and the PIT clock runs at 1.193182 MHz. Dividing the latter by the former and rounding to the nearest integer value yields 4561. This is the value that must be written to the PIT counter #2 to produce the desired tone. While the counter is above 2280 the output signal is high, below that threshold the output signal is low. Once the counter reaches 0, it automatically resets to the initial value of 4561 and the cycle repeats. To create a higher pitch, the counter value would need to be lower.

To adjust the frequency, write to port `43h` to set the PIT command register, followed by writing the desired counter value to port `42h`.

Bit #	Value	Description
0	0	Set value for counter 2 (at port `42h`).
1	1	
2	1	Because the data port is an 8 bit I/O port and the count values is 16 bit, the PIT chip needs to be instructed 16 bits are transferred as a pair, starting with the lowest 8 bits followed by the highest 8 bits.
3	1	
4	1	Set to square wave generator mode.
5	1	
6	0	
7	0	Counter is a 16-bit binary counter (0-65535).

Figure 4.54: Set PIT Command register (port `43h`) to value `b6h`[22].

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.

[21] 261.626 Hz for middle C is a consequence of the twelve-tone equal temperament, see https://en.wikipedia.org/wiki/12_equal_temperament.

[22] For details, see https://wiki.osdev.org/Programmable_Interval_Timer

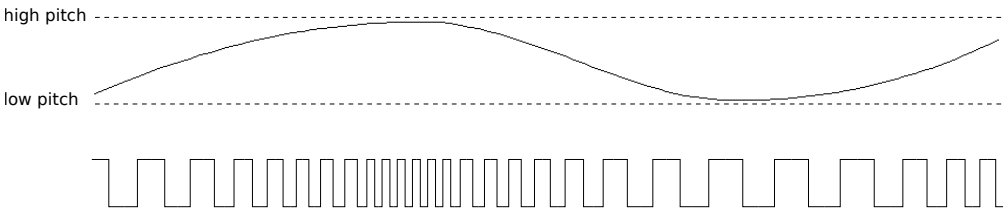


Figure 4.55: Sound pitch approximated with square wave and frequency changes.

The connection to the speaker can be deactivated without changing any timer parameters through the system's keyboard controller (Intel 8042 UPI). Setting port `61h`, bit 0 and bit 1, to 0 turns off both counter #2 and the speaker.

```
static void SDL_PCService(void)
{
  byte  s;
  word  t;
  s = *pcSound++;

  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
  }
}
```

Human hearing ranges from approximately 20 Hz to 20,000 Hz, making any counter value lower than 60 inaudible. Frequencies are encoded in a stream of bytes (0-255) and decoded using the formula

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

The lowest frequency is 78 Hz and the highest frequency is 19,886 Hz. 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.

```
word       pcSoundLookup[255];

void
SD_Startup(void)
{
    [...]

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

}
```

4.13.8 AdLib

The AdLib sound relies on the OPL2 chip. 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. On an Intel 286 it is implemented as a 10 microseconds and 25 microseconds respectively.

The engine does not know about any of the details of the OPL2. There is zero abstraction layer of transformation here. An IMF[23] 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.

```
////////////////////////////////////////////////
//
//  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
}
```

[23] id Music Format

4.14 Keyboard

Before Microsoft introduced the DirectInput API in Windows 95, developers needed to write custom drivers for each input device. This required direct communication with hardware using the vendor's protocol over physical ports.

4.14.1 Keyboard scancodes

Each key on a PC keyboard is linked to a scancode. When a key is pressed, circuitry in the keyboard generates the scancode and sends it serially to the keyboard controller. This key-down event, also known as a "make code", signals the key press. When a key is released, its scancode high-bit (80h) is set, producing a "break code" that is sent by the same mechanism. For example, pressing "A" generates the make code 1Eh, while releasing it sends the break code 9Eh. Holding down a key will type a repeating sequence of that character after a short delay, resulting in retransmitting the make code at regular intervals for as long as the key is being held.

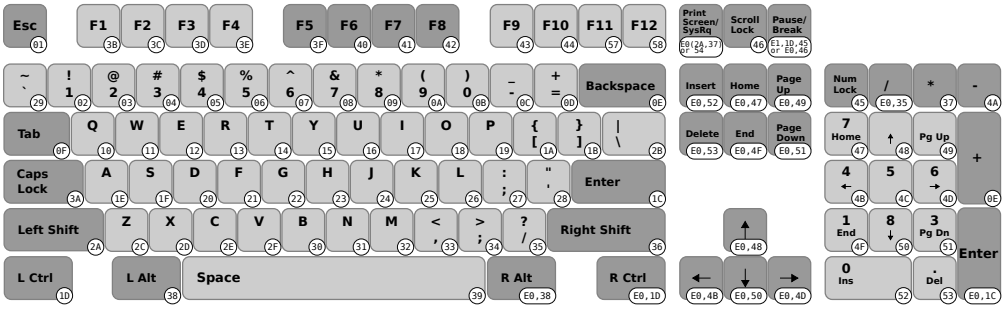


Figure 4.56: IBM PS/2 Keyboard scancode layout (scancodes in hexadecimal).

The PS/2 keyboard includes 14 keys, such as "Right Ctrl", "Right Alt", and the arrow keys, that duplicate functions from the original 84-key AT layout. To save encoding space, these keys are sent as two-byte scancodes: the first byte is always E0h, followed by the make or break code of the equivalent key from the original AT layout. For instance, the standard "/"-key produces the scancode 35h, while the numeric keypad's "Num /"-key sends E0h followed by 35h. This allows software to differentiate between the keys without expanding the scancode table.

This encoding also has the additional benefit that, if the underlying software only understands the 84-key layout, it will (presumably) discard the E0h byte in this example and only process the 35h byte. This would result in a "/" being correctly typed from the numeric keypad, even though the software doesn't understand that this key exists.

Trivia : The `E0h` scancode introduced some unexpected behaviors. For example, typing "Shift" + "Num /" should produce a "/", as indicated on the keycap. However, older software may interpret this sequence incorrectly: Shift is down, I don't understand `E0h`, and here comes `35h`. That means the user wanted to type "?".

4.14.2 Keyboard controller

Once a key is pressed or released, the scancode is decoded and processed by the keyboard controller. On the IBM XT, this is an Intel 8255 PPI[24] chip. Its main purpose is to listen for serial data from the keyboard, verify that it arrived intact, store each incoming scancode in a buffer, and request an interrupt so it can be read by the software.

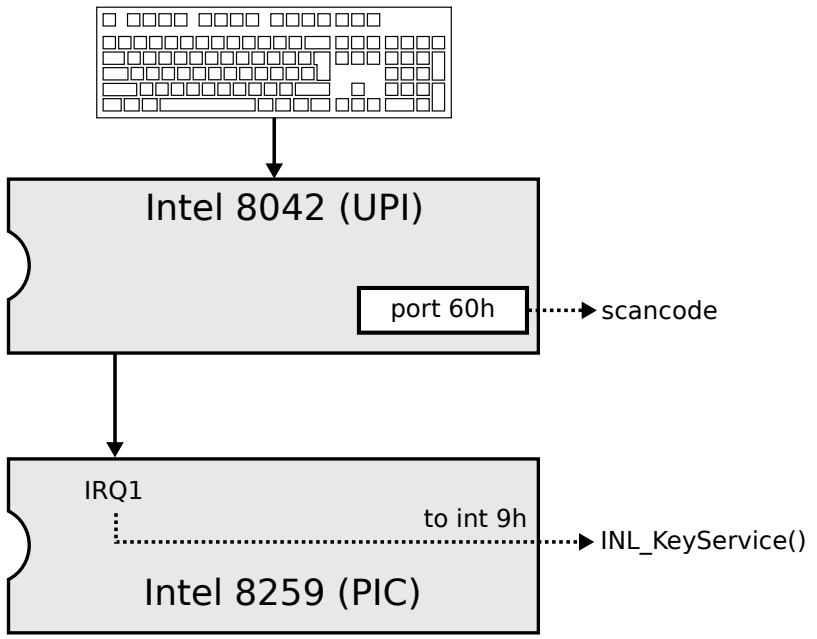


Figure 4.57: Keyboard hardware diagram.

With the advent of the IBM AT and the need for bidirectional communication with the keyboard for features like keyboard status LEDs, the Intel 8042 UPI[25] chip was introduced. The UPI retained backward compatibility with the PPI chip, handling all scancodes from the keyboard. When a key is pressed, the interrupt is routed to ISR #9 in the Vector Interrupt Table. The engine installs its own ISR there.

[24] Programmable Peripheral Interface
[25] Universal Peripheral Interface.

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

static void INL_StartKbd(void) {

  IN_ClearKeysDown();

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

static void interrupt INL_KeyService(void) {
  [...]
}
```

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];
```

When a scancode byte arrives from the keyboard, it is copied to a buffer in the keyboard controller and an IRQ #1 is raised. Scancodes arrive one byte at a time, and one interrupt at a time, even if it is a multi-byte code. The interrupt handler can read a byte from I/O port `60h` to capture the scancode for further processing. Once that is complete, the interrupt handler must explicitly acknowledge the IRQ at the interrupt controller to re-arm it for the next keyboard event. This is accomplished by writing an "end of interrupt" signal byte to I/O port `20h`.

There is a slight difference between the PPI and the UPI in terms of how the keyboard input buffer is managed. The PPI will hold a byte in its input buffer indefinitely until the software acknowledges that it has completed the read. The acknowledgment procedure is to briefly strobe the high bit of I/O port `61h` on, then off. When this occurs, the keyboard controller resets its buffer status and resumes reading from the keyboard. The UPI simplifies this process. Reading from I/O port `60h` automatically resets the buffer, eliminating the need for a separate acknowledgment. However, to maintain backward compatibility, most programs still toggle the high bit of I/O port `61h`, even though this has no effect on 286-based systems.

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

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

  if (k == 0xe0)     // Special key prefix
    special = true;
  else if (k == 0xe1) // Handle Pause key
    Paused = true;
  else
  {
    if (k & 0x80) // Break code
    {
      k &= 0x7f;
      Keyboard[k] = false;
    }
    else           // Make code
    {
      LastCode = CurCode;
      CurCode = LastScan = k;
      Keyboard[k] = true;
      [...]        //Process the key
    }
  }
  outportb (0 x20 ,0 x20 ); // ACK interrupt to interrupt
   system
}
```

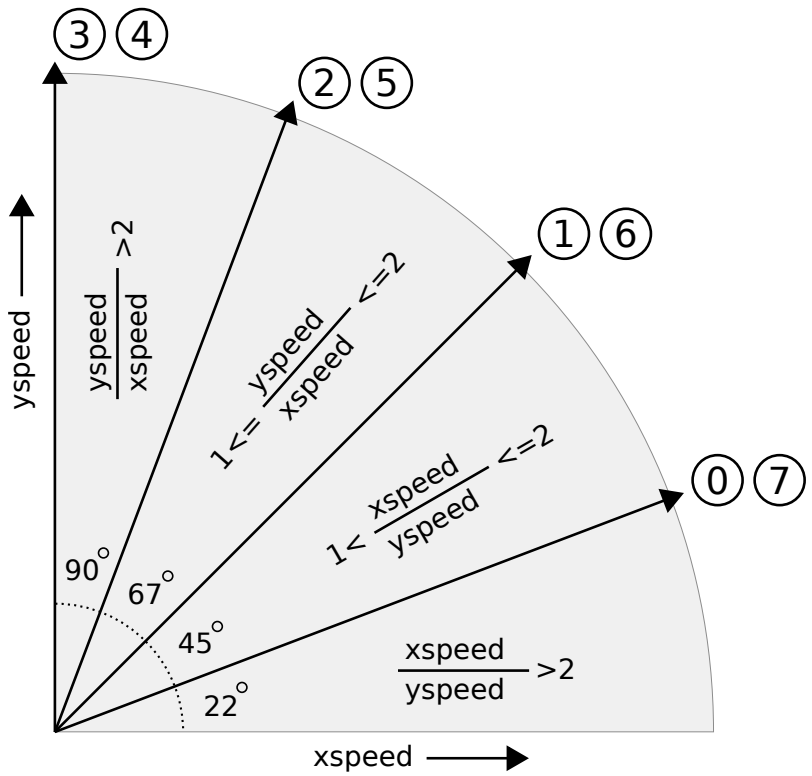
4.15 Random Tricks

Various smart techniques were employed to ensure every CPU cycle counted. This section describes random tricks used to increase game speed.

4.15.1 Bouncing Physics

When Keen throws a flower, it bounces off walls. For flat walls and floors, the bounce is easily calculated by reversing either the x-speed (for vertical walls) or the y-speed (for horizontal walls). However, handling bounces on slopes is more complex. Accurately calculating a bounce on a slope would require computationally expensive `cos` and `sin` operations.

To simplify this, the game uses an algorithm that approximates the angle to one of four values: 22°, 45°, 67° or 90°. Based on the ratio between the x-speed and y-speed, the corresponding angle and speed is calculated. The angle values 0-3 refer to a positive `xspeed`, and the values 4-7 to a negative `xspeed`.



The resulting speed is determined as a factor of either the `xspeed` or `yspeed`, depending on which of the two has the larger absolute value. For higher precision, the speed is expressed in global coordinates and therefore multiplied by 256.

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

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

    wall = ob->hitnorth;

    [...]

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

For each combination of the eight slope types and incoming angle, the bounce angle is determined using a simple lookup table.

	incoming angle							
Wall type	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	7	6	5	4	3	2	1	0
2	5	4	3	2	1	0	15	14
3	5	4	3	2	1	0	15	14
4	3	2	1	0	15	14	13	12
5	9	8	7	6	5	4	3	2
6	9	8	7	6	5	4	3	2
7	11	10	9	8	7	6	5	4

Figure 4.58: Bounce lookup table `bounceangle[8][8]`.

Each entry in the table corresponds to one of 16 possible bounce angles. For example, when a wall type 3 slope is hit with an incoming angle of 22° and postive `xspeed`, the lookup table refers to bounce angle #5. Each bounce angle is decomposed into a new `xspeed` and `yspeed`.

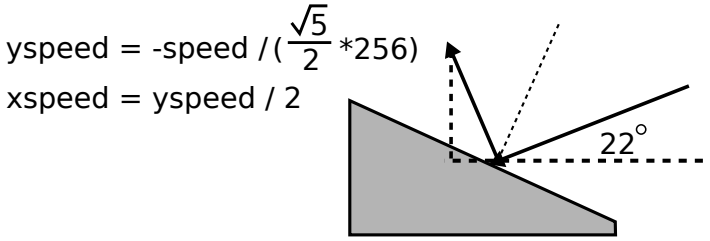


Figure 4.59: Walltype 3 with incoming angle of 22°.

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

case 3:

case 4:
  ob->xspeed = 0;
  ob->yspeed = -(speed / 256);
  break;
case 5:
  ob->yspeed = -(speed / 286);
  ob->xspeed = ob->yspeed / 2;
  break;

[...]
}
```

It is worth noting that in some cases, the resulting bounce angle does not follow the laws of physics. For instance, an incoming angle of 22° on a 45° slope results in a bounce angle of 90° instead of the expected 67°.

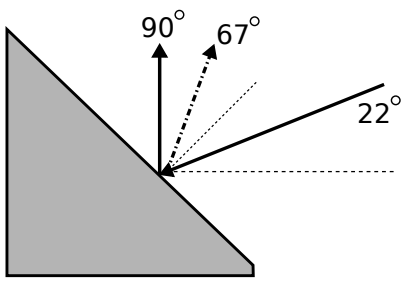


Figure 4.60: Incoming angle of 22° results in bounce angle of 90°.

4.15.2 Screen fades

When a new level is loaded, the screen fades from black to the default colors by reassigning the color palette. Each of the 16 color indices can be reprogrammed to any "RGBI" color, simply by calling the BIOS software interrupt `10h`.

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

By using `_AX=1002h`, the entire palette can be reprogrammed at once. In this process, `ES:BX` points to 17 bytes, where each byte represents an RGBI value for one of the 16 palette indices, plus one for the border.

#	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 4.61: Color fading table `colors[7][17]`.

Fading the screen from black to color is straightforward.

```
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;
}
```

4.16 Keen Dreams in CGA

The first Commander Keen serie, *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.

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!

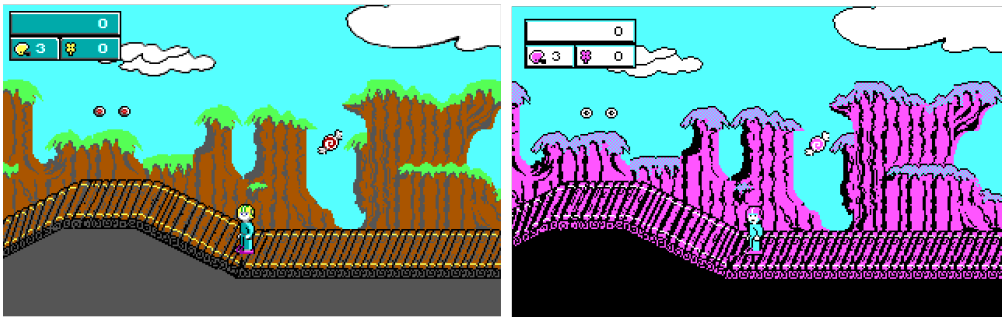


Figure 4.62: Keen Dreams EGA and CGA version.

4.16.1 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 8KiB each, resulting in 16KiB total.
- Character generator ROM, containing a 9x14 font and two 8x8 fonts. This is the same ROM as used on the MDA video card.

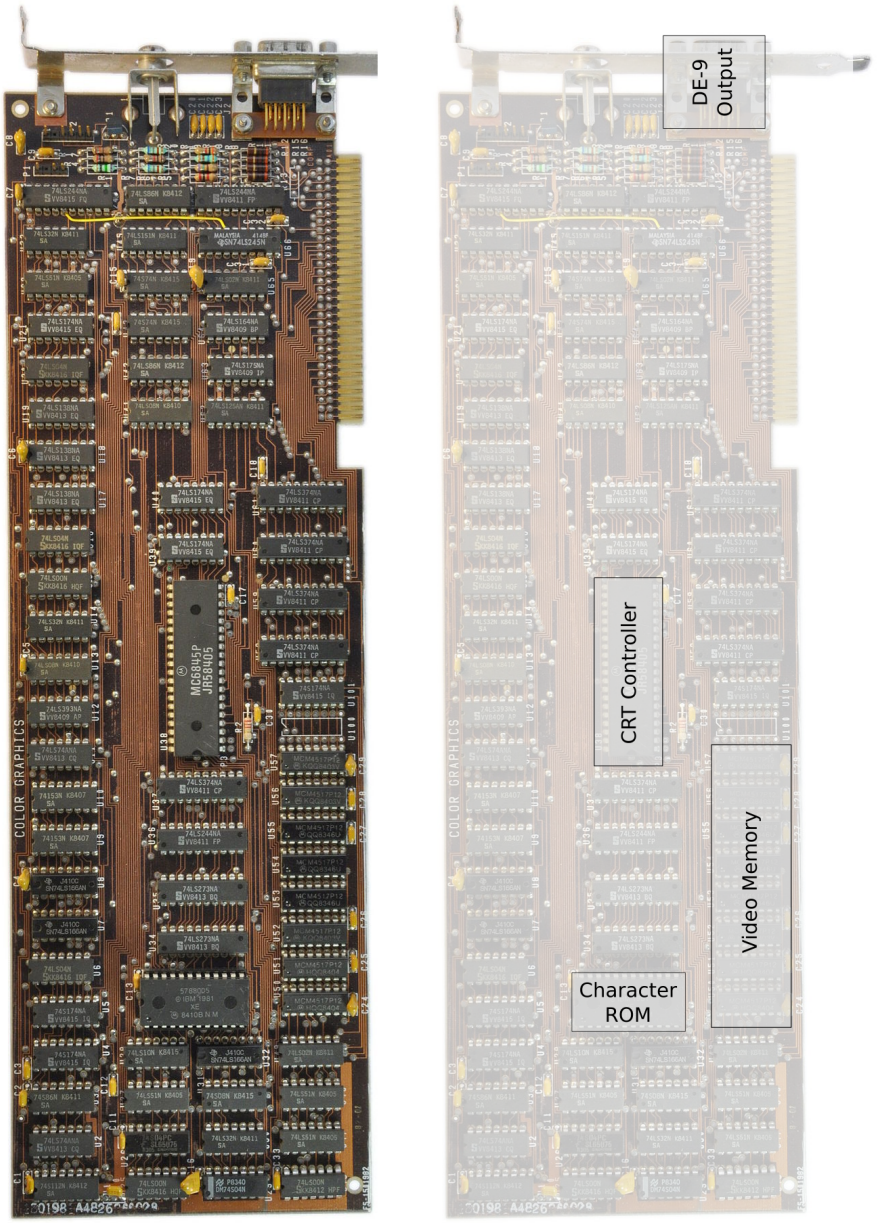


Figure 4.63: Original IBM CGA card

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 4.64: CGA Modes available.

In graphics mode 4, which is used by Commander Keen, only four colors could be 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 4.65: CGA color palettes.

The default palette when switching to Mode `04h` is palette 2 with high intensity, which is used by Commander Keen. Changing the color palette can be done using the video BIOS interrupt `10h`[26].

[26] See https://www.seasip.info/VintagePC/cga.html for more details.

```
_AH = 0x0B    // Set color palette
_BH = 1       // 4-color palette mode
_BL = 1       // 0=palette 1, 1=palette 2
geninterrupt (0x10) // Generate Video BIOS interrupt

_AH = 0x0B    // Set color palette
_BH = 0       // brightness
_BL = 0x10    // 10h=high, 0h=low
geninterrupt (0x10) // Generate Video BIOS interrupt
```

4.16.2 Interlacing

In the early days of television and computer monitors, technology was far less advanced than it is today. Picture resolution was constrained by the limitations of the hardware, and engineers were constantly devising clever solutions to maximize performance. During the 1950s and 1960s, when CRT[27] monitors and TVs were standard, one major challenge engineers faced was how to produce clear images without overwhelming the hardware. The solution was a technique called interlacing.

Interlacing worked by splitting each video frame into two halves: the odd-numbered lines and the even-numbered lines. Instead of rendering the entire frame at once, the screen would first display all the odd lines and then, after a fraction of a second, display the even lines. This approach effectively halved the amount of data processed and displayed at any given time, which was a significant advantage given the limited processing power and bandwidth of the era.

The CGA memory layout in graphics mode utilized a similar interlaced architecture. VRAM was split into two banks: VRAM bank 0, which held even rows of pixels (0, 2, 4, etc.), and VRAM bank 1, which held odd rows of pixels (1, 3, 5, etc.). This layout required additional calculation steps for many CGA graphics operations if the programmer wanted to avoid visual artifacts during screen updates.

In CGA graphics, each pair of two bits represented a single pixel, allowing for a color value between 0 and 3, based on the CGA color palette. The two leftmost bits in a byte represented pixel 0, the next two bits represented pixel 1, and so on. Each byte in VRAM corresponded to four pixels on the screen.

[27] Cathode Ray Tube

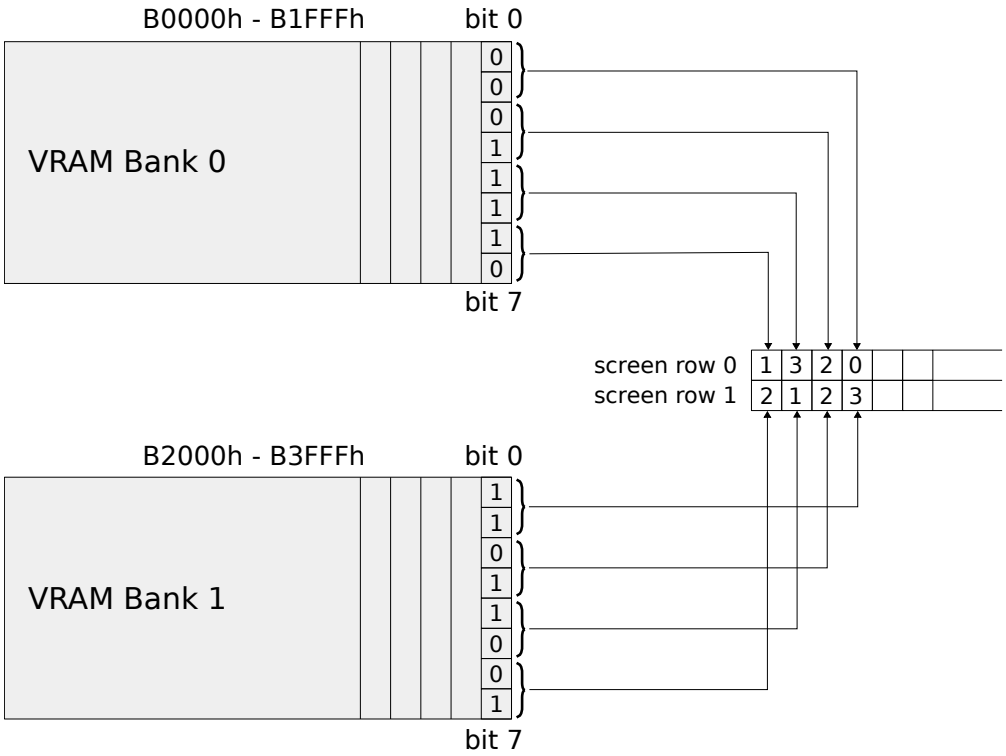


Figure 4.66: CGA interlaced memory.

Trivia : Interestingly, interlacing was never actually implemented in CGA monitors. When displaying VRAM to the screen, the CGA used a progressive (linear) scan, alternately reading from bank 0 and bank 1.

The CGA card employed memory mapping, similar to EGA. In Mode 4, VRAM bank 0 is mapped to memory addresses ranging from `B0000h` to `B1FFFh`, and VRAM bank 1 was mapped to `B2000h` to `B3FFFh`. Unlike EGA, the CGA memory model didn't require masking since the total 16 KiB of VRAM easily fit within a 64 KiB memory segment.

4.16.3 Double buffering

A full display screen in mode 4 requires 320 pixels x 2 bits per pixel x 200 lines, which equals 16,000 bytes of memory. Since the display screen consumes all 16 KiB of available VRAM, there is no capacity for additional screens. The only way to implement double buffering on a CGA card is by creating a 64 KiB buffer in RAM memory.

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

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

This memory buffer contains both the video buffer page and a master page. The buffer page starts at offset 0000h, while the master page begins at 8000h. Both pages float within the 64 KiB memory segment, leveraging the same memory wrapping mechanism described in section "Wrap around the EGA Memory" on page 129.

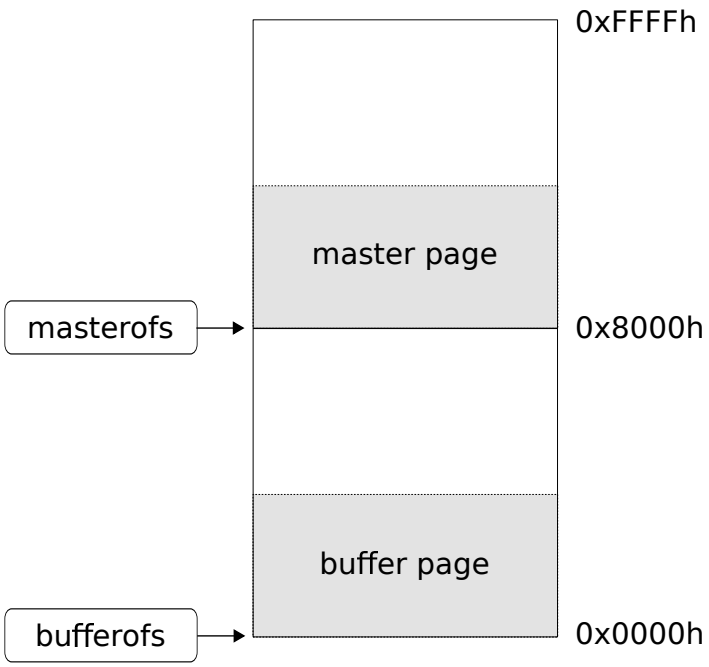


Figure 4.67: CGA double buffering memory layout.

4.16.4 Screen refresh

With double buffering in place, the same algorithm used for EGA can be applied. The final step of the algorithm involves updating the screen display by copying the buffer page to

VRAM and perform fine pixel adjustment. However, there are two complications with CGA.

The first complication is that the CGA card does not support pixel panning. As a result, the smoothest scrolling achievable is in increments of one byte. Since one byte represents four pixels, horizontal scrolling is limited to steps of four pixels, resulting in a more choppy scrolling experience compared to EGA.

```
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 is 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.

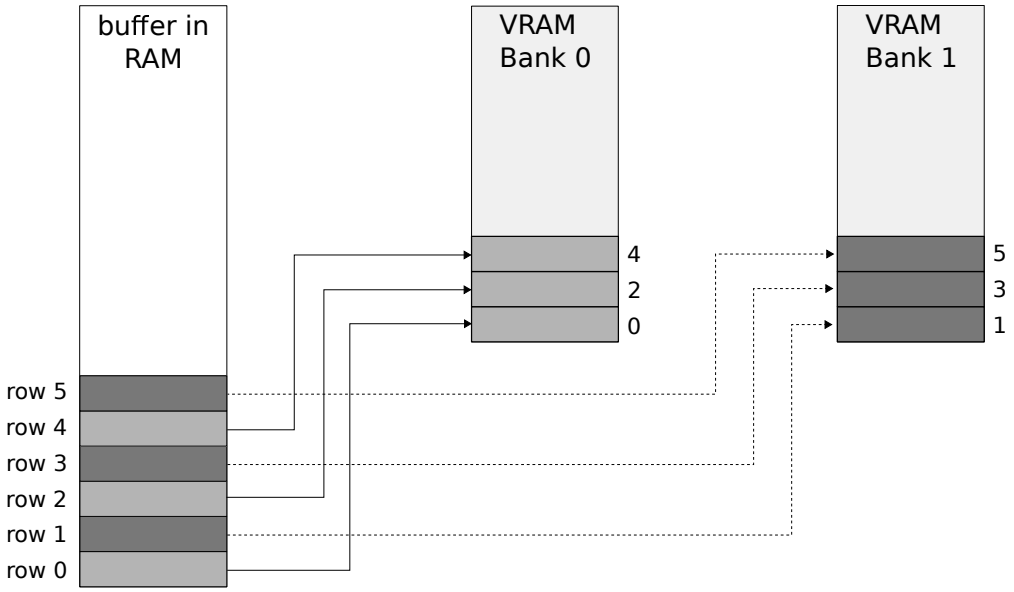


Figure 4.68: CGA memory to VRAM copy.

To avoid screen tearing, the system must wait for a vertical retrace, just as it does with EGA. However, the 286 CPU simply lacked the speed to copy all the necessary bytes from the RAM buffer to VRAM within the vertical retrace period. Consequently, in the CGA version of Commander Keen, screen tearing was unavoidable.

```
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

    [...]
}
```

Appendices

Appendix A

Unboxing the asset files

Commander Keen Dreams contains three asset files; The level maps, graphical assets and sound assets. The first table is the level map file, which contains 18 levels. Four of these levels, all starting with "Temp", are not used in the game.

Level	Title	width x height (tiles)	Size (bytes)
0	Land of Tuberia	85 x 66	33,660
1	Horse Radish Hill	136 x 37	30,192
2	Melon Mines	94 x 93	52,452
3	Bridge Bottoms	65 x 62	24,180
4	Rhubarb Rapids	74 x 46	20,424
5	Parsnip Pass	42 x 72	18,144
6	Temp1	26 x 30	4,680
7	Spud City	202 x 59	71,508
8	Temp8	26 x 30	4,680
9	Apple Acres	126 x 47	35,532
10	Grape Grove	139 x 39	32,526
11	Temp2	26 x 30	4,680
12	Brus.Sprout Bay	93 x 29	16,182
13	Temp13	26 x 30	4,680
14	Squash Swamp	74 x 29	12,876
15	Boobus' Room	35 x 30	6,300
16	Castle Tuberia	85 x 80	40,800
20	Title Screen	37 x 19	4,218
Total			**417,714**

Figure A.1: KDREAMS.MAP level map details. Each tile contains 3 planes x 2 = 6 bytes.

The second file contains all graphical assets. The majority of the file is existing out of background tiles (TILE16), foreground tiles (TILE16M) and sprites.

Asset type	quantity	unit size (bytes)	Size (bytes)
TILE16	643	128	82,304
TILE16M	542	160	86,720
NUMTILE8	72	32	2,304
NUMTILE8M	36	40	1,440
FONT	1	1900	1900
PIC	65	256-298	29,632
PICM	2	320 and 480	800
SPRITE	297	10-6180	148,975
Total			**354,075**

Figure A.2: KDREAMS.EGA graphic asset details.

The final asset file contains sounds for the PC speaker and Adlib soundcard.

Sound source	number of samples	Size (bytes)
PC Speaker	28	1,923
AdLib	28	2,649
Total		**4,572**

Figure A.3: KDREAMS.AUD audio asset details.

Appendix B

x86 Memory Models

The x86 memory models are a set of six different memory layouts for the x86 CPU operating in real mode, which control how the segment registers are used and the default size of pointers. This appendix summarizes each model.

The "Tiny model" is, as you might guess, the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64KiB for all of your code, data, and stack. Near pointers are always used. This model is used for `.com` applications, ensuring backwards compatibility with the CP/M operating system.

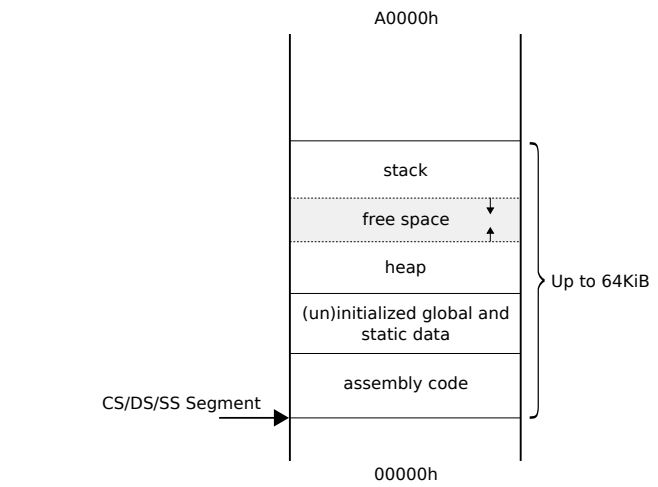


Figure B.1: Tiny memory model, total of 64 KiB.

The next model is the "small memory model". The code and data segments are different and don't overlap, so you have 64KiB of code and 64KiB of data and stack. Near pointers are always used. This is a good size for average applications.

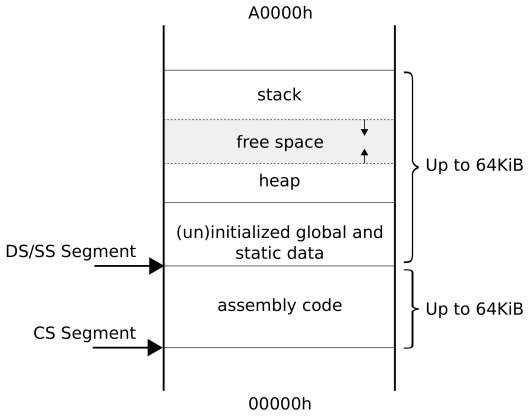


Figure B.2: Small memory model, code and data each have 64KiB.

The "medium memory model" is ideal for programs with a large amount of code but minimal data. Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 MiB.

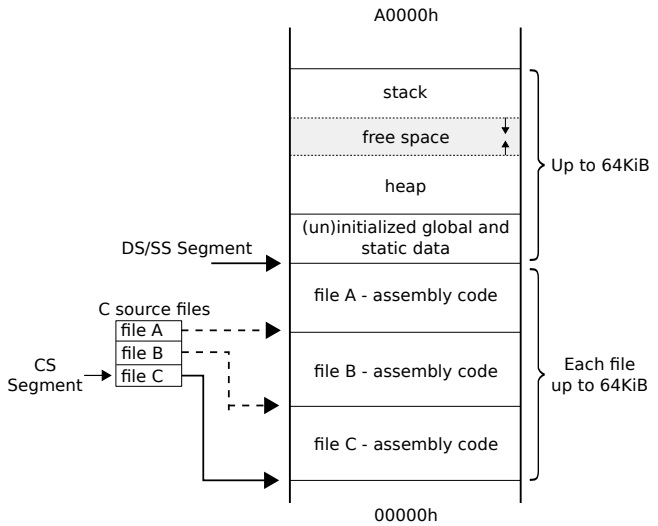


Figure B.3: Medium memory model, code can be larger than 64KiB.

The opposite of the medium model is the "compact memory model". Here the total function code cannot exceed 64KiB, but there is more space for data. This model is best if code is small but needs to address a lot of data.

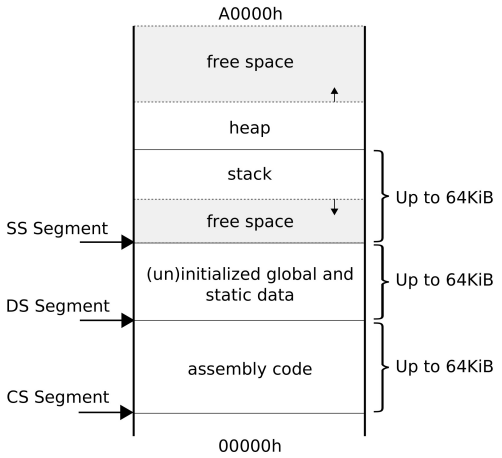


Figure B.4: Compact memory model, data can be larger than 64KiB.

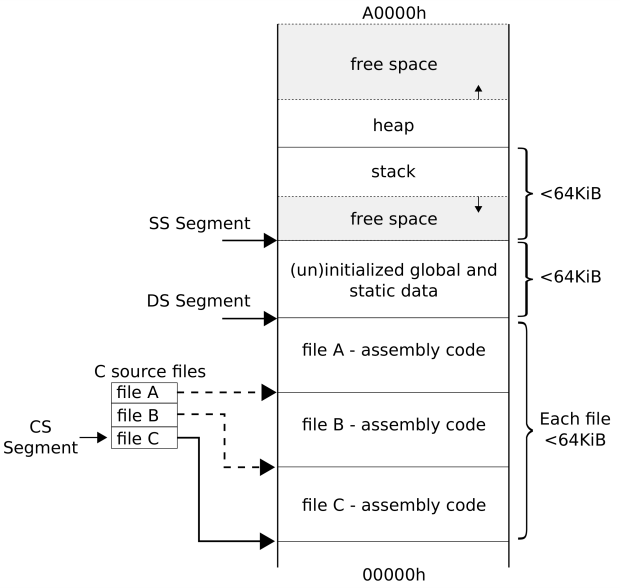


Figure B.5: Large memory model, code and data can be larger than 64KiB.

The "large memory model" can deal with function code and data larger than 64KiB. Far pointers are used for both code and data, giving both a 1 MiB range.

In all models so far, Borland C++ limits the size of global data to 64KiB. The "huge memory model" sets aside that limit, allowing global data to occupy more than 64KiB using multiple data files. If the source file is too big to fit into one 64KiB data segment, the program must be broken into smaller source files and compiled separately.

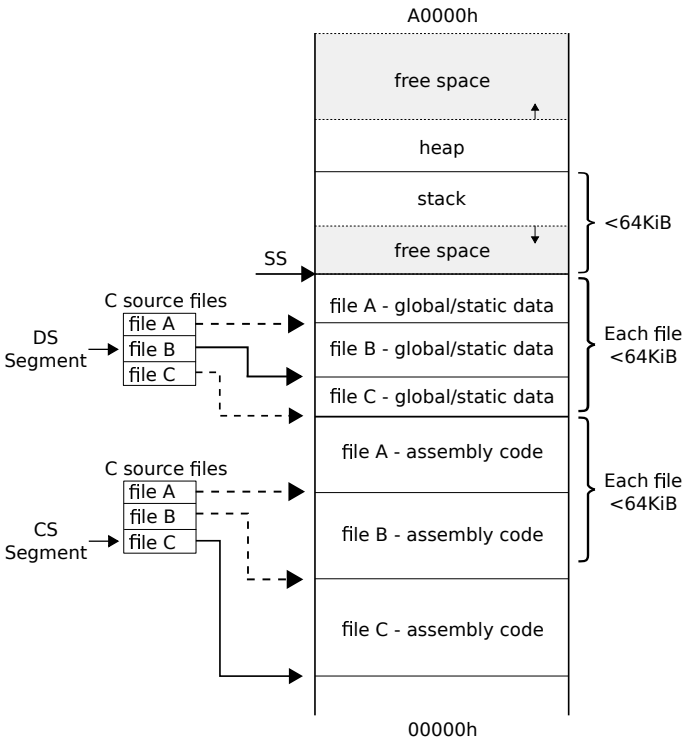


Figure B.6: Huge memory model, code and global data can be larger than 64KiB.

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[1].

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

Appendix C

Dangerous Dave in Copyright Infringement

In September 1990, John Carmack, developed his first version of *Adaptive Tile Refresh-ment*. He discussed the idea with coworker Tom Hall, who encouraged him to demonstrate it by recreating the first level of 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[1].

> “ 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.** ”

[1] Planet Romero: https://web.archive.org/web/20141231073528/http://planetromero.com/games/dangerous-dave-in-copyright-infringement.

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.

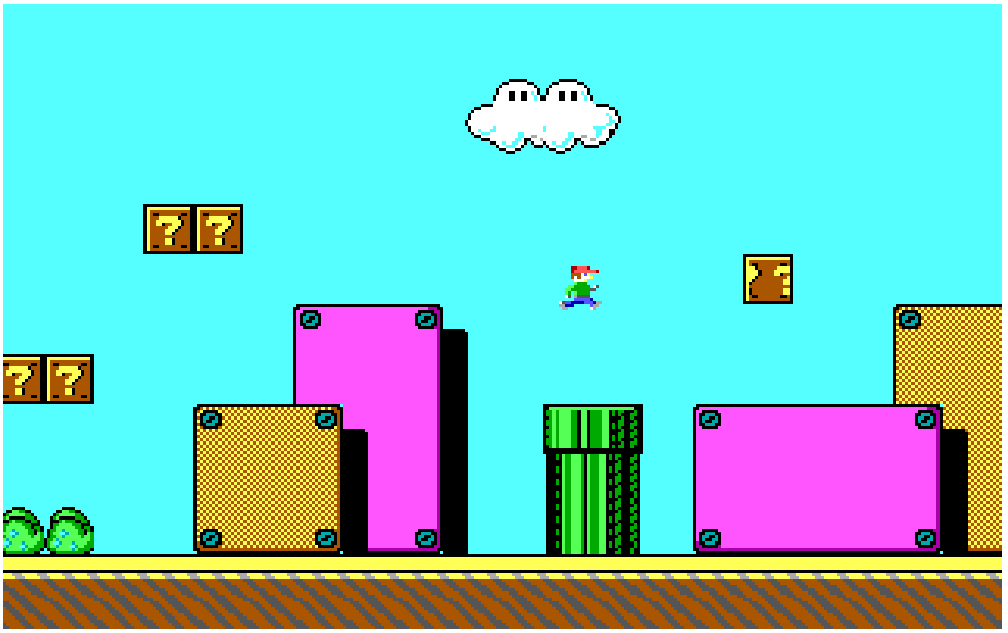


Figure C.1: Dangerous Dave in Copyright Infringement demo

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 D

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 and divided into three episodes: *Marooned on Mars*, *The Earth Explodes*, and *Keen Must Die!*. The game was published as shareware, whereby Marooned on Mars was released for free, and the other two episodes were available for purchase.

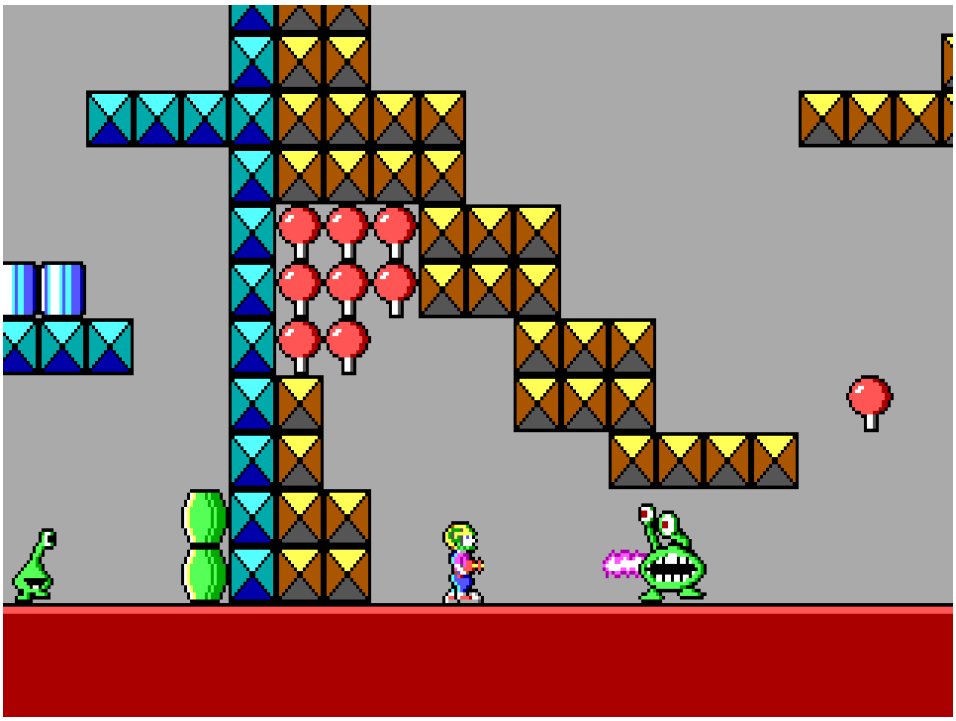
After the arrival of the first royalty check from Apogee, the team planned to quit Softdisk and start their own company. On February 1, 1991, the team founded *id Software* having four owners: John Carmack, John Romero, Tom Hall and artist Adrian Carmack[1].

> "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.**

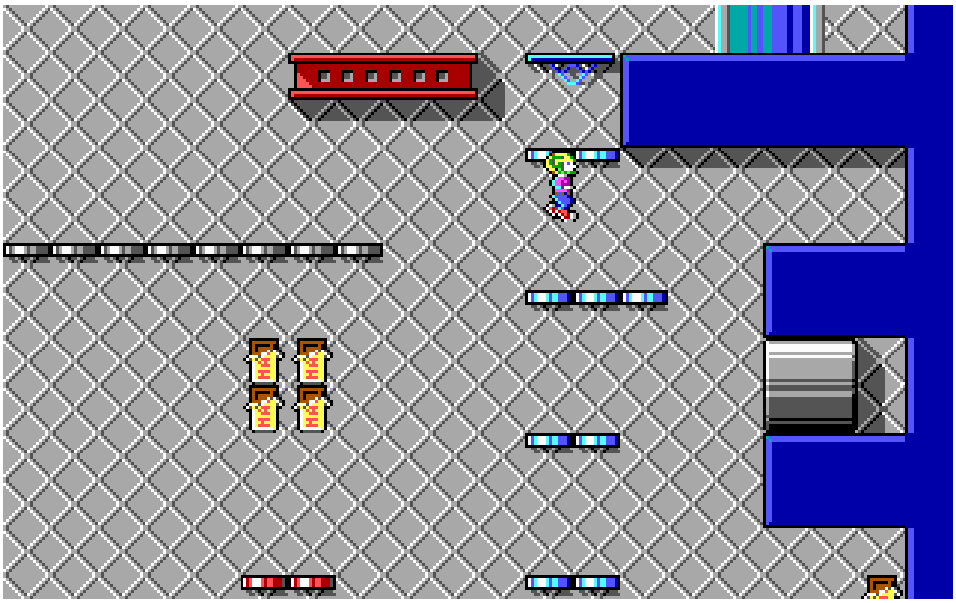
[1] See Masters of Doom, chapter 4

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 Edge, one every two months. One of the games they developed to fulfill their obligation was *Commander Keen in Keen Dreams*, which was released in June 1991.

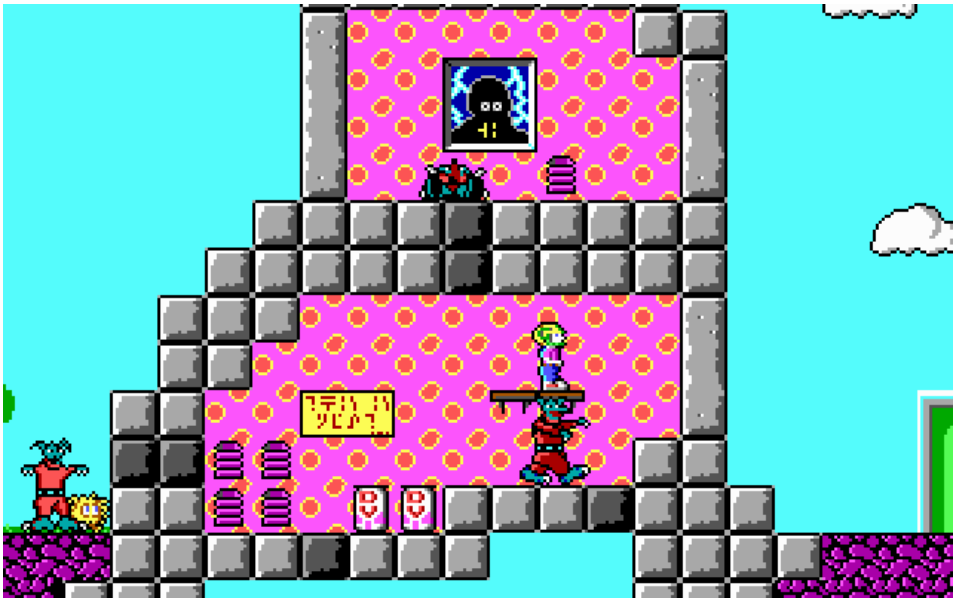
The second main game, *Commander Keen in Goodbye, Galaxy*, was released in December 1991. It consisted of episodes four and five of the series, *Secret of the Oracle* and *The Armageddon Machine*, where *Secret of the Oracle* was released for free, and the other episode sold by Apogee.



Keen I - Marooned on Mars.



Keen II - The Earth Explodes.



Keen III - Keen Must Die!

Figure D.1: Keen IV - Secret of the Oracle.

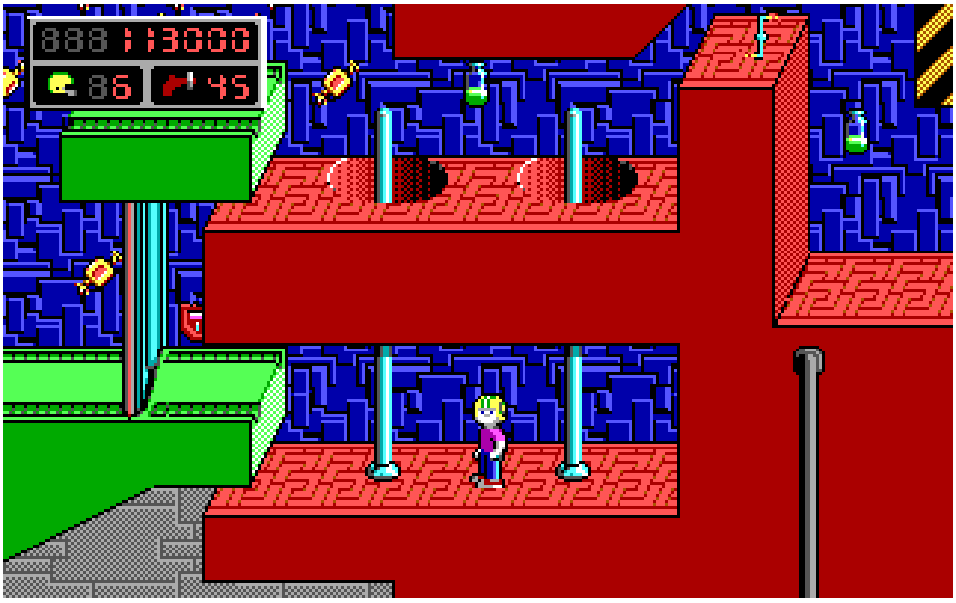


Figure D.2: Keen V - The Armageddon Machine.

The final id Software developed game was *Commander Keen in Aliens Ate My Babysitter*, also released in December 1991. Originally planned to be the third episode of *Goodbye, Galaxy* and sixth episode overall, the team decided to published it as a retail title through FormGen.

Figure D.3: Keen VI - Aliens Ate My Babysitter.

Trivia : Even though this is the last game in the series, this was not the last one to be created. *Commander Keen 5: The Armageddon Machine* was created after *Commander Keen 6: Aliens Ate My Baby Sitter!* because the latter had to go to retail and that required lead time.

Another trilogy of episodes, titled *The Universe Is Toast*, was planned for December 1992; id Software 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**...