

# GAME ENGINE BLACK BOOK

WOLFENSTEIN 3D

FABIEN SANGLARD

1st EDITION

# Copyright

In order to illustrate how the Wolfenstein 3D game engine works, a few 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:

1. All in-game screenshots, title screen, signon screen, total carnage screen.
2. All in-game menu screenshots (main menu, sound menu).
3. All 3D sequence textures such as blue wall, wood, dark wood, grid wall.
4. All 3D sequence sprites such as brown guard, dead guard, dead dog.
5. All 3D sequence images used for the HUD.
6. All screenshots of Spears of Destiny, Catacomb 3D, and Hover Tank 3D.



# Acknowledgment

Thanks to John Carmack, Romain Guy, Victoria Ho, and Aurelien Sanglard for generously helping. This project would have never materialized without them.

Thanks to Jim Leonard and Foone Turing who volunteered their fleet of 286s, 386s, and VGA cards to accurately benchmark Wolfenstein 3D.

Thanks to Jim Leonard for sharing his encyclopedic knowledge of PC system architecture and programming. His patience in explaining sound systems and extended memory systems helped his book to ship with accurate information.

Thanks to Chet Haase, Daniel Thornburgh, Xiao Yu, and Chris Forbes for proofreading and catching mistakes.

– Fabien Sanglard  
[fabiensanglard.net@gmail.com](mailto:fabiensanglard.net@gmail.com)



# Foreword

Fabien's commentary on the classic game engine codebases have been a wonderful resource on the web, so I was thrilled that he decided to start expanding them all the way to book length. While often overshadowed by Doom, Wolfenstein 3D does hold a significant place in video game history, and it remains fun to run around in today, just like dropping a quarter in a Pac Man machine.

Despite being open source, the 16 bit code and assembly language is not easy to build or experiment with, so far fewer people have looked into it than the later codebases. The most remarkable thing about the project from today's perspective is just how small it was: one little directory of code files with no external dependencies. Back then I barely trusted (with some reason!) the C standard library implementations that we had to work with, so almost everything was done in those few source files.

I was 21 years old when I wrote most of the code, and I had only been programming in C for a year, so it is far from a masterpiece of coding style, but there are still some things that were done well. Compiled scalers are a case of code specialization taken to the extreme, which, combined with the low level VGA trick of multi column writes and the progressive performance characteristics of ray casting make it much more even in framerate than a conventional approach. The choice to use ray casting was also an important pragmatic decision. I wasn't experienced enough yet to do a solid implementation of a polygon, or even line based, engine. Ray casting got me where I needed to be at an acceptable cost.

So, set the way-back machine for 1992.

ACHTUNG!

– John Carmack



# Contents

<b>Acknowledgment</b>	<b>3</b>
<b>Foreword</b>	<b>5</b>
<b>Prologue</b>	<b>11</b>
<b>1 Introduction</b>	<b>13</b>
<b>2 Hardware</b>	<b>19</b>
2.1 CPU: Central Processing Unit . . . . .	20
2.1.1 Overview . . . . .	20
2.1.2 The Intel 80386 . . . . .	21
2.1.3 Floating Point . . . . .	25
2.2 RAM . . . . .	30
2.2.1 DOS Limitations . . . . .	30
2.2.2 The Infamous Real Mode: 1MiB RAM limit . . . . .	31
2.2.3 The Infamous Real Mode: 16 bit Segmented addressing . . . . .	33
2.2.4 Extended Memory . . . . .	35
2.3 Video . . . . .	38
2.3.1 History of Video Adapters . . . . .	39
2.3.2 VGA Architecture . . . . .	41
2.3.3 VGA Planar Madness . . . . .	44
2.3.4 VGA Modes . . . . .	46
2.3.5 VGA Programming: Memory Mapping . . . . .	47
2.3.6 VGA Programming: Mode 12h . . . . .	48
2.3.7 VGA Programming: Mode 13h . . . . .	50
2.3.8 The Importance of Double Buffering . . . . .	54
2.4 Audio . . . . .	55
2.4.1 AdLib . . . . .	57
2.4.2 Sound Blaster . . . . .	58
2.4.3 Sound Blaster Pro . . . . .	59
2.4.4 Disney Sound Source . . . . .	60
2.5 Inputs . . . . .	61

2.6	Bus . . . . .	62
2.7	Summary . . . . .	64
<b>3</b>	<b>Team</b>	<b>67</b>
3.1	Organization . . . . .	71
3.2	Programming . . . . .	74
3.3	Graphic Assets . . . . .	77
3.4	Assets Workflow . . . . .	80
3.5	Maps . . . . .	85
3.6	Audio . . . . .	89
3.6.1	Sounds . . . . .	89
3.6.2	Music . . . . .	90
3.7	Distribution . . . . .	93
<b>4</b>	<b>Software</b>	<b>99</b>
4.1	Getting the Source Code . . . . .	99
4.2	First Contact . . . . .	99
4.3	Big Picture . . . . .	102
4.3.1	Unrolled Loop . . . . .	102
4.4	Architecture . . . . .	105
4.4.1	Memory Manager (MM) . . . . .	108
4.4.2	Page Manager (PM) . . . . .	111
4.4.3	Video Manager (VL & VH) . . . . .	114
4.4.4	Cache Manager (CA) . . . . .	114
4.4.5	User Manager (US) . . . . .	115
4.4.6	Sound Manager (SD) . . . . .	116
4.4.7	Input Manager (IN) . . . . .	116
4.5	Startup . . . . .	116
4.5.1	Signon . . . . .	117
4.5.2	Solving the VGA Problem . . . . .	119
4.5.3	Profound Carnage . . . . .	127
4.6	Menu Phase: 2D Renderer . . . . .	128
4.7	Action Phase: 3D Renderer . . . . .	132
4.7.1	Life of a Frame . . . . .	133
4.7.2	Life of a 3D Frame . . . . .	135
4.7.3	3D Setup . . . . .	141
4.7.4	Clearing the Screen . . . . .	148
4.7.5	Solving the CPU Problem . . . . .	150
4.7.6	Drawing Walls . . . . .	180
4.7.7	Drawing Sprites . . . . .	198
4.7.8	Drawing Weapons . . . . .	208
4.7.9	A.I . . . . .	208
4.8	Audio and Heartbeat . . . . .	215

4.8.1	IRQs and ISRs . . . . .	216
4.8.2	PIT and PIC . . . . .	218
4.8.3	Heartbeats . . . . .	218
4.8.4	Audio System . . . . .	218
4.8.5	Music . . . . .	219
4.9	Sound Effects . . . . .	221
4.9.1	Sound Effects: AdLib . . . . .	224
4.9.2	Disney Sound Source System: PCM . . . . .	224
4.9.3	SoundBlaster System: PCM . . . . .	224
4.9.4	SoundBlaster Pro System: Stereo PCM . . . . .	224
4.9.5	PC Speaker: Square Waves . . . . .	225
4.9.6	PC Speaker: PCM . . . . .	228
4.9.7	PC Speaker: PWM . . . . .	230
4.10	User Inputs . . . . .	231
4.10.1	Keyboard . . . . .	231
4.10.2	Mouse . . . . .	232
4.10.3	Joystick . . . . .	233
4.11	Tricks . . . . .	236
4.11.1	Cos/Sin Table Lookup . . . . .	236
4.11.2	FizzleFade . . . . .	237
4.11.3	Palette . . . . .	245
4.12	Pseudo Random Generator . . . . .	247
4.13	Performance . . . . .	250
<b>5</b>	<b>Sequels</b>	<b>255</b>
5.1	Spear of Destiny . . . . .	255
<b>6</b>	<b>Ports</b>	<b>261</b>
6.1	Super Nintendo . . . . .	261
6.2	Jaguar . . . . .	264
6.3	iPhone . . . . .	268
6.3.1	iPhone Development Notes . . . . .	269
6.4	Wolfenstein 3D-VR . . . . .	279
<b>7</b>	<b>Epilogue</b>	<b>283</b>
7.1	Where Are They Now? . . . . .	284
<b>Appendices</b>		<b>287</b>
<b>A</b>	<b>Before Wolfenstein 3D</b>	<b>289</b>
A.1	Hovertank 3D . . . . .	289
A.2	Catacomb 3D . . . . .	289
<b>B</b>	<b>XMS vs EMS</b>	<b>293</b>

B.1	EMS: Expanded Memory Specification . . . . .	293
B.2	XMS: eXtended Memory Specification . . . . .	295
B.3	What It Meant For Wolfenstein 3D . . . . .	296
<b>C</b>	<b>The 640KB Barrier</b>	<b>297</b>
<b>D</b>	<b>CONFIG.SYS and AUTOEXEC.BAT</b>	<b>301</b>
<b>E</b>	<b>Good Stuff</b>	<b>303</b>
<b>F</b>	<b>Release Notes by John Carmack</b>	<b>305</b>
<b>G</b>	<b>20th Anniversary Commentary</b>	<b>307</b>
G.1	The engine (4:00 mark) . . . . .	307
G.2	Modding . . . . .	308
G.3	Texture mapping (13:30 mark) . . . . .	309

# Prologue

For the past ten years, I have been writing articles explaining the internals of game engines. It all started back in 1999 when I downloaded the freshly open sourced code of Quake and eagerly opened it with Visual Studio 6.0. After a few days of struggling, I deleted `quakesrc` folder, discouraged and unable to make sense of anything.

A few years later I came across the legendary *Graphics Programming Black Book* by Michael Abrash. His articles explained the big picture of the Quake engine and detailed the now famous Binary Space Partition system<sup>1</sup>, Potentially Visible Set<sup>2</sup> and its compression techniques. Now knowing what to expect, I went back to the code and understood it deep down.

I thought many other programmers may be like me: capable but discouraged by apparent complexity. So I started to write “source code reviews” and uploaded them on my website. Over the years, I wrote more than fifty articles, selecting legendary games such as Doom, Quake, or Out Of This World. I would open the engine, explore the subsystems and the overall architecture, and draw a map that hopefully sparked interest and encouraged other adventurous programmers.

Sharing my knowledge was a rewarding experience. Not only was the feedback from readers positive, but explaining something in simple terms is an excellent way to make sure one masters a topic. It tremendously improved both my capacity to ingest a large volume of code and my communication skills. I learned to rely extensively on drawings (a picture is worth  $2^{20}$  words<sup>3</sup>) and as a result this book features hundreds of them. These skills proved invaluable in a career which ultimately led me to working at Google on Android.

---

<sup>1</sup>Chapter 59 in Michael Abrash Graphic Programming Black Book.

<sup>2</sup>Chapter 64 in Michael Abrash Graphic Programming Black Book.

<sup>3</sup>“Code: The Hidden Language of Computer Hardware and Software” by Charles Petzold is a superb example.

Eventually I decided to take my articles to the next level and came up with this book, which covers the first of the three milestone hardware and engine combinations of the 90s:

1. Wolfenstein 3D (1992) and the i386.
2. Doom (1993) and the i486-DX2.
3. Quake (1996) and the Pentium.

It may appear like a waste of time to read and write about “old” engines dedicated to extinct machines, compilers, and operating systems, but they carry tremendous value. Not only are they packed with clever tricks, they also remind us of the constraints programmers from the past had to overcome. They remind us of the spirit it once took to reach new frontiers.

Things have not changed much. These days we may deal with gigabytes, dedicated hardware accelerators, and multi-core CPUs but the spirit it takes to keep on moving forward remains the same. To those who struggle today, keep in mind you are not alone. Others have struggled before. Some have found fame and some have found fortune but in the grand scheme of things we all belong to a family of people who roll up their sleeves and try to make things better with hard work. Wherever it takes you, be proud of your labor. Be proud of your passion and keep on looking for The Right Thing to Do<sup>4</sup>!

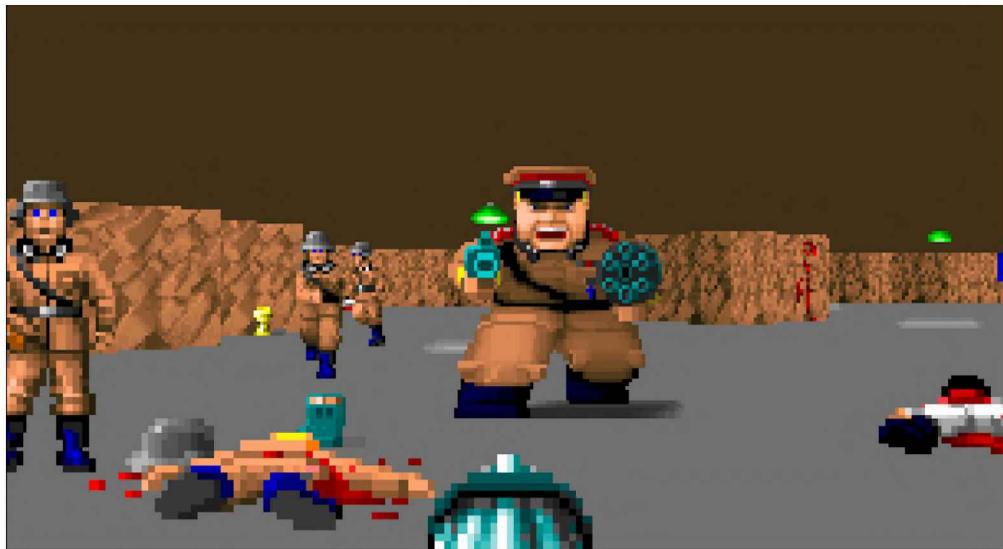
---

<sup>4</sup>“Hackers: Heroes of the Computer Revolution” by Steven Levy.

# Chapter 1

## Introduction

Wolfenstein 3D, released on May 5th, 1992, established the First Person Shooter genre. The game design, powered by an engine enabling beautiful 256 color graphics, speed, high framerate, clever AI, crisp sound effects, and engaging music, was universally acclaimed. Within a year more than 100,000 units had been sold<sup>1</sup>, bringing fame and a little bit of fortune to the team who built it: id Software.



However, fans did not stop at beating the game. Driven by a desire to modify it and make

---

<sup>1</sup>The game was distributed via shareware.

their own characters and maps, they started to explore and reverse engineer. Within a few months the asset formats were well known and mods<sup>2</sup> were released with altered graphics, sounds effects, music, and maps. However, the core of the game - the 3D engine and the secrets of its speed - remained mostly unknown.

It was kept secret for an obvious reason: a powerful engine is an essential asset for a gaming company. As a means to outperforming competitors, it's good business practice to keep other programmers clueless. This allows for maintaining a technological advantage, making better games, and generating more profit.

However, a few people within id Software did not see things that way. Instead of going along with what was common sense, they wanted to embrace players' enthusiasm and fully open the source code to the public. After much internal debate, id Software did the unthinkable: on July 21st, 1995 they uploaded a zip archive on [ftp.idsoftware.com](ftp://ftp.idsoftware.com) containing the full source code of the engine with instructions to build it<sup>3</sup>.

Programming is not a zero-sum game. Teaching something to a fellow programmer doesn't take it away from you. I'm happy to share what I can, because I'm in it for the love of programming.

**John Carmack - Programmer**

Opening the code did much more than enable programmers. It had two unforeseen consequences.

First, it allowed the software to live long after the target hardware and operating system disappeared. With access to the source, programmers were able to maintain and port the engine to new hardware and operating systems. Twenty years after the release of Wolfenstein 3D, you can still play the game on anything with a CPU, some RAM, and a framebuffer.

Second, it created a window back in time to 1991. Having reviewed complex engines such as Quake III and Doom III on [fabiensanglard.net](http://fabiensanglard.net), I thought I would have merely skimmed over the Wolfenstein 3D engine and its "simple" raycasting technology. When I took a deeper look out of curiosity, something struck me and I could not stop. The more I read, the more I came to realize how the target machine, the IBM PC, was designed for office work rather than gaming. It was meant to crunch integers and display static images for word processing and spreadsheet applications. What id Software<sup>4</sup> did in 1991 was not just

---

<sup>2</sup>MODified version.

<sup>3</sup>They were not totally crazy, they had built a new game engine which made Wolfenstein 3D obsolete: Doom was released on December 10, 1993.

<sup>4</sup>Other companies, such as Origins and LucasArts were also doing amazing things.

program a machine - they re-purposed a tool built to do office work and turned it into the best gaming platform in the world.

But why go through so much trouble? After all, if you were a game company and you wanted to make video games, you had video game consoles dedicated to this very specific task. The Genesis, the Super NES, and the Neo-Geo had sprite engines which despite limitations such as size and number allowed movement of something on the screen by simply updating its  $(x, y)$  coordinates. They were able to easily generate smooth animation at 60 frames per second, had controllers, had an audio system for sound and music, and were homogeneous (e.g. all SNES were the same). If you still really wanted to use a personal computer for a game, why not use an Amiga 500 which was packed with coprocessors designed for animation?

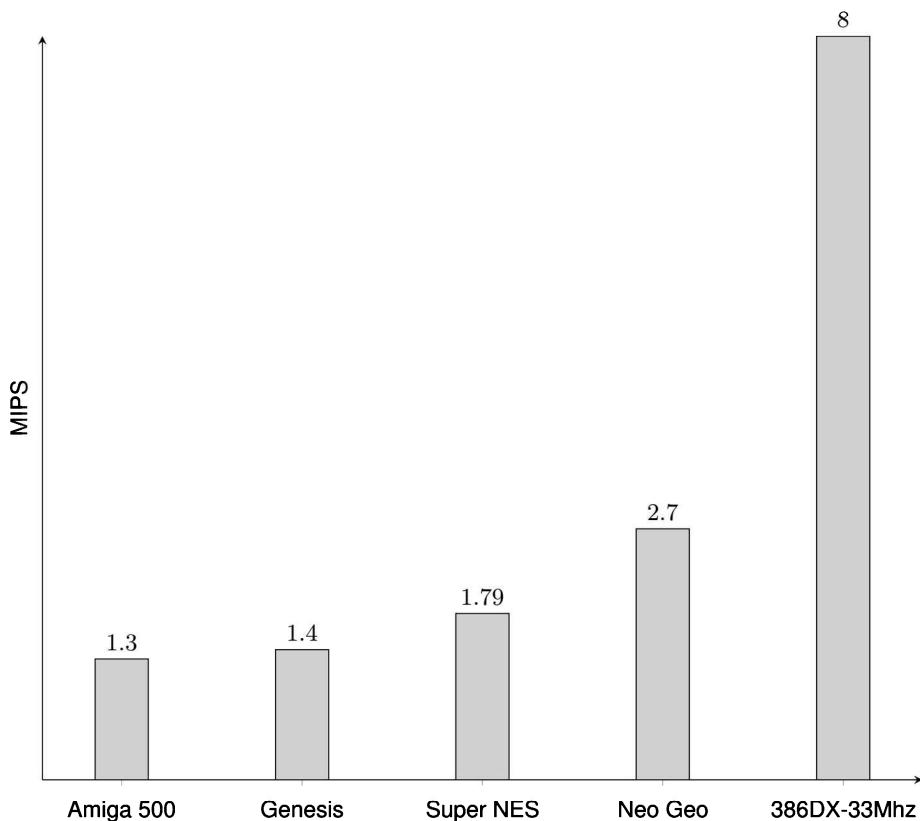
The reason fits in one word: framebuffer. The kind of game id Software wanted to create could not be done with a sprite engine or tricks from a Copper<sup>5</sup>. They wanted to shake the gaming world by providing an immersive experience in three dimensions. In order to do that they needed to draw a full screen, pixel by pixel, in a framebuffer before it was sent to the monitor.

To draw all these pixels they needed a powerful CPU, and a PC outperformed any console on the market. No Amiga<sup>6</sup>, even with its co-processor, could rival a PC in terms of raw power.

---

<sup>5</sup>Nickname of a powerful Amiga co-processor allowing operations at hsync level.

<sup>6</sup>Jimmy Maher advances an interesting theory in his book "The Future was here: The Commodore Amiga": People wanted to play First Person Shooters, which the Amiga architecture did not allow. This inability ultimately led to the downfall of Commodore's best seller.



**Figure 1.1:** Consoles Vs PC, CPU comparison with MIPS<sup>78</sup>.

With its fast CPU and 256KiB framebuffer, a 1991 PC looked promising at first. However, there were three seemingly impossible<sup>9</sup> obstacles to overcome:

- The video system (called VGA) could not double buffer. It was not possible to have smooth animations without ugly artifacts called "tears" on the screen.
- The CPU could only perform integer operations, but 3D calculations required keeping track of fractions.

<sup>7</sup>Million Instructions Per Second.

<sup>8</sup>The Amiga 500, Genesis, and Neo-Geo have a Motorola 68000 CPU respectively running at 7.16 MHz, 7.6 MHz, and 12 Mhz. The Super NES uses a WDC 65816 CPU which is a 8/16 bit version of a 6502 running at 3.58 MHz.

<sup>9</sup>The title of this book could have been "The Impossible Machine".

- The PC Speaker, the default sound device, could only produce square waves resulting in a bunch of "beeps" which were more annoying than anything else.

Beyond these major blockers were even more challenges:

- The RAM addressing mode was not flat but segmented, resulting in complex and error prone pointer arithmetics.
- VGA pixels were not square: the framebuffer was stretched vertically when transferred to the screen.
- The audio eco-system was fragmented. Each of the various sound systems had different capabilities and expectations.
- The machine could only address 1MB of RAM. To go beyond required entering a fragmented eco-system of drivers.
- The bus was slow and I/O with the VRAM<sup>10</sup> was a bottleneck. It was next to impossible to write a full framebuffer at 70 frames per second.

Overall, it looked like the machine was *doomed* to do boring things. But many around the world did not accept that and tinkered with the hardware to achieve unexpected results. How they did it is the *raison d'être* of this book. I've chosen to divide this book in three chapters:

- Chapter I: The Hardware. The five components of a PC from 1991.
- Chapter II: The Team<sup>11</sup>. The people pushing the edges.
- Chapter III: The Software. The Wolfenstein 3D game engine.

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

**Trivia :** The name "Wolfenstein 3D" was inspired by the 1981 Apple II title "Castle Wolfenstein" from Silas Warner. The Apple version was stealth oriented (Wolfenstein's "Profound Carnage" style was a clear departure from the original theme) but really stood out thanks to its unprecedeted use of digitized voices. Initially the team believed they would be unable to use the Wolfenstein name due to trademark issues. They came up with multiple possible titles only to discover that the original developer, Muse Software, had gone out of business several years earlier and let the trademark lapse, leaving them free to name their game Wolfenstein 3D.

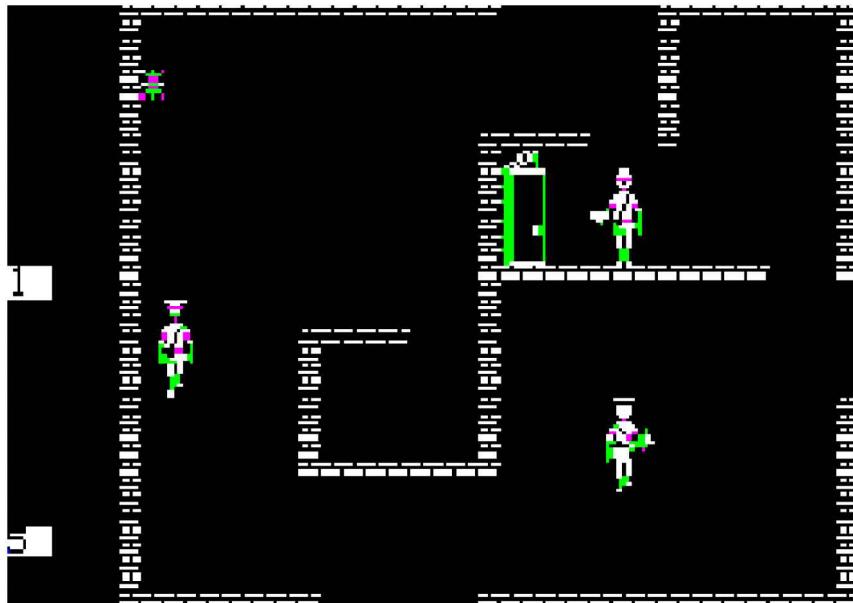
---

<sup>10</sup>VGA RAM

<sup>11</sup>This is an engineering book: for the human aspect read David Kushner's chef d'oeuvre: "Masters of Doom".



*Figure 1.2: "Castle Wolfenstein" intro.*

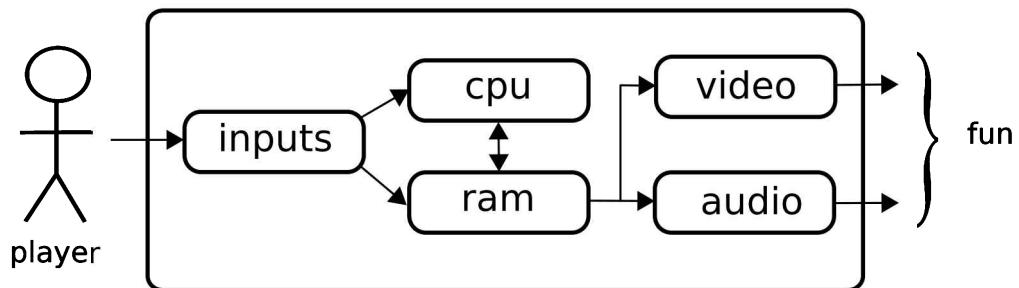


*Figure 1.3: "Castle Wolfenstein" gameplay.*

## Chapter 2

# Hardware

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



**Figure 2.1:** Hardware pipeline.

Overall, a lot of friction was present since manufacturers had not embraced the gaming industry yet. Some parts were bad, some were very bad, and some were downright impossible to deal with.

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

**Figure 2.2:** Component quality for a game engine.

## 2.1 CPU: Central Processing Unit

In 1991 there were 56 million PCs in the USA<sup>1</sup>. The performance of these machines was so overwhelmingly determined by the CPU that a PC was referred to not by its brand or GPU<sup>2</sup> but by the main chip inside. If a PC had an Intel 80386 or equivalent, it was called a "386". If it had an Intel 80286, it was a "286".

### 2.1.1 Overview

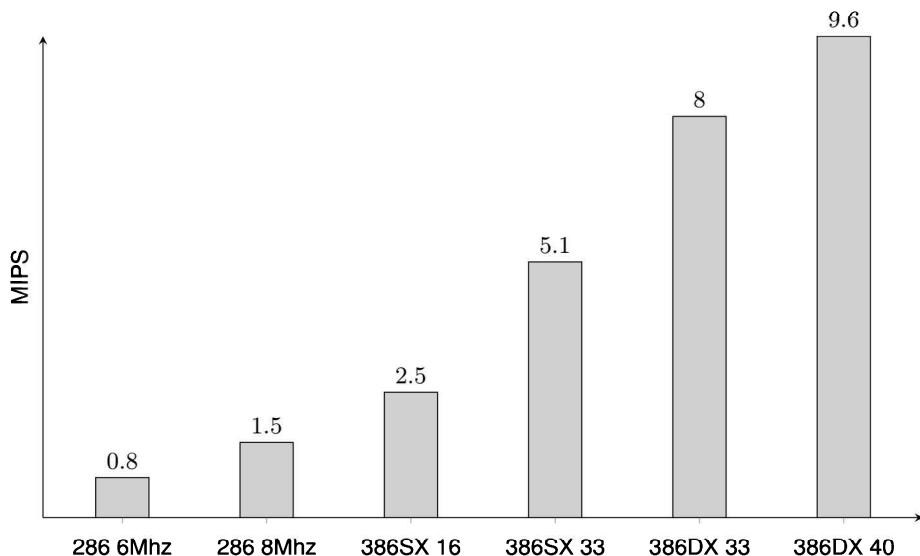
The predominant CPU manufacturer was Intel with its line of x86 microprocessors<sup>3</sup>. Machines based on the 16 bit 80286 processor did not sell well but their successors with the 80386 introduced in 1985 were immensely successful<sup>4</sup>. As a result, Wolfenstein 3D was designed to run on a PC with a 386 CPU, with degraded performance (yet still playable) on a 286.

<sup>1</sup>"Computers". Collier's Encyclopedia. Vol. 7, 1992: 114, 129.

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

<sup>3</sup>73% of PCs sold in 1989 featured an Intel CPU according to Intel Corporation's Annual Report.

<sup>4</sup>An estimated 3.74 million 386 based systems were sold in 1990 according to Dataquest in PC Magazine February 1992.



**Figure 2.3:** Comparison<sup>5</sup> of CPUs with MIPS

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

Intel built two versions of the 386: the 386-SX and the 386-DX. They were identical processors yet the DX version was almost twice as powerful as the SX (on the chart the 386 SX 33Mhz and the 386 DX 33Mhz are respectively at 5.1 and 8 MIPS). This is due to a data bus between the CPU and the RAM being twice as wide on the DX (32 bits vs 16 bits). Despite its inferiority, the SX sold well because it was cheaper and a lot of people had no idea what a "bus" was, they just wanted "a 386".

**Trivia :** Two other companies produced Intel clones: AMD and Cyrix. Their mediocre performance did not justify the lower cost and as a result they never gained significant market share. Almost all PCs featured an Intel CPU. Interestingly, AMD evolved to become a serious challenger<sup>6</sup> while Cyrix merged with National Semiconductor in 1997.

### 2.1.2 The Intel 80386

The trip from blueprint to silicon was not a pleasure cruise for the 80386. It started as a side project for a small team in San Jose while select employees in Portland worked on

<sup>5</sup>Roy Longbottom's PC Benchmark Collection: <http://www.roylongbottom.org.uk/mips.htm#anchorIntel2>.

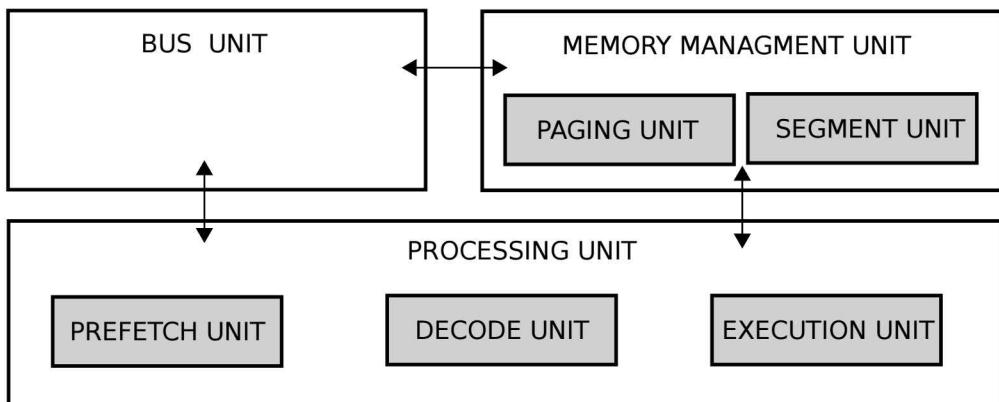
<sup>6</sup>The history of AMD vs Intel litigations could be the subject of its own book.

the flagship P7<sup>7</sup>, a CPU using a new instruction set capable of running high level language and memory garbage collection in hardware. When the Portland team hit a wall due to performance, the 386 went from step child to king<sup>8</sup>.

Two choices in the design of the 386 contributed to its success. First, the designers decided to listen to the programmers' feedback and dropped the idea of using a new instruction set. As a result the 386 is fully backward compatible with the 286. Second, they managed to add a 32 bit operating mode which solved many of the memory addressing issues of the 286.

**Trivia :** The 286 was quite unpopular among both programmers and hardware designers. Bill gates called it "brain dead"<sup>9</sup> for operating systems and Steve Morris (co-architect of the Intel 8086) called it "software poison".

The 80386 relies on three systems and a three stage instruction pipeline.

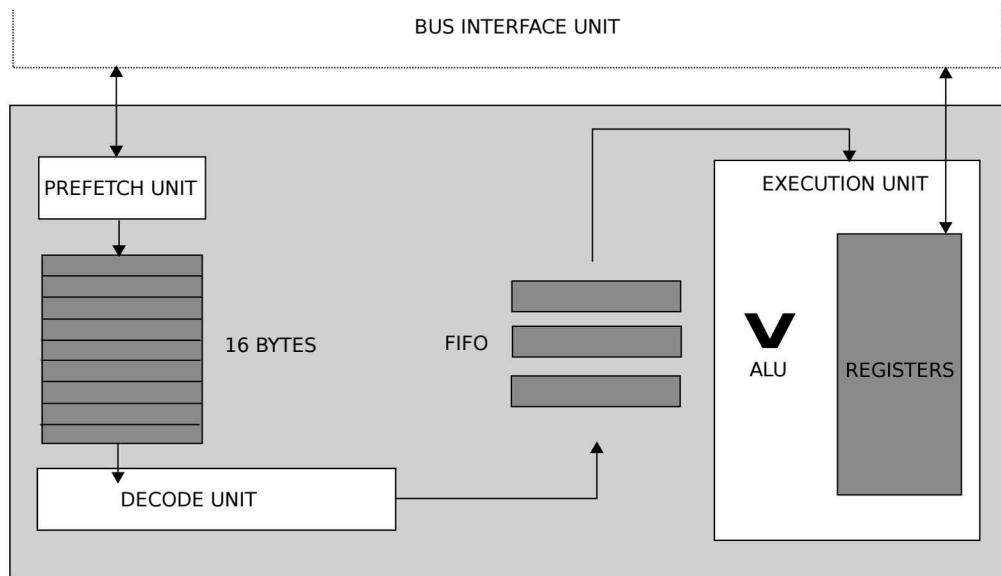


The Bus Unit is the only difference between a SX and a DX. The SX has a 16 bit bus which allowed PC manufacturers to reuse the design of the 286 motherboards and drove the price down significantly. The DX had a fully 32 bit Bus unit.

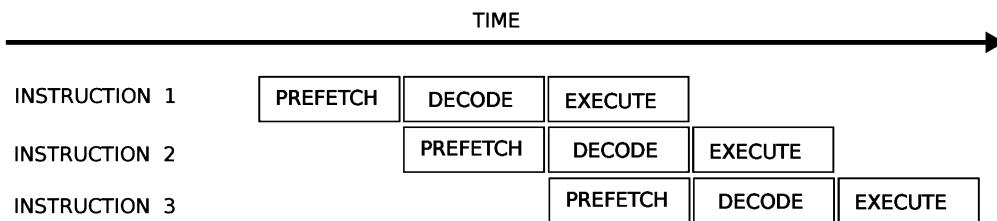
<sup>7</sup>a.k.a Intel i960.

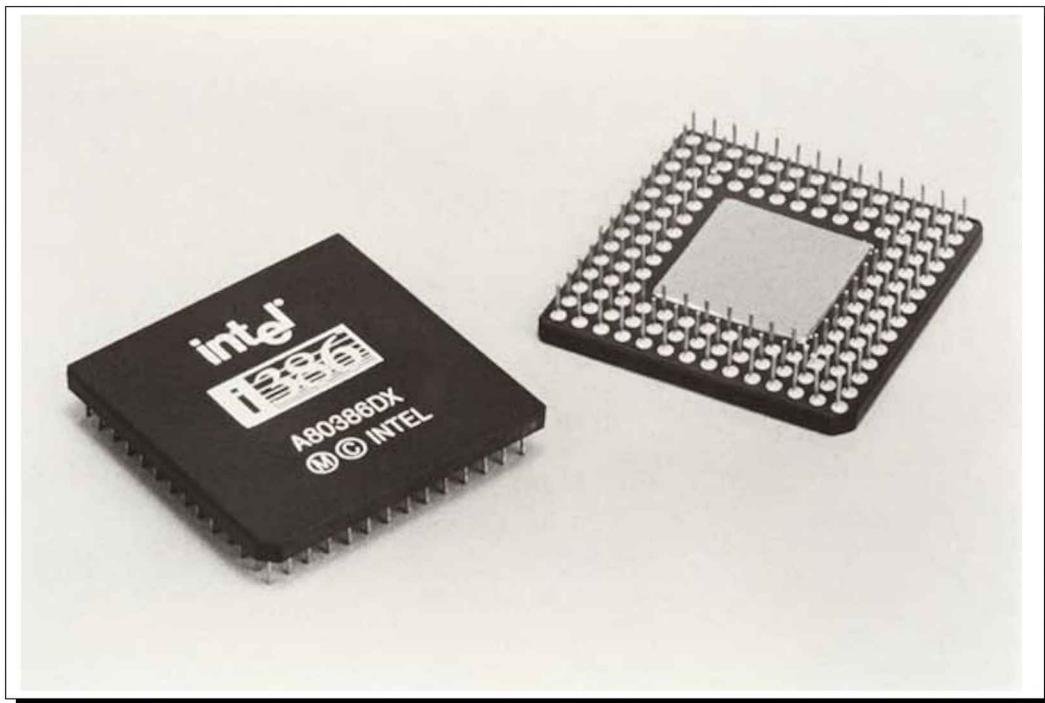
<sup>8</sup>Intel 386 Microprocessor Design and Development Oral History Panel.

<sup>9</sup>Dewar, Robert B. K.; Smosna, Matthew (1990). *Microprocessors: A Programmer's View*.



The three units in the execution group form a three stage pipeline: Prefetch, Decode, and Execute. The Prefetch Unit wakes up when the Execution unit is performing but not using the bus and fetches instructions in a 16 byte queue. The prefetcher is linear and cannot predict the result of a branch. As a result, a jump (JMP) instruction triggers a flush of the entire pipeline. Instructions go down the pipeline and are decoded by the Decode Unit: the result of the decode operation is stored in a three-element FIFO where it is picked up by the Execution Unit.





**Figure 2.4:** The Intel 386, 10mm by 10mm packing 275,000 transistors

From a programming perspective, a 386 CPU can be summarized by the following elements:

- Arithmetic Logic Unit performing add, sub, mul et cetera.
- 16 registers:
  - 32 bit General Purpose Registers: EAX, EBX, ECX, EDX
  - 32 bit Index Registers: ESI, EDI, EBP, ESP
  - 16 bit Segment Registers: CS, DS, ES, FS, GS, SS
  - 16 bit Status Register
  - 32 bit Program Counter: EIP
- A 32 bit address bus for up to 4GB of flat addressable RAM
- Memory Paging Unit

**Trivia :** Despite its pipeline design, the 386 cannot do an operation in less than two cycles. Even a simple ADD reg, reg or INC reg takes two clocks. This is due to the absence of

a SRAM on-chip cache and a slow decoding unit.

Instruction type	Clocks
ADD reg16, reg16	2
INC reg16	2
IMUL reg16, reg16	12-25 <sup>10</sup>
IDIV reg16, reg16	27
MOV [reg16], reg16	4
OUT [reg16], reg16	25
IN [reg16], reg16	26

**Figure 2.5:** 386 instruction costs<sup>11</sup>

### 2.1.3 Floating Point

All that CPU power was not necessarily useful for programming a game. In order to perform trigonometric computations for 3D effects, the engine has to keep track of the fractional part of each operation. This may not appear to be an issue since the C programming language has a type (`float`) precisely for that purpose. But in practice this was a problem, and to understand it we need to understand how `float` works.

As David Goldberg famously wrote, "*Floating-point arithmetic is considered an esoteric subject by many people*"<sup>12</sup>. I could not agree more. Yet it is important to understand in order to fully grasp how useful it is to for programming a 3D engine. In the C language, floats are 32 bit container following the IEEE 754 standard. Their purpose is to store and allow operations on approximation of real numbers. The 32 bits are divided in three sections:

- 1 bit S for the sign
- 8 bits E for the exponent
- 23 bits M for the mantissa

---

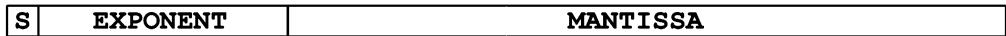
<sup>10</sup>Not all multiplications are equal. The 80386 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the optimizing multiplier.

<sup>11</sup>Intel 80386 programmer's reference manual - 1986.

<sup>12</sup>"What every Computer Scientist should know about Floating-Point" by David Goldberg.



**Figure 2.6:** Floating Point internals



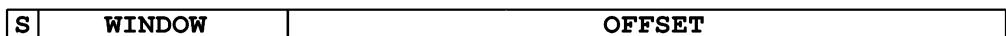
**Figure 2.7:** The three sections of a Floating Point number.

How numbers are stored and interpreted is usually explained with the formula:

$$(-1)^S * 1.M * 2^{(E-127)}$$

**Figure 2.8:** How everybody hates floating point to be explained to them.

Although correct, this way of explaining floating point usually leaves programmers completely clueless. I blame this dreadful notation for discouraging legions of programmers, scaring them to the point where they never looked back to understand how floating point actually works. Fortunately, there is a better way to explain it. Instead of an exponent, think of a window between two consecutive power of two integers. Instead of a mantissa, think of an offset within that window.



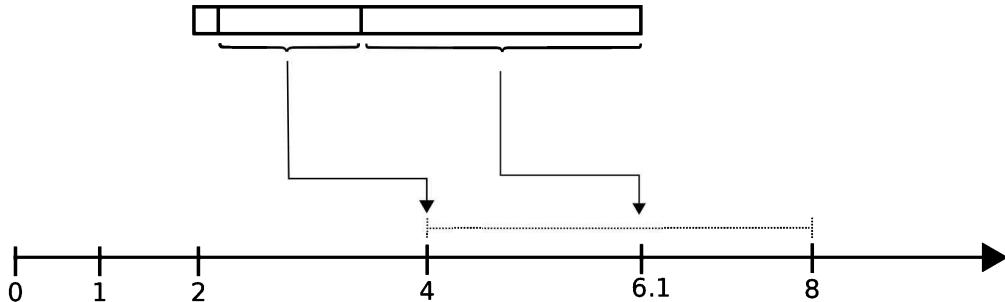
**Figure 2.9:** Alternate Floating Point internals.

The window tells within which two consecutive power-of-two the number will be: [0,1], [1,2], [2,4], [4,8] and so on (up to  $[2^{127}, 2^{128}]$ ). The offset divides the window in  $2^{23} = 8388608$  buckets. With the window and the offset you can approximate a number. The window is an excellent mechanism to protect from overflowing. Once you have reached the maximum in a window (e.g. [2,4]), you can "float" it right and represent the number within the next window (e.g. [4,8]). This only costs a little bit of precision.

**Trivia :** How much precision is lost when the window covers a wider range? Let's take an example with window [0,1] where the 8388608 offsets cover a range of 1 which gives a precision of  $(1 - 0)/8388608 = 0.00000011920929$ . In the window [2048,4096] the

8388608 offsets cover a range of  $(4096 - 2048) = 2048$  which gives a precision  $(4096 - 2048)/8388608 = 0.0002$ .

The next figure illustrates how the number 6.1 would be encoded. The window must start at 4 and span to the next power of two, 8. The offset is about half way down the window.



**Figure 2.10:** Value 6.1 approximated with floating point

Here is a detailed example that calculates the floating point representation of a number we know well: 3.14.

- The number 3.14 is positive  $\rightarrow S = 0$ .
- The number 3.14 is between the power of two 2 and 4 so the floating window must start at  $2^1 \rightarrow E = 128$  (see formula where window is  $2^{(E-127)}$ ).
- Finally there are  $2^{23}$  offsets available to express where 3.14 falls within the interval [2-4]. It is at  $\frac{3.14-2}{4-2} = 0.57$  within the interval which makes the offset  $M = 2^{23} * 0.57 = 4781507$

Which in binary translates to:

- $S = 0 = 0b$
- $E = 128 = 10000000b$
- $M = 4781507 = 1001000111010111000011b$

31	30	23	22	0
0	1	0	0	0
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
1	1	1	1	0
0	1	0	1	1
1	1	0	0	0
0	1	1	1	1

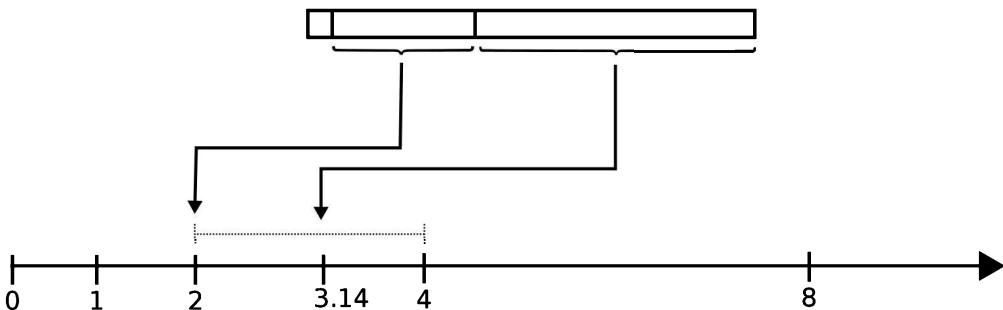
**Figure 2.11:** 3.14 floating point binary representation.

The value 3.14 is therefore approximated to 3.1400001049041748046875.

The corresponding value with the ugly formula:

$$(-1)^0 * 1.57 * 2^{(128-127)} = 3.14$$

And finally the graphic representation with window and offset:



**Figure 2.12:** 3.14 window and offset.

Floating point arithmetic is a powerful tool. It can represent very small or huge values while keeping track of fractional parts of a number, and also protecting from overflow by floating the window when necessary.

Floating point is handy, but the drawback is that it is also computationally expensive. The reason is simple. In order to add, subtract, multiply or divide two numbers, they both have to be expressed with the same window. This means converting one number to the representation used by the other, usually with higher precision than 32 bits (typically 80 bits on Intel FPU)<sup>13</sup>.

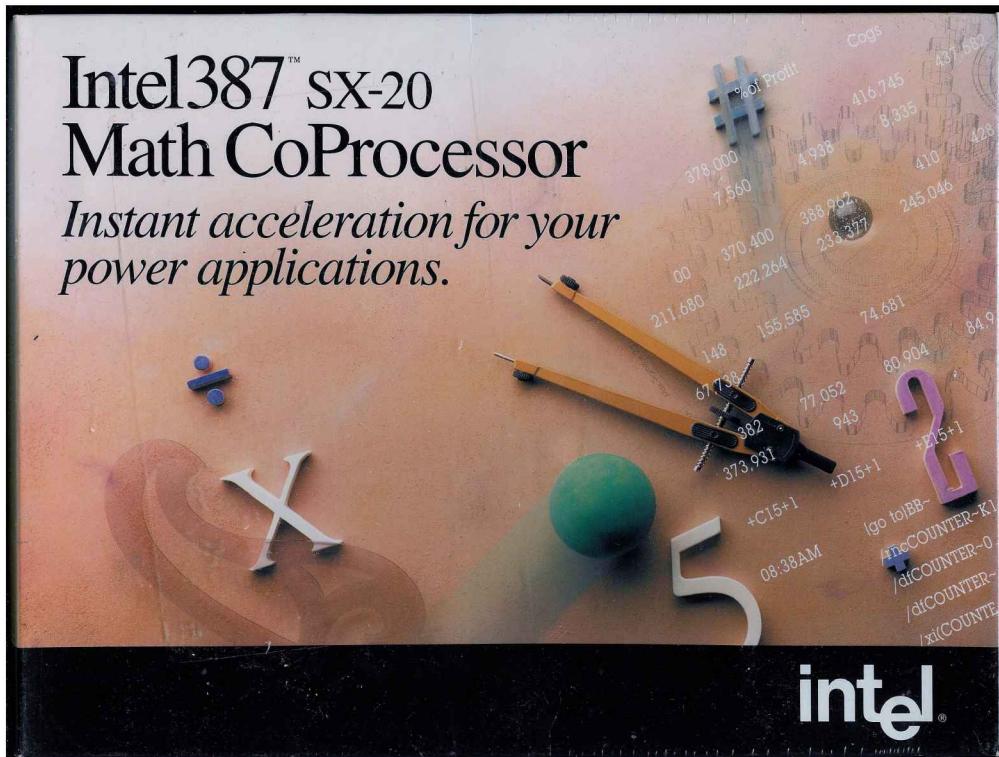
This is not a problem when everything is hardwired within a hardware floating point unit but it is a big problem for the 386. If you refer back to the architecture diagram you will notice that it only has an ALU. A 386 *doesn't have a hardware Floating Point Unit*. If float operations are found in the code, they are emulated in software by the compiler, resulting in terribly slow processing ; so slow they are not usable for anything real-time.

**Trivia :** Since floating point operations were so slow, why did the C language end up with `float` and `double` types ? After all, the machine used to invent the language (PDP-11) did not have a floating point unit! The manufacturer (DEC) had promised to Dennis Ritchie

<sup>13</sup>To fully grasp how much processing a FPU does, it helps to read a software implementation. I found Berkeley SoftFloat helpful.

and Ken Thompson the next model would have one<sup>14</sup>. Being astronomy enthusiasts, they decided to add those two types to their language.

**Trivia :** People who really wanted a hardware floating point unit could buy one. In the 90s the only people who could possibly want one would have been scientists (as per Intel's understanding of the market). The i387 chip was marketed as a "Math CoProcessor". Performance was average and price was outrageous<sup>15</sup>. As a result, sales were mediocre.



**Figure 2.13:** Intel 1991 ad for a "Math CoProcessor" i387.

A possible solution to the lack of a hardware floating point unit would have been to multiply an integer by 100 or 1000 to perform fractional operation and then divide to go back to integers. That is unfortunately not possible. On a 386, a multiply (`imul`) instruction takes 12 to 25 cycles and a divide (`div`) is even worse with 27 cycles<sup>16</sup>.

<sup>14</sup>The Development of the C Language by Dennis M. Ritchie.

<sup>15</sup>\$200 in 1993 equivalent to \$350 in 2016.

<sup>16</sup>ZSmith.com instructions at [http://zsmith.co/intel\\_i.html#imul](http://zsmith.co/intel_i.html#imul).

As a result, a game engine designer was stuck in an awkward situation with two half solutions to his problem. Integers which were fast but not accurate enough and Floats which were accurate but not fast enough.

## 2.2 RAM

The first CPUs in the Intel x86 family were designed in 1976. At a time when RAM was very expensive, the 8080 and 8086 had 16 bit registers with a 20 bit wide address bus capable of addressing 1MiB<sup>17</sup> of RAM. It is difficult to stress how big 1MiB of RAM was in the 70s but as an example the Apple II and the Commodore 64 both shipped with 64KiB<sup>18</sup> which was enough to write and run amazing things. Sixteen bit registers and 20 bit address bus were plenty even though programming was difficult and required combining two registers to build a pointer.

By 1986, hardware had gotten cheaper and Intel made a departure from the old architecture with its 286 and 386. These new CPUs could be put in what is called "protected mode" featuring a 24 bit wide address bus for up to 16 MiB of flat RAM protectable with a MMU<sup>19</sup>. The 386 also had 32 bit registers in protected mode. To make sure old programs could still run, both processors could be put in "real mode" which replicates how the Intel 8080 and 8086 operated: 16 bit registers, 20 bit address bus giving 1MiB addressable RAM with segmented addressing.

For compatibility reasons all PCs have to start in real mode. You may assume that programmers of the 90s promptly switched the CPU to protected mode to unleash the full potential of the machines and ditch the 20-year-old real mode. Unfortunately, there was a major obstacle: the operating system MS-DOS by Microsoft Corporation.

### 2.2.1 DOS Limitations

Microsoft Corporation highly valued the applications running on their operating systems. As a business priority, they were adamant to never break anything with a new system<sup>20</sup>. Since many applications were written during the 80s on machines having only real mode, DOS 5.0<sup>21</sup> kept running that way and as a result its routines and system calls were incompatible with protected mode. This created an awkward situation where the de-facto operating system delivered with every machine sold prevented programmers from using

---

<sup>17</sup>This book uses IEC notation where MiB is  $2^{20}$  and MB is  $10^6$ .

<sup>18</sup>This book uses IEC notation where KiB is  $2^{10}$  and KB is  $10^3$ .

<sup>19</sup>Memory Management Unit

<sup>20</sup>"Tales of Application Compatibility", Old New Thing by Raymond Chen.

<sup>21</sup>Released in June 1991.

the machine at its full potential. Developers were forced to ignore all the features of a 1992 CPU and instead use it like a very fast Intel 8086 CPU from 1976. They were thus limited to the following characteristics:

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

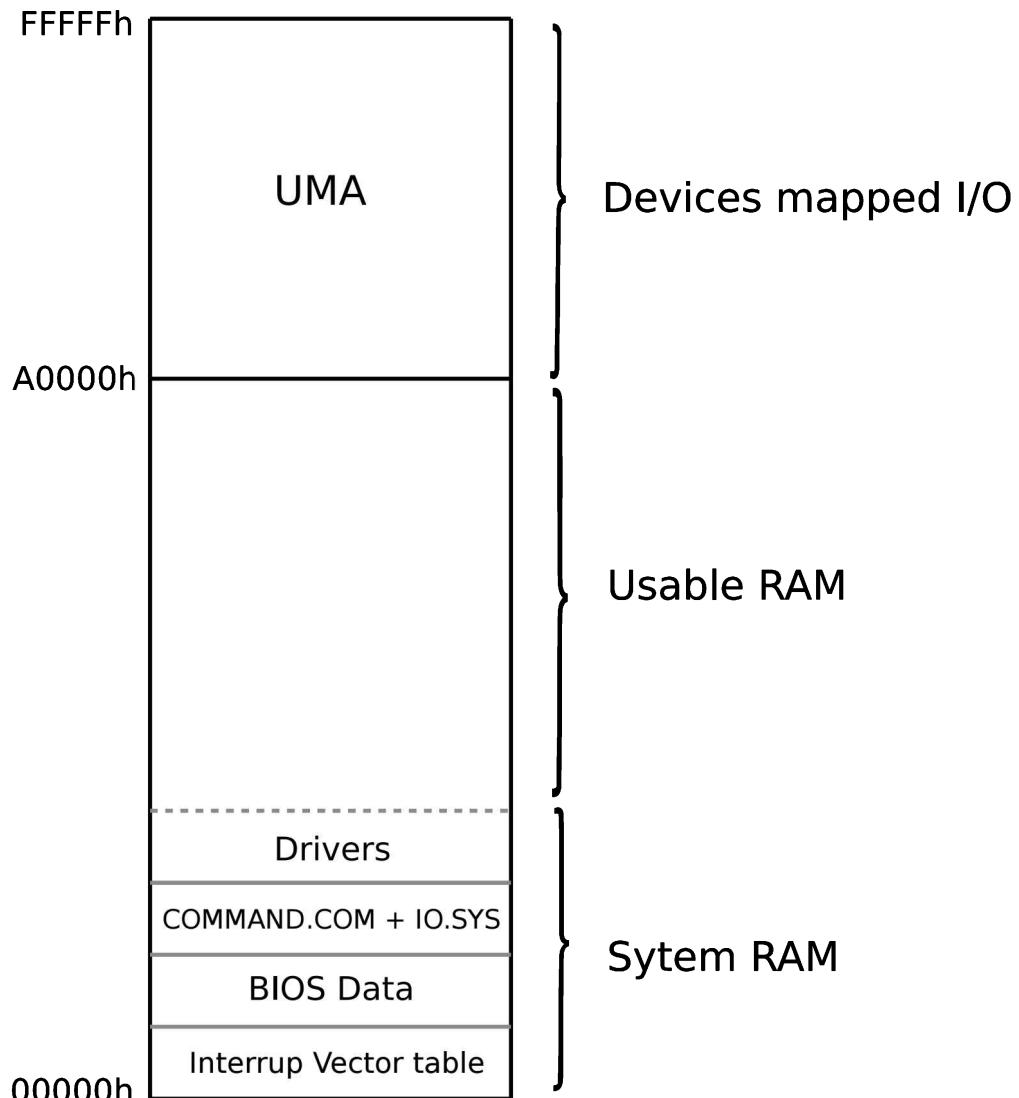
**Trivia :** One year earlier, in 1991, a student from the University of Helsinki started working on a hobby ("nothing big") of his: an operating system which contrary to DOS was able to use the CPU in protected mode and take advantage of the MMU and the 32 bits registers. It would become Microsoft's worst nightmare. Linus Torvalds had just started what would become Linux.

### 2.2.2 The Infamous Real Mode: 1MiB RAM limit

With protected mode unavailable, 1991 developers programmed like it was 1976: with a 20 bit wide address bus offering only 1MiB of addressable RAM. Regardless how much memory was installed on the machine, only 1MiB could be addressed. To top it all off, addressing had to be done not with the 32 bit registers available but by combining two 16 bit registers. One was the segment, the other an offset within that segment. Hence the name: '16 bit segmented programming'.

The memory layout is as follow:

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



**Figure 2.14:** First 1MiB of RAM layout.

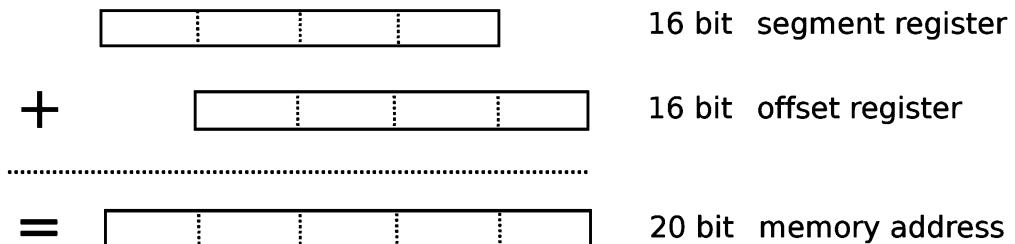
Out of the original 1024KiB, only 640KiB (called Conventional Memory) was accessible to a program. 384KiB was reserved for the UMA and every single driver installed (.SYS and .COM) took away from the remaining 640KiB.

**Trivia :** In France people had to load KEYBFR.SYS driver so AZERTY keyboard keys would be properly mapped. The driver consumed a whopping 5KiB of Conventional Mem-

ory. Needless to say French people learned pretty quickly that god mode was IDDAD<sup>22</sup>.

### 2.2.3 The Infamous Real Mode: 16 bit Segmented addressing

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



**Figure 2.15:** How registers are combined to address memory.

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

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

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

Pointer A defined as:

---

<sup>22</sup>Invincibility mode in Doom is IDDQD on a qwerty keyboard but IDDAD on an azerty keyboard without the French driver loaded.

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

Pointer B defined as:

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

Pointer C defined as:

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

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

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

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

    far void* a = FP_SEG(0x0000) + FP_OFF(0x0120);
    far void* b = FP_SEG(0x0010) + FP_OFF(0x0020);
    far void* c = FP_SEG(0x0012) + FP_OFF(0x0000);

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

Will output:

0
0
0

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

### 2.2.4 Extended Memory

The 20 bit address bus of real mode limits the addressable RAM to 1MiB. Machines of 1992 came equipped with more, typically 2MiB and even sometimes 4MiB for the most fortunate customers. This memory located beyond the addressable space is called "Extended Memory". The workaround at the time to access these resources was to install specialized drivers<sup>23</sup>.

Unfortunately extended memory access was not standardized. Users could load either of the drivers provided with DOS:

- Expanded Memory Specification (EMS) drivers: EMM368.EXE.
- eXtended Memory Specification (XMS) drivers: HIMEM.SYS.

Or they could have no idea they had to install a driver, load nothing at startup, and not use any of the RAM beyond 1MiB. This use case was a big issue. Many customers could not understand why despite installing (extremely expensive<sup>24</sup>) megabytes of RAM on their machine, the game they just purchased would refuse to start up claiming "Not enough memory". id Software had to publish an explanation (see Appendix "The 640KB Barrier") along with the game.

The two APIs had similar features but completely different architecture.

#### 2.2.4.1 XMS API

The XMS driver works<sup>25</sup> like `malloc/free` from `libc` and is the most intuitive to a programmer. It allows manipulation of data in the non-addressable extended memory via operations such as `allocate`, `free`, `realloc`, and `move`.

The key aspect of XMS is that memory has to be copied between extended RAM and conventional RAM.

---

<sup>23</sup>See CONFIG.SYS in the Appendices.

<sup>24</sup>In 1992, 4MiB of RAM cost \$149 which adjusted to inflation would be \$256 in 2017.

<sup>25</sup>eXtended Memory Specification (XMS) July 19, 1988.

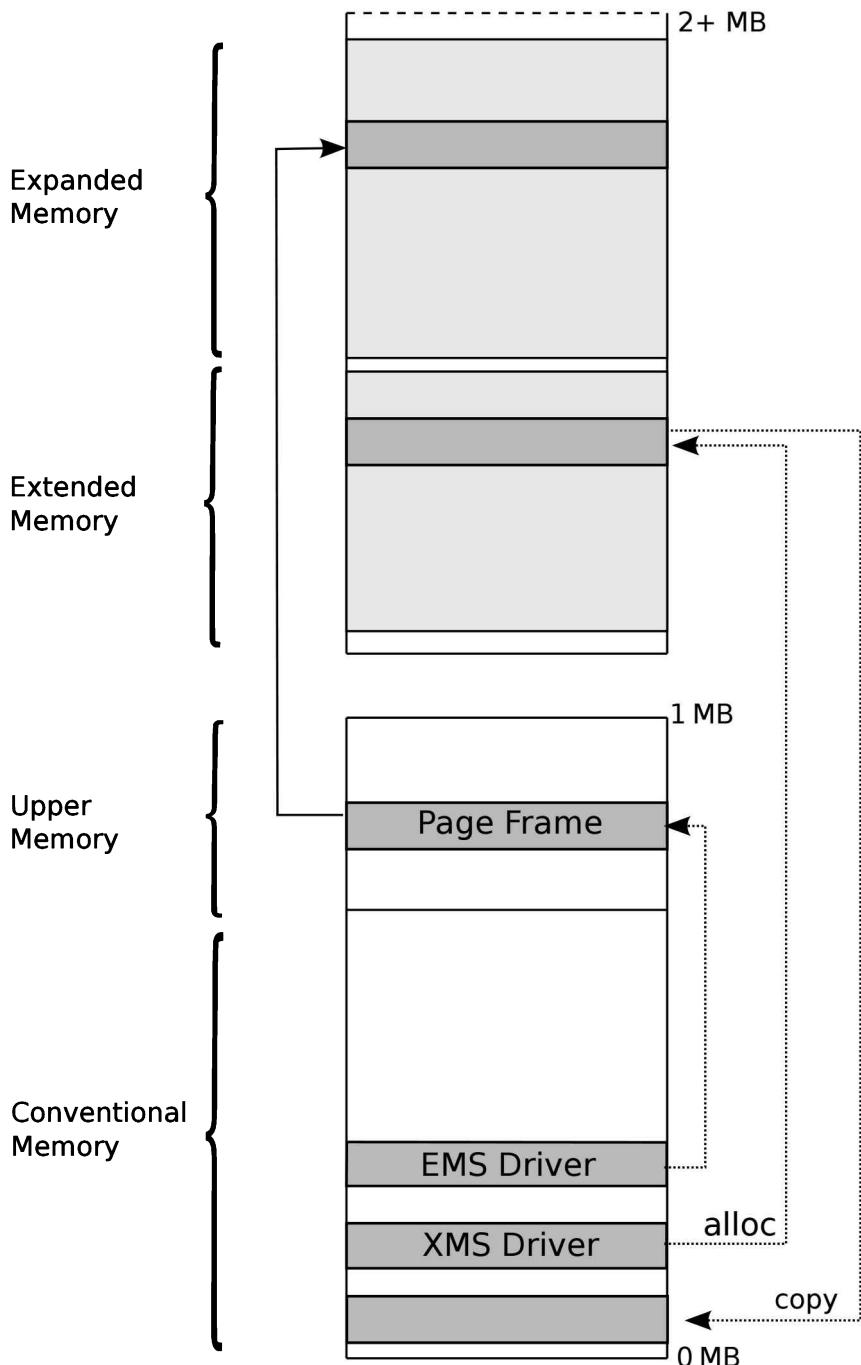


Figure 2.16: Expanded/Extended memory layout

### 2.2.4.2 EMS API

The EMS driver opens a window beyond the addressable RAM, as shown on page 36. The idea revolves around memory mapping. The driver allows for manipulating four units of 16KiB called "pages" via a 64KiB area called "Page Frame". Upon request to the driver, a page can be swapped into the page frame.

### 2.2.4.3 EMS vs XMS

How the drivers achieved the impossible (after all how do you access RAM past 1 MiB with 20 bits pointer?) and what were the trade-off of each approach is a fascinating topic explained in detail in Annexe B on page 293.

If you prefer not to dive into too much details, just remember that EMS mapping approach was several time faster than XMS copying approach. This speed consideration considerably impacted the memory management of Wolfenstein 3D.

### 2.2.4.4 A system "impossible to love"

At this point, if you are puzzled by the CPU and its design you are not alone. Over the years I came across many ways to describe this madness but three particularly stand out.

The x86 is an architecture that is difficult to explain and impossible to love.

**David Patterson & John Hennessy - Computer Organization and Design.**

That sounds odd, but Intel built it, Microsoft wrote it, and DOS grew up around it.

**Eccles-Jordan Trigger - Codeproject.com.**

Software poison.

**Steve Morris - Co-Architect of the Intel 8086.**

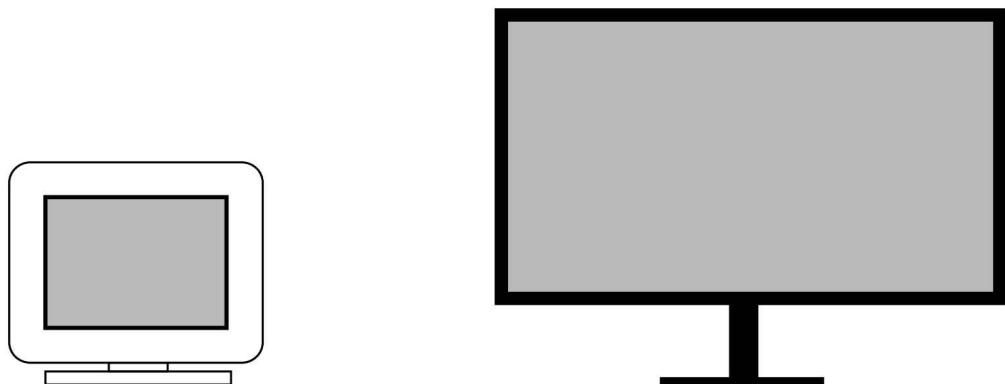
**Trivia :** 640KiB was all a game could have for executable code. But people writing compilers got clever. Strike Commander (a famous flight arcade game released in 1993) exe-

cutable is 745KiB, which obviously doesn't fit in Conventional Memory. The trick is to use a technique of "overlay" pages where only a few overlays are loaded when the game starts. When the CPU is about to reach the end of an overlay, special instructions (inserted by the compiler) load the next overlays from HDD and setup a `jmp` instruction for the CPU to follow. The technique can be seen as an exercise of graph paging similar to how Gromit places rails in front of an ongoing miniature train as he is sitting on top of it in "The Wrong Trousers".

**Trivia :** As of 2016, more than thirty five years after the introduction of the 8086, in the name of backward compatibility, all PCs in the world still start in real mode. A bootloader switches them to protected mode, loads the kernel, and then actual startup can begin.

## 2.3 Video

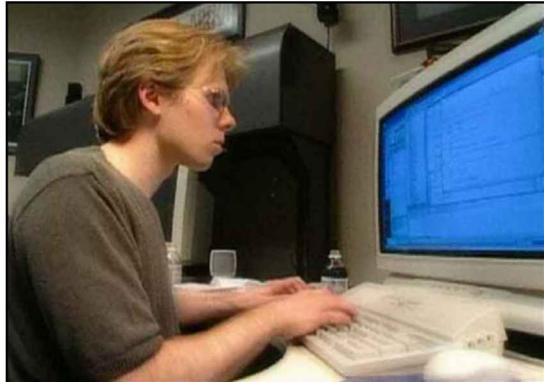
PCs were connected to CRT monitors: big, heavy, small diagonal, cathode-ray based, curved-surface screens. Most had a 14" diagonal with a 4:3 aspect ratio. To give you an idea of the size, Figure 2.17 shows a comparison between a 14" CRT from 1992 and a 30" LCD display from 2014.



**Figure 2.17:** CRT (left) vs LCD (right)

**Trivia :** How big and heavy could a CRT be? The InterView 28hd96 by Integraph had a 28" diagonal allowing a resolution of 1920x1080 at a time where most monitors displayed 640x480. It weighed 45kg (99.5lb). For comparison, a modern DELL LCD 27" weights 7.8kg (17lb).

In the photo on the right : John Carmack in 1996 working on an 28hd96 while programming Quake 2 using Visual Studio C++.



The main issue in this system was that CRTs were analog systems while computers were digital. An interface was needed between the two and it came as a series of chipsets called "Adapters".

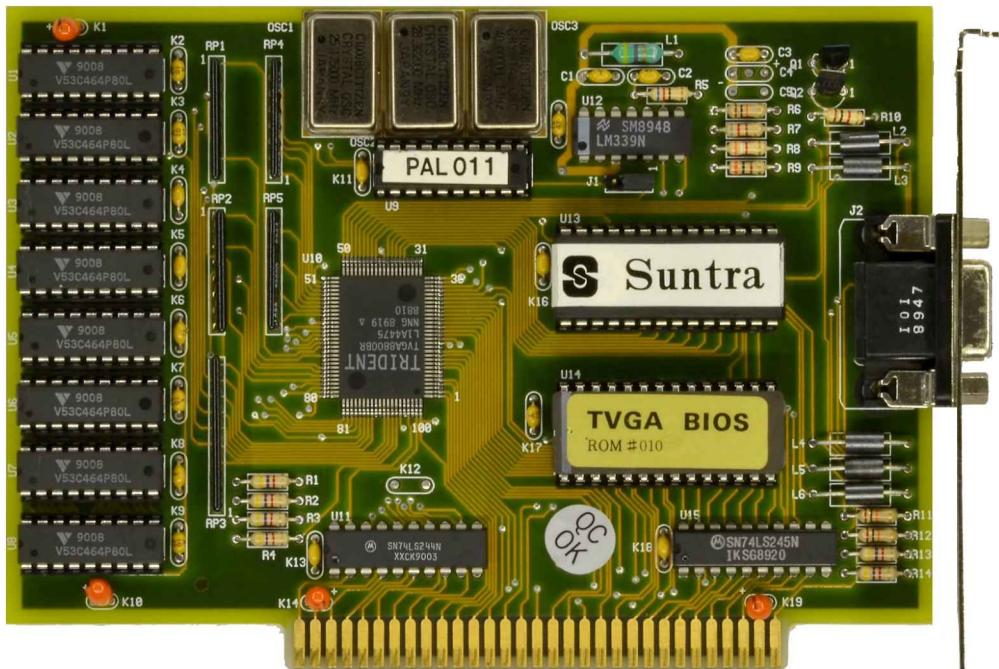
### 2.3.1 History of Video Adapters

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

Name	Year Released
MDA (Monochrome Display Adapter)	1981
CGA (Color Graphics Adapter)	1981
EGA (Enhanced Graphics Adapter)	1985
VGA (Video Graphics Array)	1987

*Figure 2.18:* Video interface history.

Each iteration added new features and by 1991 the predominant graphic system was VGA. All video cards installed on PCs had to follow the standard set by IBM. The universality of that system was a double-edged sword. While developers had to program for only one graphic system, there was no escaping its shortcomings.



**Figure 2.19:** The VGA card Trident 8800 (8 bit ISA). Courtesy of [vgamuseum.info](http://vgamuseum.info). Notice the eight chips on the left of the card forming the VRAM where the framebuffers are stored<sup>26</sup>.

<sup>26</sup>This Trident TVGA8800BR is actually more than VGA capable since its eight V53C464P80L chips can store 64KiB each, accounting for a total of 512KiB VRAM. Cards featuring more than 256KiB were called Super-VGA but this is another story altogether.



**Figure 2.20:** The VGA card Diamond Stealth (16 bit ISA). Courtesy of [vgamuseum.info](http://vgamuseum.info).

**Trivia :** There was no GPU market back then. Since all video cards "only" had to be VGA compatible with 256KiB of RAM, many just bought the cheapest thing available. Note however that some cards had an 8 bit ISA bus connector (like the Trident) and some had a 16 bit ISA connector like the Diamond, making the Diamond transfer data twice as fast.

### 2.3.2 VGA Architecture

VGA can be summarized as three major systems with input, storage, and output systems:

- The Graphic Controller and Sequence Controller controlling how VGA RAM is accessed (the CPU-VRAM interface)
- The framebuffer (the VRAM) made of four memory banks of 64KB (rather than one bank of 256KiB)
- The CRTC Controller and the DAC<sup>27</sup> taking care of converting the palette indexed framebuffer to RGB and then to analog signal (interface VRAM-CRT)

<sup>27</sup>Digital to Analog Converter.

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

Part of the explanation comes from backward compatibility. EGA, the predecessor of VGA had only 64KiB of RAM. It was very easy to design a backward compatible system that used only one bank of 64KiB.

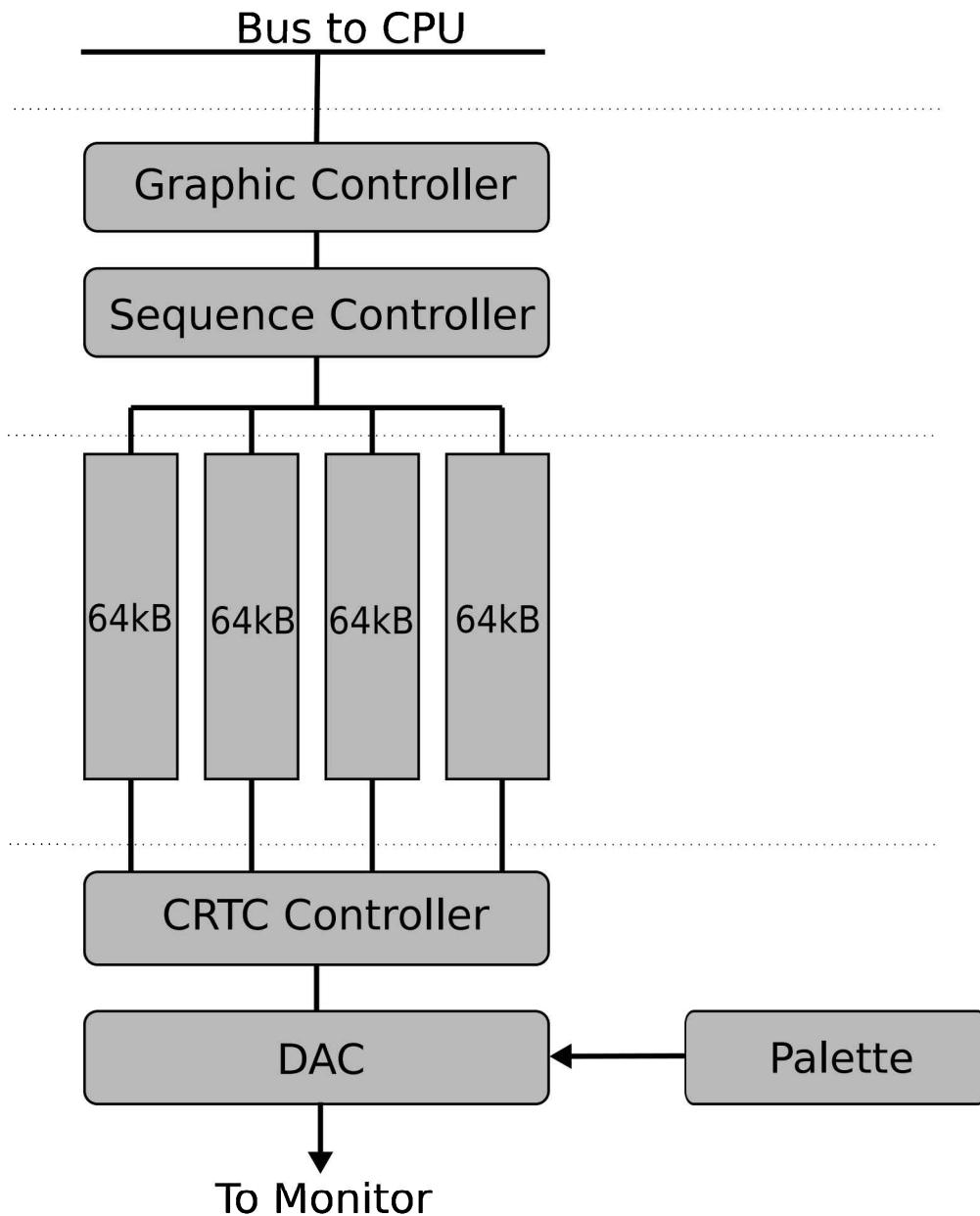
The more significant reason was RAM latency and the need for minimum bandwidth. A CRT running at 70Hz and displaying 640x480 in 16 colors needs a pixel every  $\frac{1}{640 \times 480 \times 70}$ th of a second. At this resolution, one pixel is encoded with 4 bits. Each nibble is translated to a 666 RGB color by the DAC. That encoding divides bandwidth by 4.5, but still requires one byte every 93 nano-seconds.

Unfortunately, RAM access latency was 200ns - not nearly fast enough<sup>28</sup> to refresh the screen at 70hz, so the DAC would starve. If latency could not be reduced, the throughput could still be improved by reading from four banks at a time. Reading in parallel gave an amortized RAM latency of  $200/4 = 50$ ns, which was fast enough.

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

---

<sup>28</sup>Computer Graphic: Principles and Practice 2nd Edition, page 168.



*Figure 2.21:* VGA Architecture.

### 2.3.3 VGA Planar Madness

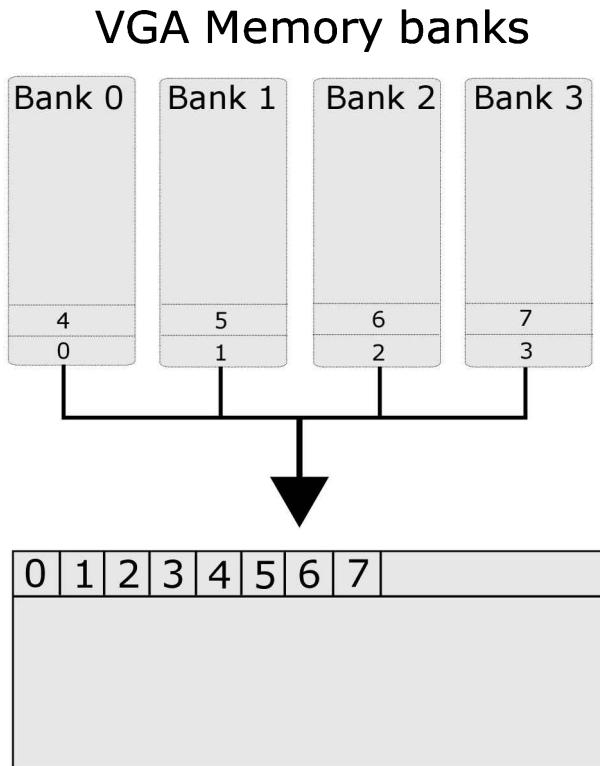
Four memory banks grant enough throughput to reach high resolutions at 70Hz. The price is complexity of programming, as acknowledged by even the best programmers of the time.

Right off the bat, I'd like to make one thing perfectly clear: The VGA is hard-sometimes very hard-to program for good performance.

#### **Michael Abrash - Graphic Programming Black Book**

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

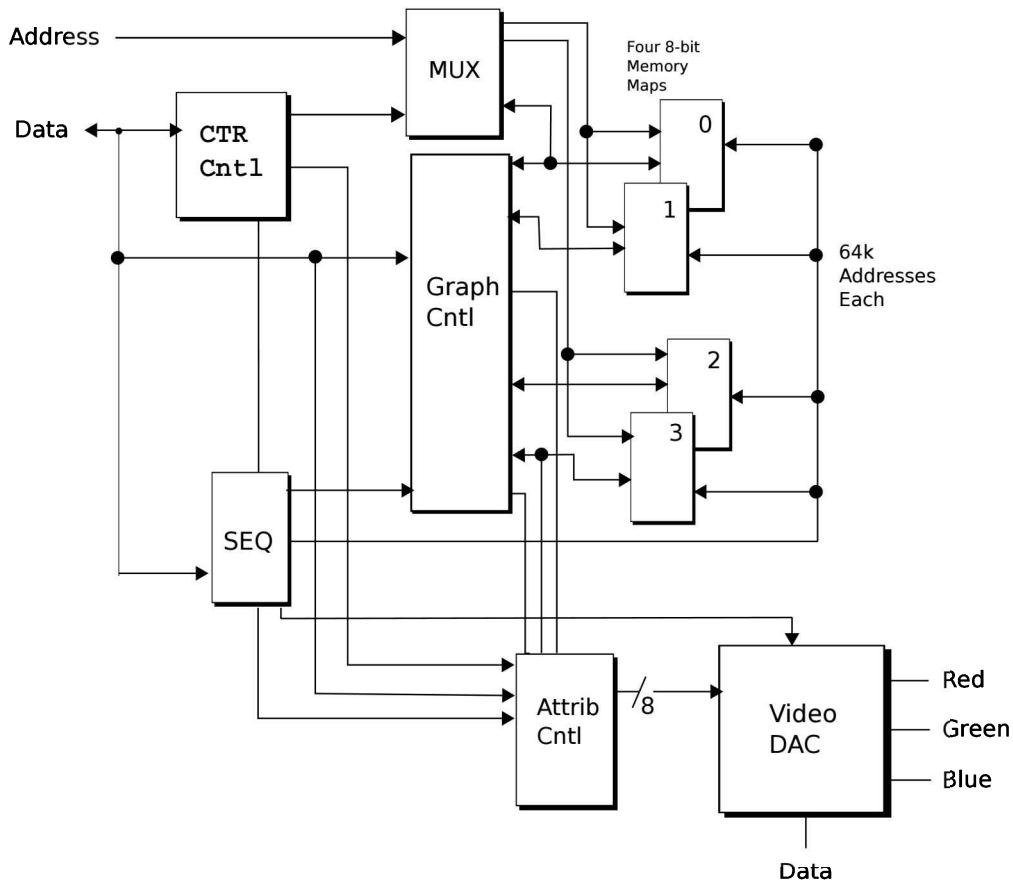
This type of architecture is called "planar". In mode 13h, where one pixel is encoded on one byte, to write four pixels next to each other on a line on the screen requires writing one byte in each bank. Each of these banks is mapped to the same UMA memory address. This layout is better explained with a drawing.



**Result on screen**

*Figure 2.22:* VGA mode 13h, How bank layout appears on screen.

In order to configure this mess of planes and the controllers, 300 poorly documented internal registers must be set. Needless to say few programmers dove into the internals of the VGA. The first figure describing its architecture is actually deceptively simplified. IBM's VGA reference documentation had its own drawing, which showcases the complexity of the system.



**Figure 2.23:** IBM's VGA Documentation.

To compensate for the complexity, IBM provided a routine to initialize all the registers via one simple BIOS call. One configuration can be selected out of 15 available (a configuration is called a "mode") with an associated resolution, number of colors, and memory layout.

### 2.3.4 VGA Modes

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

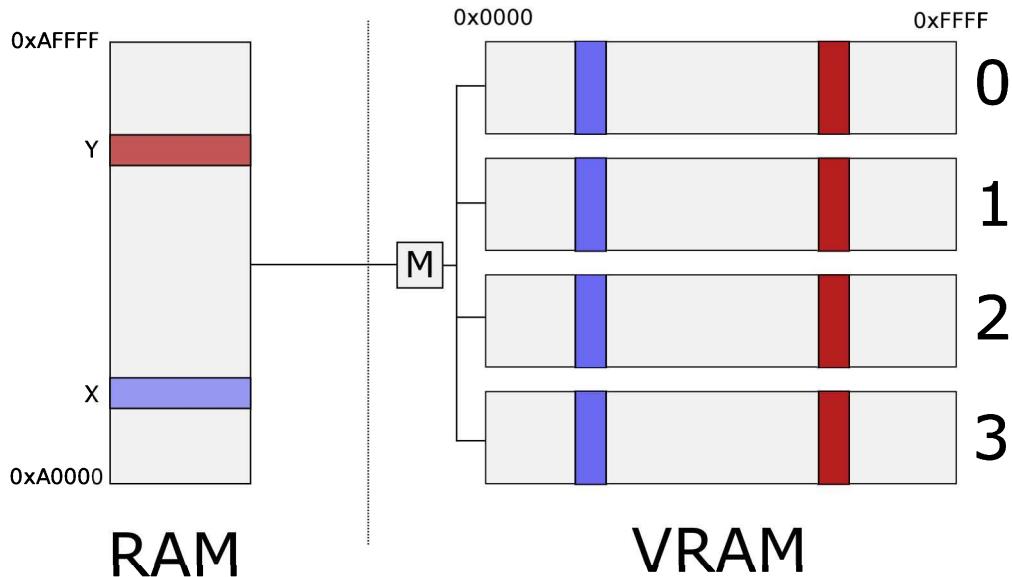
<b>Mode</b>	<b>Type</b>	<b>Format</b>	<b>Colors</b>	<b>RAM Mapping</b>
0	text	40x25	16 (monochrome)	B8000h
1	text	40x25	16	B8000h
2	text	80x25	16 (monochrome)	B8000h
3	text	80x25	16	B8000h
4	CGA Graphics	320x200	4	B8000h
5	CGA Graphics	320x200	4 (monochrome)	B8000h
6	CGA Graphics	640x200	2	B8000h
7	MDA text	9x14	3 (monochrome)	B0000h
0Dh	EGA graphic	320x200	16	A0000h
0Eh	EGA graphic	640x200	16	A0000h
0Fh	EGA graphic	640x350	3	A0000h
10h	EGA graphic	640x350	16	A0000h
11h	VGA graphic	640x480	2	A0000h
12h	VGA graphic	640x480	16	A0000h
13h	VGA graphic	320x200	256	A0000h

**Figure 2.24:** VGA Modes available.

Programmers referenced VGA modes by their ID. It was common to see tutorials about mode 12h or mode 13h, which were the two best-suited modes for game programming.

### 2.3.5 VGA Programming: Memory Mapping

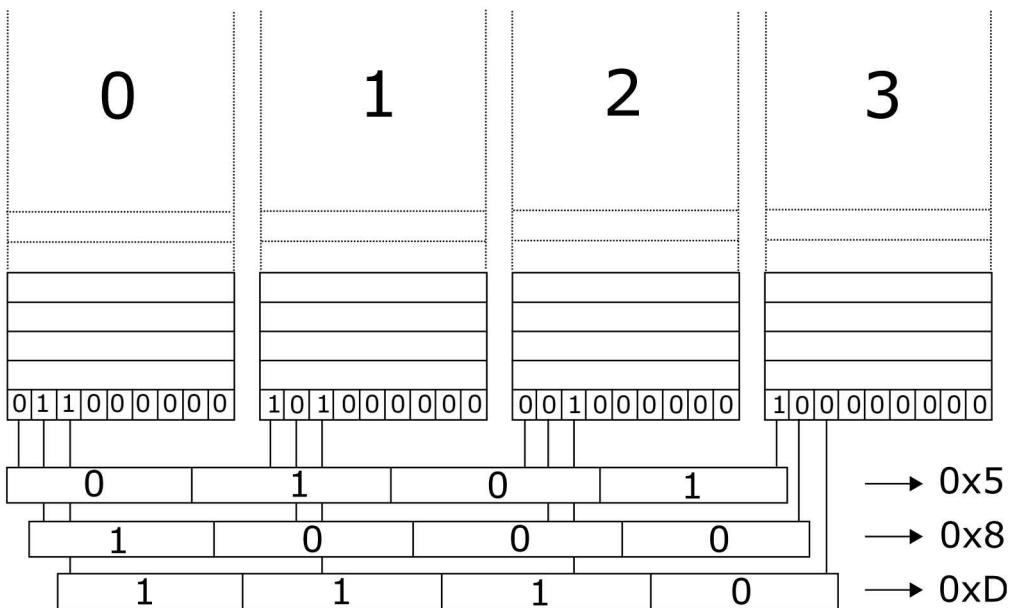
To write to the VRAM, the RAM's 1MiB address space maps 64KiB starting as indicated in the Mode table. In mode 13h for example, the VRAM is mapped from 0xA0000 to 0xFFFF. One of the first questions to come to mind is "How can I access 256KiB of RAM with only 64KiB of address space?" The answer is "bank switching". Write and Read operations are routed based on a mask register indicating which bank should be read or written to.



**Figure 2.25:** Mapping PC RAM to VGA VRAM banks.

### 2.3.6 VGA Programming: Mode 12h

The first mode commonly considered for game programming is mode 12h. It offers a resolution of 640x480 at 70hz with 16 colors. Each pixel is encoded in 4 bits (a nibble) spread across the four banks. To write the color of the first pixel, a developer has to write the first bit of the nibble in plane 0, the second in plane 1, the third in plane 2 and the fourth in plane 3. The CRT Controller then reads 4 bytes at a time (one from each plane) resulting in 8 pixels on screen.



**Figure 2.26:** VGA bank layout in mode 12h.

The only good thing about this mode is that it has square pixels. The 640x480 aspect ratio matches the 4:3 of a CRT so there is no distortion of the framebuffer when it is output to the screen. Pretty much everything else is bad:

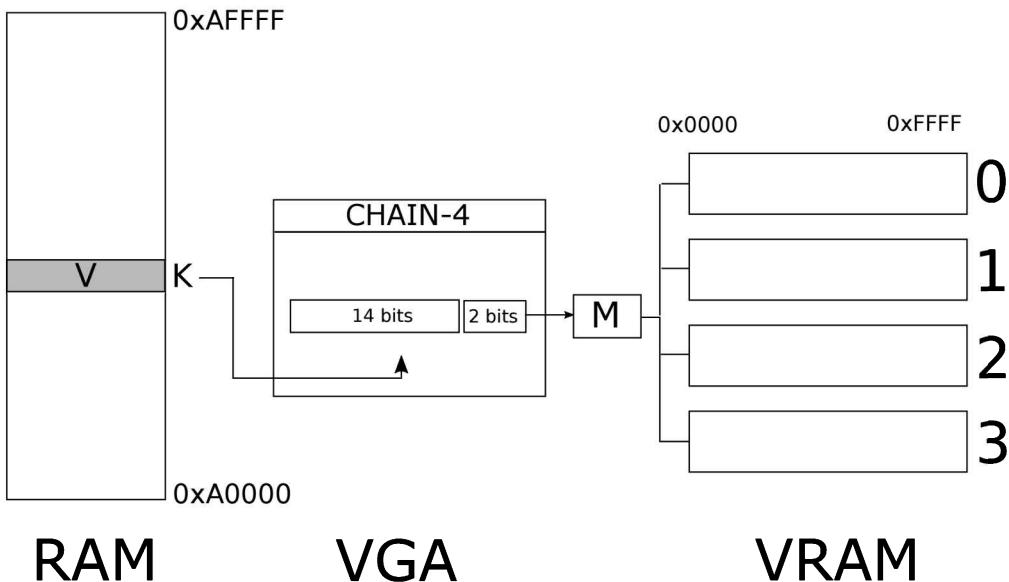
- No double buffer:  $640 \times 480 / 2 = 0x25800$  bytes which is more than half the 256KiB ( $0x40000$ ) of VRAM available.
- High resolution means more pixels, which means more calculations and more drawing.
- 16 color graphics look really, REALLY ugly.



*Figure 2.27:* Wolfenstein 3D in 16 colors.

### 2.3.7 VGA Programming: Mode 13h

Mode 13h is far more appealing since it offers a lower resolution of 320x200 with 256 colors at 70Hz. It also has the advantage of faking linear buffer. A special chip called Chain-4 uses the lower 2 bits of the RAM address to automatically program the mask and route the operation to the appropriate VRAM bank. This convenience mechanism was originally added because mode 13h was meant to display static images and a linear address space made it easy for developers to copy from RAM to VRAM.



**Figure 2.28:** The Chain-4 chip routes I/O between RAM and VRAM.

When writing or reading a value V at address K in RAM, the Chain-4 breaks the address down into two parts:

- The 2 lower bits are used to configure the mask automatically. 0x00 goes to bank 0, 0x01 to bank 1, 0x10 to bank 2 and 0x11 to bank 3.
- The 14 higher bits are right shifted by two bits and used as an offset in the bank.

For example, writing at 0xABF13 in RAM would result in writing in bank 3 at offset 0xBF13  $\gg 2 = 2FC4$ . Since this is all hardwired in the Chain-4, this extra work is fast and totally invisible to the user.

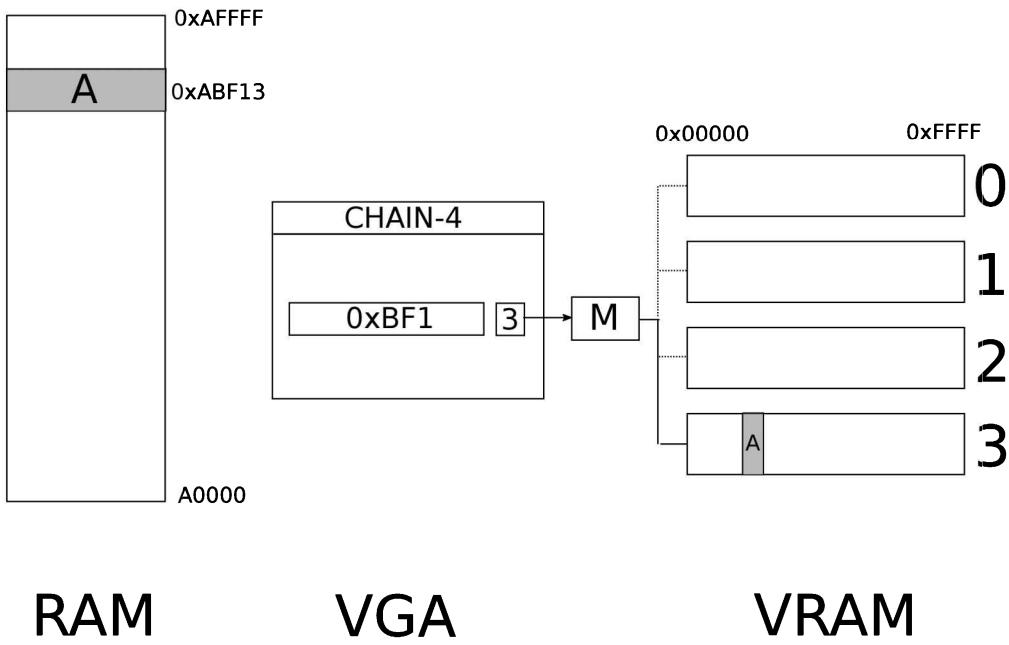


Figure 2.29

The side effect of this convenience mechanism is that 75% of the RAM is wasted (since only 14 bits are usable for offset).

### 2.3.7.1 Setup

To setup the VGA in Mode 13h using the BIOS is incredibly easy:

```
mov ax, 0x13
int 0x10
```

The int 10 instruction is a software interrupt caught by the BIOS routine in charge of graphic setup. It looks up the ax register to setup all 300 VGA registers with the corresponding mode. After the VGA is initialized one can write to the mapped memory at 0xA0000. This can be demonstrated with a code sample; here is some code to clear the screen to black.

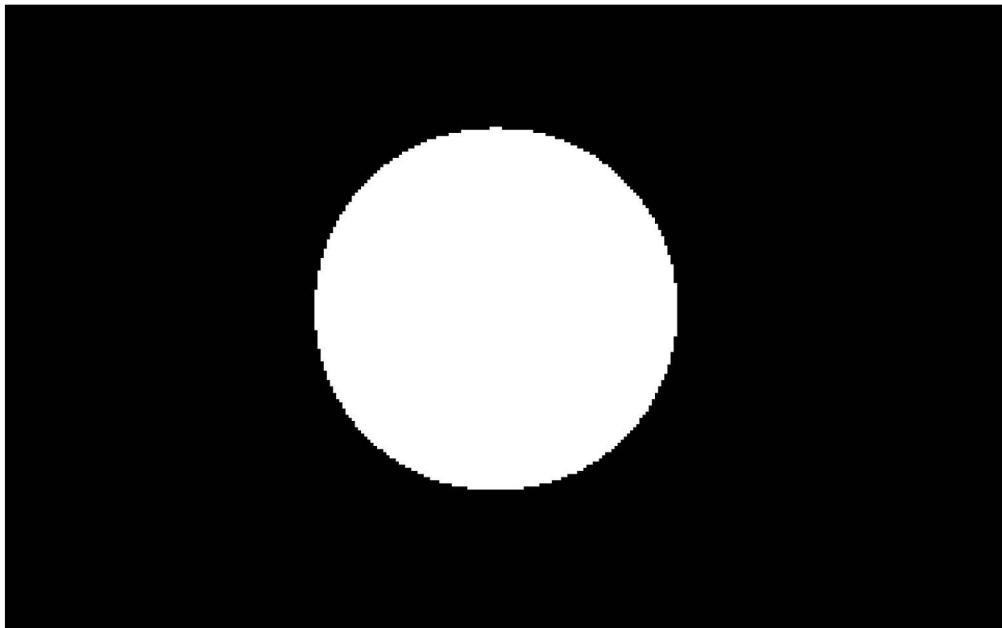
```
char far *VGA = (byte far*)0xA0000000L;

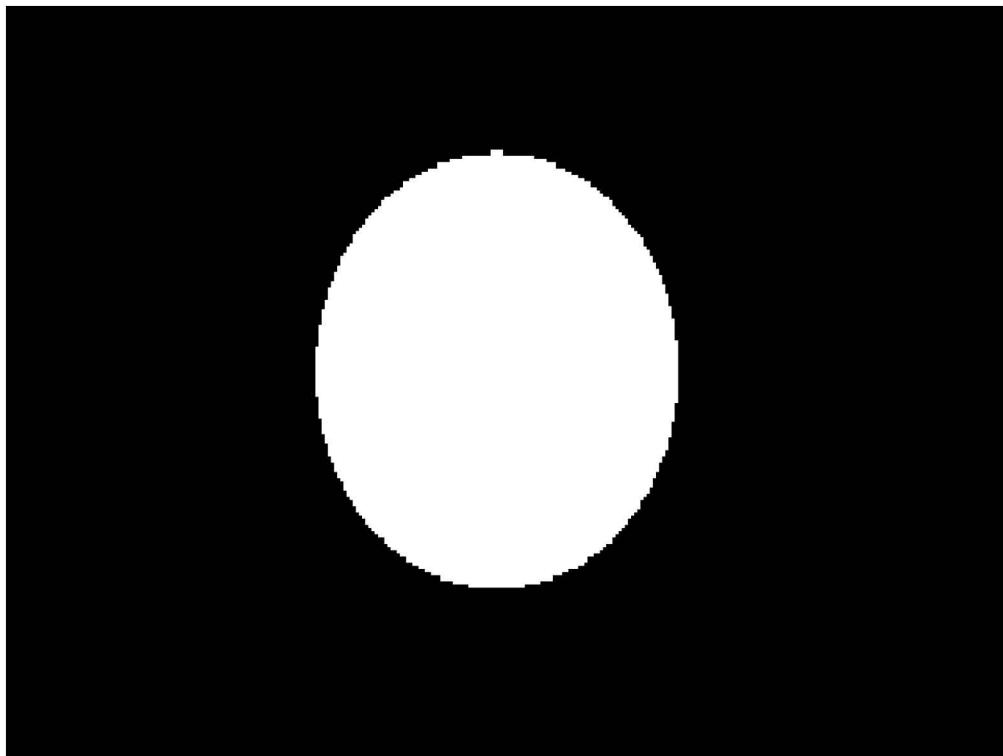
void ClearScreen(void){
    asm mov ax,0x13
    asm int 0x10

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

Mode 13h looks a bit better than 12h but is still pretty terrible for games or even static images:

- With Chain-4 all the RAM address space is used and there is no way to double buffer.
- Since the resolution is 320x200, the aspect ratio (1.6) does not match the monitor's (1.333). As a result the framebuffer stored in the VRAM is stretched when transferred to the CRT. This may seem like a small distortion but is quite problematic. An example with a circle in the framebuffer displaying as an ellipse on the screen speaks for itself.

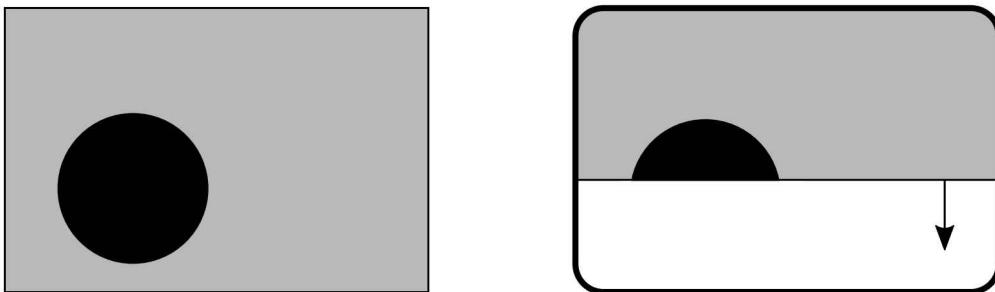




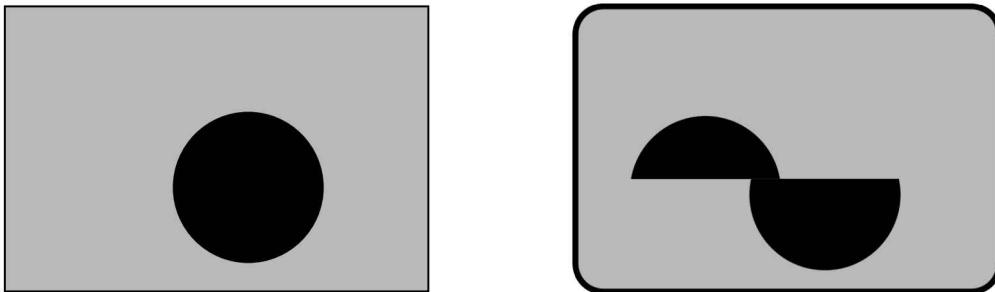
**Figure 2.30:** How the framebuffer appears once stretched on the CRT monitor.

### 2.3.8 The Importance of Double Buffering

Double buffering has been mentioned often while describing the hardware, but so far we have not reviewed why it is paramount to achieving smooth animation. With only one buffer the software has to work at exactly the frequency of the CRT (70Hz). Otherwise a phenomenon known as "tearing" appears. Let's take the example of an animation rendering a circle moving from the left to the right:



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



If the CPU is faster than the frequency of the CRT (70Hz), it can write the framebuffer again, before the scan is completed. This is what happened here. The next frame was drawn with the circle moved to the right. The electron beam did not know that and kept on scanning the framebuffer. The result on screen is now a composite of two frames. It looks like two frames were torn and taped back together. Hence the name "tearing".

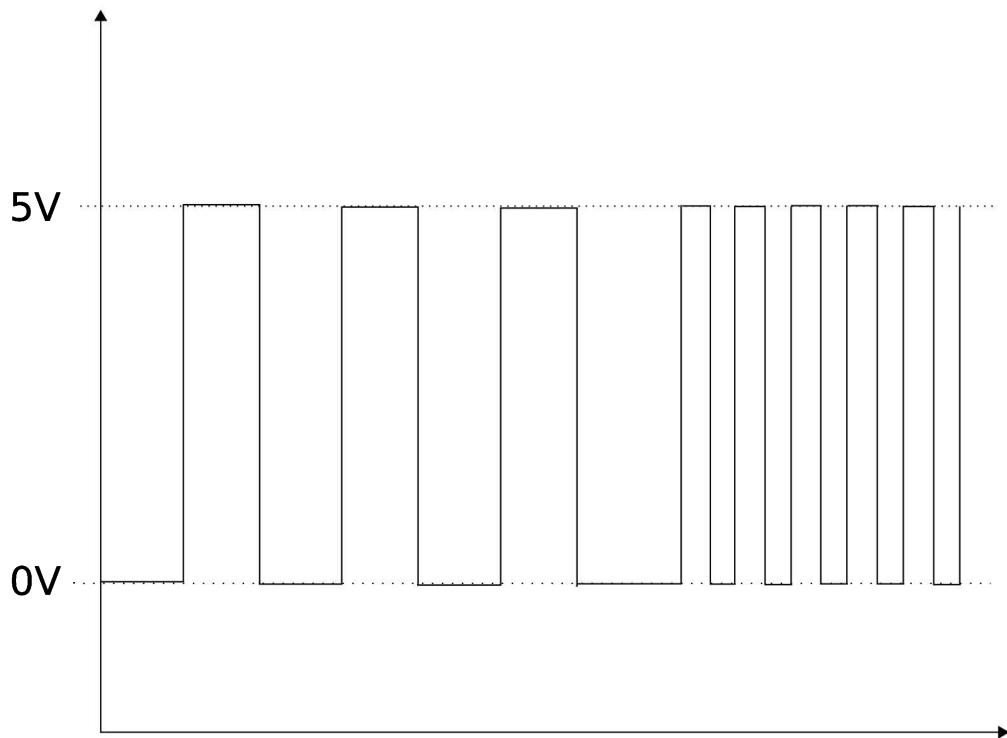
With two buffers (a.k.a double buffering) the CPU can start writing in the second framebuffer without messing with the framebuffer being scanned to the screen<sup>29</sup>. No more tearing!

## 2.4 Audio

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

---

<sup>29</sup>Now the CPU speed is capped by the CRT refresh rate. Triple buffering can solve this at the price of frame latency.



**Figure 2.31:** Two beeps of different frequencies generated via PC Speaker.

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

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

However, square waves are not useful for producing anything pleasant. Some people had started to see a market and companies began manufacturing what were known as

"sound cards". Users could buy these separately and insert them into one of the machine ISA slots. These cards could be connected to real audio speakers via 3.5mm jacks and tremendously improved sound capabilities. In 1991, there were four cards on the market:

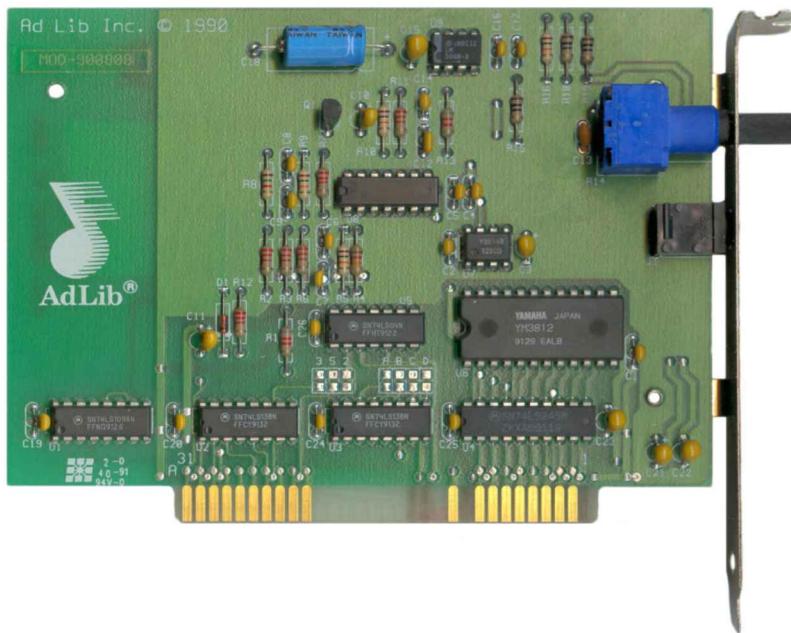
- AdLib music card
- SoundBlaster 1.0
- SoundBlaster Pro
- Disney Sound Source

Although adoption was growing (Creative would go on to sell one million SoundBlaster cards in 1991), the majority of PCs had no sound card which once again presented a huge problem for game developers.

### 2.4.1 AdLib

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

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



**Figure 2.32:** An AdLib sound card. Notice the big YM3812 chip and the 8 bit ISA connector.

**Trivia :** Canadian companies, and especially those from Quebec, were prevalent in the early 90s due to their technological prowess. AdLib manufactured Sound Card, Matrox made a killing with its Millenium Graphic Card, and Watcom sold the best DOS C compiler<sup>30</sup>. ATI<sup>31</sup> would later emerge as a major GPU innovator in the 2000s.

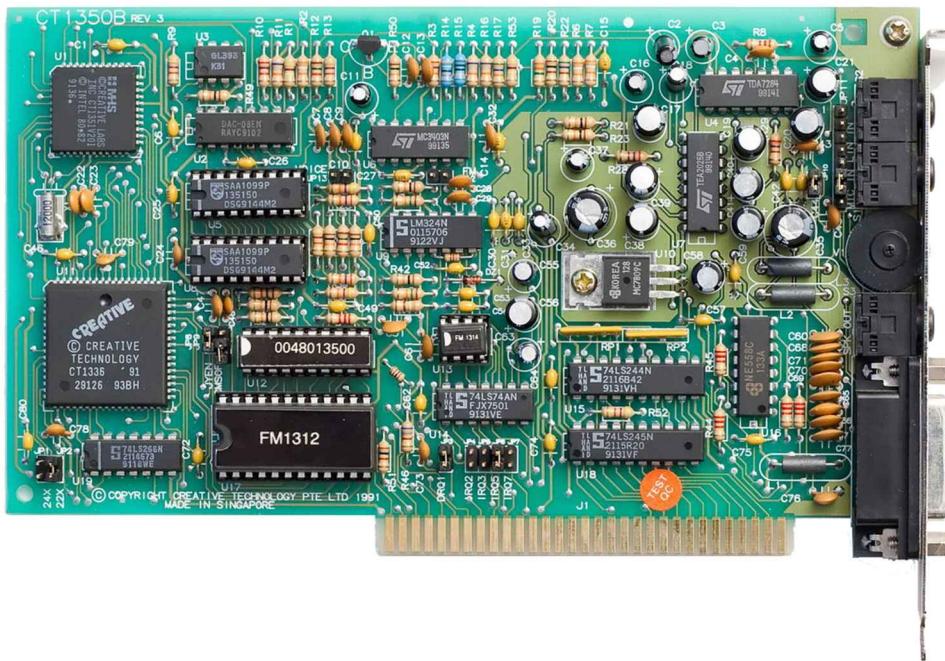
## 2.4.2 Sound Blaster

The Sound Blaster 1.0 (code named "Killer Kard"), was released in 1989 by Creative. It was a smart product which was clearly targeting AdLib's dominant position. Not only was it equipped with the same OPL2 chip, providing 100% compatibility with AdLib music playback, but it was also technologically superior with a DSP<sup>32</sup> allowing PCM playback (digitized sounds) at 8 bits per sample and up to 22.05kHz sampling rate. The card also came with a DA-15 port allowing joystick connection. Most importantly, the SoundBlaster was \$90 cheaper than the AdLib.

<sup>30</sup>Watcom's compiler was so good id would use it to compile Doom.

<sup>31</sup>History would repeat itself in the late 90s in the field of graphic cards: Nvidia vs ATI.

<sup>32</sup>An Intel MCS-51 "Digital Sound Processor", not "Digital Signal Processor".



**Figure 2.33:** A SoundBlaster (v1.2).

Figure 2.33 is the Sound Blaster model CT1350B. Notice the OPL2 chip (labeled FM1312), the big CT1336 bus interface (labeled "CREATIVE") on the center left, the CT1351 DSP on the upper left, and the 8 bit ISA bus connector.

**Trivia :** The numerous advantages of the Sound Blaster card over the AdLib made it the de-facto standard shortly after its release and eventually brought AdLib to bankruptcy<sup>33</sup>.

### 2.4.3 Sound Blaster Pro

The Sound Blaster Pro had all the capabilities of a Sound Blaster 1.0 but added support for stereo 22.05 kHz playback and 44.1 kHz in mono. It also added a "mixer" to blend audio sources (mic, line in, and CD) and choose the attenuation level of left and right outputs. Stereo was achieved with a pair of YM3812 chips (one for each audio channel).

---

<sup>33</sup>The reign of the Sound Blaster came to an end with Windows 95, which standardized the programming interface at application level and eliminated the importance of compatibility with Sound Blaster



**Figure 2.34:** A Sound Blaster Pro card.

The model above is a CT1330A. Notice the double FM1312 chip in the center. The two big chips in the upper left are the DSP (CT1341) and the Mixer. Below, labeled "Creative" is the big CT1316 bus interface.

**Trivia :** The card appears to have a 16 bit ISA bus connector (check the difference with the Sound Blaster 1.0). However it does not have 'fingers' for data transfer on the higher "AT" portion of the bus connector. It uses the 16 bit extension to the ISA bus to provide the user with an additional choice for an IRQ (10) and DMA (0) channel only found on the 16 bit portion of the edge connector.

**Trivia :** Notice on the very left of the card in black there is an IDE Data interface to connect a CD-ROM. That was the only way to connect a CD-ROM to a PC back then.

#### 2.4.4 Disney Sound Source

In 1990, Disney began selling the Disney Sound Source (DSS). Plugged into the printer port (parallel port) of the PC, an 8-bit DAC similar to the "Covox Speech Thing" was connected to a speaker box. It was incredibly easy to set up, simple to program (it could only play one type of PCM and had no FM synthesizer), and very cheap compared to the other audio solutions (\$14). It would have made programmers and customers happy if not for one serious limitation. The parallel port bandwidth<sup>34</sup> allowed a sampling rate up to 18,750 Hz but the design of the DSS limited the PCM sampling rate to 7,000Hz. This was still

<sup>34</sup>The parallel port maximum bandwidth is 150 kbytes/s.

enough to produce pleasant sounds, but fell short when compared to the 22kHz or even 44kHz of a Sound Blaster Pro.

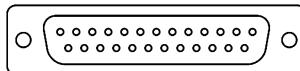


**Figure 2.35:** The speaker box (DAC not shown).

## 2.5 Inputs

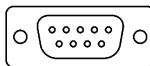
At a time before the ubiquitous USB, inputs were a mess with no less than four ports, all programmed differently.

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



**Figure 2.36:** Parallel Port

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



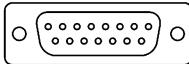
**Figure 2.37:** Serial Port

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



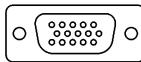
**Figure 2.38:** PS/2 Port

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



**Figure 2.39:** Game Port

**Trivia :** The CRT monitor was connected to the VGA card via a DE15 port. More than 20 years later manufacturers are still trying to get ride of the "VGA port" as it is now commonly called. It looked a lot like a serial port and newcomers often damaged it by forcing a mouse into it.



**Figure 2.40:** VGA Port

## 2.6 Bus

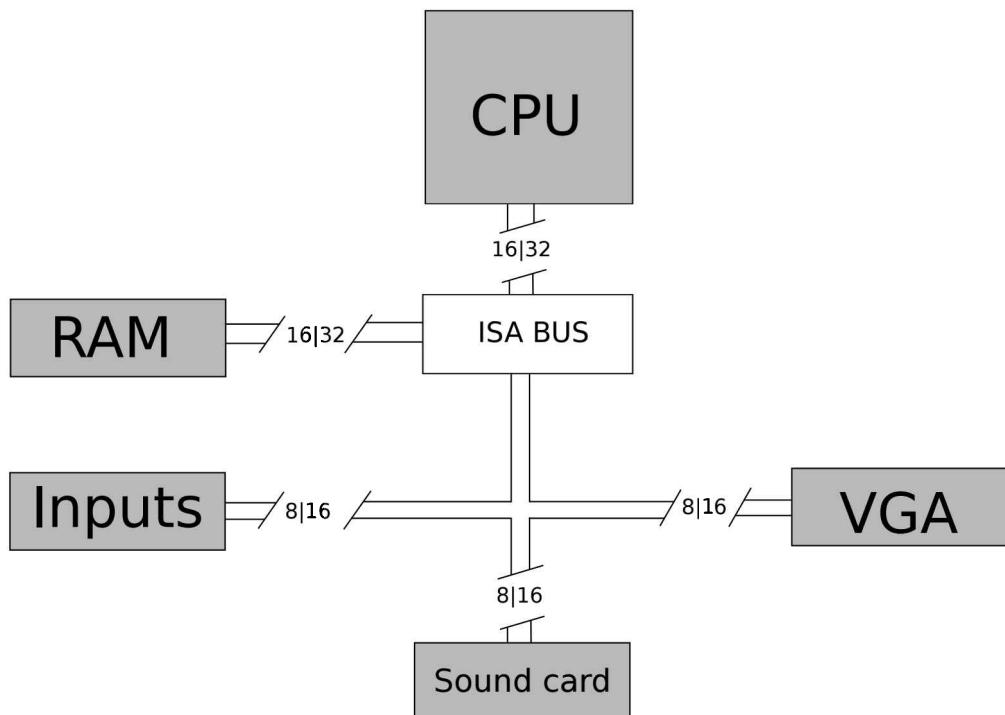
Although developers had no control over them, it is still worth mentioning how these components were connected to each other.

The ISA<sup>35</sup> bus connects the CPU to all devices, including RAM. It was already 10 years old in 1991 but still used universally in PCs. The data path to the RAM is either 16 bit wide for 286 and 386SX or 32 bit on 386DX based machines. It runs at the same frequency as the CPU.

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

- 8 bit wide at 4.77 MHz for 19.1 Mbit/s
- 16 bit wide at 8.33MHz for 66.7 Mbit/s<sup>36</sup>.

Of course it is also backward compatible and an 8 bit ISA card can be plugged into a 16 bit ISA bus.

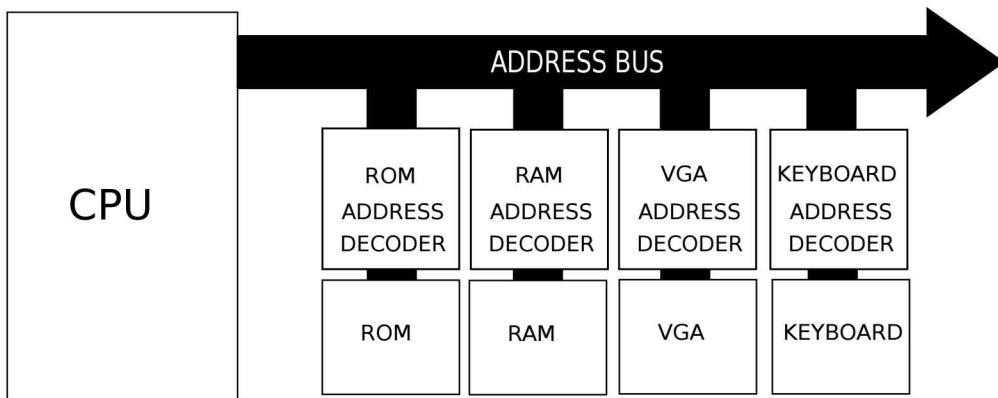


**Trivia :** On ISA all devices are connected to the bus at all times and listen on the bus address lane. Each device features an "address decoder" to detect if it should reply to a bus request. This is how the VGA RAM is "mapped" in RAM. The VGA card "address decoder" filters out everything that is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

<sup>35</sup>Industry Standard Architecture.

<sup>36</sup>[https://en.wikipedia.org/wiki/List\\_of\\_device\\_bit\\_rates](https://en.wikipedia.org/wiki/List_of_device_bit_rates) .

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



In practice the effective bandwidth of the bus is divided by two due to packet overhead and interrupts. As a result, a PC equipped with an 8 bit ISA VGA card can push  $19.1\text{Mbit}\cdot\text{s}/8/2 = 1.1\text{MB/s}$ . In mode 13h, since a frame is  $320 \times 200 = 64,000$  bytes, the theoretical maximum framerate with a CPU taking 0ms to render a frame is  $1,100,000 / 64,000 = 17$  frames per seconds.

On a 16 bit VGA card,  $33,400,000$  bits per seconds gives  $33,400,000/8/64000 = 65$  frames per seconds.

If you factor in other things which had to be transported by the bus such as palette, keyboard interrupt, mouse inputs, and music/sounds (at 23kHz sampling a digitized sound effect consumes  $23,000/70 = 328$  bytes per frame) it is easy to understand how important it was to limit data transfer and why few programs of that era could max out the VGA's 70 frames per seconds<sup>37</sup>.

## 2.7 Summary

To say a PC was difficult to program for games would be an understatement. It was a nightmare. The CPU was good at doing the wrong thing, the best graphic interface allowed neither double buffering nor square pixels, the memory model only allowed 1 standard MiB with an address composed of two separate 16 bit registers, and the near/far pointers forbade using standard C. Last but not least, the default sound system could only produce square waves and the people who did have a sound card installed could have any of the

<sup>37</sup>Specialized demos could reach 70 fps on VGA by careful management of unchanged regions or using planar writes for the 4x speedup.

three major brands.

Yet despite all these unfavorable conditions, teams of developers gathered to tame the beast and unleash its power to gamers. One of these called themselves id Software<sup>38</sup>.

---

<sup>38</sup>They originally called themselves Ideas From the Deep but then decided to shorten it to simply id, which stands for "in demand," and is pronounced as in "did" or "kid." The name also refers to id, the part of the brain that behaves by the pleasure principle in Freudian psychology.



# Chapter 3

## Team

In 1990 a small company based in Shreveport, Louisiana was doing well in the shareware market. As a video game subscription service, Softdisk produced and mailed new games every month to its members. Business was good but some of its employees had other ambitions. They thought they had the skills to make it big and they wanted to prove it.

They had created a new way to program side scrolling. They called the technology "adaptive tile refresh" and it enabled hardware scrolling on a PC capable of rivaling a NES. In early 1990 they worked non-stop over a weekend to reimplement Super Mario 3 on a PC and demonstrate their skills to Nintendo. The team was successful in building a clone of Mario, but unfortunately "Ideas from the Deep" as they called themselves failed to convince Nintendo to give them a contract. As impressed as they were, the Japanese firm wanted the Mario series to remain exclusive to Nintendo consoles.

We sent this demo to Nintendo of America, they in turn sent it to Kyoto to the mothership office, and the execs there saw the demo and were really impressed. However, they didn't want their intellectual property on anything but their own hardware, so they told us Good Job and You Can't Do This<sup>1</sup>.

**John Romero - Programmer**

---

<sup>1</sup>Super Mario Bros. 3 Demo (1990) by John Romero: <https://vimeo.com/148909578>



**Figure 3.1:** Mario 3 on PC. Notice the "IFD" for "Ideas From the Deep".

This episode was enough to convince them they had not only the talent to feed their ambitions, but also the teamwork and work ethic to potentially go all the way. In February 1991 four Softdisk employees took the leap of faith and founded their own company: "id Software"<sup>2</sup>.

Name	Age	Occupation
John Carmack	21	Programming
John Romero	25	Programming
Adrian Carmack	22	Artist
Tom Hall	28	Creative Director

**Figure 3.2:** id Software founding members.

They immediately used the technology developed for "Mario 3 PC" to release their own

<sup>2</sup>For ample details, read "Masters of Doom" by David Kushner.

titles and build their own intellectual property. Wasting no time, the team shipped no less than three titles annually.

- Commander Keen Episode 1, 2, and 3: Invasion of the Vorticons (December 14th, 1990)
- Commander Keen Episode 4, 5, and 6: Good Bye Galaxy (December 15th, 1991)
- Commander Keen standalone game: Aliens Ate My Baby Sitter (December 1991)

The games, published by FormGen, were instant successes and sold very well. They also kept on writing games for Softdisk to publish, most of which featured adaptive tile refresh:

- Commander Keen in Keen Dreams (1991)
- Dangerous Dave in the Haunted Mansion (1991)
- Rescue Rover (1991)
- Rescue Rover 2 (1991)
- Shadow Knights (1991)
- Hovertank 3D (April, 1991).
- Catacomb 3D: A New Dimension (November, 1991)

During Spring of 1991 the next generation of id Software technology started to surface<sup>3</sup>. Hovertank 3D placed the player inside a tank. There was no texture mapping yet and the pace was quite slow. Catacomb 3D marked the introduction of textures and took immersion one step further by placing the player in control of a magician in first person view.

In November 1991, the team was free from any obligations to SoftDisk. Their next game was going to feature the 3D technology they were building and would be called Wolfenstein 3D. Given the magnitude and ambition of the title, four more people were added to the team for a total of eight.

---

Name	Age	Occupation
Jay Wilbur	30	Business
Kevin Cloud <sup>4</sup>	27	Computer Artist
Robert Prince <sup>5</sup>	37	Composer
Jason Blochowiak <sup>6</sup>	??	Programming

---

<sup>3</sup>You can see screenshots in the Annex section on page 290 and 291.

“ Jason was part of id at the start, but we parted ways during Wolf development.

John Carmack - Programmer



Above is the team as shown in Spear of Destiny, in an Easter Egg created by John Romero. To view this screen, the player had to go in the Change View menu, hold down I and D on the keyboard, and press Escape.

<sup>4</sup>Jay and Kevin were recruited on April 1st, 1992 but were still given credit for participating in the game development.

<sup>5</sup>Robert had worked with id Software before on Keen 4-6. He remained a contractor, and was never a full-time employee.

<sup>6</sup>Jason wrote part of the page manager and is credited for introducing John Carmack to Unix development, which ultimately led to the purchase of a Next ColorStation.

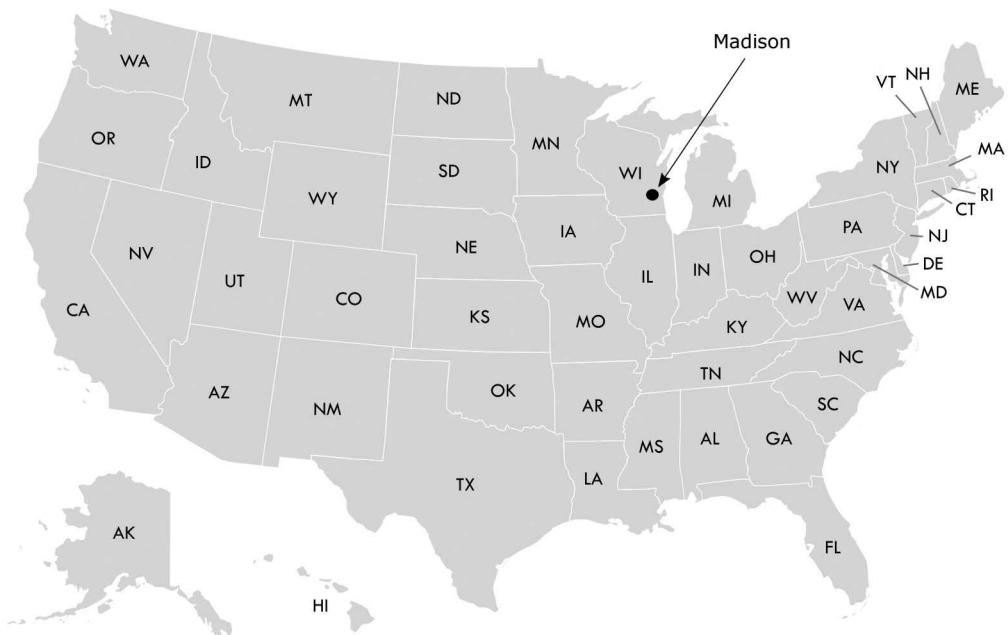


**Figure 3.3:** They were in fact wearing pants.

### 3.1 Organization

In September 1991, following Tom and Jason's high-school memories of the area, the team relocated from Shreveport, LA to Madison, WI. They established their office in a two story brick building at 2622 in The Pines complex on High Ridge Trail. They all lived in walking distance of the office except for John Carmack who, since he did not care, inherited of the second floor of the building.

The development of Wolfenstein 3D started in November 1991. As temperatures fell and snow dumped from the sky, the team kept itself increasingly busy and barely left the office. Development lasted seven months and Wolfenstein 3D was released in May 1992.



During these seven months, the organization of the team was pretty much standard for a game studio of this era: four guys crammed in one room, dictating a fast pace and strong sense of camaraderie (and a lot of noise disturbance given John Romero and Tom Hall's type of interaction).

On the map (next page) you can see on the upper floor the SNES where countless games of F-Zero were played and the Dungeons & Dragons area extensively mentioned in "Masters of Doom". To have a team member (John Carmack) with his apartment directly above the studio was not out of the ordinary<sup>7</sup>.

“

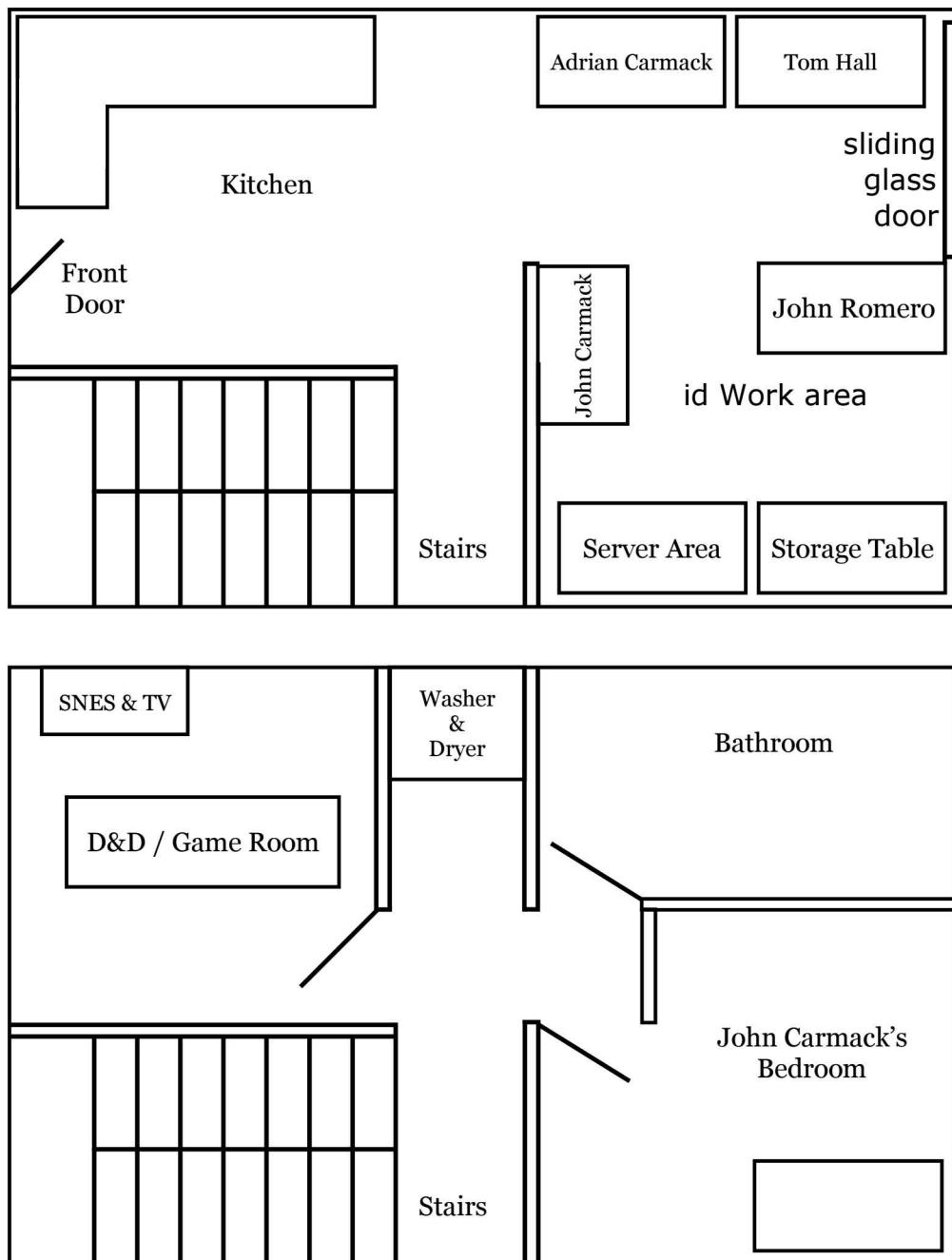
We started with floppy data transfer, but we had a Novell network on coax Ethernet by the end. We didn't have a version control system. Surprisingly, we went all the way to Quake 3 without one, then we started using Visual Source Safe.

**John Carmack - Programmer**

”

---

<sup>7</sup>"90 Hours A Week And Loving It!" by Andy Hertzfeld.



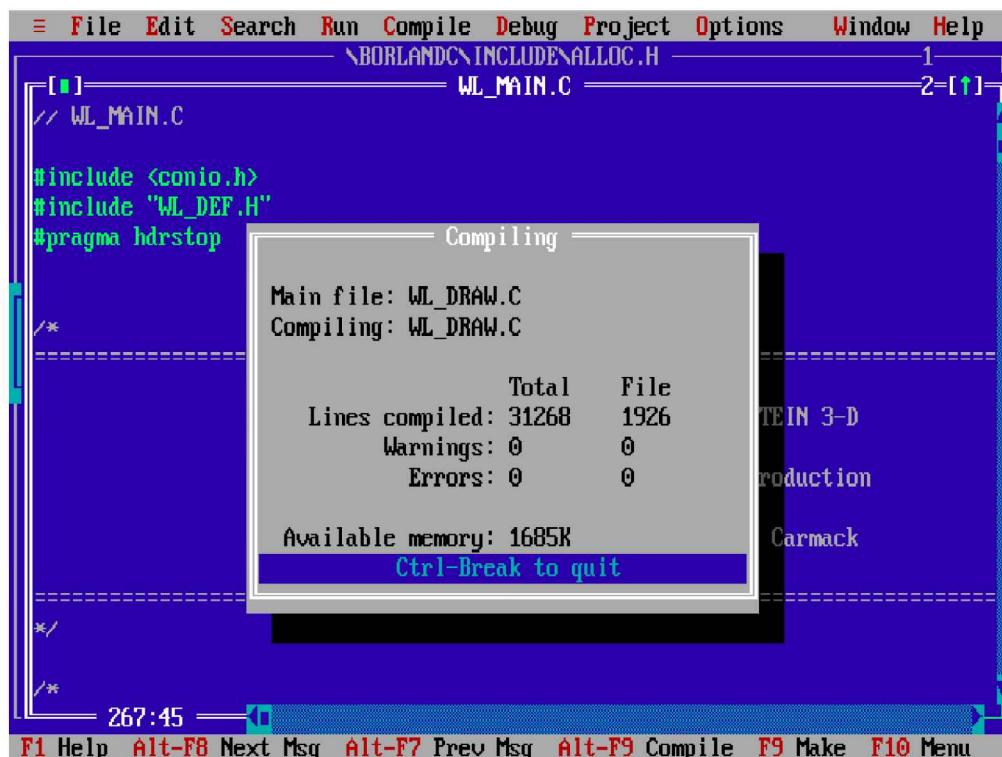
Everybody was working with the best PC money could buy, a high end 386-DX 33Mhz with

4MiB of RAM. As for combining engine, tools, and assets:

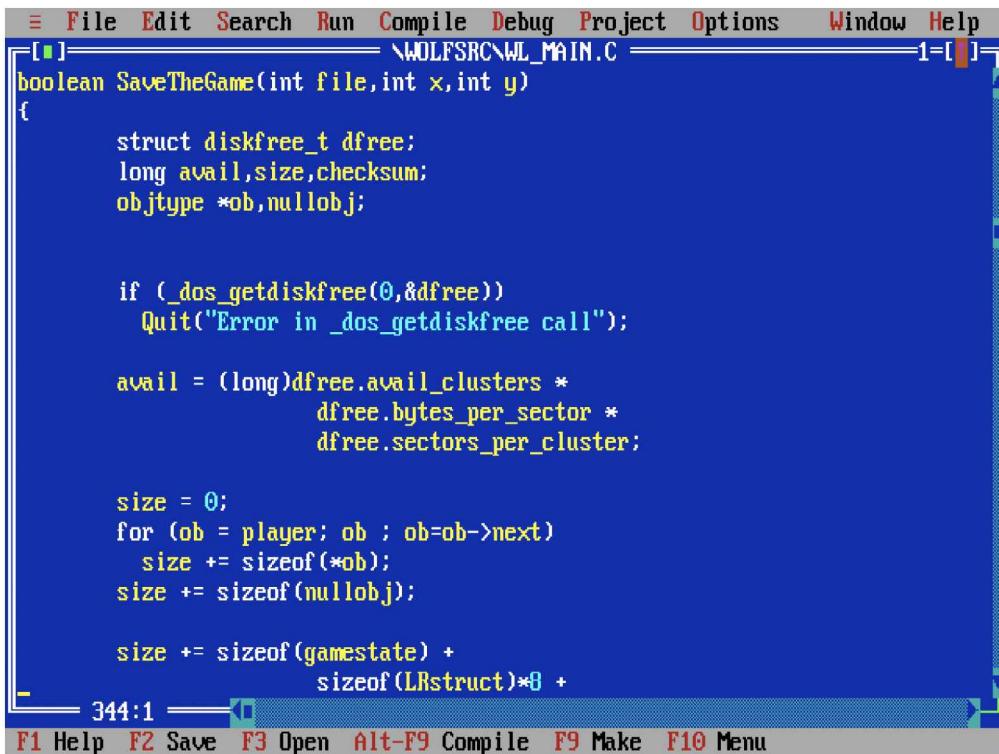
## 3.2 Programming

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

John Carmack took care of the runtime code. John Romero programmed many of the tools (TED5 map editor, IGRAB-ED graphic assets packers, MUSE sound packer). Jason Blachowiak wrote important subsystems of the game (Input manager, Page manager, Sound manager, User manager).



**Figure 3.4:** Compiling Wolfenstein 3D with Borland C++ 3.1



The screenshot shows the Borland C++ 3.1 IDE interface. The menu bar includes File, Edit, Search, Run, Compile, Debug, Project, Options, Window, and Help. The title bar displays the file name \WOLFSRC\WL\_MAIN.C and the line number 344:1. The code editor window contains the following C code:

```

boolean SaveTheGame(int file,int x,int y)
{
    struct diskfree_t dfree;
    long avail,size,checksum;
    objtype *ob,nullobj;

    if (_dos_getdiskfree(0,&dfree))
        Quit("Error in _dos_getdiskfree call");

    avail = (long)dfree.avail_clusters *
            dfree.bytes_per_sector *
            dfree.sectors_per_cluster;

    size = 0;
    for (ob = player; ob ; ob=ob->next)
        size += sizeof(*ob);
    size += sizeof(nullobj);

    size += sizeof(gamestate) +
            sizeof(LRStruct)*8 +

```

The status bar at the bottom shows keyboard shortcuts: F1 Help, F2 Save, F3 Open, Alt-F9 Compile, F9 Make, F10 Menu.

**Figure 3.5:** Borland C++ 3.1 editor

To compensate for the tiny CRT display, some of the developers used two screens (an unusual thing at the time).

“

At that point, we wanted 21" monitors, but couldn't justify them. I used a second mono monitor to allow Turbo Debugger 386 to keep the main screen in graphics mode while I stepped through the code.

**John Carmack - Programmer**

”

You may have noticed in the listing of VGA modes that mode 13h and mode 3h (see Figure 2.24, p47) don't have the same starting RAM address. This allows for a trick where two graphics cards are plugged into the same PC. One MDA setup as monochrome text mode picks up data at 0xB8000 while a VGA mapped at 0xA0000 runs the game normally.

The screenshot shows the Borland Turbo Debugger 386 interface. The menu bar includes File, Edit, View, Run, Breakpoints, Data, Options, Window, Help, and READY.... The status bar at the bottom shows function keys F1 through F10 with their corresponding labels.

The assembly code window displays the following assembly instructions:

```

main:
    cs:17FAM55    push    bp
    cs:17FB 8BEC   mov     bp,sp
#WL_MAIN#1606:
    cs:17FD 9A1C3D554F call    far _CheckForEpisodes
#WL_MAIN#1608:
    cs:1802 0E      push    cs
    cs:1803 E80BEB   call    _Patch386
#WL_MAIN#1610:
    cs:1806 0E      push    cs
    cs:1807 E863FA   call    #WL_MAIN#1145
#WL_MAIN#1612:
    cs:180A 0E      push    cs
    cs:180B E8DBFD   call    _DemoLoop
#WL_MAIN#1614:

```

The registers window on the right shows the following register values:

Register	Value
ax	FF00
bx	4768
cx	0000
dx	4768
si	474A
di	4768
bp	FFE6
sp	FFDA
ds	823E
es	823E
ss	823E
cs	4D38
ip	17FA

The memory dump window at the bottom shows memory starting at address 489F:

```

489F:0000 CD 20 FF 9F 00 EA FF FF = f 
489F:0008 AD DE E0 01 C6 15 AA 01 i |x0|S-0
489F:0010 C6 15 89 02 21 10 93 01 |S|e0!>00
489F:0018 01 01 01 00 02 FF FF FF 0000 0
489F:0020 FF FF FF FF FF FF FF FF

```

The status bar at the bottom lists function keys and their descriptions:

- F1-Help
- F2-Bkpt
- F3-Mod
- F4-Here
- F5-Zoom
- F6-Next
- F7-Trace
- F8-Step
- F9-Run
- F10-Menu

**Figure 3.6:** Monitor 1 connected to MDA card showing Borland Turbo Debugger 386.

This setup allows a developer to debug the game engine with breakpoints.



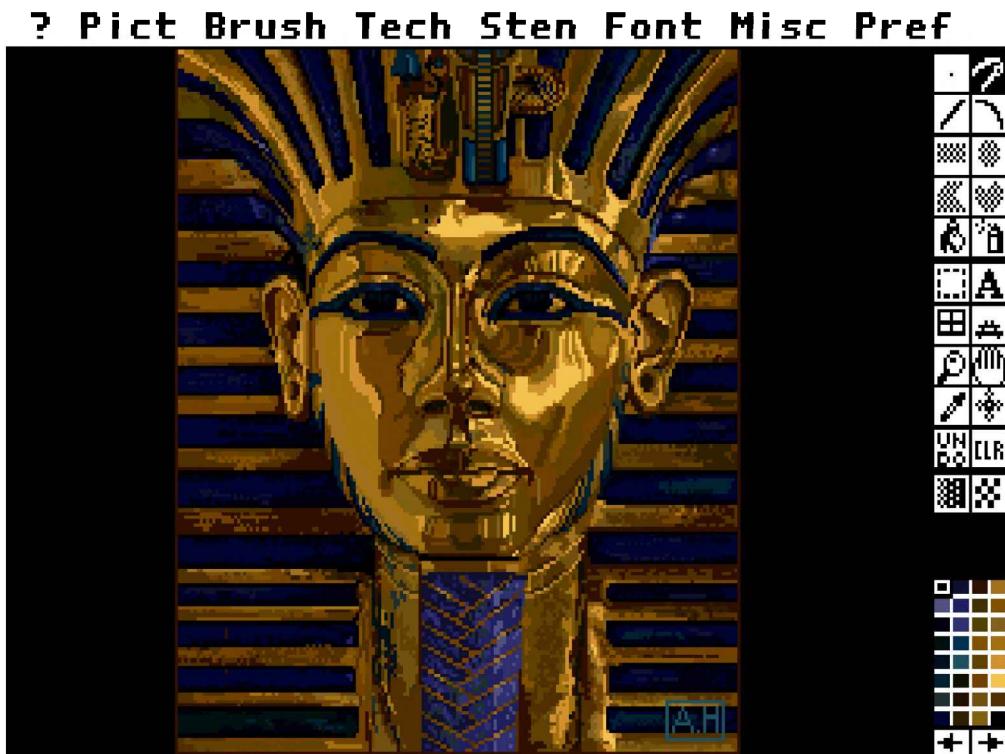
**Figure 3.7:** Monitor 2 connected to VGA card running the game normally.

### 3.3 Graphic Assets

All graphic assets were produced by Adrian Carmack<sup>8</sup>. All of the work was done with Deluxe Paint (from Electronic Arts) and saved in ILBM<sup>9</sup> files (Deluxe Paint proprietary format).

<sup>8</sup>Kevin Cloud did a few textures and also worked on the design and layout of the Wolfenstein 3D hint book.

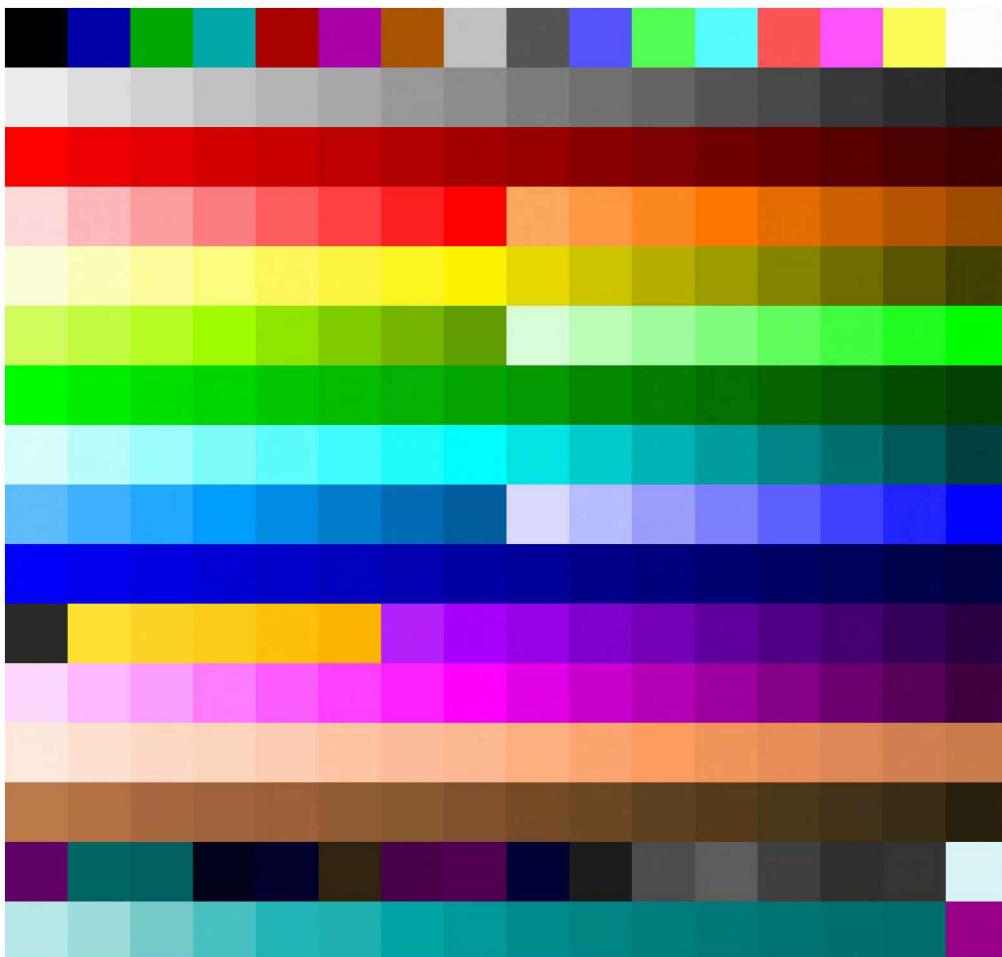
<sup>9</sup>InterLeaved BitMap.



**Figure 3.8:** Deluxe Paint was used to draw all assets in the game.

Since the VGA is palette based (colors were not specified via 24-bit RGB but via indices pointing to a 256 color table) the creative process was difficult. Adrian had to first make the key decision of which colors would go in the palette<sup>10</sup>, then draw everything with only those colors.

<sup>10</sup>Some games like "Monkey Island" used multiple palettes depending on the section of the game. id Software went for a simpler solution with one palette for the whole game.



**Figure 3.9:** Wolfenstein 3D palette. Everything in the game is drawn using these 256 colors.

The palette coordinates run 0x00 to 0x0F horizontally and 0x00 to 0xF0 vertically. The horizontal blue gradient at the bottom starts at 0xF0 and ends at 0xFE. 0xFF (represented in pink) is a special color deemed transparent by the engine and always skipped during rendition.

All assets were hand drawn with a mouse. Since the VGA stretched the framebuffer when displaying it on the screen, Adrian had to be careful to draw at the same resolution as the game would run (320x200).

Adrian and Kevin both worked directly in Deluxe Paint, we didn't have any scanning tools at the time.

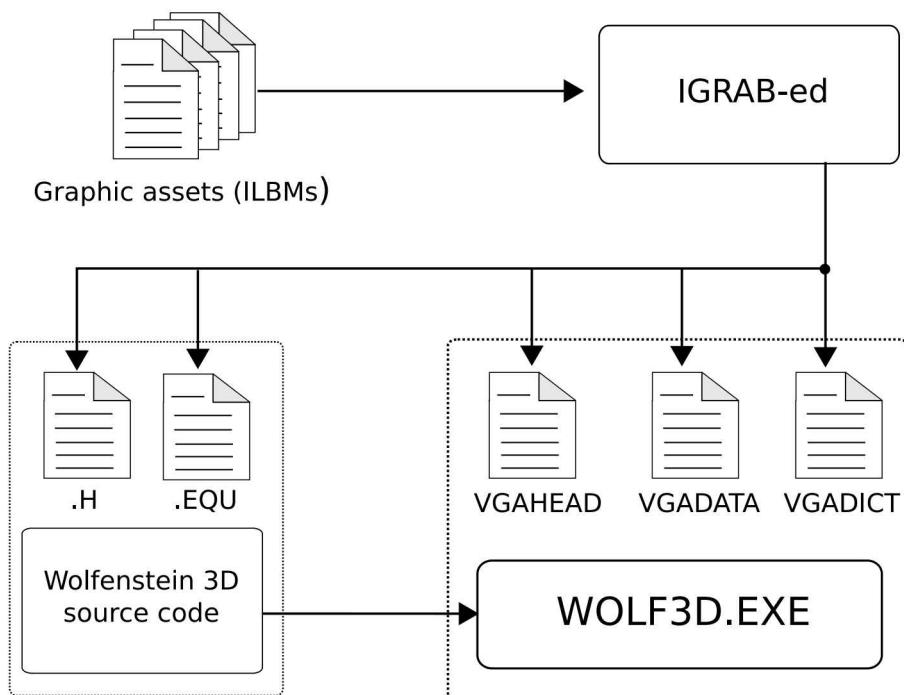
#### **John Carmack - Programmer**

Graphic assets are divided in two categories:

- 2D Menu items shipping with the game in VGADATA, VGAHEAD and, VGADICT
- 3D Action phase items (walls and sprites) shipping in the VSWAP archive

## 3.4 Assets Workflow

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



**Figure 3.10:** Asset creation pipeline for 2D Menu items

```
///////////
//
// Graphics .H file for .WL1
// IGRAB-ed on Sun May 03 01:19:32 1992
//
///////////

typedef enum {
    // Lump Start
    H_BJPIC=3,
    H_CASTLEPIC,           // 4
    H_KEYBOARDPIC,          // 5
    H_JOYPIC,              // 6
    H_HEALPIC,              // 7
    H_TREASUREPIC,          // 8
    H_GUNPIC,                // 9
    H_KEYPIC,                // 10
    H_BLAZEPIC,              // 11
    H_WEAPON1234PIC,         // 12
    H_WOLFLOGOPIC,            // 13
    ...
    PAUSEDPIC,                // 140
    GETPSYCHEDPIC,             // 141
}
```

In the engine code, asset usage is hardcoded via an enum. This enum is an offset into the HEAD table which gives an offset in the DATA archive. With this indirection layer, assets could be regenerated and reordered at will with no modification in the source code.

```
void CheckKeys (void) {
    if (Paused) {
        ...
        LatchDrawPic (20-4,80-2*8,PAUSEDPIC);
        ...
    }
}

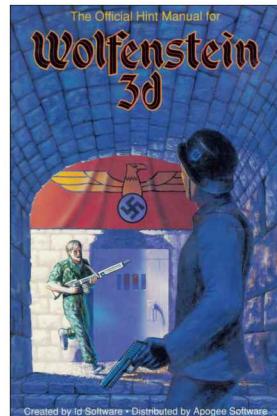
void PreloadGraphics(void) {
    ...
    LatchDrawPic (20-14,80-3*8,GETPSYCHEDPIC);
    ...
}
```

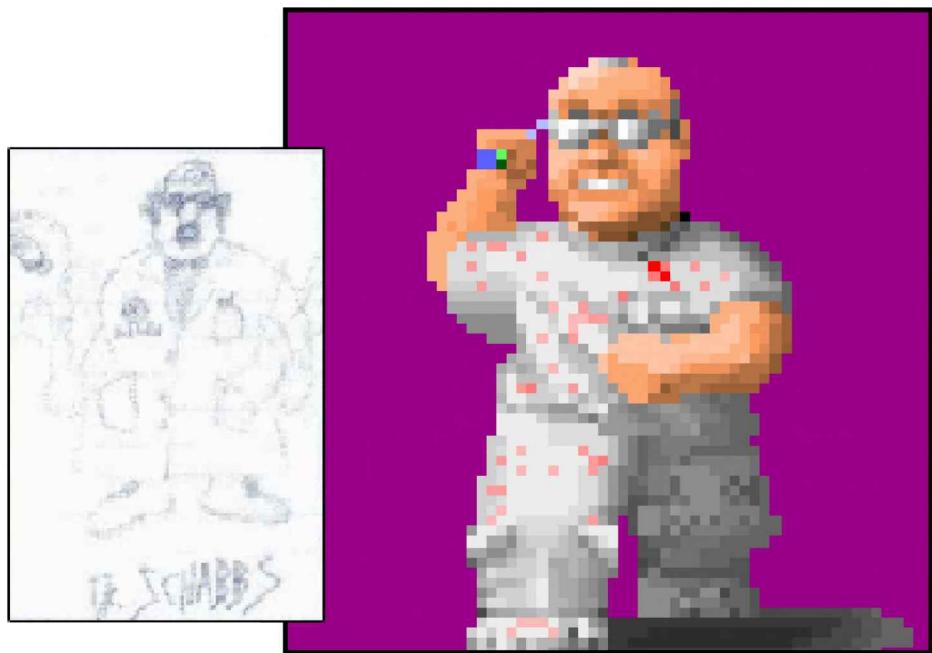
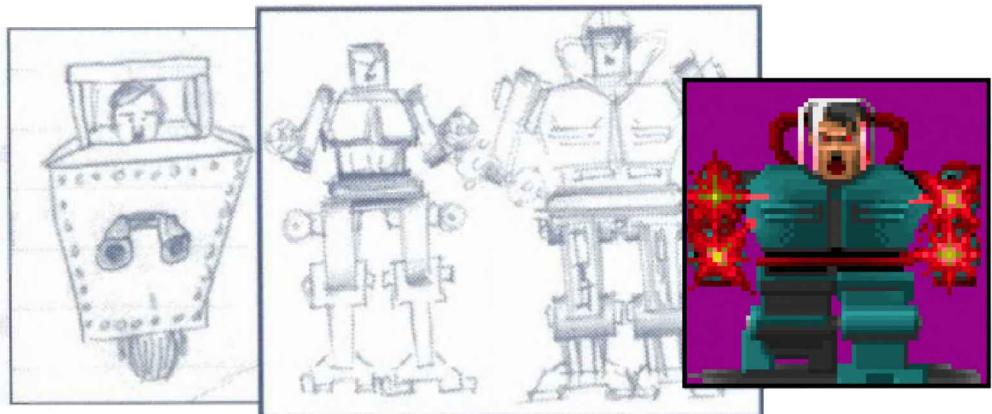
**Trivia :** This system led to issues when the source code was released. The .h header files

provided did not match the asset files from the shareware or early versions of Wolfenstein 3D. The headers released were from Spear of Destiny. You can see the kind of graphic mess this led to in the article "Let's compile like it's 1992" on [fabiensanglard.net](http://fabiensanglard.net).

"The Official Hint Manual for Wolfenstein 3D" published in 1992 explains the creative process. It contains many drawings from Tom Hall and shows many behind the scene sketches made by Tom Hall and pixelarted by the graphic team.

“ When Id's Creative Director, Tom Hall gets an idea for a screen, he provides a sketch for Adrian Carmack. Below are some of Tom's early designs for the title screen. The third sketch was chosen.







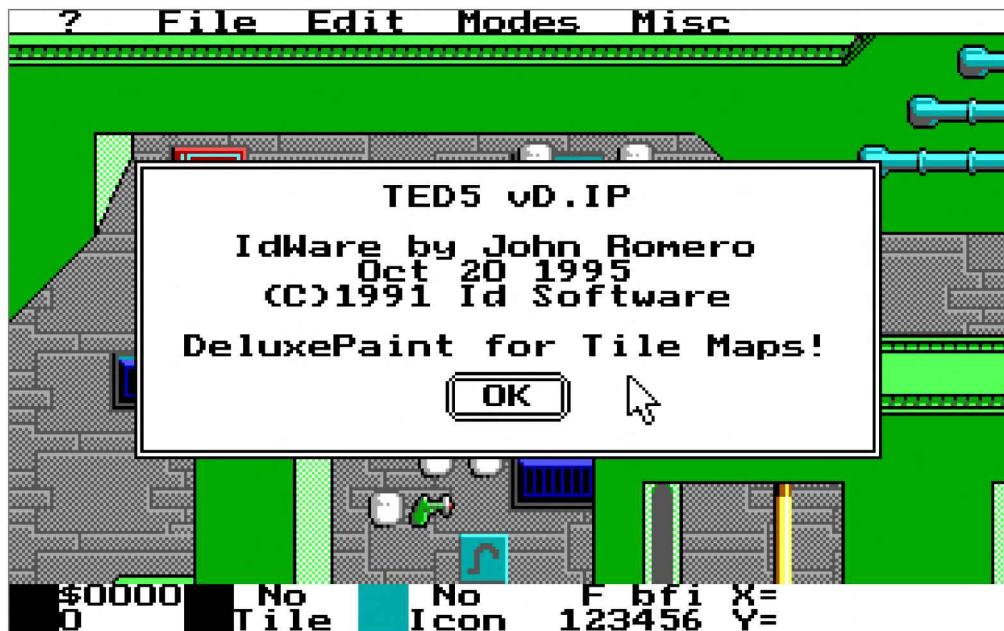


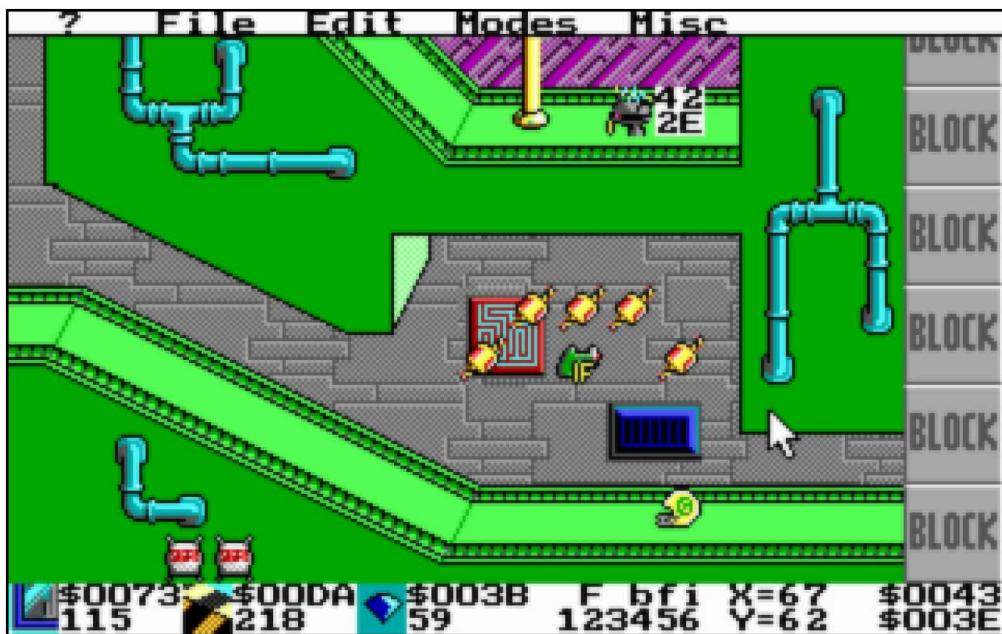
The hint manual also contains several photos of the team back in the day; it is worth a read for context<sup>11</sup>

## 3.5 Maps

Maps were created using an in-house editor called TED5, short for Tile EDitor. TED5 was not created specially for Wolf 3D, but was originally made for the Commander Keen series and improved over the years. It was a versatile tool since it allowed for creating maps of both side scroller games and top-down games like Rescue Rover and Wolf 3D. TED5 is not stand-alone; in order to start, it needs an asset archive and the associated header (as described in the graphic asset workflow Figure 3.10 on page 80). This way, texture IDs are directly encoded in the map.

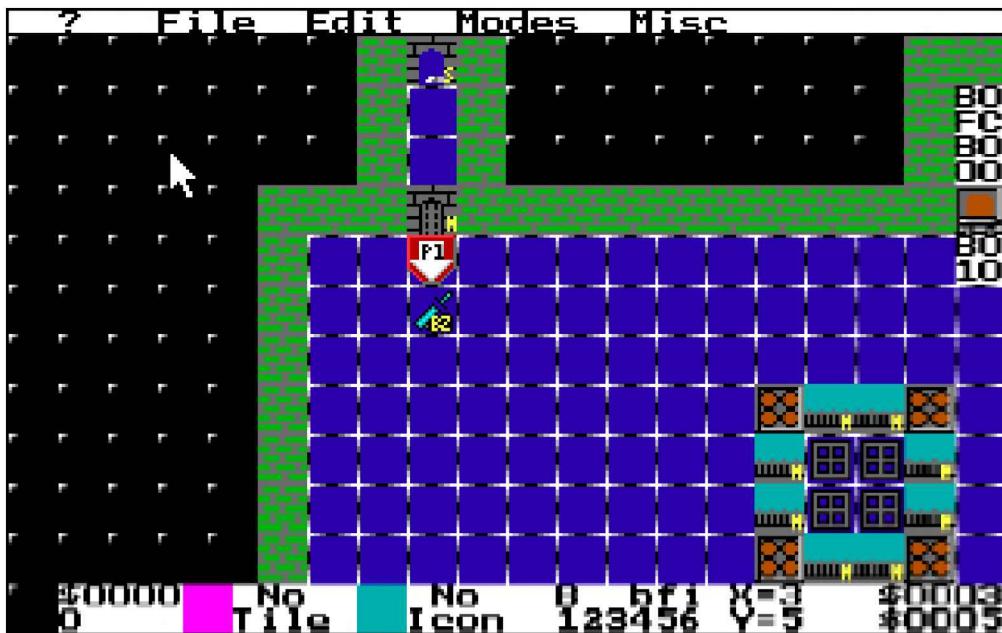
<sup>11</sup>The manual was designed on a NeXT ColorStation. It was the only usage of Steve Jobs's machine for Wolfenstein 3D even though id purchased it in December 1991. NeXT series of workstations later became a key element of the production pipeline for Doom in 1992.





**Figure 3.11:** TED5 used for Commander Keen - side scroller.

TED5 allows placement of tiles on layers called "planes". This layered approach proved powerful and versatile. In Commander Keen, layers are used for background, tiles which the hero can stand on, generating bonuses and so on. For Wolfenstein 3D, two layers are used: one for walls, and one to place bonuses and enemies on.



**Figure 3.12:** TED5 used for top view level design in Rescue Rover.

Reusing TED5 was a double win. Not only did it save tool development time, but ramp-up time was also reduced as all team members had been using it for years. TED5 was so good at doing the job that it allowed designers to make a level within minutes.

“

After talking with Romero and Tom, Scott learned that it was taking the group only about one day to make a level of the game. Ka-chung! Dollar signs! Instead of just three episodes, why not have six? Scott said, "If you can do thirty more levels, it would only take you fifteen days. And we could have it where people could buy the first trilogy for thirty-five dollars or get all six for fifty dollars, or if people buy the first episode and later want the second episodes it will be twenty dollars. So there's a reason to get them all!" After some consideration, id agreed.

- Masters of Doom

”

While everybody worked a little bit on the map, it was mostly the creation of John Romero and Tom Hall.

**Trivia :** The source code of TED5 was released several years later. Among the source code made of .C and .H was a mysterious \_TOM.PIC<sup>12</sup>. It turned out to be an adult caricature of Tom Hall made by Adrian Carmack. The explanation was provided later by John Romero in 2002:

"Hahahaha! Wow, I forgot all about that picture. I can't believe it's in the TED5 source files! It's basically a pic that Adrian drew of Tom [...] saying "Sorry!".

It's because Tom and Adrian used to share a worktable together and Tom would always bump the table while Adrian was drawing graphics with the mouse and Tom would say, "Sorry!" That picture never appears in TED5 anywhere.

**John Romero - Programmer**

## 3.6 Audio

### 3.6.1 Sounds

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

---

<sup>12</sup>Intentionally not reproduced here.



Figure 3.13: MUSE splash screen.

Three sets of each audio effect shipped with the game:

1. For PC Speaker
2. For AdLib
3. For SoundBlaster, SoundBlaster Pro and Disney Sound Source

All voices were recorded by John Romero and Tom Hall faking German accents<sup>13</sup> the best they could<sup>14</sup>.

### 3.6.2 Music

All music composition was done by Robert Prince.

<sup>13</sup>Masters of Doom by David Kushner.

<sup>14</sup>You can actually recognize their voices if you listen carefully: Tom Hall's "guten tag" is memorable.

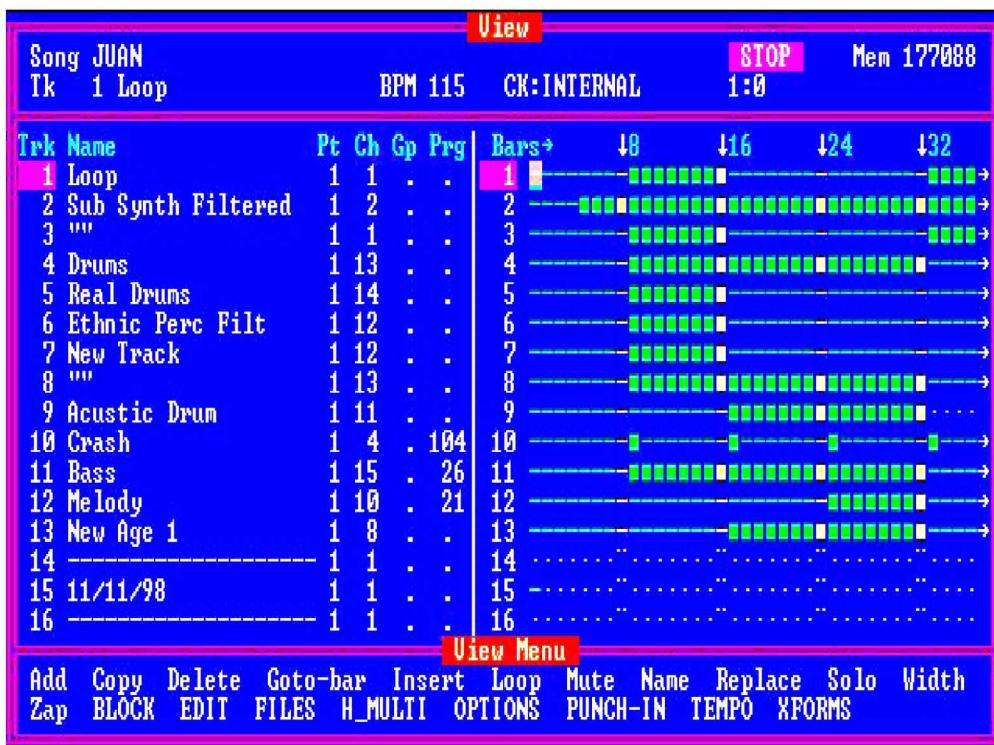
“

In the early days of the OPL soundcards, the "gold standard" sequencing software was Sequencer Plus Gold ("SPG") by Voyetra. The reason for this was it had an OPL instrument/instrument bank editor.

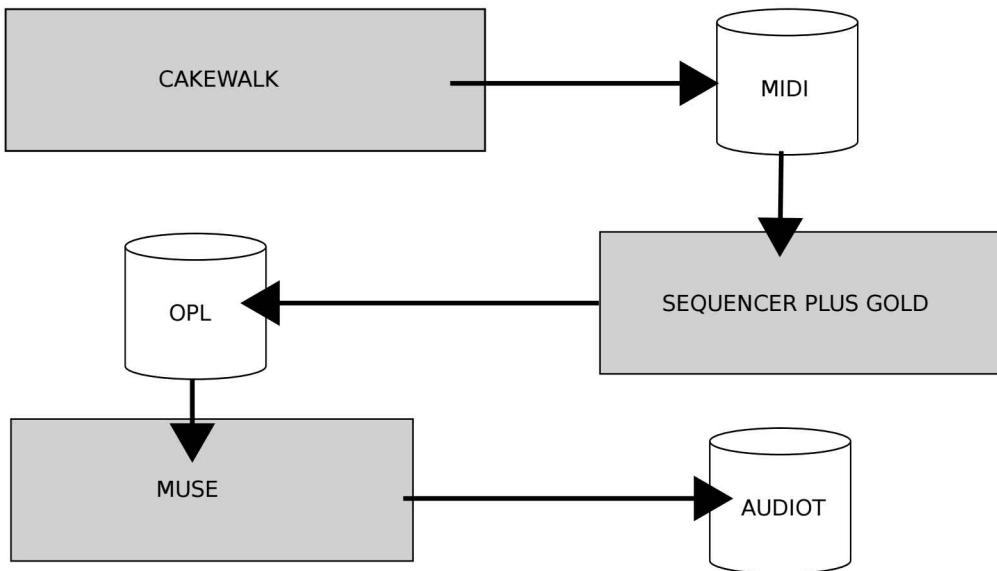
To rough out compositions, I used Cakewalk ("CW"). I had been using it for several years already and had it all set up to use the analog boxes for sound output. Having "real" sounds from those boxes helped me visualize (audiolize?) what I wanted musically. I would save the CW files in \*.mid format and load them into SPG to create the OPL instrument for each track. I built different instrument banks for the different genres of music.

**Bobby Prince - Composer**

”

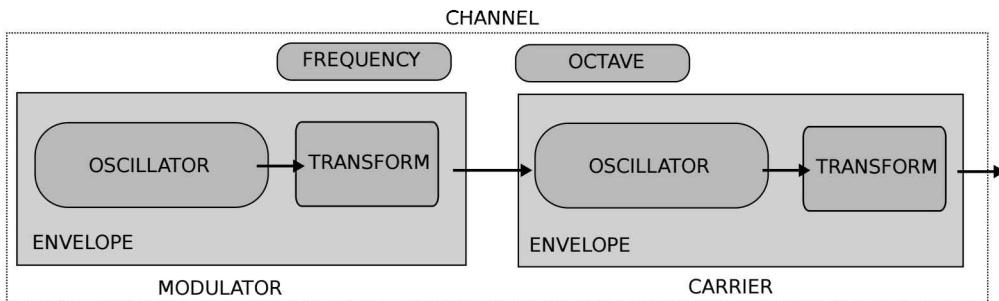


**Figure 3.14:** Sequencer Plus Gold ("SPG") by Voyetra.



**Figure 3.15:** Music pipeline as described by Bobby Prince.

What goes in the AUDIOT archive is a music format called IMF<sup>15</sup>. As it supports only the YM3812, it is tailored for the chip with zero abstraction layer. It consists of a stream of machine language commands with associated delays<sup>16</sup>. The stream pilots the nine channels in the OPL2. A channel is able to simulate an instrument and play notes thanks to two oscillators, one playing the role of a modulator and the other the role of a carrier. There are many other ways to control a channel such as envelope, frequency or octave. The way a channel is programmed is described in detail in Section 4.8.5.1, "OPL2/YM3812 Programming" on page 220.

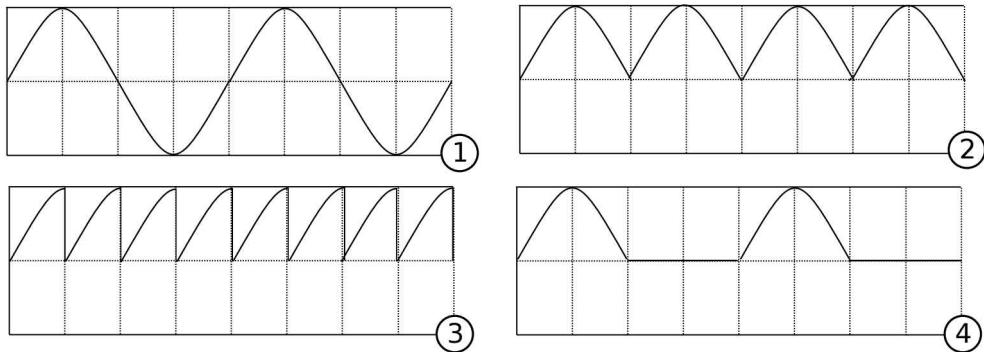


**Figure 3.16:** Architecture of a YM3812 channel.

<sup>15</sup>Id software Music File.

<sup>16</sup>IMF format is explained in detail on page 220

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



**Figure 3.17:** The four waveform transforms available.

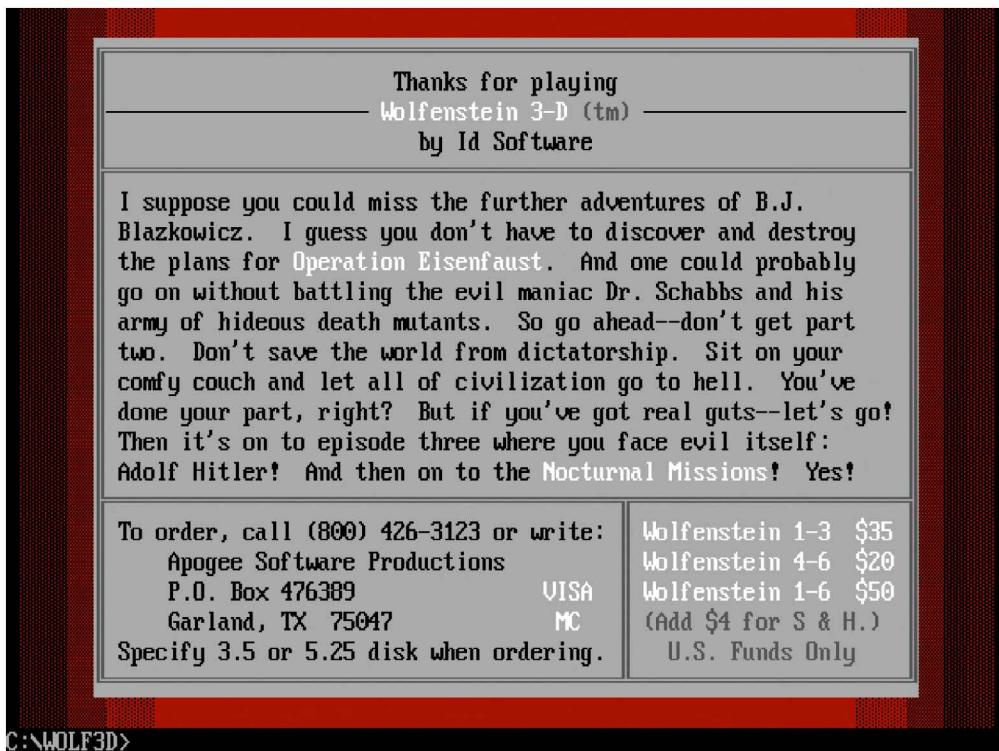
**Trivia :** In Episode 3 the music playing features a hidden Morse code message. "To Big Bad Wolf. De Little Red Riding Hood. Eliminate Hitler. Imperative. Complete mission within 24 hours. Out." The end boss of this episode is indeed Hitler in a Mech suit (see hint manual drawing on page 82).

## 3.7 Distribution

At 4am on May 5th, 1992 the first episode of the game was uploaded to Massachusetts via Software Creations BBS<sup>17</sup> server. Wolfenstein 3D was distributed as shareware; the game engine and first episode were given for free and encouraged to be copied and distributed to a maximum number of people. To receive the five other episodes, each player had to pay \$50 to id Software.

---

<sup>17</sup>Bulletin Board Systems where server allows users to connect via a console and upload/download programs.



**Figure 3.18:** Exiting the game describes how to get the full version.

In order to maximize income, it was therefore paramount to make the game easy to copy and redistribute. In 1991, Internet was still in its infancy and the best medium was the 3½-inch floppy disk. Particular attention was given to the total size of the game so it would fit on one disk. All assets combined accounted for 1,204KiB but everything was compressed to 645KiB (a 3½-inch floppy can store 720KiB). The full six episodes fit on two disks. Spear of Destiny would fit on three disks. The game shipped as follows:

```
C:\WOLF3D>dir
Directory of C:\WOLF3D\.
.
<DIR>          06-11-2016 12:16
..
<DIR>          12-11-2016 22:32
AUDIOHED.WL1      1,156 06-11-2016 12:16
AUDIOT.WL1       132,613 06-11-2016 12:16
GAMEMAPS.WL1     27,448 06-11-2016 12:16
MAPHEAD.WL1       402 06-11-2016 12:16
README.WL1        976 06-11-2016 12:16
V1                 3 06-11-2016 12:16
UGADICT.WL1      1,024 06-11-2016 12:16
UGAGRAPH.WL1    296,826 06-11-2016 12:16
UGAHEAD.WL1       462 06-11-2016 12:16
VSWAP.WL1        742,912 06-11-2016 12:16
WOLF3D.EXE       97,605 06-11-2016 12:16
11 File(s)      1,301,427 Bytes.
2 Dir(s)        262,111,744 Bytes free.
```

```
C:\WOLF3D>
```

**Figure 3.19:** All files distributed as shareware as they appear in DOS command prompt.

The files can be divided in five parts:

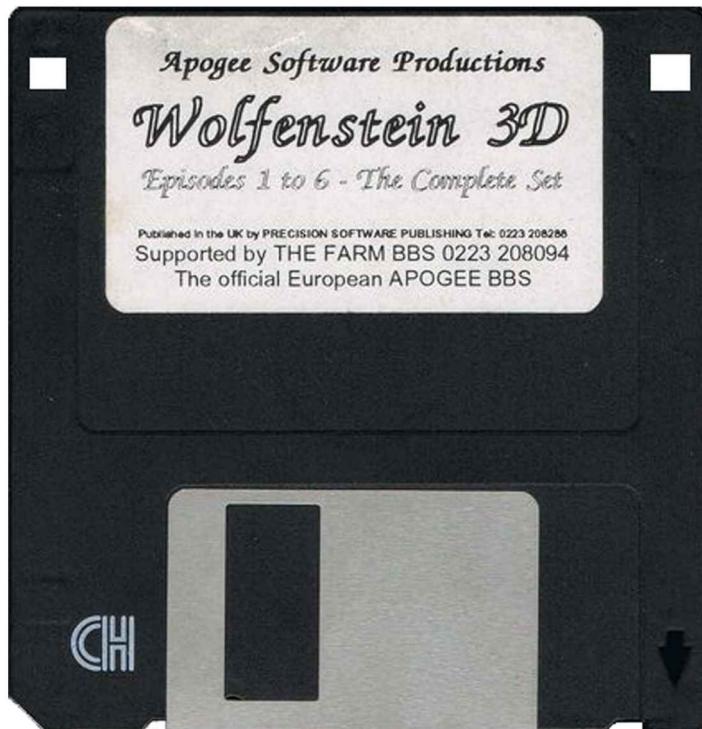
- WOLF3D.EXE: Game engine.
- VSWAP.WL1: Contains all the assets (sprites, textures, digitized sounds) needed during 3D action phases.
- Music and sound effect files used during both 3D and 2D phases:
  - AUDIOHED.WL1 : Index to payload in AUDIOT file.
  - AUDIOT.WL1: Uncompressed audio data.
- Maps:
  - MAPHEAD.WL1 : Index into GAMEMAPS file.
  - GAMEMAPS.WL1 : Compressed Map payloads.
- Pictures used during 2D menu phase:

- VGAHEAD.WL1 : Index into VGAGRAPH file.
- VGADICT.WL1 : Huffman-tree to decompress each picture.
- VGAGRAPH.WL1 : Compressed pics lumped together.

**Trivia :** The file extensions did have meaning:

- WL1: Shareware.
- WL3: Early three-episode full version (never released).
- WL6: Six-episode full version.
- WJ1: Japanese shareware.
- WJ6: Japanese full version.
- SOD: Spear of Destiny.

**Trivia :** The engine is very tiny and only uses 94KiB in total. But with 64KiB dedicated to the Signon screen (and 768B for the palette), the engine is in fact only 29KiB.



On the previous page is an image of a 3½-inch floppy widely used to share and redistribute the game. The upper left hole allows the disk reader to recognize if the disk is a low capacity (full = 720 KiB) or a high capacity (hole = 1.44 MiB version). The upper right hole features a sliding tab which when closed forbids writing to the disk, making it read-only. Once inserted in the floppy disk reader, the large metallic tab slides to the right to expose the magnetic disk.

**Trivia :** Some people (including the author) saved money by drilling holes in their 720 KiB certified floppy disks to have them recognized as 1.44 MiB. It worked!



# **Chapter 4**

## **Software**

### **4.1 Getting the Source Code**

The game engine source code was uploaded on id software's ftp<sup>1</sup> server on July 21st, 1995.

```
ftp://ftp.idsoftware.com/idstuff/source/wolfsrc.zip
```

More than twenty two years later the archive is still located at the exact same URL, a remarkable fact given the ever changing nature of the web.

### **4.2 First Contact**

Once downloaded and decompressed, the archive `wolfsrc.zip` contains another self-extracting PKZIP archive. It was a convenience back in the day, but is not practical now and is easy to deflate.

```
$unzip WOLFSRC.1
```

`cloc.pl` is a tool which looks at every file in a folder and gathers statistics about source code. It helps for getting an idea of what to expect.

---

<sup>1</sup>File Transfer Protocol.

```
$ cloc -1.64.pl WOLF3D
```

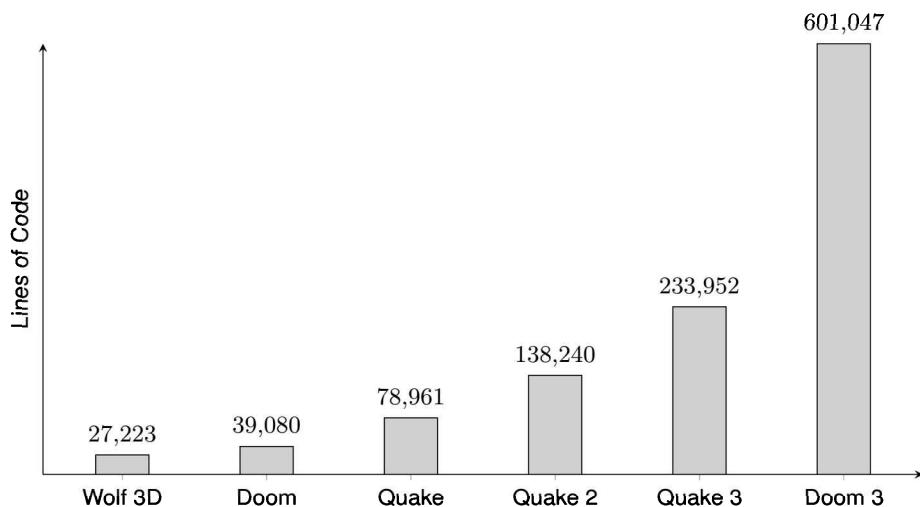
```
96 text files.  
94 unique files.  
27 files ignored.
```

Language	files	blank	comment	code
C++	26	5750	6201	21169
C/C++ Header	42	802	660	3900
Assembly	10	669	732	2150
DOS Batch	1	1	0	4
SUM:	79	7222	7593	27223

The code is 90% in C with assembly<sup>2</sup> for bottleneck optimizations and low level I/O such as video or audio<sup>3</sup>. Lines of code (SLOC) is not a meaningful metric against a single code-base but excels when it comes to extracting proportions. Wolfenstein 3D with its 27,223 SLOC is very small compared to most software. curl (a command-line tool to download url content) is 154,134 SLOC. Google's Chrome browser is 1,700,000 SLOC. Linux kernel is 15,000,000 SLOC.

<sup>2</sup>All the assembly in Wolf3D is done with TASM (a.k.a Turbo Assembler by Borland). It uses Intel notation where the destination is before the source: `instr dest source`.

<sup>3</sup>id Software would not switch to C++ until Doom 3 around 2000.



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

We didn't have spell checkers in our editors back then, and I always had poor spelling. The word "collumn" appears in the source code dozens of times. After I released the source code, one of the emails that stands out in memory read:

It's "COLUMN", you dumb FUCK!

#### **John Carmack - Programmer**

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

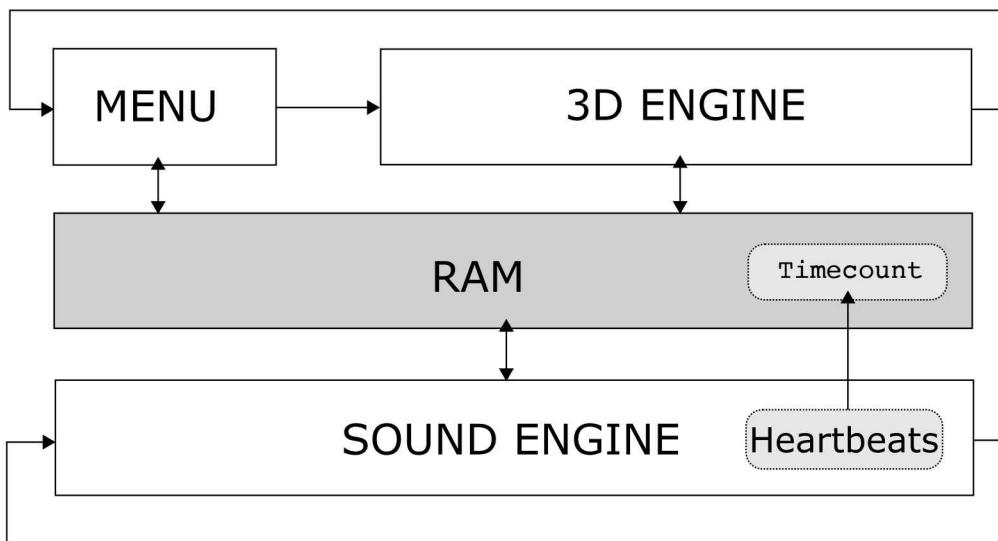
- **GOODSTUF.TXT:** Two emails from fans (an ex-POW and a Microsoft employee) demonstrating the success of the game.
- **SIGNON.OBJ:** The startup screen showing the system characteristics (RAM, EMS, XMS, Joystick, SoundCards) was linked in the binary. This weird design choice is explained later.
- **GAMEPAL.OBJ:** Game palette. Hardcoded and linked in the executable for the same reason as **SIGNON.OBJ**.
- **README:** How to build. You can also find a complete tutorial in "Let's compile like it's 1992" on [fabiensanglard.net](http://fabiensanglard.net).
- Many files resulting from a previous compilation attempt.

## 4.3 Big Picture

The game engine is divided in three blocks:

- 2D menu engine which lets the user configure the game.
- 3D game renderer where the user spends most of her time.
- Sound system which runs concurrently with either the 2D or 3D renderer.

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



*Figure 4.2: Game engine three main systems.*

### 4.3.1 Unrolled Loop

With the big picture in mind, we can dive into the code and unroll the main loop starting in `int main()`. The two renderers are regular loops but due to limitations explained later, the sound system is interrupt-driven and therefore out of `main`. Because of real mode, C types don't mean what people would expect from a 32 bit architecture.

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

The first thing the program does is check the assets available via `CheckForEpisodes()`.

```
int main (int argc, char *argv []) {  
    CheckForEpisodes();  
    Patch386 ();  
    InitGame ();  
    DemoLoop();  
}
```

Since the target machine ran in real mode, the code was compiled using 16 bit instructions only. For operations on long (32 bit), Borland used its own math library. In `Patch386` Wolfenstein 3D detects if the CPU is a 386 and patches its own code to replace Borland's integral division with instructions using 32 bit registers `eax` and `edx`.

```
    mov eax,[bp+8]  
    cdq  
    idiv [DWORD PTR bp+12]  
    mov edx,eax  
    shr edx,16
```

In `InitGame`, the engine starts up and brings up all the managers.

```
void InitGame () {  
    MM_Startup ();           // Memory manager  
  
    SignonScreen ();        // Show system configuration  
  
    VW_Startup ();           // Video Manager  
    IN_Startup ();           // Input Manager  
    PM_Startup ();           // Page Manager  
    PM_UnlockMainMem ();  
    SD_Startup ();           // Sound manager  
    CA_Startup ();           // Cache manager  
    US_Startup ();           // Font manager  
    InitDigiMap ();  
    ReadConfig ();  
    CA_CacheGrChunk(STARTFONT); // Load font  
    MM_SetLock (&grsegs[STARTFONT],true); // Lock font  
    LoadLatchMem ();         // Load picture asset to VRAM  
    BuildTables ();          // sin/cos/view loopu tables  
    SetupWalls ();           // Lookup table wall textures  
}
```

Then comes the core loop, where the 2D renderer and 3D renderer are called forever.

```

void DemoLoop() {
    StartCPMusic(INTROSONG);
    PG13(); // Show Profound Carnage screen
    while (1) {
        CA_CacheScreen (TITLEPIC);
        CA_CacheScreen (CREDITSPIC);
        DrawHighScores ();
        PlayDemo (0);
        GameLoop (); // 2D renderer (menu)
        SetupGameLevel ();
        StartMusic ();
        PM_CheckMainMem ();
        PreloadGraphics ();
        DrawLevel ();
        PlayLoop (); // 3D renderer (action)
        StopMusic ();
    }
}
Quit("Demo loop exited??");
}

```

PlayLoop contains the 3D renderer. It is pretty standard with getting inputs, update world, and render world approach.

```

void PlayLoop () {
    PollControls (); // Get player input

    MoveDoors (); // Move doors
    MovePWalls (); // Move secret wall
    for (obj = player; obj; obj = obj->next)
        DoActor (obj); // Enemies think

    ThreeDRrefresh () { // Render 3D view
        VGAClearScreen (); // Draw floor/ceiling
        WallRefresh (); // Draw walls
        DrawScaleds(); // draw scaled stuff
        DrawPlayerWeapon (); // draw weapon
        [...] // Flip framebuffer via CRT Controller
    }
    UpdateSoundLoc (); // Stereo sound loc
}

```

The sound system is started via the Sound Manager in `SDL_SetTimerSpeed`. While there

is a famous game development library called Simple DirectMedia Layer (SDL), the prefix `SDL_` has nothing to do with it. It stands for SounD Low level. (Simple DirectMedia Layer did not even exist in 1991).

The reason for interrupts is extensively explained in Chapter 4.8 "Audio and Heartbeat". In short, with an OS supporting neither process nor thread, it was the only way to have something execute concurrently with the rest of the engine.

An ISR<sup>4</sup> is installed in the Interrupt Vector Table to respond to interrupts triggered by the engine. Note how the ISR can be called at frequencies of 140Hz, 700Hz, or even 7000Hz depending on the needs of the sound system.

```
#define TickBase 70

static void SDL_SetTimerSpeed(void) {
    word rate;
    void interrupt (*isr)(void);

    if (DigiMode == PCSpeaker) && DigiPlaying) {
        rate = TickBase * 100;          // 7000 Hz
        isr = SDL_t0ExtremeAsmService;
    }
    else if (music || ((DigiMode == Dss) && DigiPlaying)) {
        rate = TickBase * 10;          // 700 Hz
        isr = SDL_t0FastAsmService;
    }
    else {
        rate = TickBase * 2;          // 140 Hz
        isr = SDL_t0SlowAsmService;
    }

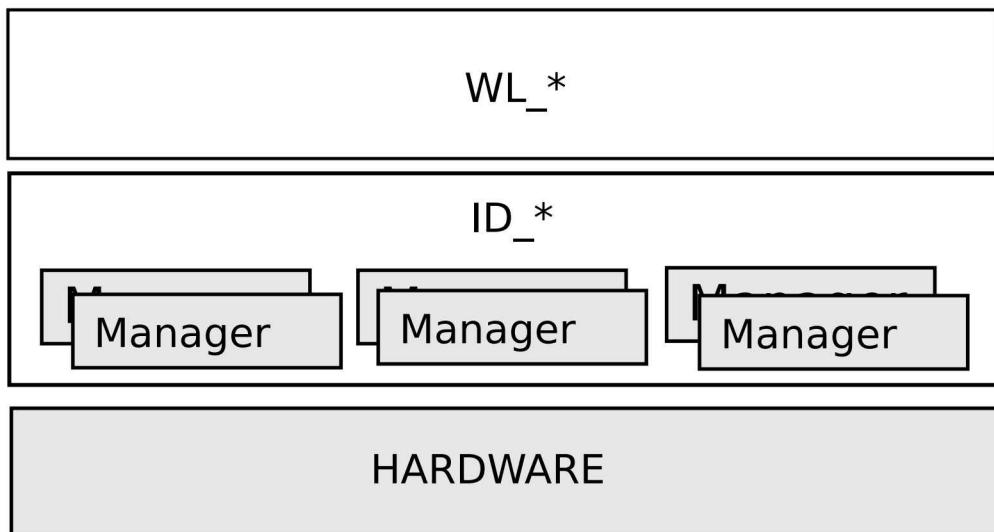
    setvect(8,isr);
    SDL_SetIntsPerSec(rate);
}
```

## 4.4 Architecture

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

---

<sup>4</sup>Interrupt Service Routine.

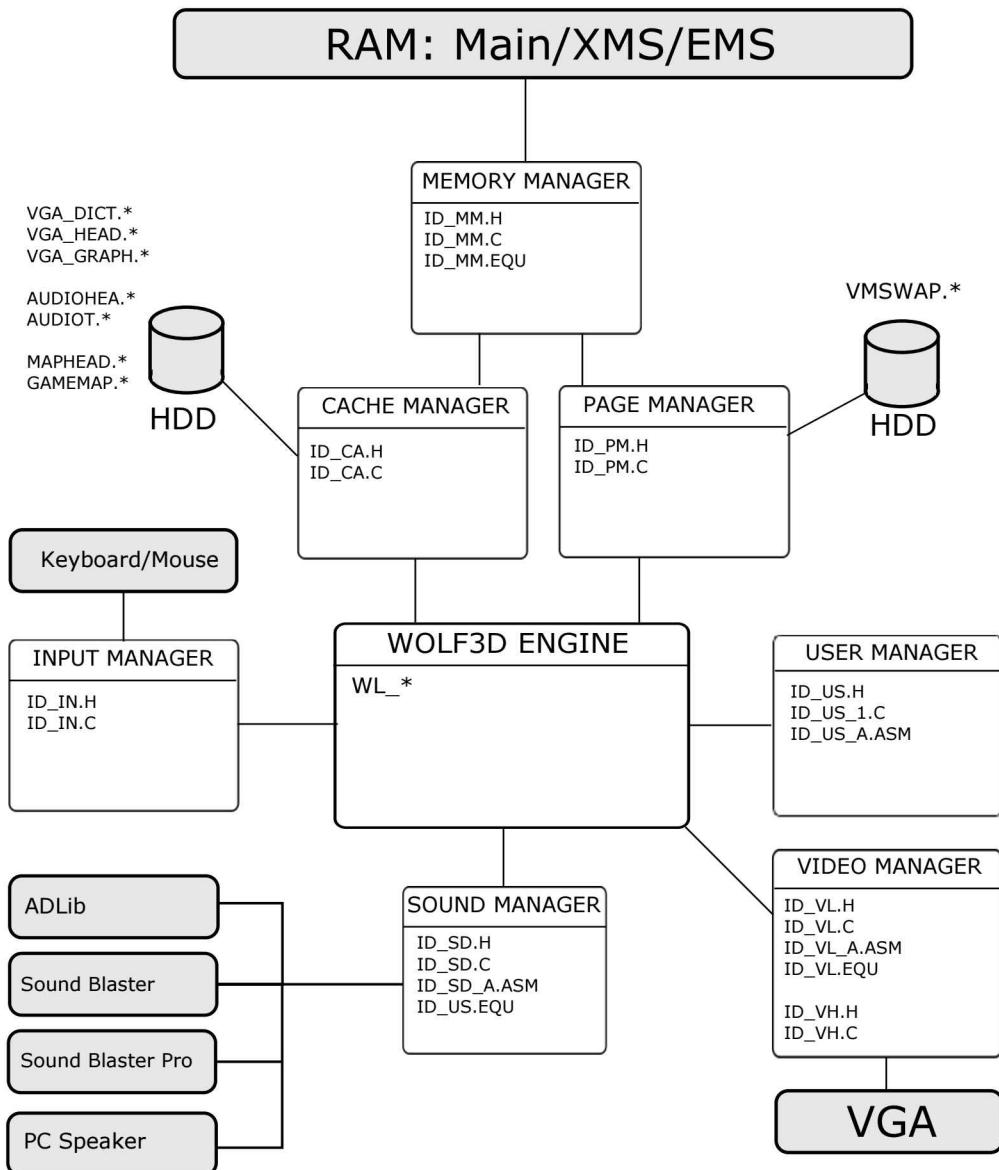


**Figure 4.3:** Wolfenstein 3D source code layers.

There are seven managers in total:

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

The WL\\_ stuff was written specifically for Wolf3D while the ID\\_ managers were reused from previous games (Hovertank 3D and Catacomb 3D) and improved for the needs of the new engine.



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

Next to the hard drives (HDD) you can see the assets packed as described in Chapter 3 Team.

### 4.4.1 Memory Manager (MM)

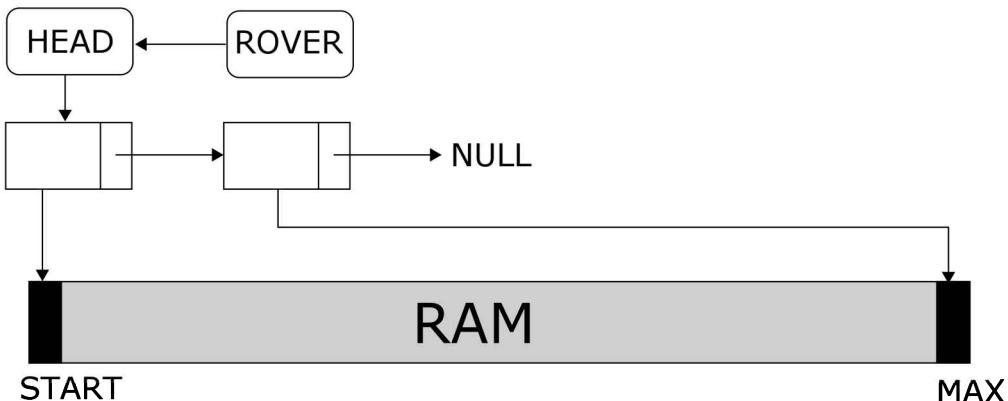
The engine does not rely on `malloc` to manage conventional memory, as this can lead to fragmented memory and no way to compact free space. It has its own memory manager made of a linked list of "blocks" keeping track of the RAM. A block points to a starting point in RAM and has a size.

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

A block can be marked with attributes:

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

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

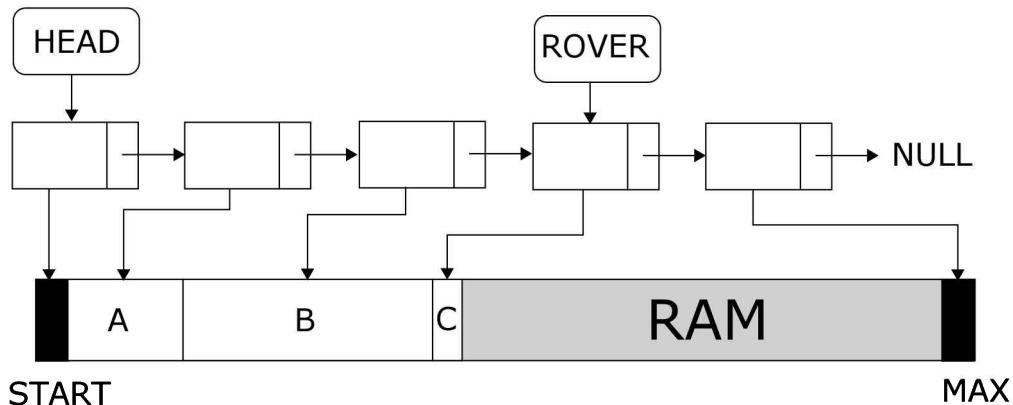


**Figure 4.5:** Initial memory manager state.

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

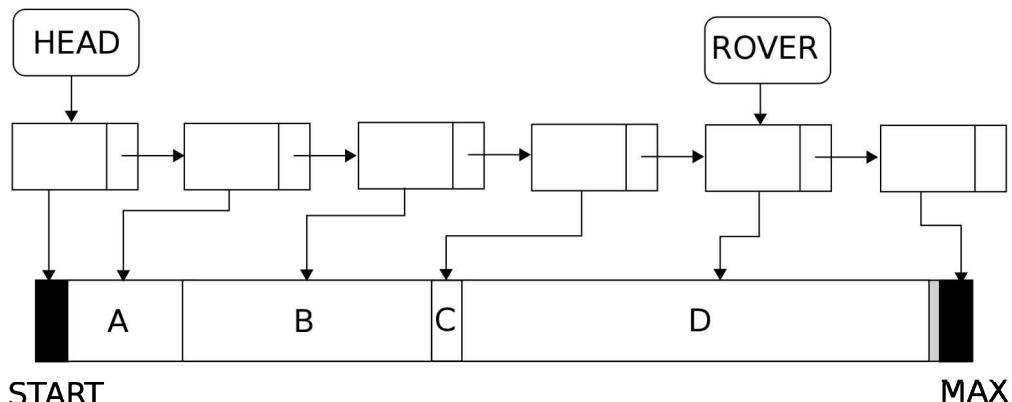
1. After rover.
2. After head.
3. Compacting and then after rover.

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



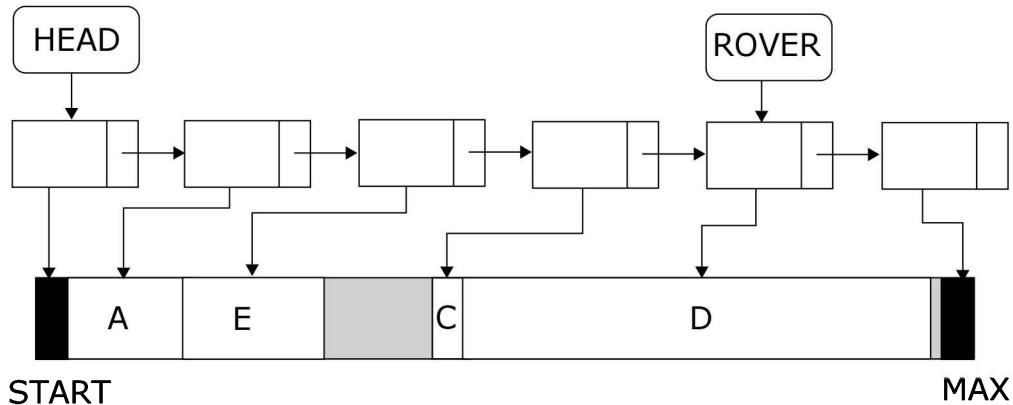
**Figure 4.6:** MM internal state after three pass 1 allocations.

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



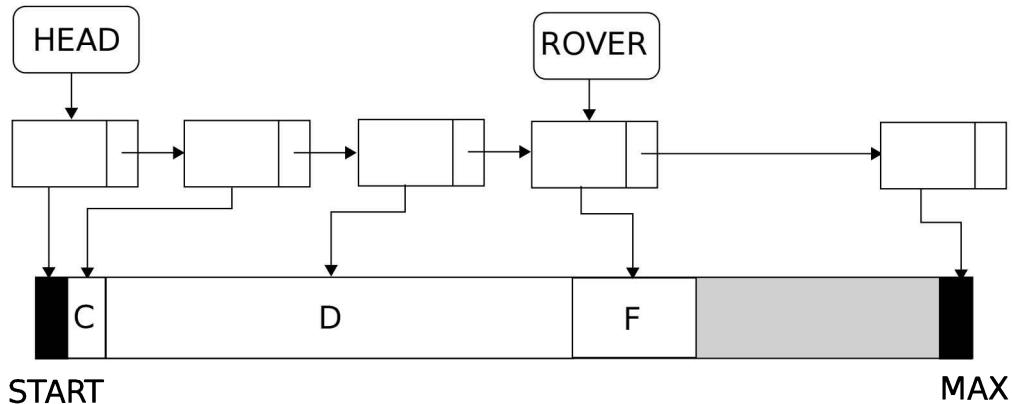
**Figure 4.7:** Pass 1 failure: Not enough RAM after the ROVER.

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



**Figure 4.8:** B was purged. E was allocated in pass 2.

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

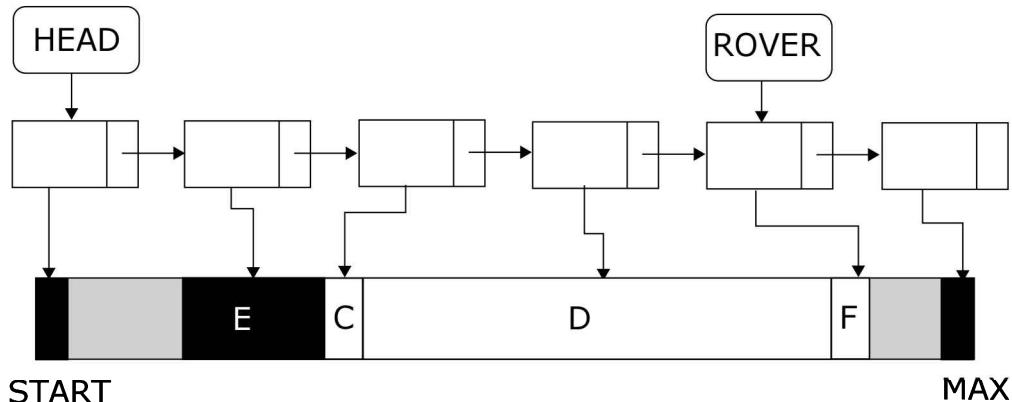


**Figure 4.9:** A and E were purged. C and D compacted. F allocated in pass3.

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

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

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



**Figure 4.10:** E is locked and cannot be compacted.

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

A dedicated memory manager was probably justified for Wolf, but they are a huge source of bugs, and I urge people not to do it!

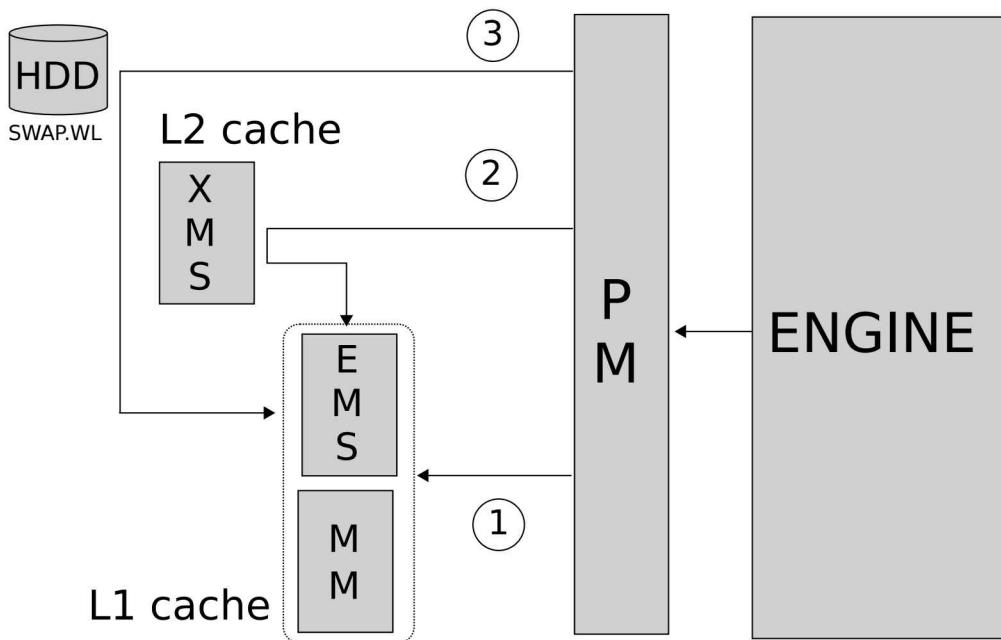
**John Carmack - Programmer**

#### 4.4.2 Page Manager (PM)

The Page Manager is dedicated to the 3D engine. Its task is to make sure assets such as wall textures, sprites, and sound effects stored on HDD are available in RAM for the CPU to use. Jason Blochowiak seems to have been the main author and his previous experience with Unix clearly influenced the design of this component. It is built around the concept of paging and swapping.

Instead of using a memory address to identify a page like Unix, an asset ID is used. These IDs are generated by IGRAB-ED. Each asset consumes a full "page". Like Unix, all pages have the same size, 4KiB. When the engine needs a resource, it requests a page with the resource ID from the Page Manager. All types of RAM (Conventional, EMS, and XMS) are leveraged but there is a hierarchy.

Originally all assets for 3D sequences are on HDD in file VSWAP.WL1. When a request for an asset is received, the L1 cache (comprised of Conventional and EMS RAM) is looked up first ①. In case of a miss, the L2 cache is consulted ②. If the page is found there, it is transferred to L1. If the page is still not found in L2, it is loaded from the HDD directly to L1 ③. L2 is only written to when a page is evicted from L1. Every time a page is accessed, it is also tagged with the current frame number. This tag is used to enforce the eviction policy.



The architecture of the Page Manager is interesting since it treats XMS as a last resort L2 cache level while EMS RAM is used like conventional memory. This is because EMS driven RAM is several times faster than XMS driven RAM and almost as fast as conventional memory. This topic is detailed in Annex B on page 293

In order to minimize the cost of page misses, the engine preloads the page cache before a level start. The user experiences this as the "Get Psyched" screen:



**Figure 4.11:** The "thermometer" showing the Page Manager precaching process.

The precaching mechanism is not particularly clever: it loads as many pages from the swap file as possible. It doesn't try to look at what is actually used in the level but instead loads assets in the order they are stored in VSWAP file on the HDD. On a machine with low memory (less than 1MB), the eviction policy (LRU) stabilizes the cache after a few minutes in a level.

This design has a small yet annoying flaw. On a low memory machine, a cache miss will occur when the player opens the last door of a level and is about to find herself confronted with the heavy-power final boss. This enemy has never been seen before, and therefore is not in the cache of the Page Manager. This incurs the worst possible case of cache miss: a long access to the hard-drive is required, leading to lag and often resulting in an undeserved (and humiliating) death.

The size of the swap file varies depending on the version of the game being played: VSWAP.WL1 (shareware) is 742KiB while VSWAP.WL6 (full version) is 1,500KiB. In both cases a machine with 2MiB of RAM on top of the factory issued 1MiB is enough to have

all assets loaded during precaching.

**Thrashing:** When the system has to evict pages but ends up reloading the same resource during the same frame, "thrashing" occurs. The HDD is put to heavy use, and framerate drops. Trashing can happen if too many different resources are visible on the screen. In order to help designers balance their creativity with the need for a decent framerate, the engine detects trashing and flashes the screen border red when running in dev-mode.

**Sounds are special:** Because sound cards are fed via a system of interrupts, the sound manager cannot recover from a page miss. Therefore, all sound resources are loaded first (they are located at the start of SWAP.WL1) and only in conventional memory.

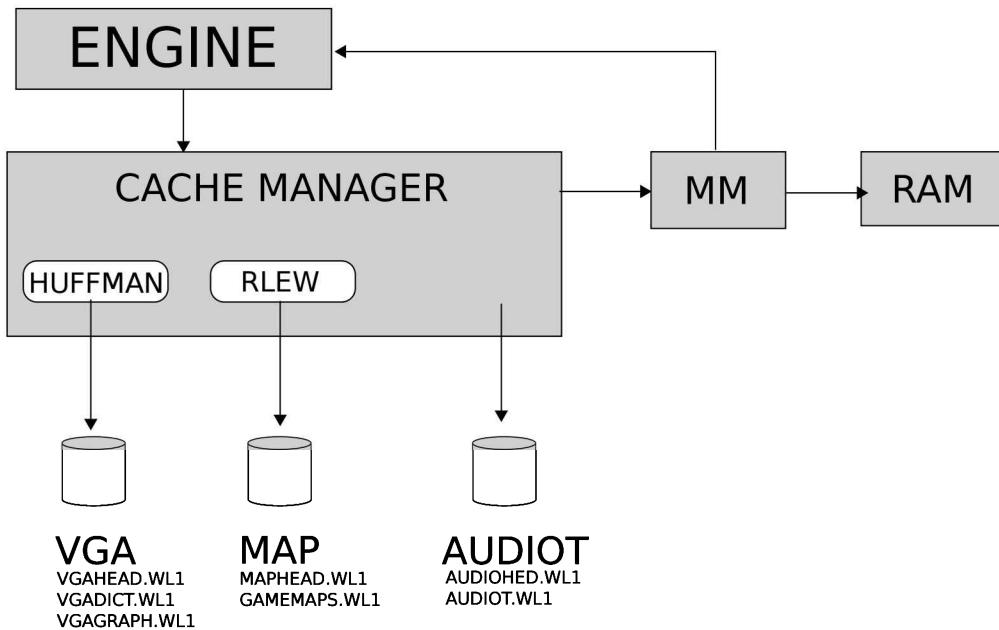
#### 4.4.3 Video Manager (VL & VH)

The video manager features two parts:

- A low level dedicated to hardware VGA register manipulation.
- A high level dedicated to 2D menu drawings.

#### 4.4.4 Cache Manager (CA)

The cache manager is a small but critical component. It loads and decompresses maps, 2D graphics, and audio resources stored on the filesystem and makes them available in RAM. Assets of each kind are stored in two files. A header file contains the offset to allow translation from asset ID to byte offset in the data file.

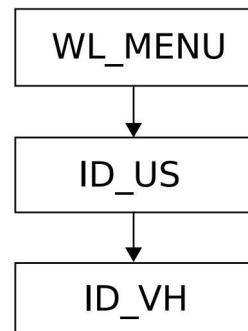


**Note:** All resources are compressed. In the case of maps and audio, compression is hard-coded in the engine. For graphics, however, a third file (DICT) contains the compression dictionary to decompress each asset. The Cache Manager handles decompression transparently.

**Trivia :** Is there a typo in the filename **AUDIOHED.WL** ? The correct spelling would be **AUDIOHEAD.WL1**. Where is the A? This is in fact a limitation of the operating system. DOS only allows 8.3 filenames (at most eight characters followed by a dot, then at most three characters for the extension).

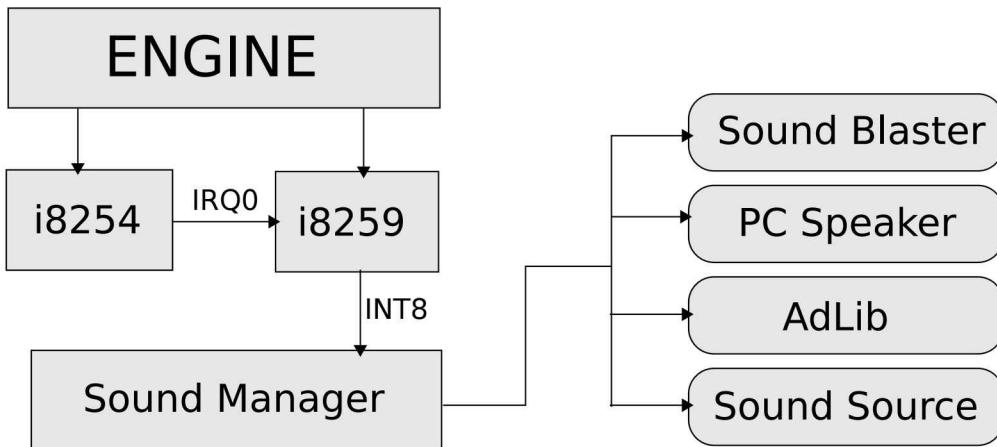
#### 4.4.5 User Manager (US)

The User Manager is largely based on the Catacomb 3D code and was written by Jason Blochowiak. The copy/paste is very visible since 90% of the functions declared in the header (**ID\_US.H**) are not actually implemented in **ID\_US.C**. It is a poorly named manager as it mostly takes care of text layout. When a **WL\_\*** high level routine needs to draw a string, it is passed to **US\_Print** which does all measurement (e.g. Draw string centered) and then passes this information to the Video Manager (**VW\_DrawPropString**), which takes care of rendition.



#### 4.4.6 Sound Manager (SD)

The Sound Manager abstracts interaction with all four sound systems supported: PC Speaker, AdLib, Sound Blaster, and Disney Sound Source. It is a beast of its own since it doesn't run inside the engine. Instead it is called via IRQ at a much higher frequency than the engine (the engine runs at a maximum 70Hz, while the sound manager ranges from 140Hz to 7000Hz). It must run quickly and is therefore not only written in assembly, but its assets are also privileged when it comes to memory allocation. All of its assets are loaded in conventional memory to avoid a cache miss in the page manager.



**Figure 4.12:** Sound system architecture.

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

#### 4.4.7 Input Manager (IN)

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

### 4.5 Startup

As the game engine starts, it must deal with the difficulties described in Chapter 2 Hardware. This is where things become really interesting.

### 4.5.1 Signon

The first (mild) issue to deal with was the heterogeneous ecosystem of PCs on the market. With different drivers loaded and different sound cards, the engine has to figure out how much RAM is installed and whether or not it will be able to run. If not, it has to let the user know what the problem is. That was important as id Software was a small team, without the resources to help troubleshoot customer issues.

That is what the "signon" screen is for: self diagnostics.



**Figure 4.13:** Signon screen.

Besides showing recognized devices such as mouse, joystick, and sound cards, the signon screen's most important metric is labeled "MAIN". Due to the architecture described in Chapter 2 Hardware, a DOS program has only 640KiB of RAM available. Each driver loaded by the user takes away from these 640KiB. If DOS cannot load the executable in RAM, the user will see the following error message:

Out of memory

The signon screen shows that Wolf3D needs at least 320KiB of Conventional RAM. John Romero wrote a release note to help people understand what was going on and avoid angry calls. You can read it in the annex under "The 640KB Barrier".

At the time signon is displayed, the only loaded manager is the Memory Manager. There is not even a filesystem for the engine to access yet. That is why the palette and the signon screen are compiled within the executable. This is done so they are loaded in RAM by the operating system (DOS) loader. All the engine does is load the palette into the VGA, copy the signon bitmap from RAM to VRAM, and fill the "thermometer" blocks with green or yellow based on what was detected.

```
extern byte far gamepal;    // content of GAMEPAL.OBJ
extern char far introscn;   // content of SIGNON.OBJ

void SignonScreen (void)
{
    unsigned segstart, seglength;

    VL_SetVGAPlaneMode ();
    VL_TestPaletteSet ();
    VL_SetPalette (&gamepal);

    VW_SetScreen(0x8000,0);
    VL_MungePic (&introscn,320,200);
    VL_MemToScreen (&introscn,320,200,0,0);
    VW_SetScreen(0,0);

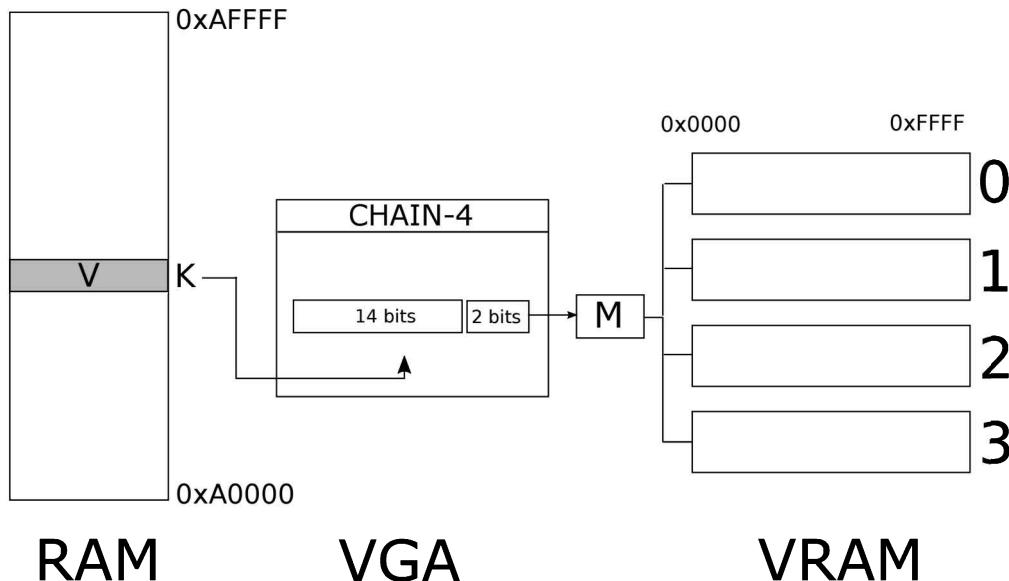
    // reclaim the memory from the linked signon screen
    segstart = FP_SEG(&introscn);
    seglength = 64000/16;
    if (FP_OFF(&introscn)){
        segstart++;
        seglength--;
    }
    MML_UseSpace (segstart,seglength);
}
```

After that screen, the `introscn` variable (using 320x200 bytes = 64,000 bytes) is unloaded from RAM to make more room for runtime.

### 4.5.2 Solving the VGA Problem

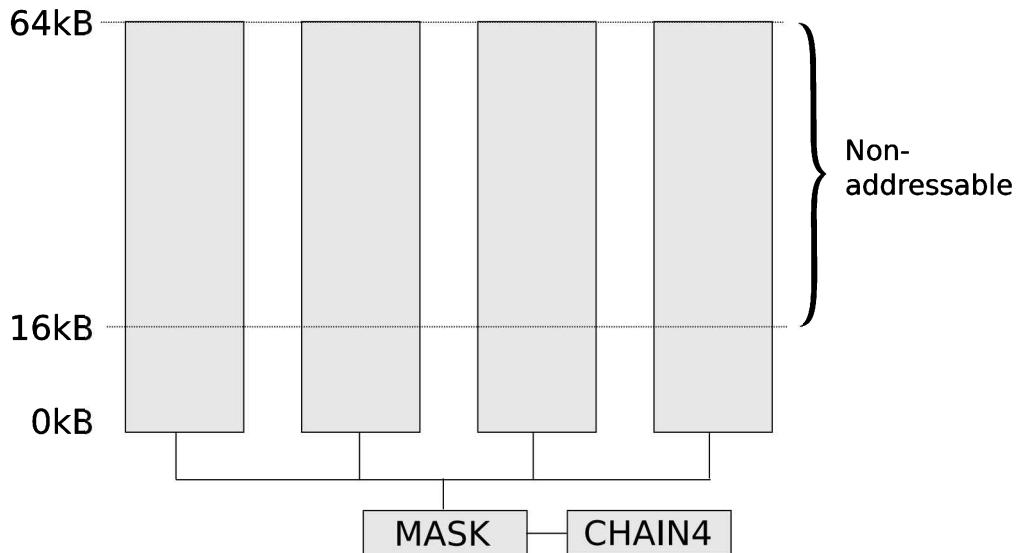
Chapter 2 Hardware left us with an unresolved issue: all the VGA modes lack double buffering capability.

The most appealing mode (13h) offers a single framebuffer at a resolution of 320x200 non-square pixels with 256 indexed colors. The Chain-4 chipset in the VGA circuitry automatically maps the RAM starting at A0000h to the four VRAM banks.



**Figure 4.14:** Chain-4 chipset between RAM and VRAM routes I/Os operations.

This mapping system is also called "chaining". Because 2 bits out of the address are used to route a write/read operation to a bank, only 14 bits are used for the actual offset in the bank. Since 14 bits can only address 16384 values, this system results in 75% of VRAM left unusable.



**Figure 4.15:** Chain-4 fakes one continuous VRAM bank but wastes 75% of the VRAM.

The waste is really the fault of the Chain-4 chip. However it turns out it is possible to disable this. The technique was popularized by Michael Abrash in Dr. Dobb's Journal of July 1991. In his article he described what he coined Mode-X. An undocumented sequence of operations disabling Chain-4 allows for a resolution of 320x240 square pixels (since the ratio is 4:3) and full access to the 256KiB of RAM.

Wolfenstein 3D does things slightly differently. It disables Chain-4 but keeps resolution at 320x200. This mode was coined Mode-Y one year later<sup>5</sup>.

There are two reasons the team did not use 320x240 square pixels. Despite its advantage for artists, a screen using Mode-Y is  $320 \times 200 = 64,000$  pixels, which represents 17% fewer pixels compared to Mode-X with  $320 \times 240 = 76,800$  pixels. The engine already struggled to reach an acceptable framerate, so Mode-X was simply too many pixels per frame. It would have also been inconvenient to artists since Deluxe Paint ran in Mode 13h which has non-square pixels. This would have created an awkward pipeline where assets would have been created and rendered at different resolutions.

---

<sup>5</sup>rec.games.programmer, February 10th, 1992

```

void VL_SetVGAPlaneMode (void) {
    // Call 19th interrupt vector with value 0x13
    // (Ask the BIOS to setup the VGA in mode 13h)
    asm mov ax,0x13
    asm int 0x10

    // Unchain (called deplane in the engine)
    VL_DePlaneVGA ();
    VGAMAPMASK(15);
    VL_SetLineWidth (40);
}

```

The magic happens in function VL\_DePlaneVGA where VGA registers are manipulated to tweak what the BIOS had setup in Mode 13h.

```

#define SC_INDEX          0x03c4
#define SC_DATA           0x03c5

#define CRTC_INDEX        0x03d4
#define CRTC_DATA          0x03d5

#define MEMORY_MODE       0x04
#define CRTC_UNDERLINE    0x14
#define CRTC_MODE          0x17

void VL_DePlaneVGA() {
    // Change how VRAM is written (Disable Chain-4)
    outp(SC_INDEX, MEMORY_MODE);
    outp(SC_DATA, (inp(SC_DATA)&~8));

    // Clear all four banks since the bios only cleared
    // the first 16K of each banks when setting up mode 13h.
    VL_ClearVideo (0);

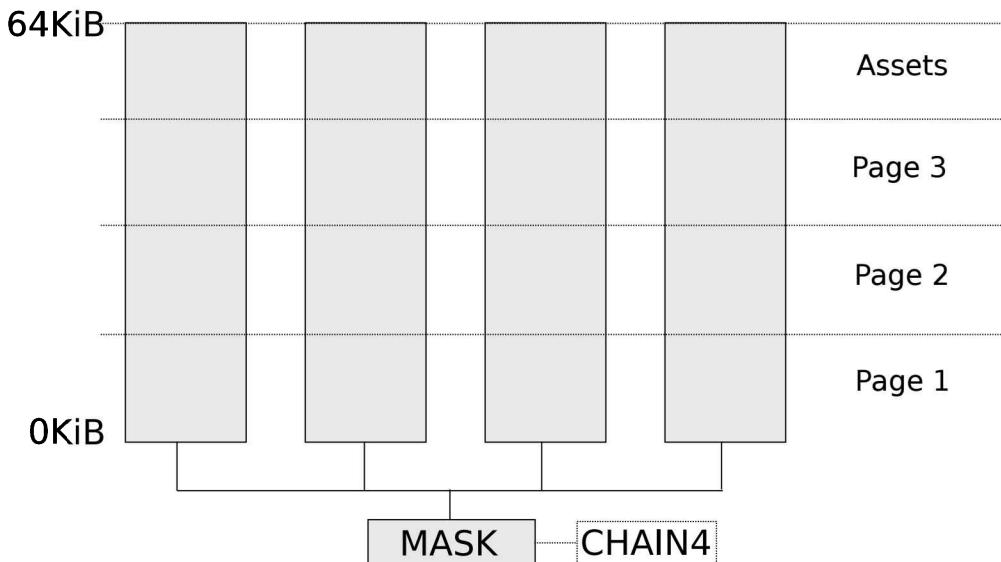
    // Change how VRAM is read by the CRTC
    // Addressing mode is selected via CRTC_MODE register.
    outp(CRTC_INDEX, CRTC_UNDERLINE);
    outp(CRTC_DATA, 0x00);
    // CRTC addressing mode set to byte.
    outp(CRTC_INDEX, CRTC_MODE);
    outp(CRTC_DATA, 0xa3);
}

```

The VGA registers of the Sequence Controller and CRT Controller are setup to divide the

256KiB VRAM into four parts.

- 64,000 bytes for Framebuffer 0
- 64,000 bytes for Framebuffer 1
- 64,000 bytes for Framebuffer 2
- 70,144 bytes for Graphic assets



But the engine is not done yet. Tweaking Mode 13h into Mode-Y solves one big problem but introduces two smaller ones: one regarding speed and one regarding correctness.

Let's look at speed first. With Chain-4 out of the picture the developer is now in charge of selecting the bank to write to. This can easily be done with a simple function.

```
#define SC_MAPMASK 0x02

void selectPlan(char plane) {
    outp(SC_INDEX, SC_MAPMASK);
    outp(SC_DATA, 1 << plane);
}
```

Now the code sample from the hardware chapter on page 53 to clear the screen only adds a modulo.

```

void CleanScreen(y, color) {
    for(int y=0 ; y < 200 ; y++) {
        for(int x=0; x < 320 ; x++) {
            selectPlan(x % 4);
            writePixel(x, y, color);
        }
    }
}

```

The code looks innocuous, but as simple as it is, it cannot run at more than a few frames per second<sup>6</sup>. The problem is we replaced something done in hardware with something done in software. The `outp` instructions are simply too slow.

The solution is to change how we draw to the screen. Instead of drawing horizontally first we need to draw vertically first in order to minimize the number of bank switches.

```

void CleanScreen(y, color) {
    for(int x=0; x < 320 ; x++) {
        selectPlan(x % 4);
        for(int y=0 ; y < 200 ; y++) {
            writePixel(x, y, color);
        }
    }
}

```

This code runs twice as fast<sup>7</sup> since just 200 slow `outp` instructions are used<sup>8</sup>.

This speed consideration has a fundamental impact on the engine: to draw anything fast with the VGA, it has to be drawn vertically. Everything in the engine is drawn this way: walls, sprites, menus. The ramifications of this hardware constraint are felt all the way down to how assets are stored in RAM: rotated 90 degrees and woven to match the VGA bank layout. This is described in detail in section 4.7.7 "Drawing Sprites".

The second issue introduced by Mode-Y is about correctness. With three pages available, the engine draws in page 1, then page 2, then page 3, and then goes back to page 1. This solves tearing and allows the engine to never block on vsync since it always has a valid framebuffer to draw to. Changing pages is done by instructing the CRT Controller that it should scan a framebuffer at a different offset after the next vsync.

The CRT Controller scan offset is a 16 bit value which is updated with a little bit of assem-

---

<sup>6</sup>5 frames per seconds on a 386DX-40 with Cirrus Logic VGA card.

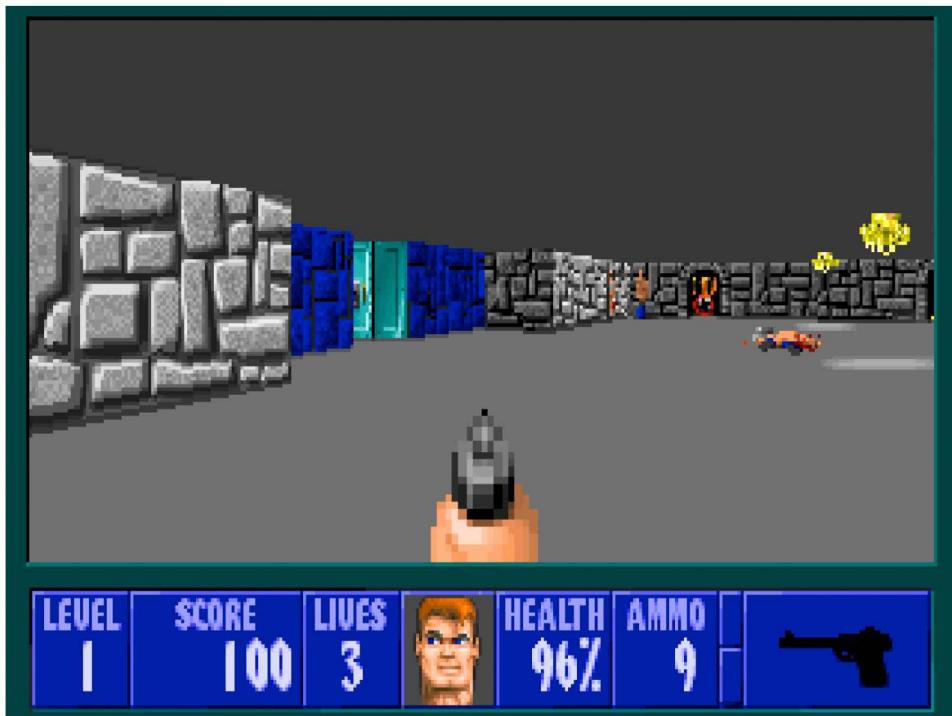
<sup>7</sup>10 frames per seconds on a 386DX-40 with Cirrus Logic VGA card.

<sup>8</sup>To reach 70 frames per second is possible by using fewer instructions thanks to `REP STOSW` instruction.

bly writing to a VGA register: first the high byte and then the low byte as follows.

```
asm mov cx, startScanOffset  
  
asm mov dx,0x3d4 ; 3d4h is the CRTC register  
  
asm mov al,0x0c ; Tell the CRTC we want to update  
asm out dx,al ; the start address high register  
asm inc dx  
asm mov al,ch  
asm out dx,al ; set the high byte  
  
asm mov al,0xd0 ; Tell the CRTC we want to update  
asm out dx,al ; the start address low register  
asm inc dx  
asm mov al,cl  
asm out dx,al ; set the low byte
```

This code looks like it would work, but there is a major flaw with it. If you were to run it, every once in a while the expected screen shown below...



...would instead appear distorted:



This glitch shows both misalignment and parts of two pages. This problem has to do with atomicity. The CRTC starting address is a 16 bit value but the `out` instruction can only write 8 bits at a time. If the pages are setup one after another like this.

0x0000      0x3E80      0x7000

Page 1	Page 2	Page 3	
--------	--------	--------	--

In VRAM, Page 1 is at 0x0000, page 2 at 0x3E80 and page 3 at 0x7000. Instructing the CRTC to use page 2 instead of page 1 requires updating the high byte 0x00 to 0x3E and the low byte 0x00 to 0x80. Since updates are not atomic, poor timing could result in the CRTC picking up a value of 0x3E00 instead of 0x3E80:

```

asm    mov    cx, startScanOffset

asm    mov    dx,0x3d4      ; 3d4h  is the CRTC register

asm    mov    al,0xc       ; Tell the CRTC we want to update
asm    out    dx,al        ; the start address high register
asm    inc    dx
asm    mov    al,ch
asm    out    dx,al        ; set the high byte

;***** CRTC SCAN START HERE !!!!!!! *****
;***** AND SHOWS 2 PARTIAL FRAMEBUFFERS *****

asm    mov    al,0xd       ; Tell the CRTC we want to update
asm    out    dx,al        ; start address low register
asm    inc    dx
asm    mov    al,cl
asm    out    dx,al        ; set the low byte

```

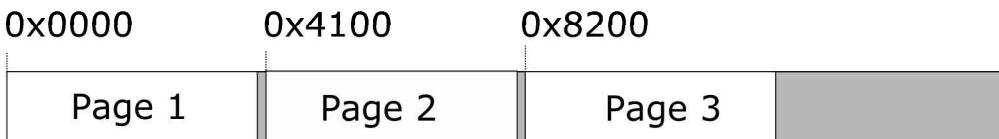
How do you update atomically a 2 byte value with a 1 byte operation? Take a look at how Wolfenstein sets up its pages:

```

#define SCREENBWIDTH     80
...
#define SCREENSIZE        (SCREENBWIDTH*208)
#define PAGE1START        0
#define PAGE2START        (SCREENSIZE)
#define PAGE3START        (SCREENSIZE*2u)
#define FREESTART         (SCREENSIZE*3u)

```

Notice how it uses a value of 208 for the height of a framebuffer. At first, this doesn't make sense as the screen is 200 pixels tall. I thought it was a typo (after all 0 and 8 are visually close) but this was done intentionally. The trick here is to use a little bit of padding after each page so the addresses only differ by their high byte value.

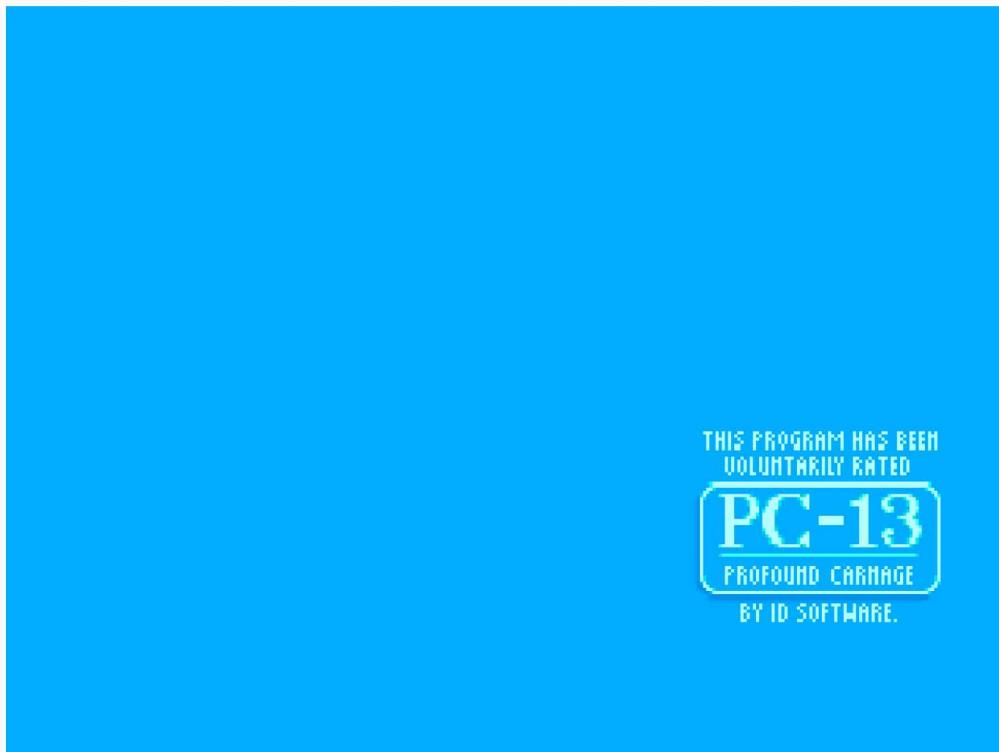


**Figure 4.16:** A little padding between pages makes all starting address a multiple of 256.

Now page 0 is at 0x0000, page 1 at 0x4100 and page 2 at 0x8200. Moving from any page to another requires updating only the high 8 bits, which makes flipping buffer an atomic operation.

### 4.5.3 Profound Carnage

After the signon screen comes the "rating" screen. There were no official ratings for video games in 1991 since the ESRB<sup>9</sup> would not be established until 1994 in response to criticisms of excessive violence (a.k.a Doom) or sexual content. The team came up with their own self-proclaimed PC-13: the now legendary "Profound Carnage-13".



**Figure 4.17:** This program has been rated PC "Profound Carnage" - 13.

---

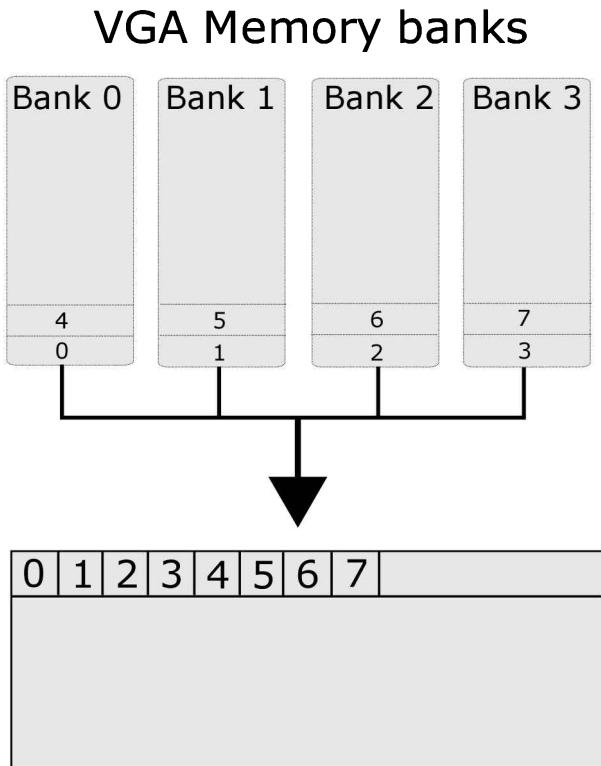
<sup>9</sup>Entertainment Software Rating Board, the organization in charge of assigning age and content ratings.

## 4.6 Menu Phase: 2D Renderer

With the VGA up and running and a robust triple buffering system ready to operate, the game finally starts. The player enters the 2D renderer which displays menus to setup the 3D game. This is pretty simple yet features a nice VGA trick to turn the four banks' weak and cumbersome design into a strength.



Notice how the background of the menu screen is full red. This is a lot of pixels to write ( $320 \times 200 = 64,000$ ). With control over the bank mask, it is possible to write up to four pixels to the VRAM with only one write operation to the RAM. In Figure 4.18, you can see how pixels 0, 1, 2, and 3 are in different banks but at the same address (0x0000). By configuring the bank mask to 8+4+2+1 (15), it is possible to write to all banks simultaneously (e.g. write to 0x0000 writes pixels 0,1,2, and 3).



### Result on screen

**Figure 4.18:** How bytes are read from the VRAM banks and displayed onto the screen.

In order to clear the screen to red before drawing the menu, the 2D engine only performs  $320 \times 200 / 4 = 16,000$  writes instead of 64000. In the drawing above, pixels 0,1,2, and 3 are written with one write operation. Then pixels 4,5,6, and 7 with another write, and so on.

Even better, since the RAM can be written with 16 bit register, 8 pixels can be written in one RAM operation. A full screen can be cleared with  $320 \times 200 / 8 = 8,000$  writes.

```

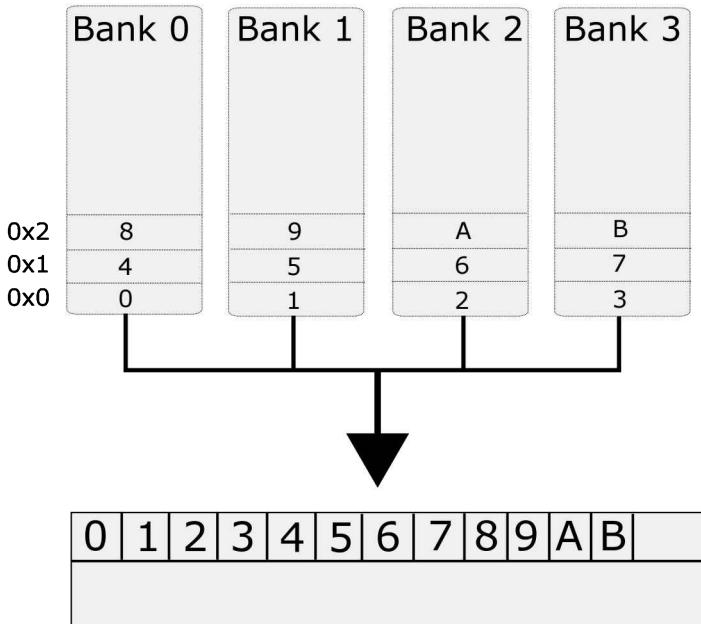
word far *VGA = (word far*)0xA0000000L;
word color = 0x0000;

/* select all planes */
outp(SC_INDEX, MAP_MASK);
outp(SC_DATA, 15);

for (int y=0 ; y < 200 ; y++) {
    for (int x = 0 ; x < 40 ; x++) {
        VGA[(y<<3)+(y<<5)+x]=color; // y*40 + x
    }
}

```

However, there is a limitation to this trick: only bytes at the same address in a bank can be written simultaneously. Pixel alignment with banks has to be carefully considered.



Pixels 0, 1, 2, 3 can be written in one write.

Pixels 3 and 4 need two writes.

Pixels 3, 4, 5, 6, 7, 8 need three writes<sup>10</sup>.

The rest of the 2D renderer is pretty straight forward. It uses the User Manager exten-

<sup>10</sup>How the mask allows for writing multiple pixels at once is extensively described on page 189.

sively to render text and the Cache Manager to retrieve assets from HDD to RAM. Note that assets are called "pic" as opposed to "sprites" in the 3D renderer. Pictures are all Huffman-compressed (VGADICT, VGAHEAD, VGAGRAPH). Menus are stored in an array of struct. Here is the code to draw the "Main Menu" shown previously.

```
#define STR_NG "New Game"
#define STR_SD "Sound"
#define STR_CL "Control"
#define STR_LG "Load Game"
#define STR_SG "Save Game"
#define STR_CV "Change View"

void CP_NewGame(void);
void CP_Sound(void);
void CP_Control(void);
void CP_LoadGame(void);
void CP_SaveGame(void);
void CP_ChangeView(void);

CP_itemtype far
MainMenu []=
{
    {1,STR_NG,CP_NewGame},
    {1,STR_SD,CP_Sound},
    {1,STR_CL,CP_Control},
    {1,STR_LG,CP_LoadGame},
    {0,STR_SG,CP_SaveGame},
    {1,STR_CV,CP_ChangeView},
    ...
}
```

```
void DrawMainMenu(void)
{
    ClearMScreen();                                // Turn screen to red
    VWB_DrawPic(112,184,C_MOUSEBACKPIC);          // Bottom image
    DrawStripes(10);                               // Draw black strip
    VWB_DrawPic(84,0,C_OPTIONSPIC);               // OPTION image
    DrawWindow(MENU_X-8,MENU_Y-3,MENU_W,MENU_H,BKGDCOLOR);
    [...]
    DrawMenu(&MainItems,&MainMenu[0]);
    VW_UpdateScreen();
}
```

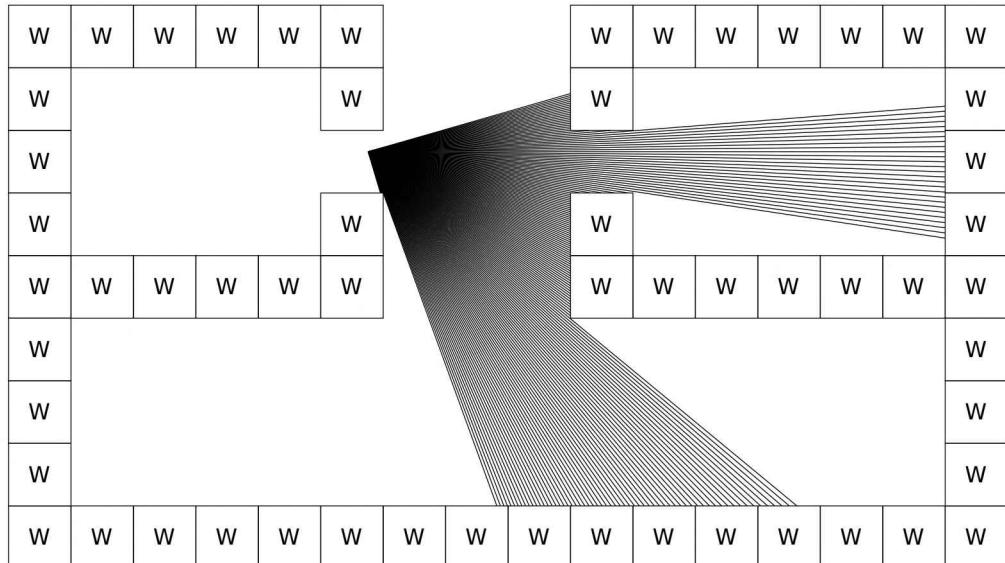
Notice how C\_MOUSEBACKPIC and C\_OPTIONSPIC are macro defined in the files generated by IGRAB-ed, the assets compiler.

## 4.7 Action Phase: 3D Renderer

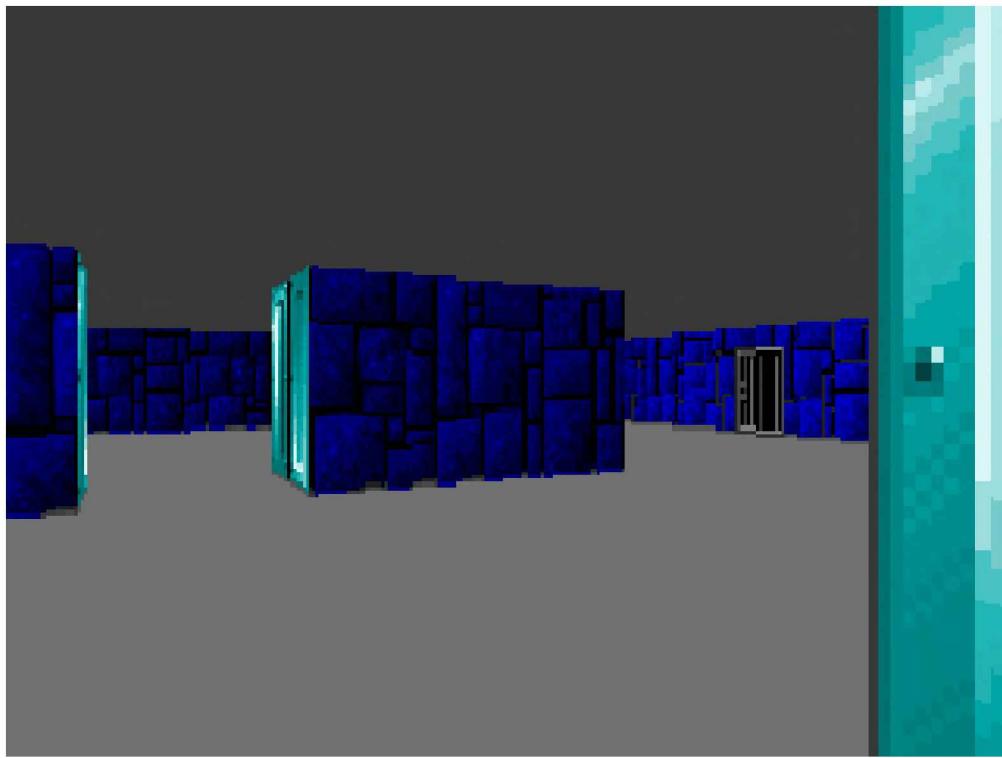
After the player is done setting up the game via the 2D renderer, it is time for the 3D engine to shine. The 3D renderer is based on a simple yet powerful technology called raycasting. The core idea is to cast a ray for each column of pixels visible on the screen. Based on the distance  $d$  from the point of view to where the ray hits a wall, a height  $h$  can be calculated ( $X$  is a simple scaling factor):

$$h = \frac{X}{d}$$

Even for a complex scene involving multiple doors and rooms, this method can deliver fast intersection calculations.



**Figure 4.19:** Casting 320 rays (one for each column) for a screen of resolution 320x200



**Figure 4.20:** Rendition of the 320 column of pixels (with texturing).

#### 4.7.1 Life of a Frame

As we saw when we unrolled the loop, the action scene is made of frames in a loop, which in pseudo code looks as follows:

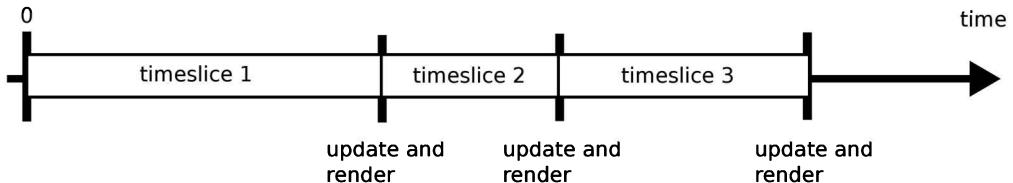
```
int lastTime = Timer_Gettime();

while (1){
    int currentTime = Timer_Gettime();
    int timeSlice = currentTime - lastTime;

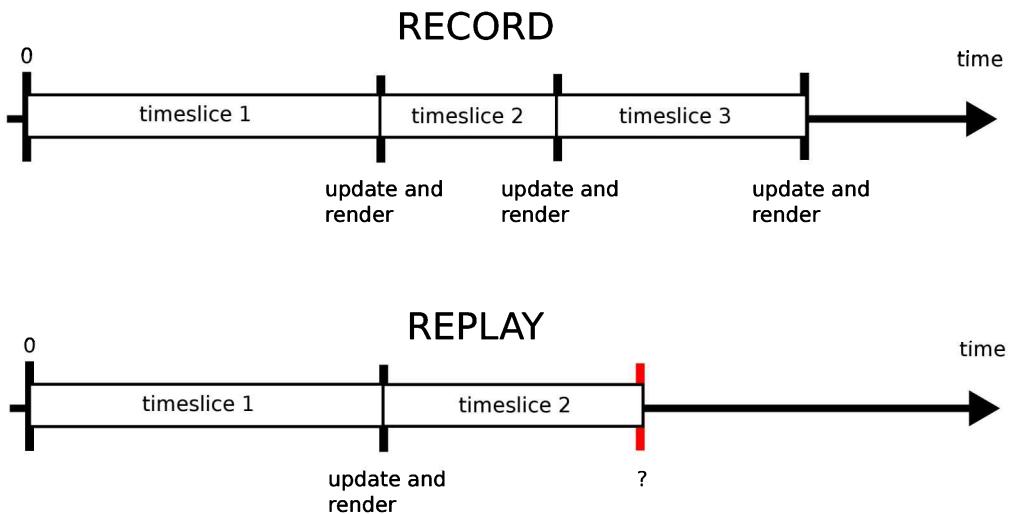
    UpdateWorld(timeSlice);
    RenderWorld();

    lastTime = currentTime;
}
```

This is a pretty standard design for an engine from the early 90s. However, it has a major flaw: each slice of the game has a different duration depending on how long it takes to render and update. This variability makes the game nondeterministic between two machines, or even between two runs on the same machine. In the next drawing, all three timeslices representing three frames have different durations.



At the beginning of each frame, the engine retrieves user inputs and combines them with the duration of the previous frame to update the world. Even if inputs are recorded for each frame, the same game could not be replayed as upon replaying a recorded game the engine would invariably reach a point where no inputs are available because a frame took more or less time to render. A simple disk access taking longer could have altered the duration of a frame, resulting in a different world outcome.



**Figure 4.21:** The second frame too longer to be rendered during the replay. No inputs are available for the player. Replay is now out of sync.

To solve this problem during the demo playback (shipped with the game), the engine dis-

regards the heartbeats and simulates events at fixed interval ( $\text{DEMOTICK} = 4$ )<sup>11</sup>. This hack resulted in a slower playback on 286 CPUs and faster playback on 486 CPUs (because it was recorded on a 386DX).

In 1993, the Doom engine would solve this issue by simulating the world at fixed intervals<sup>12</sup>.

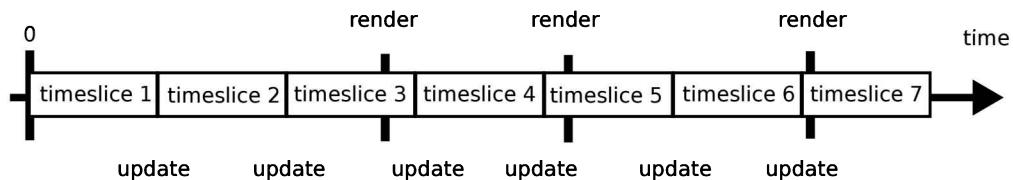
```
int gameOn = 1
int simulationTime = 0;

while (gameOn){
    int realTime = Gettime();

    while (simulationTime < realTime){
        simulationTime += 28; //Timeslice is ALWAYS 28 ms.
        UpdateWorld(28);
    }

    RenderWorld();
}
```

This design decouples the renderer from the world updates. Timeslices are always the same duration. User inputs can be recorded and replayed without getting out of sync.



“

Fixed versus variable game tic rates is still a debate today! Doom using a locked tic rate made demos easy, but it also limited it to 35 fps, which later Pentium computers could get restricted by.

**John Carmack - Programmer**

”

## 4.7.2 Life of a 3D Frame

Five stages are involved in drawing a 3D scene:

<sup>11</sup> $70/4 = 17.5$ , the machine used to record the demo ran at 17 fps. Probably a high end 386.

<sup>12</sup>Doom run at a fixed rate of 35 fps.

1. Clear the framebuffer by drawing the floor and the ceiling (both solid colors).
2. For each column of pixels on screen, cast a ray from the player to the nearest wall.  
Draw a textured column of pixels with height inversely proportional to the distance.
3. Draw the sprites (enemies, lamps, barrels).
4. Draw the weapon<sup>13</sup>.
5. Flip buffers: instruct CRT Controller to use the framebuffer just composed on next vsync.

In the next six screenshots, the engine has been modified to slow down in order to see the content of the framebuffer at the end of each stage.



**Figure 4.22:** Stage 1: Clear screen

---

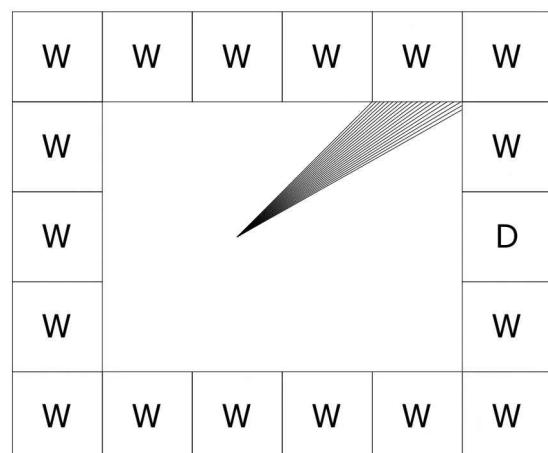
<sup>13</sup>A modern 3D engine would render the weapon first and leverage the depth buffer to save fillrate. In that era, however, memory access was so slow it was faster to accept a bit of overdraw.



**Figure 4.23:** Stage 2: Drawing Walls: 15 rays

Notice how the length of each ray corresponds to the height of a column in order to achieve perspective.

The longer the distance the ray traveled, the shorter the column is drawn on screen.

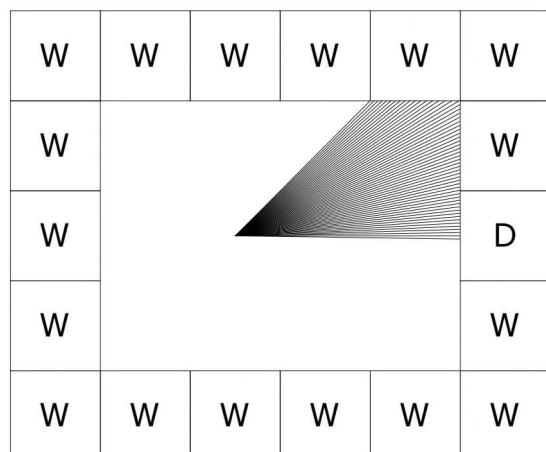


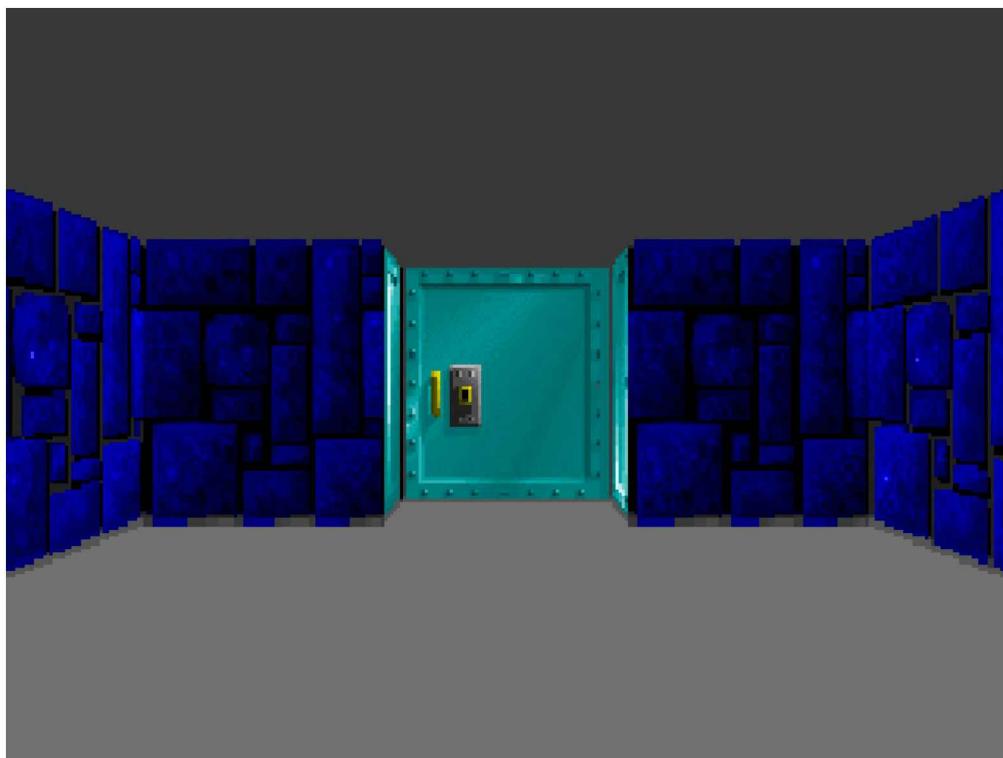


**Figure 4.24:** Stage 2: Drawing Walls: 160 rays

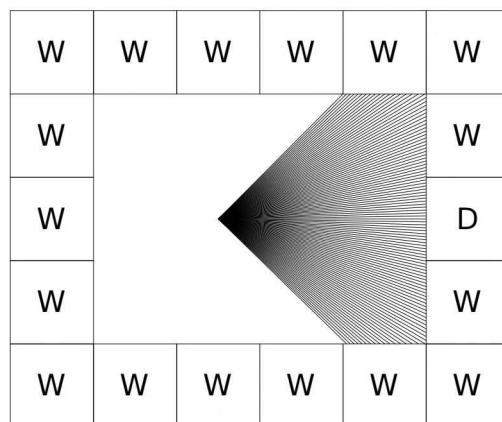
Doors are not sprites, they are part of the solid world. Notice how on the map a door is block aligned with the wall yet is rendered farther for perspective.

The raycaster recognizes door tiles and injects a delta to the distance a ray traveled. If the door is partially open, the raycaster is also able to detect if the ray should stop or be allowed to traverse the tile.





**Figure 4.25:** Stage 2: Drawing Walls has completed.





**Figure 4.26:** Stage 3: Drawing Things (a.k.a: Scaled, a.k.a: Sprites)

After the raycaster is done, "things" are rendered. During this step, clipping is performed against the wall and doors.



**Figure 4.27:** Stage 4: Drawing Weapon

### 4.7.3 3D Setup

Before starting to draw frames, the 3D renderer sets up the VRAM with static elements forming the HUD<sup>14</sup>: the green background, the blue status bar, and all labels ("LEVEL", "SCORE", "LIVES", "HEALTH", and "AMMO").

“

Wolfenstein was the last Id game that still had the concept of score and lives. We were still in a quarter-eating-arcade-game design mode. With Doom, you finally got to just keep playing as long as you wanted, which was a bit radical at the time!

**John Carmack - Programmer**

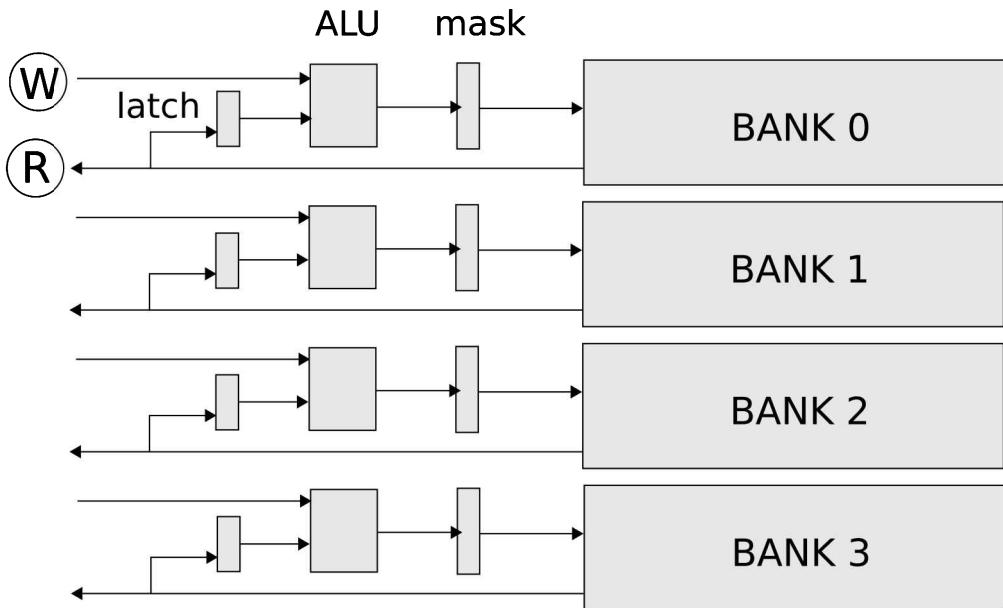
”

<sup>14</sup>Heads-Up Display.



This HUD is drawn only once at the beginning of the 3D phase. It has to be drawn in all three pages. For each new frame, the engine draws the 3D view in the reserved white canvas in the center. Small portions at the bottom of the HUD are updated: Level, Score, Lives, Status, Health, Ammo, and Current Weapon receive special treatment via another VGA trick in order to speed up rendition.

The hardware chapter described Mode 12h, which despite being unfit for games still has an interesting characteristic. Mode 12h is a 16 color mode where each pixel color index is contained in a nibble. The four bits are spread across the four VGA banks. Since all write operations are one byte wide, it is not hard to imagine the difficulty in plotting a single pixel without changing the others stored in the same byte. One would have to do four read, four xor, and four writes. Since the designers of the VGA were not complete sadists, they added some circuitry to simplify this operation. For each bank, they created a latch placed in front of a configurable ALU.



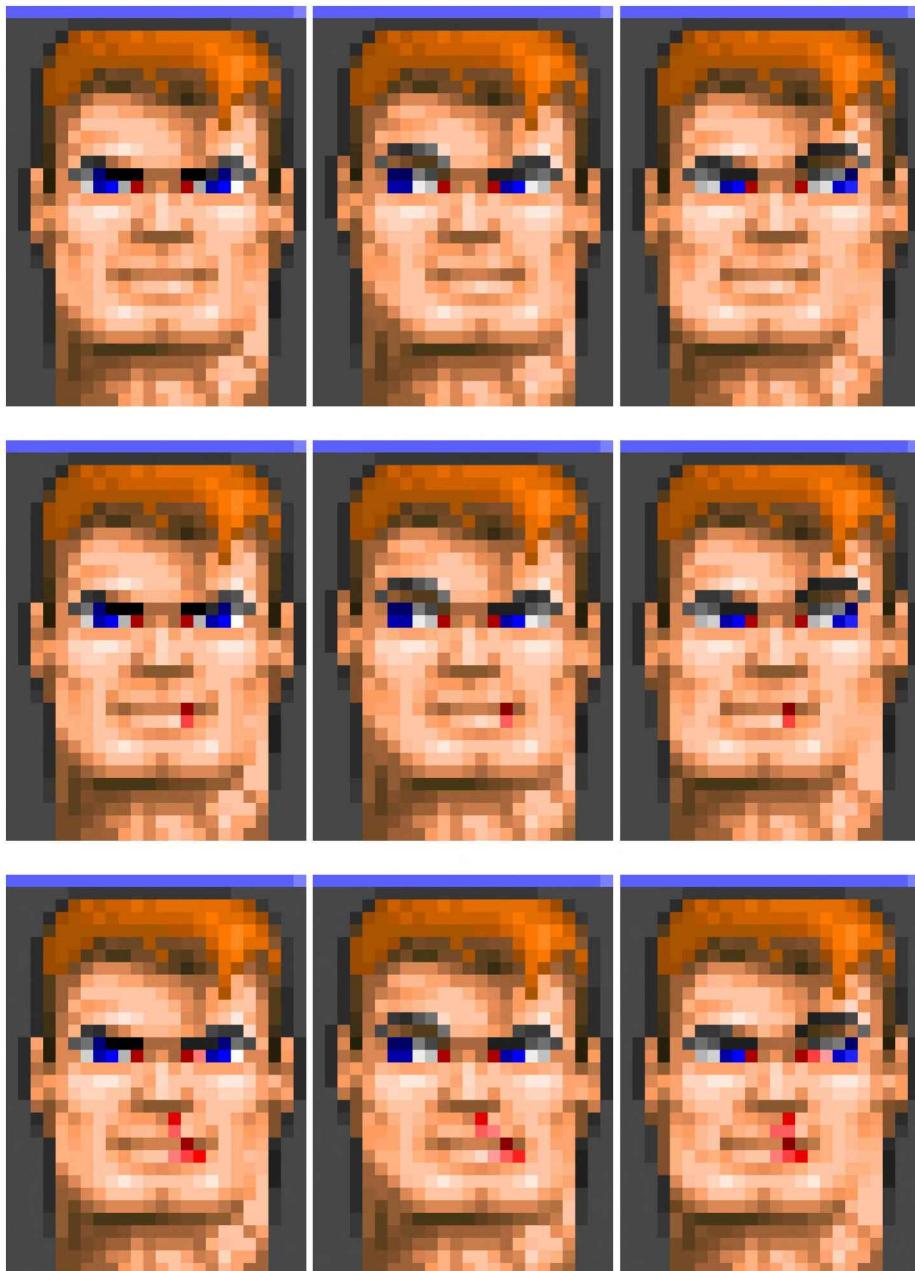
**Figure 4.28:** Latches memorize read operation from each bank. The memorized value can be used for later writes.

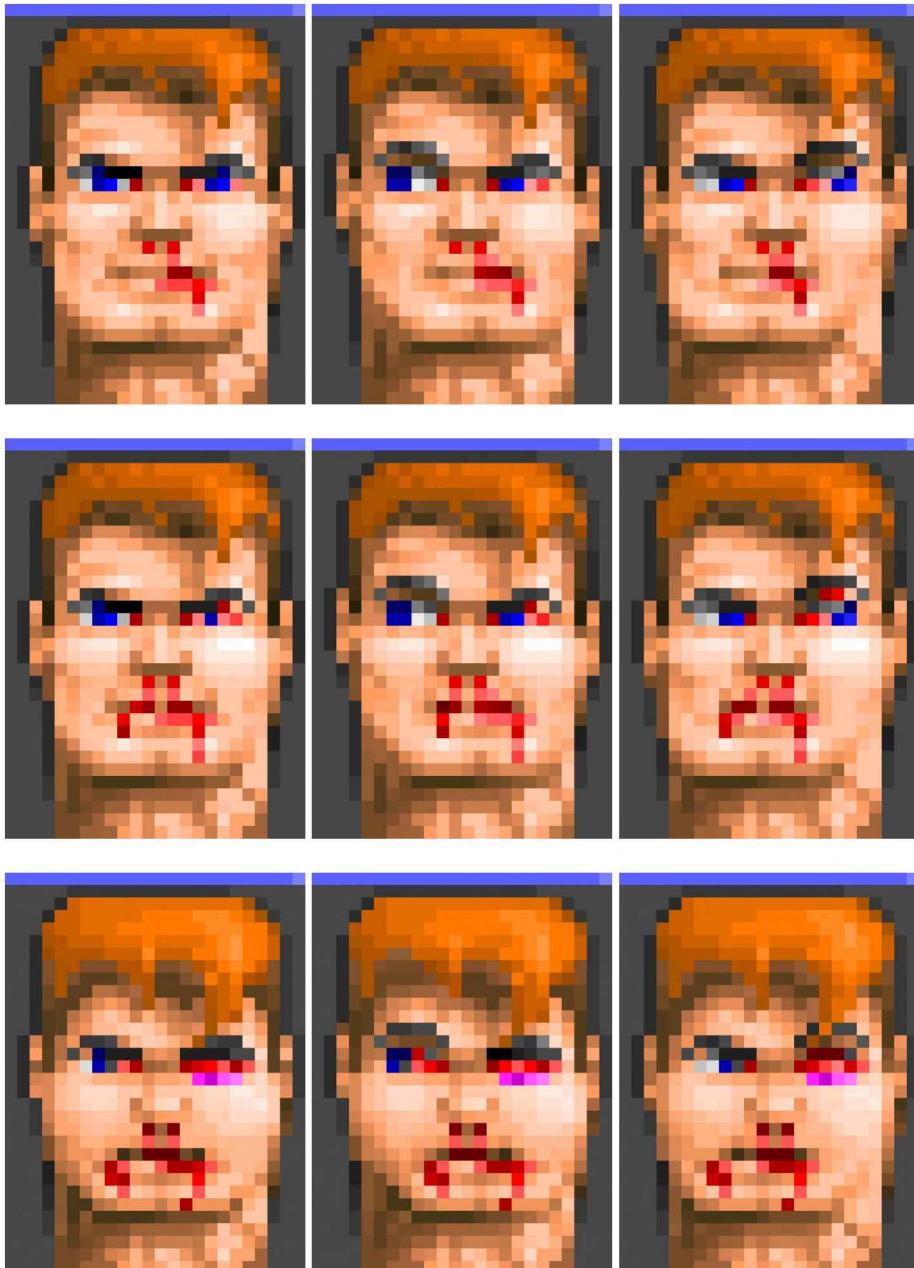
With this architecture, each time the VRAM is read (R), the latch from the corresponding bank is loaded with the read value. Each time a value is written to the VRAM (W), it can be composited by the ALU using the latched value and the written value. This design allowed mode 12h programmers to plot a pixel easily with one read, one ALU setup, and one write instead of four reads, 4 xors, and 4 writes.

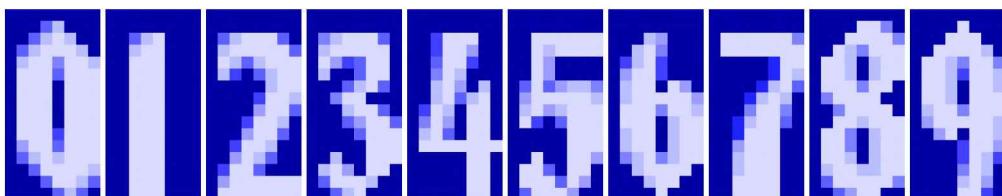
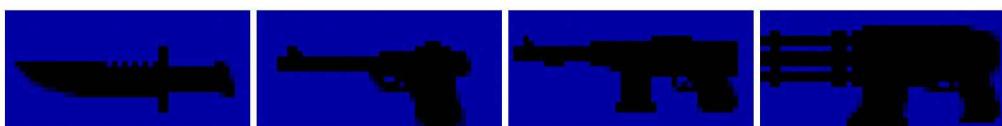
Mode Y does not need latches to operate since values are stored in bytes, and banks can be updated individually thanks to the mask. However, tinkerers<sup>15</sup> found that these latches are still active in Mode-Y. By getting a little creative, the circuitry can be re-purposed. The ALU in front of each bank can be setup to use only the latch for writing. With such a setup, upon doing one read, four latches are populated at once and four bytes in the bank are written with only one write to the RAM. This system allows transfer from VRAM to VRAM 4 bytes at a time.

To take full advantage of this optimization, the 3D renderer uploads images to the VRAM above the third page. In total, the 43 sprites used to update the HUD are loaded in the asset page when the engine starts up.

<sup>15</sup>Michael Abrash Graphic Programming Black Book: Chapter 48 – Mode X Marks the Latch.





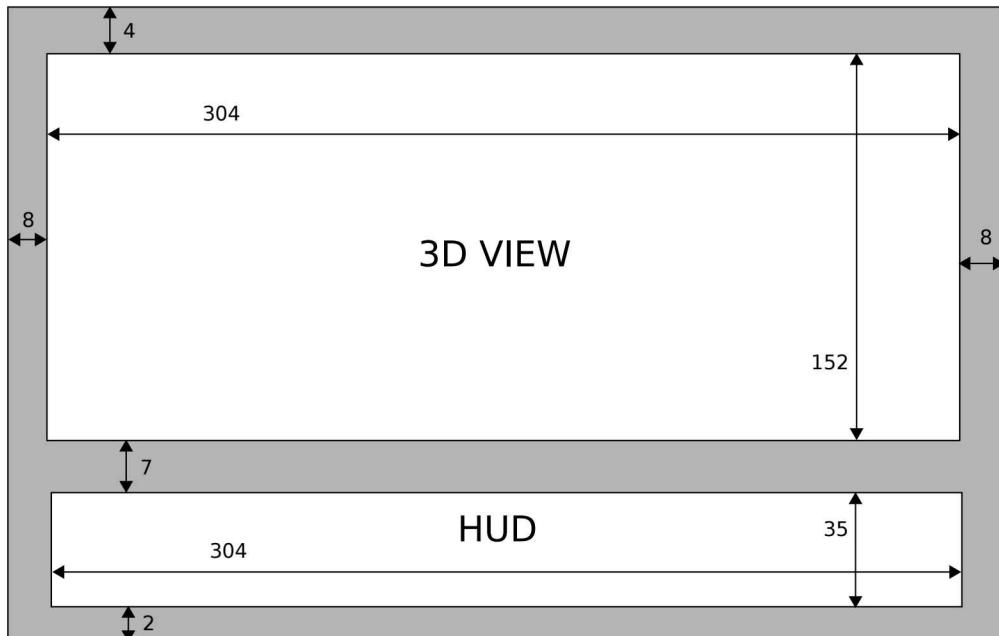




Trick : Images are stored woven. All bytes for bank 0, then all bytes for bank 1, and so on. This clever way to store data allows a bank to be loaded very fast with one `memcpy` per bank (four `memcpy` per image).

All these assets account for  $48 \times 24 \times 4$  (weapons sprites) +  $14 \times 8 \times 16$  (numeral and key sprites) +  $24 \times 24 \times 32$  (face sprites) +  $224 \times 48$  (paused + psyched) = 34,816 bytes. There are therefore 35,324 unused bytes in the fourth VGA page.

For this trick to work, image sources and destinations have to be aligned on four bytes horizontally in screen space. If you take a look at the location of each element on screen you can see that elements are aligned on four pixels horizontally but there is no such constraint vertically.



While it is faster to copy four pixels in one read and one write, this trick does not provide a full 400% speed increase as all writes have to be done in all three screen pages. This is

obvious when looking at the routine for updating the status (hero's face).

```
void StatusDrawPic (int x, int y, int picnum)
{
    unsigned temp;

    temp = bufferofs;
    bufferofs = 0;

    bufferofs = PAGE1START+(200-STATUSLINES)*SCREENWIDTH;
    LatchDrawPic (x,y,picnum);
    bufferofs = PAGE2START+(200-STATUSLINES)*SCREENWIDTH;
    LatchDrawPic (x,y,picnum);
    bufferofs = PAGE3START+(200-STATUSLINES)*SCREENWIDTH;
    LatchDrawPic (x,y,picnum);

    bufferofs = temp;
}
```

This optimization still results in a 30% overall speed increase.

#### 4.7.4 Clearing the Screen

At the beginning of a frame, the engine switches to the next page and clears the 3D area with ceiling and floor colors. To create this effect, it uses the same trick seen in the 2D renderer and sets the VGA bank mask to 15 to write to all banks simultaneously. Since values are written 16 bits at a time, it can write 8 pixels at a time.

```

void VGAClearScreen (void)
{
    unsigned ceiling=vgaCeiling[gamestate.episode*10+mapon];

    asm  mov  dx ,SC_INDEX
    asm  mov  ax ,SC_MAPMASK+15*256 // write all planes
    asm  out dx ,ax

    asm  mov  dx ,80
    asm  mov  ax ,[viewwidth]
    asm  shr  ax ,2
    asm  sub  dx ,ax           // dx = 40-viewwidth/2

    asm  mov  bx ,[viewwidth]
    asm  shr  bx ,3           // bl = viewwidth/8
    asm  mov  bh,BYTE PTR [viewheight]
    asm  shr  bh ,1           // half height

    asm  mov  es ,[screenseg]
    asm  mov  di ,[bufferofs]
    asm  mov  ax ,[ceiling]

    toploop:
    asm  mov  cl ,bl
    asm  rep  stosw
    asm  add  di ,dx
    asm  dec  bh
    asm  jnz  toploop

    asm  mov  bh,BYTE PTR [viewheight]
    asm  shr  bh ,1           // half height
    asm  mov  ax ,0x1919

    bottomloop:
    asm  mov  cl ,bl
    asm  rep  stosw
    asm  add  di ,dx
    asm  dec  bh
    asm  jnz  bottomloop

}

```

Clearing the 3D canvas made of  $304 \times 152 = 46,208$  pixels requires only 27 instructions

thanks to REP STOSW.

Colors for the floor and the ceiling do not come from map data files. The ground is always the same color (0x19) and ceiling colors are hardcoded in the engine on a per level basis.

```
byte vgaCeiling []=
{
    0x1d, 0x1d, 0x1d, 0x1d, 0x1d, 0x1d, 0x1d, 0x1d, 0xbff,
    0x4e, 0x4e, 0x4e, 0x1d, 0x8d, 0x4e, 0x1d, 0x2d, 0x1d, 0x8d,
    0x1d, 0x1d, 0x1d, 0x1d, 0x1d, 0x2d, 0xdd, 0x1d, 0x1d, 0x98,
    0x1d, 0x9d, 0x2d, 0xdd, 0xdd, 0x9d, 0x2d, 0x4d, 0x1d, 0xdd,
    0x7d, 0x1d, 0x2d, 0x2d, 0xdd, 0xd7, 0x1d, 0x1d, 0x1d, 0x2d,
    0x1d, 0x1d, 0x1d, 0x1d, 0xdd, 0xdd, 0x7d, 0xdd, 0xdd, 0xdd
};
```

Following are ceiling colors 0x1D, 0xBF, 0x4E and 0x8D.

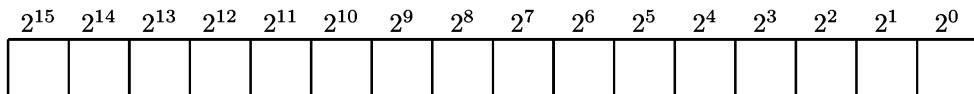


### 4.7.5 Solving the CPU Problem

In the Chapter 2 Hardware p19 describing CPU capabilities, the reader was left with a problem: the machine cannot perform floating point operations quickly enough. This is a pretty big issue for a 3D engine given all the trigonometry involved. It turns out the solution is to trick the ALU via a technique called "fixed point arithmetic".

#### 4.7.5.1 Fixed Point

The normal layout of an `int` is as follows.



**Figure 4.29:** Integer layout.

The value of the sequence of bits *0010010010010010*:

$2^{15}$	$2^{14}$	$2^{13}$	$2^{12}$	$2^{11}$	$2^{10}$	$2^9$	$2^8$	$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0

Represents  $2^{13} + 2^9 + 2^5 + 2^1 = 8738$ .

Fixed Point allows for keeping track of fractions while still using the integer operations of the CPU. The machine manipulates what are supposed to be integer numbers, but the programmer sees them as a value containing an integer part and a fractional part:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$

**Figure 4.30:** Fixed point layout 8:8 (8 bits for the integer part and 8 bits for the fractional part).

The same sequence of bits *0010010010010010* with different powers of two...

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0

Now represents:

$$\begin{aligned} 2^5 + 2^1 &= 34 \text{ for the integer part.} \\ 2^{-3} + 2^{-7} &= 0.1328125 \text{ for the fractional part.} \\ &= 34.1328125 \end{aligned}$$

The beauty of fixed point is that addition and subtraction work exactly like integers from the CPU instruction side.

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0

**Figure 4.31:** 34.75

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

**Figure 4.32:** 1.5

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0

**Figure 4.33:**  $34.75 + 1.5 = 36.25$ 

Even right shift trick (divide by power of two) and left shift trick (multiply by power of two) work...

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

**Figure 4.34:** 1.5

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

**Figure 4.35:**  $1.5 \ll 1 = 3$ 

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0

**Figure 4.36:**  $1.5 \gg 1 = 0.75$ 

The notation for fixed point is BITS\_FOR\_INTEGER\_PART:BITS\_FOR\_FRACTIONAL\_PART. In the game, the player's position is 16:16. The grid location is one shift operation away  $\text{player-}>\text{x} \gg 16$ .

Performing multiplication and division are special cases. There are two ways to do this: either multiply two 32 bit values into 64 bits (and drop the upper and lower 16 bits) or

drop the precision of both fixed point values and then multiply them into 32 bits. Both methods involve accepting a loss of data via overflow or underflow. Let's take the example of  $98.7539 * 1.5$ .

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	1	1	0	0	0	1	0	1	1	0	0	0	0	0	1

**Figure 4.37:** 98.7539

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

**Figure 4.38:** 1.5

First shift both operators by 4 bits to the right:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	1	1	0	0	0	1	0	1	1	0	0

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0

Finally multiply them together:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-5}$	$2^{-6}$	$2^{-7}$	$2^{-8}$
1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0

**Figure 4.39:**  $98.7539 * 1.5 = 148.125$

$$128 + 16 + 4 + 0.125 = 148.125$$

Notice that you have to be careful. In the previous example some precision was lost ( $2^{-8}$  bits disappeared during the  $\gg 4$  operation) and an overflow could have occurred (multiplying by 3 instead of 1.5 would have gone past the precision of 8:8).

In the source code, all fixed point variables are 32 bits wide and use a special typedef.

```
typedef long fixed;
```

Not all `fixed` are the same. Some are 16:16, some are 24:8 and operations such as multiplication can be implemented differently. It can be fairly complicated for tweaked `fixed` where the leftmost bit stores the sign:

```
fixed FixedByFrac (fixed a, fixed b) {
asm mov si,[WORD PTR b+2]    // sign of result =
                                // sign of fraction
asm mov ax,[WORD PTR a]
asm mov cx,[WORD PTR a+2]

asm or  cx,cx
asm jns aok:                // negative?
asm neg cx
asm neg ax
asm sbb cx,0
asm xor si,0x8000           // toggle sign of result
aok:

// multiply cx:ax by bx
asm mov bx,[WORD PTR b]
asm mul bx                  // fraction*fraction
asm mov di,dx                // di is low word of result
asm mov ax,cx                //
asm mul bx                  // units*fraction
asm add ax,di
asm adc dx,0

// put result dx:ax in 2's complement
asm test si,0x8000          // is the result negative?
asm jz  ansok:
asm neg dx
asm neg ax
asm sbb dx,0

ansok:; // I stick the return value in with ASMs
}
```

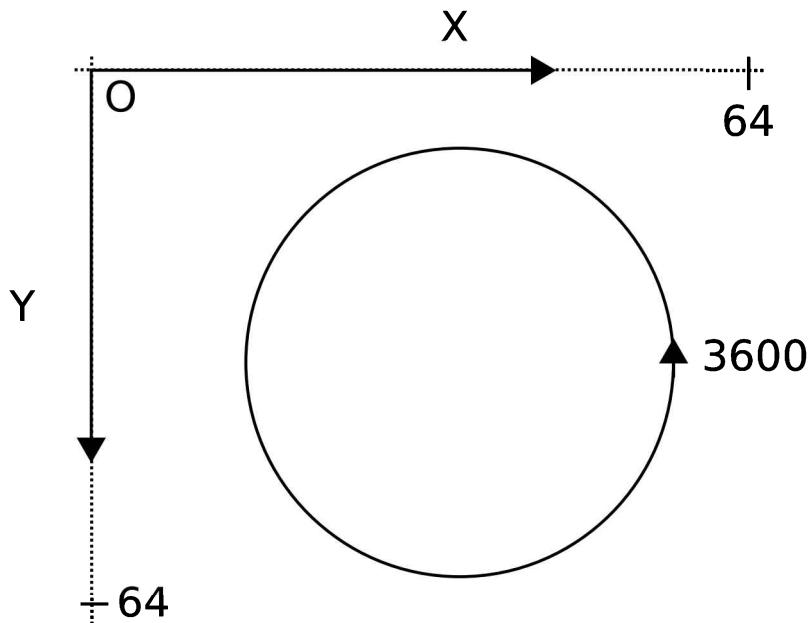
And much easier when the engine knows two `fixed` are the same sign:

```
fixed FixedMul (fixed a, fixed b) {
    return (a>>8)*(b>>8);
}
```

**Trivia :** Fixed Point Arithmetic usage was not limited to PC gaming. Many game consoles manufactured in the 90s and later did not feature floating point units in order to reduce production cost and maximize CPU pipeline throughout. Sony's original PlayStation (1994) and Sega's Saturn (1994) are examples.

#### 4.7.5.2 Coordinate System

With the CPU float/int problem out of the way, it is time to study how a ray is cast. The coordinate system finds its origin in the upper left. A map is a grid of 64 by 64 blocks.



Since one block is 8 feet, maps can be 512 feet wide and tall. Fixed point variables are used all over the place. Column heights are 29:3. Player position is 16:16. Angles however are ints representing tenths of degrees with a range [0, 3600].

#### 4.7.5.3 Square World and Ray Casting

Drawing the walls is all about determining what is visible, what is not, and what is in front of what. Michael Abrash's view on the topic tells it all:

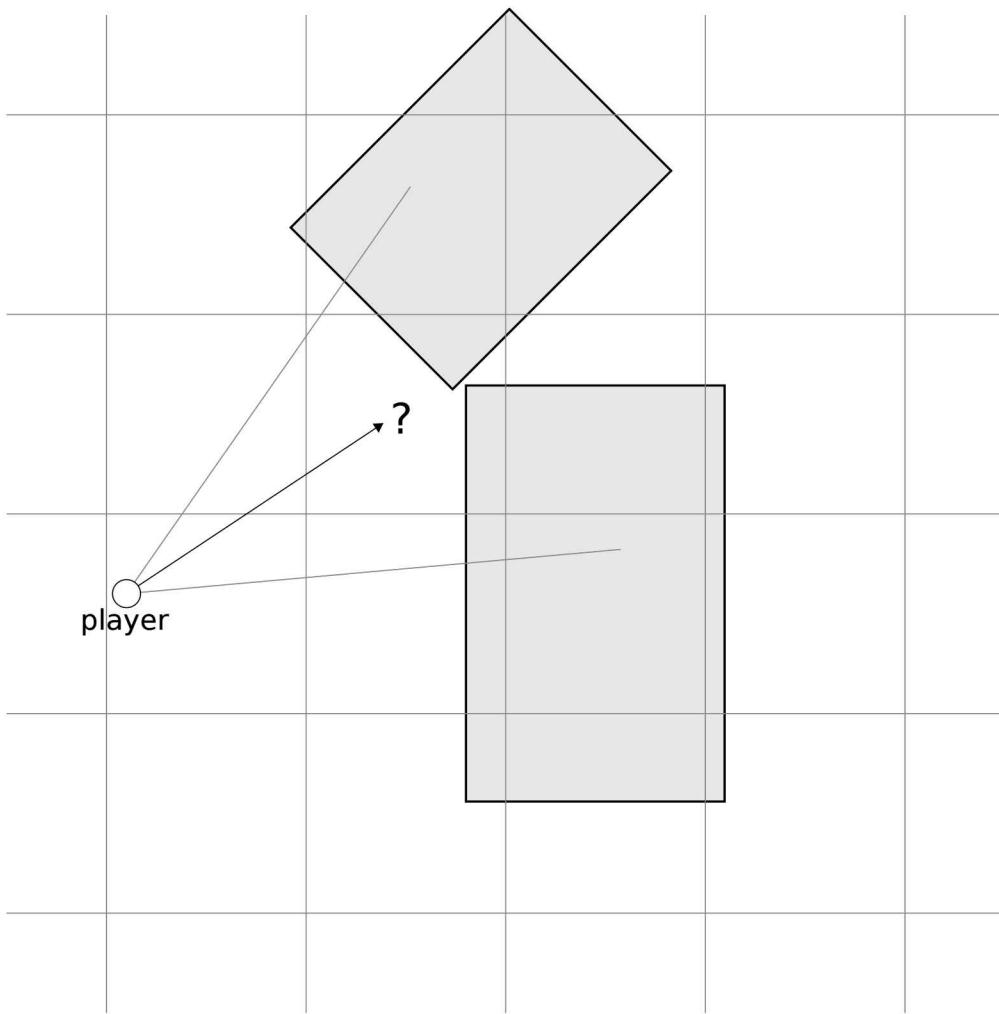
I want to talk about what is, in my book, the toughest 3-D problem of all, visible surface determination (drawing the proper surface at each pixel), and its close relative, culling (discarding non-visible polygons as quickly as possible, a way of accelerating visible surface determination). In the interests of brevity, I'll use the abbreviation VSD to mean both visible surface determination and culling from now on.

Why do I think VSD is the toughest 3-D challenge? Although rasterization issues such as texture mapping are fascinating and important, they are tasks of relatively finite scope, and are being moved into hardware as 3-D accelerators appear; also, they only scale with increases in screen resolution, which are relatively modest.

In contrast, VSD is an open-ended problem, and there are dozens of approaches currently in use. Even more significantly, the performance of VSD, done in an unsophisticated fashion, scales directly with scene complexity, which tends to increase as a square or cube function, so this very rapidly becomes the limiting factor in doing realistic worlds. I expect VSD increasingly to be the dominant issue in realtime PC 3-D over the next few years, as 3-D worlds become increasingly detailed.

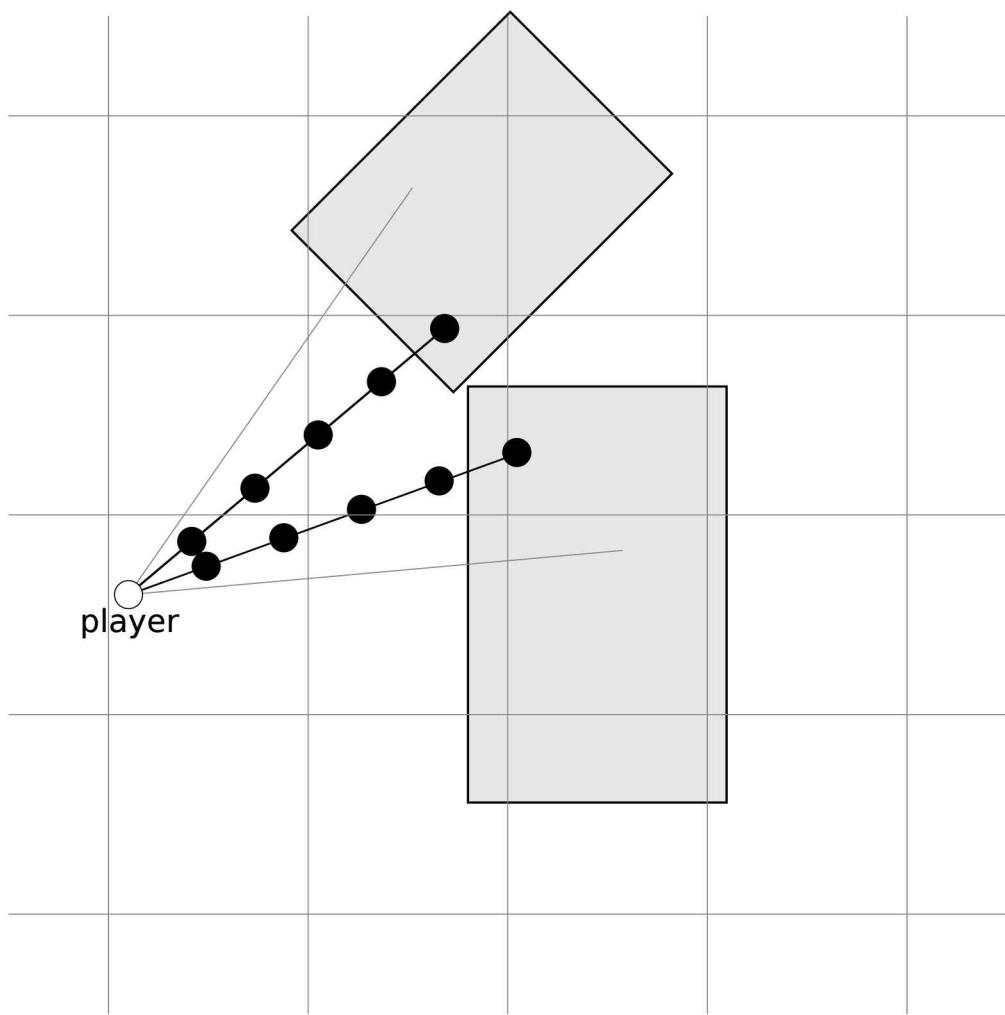
#### **Michael Abrash - Graphics Programming Black Book**

VSD is not only complicated to do right, it is also hard to do fast. It is easy to understand with a small example where objects are free form with no alignment, as follows.



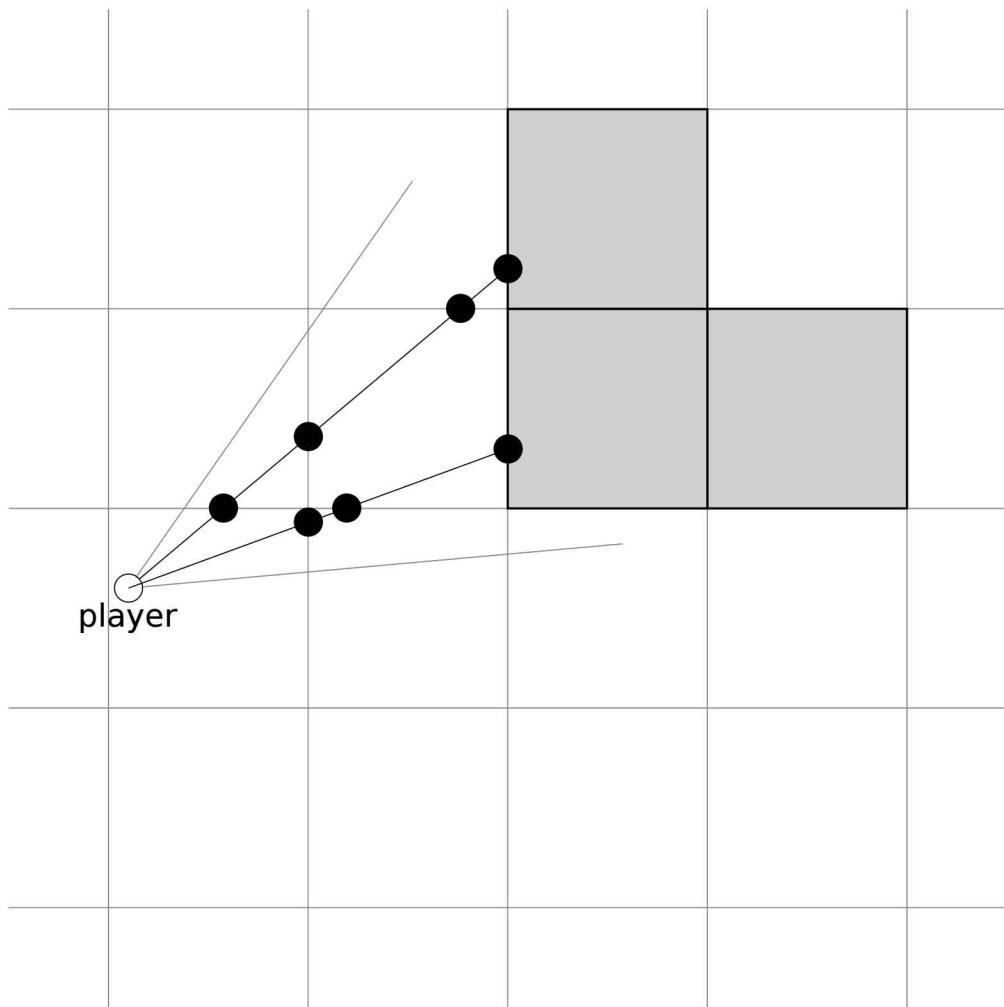
In a world with no constraints, it is difficult to find the intersection of a ray with an object.

In the example above, a valid approach would be to iterate over all objects in the world and perform ray marching, which consists of checking if a ray intersects objects at a regular interval. However, this is very CPU intensive and would not be fast enough with fixed point. Not to mention some objects could slip between the regular intervals, making VSD inaccurate.



**Figure 4.40:** Ray marching in action.

In a world with some constraints the problem becomes much simpler. If a map is made of aligned square blocks evenly distributed across a grid, a solution yielding 100% accuracy and low runtime overhead is to check for 'hits' only when a ray crosses the grid. This was the choice made for *Wolfenstein 3D*, and explains why the game can only draw perpendicular walls of 8 feet by 8 feet by 8 feet.



**Figure 4.41:** Raycasting with a grid.

These constraints enable a variant of the DDA algorithm<sup>16</sup>, which is a fast and accurate way to detect where a ray hits a wall on the map.

---

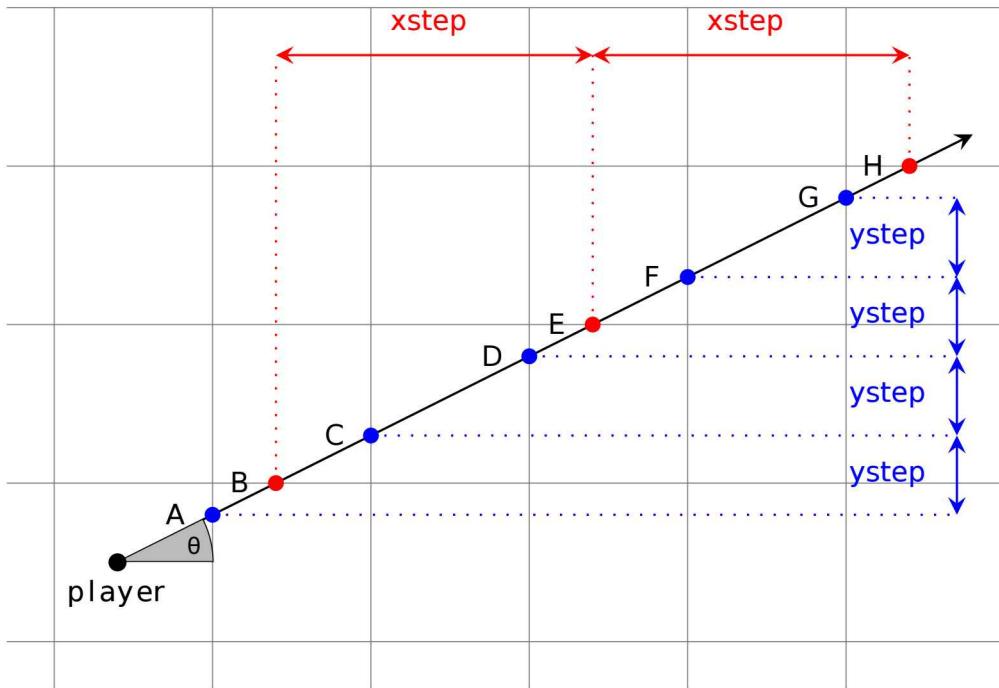
<sup>16</sup>Digital differential analyzer.

Much was made about the "ray casting" used in Wolfenstein, but the real reason for it was that I had a lot of trouble with wall-span rendering in Catacomb 3D. C3D (and Hovertank 3D before that) shipped with various graphics glitches that you could get in some combinations of map block configurations, position, and viewing angle. Some were due to fixed point precision issues not being handled optimally, and some were due to clipping and culling issues that I didn't really get a handle on until a couple years later. In any case, they bothered me a lot. Spurious graphics glitches do a lot of harm to the sense of immersion in a game, and I very much wanted Id games to feel "rock solid".

There was a clear performance cost to it - doing 320 traces through a tile map and treating each column independently is much slower than looping through a few long wall segments. However, the resulting code was small and very regular compared to the hairball of my wall span renderers, and it did deliver the rock-solid feel I wanted.

If you made extremely jagged block maps that would turn into many dozen independent wall segments, the ray casting could start to look like a good performance choice, but few scenes were even close to that. This is exactly the same ray tracing versus rasterization performance tradeoff that is still being made today, but now it is "how many tens of millions of triangles per frame to ray tracing break-even" instead of "how many dozen wall segments".

**John Carmack - Programmer**



DDA is so fast because once the first intersection of a ray with an axis is known (A for Y axis and B for X axis), all subsequent intersections' coordinates are two cheap additions away.

$$C = (X_A + 1, Y_A + ystep)$$

$$D = (X_C + 1, Y_C + ystep)$$

$$F = (X_D + 1, Y_D + ystep)$$

$$G = (X_F + 1, Y_F + ystep)$$

Similarly, for the vertical intersections:

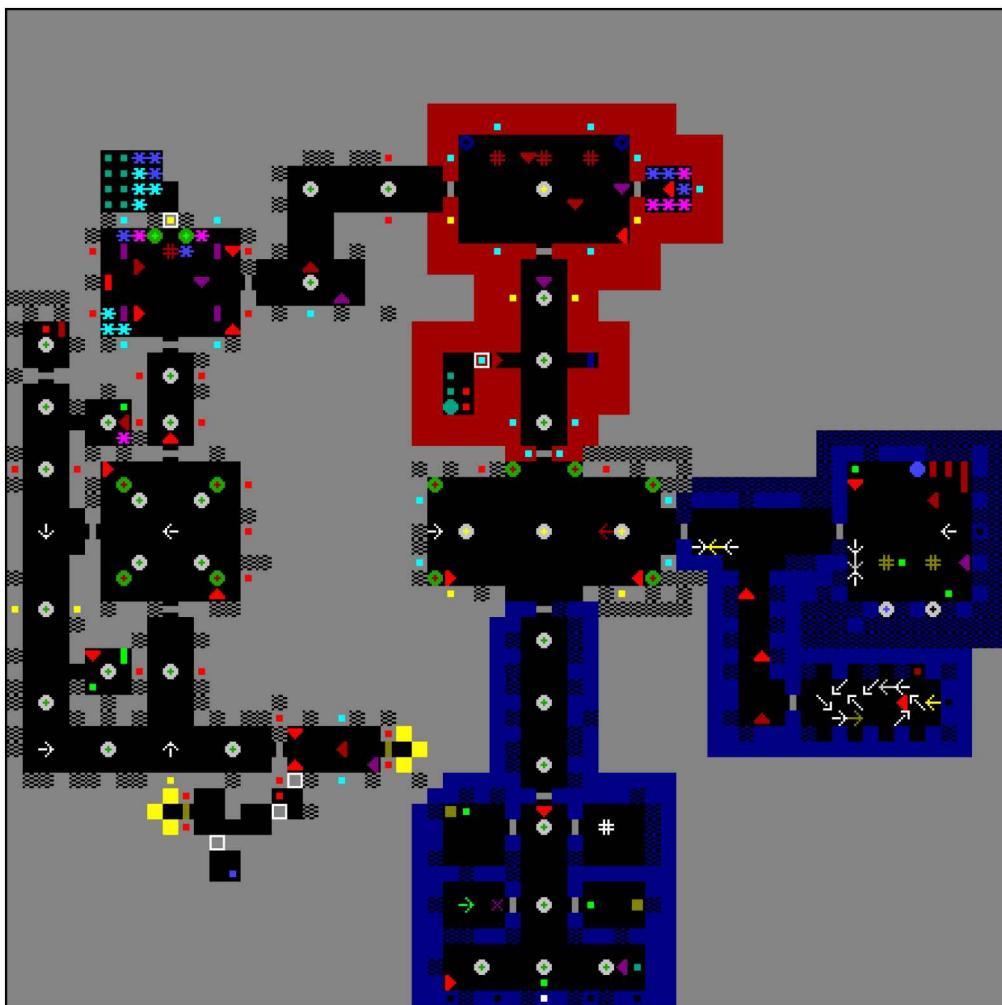
$$E = (X_B + xstep, Y_B + 1)$$

$$D = (X_E + xstep, Y_E + 1)$$

Note that  $ystep$  and  $xstep$  are simple lookups into the  $\tan$  array since  $xstep = \tan(\theta)$  and  $ystep = \tan(90 - \theta)$  where  $\theta$  is the angle of the ray in map coordinates<sup>17</sup>.

---

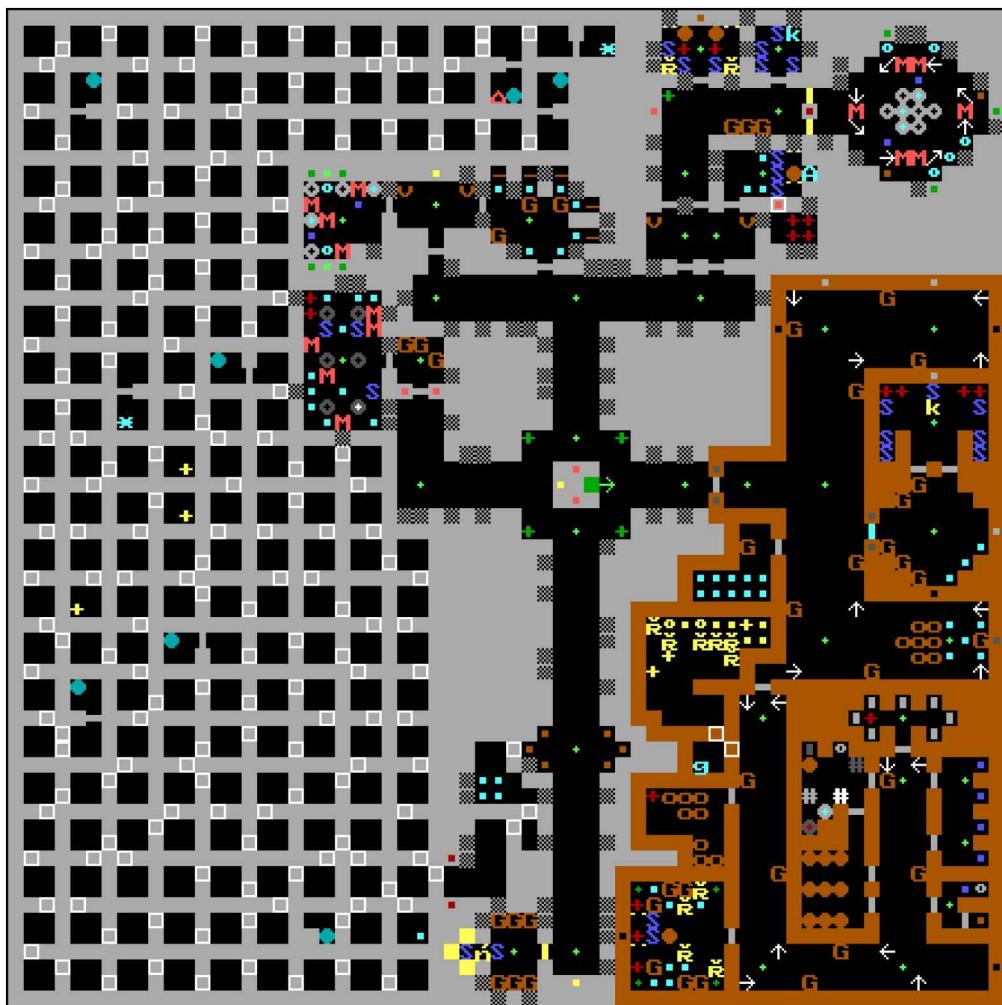
<sup>17</sup>A visual reminder of how the unit circle works can be found on page 166.



**Figure 4.42:** The legendary E1M1. Player is the green arrow at the bottom.

#### 4.7.5.4 Call Apogee

With maps being quick and easy to draw, a contest was planned. Players who found a special item in a particularly difficult to access place in the game were instructed to call the publisher (Apogee). The maze is located in Episode 2, Map 8.



**Figure 4.43:** E2M8 maze.

Behind a forest of push walls (white squares) and angry bosses (blue circles), the player finally encounters an enigmatic sign (the red triangle).

However, as players reverse engineered the map format and cheat sites emerged, the contest was called off. The sign was replaced with a skeleton in all games shipped in 1992.

In the previous map, notice there are no push walls leading to the red triangle; the room is sealed.



“

"Call Apogee and say Aardwolf." It's a sign that to this day is something that I get asked about a lot. This is a sign that appears on a wall in a particularly nasty maze in Episode 2 Level 8 of Wolfenstein 3D. The sign was to be the goal in a contest Apogee was going to have, but almost immediately after the game's release, a large amount of cheat and mapping programs were released. With these programs running around, we felt that it would have been unfair to have the contest and award a prize. The sign was still left in the game, but in hindsight, probably should have been taken out. To this day, Apogee gets letters and phone calls and asking what Aardwolf is, frequently with the question, "Has anyone seen this yet?"

Also, in a somewhat related issue, letters were shown after the highest score in the score table in some revisions of the game. These letters were to be part of another contest that got scrapped before it got started, where we were going to have people call in with their scores and tell us the code; we'd then be able to verify their score. However, with the cheat programs out there, this got scrapped too.

Basically, "Aardwolf" and the letters mean nothing now.

**Joe Siegler - Past Pioneers of the Shareware Revolution**

”

**Trivia :** What is Aardwolf? A maned striped mammal (*Proteles cristatus*) of southern and eastern Africa that resembles a hyena and feeds chiefly on carrion and insects. It was the mascot of id, appearing on Tom's Gotta Lists and the Commander Keen 6 Hint Sheet.

The reason for "aardwolf" was that it was the first image file in the NeXT dictionary on all of our NeXTStations.

**John Carmack - Programmer**

#### 4.7.5.5 Raycasting: DDA Algorithm

The DDA intersection algorithm is implemented in a fully-handcrafted 740 lines of assembly routine: `AsmRefresh`. It is represented here in pseudo-C for readability. It consists of two while loops (one each for vertical and horizontal intersections) ping-ponging with each other via goto. It is highly unorthodox and super efficient.

```

void AsmRefresh() {
    for (int i=0 ; i < pixx ; i++) {
        short angle=midangle+pixelangle[pixx];
        // Setup xstep and ystep based on angle.
        do {
            if (needed)
                goto testhorizontal;
testvertical:
        move_vertically()
        if (hitdoor)
            HitVertDoor();
        if (hitwall)
            HitVertWall();
    } while (1);

    continue;

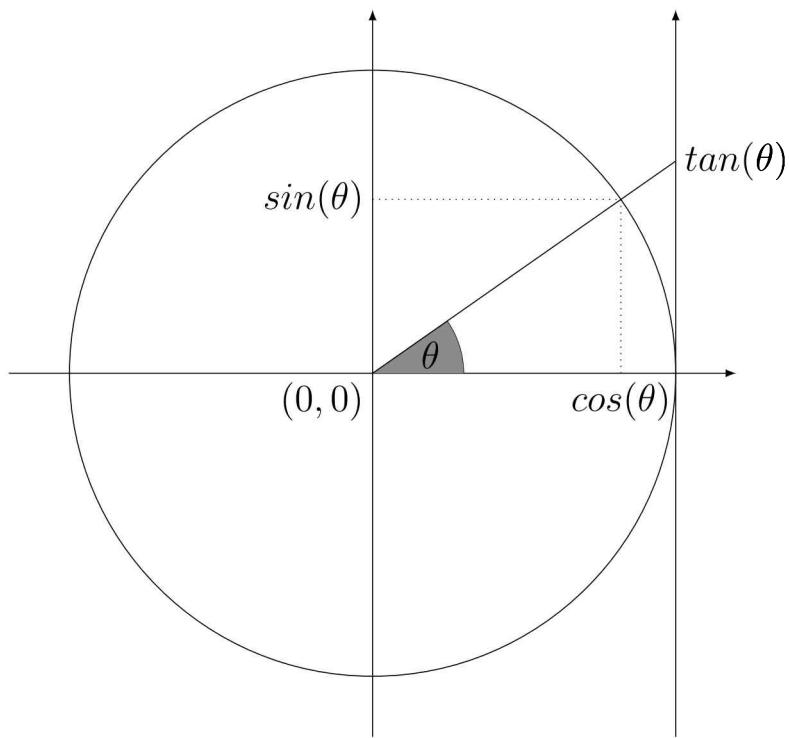
    do {
        if (needed)
            goto testvertical;
testhorizontal:
    move_horizontal()
    if (hitdoor)
        HitHorizDoor();
    if (hitwall)
        HitHorizWall();

    } while (1);
}
}

```

This implementation results in many `jmp` instructions. These would kill the icache and empty the pipeline on a modern CPU. But on an architecture with a small pipeline and no i-cache such as the 386 it is not a frame killer.

This part of the code relies heavily on unit circle principles. As we need to know how much to go vertically when advancing on the X axis and how much to go horizontally when advancing on the Y axis, the `tan` function is especially useful. This is easier to understand with the unit circle drawn (circle radius  $r$  is equal to 1).



**Figure 4.44:** Unit circle.

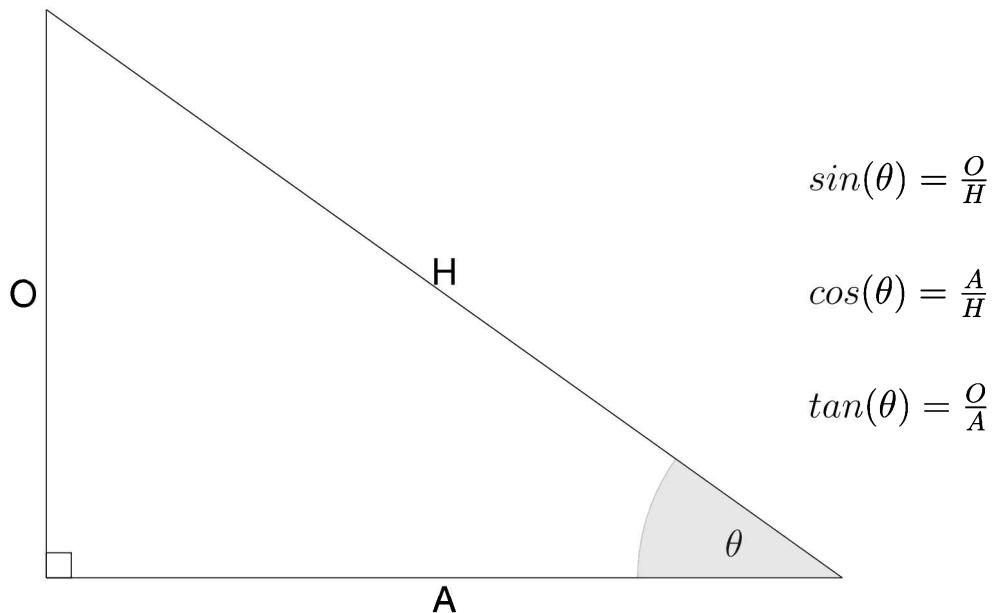
When advancing 1 on the X axis, the ray moves up  $\tan(\theta)$  on the Y axis. The reciprocal is calculated as follows: move 1 on Y axis, move  $\tan(90 - \theta)$  on X axis. To accelerate cos, sin and tan calculations, the engine uses lookup table. See 4.11.1 "Cos/Sin Table Lookup" on page 236.

#### 4.7.5.6 High School Math

Before proceeding to the next section (describing what is done with the distance from the player to the wall), here is a short refresher on a piece of high school math which forms the foundation of most calculations in the engine: SOH-CAH-TOA.

I'm no super mathematician— I learned high school math well enough to solve real world problems with it.

**John Carmack - Programmer**



**Figure 4.45:** SOH-CAH-TOA mnemonics.

This drawing is all the math you need to understand the fish eye correction and coordinate projections used to place objects (player, enemies, items) and calculate sound locations.

#### 4.7.5.7 Calculating Column Heights

Once the intersection between a ray and a wall is found from coordinate (viewx,viewy) to (xintercept,yintercept), it is time to calculate how tall the column of pixels for this ray should be. This happens in the function CalcHeight.

```

int CalcHeight (void)
{
    fixed gxt,gyt,nx,ny;
    long gx,gy;

    gx = xintercept-viewx;
    gxt = FixedByFrac(gx,viewcos);

    gy = yintercept-viewy;
    gyt = FixedByFrac(gy,viewsin);

    nx = gxt-gyt;

    //
    // calculate perspective ratio (heightnumerator/(nx>>8))
    //
    if (nx<mindist)
        nx=mindist;      // don't let divide overflow

    asm mov ax,[WORD PTR heightnumerator]
    asm mov dx,[WORD PTR heightnumerator+2]
    asm idiv  [WORD PTR nx+1]      // nx>>8
}

```

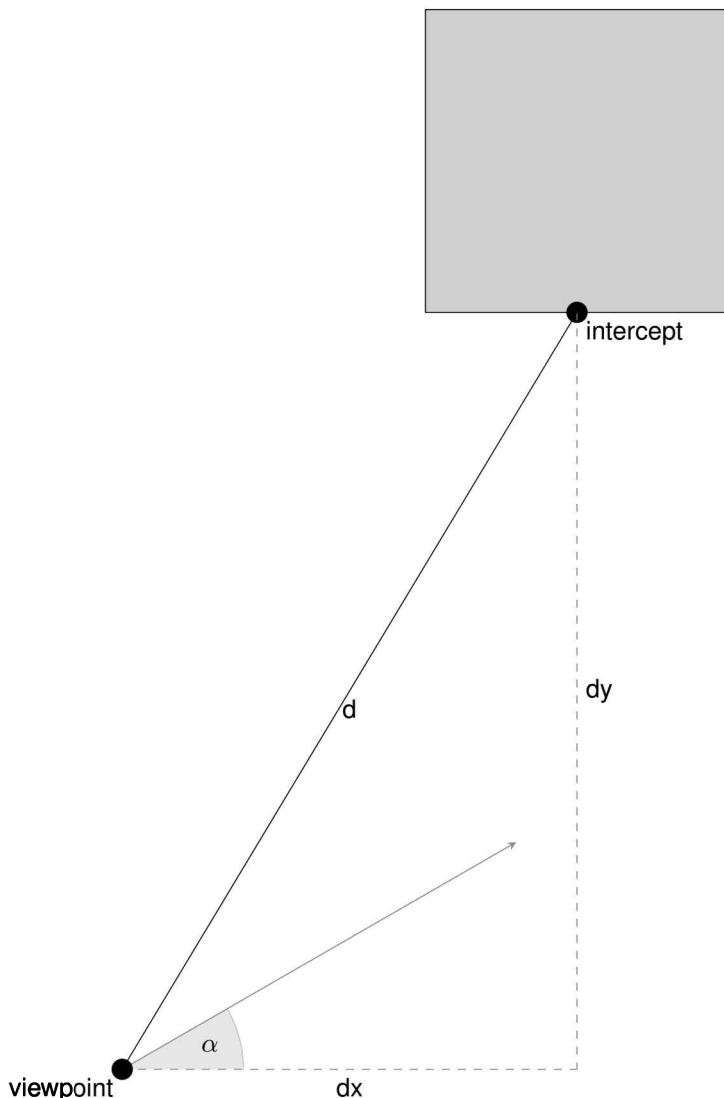
The code is not what one would expect. The raycasting algorithm is supposed to cast a ray for each pixel column and use the distance  $d$  to infer the column's height on screen. So it would have made sense to see a formula like:

$$d = \sqrt{dx^2 + dy^2}$$

Instead the code looks like it is doing:

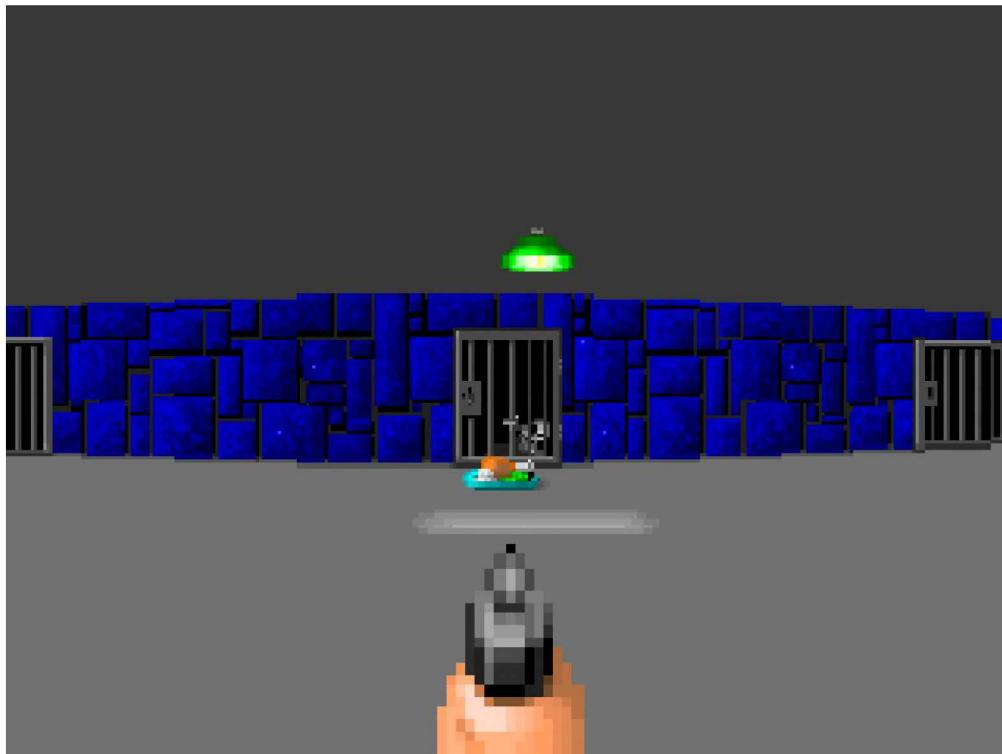
$$d = dx * \cos(\alpha) - dy * \sin(\alpha)$$

Something is fishy here. Let's explain with an example.



**Figure 4.46:** Raycasting using distance  $d$

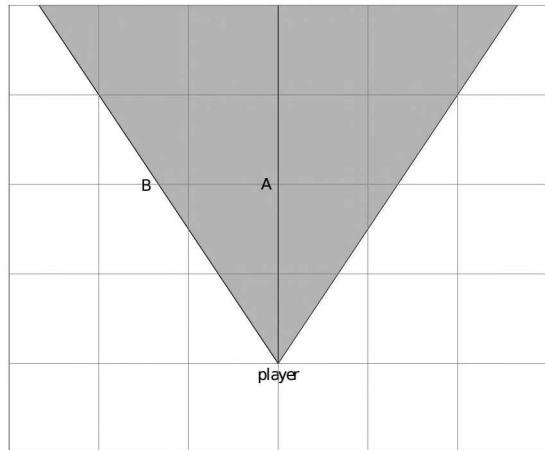
In this drawing the player is located at **viewpoint** with a view angle  $\alpha$ . A ray has been cast from **viewpoint** and it hit a wall at **intercept**. The distance  $d$  is a straight line between the player's **point of view** and the location where the ray hit the wall.  $d$  can be obtained with  $d = \sqrt{dx^2 + dy^2}$ . Repeated for all rays, such an algorithm would result in a "fisheye effect".



**Figure 4.47:** Fish eye effect: Mild

This visual artifact happens because the straight distance from the player to the wall is not constant. Walls are further away on the side of the screen and therefore represented smaller.

To demonstrate the fish eye distortion, here are three screenshots from a modified version of the engine. It was altered to use direct distance  $d$  instead of "something else" to calculate column heights. At first from a distance of 32 feet, the distortion is barely noticeable.

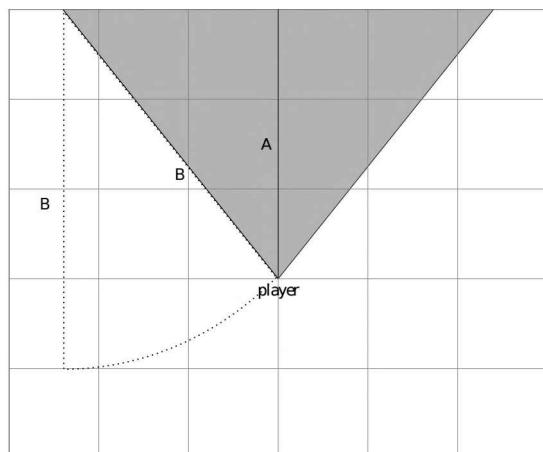




**Figure 4.48:** Fish eye effect: Bad

From a distance of 24 feet, the distortion cannot be ignored and the problem is clear.

Note that even though the player is getting closer to the wall, the ratio of A to B remains the same. Only the absolute difference in pixel height when the column is rendered on screen is increasing.

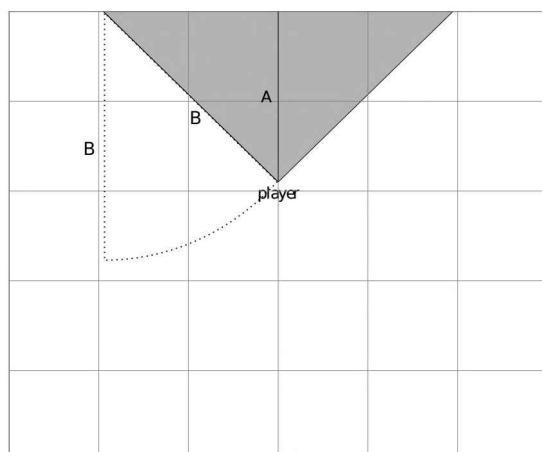


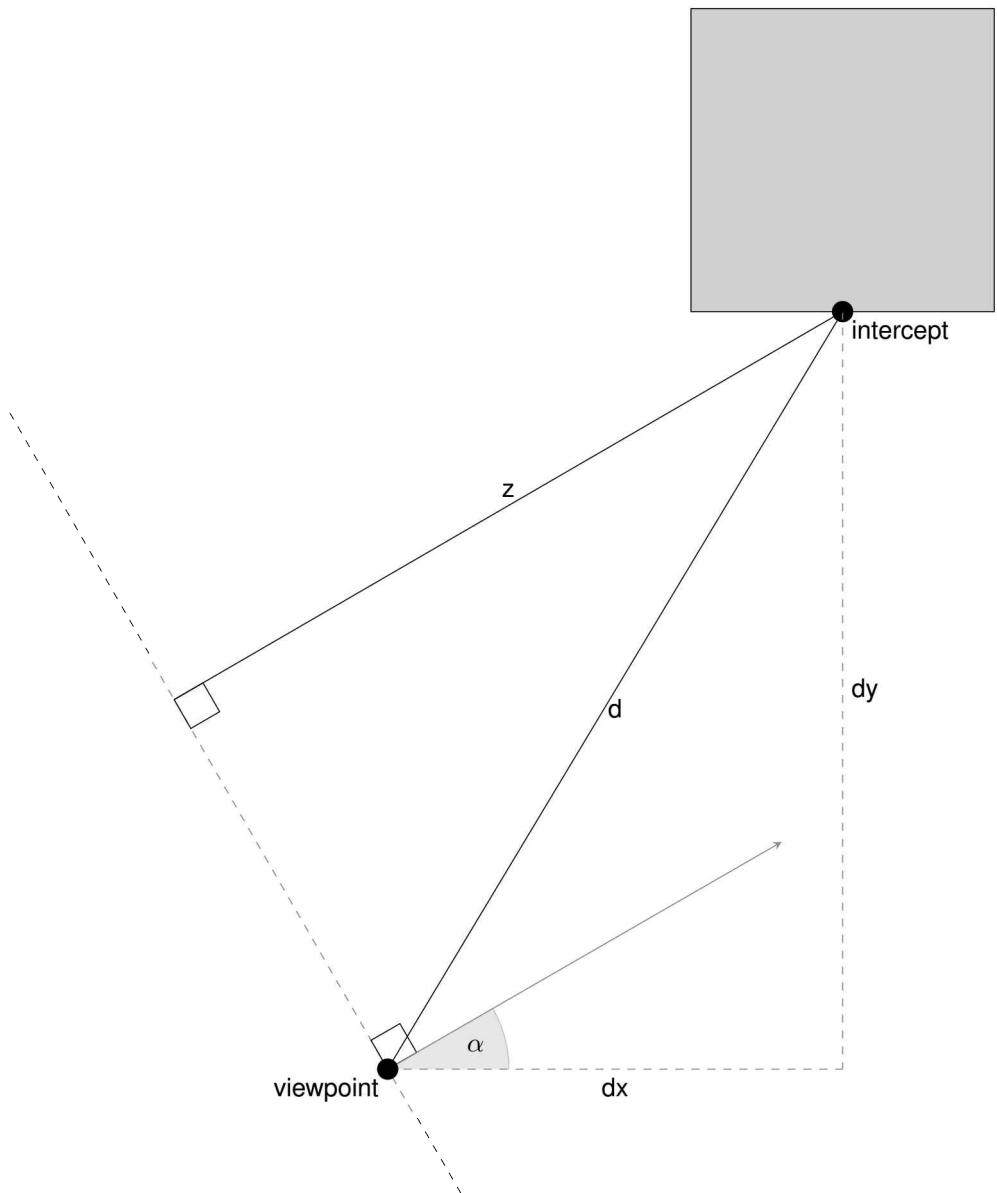


**Figure 4.49:** Fish eye effect: AAAAARGH

At 12 feet, the distortion is straight up unpleasant with a claustrophobic touch.

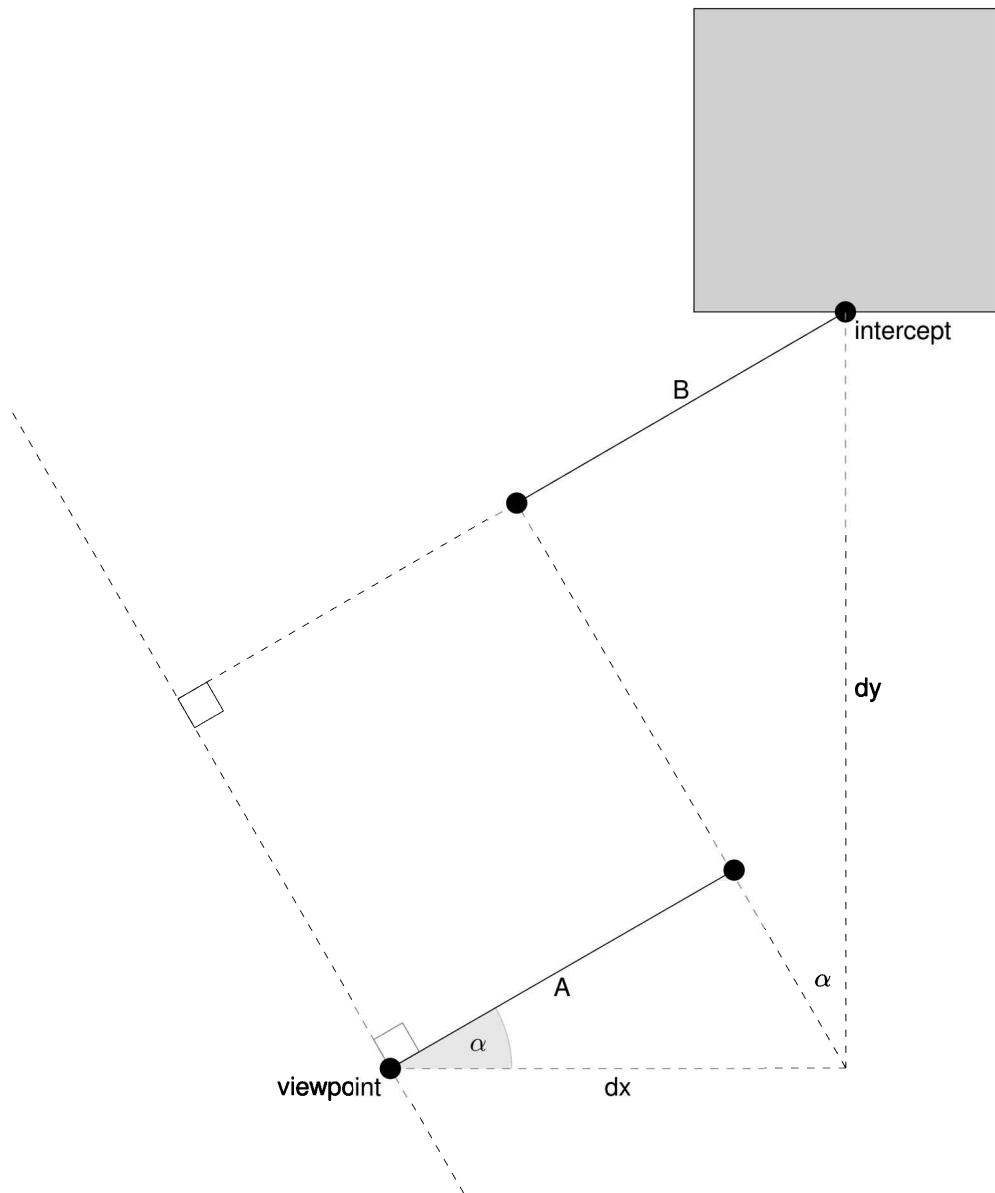
To avoid this distortion and produce a more pleasant rendition, what must be used is not the direct distance  $d$  but distance projected on the camera plane (perpendicular to the view direction ( $z$ )).





**Figure 4.50:** Direct distance  $d$  and projected distance  $z$ .

This projection ( $z$ ) is mathematically hard to calculate in one go (especially with fixed point). The trick is to break it down into two components and use the mnemonic SOH-CAH-TOA.



**Figure 4.51:** Projected distance  $z$  is calculated via two sub-components.

Using CAH gives  $A = dx * \cos(\alpha)$  and SOH gives  $B = dy * \sin(\alpha)$ .

Adding them together becomes:

$$z = A + B = dx * \cos(\alpha) + dy * \sin(\alpha).$$

Since dx and dy are not distances but vectors (with a sign) the equation becomes:

$$z = A + B = dx * \cos(-\alpha) + dy * \sin(-\alpha)$$

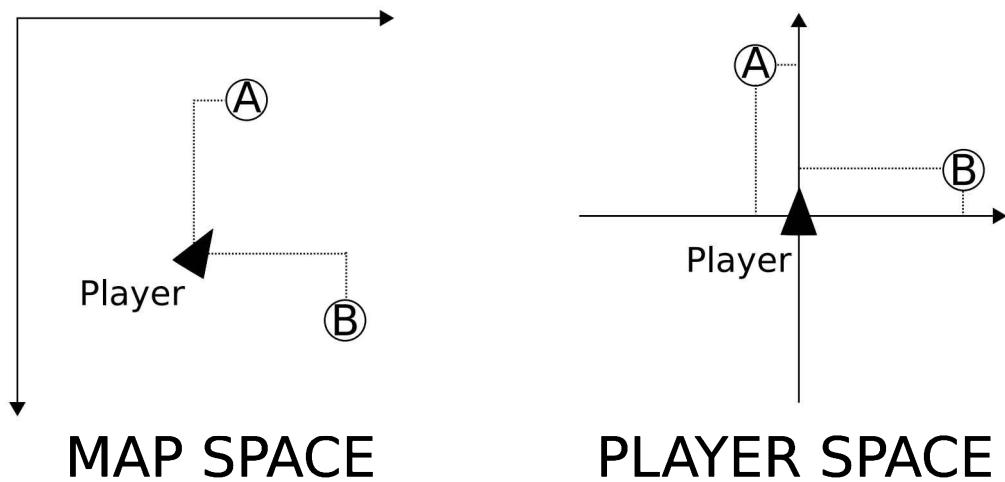
which simplified becomes:

$$z = A + B = dx * \cos(\alpha) - dy * \sin(\alpha)$$

For people who like equations, the overall operation can be seen as the multiplication of a rotation matrix with the vector intercept (dx,dy). It would rotate coordinates from map space to player space (where axis orientation is dictated by the player view angle).

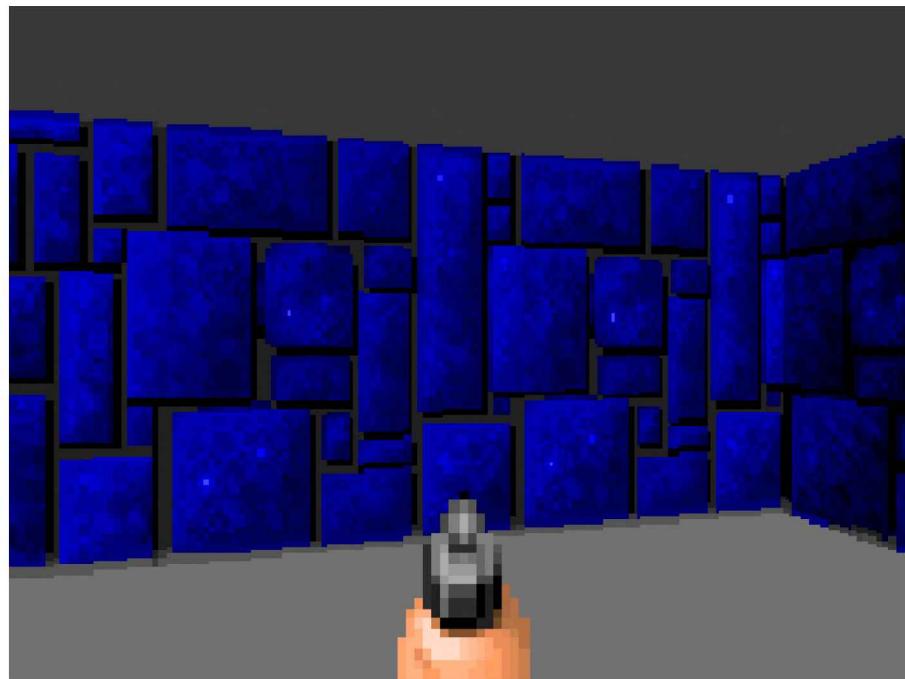
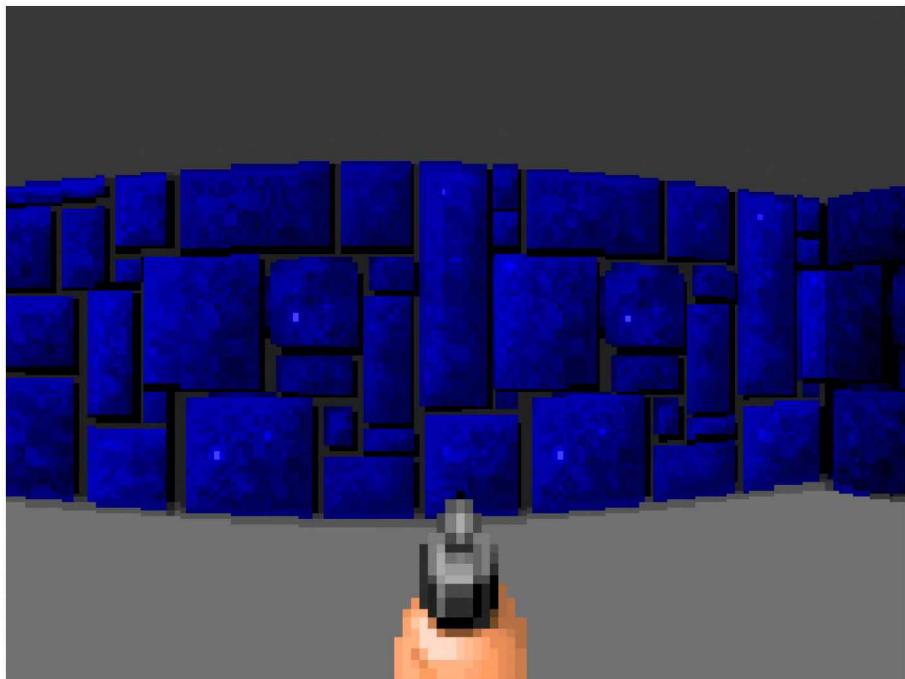
$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} * \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} dx * \cos(\alpha) - dy * \sin(\alpha) \\ dx * \sin(\alpha) + dy * \cos(\alpha) \end{bmatrix}$$

I personally find the graphic explanation with SOH-CAH-TOA clearer.



**Figure 4.52:** Translating from map space to player space.

This correction is enough to give pleasant straight lines for the walls. To demonstrate the difference, here are a few screenshots showing two versions of the same scene next to each other. First uncorrected (with fisheye) and then with projected distance  $z$  instead of direct distance  $d$ .





## 4.7.6 Drawing Walls

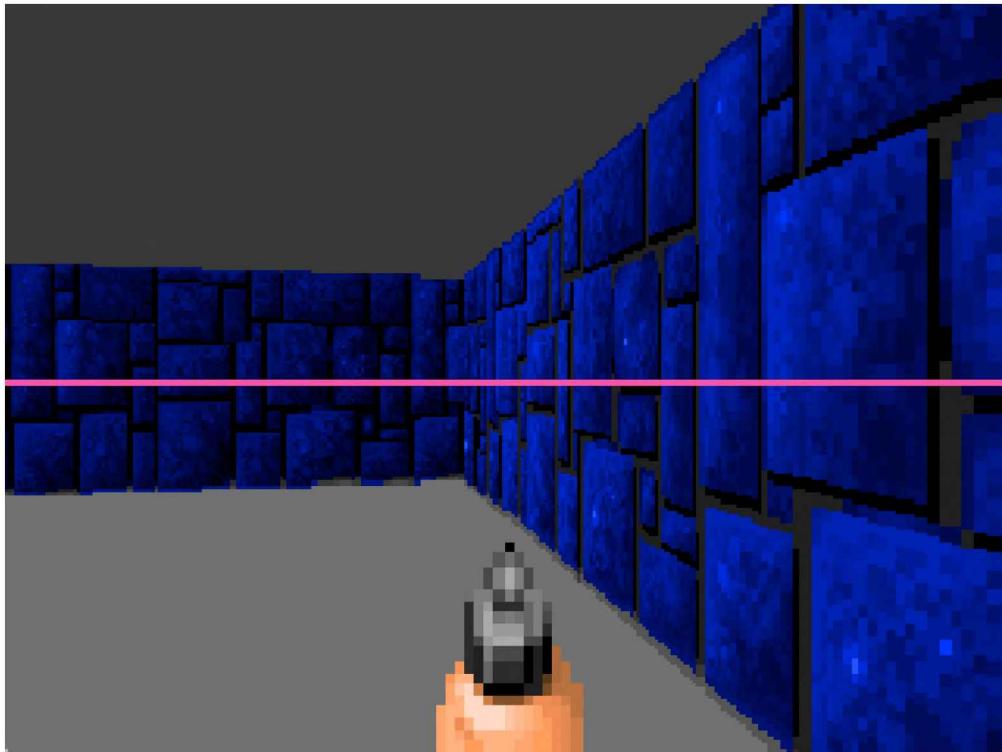
With column height calculated, all that remains is to draw a column of textured pixels.

That may sound easy but is in fact difficult on a CPU with as little power as a 386. Scaling a column of 64 texels is expensive. It turns out you need a few optimizations to do it fast enough. This is where Wolfenstein 3D leaves other 3D engines of the era in the dust. The secret to its speed lies in two tricks called:

- Compiled scalers
- Deferred column rendering

### 4.7.6.1 Compiled Scalers

All columns of pixels representing the walls are centered vertically and either magnified or minified. The goal is to scale a column of 64 texels to any height ranging from 2 pixels to the max 3D canvas height (152 pixels); as quickly as possible.



As the photoshopped pink line shows, each column of pixels for a wall is centered vertically with no offset added. In order to do this, a naive approach would be to use a generic routine looking something like this.

```
void scaleTextureToHeight (int height , void * src , void *
    dst ){
    fixed_t src_cursor = 0; // 24:8 format
    int dest_cursor = 0;
    fixed_t step = FixedDiv(64, height);
    while (height > 0) {
        if (dst_not_clipped(dest_cursor)) {
            dst[dest_cursor] = src [src_cursor >> 8];
        }
        src_cursor += step;
        height--;
        dest_cursor++;
    }
}
```

This would involve a loop resulting in at least one `jmp`<sup>18</sup>, a fixed point accumulator, and intermediate variables. That would be a lot of instructions due mostly to the genericity of the function. Indeed this flexible function allows `scaleTextureToHeight` to accept any height from 0 to `INT_MAX`.

There is a faster way to do this which involves the usual RAM vs CPU tradeoff. In this case we can invest a little bit of RAM to gain a lot of CPU time. The idea is to make the scaler less generic and instead generate hard-coded functions.

```
void scaleTextureTo2(void* src, void* dst) {
    dst[0] = src[16];
    dst[1] = src[48];
}

void scaleTextureTo4(void* src, void* dst) {
    dst[0] = src[0];
    dst[1] = src[16];
    dst[2] = src[32];
    dst[3] = src[63];
}
```

The engine generates machine code at runtime when it starts; this happens in `Build-CompScale`.

---

<sup>18</sup>`jmp` always cause a pipeline flush.

```

// Call with
// DS:SI    Source for scale
// ES:DI    Dest for scale
unsigned BuildCompScale (int height, byte far *code)
{
    [...]
    work = (t_compscale far *)code;
    code = &work->code[0];
    [...]
    for (src=0;src<=64;src++) {

        if (not_result_in_written_pixel)
            continue;

        // mov al,[si+src]      (Read src into register al)
        *code++ = 0x8a;
        *code++ = 0x44;
        *code++ = src;

        for (magnification_size) {
            // mov [es:di+heightofs],al  (Write al to dest)
            *code++ = 0x26;
            *code++ = 0x88;
            *code++ = 0x85;
            *((unsigned far *)code)++ = startpix*SCREENBWIDTH;
        }
    }
    // retf
    *code++ = 0xcb;
}

```

Given a height, `BuildCompScale` generates x86 instructions in the `out` variable `code`. As a result, instead of having one generic function accepting any height, *Wolfenstein* has 256 functions with hard-coded heights. Hard-coding the height allows for unrolling the loop and reducing overhead.

With this optimization, minifying a column of 64 texels to a 2 pixel tall column takes 15 instructions. Minifying to 4 pixels take 25 instructions. Magnifying to 128 pixels takes 705 ( $64 \times 11 + 1$ ) instructions. A precompiled scaler function has a fixed cost of 7 instructions per pixel when minifying and  $3 + 4 * scalingFactor$  instructions per pixel when magnifying. It doesn't get any faster than that.

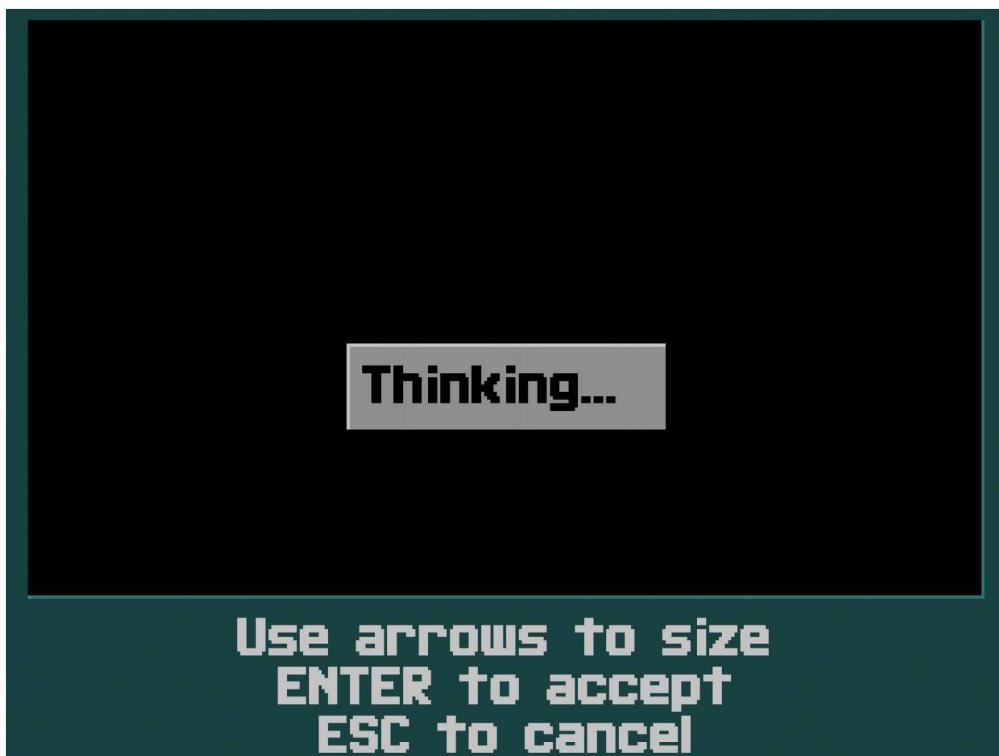
What is the RAM cost incurred by precompiling and caching these scalers? When render-

ing to its maximum dimension in the 3D canvas, the engine generates all scalers of even height from 2 to 512<sup>19</sup>. That's 255 scalers totaling 178,479 bytes of generated instructions. This cost was deemed too high.

To save RAM, past size 76 only every other even size is generated (2,4,6,...,72,74,76) and (78,82,86,...,504,508,512). This trick generates only 136 scalers for a total instruction size of 83,160 bytes.

Skipping compiled scaler generation involves using the wrong scaler when one is not available and introduces small visual artifacts, but they are barely noticeable.

Notice that scalers are generated at startup but if the 3D canvas' dimensions are changed they have to be re-generated. This is what happens after a resize while the "thinking" screen is showing.



---

<sup>19</sup>512 is taller than the height of the 3D canvas (which is 152). However, the engine needs to render sprites magnified to the point where sprites are clipped vertically.

#### 4.7.6.2 Deferred Column Drawing

The second level of optimization is based on deferred rendering. When a column of pixels meets certain conditions, it is buffered and rendered later in a batch, leveraging the VGA mask to save write operations.

A simple raycaster with compiled scalers would have looked like this.

```
for (int x=0 ; x< 320 ; x++) {
    castRay();
    height = CalculateWallHeight();
    drawColumn(x, height);
}
```

Instead, the engine buffers what to draw as follows.

```
for (int x=0; x<320 ; x++){
    castRay();
    if (raySimilarToOnesInBuffer){
        AddColumnToBuffer();
        continue;
    } else {
        DrawBuffer();
        height = CalcWallHeight();
        AddColumnToBuffer();
    }
}
```

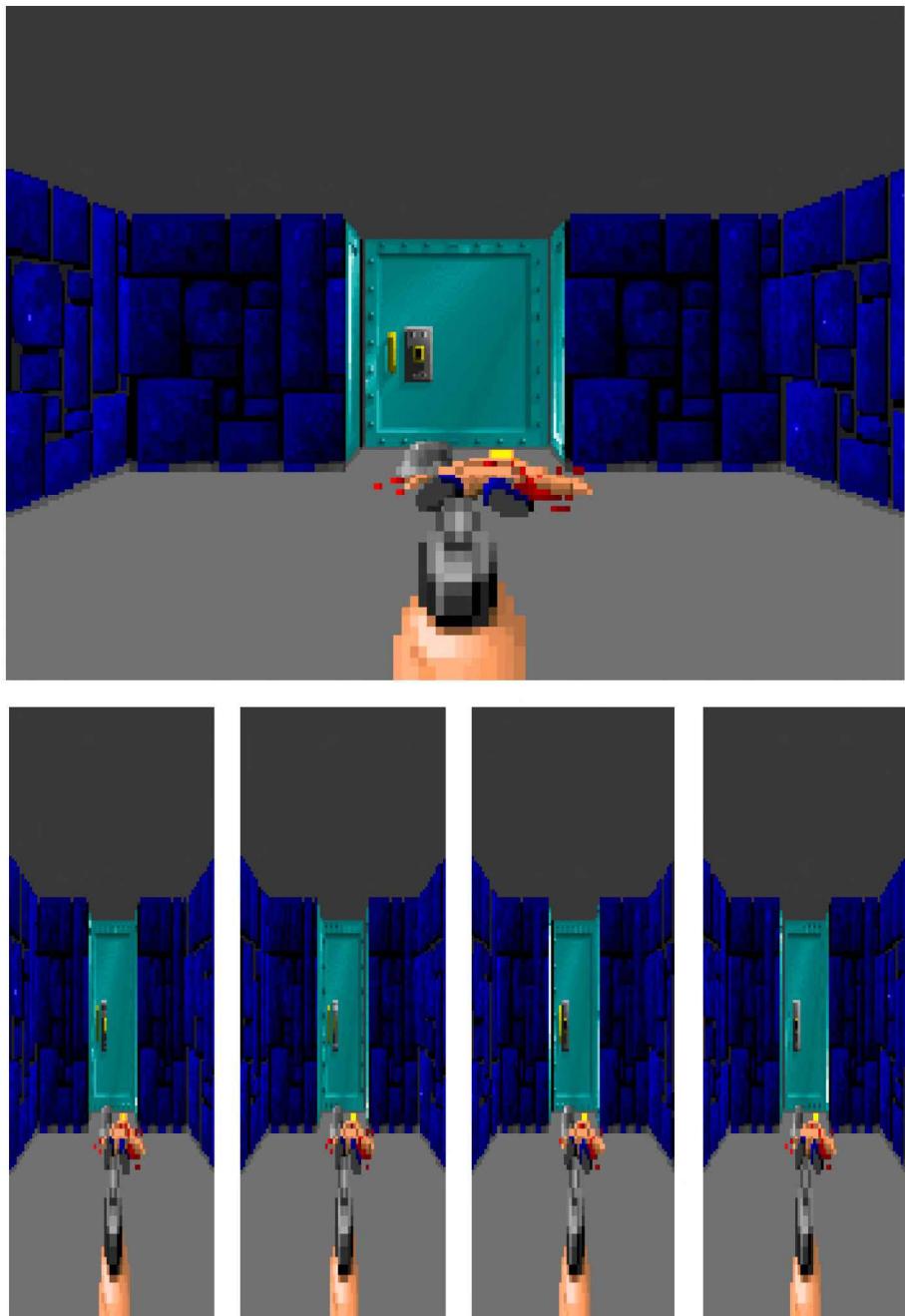
This buffering is done because the renderer allows itself to cheat a little when two rays are deemed similar enough. If consecutive rays share characteristics, they are grouped together and drawn at the same height, regardless of their exact distance from the player's point of view. This process introduces a little distortion but unlocks a formidable speed increase: the engine can write multiple columns on the screen, up to eight pixels in 3 writes.

Rays are similar if they hit the same wall and result in the same v horizontal texture coordinate. In short, this optimization leverages wall magnification.

The best case for this trick is shown in the following screenshots. The player is as close as possible to a wall. In this example, the wall completely covers the left part of the screen. The magnification is obvious given how texels spread across multiple pixels horizontally. In this case, multiple consecutive rays hit a texture at the same horizontal coordinate.



This trick is only possible because of an interesting property of the VGA bank layout. If you take a look on page 186, you see a 3D view with its associated bank storage underneath. Each bank looks like a compressed version of the screen because it stores every fourth column, 80 columns to be exact. Bank 0 stores column 0, column 4, column 8, and so on. Bank 1 stores column 1, column 5, column 9 and so on. Column 0 and Column 1 are at the same address in bank 0 and bank 1.



To achieve this trick, the engine must be careful in factoring in the four bank bytes align-

ment and the position of the columns on screen. The details of this are in the method ScalePost in WL\_DRAW.C.

```

void    near ScalePost (void)           // VGA version
{
    ...
    // scale a byte wide strip of wall
    asm  mov  bx,[postx]                // posx = x coordinate..
    asm  mov  di,bx                   // . where to draw
    asm  shr  di,2                  // X in bytes
    asm  add  di,[bufferofs]

    asm  and  bx,3
    asm  shl  bx,3                  // bx = pixel*8+
postwidth
    asm  add  bx,[postwidth]

    // First pass.
    asm  mov  al,BYTE PTR [mapmasks1-1+bx]
    asm  mov  dx,SC_INDEX+1
    asm  out  dx,al                  // set VGA bank mask
    asm  lds  si,DWORD PTR [postsource]
    asm  call DWORD PTR [bp]         // call compiled scaler

    // Second pass.
    asm  mov  al,BYTE PTR [ss:mapmasks2-1+bx]
    asm  or   al,al
    asm  jz   nomore
    asm  inc  di
    asm  out  dx,al                  // set VGA bank mask
    asm  call DWORD PTR [bp]         // call compiled
scaler
    ...
    // Third pass.
nomore:
}

```

The engine will group up to 8 columns together. Because of the VGA alignment it can take up to three passes to write them all. Since there can be multiple combinations of VGA bank alignment and number of pixels to draw, the VGA mask is precalculated and looked up at runtime. Each pass has its own settings. A pass runs only if the VGA mask is not zero.

```

byte mapmasks1[4][8] = {
{1 ,3 ,7 ,15,15,15,15,15} ,
{2 ,6 ,14,14,14,14,14,14} ,
{4 ,12,12,12,12,12,12,12} ,
{8 ,8 ,8 ,8 ,8 ,8 ,8 } };

byte mapmasks2[4][8] = {
{0 ,0 ,0 ,0 ,1 ,3 ,7 ,15} ,
{0 ,0 ,0 ,1 ,3 ,7 ,15,15} ,
{0 ,0 ,1 ,3 ,7 ,15,15,15} ,
{0 ,1 ,3 ,7 ,15,15,15,15} };

byte mapmasks3[4][8] = {
{0 ,0 ,0 ,0 ,0 ,0 ,0 ,0} ,
{0 ,0 ,0 ,0 ,0 ,0 ,0 ,1} ,
{0 ,0 ,0 ,0 ,0 ,0 ,1 ,3} ,
{0 ,0 ,0 ,0 ,0 ,1 ,3 ,7} };

```

Each `mapmasks` array is used during each pass X as:

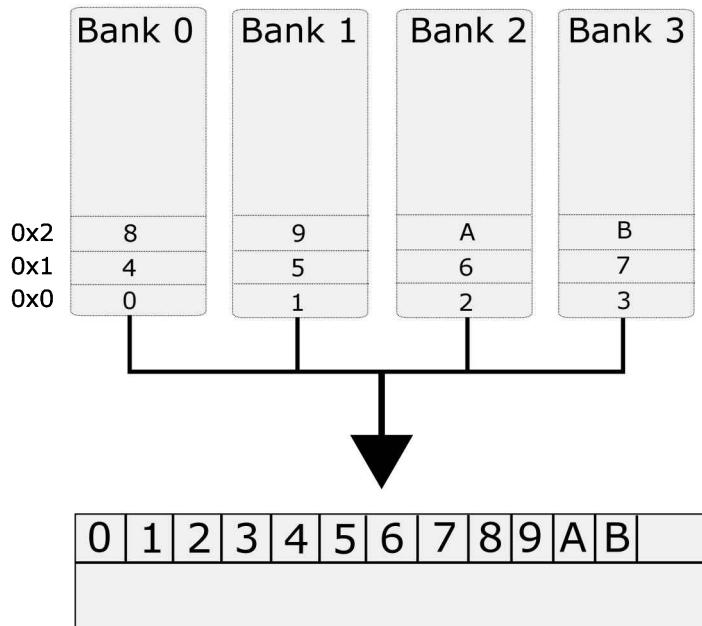
`VGA_MASK = mapmasksX[first_column_x_coordinate%4][numcolumn - 1]`

The engine does an early return when `VGA_mask` is equal to 0 (which is equivalent to "no more pixels are to be drawn"). This is easier to demonstrate with drawings. First a recap of how the VGA byte mask works.

- Bank 0 is selected when bit  $1 \ll 1$  (1) is set to 1.
- Bank 1 is selected when bit  $1 \ll 2$  (2) is set to 1.
- Bank 2 is selected when bit  $1 \ll 3$  (4) is set to 1.
- Bank 3 is selected when bit  $1 \ll 4$  (8) is set to 1.

Unused	Bank3	Bank2	Bank1	Bank0
	8	4	2	1

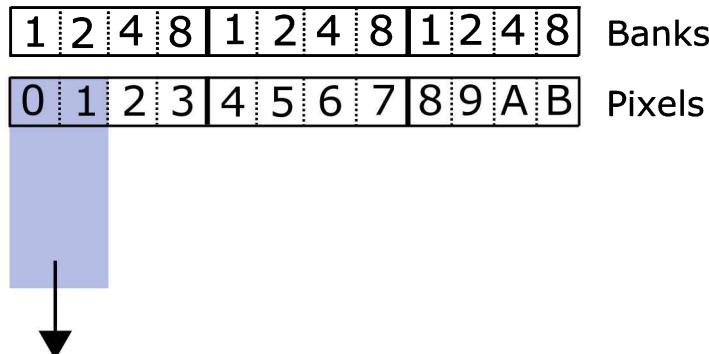
With these in mind, here are a few examples using the following layout where the engine wants to draw a column of pixels.



**Figure 4.53:** All following examples are based on this VRAM/Screen layout.

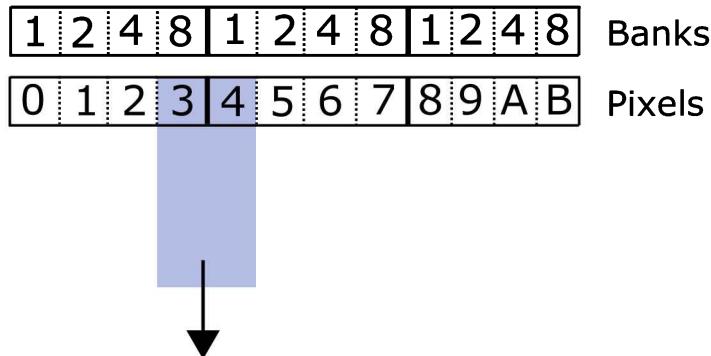
Drawing two columns (under pixel 0 and 1) can be done with only one pass using a VGA mask set to 3 to write in banks 0 and 1.

```
PASS1: mapmasks1 [0] [1] = 3
PASS2: mapmasks2 [0] [1] = 0 -> EXIT
```



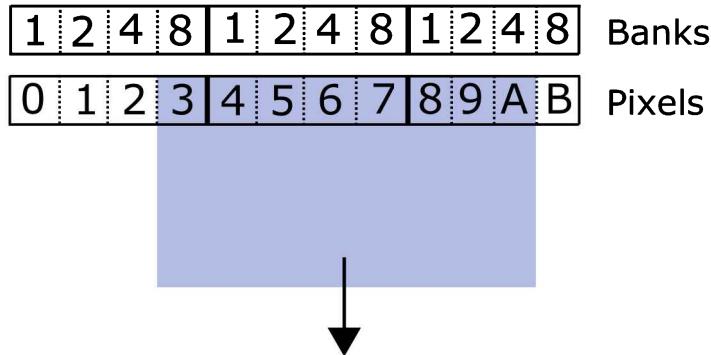
If the two columns are not properly aligned (like columns under pixels 3 and 4), the mask is of no help. Two passes with mask set to 8 and then 1 will be needed.

```
PASS1: mapmasks1 [3] [1] = 8
PASS2: mapmasks2 [3] [1] = 1
PASS3: mapmasks3 [3] [1] = 0 -> EXIT
```



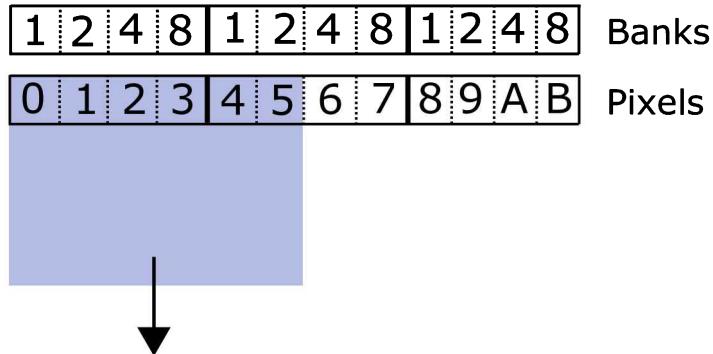
In the worst case scenario, the engine needs to draw eight columns of pixels, under pixels 3,4,5,6,7,8,9 and A. Because of the poor alignment, three passes are needed with mask set to 8, 15 and, 7.

```
PASS1: mapmasks1 [3] [7] = 8
PASS2: mapmasks2 [3] [7] = 15
PASS3: mapmasks3 [3] [7] = 7 -> EXIT
```

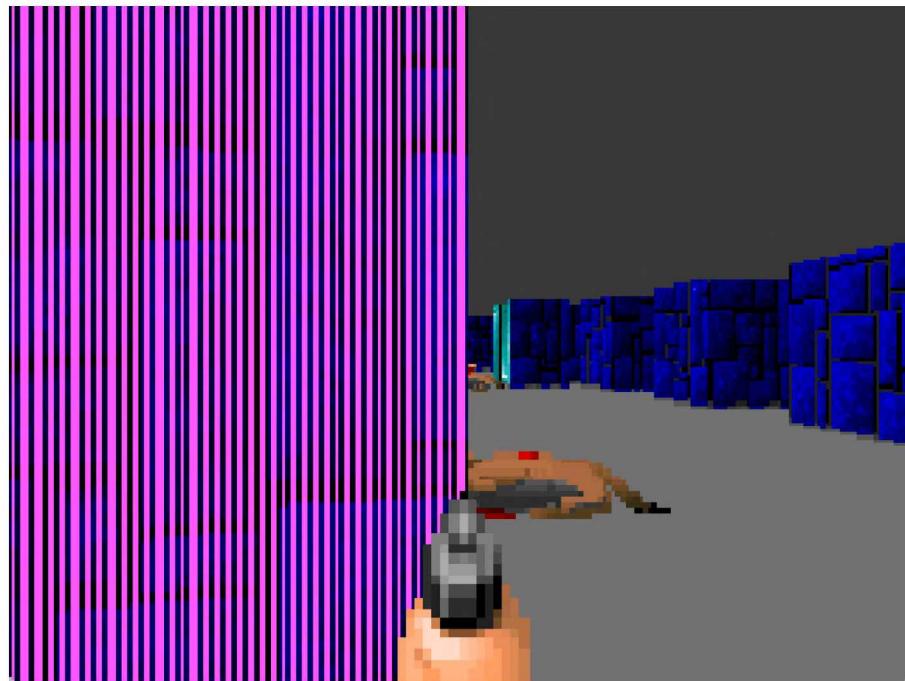


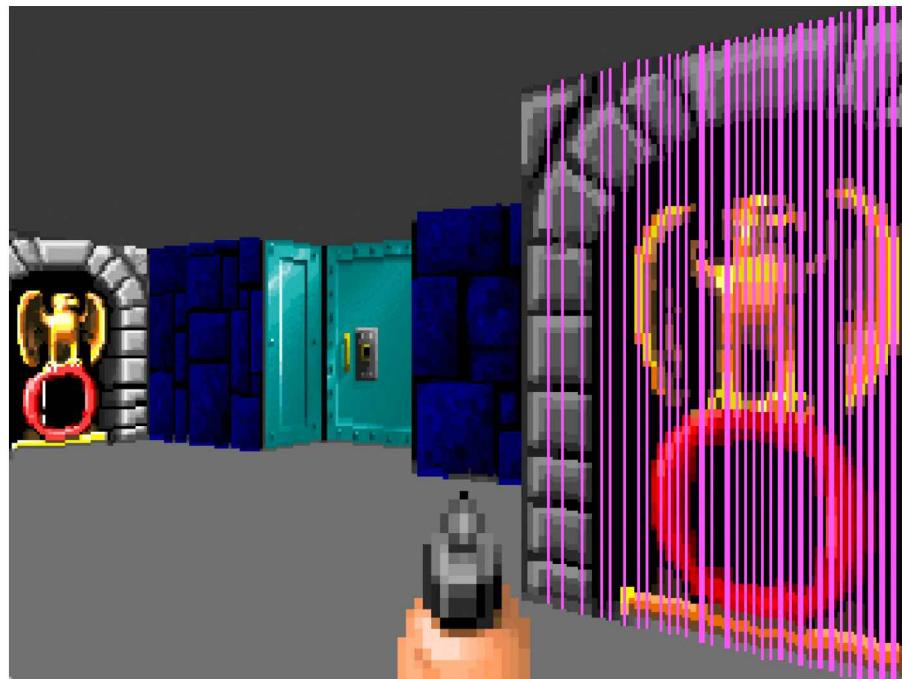
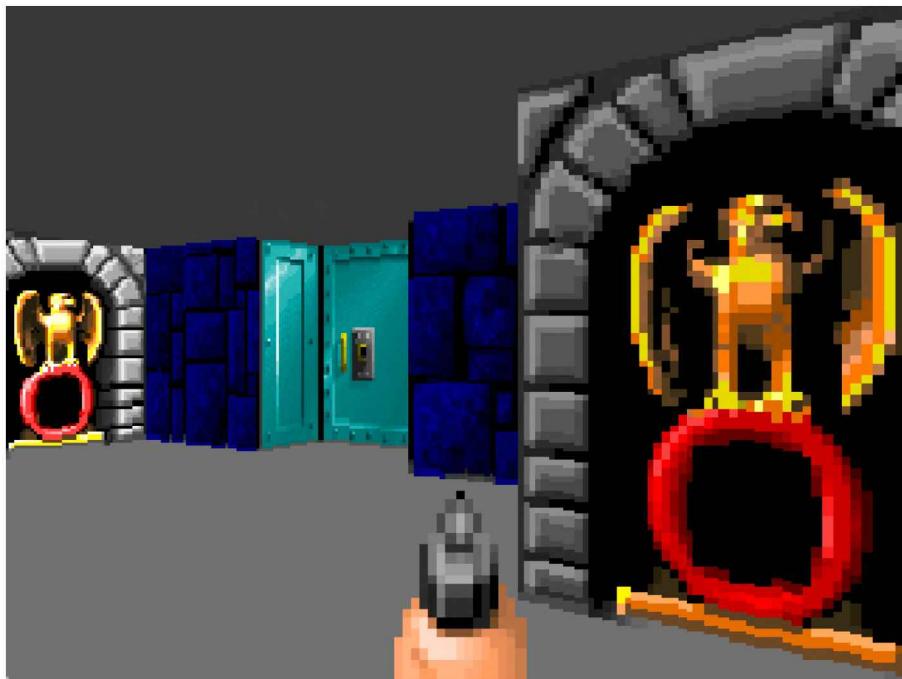
A better case where two passes with mask set to 15 then 3 allows drawing 6 columns of pixels :

```
PASS1: mapmasks1 [0] [5] = 15  
PASS2: mapmasks2 [0] [5] = 3  
PASS3: mapmasks3 [0] [5] = 0 -> EXIT
```



To visualize the performance gain in real cases, the engine has been modified to draw columns of pixels which were written "for free" thanks to the VGA mask manipulation in pink. We can see the performance improvement is significant, with 50% of write operations avoided.





This technique helps solve for scenarios where a large number of pixels have to be written due to the size of the wall. It only really shines when a lot of magnification occurs. For walls far away (minified) this technique doesn't help at all, but this matters less as small walls are cheap because they have fewer pixels to render.

**Trivia :** Generating code at runtime may sound like an outdated technology. However, it was used as recently as 2016 when Android emulator used PixelFlinger to render the entire phone screen.

#### 4.7.6.3 Texturing

A subtle but extremely efficient trick used to improve the quality of the wall rendering is pre-baked light texturing. Wall texture assets were generated twice by the artists: once lit and once unlit.

At runtime, upon finding ray-wall intersection, if the ray hits a vertical wall (looking at the map from above) on the Y axis, the engine uses a lit texture. If the ray hits an horizontal wall (on the X axis), it uses the unlit version of the same texture. The difference is not obvious at first, but with the same scene rendered side by side with and without this effect, the visual difference is striking, giving the scene more realism because of this directional-light effect.



**Figure 4.54:** Light and Dark wood textures.



**Figure 4.55:** Above: Baked texture off. Below: Baked texture on.



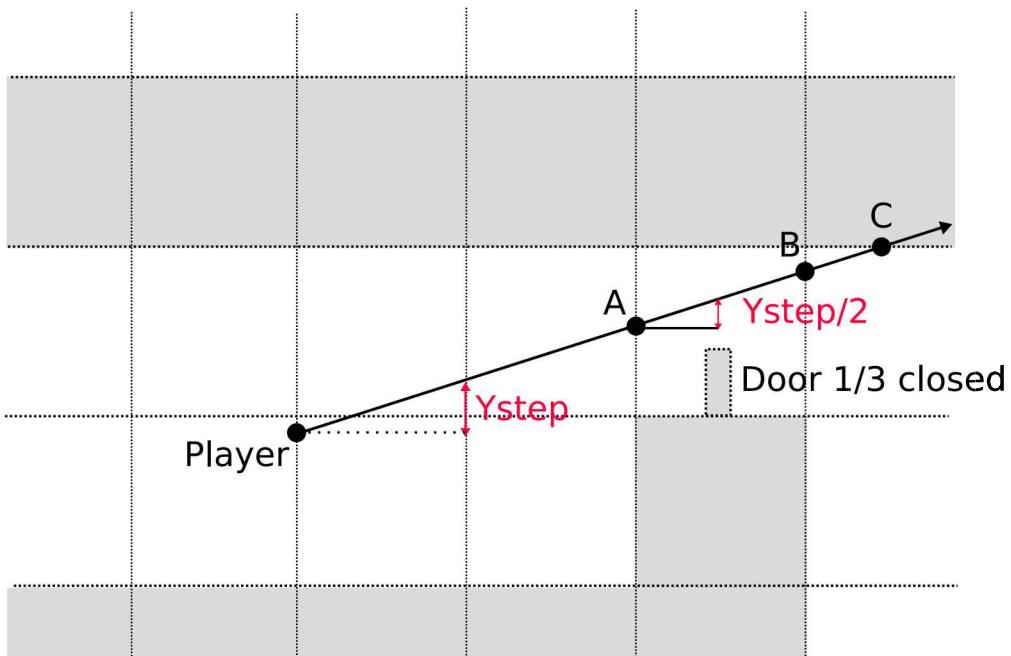
#### 4.7.6.4 Doors

Doors are rendered directly via the raycaster. Therefore, they have no thickness as that would have been much more complicated to implement.



Upon hitting a "DOOR" tile, the raycaster consults the `doorposition` array. The raycaster is able to calculate how far along a door is opened and if it should either stop at the door or traverse through.

```
#define MAXDOORS 64 // max number of sliding doors  
  
// leading edge of door 0=closed, 0xffff = fully open  
unsigned doorposition[MAXDOORS];
```



**Figure 4.56:** A ray traversing a partially opened door.

Testing if a ray is stopped by a door or not costs almost nothing:

$$A_X + \frac{ystep}{2} < \text{doorposition}[doorIndex]$$

In case of success, the ray traverses the wall tile and continues being tested against the grid. If the ray is a hit, the coordinates of the point of interception are calculated as  $\text{Intercept}_X = A_X + \frac{ystep}{2}$  and  $\text{Intercept}_Y = A_Y + \frac{\text{TILE\_SIZE}}{2}$  and passed to the renderer.

#### 4.7.6.5 Push Walls

Push walls are also implemented via the raycaster. Like doors, these moving walls are also treated as special cases. If the push wall has been activated, the raycaster adds an offset to the ray intercept coordinates.

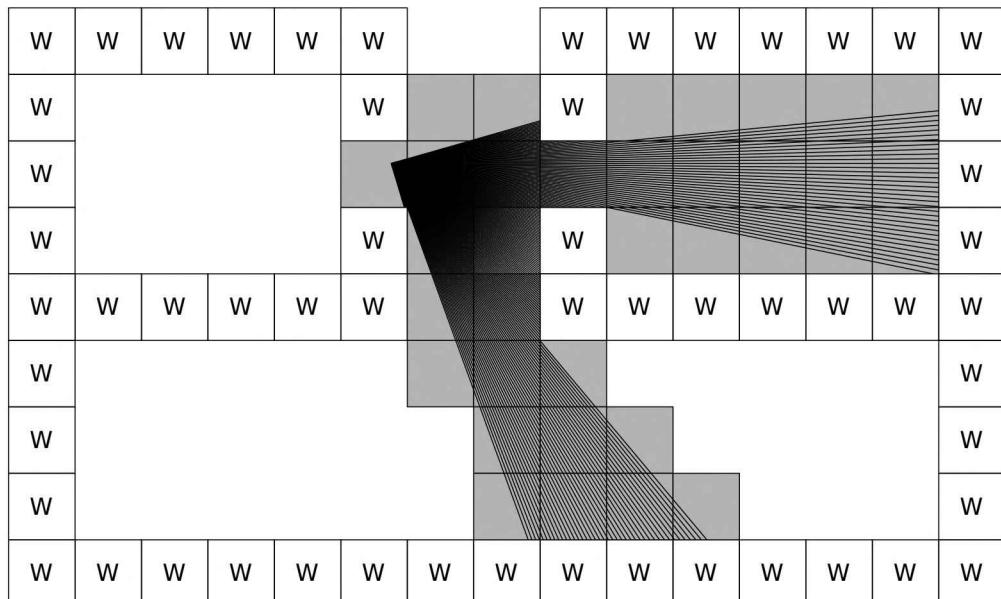
### 4.7.7 Drawing Sprites

Once walls are drawn, it is time to render sprites such as enemies, items (ammo, weapons), and decorations (lamps, table, etc). This is a three step operation.

1. Identify which sprites are visible.
2. Determine which part of the sprite is visible (not hidden by a wall).
3. Draw what is visible.

#### 4.7.7.1 Visible Sprite Determination

Building a list of visible sprites is done indirectly by leveraging information gathered by the raycaster. This relies on the assumption that visible sprites are on visible tiles. The raycaster marks all tiles visited by each ray while it travels the map looking for walls.



**Figure 4.57:** Ray casted and tiles marked as visible.

Visible tile tracking is done in the simplest way with a 64x64 boolean array indicating if a tile was visited.

```
#define MAPSIZE 64 // maps are 64*64 max
extern byte spotvis[MAPSIZE][MAPSIZE];
```

At the beginning of each frame, the engine clears the array.

```
asm mov ax,ds
asm mov es,ax
asm mov di,OFFSET spotvis // Array to clear
asm xor ax,ax    // Put 0 in ax
asm movc x,2048 // repeat next instruction 64*64/2
asm rep stosw // store ax at es:di
```

The ray caster (AsmRefresh) writes true in the spotvis array as the ray progresses towards a wall. The array is used to select only visible objects:

```
#define MAXVISABLE 50

typedef struct {
    int viewx,viewheight,shapenum;
} visobj_t;

visobj_t vislist[MAXVISABLE],*visptr,*visstep,*farthest;

void DrawScaleds (void) {
    int numvisible=0;
    visptr = &vislist[0];

    // Use spotvis[] to add objects to visptr. Increase
    // numvisible

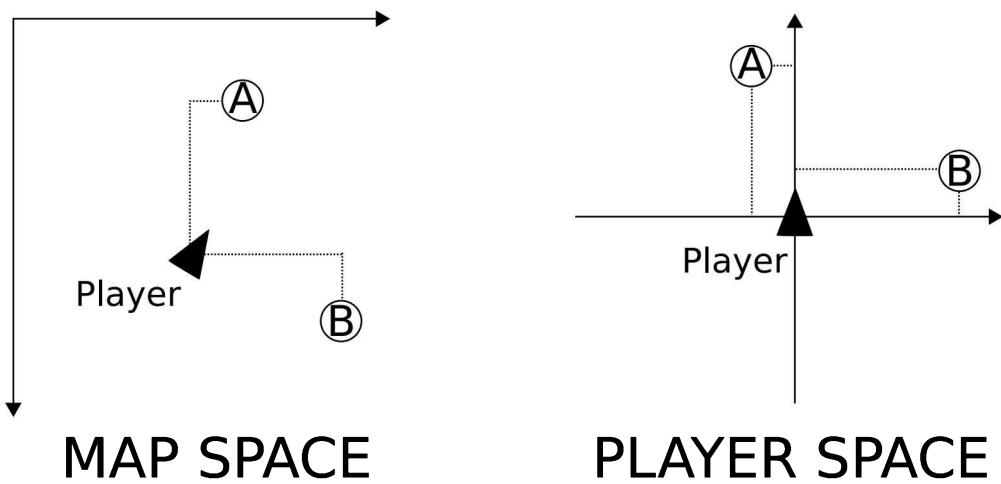
    // draw from back to front
    for (i = 0; i<numvisible; i++){
        least = 32000;
        for (visstep=&vislist[0] ; visstep<visptr ; visstep++){
            height = visstep->viewheight;
            if (height < least){
                least = height;
                farthest = visstep;
            }
        }
        ScaleShape(farthest->x,farthest->id,farthest->height);
        farthest->viewheight = 32000;
    }
}
```

While the way the visible tile array is used is not the most subtle part of the engine (it runs

in  $\mathcal{O}(n^2)$  time), it works well due to the low number of sprites. All sprites on the map are tested for visibility with `spotvis`, then their height is calculated. They are all added to an array unsorted. A final double loop draws all sprites from far to near (using sprite height to determine distance on each iteration).

#### 4.7.7.2 Rendition

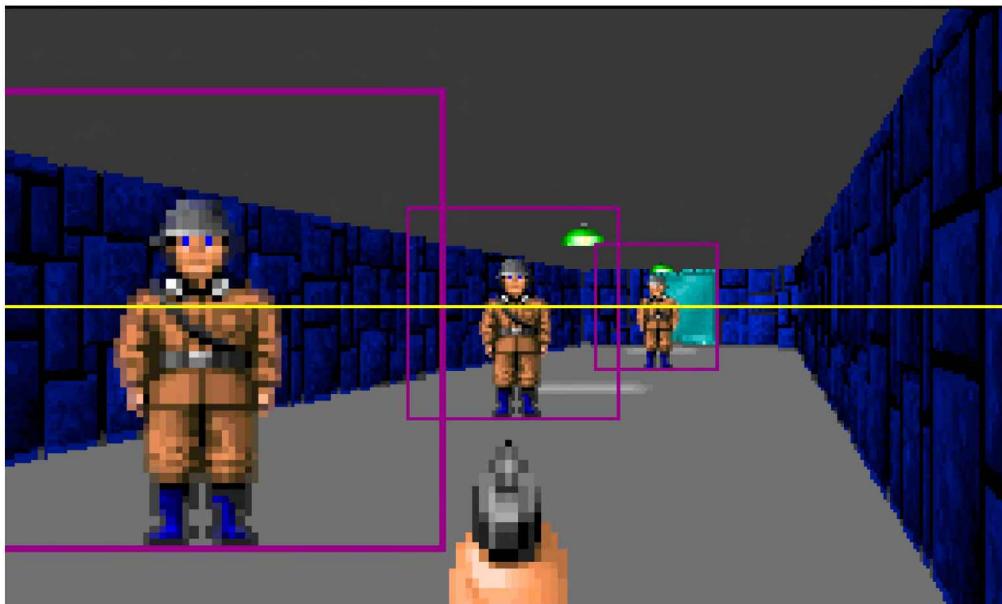
Each sprite is rendered individually in the function `ScaleShape (int xcenter, int shapenum, unsigned height)`. The sprite is transformed from map space to player space using the same SOH-CAH-TOA math seen in the "Drawing Walls" section.



The X coordinate is used to place a sprite horizontally on the screen, while the Y coordinate is used for clipping. Like walls, sprites don't need to be placed vertically since they are drawn in the same 64x64 texels space.

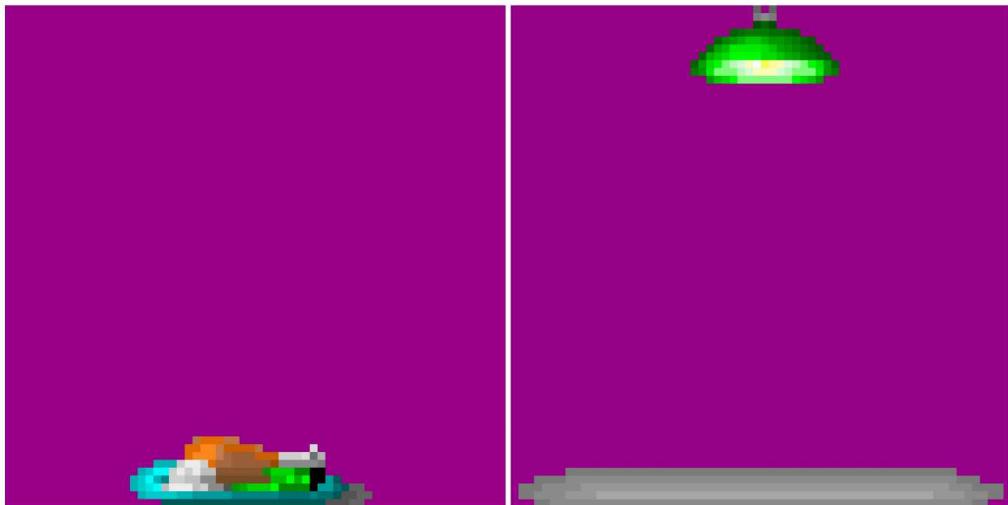


As shown in the following screenshot, scaling a vertically centered sprite is enough to give the illusion of perspective.



Sprites are special, however. Walls fill the full 64x64 space and are fully opaque, while sprites have transparent parts. For example, the "food" sprite is only 9 texels tall. The

lamp sprite uses the full height but the middle is transparent.



**Trivia :** Sprites with a lot of transparency (such as the two shown previously) would later turn out to be a major fillrate issue with the hardware accelerated renderer for the iOS port.

“

Wolfenstein (and Doom) originally drew the characters as sparse stretched columns of solid pixels (vertical instead of horizontal for efficiency in interleaved planar mode-X VGA), but OpenGL versions need to generate a square texture with transparent pixels. Typically this is then drawn by either alpha blending or alpha testing a big quad that is mostly empty space. You could play through several early levels of Wolf without this being a problem, but in later levels there are often large fields of dozens of items that stack up to enough overdraw to max out the GPU and drop the framerate to 20 fps. The solution is to bound the solid pixels in the texture and only draw that restricted area, which solves the problem with most items, but Wolf has a few different heavily used ceiling lamp textures that have a small lamp at the top and a thin but full width shadow at the bottom. A single bounds doesn't exclude many texels, so I wound up including two bounds, which made them render many times faster.

**John Carmack - Programmer**

”

### 4.7.7.3 Clipping

Like everything in the game, sprites are made of and drawn as columns. Before drawing each column in a sprite, the engine determines for each column if it is occluded by a wall. This is called clipping and is an easy step thanks to walls and sprites being in the same 64x64 coordinate system. Once the sprite's position is transformed in player space, height is generated based on the distance Y. That height is the same as calculated for the walls. Therefore to determine if a sprite column is behind a wall, the engine simply compares its height to the height of the wall.

In order to do that, the raycaster keeps tracks of each height calculated for each column in an occlusion array called `wallheight`.

```
#define MAXVIEWWIDTH 320
unsigned wallheight[MAXVIEWWIDTH];
```

The occlusion array is written to during raycasting.

```
// Write in methods HitVertWall, HitHorizWall, HitVertDoor,
// HitHorizDoor, HitHorizPWall and HitVertPWall
wallheight [pixx] = CalcHeight();
```

While drawing sprites, the occlusion array is read. If the height of a wall is greater than that of a sprite, it means the wall is in front of the sprite and the full sprite rendition is skipped.

```
if (wallheight [slinex] >= height)
    continue; // obscured by closer wall
```

Since the entire screen is refreshed on the next frame, the occlusion array does not need to be cleared at the beginning of a frame.

“

There was still a little more room for improvement in compiled scaler performance for wall textures: if the textures were reorganized so that they went "middle-out", you could avoid half the loads by doing a 16 bit read and writing out AL for the top pixels and AH for the bottom pixels.

”

The combination of the compiled scalers getting more efficient at higher magnifications and the VGA latch writes avoiding entire columns let Wolf maintain almost the same speed with big walls and small walls, which was important.

**John Carmack - Programmer**

#### 4.7.7.4 Drawing Things

Sprite rendering benefits from the same optimization we saw in walls (compiled scalers and deferred rendering). However, since sprites introduce transparency the techniques are slightly adjusted.

#### 4.7.7.5 Compiled Scalers

Visible sprite column rendition is also done with the compiled scalers, but the scalers cannot be used directly. A compiled scaler draws its speed from its absence of parameters. It is an unrolled loop hardcoded with x86 instructions to read from a texture 64 texels tall and write/scale a predetermined number of pixels. For example, compiled scaler #112 always reads 64 pixels of texture and magnifies it to 134 pixels on the screen. Transparency is not handled at all, used as is the scalers would draw transparent pixels in pink. To solve this problem, sprites are stored in a special way that allows tweaked compiled scalers to skip transparency.

A sprite is stored as an array of 64 entries. Each entry is a series of "commands" forming a column. A command features:

1. A vertical offset.
2. A vertical length.
3. A payload of texels.

The last command in a column is marked with an offset of 0x00.

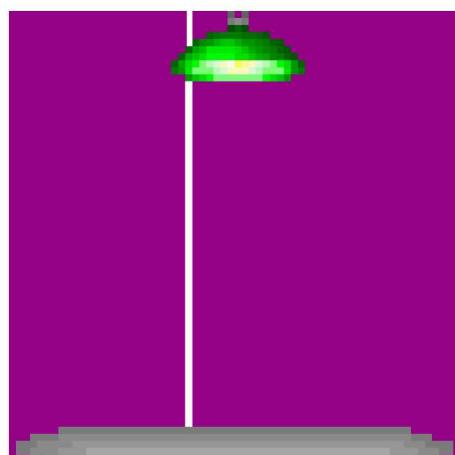
**Example :** Column #25 in the lamp sprite is made of 5 transparent pixels, followed by one sequence of 5 pixels for the lamp:



Followed by 49 transparent pixels, and finally one sequence of 5 pixels for the light halo.



Total: 64 texels. This column is encoded with 2 commands of 1 byte offset + 1 byte length + 5 bytes of payload. A payloadless command with length 0x00 marks the end. Total size is 17 bytes.



The idea is to prevent a scaler from consuming 64 texels by patching the x86 instructions with an early return instruction `ret`. In order to do that, the code generator also generates patch locations. The `t_compscale` structure containing a compiled scaler not only features the x86 instruction in code, it also features patching offset.

```
typedef struct
{
    unsigned   codeofs[65];
    unsigned   width[65];
    byte      code[];
} t_compscale;
```

There are 64 patch locations per scaler (one for each length of pixels from 0 to 63, stored in `codeofs[]`), which allows for writing a `RETF` instruction causing an early return. For each command in a sprite column, the engine looks up how many texels the command payload contains and where to patch the scaler, then saves the instruction at this location and overwrites it with a `RETF`. After the scaler returns (and the command payload has been rendered), the scaler is unpatched.

In the assembly optimized code, note how the engine knows where to patch the code thanks to a `BX` register which points to `codeofs`.

```
asm mov bx,[ds:bp]           ; table location of rtl to patch
asm or  bx,bx
asm jz  linedone             ; 0 signals end of segment list
asm mov bx,[es:bx]
asm mov dl,[es:bx]           ; save old value
asm mov BYTE PTR es:[bx],OP_RETF ; patch a RETF in
asm mov si,[ds:bp+4]          ; table location of entry spot
asm mov ax,[es:si]
asm mov WORD PTR ss:[linescale],ax ; call here to start
                                   scaling
asm mov si,[ds:bp+2]          ; corrected top of shape for this
                                   segment
asm add bp,6                  ; next segment list

asm mov ax,SCREENSEG
asm mov es,ax
asm call ss:[linescale]       ; scale the segment of pixels

asm mov es,cx                 ; segment of scaler
asm mov BYTE PTR es:[bx],dl   ; unpatch the RETF
asm jmp scalesingle          ; do the next segment
```

Patching the scaler is not enough. It only allows for consuming less than 64 texels. In order to draw a "hole" properly on screen, the engine also needs to know how many pixels to skip vertically in screenspace before starting the scaler on the next command. This is where the `width[]` array is used. The scaler generator also saves 64 entries to convert sprites' transparent height into screen space height.

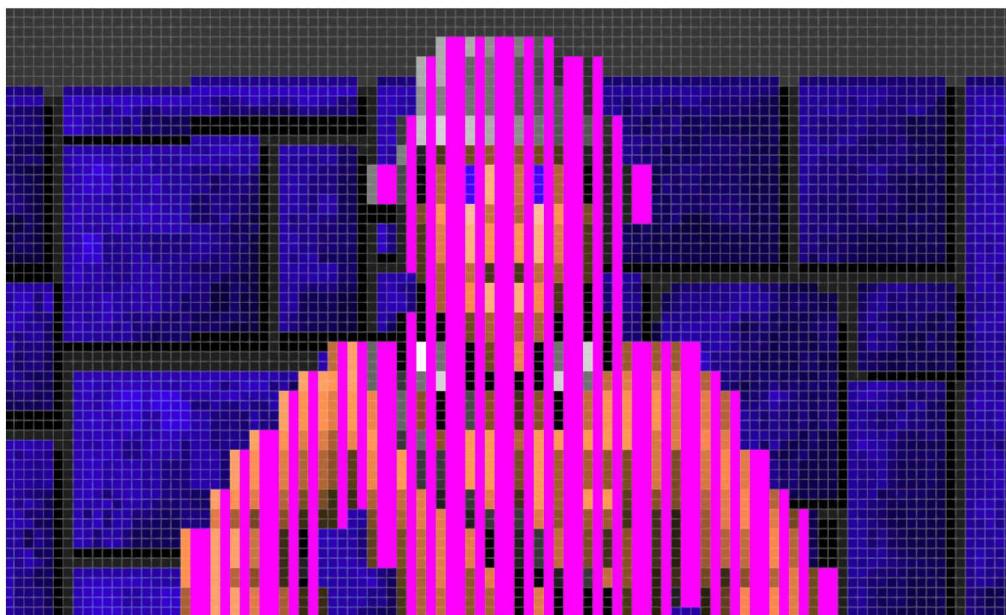
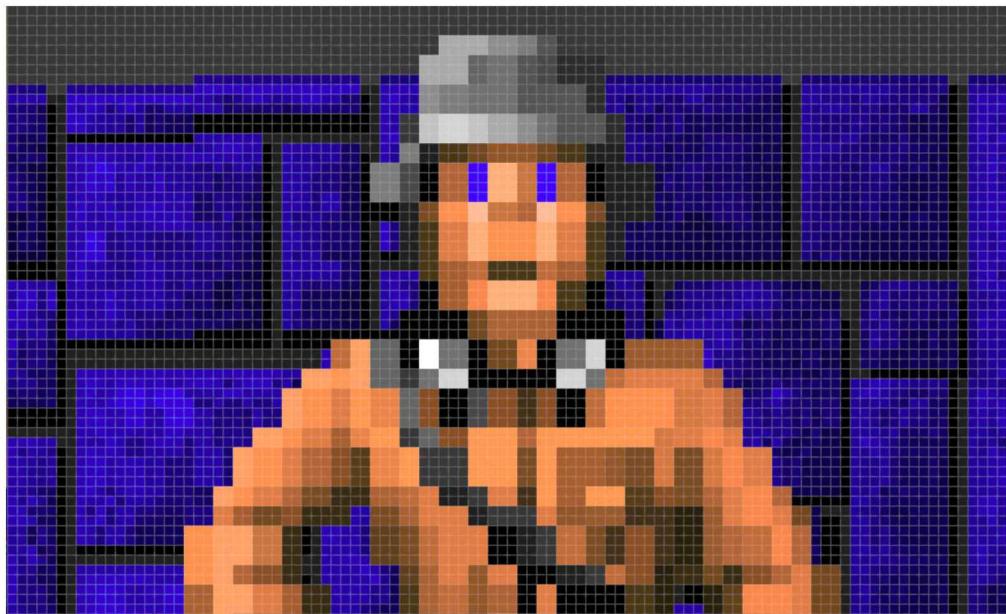
#### 4.7.7.6 Deferred Rendering

The same deferred drawing technique we saw for walls is also used for sprites, and is especially powerful when a sprite is magnified.



In this scene, the 64x64 guard sprite is magnified 2.2 times. By zooming in, we see that each column is repeated at least twice and sometimes three times. It is a perfect optimization case for VGA mask manipulation.

A modified version of the engine (which draws "free" columns in pink) is again used to demonstrate. In this case most columns were drawn only once. Magnification was a totally free operation despite more than twice the number of texels written to the screen.



### 4.7.8 Drawing Weapons

Drawing the weapon at the bottom of the canvas is straight forward. It uses the same type of rendering as sprites but with clipping disabled (in the function `ScaleShape`). The same compiled scaler and deferred rendering tricks are used.

### 4.7.9 A.I

To simulate enemies, some objects are allowed to "think" and take actions like firing, walking, or emitting sounds. Those thinking objects are called "actors".

Actors are programmed via a state machine. They can be aggressive, sneaky, or dumb (rockets for instance). To model their behavior, all enemies have an associated state:

- Standing
- Attack
- Path
- Pain
- Shoot
- Chase
- Die
- Special Boss

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

```
typedef struct      statestruct
{
    boolean      rotate;
    int          shapenum;    // Sprite to render on screen
    int          tictime;     // How long stay in that state
    void        (*think) (),(*action) ();
    struct      statestruct *next;
} statetype;
```

A guard in standing position always stays in the same state (`next` points to itself):

```
statetype s_grdstand = {true ,SPR_GRD_S_1,0 ,T_Stand ,NULL,&s_grdstand};
```

Some state chains are more complex, as with chasing:

```
statetype s_grdchase1 ={true ,SPR_GRD_W1_1,10 ,T_Chase ,NULL ,&s_grdchase1s};
statetype s_grdchase1s={true ,SPR_GRD_W1_1,3 ,NULL ,NULL ,&s_grdchase2 };
statetype s_grdchase2 ={true ,SPR_GRD_W2_1,8 ,T_Chase ,NULL ,&s_grdchase3 };
statetype s_grdchase3 ={true ,SPR_GRD_W3_1,10 ,T_Chase ,NULL ,&s_grdchase3s };
statetype s_grdchase3s={true ,SPR_GRD_W3_1,3 ,NULL ,NULL ,&s_grdchase4 };
statetype s_grdchase4 ={true ,SPR_GRD_W4_1,8 ,T_Chase ,NULL ,&s_grdchase1 };
```

“

The enemies in Wolf felt different based on the decision to close distance to the player either largest or smallest axis delta first - Closing smallest first made the brown shirts line up with you a long way away, making them easy to shoot. Closing largest first made the officers come at you in a ragged diagonal, as if they were dodging your shots.

**John Carmack - Programmer**

”

All types of enemies (including bosses) have their own state machine. They often share actions (e.g. T\_Stand and T\_Path) but also occasionally have their own.

What makes enemies interesting is how they trigger from standing to aggressive via T\_Stand. They have three ways to detect the player:

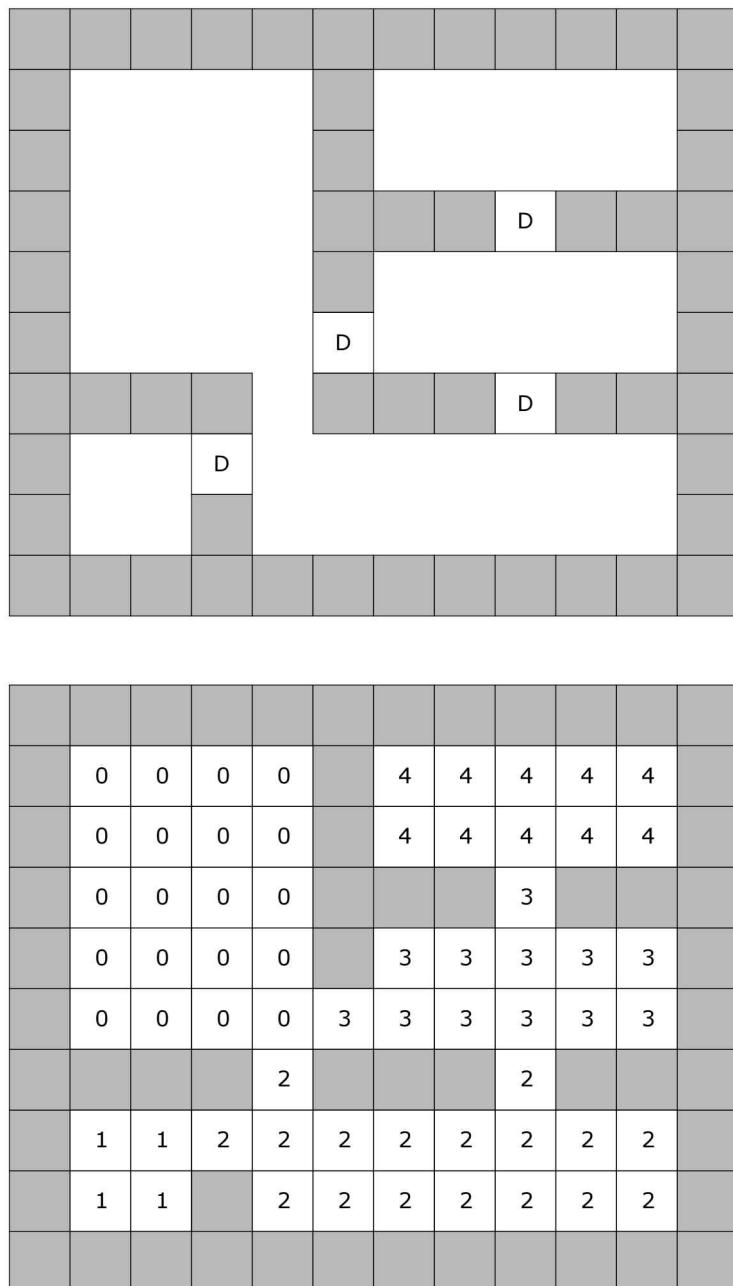
- Proximity
- Sight
- Noise

By far the most important stimuli and, what makes the player feel like the A.I is "smart", is reaction to noise.

#### 4.7.9.1 Sound Propagation

Early on the game teaches the player that enemies react to gunfire and seek out the source. Sound is an essential part of the experience and propagating it realistically in realtime is hard.

A floodfill algorithm could have been used but that would have been slow. To speed things up, maps are preprocessed. Each room delimits an area. At runtime the engine maintains a matrix of portals connecting areas and updates it when a door opens or closes. Determining if an enemy can hear the player's gunfire is then a simple lookup in a table.



**Figure 4.58:** A map generated by TED5 before and after preprocessing for audio propagation. After processing each block must belong to an area.

	0	1	2	3	4
0	-	0	0	0	0
1	0	-	0	0	0
2	0	0	-	0	0
3	0	0	0	-	1
4	0	0	0	1	-

At runtime the engine maintains a matrix of portals. Each time a door is opened or closed, the array `areaconnect[][]` is updated and `areabyplayer[]` array is populated recursively starting with `areabyplayer[player.tile] = 1`. If player is in area 4 and open the door to 3, `areabyplayer[]` will look as follow.

	0	1	2	3	4
	0	0	0	1	1

Determining if an enemy in area X can hear the player is achieved with an inexpensive lookup in `areabyplayer[X]`.

```
#define NUMAREAS 37

byte far areaconnect[NUMAREAS][NUMAREAS];
boolean areabyplayer[NUMAREAS];
```

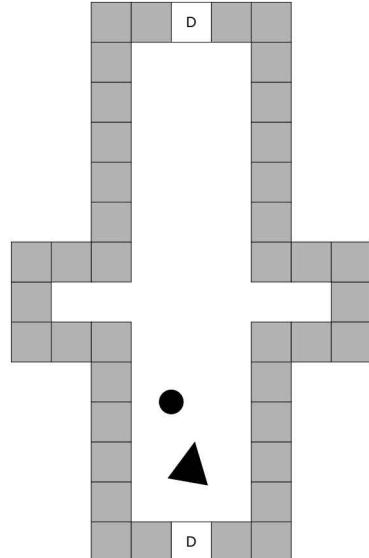
#### 4.7.9.2 Sound Non-propagation

Perfect sound propagation is a simplistic model. When a gun is fired, all enemies which can hear it will shout "Achtung!" (Attention!) and converge toward the origin of the sound. This would get old pretty fast, though. To make gameplay spicier, the designers introduced a little hack that goes a long way in making the AI appear smarter than it is.



There is a perfect example early on in the game in map E1M1.

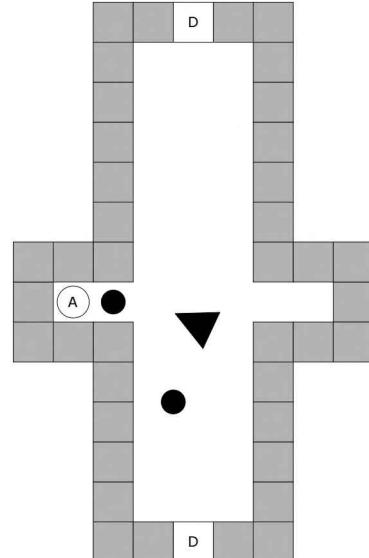
At this point, the player has learned that enemies will react to gunfire and move toward its origin. Upon entering this room and dispatching the guard, she naturally assumes this is a safe place – a bounded area where no other enemies can be encountered (otherwise they would have shown up or at least shouted by now).





Feeling safe, she will either run straight to the door or go left (or worse: right) to see what is in these corners. Surprise! An enemy was "hiding". This behavior is possible thanks to special tiles marked "AMBUSH" which make the engine not propagate sound to actors standing on these tiles.

This is probably one of the cheapest features in the engine, yet it results in what most would agree is the soul of the game - keeping the player on her toes.



**Trivia :** The ambush behavior is explained in the Hint Book as follows:

“ Each enemy is given specific orders which dictate his actions once he knows of your presence. Some are ordered to immediately attack, while others are trained to act only upon visual contact.

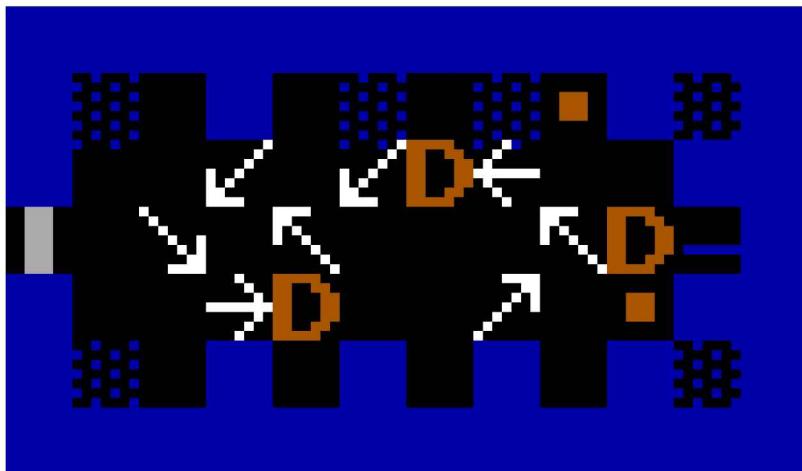
Kevin Cloud. **The Official Hint Manual for Wolfenstein 3D**

**Trivia :** Be it bug or dedication, a guard on an AMBUSH tile will react ONLY to seeing the player. Seeing another actor/dog die right in front of him will not activate him.

#### 4.7.9.3 Patrolling

In "path" mode, an agent simulates patrolling. There is no path finding; this is all done via waypoints which change the direction of the agent. After stepping on a "change direction" tile, the agent keeps on walking straight until it hits a wall. All waypoints are manually placed by the map designer with TED5.

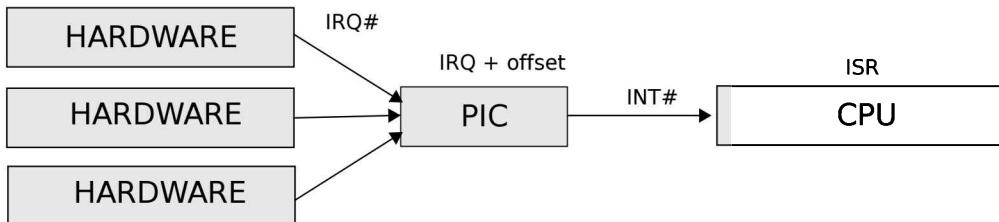
The waypoint system was used in other creative ways, like in E1M1 to simulate the agitation of German shepherds (marked a "D"). A detailed view of the right wings of the map from page 162 shows no less than eight waypoints making the dogs run in circles and zigzag.



## 4.8 Audio and Heartbeat

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

The idea is to configure the hardware to trigger a hardware interrupt at a regular interval. This interrupt is caught by a system called PIC which transforms it into a software interrupt. The software interrupt ID is used as an offset in a vector to look up a function belonging to the engine. At this point, the CPU is stopped (a.k.a: interrupted) from doing whatever it was doing (likely running the 3D renderer), and it starts running the interrupt handler which is called an ISR<sup>20</sup>. We now have two systems running in parallel.



**Figure 4.59:** Hardware interrupts are translated to software interrupt via the PIC.

Since interrupts keep triggering constantly from various sources, an ISR must choose what should happen if an IRQ is raised while it is still running. There are two options. The ISR can decide it needs a "long" time to run and disable other IRQs via the IMR<sup>21</sup>. This path introduces the problem of discarding important information such as keyboard or mouse inputs.

Alternately, the ISR can decide not to mask other IRQs and do what it is supposed to do as fast as possible so as to not delay the firing of other important interrupts that may lose data if they aren't serviced quickly enough.

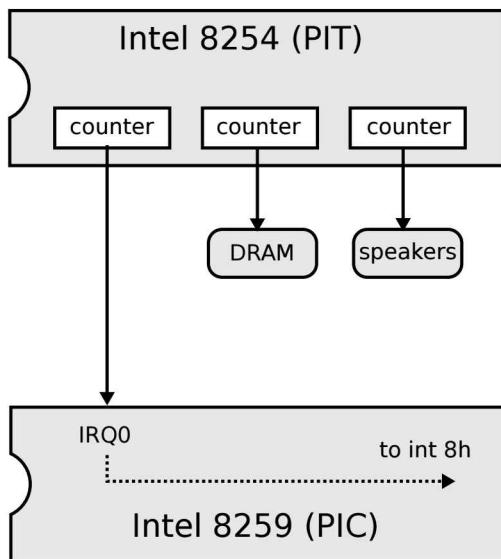
Wolfenstein 3D uses the latter approach and keeps tasks in its ISR very small and short. To this effect everything in the audio and heartbeat system is written in assembly and avoids "heavy" processing.

<sup>20</sup>Interrupt Service Routine

<sup>21</sup>Interrupt Mask Register

### 4.8.1 IRQs and ISRs

The IRQ and ISR system relies on two chips: the Intel 8254 which is a PIT<sup>22</sup> and the Intel 8259 which is a PIC<sup>23</sup>. The PIT features a crystal oscillating in square waves. On each period, it decrements its three counters. Counter #2 is connected to the RAM in order to automatically perform something called "memory refresh"<sup>24</sup>. Counter #1 is connected to the buzzer and generates sounds. Counter #0 is connected to the PIC. When counter #0 hits zero it generates an IRQ<sup>25</sup> and sends it to the PIC.



**Figure 4.60:** Interactions between PIT and PIC.

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

<sup>22</sup>Programmable Interval Timer

<sup>23</sup>Programmable Interrupt Counter

<sup>24</sup>Without frequent refresh, DRAM will lose its content. This is one of the reasons it is slower and SRAM is preferred in the caching system.

<sup>25</sup>Interrupt Request Line: Hardware lines over which devices can send interrupt signals to the CPU.

I.V.T Entry #	Type
00h	CPU divide by zero
01h	Debug single step
02h	Non Maskable Interrupt
03h	Debug breakpoints
04h	Arithmetic overflow
05h	BIOS provided Print Screen routine
06h	Reserved
07h	Reserved
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 2
0Eh	IRQ6, Floppy Disk Controller
0Fh	IRQ7, LPT2, Parallel port 1
10h	Video services (VGA)
11h	Equipment check
12h	Memory size determination

**Figure 4.61:** The Interrupt Vector Table (entries 0 to 18).

Notice #8 which is associated with the System timer and usually updates the operating system clock. Because IRQ 0 was hijacked, the operating system clock is not updated updated while Wolfenstein 3D runs. Upon exiting the game, DOS will run late by the amount of time played.

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

The engine can decide at what frequency to be interrupted, depending on the type of sound/music it needs to play and what devices will be used. As a result, three different ISRs can be found at IVT #8:

1. `SDL_t0SlowAsmService` when running at 140Hz, to play sound effects on the beeper via PWM and PCM audio effects on SoundBlaster.
2. `SDL_t0FastAsmService` when running at 700Hz, to play FM music, FM sound effects on AdLib, and PCM audio effects on Disney Sound Source.
3. `SDL_t0ExtremeAsmService` when running at 7000Hz, to play PCM sound in an alternate way. This mode was never enabled in shipping products, see page 228 for

a detailed explanation.

### 4.8.2 PIT and PIC

The PIT chip runs at 1.193182 MHz. This initially seems like an odd choice from the hardware designers, but has a logical origin. In 1980 when the first IBM PC 5150 was designed, the common oscillator used in television circuitry was running at 14.31818 MHz. As it was mass produced, the TV oscillator was very cheap so utilizing it in the PC drove down cost. Engineers built the PC timer around it, dividing the frequency by 3 for the CPU (which is why the Intel ran at 4.7Mhz), and dividing by 4 to 3.57Mhz for the CGA video card. By logically ANDing these signals together, a frequency equivalent to the base frequency divided by 12 was created. This frequency is 1.1931816666 MHz. By 1991, oscillators were much cheaper and could have used any frequency but backward compatibility prevented this.

### 4.8.3 Heartbeats

Each time the interrupt system triggers, it runs another small (yet paramount) system before taking care of audio requests. The sole goal of this heartbeat system is to maintain a 64 bit variable: TimeCount.

```
longword TimeCount;
```

It is updated at a rate of 70 units per seconds (to match the VGA update rate of 70Hz). These units are called "ticks". Depending on how fast the audio system runs (from 150Hz to 7000Hz), it adjusts how much it should increase TimeCount.

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

### 4.8.4 Audio System

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

```

void      SD_Startup(void),
SD_Shutdown(void),

boolean   SD_PlaySound(soundnames sound);
          SD_PositionSound(int leftvol,int rightvol);
void      SD_SetPosition(int leftvol,int rightvol),
          SD_StopSound(void),
          SD_WaitSoundDone(void),
word      SD_SoundPlaying(void);
          SD_SetSoundMode(SDMode mode),

void      SD_StartMusic(MusicGroup far *music),
          SD_MusicOn(void),
          SD_MusicOff(void),
          SD_FadeOutMusic(void),
boolean   SD_MusicPlaying(void),
          SD_SetMusicMode(SMMode mode);

```

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

#### 4.8.5 Music

Playing music is not too messy since only PCs equipped with a Yamaha YM3812 FM synthesizer could play tracks (a.k.a: with an AdLib or a SoundBlaster inside). As SoundBlaster made its programming interface compatible with AdLib there is only one code path to both cards. There is not a lot of magic here since this uses a piece of well-designed hardware dedicated to this specific task. There are a few cool tricks, though.

The music system streams data to the sound cards. Music in the 90s was not in digitized formats like today's CD or MP3 formats (that would have taken too much storage space and bandwidth). Instead music was stored as series of notes played on channels simulating instruments. The format used is close to the notorious MIDI but with a few variations and is called IMF<sup>26</sup>. It is proprietary to id Software and designed with OPL2 in mind (the raw format is exactly what is sent to the AdLib/Soundblaster synthesizer with no transformations). IMF has a hardcoded playback rate and music notes are played at 700 Hz.

Hardware limitations dictated certain aspects of music design. The FM synthesizer (OPL2) has 9 channels (a.k.a instruments) yet the composer, Bobby Prince, was asked to use only

---

<sup>26</sup>Id Music Format

channels 1 to 8. This little trick allows for multiplexing music and sound effects on AdLib cards since it leaves channel 0 available at all times (the SoundBlaster plays sound differently).

#### 4.8.5.1 OPL2/YM3812 Programming

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

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

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

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

When the AdLib was released in 1986, developers were instructed to send data "as fast as possible". At 4.77Mhz, a PC was unable to out-pace the AdLib. Yet as CPUs got faster, issues started to arise and the card was unable to keep up.

The original AdLib manual (before they shipped) did not call for ANY delays. The original IBM PC (4.77 Mhz) couldn't get ahead of the card. By the time it shipped they were telling use to do one IN instruction, and every time a new faster processor came out they added some delay to their recommendation. The old 8088 machines would not have worked with the 35 IN instructions now required, it would have slowed the machine down so much nothing else could get done.

**Jason Linhart**

Later the Programming Guide was amended with reliable specs.

---

<sup>26</sup><http://www.oldskool.org/guides/oldonnew/sound> .

“

Wait three point three (3.3) microseconds for the address, and twenty-three (23) microseconds for the data.

**AdLib manual**

”

The engine does not know about any of the details of the OPL2. There is zero abstraction layer or transformation here. An IMF song is made of a series of messages containing exactly the values to write to the register and data ports of the OPL2. Each message is four bytes:

```
struct music_packet {  
    char reg;      // Sent to register port.  
    char data;     // Sent to data port.  
    int delay;     // How much to wait.  
}
```

The `reg` byte is sent to port 0x388, the `data` byte is sent to 0x389, and the `delay` 2 bytes are used to tell how much time to wait before sending the next register/data to the card. The stream is hard-coded at 700Hz and the delay is expressed in this unit: a value of 700 means to wait 1000ms before sending another command. Whenever there is music playing the engine runs at no less than 700Hz. A value of zero means the next message should be sent immediately.

Overall, music is simple to execute because almost everything has been pre-processed via IMF. Every time the audio system wakes up, it checks if music packets should be sent, sends them, and moves on to the sound effects.

## 4.9 Sound Effects

Sound effects are where things become complicated. None of the cards use the same format and audio configurations are numerous. The sound settings screen illustrates how complex this is.



Sounds are stored in three formats. Once for PC Speaker, once for AdLib, and once for SoundBlaster/Disney Sound Source in the `AudioT` archive created by Muse. Sounds are segregated by format but always stored in the same order. This way a sound can be accessed in three formats by using `STARTPCSOUNDS + sound_ID` or `STARTADLIBSOUNDS + sound_ID`.

```
///////////////////////////////
//  
// MUSE Header for .SDM
// Created Thu Aug 27 07:12:39 1992
//  
///////////////////////////////  
  
#define STARTPCSOUNDS      0  
#define STARTADLIBSOUNDS   81  
#define STARTDIGISOUNDS    162  
#define STARTMUSIC         243  
  
// Sound names & indexes  
typedef enum {  
    HITWALLSND ,           // 0  
    MISSILEHITSND ,        // 1  
    /* */  
    DEATHSCREAM1SND ,      // 29  
    GETMACHINESND ,        // 30  
    GETAMMOSND ,           // 31  
    SHOOTSND ,             // 32  
    HEALTH1SND ,            // 33  
    HEALTH2SND ,            // 34  
    BONUS1SND ,             // 35  
    BONUS2SND ,             // 36  
    BONUS3SND ,             // 37  
    /* */  
    ANGELTIRED SND ,       // 80  
    LASTSOUND  
} soundnames;  
  
// Music names & indexes  
typedef enum {  
    XFUNKIE_MUS ,          // 0  
    DUNGEON_MUS ,           // 1  
    XDEATH_MUS ,            // 2  
    /* */  
    XTOWER2_MUS ,           // 23  
    LASTMUSIC  
} musicnames;
```

Strangely, only PC speaker and AdLib sounds are stored in the AUDIOT.\* files; the digitized sounds are in the VMSWAP.\* archive. As a result, offset STARTDIGISOUNDS is never

used. The authors of this code were asked, but it seems nobody can remember why.

### 4.9.1 Sound Effects: AdLib

AdLib only has an FM synthesizer, with sounds played on its Channel 0. Sounds are played the same way as music via IMF, as previously described.

### 4.9.2 Disney Sound Source System: PCM

The Disney Sound Source is simple to program<sup>27</sup> because it can do only one thing. It plays 8 bit PCM audio at 7000Hz on a single channel. That's it – nothing more, nothing less. Data is fed through the parallel port to the device, which is stored into a 16-byte FIFO queue, which then feeds an 8-bit DAC connected to its integrated loudspeaker.

Even though the sampling rate is 7000Hz, the sound system needs not to run at such a high frequency. The 16 byte buffer allows to run at 700Hz and send a batch of 9-10 bytes every iterations. Every time the audio system wakes, it reads the DAC status to check how many bytes have been consumed in the FIFO. The engine pushes as many bytes as possible until the FIFO is full and returns. When the FIFO empties, the Disney Sound Source stops making noise.

### 4.9.3 SoundBlaster System: PCM

Since the SoundBlaster also supports 7000Hz PCM, it uses the same sound effects files as the Disney Sound Source. However, it also features a DSP which is DMA capable. As a result, the CPU does not have to waste cycles transferring data. The audio system has little to do and call allow itself to run in "slow" mode at 140Hz. It wakes up to point the DMA to the right memory address each time it crosses the end of a 16K segment via a DMA routine callback.

### 4.9.4 SoundBlaster Pro System: Stereo PCM

On a "high-end" SoundBlaster Pro, the Mixer described on page 59 is used to simulate 3D sounds. The source of the sound is first rotated with the same formula seen on page 176. The player's position is used to generate two attenuation values (between 0 and 15) which are then packed in a byte and sent to the mixer. The difference in volumes tricks the player into perceiving the sound origin anywhere within the 180 degrees facing her.

---

<sup>27</sup>The Programmer's Guide to the Disney Sound Source is a whopping 2 pages!

```
#define      sbOut(n,b)      outportb((n) + sbLocation,b)

#define      sbpMixerAddr      0x204
#define      sbpMixerData      0x205

//      SBPro Mixer addresses
#define      sbpmVoiceVol      0x04

static void SDL_PositionSBP(int leftpos,int rightpos) {
    byte      v;

    if (!SBProPresent)
        return;

    leftpos = 15 - leftpos;
    rightpos = 15 - rightpos;
    v = ((leftpos & 0x0f) << 4) | (rightpos & 0x0f);

    asm      pushf
    asm      cli

    sbOut(sbpMixerAddr,sbpmVoiceVol);
    sbOut(sbpMixerData,v);

    asm      popf
}
```

**Trivia :** Plugging in a SoundBlaster card was not enough to produce sound. This was before "plug & play" was introduced by Windows 95. The user had to write a special line in the startup command of the PC (autoexec.bat).

```
SET BLASTER=A220 I5 D1
```

This line defines a variable BLASTER which the engine retrieves and parses at runtime with getenv. A tells what port the card is using. I gives away the interrupt vector it is associated with. Finally D gives the DMA channel to use for data transfer. For all this to work the sound card had to be configured accordingly via its jumper connectors. See SDL\_SBPlaySeg and the ISR SDL\_SBSERVICE in the source code.

#### 4.9.5 PC Speaker: Square Waves

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

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



**Figure 4.62:** The original sound.



**Figure 4.63:** The same sound approximated with square wave and frequency changes.

To do this, the audio system once again relies on the PIT chipset. Counter 0 is used to trigger the audio system. Counter 1 is used to refresh the RAM periodically. Counter 2, however, is directly connected to the PC Speaker. The trick is to set this Counter 2 to square wave mode (Mode 3) so it will repeat after it triggers and program the desired square wave frequency.

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

**Figure 4.64:** Available modes of a PIT counter.

When instructed to play a PC Speaker sound effect, the audio system sets itself to run at 140Hz via PIC 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. The frequencies to use are encoded as a stream of bytes, the value of which is decoded as follows:

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

In the assembly, three I/O ports are accessed:

```
#define pcTimer    0x42
#define pcTAccess  0x43
#define pcSpeaker   0x61
```

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

**Trivia :** LucasArts obtained surprisingly good results for their game Monkey Island. See the video "LGR - Evolution of PC Audio - As Told by Secret of Monkey Island"<sup>28</sup>.

```
MACRO DOFX
les di,[pcSound]           ; PC sound effects
mov ax,es
or ax,di
jz @@nopc                 ; no PC sound effect going

mov bl,[es:di]              ; Get the byte
inc [WORD PTR pcSound]     ; Increment pointer
cmp [pcLastSample],bl       ; Is this sample same as last?
jz @@pcsame                ; Yep - don't do anything
mov [pcLastSample],bl       ; No, save it for next time

or bl,bl
jz @@pcoff                 ; If 0, turn sounds off
xor bh,bh
shl bx,1
mov bx,[pcSoundLookup+bx]  ; Use byte as lookup

mov al,0b6h                 ; Select channel 2 (speaker) timer
out pcTAccess,al
mov al,bl
out pcTimer,al              ; Low byte
mov al,bh
out pcTimer,al              ; High byte

in al , pcSpeaker          ; Turn the speaker & gate on
or al ,3
out pcSpeaker ,al
```

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

---

<sup>28</sup><https://www.youtube.com/watch?v=a324ykKV-7Y>

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

Notice how 6b (10110110) is sent to the PIC Command register:

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

#### 4.9.6 PC Speaker: PCM

The source code features an audio code path which plays PCM digitized sound via the PC Speaker. The function `SDL_PlayDigiSegment` has a switch to route playback. Notice the branch leading to `SDL_PCPlaySample` allowing PCM to be played on the buzzer.

```
void  SDL_PlayDigiSegment(memptr addr, word len) {
    switch (DigiMode)
    {
        case sds_PC: // Never used :
            SDL_PCPlaySample(addr, len); break;
        case sds_SoundSource:
            SDL_SSPlaySample(addr, len); break;
        case sds_SoundBlaster:
            SDL_SBPlaySample(addr, len); break;
    }
}
```

The problem of this method is to find a way to fit 8 bit samples into a 1 bit DAC. Let's take the example of the digitized sound "Mein Liben" which has 6,896 samples of 8 bits.



**Figure 4.65:** Mein Liben PCM 8 bit. You can clearly see the three syllables.

The audio system goes for a surprisingly straightforward solution. It set itself to run at 7000Hz and manually move the cone of the speaker, using quantization to map 256 values to either 1 or 0.

```

PROC      SDL_t0ExtremeAsmService
PUBLIC    SDL_t0ExtremeAsmService

    ...

les di,[pcSound]          ; Prepare to get byte.
mov ax,es
or  ax,di
jz  @@donereg           ; nil pointer

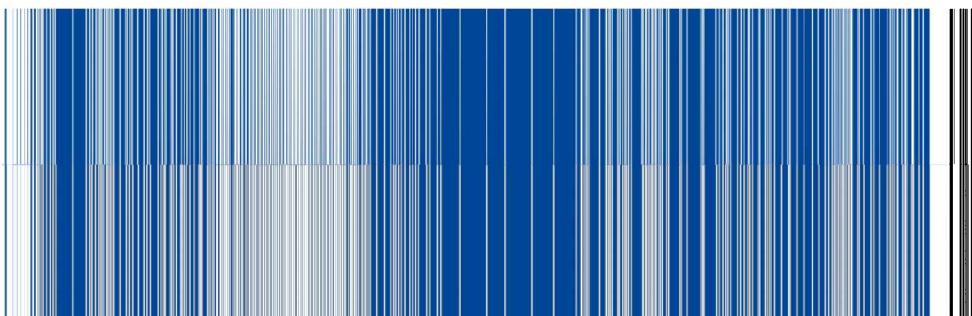
mov bl,[es:di]            ; Get the PCM byte
inc [WORD PTR pcSound]   ; Increment pointer

and bl,11100000b          ; Nuke some of the precision
                           ; (DEBUG: do this with table)
xor bh,bh
mov ah,[pcdtab+bx]        ; Translate the byte

in  al,pcSpeaker
and al,11111100b
or  al,ah
out pcSpeaker,al

```

`SDL_t0ExtremeAsmService` takes an 8 bit value and convert it to 00 or 10 to drive bit 1 in I/O port 61h. When bit 1 is set to 1, the beeper cone will go in position high and when set to 0, it will go to position low.



**Figure 4.66:** Mein Liben resampled to 1 bit.

The visual representation of a 1 bit per sample "Mein Liben" PCM looks like mashed potatoes but it sounds remarkably good. You can hear it thanks to a passionate YouTuber named Dafe Allen who recompiled the engine with PCM playback enabled<sup>29</sup>.

As good as it sounds, this codepath shipped but was never enabled. According to John Carmack overhead was too much a problem. Jim Leonard provided a more in depth explanation.

PCM playback over the PC speaker was likely cut because it would require `SDL_t0ExtremeAsmService`. The Disney Sound Source had a 16 byte buffer but the PC Speaker had no such thing, the only way to feed it fast enough is to run the interrupt audio system at 7000 Hz.

High-end 386s could handle this, but this was nearly unsustainable on the 286. Sending data to the parallel port does not take a lot of time – but interrupt overhead does, and 7000 Hz on a 12 MHz 286 would definitely have been noticeable.

Running at a frequency this high would have caused the game to freeze on slower machines while the sample played through the speaker.

Jim Leonard

#### 4.9.7 PC Speaker: PWM

Even though this method was not used in Wolfenstein 3D, it is worth mentioning a third way hackers managed to play pleasant sounds via the PC speaker. The method produced audio quality superior to both the square waves and the 1 bit conversion we just saw. It was called Pulse Width Modulation.

The idea is to manipulate the speaker cone manually and instruct it to move faster than it can, interrupting it "somewhere" between its position up or down.

The technique was patented (US US5054086 A) by Access Software and called RealSound. Many studio licensed it during the 80s but the advent of dedicated sound cards with FM and PCM capabilities made RealSound obsolete in the early 90s.

---

<sup>29</sup>"Wolfenstein 3D Hack - Digitized PC Speaker Sound Effects": "<https://www.youtube.com/watch?v=1BtIsjJRNFU>".

“

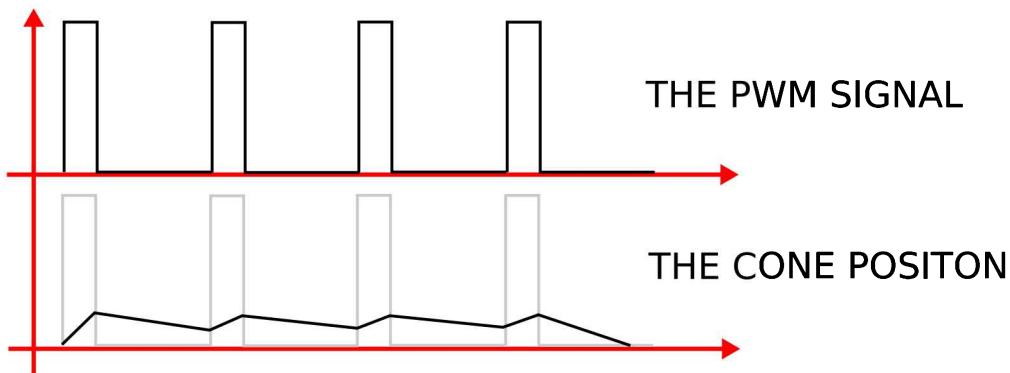
The PC speaker is normally meant to reproduce a square wave via only 2 levels of output (the speaker is driven by only two voltage levels, typically 0 V and 5 V). However, by carefully timing a short pulse (i.e. going from one output level to the other and then back to the first), and by relying on the speaker's physical filtering properties (limited frequency response, self-inductance, etc.), the end result corresponds to intermediate sound levels. This effectively allows the speaker to function as a crude 6 bit DAC, thereby enabling approximate playback of PCM audio.

This technique is called pulse-width modulation (PWM).

Jim Leonard - oldskool.org

”

In the next drawing we see how commanding the cone faster than it can move results in intermediate positions.



## 4.10 User Inputs

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

### 4.10.1 Keyboard

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

installs its own ISR there.

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

static void INL_StartKbd(void) {
    INL_KeyHook = NULL;        // no key hook routine

    IN_ClearKeysDown();

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

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

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

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

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

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

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

## 4.10.2 Mouse

A driver has to be loaded at startup for the mouse to be accessible. Beginning with DOS 2.1, operating systems had a default driver. It had to be added to `config.sys` so it would reside in RAM.

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

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

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

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

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

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

**Figure 4.67:** Mouse request/response.

### 4.10.3 Joystick

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

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

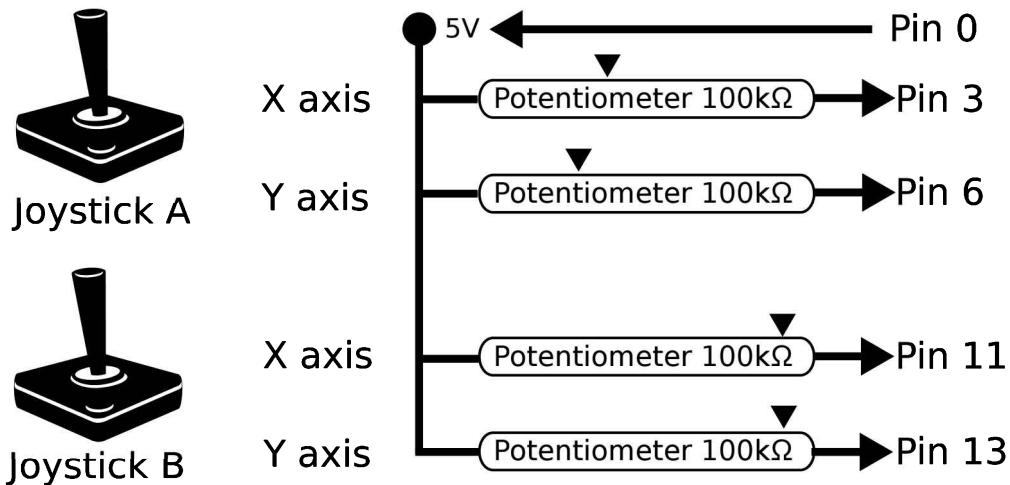
    result = inportb(0x201); // Get all the joystick buttons
    result >= joy? 6 : 4;   // Shift into bits 0-1
    result &= 3;           // Mask off the useless bits
    result ^= 3;
    return(result);
}
```

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

**Figure 4.68:** Joystick sampling bit and their meaning.

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

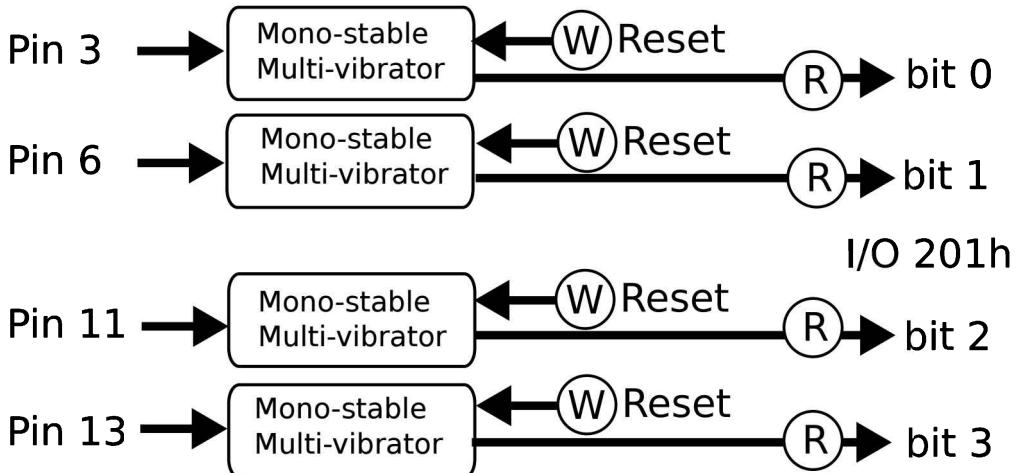
On the joystick side, each axis is connected to a  $100\text{k}\Omega$  potentiometer. An applied 5V voltage generates a variable current based on the stick position (from Ohm's law where  $I = \frac{V}{R}$ ).



**Figure 4.69:** Two joysticks and the four potentiometers connected to the game port pins.

On the joystick side each pin carrying the current is connected to monostable multivibrators (which is a complicated name for a capacitor able to output 1 when it is charged and

0 when it is charging). The idea is to infer the position of the stick by measuring how long the vibrator takes to charge (a strong current will charge the capacitor faster than a weak current).



**Figure 4.70:** Each potentiometer is connected to a capacitor able to output either 0 or 1 depending on its charging state.

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

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

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

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

combination.<sup>30</sup>



**Figure 4.71:** Strike Commander startup screen makes you calibrate your joystick.

There is no calibration process in Wolfenstein 3D because when the engine starts up it samples the loop count and assumes the joystick is in neutral position. When the game runs and joystick position is needed, the engine samples loop count and compares the count to what was measured with neutral. It is not enough to calculate the exact stick position on each axis but it is enough to determine if it was up/down and left/right using  $>$ ,  $==$  (with epsilon) and  $<$  comparison operations.

## 4.11 Tricks

This section describes random tricks used to speed up rendering. They range from simple precomputed cos/sin tables to what I consider one of the most beautiful hacks in the engine: the Linear Feedback Shift Register.

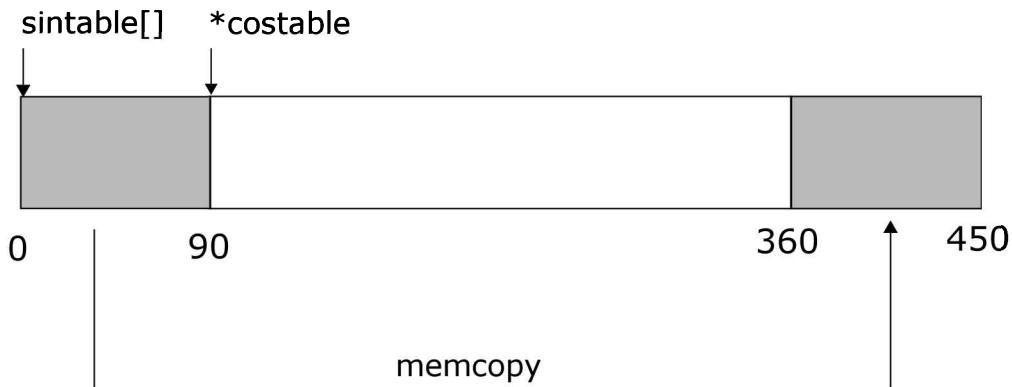
### 4.11.1 Cos/Sin Table Lookup

`cos` and `sin` are expensive methods involving floating point calculations. They are extensively used at runtime. To speed things up, the engine generates and caches them in a lookup array (one value per angle) at startup. To save RAM, it exploits a math property ( $\cos(X) = \sin(X + 90)$ ) to avoid 360 `cos` method calls and 240 bytes of RAM by reusing the `sin` table as follows:

```
#define ANGLES 360
fixed far sintable[ANGLES+ANGLES/4];
far *costable = sintable+(ANGLES/4);
```

---

<sup>30</sup>The Mark-1 FCS by Thrustmaster and Flightstick Pro by CH were the best flight controller of the 90s. They used they used all bits for one controller, offering a devices with four buttons with the extra two axis serving as a four-way view hat.



**Figure 4.72:** In grey the 90 `sin` values duplicated at the end of the array to complete the `cos` lookup table.

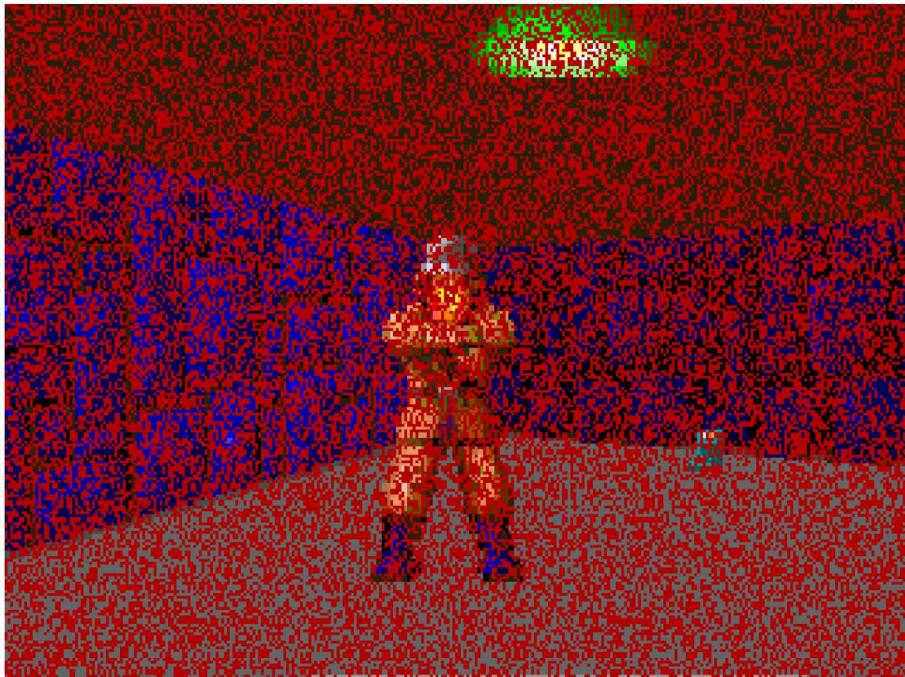
### 4.11.2 FizzleFade

While most screen transitions are done with a fade to black (by shifting the palette), there are two instances when the screen transitions via fizzling:

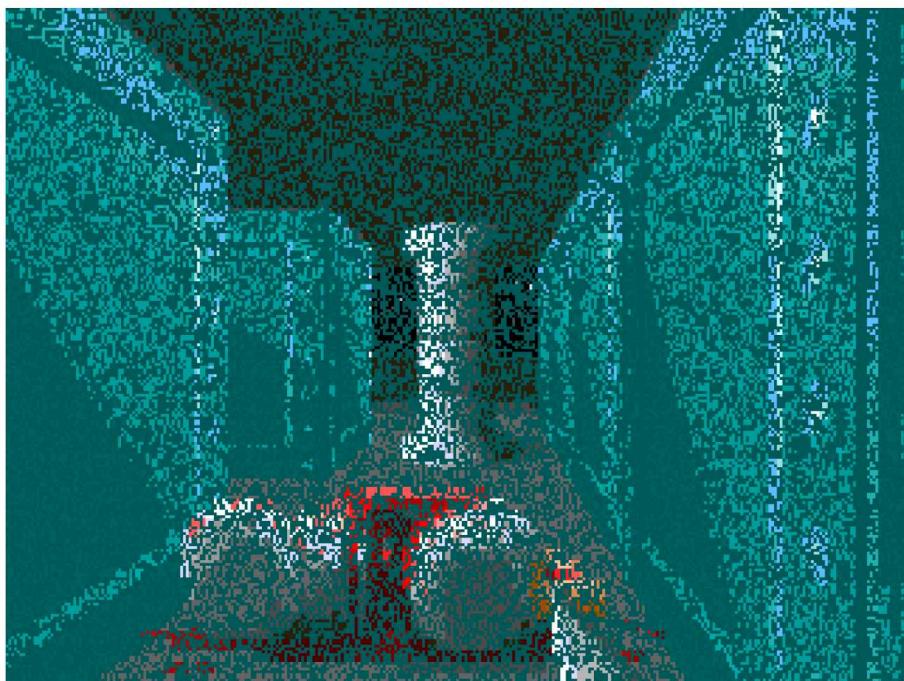
- When dying
- When killing a boss

Below are a series of screenshots to illustrate fizzling. During the transition, each pixel on the screen is turned to red (when dying) or blue (when dispatching a boss). Each pixel is written only once and seemingly at random.









To implement this effect, a naive approach would have been to use the pseudo random generator US\_RndT and keep track of which pixels had been fizzled. However, this would make the fade non-deterministic with regard to duration and would also waste CPU cycles since the same pixel coordinates (X,Y) could come up several times. There is a faster and more elegant way to implement a pseudo-random value generator. The code responsible for this effect can be found in `id_vh.cpp`, in the function `FizzleFade`. At first, it is not obvious how it works.

```
boolean FizzleFade {
    long rndval = 1;
    int x,y;
    do
    {
        // seperate random value into x/y pair
        asm mov ax,[WORD PTR rndval]
        asm mov dx,[WORD PTR rndval+2]
        asm mov bx,ax
        asm dec bl
        asm mov [BYTE PTR y],bl      // low 8 bits - 1 = y
        asm mov bx,ax
        asm mov cx,dx
        asm mov [BYTE PTR x],ah      // next 9 bits = x
        asm mov [BYTE PTR x+1],dl

        // advance to next random element
        asm shr dx,1
        asm rcr ax,1
        asm jnc noxor
        asm xor dx,0x0001
        asm xor ax,0x2000
    noxor:
        asm mov [WORD PTR rndval],ax
        asm mov [WORD PTR rndval+2],dx

        if (x>width || y>height) continue;

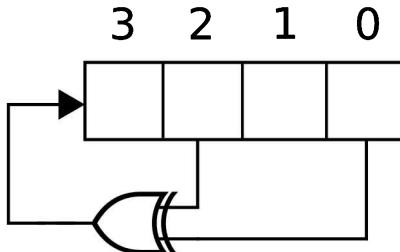
        copy_screen_pixel(x,y);

        if (rndval == 1) return false; // end sequence
    } while (1)
}
```

This code can be read as:

- Initialize `rndval` to 1.
- Break it down in 8 + 9 bits: use 8 bits to generate a Y coordinate and 9 bits for a X coordinate. Turn this pixel to red.
- Subject `rndval` to a soup of XORing.
- When `rndval` value is somehow back to 1: Stop.

This feels like dark magic. How is `rndval` supposed to return to value 1? Via a technique called Linear Feedback Shift Register. The idea is to use one register to store a state, generate the next state, and also generate a value. To get the next value, you do a right shift. Since the rightmost bit disappears, a new one to the left is needed. To generate this new bit, the register uses "taps" which are bit offsets used to XOR together values and generate the new bit value. A Fibonacci representation shows a simple LFSR with two taps.



The register depicted above is able to generate 6 values before it cycles back to its original state. The following listing shows all of them.

*	*		value
<hr/>			
0	0	0	1
1	0	0	8
0	1	0	4
1	0	1	A
0	1	0	5
0	0	1	2
0	0	0	1 -> CYCLE

Sequence of 6 numbers before cycling.

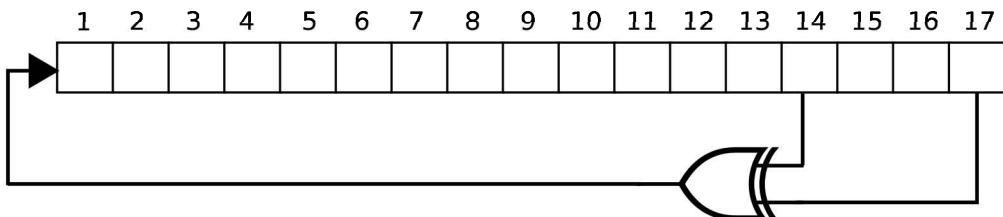
Various arrangements of taps will produce different series. In the case of this four bit register, the maximum number of values in a series is  $16-1 = 15$  (zero cannot be reached).

This can be achieved with taps on bits 0 and 1. This is called a "Maximum-Length" LFSR.

**		value
0001		1
1000		8
0100		4
0010		2
1001		9
1100		C
0110		6
1011		B
0101		5
1010		A
1101		D
1110		E
1111		F
0111		7
0011		3
0001		1 -> CYCLE

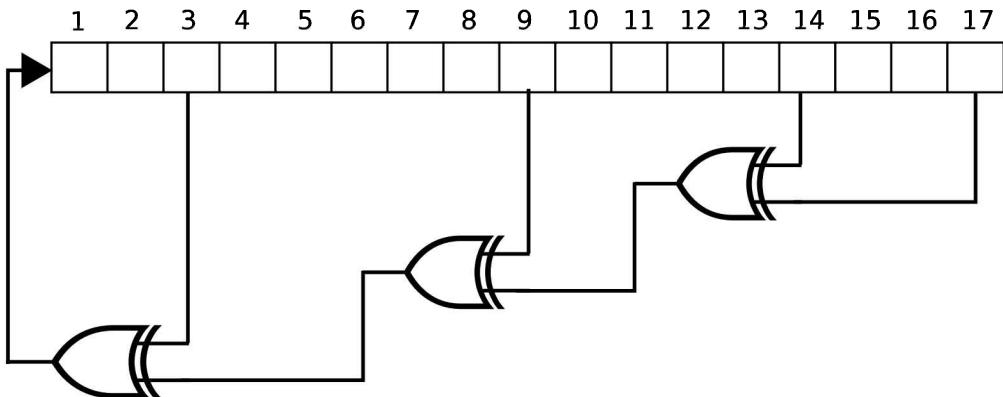
Sequence of 15 numbers before cycling.

Wolfenstein 3D uses a 17 bit Maximum-Length LFSR with two taps to generate a series of pseudo-random values. Of these 17 bits, on each iteration, 9 are used to generate a X coordinate and 8 for a Y coordinate. The corresponding pixel on screen is turned red/blue.



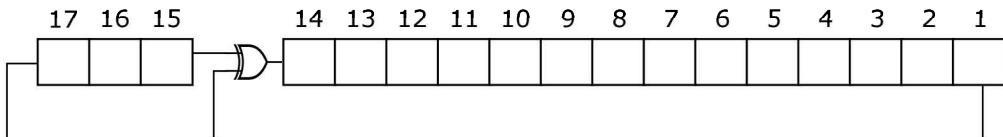
**Figure 4.73:** 17 bit Maximum-LFSR (Fibonacci representation).

The Fibonacci representation helps to understand the general idea. But it is not how a LFSR is usually implemented in software. The reason is that it scales linearly with the number of taps. With four taps, you need three sequential XOR operations:



**Figure 4.74:** Four taps on a 17 bit register; each XOR requires an instruction.

There is an alternate way to represent a LFSR called "Galois", which allows one XOR operation regardless of how many taps are involved. It is the way Wolfenstein 3D implements its LFSR and writes  $320 \times 200 = 64000$  pixels exactly once with a deterministic duration.



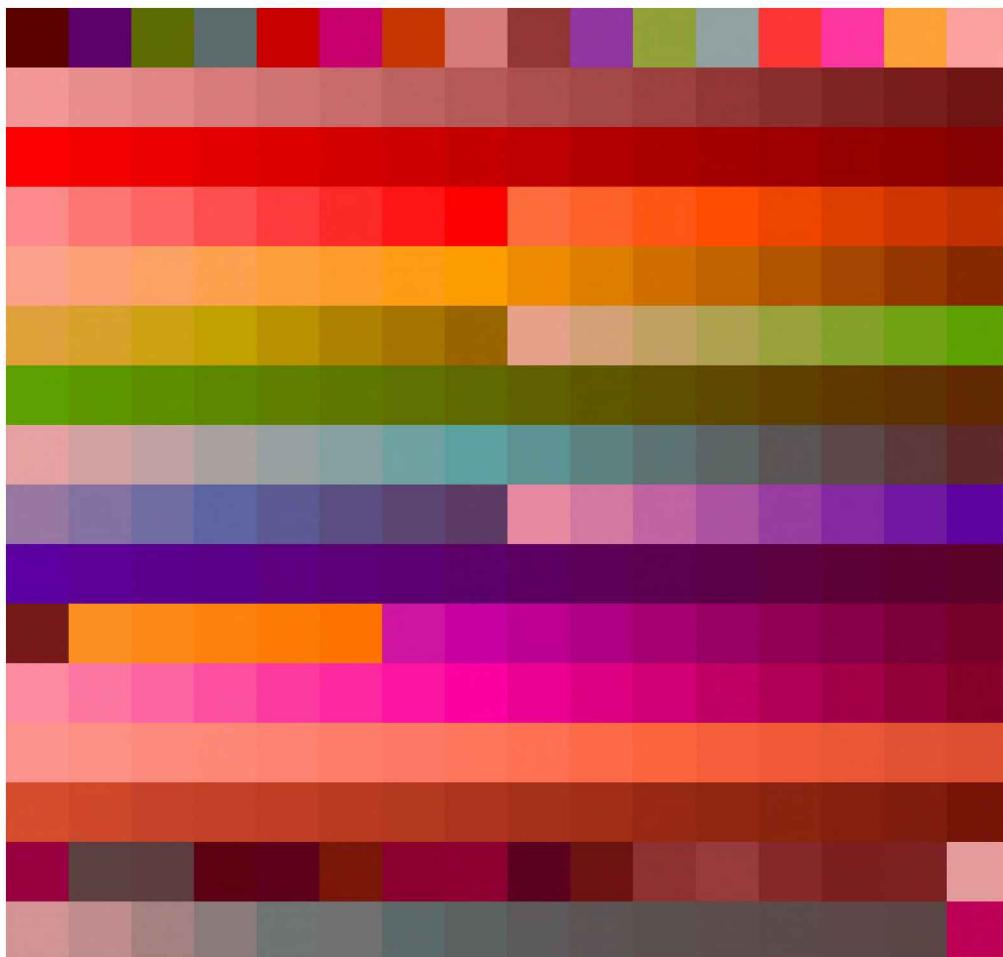
**Figure 4.75:** Galois representation allows for one XOR operation regardless of the number of taps. This configuration is equivalent to figure 4.73.

**Note :** Because the effect works by plotting pixels individually, it was hard to replicate when developers tried to port the game to hardware accelerated GPU. None of the ports managed to replicate the fizzfade except Wolf4SDL, which even found maximum-length LFSR taps configuration to reach resolution higher than  $320 \times 200$ .

**Note :** The tap configuration on 17 bits generates 131072 values before cycling. Since  $320 \times 200 = 64000$ , it could have been implemented with a 16 bit Maximum-length register with taps on 16, 15, 13 and 4 (in "Galois" notation).

### 4.11.3 Palette

Even though it limited the graphic capabilities of the game, the palette system can be turned into a strength. It is easy to fade the screen to white (when picking up an item), red (when taking damage), or black (when transitioning between 2D menus). It only takes  $256 \times 3 = 768$  bytes and 768 out instructions to modify the full screen.



**Figure 4.76:** The palette RGB colors is altered when taking damage.

A fast path is provided (which ironically only works if the CPU is as slow as the VGA processor) to update the palette via one `rep outsb` instruction. Otherwise, if not supported, a loop of 768 `outsb` is used.

```
void VL_SetPalette (byte far *palette) {
    asm mov dx,PEL_WRITE_ADR
    asm mov al,0
    asm out dx,al
    asm mov dx,PEL_DATA
    asm lds si,[palette]

    asm test [ss:fastpalette],1
    asm jz slowset
// set palette fast for cards that can take it
    asm mov cx,768
    asm rep outsb
    asm jmp done

// set palette slowly for some video cards
slowset:
    asm mov cx,256
setloop:
    asm lodsb
    asm out dx,al
    asm lodsb
    asm out dx,al
    asm lodsb
    asm out dx,al
    asm loop setloop

done:
    asm mov ax,ss
    asm mov ds,ax
}
```

## 4.12 Pseudo Random Generator

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

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

The pseudo-random series is initialized using the current time modulo 256 when the engine starts up.

“

The random table wasn't even a shuffle - note that there are no 1s and two 2s in the table. I built it by just storing out 256 randoms from a little C program. This was bad!

John Carmack - Programmer

”

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

PROC  US_InitRndT randomize:word
    uses  si,di
    public US_InitRndT

    mov ax,SEG rndtable
    mov es,ax

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

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

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

@@setit:
    mov [es:rndindex],dx
    ret

ENDP
```

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

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

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

ENDP
```

This pseudo random series of 256 values could have been generated with an 8 bit Maximum length LFSR (8,6,5,4). My assumption is that LFSR literature was hard to find at the time and finding the correct tap for a 16 bit maximum length register was not worth the effort.

## 4.13 Performance

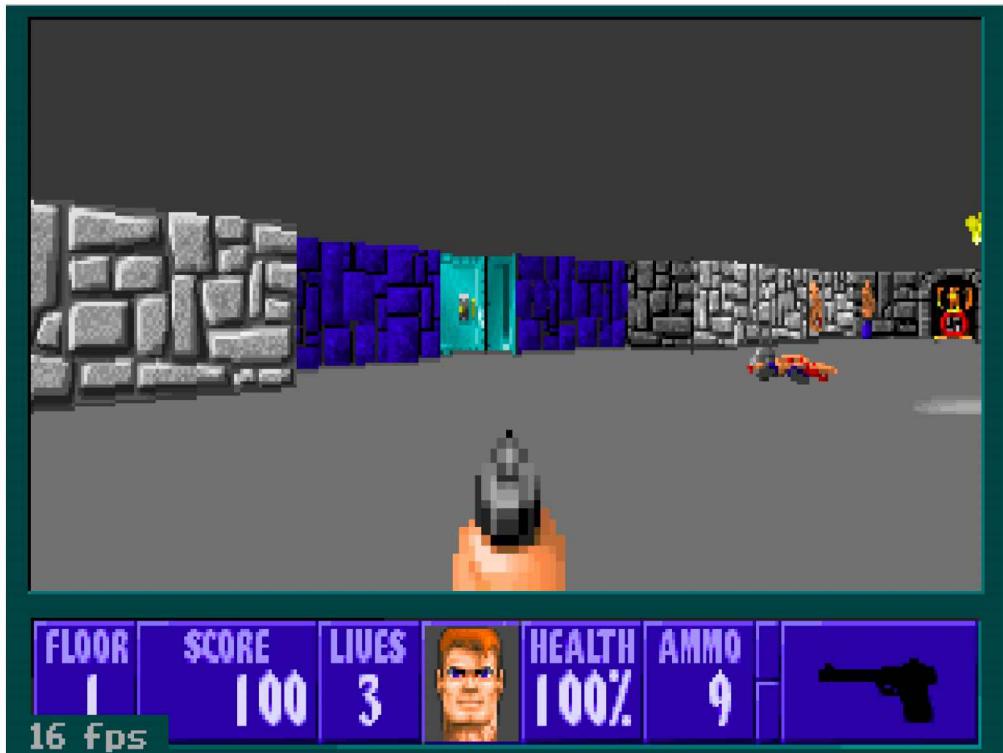
With all the tricks and optimizations described, how well did the game run on the machines of the early 90s? With most of these computers gone, it is hard to tell. But thanks to the personnal collections of Jim Leonard and Foone Turing, and a special version of Wolfenstein 3D displaying "frames per seconds", we were able to come up with the numbers in figure 4.77.

By far the most important component after the CPU was the VGA card. Cirrus Logic RAM was optimized for fast write (and slow read) which explains the huge performance boost compared to other cards. A 386DX-40 with a bad VGA card can be brought to the same level as a 386SX-16 with a good card. Likewise, upgrading the VGA card could double performances on a 386DX-40.

CPU	Mhz	Cache	Audio	VGA Card	Bus	Avg FPS
286	6mhz	0k	AdLib	Hercules VGA	8	5
286	12mhz	0k	AdLib	Paradise PVGA1A	16	7
286	25mhz	0k	ESS1868	CirrusLogic 5420	16	19
386SX	16mhz	512k	No	Oak Technology OTIVGA	16	10
386SX	16mhz	512k	No	ATI VGA Wonder	16	10
386SX	16mhz	512k	No	ATI VGA Wonder	8	10
386SX	16mhz	512k	No	Tseng Labs et300ax	8	11
386SX	16mhz	512k	No	Trident Tvga8900c	16	12
386SX	16mhz	512k	No	Headland tech GC208-PC	16	13
386SX	16mhz	512k	No	Cirrus Logic AVGA3M-C03	16	14
386SX	40mhz	0k	No	ATI VGA Wonder	16	16
386SX	40mhz	0k	No	ATI VGA Wonder	8	16
386SX	40mhz	0k	No	Tseng Labs et300ax	8	17
386SX	40mhz	0k	No	Oak Technology OTIVGA	16	17
386SX	40mhz	0k	No	Trident Tvga8900c	16	21
386SX	40mhz	0k	No	Headland tech GC208-PC	16	24
386SX	40mhz	0k	No	Cirrus Logic AVGA3M-C03	16	26
386SX	40mhz	0k	SB	ATI VGA Wonder	8	16
386SX	40mhz	0k	SB	ATI VGA Wonder	16	17
386SX	40mhz	0k	SB	Oak Technology OTIVGA	16	17
386SX	40mhz	0k	SB	Tseng Labs et300ax	8	18
386SX	40mhz	0k	SB	Trident Tvga8900c	16	21
386SX	40mhz	0k	SB	Headland tech GC208-PC	16	21
386SX	40mhz	0k	SB	Cirrus Logic AVGA3M-C03	16	25
386DX	40mhz	128kb	No	Tseng Labs et300ax	8	21
386DX	40mhz	128kb	No	ATI VGA Wonder	16	21
386DX	40mhz	128kb	No	ATI VGA Wonder	8	21
386DX	40mhz	128kb	No	Oak Technology OTIVGA	16	22
386DX	40mhz	128kb	No	Trident Tvga8900c	16	26
386DX	40mhz	128kb	No	Headland tech GC208-PC	16	32
386DX	40mhz	128kb	No	Cirrus Logic AVGA3M-C03	16	33
386DX	40mhz	128kb	SB	ATI VGA Wonder	16	20
386DX	40mhz	128kb	SB	ATI VGA Wonder	8	20
386DX	40mhz	128kb	SB	Tseng Labs et300ax	8	21
386DX	40mhz	128kb	SB	Oak Technology OTIVGA	16	21
386DX	40mhz	128kb	SB	Trident Tvga8900c	16	26
386DX	40mhz	128kb	SB	Headland tech GC208-PC	16	32
386DX	40mhz	128kb	SB	Cirrus Logic AVGA3M-C03	16	34

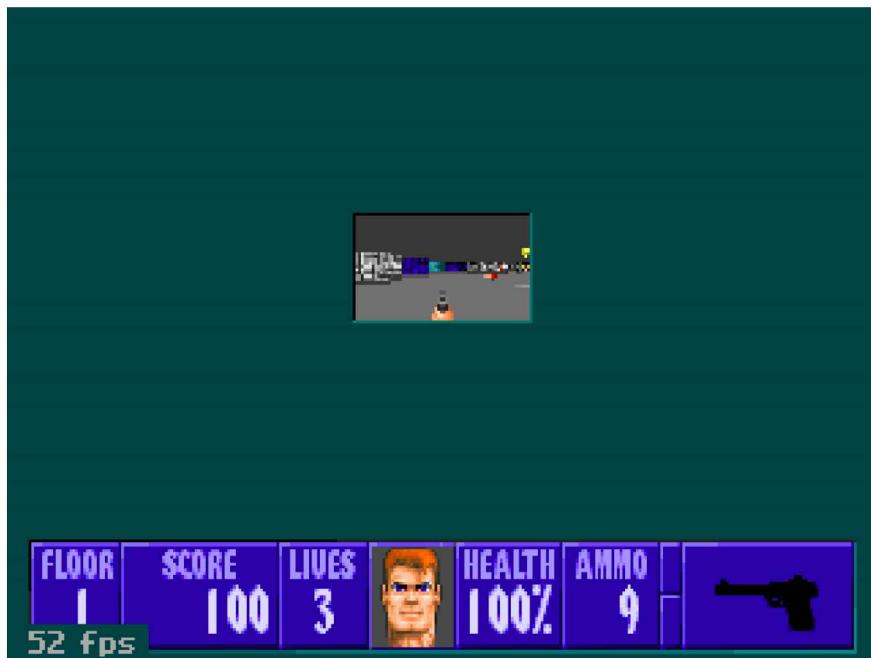
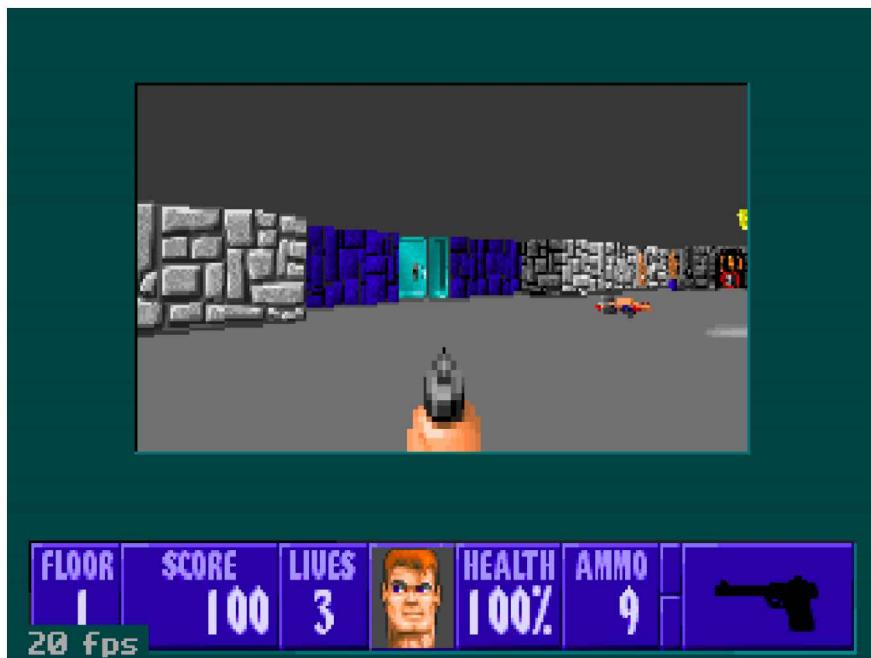
**Figure 4.77:** Average fps per machine.

To compensate for power differences, the engine can reduce the 3D canvas to lower the number of rays cast and the number of pixels to render. The maximum is 304 rays resulting in 46208 pixels to render ( $304 \times 152$ ) and the minimum is 64 rays resulting in 2432 pixels to render ( $64 \times 38$ ).



In the previous screenshot the game is running at maximum resolution: 304 rays, resolution of 304x152. This 386SX-16Mhz achieved 16 frames per second.

In the next two screenshots the framerate improves as 3D canvas size is reduced. 224 rays and a resolution of 224x112 raises the rate to 20 fps. 64 rays and a resolution of 64x38 brings it up to 52 fps.



**Trivia :** In 1994, 3D Realms published Rise Of the Triad (a.k.a ROTT), directed by Tom Hall. The game uses a highly-modified<sup>31</sup> Wolfenstein 3D engine and also features the same window size adjustment system. True to the humor found in early 90s video games, it features an Easter egg teasing the player to buy a 486 when the lowest setting is chosen.



<sup>31</sup>Textured floors and ceiling, different wall heights, stairs, trampolines, and even jumps.

# **Chapter 5**

## **Sequels**

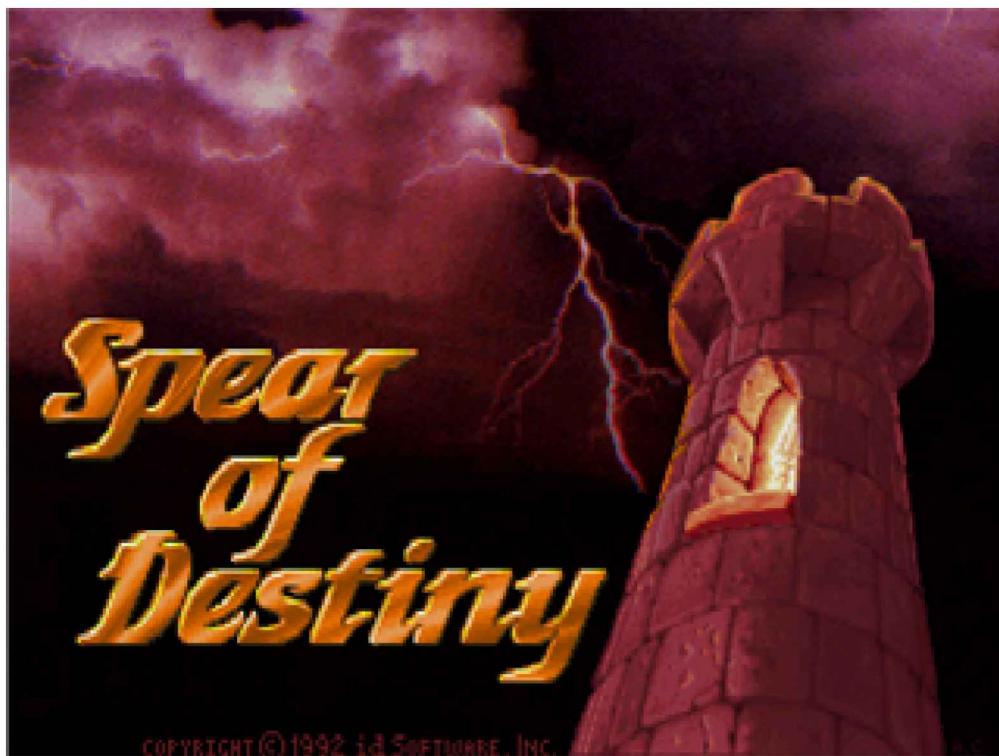
Wolfenstein 3D was a colossal financial success. Despite having multiple episodes, each made of 10 levels, the game was not long enough to satisfy many of its fans:

- Episode 1: Escape from Wolfenstein
- Episode 2: Operation: Eisenfaust
- Episode 3: Die, Fuhrer, Die!
- Episode 4: A Dark Secret
- Episode 5: Trail of the Madman
- Episode 6: Confrontation

id Software contracted other companies to create more maps and also worked on a sequel of its own.

### **5.1 Spear of Destiny**

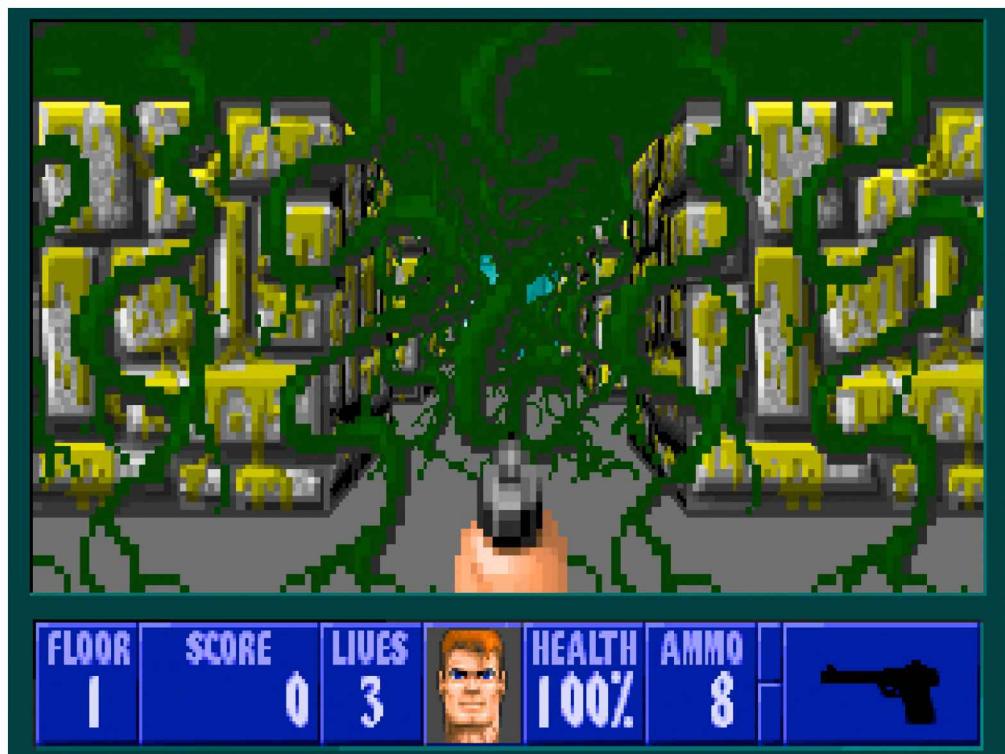
Released on September 18, 1992 and entirely done by id Software, Spear Of Destiny used the same game engine but with new graphics, music, and levels. It was a prequel to the adventures of Wolfenstein 3D's hero B.J. Blazkowicz and consisted of one additional episode made of 21 levels.



Working on a sequel made a lot of sense:

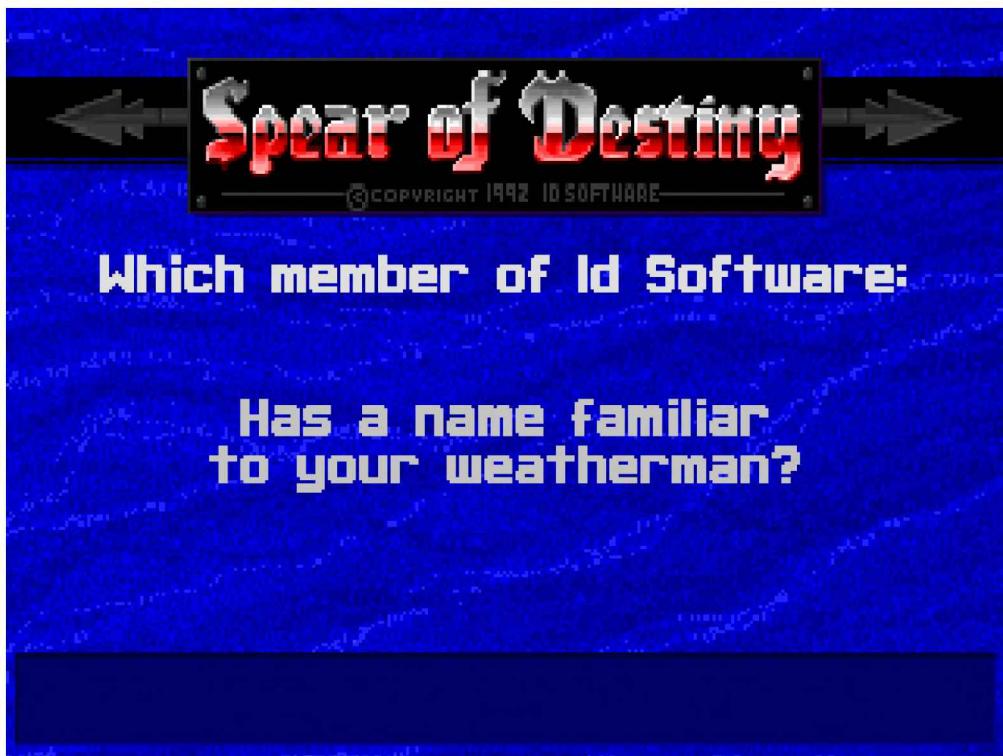
1. Sales were good, so it would have been silly not to take advantage of the momentum.
2. It gave the engineering team time to build the next generation of engine and tools while keeping the design and graphics teams busy.

Although it used the same engine, Spear Of Destiny was innovative in several ways. It introduced huge transparent sprites to simulate vegetation and attempted to break away from the orthogonal world constraint with clever map design leveraging all of each map's real estate.





To fight piracy, SOD shipped with copy protection typical of the early 90s. Since copying a disk was easy but photocopies were much more cumbersome, the game would refuse to start unless the player answered a question about something found in the game manual.



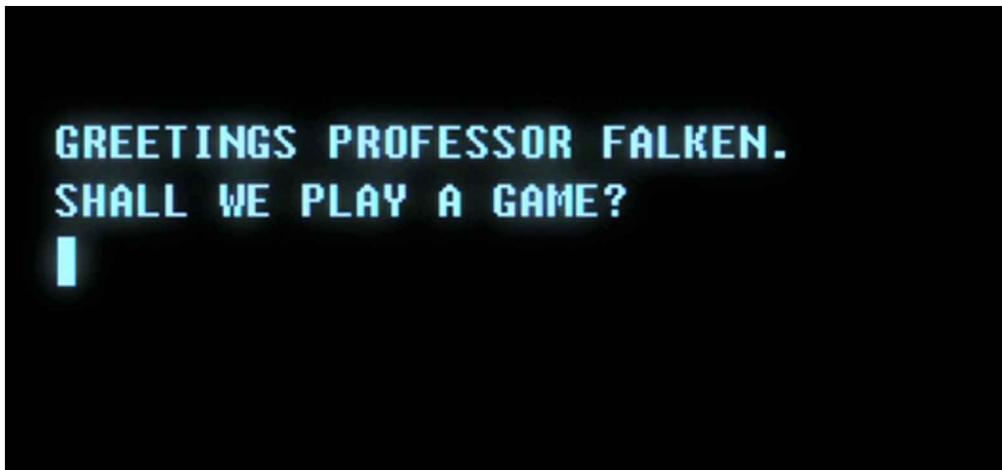
The copy protection mechanism featured backdoors which are probably private jokes.

```
BackDoorStrs[5][16] = {  
    "a spoon?",  
    "bite me!",  
    "joshua",  
    "pelt",  
    "beta"  
    "snoops"  
},
```

And the associated responses from the protection system:

```
GoodBoyStrs[10][40] = {  
    "...is the CORRECT ANSWER!",  
    "",  
  
    "Consider yourself bitten, sir.",  
    "",  
  
    "Greetings Professor Falken, would you",  
    "like to play Spear of Destiny?",  
  
    "Do you have any gold spray paint?",  
    "",  
  
    "Beta testing approved.",  
    "",  
  
    "I wish I had a 21\" monitor...",  
    ""  
},
```

Note the reference to the 1984 movie, WarGames:



# **Chapter 6**

## **Ports**

The software chapter described how tightly coupled the engine and PC hardware were. The architecture of the VGA, audio systems, or even a simple trick like the Linear Feedback Shift Register proved sometimes impossible to replicate with machines even 100 times more powerful than a 386<sup>1</sup>.

Except for Wolf4SDL (and Chocolate Wolfenstein 3D derived from it) no port was ever able to perfectly replicate the original PC DOS experience.

### **6.1 Super Nintendo**

Released in the United States in March 1994, the port to Super Nintendo was done during the development of Doom. id Software originally intended to subcontract the port but the programmer in charge failed to deliver.

---

<sup>1</sup>This is in contrast to Doom, which was created with portability as a core part of the development process.

“

Disaster Struck!

We all had to stop working on Doom. The Super Nintendo version that we had gotten a guy to work on was not finished. And Imagineer was not pleased. It had been since the summertime so it had been about nine months and they hadn't heard anything from us. That guy was working on the port and we could not get ahold of the guy. It was a nightmare situation. We could not get it done with somebody else. So, basically, we stopped working on Doom and ported Wolfenstein to the Super Nintendo. The whole team jumped on it. It took us like three weeks to just blast out the port. And that is including all rats in there and green blood.

John Romero - Programmer

”



The box is as sober as the content of the cartridge. In order to comply with Nintendo's "Game Content Guidelines", id had to drastically alter the graphics of the game and remove blood, dogs<sup>2</sup>, and all World War II-era German references. Things were also interesting from a technological stand point:

<sup>2</sup>They were replaced with rats.

When I ported Wolf to the SNES, the ray casting performance cost was too much, so I had to make a new wall span renderer. Learning about BSP trees allowed me to much more accurately resolve the culling challenges, and it worked out ok, leading the way to the Doom renderer.

Many years later, I made a very similar programming tradeoff for the mobile BREW version of Doom RPG. The J2ME (java) Doom RPG looked like Wolfenstein, with a tile map world, textured walls, and solid color floor and ceiling, but it was done with a quite nice wall span renderer. I had learned a thing or two since writing Wolf. For the ARM native code BREW version, I wanted to add texture mapped floors and ceilings with per-tile texture choices. I turned the tile maps into polygon windings and started writing a full texture mapped clipped polygon rasterizer for them, but I only had a couple days to work on the mobile renderer, and it became clear that I wasn't going to be able to deliver a really solid implementation in that time.

The solution was to sacrifice performance for implementation simplicity. Instead of trying to fill in just the empty pixels around the walls with floor / ceiling textures, I completely textured the entire screen with floor and ceiling textures before drawing the walls on top of them. With floor / ceiling symmetry and fixed texture sizes, it was pretty fast even with per-pixel tile map lookup, but most importantly it was rock solid, crack free, and completed in the window of time I had allotted to it.

#### **John Carmack - Programmer**

Switching the core algorithm from raycasting to BSP sorted wall span rasterization was not the only trick John Carmack had to use to get the game to run on Super NES. You may remember from page 15 that console graphics are based on sprite engines, which are orthogonal to Wolfenstein 3D's requirements for a framebuffer. The solution to this problem was to use the SNES Mode 7<sup>3</sup> where one background layer made of tiles 14x12 simulates a 112x96 framebuffer which is scaled up 2x to 224x160 thanks to the Mode 7 transformation matrix. The SNES PPU<sup>4</sup> sprite engine is used to draw the weapon and the status bar with 32x32 tiles.

---

<sup>3</sup>The same Mode 7 used for Super Mario Kart and F-Zero

<sup>4</sup>Pixel Processing Unit



On the screenshot above, the scaling is quite visible. The SNES resolution is 224x192 with the 3D scene tiles scaled 2x from 112x80 to 224x160. The status bar/weapon has a high resolution thanks to non-scaled 32x32 tiles.

## 6.2 Jaguar

The Jaguar port was released in 1994 and was the only other port (along with SNES) made by id Software. It was the only port able to reach 60 frames per seconds and ran at 240p resolution (320x240) with bilinear filtering allowing pleasant magnification. It also introduced two new weapons: the Flamethrower and Rocket Launcher.

The console was a mix of powerful components and severe bottlenecks as described by John Carmack on [slashdot.com](http://slashdot.com) in March 2000.

The memory, bus, bliter and video processor were 64 bits wide, but the processors (68k and two custom risc processors) were 32 bit.

The bliter could do basic texture mapping of horizontal and vertical spans, but because there wasn't any caching involved, every pixel caused two ram page misses and only used 1/4 of the 64 bit bus. Two 64 bit buffers would have easily tripled texture mapping performance. Unfortunate.

It could make better use of the 64 bit bus with Z buffered, shaded triangles, but that didn't make for compelling games.

It offered a usefull color space option that allowed you to do lighting effects based on a single channel, instead of RGB.

The video compositing engine was the most innovative part of the console. All of the characters in Wolf3D were done with just the back end scalar instead of bliting. Still, the experience with the limitations and hard failure cases of that gave me good ammunition to rail against Microsoft's (thankfully aborted) talisman project.

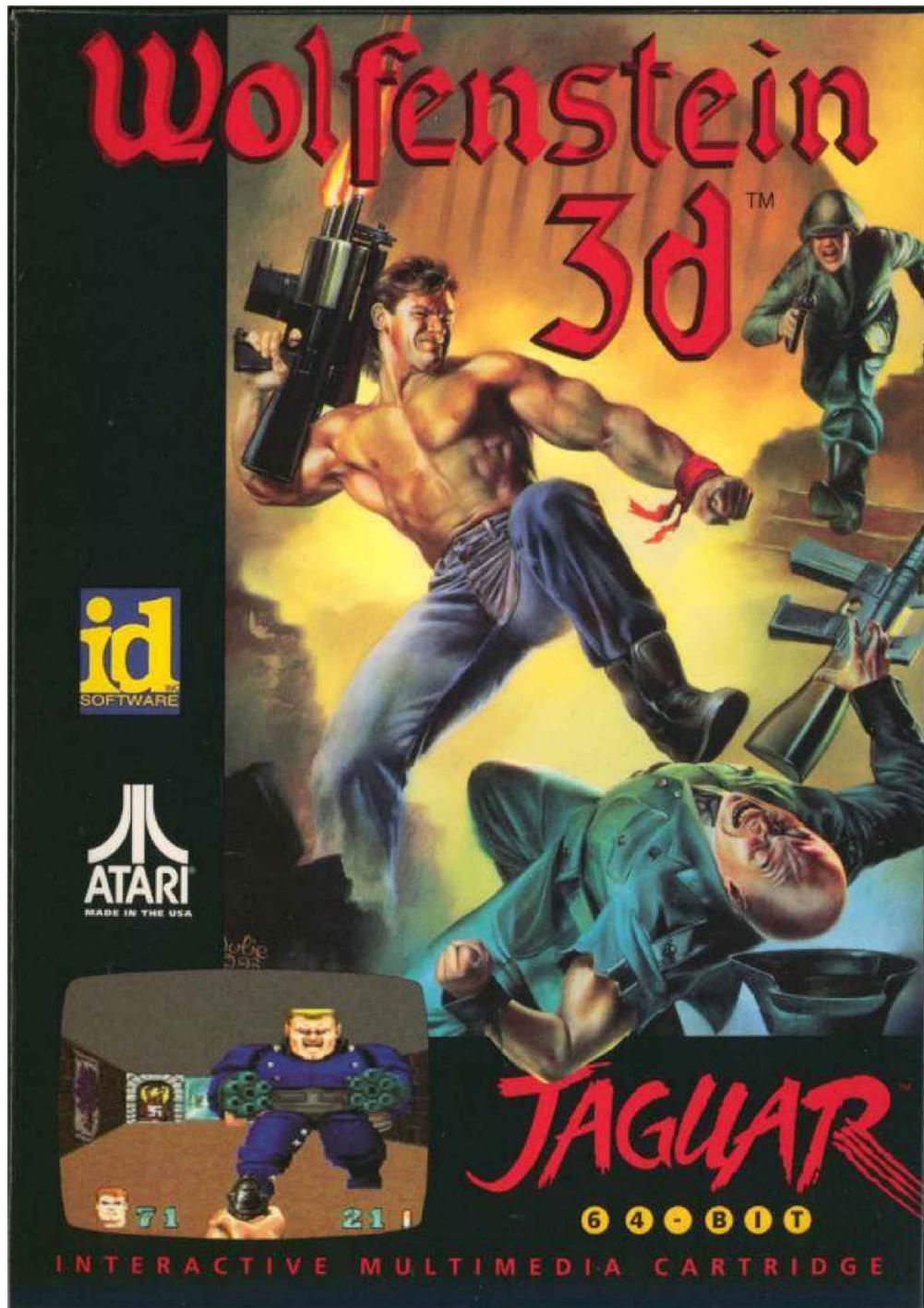
The little risc engines were decent processors. I was surprised that they didn't use off the shelf designs, but they basically worked ok. They had some design hazards (write after write) that didn't get fixed, but the only thing truly wrong with them was that they had scratchpad memory instead of caches, and couldn't execute code from main memory. I had to chunk the DOOM renderer into nine sequentially loaded overlays to get it working (with hindsight, I would have done it differently in about three...) ..

The 68k was slow. This was the primary problem of the system. Your options were either taking it easy, running everything on the 68k, and going slow, or sweating over lots of overlayed parallel asm chunks to make something go fast on the risc processors.

That is why playstation kicked so much ass for development – it was programmed like a single serial processor with a single fast accelerator.

If the Jaguar had dumped the 68k and offered a dynamic cache on the risc processors and had a tiny bit of buffering on the bliter, it could have put up a reasonable fight against Sony.

**John Carmack - Programmer (March 4th, 2000)**





Note the 4:3 aspect ratio mandated by NTSC standard. The machine gun sprite was not the one from the original Wolf3D, it was borrowed from Doom.

“

Jaguar wolf ran at 30 fps. In some places it could do 60, which was my first experience with 60 fps 3D graphics, but it was rare and you felt let down when it dropped to 30, so I forced it.

**John Carmack - Programmer**

”



**Figure 6.1:** A new weapon was introduced: flamethrower.

### 6.3 iPhone

Wolfenstein 3D was ported by John Carmack himself to iOS. The source code was released on the same day it was published on the Apple appstore.

The most notorious improvement is bilinear filtering, thanks to hardware accelerated rendering. Bilinear filtering is not for everybody. Some people complained the enemies were "blurry" and wanted to see the pixels. Following are the release notes of Wolfenstein3D iOS on March 25, 2009.



### 6.3.1 iPhone Development Notes

By John Carmack, Technical Director, Id Software

I had been frustrated for over a year with the fact that we didn't have any iPhone development projects going internally at Id. I love my iPhone, and I think the App Store is an extremely important model for the software business. Unfortunately, things have conspired against us being out early on the platform.

Robert Duffy and I spent a week early on starting to bring up the Orcs & Elves DS codebase on the iPhone, which would have been a nice project for a launch title, but it wasn't going to be a slam dunk. The iPhone graphics hardware is a more capable superset of the DS hardware (the driver overhead is far, far worse, though), but the codebase was fairly DS specific, with lots of Nintendo API calls all over the place. I got the basics drawing by converting things to OpenGL ES, but I was still on the fence as to whether the best approach to get all the picky little special effects working would be a complete GL conversion, or a DS graphics library emulation layer. Coupled with the fact that the entire user interface would need to be re-thought and re-tested, it was clear that the project would take several months of development time, and need artists and designers as well as coding work. I made the pitch that this would still be a good plan, but the idMobile team was already committed to

the Wolfenstein RPG project for conventional Java and BREW mobile phones, and Anna didn't want to slip a scheduled milestone on the established, successful development directions there for a speculative iPhone project.

After thinking about the platform's capabilities a bit more, I had a plan for an aggressive, iPhone specific project that we actually started putting some internal resources on, but the programmer tasked with it didn't work out and was let go. In an odd coincidence, an outside development team came to us with a proposal for a similar project on the Wii, and we decided to have them work on the iPhone project with us instead. We should be announcing this project soon, and it is cool. It is also late, but that's software development...

Late last year, the mobile team had finished up all the planned versions of Wolfenstein RPG, but EA had suggested that in addition to the hundreds of customized versions they normally produce for all the various mobile phones, they were interested in having another team do a significant media quality improvement on it for the iPhone. While Wolf RPG is a very finely crafted product for traditional cell phones, it wasn't designed for the iPhone's interface or capabilities, so it wouldn't be an ideal project, but it should still be worth doing. When we got the first build to test, I was pleased with how the high res artwork looked, but I was appalled at how slow it ran. It felt like one of the mid range java versions, not better than the high end BREW as I expected. I started to get a sinking feeling. I searched around in the level for a view that would confirm my suspicion, and when I found a clear enough view of some angled geometry I saw the tell-tale mid-polygon affine swim in the texture as I rotated. They were using the software rasterizer on the iPhone. I patted myself on the back a bit for the fact that the combination of my updated mobile renderer, the intelligent level design / restricted movement, and the hi-res artwork made the software renderer almost visually indistinguishable from a hardware renderer, but I was very unhappy about the implementation.

I told EA that we were NOT going to ship that as the first Id Software product on the iPhone. Using the iPhone's hardware 3D acceleration was a requirement, and it should be easy - when I did the second generation mobile renderer (written originally in java) it was layered on top of a class I named TinyGL that did the transform / clip / rasterize operations fairly close to OpenGL semantics, but in fixed point and with both horizontal and vertical rasterization options for perspective correction. The developers came back and said it would take two months and exceed their budget.

Rather than having a big confrontation over the issue, I told them to just send the project to me and I would do it myself. Cass Everitt had been doing some personal work on the iPhone, so he helped me get everything set up for local iPhone development here, which is a lot more tortuous than you would expect from an Apple product. As usual, my off the cuff estimate of "Two days!" was optimistic, but I did get it done in four, and the game is definitely more pleasant at 8x the frame rate.

And I had fun doing it.

Since we now were doing something resembling "real work" on the iPhone at the office, we kept it going at a low priority. One of the projects Cass was tinkering around with at home was a port of Quake 3, and we talked about different interface strategies every now and then.

Unfortunately, when we sat down to try a few things out, we found that Q3 wasn't really running fast enough to make good judgments on iPhone control systems. The hardware should be capable enough, but it will take some architectural changes to the rendering code to get the most out of it.

I was just starting to set up a framework to significantly revise Q3 when I considered the possibility of just going to an earlier codebase to experiment with initially. If we wanted to factor performance out of the equation, we could go all the way back to Wolfenstein 3D, the grandfather of FPS games. It had the basic run and gun play that has been built on for fifteen years, but it originally ran on 286 computers, so it should be pretty trivial to hold a good framerate on the iPhone.

Wolfenstein was originally written in Borland C and TASM for DOS, but I had open sourced the code long ago, and there were several projects that had updated the original code to work on OpenGL and modern operating systems. After a little looking around, I found Wolf3D Redux at <http://wolf3dredux.sourceforge.net/>. One of the development comments about "removal of the gangrenous 16 bit code" made me smile.

It was nice and simple to download, extract data from a commercial copy of Wolfenstein, and start playing on a PC at high resolution. Things weren't as smooth as they should be at first, but two little changes made a huge difference - going at VBL synced update rates with one tic per cycle instead of counting milliseconds to match 70 hz game tics, and fixing a bug with premature integralization in the angle update code that caused mouse movement to be notchier than it should be. The game was still fun to play after all these years, and I began to think that it might be worthwhile to actually make a product out of Wolfenstein on the iPhone, rather than just using it as a testbed, assuming the controls worked out as fun to play. The simple episodic nature of the game would make it easy to split up into a \$0.99 version with just the first episode, a more expensive version with all sixty levels, and we could release Spear of Destiny if there was additional demand. I was getting a little ahead of myself without a fun-to-play demonstration of feasibility on the iPhone, but the idea of moving the entire line of classic Id titles over - Wolf, Doom, Quake, Quake 2, and Quake Arena, was starting to sound like a real good idea.

I sent an email to the Wolf 3D Redux project maintainer to see if he might be interested in working on an iPhone project with us, but it had been over a year since the last update, and he must have moved on to other things. I thought about it a bit, and decided that I would go ahead and do the project myself. The "big projects" at Id are always top priority, but the systems programming work in Rage is largely completed, and the team hasn't

been gated on me for anything in a while. There is going to be memory and framerate optimization work going on until it ships, but I decided that I could spend a couple weeks away from Rage to work on the iPhone exclusively. Cass continued to help with iPhone system issues, I drafted Eric Will to create the few new art assets, and Christian Antkow did the audio work, but this was the first time I had taken full responsibility for an entire product in a very long time.

### Design Notes

The big question was how "classic" should we leave the game? I have bought various incarnations of Super Mario Bros on at least four Nintendo platforms, so I think there is something to be said for the classics, but there were so many options for improvement. The walls and sprites in the game were originally all 64 x 64 x 8 bit color, and the sound effects were either 8khz / 8 bit mono or (sometimes truly awful) FM synth sounds. Changing these would be trivial from a coding standpoint. In the end, I decided to leave the game media pretty much unchanged, but tweak the game play a little bit, and build a new user framework around the core play experience. This decision was made a lot easier by the fact that we were right around the 10 meg over-the-air app download limit with the converted media. This would probably be the only Id project to ever be within hailing distance of that mark, so we should try to fit it in.

The original in-game status bar display had to go, because the user's thumbs were expected to cover much of that area. We could have gone with just floating stats, but I thought that BJ's face added a lot of personality to the game, so I wanted to leave that in the middle of the screen. Unfortunately, the way the weapon graphics were drawn, especially the knife, caused issues if they were just drawn above the existing face graphics. I had a wider background created for the face, and used the extra space for directional damage indicators, which was a nice improvement in the gameplay. It was a tough decision to stop there on damage feedback, because a lot of little things with view roll kicks, shaped screen blends, and even double vision or blurring effects, are all pretty easy to add and quite effective, but getting farther away from "classic".

I started out with an explicit "open door" button like the original game, but I quickly decided to just make that automatic. Wolf and Doom had explicit "use" buttons, but we did away with them on Quake with contact or proximity activation on everything. Modern games have generally brought explicit activation back by situationally overriding attack, but hunting for push walls in Wolf by shooting every tile wouldn't work out. There were some combat tactics involving explicitly shutting doors that are gone with automatic-use, and some secret push walls are trivially found when you pick up an item in front of them now, but this was definitely the right decision.

You could switch weapons in Wolf, but almost nobody actually did, except for occasion-

ally conserving ammo with the chain gun, or challenges like "beat the game with only the knife". That functionality didn't justify the interface clutter.

The concept of "lives" was still in wolf, with 1-ups and extras at certain scores. We ditched that in Doom, which was actually sort of innovative at the time, since action games on computers and consoles were still very much take-the-quarter arcade oriented. I miss the concept of "score" in a lot of games today, but I think the finite and granular nature of the enemies, tasks, and items in Wolf is better suited to end-of-level stats, so I removed both lives and score, but added persistent awards for par time, 100% kills, 100% secrets, and 100% treasures. The award alone wasn't enough incentive to make treasures relevant, so I turned them into uncapped +1 health crumbs, which makes you always happy to find them.

I increased the pickup radius for items, which avoided the mild frustration of having to sometimes make a couple passes at an item when you are cleaning up a room full of stuff.

I doubled the starting ammo on a fresh level start. If a player just got killed, it isn't good to frustrate them even more with a severe ammo conservation constraint. There was some debate about the right way to handle death: respawn with the level as is (good in that you can keep making progress if you just get one more shot off each time, bad in that weapon pickups are no longer available), respawn just as you entered the level (good - keep your machinegun / chaingun, bad - you might have 1 health), or, what I chose, restart the map with basic stats just as if you had started the map from the menu.

There are 60 levels in the original Wolf dataset, and I wanted people to have the freedom to easily jump around between different levels and skills, so there is no enforcement of starting at the beginning. The challenge is to /complete /a level, not /get to/ a level. It is fun to start filling in the grid of level completions and awards, and it often feels better to try a different level after a death. The only exception to the start-anywhere option is that you must find the entrance to the secret levels before you can start a new game there.

In watching the early testers, the biggest issue I saw was people sliding off doors before they opened, and having to maneuver back around to go through. In Wolf, as far as collision detection was concerned, everything was just a 64 x 64 tile map that was either solid or passable. Doors changed the tile state when they completed opening or began closing. There was discussion about magnetizing the view angle towards doors, or somehow beveling the areas around the doors, but it turned out to be pretty easy to make the door tiles only have a solid central core against the player, so players would slide into the "notch" with the door until it opened. This made a huge improvement in playability.

There is definitely something to be said for a game that loads in a few seconds, with automatic save of your position when you exit. I did a lot of testing by playing the game, exiting to take notes in the iPhone notepad, then restarting Wolf to resume playing. Not having to skip through animated logos at the start is nice. We got this pretty much by accident with

the very small and simple nature of Wolf, but I think it is worth specifically optimizing for in future titles.

The original point of this project was to investigate FPS control schemes for the iPhone, and a lot of testing was done with different schemes and parameters. I was sort of hoping that there would be one "obviously correct" way to control it, but it doesn't turn out to be the case.

For a casual first time player, it is clearly best to have a single forward / back / turn control stick and a fire button.

Tilt control is confusing for first exposure to the game, but I think it does add to the fun factor when you use it. I like the tilt-to-move option, but people that play a lot of driving games on the iPhone seem to like tilt-to-turn, where you are sort of driving BJ through the levels. Tilt needs a decent deadband, and a little bit of filtering is good. I was surprised that the precision on the accelerometer was only a couple degrees, which makes it poorly suited for any direct mapped usage, but it works well enough as a relative speed control.

Serious console gamers tend to take to the "dual stick" control modes easily for movement, but the placement of the fire button is problematic. Using an index finger to fire is effective but uncomfortable. I see many players just move the thumb to fire, using strafe movement for fine tuning aim. It is almost tempting to try to hijack the side volume switch for fire, but the ergonomics aren't quite right, and it would be very un-Apple-like, and wouldn't be available on the iPod touch (plus I couldn't figure out how...) ..

We tried a tilt-forward to fire to allow you to keep your thumbs on the dual control sticks, but it didn't work out very well. Forward / back tilt has the inherent variable holding angle problem for anything, and a binary transition point is hard for people to hold without continuous feedback. Better visual feedback on the current angle and trip point would help, but we didn't pursue it much. For a game with just, say, a rocket launcher, shake/shove-to-fire might be interesting, but it isn't any good for wolf.

It was critical for the control sticks to be analog, since digital direction pads have proven quite ineffective on touch screens due to progressive lack of registration during play. With an analog stick, the player has continuous visual feedback of the stick position in most cases, so they can self correct. Tuning the deadband and slide off behavior are important.

Level design criteria has advanced a lot since Wolfenstein, but I wasn't going to open up the option of us modifying the levels, even though the start of the first level is painfully bad for a first time player, with the tiny, symmetric rooms for them to get their nose mashed into walls and turned around in. The idea is that you started the game in a prison cell after bashing your guard over the head, but even with the exact same game tools, we would lead the player through the experience much better now. Some of the levels are still great

fun to play, and it is interesting to read Tom Hall and John Romero's designer notes in the old hint manuals, but the truth is that some levels were scrubbed out in only a couple hours, unlike the long process of testing and adjustment that goes on today.

It was only after I thought I was basically done with the game that Tim Willits pointed out the elephant in the gameplay room - for 95% of players, wandering around lost in a maze isn't very much fun. Implementing an automap was pretty straightforward, and it probably added more to the enjoyment of the game than anything else. Before adding this, I thought that only a truly negligible amount of people would actually finish all 60 levels, but now I think there might be enough people that get through them to justify bringing the Spear of Destiny levels over later.

When I was first thinking about the project I sort of assumed that we wouldn't bother with music, but Wolf3D Redux already had code that converted the old id music format into ogg, so we would end up with support at the beginning, and it turned out pretty good. We wound up ripping the red book audio tracks from one of the later commercial Wolf releases and encoding at a different bitrate, but I probably wouldn't have bothered if not for the initial support. It would have been nice to re-record the music with a high quality MIDI synth, but we didn't have the original MIDI source, and Christian said that the conversion back from the id music format to midi was a little spotty, and would take a fair amount of work to get right. I emailed Bobby Prince, the original composer, to see if he had any high quality versions still around, but he didn't get back with me.

The game is definitely simplistic by modern standards, but it still has its moments. Getting the drop on a brown shirt just as he is pulling his pistol from the holster. Making an SS do the "twitchy dance" with your machine gun. Rounding a corner and unloading your weapon on ... a potted plant. Simplistic plays well on the iPhone.

### Programming Notes

Cass and I got the game running on the iPhone very quickly, but I was a little disappointed that various issues around the graphics driver, the input processing, and the process scheduling meant that doing a locked-at-60-hz game on the iPhone wasn't really possible. I hope to take these up with Apple at some point in the future, but it meant that Wolf would be a roughly two tick game. It is only "roughly" because there is no swapinterval support, and the timer scheduling has a lot of variability in it. It doesn't seem to matter all that much, the play is still smooth and fun, but I would have liked to at least contrast it with the perfect limit case.

It turns out that there were a couple issues that required work even at 30hz. For a game like Wolf, any PC that is in use today is essentially infinitely fast, and the Wolf3D Redux code did some things that were convenient but wasteful. That is often exactly the right

thing to do, but the iPhone isn't quite as infinitely fast as a desktop PC.

Wolfenstein (and Doom) originally drew the characters as sparse stretched columns of solid pixels (vertical instead of horizontal for efficiency in interleaved planar mode-X VGA), but OpenGL versions need to generate a square texture with transparent pixels. Typically this is then drawn by either alpha blending or alpha testing a big quad that is mostly empty space. You could play through several early levels of Wolf without this being a problem, but in later levels there are often large fields of dozens of items that stack up to enough overdraw to max out the GPU and drop the framerate to 20 fps. The solution is to bound the solid pixels in the texture and only draw that restricted area, which solves the problem with most items, but Wolf has a few different heavily used ceiling lamp textures that have a small lamp at the top and a thin but full width shadow at the bottom. A single bounds doesn't exclude many texels, so I wound up including two bounds, which made them render many times faster.

The other problem was CPU related. Wolf3d Redux used the original ray casting scheme to find out which walls were visible, then called a routine to draw each wall tile with OpenGL calls. The code looked something like this:

```
DrawWall( int wallNum ) {
    char name[128];
    texture_t *tex;
    sprintf( name, "walls/%d.tga", wallNum );
    tex = FindTexture( name );
    ...
}
texture_t FindTexture( const char *name ) {
    int i;
    for ( i = 0 ; i < numTextures ; i++ ) {
        if ( !strcmp( name, texture[name]->name ) ) {
            return texture[name];
        }
    }
    ...
}
```

I winced when I saw that at the top of the instruments profile, but again, you could play all the early levels that only had twenty or thirty visible tiles at a time without it actually being a problem.

However, some later levels with huge open areas could have over a hundred visible tiles, and that led to 20hz again. The solution was a trivial change to something resembling:

```
DrawWall( int wallNum ) {
    texture_t *tex = wallTextures[wallNum];
    ...
}
```

Wolf3D Redux included a utility that extracted the variously packed media from the original games and turned them into cleaner files with modern formats. Unfortunately, an attempt at increasing the quality of the original art assets by using hq2x graphics scaling to turn the 64x64 art into better filtered 128x128 arts was causing lots of sprites to have fringes around them due to incorrect handling of alpha borders. It wasn't possible to fix it up at load time, so I had to do the proper outline-with-color-but-0-alpha operations in a modified version of the extractor. I also decided to do all the format conversion and mip generation there, so there was no significant CPU time spent during texture loading, helping to keep the load time down. I experimented with the PVRTC formats, but while it would have been ok for the walls, unlike with DXT you can't get a lossless alpha mask out of it, so it wouldn't have worked for the sprites. Besides, you really don't want to mess with the carefully chosen pixels in a 64x64 block very much when you scale it larger than the screen on occasion.

I also had to make one last minute hack change to the original media - the Red Cross organization had asserted their trademark rights over red crosses (sigh) some time after we released the original Wolfenstein 3D game, and all new game releases must not use red crosses on white backgrounds as health symbols. One single, solitary sprite graphic got modified for this release.

User interface code was the first thing I started making other programmers do at Id when I no longer had to write every line of code in a project, because I usually find it tedious and unrewarding. This was such a small project that I went ahead and did it myself, and I learned an interesting little thing. Traditionally, UI code has separate drawing and input processing code, but on a touchscreen device, it often works well to do a combined "immediate mode interface", with code like this:

```
if ( DrawPicWithTouch( x, y, w, h, name ) ) {
    menuState = newState;
}
```

Doing that for the floating user gameplay input controls would introduce a frame of response latency, but for menus and such, it works very well.

One of the worst moments during the development was when I was getting ready to hook up the automatic savegame on app exit. There wasn't any savegame code. I went back and grabbed the original 16 bit dos code for load / save game, but when I compiled I found out that the Wolf3d Redux codebase had changed a lot more than just the near / far pointer issues, asm code, and comment blocks. The changes were sensible things, like grouping more variables into structures and defining enums for more things, but it did mean that I

wasn't dealing with the commercially tested core that I thought I was. It also meant that I was a lot more concerned about a strange enemy lerping through the world bug I had seen a couple times.

I seriously considered going back to the virgin codebase and reimplementing the OpenGL rendering from scratch. The other thing that bothered me about the Redux codebase was that it was basically a graft of the Wolf3D code into the middle of a gutted Quake 2 codebase. This was cool in some ways, because it gave us a console, cvars, and the system / OpenGL portable framework, and it was clear the original intention was to move towards multiplayer functionality, but it was a lot of bloat. The original wolf code was only a few dozen C files, while the framework around it here was several times that.

Looking through the original code brought back some memories. I stopped signing code files years ago, but the top of WL\_MAIN.C made me smile:

```
/*
=====
          WOLFENSTEIN 3-D
          An Id Software production
          by John Carmack
=====
*/
```

It wasn't dated, but that would have been in 1991.

In the end, I decided to stick with the Redux codebase, but I got a lot more free with hacking big chunks of it out. I reimplemented load / save game (fixing the inevitable pointer bugs involved), and by littering asserts throughout the code, I tracked the other problem down to an issue with making a signed comparison against one of the new enum types that compare as unsigned. I'm still not positive if this was the right call, since the codebase is sort of a mess with lots of vestigial code that doesn't really do anything, and I don't have time to clean it all up right now.

Of course, someone else is welcome to do that. The full source code for the commercial app is available on the web site. There was a little thought given to the fact that if I had reverted to the virgin source, the project wouldn't be required to be under the GPL. Wolf and the app store presents a sort of unique situation - a user can't just compile the code and choose not to pay for the app, because most users aren't registered developers, and the data isn't readily available, but there is actually some level of commercial risk in the fast-moving iPhone development community. It will not be hard to take the code that is already fun to play, pull a bunch of fun things off the net out of various projects people have done with the code over the years, dust off some old map editors, and load up with some modern quality art and sound.

Everyone is perfectly within their rights to go do that, and they can aggressively try to bury the original game if they want. However, I think there is actually a pretty good opportunity for cooperation. If anyone makes a quality product and links to the original Wolf app, we can start having links to "wolf derived" or "wolf related" projects. That should turn out to be a win for everyone.

I'm going back to Rage for a while, but I do expect Classic Doom to come fairly soon for the iPhone.

## 6.4 Wolfenstein 3D-VR

Around 1994 a company called Alternate World Technologies worked on an adaptation of the Wolfenstein 3D engine, designed to work with head-mounted displays, under special license from id Software. AWT worked on three games: Wolfenstein VR, Blake Stone VR, and Cybertag VR, all based on the Wolfenstein engine code, with Cybertag being the only one playable in multiplayer for up to four players. A 3D artist by the name of Tom Roe, who uploaded a video on YouTube<sup>5</sup>, helped with some of the finer details of this project.

---

<sup>5</sup>[https://www.youtube.com/watch?v=l9n\\_TkFltk](https://www.youtube.com/watch?v=l9n_TkFltk) .

The Wolfenstein VR game was developed by a small game development company based in Louisville, KY back in 1993. Through an agreement with Id, Alternate Worlds Technologies, Inc., (AWT) licensed the original Wolfenstein 3D engine and used Polhemus tracker technology to create a head mounted display version of the game. The design team also modified all of the animation sequences to look more like green alien blood rather than red blood; an attempt to reduce the apparent violence in the game. Amusing by today's standards for sure.

I was not personally involved in the design work on Wolfenstein as I arrived after they were already shipping arcade units with this title. They also created a similar experience with Blake Stone. The Wolfenstein VR game was a single player game, though the Wolfenstein engine was later used to create a multi-player game which you mentioned, Cybertag VR. Where four players could speak to each other through a headset while playing tag in a virtual environment. I began my work with AWT as a 3d artist developing levels for a new game engine which used character and level models designed in 3d Studio. I also used Deluxe Paint to create animation sequences for Cybertag VR.

**Tom Roe - 3D artist**

The Wolfenstein VR project had no chance of success. You could rotate your head to turn in the game, but people playing the game never did, continuing to face forward and just using the joystick. The developers that put it together were very enthusiastic, but it was just premature. Even with my knowledge today, I don't think I could do a decent VR experience on the PC hardware of the day back then.

**John Carmack**





# Chapter 7

## Epilogue

After the release of Wolfenstein 3D, id Software kept itself busy and became a video game industry powerhouse. id went on to publish ten more games over the next two decades.

Name	Engine	Release date
Doom	idTech 1	December 1993
Doom II	idTech 1	September 1994
Quake	idTech 2	June 1996
Quake II	idTech 2	December 1997
Quake III	idTech 3	December 1999
Doom III	idTech 4	August 2004
Wolfenstein 3D: iOS	idTech X	March 2009
Rage	idTech 5	October 2011
Doom 3: BFG Edition	idTech 4.5 <sup>1</sup>	October 2012
Doom	idTech 5	May 2016

*Figure 7.1:* id Software titles.

Not only did the team drive technological progress by producing new game engines, they also heavily influenced the industry by licensing their technology throughout the early 90s. A rough estimate accounts for 80 games powered by id Software's engines.

Beyond software, id heavily influenced the graphic hardware industry. With VQuake and GLQuake offering support for 3D acceleration cards, id helped to drive sales for Rendition's Verite 1000, 3dFx's Voodoo, and later PowerVR. Doom 3 famously used OpenGL at a time where many studios succumbed to Microsoft's DirectX API.

<sup>1</sup>idTech 4 boosted with subsystems from idTech 5 such as the job system to take advantage of multi-core architecture.

id remained true to its Right Thing to Do roots and continued releasing the source code of each previous engine, helping countless programmers educate themselves with recent technology. Sadly, this practice stopped with the Rage (id Tech 5) engine.

Source Code name	Release date	Delta from game
Wolfenstein 3D	July 21, 1995	3 years, 2 months
Doom	December 23, 1997	3 years, 2 months
Quake	December 21, 1999	3 years, 6 months
Quake II	December 21, 2001	4 years, 0 months
Quake III	August 19, 2005,	5 years, 8 months
Doom III	November 22, 2011	7 years, 3 months
Wolfenstein 3D: iOS	March 25, 2009	Same day
Doom 3: BFG Edition	October 2012	Same day

**Figure 7.2:** id Software source code releases.

The Wolfenstein 3D franchise lives on to this day, having been licensed to numerous studios. At E3 2017, Bethesda announced Wolfenstein II: The New Colossus, a sequel to The New Order. At the time of the writing of this book, it is scheduled for release on October 27, 2017.

## 7.1 Where Are They Now?

**Tom Hall** left id Software shortly after the release of Spear of Destiny, during the development of Doom in 1992. He worked at Apogee and later Ion Storm and was involved in some great games of the 90s such as Rise of the Triad, Terminal Velocity and Anachronox. Tom now lives in the San Francisco Bay Area and is Senior Creative Director of mobile games at Glu's GluPlay studio, producing titles such as Cooking DASH, Diner DASH, and Gordon Ramsay DASH.

**John Romero** left id Software after the release of Quake. He contributed massively to Doom/Quake and also developed the licensing business branch of id. His "all in-hand"<sup>2</sup> solution based on the id Tech 1 engine (the same powering Doom) notably helped Raven publish Heretic and Hexen. He later founded other companies such as Ion Storm and Monkeystone Games, the latter of which was a precursor to the field of mobile game development. He now lives in Ireland and is working on an upcoming title called Blackroom.

---

<sup>2</sup>Engine + Tools recommendation (NeXT) + Mentoring.

**Adrian Carmack** left id Software after the release of Doom III and retired from the video game industry. He lives in USA and recently announced his collaboration with John Romero on Blackroom.

**John Carmack** remained with id Software until 2013, after which he joined Oculus VR as CTO. He has received several awards for his accomplishments, including two Emmy awards for "Science, Engineering & Technology ", a Game Developers Conference Lifetime Achievement, and a BAFTA Fellowship Award.

**Jay Wilbur** worked on Doom, Doom II, Final Doom, and Quake. He left to become Vice President of Business Development at Epic Games.

**Kevin Cloud** is the last man standing at id Software of the original Wolfenstein 3D team. He was an owner until Zenimax Media's acquisition of the studio and is now an Executive Producer.



# **Appendices**



# **Appendix A**

## **Before Wolfenstein 3D**

Wolfenstein 3D was not the first FPS the team produced. Back when they still had obligations to Softdisk, they worked on two games: Hovertank 3D and Catacomb 3D.

### **A.1 Hovertank 3D**

Hovertank 3D (a.k.a Hovertank, Hovertank 3-D or Hovertank One) is a vehicular combat game published by Softdisk in April, 1991. Set during a nuclear war, the game puts the player in control of a tank. The goal of the game is to kill mutated monsters and rescue survivors. John Carmack's research for the game's engine took six weeks, two weeks longer than any engine he wrote before. There was no texture mapping on the walls or the floor/ceiling and the 3D engine was targeted at EGA (16 colors). The pace of the game was slow and it had no music. The digitized audio effects were just Romero making noises into a microphone!

### **A.2 Catacomb 3D**

Catacomb 3D (a.k.a Catacomb 3-D: A New Dimension, Catacomb 3-D: The Descent, and Catacombs 3) was released in November 1991. The game put the player in the shoes of a magician fighting goblins and orcs. The engine was improved with texture mapping for the walls. While still using EGA and in 16 colors, the game looked much better than Hovertank 3D thanks to improved assets. The pace of the game was also set to be faster. Note that players could destroy walls with fireballs.







## Appendix B

# XMS vs EMS

Accessing past the first MiB of RAM was still difficult in 1991. Stuck in real mode with a 20 bit address bus, games and applications had to rely on two types of RAM drivers: EMS and XMS.

### B.1 EMS: Expanded Memory Specification

Developed in 1985 by Lotus, Intel, and Microsoft, LIM EMS was originally designed to pilot hardware memory boards. Like with sound cards, graphic cards, and network cards, a customer could purchase an EMS memory board to increase the RAM capacity of a machine. Applications could access the newly added RAM via the EMS driver.

It's garbage! It's a kludge! ... But we're going to do it.

Bill Gates - Microsoft

EMS is built on the idea of memory mapping, where 64 KiB of RAM in conventional memory called a "page frame" is divided into four units of 16KiB called "pages". These pages are windows into extended memory which can be read or written.

Many memory boards were available for Intel 286 machines but they were expensive. In 1989 a RapidRAM 2 MiB board<sup>1</sup> could be purchased for \$1,495<sup>2</sup>. A SUPERAM 4 MiB

---

<sup>1</sup>[http://www.atarimagazines.com/compute/issue112/Memory\\_Expansion\\_Blocks.php](http://www.atarimagazines.com/compute/issue112/Memory_Expansion_Blocks.php)

<sup>2</sup>Adjusted to inflation, \$1,495 in 1989 is equivalent to \$2,951.25 in 2017.

cost \$3,199<sup>3</sup>.

Upon arriving on the market, the Intel 386 changed the EMS landscape dramatically. Thanks to its new Virtual 8086 Mode, the driver EMM386.EXE<sup>4</sup> was able to run DOS in a virtual machine. EMS memory could be emulated in software using normal RAM which made hardware EMS boards obsolete.

The purpose of a V86 task is to form a "virtual machine" with which to execute an 8086 program. A complete virtual machine consists not only of 80386 hardware but also of systems software. Thus, the emulation of an 8086 is the result of cooperation between hardware and software:

The hardware provides a virtual set of registers (via the TSS), a virtual memory space (the first megabyte of the linear address space of the task), and directly executes all instructions that deal with these registers and with this address space.

#### INTEL 80386 PROGRAMMER'S REFERENCE MANUAL

Michal Necasek explains well how the EMM386 driver operated:

The idea was simple in principle, but quite complex in its implementation. The V86 mode requires the CPU to be in protected mode, but DOS is not a protected-mode operating system. Therefore, EMM386 had to include a miniature 32-bit protected-mode operating system; that was a necessity, not a feature. One of the most important tasks of this mini-OS was setting up page tables and enabling paging, a major new feature of the i386 CPU.

Paging was how EMM386 did its magic. Swapping memory blocks in and out of the EMS page frame (located in the first megabyte of RAM and directly accessible by real-mode DOS applications) was accomplished by reprogramming the page tables and thus controlling which physical memory pages mapped to a given linear address. No memory copying was involved and the mechanism was very similar to how 8086 EMS boards worked, but used only the CPU's built-in memory management facilities instead of relying on external hardware.

Michal Necasek - os2museum.com

---

<sup>3</sup>Adjusted to inflation, \$3,199 in 1989 is equivalent to \$6,315.08 in 2017.

<sup>4</sup>Short for Extended Memory Manager for 386.

The implementation relying on virtual memory meant EMS-emulated RAM suffered no performance penalty. The only limitation was the 16 KiB size of each page.

## B.2 XMS: eXtended Memory Specification

In 1988, Lotus, Intel, Microsoft, and AST gathered to produce another standard: eXtended Memory Specification. The goal was to provide a more flexible API than EMS, closer to what programmers used, and allow for bigger chunks of data to be manipulated. The driver emulating XMS RAM, `HIMEM.SYS`, also shipped with MS-DOS. A user could elect to create XMS RAM by adding a line to `CONFIG.SYS`.

Because the size of data to manipulate was arbitrary, `HIMEM.SYS` could not use the same V86 trick as `EMM386.EXE`. How does one access RAM outside of the addressable space without virtual memory? On a 386, it was easy: the driver switched the CPU into protected mode, performed whatever was asked, and switched the CPU back to real mode.

On a 286 it was more complicated. Intel architects never envisioned programmers would want to go back to real mode from protected mode. As a result, there was no documented way to transition from protected mode to real mode.

That did not stop Microsoft engineers from trying to do it anyway. They figured they could use the soft reboot provided via the keyboard combination of Control-Alt-Del. When detecting this pattern the i8042 keyboard controller resets the CPU and asks the BIOS to initialize the machine. This operation takes several seconds and would not have been usable. By writing a special value to a special memory location it was possible to prevent the long BIOS initialization and "just" reset the CPU. This trick effectively transitioned a 286 from protected mode to real mode but it was slow and took a full millisecond to complete.

A different (and faster) technique surfaced later, involving generating a triple fault (faulting in a fault handler by invalidating the IDTR and causing an interrupt).

Finally, when `HIMEM.SYS` v2.06 came out, it removed the need to even leave real mode. It used the undocumented `LOADALL` instruction to control hidden registers offsetting all RAM access<sup>5</sup>.

---

<sup>5</sup>"HIMEM.SYS, Unreal mode, and LOADALL" - <http://www.os2museum.com/wp/himem-sys-unreal-mode-and-loadall/>.

### B.3 What It Meant For Wolfenstein 3D

The XMS API provided the ability to work on a dataset bigger than 64 KiB but Wolfenstein 3D did not require this since it works on small textures/sprites sequentially. The need for copy between extended RAM and conventional RAM made XMS RAM 10 times slower than EMS RAM. In this context EMS was far more attractive than XMS for Wolfenstein 3D.

## Appendix C

# The 640KB Barrier

The problems with conventional memory limitations were so bad that most games has to ship with explanations about how everything worked. Here is an extract from W3DHELP.EXE.

### THE 640K BARRIER

---

This section isn't actually needed in order to get our programs running. What is contained in here is for the most part background information to better assist our customers in understanding why they need to make more conventional memory available.

When MicroSoft first made DOS 1.0, 640 kilobytes (KB) was set aside as the highest amount of memory that a computer could have. The 640KB of memory is what is called "conventional memory". To maintain compatibility with older versions, this was never changed. Advances in memory management have made access to memory beyond 640KB, but this memory can only hold data; the program actually has to run in the first 640KB. This first 640k is called "Conventional Memory".

Here is a brief discussion of the different types of memory available on your computer. The most important one is Conventional memory.

CONVENTIONNAL MEMORY starts at 0k and normally ends at 640k . (The instances where this is not the case are EXTREMELY

rare) If you are not using some sort of memory manager (such as DOS's EMM386, Quarterdeck's QEMM or Qualitas'386MAX), this is the only type of memory you have. Conventional memory is used by DOS as well as device drivers and TSR's (Terminate and Stay Resident Programs). A TSR is a program that is loaded into your computer's memory (usually from the CONFIG.SYS or AUTOEXEC.BAT files) and stays there. Host programs remove themselves from memory after execution, a TSR does not. Device drivers and TSR's are programs that enable the computer to use additional hardware such as a mouse, scanner, CD-ROM, expanded or extended memory, etc. A program such as an Apogee game is NOT a program that can be loaded as a TSR. If all you have is conventional memory, anything that you would load as a TSR would come out of this section of memory. Take too much away, and you're not left over with enough memory to run our product.

If you are getting an out of memory error from our program, it is this memory that you are running out of. Whether you have 1 meg, 8 meg of memory, or 32 meg of memory, it's irrelevant. Only the first 640k of memory is available for program execution. Please do not confuse this with hard drive space. Your hard drive space is not memory, and is not relevant nor should be considered in this example.

UPPER MEMORY starts at 640k and ends at 1024k. Normally, this area is used for things such as system ROM, video and hardware cards, and the like. On most PC's hardware does not use the entire upper memory area, and with the use of the aforementioned memory managers, (EMM386, QEMM, 386MAX, etc). you can move some TSR's into this memory area. These unused areas are called Upper Memory Blocks (UME'S), and this is where some TSR's can be loaded.

EXTENDED MEMORY (XMS) is the memory addressed above 1024k. Extended memory requires the use of a memory manager, such as MS/DOS's HIMEM.SYS. This region of memory is not usable for standard program execution; it can only be used for data storage. Aogee programs that use this type of memory (such as Wolfenstein & Blake Stone), only use this to store level or graphic data. The actual program itself is running in conventional memory.

HIGH MEMORY HREH (HMH) is the first 64k of extended memory. This is a special region of memory that is most commonly used to load DOS high. When you issue the DOS:HIGH command in your config.sys file, the amount of conventional memory that was previously being occupied by DOS itself is moved into this region.

EXPANDED MEMORY (EMS) is another type of memory that some MS/DOS programs can make use of. Like XMS, this memory is not available for program execution, it's only used for data storage due to it's nature. An explanation of this type of memory is rather technical, so it will not be delved into here. If you're curious, check your DOS manual, or your memory manager manual.

When you first start up your computer, there are two files that your computer looks at: CONFIG.SYS and AUTUEXEC.BAT. These two files contain lists of device drivers and TSR's that are automatically run when starting your computer. Each of these takes up space, and it is taken away from the 640k of conventional memory. As more and more programs are loaded from the autoexec.bat and config.sys files, you have less and less available from the original 640k. Since it is this memory that programs run in, you can see that the amount taken away from the programs executed in config.sys and autoexec.bat would want to be kept to a minimum. This can be accomplished by either reducing the amount of programs loaded in from config.sys and autoexec.bat, or moving them to high memory via the use of EMM386, QEMM, 386 MAX, or some other memory management program.



## Appendix D

# CONFIG.SYS and AUTOEXEC.BAT

At startup, the operating system reads two files automatically from the booting device (which can be either the hard-drive C: or floppy disk A:). Each line in CONFIG.SYS instructs DOS to load a device drive or configure where to load something in RAM.

CONFIG.SYS:

```
DEVICE=C:\WINDOWS\HIMEM.SYS  
DOS=HIGH,UMB  
DEVICEHIGH=C:\WINDOWS\EMM386.EXE AUTO RAM  
DEVICE=C:\WINDOWS\MOUSE.SYS
```

The file AUTOEXEC.BAT is more like a batch file used to define variable. Notice BLASTER variable which is parsed in Wolfenstein3D to know how to talk to the sound card.

AUTOEXEC.BAT:

```
@echo off  
SET SOUND=C:\CREATIVE\CTSND  
SET BLASTER=A220 I5 D1 H5 P330 E620 T6  
SET PATH=C:\DOS;C:\
```



## Appendix E

### Good Stuff

Two emails showing the impact of the game:

```
To : ROMERO , TOM
From : LOTHAR/JAY
Date : 9 Aug 92 21:06:46
Subject : AOL Message
X-mailer : Pegasus Mail v2.3 (R2).

Subj : DREAMS , FLASHBACKS
Date : 92-08-09 03:59:55 EDT
From : Tug Hill 2
Posted on: America Online
```

On a serious note...

As a former POW (Vietnam), I hesitated to play WOLF for over a month after downloading as I feared flashbacks. I didn't want to remember all that I had been through all those years ago, when, as POW's, my friend and I decided an escape attempt would be better than a slow death by torture and starvation.

My friend and I made crude maps and hoarded food. The day of the escape we clubbed the guard with stones, took his gun and fought our way through two levels of underground tunnels (only a few guards and had to crawl). I made it, my friend didn't.

Dreams...NO! NIGHTMARES...YES!! However, the more I play WOLF the less frequently I have nightmares. The chilling part is turning a corner and seeing a guard with his gun drawn.

WOLF is a powerful game. Fearful as well. I believe that a person should face the past. So... when I can play EPISODE 1 comfortably (no nightmares), I plan on ordering the full series.

Don't let a few bad dreams make you discard this game.

Subj: Wolf-3D    Section: Action/Arcade  
Games  
From: Ty Graham 72350,2636    # 191387, \* No Replies \*  
To: Id Software 72600,1333 Date: 24-Jul-92 18:27:27

Jay, just thought I'd drop a note to let you know how popular Wolf3D is here at Microsoft. It seems like I can't walk down a hall without hearing 'Mein Leben' from someone's office. I hope you guys are getting revenue from all this.

Anyway, we were sitting around talking the other day, discussing games for Windows, and someone said 'What are those cool guys at Id doing?'. So how about it. Are you guys looking at Win games at all? Win32?

In a perfect world, I'd have you guys port the Wolf engine to a multiusermaze game for Windows for Workgroups. We need a good M'user Win game.

Anyway some thoughts.

Ty Graham (Microsoft)

## Appendix F

# Release Notes by John Carmack

RELEASE.TXT

-----

::

We are releasing this code for the entertainment of the user community. We don't guarantee that anything even builds in here. Projects just seem to rot when you leave them alone for long periods of time.

This is all the source we have relating to the original PC wolfenstein 3D project. We haven't looked at this stuff in years, and I would probably be horribly embarrassed to dig through my old code, so please don't ask any questions about it. The original project was built in borland c++ 3.0. I think some minor changes were required for later versions.

You will need the data from a released version of wolf or spear to use the exe built from this code. You can just use a shareware version if you are really cheap.

Some coding comments in retrospect:

The ray casting refresh architecture is still reasonably appropriate for the game. A BSP based texture mapper could go faster, but ray casting was a lot simpler to do at the

time.

The dynamically compiled scaling routines are now a Bad Thing. On uncached machines (the original target) they are the fastest possible way to scale walls, but on modern processors you just wind up thrashing the code cache and wrecking performance. A simple looping texture mapper would be faster on 486+ machines.

The whole page manager caching scheme is unnecessarily complex.

Way too many #ifdefs in the code!

Some project ideas with this code:

Add new monsters or weapons.

Add taller walls and vertical motion. This should only be done if the texture mapper is rewritten.

Convert to a 32 bit compiler. This would be a fair amount of work, but I would hate to even mess with crusty old 16 bit code. The code would get a LOT smaller.

Make a multi-player game that runs on DOOM sersetup / ipxsetup drivers.

Have fun...

John Carmack  
Technical Director  
Id Software

README.TXT

-----

NOTES:

This version will compile under BORLAND C++ 3.0/3.1 and compiled perfectly before it was uploaded. Please do not send your questions to id Software.

## Appendix G

# 20th Anniversary Commentary

For the 20th anniversary, John Carmack revisited and played the game with commentaries.  
Here is the transcript :

### G.1 The engine (4:00 mark)

The two previous 3D games Hovertank and catacombs 3D were done in an object space rendering where it drew limited polygons. They were one-dimensional in terms that they were just line segments that were restricted on axeses.

We had something that resembled a polygon rasterizer and a polygon clipper and those were both done in four to six weeks a piece. I had really quite a bit difficulty with it. Going back in time twenty years, there weren't all of the references in existence, books and tutorials on this subject.

I was having a hard time getting some of that stuff to be as robust and reliable as it needed to be. You could get a few freak out cases in Catacombs 3D and one of my real goals was to simplify it enough that it would be really rock solid and robust.

The fact that Wolfenstein 3D came out with a speedup have more to do with the poor quality of my previous implementation because when Wolfenstein needed to gain some speed on much poorer platforms like the Super Nintendo I went back to more of a rasterization approach with BSP trees rather than ray casting.

Wolfenstein did wind up being both more efficient and more robust than my previous implementations but it was all wild-west for me back then :I was figuring it all out as I went along and there were a lot of things to kinda look at.

As example I remember that we had the first level running at about three months in. We followed it up with spear destiny on there before moving to doom.

It was very short amount of time. We leveraged the toolset that we were using to create the 2d games the Commander Keen series where we had a good tile editor that John Romero had developed and since the Wolfenstein maps were basically simple tile maps we were able to make things happen really quickly on that and also the maps were just so quick to create: we had several maps in shipping products that were literally done in one day somebody would go scrub out a map we played a bunch of times tweak things a little bit and it would go in the project there. and it's always interesting looking back at the games there where you look at the source code and it all fits in one directory its one handful C files and a few ASM files and there's just not that much to it when we look at what we're doing today and it seems that you know we can't throw a dialog up on the screen without invoking ten different frameworks and fifty thousand lines of code.

## G.2 Modding

The thing that stands out a lot was the first (and it influenced a lot of what iD Software did afterwards) was that people kinda tweaking with the projects. People figuring out how to unpacked the levels. That wasn't straightforward because We were trying to fit on floppy disks at this time and I had developed all this compression technology course: The funny thing is in hindsight I independently reinvented LZSS coding (in very ad hoc ways).

People figured all these out extracted everything and started making character editors and level editors. Neither of those were designed to be done and they weren't straightforward. The characters were in this column packed format that made it more efficient to draw without transparency test but made it really difficult to figure out what these original pictures looked like.

These people started doing these really neat things with it. Once we just recognized that there was a large body of people that wanted to do this, it influenced a lot of our future decisions in Doom and Quake about making it actually easy and straightforward and encouraging people to undertake modding.

I have cleared recollections to this day and especially at that time about thinking back to when I was getting into gaming when I was a teenager about how I wish that I could have that kinda access to the inside and guts of the games that I played. I can remember breaking out the Apple II sector editors to give myself lots of gold in Ultima III and that type of stuff and wishing that I had the ability to look at the source code for those old titles and being able to make that type of things come true as we were later able to later release the full source code as well as the modding tools for the games has been something that am

really proud of in my career.

### G.3 Texture mapping (13:30 mark)

Normally when you're decent distance from things everything smoothly expands on there but if you notice when you get up close and start going off the edge of the screen you start getting more quantization in that. That is a result for the fact that the core graphics technology behind the texture mapping in this timeframe was compiled scalars: There is actually a different little section of assembly language code that was programmatically generated to draw a 64 tall piece of graphics:

```
two pixels tall  
four pixels tall  
six pixels tall  
  
and so on...
```

all the way up to the full height of the screen but it started taking up too much memory and would run out in a 640k system if I let them stretch all the way up to the largest possible scale you get there in steps by one so it started taking some shortcuts and saving a little bit has it got bigger





**How was Wolfenstein 3D made and what were the secrets of its speed? How did id Software manage to turn a machine designed to display static images for word processing and spreadsheet applications into the best gaming platform in the world, capable of running games at seventy frames per seconds? If you have ever asked yourself these questions, *Game Engine Black Book* is for you.**

This is an engineering book. You will not find much prose in here (the author's English is broken anyway.) Instead, this book has only bit of text and plenty of drawings attempting to describe in great detail the Wolfenstein 3D game engine and its hardware, the IBM PC with an Intel 386 CPU and a VGA graphic card.

***Game Engine Black Book* details techniques such as raycasting, compiled scalers, deferred rendition, VGA Mode-Y, linear feedback shift register, fixed point arithmetic, pulse width modulation, runtime generated code, self-modifying code, and many others tricks. Open up to discover the architecture of the software which pioneered the First Person Shooter genre.**

