GAME ENGINE BLACK BOOK

Wolfenstein 3D

FABIEN SANGLARD

Contents

Preface

For the past 10 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 .NET`. After a few days I deleted `quakesrc` folder, unable to make sense of anything and discouraged.

A few years later I came across legendary programmer and technical writer Michael Abrash's Graphics Programming Black Book. His marvellous articles gave away the big picture of Quake engine and detailed some of its algorithms and data structures. Equiped with this precious knowledge I went back to the code, read and understood it to the 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 published them on *fabiensanglard.net*. Over the years, I proceeded to publish more than fifty articles, selecting legendary games such as Doom, Quake or Out Of This World. I would open the engine, explore the subsystems, understand the overall architecture and draw a map that hopefully sparkled interest and encouraged other adventurous programmers.

Sharing my knowledge was a rewarding experience: Not only the feedback from readers was overwhelmingly positive, to try to explain something in simple terms was also a way to make sure I mastered a topic. It also allowed me to become a better teacher and learn to rely extensively on good drawings. A picture is worth more than a thousand words[1]. Those skills proved invaluable in my career which lead me to Google.

Eventually I decided to take my articles to the next level and came up with this book which I hope will be the first of a trilogy:

1. Wolfenstein 3D and the i386 CPU.
2. Doom and the i486 CPU.
3. Quake and the Pentium CPU.

Focusing first on what the hardware was capable of and then how the software brought it to live.

[1] "Code: The Hidden Language of Computer Hardware and Software" by Charles Petzold is a superb example.

It may seem like a waste of time to read those “old” engine dedicated to extinct machines, compilers and operating systems. But there are tremendous values in them: Not only they are packed with clever tricks, they remind us of the constraints programmers from the past had to overcome. They remind us of the spirit it took to reach new frontiers. That spirit never died, it is in all off us who keep on trying to build things. If anything, I hope this book will remind to people who struggle today that others have struggled before. You are not alone. Believe in yourself and keep on aiming for The Right Think to Do[2] :) !

THIS DOS NOT BELONG HERE: delete everything after this!!

I started to program in the years 2000 and was dismayed to have missed the golden era of the 90s where developers wrote their own engine from scratch (idtech serie, build, src, goldSrc, unreal, dark, trespasser, ...). Only to catch on the mobile revolution where I did get to enjoy some success, even writing an app that was #1 worldwide in 2009. I later found out that other developers felt the same about their era. John Carmack recently described the same feeling during his Fellowship, BAFTA speech:

> “I can remember when I was a teenager: I thought I had missed the golden age of 8 bits Apple II gaming. And I was never gonna be Richard Garriot. And time went by and I got to make my own marks and things after that.
>
> The opportunities that I had are not there for people today but they are new and better one.”

I agree: There is no golden era. There are only dreams and the difficult path of excellence leading to them. This source code is a testimony of the talent and determination it once took to make a dream become true. It inspired me to emulate myself and I hope it will benefit you as well.

[2]The Right Thing To Do was first brought to fame and extensively discussed in "Hackers: Heroes of the Computer Revolution" by Steven Levy". It was also often mentioned in John Carmack journal using the finger protocol.

Chapter 1

Introduction

Wolfenstein 3D was released in May 1992 and created the First Person Shooter genre. The beautiful graphics, high framerate, crisp sounds effect and engaging musics were universally acclaimed. Within a year more than 100,000 units had been sold over the mail (the game was distributed via shareware: You had to mail money to get the full version), bringing fame and a little bit of fortune to the team of people who built it: id Software.

The many fans did not stop at beating the game. Because they wanted to modify it and make their own characters and maps, they started to explore and reverse engineer the engine. Within a few months the assets formats were well known and people released mods[1] with altered graphics, sounds effects, music and maps. The core of the game however, the 3D engine and the secrets of its speed remained mostly unknown.

It was kept secret for an obvious reason: A good game engine was considered the main asset of a game company. As a mean to outperform competitors it was a good business practice to keep other programmers uneducated in order to gain technological advantage and generate profit.

But 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 many internal debate, id Software did the unthinkable: On July 21, 1995 they uploaded a zip archive on *ftp.idsoftware.com* containing the full source code of the engine with all instructions to build it[2].

[1] MODified version.

[2] They were not totally crazy: They had built a game engine making Wolfenstein 3D obsolete: Doom was released on December 10, 1993

> 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 not only allow MODding and educate programers: It had two unforeseen consequences:

It allowed the software to live long after the target hardware and operating system died. 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.

A second unexpected side effect was the creation of a window back in time looking right into 1992. As a technical writer on *fabiensanglard.net* I thought i would never take a look at these old thing. But when out of curiosity I took a peek, I could not stop reading: I was mesmerized. The more I read, the more I came to realize one very important thing: The IBM PC was designed for office work, not gaming. It was meant to crunch integers, word processing, spreadsheet and static screens. What id Software did in 1993 was not just program a machine: They repurposed something built to do office work and turned it into the best gaming platform in the world[3].

But why go through so much trouble? After all if you were a game company and you wanted to make video games, you had hardware dedicated to this very specific thing: video game console! The Megadrive, the NES and the Neo-Geo had sprite engine which albeit small limitation such as size and number allowed to move something on the screen by simply updating its (x, y) coordinates. Heck if really you wanted to use a personal computer for gaming why not use an Amiga which was packed with coprocessors dedicated to make smooth animation?

The reason is: framebuffer. The kind of game they wanted to do could not be done with a sprite engine. Every frames, 70 times per seconds, it has to draw a full screen pixel per pixel in the framebuffer and send it to the screen. That task required a particularly powerful CPU. No console or Amiga could do the job as the mips[4] graph shows.

[3] A few other companies, such as Origins and LucasArts where also doing amazing things: I focus on Id because we have the source code!

[4] Million Instructions Per Second.

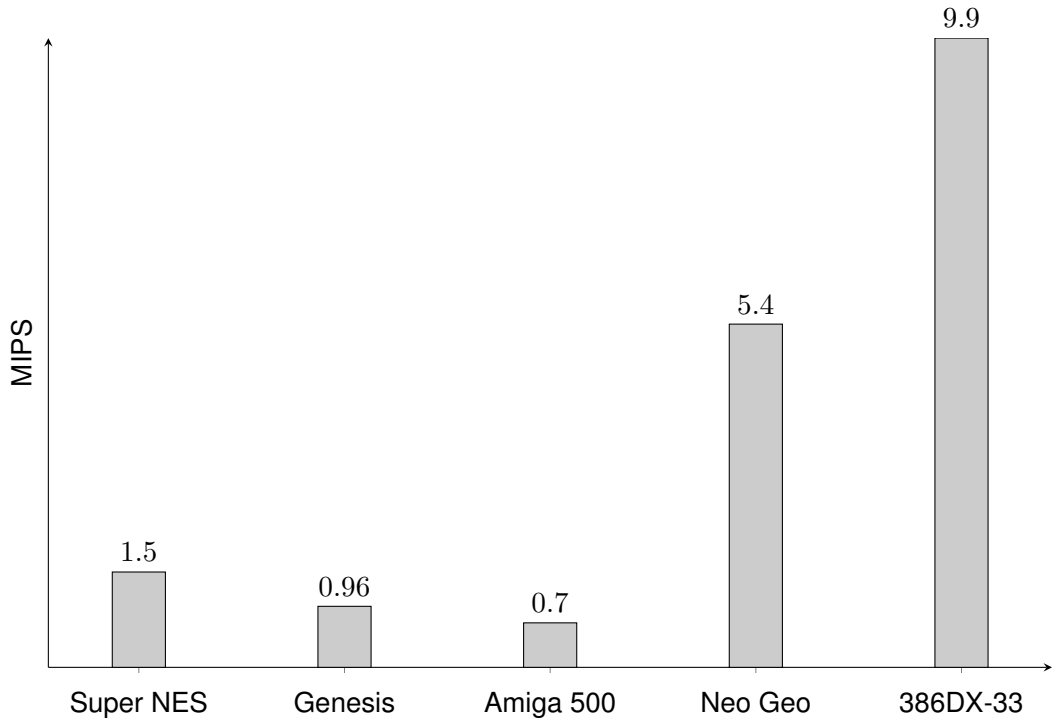


Figure 1.1: Game Console Vs PC: Raw power of the CPUs.

PC had a fast CPU and a framebuffer but everything else was a nightmare:

- Smooth animation was next to impossible.
- CPU could only do integer operations, 3D requires fractions.
- RAM was segmented resulting in complex and error prone model.
- The pixel of the VGA were not square: the framebuffer was distorted on screen.
- Various sound systems had various capabilities and expectations.
- Various RAM drivers exposed the RAM and made it available differently.

At first sight it did not look like much could be done to make a 3D game come true. Yet id made it happen thanks to creativity and hard work.
The book emphasis on the contrast between what the machine were theoritically capable of, the people who worked on it and the software they produced to innovate and reach new grounds:

- Chapter I: Hardware: The limits.
- Chapter II: People: The team pushing the edges.
- Chapter III: The result: Wolfenstein 3D engine.

Credited as the demise of all other types of computer such as the Amiga (reference HERE as footnote).
Ultimately all consoles abandonned the sprite engine and onboarded big fast cpus.
Racing the beam.
Disclaimer : The description that follows is technical and will most likely appeal to programmers. If you are more into the human aspect of game programming I would recommend instead to read David Kushner's chef d'oeuvre: "Masters of Doom".

Chapter 2

Hardware

PC performances in the 90s were so overwhelmingly dominated by their CPU that they were simply referred to not by their brand or model but by their Intel chip. If your PC has an Intel 80386SX in it, it was a 386. If it had an Intel 80286, it was a 286. Wolfenstein was target at 386s with degraded performances (yet still playable) on 286s.

In order to understand the machine I broke it down in five components part of a pipeline: Input, Cpu, Ram, Video and Audio.

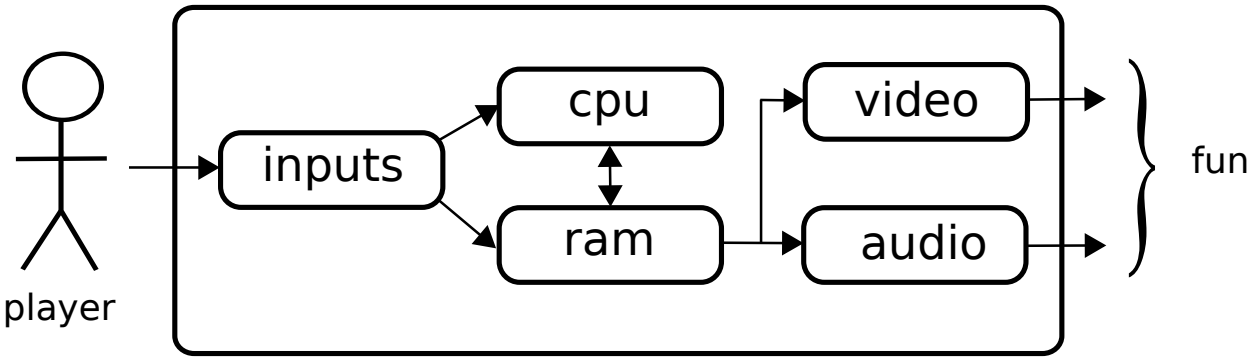


Figure 2.1: Hardware pipeline.

None of theses part were good for game although some were worse than others:

Stage	Quality
RAM	Very Poor
Video	Very Poor
Audio	Poor
Inputs	Good
CPU	Poor

Figure 2.2: Component quality for a game engine.

[1]

Overall, the pipeline offered a lot of resistance: hardware manufacturers had not embraced the game industry yet and it showed a lot. Ever since their inception in the 70s, IBM Personal Computers were designed to display static images and crunch integers. Real-time 3D, fractions and smooth sixty-frames-per-seconds animations were part of the blueprints.

2.1 Central Processing Unit

2.1.1 Overview

The ubiquitous CPU manufacturer was Intel with its x86 line of microprocessors. The machines based on the 80286 released in 1982 were on the decline and progressively replaced by Intel's first 32 bits processor: The 80386. Moore's law was in full swing as can be seen on a mips[2] histogram:

[1] Even though the CPU was powerful it was very good only at something next to useless for a 3D engine: Integer arithmetic.

[2] Million Instructions Per Second.

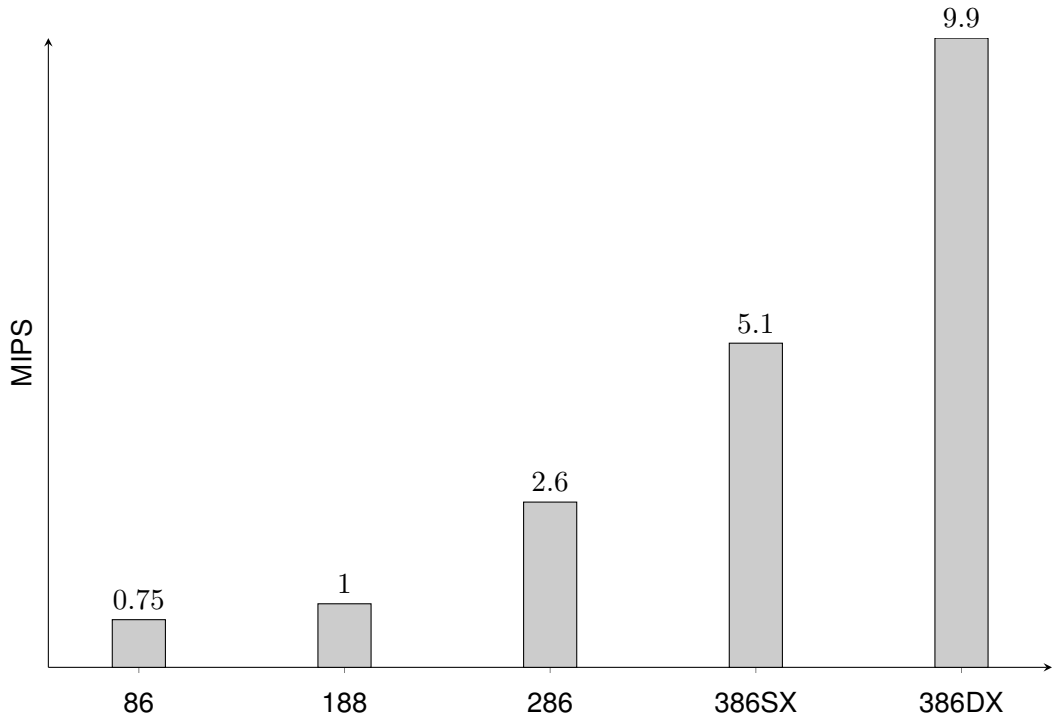


Figure 2.3: Processor speeds comparison.

Trivia : A modern processor such as the Intel Core i7 3.33 GHz operates at close to 180,000 Mips: Five orders of magnitude faster!

Trivia : Two other companies were producing Intel clones: AMD and Cyrix. The mediocre performances did not justify the lower cost and as a result they never gathered a significant market share. Interestingly AMD evolved to become a serious challenger while Cyrix merged with National Semiconductor in 1997.

Trivia : The 386SX and 386DX were identical processor. Yet the DX is twice faster than the SX thanks to a data bus between the CPU and the RAM twice as wide (32 bits vs 16 bits) !

2.1.2 Floating Point

PC 286s and 386s were powerful machines far outperforming any game console on the market. But all that power was not necessarily useful. In order to perform all the trigonometry for a '3D' game, the machines had to keep track of the fractional part of each operations. This is not an issue since the C language offers the types `float` and `double` precisely for that.

It is important to understand what floating point is and how it works to fully grasp how useful it is when doing maths. `float` are 32 bits 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:

- One bit S for the sign.
- Seven bits E for the exponent.
- Twenty four bits for the mantissa.

Figure 2.4: Floating Point internals

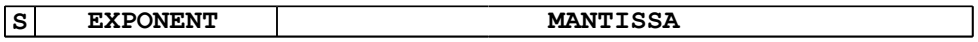


Figure 2.5: Floating Point three sections

How numbers are stored and interpreted is usually explained with the formula 2.1).

$$(-1)^S * 1.M * 2^{(E-127)} \tag{2.1}$$

Although correct, this way to explain floating point usually leaves programmers completely clueless. I blame this ignoble notation for discouraging generations of programmers, scaring them to the point they never looked back to understand how floating point actually works. Fortunately, there is a better way to explain it: Instead of Exponent, think of a floating window between two consecutive power of two integers. And instead of a Mantissa think of an Offset within that window as in Figure 2.11:

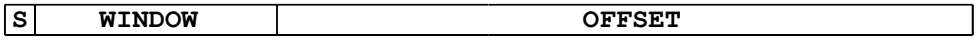


Figure 2.6: Alternative Floating Point internals

Figure 2.7 illustrate how the number 6.1 would be encoded, with the window starting at 4 (and therefore spanning up to next power of two: 8). The offset about half way down the window.

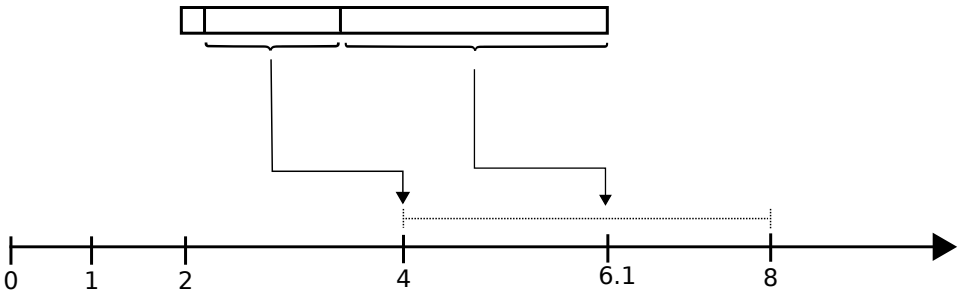


Figure 2.7: Value 6.1 approximated with floating point

Here is a detailled example which calculate the floating point representation of the number 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 $2^1 \rightarrow E = 128$.
- Finally there are 2^{23} offsets available so 3.14 is at $\frac{4-3.14}{2} = 0.43$. The mantissa/offset $\rightarrow M = 2^{23} * 0.43 = 4781506$

Which in binary translate to:

- S = 0.
- E = 10000000
- M = 10010001111011111000011

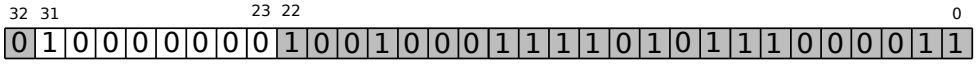


Figure 2.8: Floating Point internals

The value 3.14 is therefore approximated to 3.140000104904175.

The corresponding value with the ugly and useless formula:

$$3.14 = (-1)^0 * 1.57 * 2^{(128-127)} \tag{2.2}$$

And finally the graphic representation:

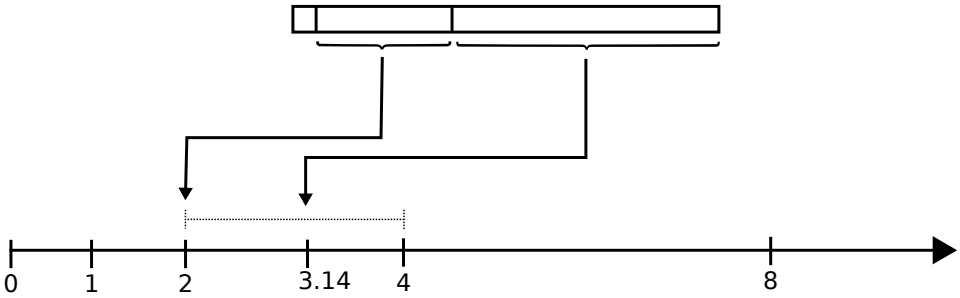


Figure 2.9: 3.14 window and offset.

Floating Point are very handy: They can represent very small values and huge ones while keeping track of fractional part of a number and also protecting from overflow by floating the

window when necessary.

They are neat but computational intensive: In order to add, subtract, multiply or divide two numbers, they have to be expressed with the same window. Which mean converting one number to the representation used by the other usually with higher precision than 32 bits (typically 80 bits on Intel FPUs).

This is not a problem when everything is hardwired...but it was a big problem for the 286 and the 386: *They did not have a hardware Floating Point Unit*. Those operations were emulated in software by the compiler and therefore terribly slow. So slow they were not usable for anything real-time.

Trivia : Since floating point units where so slow, why did the C language end up with a `float` and `double` types ? The machine used to invent the language (PDP-11) did not have a floating point but the manufacturer (DEC) had promised Dennis Ritchie and Ken Thompson the next model would have one[3]. Being astronomy enthusiasts they decided to add those two types to their language. Had they decided not to, C and C++ would have never enjoyed the same popularity!

[3]The Development of the C Language by Dennis M. Ritchie

Trivia : Some PCs did have hardware FPU. Intel sold what was marked as "Math CoProcessor". Intented to scientists, performances were average, price outrageous and sales mediocre :

Figure 2.10: Intel 1991 ad for "Math CoProcessor".

Overall the CPU was not promising for a 3D game: 286 and 386 *int* operations were fast but not accurate enough and *float* operations were accurate but not fast enough.

2.2 RAM

Intel CPUs could operate in two modes:

- The old backward compatible Real Mode. Available *only* to allow old software to run. This memory model replicated how the Intel 8086 CPU from the 70s operated: 20 bits memory bus, 1MB addressable RAM with an awful segmented addressing system.
- The new Protected Mode with a 32 bits memory bus wide offering up to 4 GB of RAM protectable with a MMU[4].

[4]Memory management Unit

The machine started in Real Mode for compatibility reasons. You may assume that programers of the 90s switched the CPU in Protected Mode to unleash the full potential of the machines and ditch the 20 years old Real Mode. Unfortunately, there was a major obstacle between this nice feature and the programmer. The infamous operating system, MS-DOS 5.0 by Microsoft Corporation.

2.2.1 DOS limitations

Microsoft Corporation highly valued the applications running on their operating systems. They were adamant to never ever break anything with a new system. Since many applications were written during the 80s on machine having only Real-Mode, DOS 5.0 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 that came with every machine sold prevented programmers from using Protected Mode. They were forced to ignore all the neat features of their 1992 PC and use the machine like it was a very fast Intel 8086 CPU from 1976.

Trivia : One year earlier, in 1991, a student from the University of Helsinki started working on a "hobby" of his: An operating system able to use the CPU in Protected Mode taking advantage of the MMU and the 32 bits registers. It would become Microsoft worse nightmare. Linus Torvalds had just started what would become Linux.

2.2.2 The infamous Real Mode: 1MB RAM max

Protected mode was unavailable. What had the Real-Mode to offer ? Essentially a trip back in time to 1976: A 20 bits wide address bus offering only 1MB of addressable RAM. No matter how much memory was installed on the machine in 1992, only 1MB could be addressed. And to top it all, addressing had to be done not with the 32 bits registers available but by combining two 16 bits register together: One was the segment, the other an offset within that segment. Hence the name: ’16 bits programming’.

Here is a brief description of the memory layout:

- 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 (around 620KB).
- From A0000h to FFFFFh : UMA (Upper Memory Area): Reserved to bios ROM, video card and sound card mapped I/O.

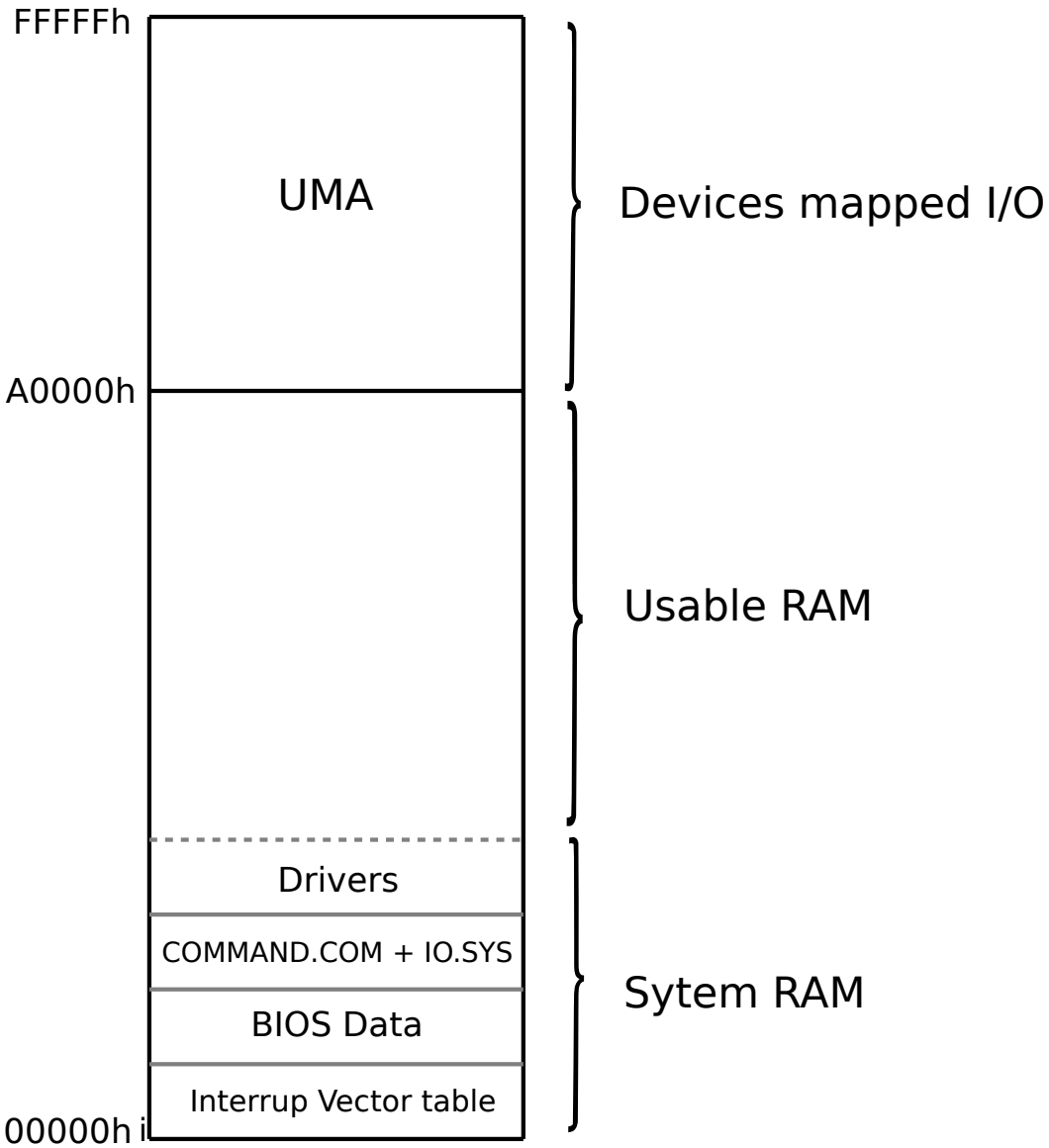


Figure 2.11: 3.14 window and offset.

As a result, out of the original 1024KB, only 640KB (called Conventional Memory) were usable. 384KB were reserved for the UMA[5] and every single driver installed (`.SYS` and `.COM`) took away from the remaining 640KB.

Trivia : In France people had to load KEYBFR.SYS driver so their AZERTY keyboard keys woud be properly mapped. The driver consumed a whopping 5KB of Conventional Memory.

[5]Upper Memory Area

2.2.3 The infamous Real Mode: 16 bits Segmented addressing

Real Mode could not use 32 bits registers for addressing. Everything had to be done by combining two 16 bits registers as seen in Figure 2.12. There were two kinds of RAM pointers: near and far. In the case of a far pointer, a 16 bits segment register would be shifted left 4 bits and combined with an other 16 bits offset register

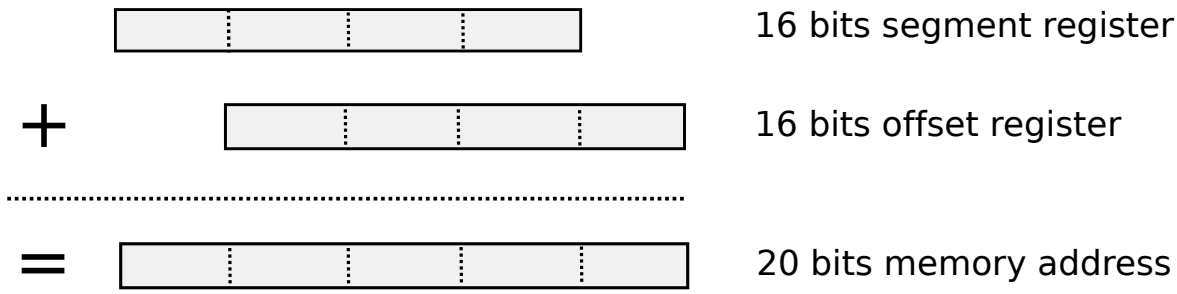


Figure 2.12: How registers were combined to address memory.

A near pointer was just a 16 bits offset that would point in the same segment. That may not

sound too bad but in practice this segmented addressing led to many issues. The least problematic was about the language: Since the C language was invented on 32 bits machine, the language had to be augmented by compiler manufacturers. That is how the `near` and `far` keywords came to existence. To build pointers, a set of macro would be provided, respectively `FP_SEG` and `FP_OFF`. That was an ugly wart on the beautiful C but that was manageable.

The really big issue however was how two pointers, referring to the same address could fail an equality test: In this model, the 1MB of RAM is divided in 65536 paragraphs by the segment pointer. So a paragraph is 16 bytes but an offset can be up to 65536 !!! See the following example:

Pointer A defined as:

```
    0000 0000 0000 0000 0000  Segment, 16 bits shifted 4 bits left
  +      0000 0001 0010 0000  Offet,   16 bits
============================
    0000 0000 0001 0010 0000  Address, 20 bits
```

Pointer B defined as:

```
    0000 0000 0001 0000 0000  Segment, 16 bits shifted 4 bits left
  +      0000 0000 0010 0000  Offet,   16 bits
============================
    0000 0000 0001 0010 0000  Address, 20 bits
```

Pointer C defined as:

```
    0000 0000 0001 0010 0000  Segment, 16 bits shifted 4 bits left
  +      0000 0000 0000 0000  Offet,   16 bits
============================
    0000 0000 0001 0010 0000  Address, 20 bits
```

As defined A, B and C all points to the same memory location however they would fail a comparison test in the following code.

```
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);
}
```

Woud output:

```
0
0
0
```

2.2.4 Extended Memory

The 20 bits address bus of Real Mode limited the addressable RAM to 1MB. But machines of 1992 came equipped with more, typically 2MB and even sometimes 4MB. The workaround at the time was to install special drivers that would open a window beyond the addressable RAM as shown in Figure 2.13.

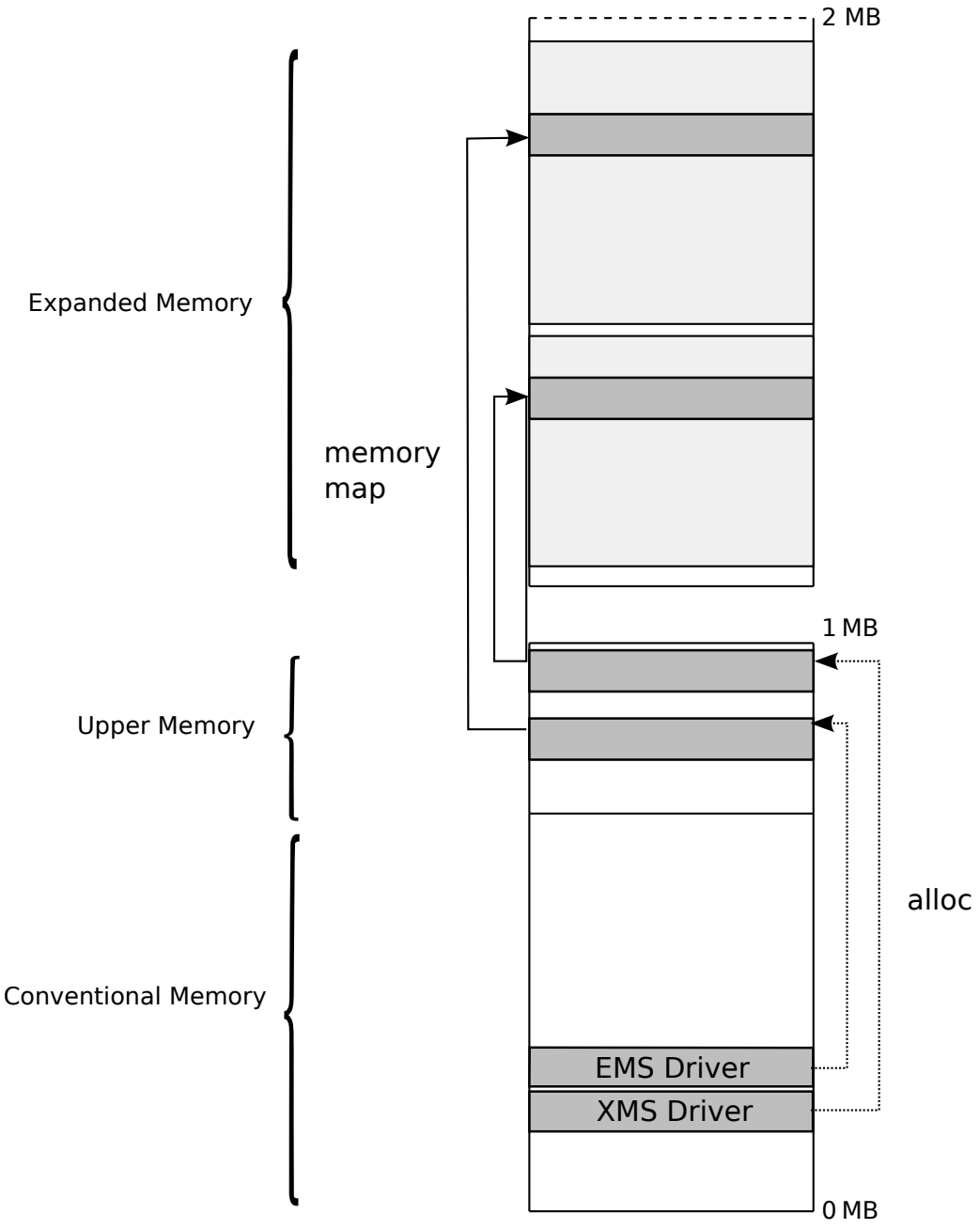


Figure 2.13: Expanded memory layout

With this system, programmer could map part of the upper memory to RAM located beyond the addressable space. Unfortunately once again, nothing was standardized. Users could load different drivers providing different memory mapping systems:

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

Or they could decide to load no drivers at all at startup and not use any of the RAM beyond 1MB.

Trivia : This 640KB barrier was a big issue. Many customer could not understand why even though they had many MB installed on their machine, some game would refuse to start up claiming "Not enough memory". id Software had to publish an explanation (Appendix "The 640KB Barrier") along with the game to make it clear that it was not their fault.

Trivia : As of 2014, thirty five years after the introduction of the 8086, most PC in the world still start in Real Mode. A bootloader switch them to protected mode, load the kernel and then real startup can begin. Mac computers don't have this problem.

TODO: Include reference to XMS and EMS programming reference. [6]

TODO: Include starting screen of Wolf3D showing available memory.

2.3 Video

PC were connected to CRT monitors: Big, heavy, small diagonal, cathodic ray based with curved surface screens. Most monitors had a 13" diagonal with a 4:3 aspect ratio. Figure 4.28 shows a comparison between a 13" CRT from 1992 and a 28" LCD display from 2014.

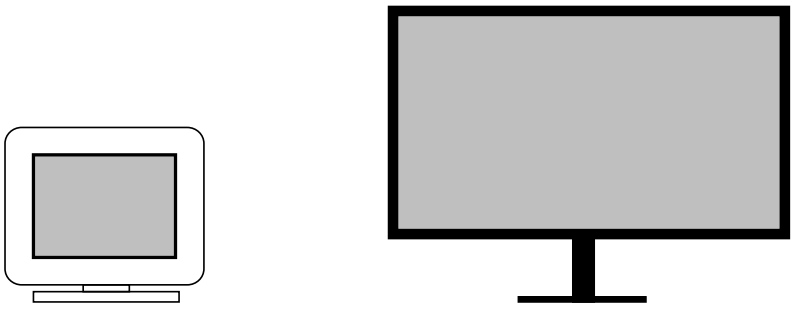


Figure 2.14: CRT (left) vs LCD (right)

Trivia : How big and heavy could a CRT be ? The InterView 28hd96 by Integraph weighted 45kg (99.5lb). For comparison, a modern DELL LCD 27" weights 7.8kg (17lb).

The main issue in this system is that CRT are analogic system while computers are digital. An interface system is needed between the two and it came as a serie of chipsets called "Adapters".

2.3.1 History

The first adapter of the family was released in 1981. The Monochrome Display Adapter (`MDA`) was monochrome with low resolutions but it had the merit to be on every PC sold. Many

[6] eXtended Memory Specification (XMS) July 19, 1988.

other systems followed over the years, always preserving backward compatibility like it was the tradition at the time:

Name	Year Released
MDA (**M**onochrome **D**isplay **A**dapater)	1981
CGA (**C**olor **G**raphics **A**dapter)	1981
EGA (**E**nhanced **G**raphics **A**dapter)	1985
VGA (**V**ideo **G**raphics **A**rray)	1987

Figure 2.15: Video interface history.

Each iteration added new features and in 1993 the ubiquitous graphic system was the VGA[7]. The universality of that system was a two edged sword: One one side developers had to program for one graphic system only. But on the other side, there was no escape to the shortcoming of the Adapter (and people did not purchase separate GPU back then).

2.3.2 VGA Architecture

The VGA can be summarized as two major systems :

- The Graphic Controller and Sequence Controller controlled how the VGA RAM was accessed (interface CPU-VRAM).
- The framebuffer (VGA RAM) made not of one memory bank of 256KB but four memory banks of 64KB.
- The CRTC Controller and the DAC (Digital To Analog Converter) took care of converting the Palette indexed framebuffer to RGB and then to analogic signal (interface VRAM-CRT).

The most surprising part of the architecture is obviously the framebuffer. The first reaction of any engineer would be to ask themselves: Why four banks instead of one clean big one ?

The first part of the explanation comes from backward compatibility: The version before VGA, the EGA had only 64KB of RAM. It was very easy to design a backward compatible system that used only one bank of 64KB.

But the real reason was physical limitations: A CRT running at 60Hz and displaying 640x480 needed a pixel every 1/(640*480*60)th of a second . That means one pixel each 54ns. The problem was that RAM access time was 200ns: Not nearly fast enough[8] to refresh the screen at 60hz. If latency could not be reduced, the throughput could still be improved by accessing

[7] Video Graphic Adapter

[8] Computer Graphic: Principles and Practice 2nd Edition, page 168

four banks at the time. The memory banks read in parallel had an amortized latency of 200/4 = 50ns/pixel which was fast enough.

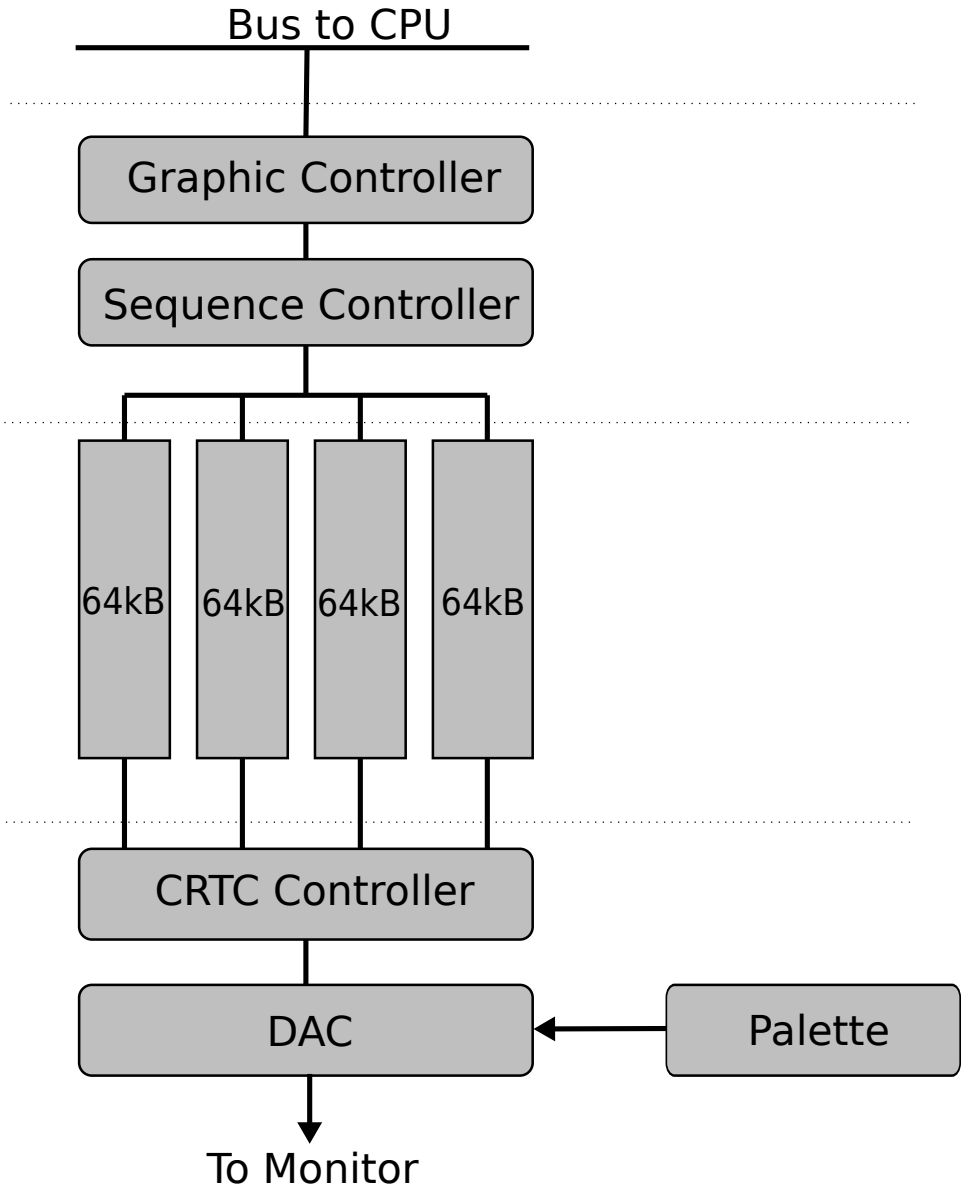


Figure 2.16: Video Graphic Array Architecture.

2.3.3 VGA Planar madness

Four memory banks granted enough throughput to reach high resolutions. The price was complexity of programming aknowledged 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**

My first drawing of the VGA was actually deceiving. It was a hell of a system to work with. The diagram from IBM VGA Reference illustrate this pretty well:

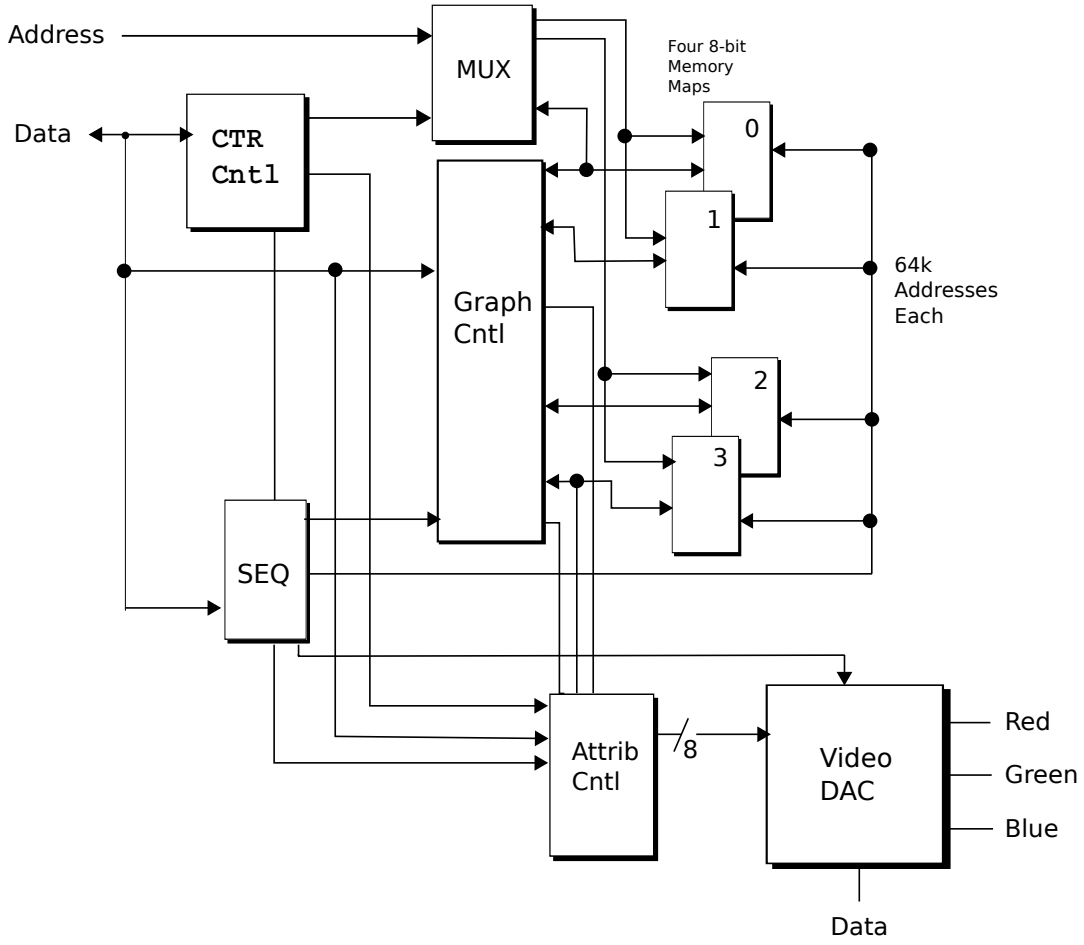


Figure 2.17: IBM's VGA Documentation

How memory banks were populated vs how they were displayed:

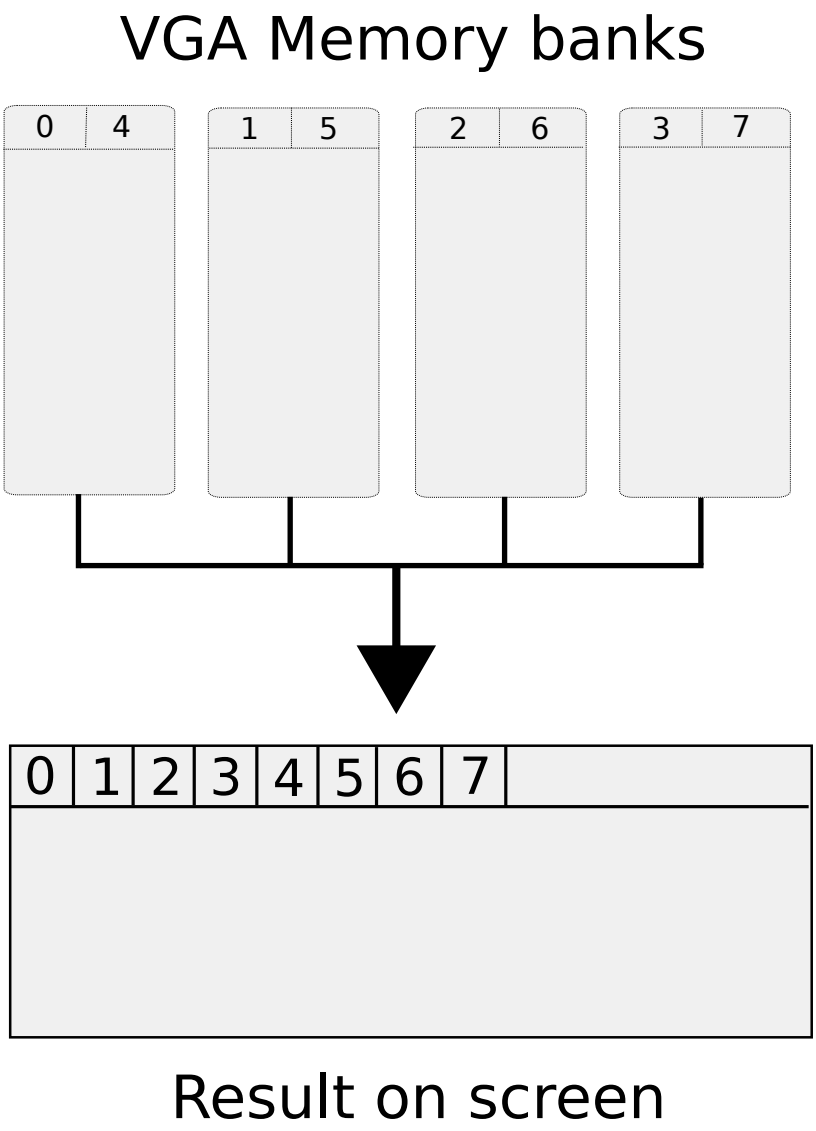


Figure 2.18: Video Graphic Array Architecture.

To configure this mess of planes, registers and controllers meant setting up more than 300 poorly documented registers to interact together. Needless to say few programmers got to dive in the internal of this thing. Luckily IBM BIOS had a few preset configuration (called "mode") with an associated resolution and color numbers.

2.3.4 VGA Modes

The BIOS could be called to have the VGA configured with preset registers.

Mode	Type	Format	Colors	Segment
0	text	40x25	16 gradient (monochrome)	b800h
1	text	40x25	16	b800h
2	text	80x25	16 gradient (monochrome)	b800h
3	text	80x25	16	b800h
4	CGA Graphics	320x200	4	b800h
5	CGA Graphics	320x200	4 gradient (monochrome)	b800h
6	CGA Graphics	640x200	2	b800h
7	MDA text	9x14	3 gradient (monochrome	b000h
0Dh	EGA graphic	320x200	16	A000h
0Eh	EGA graphic	640x200	16	A000h
0Fh	EGA graphic	640x350	3	A000h
10h	EGA graphic	640x350	16	A000h
11h	VGA graphic	640x480	2	A000h
12h	VGA graphic	640x480	16	A000h
13h	VGA graphic	320x200	256	A000h

Figure 2.19: VGA Modes available from BIOS.

Programmers referenced VGA modes by their ID. It was a common thing to see tutorial about Mode 12h or Mode 13 which were the two most appealing mode for game programming.

2.3.5 VGA Programming: Mode 13h

Using the VGA as per the manual is easy since two instructions are enough to ask the BIOS to setup the environment with a mode. Here is an example that setup the a VGA to operate at 320x200 with 256 indexed colors: Mode 13h.

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

Two instructions and that is it. The `int 10` is a software interrupt call to the BIOS routine in charge of Graphic setup. It looks up the `ax` register to setup all 300 VGA register with the corresponding mode.

A convenient thing with those was how the four memory banks were hidden to the programmer. By using 2 bits from the RAM, the VGA automatically determined in which bank to store a value. The side effect was to waste 75% of the RAM. Memory Mapped area:
DRAWING
The programmer has access to a linear framebuffer of 64KB mapped at `A0000h`.

DRAWING

To clear the screen to black for example, the following code was enough:

```
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] = 0;
}
```

The mode 13h may look good but it is in fact terrible for game and animation. The intend was clearly to display static images:

- All the RAM is used and there is no way to have a double buffer.
- Since the resolution is 320x200, the aspect ration does not feet the monitor. As a result the image is streched when transfered from the framebuffer to the CRT.

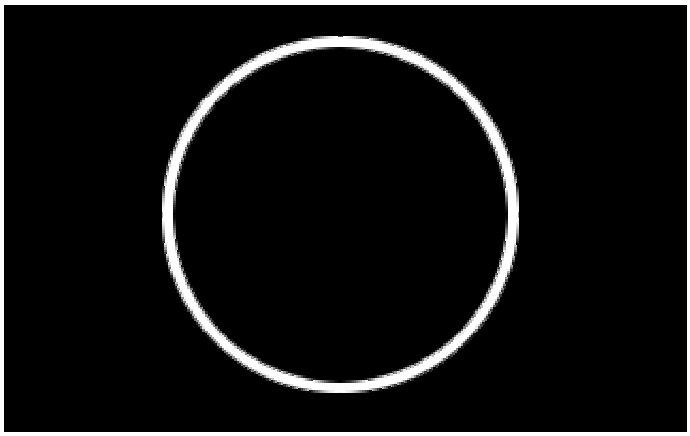


Figure 2.20: Drawing a circle in the framebuffer

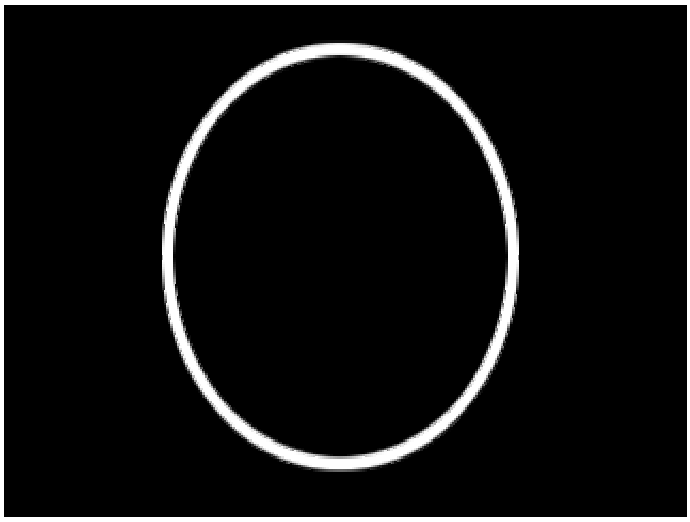


Figure 2.21: How the framebuffer looked once streshed on the CRT monitor :(!

- VGA MAPPING I/O WASTED SPACE.

2.3.6 What VGA meant for games

Not sprite engine meant a full framebuffer was had to be populated for any type of animation. Off all parts of the system, the VGA was in appearance the weakest of them all. Single-handedly preventing any type of smooth, tearless animation.

2.4 Audio

Off all the parts in a PC, the audio system was the best. The card manufacturer had started to see a market and player actually had the opportunity to buy decent extension "sound card" from Ad Lib or SoundBlaster.

2.4.1 Speaker

PC came out of the box with a binary beeper. It was capable to produce two frequencies with surprisingly good results such as in Monkey Island game.[9]

2.4.2 Ad Lib

Ad Lib was a Canadian manufacturer of sound cards. As early as XXXX they identified the limitations of the PC Speakers and started to manufactur a card to connect othe ISA bus. Allowd MIDI.

[9] https://www.youtube.com/watch?v=a324ykKV-7Y

Note: Canadian company and especially from Quebec were very innovative in the early 90s. Ad Lib manufactured Sound Card, Matrox made a killing with Millenium Graphic Card, Watcom compiler, ATI.

Music (midi) Not about canada: adlib, watcom and matrox millenium

2.4.3 Sound Blaster

The Sound Blaster 1.0 (code named "Killer Kard"),[1] CT1320A, was released in 1989 Music (midi+digitized sound)] But it also added two key features absent from the Adlib: a PCM audio channel, and a game port.

2.4.4 Sound Blaster Pro

Model CT1330, announced in May 1991, was the first significant redesign of the card's core features, and complied with the Microsoft MPC standard.[6]. The Sound Blaster Pro supported faster digital sampling rates (up to 22.05 kHz stereo or 44.1 kHz mono), added a "mixer" to provide a crude master volume control (independent of the volume of sound sources feeding the mixer), and a crude high pass or low pass filter. The Sound Blaster Pro used a pair of YM3812 chips to provide stereo music-synthesis (one for each channel).

2.5 Inputs

Even though it is anecdotic I still wanted to write a part about the connectors found on a PC at the time. A time before the ubiquitous USB. In short, inputs were a mess.

The parallel port (DB-25) was on every computer and usually used to connect matrix printers (loud thing that printed with aiguilles).

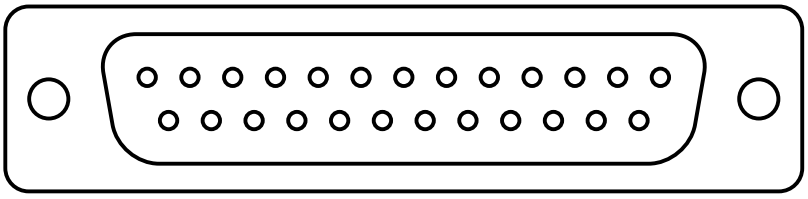


Figure 2.22: Parallele Port

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

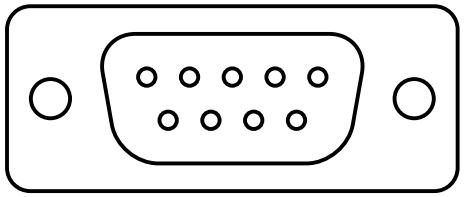


Figure 2.23: Serial Port

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

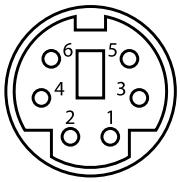


Figure 2.24: PS/2 Port

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

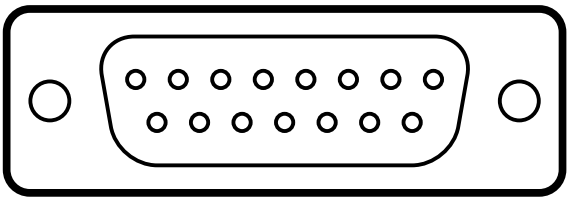


Figure 2.25: Game Port

Note that without a sound card you were unable to connect a joystick !

2.6 Summary

Chapter 3

Team

id Software was funded during February 1991 by four people:

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

Figure 3.1: id Software founding members.

Wolfenstein 3D was id first title but the team had already shipped no less than 13 games will working for their previous employer SoftDisk:

- Dangerous Dave (1988)[1]
- Commander Keen
  - Episode 1: Marooned on Mars (1990)
  - Episode 2: The Earth Explodes (1991)
  - Episode 3: Keen Must Die (1991)
  - Keen Dreams (1991)
  - Episode 4: Secret of the Oracle (1991)
  - Episode 5: The Armageddon Machine (1991)
  - Episode 6: Aliens Ate My Baby Sitter (1991)
- Dangerous Dave in the Haunted Mansion (1991)

[1] Dangerous Dave is a solo project of John Romero predating Id's formation, but Id Software produced its first sequel and it is sometimes regarded as an early Id Software title. Later Dangerous Dave sequels were not made by Id, nor were later Catacomb titles.

- Rescue Rover (1991)
- Rescue Rover 2 (1991)
- Shadow Knights (1991)
- Hovertank 3D (1991)
- Catacomb 3D: A New Dimension (1991)

Considering the magnitude and ambitions of the title, four more people were added to the team for a total of eight.

Name	Age	Occupation
Jay Wilbur	??	Business
Kevin Cloud	27	Computer Artist
Robert Prince	??	Composer
Jason Blochowiak	??	Programming

Figure 3.2: id Software new hires.

> “ Jason was part of Id at the start, but we parted ways during Wolf development.
>
> **John Carmack - Programmer** ”

Figure 3.3: id software team circa 1993 as show from Wolfenstein 3D.

Figure 3.4: They were in fact wearing pants.

Every member was working with an high end 386 DX. As for combining game, tools and assets:

> 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**

3.1 Programming

Development was done with Borland C++ 3.1:

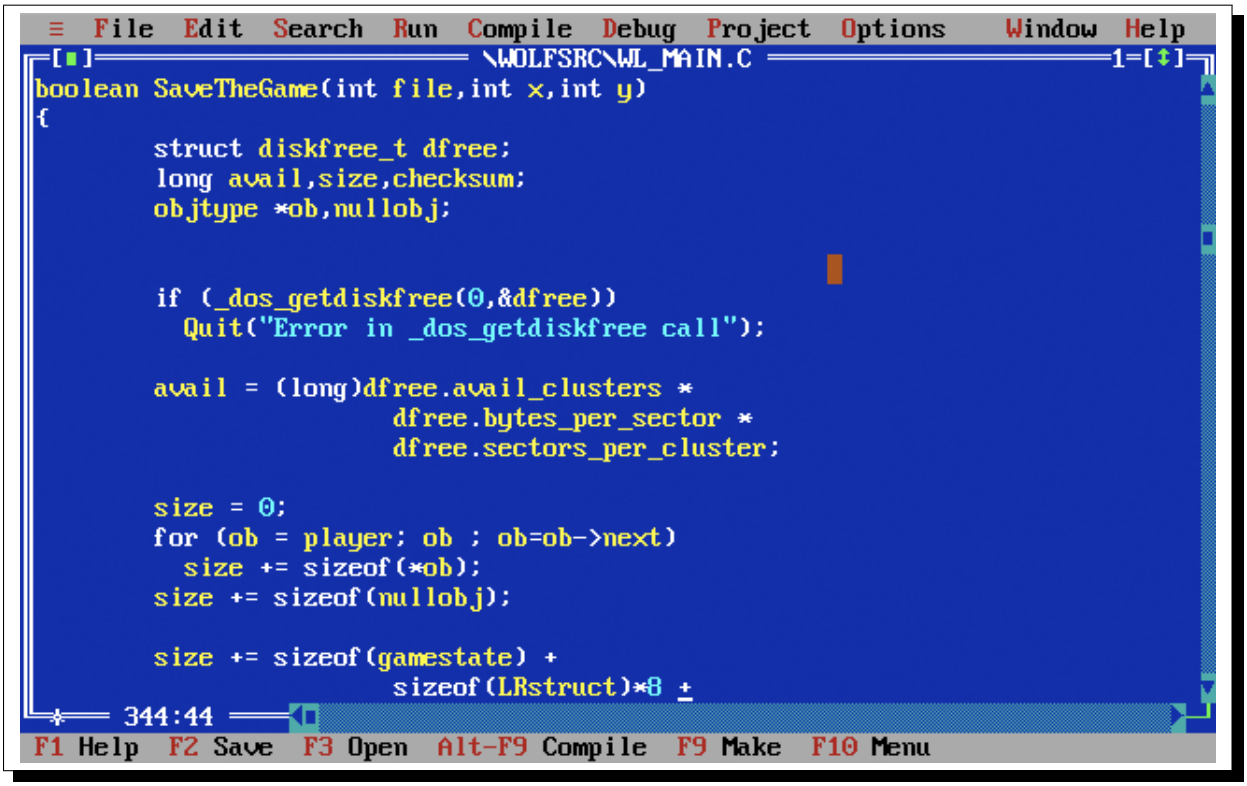


Figure 3.5: Dual-monitor: Screen 2.

The editor used VGA mode 3: 80 characters wide and 25 characters tall.
John Carmack took care of the runtime code. John Romero programmed many of the tools (editor, packers, ...). Jason Blochowiak was contracted to write important subsystems of the game (Input manager, Page Manager, Sound Manager, User Manager).

The compiler was Borland C++ 3.1. To compensate for the tiny CRT, some of the developers used two screens (a very 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 mode (See figure 2.19 on page 30): they don't have the same starting segment. This allows to add a CGA video card to the PC to run in dual-monitor mode: One in monochrome text and the other one in regular VGA. This allows a developer to debug the game engine in real-time.

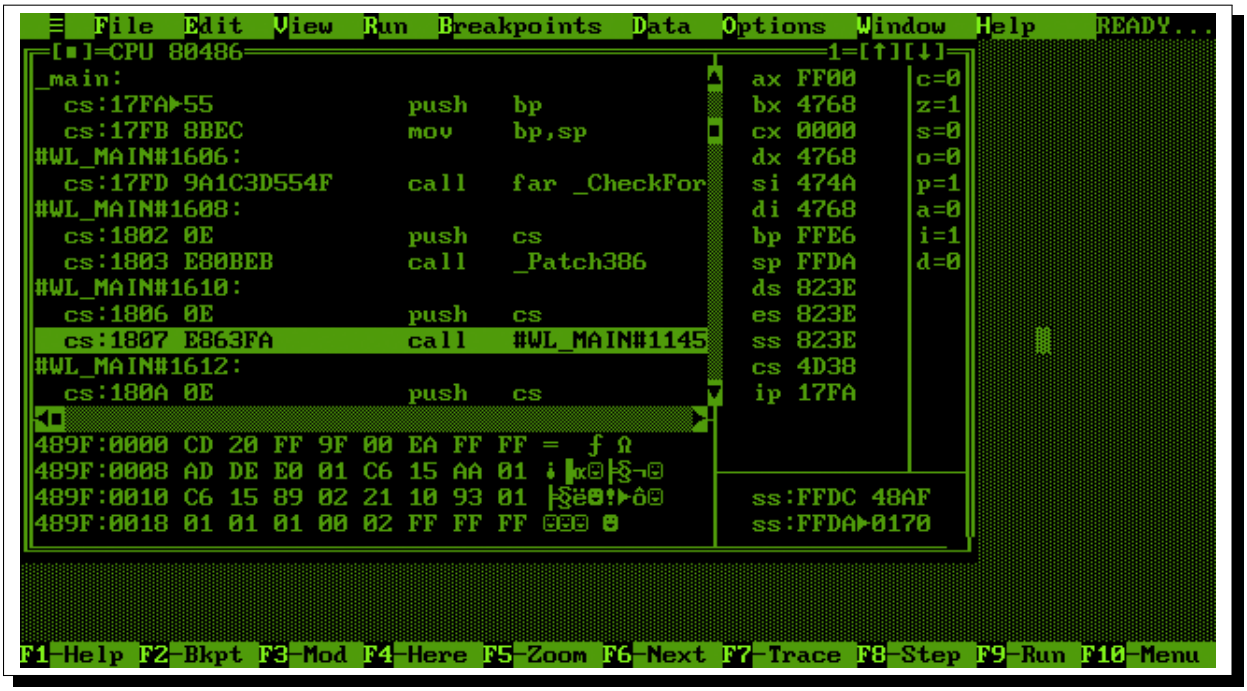


Figure 3.6: Dual-monitor: Screen 1.

Figure 3.7: Dual-monitor: Screen 2.

3.2 Graphics assets

The graphic assets are divided as follow:

- 2D Menus items.
- 2D Action phase items.
- Wall textures.
- Entities animations.

Those were the work of Adrian Carmack and Kevin Cloud. All of the work was done with Deluxe Paint on high-end 386 DX computers. The most difficult task was not to draw everything but to make the key decision of what colors would go in the palette.

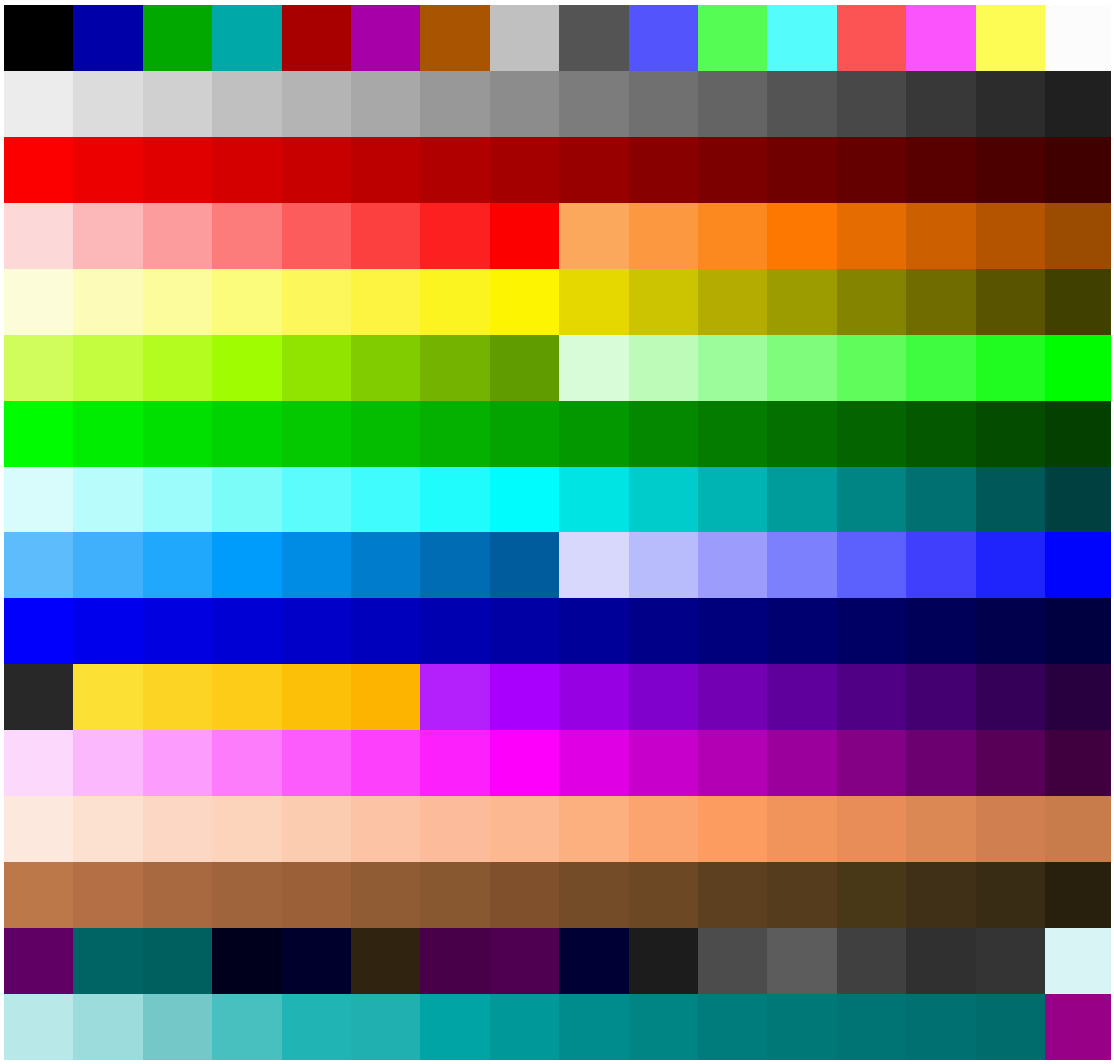


Figure 3.8: The 3D sequence palette.

Since all textures for the wall and sprites would have to use it.
All assets were hand drawn:

> We didn't have any scanning tools at the time.
>
> **John Carmack - Programmer**

I was unable to get in touch with Adrian Carmack or Kevin Cloud but I would have loved to know if they have some sort of stylus or if he was just very good with a mouse.

3.3 Asset workflow

Once all the Deluxe Paint images were ready, a tool, IGRAB-ED, packed all the ILBMs files and generated a C header file:
TODO: entry example. All asset were given a name (e.g:). The program could reference an asset directly by using its macro.

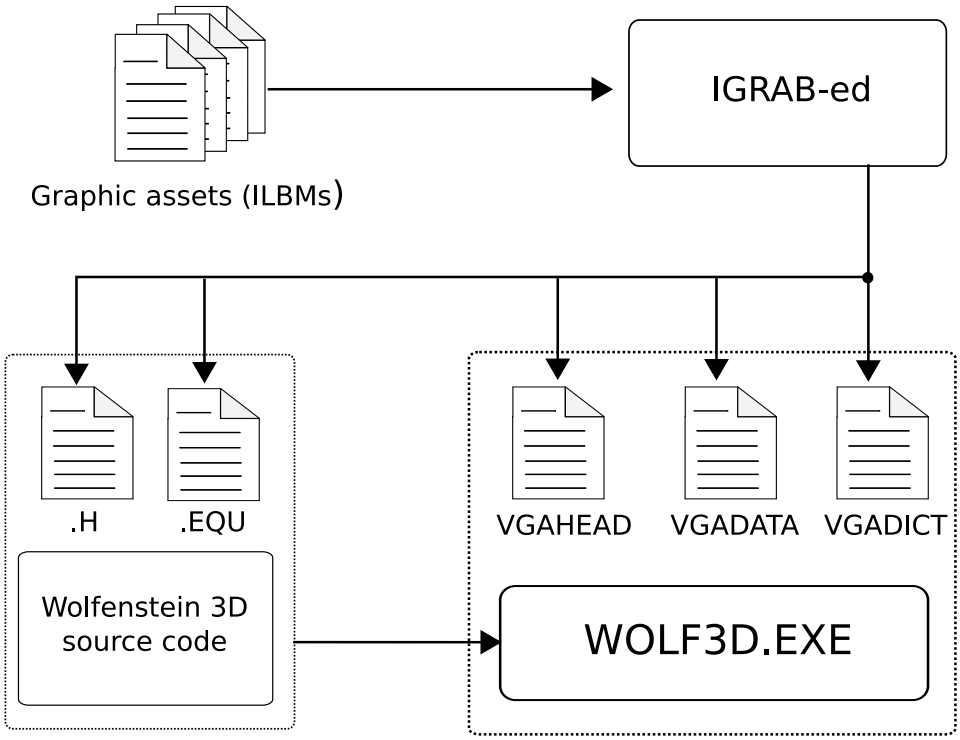


Figure 3.9: Assets creation

Trivia : This system let to issues when the source code was released: The header provided did not match the asset file from the shareware or early version of Wolfenstein 3D: The header released were from Spears of Destiny. You can see the kind of graphic mess this let to in the

appendix "Let's compile like it is 1994".

Insight into asset production show Tom Hall sketches before they were created by Adrian Carmack.

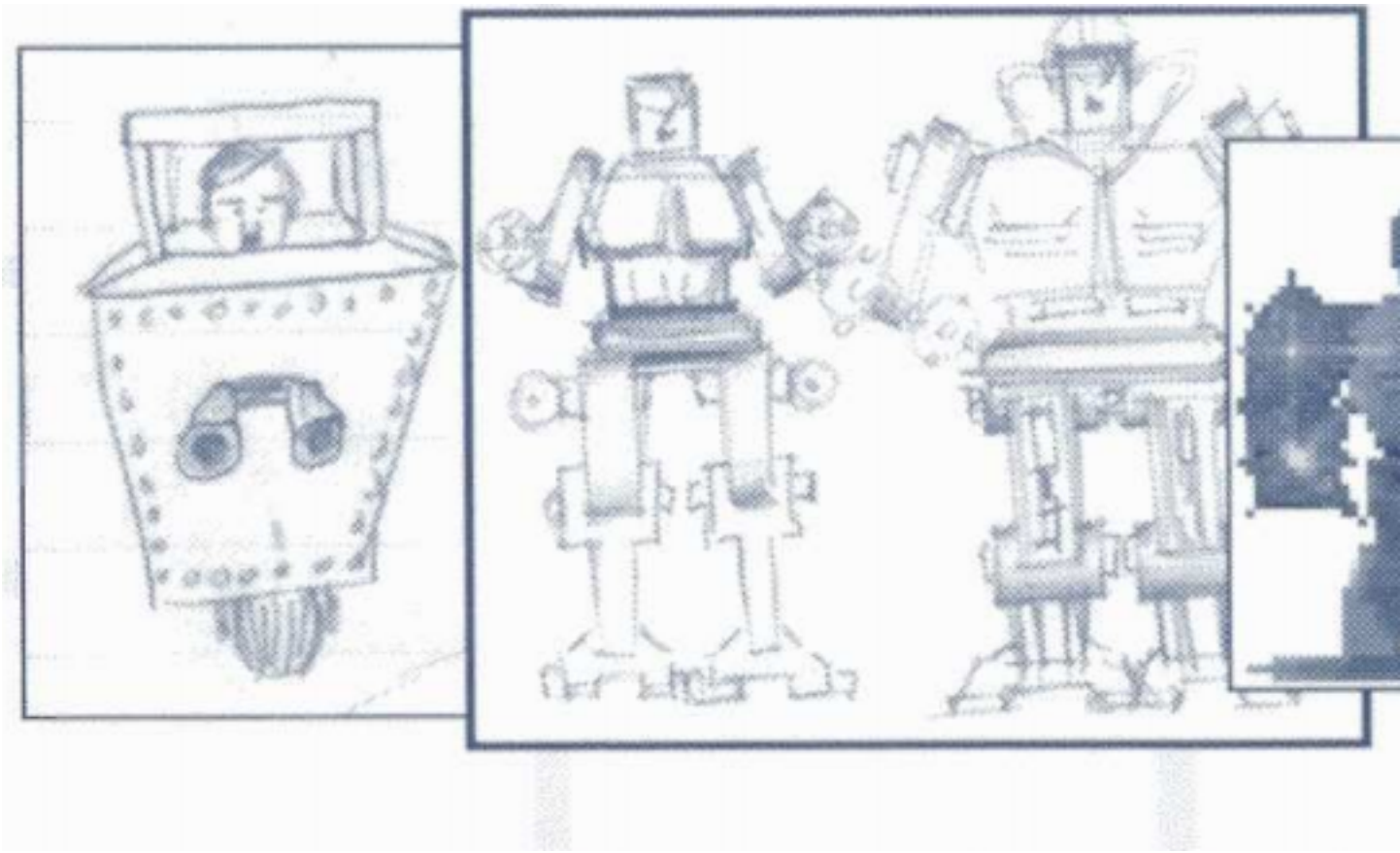


Figure 3.10: 3D Rendere Phase 1: Backed lights

Figure 3.11: 3D Rendere Phase 1: Backed lights

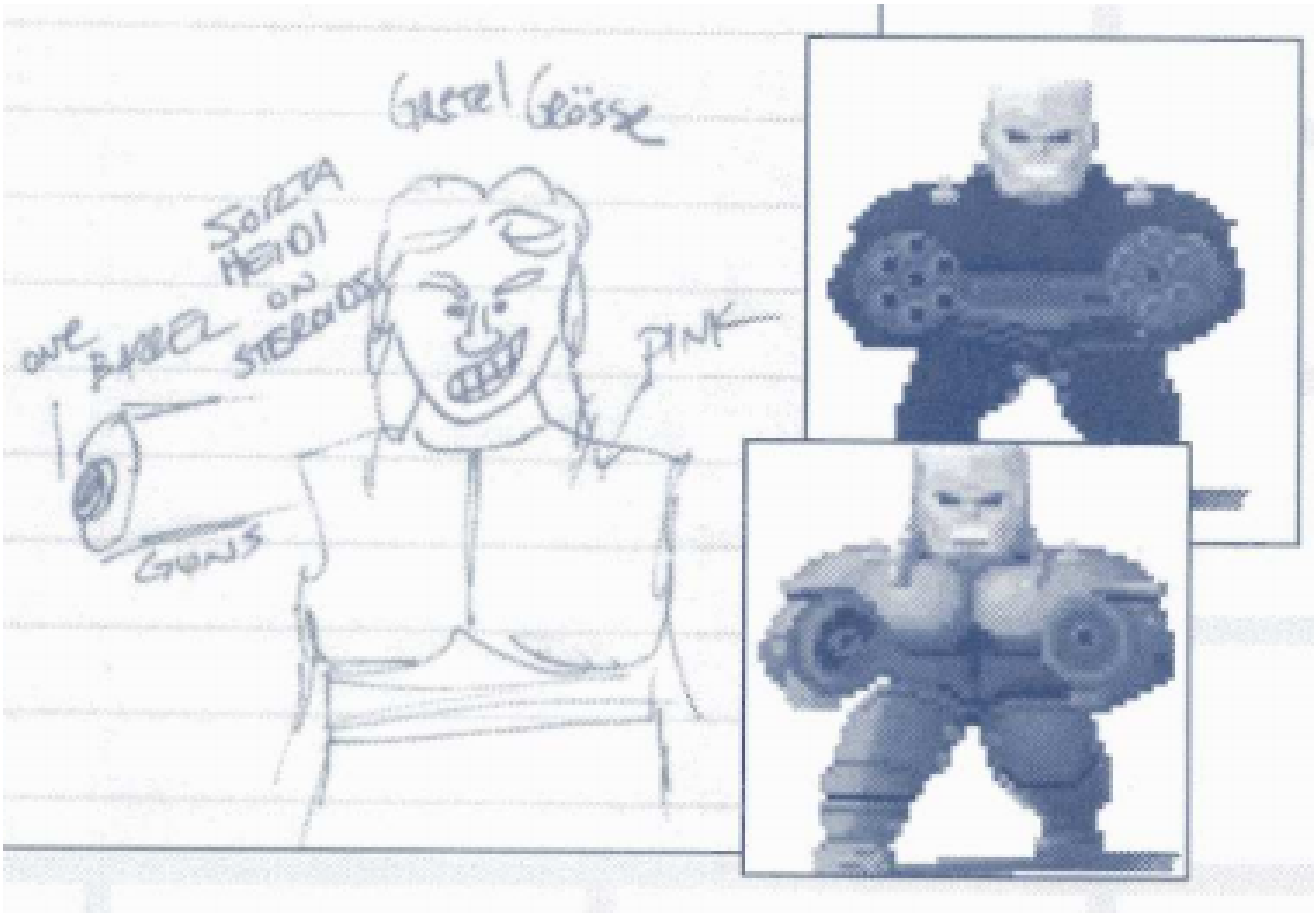


Figure 3.12: 3D Rendere Phase 1: Backed lights

Figure 3.13: 3D Rendere Phase 1: Backed lights

Figure 3.14: 3D Rendere Phase 1: Backed lights

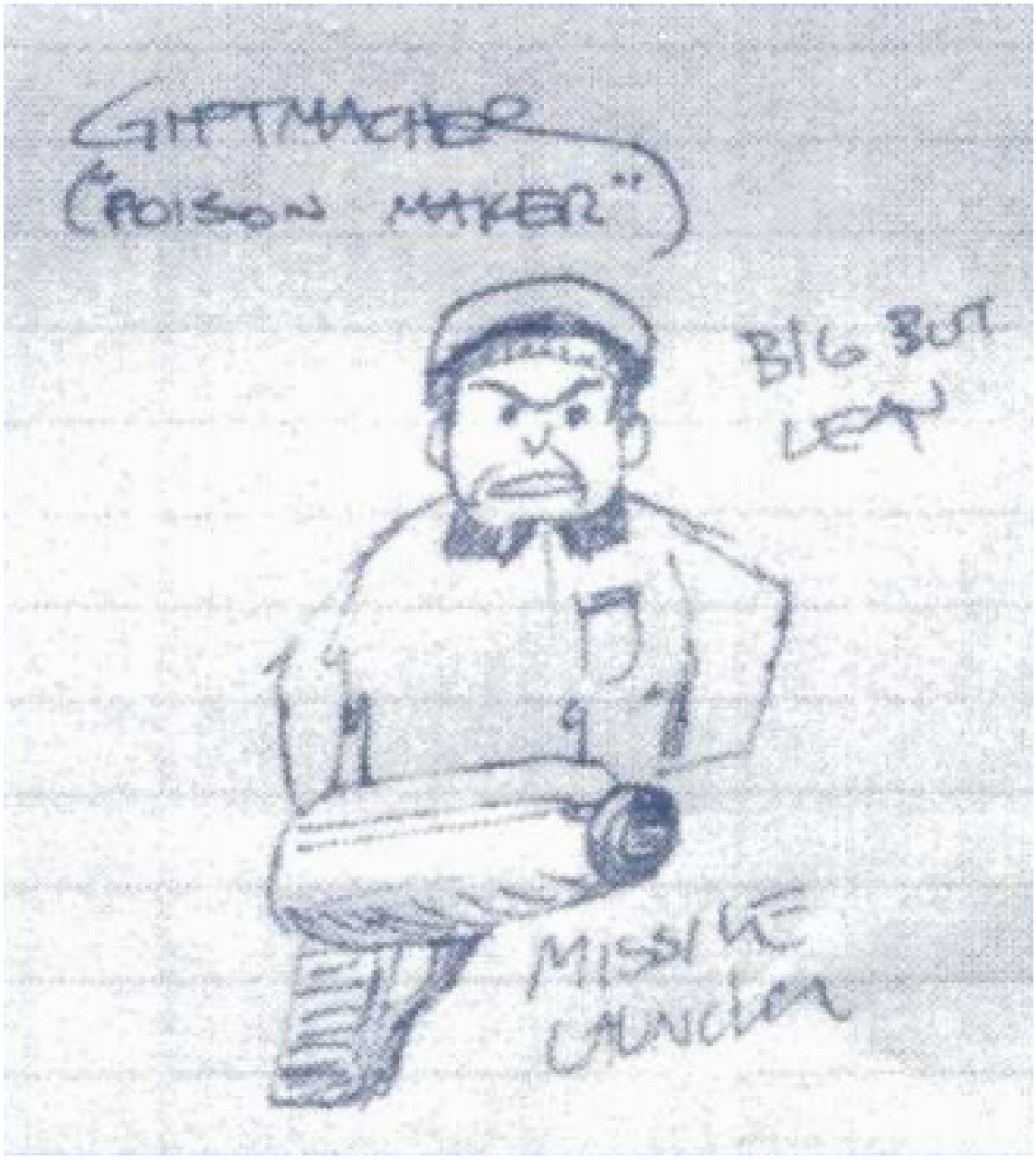


Figure 3.15: 3D Rendere Phase 1: Backed lights

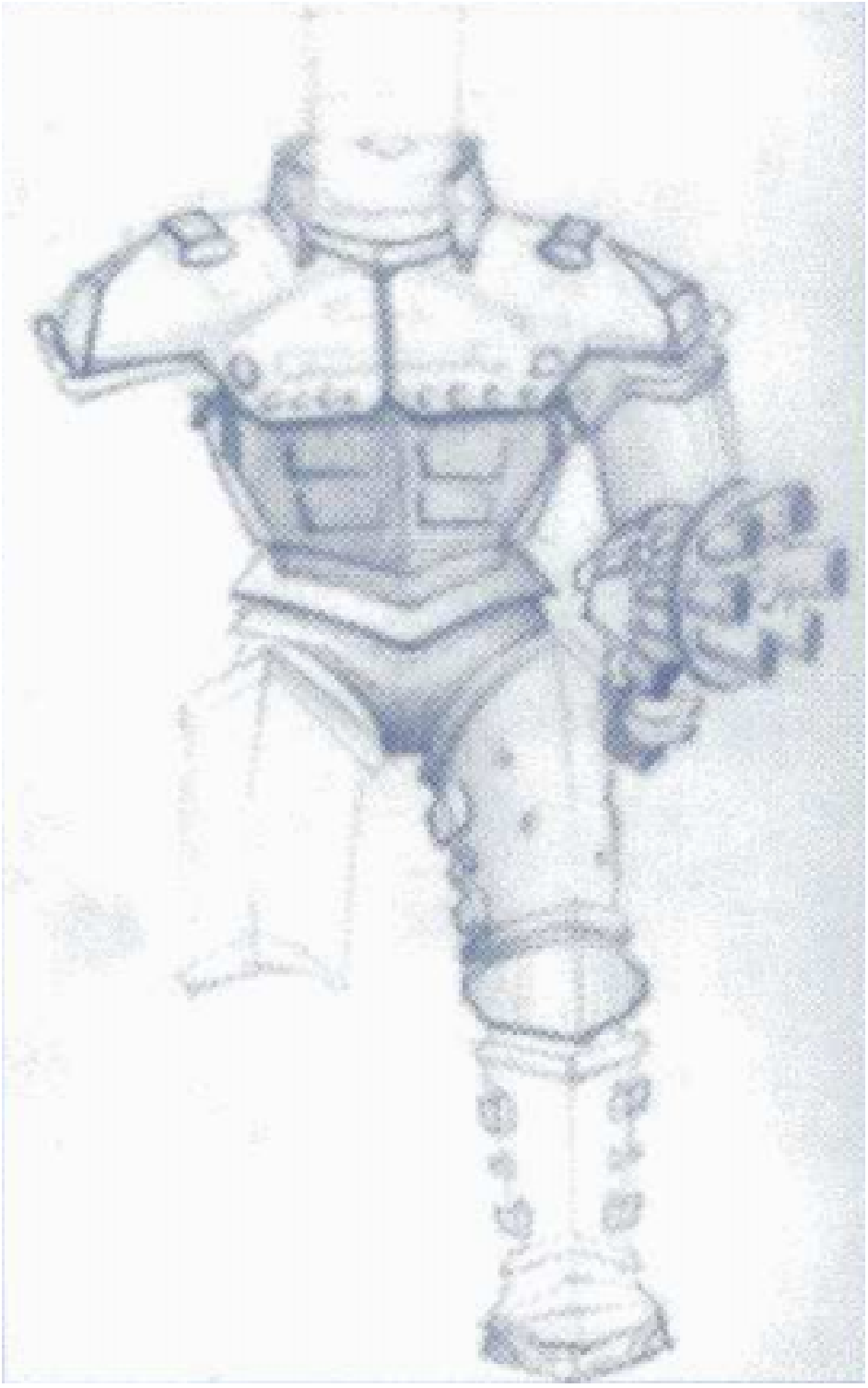


Figure 3.16: 3D Rendere Phase 1: Backed lights

2

[2]The official hint manual for Wolfenstein 3D

3.4 Maps

trivia: speed run.
trivia: the ceiling color is not part of the asset. Instead it is hard-coded in the game engine. TODO: Paste code and file here.
Maps were created using the in-house editor called TED which is short for Tile EDitor. That tool was used for other games, the 2D scroller Commander Keen serie was entirely designed using TED.

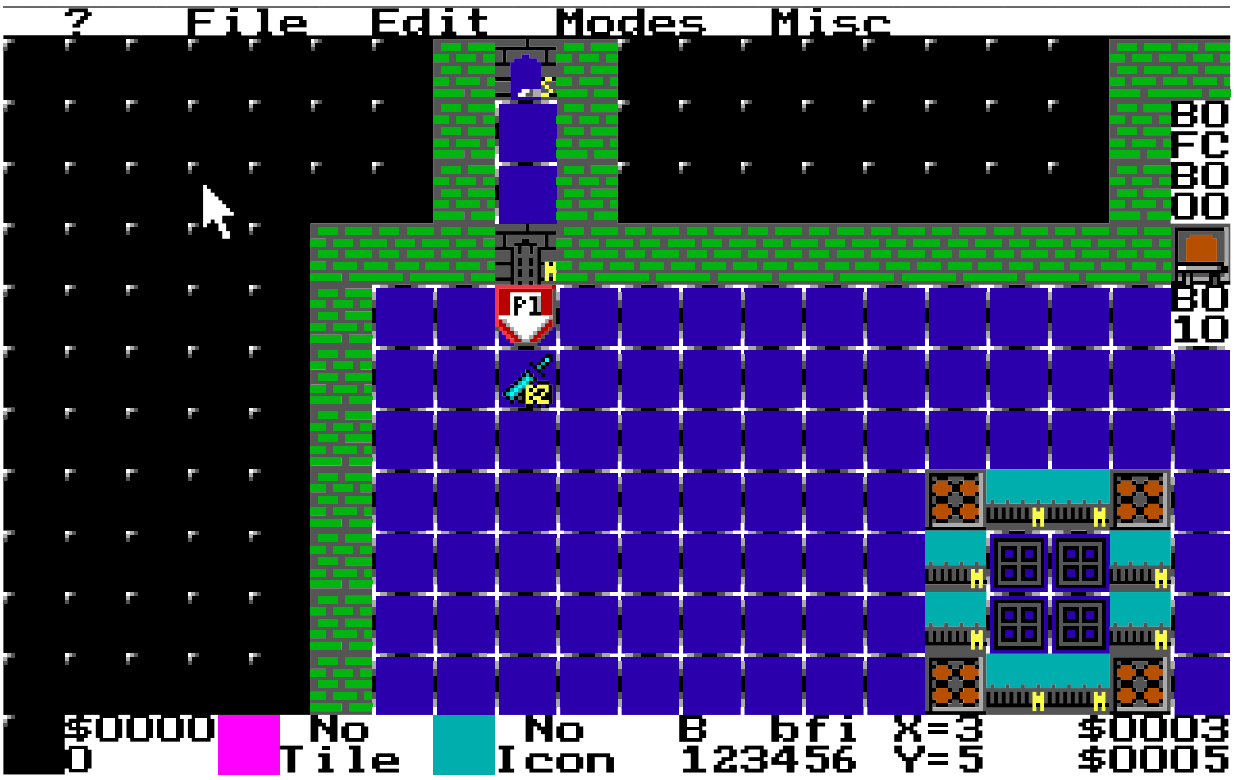


Figure 3.17: Tile EDitor screen

Everybody worked a little bit on the map but those were mostly the work of John Romero and Tom Hall. John Romero updated the editor TileEDitor (TED). Maps were plans based, 64x64.

Trivia : The source code of TED was released several years later. Inspecting the source was a mysterious `_TOM.PIC`. Converted to PNG it looks as follow:

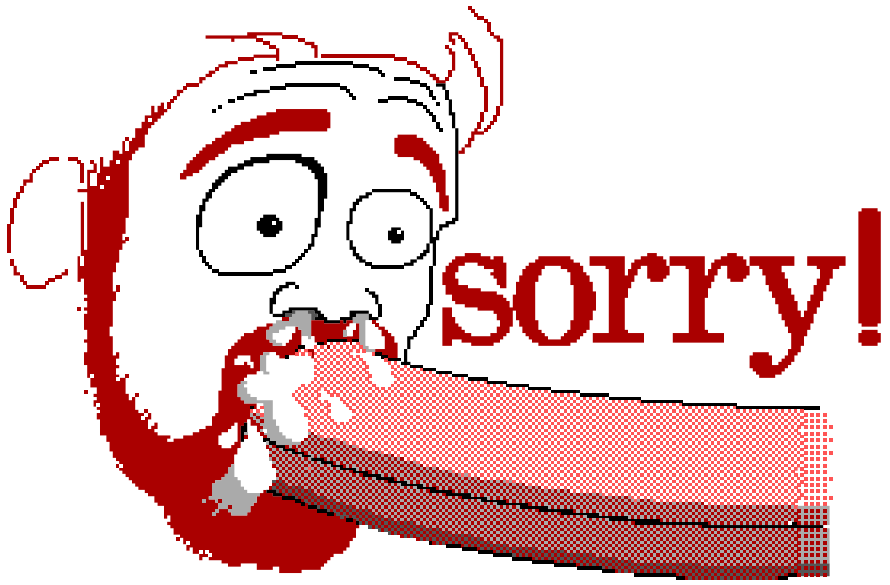


Figure 3.18: Not so politically correct caricature.

The explanation was provided later by John Romero:

> "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 getting Adrian's dick blasted into his face with Adrian 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**

3.5 Business

I was unable to get in touch with Jay Wilbur. Little is known...for now :) !

3.6 Sounds

Despite multiple emails, I was unable to get in touch with Robert Prince. Little is known...for now :) !

Chapter 4

Software

4.1 Source Code

The game engine was released on July 21, 1995. 20 years later it is still there on the company ftp[1]:

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

Trivia : Before the times of nmap and WireShark, ftp was commonly used to transfer files: It was simple and naively transmitted username and password in clear. Needless to say it did not age well...

4.2 First Contact

The file `woldsrc.zip` contains an other self-extracting PKZIP archive. It was a convenience back in the day but it is not practical nowadays. You can deflate it with:

`unzip WOLFSRC.1`.

The source code actually contains more than just `.H` (headers) and `.C` (code) files. Also present are:

- `.EQU`
- `_WL1.H`
- `_WL6.H`
- `_SOD.H`

[1] Transfer File Protocol (along with Finger where J.C.Carmack used to blog) was commonly used in the 90s in the blessed time where you did not have to encrypt everything on the wire.

- `_SDM.H`
- `GOODSTUF.TXT` A letter from a POW playing Wolfenstein 3D.
- `.ASM` Assembly optimized routines. Also contains routines to access VGA and refresh screen.
- `SIGNON.OBJ/`: The startup screen showing the system characteristic (RAM, EMS, XMS, Joystick, SoundCards) was linked in the binary. Because that screen was showed before any sub-system was started. TODO: This should be a trivia.
- `GAMEPAL.OBJ/` Game palette. Hardcoded and linked in the executable for the same reason described previously.
- `README/` How to build. You can find a complete tutorial in the Annexe of this book.
- `RULES.ASI` ???
- `SV.EXE` ???
- Many files resulting in a previous compilation attempt.

A quick stats:

```
wolfsrc\$ cloc-1.64.pl .

      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 game engine is mostly C with a few ASM routines for optimized stuff and I/O.

> 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 @#$% !
>
> **John Carmack - Programmer**

> Estimated completion time ? **John Carmack - Programmer**

4.3 Architecture

The engine is made of sub-systems called Managers:

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

Together they harness the machine to play sound, music and display the two phases of the game:

- 2D Menus
- 3D Action

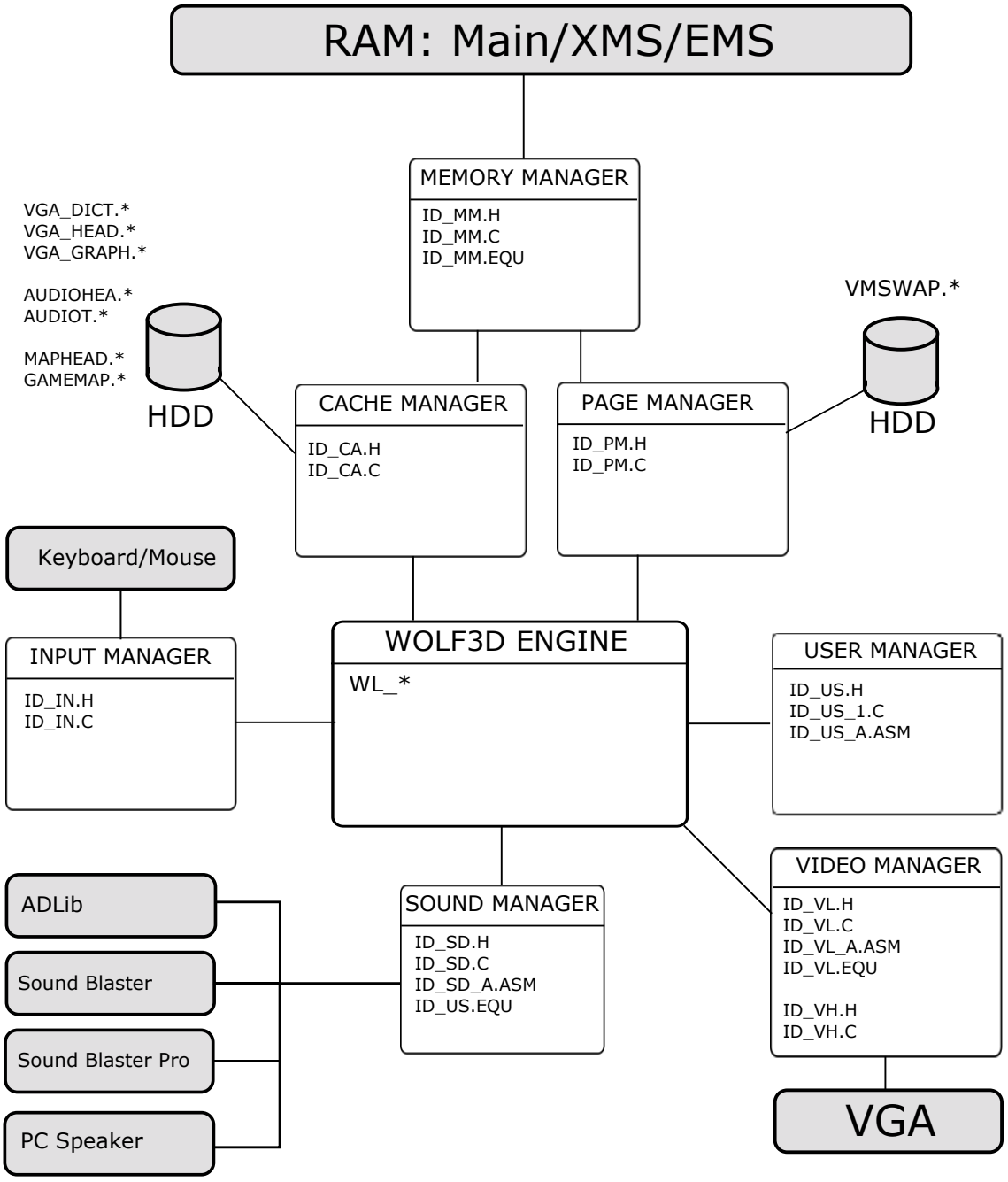


Figure 4.1: Architecture and sub-systems.

4.4 Palette tricks

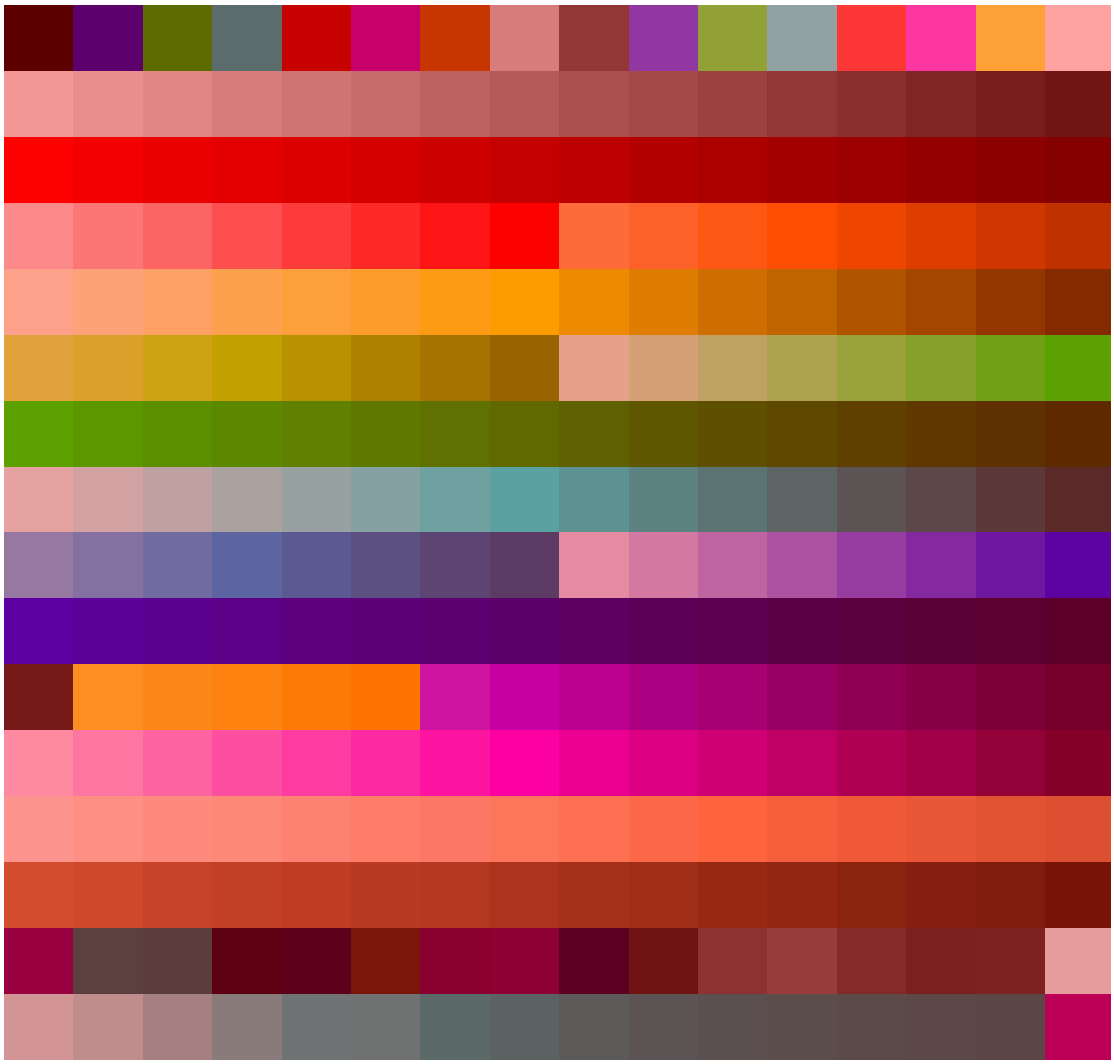


Figure 4.2: The palette RGB colors altered when taking damage.

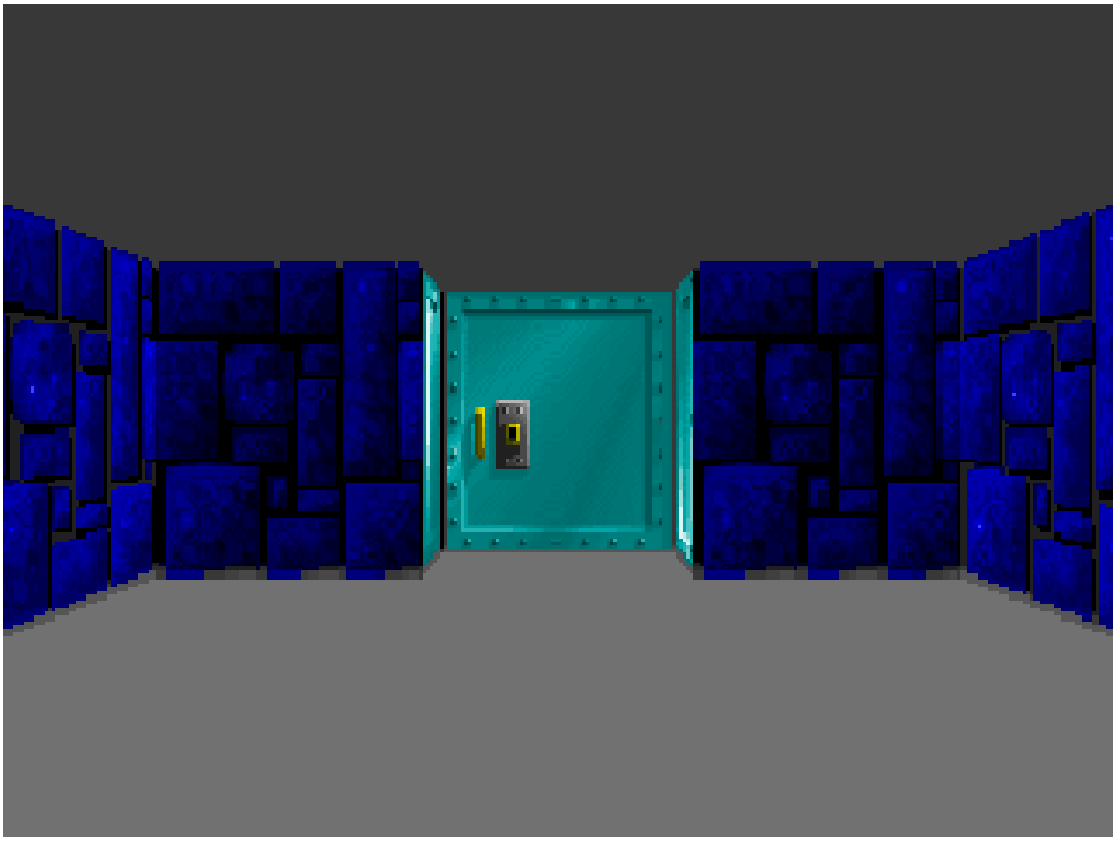


Figure 4.3: Raycasting 1: Screenshot

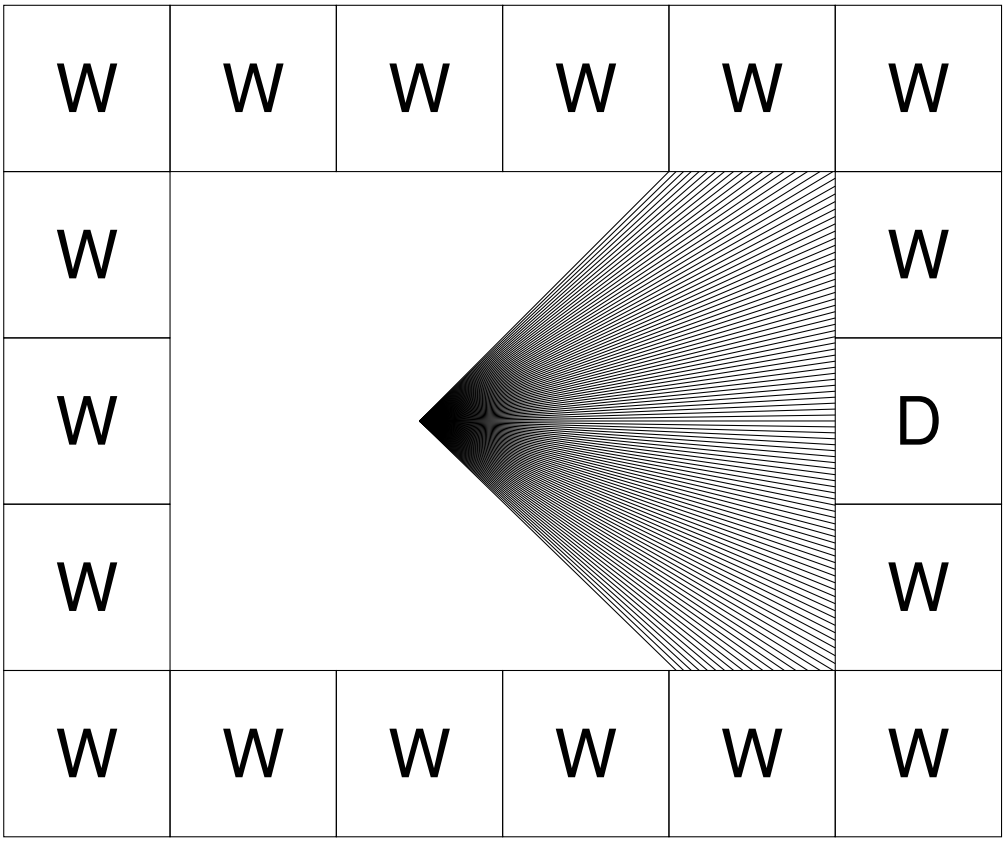


Figure 4.4: Raycasting 1: Drawing

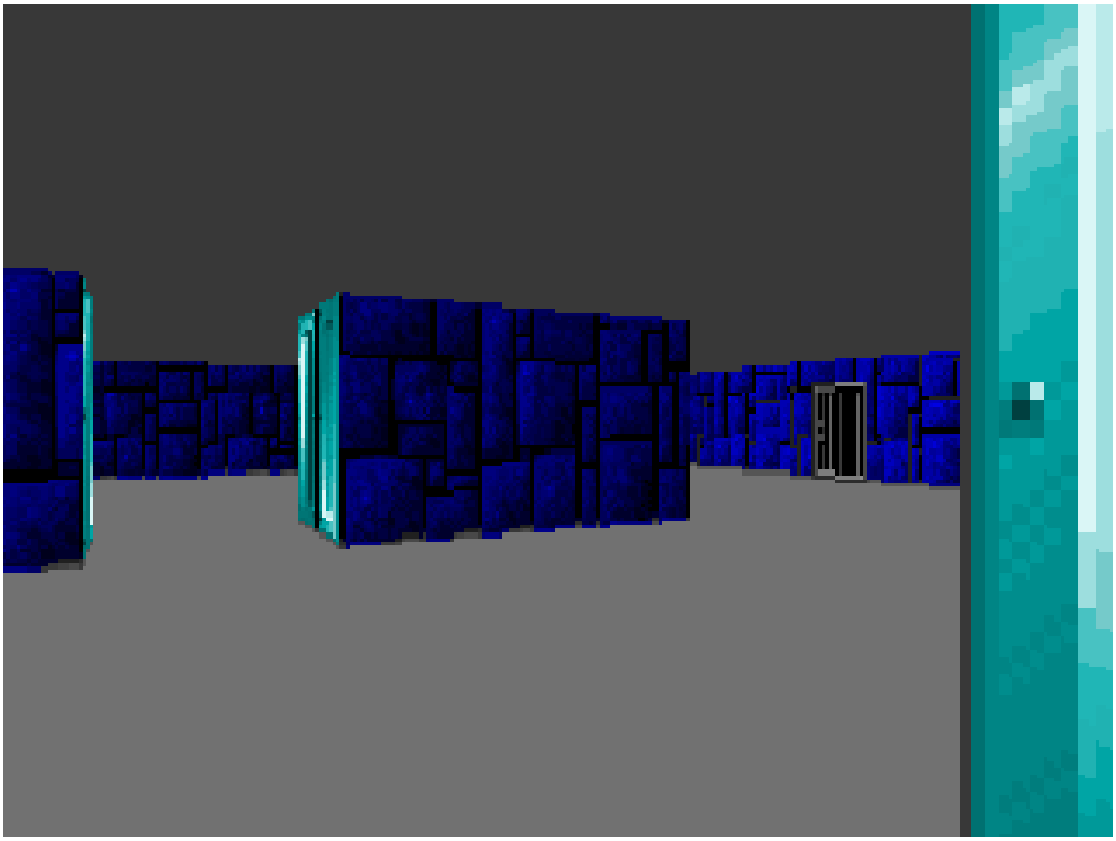


Figure 4.5: Raycasting 2: Screenshot

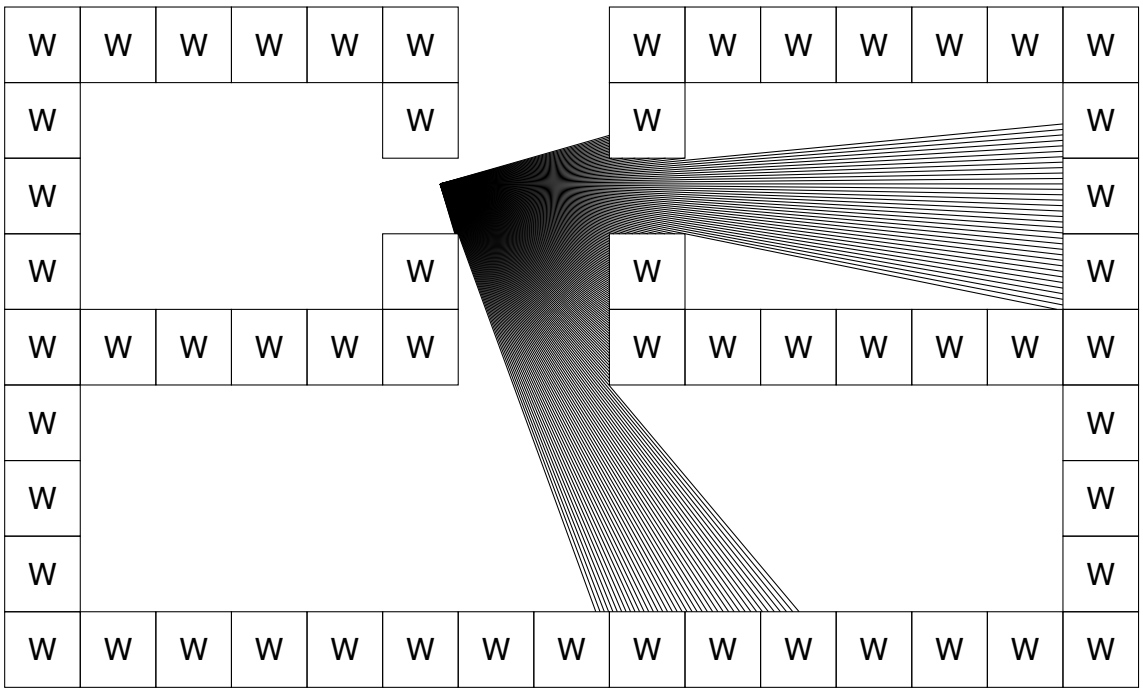


Figure 4.6: The 320 rays cast to render the walls in the previous scene.

Trivia: Actually the engine casted less than 320 rays: Trick used.

4.5 Unrolled Loop

Game runs at 70Hz (twice as doom which ran at 35Hz). Framerate comes from the refresh rate of VGA at 320x240. At 640*480, rate it 60hz.

4.5.1 Memory Manager (MM)

4.5.2 Page Manager (PM)

The Page Manager harness the three types of RAM available: Conventional, XMS and XML. It abstract this complexity behind a Paging system.
Is it where the loading screen shows a progress bar ? "Get Psyched !!!": Fill up the cache ? Trivia: The progress bar was called a "thermometer". The Page Manager provides all asset during 3D rendition. It loads as many "Pages" as possible in Main, XMS and EMS memory during the get Psyched screen. TODO: Add a screenshot. It uses a Least Frequency Used eviction policy if not all asset fit in RAM. VSWAP.WL6 is 1.6MB -> Recommended configuration 2MB RAM..but the game would run with just 640KB.

4.5.3 Video Manager (VW)

4.5.4 Cache Manager (CA)

The Cache Manager takes care of all the assets during Menu Phases: Sprites, Music, Sounds.

4.5.5 Sound Manager (SD)

The Sound Manager abstract interaction with all four sound systems supported: PC Speaker, ADLib (Music only), Sound Blaster (Mono), Sound Blaster Pro (Stereo).

4.5.6 User Manager (US)

4.5.7 Input Manager (IN)

Abstract interaction with keyboard and mouse.

4.6 Introduction Phase

The game starts with the "resource summary" which display the available RAM, Inputs and Audio systems. Then it is followed by the title screen and the the disclaimer screen. Subversive spirit. TODO: Show screenshot.

4.7 Renderer: Mode X and Mode Y

As seen in the Hardware chapter, none of the VGA modes offered could do what a game needed. But game developers found
a way: Mode X [2]. Nobody knows who discovered mode-X but the person who popularized it is Michael Abrash [3]. The idea is to disable the "convenience" mechanism making the 4 banks appear as one. It is called putting the VGA in unchained mode where each bank of memory (called "plane") is written to individually.
With this technique the 64KB of mapped RAM can access the full 256KB of the VGA. This enables a mode offering:

- Resolution of 320x200
- Double buffering
- Storage of sprites in VRAM.

Compared to figure TODO: New Drawing of VGA RAM.
But there is still a problem: Switching the target RAM was very slow. So a routine drawing an horizontal line on the screen:

```
void DrawLine(y, color) {
  for(int x=0; x < 320 ; x++) {
    SelectBrank(x % 4);
    WritePixel(x, y, color);
  }
}
```

Would have been unpractical. But something else can be done if you notice that 320 is a multiple of 4 (320/4=80): If you draw only columns you don't have to change the target plan often.
TOOD: Drawing of how column line up in VGA RAM.

4.8 Menu Phase: 2D Renderer

The menu phase is renderer by the 2D renderer which is tile based. The screen is divided into tiles as follow:
DRAWING
Bla bla bla

[2] X-Mode Frequently Asked Questions - By Zoombapup (Phil) 2-Oct-94
[3] Michael Abrash's Graphic Programming Black Book: Chapter 47, page 877.

4.9 Action Phase: 3D Renderer

Raycasting DDA: "http://lodev.org/cgtutor/raycasting.html"

4.9.1 Map

Show map.
Show 64x64.
Show angle orientation.

Call apogee:

Figure 4.7: Raycasting 2: Call apogee texture

Figure 4.8: Raycasting 2: Call Apogee as seen in Episode 2, Floor 8

> "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. Also note that if you found the Aardwolf sign in the game (without cheating), there's a VERY strong chance that you're stuck in there. The only way out may be to restart, or load a saved game from before you went into that maze.
>
> **Joe Siegler - Past Pioneers of the Shareware Revolution**

4.9.2 Fixed point

4.9.3 Raytracing: DDA Algorithm

Digital Differential Analysis
Some ray are not traced ? How does that work ?
Precalculated sin-cos.

4.9.4 FishEye

The function in charge of calculating the heigh of a wall is 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 raytracing algorithm is supposed to cast a ray for each pixel column and use the distance `d` to infer an other value. So I would have expected to see something like: $d = \sqrt{dx^2 + dy^2}$
but instead it looks like: $d = dx * \cos(\alpha) - dy * \sin(\alpha)$. Something is not right. Or misunderstood here.

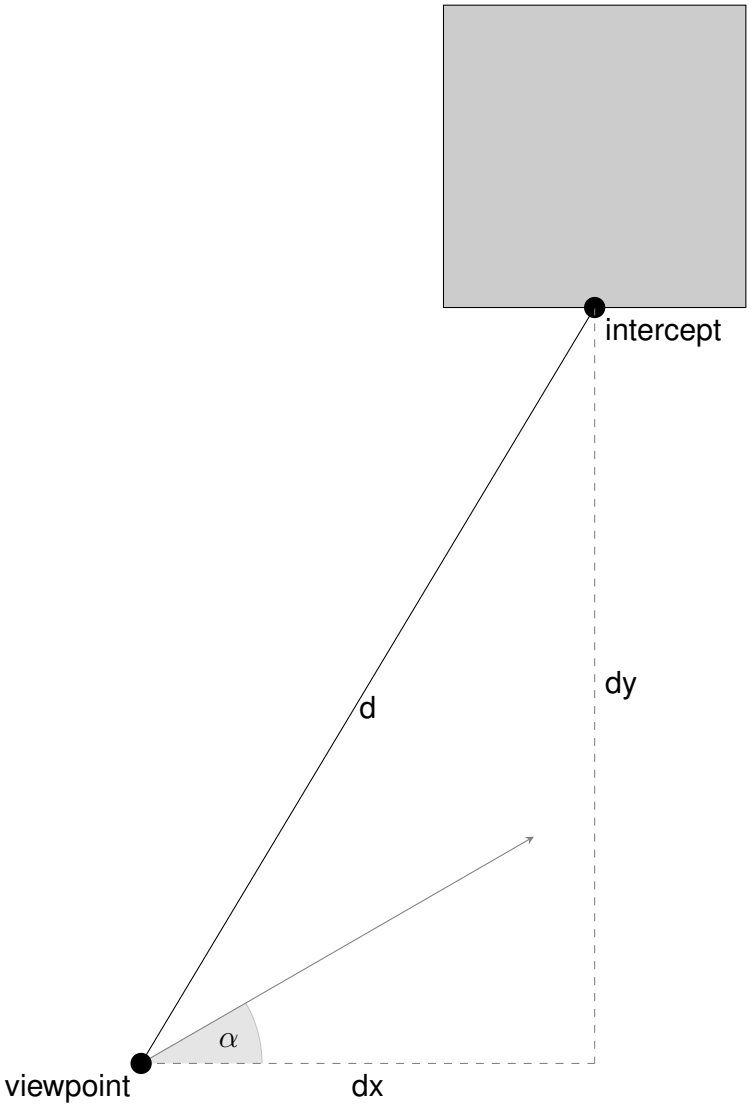


Figure 4.9: Raycasting using distance d

In this drawing the player is located at viewpoint with a view angle α. The distance `d` is a straight line between the player point of view and the location where the ray hit the line. Such an algorithm would result in a "fisheye effect". It is not very noticeable when a wall is far:

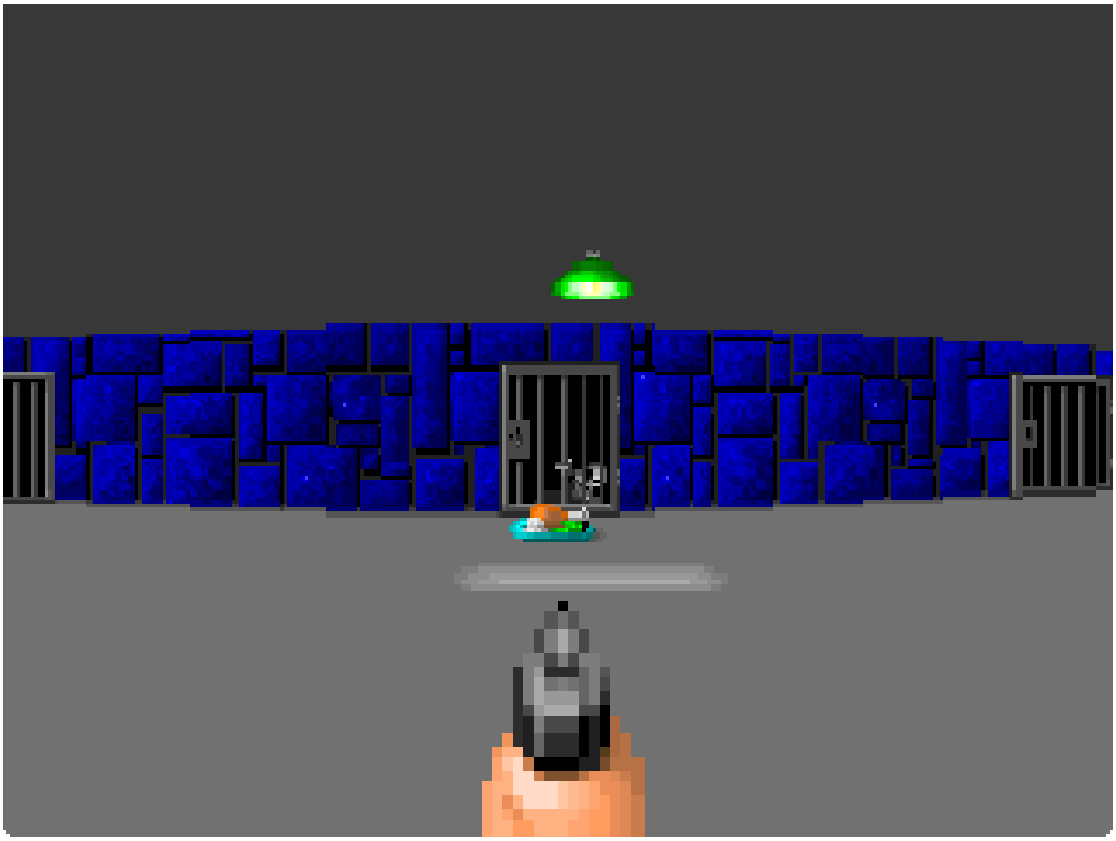


Figure 4.10: Fish eye effect: Mild

.

But it gets worse when you get closer:

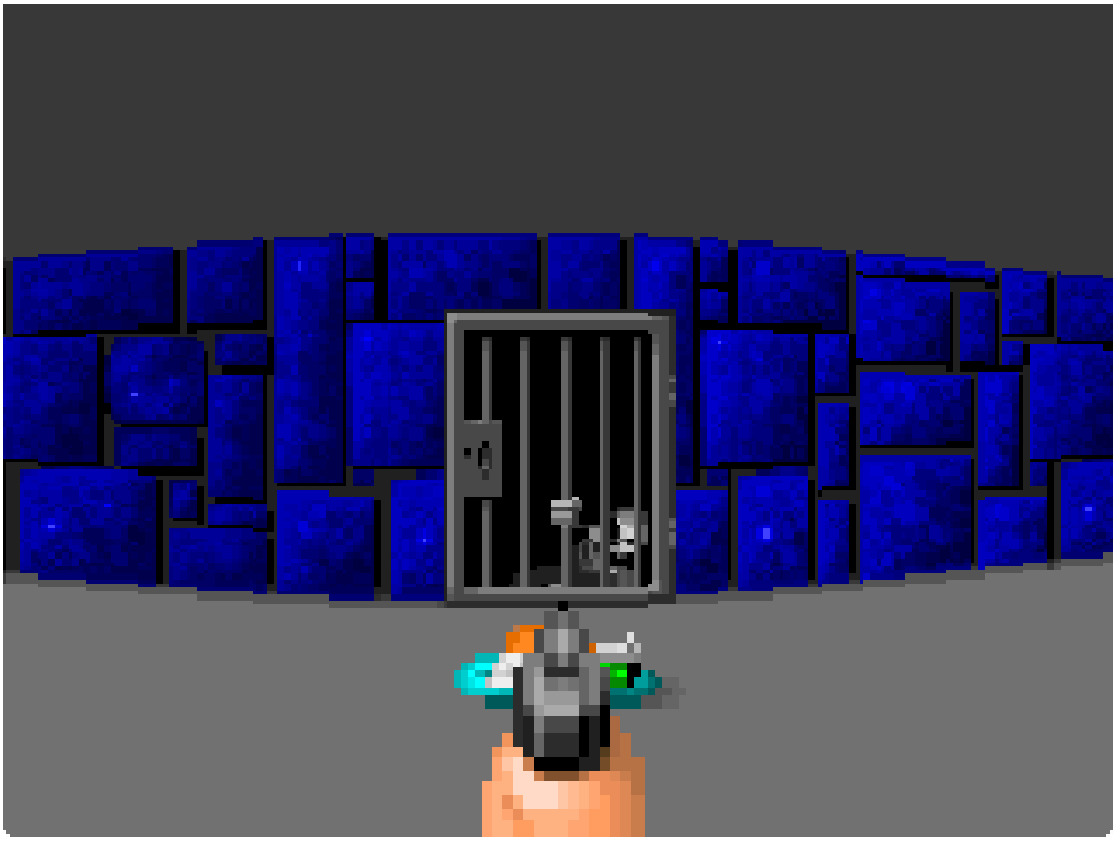


Figure 4.11: Fish eye effect: Bad

And become unbearable when close:

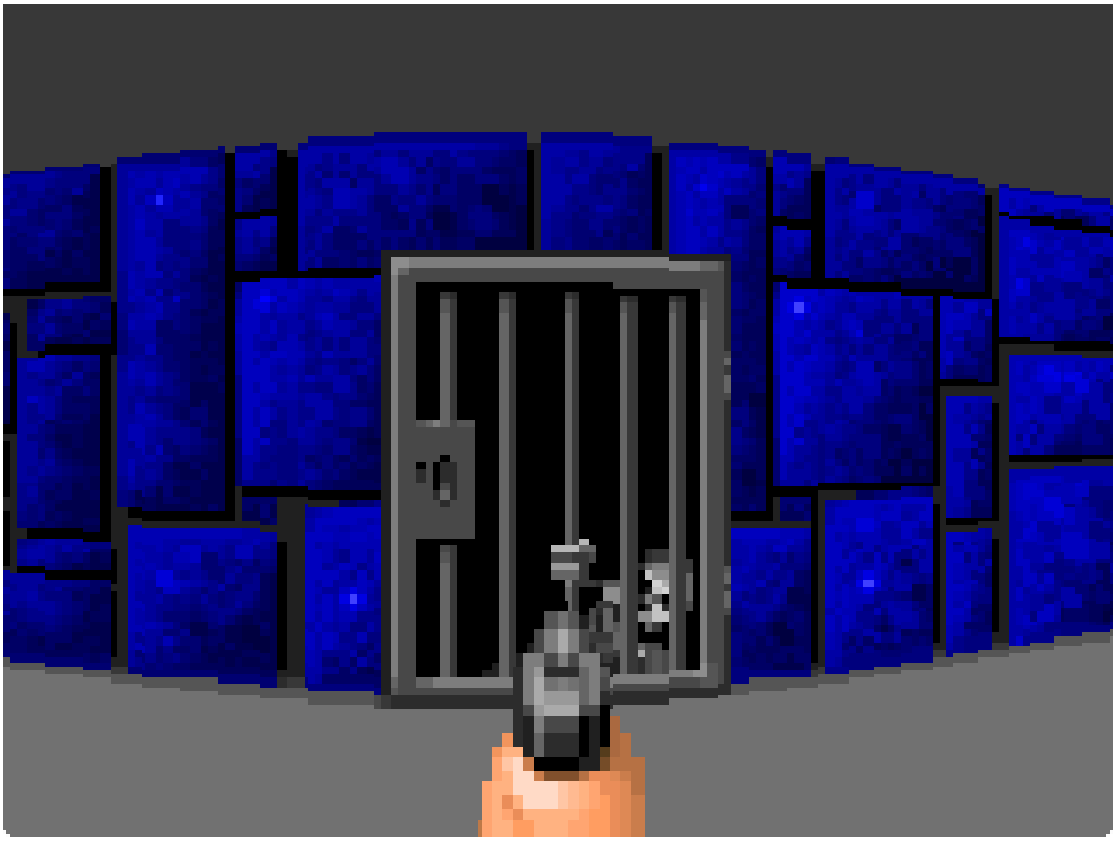


Figure 4.12: Fish eye effect: AAAAARG

To avoid this visual annoyance, what must be used is not the direct distance d but d projected on the perpendicular of the view direction:

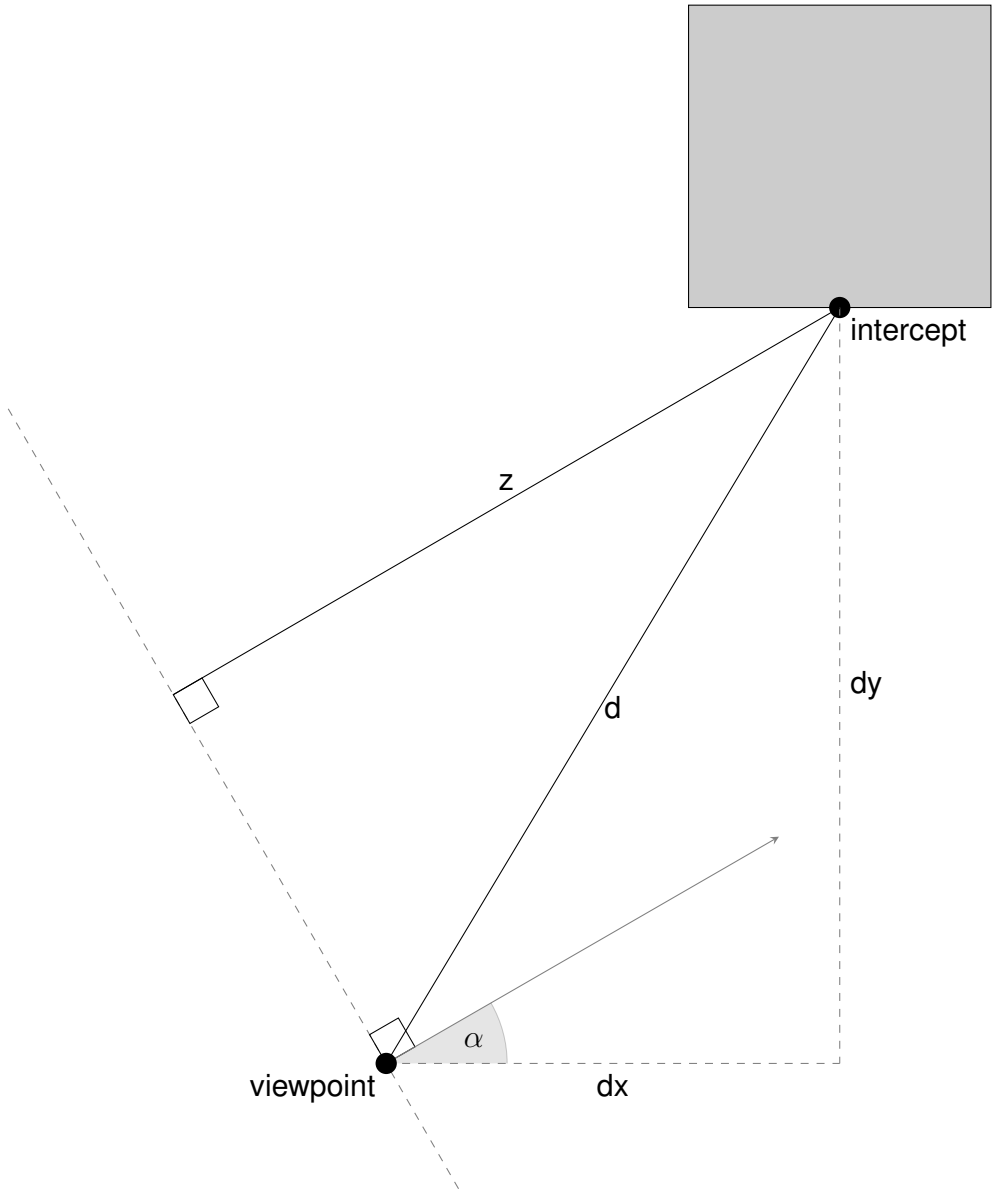


Figure 4.13: blabla.

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

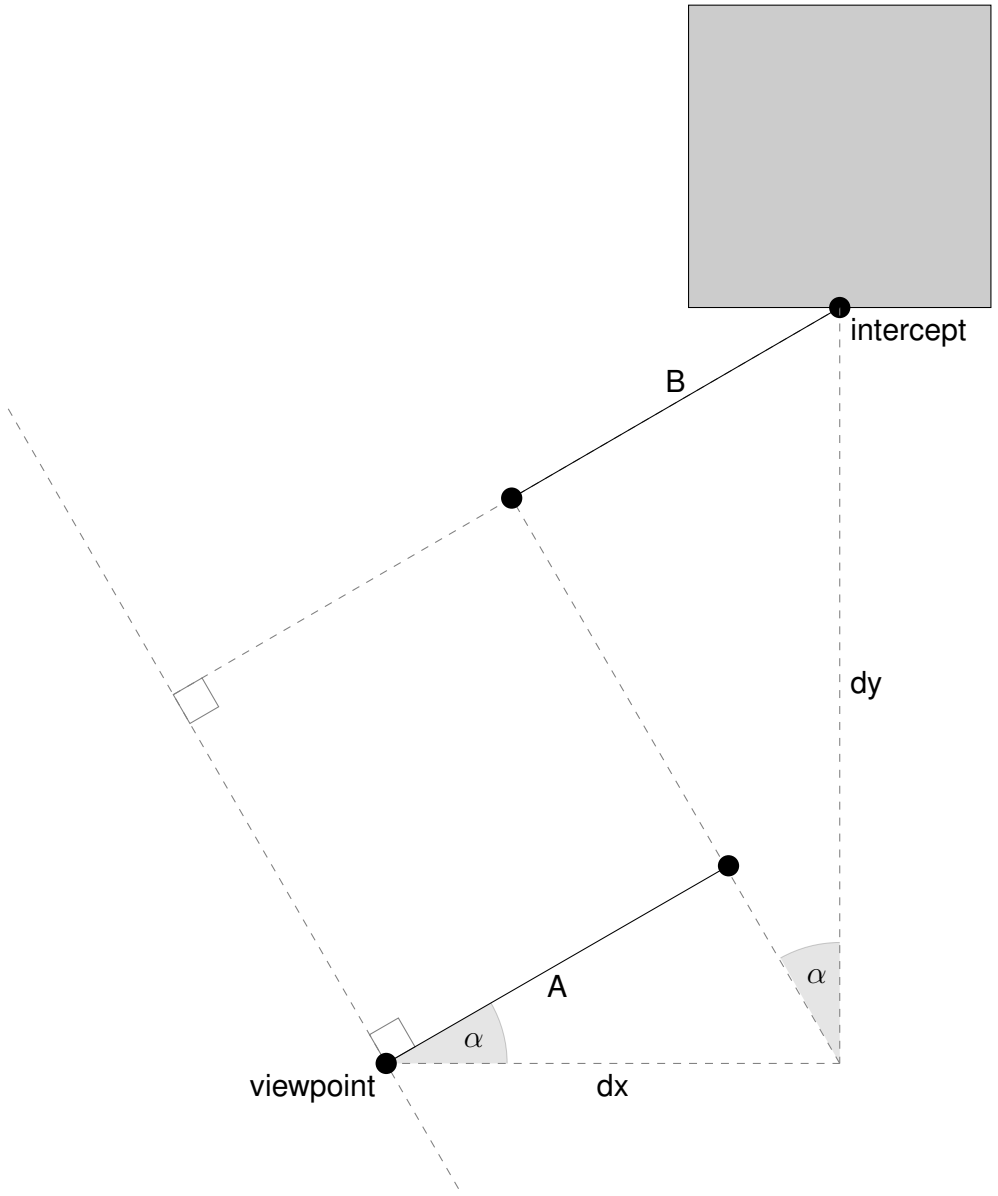


Figure 4.14: blabla.

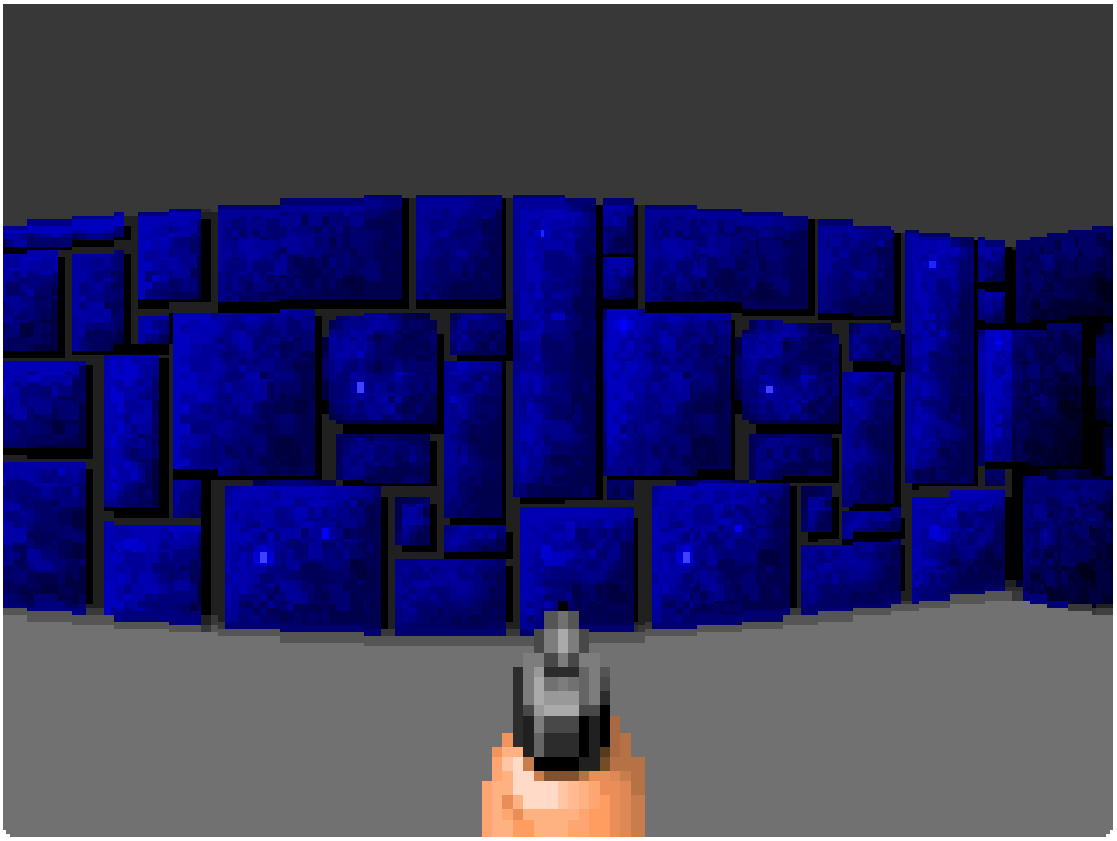


Figure 4.15: 3D Rendere Phase 1: Background

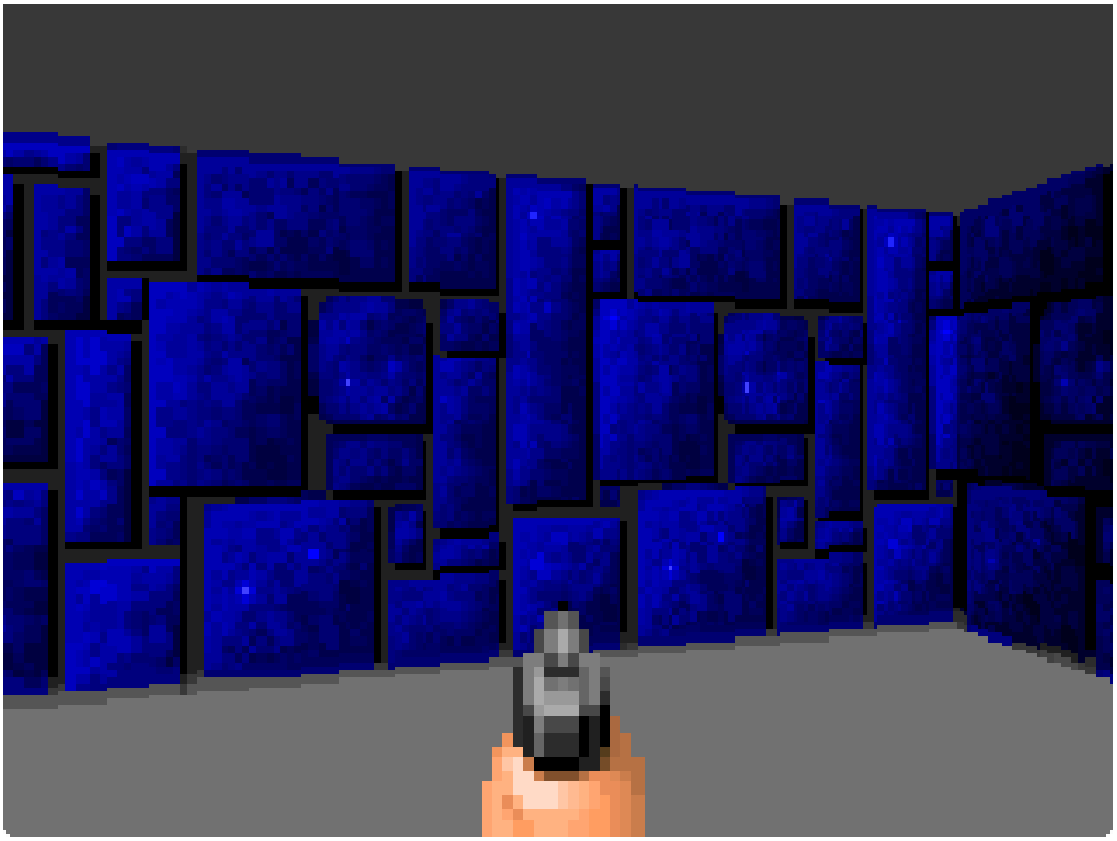


Figure 4.16: Fish eye: Uncorrected

Figure 4.17: Fish eye: Corrected

Note: Why substract values insted of adding them ? The Wolf3D coordinate system has its origin at the upper left. This inverse the sin value. To compensate for it, the correction math uses substraction instead of addition.

4.9.5 Ray skipping trick

A naive implementation of the raycaster would be:

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

But Wolf3D is different: It buffers what to draw:

```
for (int x=0; x<320 ; x++){
  castRay();
  if (raySimilarToOnesInBuffer){
      AddColumnToBuffer();
    continue;
```

```
  } else {
      DrawBuffer();
      height = CalcWallHeight();
      AddColumnToBuffer();
  }
}
```

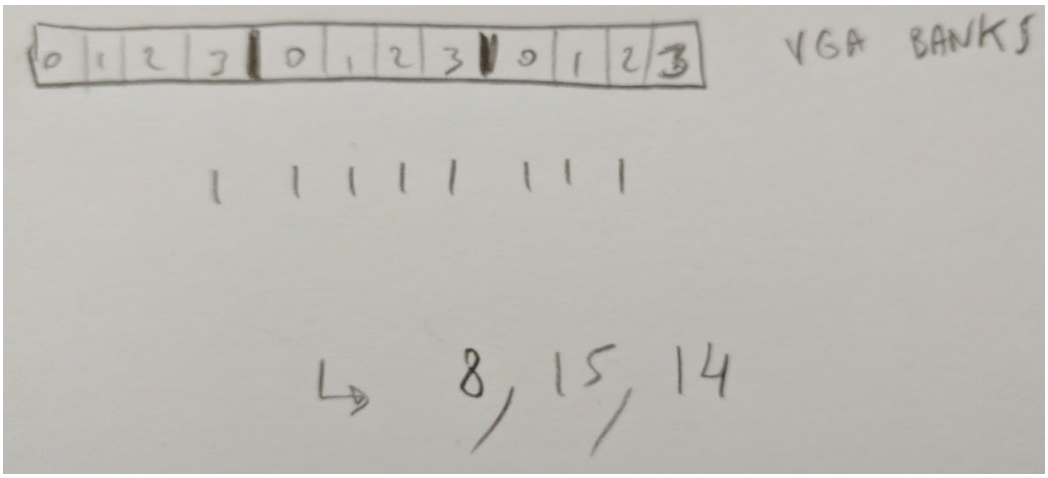
Rendition is deferred because the engine allows itself to cheat a little: If a ray is cast and hits the same block and in the same texture column, the two rays are considered similar. If two rays are similar they are drawn at the same height. The visual result is barely noticable. But this cheat allows to draw up to eight column of pixel in three write operations (in practice it is more like two columns with one write operation). The details of this are in the method `ScalePost` in `WL_DRAW.C`.

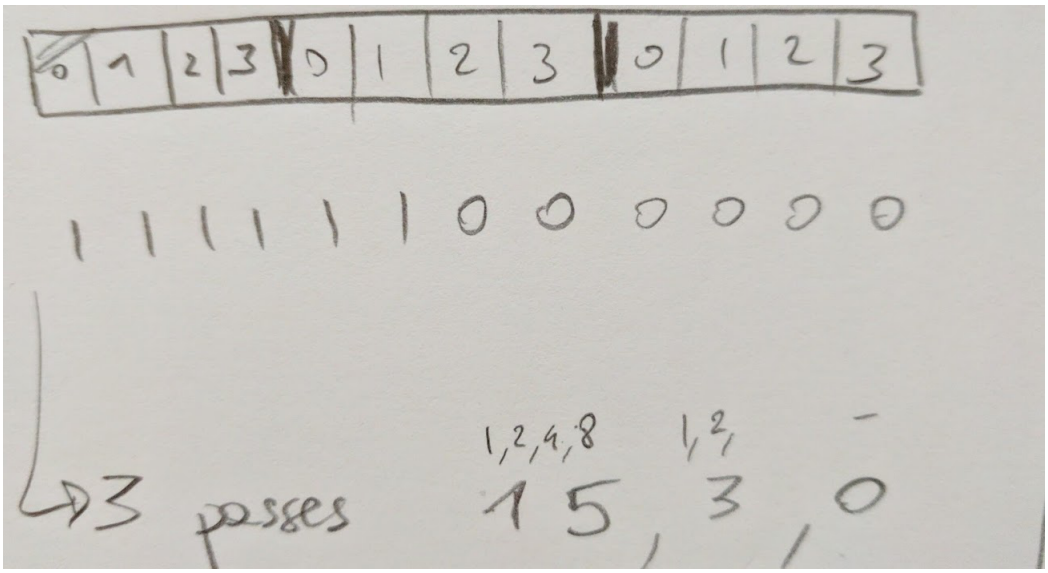
Note: A “post” is a column of pixel belonging to a wall.
ScalePost is written in assembly and performs a maximum of three pass to draw a maximum of eight columns of pixels.

```
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:
}
```

Because there can be many combinations of VGA bank alignment and number of pixels to draw:



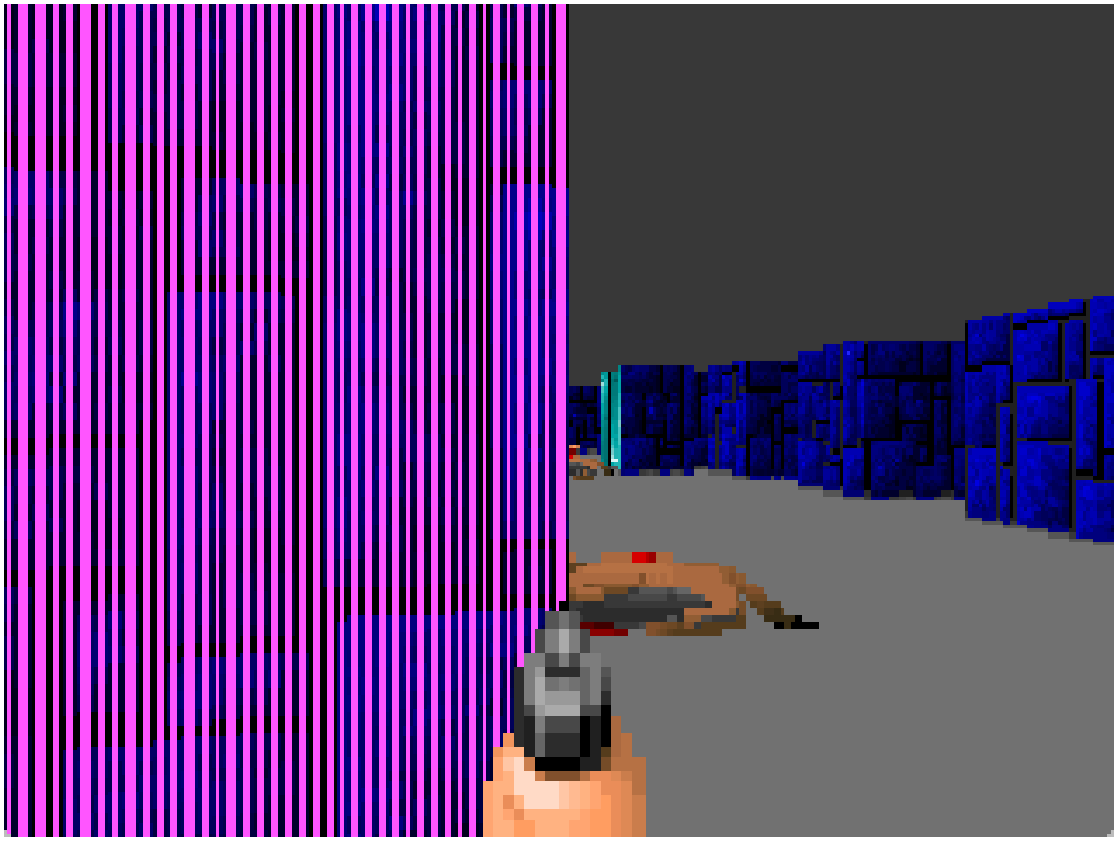


The instruction `or al , al` can be surprising. It means test if al is equal to zero and set the flag. Back in the day it was more populate than `test al, al` the VGA bank masks are hard-coded in an array: One for each pass:

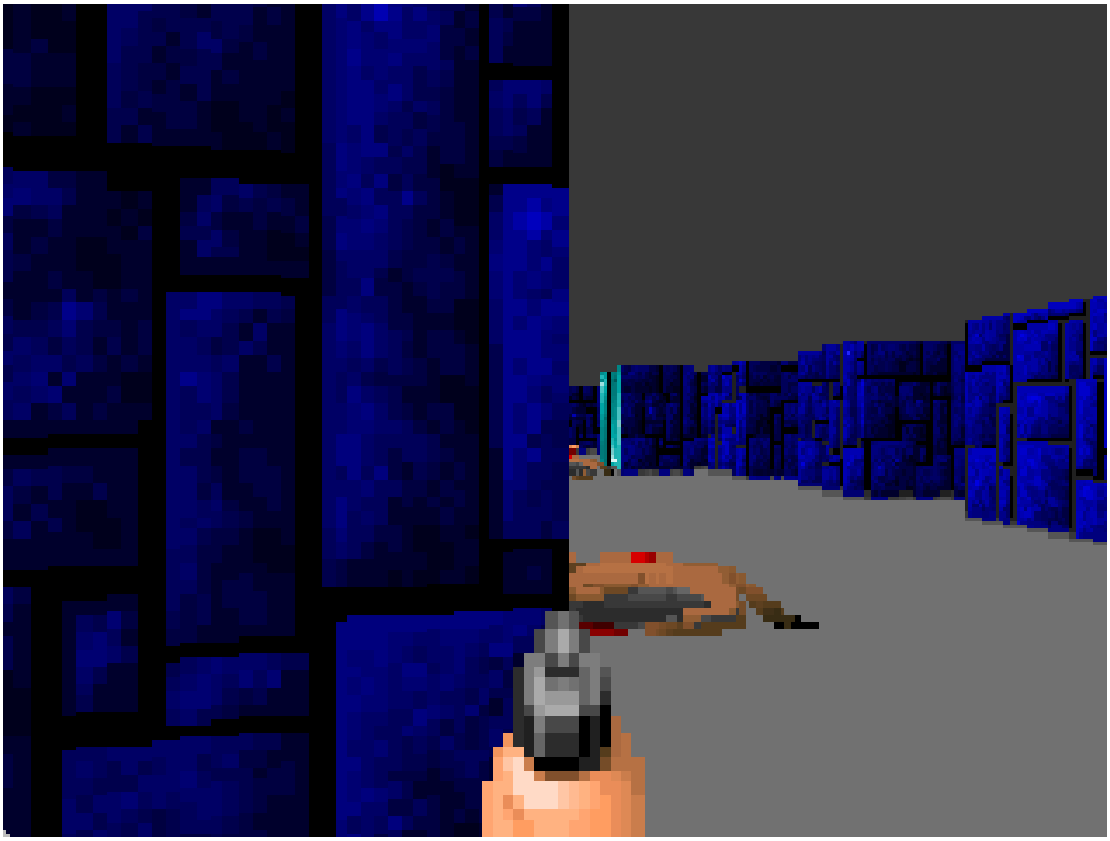
```
byte  mapmasks1[4][8] = {
{1 ,3 ,7 ,15,15,15,15,15},
{2 ,6 ,14,14,14,14,14,14},
{4 ,12,12,12,12,12,12,12},
```

```
{8 ,8 ,8 ,8 ,8 ,8 ,8 ,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} };
```

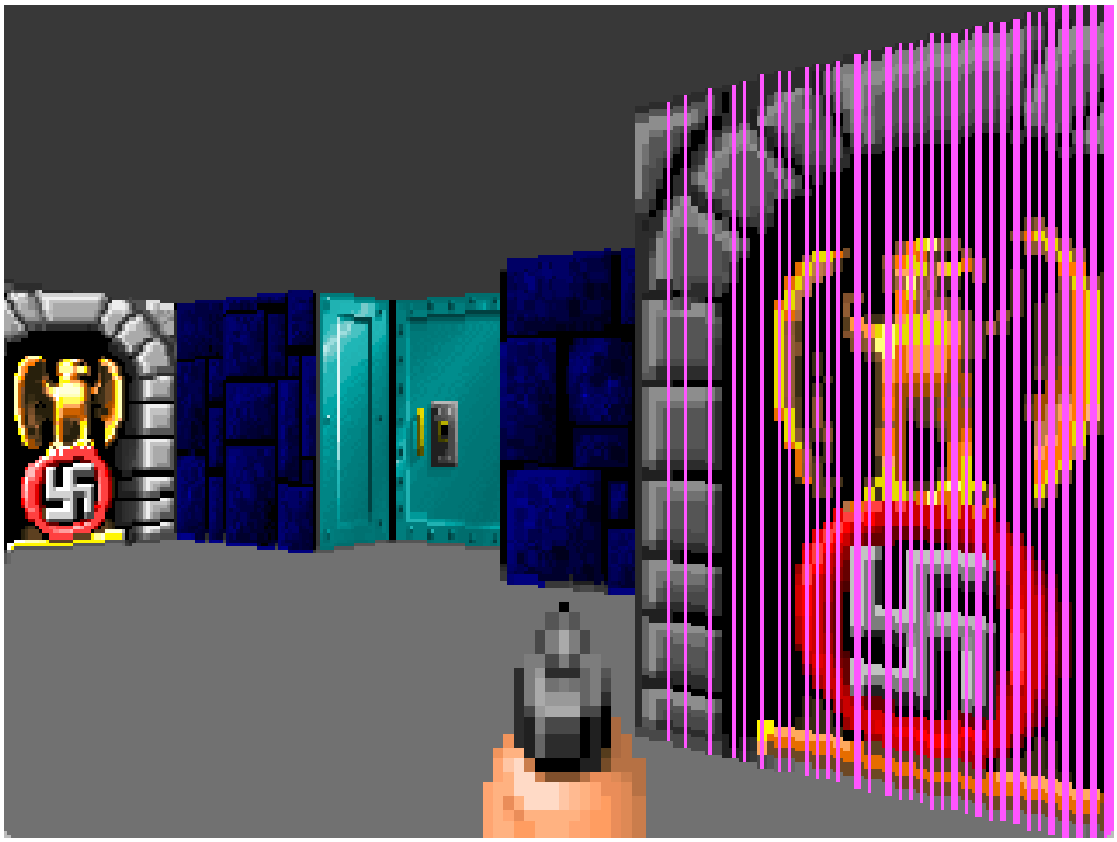
TODO: Were images stored rotated 90degres to increase cache hit ? No way, this was aimed at 386. BUT that would make the drawing easier !!
In the following example, the left wall is magnified. several rays hit the wall at the same texture location.



The engine takes advantage of it and draw several columns (posts) efficiently with the VGA. The engine was altered to show the optimized posts in pink.



In the previous scene, 20% of the wall was optimized away. In the next two screenshot, the same process, this time with the ‘eagle’ texture:



To draw several column of pixels at the same time, the engine exploit the VGA banks and the masking mechanism seen in the Hardware section. Since up to eight column can be similar, there are many case of figure depending on the aligment with the VGA banks and how many pixels to draw:
Example

4.9.6 Door

4.9.7 Texturing

Pre-Lighted texture.
Scaler (precompiled).

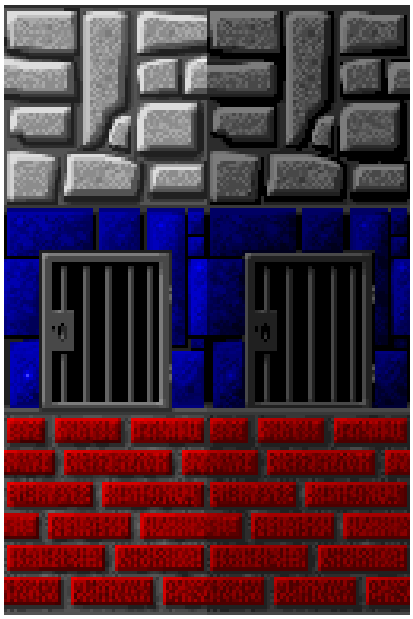


Figure 4.18: 3D Rendere Phase 1: Backed lights

4.9.8 Rendering phases

Upon finishing parameter selection, the engine would switch from Menu phase to Action Phase which was mostly about rendering the 3D environment. The renderer operated in four phases:

- Draw Background
- Draw Walls
- Draw scaled (entities such as enemies and items)
- Draw current weapon

Trivia: Hard-coded cos and sin lookup table. Budget ?

4.9.9 Square World and RayCasting

TODO: Wolfenstein 3D referencial (64 grid, with origin and rotation

4.9.10 Compiled Scalars

Builds a compiled scaler object that will scale a 64 tall object to the given height (centered vertically on the screen). `SetupScaling`
`BuildCompScale`

```
unsigned BuildCompScale (int height, byte far *code)
  ...
  work = (t_compscale far *)code;
  code = &work->code[0];

  // mov al,[si+src]
  *code++ = 0x8a;
  *code++ = 0x44;
  *code++ = src;

  for (;startpix<endpix;startpix++) {
    // mov [es:di+heightofs],al
    *code++ = 0x26;
    *code++ = 0x88;
    *code++ = 0x85;
  }
  // retf
  *code++ = 0xcb;
}
```

Figure 4.19: 3D Rendere Phase 1: Backed lights

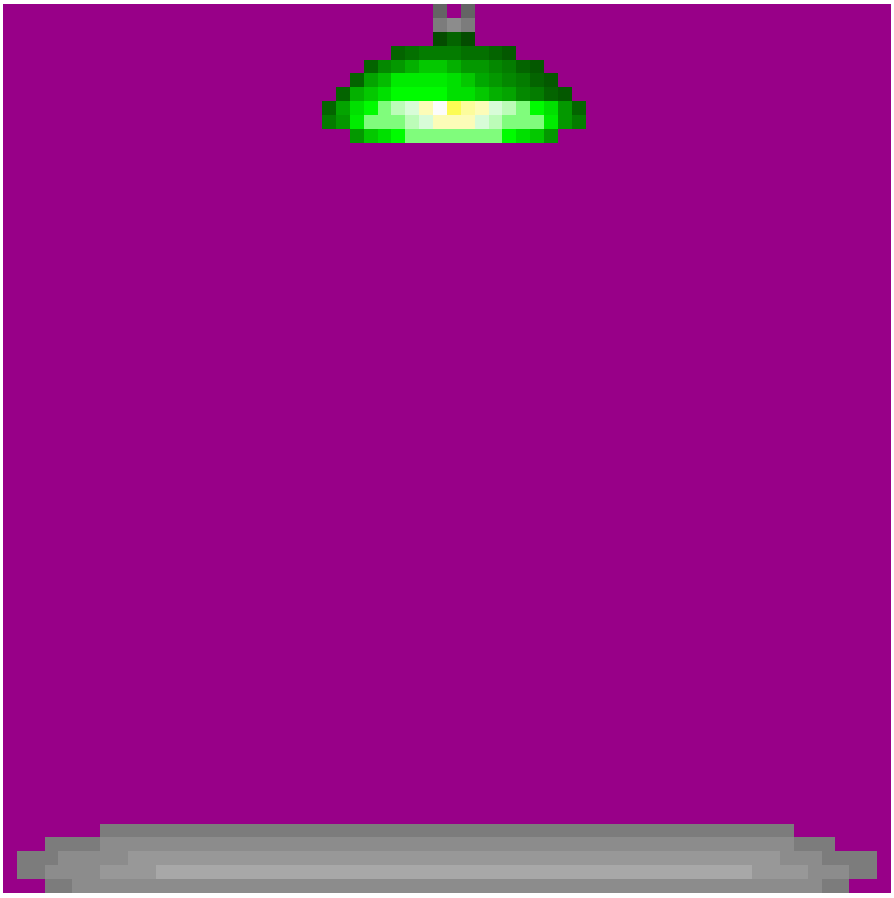


Figure 4.20: 3D Rendere Phase 1: Backed lights

Trivia: This type of sprite would later turn out a major issue with 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**

Figure 4.21: 3D Rendere Phase 1: Backed lights

Figure 4.22: 3D Rendere Phase 1: Background

The floor was always the same color. The ceiling however changed see `WL_DRAW.C` for a hard-coded index. TODO: Include a clear screen from a later level.

Figure 4.23: 3D Rendere Phase 2: Walls (160 rays)

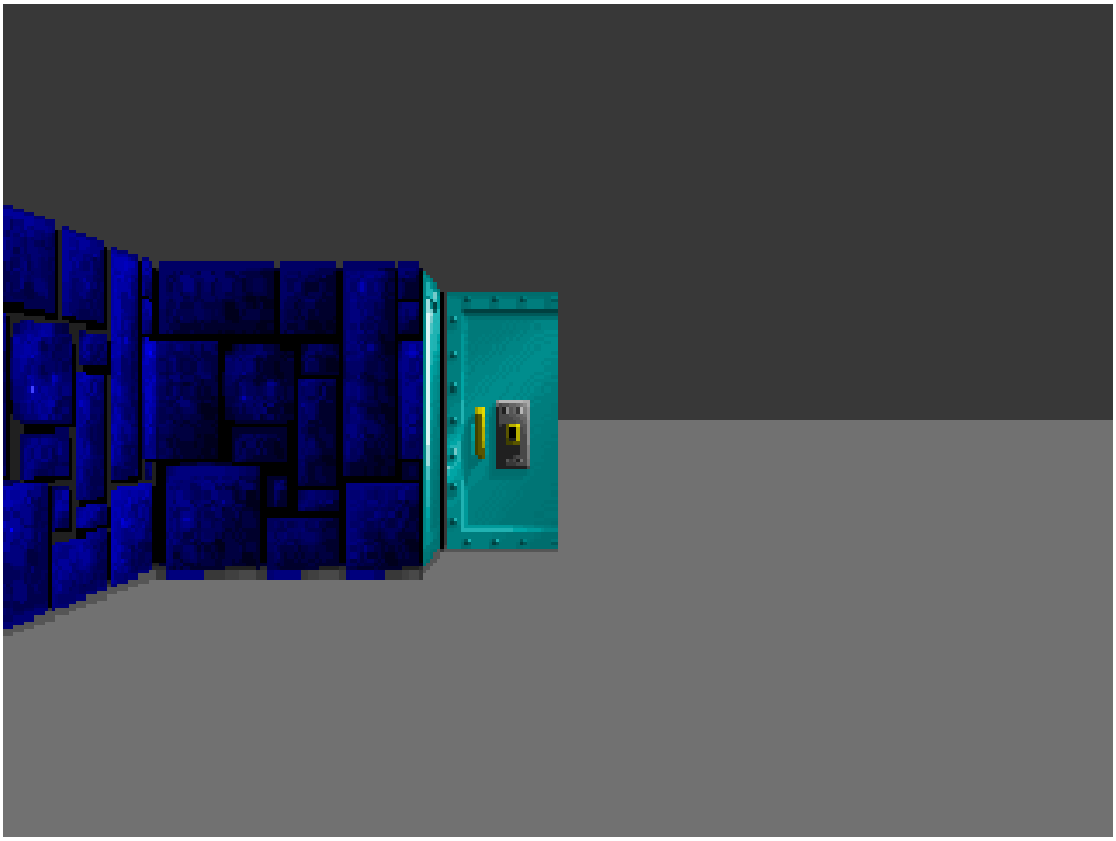


Figure 4.24: Finalez frame

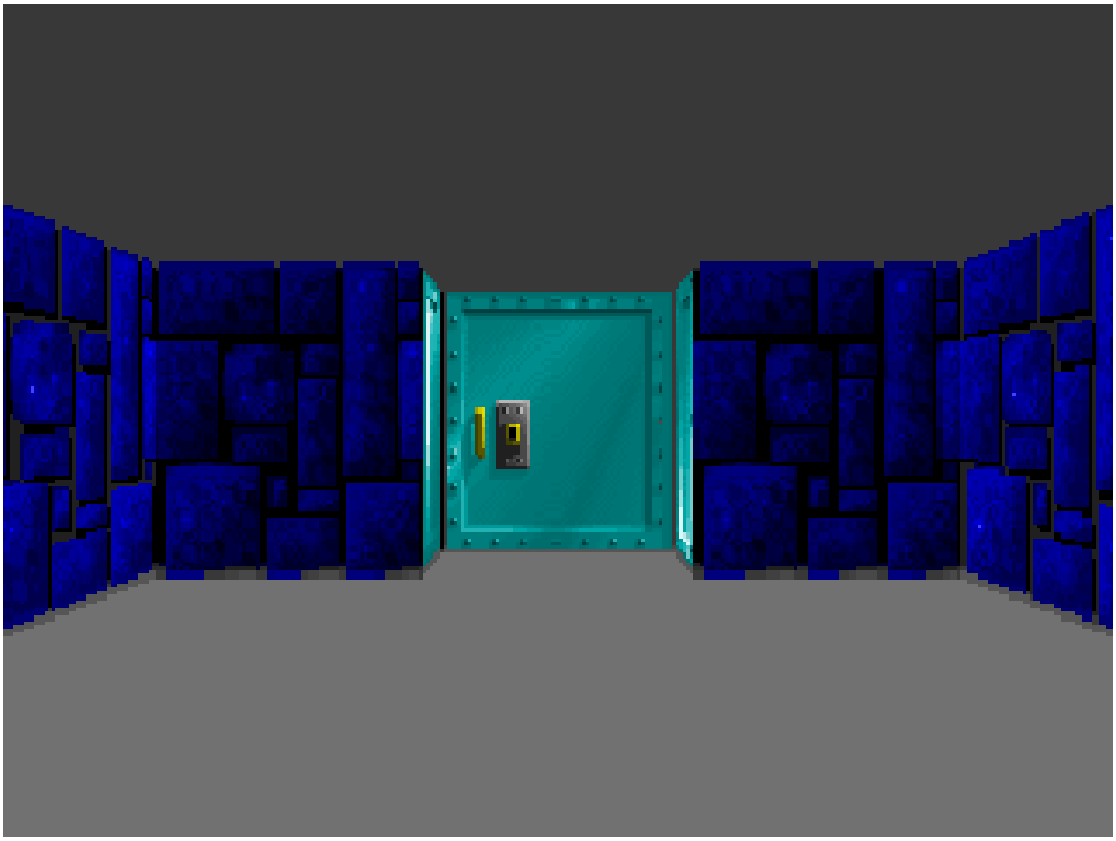


Figure 4.25: 3D Rendere Phase 2: Walls completed

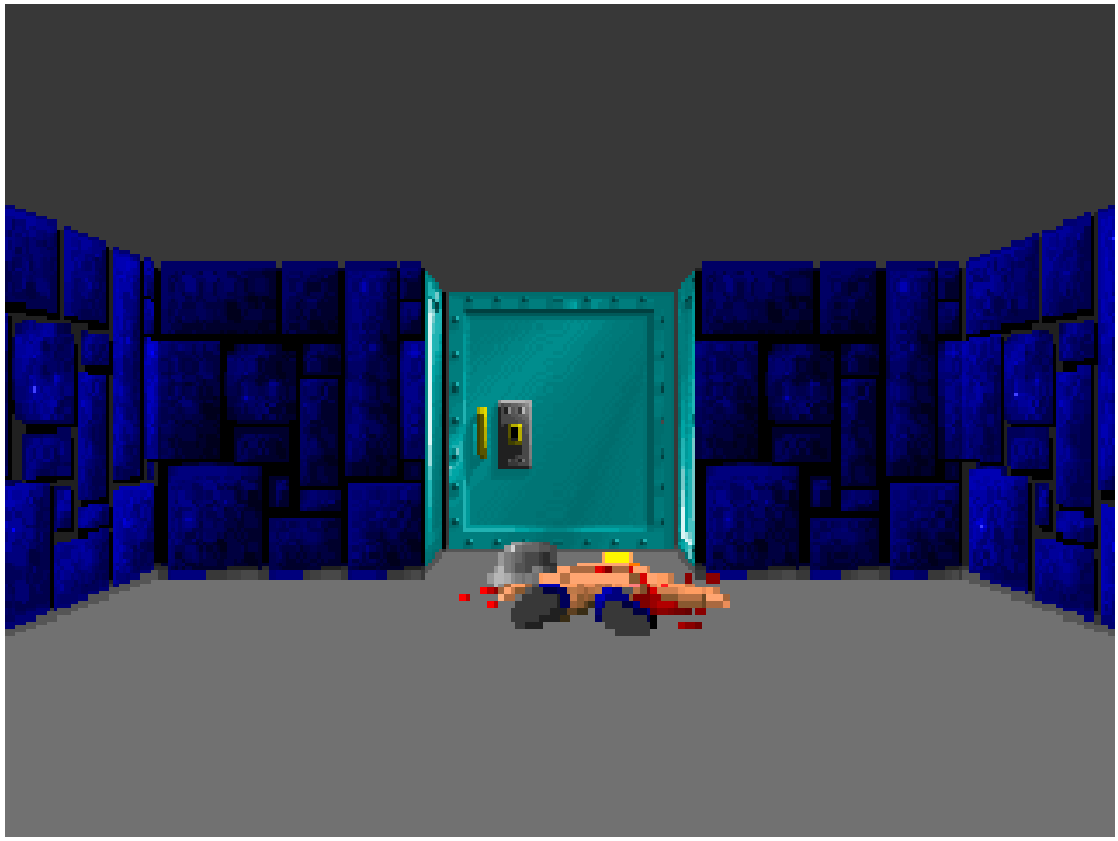


Figure 4.26: 3D Rendere Phase 3: Scaled

Figure 4.27: 3D Rendere Phase 2: Weapon

Synergy: VGA column drawing, raycasting, occlusion array

4.9.11 Rendering things

4.9.12 Loading screen

One interesting aspect: Wall sprites were stored uncompressed but things are compressed in memory and decompressed while drawing occurs via bytecode.

4.10 I.A: State Machines

If integers were not accurate enough and floating points were too slow, how could trigonometry be done on those machines? The solution was to re-purpose the hardware via Fixed Points! Fixed Point is a method that allows to keep track of fractions while still using the integer operations of the CPU. The machine manipulates what are supposed to be integer numbers but the programmers sees them as a real numbers. As usual it is easier to describe with a drawing. Integer are encoded following figure 4.28:

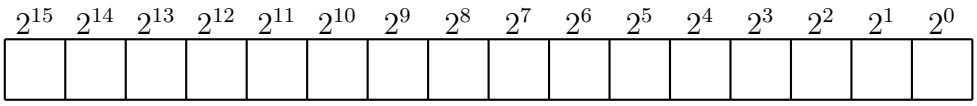


Figure 4.28: Integer layout.

So the value of the sequence of bits *0010010010010010*:

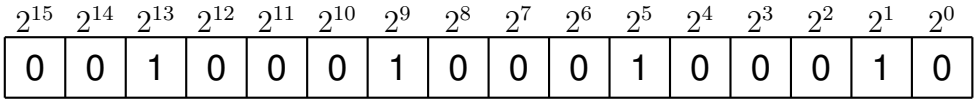


Figure 4.29: Integer example.

Is equal to $2^{13} + 2^9 + 2^5 + 2^1 = 8738$.

Fixed Point arithmetic works by simply shifting the layout 8 bits to the left:

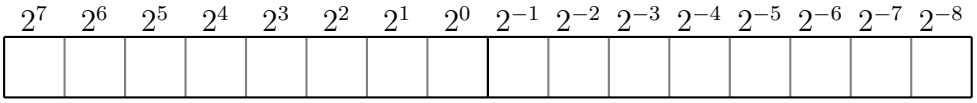


Figure 4.30: Fixed point layout.

So the same sequence of bits *0010010010010010*:

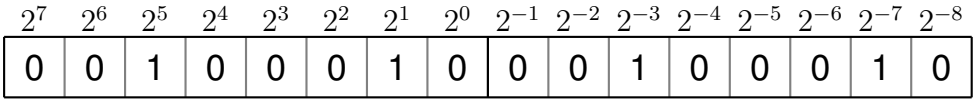


Figure 4.31: Fixed point example.

Now represents:

$2^5 + 2^1 = 34$ for the integer part.
$2^{-3} + 2^{-7} = 0.1328125$ for the fractional part.
$= 34.1328125$

The beauty of this method is that addition and subtraction work exactly like integers from the CPU instruction side: DRAWING

The only special case is when performing operation.

DRAWING

Trivia : Fixed Point Arithmetic usage was not limited to PC gaming. Many game console manufactured in the 90s and later had no hardware floating point unit: Sony's original PlayStation (1994) and Sega's Saturn (1994) are few examples among many with a design choice that not only reduced the production cost but also maximized the CPU pipeline throughput.

> 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 Catacombs 3D. C3D (and Hovertank 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**

Chapter 5

After Woftenstein 3D

Wolfenstein layed out the fundation of first person shooter and established id Software as a major development studio. The team embraced their success:

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

5.1 Spears of Destiny

Released on September 18, 1992 Spears Of Destiny used the same game engine but with new graphics, music and levels.
It was essentially a mod of Wolfenstein 3D. SOD helps to put things into perspective when it comes to being open source and allow people to figure out the file formats. SOD may not have sold as well had the game been open.
No a shareware version but 2-level playable demo was distributed
Lost Episodes, Each of these, consists of 21 levels. New level textures, new enemies, and new appearances for old enemies. Published by FormGen Corporation in May 1994."
Mission Packs(released by FormGen Corporation in May 1994, resemble many fan-made mods a.k.a: Lost Episodes):
Mission 2: Return to Danger
Mission 3: Ultimate Challenge

Trivia: Angel of Death looked like a demon and was a good insight into what would come next: Doom.

5.2 Mission Pack: Return to Danger

5.3 Mission Pack: Ultimate Challenge

5.4 Port: Super Nintendo

Story of Doom development to stop.

> 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 wasnâĂŹt 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**

5.5 Port: Jaguar

The jaguar port was the only one able to reach 60 frames per seconds. The console was a mix of powerful components with severe bottleneck:

“

The memory, bus, blitter 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 engined 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. You 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 blitter, it could have put up a reasonable fight against sony.

John Carmack - Programmer (Mar 04, 2000)

”

5.6 Port: PC-98

Essentially Japanese port. Explain the source code diff packages.

5.7 Port: 3DO

5.8 Macintosh

Published with authorization
Unlike newer games (Doom, Unreal, etc) the Macintosh version is not just a port (exact translation) of the PC version. The levels and the game engines are actually quite different. Thus, you can not use levels and mods for one version with the other version.

5.8.1 Levels

The first major difference is the level design. While the PC version contains 6 episodes with each episode containing 10 levels (1 of those is a hidden bonus level), the Macintosh version (reffered to as the 2nd Encounter) contains 30 levels with no divisions into episodes. The Mac version is divided into floors, but you can not start on a specific floor and every floor has a different amount of levels. A 3rd Encounter was later released for Macintosh containing 60 more levels which were supposed to make up episodes 2 to 6.
Obviously, the number of levels aren't the same but this doesn't necessarily mean that the Macintosh version is giving you more. Many of the levels are similar, but the Macintosh levels are usually much less intricate and much shorter. This is apparent in the first level where the path to the exit is exacly the same in both versions, but the PC version contains many more possible side-trips and many more rooms. Other levels in the games are completely different. Basically, the 2 games play extremely differently and offer unique experiences.

5.8.2 Game Engine

Another major difference is in the game engines. At first glance, the Macintosh engine, which is 2 years newer, seems to be the more advanced. The graphics in the Macintosh version are much sharper, clearer, and better looking at twice the resolution of the PC graphics. Also, the game takes up a slightly larger portion of the screen and the sounds are much higher quality in the Macintosh version.
However, the enemies in the Macintosh version are only 2D sprites whereas the enemies in the PC version are 3D sprites. This means that the Macintosh enemies only have 1 side, a front side, and thus, they are always facing you. In contrast, the PC enemies have 8 sides, making for much more realist play; in the PC version, it is possible to sneak up behind guards and kill them before they see you - something that can't be done in the Macintosh version.

The enemies in the PC version also offer an extra level of realism by being able to patrol an area. In the Macintosh version, the enemies will just stand in one spot until they see you or hear you; in the PC version, enemies will stand in place or patrol an area until they see or hear you. These differences make the PC version's game engine more advanced despite being 2 years older.

5.8.3 Weapons, Items, and Enemies

Several other difference worth noting include the addition of 2 extra weapons in the Macintosh version: the flamethrower and the rocket launcher. Those two weapons and some other items (bullet crates, flame thrower and rocket ammo, backpacks) are not found in the PC version. Also, some other pick-ups and dropped items are different. For example, in the PC version, dropped clips give 4 ammo and other clips give 8 ammo; in the Macintosh version, all clips give 5 ammo. Also, picking up treasure in the PC version adds to your score (40000 points gets you an extra life); in the Macintosh version, picking up treasure increases your item count by one until you get an extra life at 50 items.
Other differences include an additional enemy, the Flying Hitler, in the PC version and different bosses in the 2 versions (see the enemies and bosses pages); the Mac version actually uses some bosses from Wolfenstein 3D and some bosses from Spear of Destiny.

5.8.4 Miscellaneous

Yet another difference is that changing the difficulty level in the Macintosh version only affects the damage taken and starting ammo whereas in the PC version it affects the damage taken and the number of enemies; the PC version tends to be much more difficult. The Macintosh version also has an auto-map that is helpful in finding you way around; there is no mapping feauture in the PC version. The last difference I can think of is the pools of blood in the PC version. In the Mac version, you only have pools of water - no pools of blood. If your health is less than 11 in the PC version, you can drink the blood to gain back some health.

5.9 Port: Game Boy Advance

Released in 2002, the Game Boy Advance version is an nearly exact port of the PC version. It contains all 6 Episodes. The only difference is that you can only save the game at the end of a level - not during.

5.10 Modding

Appendices

Appendix A

Let’s compile like it’s 1992

Appendix B

The 640KB Barrier

The problem with conventionnal memory limitation was 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 compatibi-
lity 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 withenough 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, 386MAX, or some other memory management program.

Appendix C

CONFIG.SYS and AUTOEXEC.BAT

Upon startup, the operating system read two files automatically from the booting device (which could be either the hard-drive(C:) or floppy disk (A:) CONFIG.SYS:

```
DEVICE=C:\WINDOWS\HIMEM.SYS
DOS=HIGH,UMB
DEVICEHIGH=C:\WINDOWS\EMM386.EXE AUTO RAM
DEVICE=C:\WINDOWS\MOUSE.SYS
```

AUTOEXEC.BAT:

```
@echo off
SET SOUND=C:\\CREATIVE\CTSND
SET BLASTER=A220 I5 D1 H5 P330 E620 T6
SET PATH=C:\DOS;C:\
```

Trivia: To this day, the old way to refer to drives is still in Windows (C:):

- A: used to be the floppy drive containing the program code.
- B: the floppy drive containing the data.
- C: became used when hard drives became more wide-spread.
- D: was adopter when CD-ROM reader apperared later.

Appendix D

Letter from a war veteran

Appendix E

Chocolate Wolfenstein 3D

Appendix F

Release Notes by John Carmack

```
   RELEASE.TXT
-----------

::

  We are releasing this code for the entertainment of the
  user community.  We don't guarentee 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 embarassed
  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 cash and wrecking performance.
   A simple looping texture mapper would be faster on 486+ machines.

   The whole page manager caching scheme is unecessarily 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

20 years anniversary commentary by John Carmack

For the 20 years anniversary, JOhn Carmack revisited and played the game with commentaries. Here is the transcript :

```
The engine (4mn 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 axises.

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 dialogue up on the screen without invoking ten
different frameworks and fifty thousand lines of code.

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.



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
```

Appendix H

File formats

H.1 Maps

H.2 3D Assets

H.3 2D Assets

H.4 Music

H.5 Sound effects