GAME ENGINE WHITE PAPERS

COMMANDER KEEN

v1.0 by BAS SMITS

Copyright

To illustrate how the Commander Keen game engine works, several screenshots, images, sprites, and textures—belonging to and copyrighted by id Software—are reproduced in this book.

The following items are used under the “fair use” doctrine from id Software:

1. All in-game screenshots, title screen and retail cover.
2. All in-game sound menu screenshots.
3. All 2D tile textures (background and foreground tiles).
4. All 2D picture assets.
5. All 2D sequence sprites (Commander Keen, BROCCOLASH, CARROT, MELONLIPS, TOMATOOTH).
6. All TED5 editor screenshots.

How To Send Feedback

This book strives for accuracy. If you find mistakes, omissions, or typos, please take the time to report these issues.

Or even better, patch the book with a pull request since the full source code is available online on github.

Report an issue: **https://github.com/bsmits74/Keen_White_Papers/issues**.

Send a pull request: **https://github.com/bsmits74/Keen_White_Papers/pulls**.

Many thanks!

Contents

Prologue

I was 11 years old when I wrote my first lines of software code. It happened on an MSX-1 computer in BASIC, and from that moment a completely new, magical world opened up to me. Five years later I got my first PC, an Intel 80286, and my fascination with computers only grew stronger. I witnessed the rise of the PC as a gaming platform and spent countless hours playing many of its groundbreaking titles: *Prince of Persia*, *Wolfenstein 3D*, *Doom*, *Dune II*, *Command & Conquer*, and many more.

One of the first games I played on the PC was *Commander Keen*. The moment I saw the game running, I was amazed. It was the first time I had seen smooth scrolling on a PC. Commander Keen was also the first major title from id Software. Thanks to the financial success of Commander Keen, the team—then still employed by Softdisk—decided to start their own game development company. The same studio would later go on to release the groundbreaking games *Wolfenstein 3D* and *Doom*.

Fast forward to 2021, I discovered Fabien Sanglard's website and began reading his Game Engine Black Books on Wolfenstein 3D and Doom. Inspired by those works, I wondered whether I could do something similar for Commander Keen: open up the source code, explore the files, and piece together a picture of the overall architecture and the clever tricks used. The style, dimensions, and structure of this book are intentionally similar to Fabien's Game Engine Black Books, as an homage to those masterpieces. To give it a personal twist, I inverted the title and cover to white.

The entire experience was immensely rewarding. It felt like returning to my early teenage years, debugging and experimenting with C and assembly code on an MS-DOS computer. I learned a great deal about the hardware of that era and the challenges developers faced in getting everything to work together.

You might ask why I "wasted" my time on a game and hardware that are now more than 35 years old. Well, even though we now have vastly more transistors on a chip and have gone from megahertz to gigahertz CPUs, from kilobytes to gigabytes of RAM, and from simple video cards to powerful GPUs, the underlying architecture remains surprisingly similar to what it was three decades ago.

The result of my journey is in this book—a mix of engineering, history, and nostalgia. It is written by a Dutch guy, so please forgive my imperfect English (ChatGPT was a great help in reviewing my grammar and style). To compensate for my limitations in prose, I've included many detailed drawings to illustrate the hardware and software techniques discussed. All source code in this book is based on publicly available material. I have not altered the code, except for formatting and minor edits to improve clarity.

So turn this page, sit back, and let's travel together to the early '90s.

I hope you enjoy reading it.

– Bas Smits

Helmond, the Netherlands
March, 2026

Chapter 1

Introduction

My personal introduction to computer gaming began in 1985, when my parents bought an MSX-1 home computer. I was fascinated by games such as Konami's *Knightmare* and *Nemesis 2*. It was not only the gameplay that intrigued me, but also how such games were created. This curiosity sparked my interest in programming and game development, and I soon began creating my first games in MSX-BASIC.

Around the same time I was introduced to the MSX, Nintendo released *Super Mario Bros.* for the Nintendo Entertainment System (NES). It became an instant blockbuster, combining colorful graphics with remarkably smooth side-scrolling. By comparison, the side-scrolling games I played on the MSX-1, such as Knightmare and Nemesis 2, moved at a constant, noticeably choppy speed.

Super Mario Bros. was different. The player dictated the scrolling speed: you could accelerate from walking to running or jumping, and the screen would smoothly follow your actions. The game was immensely successful, both commercially and critically. It helped popularize the side-scrolling platform genre and served as a killer app for the NES[1]. More importantly, it demonstrated the true power of the NES by utilizing a dedicated Picture Processing Unit (PPU) that handled sprites and scrolling independently of the CPU.

My MSX-1 home computer lacked any hardware support for smooth scrolling. Other computers, like the MSX-2 and Commodore 64, could scroll up to 8 pixels (one character) horizontally or vertically. However, once that limit was reached, developers had to rely on all kinds of tricks to achieve continuous, full-screen smooth scrolling. This typically involved copying large amounts of data, simply to move the screen by a single character. Such approaches were extremely CPU-intensive and often beyond the capabilities of the hardware.

[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.

Trivia : Some great programmers succeeded in implementing smooth scrolling on the Commodore 64 and MSX-2, most notably in games such as *Uridium* (1986) and *Armalyte* (1988) on the Commodore 64, and *Space Manbow* (1989) on the MSX-2.

The NES was one of the very first home consoles to offer true hardware-supported smooth scrolling. Essentially, the PPU had registers that could be written to, in order to set the fine (pixel) scroll. For example, to scroll the background 120 pixels horizontally and then 22 pixels vertically, you would first write the horizontal scroll value 120, followed by the vertical value 22 to the PPU[2]. Done deal! The video chip takes care of the rest, running at the same constant speed as it always has.

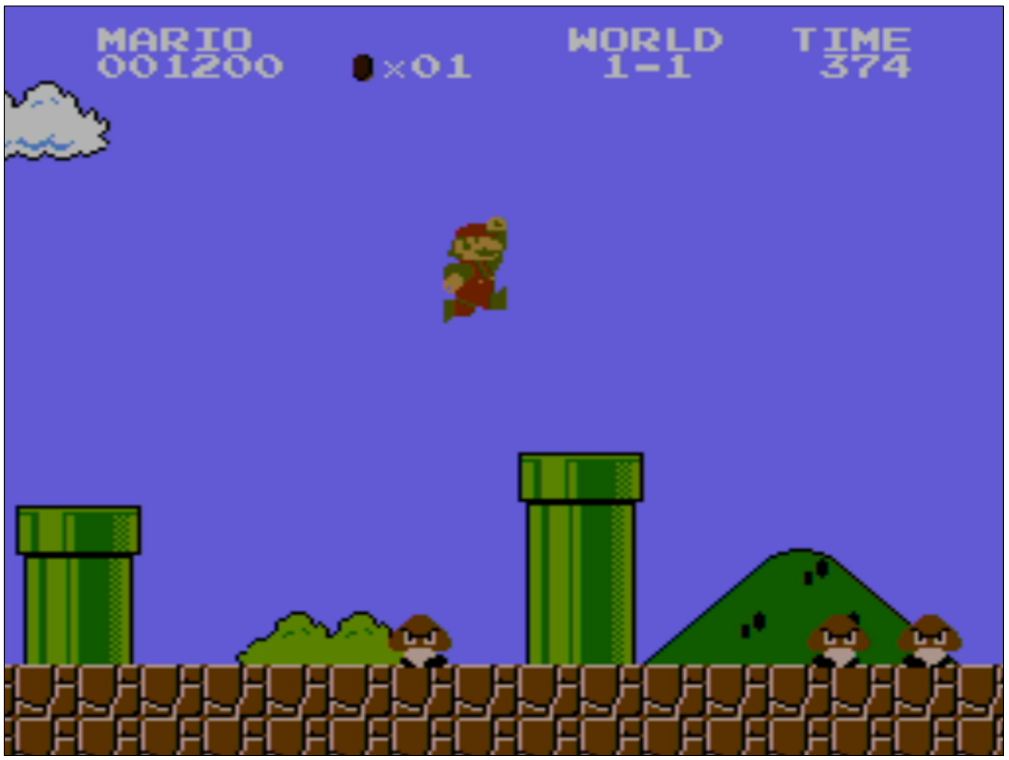


Figure 1.1: Super Mario Bros on Nintendo Entertainment System

In the late 1980s, the IBM PC lagged far behind the gaming power of the NES. It was designed for office work rather than gaming. It excelled at word processing and crunching numbers in spreadsheets. Although the PC game market was growing rapidly, it was

[2]The PPU could only handle one-directional scrolling at a time. So bi-directional scrolling requires twice updating the PPU registers.

dominated by adventure games (*King's Quest*), role-playing games (*Ultima series*), strategy titles, and simulations (*SimCity*). Action and platform games remained largely the domain of consoles and home computers, as the PC lacked 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 PC. It was the first PC game to feature smooth side-scrolling, similar to Super Mario Bros on the NES.

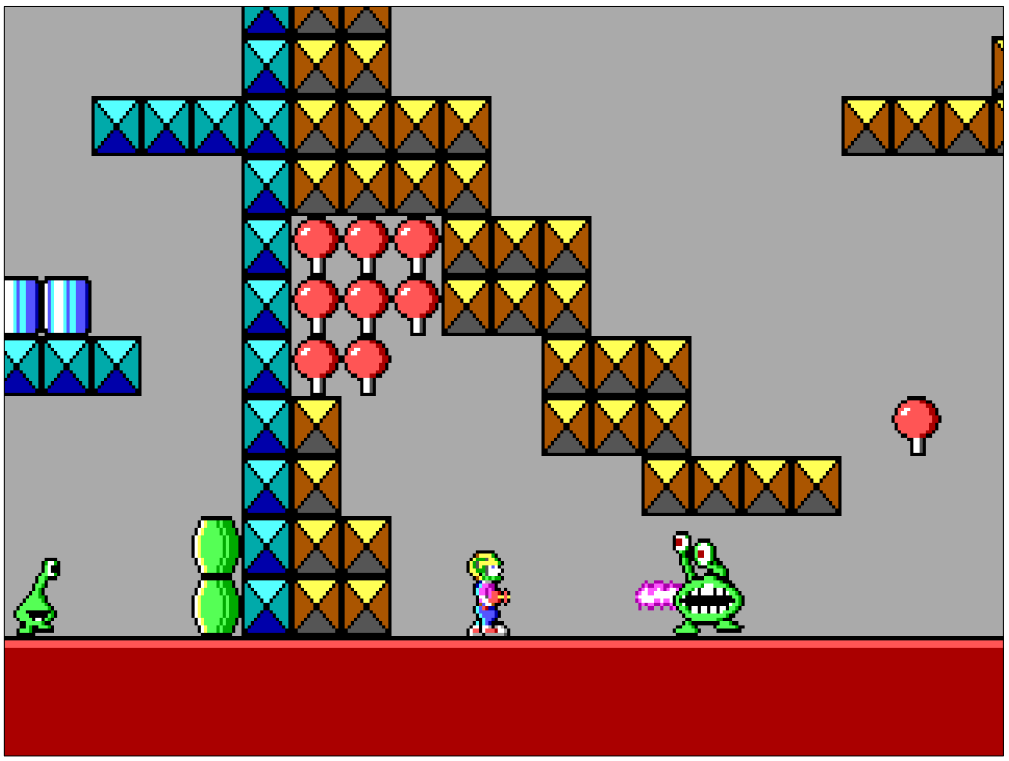


Figure 1.2: Commander Keen in Invasion of the Vorticons

So how was it possible to create a game like Commander Keen on the IBM PC?

With the introduction of the IBM AT, powered by the Intel 80286, the PC began to outperform consoles and home computers on the market. In terms of raw processing power, not even the NES could compete with the IBM AT.

But raw power was about the only area in which the PC excelled. Its other components

were notoriously difficult to master:

- The video adapter (called EGA) did not support hardware scrolling. It also lacked hardware sprites, which would have allowed objects to move on the screen simply by updating their `(x,y)` coordinates.
- The PC speaker could produce little more than a series of (mostly) annoying "beeps". Sound cards did exist, but they were still in their infancy and far from standardized.
- Memory available to applications was 640KiB, much more than an average home computer, which typically had 64KiB. However, RAM was not organized as a flat address space but accessed through segments and offsets. Different memory models existed, each involving trade-offs between program size and performance.

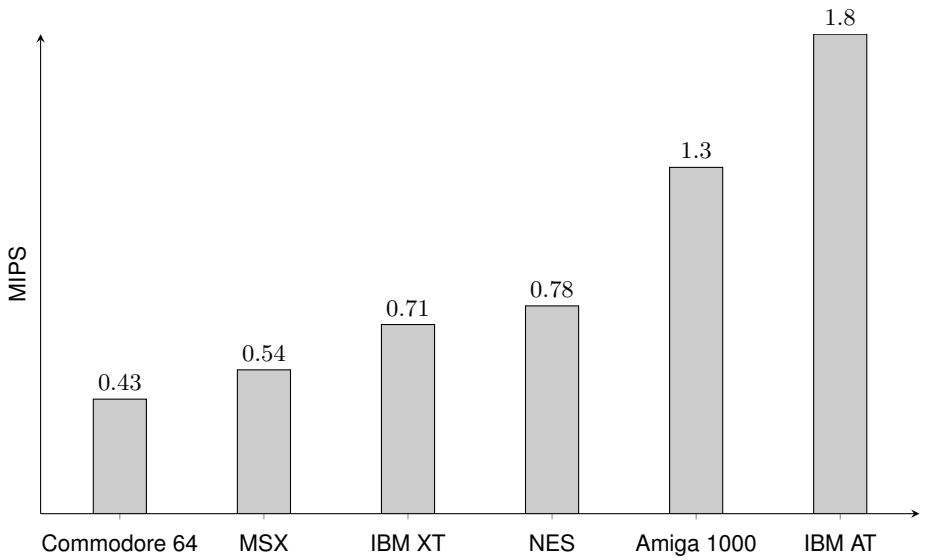


Figure 1.3: Consoles[3]and IBM PC CPU comparison (MIPS)[4,5].

Overall, creating a fast side-scrolling action game on the PC seemed nearly impossible. Nevertheless, programmers around the world embraced the challenge, exploring the depths of the hardware and produced remarkable games. Their ingenuity is what inspired me to write this book.

[3]The Commodore 64 uses a MOS 6510 CPU running at 1MHz. The MSX uses a Zilog Z80 running at 3.6MHz. The Amiga 1000 has a Motorola 68000 CPU running at 7.16 MHz. The NES uses a Ricoh 2A03 CPU running at 1.8 MHz.

[4]Million Instructions Per Second.

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

This book is divided into three chapters:

- Chapter 2: The Hardware. An exploration of the main PC components from 1990 and the challenges they presented.
- Chapter 3: The tools and assets. Which tools are used for game development and how assets are created and structured.
- Chapter 4: The Software. A deep dive into the Commander Keen game engine and how the team bent the hardware to their will. Commander Keen targeted the 16-color EGA graphics card, but a CGA version was also released, which is discussed in this chapter as well.

This book focuses on *Commander Keen in Keen Dreams*, which was 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

The original IBM PC 5150 was released in September 1981 and marked a turning point in personal computing. Unlike many earlier systems, it was intentionally designed with an open architecture and built using widely available, off-the-shelf components rather than proprietary hardware developed exclusively by IBM. This strategic decision significantly reduced development time and helped make the system more affordable.

The system included internal expansion slots that allowed third-party manufacturers to develop compatible hardware such as graphics adapters, memory expansion cards, sound cards and (hard)disk controllers. This openness encouraged a rapidly growing ecosystem of compatible components and clone systems, which played a major role in establishing the IBM-compatible PC as an industry standard.

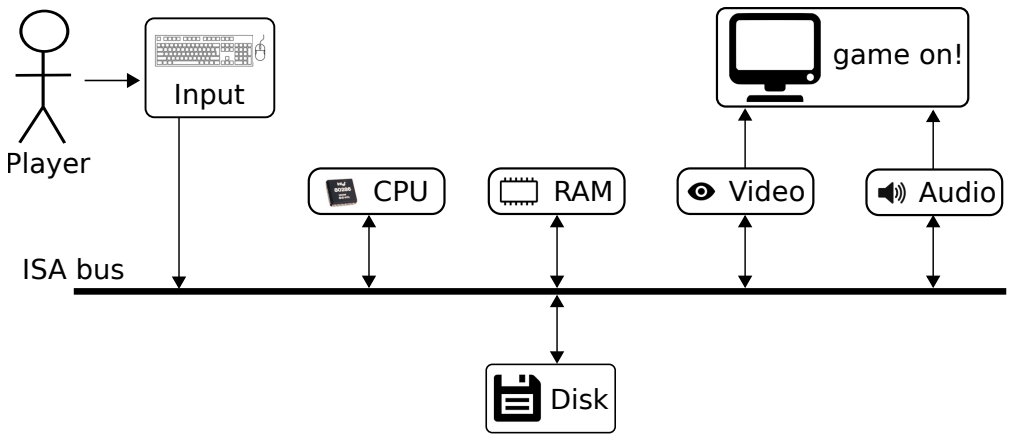


Figure 2.1: IBM PC subsystems.

The IBM PC can be divided into six essential subsystems that determine the overall system performance and gaming experience:

- CPU
- RAM
- Video
- Disk
- Audio
- Input devices

Each of these subsystems is described in this chapter.

2.1 CPU: Central Processing Unit

In the late 1980s, Intel's x86 architecture dominated the IBM-compatible PC market, while Motorola's 68000 series powered systems such as the Apple Macintosh, the Commodore Amiga, and workstations from Sun Microsystems. By 1989, approximately 15% of US households owned a personal computer[1].

2.1.1 Introduction of Intel x86 CPU

Intel released the 8086 in 1978, 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. IBM chose the 8088 over the 8086 for its PC/XT, because Intel offered a better price for the former and could supply more units.

In 1982 Intel released the 80286 microchip, simply called the "286". A typical 8088 chip was running at 4.77MHz, where the 80286 was running at 6MHz and later popular 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[2]. It would be the last, fastest 16-bit PC processor Intel made. Its successor, the Intel 80386, was a true 32-bit processor with a 32-bit data bus.

Although Commander Keen was 100% compatible with the IBM PC, it was hardly playable on an 8088 CPU. A 286 or higher was recommended to play the game primarily due to

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

[2] By the end of 1988, Intel estimates there were around 15 million 286-based PCs in use worldwide.

its reliance on the 16-bit data bus, as well as its raw speed. This section (and the bulk of this book) is written from the perspective of a 286 CPU, but most of the information is applicable to the 8086, as well as PCs many generations newer.

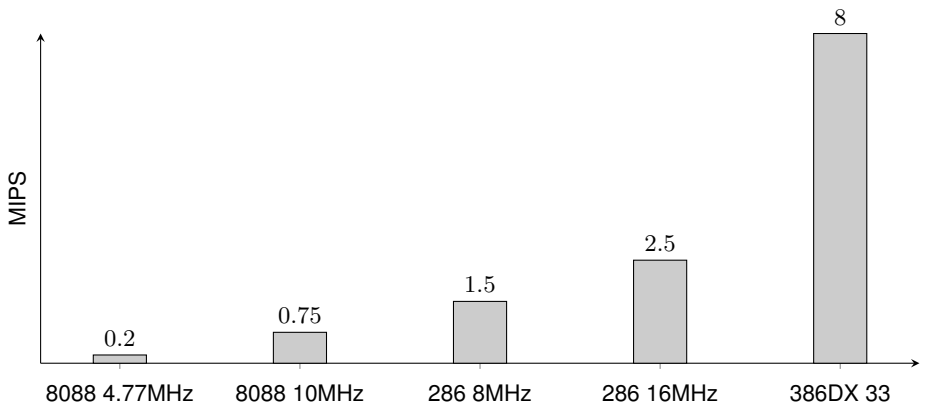


Figure 2.2: Comparison[3] of Intel CPUs in MIPS

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 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 main reason the 286 computer was faster than its predecessor is that it executes instructions much more efficiently. An average instruction takes 12 clock cycles on the 8088,

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

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 CPU has two operating modes: real mode and protected mode. These modes are sufficiently distinct that the 286 can be regarded as two processors in one. In real mode, the 286 behaves essentially like an 8086 and is fully compatible with the 8088. In protected mode, however, the 286 introduced significant new capabilities. Programs designed to take advantage of this mode can access up to 16 MB of physical memory through its 24-bit address bus.

A major limitation of the 286 is that it cannot switch from protected mode back 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 80286 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 an 8088 chip.

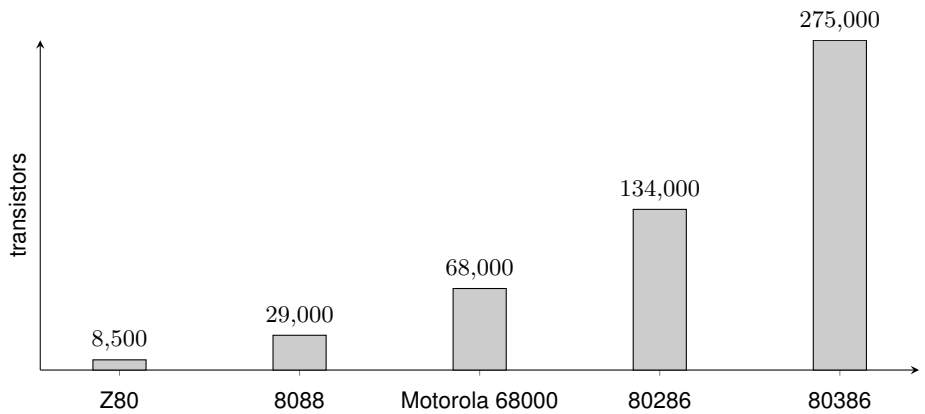
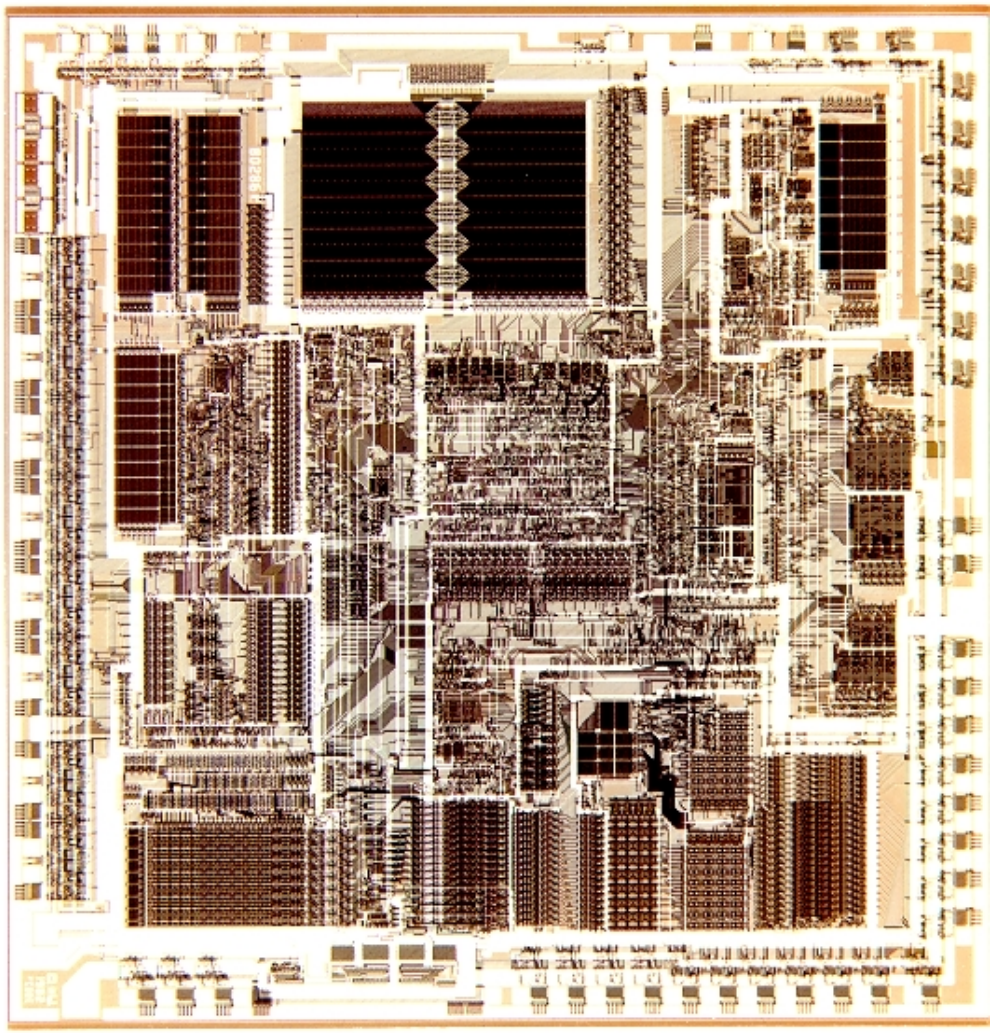


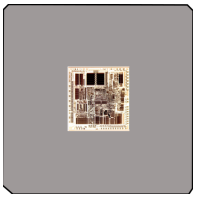
Figure 2.3: Comparison of CPUs with # of transistors.

Trivia : Fast forward to today, the AMD Ryzen 9 7950X contains over 13 billion transistors.

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

If you are holding a physical 9.25"×7.5" copy of this book, the CPU packaging is 25×25 mm square and the die is 7×7 mm, at 1:1 scale.





To better understand why the 286 is much faster than the 8088, we need to understand the architecture of the chip. The 286 architecture can be summarized by four functional units as illustrated below.

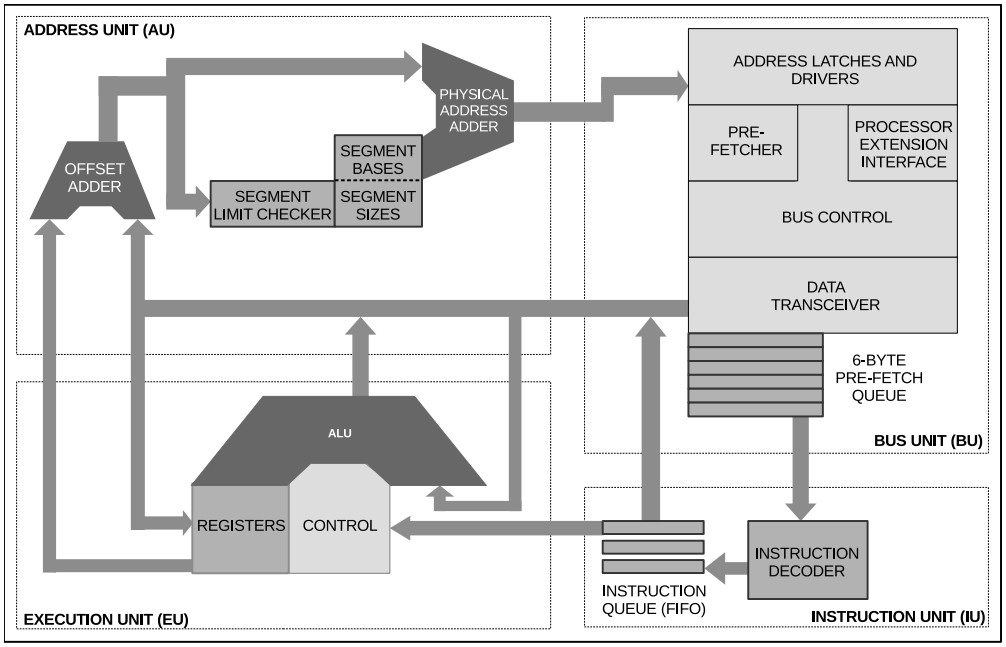


Figure 2.4: Internal block diagram of the 80286 processor

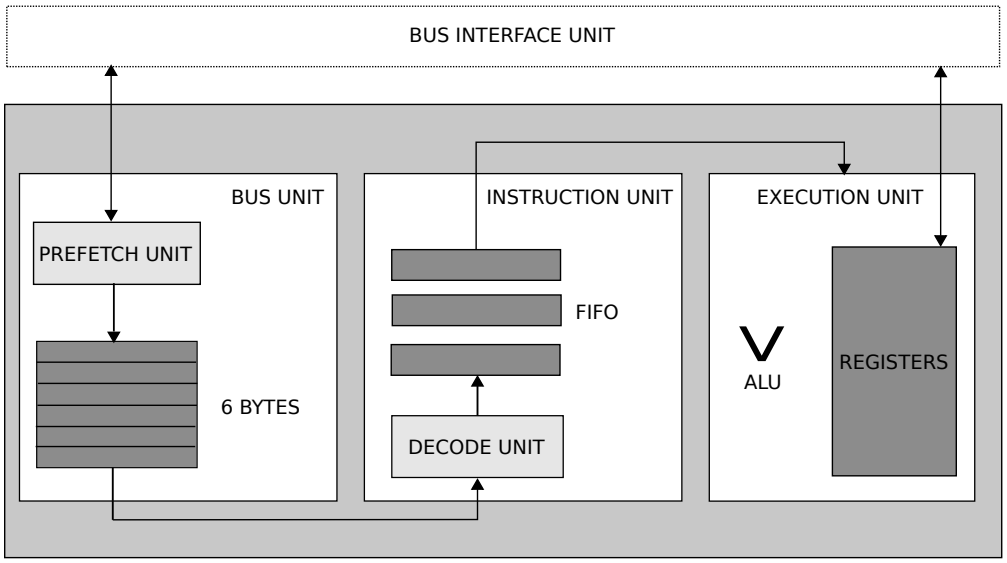
The four functional units can be described as follows:

- **Address Unit (AU)** is used to determine the physical addresses of instructions and operands which are stored in memory. The address lines derived by the AU can be used to address different peripheral devices such as memory and I/O devices.
- **Bus Unit (BU)** interfaces the CPU with memory and I/O devices. The BU is used to fetch instruction bytes from the memory and store them in the prefetch queue.
- **Instruction Unit (IU)** 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 EU consists of the register bank, arithmetic and logic unit (ALU) and control block. The ALU is the core of the EU and performs all the arithmetic and logical operations.

The 8088 had a two-unit architecture consisting of the Execution Unit and the Bus Interface Unit. The introduction of four independent units operating in parallel was one of the primary reasons why the 286 was significantly faster than the 8088.

A good example is the calculation of physical memory addresses. In the 286, address calculations are performed by the dedicated address unit, whereas the 8088 computes effective addresses using the ALU, often consuming several additional clock cycles.

The remaining three units form a three-stage pipeline. When the execution unit is not using the system bus, the bus unit fetches instructions and stores them in a 6-byte prefetch queue. The decode unit reads bytes from this queue, translates them into decoded instructions, and places them into a secondary three-instruction queue. Finally, the execution unit retrieves these decoded instructions and executes them.



In practical terms, a 286 system running at the same clock frequency as an 8088 will typically be three to six times faster due to its pipelined architecture and dedicated functional units[5]. Still, it was important to understand the cost of each assembly instruction. Figure 2.5 shows the average instruction costs for the 286. Multiplication and division operations require significantly more cycles than most other instructions. For applications where every cycle counts, such as games, minimizing the use of multiplication and division during runtime was critical.

[5] https://www.tomshardware.com/reviews/processor-cpu-apu-specifications-upgrade,3566-6.html.

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 2.5: 286 instruction costs[6].

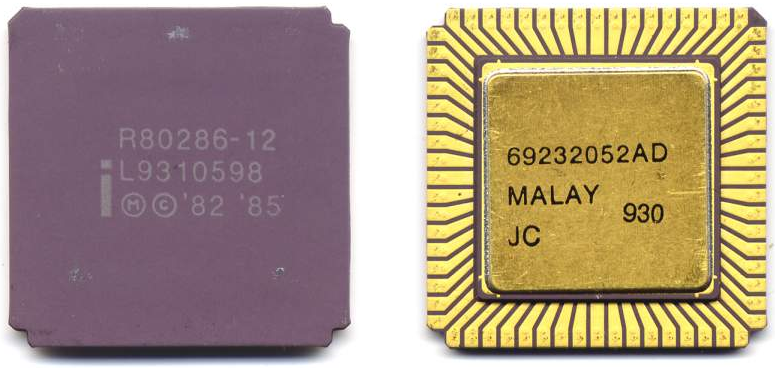


Figure 2.6: The Intel 286, 7mm by 7mm packing 134,000 transistors.

For programming purposes, the 286 CPU can be summarized as follows:

- Arithmetic Logic Unit performing `add`, `sub`, `mul`, etc.
- 14 registers, all 16-bits wide:
  - General Purpose Registers: AX, BX, CX, DX
  - Index Registers: SI, DI, BP, SP
  - Segment Registers: CS, DS, ES, SS
  - Status and Control Register
  - Instruction Counter: IP
- A 24-bit address bus for up to 16MiB of addressable RAM
- Memory Management Unit (MMU), used in Protected Mode

[6] Intel 80286 programmer's reference manual - 1987.

2.2 RAM

When the first IBM PC was introduced in 1981, it came with a base memory of 16KiB or 64KiB and could be expanded up to 256KiB[7]. With the introduction of the 8088 CPU and the IBM XT, the maximum memory increased to 640KiB. This was a large amount of memory at the time, considering that the average home computer, such as the Commodore 64 or MSX-1 system, contained a maximum of 64KiB. Later, with the introduction of the 80286, one could, in theory, address up to 16MiB of memory. In practice, however, this was never fully utilized due to the constraints of the DOS operating system.

2.2.1 x86 Real and Protected Mode

With the 16-bit architecture of the 8088, only 64KiB could be addressed using a single 16-bit address register. To address more memory, Intel introduced the real mode architecture. It combined two 16-bit address values into a 20-bit physical address, thereby providing access to up to 1MiB of RAM. There was no memory protection and direct access to all hardware, meaning that applications could (un)intentionally overwrite each other's memory.

As memory requirements increased, there was a need for an improved CPU architecture supporting larger address spaces and memory access protection. With the introduction of the 80286, protected mode was added to the architecture. It provided 24-bit memory addressing, allowing up to 16MiB of memory, along with memory protection through a Memory Management Unit that prevented applications from accessing or overwriting memory belonging to other programs.

To preserve compatibility with the vast library of existing software written for the 8086, the 80286 retained real mode. In fact, every time the CPU powered on, it started in real mode and could then switch to protected mode. This design decision ensured that older software would continue to run unchanged, a critical factor in IBM PC's commercial success.

Trivia : Even in 2026, a modern Intel Core i9 still begins its boot process in 16-bit real mode. Beneath layers of gigahertz clocks and multi-core designs, the ghost of the 8088 still stirs at startup—a quiet reminder of where it all began.

Protected mode could not run existing MS-DOS software. To use it, a completely new operating system had to be rewritten, and there was little incentive for developers to do this when they could target the 8086, which held the largest market share. As a result, developers using DOS continued programming within the limitations of the 8088 architecture, where only 1MiB of RAM could be addressed without memory protection.

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

To make matters worse, only the first 640KiB of the 1MiB address space was available for applications. This area, also called "Conventional Memory", included system-specific data, the DOS operating system, as well as drivers needed to run the system (e.g., a mouse driver), which effectively left ~620KiB of usable memory for applications. The remaining 384KiB, known as the "Upper Memory Area (UMA)", was reserved for video memory, BIOS ROM, and memory-mapped I/O.

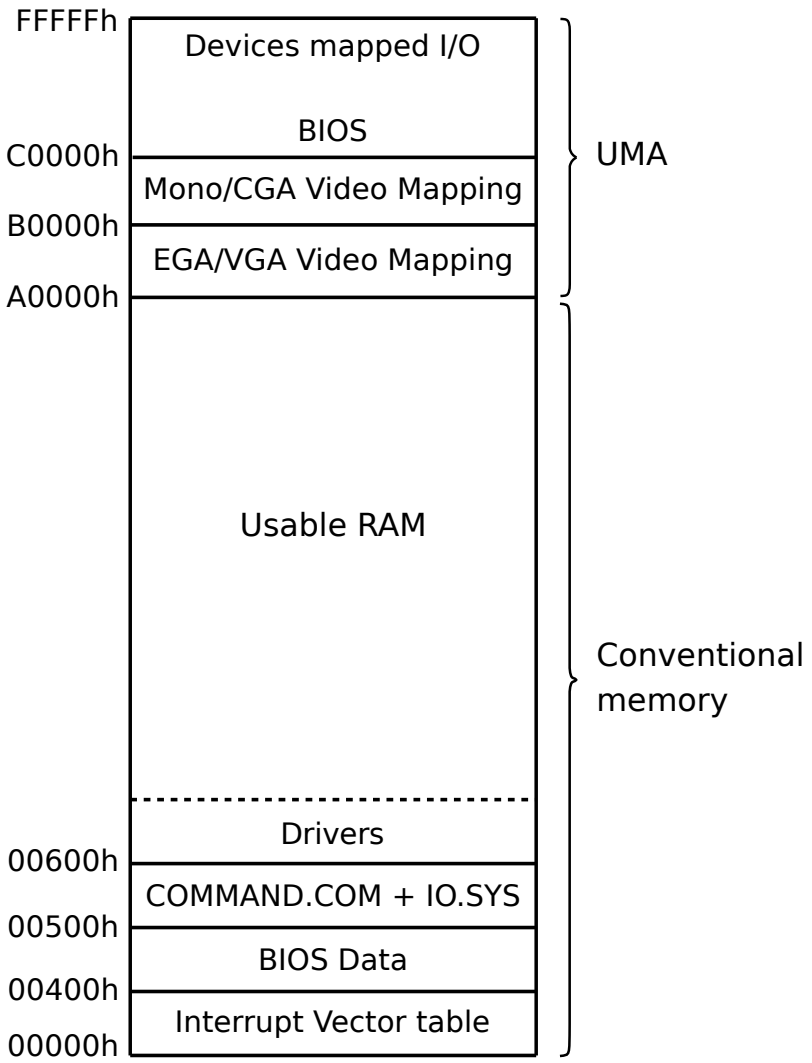


Figure 2.7: First 1MiB of RAM layout.

2.2.2 Memory addressing

If Intel had allowed the CPU to combine two registers into a high and low pair of 32-bits, it could have referenced up to 4GiB of memory in a linear fashion. Keep in mind, however, this was at a time when many never dreamed we would need a PC with more than 640KiB of memory for user applications and data.

> “ I've said some stupid things and some wrong things, but not "640K is enough". No one involved in computers would ever say that a certain amount of memory is enough for all time.
>
> **Bill Gates - Founder of Microsoft** ”

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 1MiB[8] of memory. The segment:offset scheme combines two 16-bit registers, one designating a segment and the other an offset within that segment.

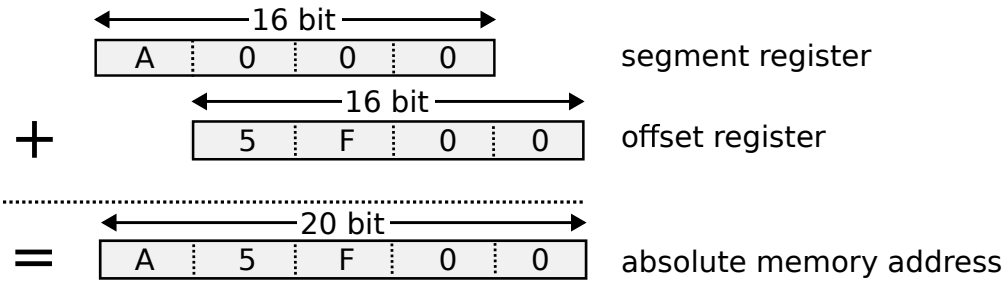


Figure 2.8: How registers are combined to address memory.

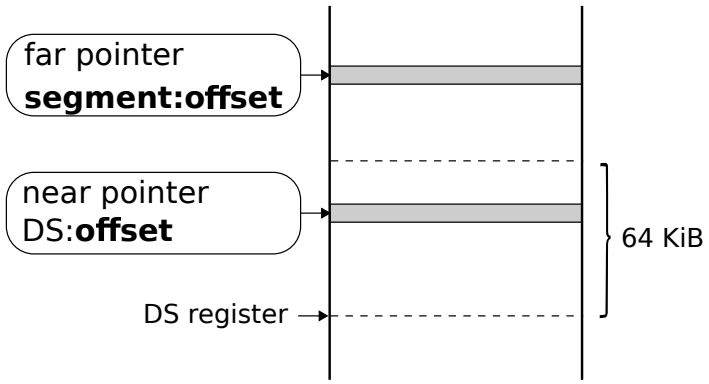
To support this architecture, four segment registers are introduced. Each of these segment registers has its own purpose:

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

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

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 code or data segment. It is 16 bits long and contains an offset into the current DS data segment if it is a data pointer, or into the current CS code segment if it is a function pointer. A far pointer can refer to a function or data that is in a different segment from 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 linear) 20-bit memory location and the 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 it was really needed.

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, aligned with the segment registers:

- **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[9] exists, 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

[9] 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.

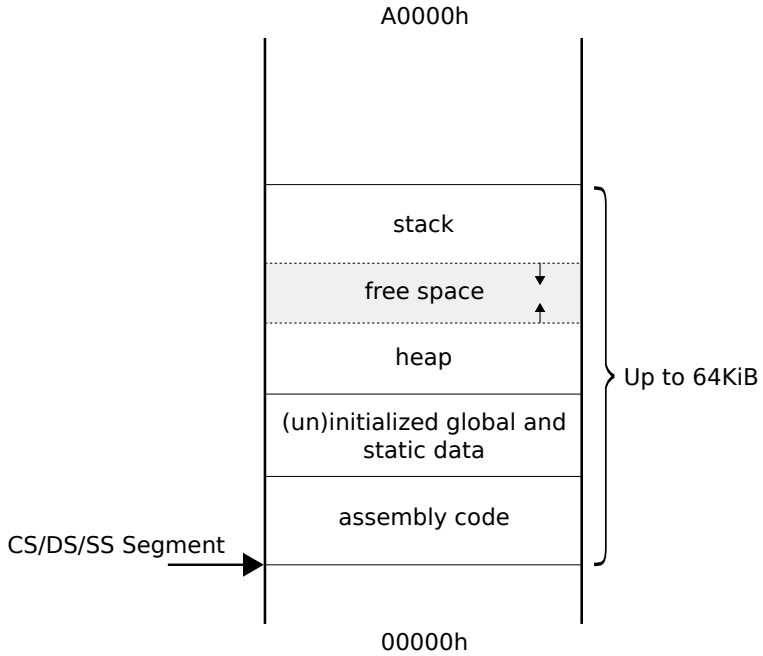


Figure 2.9: Tiny memory model layout.

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, like 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.

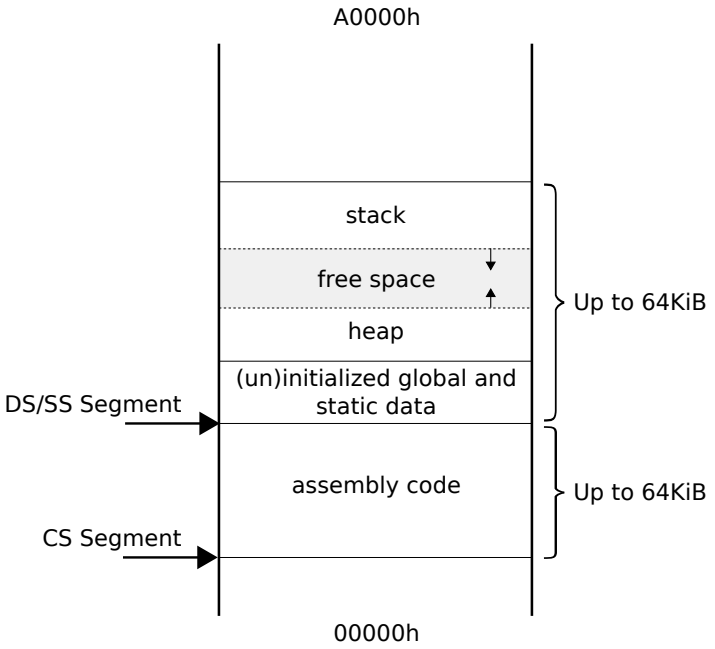


Figure 2.10: Small memory model layout, code and data each have 64KiB.

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.

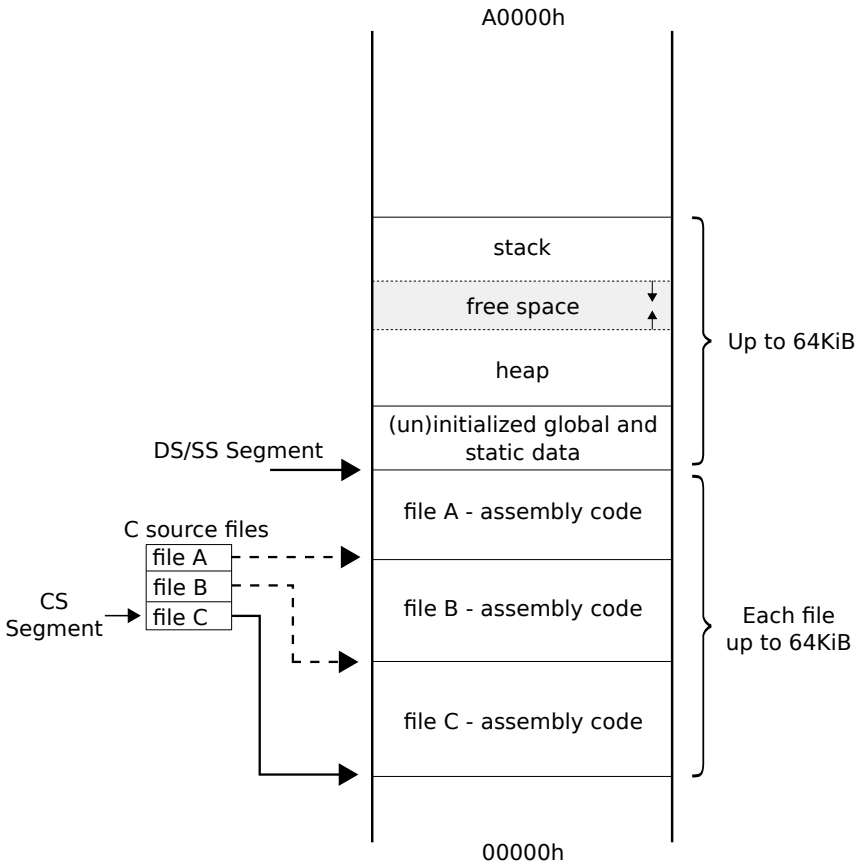


Figure 2.11: Medium memory model layout, code can be larger than 64KiB.

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 maps. 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 game maps.

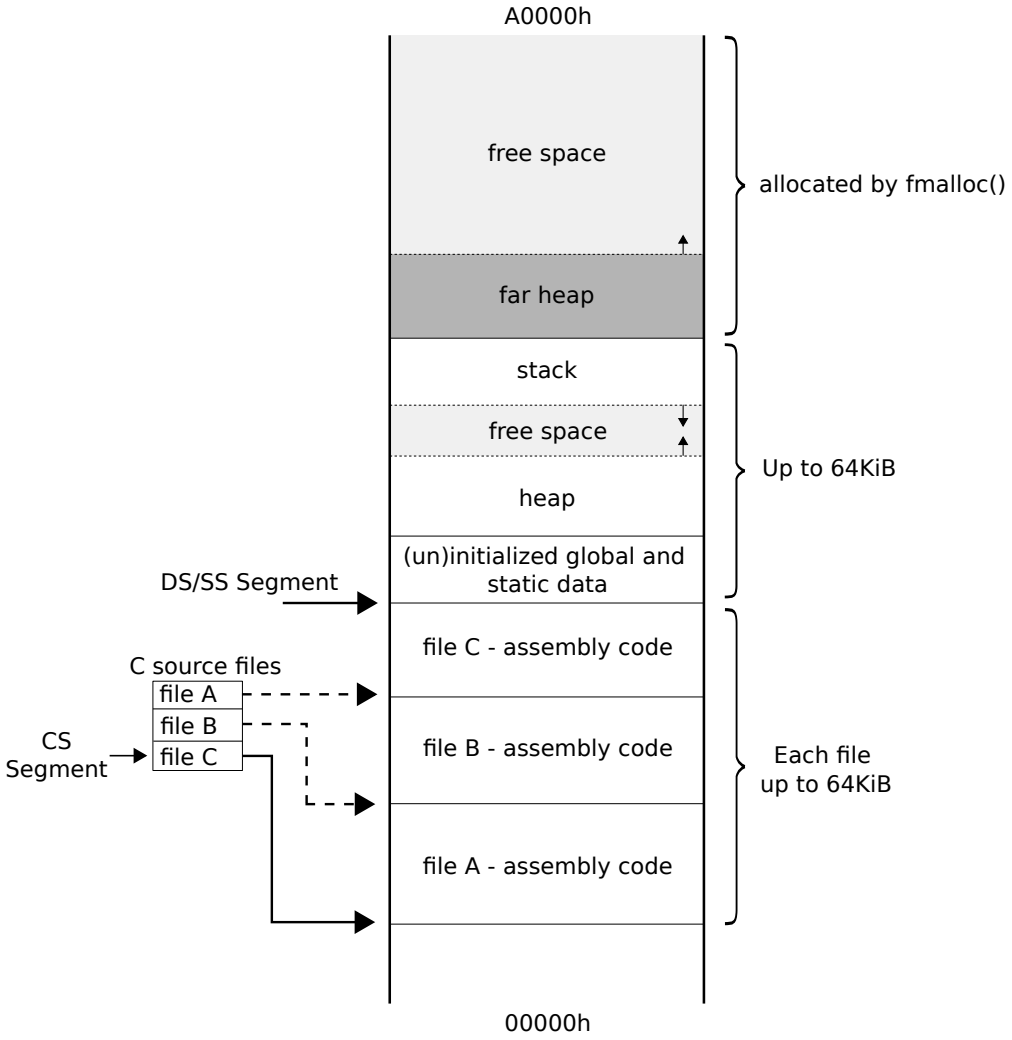
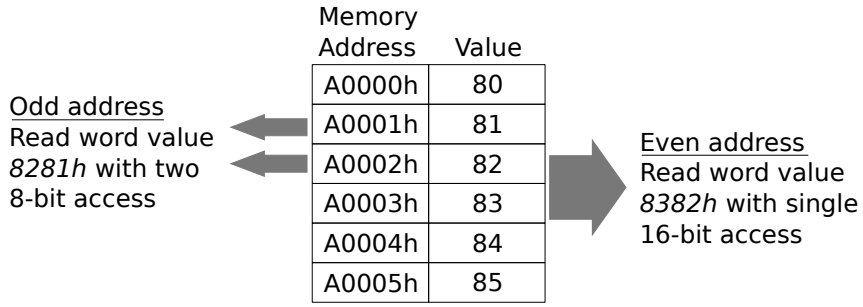


Figure 2.12: Medium memory model layout with far heap memory.

2.2.5 16-bit Data alignment

Compared to the Intel 8088, the 286 CPU contained a 16-bit external data bus, whereas the 8088 only had an 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: this is 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[10].



The way to deal with the data alignment cycle-eater is straightforward: Don't perform word-sized access 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.

2.3 Video

Unlike home computers, the IBM PC did not include onboard video hardware. Instead, it required a separate video adapter card, which was installed in one of the system's expansion slots. Each video adapter was typically paired with its own dedicated CRT monitor—a large, heavy display, usually with a 14" diagonal screen.

2.3.1 Inner workings of a CRT monitor

The first computer monitors were built around a Cathode Ray Tube (CRT). The picture on a CRT is generated by an electron gun, which converts electrical power into a narrow beam of electrons moving with substantial kinetic energy. The inside of the screen is coated with phosphor, which glows when struck by the electron beam (and for a short time thereafter).

[10] See Michael Abrash's Graphics Programming Black Book Special Edition, chapter 11.

The electron beam is highly susceptible to magnetic fields, allowing it to be deflected using magnets. By placing horizontal and vertical electromagnets around the neck of the tube (the deflection yoke), the beam can be positioned anywhere on the screen. Manually controlling the beam would be tedious and impractical, so this process is automated by connecting the electromagnets to an oscillator that generates sawtooth waveforms.

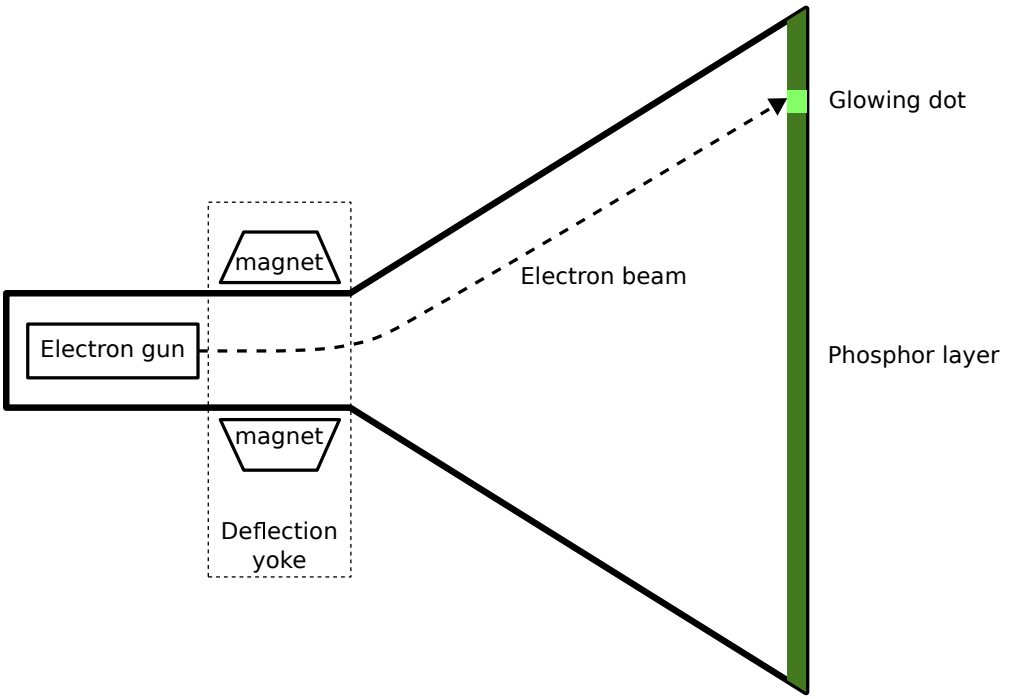


Figure 2.13: Inner workings of CRT monitor.

When both oscillators run together at precisely controlled frequencies, the electron beam scans the phosphor-coated screen from left to right and from top to bottom. The vertical oscillator determines how often the entire screen is redrawn, also known as the refresh rate. If the refresh rate is too low, the display will flicker. Most people can perceive flicker when the refresh rate drops below about 60Hz, which is why most CRT displays operate at a refresh rate of 60Hz or higher.

The CGA standard has a maximum vertical resolution of 200 lines, which requires the horizontal oscillator to operate at a much higher frequency of approximately 12kHz in order to maintain a 60Hz refresh rate. Standard CGA monitors are equipped with a horizontal frequency of 15.75kHz, which is sufficient to refresh 200 lines. EGA monitors support two synchronization modes: the CGA-compatible 15.75kHz / 60Hz mode, and an EGA-specific

horizontal rate of 21.85kHz to support the newer 350-line display.

Trivia : The 15.75kHz horizontal frequency is the standard used for NTSC television signals in North America and Japan. By adopting this frequency rate, IBM made it possible to connect a CGA card to standard televisions.

To generate an image, the output of the electron gun is modulated on and off, controlling which regions of phosphor get hit and which don't. Each horizontal sweep of the beam is called a scan line. The time during which the beam returns to the left edge of the screen is known as the horizontal retrace, while the time required to return to the top is called the vertical retrace.

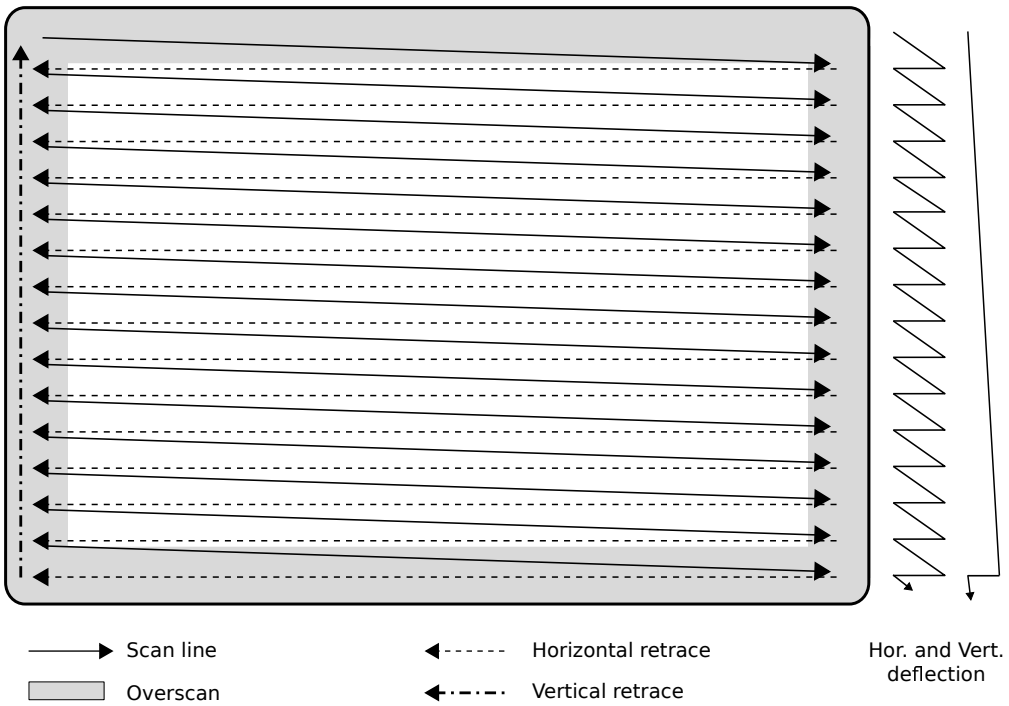


Figure 2.14: Horizontal/vertical oscillator and CRT monitor scan.

The electron beam cannot instantly stop and reverse direction; it must slow down, change direction, and accelerate again at the edges of the screen. This is handled by moving the beam outside the visible display area during retrace. This practice, known as overscan, intentionally directs the beam over regions of the screen that are not visible to the viewer, resulting in an image that is slightly larger than the active display area (the area that contains the actual character and/or graphics data).

2.3.2 And then there was color

So far, this explanation has described a monochrome display, which uses a single color of phosphor. To produce a color image, the screen is coated with a red, a green, and a blue phosphor, arranged in a triangular shape.

The color CRT contains three separate electron guns, each responsible for one of the RGB components of the image. In the figure below, the rays from the cannons are colored so the reader can follow, but electrons have no color. To ensure that each electron beam strikes only the phosphor of its corresponding color, a shadow mask is placed between the electron guns and the phosphor-coated screen.

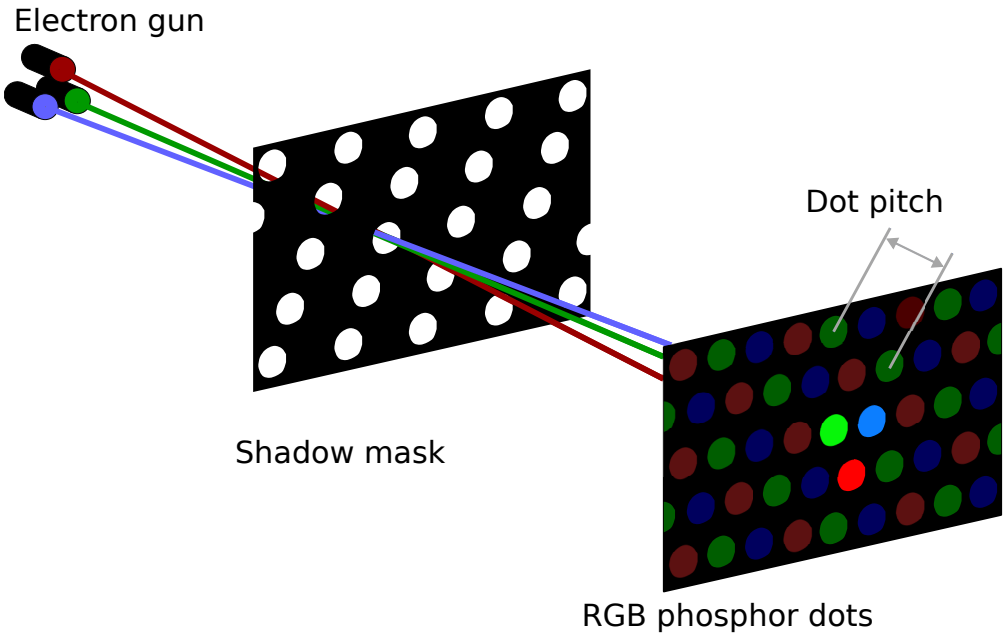


Figure 2.15: CRT color display.

The distance between the same-colored phosphor dots on the screen is called the "dot pitch". It dictates the image sharpness and for the EGA monitor it generally ranged from 0.28mm to 0.40mm.

The three dots in a triangle give the appearance of discrete pixels, but in practice it is nearly impossible to control individual phosphor elements directly. Take the example in Figure 2.16, where we light up green. The beam, represented by the circle, will usually hit multiple adjacent green phosphor elements at once. As a result, the effective “dot” (or

pixel)—the smallest addressable unit of information on the screen—is defined by the combined effects of the RGB phosphor pattern density and the diameter of the electron beam.

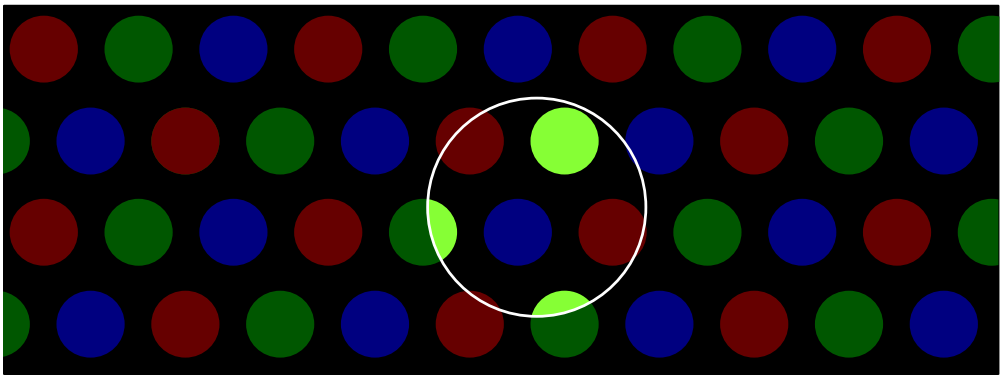


Figure 2.16: "dot" or "pixel" on a CRT screen.

Everything described up to this point occurs entirely inside the monitor. This is where the video adapter comes into play, controlling the display via the DE-9 video connector.

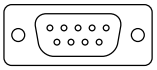


Figure 2.17: DE-9 video output connector.

The CGA and EGA adapters use

- Two pins for horizontal and vertical synchronization signals
- Four (CGA) or six (EGA) pins to control the intensity of the electron guns

The horizontal and vertical synchronization pins carry timing signals that match the monitor's operating frequencies (15.7kHz / 60Hz for CGA, and 21.85kHz / 60Hz for EGA's high-resolution mode). The monitor listens for these signals, synchronizes its internal oscillators to them, and deflects the electron beam accordingly.

The CRT draws each scan line from left to right, while the control pins from the video adapter determine which portions of the scan line are illuminated. When a scan line is completed, the video adapter generates a horizontal synchronization pulse (HSYNC), to indicate a return to the left side of the screen. This process continues line by line until the bottom of the screen is reached. At that point, the adapter generates a vertical synchronization pulse (VSYNC), and the electron beam retraces back to the top of the screen to begin drawing the next frame.

2.3.3 History of Video Adapters

The original IBM PC and XT could be equipped with the Monochrome Display Adapter (MDA) or the Color Graphics Adapter (CGA) expansion cards. With the introduction of the IBM PC AT in 1984, IBM introduced the Enhanced Graphics Adapter (EGA). Three years later, the EGA was succeeded by the Video Graphics Array (VGA).

Name	Year Released	Memory	Max Resolution
`MDA` (**M**onochrome **D**isplay **A**dapter)	1981	4KiB	80x25[11]
`CGA` (**C**olor **G**raphics **A**dapter)	1981	16KiB	640x200
`Hercules`	1982	64KiB	720x348
`EGA` (**E**nhanced **G**raphics **A**dapter)	1984	64KiB	640x350
`VGA` (**V**ideo **G**raphics **A**rray)	1987	256KiB	640x480

Figure 2.18: Video interface history.

The EGA represented an interesting middle ground for game developers in the late 1980s and early 1990s. It supported a sufficiently high display resolution and enough colors to produce respectable games. However, programming the EGA proved to be incredibly difficult. The card had many dark corners and undocumented behaviors. To fully master the hardware, one must delve deeply into its inner workings.

2.3.4 Introduction of EGA Video Card

IBM introduced the Enhanced Graphics Adapter (EGA) in 1984 as the successor to 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 2.20 shows the original IBM EGA card, a clunky beast full of discrete components. The memory consists of TMS4416 RAM, a common memory chip for (home) computers around that period. Each chip contains 16KiB of 4-bits memory, so two chips are needed to provide 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[12]. Still, it was not until 1984 that they became widely available and lower-priced than four TMS4116 chips (16KiB by 1-bit). However, at that time 64KiB RAM was the way to go for new designs. Computers with only 16KiB as base memory, and that's where TMS4416 would have been a cost saver, were already on the way out.

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

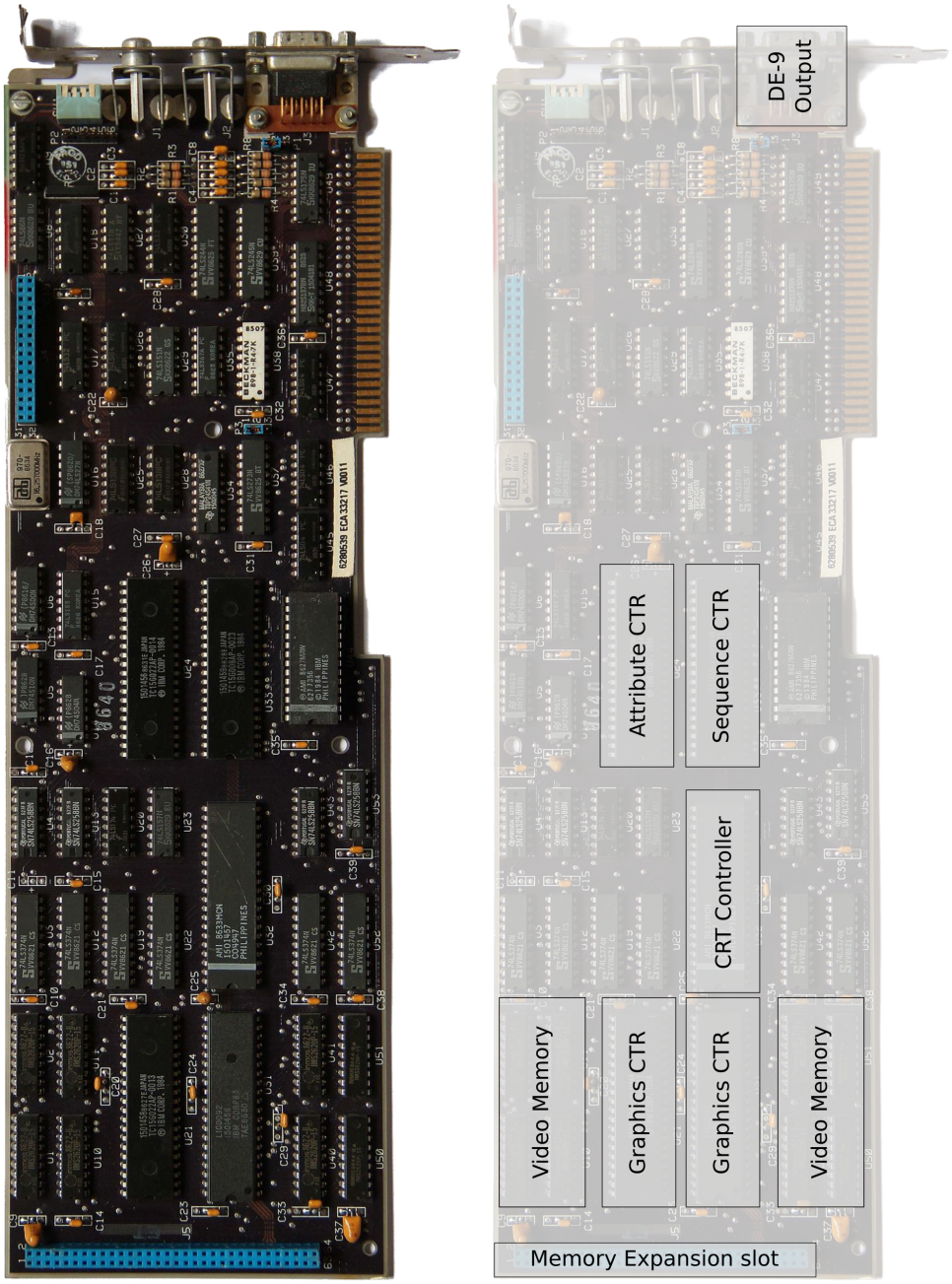


Figure 2.19: Original IBM EGA card

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 dual in-line package (DIP) 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.

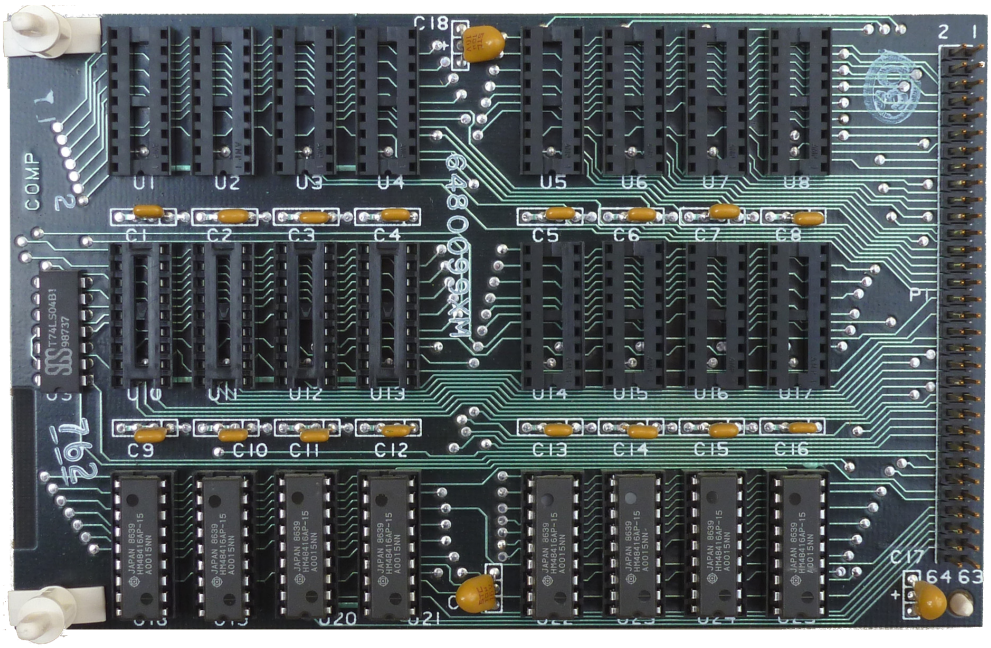
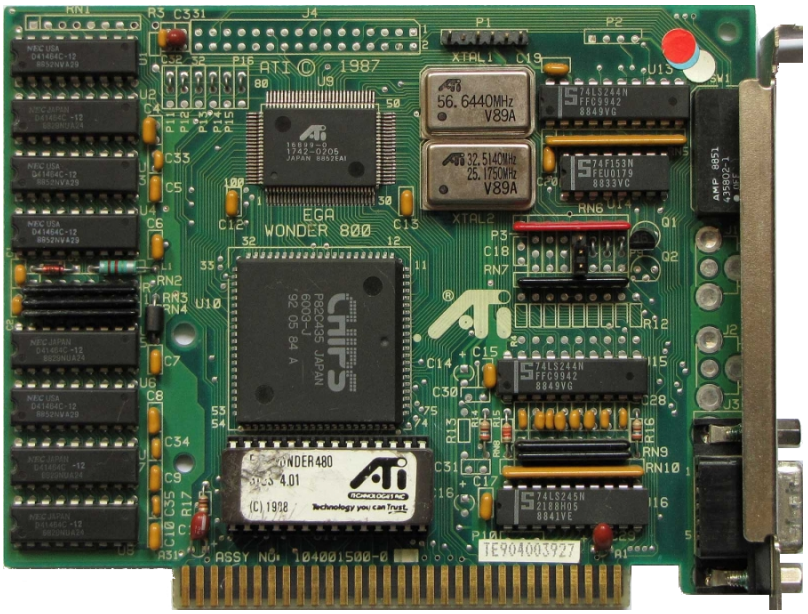


Figure 2.20: EGA Memory Expansion Card, bottom row populated with memory chips.

The EGA clones that started appearing 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[13].

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

[13]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
1E904003927


PARADISE
PEGA1A
38304 8841
015567190502
Paradise Basic EGA Card
ASSY 28250 REV 1
FCC ID:DBM8FL24LMRP AEGA1A
MADE IN HONG KONG
R2TP 553645
FOXCONN
DT1122
SI 8829

2.3.5 EGA Architecture

The EGA card was documented with a highly technical architectural diagram, filled with cryptic components and many wires between them.

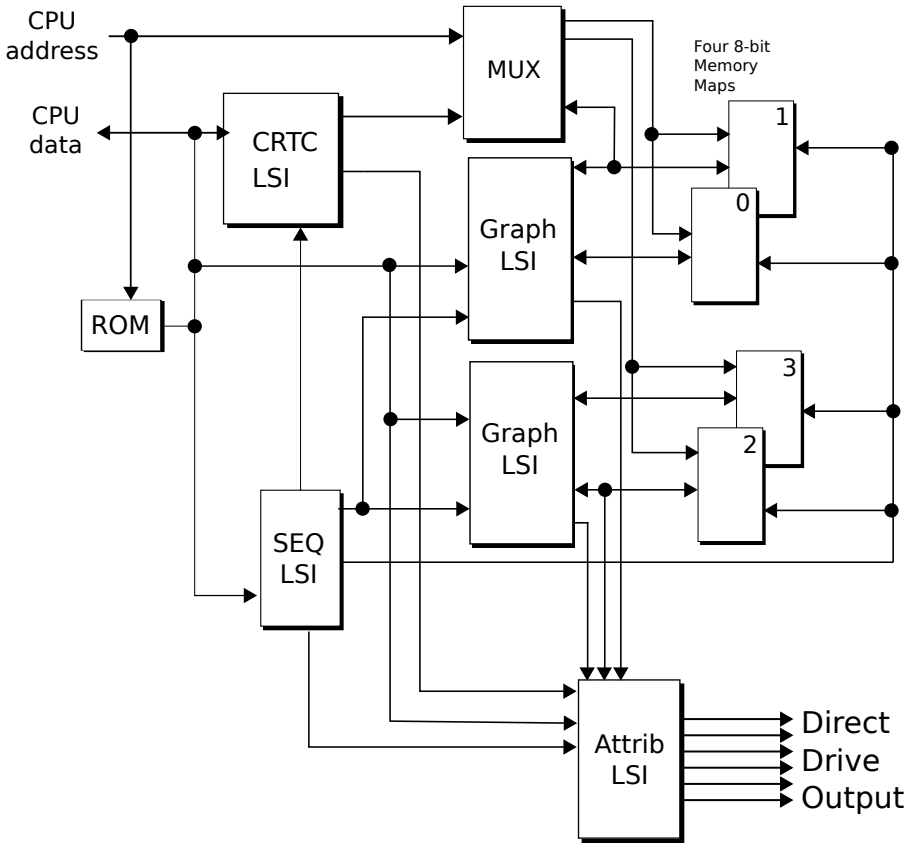


Figure 2.21: IBM's EGA documentation.

The overall design can be simplified into several main components responsible for input, storage, and output:

- Two discrete Graphics Controllers to control and translate data between the CPU and video memory.
- The framebuffer (VRAM), organized into four memory banks (planes), which stores the image data.

- The Sequencer Controller, which reads data from the four memory banks and converts it into color indices.
- The CRT Controller and Attribute Controller, which convert color indices into synchronized RGBI signals and transmit them to the display using TTL[14] signaling.

Trivia : In the 1980s integrated video DACs[15] 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.

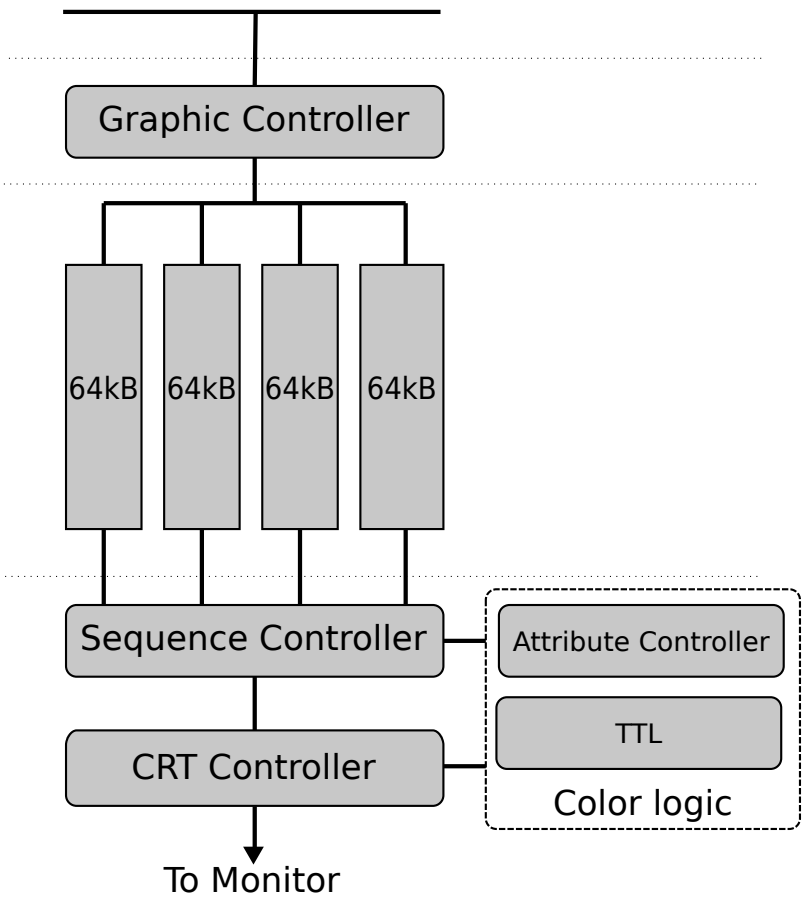


Figure 2.22: Simplified EGA architecture, it should not be considered 100% correct.

[14] Transistor-Transistor Logic
[15] Digital to Analog Converter

A particularly interesting aspect of the architecture is the use of four memory banks. Let's do some math to better understand why this architecture is chosen. A 640×350 image with 16 colors (4 bits per pixel) requires 640×350×4 bits = 896,000 bits, or 112KB of VRAM. This exceeds the 64KiB segment size imposed by the x86 Real Mode architecture, meaning the CPU could not access the entire framebuffer without performing relatively slow segment register changes while updating the screen.

A second, more serious issue was memory bandwidth. To support a CRT refresh rate of 60Hz, the adapter needed to provide one byte every $\frac{1}{112{,}000 \times 60} = 149$ nanoseconds. At the time, typical RAM access latency was around 200ns, which would have choked the video adapter.

The EGA card solved these limitations by dividing memory into four planes and accessing them in parallel. This type of architecture is called "planar". This reduced the effective address space per plane by a factor of four, meaning each plane only needed to supply one byte every 596 nanoseconds, thereby relaxing the timing constraints on memory access.

2.3.6 EGA Modes

The EGA card remained backward compatible with previous video adapters. In addition to introducing new, higher-resolution graphics modes, EGA retained support for both Monochrome Display Adapter (MDA) and Color Graphics Adapter (CGA) modes. This ensured that existing software written for earlier IBM PCs would continue to function without modification.

Mode	Type	Format	Colors	RAM Mapping
00h	text	40x25	16 (monochrome)	B8000h
01h	text	40x25	16	B8000h
02h	text	80x25	16 (monochrome)	B8000h
03h	text	80x25	16	B8000h
04h	CGA Graphics	320x200	4	B8000h
05h	CGA Graphics	320x200	4 (monochrome)	B8000h
06h	CGA Graphics	640x200	2	B8000h
07h	MDA text	9x14	4 (monochrome)	B0000h
0Dh	EGA graphic	320x200	16	A0000h
0Eh	EGA graphic	640x200	16	A0000h
0Fh	EGA graphic	640x350	4	A0000h
10h	EGA graphic	640x350	16 (out of 64)	A0000h

Figure 2.23: EGA Modes available.

Trivia : The modes 08h-0Ah are reserved for PCjr (or Tandy Graphics Adapter) graphics modes, which offered 160×200 with 16 colors, 320×200 with 16 colors and 640×200 with 4 colors. Modes 0Bh and 0Ch are reserved for internal EGA BIOS.

Changing the video mode is usually done using the 0x10h BIOS interrupt, which handles all kinds of video display operations. Changes via the BIOS are regarded as a safe way to ensure all registers are setup properly.

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

2.3.7 EGA Memory Mapping

Accessing video memory on the PC was not as straightforward as it might seem. Instead of communicating with the video card through special I/O instructions, the system relied on memory mapping. The video card's VRAM was mapped directly into the CPU's address space—on the EGA beginning at A0000h—so the processor could treat it like conventional system RAM. This allowed for faster and more flexible data transfers.

However, this design introduced a new challenge: how do you fit 256KiB of video memory into a 64KiB segment? The solution was "bank switching", implemented through a mapping mask configured via the EGA registers. By configuring the mask first, a program can select which memory plane it wants to write to.

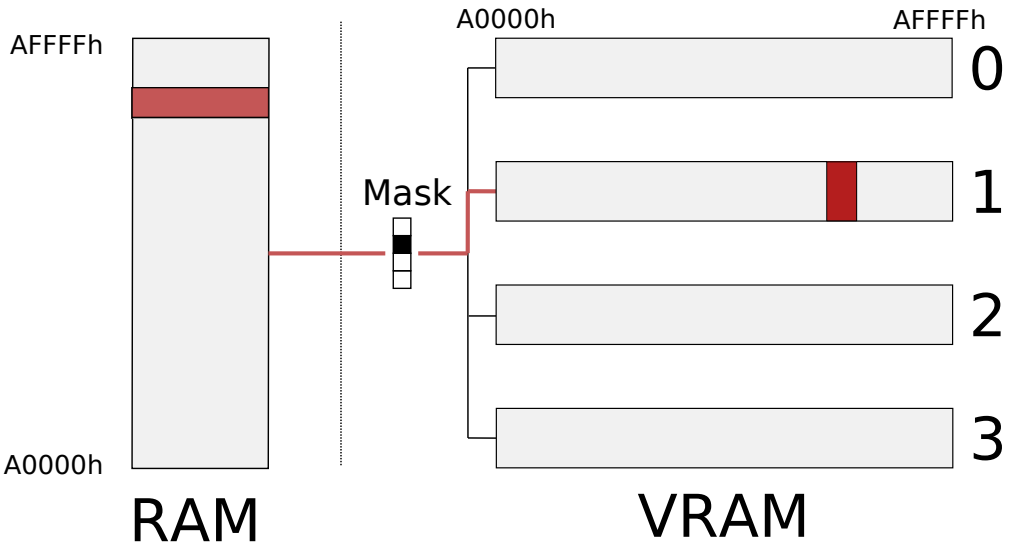


Figure 2.24: Mapping PC RAM to EGA VRAM banks.

In mode `0Dh`, each plane carries one bit of a pixel's palette index. For EGA there are four planes, where combining one bit from each plane produces a 4-bit color index. This index is then translated into an on-screen color by the attribute controller, as illustrated below.

To set the color of a single pixel, the programmer must write one bit in each of the four planes at the corresponding byte and bit position. Writing an individual pixel color was therefore CPU-intensive and should be minimized whenever possible.

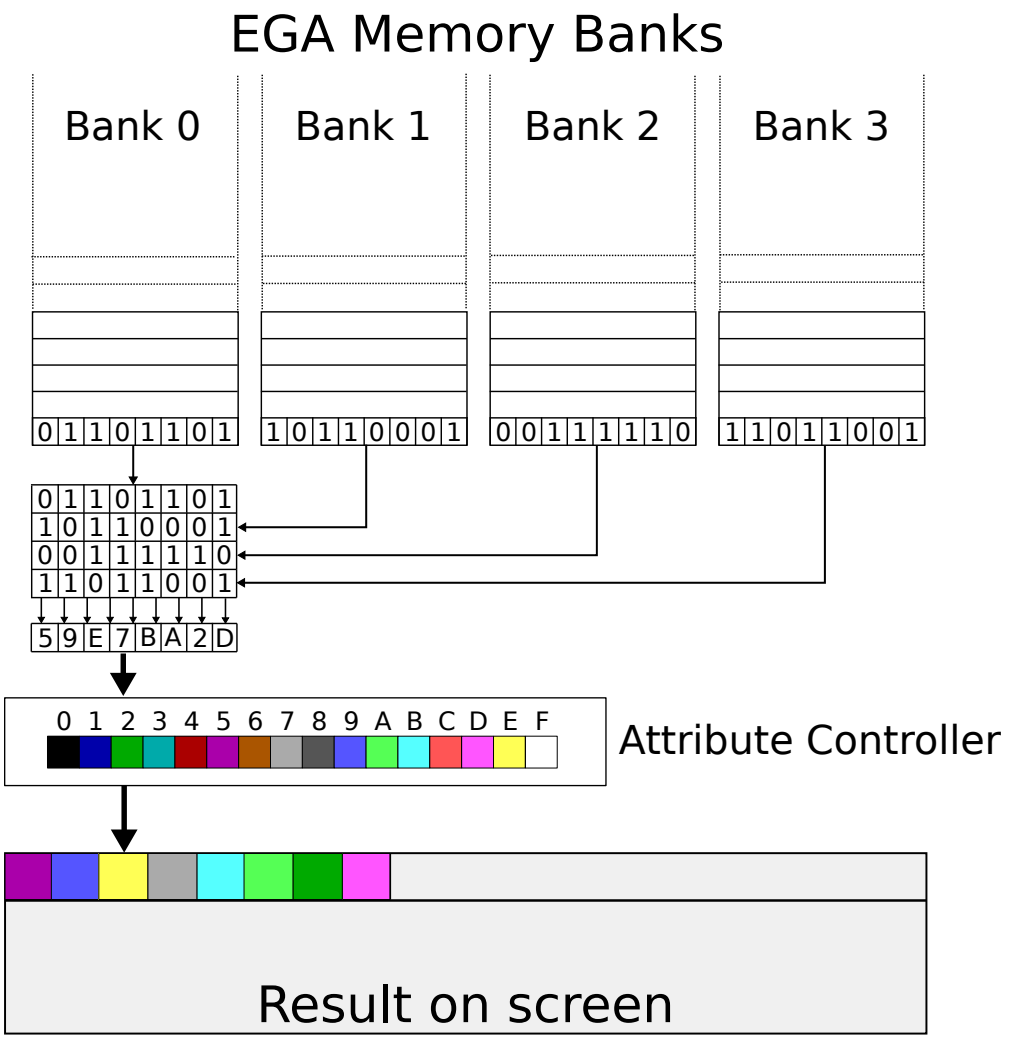


Figure 2.25: EGA mode 0Dh, how bank layout appears on screen.

2.3.8 Programming on the EGA

In the days of the EGA, there were no graphics drivers to rely on and no Windows DirectX to smooth things over. Programming graphics truly meant programming on the bare metal via I/O port addresses. The EGA card had multiple I/O port addresses that are generally directed to one of the sub-components.

Component	I/O port address
Miscellaneous	`3B2h`, `3BAh`, `3C2h`, `3D2h`, `2DAh`
CRT Controller	`3B4h`, `3B5h`, `3D4h`, `3D5h`
Attribute Controller	`3C0h`
Sequence Controller	`3C4h`, `3C5h`
Graphics Controller	`3CCh`, `3CAh`, `3CEh`, `3CFh`

Figure 2.26: EGA I/O port addresses.

In most cases, a component used one I/O address to select a parameter and another as a data register, allowing dozens of settings to be exposed and configured on the EGA card. Mastering the EGA card, however, was far from straightforward. Its many registers, bit masks, and undocumented quirks demanded patience and deep hardware knowledge. Only a few programmers truly learned how to control the card to its full potential.

2.3.9 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 is 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 indices 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 2.27: EGA "rgbRGB" color palette (64 values from 00h to 3Fh)

Both CGA and EGA cards use a female nine-pin D-subminiature (DE-9) connector for output to the monitor. However, the pin configurations for CGA and EGA are different. CGA's color output follows the "RGBI" model, where "I" stands for Intensity, adding brightness to the RGB color. As a result, CGA can produce up to 16 colors. Compared to CGA, the EGA card 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.

For example, if the color brown (rgbRGB = `010100b`) is assigned to a color index, the resulting output on a CGA pin configuration appears as light red. This occurs because the secondary green pin ("r" in rgbRGB) is mapped to the Intensity pin in CGA mode, producing the color red with intensity rather than the expected brown.

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

So how did an EGA monitor recognize that it was connected to an EGA video card? In fact, it could only distinguish EGA and CGA cards based on the Vertical Sync signal (pin 9), which is either 200-line or 350-line mode. If the Vertical Sync signal indicates 350-line mode, the monitor switched to Mode 2 operation, which supported the extended rgbRGB-color information[16]. However, in the 200-line mode the monitor cannot distinguish between being connected to a CGA or an EGA card.

[16] IBM Enhanced Color Display technical documentation

For this reason, EGA monitors use the CGA pin assignment for all 200-line modes (mode 1), allowing them to accept the same RGBI signal format as a CGA card. This makes an EGA monitor compatible with CGA output in those modes. Thereby, it is able to show all 16 CGA colors simultaneously, whereas a CGA card is limited to four colors at once in graphics modes. The downside is that the 64-color EGA palette can only be used in 350-line mode.

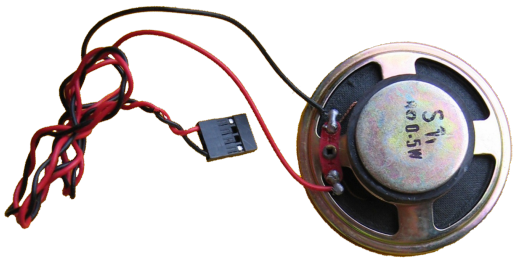
Index Number	Color	rgbRGB	RGBI
00h	Black	000000b	0000b
01h	Blue	000001b	0010b
02h	Green	000010b	0100b
03h	Cyan	000011b	0110b
04h	Red	000100b	1000b
05h	Magenta	000101b	1010b
06h	Brown	010100b	1100b
07h	Light grey	000111b	1110b
08h	Dark grey	111000b	0001b
09h	Bright blue	111001b	0011b
0Ah	Bright green	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 2.29: Default EGA 16-color palette

Trivia : The color brown (`06h`) is actually dark yellow. IBM engineers found this shade unappealing, so they implemented a special hack in the monitor that targets color `06h`. When detected, the monitor halves the green intensity, producing a golden brown instead of dark yellow.

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, commonly known as a "PC Speaker", 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!



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, 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 sound cards.

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 needed. 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 could rival even today's sound cards. Connecting the MT-32 to a PC required what Roland called an MPU-401[17], in one of the PC's expansion slots. Sierra sold the MT-32 with a necessary MPU-401 interface for $550. The high

[17] Midi Processing Unit-401

price prevented it from dominating the end-user gaming market.

Figure 2.30: Roland MT-32 synthesizer box.

The IBM Music Feature Card was launched in March 1987 as a collaboration between IBM and Yamaha. Essentially, the Music Feature Card was a synthesizer installed on an 8-bit expansion card[18]. The Music Feature Card had 8 FM voices, controlled via 4 frequency operators. It came with over 300 high-quality synthesized instruments onboard, 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.

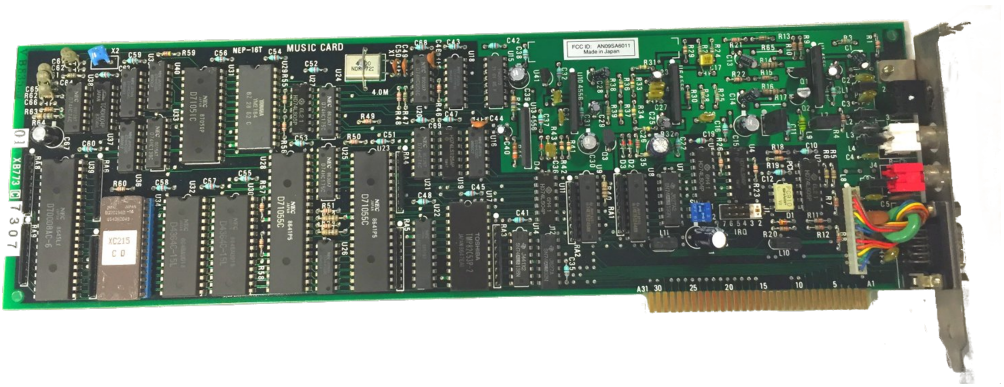


Figure 2.31: IBM Music Feature Card.

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

2.4.2 AdLib

AdLib was the other company, besides 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 sound card used a technology called FM synthesis. The technology is based on the idea of generating superimposed 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 percussion 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 the early PC sound card market.

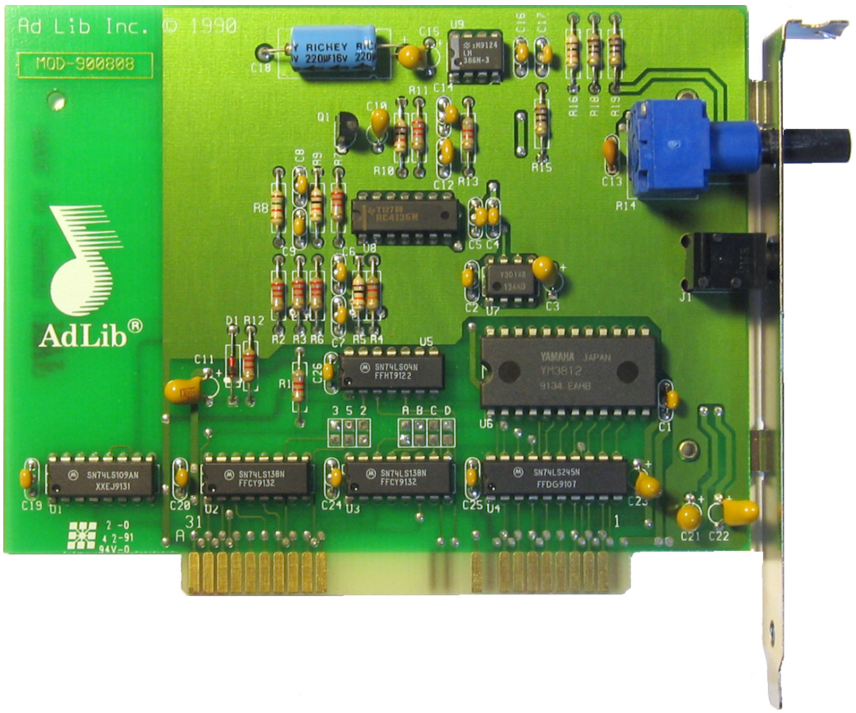


Figure 2.32: An AdLib sound card. Notice the big YM3812 chip.

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

2.4.3 Sound Blaster

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

The original Sound Blaster v1.0 and v1.5 were released in 1989 as successors to the Game Blaster. The Sound Blaster was equipped with the same OPL2 chip used by AdLib, providing 100% compatibility with AdLib music. In addition, it included a Digital Sound Processor (DSP), allowing digitized sound playback at 8 bits per sample and sampling rates of up to 22.05 kHz.

The card also included a DA-15 port for joystick connections. Most importantly, the Sound Blaster was about $90 cheaper than the AdLib, which led many consumers to choose it instead. In less than a year, the Sound Blaster became the best-selling expansion card for the PC.

> "At Comdex [1989], we sold one Sound Blaster every four minutes."
>
> **Sim Wong Hoo, founder, on their rapid success**

Figure 2.33 is the Sound Blaster 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 backward compatibility with the Creative Music System. The CT1320B model (Sound Blaster 1.5) was a cost-cutting measure. Having recognized that C/MS was unpopular, Creative Labs 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.

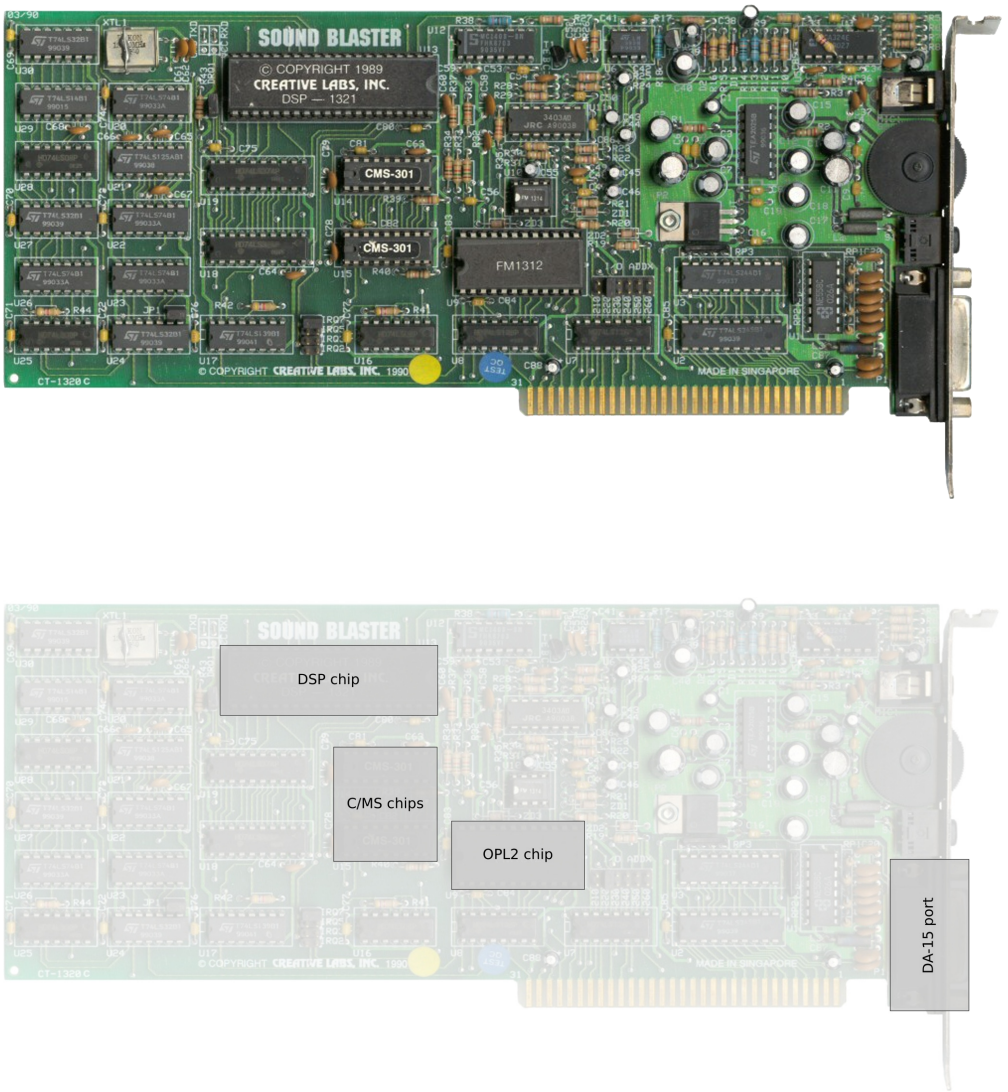


Figure 2.33: A Sound Blaster 1.5, model CT1320B

Trivia : Creative Labs boasts in its own advert about AdLib compatibility and even uses an image of AdLib Inc's product[19]! On the other hand, they sued every company that tried to market a 'Sound Blaster compatible' product for using the name of their product.

[19] Advert p20 in Compute!, April 1990.

Figure 2.34: Sound Blaster advertisement with Adlib compatibility.

2.5 Floppy Disk Drive

Before the internet, floppy disks were the primary medium for sharing and distributing software and data. The original XT systems were equipped with 5¼-inch floppy disk with a capacity of 360KiB[20]. In 1984, IBM introduced with its PC AT the 1.2MB dual-sided 5¼-inch floppy disk, but it never became very popular. IBM started using the 720KiB double density 3½-inch floppy disk in 1986 and the 1.44MB 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.

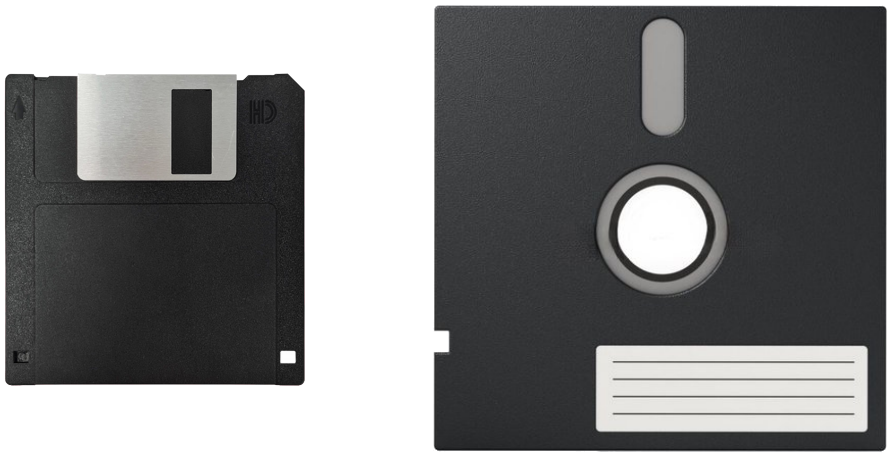


Figure 2.35: 3½-inch and 5¼-inch floppy disk.

A floppy disk is essentially a very flexible piece (hence the term floppy disk) of plastic coated on both sides with 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 to read data from it.

[20] Manufacturers often rounded this down to 360K or 360KB for simplicity, using the decimal definition of KB (1000 bytes) in marketing.

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.

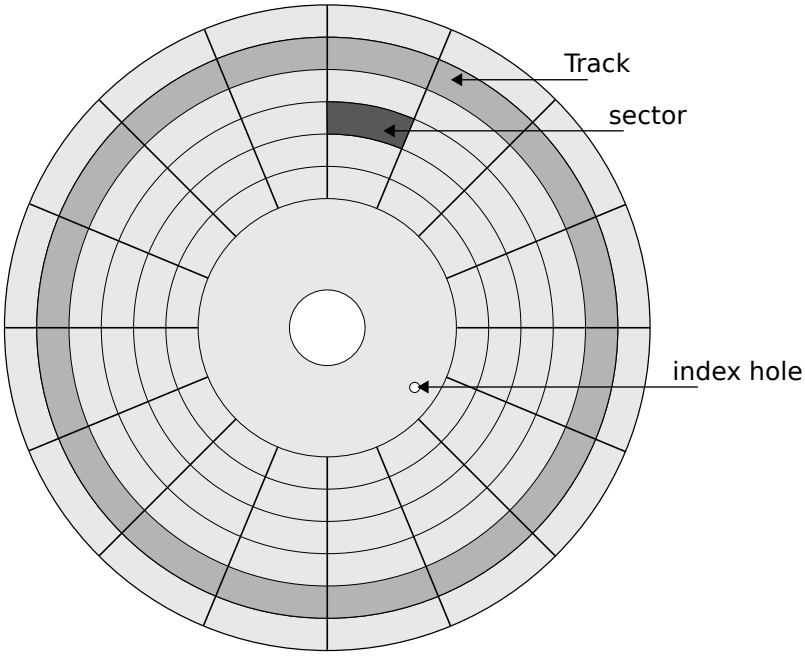


Figure 2.36: Floppy disk with tracks, sectors and the index hole.

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, moving 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 waits 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 for 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 2.37: 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 (ANSI) or 102 (ISO).

The ANSI layout has 101 keys and was used for the US and Canada, and the ISO layout has 102 keys, primarily used in Europe and other international markets. The primary differences between both layouts are the shape of the Enter key, the size of the left Shift

key, and the presence of an extra key on the ISO variant to provide better support for typing in multiple languages with accents and special characters.

Figure 2.38: IBM Keyboard model M 101-key ANSI layout.

2.7 Summary

By the late 1980s, programming games for the PC could feel like an uphill battle. The CPU still behaved as if it were 1978, constraining developers to a 1MiB address space. Its memory model with near and far pointers was complex, and could turn even simple tasks into careful memory juggling.

Graphics offered little relief. The EGA video card featured a complex planar architecture that made something as straightforward as setting a colored pixel surprisingly cumbersome. Writing to video memory required extensive hardware knowledge and a fair amount of patience.

Sound was equally restrictive. Most PCs relied on the PC speaker, capable of little more than beeps and simple tones. Dedicated sound cards existed, but they were still in their infancy and far from standardized.

And yet, despite these formidable technical constraints, small and determined development teams managed to create remarkable games. Among them was a group calling themselves *Ideas From the Deep*[21].

[21] Throughout the book, *Ideas from the Deep* and *id Software* will be used interchangeably.

Chapter 3

Assets

3.1 Programming setup

Development was done with Borland C++ 3.1 using 80386 computers; however, none of the 80386-specific features were utilized for the game. John Carmack took care of the game engine. 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). As the team did not have money to buy their own computers, they used the Softdisk computers after hours and on weekends to develop the game.

In the late 1980s, the concept of an Integrated Development Environment (IDE) was just beginning to mature. One of the first all-in-one packages was Borland Turbo C++. It was considered the standard for development on the DOS system.

The software came with two thick manuals, explaining everything about the IDE (238 pages) and programming in C++ (483 pages). The IDE, `BC.EXE`, featured a blue-window text-based user interface that integrated the editor, compiler (`BCC.EXE`) and debugger into a single application. The IDE's standard resolution mode was 80×25, but also included a "high resolution" 80×50 text mode.

Figure 3.1: Borland C++ 3.1 manuals.

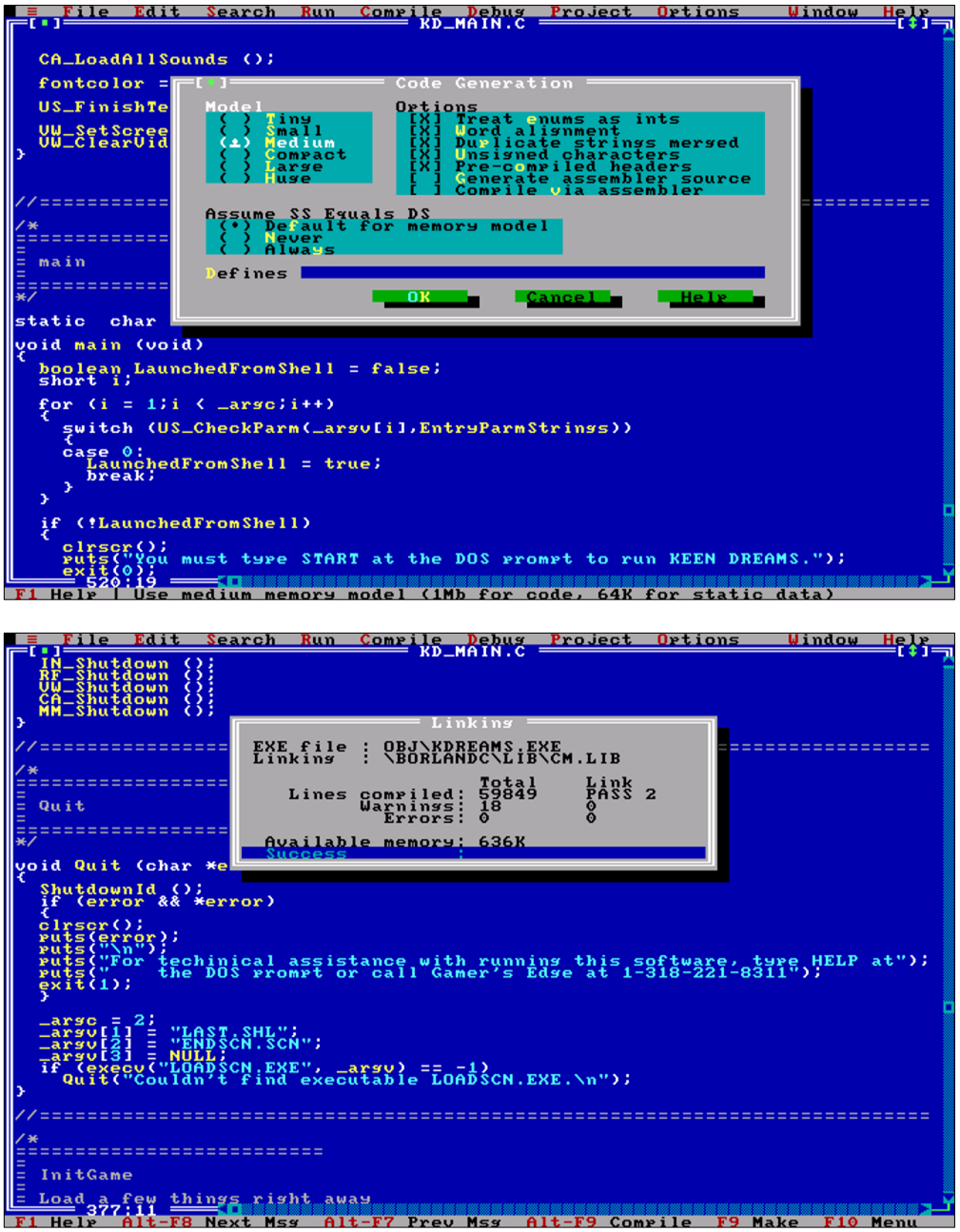


Figure 3.2: Borland C++ 3.1 IDE in high resolution mode.

3.2 Graphical Assets

All graphical assets were produced by Adrian Carmack, using Deluxe Paint II (by Brent Iverson, Electronic Arts) and saved in ILBM[1] files (a proprietary Deluxe Paint format). Deluxe Paint II supported many graphic modes, including all EGA modes.

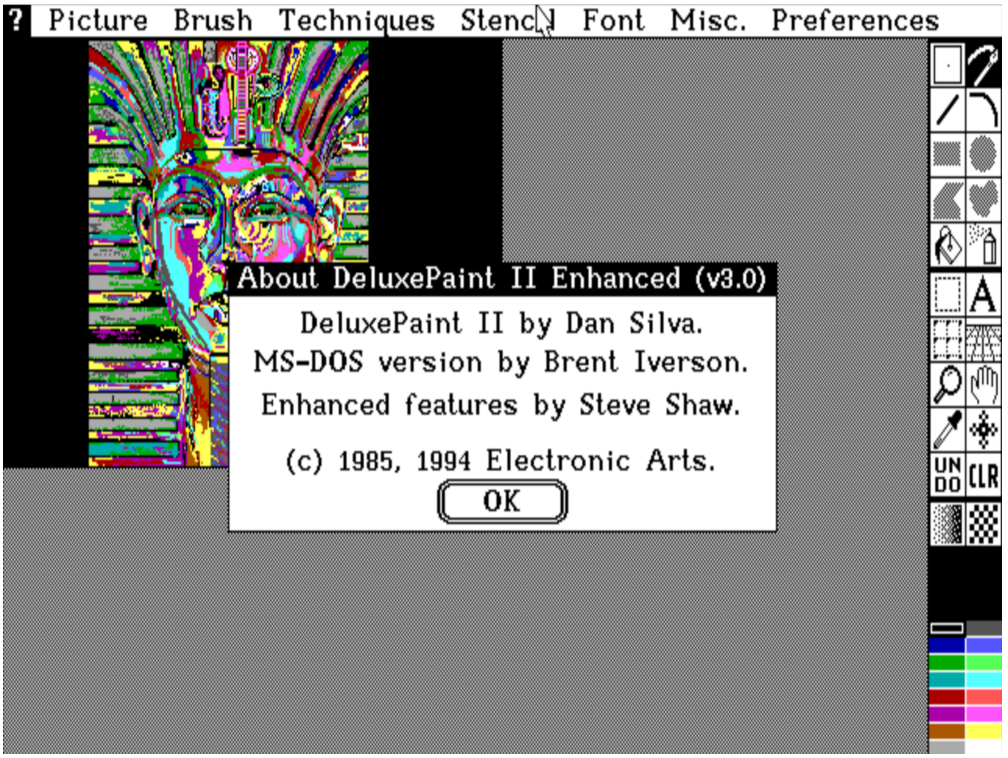


Figure 3.3: Deluxe Paint II in EGA 640×350 mode with 16 colors.

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 simply a "tile"—is a picture that measures 16×16 pixels. Each tile occupies 128 bytes (2 bytes × 16 lines × 4 planes) of storage space in the graphics assets file.

[1] InterLeaved BitMap.

Individual tile images are stored in a planar format, with blue, green, red, and intensity bits separated. This arrangement, called "planar graphics", stores complete planes sequentially, with each plane containing data for an entire tile. This structure mirrors the layout of EGA VRAM, enabling efficient data transfer. Each plane can be copied directly into its corresponding VRAM bank using a single `memcpy`. As a result, rendering a tile requires only four bank switches.

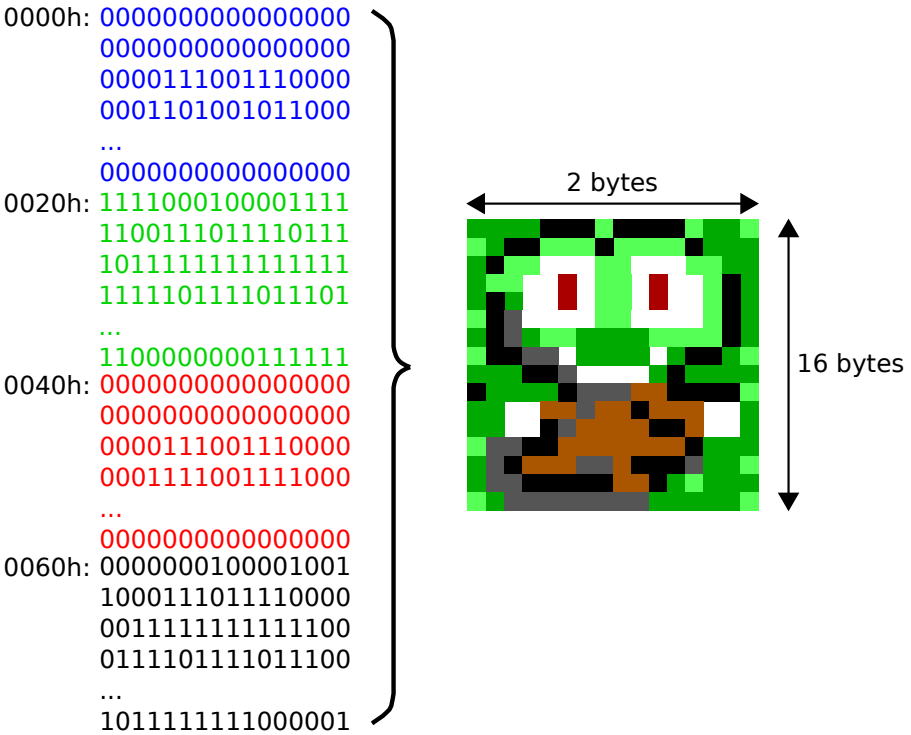


Figure 3.4: Graphic planar data storage.

Foreground tiles, or "masked tiles", are similar to normal tiles but contain five planes instead of four. The extra plane stores the mask. 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, whereas a mask bit of '1' preserves the background color behind the foreground tile.

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 include 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 graphical assets were generated, a tool (IGRAB) packed all ILBM files into a compressed asset archive file (`*.EGA`) and generated a `HEAD` table and Huffman `DICT` file (`*.KDR`-files[2]), and a C header file containing asset IDs. The engine references an asset directly using these IDs via an enum. This enum serves as an offset into the `HEAD` table, which then provides an offset within the asset archive file.

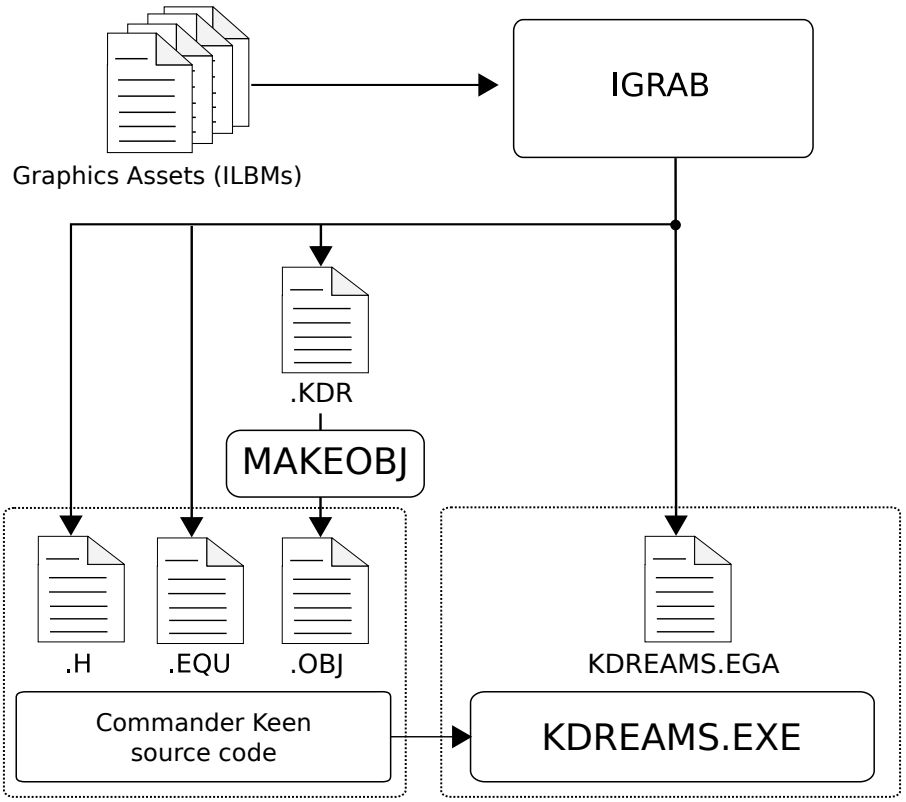


Figure 3.5: Asset creation pipeline for graphics items

The `HEAD` and `DICT` files in the source code must match the asset archive file from the shareware version of *Keen Dreams*. However, the latest release of the source code (v1.93) does not match shareware version v1.13. To ensure that the `HEAD`, `DICT`, and asset archive file are compatible, you will need to retrieve a specific git commit, which will be explained in the next chapter.

[2]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
#define CARROT_LUMP_START         261
[...]
```

3.2.3 Asset file structure

Figure 3.6 illustrates the structure of the `KDREAMS.EGA` archive file. The first section contains data tables for pictures and sprites, followed by the font, and all sprite and tile graphs.

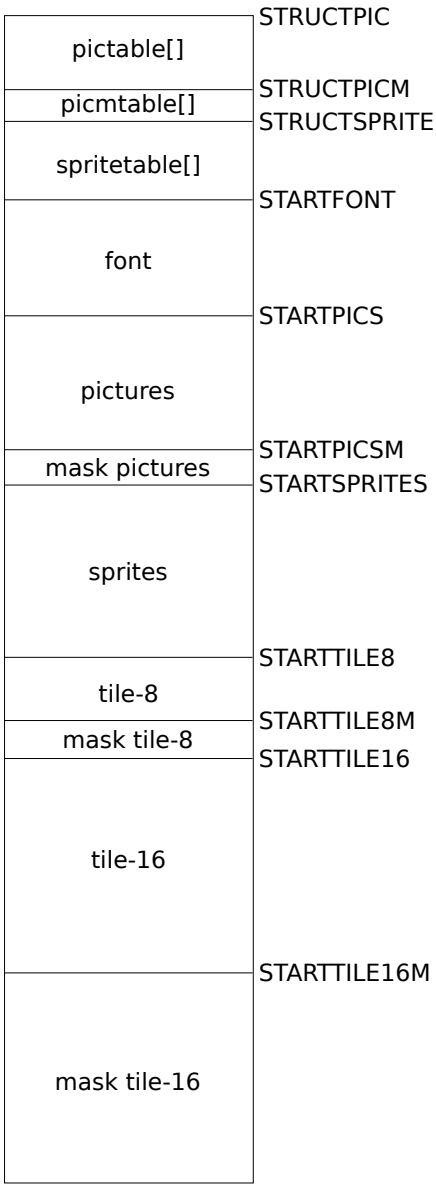


Figure 3.6: File structure of `KDREAMS.EGA` archive file.

Both the `pictable[]` and `picmtable[]` contain the width and height for each picture in the asset file. The `spritetable[]` includes not only width and height but also information on the sprite's center, hitbox, and number of bit shifts (explained in 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.

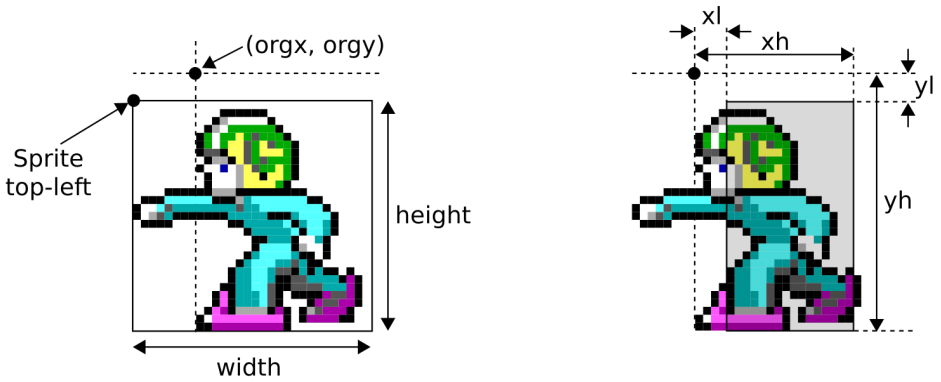


Figure 3.7: Sprite dimensions, origin and hitbox.

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

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

The remainder of the archive file contains all graphical assets, including pictures, sprites, and tiles. The picture assets are illustrated below; they consist solely of buttons used in the control panels.

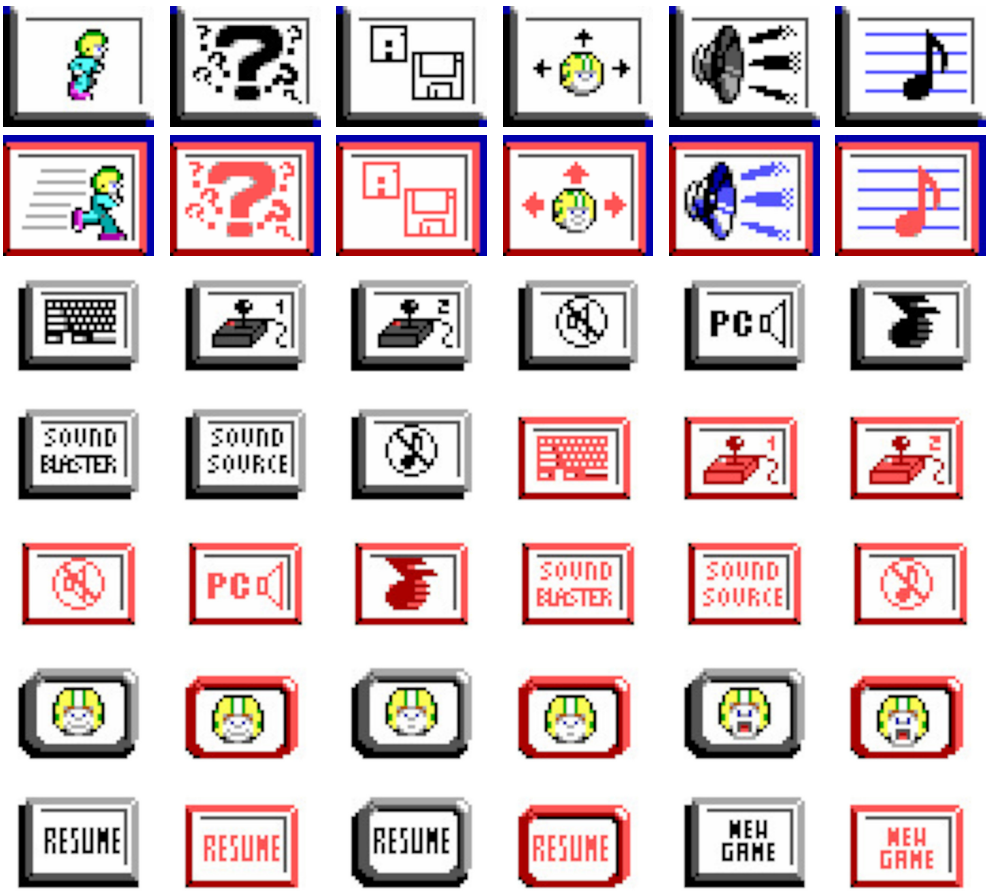


Figure 3.8: Picture assets used in control panel.

On the next two pages, you will find several sprite assets, as well as background and foreground tiles. The intersection of the horizontal and vertical axes indicates the corresponding tile asset ID.

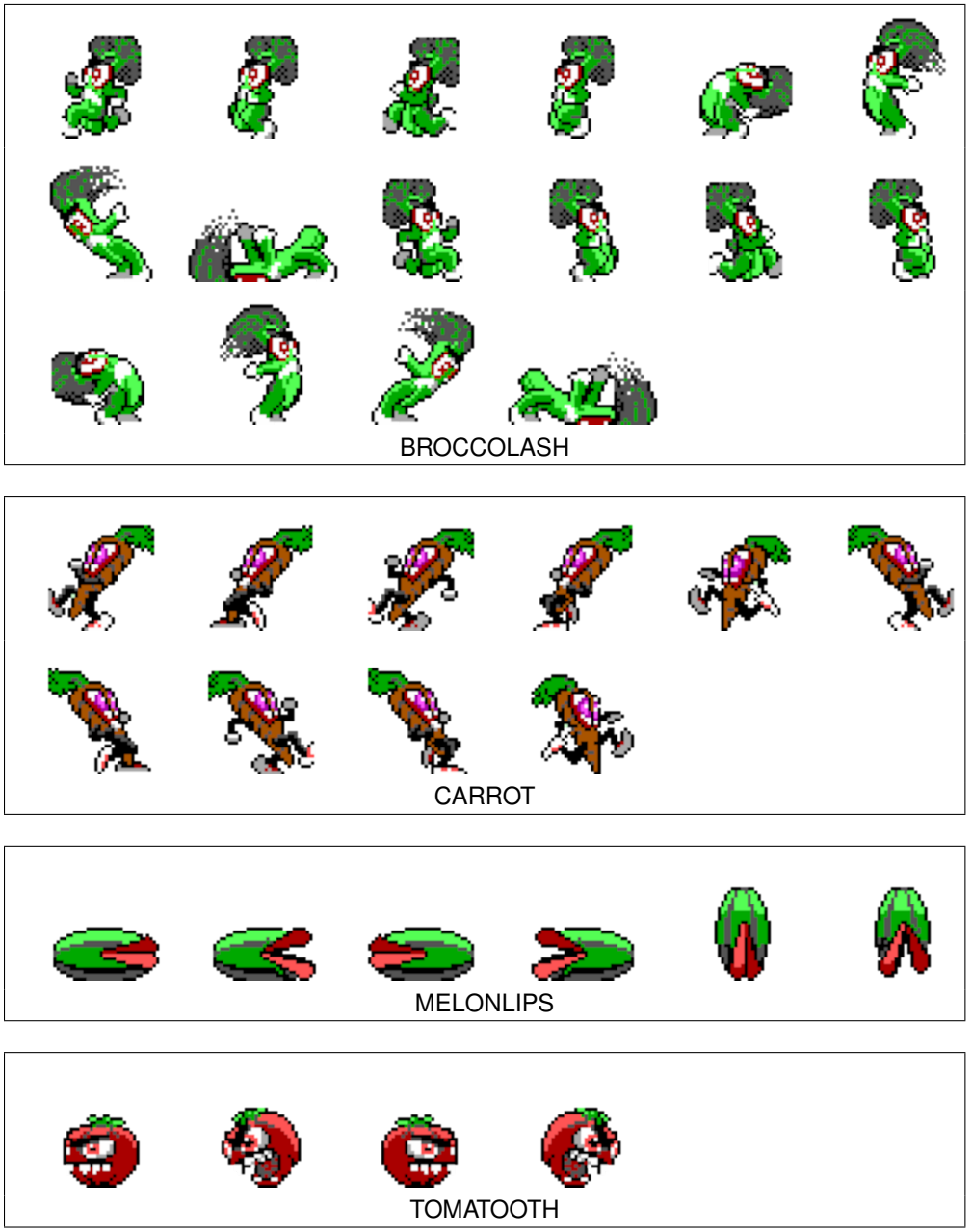


Figure 3.9: Sprite assets.

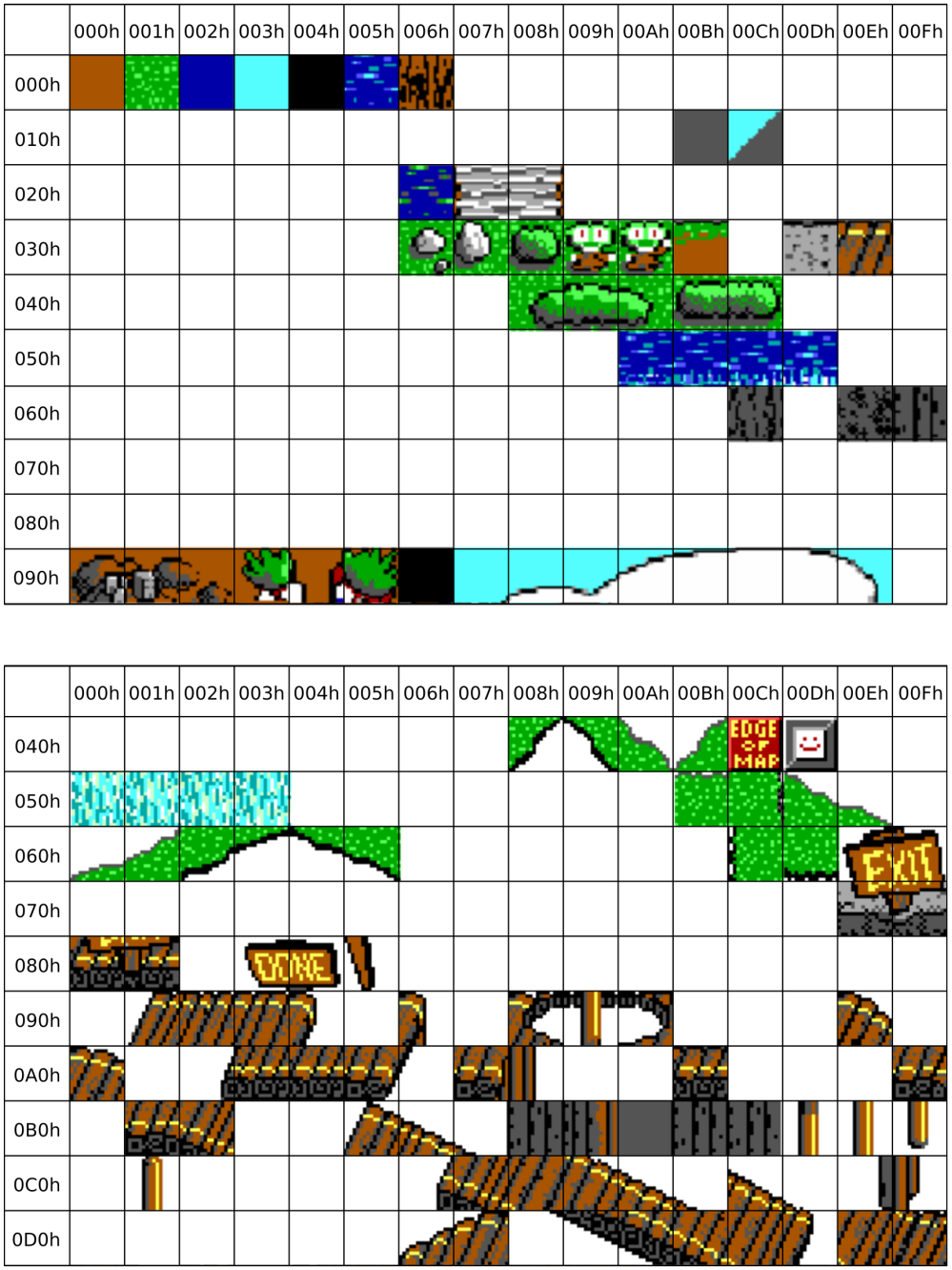


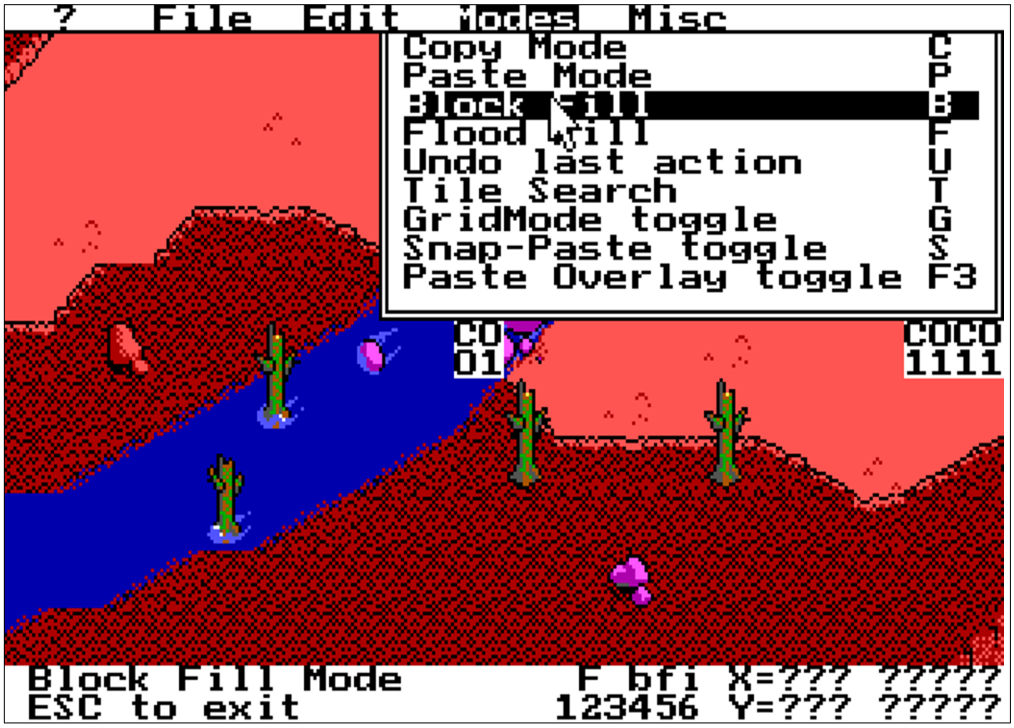
Figure 3.10: Background (top) and foreground (bottom) tile assets.

3.3 Maps

All maps were produced by Tom Hall and John Romero, using an in-house editor called TED, short for Tile EDitor. Over the years TED was improved, and the same tool was later used for creating maps for both side-scrolling games and top-down games like *Wolfenstein 3D*.

> When we were first starting to make stuff together, I made TEd 1.0, and it evolved over the next six months to be TEd 5.0, which was used for 33 shipped games.
>
> **John Romero**[3]

TED is not standalone; in order to start, it needs an asset archive and the associated header and dictionary files (as described in Figure 3.5 on page 67). This way, tile and sprite IDs are directly encoded in the map.



[3] Interview with David Lightbown from Gamasutra, February 2017

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

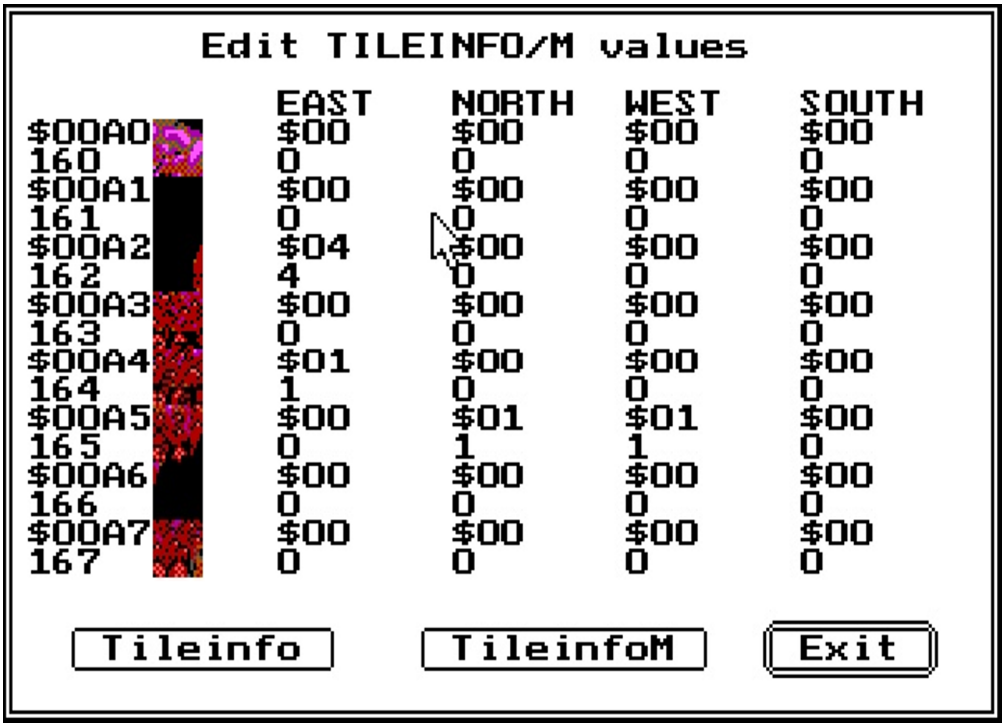


Figure 3.11: "Tile info" adding properties to tiles.

Similar to IGRAB, TED5 compresses all maps 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;
```

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 3.1, 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 as the number of ticks before the next tile is displayed.

tile #	tile animation	tile animation speed
...	...	...
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, such as 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 example, for 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
...	...	...	...	...	...

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

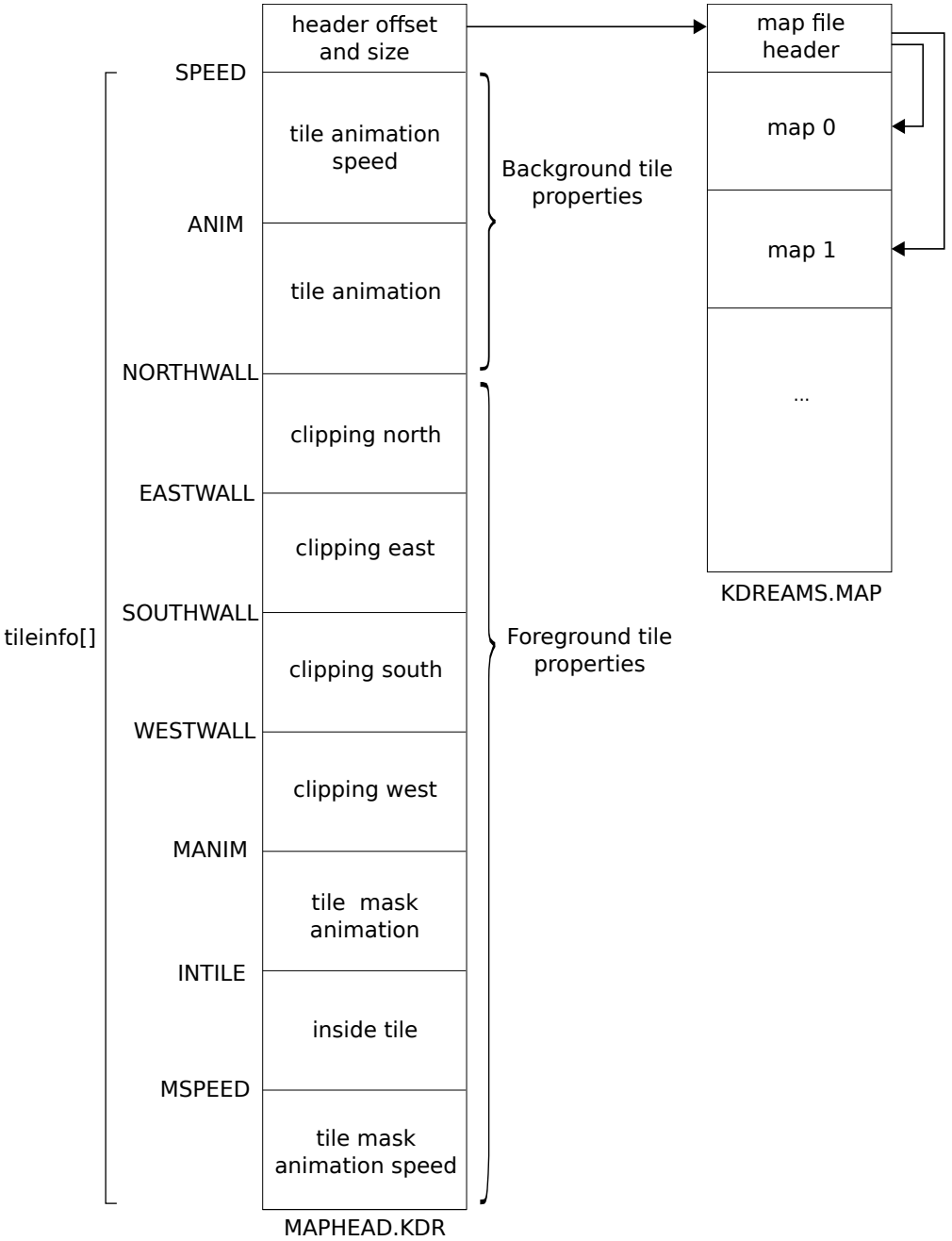


Figure 3.12: Structure of `MAPHEAD.KDR` header file.

3.3.2 Map archive structure

The structure of the KDREAMS.MAP archive is illustrated in Figure 3.13. 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;
```

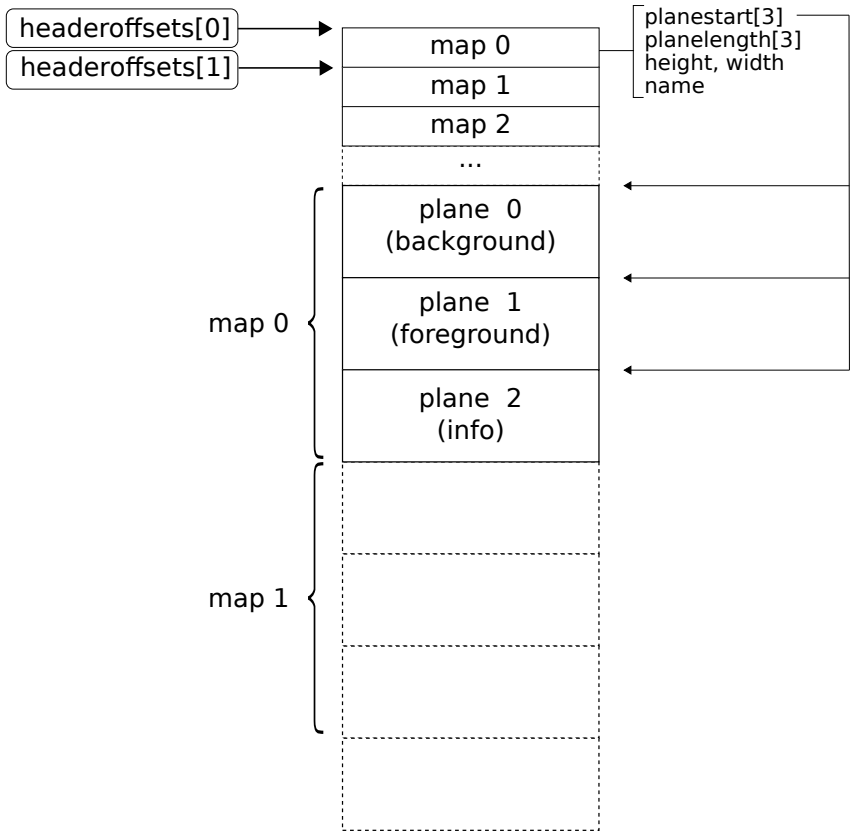


Figure 3.13: Structure of KDREAMS.MAP file.

3.4 Audio

The original Commander Keen Trilogy only supported the default PC speaker. With the release of Commander Keen in Keen Dreams, the team decided to add support for sound cards as well.

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

id Software intended Keen Dreams to include music for the Sound Blaster and 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 360KiB floppy disk, and in order to do this, id Software had to scrap the game's music at the last minute[4]. The team did not even have time to remove the music setup menu.

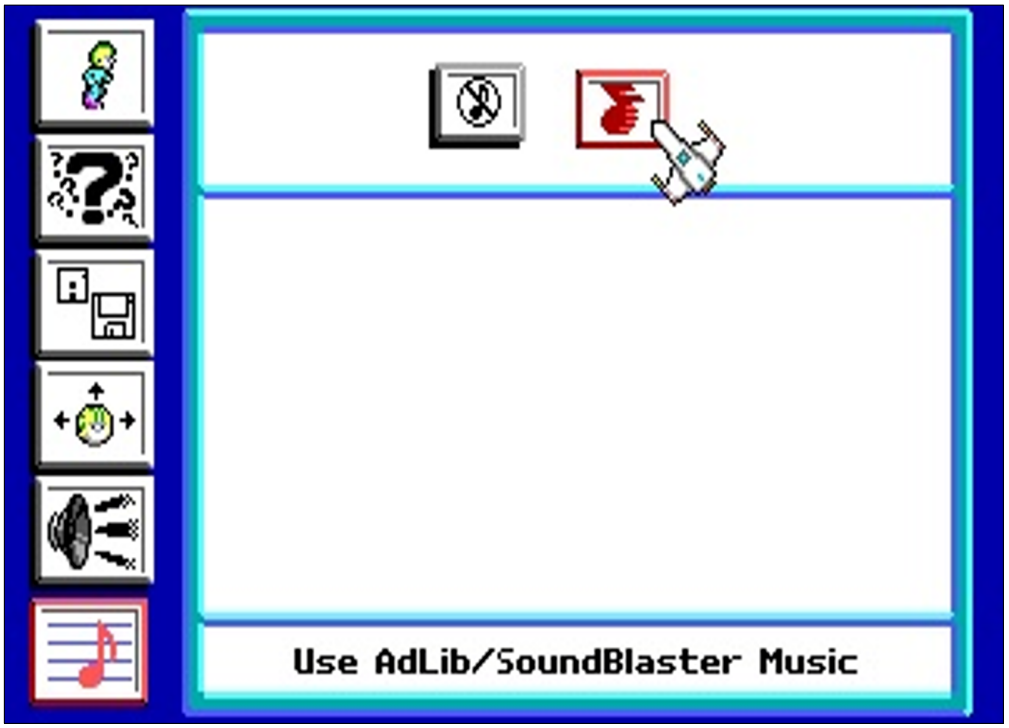


Figure 3.14: Music setup in Keen Dreams, even though the game contains no music.

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

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*.

As a result, the game shipped with sound effects only. The sound menu also suggests that digitized sound effects were planned, although they were never included.

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

Consequently, only two versions of each sound effect were included with the game:

1. For PC speaker
2. For AdLib

All sound effects were done by Robert Prince. Sound effects were created using Cakewalk and the inhouse MUSE tool[5]. The MUSE tool packed all sound effects together in a compressed `SOUND` archive and generated `HEAD` and `DICT` files and a C header file with asset IDs.

> Muse — Torture in the form of an executable. Oh, yeah, you can edit PC and AdLib sounds in it. And import music. And see a digitized sound import that doesn't do anything. And see funny blue lines march across your screen. And save out five meg sound files. And have two red lines that show the end of the sound. And see a red line and no sound, though you can play it. And have all the maximum values suddenly disappear.
>
> All these features, and no damn cut and paste! For a fuckin' year!!! (But i'm not bitter ...)
>
> **Tom Hall[6]**

[5] See *Game Engine Black Book - Wolfenstein 3D*, section 3.6.
[6] Tom Hall, The Doom Bible (http://5years.doomworld.com/doombible/).

3.5 Distribution

On December 14, 1990, the first episode was released via Apogee. Episodes 1–5 were all published by Apogee Software. The game engine and first episode were distributed for free and were intended to be copied and shared as widely as possible. 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[7].

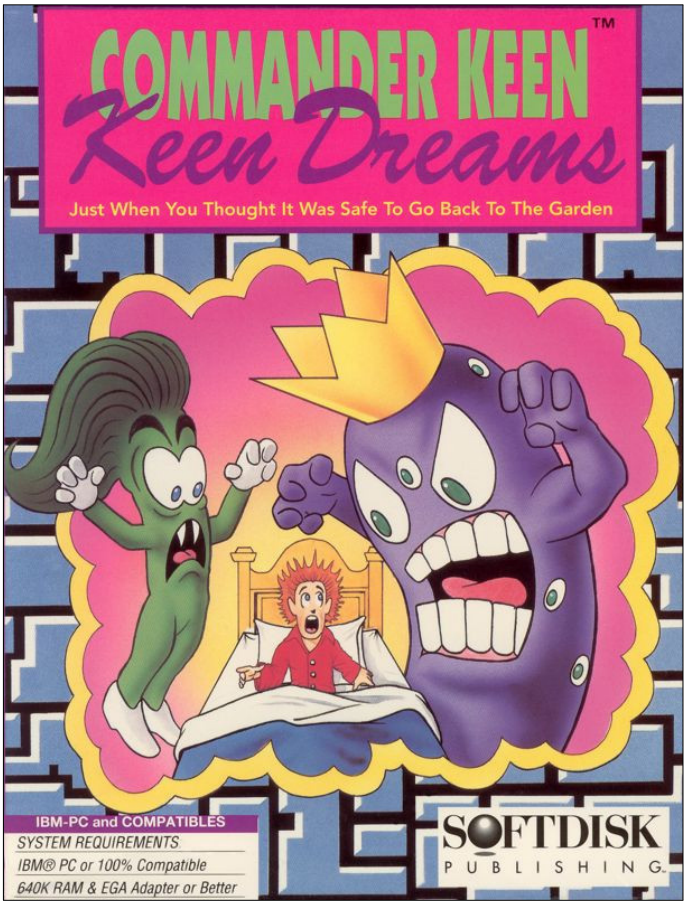


Figure 3.16: Retail version of Commander Keen in Keen Dreams by Softdisk.

[7]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 into seven parts:

- KDREAMS.EXE: Game engine.
- KDREAMS.EGA: Contains all the assets (sprites, tiles) needed during the game.
- KDREAMS.AUD: Sound effect files.
- KDREAMS.MAP: Contains all maps.
- 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 by executing the corresponding *.BAT file.

```
 Directory of C:\KDREAM

.            <DIR>           10-11-2025 21:39
..           <DIR>           08-11-2025 14:07
BKGND    SHL               285 10-11-2025 21:39
FILE_ID  DIZ               508 10-11-2025 21:39
HELP     BAT                34 10-11-2025 21:39
HELPINFO TXT             1.038 10-11-2025 21:39
INSTRUCT SHL             2.763 10-11-2025 21:39
KDREAMS  AUD             3.498 10-11-2025 21:39
KDREAMS  CFG               656 10-11-2025 21:42
KDREAMS  CMP            14.189 10-11-2025 21:39
KDREAMS  EGA           213.045 10-11-2025 21:39
KDREAMS  EXE            81.619 10-11-2025 21:39
KDREAMS  MAP            65.673 10-11-2025 21:39
LAST     SHL             1.634 10-11-2025 21:39
LICENSE  DOC             8.347 10-11-2025 21:39
LOADSCN  EXE             9.959 10-11-2025 21:39
MENU     SHL               447 10-11-2025 21:39
NAME     SHL                21 10-11-2025 21:39
ORDER    SHL             1.407 10-11-2025 21:39
PRODUCTS SHL             4.629 10-11-2025 21:39
QUICK    SHL             3.211 10-11-2025 21:39
README   TXT             1.714 10-11-2025 21:39
START    EXE            17.446 10-11-2025 21:39
VENDOR   BAT                32 10-11-2025 21:39
VENDOR   DOC            11.593 10-11-2025 21:39
VENDOR   TXT               810 10-11-2025 21:39
   24 File(s)          444.558 Bytes
    2 Dir(s)         733,29 G Bytes free

C:\KDREAM>_
```

Figure 3.17: All Keen Dreams files as they appear in DOS command prompt.

Chapter 4

Software

4.1 About the source code

The source code for Commander Keen Episodes 1–5 is unavailable, as the current owner Zenimax[1] has shown no interest at the time of writing in selling the intellectual property. Fortunately, 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 launched to purchase the rights from Flat Rock for US$1500 in order to release the game's source code and publish it on multiple platforms. The campaign did not reach the goal, but its 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 shareware version 1.13, otherwise you may encounter compatibility issues due to incompatible map and graphics headers[2]. To get the correct source code, run the following commands in the console

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

[1] June 24, 2009, id Software was acquired by ZeniMax Media (owner of Bethesda Softworks).
[2] See issue #7 on https://github.com/keendreams/keen.

4.3 Exploring the source code

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 used for performance-critical optimizations and low-level I/O operations such as video and audio handling.

Source lines of code (SLOC) is helpful for understanding the relative size of the codebase compared to modern software. For instance, Inkscape (a vector graphics editor tool I used to create the drawings in this book) has 544,508 SLOC, Unreal Engine 4 has 2,300,000 SLOC and Microsoft Office 2013 is estimated to have 45,000,000 SLOC. This illustrates how small Commander Keen is compared to modern software codebases.

Trivia : The game *Red Dead Redemption II* is estimated to have 60 million SLOC, the highest amount for a game at the time of writing this book.

Besides the source code, the repository also contains two folders featuring:

- A `static` folder with header and dictionary files for loading assets and maps, and a tool `makeobj.c` to convert these files into OBJ-files (this is explained in the next section).
- An `lscr` folder for loading and decompressing Softdisk text files.

4.4 Compiling source code

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

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

After setting up DOSBox and installing Borland C++ 3.1[3], download the source code from GitHub. Once DOSBox is running and you have navigated to the `keen` folder, create a folder to store your compiled object files.

```
mkdir OBJ
```

The `static` folder contains the Huffman dictionaries (`DICT`) and headers (`HEAD`) from the generated graphics, sound and map assets, stored as raw game data files. The team created a special tool called `makeobj` that takes these data files and wraps them into DOS-compatible `.OBJ` files, so that a C compiler (such as Borland C++) can link them directly into the game's executable. The tool creates a "symbol" within the object file so that the game's source code can access the embedded data just like any other global variable.

```
REM Create EGAhead as global variable in the source code
makeobj f EGAHEAD.KDR ..\kdrehead.obj EGA_grafixheader
    _EGAhead
```

To generate all static `OBJ` header files, simply enter the following command:

```
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 game executable is stored in the \OBJ folder. The final step is to copy `kdreams.exe` into the Keen shareware folder. You can now play your compiled version of *Commander Keen*.

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

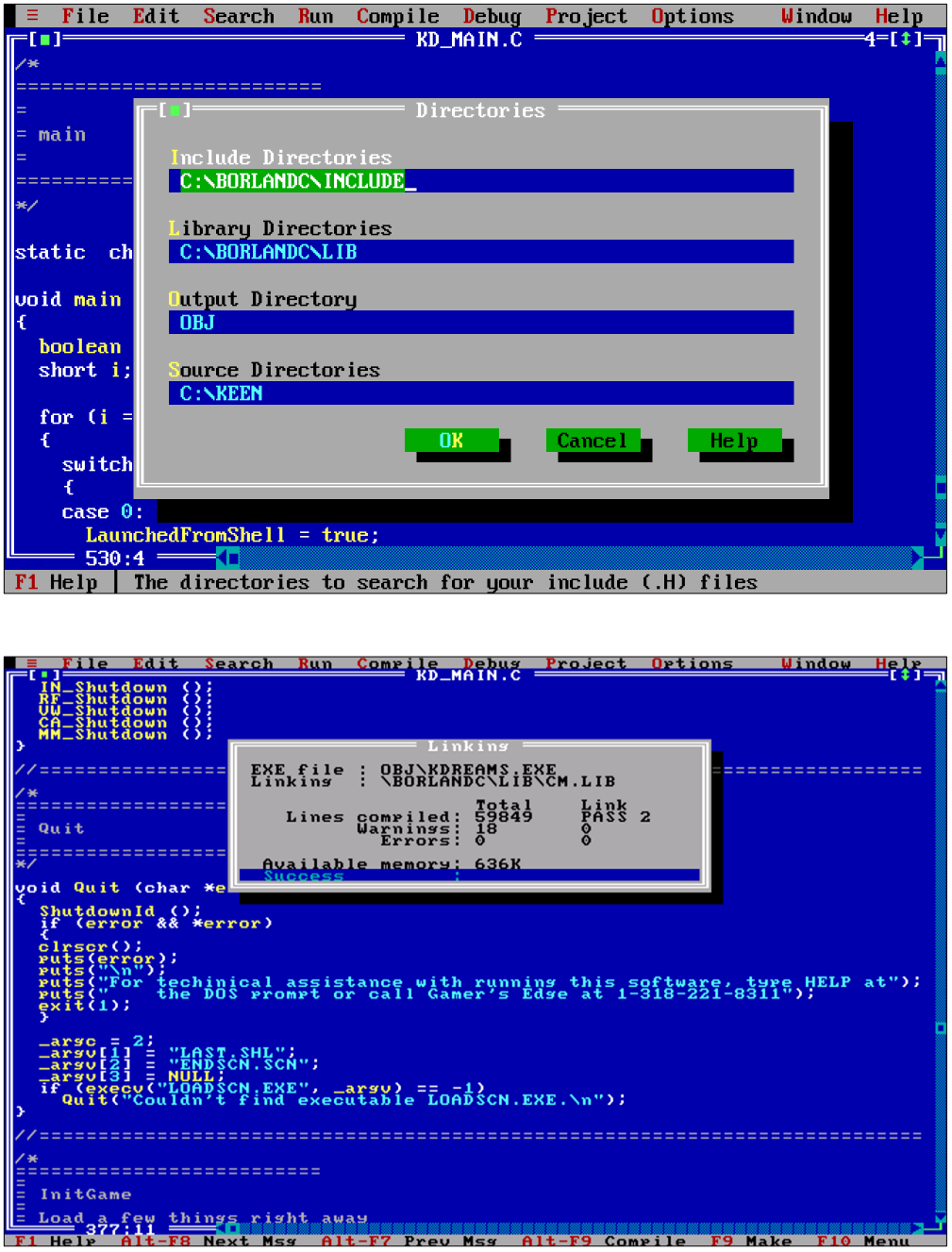


Figure 4.1: Compiling Commander Keen in Borland C++ 3.1.

4.5 Executable compression

Checking the size of the compiled program, we see

```
KDREAMS.EXE          357,293 Bytes
```

As mentioned earlier, Softdisk did not allow distribution of games larger than 360 KiB. The compiled executable alone already occupies 349 KiB, and it still needed to accommodate graphics, map, and audio assets. With only 11 KiB remaining, the team needed a way to compress the executable to ensure everything would fit on one floppy disk.

To compress the executable, the team used a small utility called LZEXE. It was one of the first widely used executable file compression tools for DOS computers. LZEXE allowed DOS executables to be launched without requiring an explicit decompression step. Most other compression or packing utilities required an additional step (and often additional disk space) to accomplish this.

> “ I wrote LZEXE in 1989 and 1990 when I was 17. At that time, hard disks were small and expensive, and 5¼-inch floppies were small (360KiB). Since I had only two floppy drives on my PC (an Amstrad PC1512), saving space was really an issue.
>
> Although I wrote LZEXE for my own use, I gave it to some friends, and it was then put on some BBS's. LZEXE became then very famous, although I did not do anything to promote it. This success was quite unexpected for me.
>
> **Fabrice Bellard, LZEXE creator**[4] ”

The compression engine was built around the Lempel–Ziv–Storer–Szymanski (LZSS) algorithm, and the decompressor was designed to operate without using more memory than the final decompressed program required. This was more difficult than it might seem at first, because the tool had to ensure that there was never a moment when decompressed data overwrote an unread portion of the compressed data.

An LZEXE-compressed program is a standard DOS executable, loaded in the standard way without any special trickery. Each executable file contains an EXE header, which specifies the initial layout of the executable in memory. One of its tasks is to initalize the stack, which is set up to a high area of uninitialized memory.

[4] See homepage of Fabrice Bellard: https://bellard.org/lzexe/.

The first task performed by the program is to move itself, including the decompression code and relocation table. This step is necessary because, if the compressed payload were decompressed in the same location where DOS originally loaded it, the decompressor would begin writing decompressed program data faster than it could read the compressed data, destroying its own source material.

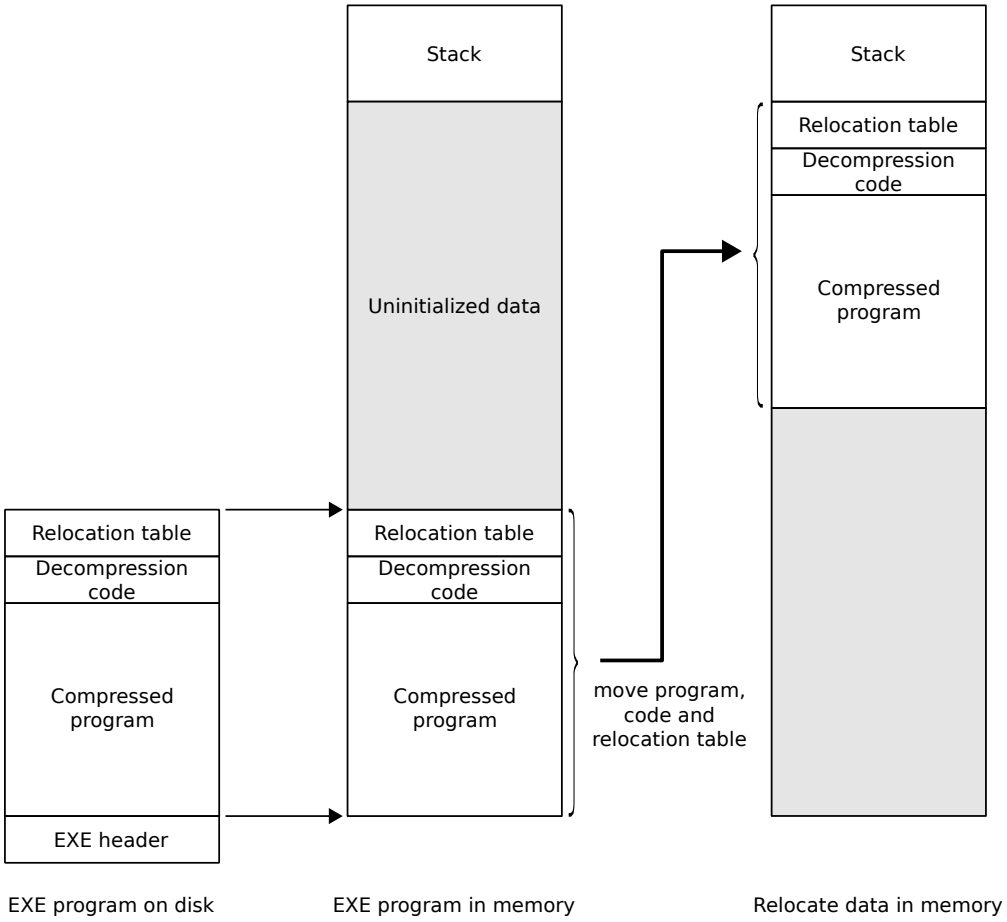


Figure 4.2: Loading and relocating EXE program in memory (simplified).

Once relocated, the decompression code begins reading the compressed program and decompresses it back into the program's original starting address. The heart of the decompressor is the LZSS algorithm, which operates by replacing repeated occurrences of

data with references to earlier instances of that same data in the decompressed output[5].

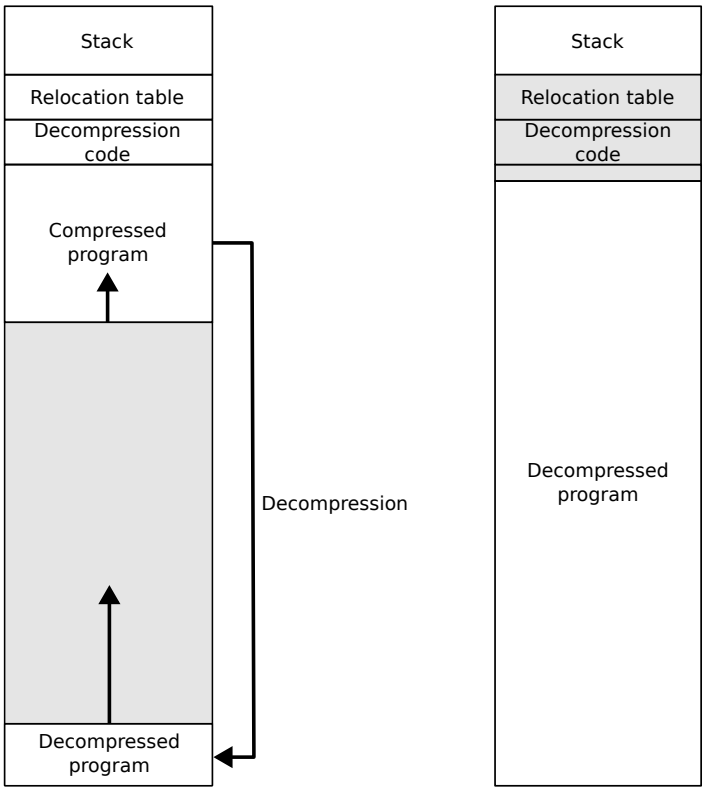


Figure 4.3: Decompression and final result in memory.

After the entire program has been decompressed, special attention is required for `JMP` (jump) and `CALL` instructions. Normal program execution is linear, with each instruction directly following the previous one. Jump and call instructions interrupt this flow and force the program to branch to arbitrary absolute addresses in the code. However, under DOS there are no guarantees about the memory address at which an executable will be loaded. Depending on DOS itself, device drivers, reserved memory regions, and other factors, this address can — and frequently does — change from run to run.

To solve this problem, DOS uses a relocation table. When an EXE file is created, all absolute addresses in the code are calculated under the assumption that the program will be loaded at address `0000:0000h`, the absolute beginning of memory. The relocation table contains a list of locations within the executable where such absolute addresses appear.

[5]See https://cosmodoc.org/topics/lzexe/ for a detailed explanation of LZSS compression.

The DOS loader iterates through this table and patches each address ① by adding the program's actual load address ②.

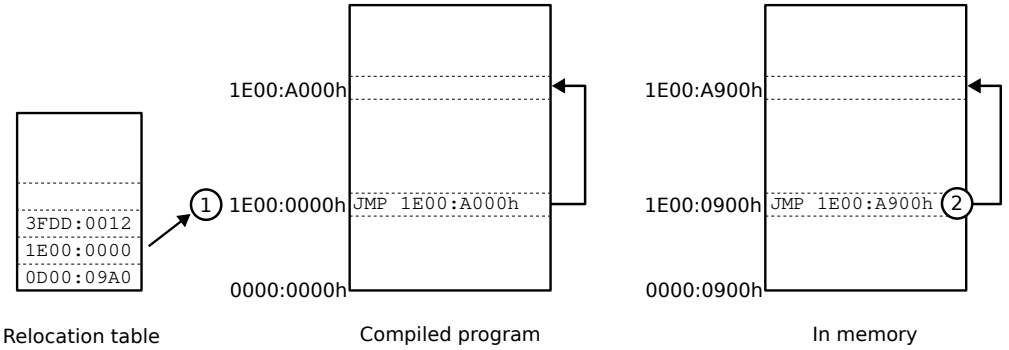


Figure 4.4: DOS relocation table.

DOS stores each relocation table entry as a 4-byte segment:offset pointer. This takes up quite a bit of space, since an average game has over 1,300 relocation table entries, which would occupy well over 5 KiB of space using the DOS scheme. LZEXE, by contrast, encodes only the distance from the previous fixup. This results in a far more efficient format, where most relocations require only a single byte. In practice, an LZEXE-compressed relocation table can be one-quarter the size of the equivalent DOS relocation table.

At this point, memory contains exactly what it would have contained if the executable had never been compressed with LZEXE in the first place. And that's how it all worked — everything happening in the split second between the user pressing the Enter key at the DOS prompt and the screen clearing to launch the game. The executable file is ultimately compressed to 80 KiB, achieving a 77% reduction in file size.

> 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] History of Commander Keen on https://legacy.3drealms.com/keenhistory.

4.6 Main overview

The game engine is divided into three blocks:

- Control panel, which lets players configure and start the game.
- 2D game engine, where players spend most of their time.
- Sound system, which runs concurrently with either the control panel or 2D engine.

The 2D engine communicates with the sound engine via function calls. The 2D engine sends sound requests to the sound engine, which processes them in its main loop. The sound engine controls the game's heartbeat. It updates the global `TimeCount` variable, which is used to time-control the game flow and screen refresh.

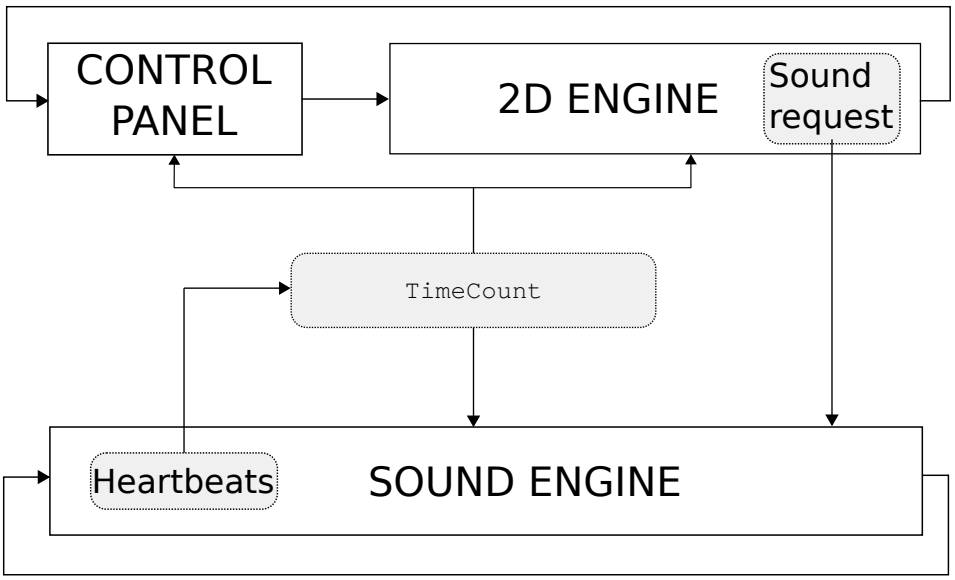


Figure 4.5: Game engine three main systems.

4.6.1 Unrolled Loop

The first programmer-defined function that runs in any C program is named `main()`, which resides in `kd_main.c`. When started with the `/?` argument, the program displays an overview of available configuration parameters, such as disabling sound cards or mouse. The control panel and 2D engine follow regular loops, but due to limitations explained later, the sound system is interrupt-driven and therefore operates outside the main loop.

The first action taken by the program is to set the text color to light grey and the background to black, followed by game initialization, after which the main loop begins.

```
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. Next, it loads all sounds and frequently used graphics (e.g. font and Commander Keen sprites) into memory.

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

  MM_Startup ();     // Memory Manager
  if (mminfo.mainmem < 3351*1024)
  { ... (Not enough memory)}

  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 ();

  // load in and lock down some basic chunks
  CA_CacheMarks (NULL, 0);
  CA_LoadAllSounds ();
  //wait for a keypress and then clears the screen
  US_FinishTextScreen();
}
```

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

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

`PlayLoop` contains the 2D engine. It is pretty standard with getting input, updating the state machine, scrolling, and refreshing the screen.

```
void PlayLoop (void)
{
  FixScoreBox ();     // draw bomb/flower scorebox
  do
  {
    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);

    ScrollScreen(); // Scroll if Keen is nearing an edge.
    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)`. The reason for interrupts is extensively explained in Chapter 4.13 "Audio and Heartbeat". In short, with DOS supporting neither processes nor threads, it was the only way to have something executed concurrently with the rest of the engine.

4.7 Architecture

The source code is structured in two layers. The ID_* files are sub-systems called managers interacting with the hardware. These ID_* managers were generic and reused in other id Software games (e.g. *Hovertank One*, *Catacomb 3-D* and *Wolfenstein 3D*). The KD_* files were written specifically for Commander Keen.

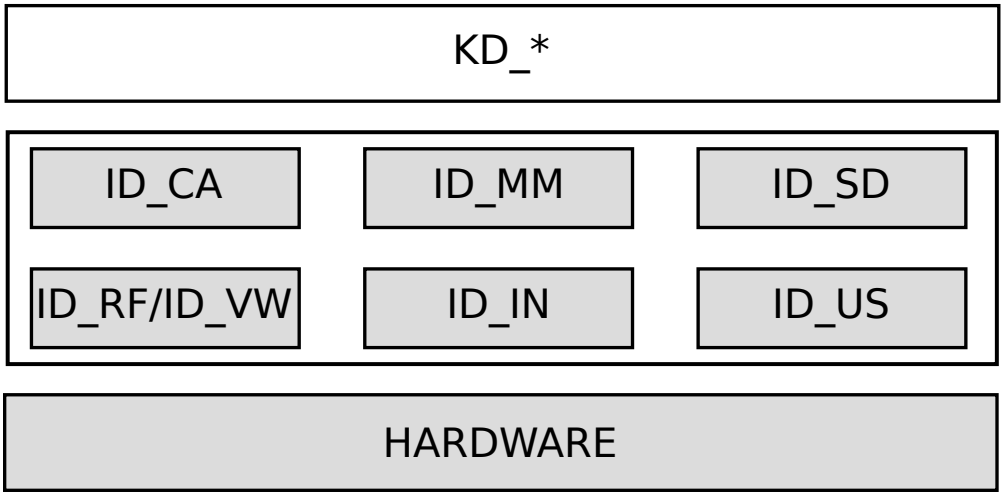


Figure 4.6: Commander Keen source code layers.

There are six managers in total:

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

The figure below shows a more detailed architectural overview, illustrating the interaction between the different sub-systems. Next to the hard drive (HDD) you can see the graphics, map and sound assets files as described in Chapter 3.

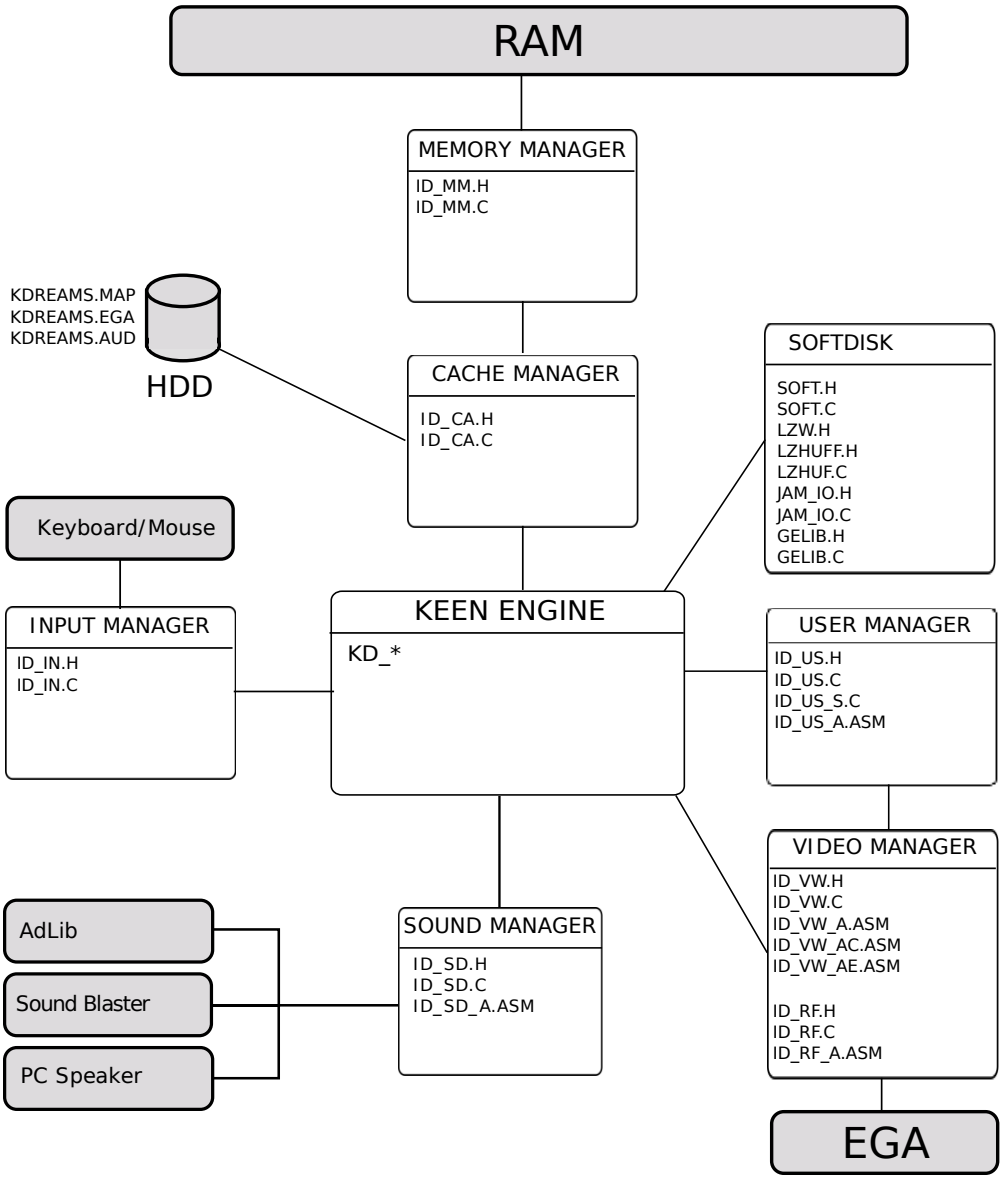


Figure 4.7: Architecture with engine and sub-systems connected to devices (in gray).

4.7.1 Memory Manager (MM)

The engine has its own Memory Manager to have more control over memory fragmentation and optimization. The Memory Manager consists of a linked list of "blocks" keeping track of 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 in the data segment 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, called "paragraphs".

```
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 always be the first node
mmnew->start = 0;
mmnew->length = segstart;
mmnew->attributes = LOCKBIT;
```

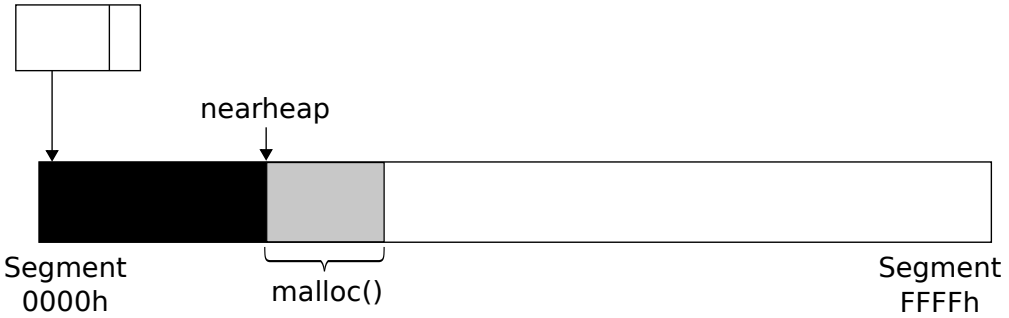


Figure 4.8: First locked block of unusable memory.

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 allocating the far heap, which can be obtained by

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

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

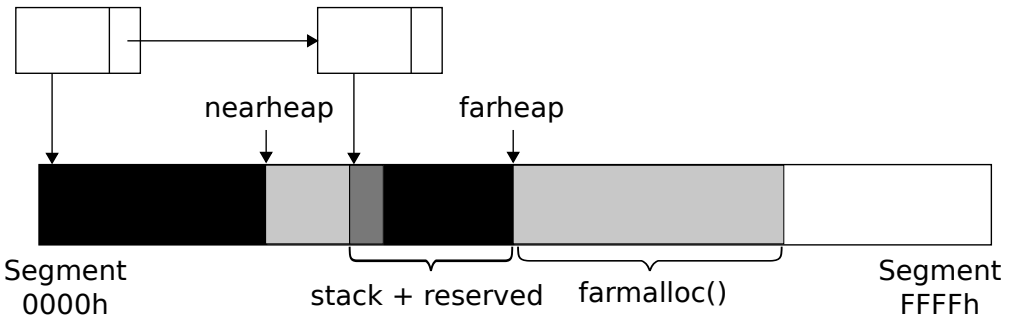


Figure 4.9: Second locked block: stack.

Finally, the last unusable memory block 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.

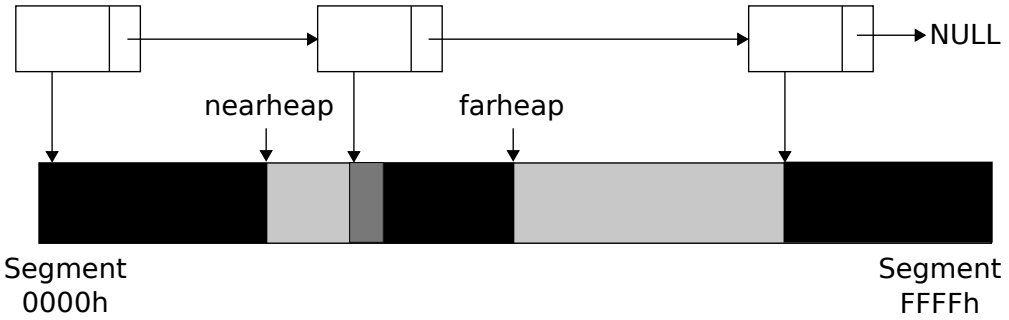


Figure 4.10: Final initial Memory Manager state.

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[7].

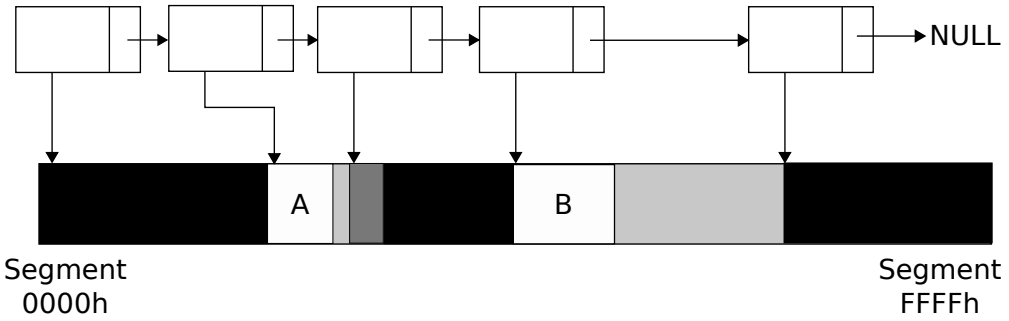


Figure 4.11: Allocation of memory.

The current memory usage during gameplay can be inspected by calling `MM_ShowMemory`. Each pixel represents one memory paragraph of 16 bytes. Red indicates locked 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!

[7]See the "Game Engine Black book Wolfenstein 3D" for a detailed description how the Memory Manager allocates memory

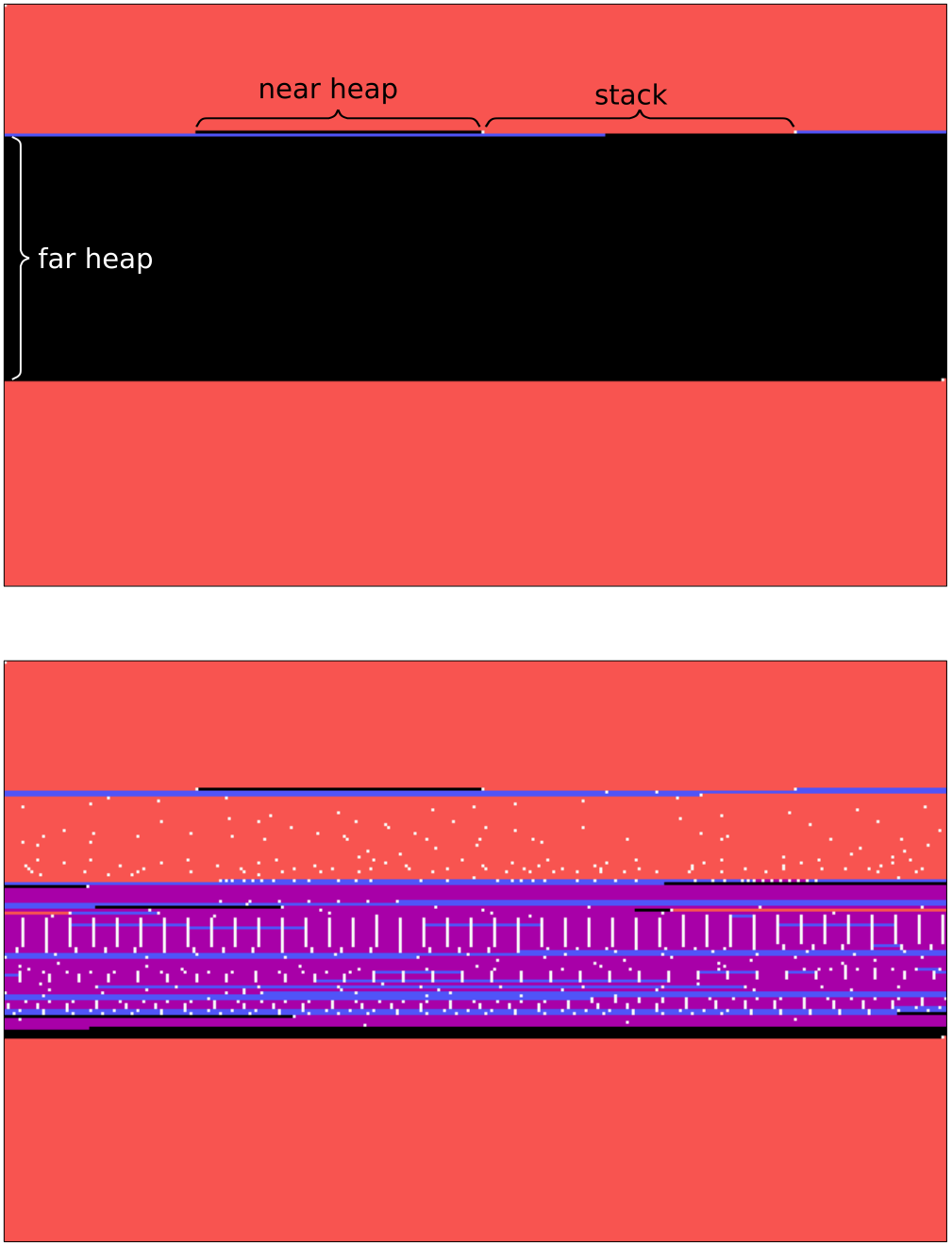


Figure 4.12: Memory dump after (i) initializing Memory Manager and (ii) running the game.

4.7.2 Video Manager (VW & RF)

The Video Manager consists of 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.

4.7.3 Cache Manager (CA)

The Cache Manager loads and decompresses maps, graphics and audio resources stored on the filesystem and makes them available in RAM. Assets are stored in 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

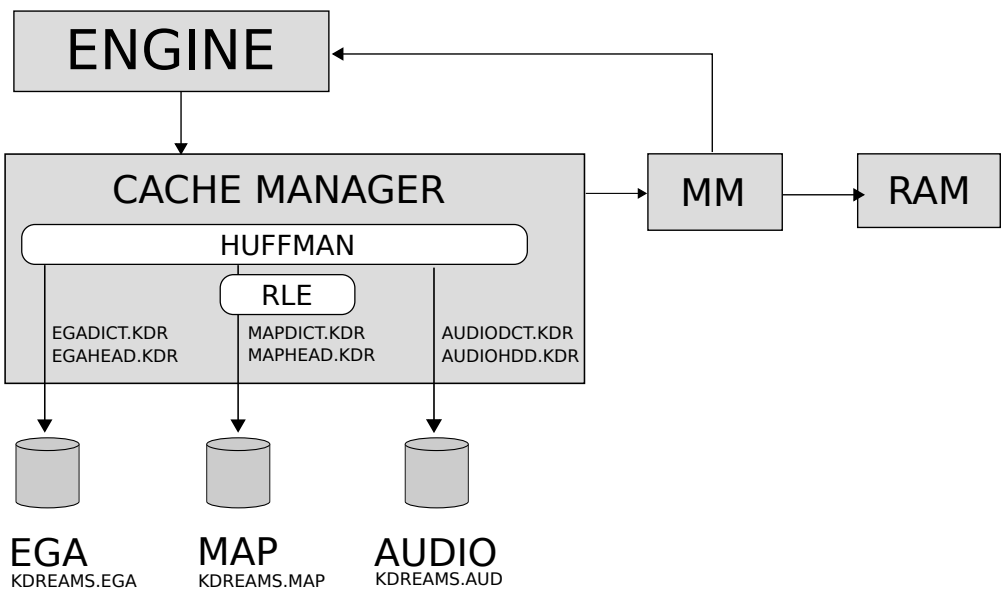


Figure 4.13: Cache manager architecture.

The header (`HEAD`) and dictionary (`DICT`) files are provided in the `static` folder after the conversion into `*.OBJ` file using `makeobj.c`. 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 on page 106.

4.7.4 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 performs all measurements (`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 4.14.

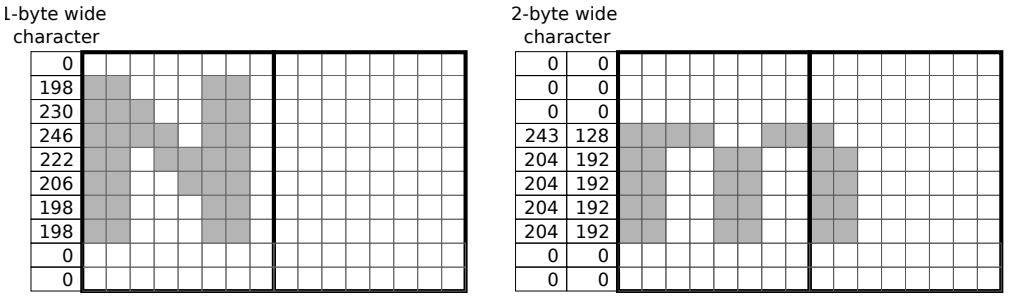


Figure 4.14: Character bitmaps of 'N' (7 bits wide) and 'm' (11 bits wide)

On the EGA video card each 8 pixels take up 1 byte of VRAM, as explained on page 47. When you 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. The process works as follows:

1. Start with a base table (called `shiftdata0`) that contains all possible byte values, from 0 to 255, each stored as a 16-bit integer.

2. Next, seven additional tables are generated by shifting each value in `shiftdata0` one bit to the right for each successive table. For example, in `shiftdata1` the value 198 becomes 99, and in `shiftdata2` it becomes 32817, and so on.

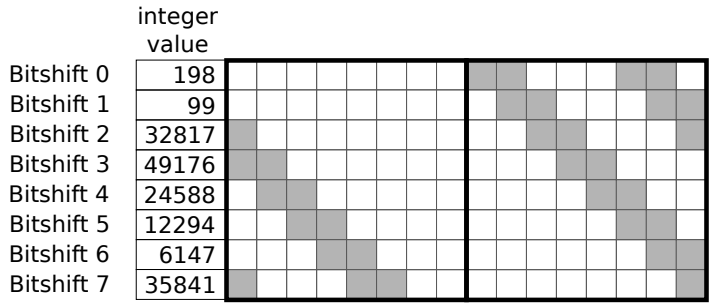


Figure 4.15: Right bitshift for integer 198.

The pre-calculated bitshift tables are stored in `id_vw_a.asm`, below the `bitshift3` table. Figure 4.16 illustrates how a 3-pixel shifted 'N' is generated using this 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
```

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

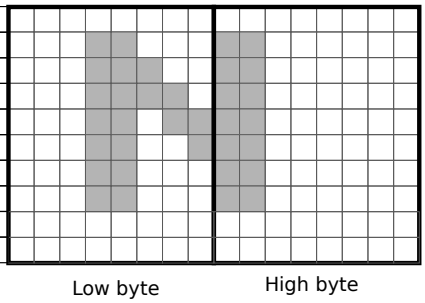


Figure 4.16: Bitshift 'N' over 3 bits using bit shift tables.

The assembly routine `ShiftPropChar` takes care of drawing the character to the `(x,y)` location on screen. By sacrificing 4096 bytes of memory, a lot of CPU time is saved.

```
MACRO SHIFTNOXOR    ; Macros to table shift a byte of font
  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

PROC  ShiftPropChar NEAR
  [...]
; 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] ;pointer to shift table
  [...]

@@loop1:
  SHIFTNOXOR
  add di,dx         ; next line in buffer
  loop  @@loop1
  ret
ENDP
```

4.7.5 Sound Manager (SD)

The Sound Manager takes care of interaction with the different sound systems: PC speaker, AdLib and Sound Blaster. It is not part of the game engine loop, but runs on a separate loop via an interrupt system. Where the game engine runs at a maximum of 70Hz, the Sound Manager is capable of running between 140Hz to 700Hz. The Sound Manager is described extensively in the "Audio and Heartbeat" section on page 160.

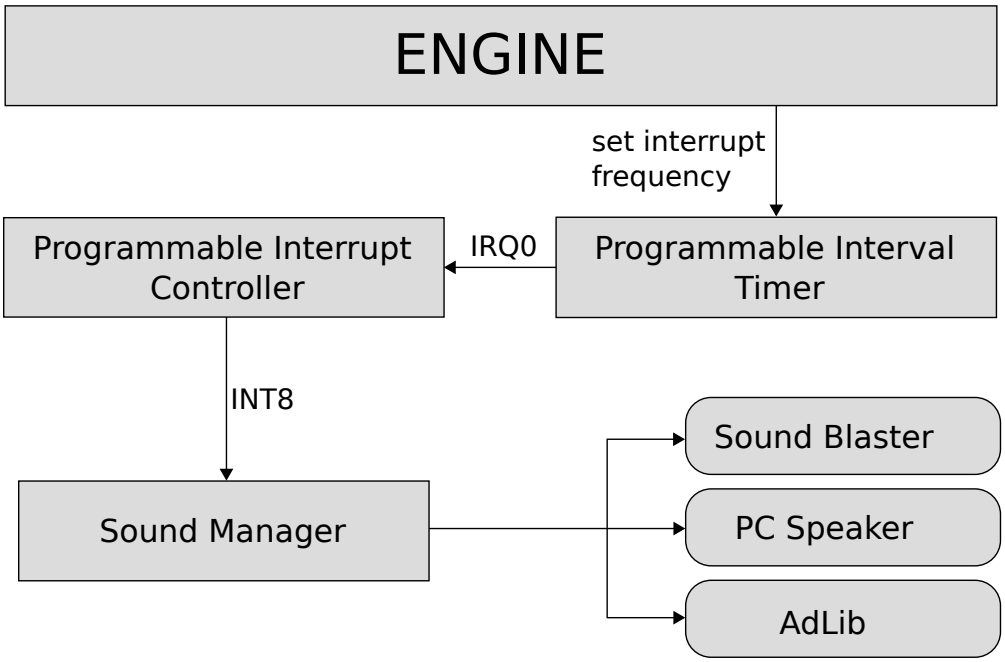


Figure 4.17: Sound system architecture.

4.7.6 Input Manager (IN)

The Input Manager is responsible for handling input from the keyboard, mouse, and joystick. It manages hardware-level details such as I/O port addresses, scan codes, and custom interrupt routines to ensure reliable communication between the input devices and the game engine.

Because most players used the keyboard to play Commander Keen, section “Inputs” on page 176 provides a detailed explanation of the keyboard input system and its implementation.

4.7.7 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.

4.8 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 continue. However, the game will likely crash or display an error shortly thereafter.

```
Out of memory! Please make sure you have enough free memory
```

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 64KB) is unloaded from RAM to free up memory for runtime, and the control panel is displayed.

Figure 4.18: Keen Dreams intro screen

4.9 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.

4.9.1 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								
TILE16 #3								
...								

Figure 4.19: Initiating `grneeded[]` array, current cache level is 1.

When new graphics for a map need to be cached in memory, all required assets are marked by setting the current level bit (bit 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								
TILE16 #3								■
...								

Figure 4.20: Mark and load assets required for the new map in cache level 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.

```
void CA_CacheMarks (char *title, boolean cachedownlevel)
{
  // save title so cache down level can redraw it
  titleptr[ca_levelnum] = title;

  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 already 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;
  [...]
  //
  // go through and load in anything still needed
  //
  for (i=0;i<NUMCHUNKS;i++)
    if ( (grneeded[i]&ca_levelbit) && !grsegs[i])
    {
            //load asset from disk into memory buffer
            lseek(grhandle,pos,SEEK_SET);
            CA_FarRead(grhandle,bufferseg,endpos-pos);
            source = bufferseg;

            CAL_ExpandGrChunk (i,source);   // decompress
    }
  }
}
```

Now, during gameplay, the user opens the control panel (e.g. to pause the game), which

requires loading new 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.

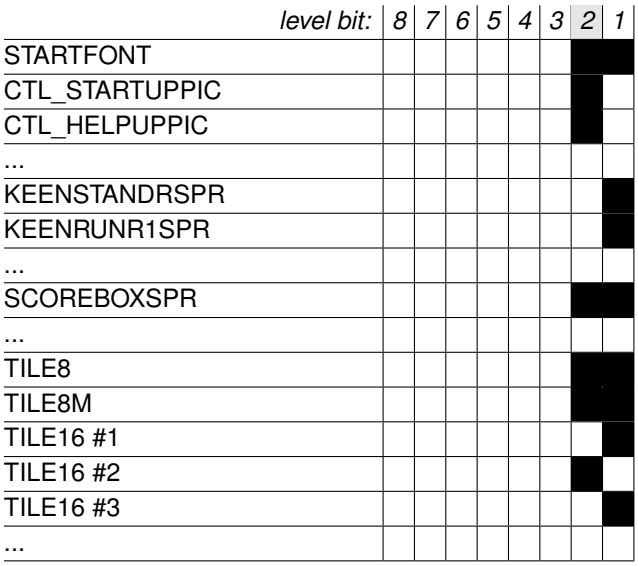


Figure 4.21: Mark all assets required for the control panel in cache level 2.

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` and `TILE16 #3`.

When the user closes the control panel, the engine lowers the cache level back to 1, the cache level where all assets for the current map are memorized, by calling `CA_DownLevel`. After the cache is lowered, `CA_CacheMarks` is called again to purge unnecessary assets and reload 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 everything from the previous level
  CA_CacheMarks(titleptr[ca_levelnum], 1);
}
```

To prevent frequently used assets—such as fonts and Commander Keen sprites—from being reloaded from disk repeatedly, they are kept permanently in memory by flagging them as locked memory (This is illustrated by the red block after the stack in Figure 4.12).

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

4.9.2 Asset compression

Given that the floppy disk is limited in both speed (100-250 kbps[8]) and storage capacity (3½-inch disk size is either 720KiB or 1.44MB), file compression was essential. Compression ensures that the game occupies less space and loads more quickly. Ideas from the Deep 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
```

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

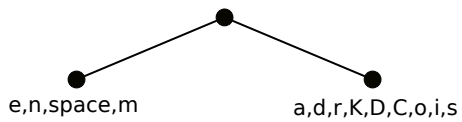
The first step is to make a character frequency table.

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

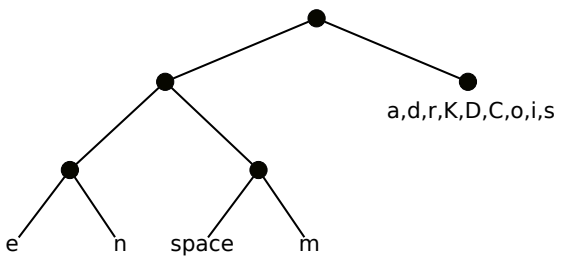
Character	Frequency
d	1
D	1
C	1
o	1
i	1
s	1

Figure 4.22: Character frequency table.

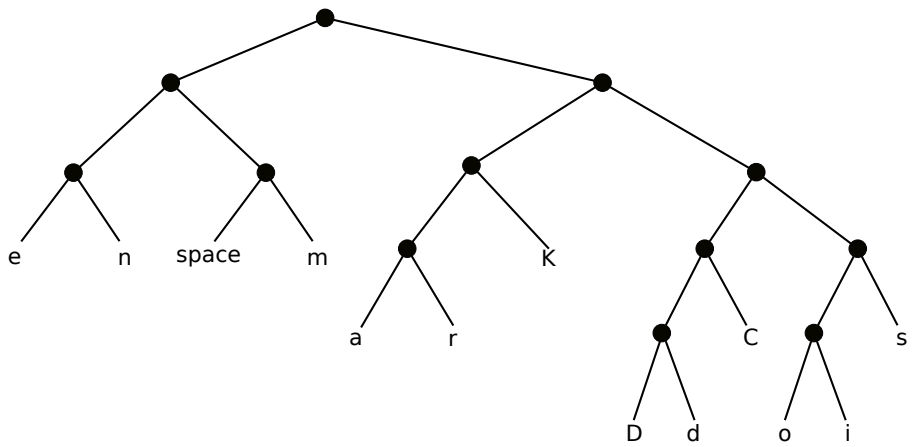
Next, an optimal binary tree, also known as a dictionary, is created. This is done by starting with the most common character and checking if it occurs more frequently than all the other characters combined. If not, then the second most common character is added, and so on. In our example, four characters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



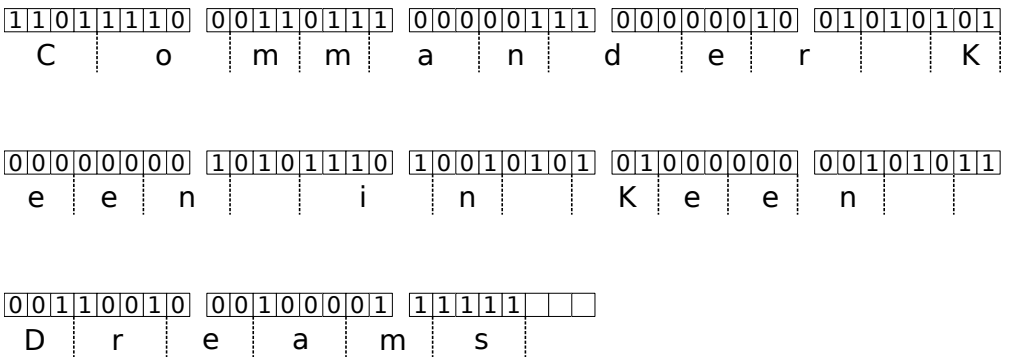
The process continues by creating a new node with left and right branches and repeating the steps. The two most common characters in the left node, 'e' and 'n', are assigned to the left branch of this new node. A third node is created, and since there are only two options, each is given a branch. This 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[9].

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

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.

```
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 map, you'll notice large chunks of the same tile. For example, consider the first map you enter (Horse Radish Hill), which contains extensive areas of blue sky. In this case, *Run-length encoding* (RLE) compression is particularly effective.

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 word-size data, and therefore is named RLEW. The RLEW tag in Commander Keen is defined as `ABCDh`.

The overall result of applying Huffman and RLEW compression results in almost 65% asset file size reduction. Together with the LZEXE-compressed executable file (80KiB), the entire game fits nicely on a 360KiB floppy disk, just as was required by Softdisk.

Asset type	**filename**	**Uncompressed size**[10]	**Compressed size**
Graphic assets	KDREAMS.EGA	354,075 bytes	213,045 bytes
Game maps	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

4.10 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 second.

```
# 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**[11] ”

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

- By exploiting specific EGA hardware tricks.
- Updating only the portions of the screen that had actually changed between frames

[10] See Appendix A for details per asset file.
[11] Interview with Lex Fridman in 2022, this quote is around 1h:41m.

In the later episodes of Commander Keen, a different – and significantly simpler – approach was adopted for updating the display. Each of these solutions is described in detail in the following sections, starting with the EGA hardware.

4.10.1 Moving one pixel at a time in EGA

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.

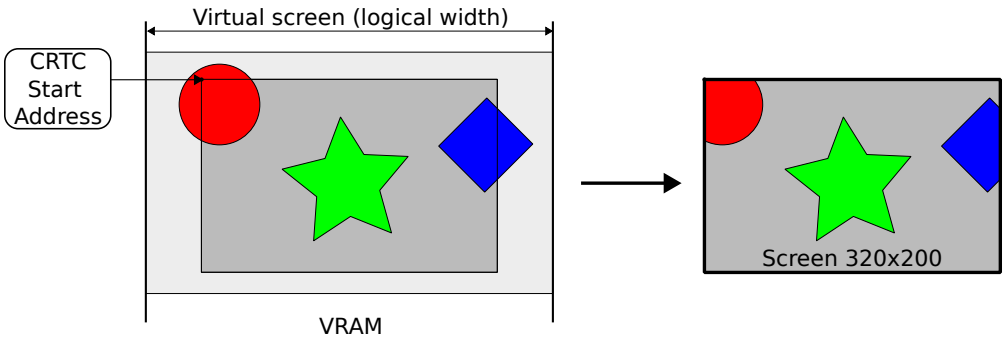


Figure 4.23: Virtual screen in VRAM.

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
;==============

  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
```

To pan down a scan line, it only requires increasing the CRTC start address by the logical width.

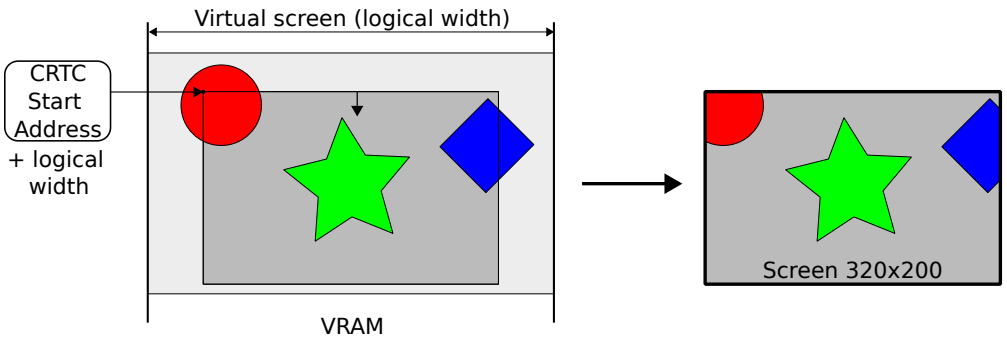


Figure 4.24: Move screen one pixel down.

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 smooth horizontal scrolling. Fortunately, another EGA register addresses this limitation.

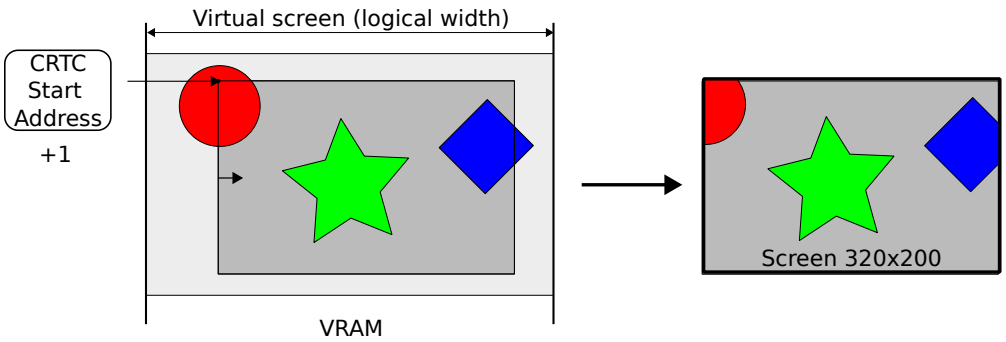


Figure 4.25: Move screen eight pixels (1 byte) right.

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. 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 sent 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 ;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
```

Trivia : "Pel" stands for "Picture element" and was introduced alongside pixel in the 1960s. While pixel became the standard term in graphics and modern computing, pel was frequently used in early image processing, computer technical references (notably IBM), and video coding.

By combining both EGA hardware tricks, smooth scrolling on EGA is achieved by adjusting the CRTC start address and fine-tuning the horizontal display using the horizontal pel panning register. The process consists of the following steps:

- Calculate the panning offsets in pixels for both the x and y directions.
- Achieve smooth vertical scrolling by adding *(logical width × y)* to the CRTC start address.
- For coarse horizontal scrolling in 8-pixel increments, increase the CRTC start address by *x / 8* bytes.
- Finally, apply fine horizontal pixel alignment by writing *(x % 8)* to the horizontal pel panning register.

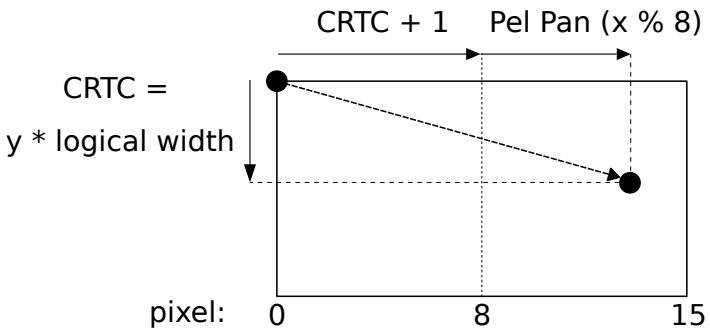


Figure 4.26: Smooth scrolling in EGA.

The engine is now capable of moving the entire screen one pixel at a time in all directions, just by tweaking the EGA registers, enabling smooth scrolling across the display.

4.10.2 Adaptive Tile Refresh

So far, we have built a virtual screen in VRAM allowing smooth one pixel moves in both axes. However, once the virtual screen edge is reached, a full-screen redraw would be too slow, causing a drop to 5 fps.

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. unfortunately, loading the entire level requires far more VRAM than is available. The first level, *Horse Radish Hill*, is 136 tiles wide and 37 tiles high. Each tile is 128 bytes, which means a total of $136 \times 37 \times 128$ bytes = 644KB is required to store the entire level in VRAM. Since we only have 256KiB 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[12]. Let's look at *Commander Keen 1: Marooned on Mars* in Figure 4.27 on the next page. 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 total, only 69 tiles out of the 260 tiles need to be refreshed, which is 27% of the screen!

> "The trick behind the scrolling in the first Commander Keen games was to only redraw tiles that actually changed after panning 16 pixels, since most maps had large swathes of constant background.
>
> Text consoles have similar statistics with spaces, so the same trick is a big win.
>
> **John Carmack[13]**"

This is also the part where the game designers Tom Hall and John 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 repeating tiles.

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

[13] John Carmack on X, July 11, 2020.

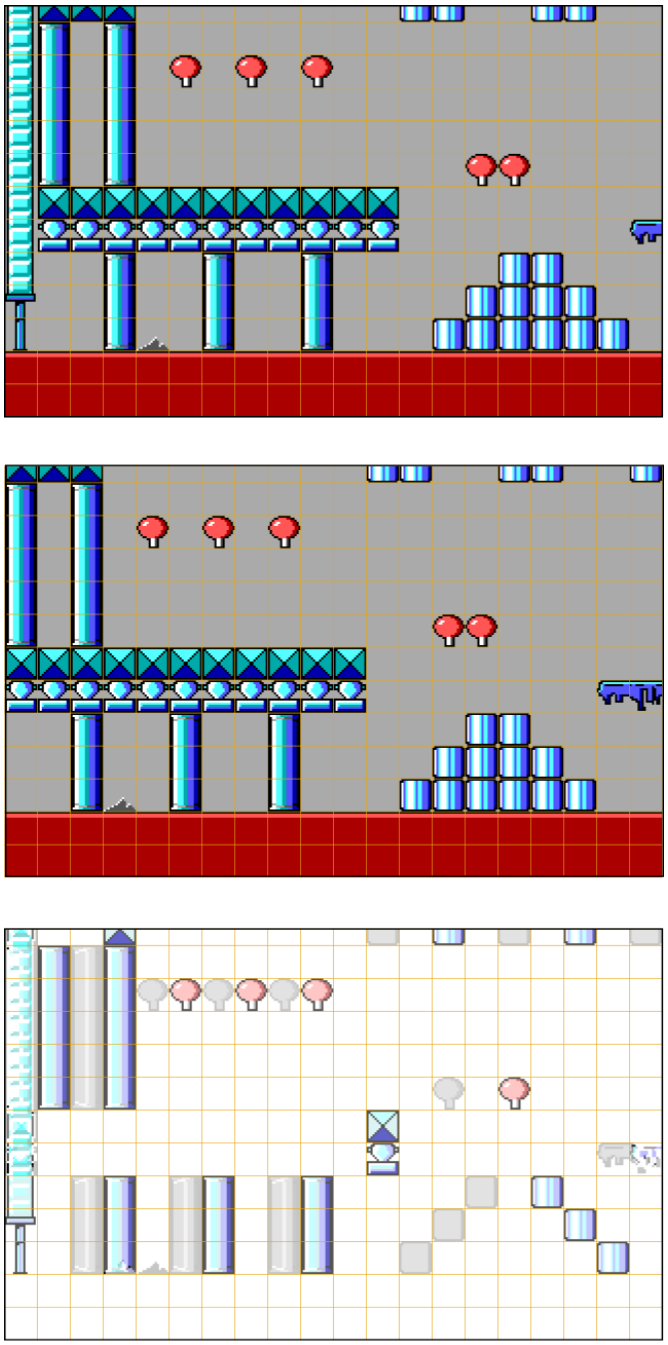


Figure 4.27: Start of the world, moved one tile to the right and difference.

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

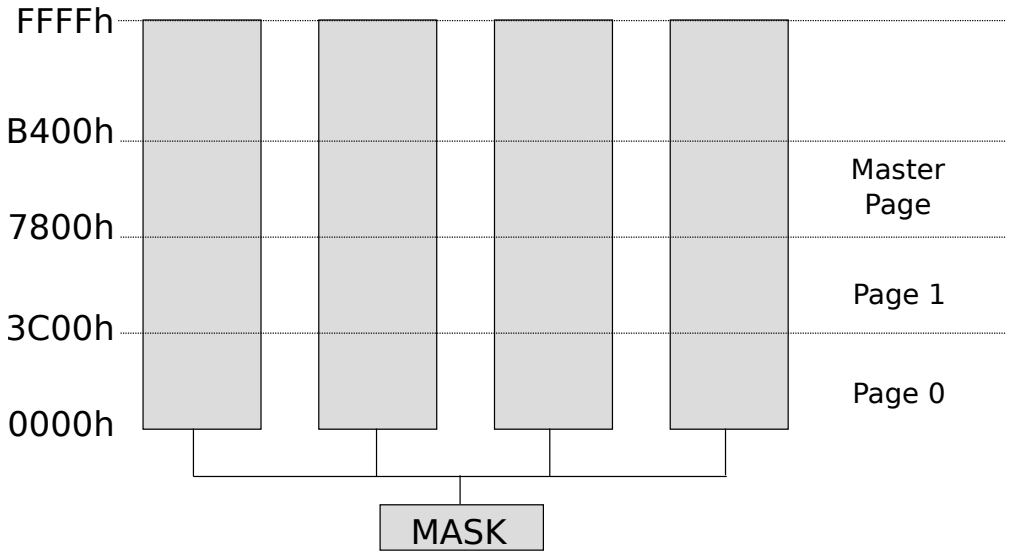
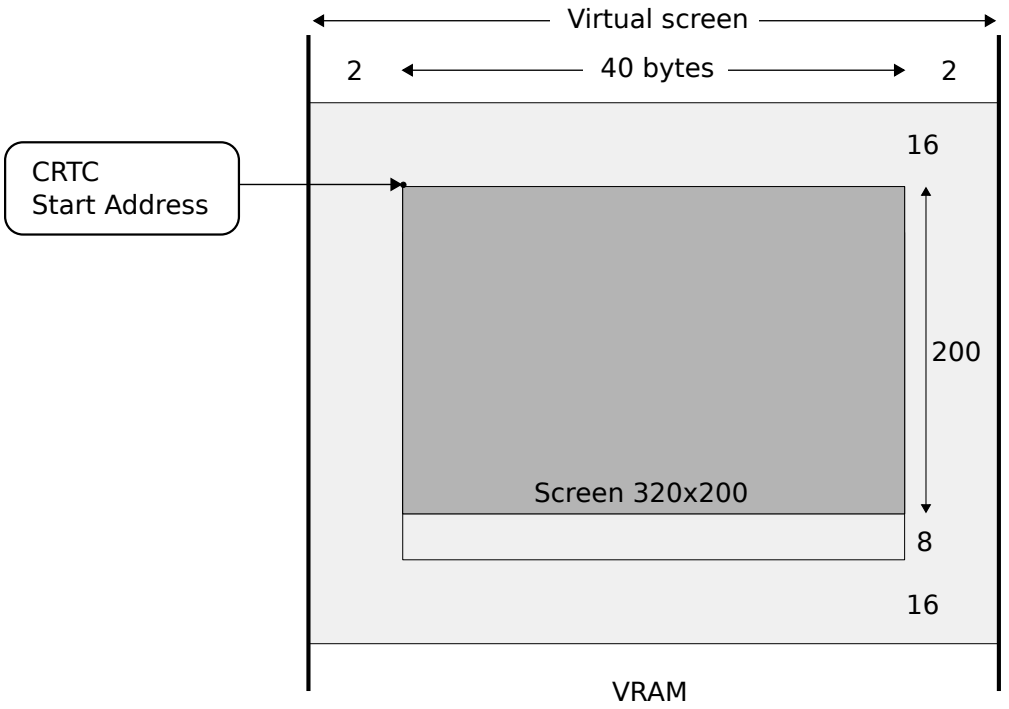


Figure 4.28: Virtual screen layout on EGA card.

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.

Each virtual screen has a size of $44 \times 240 \times 4$ = 42,420 bytes. So, within a 256KiB 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.



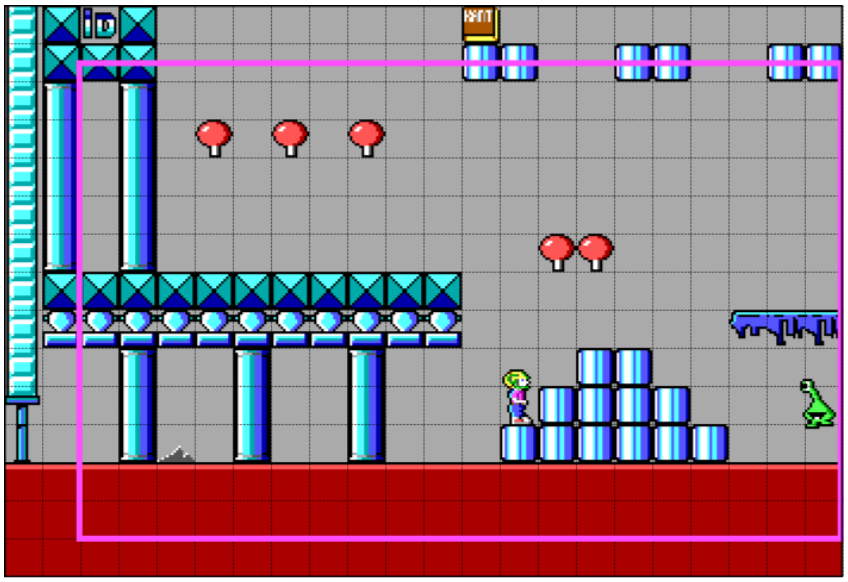
For both the buffer and visible screen a tile array is created to maintain which tiles are updated since last refresh.

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

The steps to refresh the screen are as follows (ignore sprites for now):

1. Check if the player has moved one tile in any direction.
2. The ATR function scans through every tile in the current display page and compares them against the corresponding tile from the level map one by one.
3. If the stored tile index number doesn't match the one in the level map, the corresponding tile is copied to the master page, and marked in both tile update arrays.
4. Refresh the buffer page by copying marked tiles from the master page.
5. Switch the view and buffer pages by adjusting the CRTC Start Address and Pel Panning registers.

In the next three snapshots, 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.



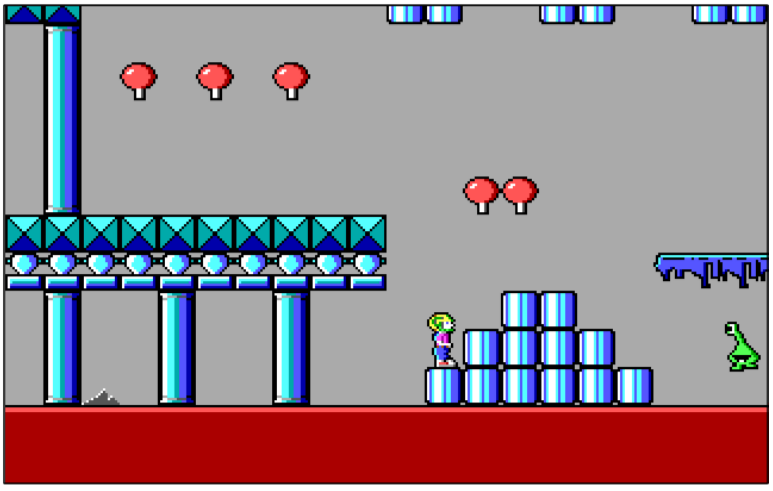


Figure 4.29: Start: Reach the end of the virtual screen.

The engine keeps track of which tile numbers are part of the virtual screen. Since the engine only refreshes 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.

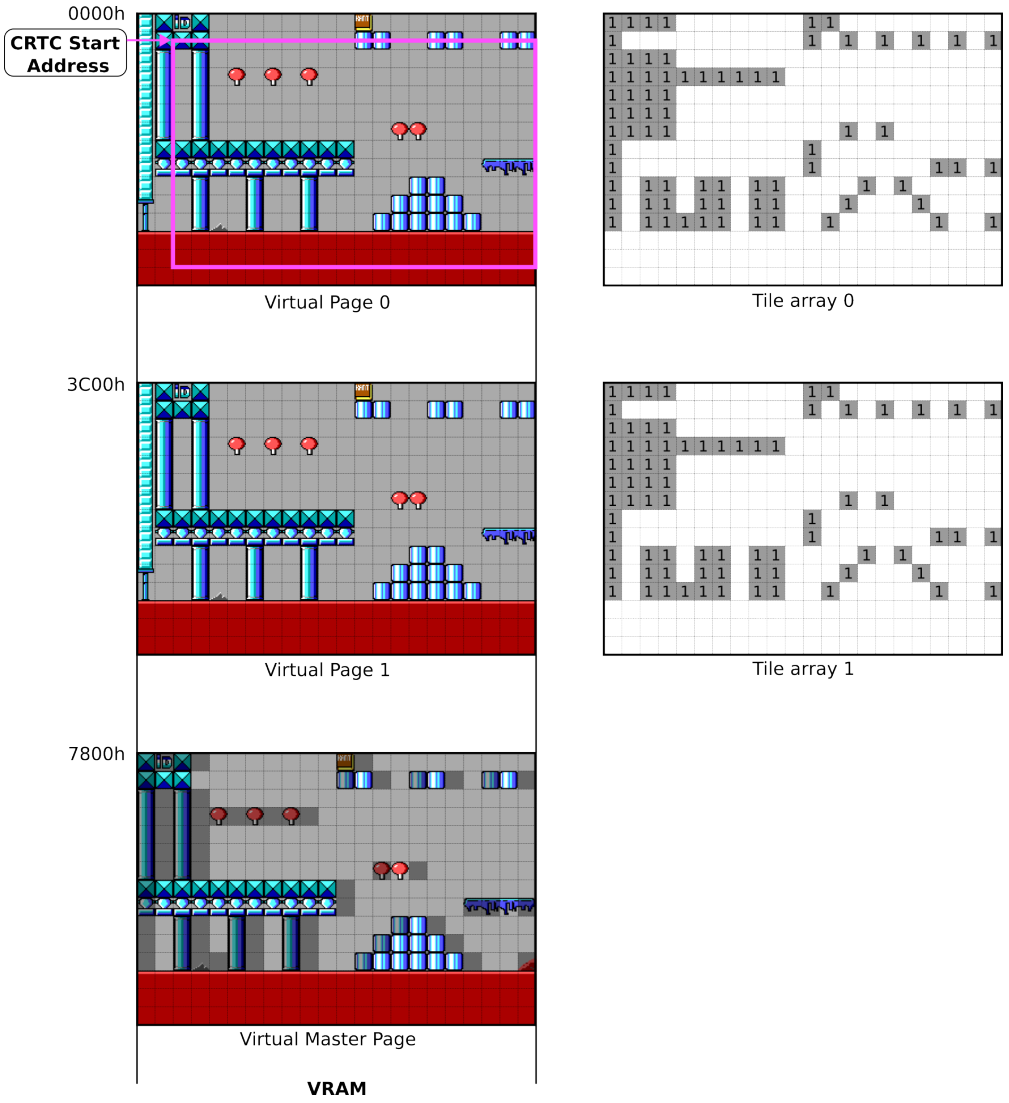


Figure 4.30: Update tiles in master page and update both tile update arrays.

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

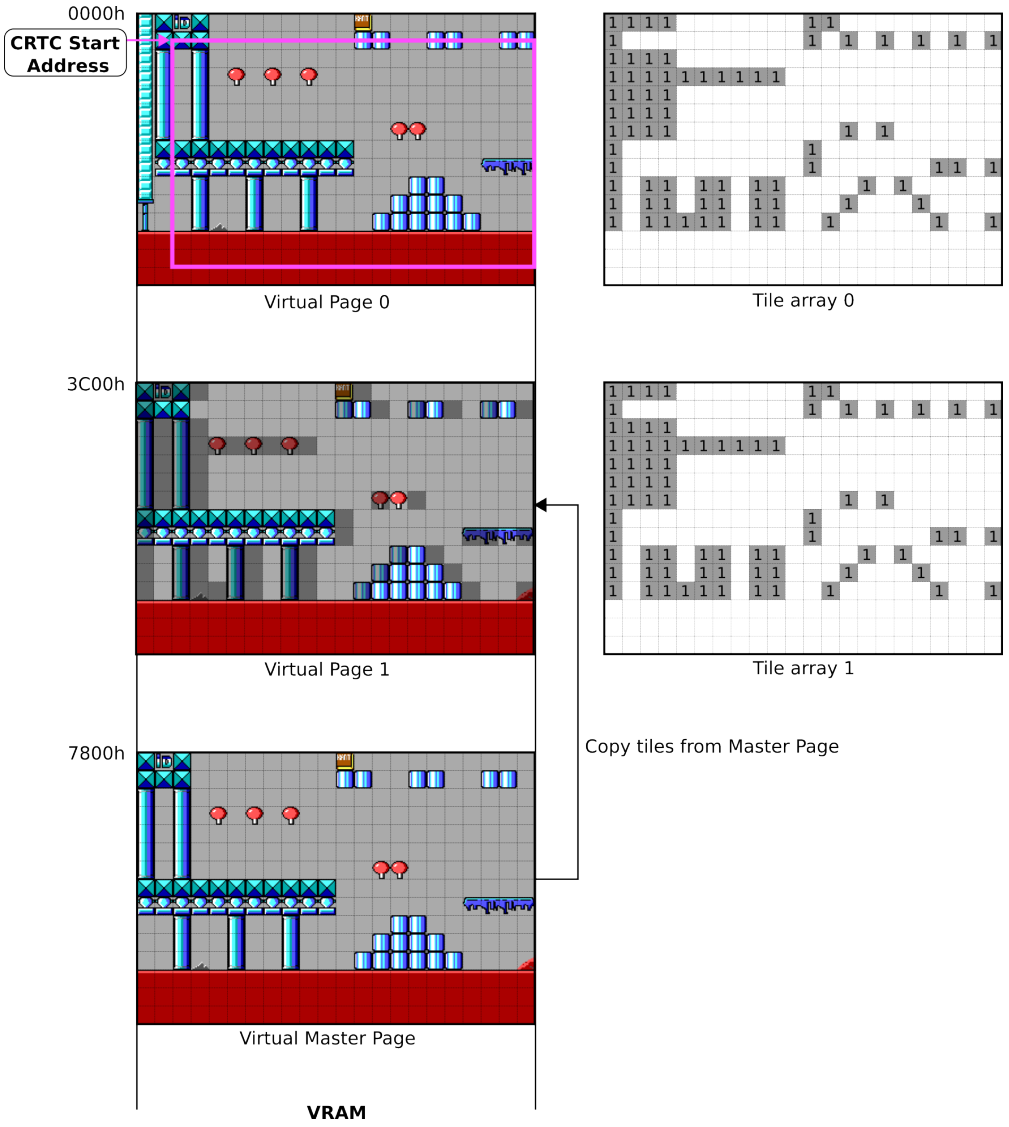


Figure 4.31: Copy changed tiles from virtual master page to page 1.

In the final step, point the visible screen to virtual page 1 by updating the CRTC start address. Finally, tile update 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 update 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.

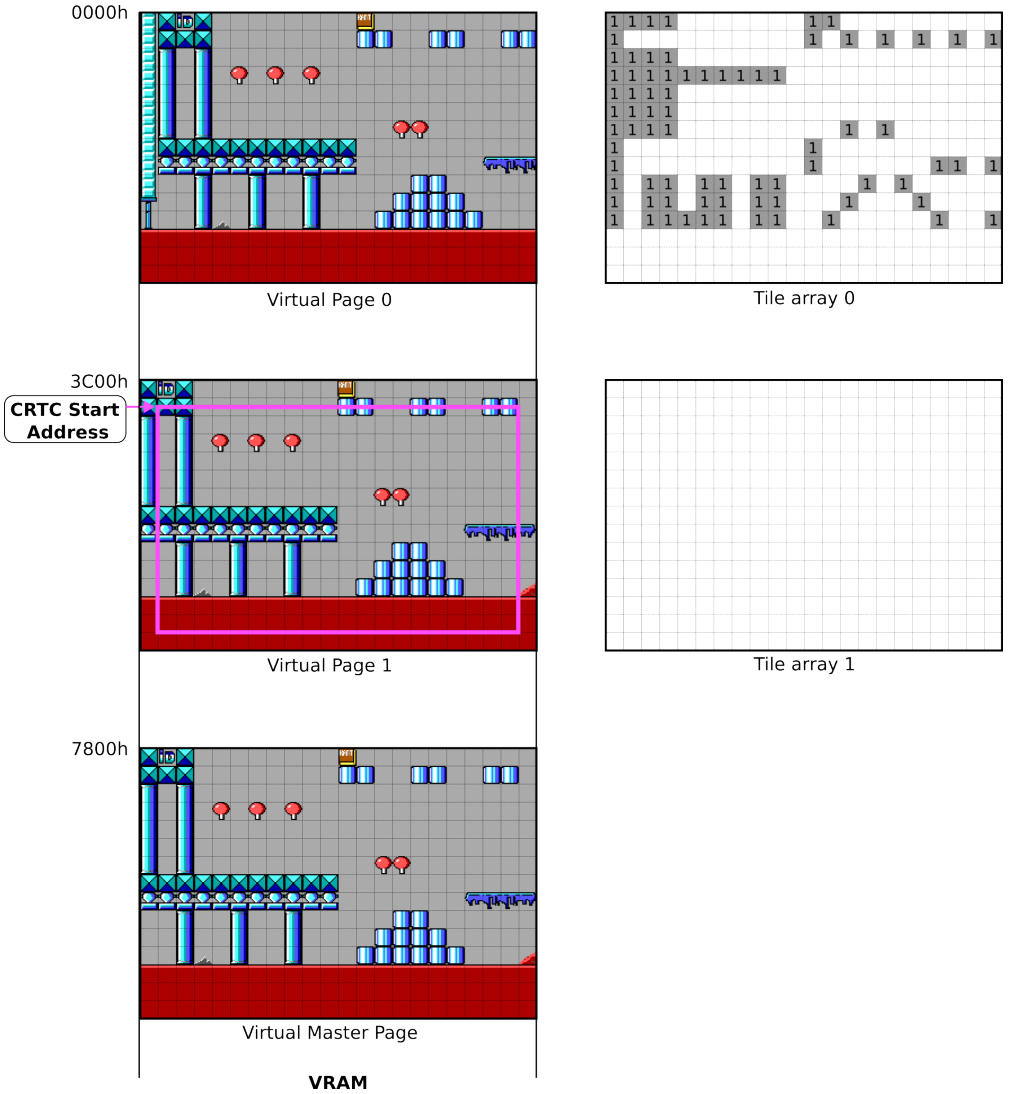


Figure 4.32: Update CRTC Start Address to virtual page 1 and empty tile update array 1.

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

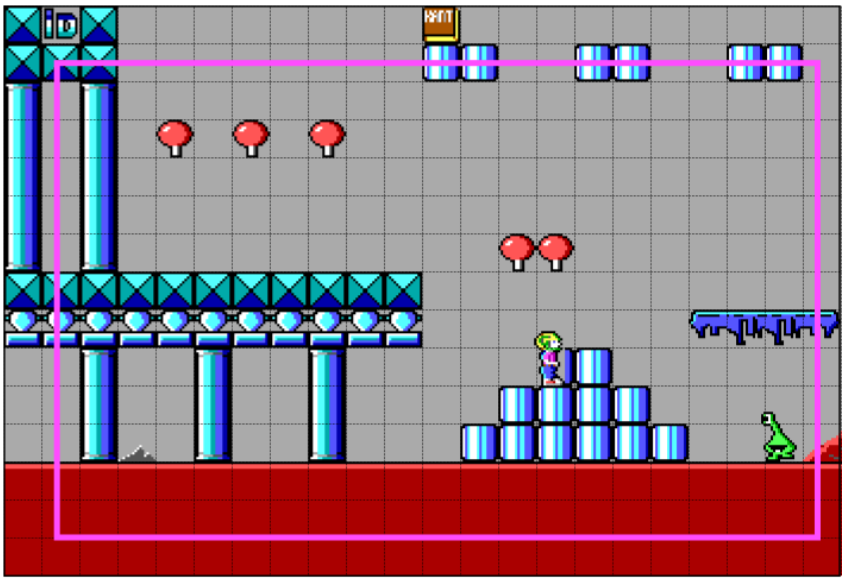


Figure 4.33: Screen is refreshed and scrolled to the right.

4.10.3 Virtual Screen Tile Refresh

John Carmack explored what would happen if you push the virtual screen over the 64KiB 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.

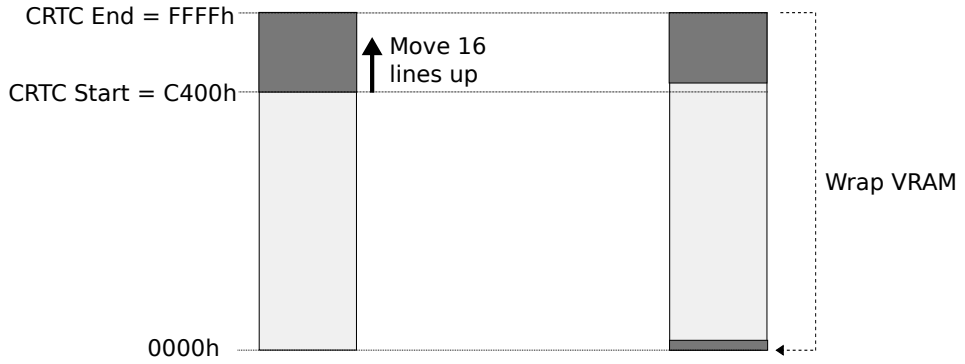


Figure 4.34: EGA memory wrap-around.

> 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**[14]

[14] An explanation further elaborated during the same interview with Lex Fridman in 2022.

4.10.4 Virtual Screen Tile Refresh in Keen Dreams

The EGA memory wrapping results in an improved, more simplified screen refresh algorithm. This algorithm was called "Virtual Screen Tile Refresh".

> The second Keen trilogy used a better trick – just keep panning and redrawing the leading edge, letting the screen wrap around at the 64k aperture edge.
>
> **John Carmack**[15]

First, a virtual screen and tile update 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.

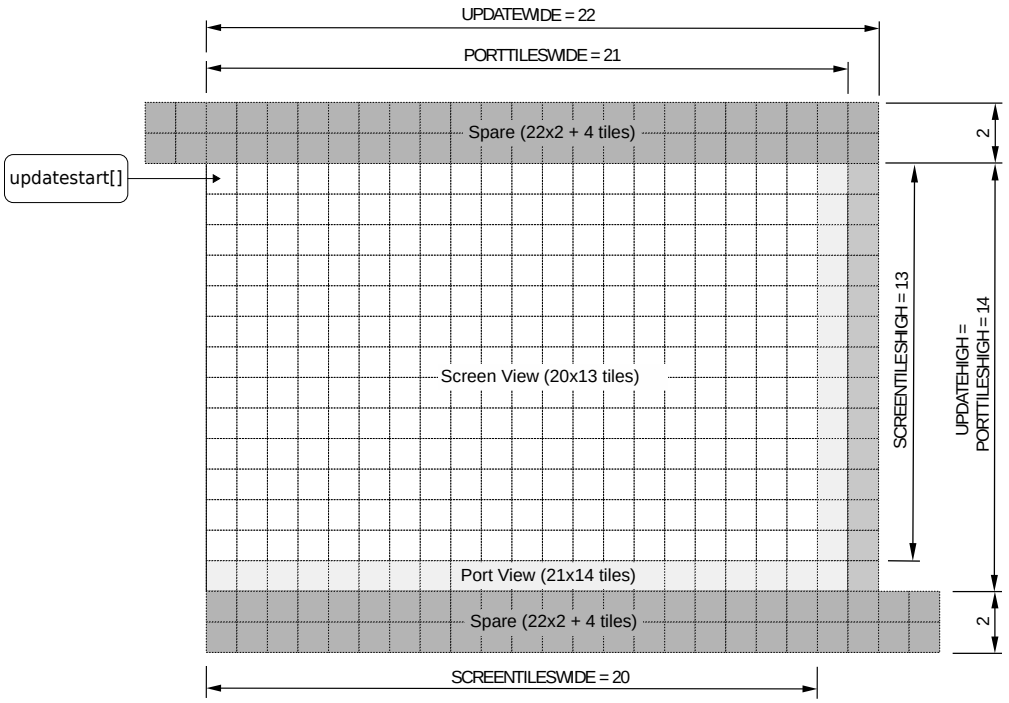


Figure 4.35: Tile update array layout.

[15] John Carmack on Twitter.

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

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 update 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);
}
```

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 update 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.

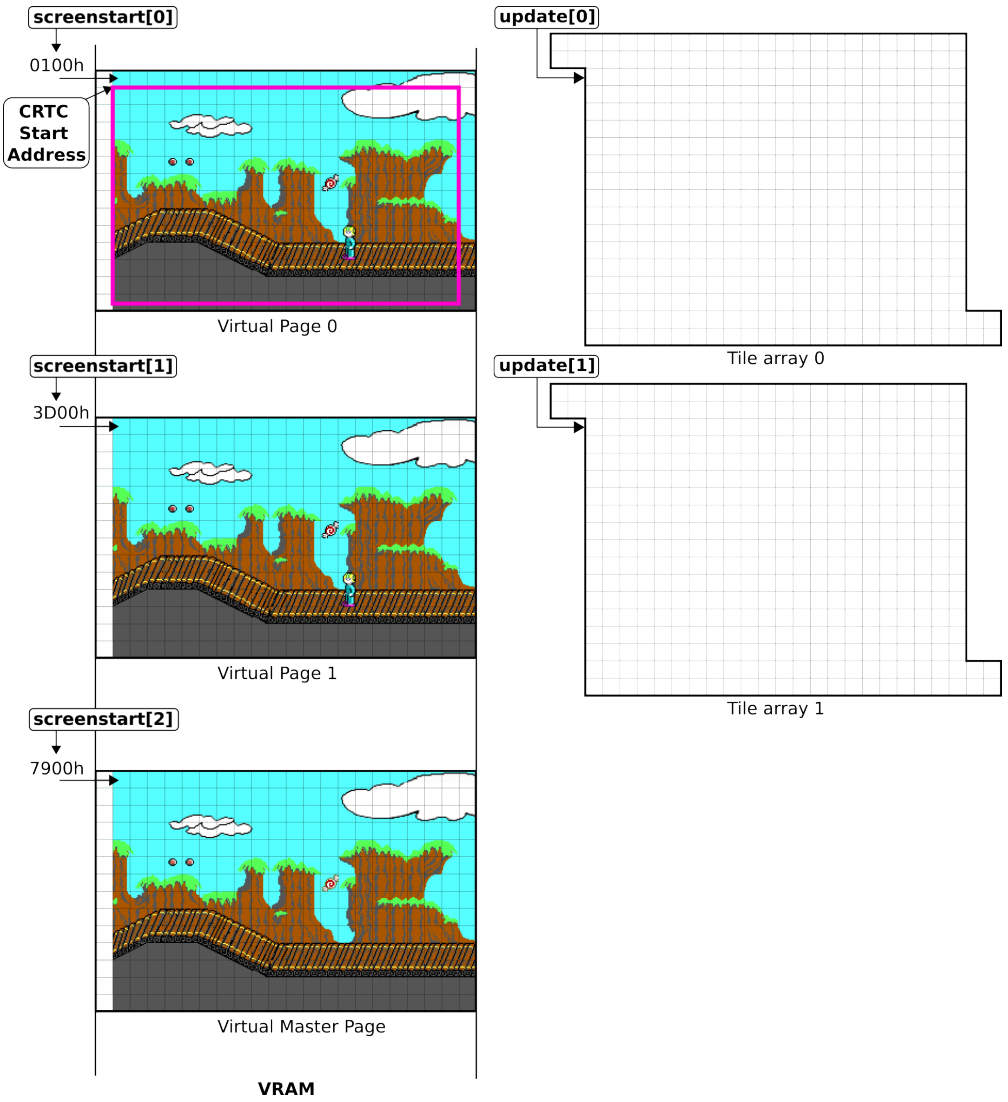


Figure 4.36: VRAM and tile update arrays before scrolling to the left.

Both VRAM `screenstart[]` pointer and tile update array pointer shift left to introduce a new column of tiles. The new column of tiles is copied from RAM to the master page, while the leftmost column in both tile update 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'.

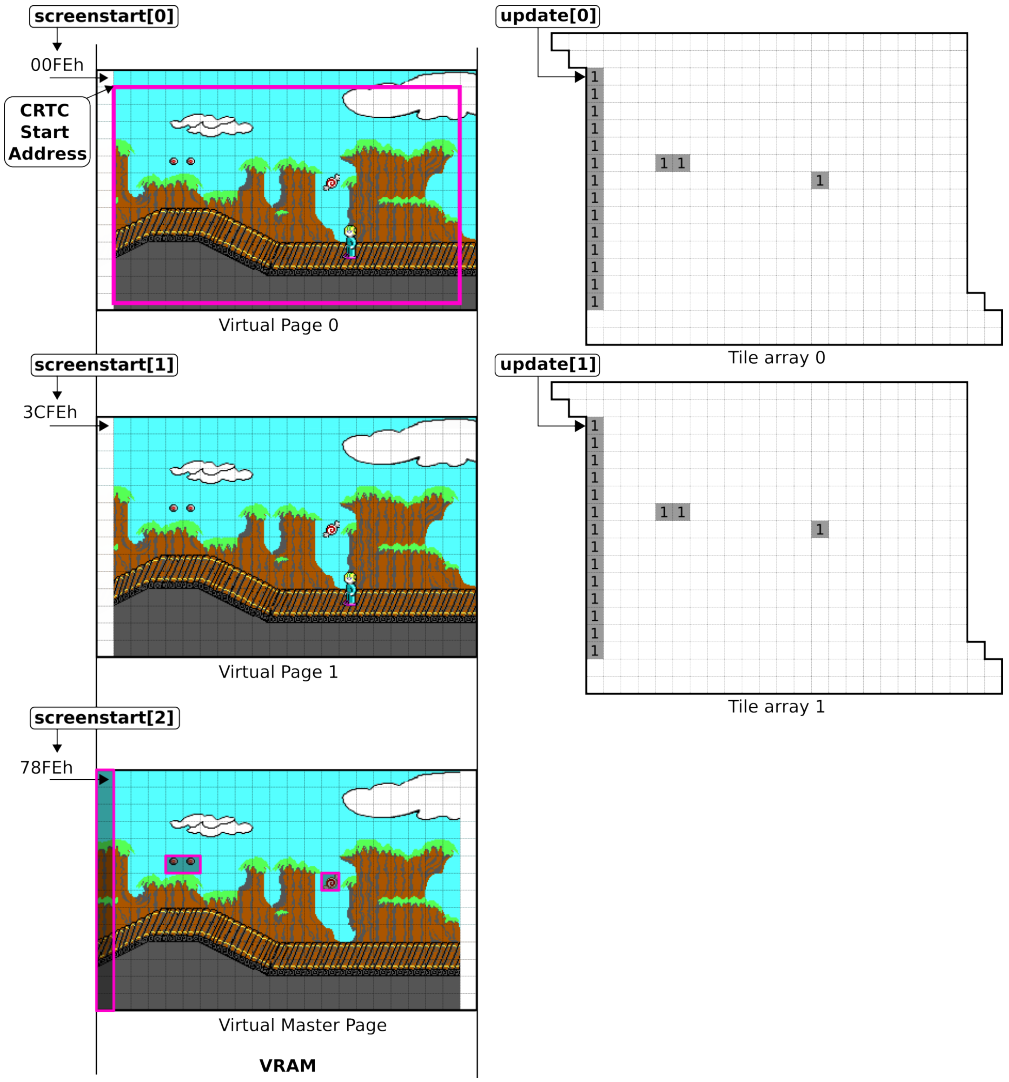


Figure 4.37: Update Virtual master with new left column and animated tiles.

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

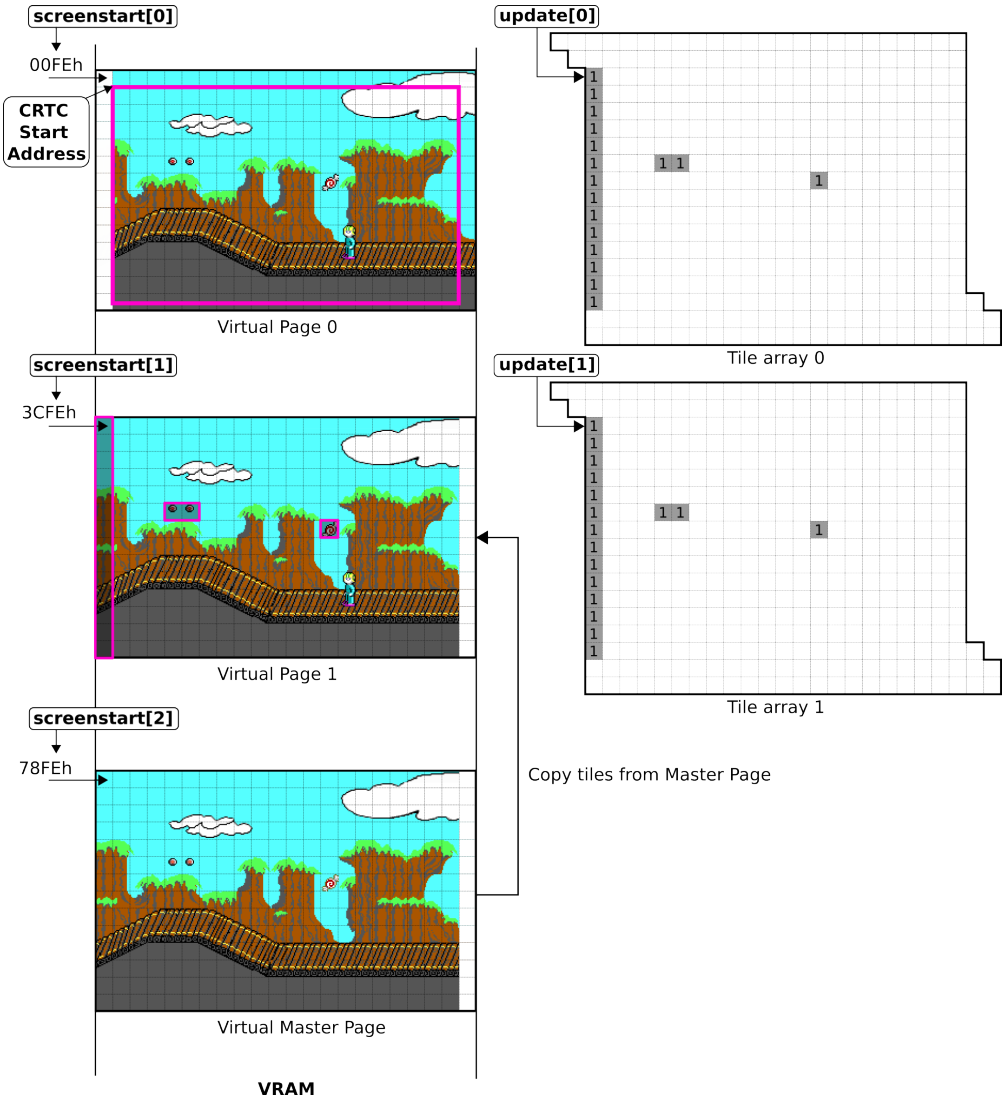


Figure 4.38: Copy changed tiles from master to buffer page.

The final step is to erase all sprites listed in the sprite removal list from the buffer page and copy new sprites from RAM to the buffer page. Tiles overlapping with erased blocks are

marked with '2' in the tile update array, while tiles containing new sprites are marked with '3'. Note that the '2' marker is not used anywhere in the engine; it was most likely part of the original ATR algorithm.

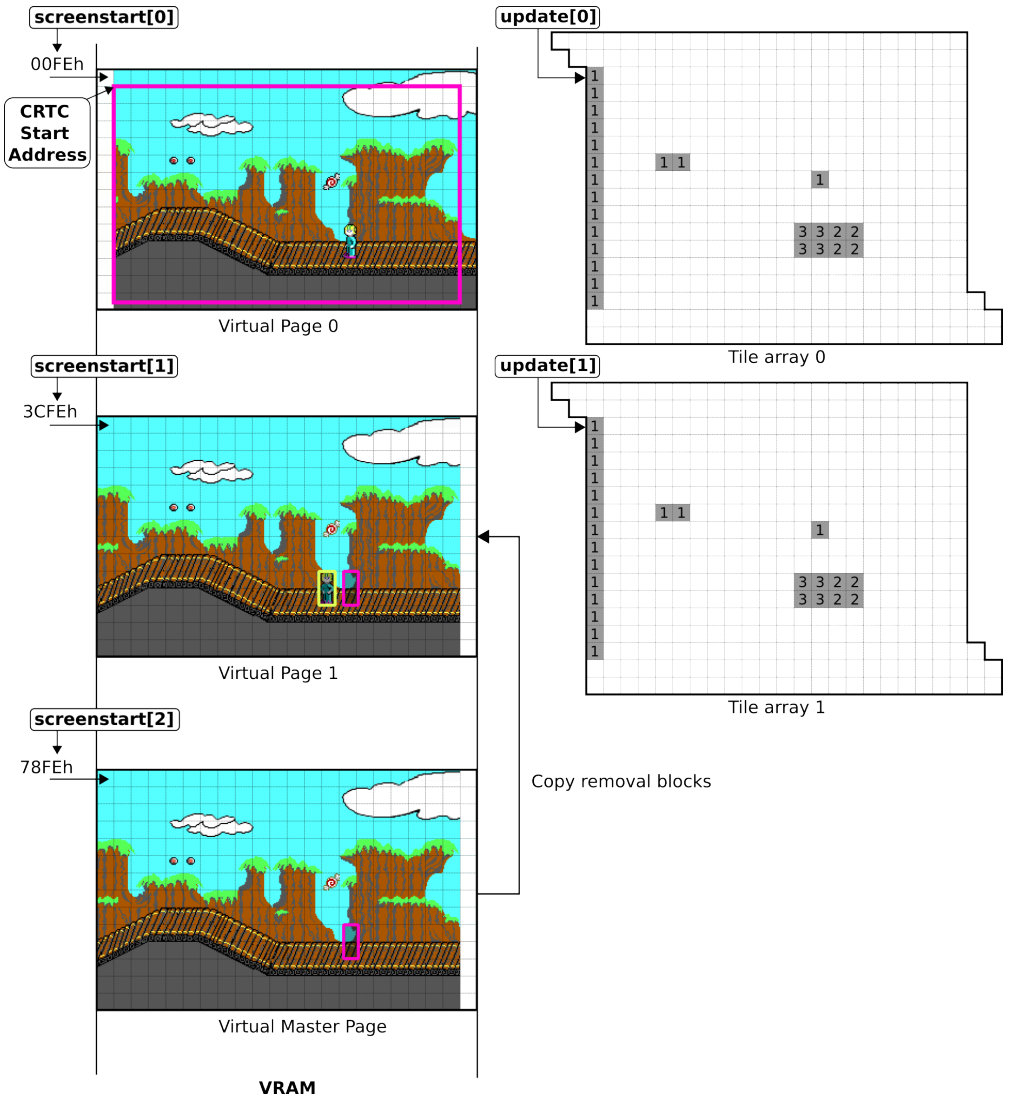


Figure 4.39: Removing and updating sprites to the buffer screen.

For each sprite that needs to be removed from the buffer screen, its location and size are stored in the sprite removal list, allowing the sprite to be removed by copying the corresponding section from the master screen to the virtual page.

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

//sprite removal list for Page 0 and Page 1
eraseblocktype  eraselist[2][MAXSPRITES],*eraselistptr[2];
```

Notice that the removal block is perfectly aligned on an even memory address by using the `& ~1` bitwise operator to clear the least significant bit, ensuring 16-bit transfers per CPU cycle.

```
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++;
  }
}
```

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 update 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 been refreshed. Finally, the refresh cycle restarts, with virtual page 0 now functioning as the buffer.

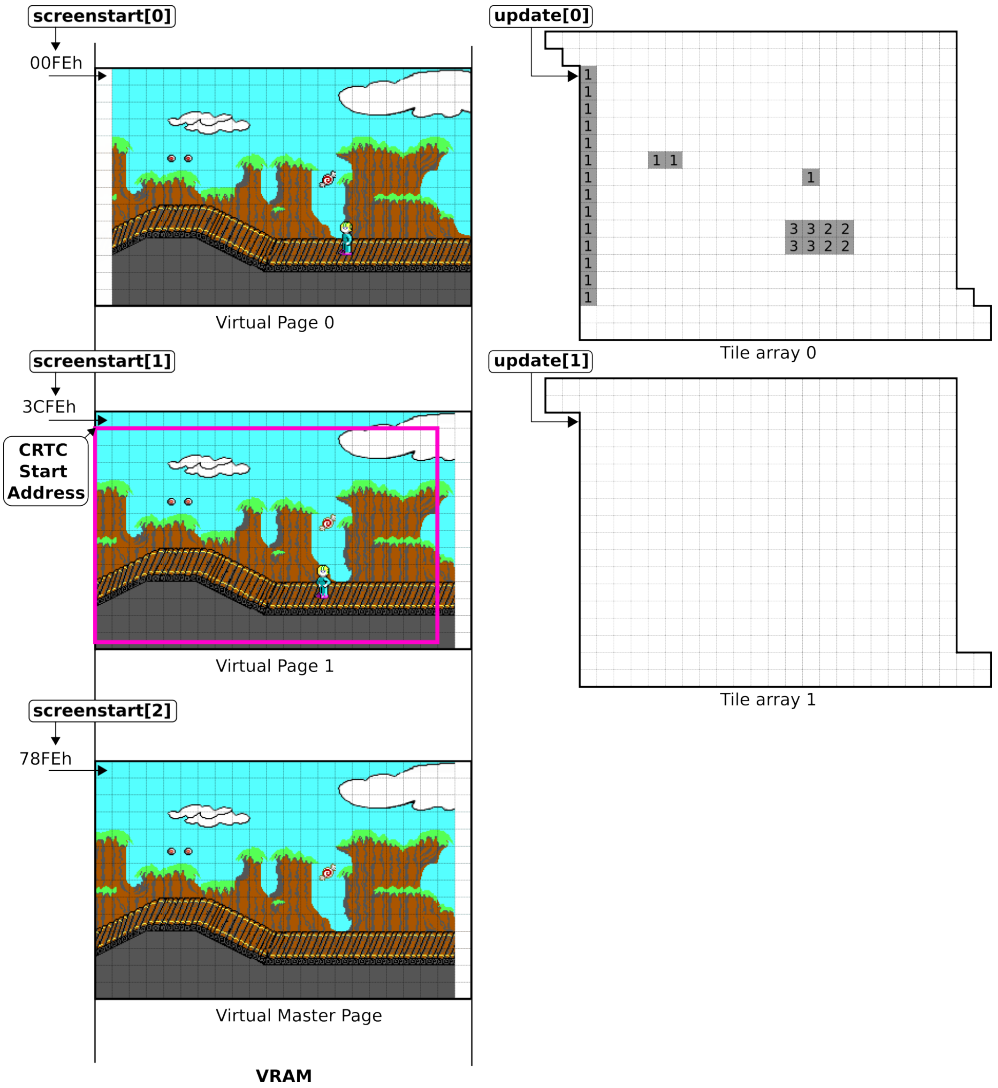


Figure 4.40: Swap buffer and visible screen by updating CRTC start address.

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.

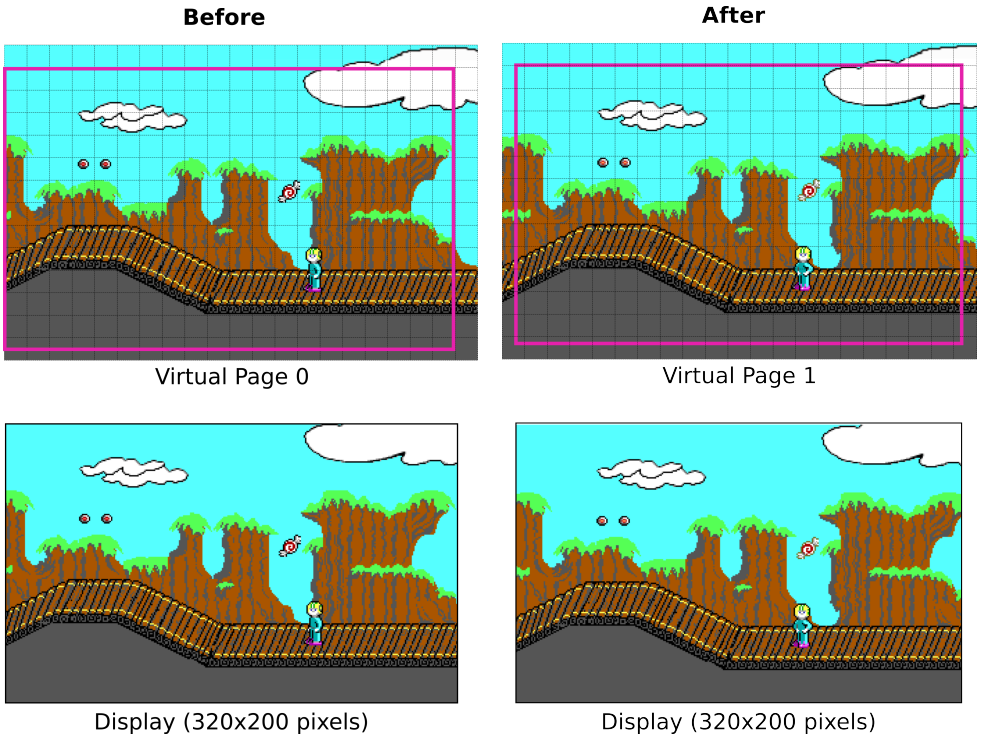
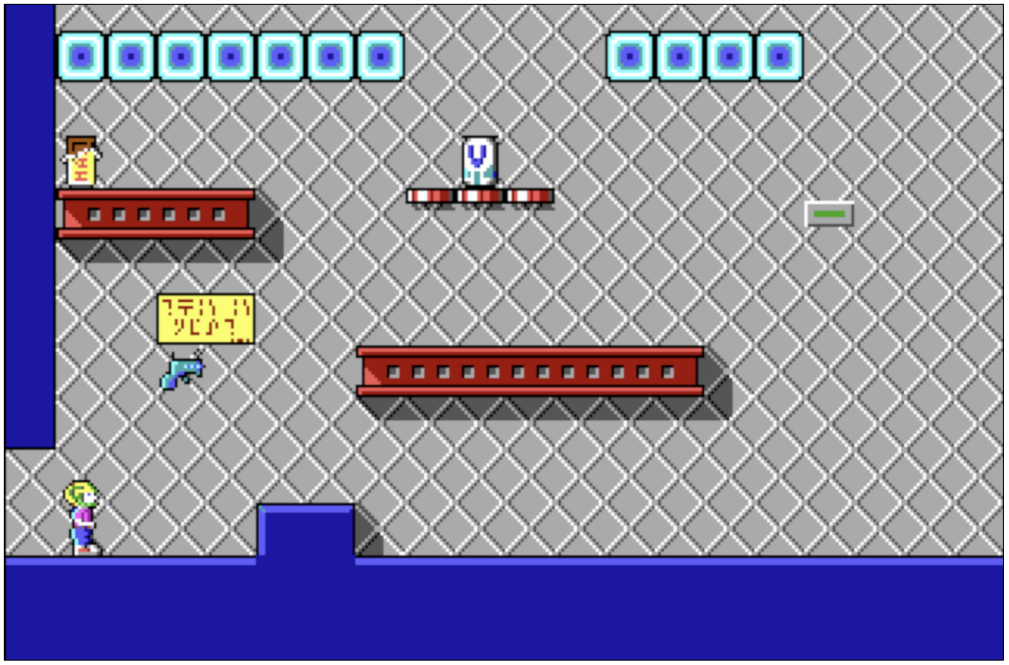


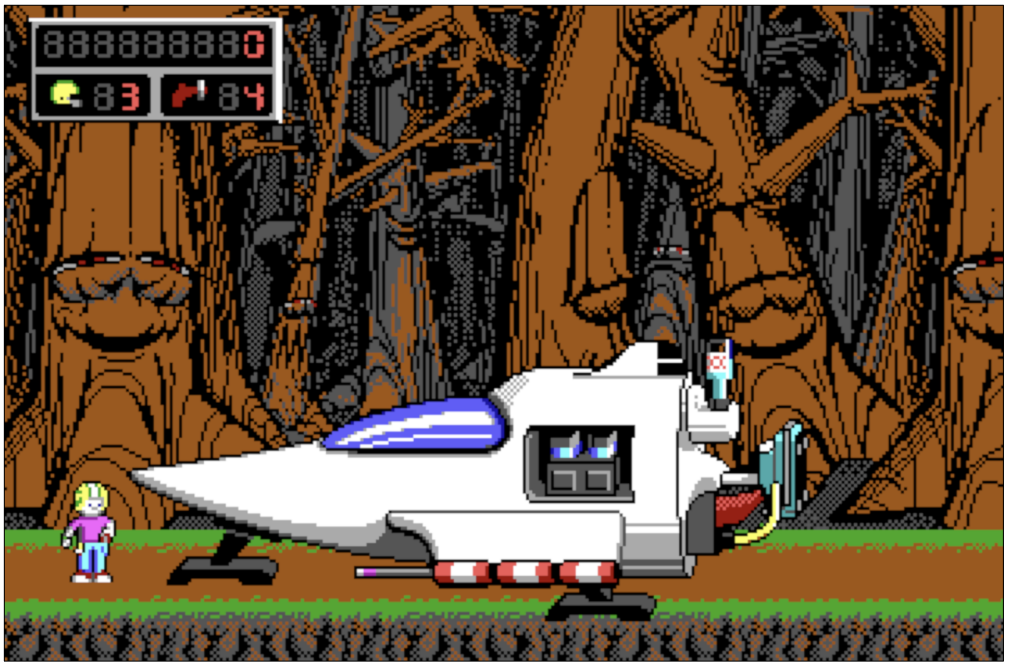
Figure 4.41: Left screen is before scrolling, right screen after scrolling.

If the player continues to move to the left, the tile update array 0 pointer lowers a second time, underscoring why the viewport needs to float up to two tiles in all directions.

This scrolling process is significantly more efficient than the original adaptive tile refresh engine, requiring only 8% of the tiles to be refreshed. In addition, level design is no longer constrained by the need to use large repeating patterns.

This is illustrated on the next page with a comparison between a level from Commander Keen II - *The Earth Explodes* (top) and one from Keen IV - *Secret of the Oracle* (bottom). Notice how in Commander Keen IV almost nothing repeats except for the minimal path and underground sections.




00000000
03
04

4.10.5 Crippled Super VGA compatibility

The introduction of Super VGA cards created a compatibility issue, as these cards typically contained more than 256KiB of VRAM[16]. This resulted in crippled backwards compatibility and the wrapping around `FFFFh` did not work anymore on these cards.

> “ 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**[17] ”

A simple solution can be used to resolve this issue. As illustrated in Figure 4.28 on page 121, the space between `B400h` and `FFFFh` is unused and provides enough room for another virtual screen. Instead of relying on memory wrapping, the corresponding screen page is copied to the opposite end of the VRAM buffer, as illustrated in Figure 4.43. This ensures that the display remains contiguous.

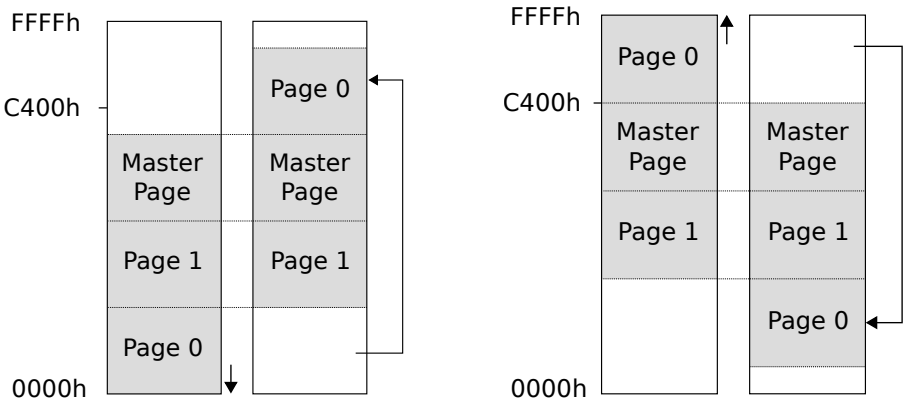


Figure 4.43: Move screen to opposite end of VRAM buffer

[16] 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 more than 256KiB VRAM, which from a hardware constraint resulted in 512KiB.

[17] And again the same interview with Lex Fridman in 2022.

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

//Calculate new startring point of the screen
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 Memory Mapping" on page 47, each pixel is encoded by four bits, which are distributed across the four EGA banks. Copying the entire page requires reading all data from one EGA bank into RAM, writing it back to VRAM at its new location, and repeating this process for each of the four EGA banks. So, how can we copy four VRAM planes fast enough without introducing a noticeable performance hit?

A closer look at the EGA card reveals that a latch is placed in front of the ALU, which can be repurposed for this task. The ALU for each bank can be configured to use only the latch value when writing to VRAM, ignoring the CPU-provided value[18]. By enabling the mask to write to all four memory banks, the values stored in the latches are written simultaneously to VRAM. With this setup, a single read operation ① loads all four latches at once, and ② four bytes—one per bank—are written with a single write operation. This makes it possible to copy the entire buffer fast enough without causing any noticeable performance impact.

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

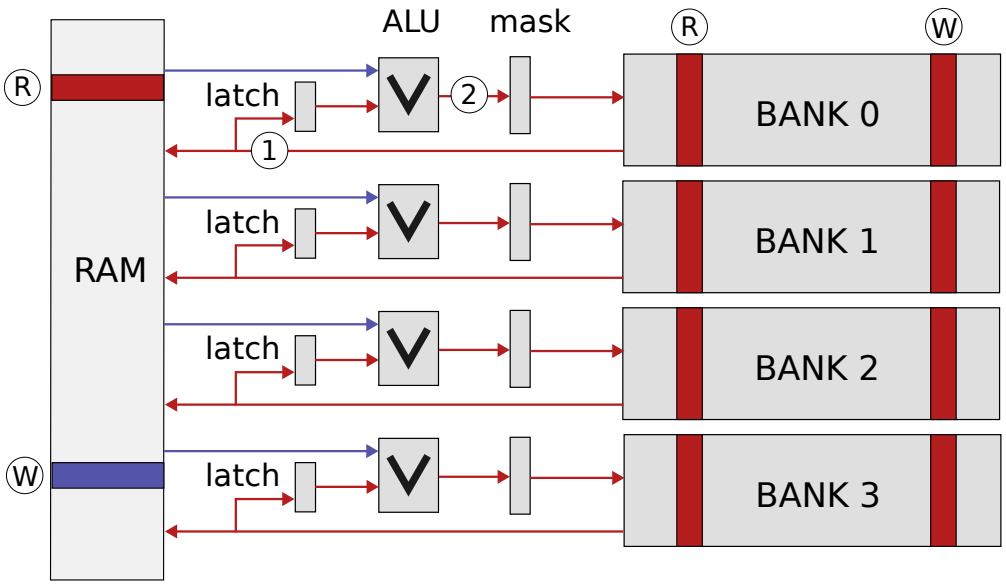


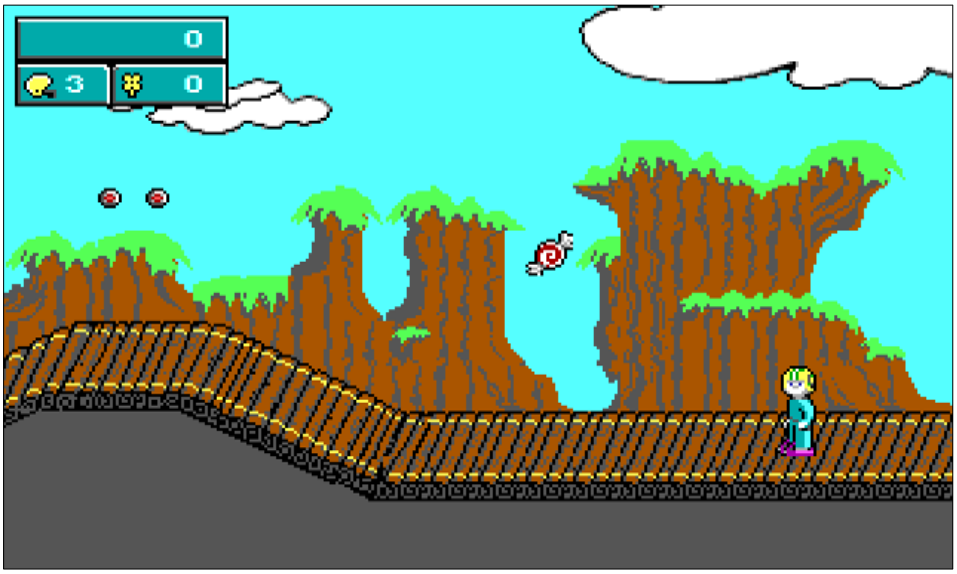
Figure 4.44: Read operation (in red) stored in latches which are later used to write back into VRAM, ignoring the RAM value (in blue).

The setup requires to configure the grapics controller to only read the content of the latches and the map mask register to enable writing to all 4 planes at once.

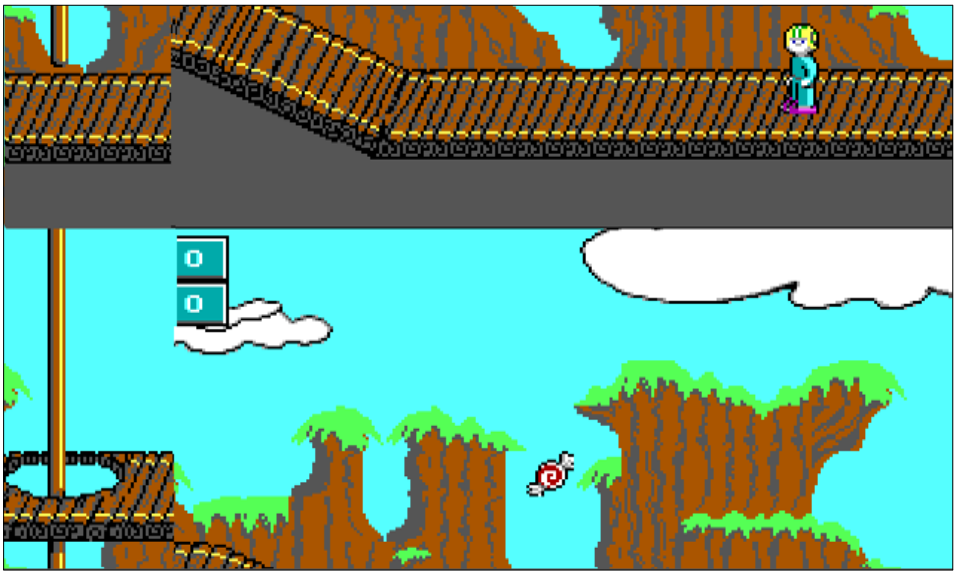
```
GC_INDEX     =    0x3CE     ;Graphics Controller register
GC_MODE      =    5         ;mode register
SC_INDEX     =    0x3C4     ;Sequence register
SC_MAPMASK   =    2         ;map mask register

;==============
; 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
```

4.10.6 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 approach to this problem is to update the start address when the vertical retrace signal is detected via the Input Status 1 Register, specifically bit 3 at port `3DAh`. Unfortunately, by the time the program observes this status, the start address for the next frame has already been latched, as this occurs immediately at the beginning of the vertical retrace pulse.

An alternative approach is to update the CRTC start address at a point sufficiently distant

from the start of the vertical retrace. The Display Enable Status signal at bit 1 from the same Input Status Register can be used for this purpose; a value of 1 indicates that the display is currently within a horizontal or vertical retrace[19]. What we want to detect is the completion of a horizontal or vertical retrace and the beginning of a new scan line, which is far enough from the vertical retrace to ensure that the new start address is latched during the next vertical sync.

To guarantee that the start address is updated at the very beginning of a new scan line, it first waits for the current scan line to complete. It then waits for a full retrace, verifying that the Display Enable Status returns to 0. At this point, the CRTC start address is updated. Once the CRTC start address has been updated, the engine waits for a vertical retrace to set the Pel panning register.

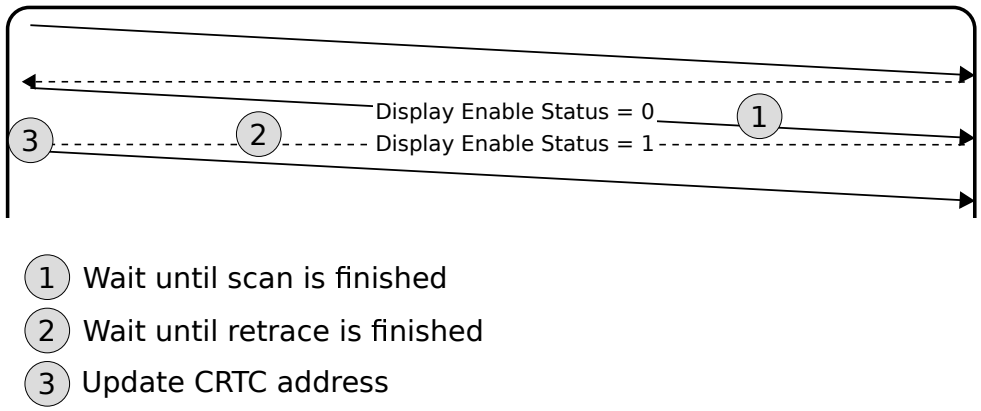


Figure 4.45: Update CRTC start address at beginning of new scan line.

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

Wolfenstein 3D is using an even faster solution. Here, each virtual screen has a fixed starting address and only differ by their high byte value: page 0 is at `0000h`, page 1 at `4100h` and the master page at `8200h`. Moving from any page to another requires updating only the high 8 bits, ensuring a smooth screen refresh[20]. This solution is not possible for Commander Keen as the virtual screen moves around in VRAM.

[19] 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'.

[20] See *Game Engine Black Book: Wolfenstein 3D* section "Solving the VGA Problem".

```
PROC  VW_SetScreen  crtc:WORD, pel:WORD
PUBLIC  VW_SetScreen

  mov dx,03DAh          ;Status Register 1
;
; wait util the CRTC just starts scanning 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


;
; set CRTC start
;
  [...]

;
; wait for a vertical retrace to set pel panning
;
  mov dx,03DAh
@@waitvbl:
  sti                   ;service interrupts
  jmp $+2
  cli
  in  al,dx
  test  al,00001000b  ;look for vertical retrace
  jz  @@waitvbl

;
; set horizontal pel panning
;
  [...]
```

4.11 Actors and AI

All objects in the game, such as Commander Keen, enemies, bonus points, doors, and even the scoreboard, are called "actors". Each actor is controlled via a state machine, enabling them to think and take actions such as walking, throwing objects, jumping over a gap, or playing sound. To model their behavior, each enemy has an associated state, which can include:

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

4.11.1 State Machine

A state machine is a design pattern that manages an object's behavior by breaking it into distinct states (e.g., idle, running, jumping) and defining rules for transitioning between them. In Commander Keen, each state has associated think, reaction, and contact method pointers. Additionally, there is a `tictime` value 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, for 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 functions (e.g., `WalkReact` and `ProjectileReact`) but usually having their own unique "thinking" states.

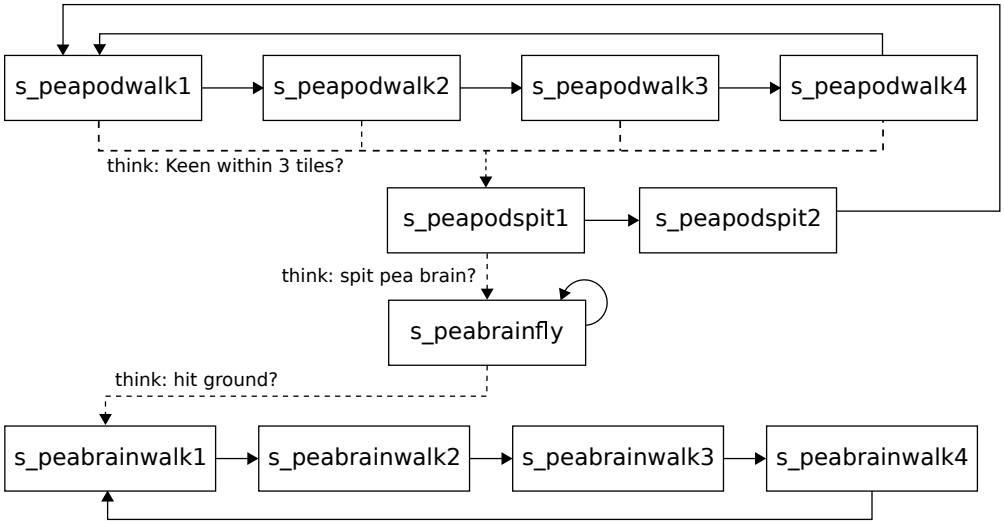


Figure 4.46: State machine for Pea pod and Pea brain.

The `*react` function determines how an enemy reacts to the environment, such as turning around when hitting a wall or reaching 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 losing a life.

4.11.2 Clipping

Whether an actor can move through 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 an actor moves to the left and encounters a solid tile from the east, the engine adjusts the actor's movement to prevent it from crossing the tile.

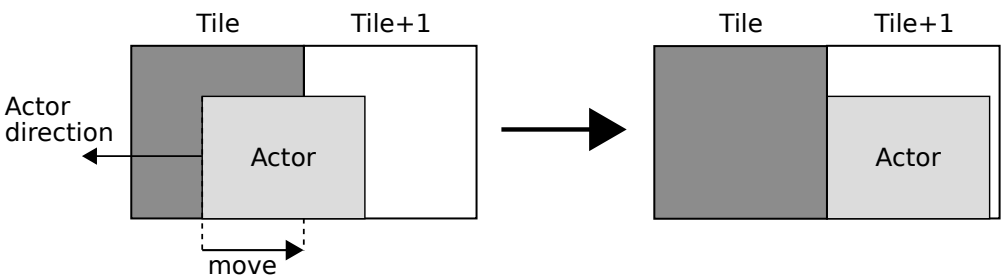


Figure 4.47: Clipping to east wall when actor moves left.

```
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. When 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.

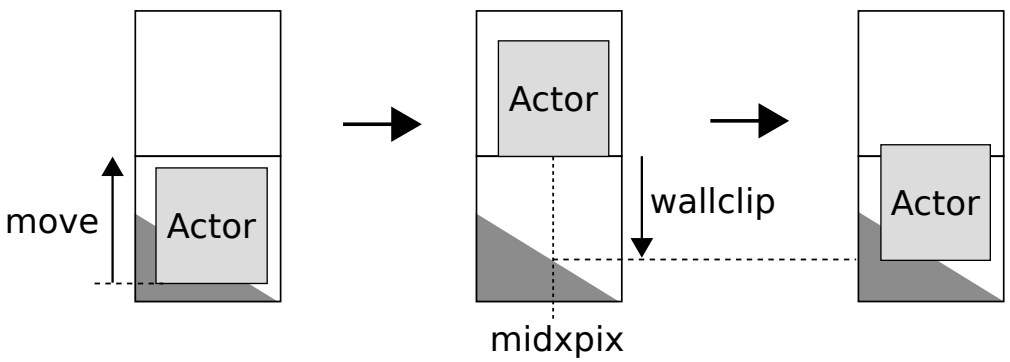


Figure 4.48: Clipping north wall with slope.

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

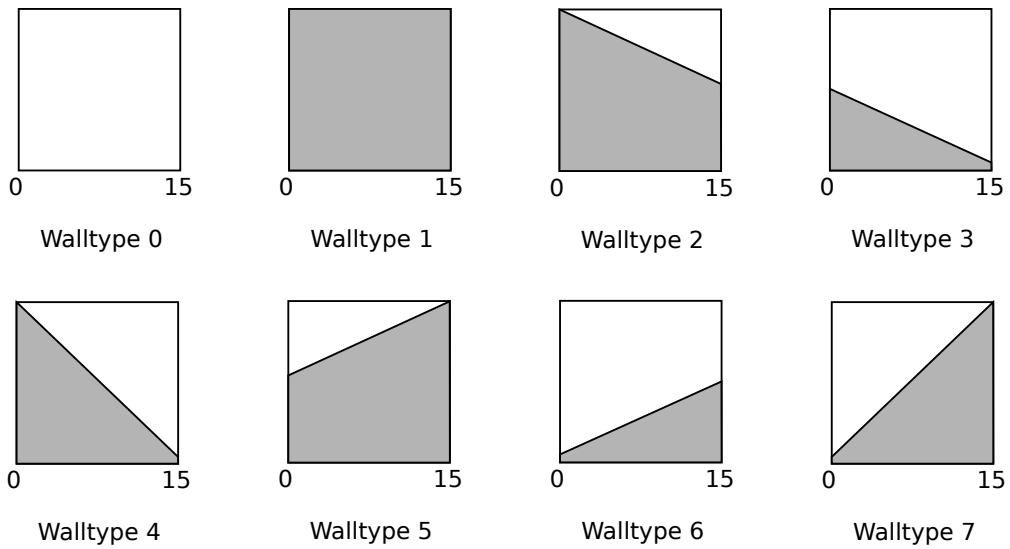


Figure 4.49: `wallclip[8][16]`: Fall through, solid and 6 slope types.

```
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 Rendering the layers

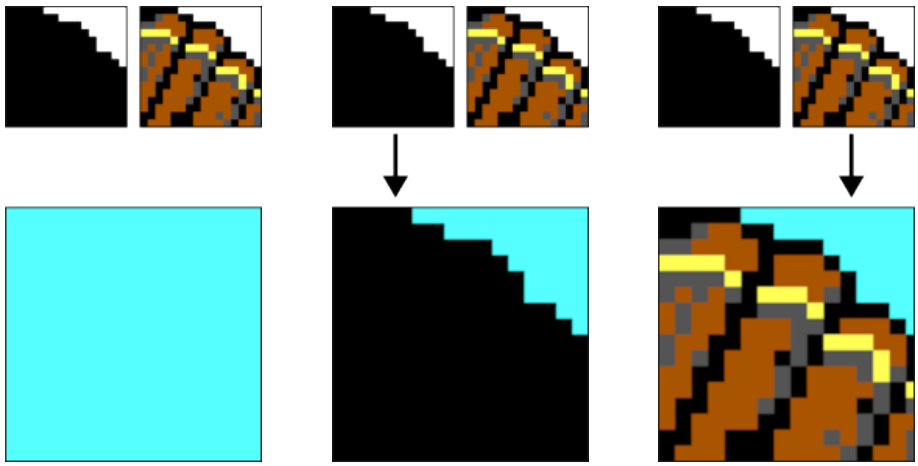
Each tile on the screen can contain up to three layers: the background tile layer, the foreground tile layer and a sprite layer. Rendering the final result on screen requires multiple redraws of the screen:

1. Draw the background tile or a combined background and foreground tile.
2. Draw the sprites.
3. Re-draw the foreground tile 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: it only requires copying 128 bytes to VRAM. Drawing a foreground tile on top of the background tile 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 4.50: Draw masked foreground tile.

Combining the background and foreground layers involves using a bitwise `AND`-operation to clear the background pixels, followed by a bitwise `OR`-operation to write the foreground pixels. Because each tile is 2 bytes wide, writes to VRAM therefore always occur on even memory addresses, ensuring full utilization of the 80286 CPU's 16-bit data bus.

```
PUBLIC  RFL_NewTile

[...]
  es,[mapsegs+2]        ;foreground plane
  mov bx,[es:si]
  mov es,[mapsegs]      ;background plane
  mov si,[es:si]
  mov es,[screenseg]
  mov dx,SC_INDEX       ;stepping through map mask planes

[...]
  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:       ;start writing masked tile to VRAM
  WORDOUT           ;set mask plane (OUT DX,AX)
  tileofs = 0
  lineoffset  = 0
REPT  16
  mov bx,[si+tileofs]           ;background tile
  and bx,[ss:tileofs]           ;mask background
  or  bx,[ss:si+tileofs+32]     ;masked foreground data
  mov [es:di+lineoffset],bx     ;move to next line in VRAM
  tileofs   = tileofs + 2
  lineoffset  = lineoffset + SCREENWIDTH
ENDM
  add si,32
  shl ah,             ;shift plane mask over for next plane
  cmp ah,10000b
  je  @@done          ;drawn all four planes
  jmp @@planeloopm
```

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 an 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 101 is used. When caching a sprite in memory, each sprite is stored 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.

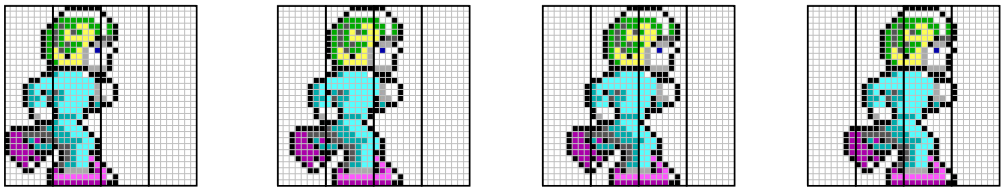


Figure 4.51: Sprite shifted in 4 steps.

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
```

Creating and storing the bit-shifted sprites takes place during loading and caching the sprites from disk. The first sprite keeps the original width (`smallplane`), every shifted copy is one byte wider to accommodate the 8-bit shift (`bigplane`).

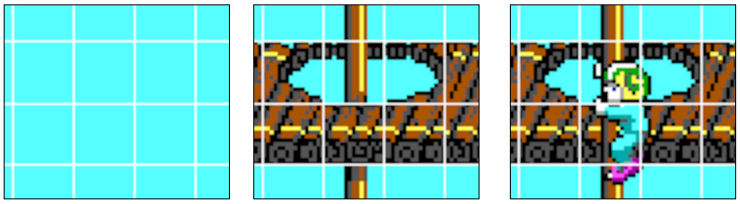
```
void CAL_CacheSprite (int chunk, char far *compressed)
{
    // make the shifts!
    switch (spr->shifts)
    {
    [...]
    case  4:
    dest->sourceoffset[0] = shiftstarts[0];
    dest->planesize[0] = smallplane;
    dest->width[0] = spr->width;

    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);

    [...]    // Shift 4 and 6
    break;
  }
}
```

One might ask why only four shifts are used instead of eight to achieve a perfect one-pixel shift. Consider Commander Keen, which uses 91 different sprites that are all loaded permanently into memory. These sprites consume 41,095 bytes of RAM, and after creating seven additional shifted copies, this grows to a total of 328,760 bytes. In addition, the engine still needs to load and copy all other sprites required for a level. This memory cost was deemed too high. To save RAM, the number of bit-shifted copies is therefore limited to four, resulting in a two-pixel alignment.

Recall that the map itself scrolls with one-pixel alignment using the EGA Pel Pan register. To keep the sprites and the world synchronized, pixel panning must therefore also be restricted to two pixels. A simple technique is used to enforce this constraint: the least significant bit of the Pel Pan value is cleared using a mask.

```
unsigned  xpanmask = 6;   // prevent panning to odd pixels
void RF_Scroll (int x, int y)
{
   [...]
   VW_SetScreen(newscreen+oldpanadjust,oldpanx & xpanmask);
}
```

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 4.52.

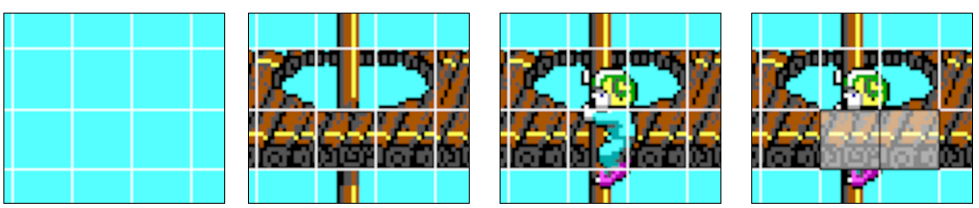


(a) Background tile. ***(b)*** Foreground tile. ***(c)*** Sprite on top.

Figure 4.52: 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. 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 4.40 on page 136.
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 4.53: 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

The minimum number of read/write operations per tile, when only a background tile is required, is 128 bytes (2×16 bytes×4 memory banks). Worst case, up to 512 bytes of read/write operations are required (background tile, foreground tile, sprite, and foreground tile 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. In section "Virtual Screen Tile Refresh" on page 128 it was explained 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 Writing to word-aligned memory

The 286 CPU can read and write 16 bits in a single cycle, but there is a caveat: writing a word at offset `FFFFh` causes the CPU to crash. This is not an issue for tiles, because they are word-aligned (16 bits wide). Since the screen scrolls in tile-sized steps, they always reside at even memory addresses.

Sprites, however, are byte-aligned. When a sprite is drawn starting at an odd memory address, it may cross the `FFFFh` offset and crash the CPU. To avoid this, the engine uses small, specialized routines for each combination of sprite width and even/odd alignment, ensuring correct wrapping at `FFFFh`.

For example, when drawing a 6-byte-wide sprite that starts at an even address, three `word` writes wrap around the segment. If it starts at an odd address, the routine writes a `byte`, followed by two `word` writes, and finally another `byte`.

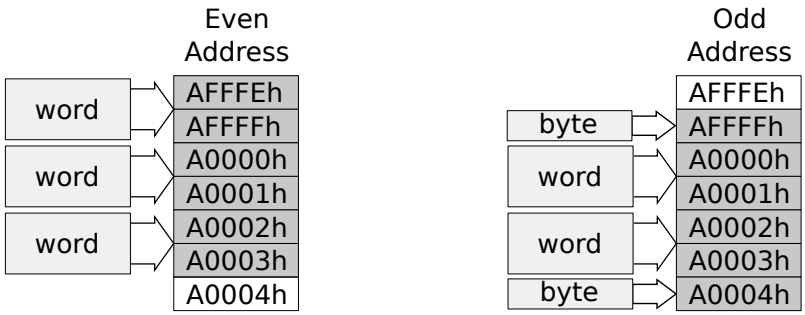


Figure 4.54: Word-optimized 6 byte wide sprite to even (`MASK6E`) and odd (`MASK60`) address.

In total, 23 routines handle aligned `word`-sized writes for sprites up to 10 bytes wide. Their function pointers are 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

[...]

;Optimized functions for word-size writing to memory
EVEN
mask60:
  MASKBYTE
  MASKWORD
  MASKWORD
  MASKBYTE
  SPRITELOOP  mask60
```

4.13 Audio and Heartbeat

Since DOS had no multitasking or thread support, programs could not run parallel tasks such as system heartbeat and sound playback independently. Therefore, they often relied on system timer interrupts to maintain a consistent game loop and timing.

4.13.1 Introduction to 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 unserviced. 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 the Programmable Interrupt Controller (PIC), that causes the flow of program execution (e.g., running the 2D engine) 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 back to the exact point where it was interrupted, and the original program continues as if nothing had happened.

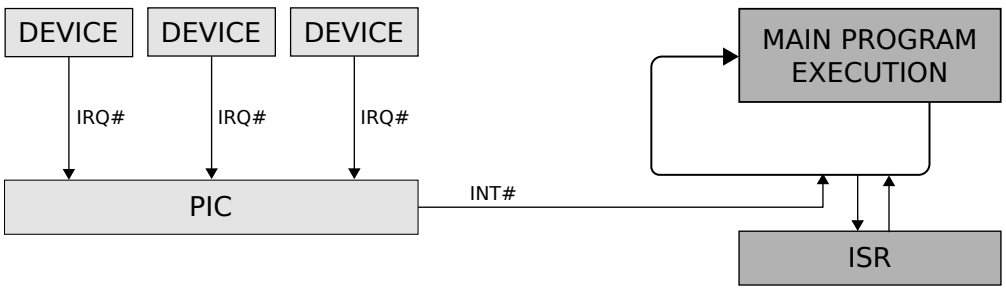


Figure 4.55: Hardware interrupts are translated to software interrupts via the PIC.

Because interrupts can arrive from many sources at any time, a mechanism is required to prevent one interrupt from being pre-empted by another. Therefore interrupts can be disabled using the Interrupt Mask Register (IMR), which enables or disables individual hardware interrupts. To enable or disable all interrupts at once, the interrupt flag (IF) can be used.

```
CLI ; maskable interrupts disabled (IF=0)
STI ; maskable interrupts enabled (IF=1)
```

Notice that masking interrupts potentially introduces other problems, such as discarding important information like keyboard or mouse input. Therefore it was common practice to keep interrupt tasks very small and short.

4.13.2 The PIT and PIC

The audio and heartbeat system relies on two chips: the Intel 8253 Programmable Interval Timer (PIT) and the Intel 8259 PIC. The PIT uses a crystal oscillator running at 1.193182MHz. At its core, the PIT contains programmable counters. 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.

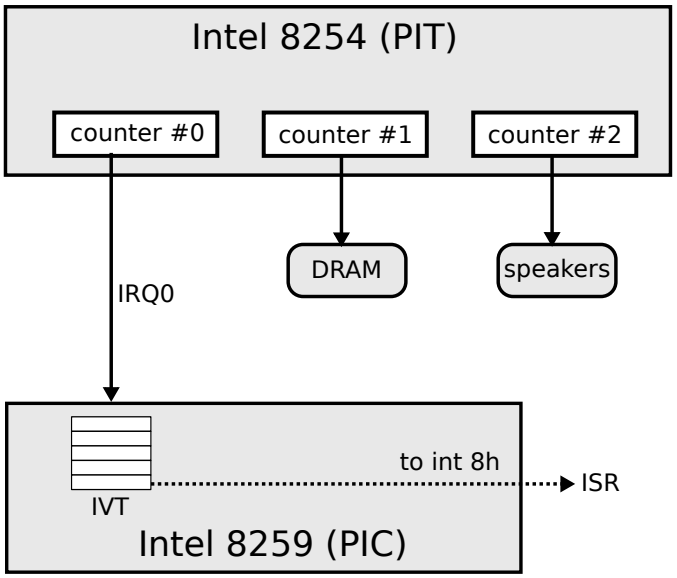


Figure 4.56: Interactions between PIT and PIC.

The PIT contains three counters in total. 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. Counter #2 is connected to the PC speaker and generates sounds. This 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 to the PIC. The PIC routes the IRQ to the Interrupt Vector Table (IVT), which is a list of pointers to corresponding ISR addresses in memory. The first eight interrupts in the IVT are processor exceptions, followed by hardware interrupts (IRQ 0–7), and finally software interrupts. IRQ 0 is mapped to interrupt #8 and usually updates the operating system clock.

Interrupt #	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
11h	Equipment determination
12h	Memory size determination

Figure 4.57: The Interrupt Vector Table as implemented in DOS (entries 0 to 18).

4.13.3 Hijacking the System Timer

Each entry in the IVT can be reprogrammed to point to a custom ISR. By modifying the IVT #8 pointer, an application can hijack the interrupt to serve its own purposes. When this occurs, the engine halts execution at regular intervals and jumps to a custom interrupt function. This effectively creates two systems that run in parallel.

```
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 ISR, 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 describing the state of the computer[21].

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 4.58: Partial list of BIOS Data Area variables.

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

BIOS data address 40:3Fh stores the motor status, where bit 0 indicates whether 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 shut-off 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 PIT counter is 16-bit and decrements every clock cycle. When it wraps around after 2^{16} = 65,536 counts, it generates IRQ 0 to the PIC. With the PIT running at 1.193182MHz, this results in an interrupt frequency of 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 18.2Hz * 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 speaker, AdLib and Sound Blaster.
2. Running at 700Hz to play sound effects and music on the (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;
  // Interrupt refresh to either 140Hz or 700Hz
  rate = TickBase * t0CountTable[SoundMode];
  SDL_SetIntsPerSec(rate);
}
```

After handling the audio-related requests, the interrupt routine updates the system heartbeat. This step maintains a global timing reference that the system relies on for basic timekeeping. Specifically, the routine increments a 32-bit variable called `TimeCount`.

```
longword TimeCount;

static void interrupt SDL_t0Service(void)
{
  static word count = 1,
  switch (SoundMode)
  {
  case sdm_PC:
    SDL_PCService();
    break;
  case sdm_AdLib:
    SDL_ALSoundService();
    break;
  }

  if (!(--count))
  {
    // Set count to match 70Hz update
    count = t0CountTable[SoundMode];
    TimeCount++;
  }
  outportb(0x20,0x20);  // Acknowledge the interrupt
}
```

The heartbeat is updated at a rate of 70 units per second. These units are called "ticks". Every system in the engine uses this variable to pace itself; the engine will not refresh the screen until at least two ticks have passed, the game state machine expresses actor action duration in tick units, and the list goes on.

Trivia : Even if the game were played for a very long period, the 32-bit `TimeCount` counter would only overflow after almost two years.

4.13.5 Manage Refresh Timing

After each screen refresh, a certain number of ticks has passed. The tick count depends on several factors, such as the number of tiles refreshed, number of sprites and waiting factors such as the 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

In the time before Windows 95, there was no standard API for programming sound cards. Each game studio had to implement its own audio interfaces, and id Software was no exception. The Sound Manager provided support for both sound effects and music. Keen Dreams only used sound effects.

```
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);

boolean SD_SetSoundMode(SDMode mode);
```

Each implementation of sound effects relies on directly accessing the I/O port of one of three sound cards: AdLib, Sound Blaster (using the AdLib sound) and PC speaker.

Sound 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`.

```
//////////////////////////////////////////////////
// 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 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
```

Although the Sound Manager was designed to support music and digital sound playback on Sound Blaster and Disney Sound Source, this functionality was never implemented in Keen Dreams (as explained in section "Audio" on page 79) and will therefore not be further explained in this book.

4.13.7 PC Speaker

The hardware chapter highlighted a key challenge for sound effects. The default PC speaker was only capable of producing simple square waves. This limitation made the default speaker poorly suited for sound effects, highlighting a significant obstacle for game developers.

Earlier it was hinted that the PIT had three counters, of which counter #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.

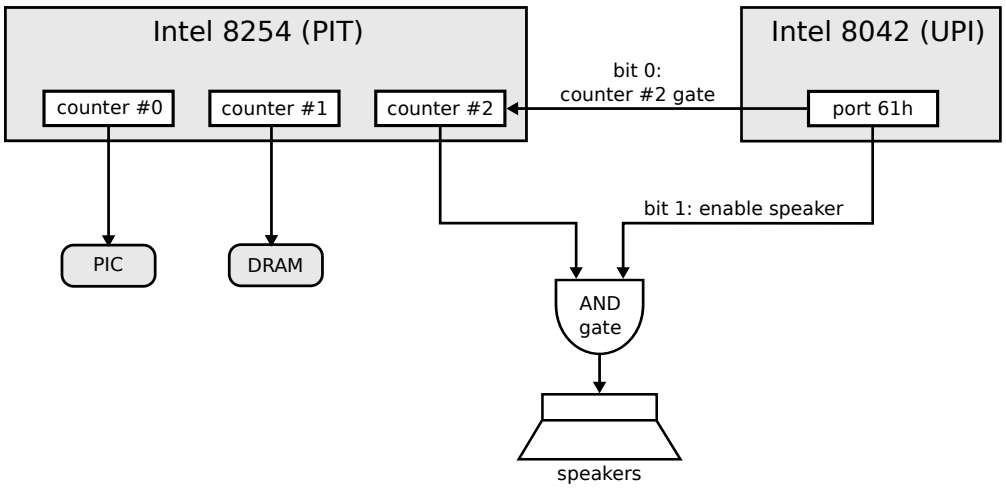


Figure 4.59: Built-in speaker hardware diagram.

A simple tone has only one frequency. As a practical example, say we want to play a middle C through the speaker, which is 261.626Hz[22]. With the PIT clock running at 1.193182MHz,

[22] 261.626Hz for middle C is a consequence of the twelve-tone equal temperament, see https://en.wikipedia.org/wiki/12_equal_temperament.

setting counter #2 to $\frac{1.193182MHz}{261.626Hz} = 4561$ results in playing the middle C through the speaker. To create a higher pitch, the counter value would need to be lower.

This integer value 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 adjust the frequency, write value `b6h` 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 4.60: Set PIT Command register (port `43h`) to value `b6h`[23].

When instructed to play a PC Speaker sound effect, the audio system sets itself to run at 140Hz via PIT Counter #0. Every time it wakes up, it reads the frequency to maintain for the next 1/140th of a second and writes it to Counter #2.

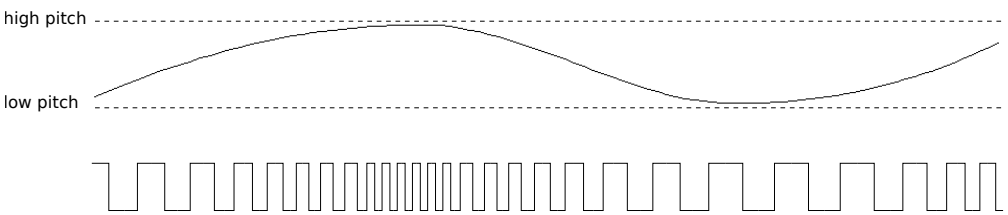


Figure 4.61: Sound pitch approximated with square wave and frequency changes.

The connection to the speaker can be deactivated without changing any timer parameters

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

through the system's keyboard controller (Intel 8042 UPI[24]). Setting bits 0 and 1 of port `61h` to 0 turns off both counter #2 and the speaker.

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

  s = *pcSound++;
  if (s != pcLastSample)
  {
  asm pushf
  asm cli
    pcLastSample = s;
    if (s)              // We have a frequency!
    {
      t = pcSoundLookup[s];
      asm mov bx,[t]

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

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

[24] Universal Peripheral Interface.

Frequencies are encoded in a stream of bytes (0-255) and decoded using the formula

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

Human hearing ranges from approximately 20Hz to 20,000Hz, making any counter value lower than 60 inaudible. The lowest frequency using the formula is 78Hz and the highest frequency is 19,886Hz. 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

At the heart of the AdLib was a Yamaha YM3812 music synthesis chip, which Yamaha called OPL2. The foundation of the OPL2 consists of 18 operators. Each operator contains an oscillator, envelope generator, and level controller. The OPL2 generates sound using a combination of oscillators and envelope generators.

The oscillator generates sine waves but can also apply several transformations to produce other waveforms. Besides the unmodified sine wave, it can generate three other waveforms.

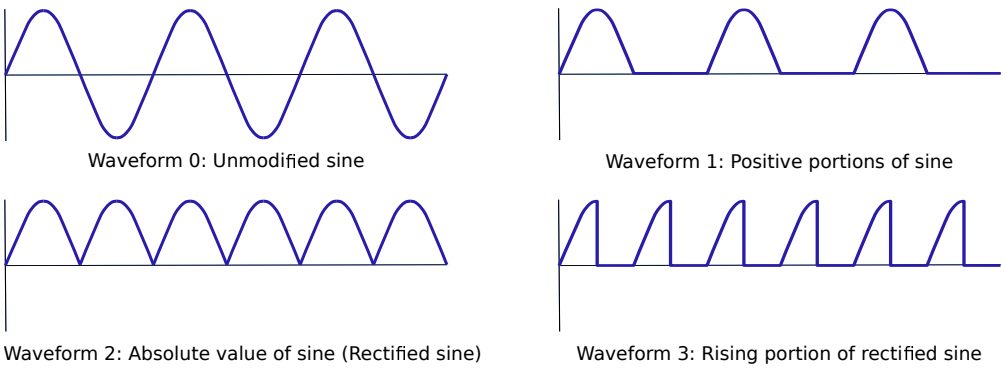


Figure 4.62: OPL2 oscillator waveforms.

The waves generated by the oscillator are fed into an envelope generator, which is responsible for creating a more realistic approximation of a real instrument. Each individual sound goes through several distinct stages that affect the output amplitude over time. These stages are called attack, decay, sustain, and release.

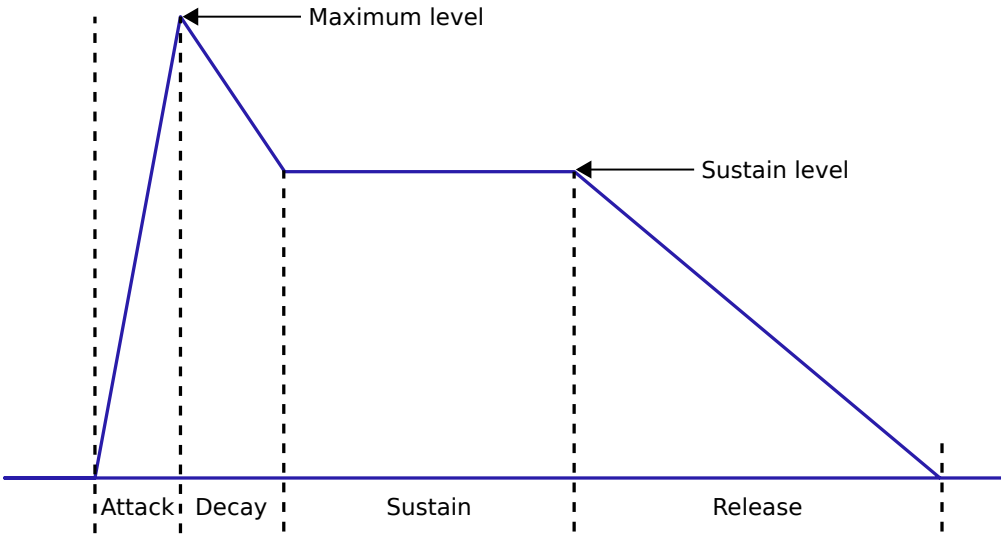


Figure 4.63: OPL2 envelope lifecycle.

The attack stage begins when a sound event occurs, starting from silence. Once maximum amplitude is reached, the decay stage reduces the amplitude until it reaches the sustain level. The amplitude is held at the sustain level as long as the composer holds the note. When a sound-off event occurs, the release stage gradually reduces the amplitude back to silence. By combining the oscillator and envelope generator, the OPL2 can produce surprisingly convincing simulations of real instruments. Finally, the level controller sets the overall volume sent to the speakers.

The OPL2 is organized into nine channels, where each channel consists of two operators. Each channel can control how its operators are connected to each other:

- FM synthesis uses two operators in series. The first operator ("modulator") modulates the second operator ("carrier") via its modulation data input.
- Additive synthesis uses two operators in parallel, adding both outputs together.

Most games used FM synthesis because it was easier to use and required less CPU overhead than additive synthesis. The output of all nine channels is summed together to produce the final sound sent to the speaker.

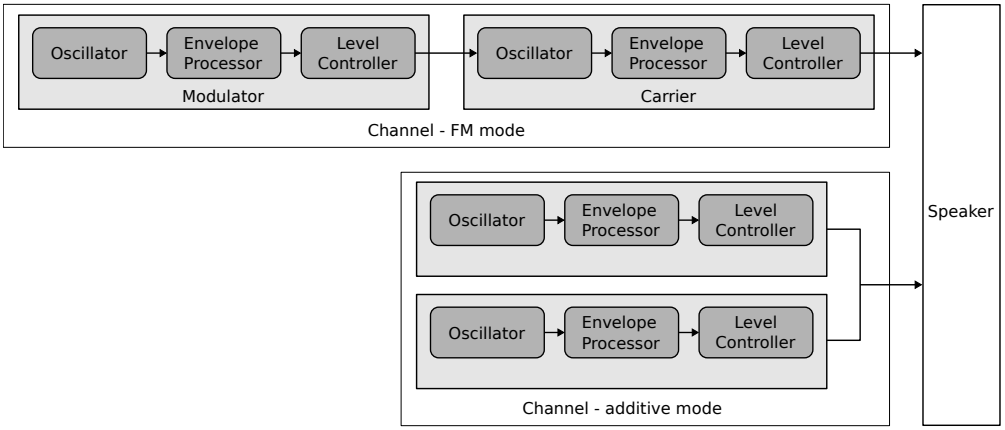


Figure 4.64: OPL2 channel architecture.

The OPL2 chip is controlled by three kinds of internal registers: chip-wide registers, per-channel registers, and per-operator registers. In total, 288 parameters could be configured. The available documentation for programming all these parameters from AdLib and Creative Labs was both cryptic and expensive. Programming the OPL2 was therefore, to say the least, quite challenging. Nevertheless, musicians were able to create wonderful sounds with the OPL2 chip and make lasting impressions on listeners.

Index reg.	D7	D6	D5	D4	D3	D2	D1	D0
01h			WSEnable					
02h	Timer 1 count (80 μsec resolution)							
03h	Timer 1 count (320 μsec resolution)							
04h	IRQReset	T1Mask	T2Mask				T2 Start	T1 Start
08h	CSW	NOTE-SEL						
20h-35h	Tremolo	Vibrato	Sustain	KSR	Frequency Multiplication Factor			
40h-55h	Key Scale Level		Output level					
60h-75h	Attack Rate				Decay Rate			
80h-95h	Sustain Level				Release Rate			
A0h-A8h	Frequency Number (lower 8 bits)							
B0h-B8h			KEY-ON	Block number			F-Num (hi bits)	
BDh	Trem Dep	Vibr Dep	PercMode	BD on	SD on	TT on	CY on	HH on
C0h-C8h					FeedBack Modulation Factor			Additive
E0h-F5h							Waveform Select	

chip-wide per channel per operator

Figure 4.65: OPL2 parameters reference overview.

All these parameters were programmed via two external I/O ports:

- `388h` - Index register (write), Status register (read)
- `389h` - Data register (write only)

Every time the timer interrupt is triggered, it checks whether a sound effect has been received and plays the next effect 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
}
```

One notable detail is the `SDL_Delay()` call. At first glance, introducing delays in the time-critical audio routine seems counterintuitive. The OPL2 chip was designed with a "fire-and-forget" register system. When the CPU wrote a value to a register, the chip required a short amount of time to apply the change before accepting the next write.

When the AdLib card was released in 1987, the PC XT and AT were not fast enough to outpace writing to the AdLib card. However, with the introduction of the Intel 386 (on which Commander Keen was developed), commands were sent to the card more quickly than AdLib had anticipated. If a program sent data too quickly, the new command would overwrite the previous one before it was processed, leading to garbled sound. AdLib therefore recommended inserting artificial delays between register writes.

> “ Because of the nature of the card, you must wait 3.3 µsec after a register select write, and 23 µsec after a data write.
>
> **- AdLib programming guide, by Tero Töttö** ”

It is not readily apparent how the value of `TimerDelay10` was chosen to produce a delay of 3.3 μs. Later id Software games replaced this delay with six (`in al,dx`) instructions that repeatedly read from I/O port 388h.

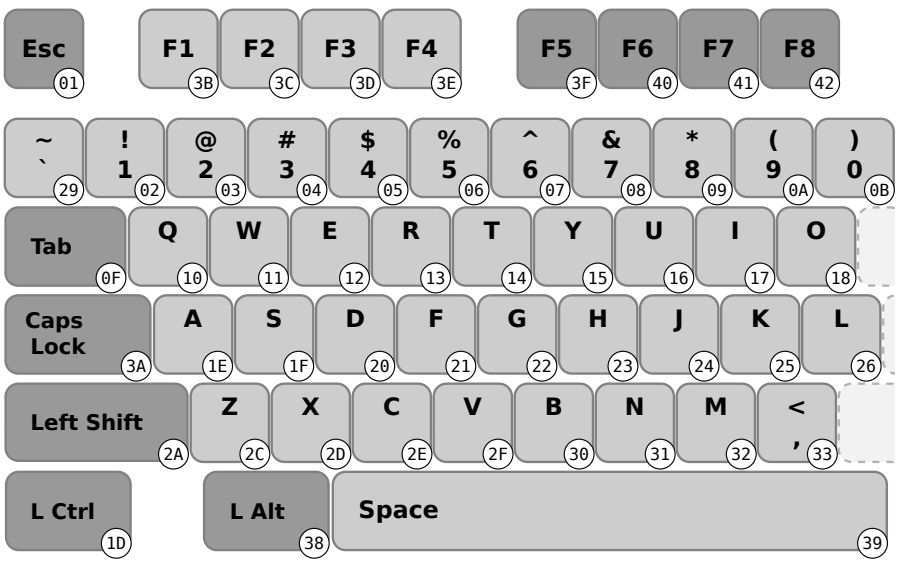
4.14 Inputs

In the time before Windows, developers needed to write custom drivers for each input device. This required direct communication with hardware using the vendor's protocol over physical ports, and the keyboard was no exception.

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.

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.



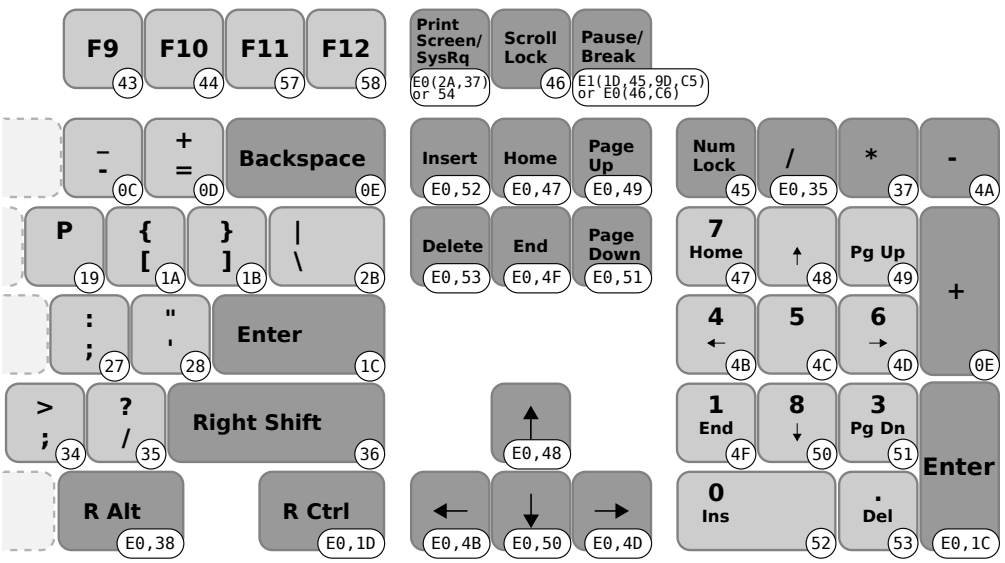
To save encoding space, these keys are sent as two-byte scancodes: the first byte is always `E0h`, followed by the make or break code of the equivalent key from the original AT layout. For instance, the standard "/"-key produces the scancode `35h`, while the numeric keypad's "Num /"-key sends `E0h` followed 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. The picture below on this and previous page shows the full IBM PS/2 keyboard scancode layout.

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 "?".*

The Input Manager maintains several lookup tables to translate the scancodes to ASCII or special codes

- Shifted and unshifted ASCII codes tables.
- Table for the 14 duplicate keys with `E0h` prefix.
- Special keys like "ESC", "Shift", "Backspace", etc.



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[25] 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.

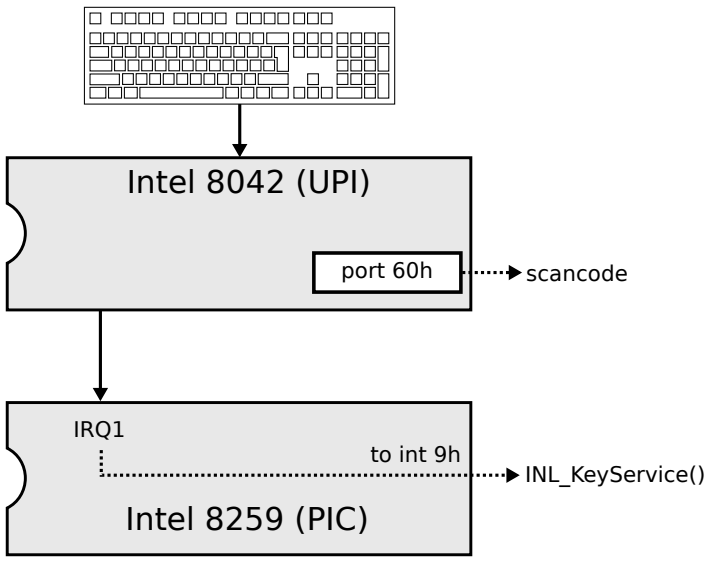


Figure 4.66: Keyboard hardware diagram.

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 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. Again, the engine installs its own ISR there.

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

static void INL_StartKbd(void) {
  IN_ClearKeysDown();
  OldKeyVect = getvect(KeyInt);
  setvect(KeyInt,INL_KeyService); //install custom ISR
}
```

[25] Programmable Peripheral Interface

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];
```

```
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
}
```

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 or higher based systems.

4.15 Random Tricks

In this section a few small but useful tricks are highlighted, that don't really fit anywhere else in the book.

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 xspeed (for vertical walls) or the yspeed (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.

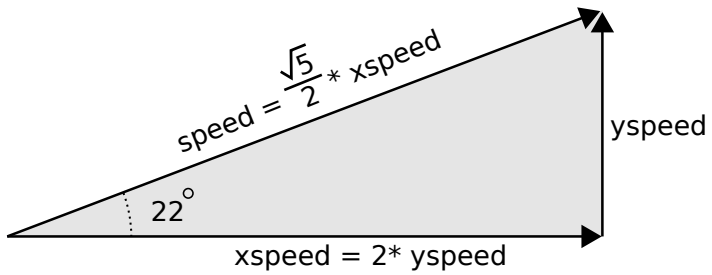


Figure 4.67: Simplify to 22° for xspeed more than twice the yspeed.

To simplify the calculations, the game restricts incoming angles to four discrete values – 22°, 45°, 67° or 90° – based on the ratio of `xspeed` to `yspeed`. Eight angle indices are then defined, where indices 0–3 represent negative `xspeed`, while indices 4–7 represent positive `xspeed`. The resulting speed vector magnitude is calculated using either `xspeed` or `yspeed`, depending on which has the greater absolute value. This magnitude is multiplied by 256 to achieve higher precision.

```
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*256
            }
            else                   // 45 degrees
            {
                angle = 1;
                speed = absx*362; // x*sqrt(2)*256
            }
        }
        [...]        // Handle 67 and 90 degrees
    }
    if (ob->xspeed > 0)
        angle = 7-angle;
}
```

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

	incoming angle index							
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 4.68: 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° (angle index 0) and positive `xspeed`, the lookup table specifies bounce angle #5. Each bounce angle is then decomposed into a new `xspeed` and `yspeed`.

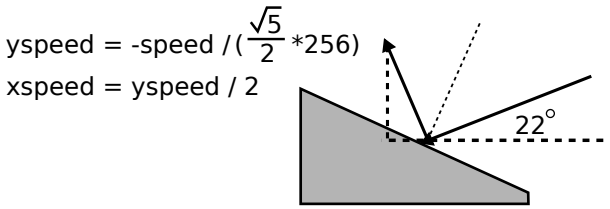


Figure 4.69: Wall type 3 with incoming angle of 22°.

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° (angle index 0) on a 45° slope (wall type 4) results in a bounce angle of 90°, rather than the expected 67°.

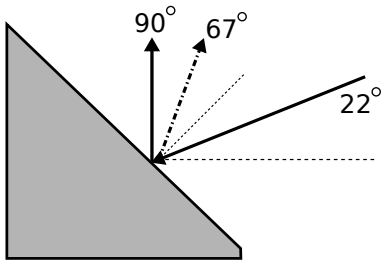


Figure 4.70: Incoming angle of 22° results in bounce angle of 90°.

```
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;
case 6:
  ob->xspeed = ob->yspeed = -(speed / 362);
  break;

[...]
}
```

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.

In total, six pre-defined color palettes are stored in the `color[][]` table, ranging from completely black to completely white. The default table with CGA colors is defined in row 3.

#	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 4.71: Color palette table `colors[][]`.

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 series, *Commander Keen in Invasion of the Vorticons*, was only released for the EGA video card. Keen Dreams 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!

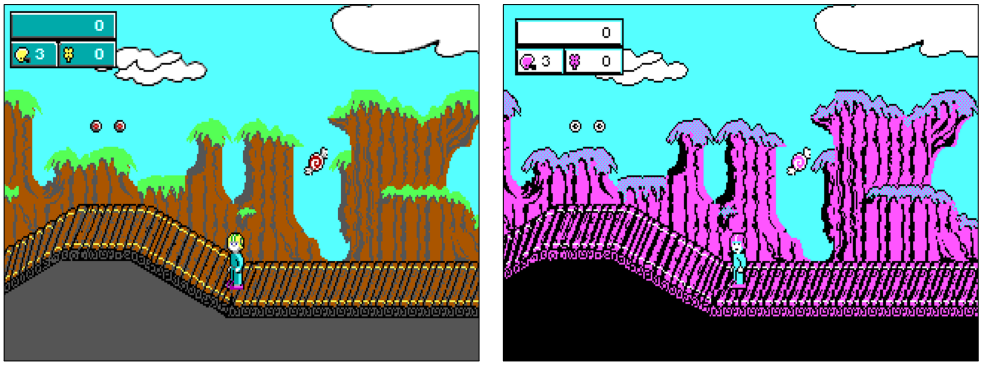


Figure 4.72: Keen Dreams EGA and CGA version.

4.16.1 CGA Video card

The Color Graphics Adapter (CGA), originally also called the 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.

The next page shows the original IBM CGA card, a clunky beast full of discrete components.

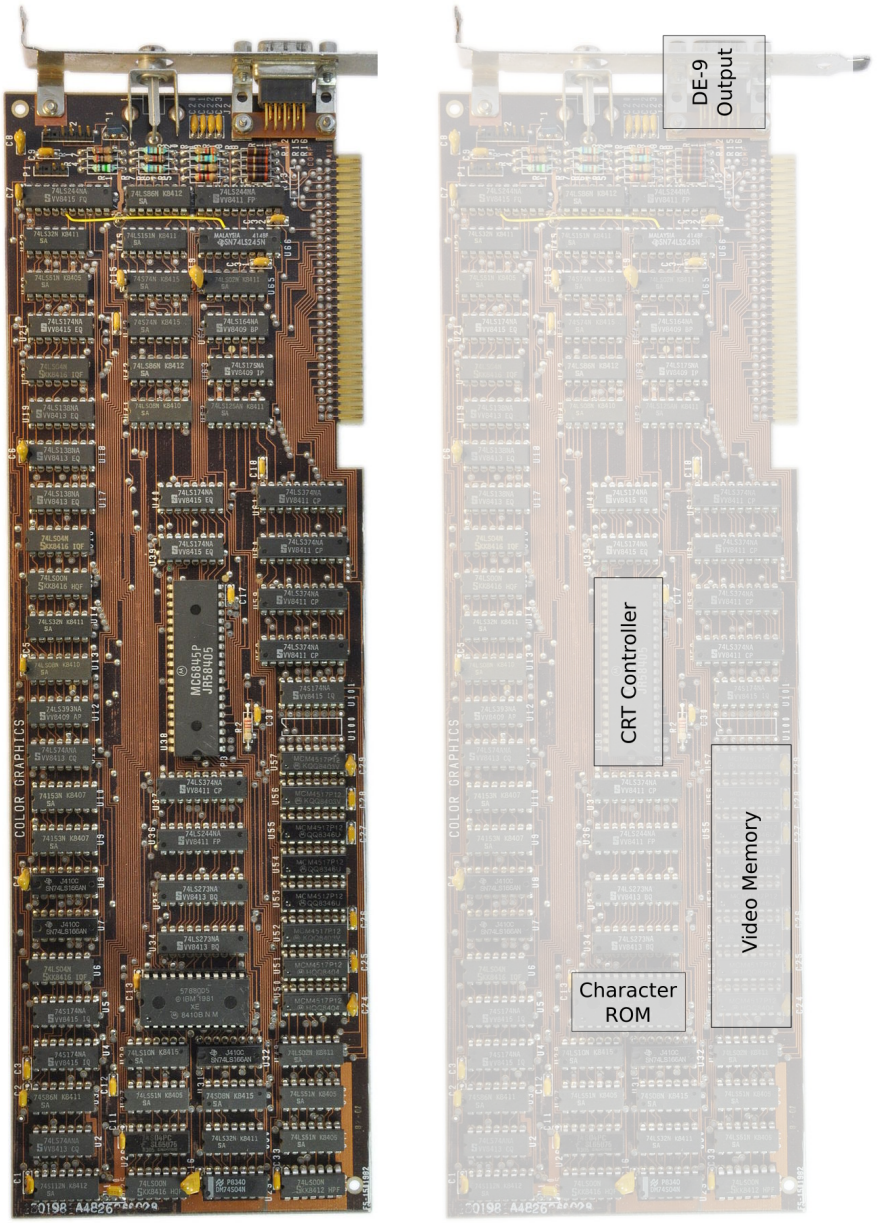


Figure 4.73: Original IBM CGA card

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 4.74: 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 4.75: 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 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.

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.

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.

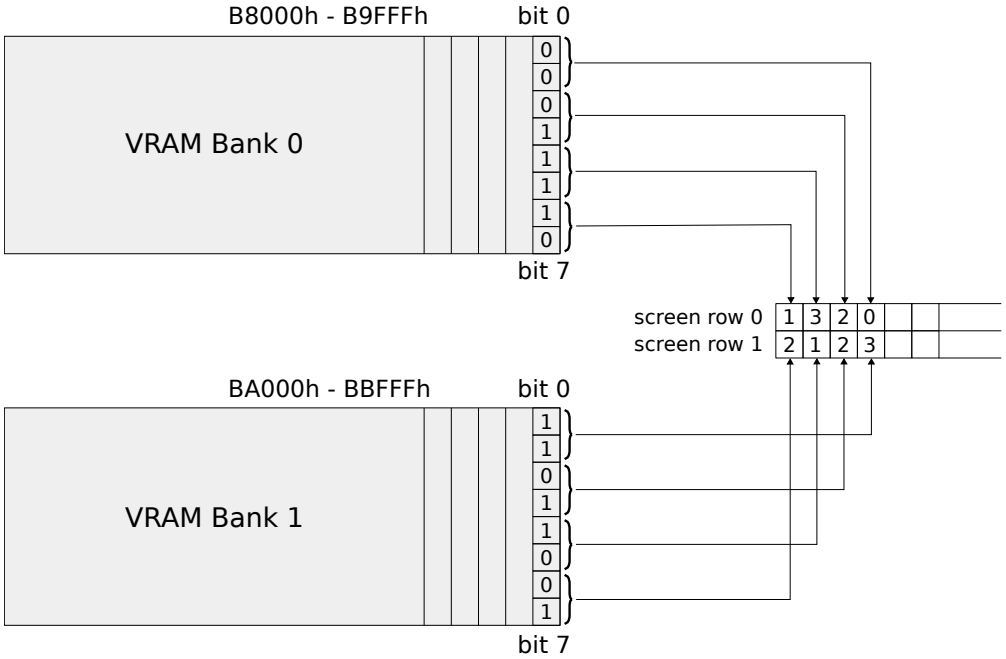


Figure 4.76: CGA interlaced memory.

The CGA card employed memory mapping similar to EGA. In Mode 4, VRAM bank 0 is mapped to memory addresses ranging from `B800:0000h` to `B800:1FFFh` (absolute address `B8000h`–`B9FFFh`), and VRAM bank 1 is mapped to `B800:2000h` to `B800:3FFFh` (absolute address `BA000h`–`BBFFFh`). Unlike EGA, the CGA memory model does not require masking because the total 16KiB of VRAM easily fits within a 64KiB memory segment.

4.16.3 Double buffering

A full display screen in mode 4 requires 320 pixels×2 bits per pixel×200 lines, which equals 16,000 bytes of memory. Since the display screen consumes all 16KiB 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 64KiB 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 master page. The buffer page starts at offset 0000h, while the master page begins at 8000h. Both pages float within the 64KiB memory segment, leveraging the same memory wrapping mechanism described in section "Virtual Screen Tile Refresh" on page 128.

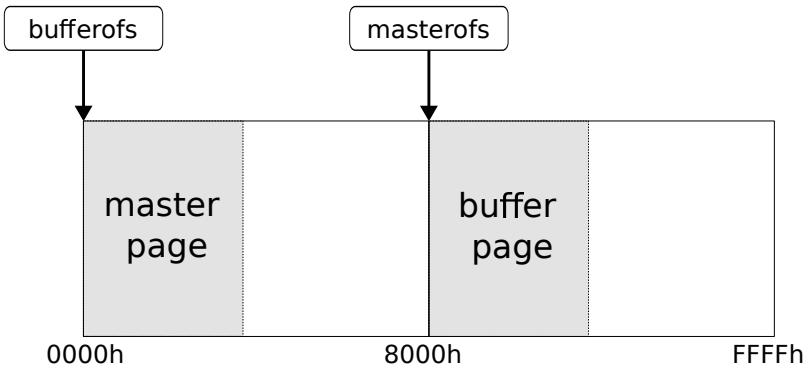


Figure 4.77: CGA double buffering memory layout in RAM.

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, smooth horizontal scrolling is limited to steps of four pixels, which means a more choppier 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 splitting the linear memory buffer into copying all even rows to VRAM bank 0 and odd rows to VRAM bank 1.

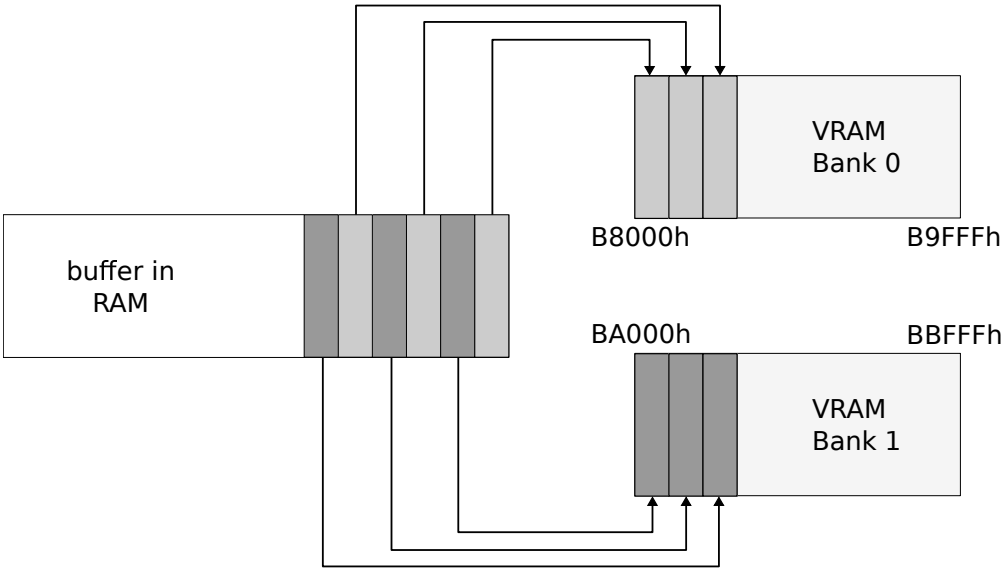


Figure 4.78: Copy RAM buffer to VRAM in CGA.

The CPU lacked the speed to copy the entire RAM buffer to VRAM during the vertical retrace, which was the only window where the CGA hardware was not actively scanning to the display. Consequently, screen tearing in the CGA version of Commander Keen was unavoidable.

Trivia : Because the video memory on the original IBM CGA was not dual-ported, when both the CPU and the video output attempted to access the same byte of video RAM, the CPU took precedence. This causes the card to read random values, resulting in 'snow' artifacts on the display.

Chapter 5

Epilogue

Upon release, Commander Keen in Invasion of the Vorticons was an immediate hit, generating over US$60,000 per month. In the summer of 1991, id Software hosted a seminar for game developers with the intention of licensing the Commander Keen engine. This effort became a spiritual predecessor to id Software's later, more formal practice of licensing its game engines.

Although this licensing initiative was not as successful as the later idTech engines, the Commander Keen engine was used in several games, including:

- Dangerous Dave
- Shadow Knights
- Bio Menace
- ScubaVenture

A third trilogy of episodes, titled *The Universe Is Toast*, was planned for release in December 1992. id Software worked on it for a couple of weeks before shifting its focus to *Wolfenstein 3D*, and the project was never resumed.

In 1999, John Carmack was considering developing a Commander Keen game for the Game Boy Color. Sometime afterward, id Software approached Activision with the idea. Activision recommended David A. Palmer Productions to develop the game, with oversight from id Software. In 2001, ten years after the series' first release, Commander Keen appeared on the Game Boy Color.

In June 2019, during Bethesda's E3 conference, a new Commander Keen game for iOS and Android devices was announced. It was planned for release in the summer of the same year; however, no release or further announcements followed, and by June 2020 all references to the game had been removed from Bethesda and ZeniMax websites.

5.1 The Dopefish legacy

While the Commander Keen franchise declined after its initial success, one character from the series created its own legacy: the Dopefish. It was created by Tom Hall and it first appeared in the "*Well of Wishes*" level of Commander Keen 4.

> The second-dumbest creature in the universe, this creature's thought patterns go "swim swim hungry, swim swim hungry." They'll eat anything alive and moving near them, though they prefer heroes.

Soon, the Dopefish began appearing in other games, often as a quiet in-joke: carved into walls in *Doom*, hidden on computer screens in *Quake*, floating on signs and textures in *Quake II* and *Quake III Arena*. Later, it surfaced in unexpected places, such as graffiti in *Max Payne*, posters in *Daikatana*, and as a hobblehead in *Hitman 2*. Its most recent appearance, as of writing this book, is in *Doom Eternal* (2020)[1].

[1] For all appearances, see https://www.dopefish.com/fishinfo.html.

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.

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.

Level 6, 8, 11 and 13 are secret levels in Keen Dreams that cannot be reached from the map. They can only be accessed using the cheat code `F10+W` (warp). It is unknown whether or not these incomplete levels were intentional, development of Keen Dreams was rushed, or because of some other problem. It is also worth noting that all these levels have been removed from the original registered versions.

The second file contains all graphical assets. Most 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 or 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.

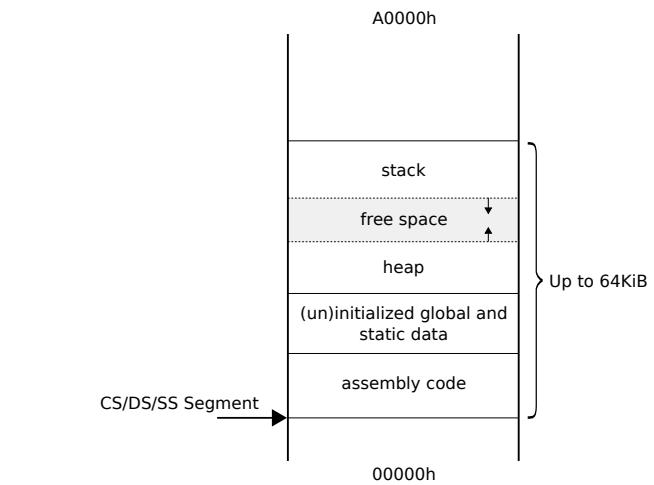


Figure B.1: Tiny memory model layout, total of 64 KiB.

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.

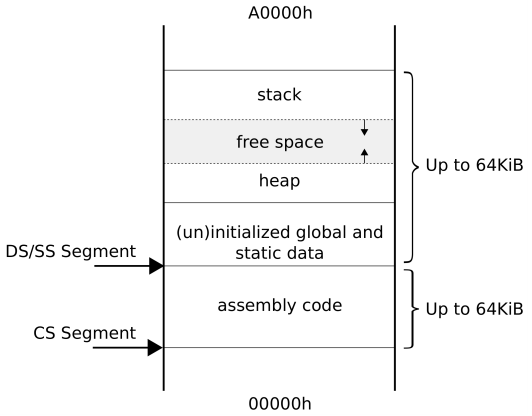


Figure B.2: Small memory model layout, code and data each have 64KiB.

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.

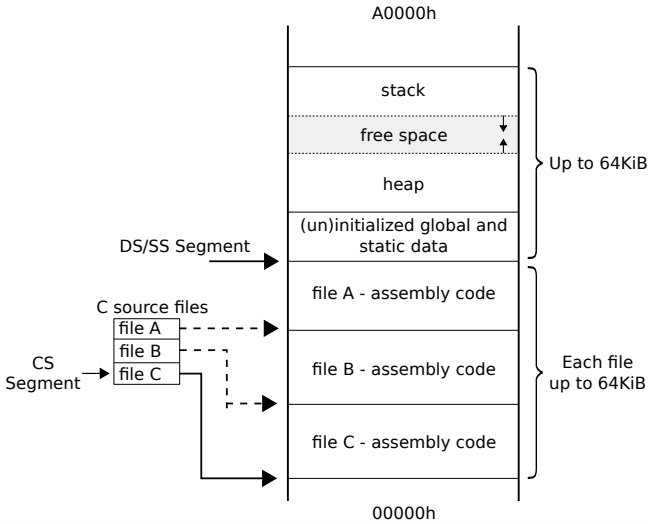


Figure B.3: Medium memory model layout, code can be larger than 64KiB.

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.

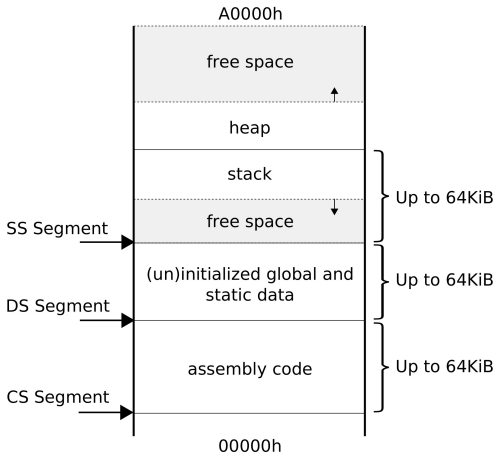


Figure B.4: Compact memory model layout, data can be larger than 64KiB.

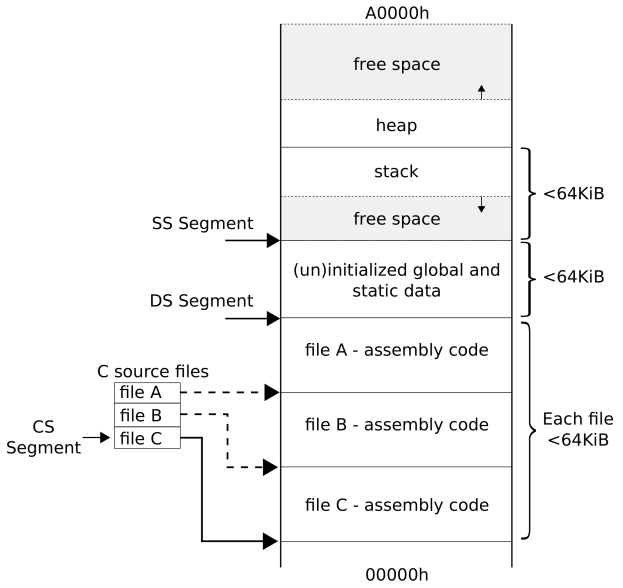


Figure B.5: Large memory model layout, code and data can be larger than 64KiB.

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.

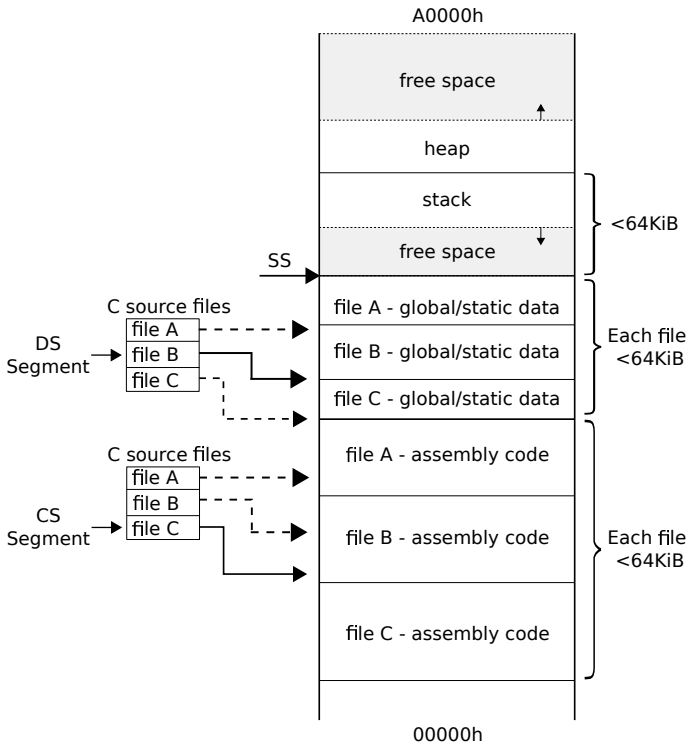


Figure B.6: Huge memory model layout, code and global data can be larger than 64KiB.

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 Refreshment*. He discussed the idea with his 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.

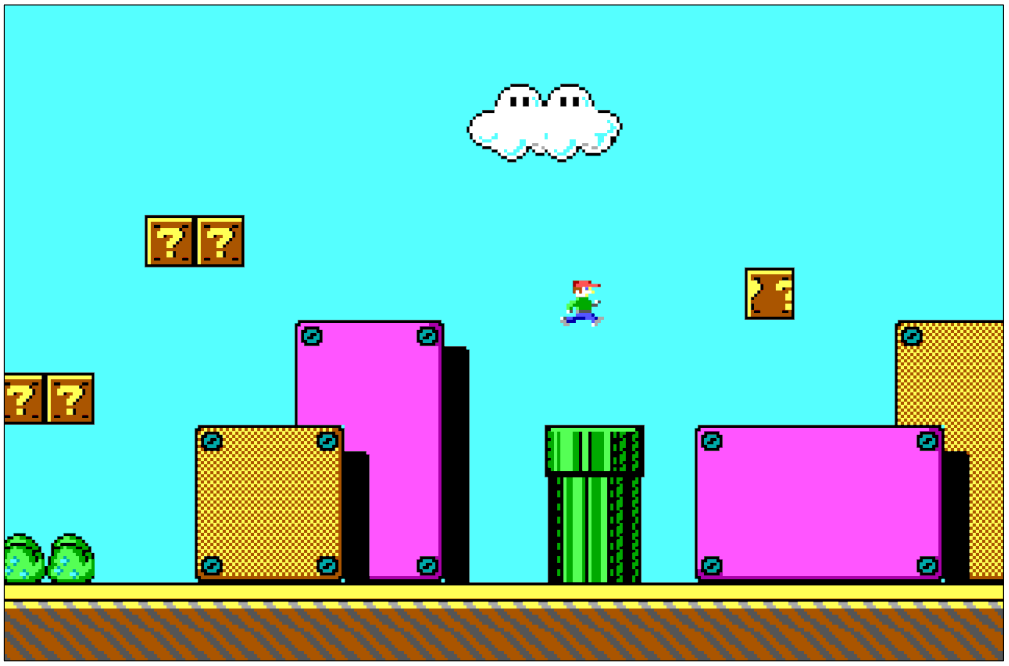


Figure C.1: Dangerous Dave in Copyright Infringement demo

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 sent 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

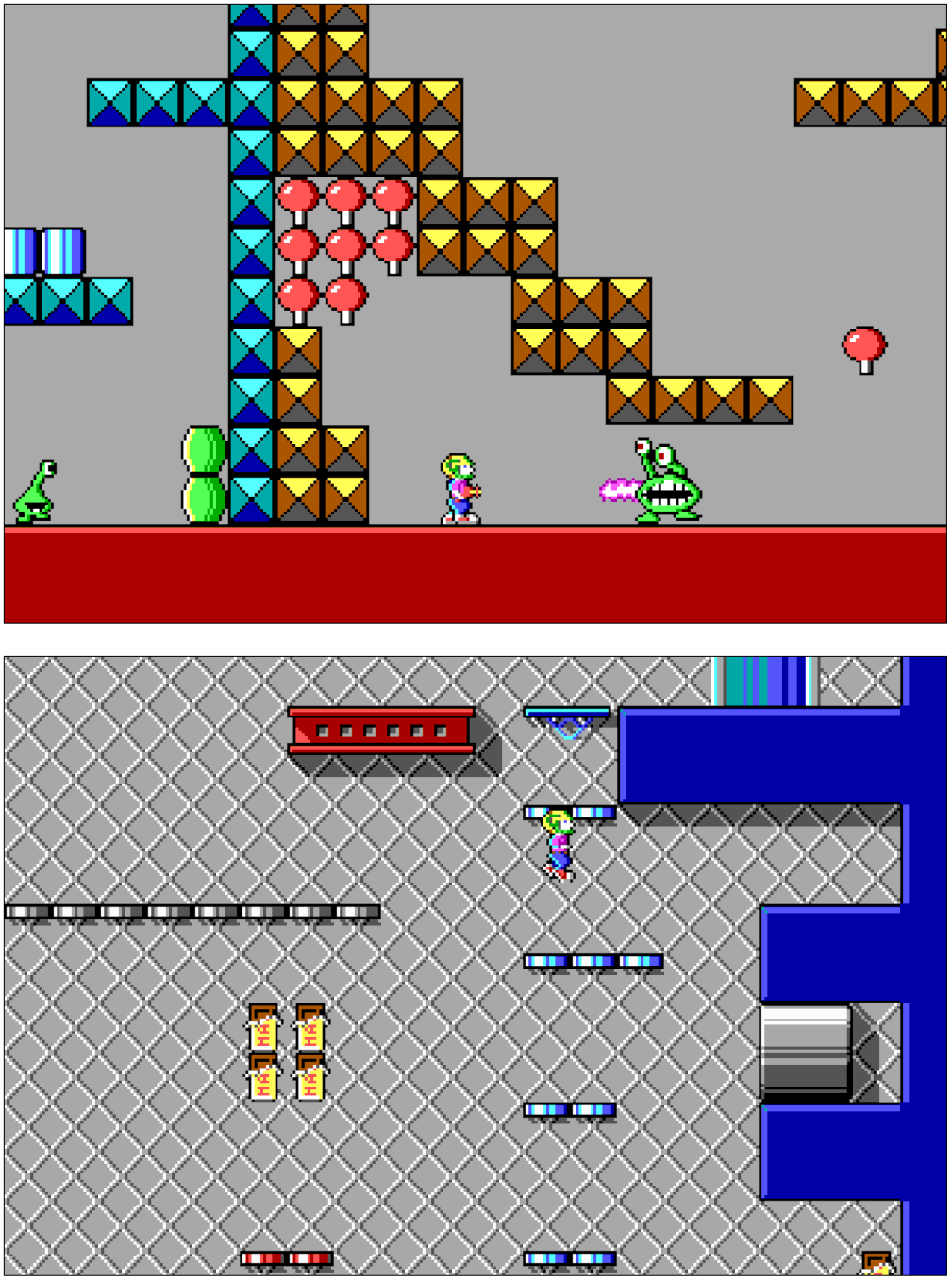
Around the same time the group was rejected by Nintendo, Scott Miller of Apogee Software approached Romero. They agreed to make *Commander Keen in Invasion of the Vorticons*, to be published by Apogee Software. The team could not afford to leave their jobs to work on the game full-time, so they continued 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].

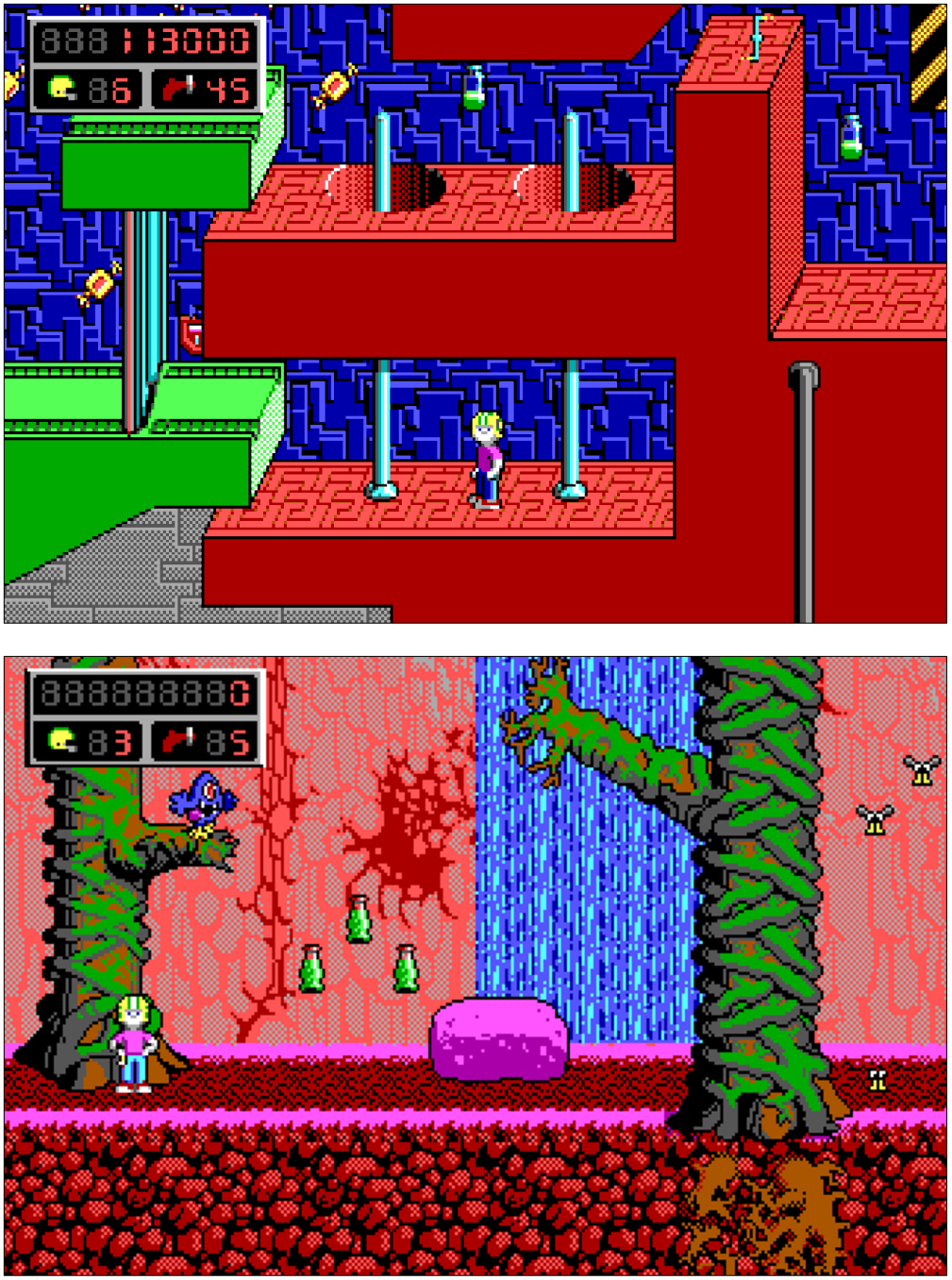
> "I told them we need to start a company, do our own game and publish it, outside of Softdisk. Jay Wilbur happened by the office and I told him that after what had been done by John and Tom the night before, we were outta there. He kinda laughed and said, "Heheh, yeah..." and I said, "No. I'm serious - 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



Keen I - *Marooned on Mars* (above) and Keen II - *The Earth Explodes* (below).

Keen III - *Keen Must Die!* (top) and Keen IV - *Secret of the Oracle* (bottom).



Keen V - *The Armageddon Machine* (top) and Keen VI - *Aliens Ate My Babysitter* (bottom).

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.

The final Commander Keen 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 publish it as a retail title through FormGen.

Trivia : Even though this is the last game in the series, it was not the last one developed. *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.

In the years that followed, id Software became one of the most successful and influential game development studios in the industry, largely due to the Wolfenstein 3D, Doom, and Quake franchises, as well as its pioneering work in game engine licensing. id Software was acquired by ZeniMax Media in 2009, which was subsequently acquired by Microsoft in 2020.