

GAME ENGINE BLACK BOOK

DOOM

FABIEN SANGLARD

Copyright

In order to illustrate how the DOOM game engine works, a few screenshots, images, sprites, and textures belonging to and copyrighted by id Software are reproduced in this book. The following items are used under the "fair use" doctrine:

1. All in-game screenshots, title screen.
2. All in-game menu screenshots.
3. All 3D sequence textures.
4. All 3D sequence sprites.
5. All screenshots of DOOM.
6. DOOM name.

Photographs with "ROME.RO" watermark belong to John Romero and are reproduced with his authorization.

DOOM Survivor's Strategies & Secrets essays are copyrighted by Jonathan Mendoza and reproduced with his authorization.

Acknowledgments

Many people helped completing this book. Many thanks are due:

To John Carmack, John Romero, and Dave Taylor for sharing their memories of DOOM development and answering my many questions.

To people who kindly devoted time to the painful proofreading process, Aurelien Sanglard, Jim Leonard, Dave Taylor, Jonathan Dowland, Christopher Van Der Westhuizen, Eluan Miranda, Luciano Dadda, Mikhail Naganov, Leon Sodhi, Olivier Cahagne, Andrew Stine, and John Corrado.

To Simon Howard, for not only proofreading but also sending pull requests to the git repo. His efforts saved countless hours at a time where the deadline was concernedly close.

To Jim Leonard who once again volunteered his time and encyclopedic knowledge of audio hardware and software (the Roland section was heavily based on his articles).

To Foone Turing who volunteered his fleet of 386s, 486s, and ISA/VLB VGA cards to accurately benchmark DOOM.

To Andrew Stine, founder of doomworld.com, for sharing his encyclopedic knowledge of DOOM and putting me in touch with the right people.

To James Miller and Leon Zawada who researched and discovered the origin of the backgrounds. James also collected and photographed all toy props used to shape DOOM weapons.

To Rob Blessin, founder and owner of Black Hole, Inc for answering all my questions about NeXT, helping me assemble a NeXTstation, and lending me a rare NeXTdimension board. If you ever want to restore a NeXT or acquire your own, Rob is likely a good starting point.

To Alexey Khokholov, author of PCDoom-v2. His backport helped to generate accurate performance metrics.

To Alexandre-Xavier Labonté-Lamoureux for his patch restoring C drawing routines in PC Doom-v2.

To Simon Judd, author of Slade3, a map editor used to create maps showcasing special aspects of the renderer.

To Colin Reed and Lee Killough for their node builder, BSP 5.2, which was used to inject maps into the DOOM engine.

To the developers of Chocolate DOOM which was heavily hacked to generate many explanatory screenshots.

To Bruce Naylor for kindly making time for an interview and enlightening me with his master knowledge of BSPs.

To John McMaster for his insanely high resolution photos of Intel 486 and Motorola 68040 CPUs.

To Romain Guy for taking the pictures of my 486 motherboard, my NeXTCube motherboard, and my NeXTDimension motherboard.

To Samuel Villarreal for finding the origin of the BFG artwork and reverse engineering DOOM console animated fire.

To Rebecca Heineman (author of DOOM 3DO) for proof-reading and fact checking the 3DO section.

To Carl Forhan, owner and founder of Songbird Productions for releasing DOOM Jaguar source code and answering my questions.

To Leon Sodhi, for sharing his studies of DOOM wall rendition.

To John Corrado, for sharing his knowledge of visplanes.

To Matthew S Fell, author of the Unofficial DOOM specs which was instrumental in building the map visualizer featured in this book.

To Alexandre-Xavier Labonté-Lamoureux for providing the SNES screenshot demonstrating dithering and diminished lightning.

To Aiden Hoopes, Alexandre-Xavier Labonté-Lamoureux (axdoomer), Anders Montonen, coucouf, Bartosz Pikacz, Bartosz Taudul, Boris Faure @billiob, Brandon Long, Brian Gilbert

@troidann, Chris @JayceAndTheNews, Daniel Lo Nigro, Daniel Monteiro , Davide Gualano @davesio, George Todd, Guilherme Manika, Jamis Eichenauer, John Corrado, Klaus Post, Marcel Lanz, Marcell Baranyai, Marco Pesce, Marcus Dicander, Matt Riggott, Matthieu Nelmes, Miltiadis Koutsokeras, Olivier Cahagne, Olivier Neveu, Patrick Hresko, phg, Richard Adem @richy486, Rory Driscoll, Ryan Cook, Sam Williamson, Steve Hoelzer, Tor H. Haugen @torh, tronster, Tzvetan Mikov, Vasil Yonkov, Frank Polster, Boris Chuprin, Rory O’Kenny and Vincent Bernat for reporting errata.

– Fabien Sanglard
fabiansanglard.net@gmail.com

How To Send Feedback

This book strives to be as accurate and as clear as possible. If you find factual errors, spelling mistakes, or merely ambiguities, please take a few minutes to report them on the *Game Engine Black Book: DOOM* companion webpage located at:

<http://fabiansanglard.net/gebbdoom>

Thanks :) !

Foreword by John Carmack

In many ways, DOOM was almost a "perfect" game.

With hindsight and two decades more skill building, I can think of better ways to implement almost everything, but even if I could time machine back and make all the changes, it wouldn't have really mattered. DOOM hit a saturation level of success, and the legacy wouldn't be any different if it was 25% faster and had a few more features.

The giant aliased pixels make it hard to look at from a modern perspective, but DOOM felt "solid" in a way that few 3D games of the time did, largely due to perspective correct, subpixel accurate texture mapping, and a generally high level of robustness.

Moving to a fully textured and lit world with arbitrary 2D geometry let designers do meaningful things with the levels. Wolfenstein 3D could still be thought of as a "maze game", but DOOM had architecture, and there were hints of grandeur in some of the compositions.

Sound effects were actually processed, with attenuation and spatialization, instead of simply being played back, and many of them were iconic enough that people still recognize them decades later.

The engine was built for user modification from the ground up, and the synergy of shareware distribution, public tool source release, and early online communities led to the original game being only a tiny fraction of the content created for it. Many careers in the gaming industry started with someone hacking on DOOM.

Blasting through the game in cooperative mode with a friend was a lot of fun, but competitive FPS deathmatch is one of the greatest legacies of the game. Seeing another player running across your screen, converging with the path of the rocket that you just launched, is something that still makes millions of gamers grin today.

There was a lot of clever smoke and mirrors involved in making DOOM look and feel as good as it did, and it is a testament to the quality of the decisions that so many people thought it was doing more than it actually was. This remains the key lesson that still mat-

ters today: there are often tradeoffs that can be made that gets you a significant advantage in exchange for limitations that you can successfully cover up with good design.

– John Carmack

Foreword by Dave Taylor

I find the technology behind great products fascinating, but I'm even more fascinated by the conditions that lead to that technology.

I don't feel in any way responsible for the greatness of DOOM. Technologically, that was all Carmack, and anything that came close in my code was only due to his influence.

In the course of struggling to keep up with Carmack, I would sometimes fall asleep in the office. I remember hearing that at least for a window there, Carmack felt guilty that I was working so hard. Knowing his intense work ethic, perhaps that put a little more spring in his already indefatigable step.

What I do know is that I had joined an already well-oiled development team in the form of Carmack, Romero, Adrian, and Kevin. This wasn't their first rodeo.

I later learned the mother of their invention was a harsh time constraint at Soft Disk, a company that sold a subscription of curious demos on diskette. They wanted to start making games instead of just toy applications, and their manager allowed them, but only if they could deliver a game every two months like clockwork.

Back then, there were no game engines. In fact, DOS was not a complete operating system, and you had to finish writing the drivers for your game to work. So this was an incredibly harsh time constraint, and it forced them into doing constant triage on what they wanted to achieve with each game.

I was so convinced that this painful constraint is what led to the team's impressive work on DOOM, that years later, I would teach a university class inspired by id's path to success. The other teachers warned me not to be nice to the students, or they would take advantage of me. Relishing an acting challenge, these were my first words to the class: "My name is Dave Taylor. I'm a working producer, and I don't have time for you. You owe me a shipped game every week, and if you don't ship, you'll get an F."

Three classes and 127 games later, almost all of my students went on to jobs in the game

industry, no mean feat with a game design degree, and many of them credit the painful time constraints of the class. Terror works. I recommend it.

Which brings me to our newest source of terror. I am not popular amongst my peers for having a low opinion of games. I consider what we make an opiate for the mind with about as much redeeming value. I met one too many DOOM fans who expressed their enthusiasm in how it threatened their GPA's, their relationships with their significant others, and their employment. That used to be funny and flattering until I integrated the area under the curve.

Scalar money commerce does not formally motivate us to make games that are good for you, just to make money, which unfortunately makes us complicit in the Holocene Extinction now threatening the planet. We need to start changing behaviors quickly, which means we desperately need games to help us change those behaviors.

Terrified? Good. Get to work!

=-ddt->

Foreword by John Romero

The year of 1993 was a magical one, more so than any other. It was the only time we challenged ourselves as a group to create a game that was as good as anything we could have imagined at the time. We didn't challenge ourselves like that before DOOM, nor after it. It was the right time to shoot for the stars.

Incredibly, and perhaps a bit naively, we made a list of the technological wizardry we planned to create, and boldly stated in a press release in January 1993, that DOOM would be a major source of productivity loss around the world. We truly believed it, and worked hard that year to make it happen. I don't recommend writing a press release at the start of your project, especially one like that.

We did so many new things while creating DOOM. It was our first 3D game to use an engine that broke away from the 2D paradigm we were in from the start of the company, and even stayed in with Wolfenstein 3D and Spear of Destiny, at least for the map layouts. We wanted to use a video camera to scan in our weapons and monsters because we were using real workstations this time around - the mighty NeXTSTEP computers and operating system of Steve Jobs.

Making DOOM was difficult. We were creating a darker-themed game with our creative director Tom Hall who is an absolutely positive guy, and it was anathema to his design ethos. He laid the initial design groundwork by creating the DOOM Bible which outlined several design concepts we never implemented, some of which were included in 2016's reboot.

The engine was revolutionary in that it represented a type of world that no one had seen on a computer screen before. Angled walls and halls that darken in the distance. A high-framerate nightmare some would call it, but it was a high octane blastfest that opened everyone's eyes to the potential of the PC's gaming future. Today's first-person shooters trace their lineage back to this game that bears the distilled essence of what a shooter should be: balanced weapons, insidious level design, a complementary enemy menagerie, and lots of fast action.

Throughout the year we tweaked, and added, and removed elements of the game to make it just right. Gone were the score and lives, remnants of the arcades we grew up in. The items that supported a score were removed. The game was far better for it, and those choices influenced our future designs.

The application of Bruce Naylor's binary space partition was a huge advance for 3D rendering speed, and the abstract level design style broke games out of the 90-degree maze wall design rut they had been in for 20 years. This was something new, with textured floors and ceilings, stairs, platforms, doors, and blinking lights. We loved having this design palette to work with, and it fit well with the subject matter we based the game upon: Hell.

As a group, we played Dungeons & Dragons for years. Our main campaign was destroyed by demons teleporting onto the material plane and destroying everything in it. This gave us the idea of a demonic invasion, but we decided to base it in the future where we could have some really powerful weapons. Besides, the combination of Hell and science fiction was too great to ignore. We felt even the storyline was slightly new because of it.

Writing the DoomEd map editor to create levels was a dream. I was finally using a real operating system with an incredible programming language, Objective-C, and getting to program in a way I had never known. The fact that we had monitors at 1120x832 let us see our game in a way we couldn't under DOS. Using these tools of the future helped us immensely.

There was so much we did that was new, it was a little mind-boggling. We were using high-end workstations, a brand-new 3D engine that allowed for incredible graphics and design expression, graphical scanning of our game sprites, and for the first time we were putting multiplayer into our game with a mode I called Deathmatch because that name just made sense.

The inclusion of multiplayer co-op and deathmatch modes changed everything about games. We knew that playing a game as fast and over-the-top as DOOM would signal a new era. I visualized what E1M7 would look like with two players shooting rockets at each other over a large room and it got me more excited than I had been since Wolfenstein 3D's chaingun audio.

We couldn't wait to see what players would do with our game, so we made sure it was open and available to modify all the data we had. We had hoped people would change textures, sounds, and make lots of new levels. We were enabling players to let us play their creations finally. It was a major move that would eventually end up with us releasing the source code. Open your game and your fans will own it, and keep it alive after you're gone.

For our small team, we took these huge changes in stride and tried to use them to the

edge of their capabilities. The technical stretches we made matched the design stretches we were exploring. I felt that we hit a lot of walls and climbed right over them. When Tom Hall left in August 1993, we quickly hired Sandy Petersen to help us in the final stretch. Dave Taylor came aboard to help us fill out the game.

At six developers, we were a tight team. Adrian and Kevin held down the art side confidently, while John Carmack handled the meat of the code. I loved being able to play with all of their output, and added a lot of my own code into the game's environments to support my level designs and Sandy's. When we were finished, we knew that we made something pretty great. We couldn't wait for everyone else to see it.

It's been an amazing 25 years, and I must first and foremost thank the fans that made it possible and kept it alive all this time as well as the game press who have always supported DOOM through its many iterations. Your appreciation of our work means everything. I also must thank John, Adrian, Tom, Sandy, Dave and Kevin. It was our crazy dream that made DOOM possible. Lastly, I want to thank the current DOOM team for their great work on the latest DOOM (I'm not at all involved in it, except as a player). Like everyone else, I am super excited for DOOM Eternal.

Then, here's to a quarter century of Rip and Tear!

Cheers,

— John Romero

Contents

Acknowledgments	5
Foreword by John Carmack	11
Foreword by Dave Taylor	13
Foreword by John Romero	15
Preface	25
1 Introduction	29
2 IBM PC	33
2.1 The Intel 486	39
2.1.1 Pipeline improvements	42
2.1.2 Caching	44
2.1.3 L1 Cache	46
2.1.4 Bus Burst Transfer	50
2.1.5 Overdrive and L1 Writeback	50
2.1.6 Die	51
2.1.7 Programming the 486	54
2.2 Video System	56
2.3 Hidden improvements	59
2.3.1 VGA Chip manufacturers	60
2.3.2 VL-Bus	62
2.4 Sound System	66
2.4.1 Sound Blaster 16	66
2.4.2 Gravis UltraSound	67
2.4.3 Roland	69
2.5 Network	71
2.5.1 Null-Modem Cable	72
2.5.2 BNC 10Base2 LAN (Local Area Network)	72
2.5.3 Modem	74

2.6	RAM	77
2.6.1	DOS/4GW Extender	78
2.7	Watcom	80
2.7.1	ANSI C	83
3	NeXT	85
3.1	History	85
3.2	The NeXT Computer	87
3.3	Line of Products	90
3.4	NeXTcube	91
3.5	NeXTstation	94
3.6	NeXTdimension	96
3.7	NeXTSTEP	102
3.7.1	GUI	103
3.8	NeXT at id Software	105
3.9	Roller coaster	108
3.9.1	Downfall	108
3.9.2	Rebirth	110
4	Team and Tools	111
4.1	Location	114
4.2	Creative direction	116
4.3	Graphic assets	117
4.3.1	Sprites	117
4.3.2	Weapons	122
4.3.3	Skies	125
4.4	Maps	130
4.4.1	Map Editor (DoomED)	132
4.5	Map Preprocessor (Node Builder)	135
4.6	Public Relations	139
4.7	Music	142
4.8	Sounds	142
4.9	Programming	143
4.9.1	Interface Builder, OOP and Objective-C	145
4.10	Distribution	148
4.10.1	WAD archives: Where's All the Data?	151
5	Software: idTech 1	155
5.1	Source Code	155
5.2	Architecture	156
5.2.1	Solving Endianness	157
5.2.2	Solving APIs	159
5.3	Diving In!	164
5.3.1	Where Is My Main?	165

5.4	Fixed Time Steps	168
5.5	Game Thread/Sound Thread	169
5.6	Fixed-point arithmetic	170
5.7	Zone Memory Manager	172
5.8	Filesystem	176
5.8.1	Lumps	178
5.9	Video Manager	182
5.10	Renderers	186
5.11	2D Renderers (Drawers)	187
5.11.1	Intermission	187
5.11.2	Status Bar	188
5.11.3	Menus	191
5.11.4	HUD (Head-Up Display)	192
5.11.5	Automap	192
5.11.6	Wipe	193
5.12	3D Renderer	197
5.12.1	Binary Space Partitioning: Theory	202
5.12.2	Binary Space Partitioning: Practice	206
5.12.3	Drawing Walls	208
5.12.4	Subpixel Accuracy	216
5.12.5	Perspective-Correct Texture Mapping	218
5.12.6	Drawing Flats	222
5.12.7	Drawing Flats (For Real)	229
5.12.8	Diminishing Lighting	230
5.12.9	Drawing Masked	236
5.12.10	Drawing Masked Player	243
5.12.11	Picture format	243
5.12.12	Sprite aspect ratio	244
5.13	Palette Effects	246
5.14	Input	248
5.15	Audio System	252
5.15.1	Audio Data: Formats and Lumps	255
5.16	Sound Propagation	258
5.17	Collision Detection	262
5.18	Artificial Intelligence	264
5.18.1	Optimization	271
5.19	Map Intelligence	272
5.20	Game Tics Architecture	276
5.21	Networking	278
5.21.1	Architecture	278
5.21.2	PC Network drivers	280
5.21.3	Implementation	281
5.21.4	DeathManager	284

5.22 Performance	286
5.22.1 Profiling	287
5.22.2 Profiling With A Profiler	288
5.22.3 DOS Optimizations	289
5.23 Performance Tuning	292
5.24 High/Low detail mode	292
5.25 3D Canvas size adjustment	294
6 Game Console Ports	297
6.1 Jaguar (1994)	298
6.1.1 Programming The Jaguar	302
6.1.2 Doom On Jaguar	304
6.2 Sega 32X (1994)	308
6.2.1 Doom On 32X	313
6.3 Super Nintendo (1995)	316
6.3.1 Argonaut Games	317
6.3.2 Doom On Super Nintendo	324
6.4 Playstation 1 (1995)	328
6.4.1 Doom on PlayStation	334
6.5 3DO (1996)	342
6.5.1 3DO Programming	346
6.5.2 Doom on 3DO	348
6.6 Saturn (1997)	352
6.6.1 Programming the Saturn	354
6.6.2 Doom on Saturn	358
Epilogue	363
Appendices	367
A Bugs	369
A.1 Bugs	369
A.1.1 Flawed collision detection	369
A.1.2 Slime trail	373
A.2 Barrel suicide	377
B Dots	379
B.1 Waiting for the Dots	379
B.2 Reload Hack	380
C NeXTstation TurboColor	383
C.1 Developing The Game	384
C.2 Compiling Maps	386
C.3 Running The Game	387

C.4	Framebuffer Non-distortion	388
D	Press Release	391
E	Source Code Release Notes	397
F	doombsp Release Note	401
G	Survivor's Strategies & Secrets	403
G.1	John Carmack	403
G.1.1	GOALS	403
G.1.2	IMPLEMENTATION	404
G.2	Sandy Petersen	406
G.2.1	How Does It Look?	407
G.2.2	Is It Fun?	408
G.2.3	Did You Remember to Clean Up?	408
G.3	Kevin Cloud	409
G.3.1	HAPPY DOOMING	410
H	Interview with Dave Taylor	411
H.1	Q & A	411
I	Interview with Randy Linden	415
J	OpenGL vs Direct3D .plan	423
K	Black Book Internals	429

Preface

This is the second *Game Engine Black Book*. It picks up right where the first one ended with the release of *Wolfenstein 3D* in May 1992. It carries on all the way up to December 1993 with id Software's second breakthrough of the 90s, *DOOM*.

Like its predecessor, this volume attempts to describe in great detail both the hardware and the software of the era. It opens a window back in time peeking over the engineering used to solve the various problems id Software encountered during the eleven months it took them to ship their next title.

It may seem odd to write a book about a game twenty-five years after its release. After all, who would be interested in seemingly outdated technology found in extinct hardware running obsolete operating systems? Given the success of the first *Black Book*, it turns out many. Whether readers are into history, nostalgia, engineering, or even philosophy, it seems there is an edge for everyone.

DOOM has had such a profound and sustained impact that it has become part of modern history. It is an unquestionable milestone that entertained millions and catalyzed vocations. Because the source code was made available, programmers have learned the game engine's architecture with it. Because it was easy to modify and the tools were available, countless aspiring game makers had their first experiences designing or drawing assets. To this day, because it is such an icon, it is often the go-to title for hackers wanting to demonstrate their skills¹. From the MacBook Touchbar, to ATMs, CT scanners, watches, and even fridges, pretty much any piece of electronics has run *DOOM*².

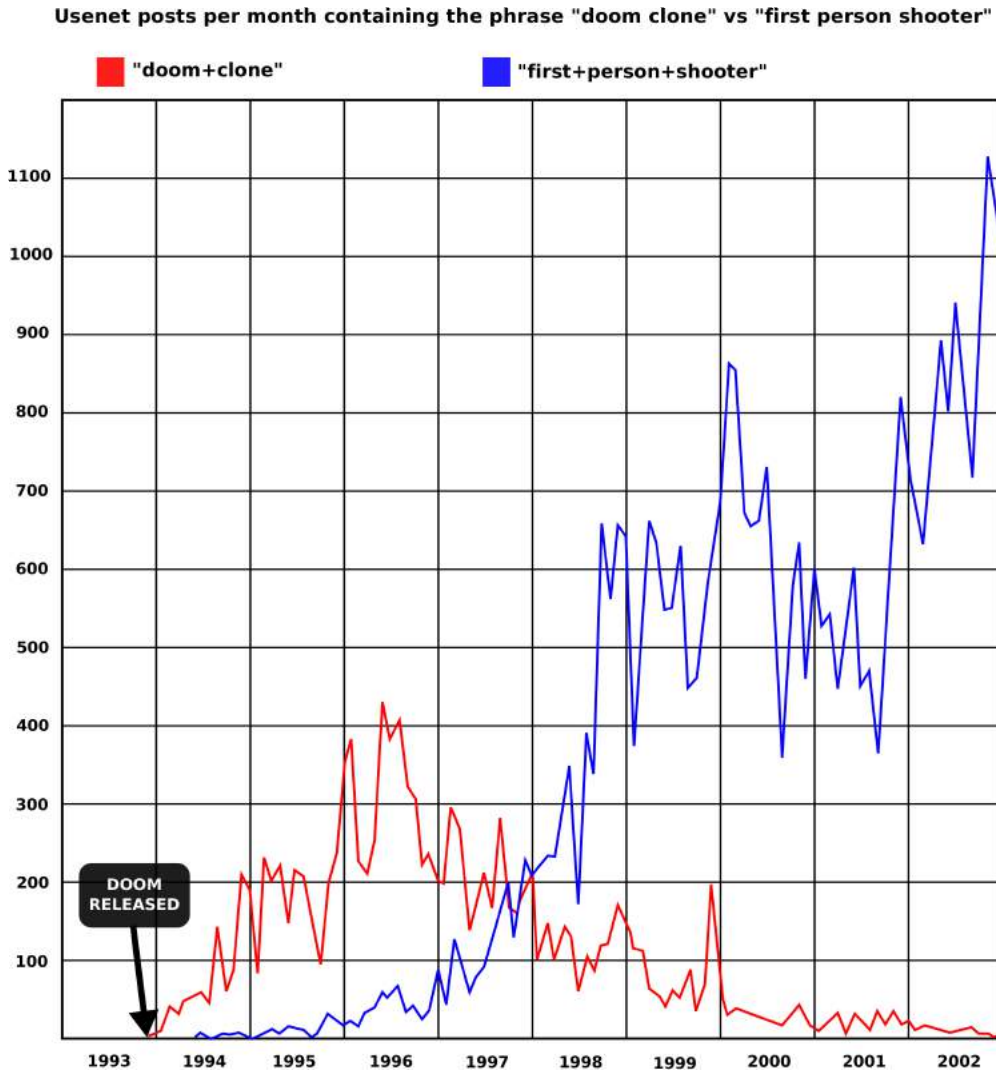
It was a financial and critical success that reshaped the PC gaming industry³. During 1994 it received several awards, including *Game of the Year* by both *PC Gamer* and *Computer Gaming World*, *Award for Technical Excellence* from *PC Magazine*, and *Best Action Ad-*

¹Upon cracking the BitFi wallet in Aug 2018, the hacker team demonstrated it by running *DOOM* on the device.

²"Will it run *DOOM*?" has become a common joke in the computer/gaming worlds. There is even a website, "itrunsdoom.tumblr.com", to provide the answer.

³And even killed the Amiga. Source: "Commodore: The Amiga Years" by Brian Bagnall.

venture Game Award from the Academy of Interactive Arts & Sciences. With more than two million copies sold and an estimated 20 million shareware installations, at its height the phenomenon generated close to \$100,000 per day. Before the term was overtaken by "First Person Shooter" people talked about "Doom clones".



There is also tremendous sentimental value. DOOM is one of those titles that made an ever-lasting impression upon first contact. Those who were able to play it upon release or shortly after are still able to remember the circumstance under which they first saw it running. It is an exhilarating feeling to learn the internals of something once deemed magical.

Beyond the nostalgia, and this is the most important reason this book is relevant, the making of DOOM is the ever-repeating story of inventors, engineers and builders gathered around a common vision. There was no clear path from where id Software was to where they wanted to be – only the certainty that nobody else had gone there before. They worked days and nights, slept on the floors, and waded across rivers to make their dreams come true.

The making of DOOM summarizes well how achieving a colossal task breaks down to a thousand small things done right. This is the story of a group of dreamers who combined skills, dedication and good fortune, resulting in a breathtaking combination of technology, artwork and design.

To narrate this wonderful adventure, the black book had to respond to two seemingly orthogonal constraints. On the one hand, to be able to stand alone without need for supplemental information or cross-references. On the other hand, to avoid boring faithful readers with content already visited in the series. The middle ground was to allow people who had read about Wolfenstein 3D to get more out of this book without making it a necessity.

Topics which would have been interesting to re-visit, such as the architecture of the VGA hardware, DOS TSRs, 386 Real-Mode, PC Speaker sound synthesis, the PIC and PIT, DDA algorithms and a few others are mentioned but not extensively described since they were part of *Game Engine Black Book: Wolfenstein 3D*. This trade-off allowed reaching the target, which was a book around 400 pages that can be held in one hand while sipping a cup of tea.

A few liberties were taken with regard to code samples. Due to the restricted real-estate of the paper version, code sometimes had to be slightly modified to fit. Other times, in order to introduce complexity progressively and not overwhelm the reader, portions of the code in functions were removed. Some code samples are from the original source code before it was cleaned up during the open sourcing effort so it may differ from what you can find on github.com. Rest assured the semantics and spirit remain intact.

This book is the fruit of an exercise inspired by Nicolas Boileau who reportedly stated: "Whatever we well understand, we express clearly". It is also the volume I wish someone else had written so I could just have purchased it (I am quite lazy).

I hope you will enjoy reading it!

– Fabien Sanglard (fabiansanglard.net@gmail.com)

Chapter 1

Introduction

In May of 1992, id Software was the rising star of the PC gaming industry. Wolfenstein 3D had established the First Person Shooter genre and sales of the sequel "Spear of Destiny" were skyrocketing¹. The game engine and the associated tools which had taken years to develop were far above the competition. They had an efficient game production pipeline and the talent to use it well with gorgeous levels and assets. Nobody even came close to challenging them... but for how long? They could have kept milking their technology but the evolution of hardware would have *doomed* them.

“

Because of the nature of Moore's law, anything that an extremely clever graphics programmer can do at one point can be replicated by a merely competent programmer some number of years later.

— John Carmack

”

Competitors were coming with their own games. Since its inception around the technological breakthrough named "Adaptive Tile Refresh", id Software's core value had been innovation. They had already released a sequel to Wolfenstein 3D and it was time to move on. The Right Thing to Do (and the most risky²) was to throw away everything they had worked so hard to build and start their next game from a blank sheet. Assets, levels, tools, and game engine – everything would be new and innovative.

Before getting started, id Software had to decide what hardware they would target, and then what tools to use. A summary assessment of the consumer landscape showed that

¹By the end of 1993, combined sales of Wolfenstein 3D and Spear of Destiny reached over 200,000 units. By the end of 1994, that figure increased to 300,000 units.

²Things You Should Never Do (Netscape 6 development), by Joel Spolsky.

PCs had significantly evolved since their last hit:

- Intel's latest CPU, the 486 announced in 1989, was finally becoming affordable. Providing twice the processing power of the previous generation, more and more customers now declined to go for an "old", twice as slow, Intel 386.
- The advent of Microsoft Windows 3.1 and its hungry GUI had prompted hardware manufacturers to offer more powerful graphic adapters. Rendering still had to be done in software but chipsets were faster and had more capacity.
- Frustrated with the bus bottleneck, vendors had teamed up to produce a new standard. PCs often came equipped with a bus ten times faster than the old legacy ISA, called VESA Local Bus (VLB/VL-Bus).
- The price of RAM was dropping significantly. The once-standard 2 MiB of RAM was now forecast to be 4 MiB..
- The audio ecosystem had become even more fragmented with many SoundBlaster "compatible" audio card clones on the market and also new innovative technology such as the Gravis Ultrasound's wavetable synthesis.

Not only had the hardware evolved, the software was also different. Better compilers such as Watcom allowed faster code to be generated. There was less need for time-consuming hand-crafted assembly, which was slowly becoming a thing of the past³. DOS extenders broke the machine free from 16-bit programming and its infamous limited 1 MiB address space.

On the developer hardware side, new options had appeared. Powerful workstations were now available and affordable to professionals. One company in particular, founded by Steve Jobs after his departure from Apple, combined strong hardware with efficient development tools. NeXT produced impressive machines running on their UNIX-based OS called NeXTSTEP.

In this whirlwind of novelties, it would have been easy to go in the wrong direction. Yet, id Software seems to have made all the right choices. How did they manage to start from nothing and make one of the best games of all time in just eleven months? This is the question this book will attempt to answer.

To do so, the two first chapters take a close look at the hardware of the time – not only the IBM PC on which DOOM ran but also the NeXT machines which id Software elected as the foundation of its production pipeline. The third chapter focuses on the team and the tools they wrote to bridge the hardware and the software. With all these capabilities and

³Intel would bring that trend back with its super-scalar processor, the Pentium, and make Quake development ASM intensive, but this is another story altogether.

restrictions in mind the last chapters are a deep dive into the game engine which hopefully will help the reader to appreciate why things are designed the way they are.

Now load your shotgun, pack a few medkits and let's dive!



Figure 1.1: "DOOM means two things: demons and shotguns!" – John Carmack

Chapter 2

IBM PC

The PC environment had morphed significantly between the development of Wolfenstein 3D in 1991 and the development of DOOM in 1993. The previous "recommended configuration" based on an Intel 386 with 2 MiB of RAM and a VGA graphics card was no more.

The "new" top of the line PC still had six subsystems: ① Inputs, ② Bus, ③ CPU, ④ RAM, ⑤ Video output, and ⑥ Audio output together forming a pipeline. Each of them had become faster or increased in capacity.

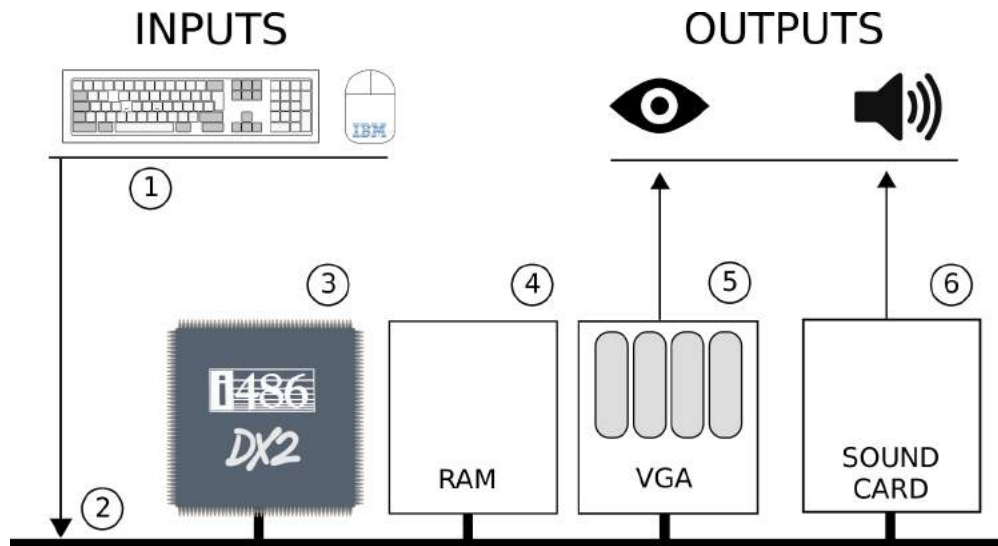


Figure 2.1: The six components of an IBM PC.

Before describing each component in detail, an important clarification is required. Since this chapter is structured as a delta comparing what was available for DOOM to what was available for Wolfenstein 3D, it carries a feeling of abounding power which is deceiving.

Despite the impressive list of improvements listed next, keep in mind that the IBM PCs were still not machines well suited for video games. They were riddled with limitations due to the original target, which was office work. They were designed to perform word processing, crunch spreadsheets and maybe occasionally display a static graph – the intent was never to build something allowing real time, 70Hz¹ animations.

Looking back it is hard to believe that studios focused solely on producing titles for a machine "obviously" less capable than consoles. The list of problems was substantial:

- A CPU unable to perform floating-point operations and no co-processors.
- An archaic graphic system seemingly unable to double-buffer and with an aspect ratio different from the monitor, resulting in distorted images.
- A de-facto sound system only capable of irritating "beeps" and a fragmented sound card ecosystem when the customer had elected to buy one.
- A price tag where a top of the line machine fetched close to \$3,000. To compare with the competition, the SNES and the SEGA Genesis were both priced at US\$199 while the Neo-Geo which provided arcade-like experience was priced at US\$649.99².

A PC was unappealing at best and seemingly less likely to generate good games, especially compared to cheaper systems which had been built with 60Hz animation in mind and benefited from sprite engines.

Obviously, given the title of the book in your hands, with a few software tricks the hardware of the PC was capable of far more than what it was designed for. PCs were not good at certain types of games but they could excel at certain types requiring a framebuffer. In a world without Internet and little documentation, it was far from a trivial challenge.

Figure 2.2 on the opposite page reproduces the kind of advertisement one could find in abundance in the many computer magazines of the 90s. Notice the featured IBM PS/1 with an Intel 486 CPU emphasizes office work and its ability to run static-screened office applications. Productivity and profit were the only way to justify the high price of a machine that represented 5% of the US median yearly income in 1993³.

¹VGA's most common refresh rate used for games (320x200) was 70Hz, contrary to today's ubiquitous 60Hz.

²Adjusted for inflation the figure would be, as of 2018: \$10,476 for a PC, \$377.00 for a SNES/Genesis, and \$1,134 for a Neo-Geo.

³"statista.com" lists 1993 US median income at \$52,335. Byte Magazine's Spring 1993 ads show 486 DX2-66 VESA PCs at \$2,575.

IBM **PS/1**
IMAGINATION SYSTEM



Figure 2.2: IBM PS/1 ad circa 1993. Notice the ridiculously small 14" CRT standard monitor allowing a resolution up to 800x600.

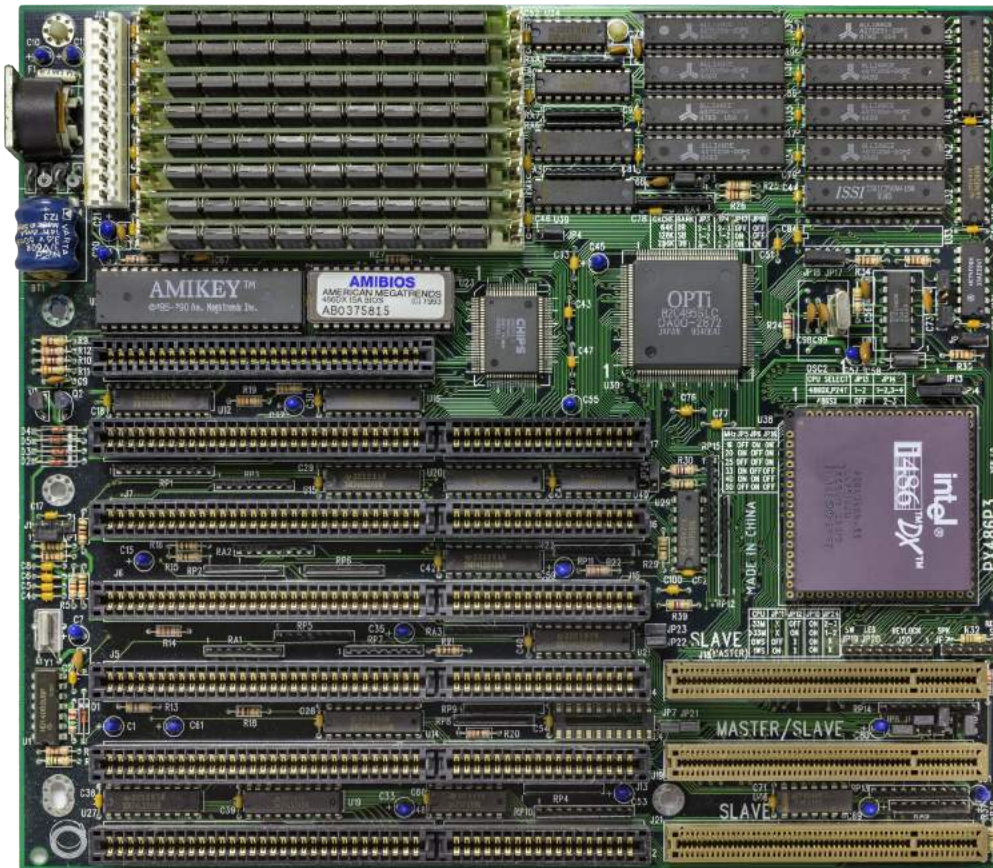


Figure 2.3: Motherboard PX486P3 by QDI Computer, Inc

A practical way to get an overview of the hardware available is to open up a PC and take a look at the component that connects everything together. In 1994, the best-selling motherboard was the PX486P3 by QDI Computer, Inc⁴.

The most prominent novelty is of course the heart of the computer, the Intel i486 CPU ①. A closer look reveals many more features which would turn out to be of paramount importance for the architecture of DOOM.

The black connectors show the traditional ISA bus expansion ports. One 8-bit ② and three dual-slot 16-bit ③ allow four ISA cards. Also present are three connectors of a new kind with an additional brown slot ④. These are VLB slots⁵, a bus up to 10x faster than ISA.

⁴A Canadian company 🇨🇦!

⁵a.k.a: VL-Bus, a.k.a: VESA Local Bus.

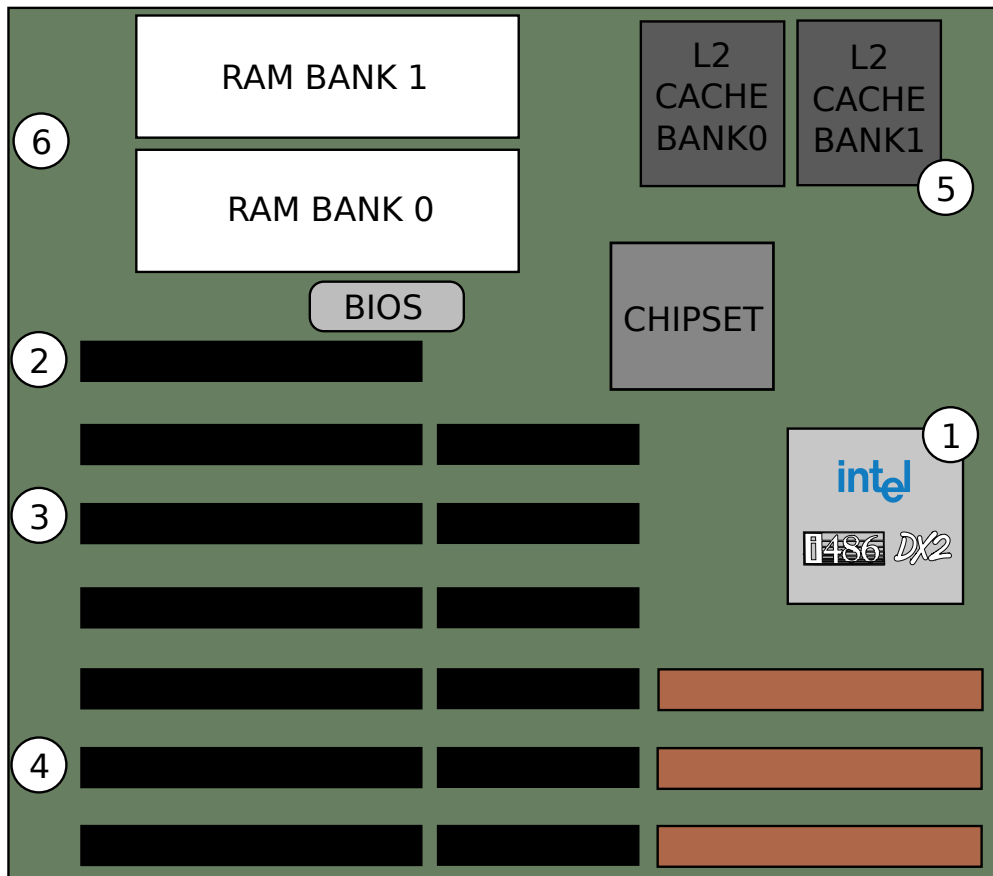


Figure 2.4: Component diagram of the PX486P3.

In the upper left ⑥, the main memory of the system had grown in capacity, speed and complexity. Thanks to a sharp decline in manufacturing price, the standard DRAM⁶ installed would be 4 MiB⁷.

Finally, in the upper right ⑤, a new type of RAM had found its way into these new PCs. Eight black chips of SRAM⁸ offered a total of 256 KiB acting as L2 "cache". Used in a new system designed to prevent CPU data and instruction starvation, the SRAM was much faster (access time of 20ns, which was 10x faster than DRAM) but had the double drawbacks of being far more expensive to produce and being less dense than DRAM, limiting its usability.

⁶Dynamic RAM.

⁷DOOM would not run on PCs equipped with only 2 MiB.

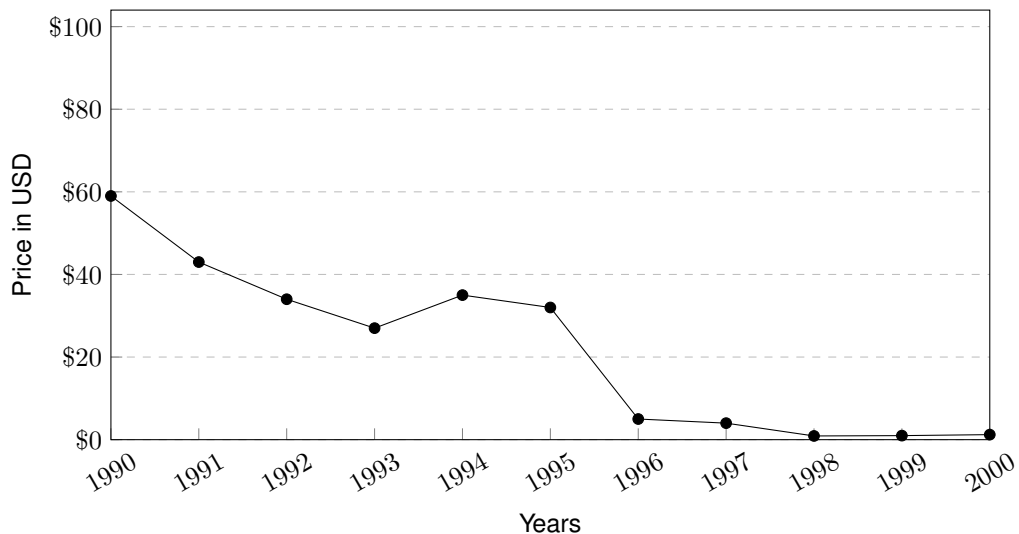
⁸Static RAM.

Trivia : Several of the most impressive game studios of the era speculated on the projected RAM price drop. The most impressive titles of 1994 (including DOOM) required a minimum of 4 MiB installed. Strike Commander, Ultima 8, and Comanche Maximum Overkill are prime examples.



Ironically this time period would end up coinciding with the great RAM shortage of 1994 which saw the price go back up. The legend attributes the surge to a resin factory burning down in Taiwan. In all likelihood the fluctuation was probably due to Microsoft's announcement of Windows 95, which recommended a machine with at least 8 MiB of RAM.

Average Price of 1 MiB of RAM 1990-2000⁹



2.1 The Intel 486

Announced in 1989, the 80486 was a performance evolution that addressed all the bottlenecks of the 80386. However, its price tag of \$950 (\$1,920 in 2018) kept it away from most consumers. By 1993 it was finally becoming affordable (\$500) and would become DOOM's recommended CPU.



The design had changed significantly compared to its predecessor. The pipeline was gifted with two extra stages, extending its depth to five elements. The FPU¹⁰, which used to be optional and somewhere on the motherboard was brought on-die. Most importantly, manufacturing improvements¹¹ allowed the 486 a more elaborate design that finally featured an integrated L1 cache – something Intel had attempted with the 386 without success.

“ The 386 actually had a small cache that eventually got exited because it didn't have enough performance for the size of the cache that we could put on board the chip. The problem was that if you made the chip bigger, it literally wouldn't fit inside the lithography machine's field of view, to flash on the chip.

— Gene Hill - Intel 386 Microprocessor Design and Development ”

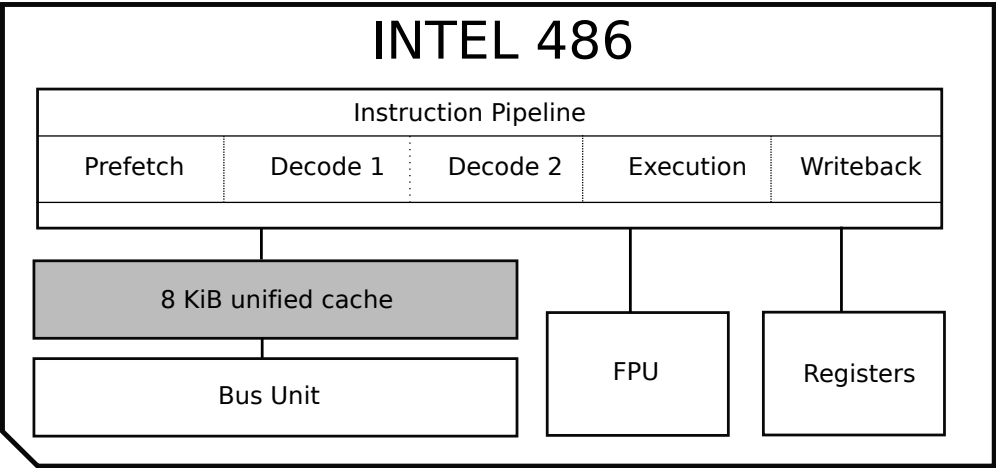


Figure 2.5: Intel 486 architecture

⁹Source: John C. McCallum survey.

¹⁰Floating Point Unit.

¹¹Manufacturing technology improved from 1.5 μ to 1.0 μ allowing five times more transistors on die.

Like with the 386, Intel marketed its new CPU in two flavors. The DX version was the pure technology, while the SX version had an unavailable FPU. A decades-old diehard myth is that the DX/SX distinction was a marketing stunt from Intel to sell chips coming out of the factory with malfunctioning parts due to manufacturing problems. It was in fact an intentional commercial operation¹² to provide a discounted (50%) price and an opportunity to sell i487 FPU co-processors.¹³

If in retrospect the 486 was Intel's 1994 champion and an unquestionable powerhouse (both in terms of performance and sales¹⁴), it had to sustain a period of uncertainty. Just as the 386 had to face its brother (the i960), the i486 also had to face a challenger from the same company. The competing sibling was named the "Intel 860".

“

...We now had two very powerful chips that we were introducing at just about the same time: the 486, largely based on CISC technology and compatible with all the PC software, and the i860, based on RISC technology, which was very fast but compatible with nothing. We didn't know what to do. So we introduced both, figuring we'd let



the marketplace decide. However, things were not that simple. Supporting a microprocessor architecture with all the necessary computer-related products — software, sales, and technical support — takes enormous resources. Even a company like Intel had to strain to do an adequate job with just one architecture. And now we had two different and competing efforts, each demanding more and more internal resources. Development projects have a tendency to want to grow like the proverbial mustard seed. The fight for resources and for marketing attention (for example, when meeting with the customer, which processor should we highlight) led to internal debates that were fierce enough to tear apart our microprocessor organization. Meanwhile, our equivocation caused our customers to wonder what Intel really stood for, the 486 or i860?

— Andy Grove, "Only the paranoid survive".

”

¹²Source: "Lies, Damn Lies, and Wikipedia" by Michal Necasek; The timeline did not make sense since "The 486DX started shipping in volume in late 1989. The 486SX was only introduced in mid-1991. In the first 18 months or so when yield problems would have been the worst, there was no SX."

¹³Amusingly, the i487 FPU upgrade was a full-blown 486DX that disabled the 486SX completely!

¹⁴As of late 2015, the 486 was still manufactured and used inside network routers.

On paper, the i860 was impressive and a serious opponent. Relying on a heavily pipelined super-scalar architecture crushing VLIWs¹⁵, it had three units (X, Y, and Z) allowing parallel processing and when used efficiently could outperform the Intel 486.

But whereas later CPUs such as the Pentium chose to hide the chip's complexity by automatically executing instructions in parallel when possible, the i860's architecture mandated direct manipulation of its parallel pipelines. The chip did nothing behind the scenes and relied on compiler writers to sequence instructions appropriately.

Unfortunately, compiler technology was not there yet. Without Intel's full backing to generate the precious tool, none of the compilers available came even remotely close to generating instructions able to exploit its super-scalar capability. The i860 was never able to reach its full potential. If only Intel had been willing to build the tools it desperately needed, the history of the i860 could have been different.



Figure 2.6: The Intel 80486 package

Figure 2.6 shows the Intel 486 die featuring 1,180,235 transistors inside its package. Around this time period, Intel started to stamp its CPU with a trademarked logo in an attempt to distance itself from increasingly aggressive AMD and Cyrix clones.

Trivia : The i860 played a part in DOOM anyway since it was used in the NeXTDimension's video processing boards.

¹⁵Very Long Instruction Word.

2.1.1 Pipeline improvements

Charting the 486's MIPS performance against the previous generation makes the performance boost appear vividly. Thanks to a better manufacturing process, top of the line 486s were able to run at 50Mhz¹⁶, but frequency increase was not the main source of the improvement.

Looking closely at the chart, one will notice that even at equal frequencies, a 486 offered more than twice the processing power of a 386.

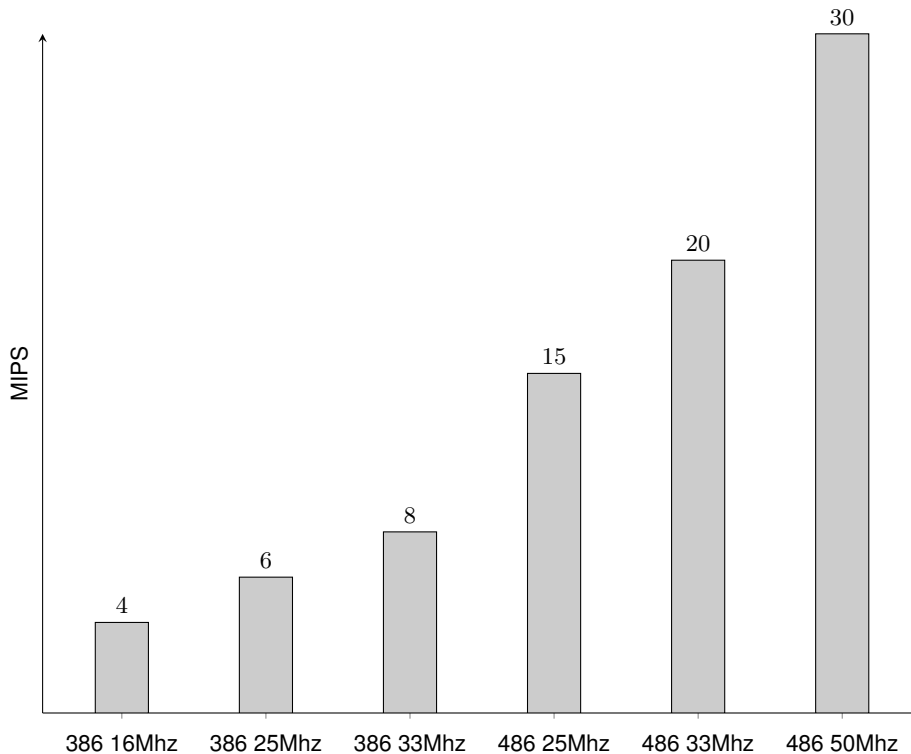


Figure 2.7: Comparison of Intel CPUs with MIPS ¹⁷.

The way it achieves higher performance is through a higher average throughput. According to its documentation, the 386 was a smooth three-stage pipelined processor. Under ideal conditions, figure 2.8 shows how it should have in theory been able to execute one instruction per cycle. In practice the CPU behaved as shown in figure 2.9, twice slower than suggested.

¹⁶To reach this frequency, 486 DX 50Mhz were manufactured at 1.0 μ .

¹⁷Source: "Roy Longbottom's PC Benchmark Collection: <http://www.roylongbottom.org.uk/mips.htm>".

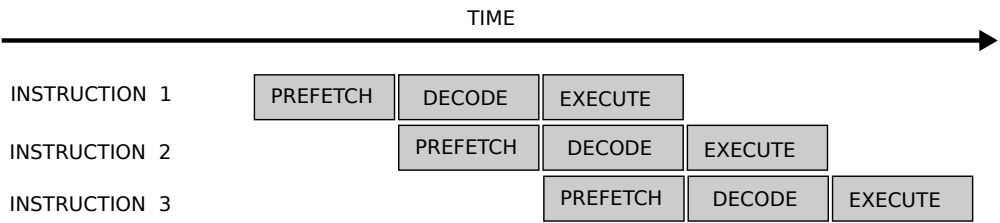


Figure 2.8: 386 pipeline in theory according to Intel documentation.

Even if the Prefetch Unit and the Execution Unit were properly fed, the Decode unit always took a minimum of two cycles to decode an instruction¹⁸. Since the maximum throughput of a pipeline cannot exceed the speed of its slowest stage, the Intel 386 could process at most one instruction every two cycles.

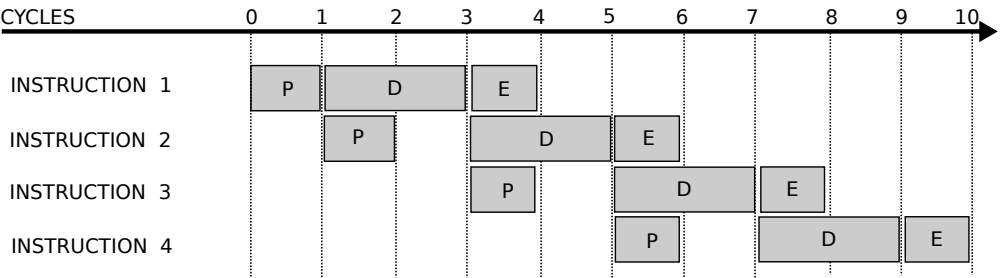


Figure 2.9: 386 pipeline in practice: Two cycles per instruction.

To solve this problem, Intel broke down the three stage pipeline into five (Prefetch, Decode1, Decode2, Execute, WriteBack). With all stages performing at 1 CPI¹⁹, the total throughput of the 486 was doubled (as long as the pipeline never starved).

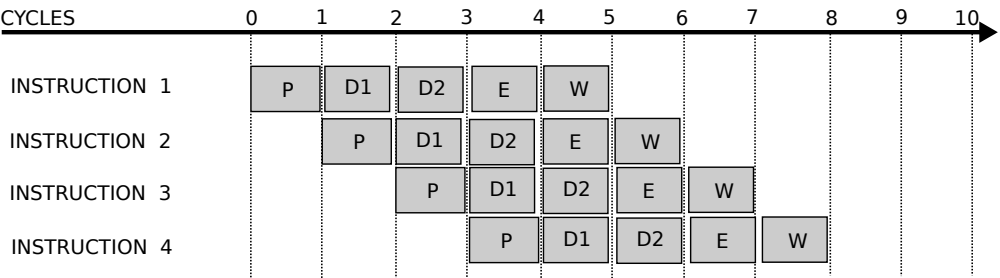


Figure 2.10: 486 pipeline: One cycle per instruction.

¹⁸Decoding is complex since x86 uses variable-length instructions, contrary to RISC's fixed-length approach.

¹⁹Cycle Per Instruction.

2.1.2 Caching

Modifying the pipeline and making each stage run as fast as the others was one step in the right direction. But making the pipeline deeper also made it more vulnerable to starvation. Starting from an empty pipeline, the 486 had a latency of 5 cycles compared to the 386 which had only three stages. A frequently stalling 486 would have been slower than a 386. Halting processing due to missing data or instructions was to be avoided at all cost.

It was a difficult constraint to fulfill for physical reasons. Since 1980, RAM performance had been lagging behind CPU performance. Each year, CPUs performance improved by 60% while DRAM had only improved by 7%, the gap increasing by 50%/year. By 1989, DRAM access time was 10 times slower than CPU cycle time.

PERFORMANCE

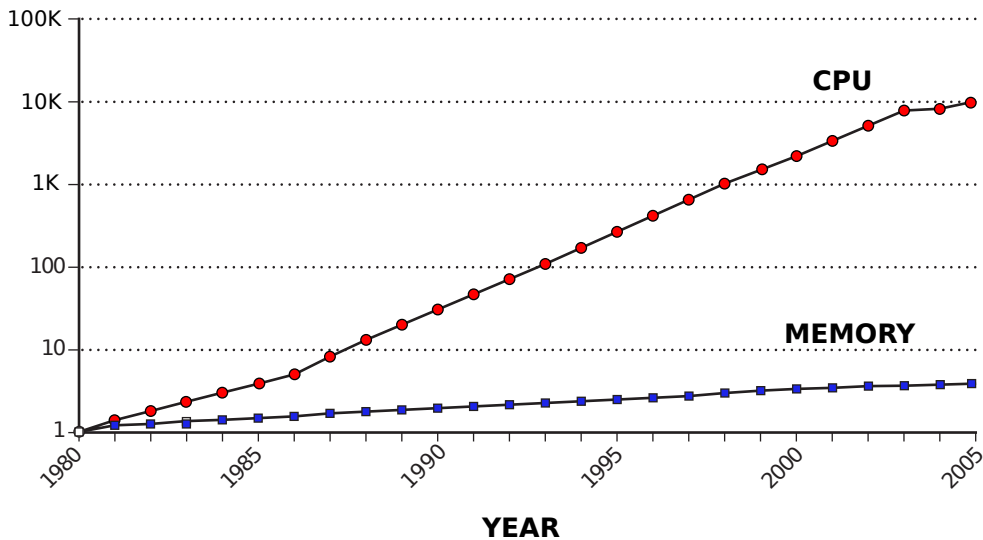


Figure 2.11: Source: "Computer Organization and Design" by Hennessy and Patterson

Until the 486, a CPU requesting either instructions or data from DRAM always had to stall and go through its Bus Unit to talk to the motherboard memory controller. As optimized as the ISA bus protocol was, it took at the very minimum two cycles.

A first cycle initialized the bus request, placed the address on the address line and set the control line (Read/Write). Then, a wait cycle ran (which Intel called Wait State since while waiting on the Bus Unit, the CPU did absolutely nothing) while the device on the other side of the bus fulfilled the request.

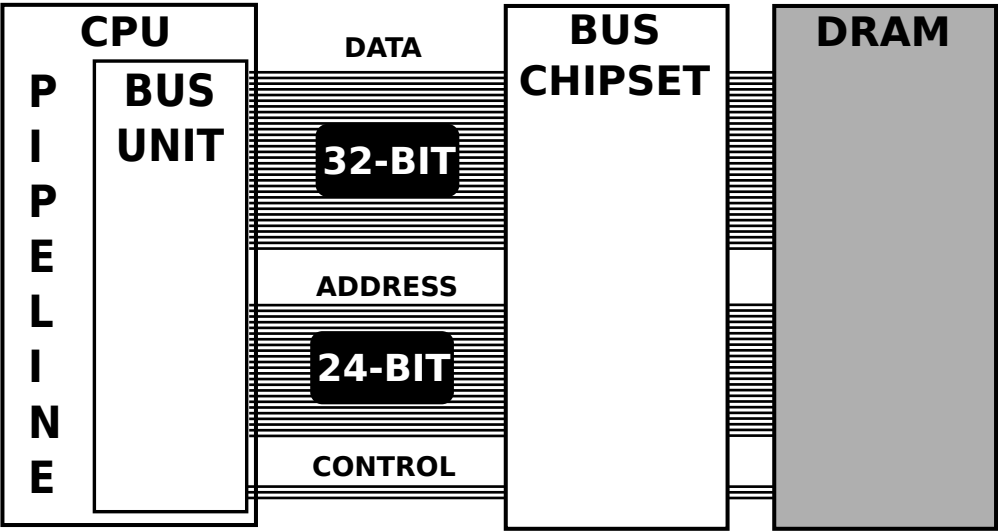
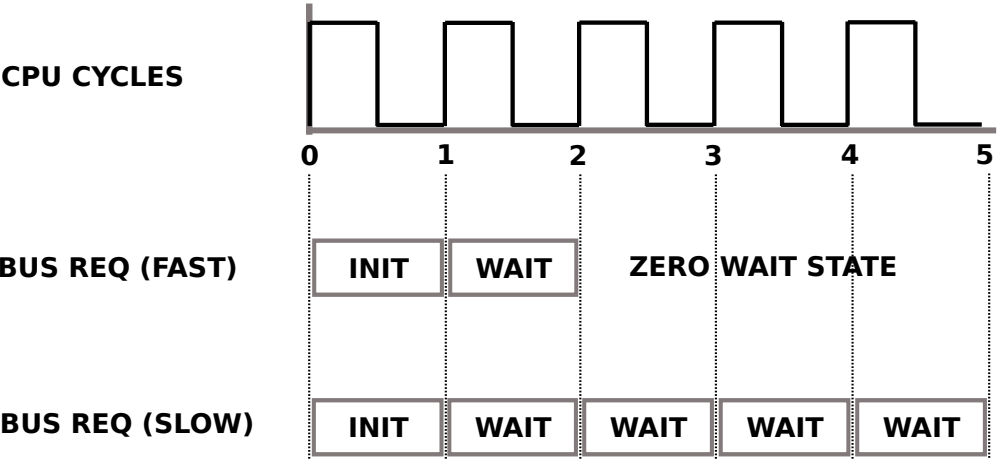


Figure 2.12: 386 CPU-RAM communication elements

If a device was able to answer the bus request within the first wait state cycle, the CPU was able to resume operations having reached Zero Wait state. Otherwise, additional Wait States were inserted in order to wait for the request to complete. From a performance perspective, these Wait States were a disaster. Not only because it took longer for the instruction to finish but also because it stalled all other instructions in the pipeline.



A two cycle bus request was the fastest a CPU could achieve. In practice, DRAM access required several Wait State insertions. To avoid this meant avoiding using the bus com-

pletely. Therefore, Intel inserted a new component between the pipeline and the bus unit called the L1 (Level 1) cache. The idea was to exploit both the spatial and temporal locality of a program.

Temporal locality relies on the iterative nature of programs. While in a loop, a recently accessed instruction is likely to be accessed again on the next iteration. Spatial locality has to do with the way programs sequentially read or write arrays containing data. If a memory address is accessed, it is likely a neighboring address will also be accessed shortly after.

By leveraging these two properties, a well-designed cache located between the CPU and the Bus Unit would often already contain the requested data or instruction, making a bus request unnecessary.

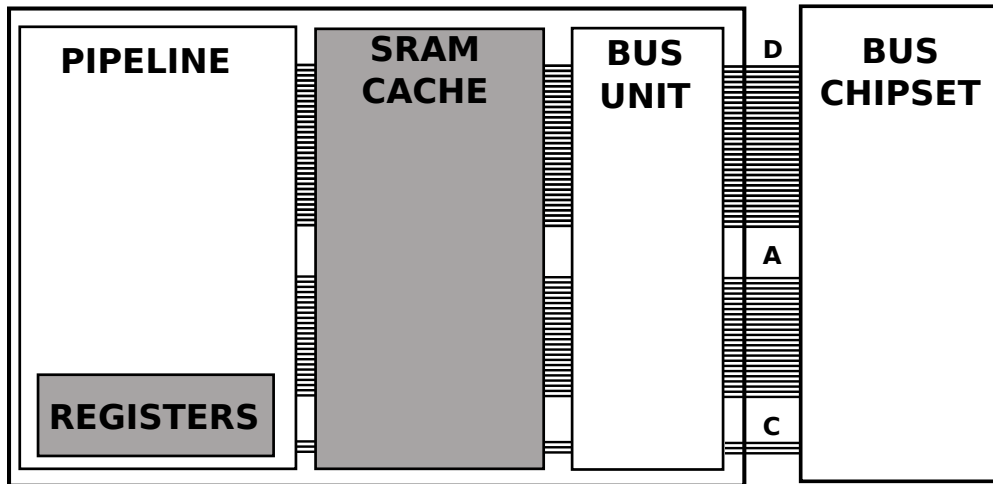


Figure 2.13: 486 CPU-RAM communication elements

2.1.3 L1 Cache

Hopefully it is now abundantly clear that the cache was the cornerstone of the entire CPU. Designing the cache to yield the highest hit rate possible and making it as fast as possible were paramount.

2.1.3.1 DRAM vs SRAM

The first thing the cache had going for itself was the lower latency of its RAM. While the main RAM in the SIMM slots used DRAM (Dynamic RAM), the cache was made of a different type called SRAM (Static RAM), with a much faster access time. DRAM typically had an access time of 200ns while SRAM was capable of 20ns, 10x faster.

The difference in speed comes from the design of their elementary cells.

A DRAM cell holds a single bit. Its simple design features one transistor and one storage capacitor which allow tight packing and high capacity. However, the capacitor loses its charge over time and each time it is accessed. Every time the cell is read, it must be written back with its value. Even if it is not accessed, it must be refreshed every $15\mu s$.

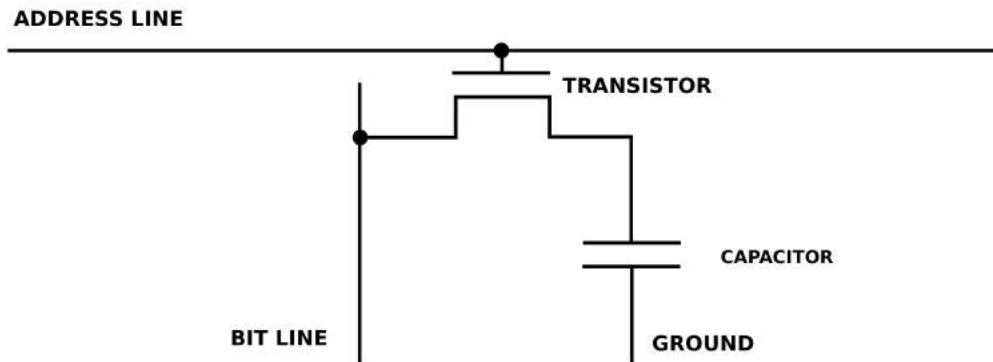


Figure 2.14: Dynamic RAM and its two elements holding one bit of data

The slowness comes from the high maintenance cost of each cell. The DRAM also has the disadvantage of being far away. Located somewhere on the motherboard, it requires using the ISA bus which is shared with other devices.

Thanks to a more elaborate design (which made it less dense but more expensive to manufacture), an SRAM cell has none of these disadvantages.

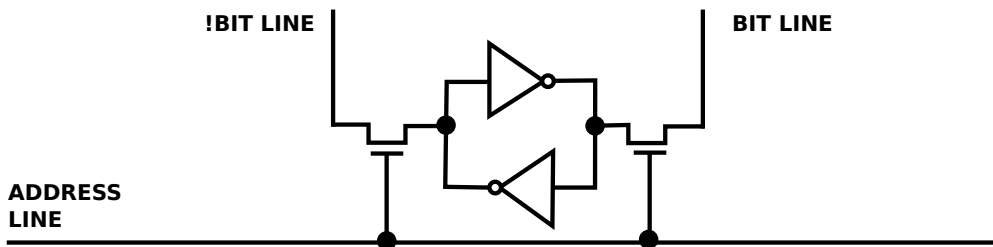


Figure 2.15: Static RAM made of six elements

Without a capacitor, an SRAM cell doesn't leak. It does not need a periodic refresh and it does not need to be written back each time it is accessed. Its two bit lines allow twice as fast voltage variation detection and faster timing. Since it is located inside the chip, accessing it doesn't require an expensive bus request and there is no contention with

other devices²⁰

2.1.3.2 Cachelines

Not only did the L1 cache have better hardware, it was also cleverly designed. Its small size (8 KiB) and heavy duty (unified cache for both code and data) placed a considerable stress on it, yet it managed an impressive 92% hit rate²¹ under normal operation.

To achieve this, Intel engineers used a four-way associative design where the 2^{32} address space is divided into 2,097,152 pages of 2 KiB. Within each page there are 128 lines of 16 bytes (called cachelines).



Figure 2.16: The 16 bytes in a cacheline.

The cache system is made of one directory and four banks (also called ways). Each way can store 128 cachelines of 16 bytes and therefore has a capacity of 2 KiB. These lines of 16 bytes are the elementary units of the cache.

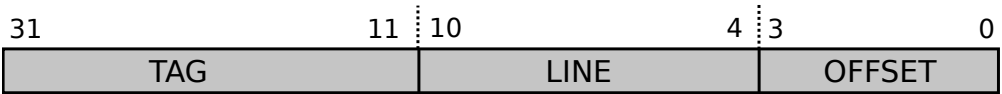


Figure 2.17: How a memory address is interpreted by the cache controller

Upon receiving a 32-bit address access request, the cache controller splits it into three fields.

- 1. Use the LINE field [4-10] to look up one of the 128 dictionary entries.
- 2. Look at the four tags in the entry. If one matches the TAG [11-31] then it means the cacheline is present in one of the four ways.
- 3. Check the flag F in the directory entry to make sure the cacheline is valid.
- 4. Use the OFFSET [0-3] field to access one of the 16 values in the cacheline.
- 5. Update the flag F in the directory entry to update the LRU value.

A memory address' content can be in any of the four ways but always at the same LINE offset. With $2^{32}/128 = 33,554,432$ addresses competing for four slots, the unavoidable

²⁰DRAM speed improved over the years. Fast Page Mode "cached" rows of DRAM cells with an SDRAM row buffer. udacity.com's UPCF courses are excellent if you want to learn more about this topic.
²¹Source: "The i486 CPU: Executing Instructions in One Clock Cycle".

cacheline evictions are arbitrated via an LRU²² policy²³.

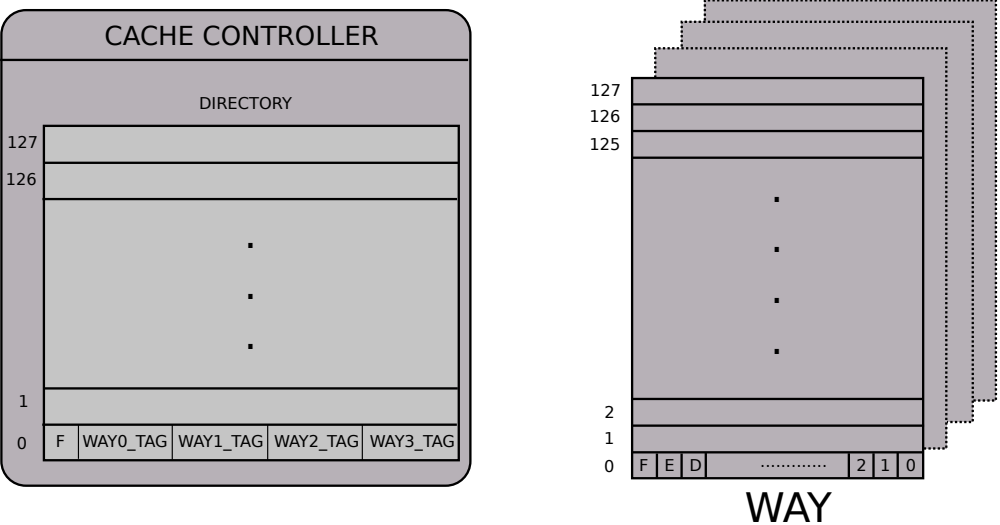
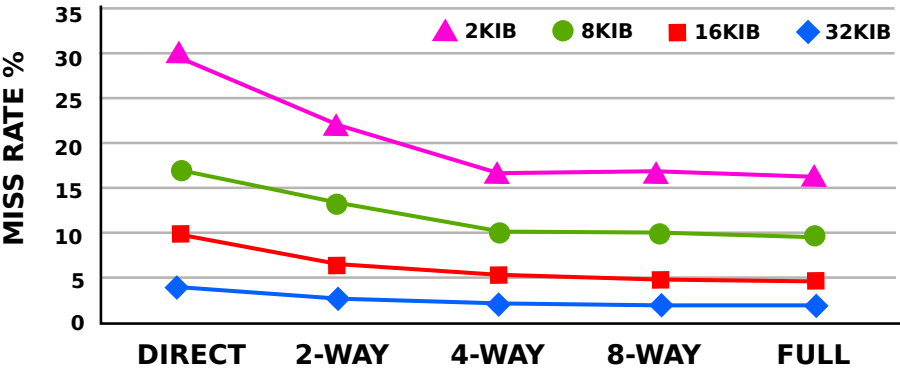


Figure 2.18: The cache controller and its four ways (banks).

Trivia : What about increasing the number of ways or the cache size? With 8 KiB of cache, four ways grant the best trade-off²⁴. A two-ways cache yields a 14% miss rate and a 4-ways cache yields a 10.5% miss rate. However, going up to eight only improves the percentage to 10%, and fully associative to 9%.



²²Least Recently Used.
²³Eviction can happen on read but also on write if the cache is write-allocate, which all Intel 486s were (Source: "Internal Cache Architecture of X86 Processors").
²⁴Source: "Computer Architecture: A Quantitative Approach" by Hennessy/Patterson.

2.1.4 Bus Burst Transfer

Any cache miss within the 486 pipeline triggered the eviction of a cacheline and a full 16 bytes had to be transferred from DRAM to SRAM²⁵. Normally this would have been a very costly operation and a huge issue for the CPU. But Intel added something called "Burst Transfer" capability to make it all work together.

The principle was simple: While waiting for data to arrive, latch the next request so the bus controller can use it right away without waiting for the CPU to initialize a bus request.

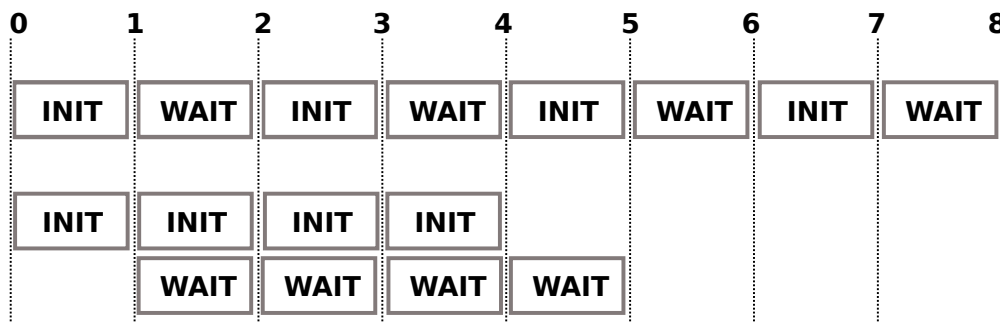


Figure 2.19: "Burst Transfer" allows for 65% faster cacheline filling.

2.1.5 Overdrive and L1 Writeback

Intel managed to improve performance by 33% with its line of 80486 OverDrive chips. These CPUs featured a frequency multiplier that made them run two times faster than the bus (the 33Mhz model CPU ran at 66Mhz)²⁶. Furthermore, the L1 cache policy became write-back (instead of write-through) which reduced bus traffic significantly.



Figure 2.20: The "DX2-66" was the golden standard and absolute best to run DOOM

²⁵The prefetcher also worked with units of 16 bytes. It retrieved and stored cachelines into a prefetch queue of 32 bytes.

²⁶To this day, designers still try to solve the problem of having a CPU so much faster than the bus.

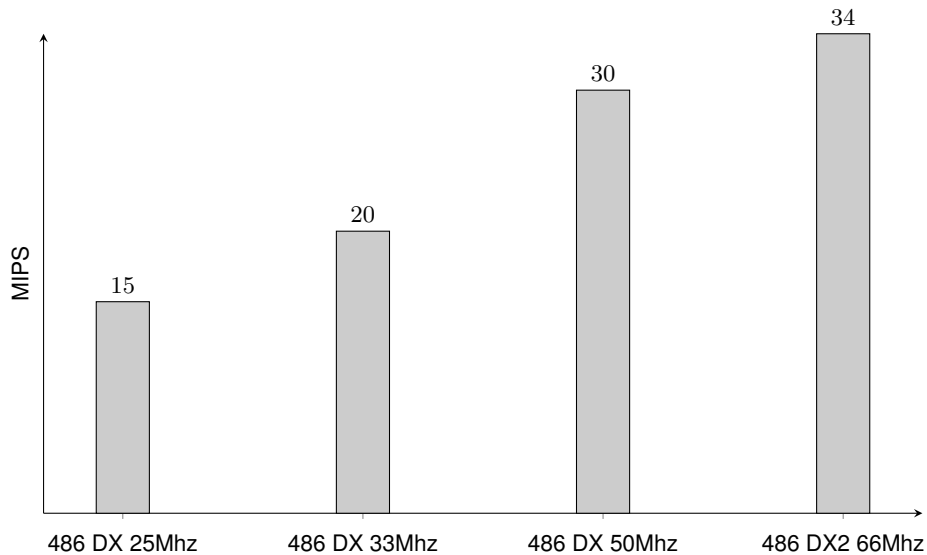
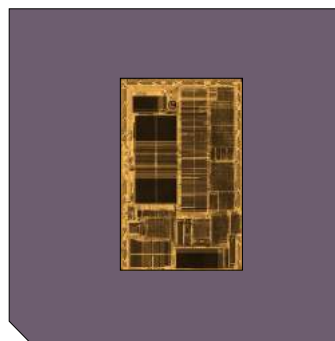


Figure 2.21: Comparison of CPUs with MIPS ²⁷.

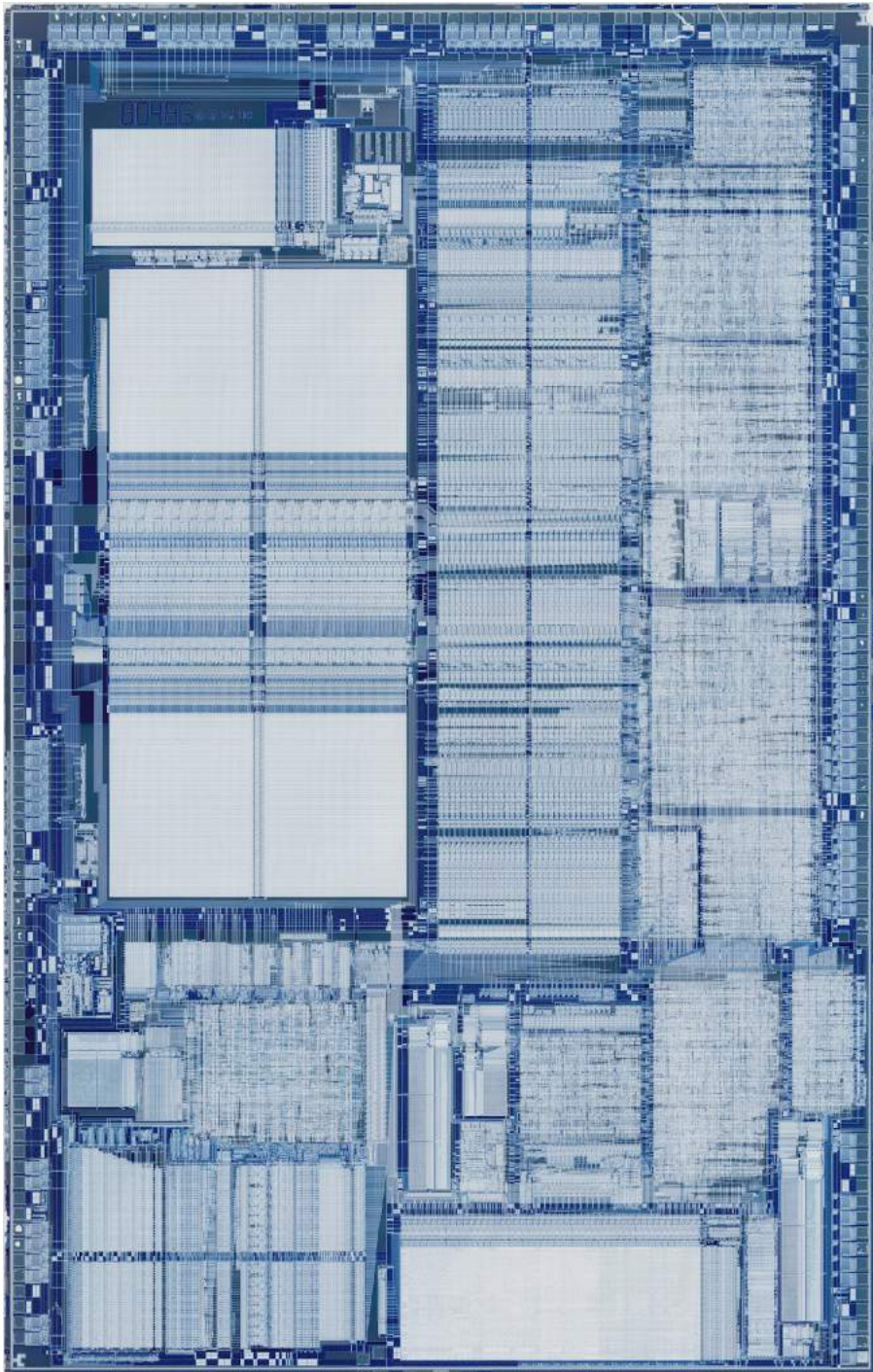
On the chart above, notice how a 486DX2-66Mhz is faster than a 486DX-50Mhz but not by the full 20% that frequency would make us expect. This is because the DX2 bus runs at 33Mhz while on the DX, both the CPU and bus run at 50Mhz.

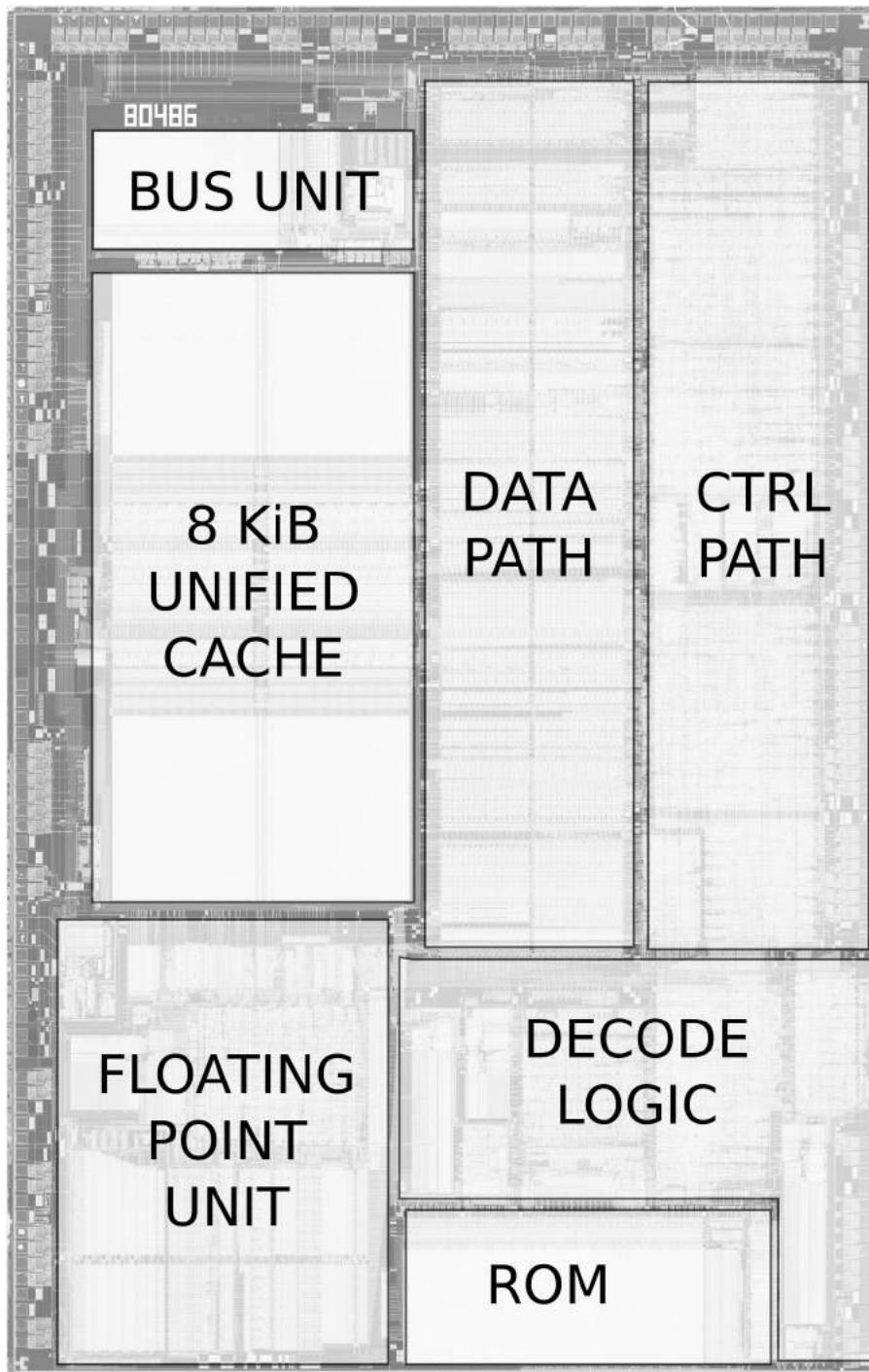
2.1.6 Die

If you are holding a physical 9.25"x7.5" copy of this book, the CPU packaging is 30mm square and the die is 15.5 x 9.9 mm, both represented at 1:1 scale.



²⁷Source: "Roy Longbottom's PC Benchmark Collection: <http://www.roylongbottom.org.uk/mips.htm>".





Trivia : On the previous page, the transistor layout differs between the data path, which was hand-crafted, and the control path, which was created with CAD tools built especially for the 486²⁸.

2.1.7 Programming the 486

With the architecture in mind we can now understand how a programmer could take best advantage of the 486. The good news was that most of the performance improvement was the characterized free-lunch of the 90s. The exact same binary would run twice as fast on the new CPU.

Apart from a few peculiarities²⁹, as long as the programmer was mindful of the cache-lines and maximized time and space locality³⁰, the CPU would fly in processing integers. Floating-point operations had improved by a factor of 2 compared to the i386's FPU (i387). In fact, floating point operations had improved so much that the i487 FPU could FMUL faster than the i386 could IMUL.

CPU	FADD	FMUL	FDIV	FXCH
Intel 387	23-34	29-57	88-91	18
Intel 487	8-20	16	73	4

Figure 2.22: FPU cycles per instruction: 387 vs 487.

But the FPU performance was still a far cry compared to the ALU and its barrel shifter. This mandated DOOM to use integer operations exclusively³¹.

CPU	ADD	MUL	DIV
i487 (FPU)	8-20	16	73
i486 (ALU)	1	12-42	43

Figure 2.23: Cycles per instruction: ALU vs FPU.

Trivia : Difficulty in accessing information birthed myths about DOOM and floating point units. One endless thread that occurred on `alt.games.doom` in 1994 helps to appreciate the state of things. The topic, "Does a 486DX run Doom faster than an SX?" from July 1994 and its (filtered) five answers shows how difficult it was to reach the truth.

²⁸Source: Coping with the Complexity of Microprocessor Design at Intel – A CAD History.

²⁹"Pushing the 486" by Michael Abrash.

³⁰And avoided branching. Without a branch predictor, `jmps` are ignored and usually incur a two cycle stall.

³¹The dawn of floating-point in games would begin with Intel's Pentium and Quake in 1996.

“ My friend is buying a computer and doesn't see much reason to buy a DX. Any opinions? (or hard facts :) ?).

— Dave Gates@bestsd.sdsu.edu - 23 Jul 1994 05:28 ”

“ DOOM runs *MUCH* faster on a 486DX/33 than on a 486SX/33. Believe me, I've seen it running on the 2 different machines in the same room.

— BillyBoB 4@aol.com - 23 Jul 1994 10:45 ”

“ They are *NOT* any different as far as CPU speed go. Period. The reason one (DX) is faster must have something to do with probably the SX has an ISA video card, or no cache, or less memory. Doom does NOT NOT NOT NOT use an FPU (math co-processor) so there will be no slowdown for the SX.

— Chad Anson@daisy.cc.utexas.edu - 23 Jul 1994 11:48 ”

“ We have a 486SX/25 and a 486DX/50 and the DX 50 runs faster at full screen high detail then the SX runs at postage stamp. It is so slow as to be almost unplayable.

— BonesBro@aol.com - 23 Jul 1994 12:36 ”

“ An SX is considerably slower than a DX for most processor-intensive applications and games, including DOOM.

— Neal W.Miller@@rebecca.its.rpi.edu - 23 Jul 1994 13:34 ”

“ Thats wrong ! A 486 SX runs Doom with exactly the same speed like a 486 DX (if you use the same VGA Card and Motherboard).

— Grassl Wolfgang@papin.HRZ.Uni-Marburg.DE - 23 Jul 1994 14:24 ”

2.2 Video System

At first glance, the video output system, the VGA (Video Graphic Array), was still the same weird beast Wolfenstein 3D had to deal with. With its infamous 50 registers to configure, its palette system limiting colors to 256, and the awkward four banks of 64 KiB mandating interleaved framebuffers, the VGA was an unappealing programming interface.

A GC (Graphic Controller) and an SC (Sequence Controller) controlled access to 256 KiB of Video RAM. A CRTC (Cathode Ray Tube Controller) controlled how the framebuffer was sampled. Finally, a DAC converted digital levels to analog levels for output to a CRT.

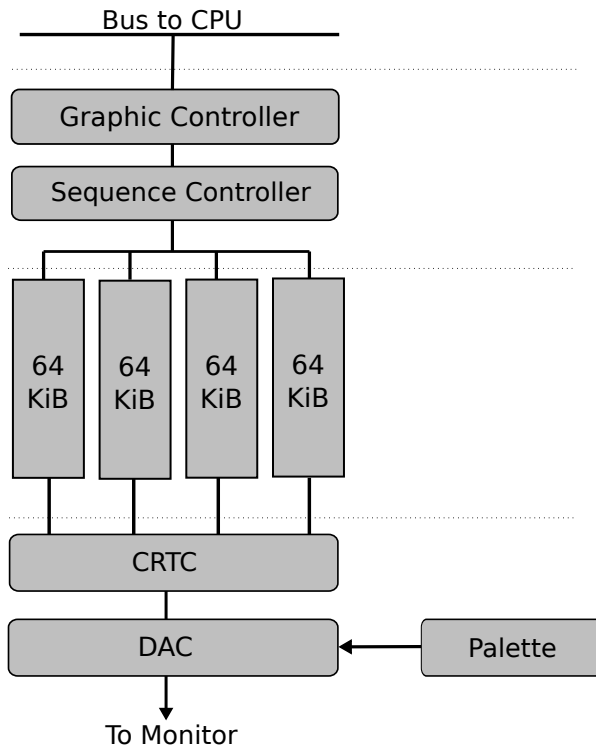
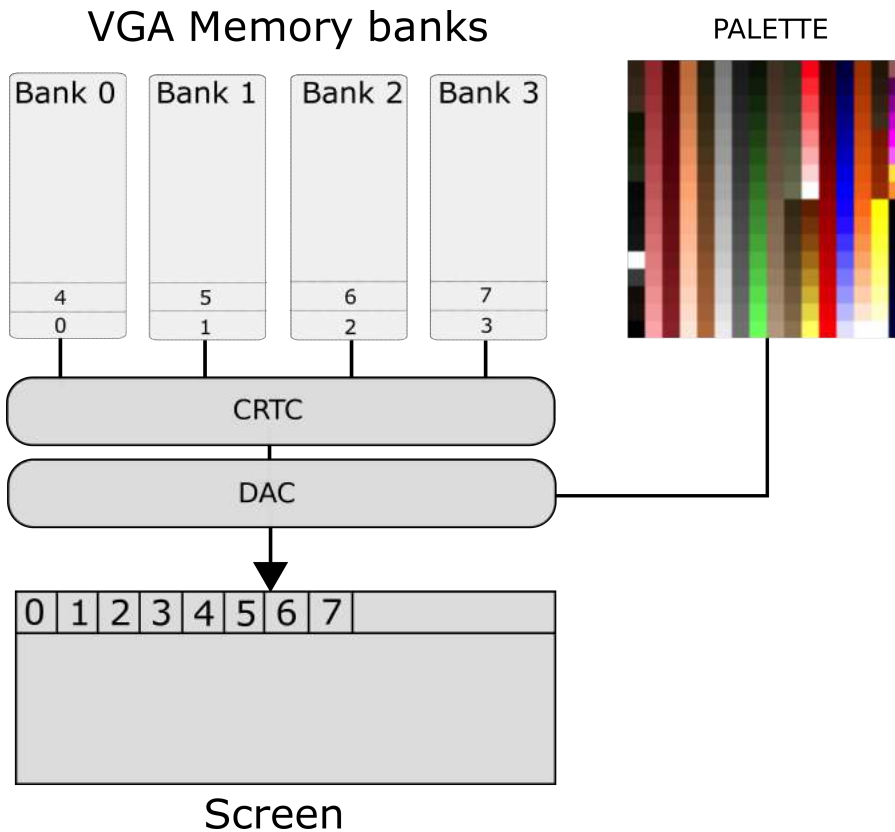


Figure 2.24: Architecture of the VGA

During the early 90s, a vast majority of PC video games used the VGA in tweaked mode 13h (also called Mode-Y) which offered a resolution of 320x200, 1 byte per pixel and a 256 color palette. Using this undocumented mode, developers manipulated the four banks in the video card directly. How the framebuffer was laid out across banks was far from obvious. As figure 2.25 shows, due to historically slow RAM access times, pixels were interleaved four by four.

**Figure 2.25**

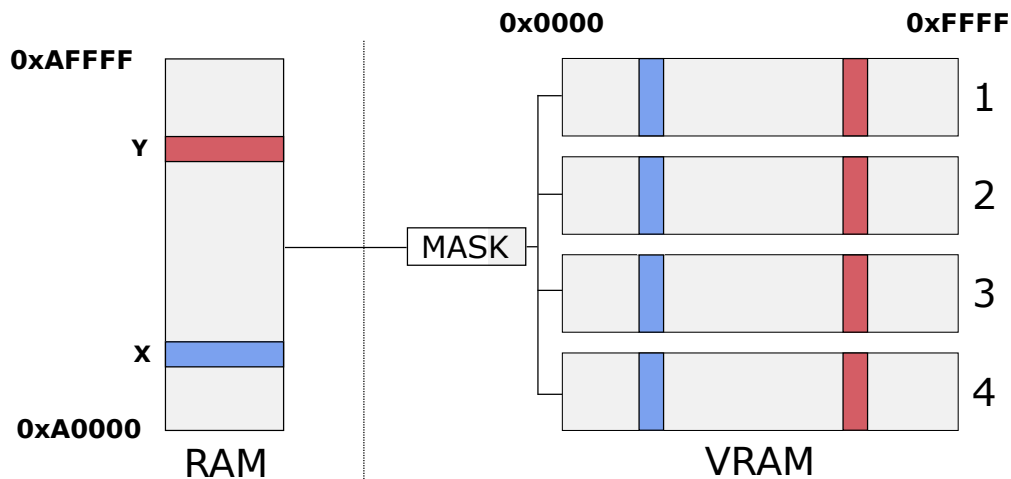
Notice how the first pixel 0 is stored in bank 0, pixel 1 is stored in bank 1 and so on. With a horizontal resolution of 320 columns, 80 non-horizontally adjacent yet vertically adjacent pixels are stored in each bank³².

To access the 256 KiB of VRAM, IBM had established a hard-coded memory mapping in RAM from 0xA0000 to 0xAFFFF. An eye accustomed to hexadecimal will immediately notice that 0xFFFF translates to 64KiB addresses, far fewer than the total available VRAM.

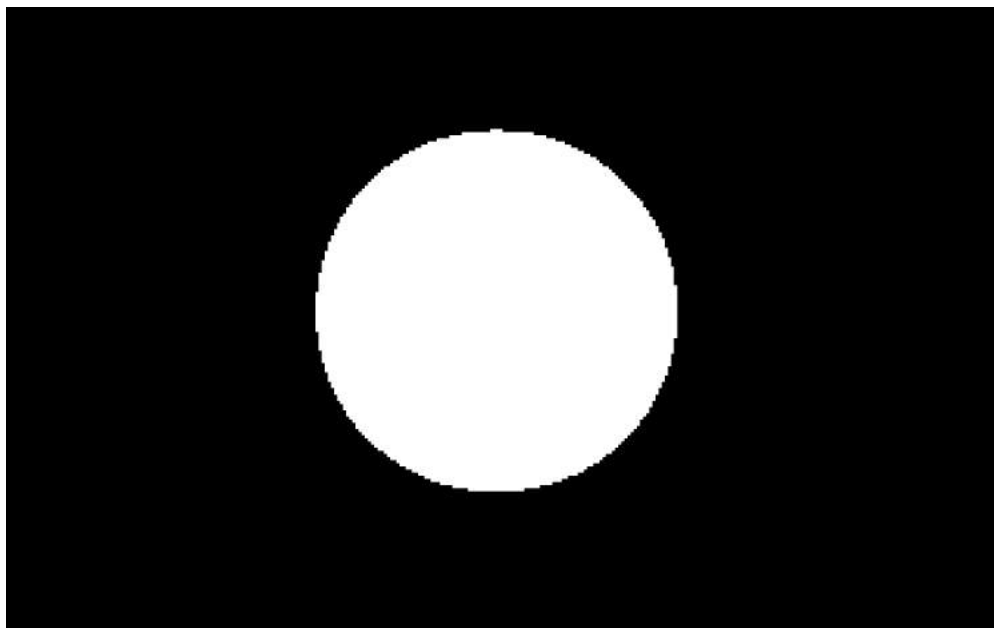
To compensate for the lack of addresses, IBM designed a bank-switching system managed through a *map mask register*. In practice this meant a RAM address in the range 0xA0000 to 0xAFFFF could correspond to four locations in VRAM as shown in figure 2.26. The mask was cumbersome but allowed magic, such as writing four pixels in one write operation³³.

³²The design of the VGA are a huge topic covered extensively in *Game Engine Black Book: Wolfenstein 3D*.

³³VGA mask tricks are discussed in *Game Engine Black Book: Wolfenstein 3D*.

**Figure 2.26**

Another difficulty came from the differing aspect ratios of the mode Y framebuffer layout and the CRT display which resulted in distortion.

**Figure 2.27**

In figure 2.27 a programmer drew a circle into the framebuffer; notice the $320/200 = 1.6$ aspect ratio.

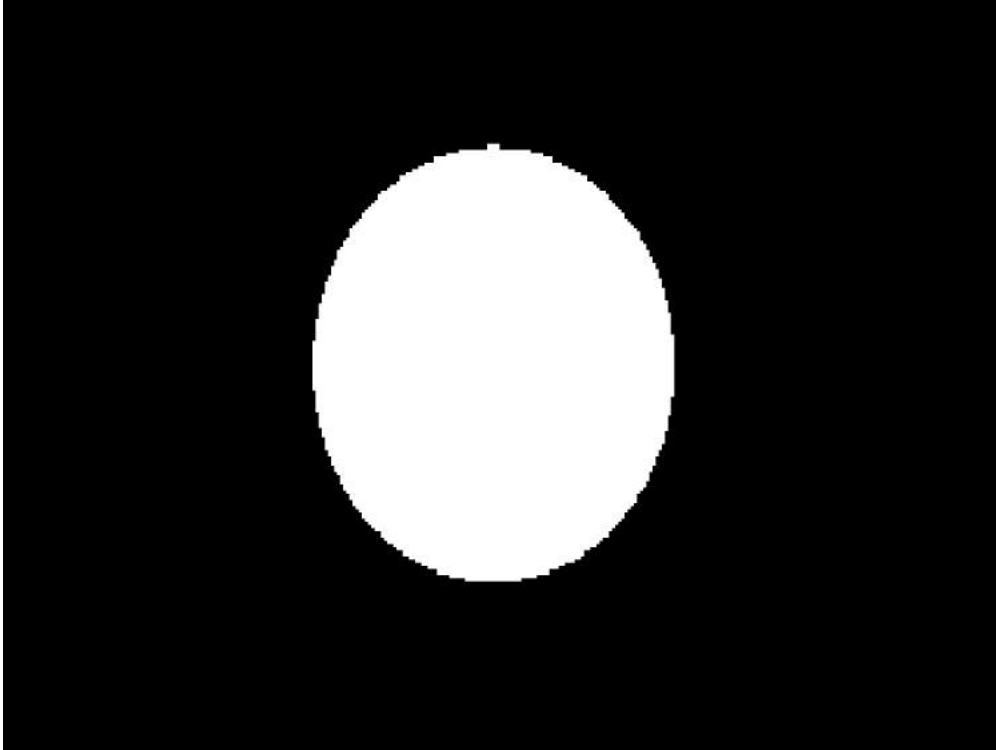


Figure 2.28

Figure 2.28 shows how the same framebuffer appears when displayed on the monitor. Notice the $320/240 = 1.333$ aspect ratio. The circle appears as an ellipse.

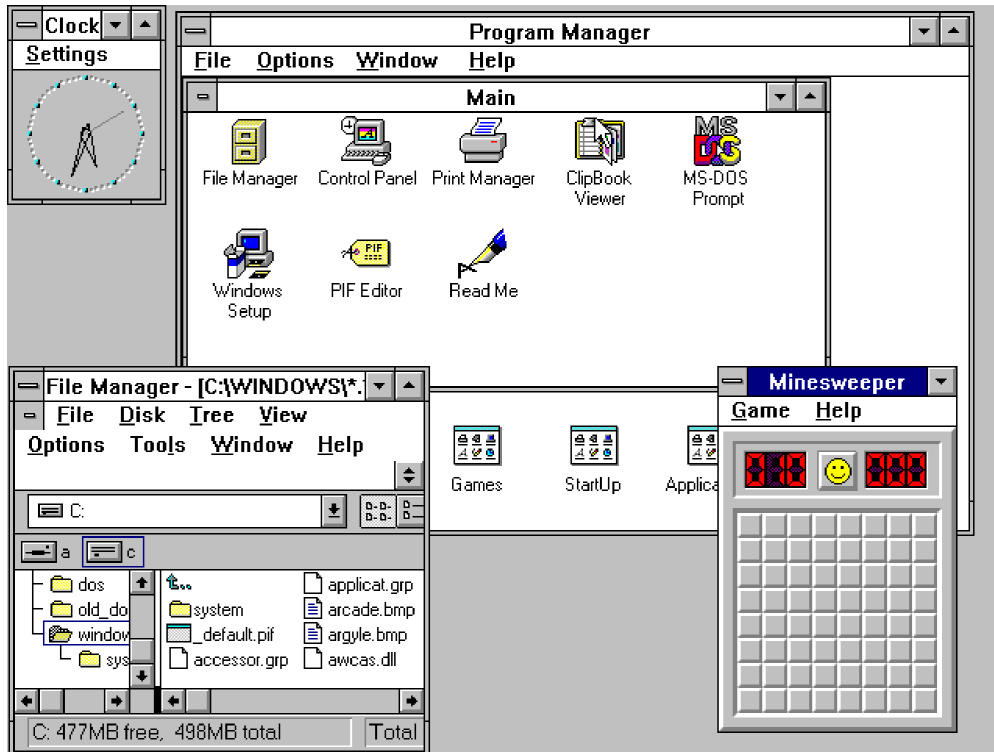
2.3 Hidden improvements

Despite this bleak description, a closer inspection of the world of graphic cards unveiled two tremendous changes which ended up deeply impacting DOOM.

Since 1992 with the release of new operating systems by Microsoft and IBM (Windows 3.1 and OS/2 2.0, respectively), demand for fast graphic cards had been growing strong. It was a huge technological leap for devices designed to push 4,000 bytes of information³⁴ in text mode to instead move 153,600 bytes³⁵ for GUIs.

³⁴2,000 bytes for the characters, and 2,000 bytes for screen attributes

³⁵16 colors at 640x480



Despite its simple interface, Windows 3.1's 640x480, 16 colors was able to bring PCs to their knees. Moving a window had to be done via its outlines since no hardware was capable of refreshing the screen fast enough in order to also show the content³⁶.

2.3.1 VGA Chip manufacturers

The first improvement came from VGA chip manufacturers. Sensing that demand for performance was growing, companies such as ATI, Cirrus Logic and Tseng Labs went to great effort to compete and achieve higher performance. Hardware GUI acceleration had not yet become mainstream so host-throughput was the dominating factor in redraw speed for graphical applications. They started to tightly integrate every component of a VGA card into a single chip (RAM, RamDAC, BIOS, Memory controller, Blitter, Ram Refresh, Cache controller, Timing Sequencer³⁷ to name only a few).

Some manufacturers such as Cirrus Logic even adopted a fabless business model where they sold semiconductor designs while outsourcing the fabrication.

³⁶NeXT workstations could do it and Steve Jobs mentioned it often during demos :)!

³⁷Tseng Labs ET4000

One optimization among many others was to leverage the fact that VGA RAM was more subject to write operations than reads. Using a FIFO SRAM cache to buffer operations and return right away tremendously improved screen blitting. Peeking at a card featuring one of the most notorious chip of the era by Tseng Labs, the ET4000 gives a good overview of what a customer could purchase.



Figure 2.29: ILLETW32 Britek Electronics. Photo courtesy <http://www.amoretro.de/>

By licensing the ET4000 ①, Britek Electronics only had to provide RAM ②, RAMDAC ③, a Timer ④ and apply a few customizations via a Programmable Array Logic TIBPAL16L8 ⑤. The VGA BIOS chip ⑥ could also be purchased from Tseng Labs.

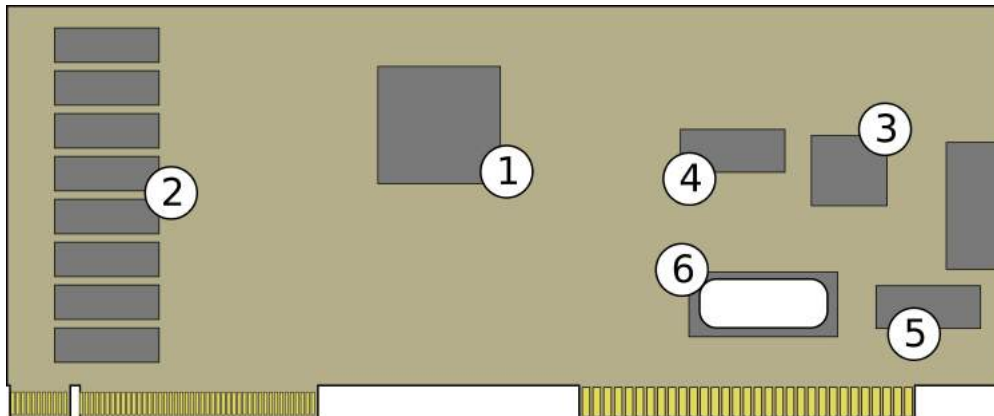


Figure 2.30

2.3.2 VL-Bus

As much as the video card manufacturer could optimize their product, there was still a huge bottleneck that was out of their control. Information written by the CPU still had to transit over the ISA bus.

Introduced in 1981, the first incarnation of the ISA bus had an 8-bit data path running at 4.77 MHz. It was upgraded in 1984, bringing its width to 16 bits and running at 6 MHz. After 10 years of service it was starting to show its age and was considered a performance killer.

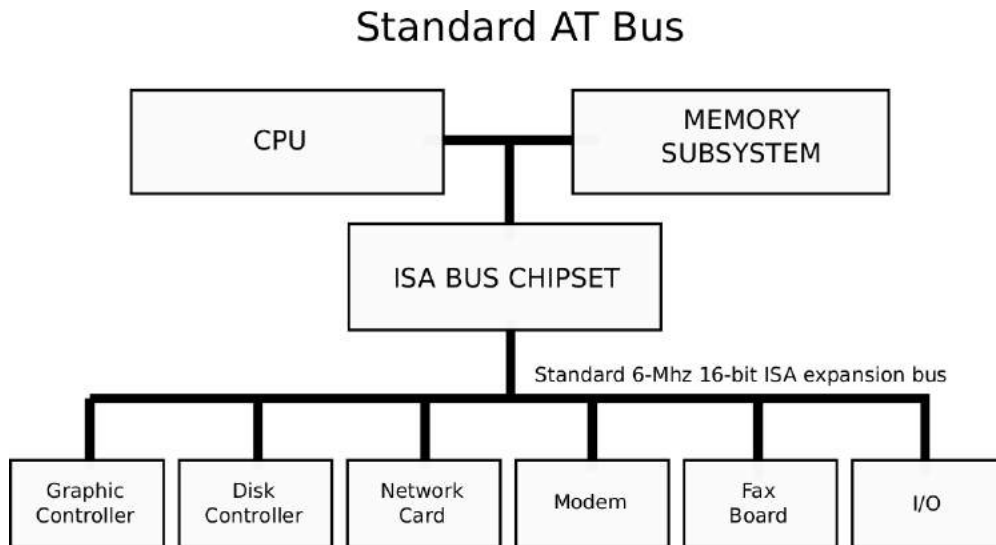


Figure 2.31: ISA Bus

Fed up with the state of things, hardware manufacturers teamed up to form VESA (Video Electronics Standards Association) and created a new Bus standard. They did not go for something complicated – the protocol was exactly the Intel 486 Bus Unit protocol, which made it a frictionless medium.

The VLB (VESA Local Bus) doubled ISA's bus data lines to 32 bits and increased its frequency to 33 Mhz, making it up to 10x faster when compared to the slowest ISA bus.

The chip design for the VLB controller was relatively simple because many of the core instructions (interrupts and port-mapped I/O) were still hosted by the ISA circuits already on the motherboard, while memory-mapped I/O and DMA data paths were on the same local bus as the one used by the CPU (see figure 2.32). The speed of the system data bus was based on the clock rate of the motherboard's crystal which meant the bus ran at the same speed as the CPU.

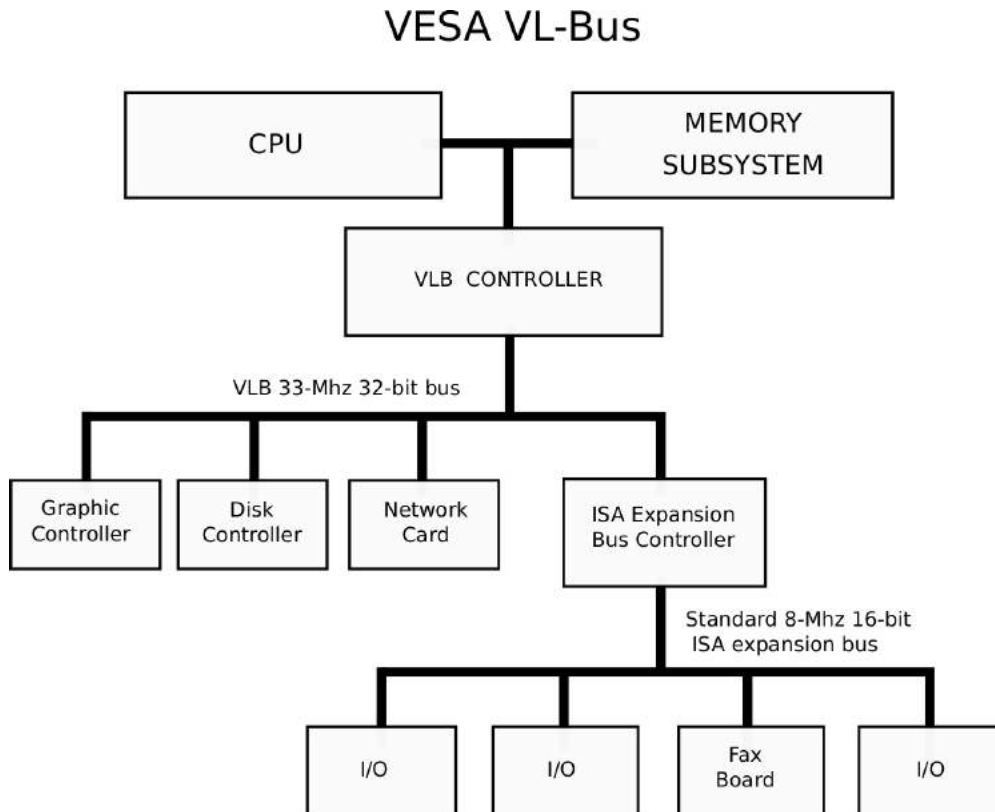


Figure 2.32: VL-Bus, a.k.a VESA Local Bus, a.k.a VLB

Closely tying the VL-Bus architecture to the 486 Bus Unit brought unmatched performance and considerably facilitated adoption since there was no need for a chipset. The term 'local bus' meant that the address, data and control signals were directly connected to the processor, so devices on the bus were connected via nothing more than some electrical buffering. This is one of the reasons for its simplicity, but it is also the reason for many of its limitations.

Forced to run at the same speed as the CPU, the VL-Bus suffered instability as frequencies reached 40 Mhz, resulting in crashes. Past this speed, the system became increasingly intolerant to timing variations³⁸. The root problem is that a local bus is by definition synchronous. Expansion card vendors had the difficult task of ensuring their products could run at a range of speeds, the upper limit of the range being undefined as new processors

³⁸The issue did not affect 486 DX-66Mhz, where the bus ran at only 33Mhz.

were introduced. This was a recipe for compatibility problems³⁹.

The second problem was that the electrical load driven by the CPU onto the bus decreased as the clock speed increased. Three slots could be provided at 33MHz, but only two at 40MHz and just one at 50MHz. This resulted in configuration hell since motherboard speed was configurable and came with three slots. Users would find some VLB slots "did not work" or "stopped working" as they tuned the frequency.

Cards were also hard to install due to their length and required pressure to force them into the VLB slot resulting in physical breakage⁴⁰.

Worst of all, Intel's 1993 Pentium Bus Unit protocol was instead based on PCI which was entirely incompatible. Unable to adapt, the VL-Bus found itself obsolete and the standard died within a year.

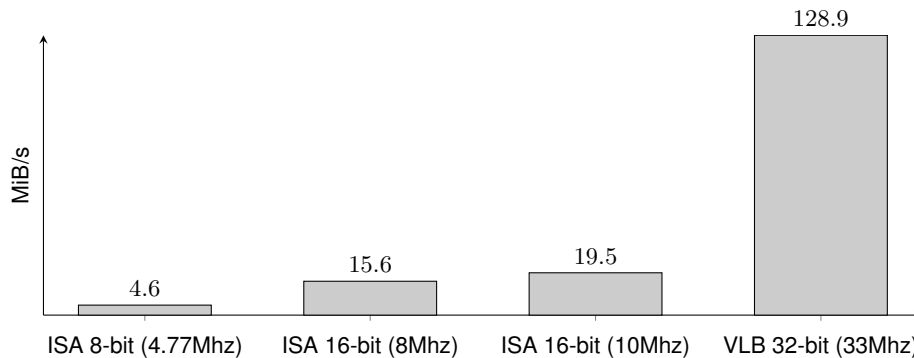


Figure 2.33: Theoretical Maximum Speeds (MiB/sec)⁴¹.

The next page shows three VGA cards available in 1994. The connectors instantly tell you what kind of bus and performance to expect.

- Top: an ATI 8800, with an ISA 8-bit interface.
- Middle: an ATI Mach32, with an ISA 16-bit interface.
- Bottom: a Cirrus Logic MachSpeed, with a VLB 32-bit interface.

Notice how the VLB connector uses only the 8-bit part of the ISA connector but has no teeth for the 16 other bits.

³⁹Such problems were experienced by users of VL-Bus systems using the AMD 80MHz processors, which had a 40MHz bus clock.

⁴⁰Friends jokingly renamed VLB to "Very Long Bus".

⁴¹At least one cycle is used to place the address on the bus, which halves payload bandwidth.



2.4 Sound System

PC were equipped with a "PC speaker", a device able to produce monotonic and annoying "beeps". The intent was to help diagnose system health at startup (one short beep meant the system was okay). But serious gamers always invested in a sound card. Thanks to its aggressive marketing, superior technology, and cheaper cards, Creative Labs dominated the market. In order to survive, any newcomer had to label itself "SoundBlaster-compatible". The unofficial standard meant OPL2-based FM synthesizer capability for music and a DSP able to play back digitized sounds at 22Khz, 8-bit per sample in stereo.

The early 90s were the theater of the last wave of innovation for gaming audio and saw the extinction of a previously key manufacturer named AdLib⁴². Two cards nonetheless managed to bring something new to the table. They were the Sound Blaster 16 by Creative Labs and the Gravis Ultrasound by Advanced Gravis Computer Technology Ltd.

2.4.1 Sound Blaster 16

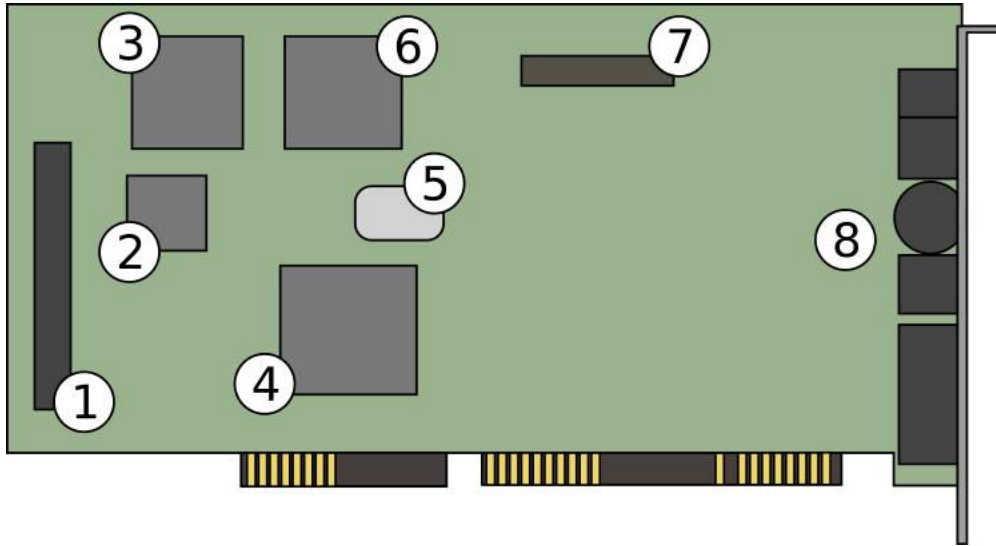
In June 1992, with the release of the Sound Blaster 16, Creative Labs solved the problem of PC audio for gaming forever with a card capable of CD quality playback – 44Khz 16-bit stereo samples. It was an instant hit and immensely successful with customers.



⁴²Which ironically had established the OPL2 chipset necessity.

were made but consumers remained deaf to the melody of these improvements. As audio chips became cheaper and with technical requirements stagnating, manufacturers started to provide audio capability built in on motherboards.

For a short time the extra Panasonic/Matsushita connectors permitting connection of a CD-ROM allowed sound cards to survive, in bundle products. But that was not enough to save them. Within ten years the market for sound cards disappeared.

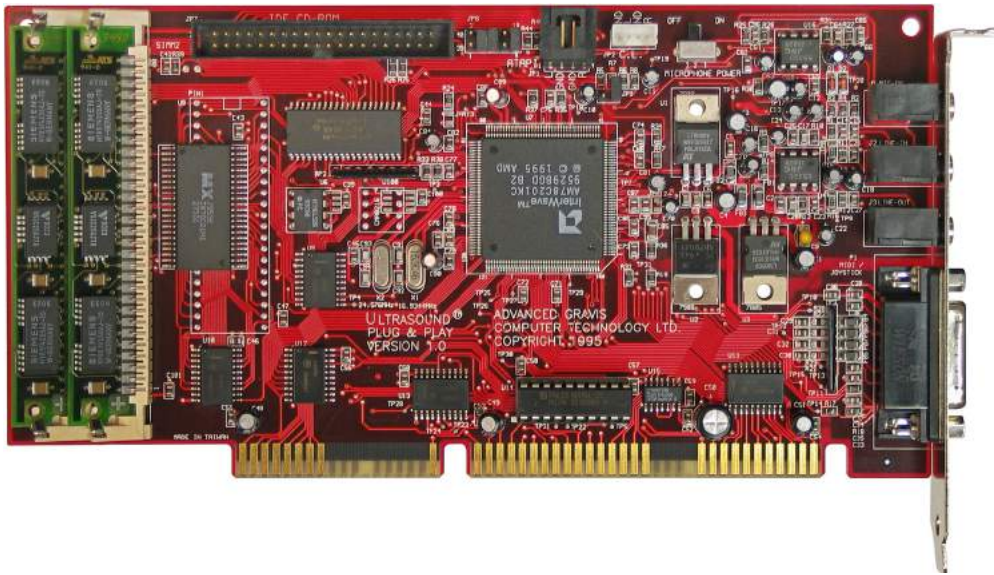


Above, a Sound Blaster 16 model CT1740 from 1994. ① Panasonic/Matsushita connector (for CD-ROM), ② C1741 DSP Chip, ③ C1748 ASP chip, ④ CT1746B Bus Interface, ⑤ 46.61512 Mhz oscillator, ⑥ CT1745A Mixer, ⑦ WaveBlaster Connector for MIDI "wavetable synthesizer" daughterboard, ⑧ (top to bottom) line-in, mic-in, volume wheel, line/speaker-out and MIDI/joystick port.

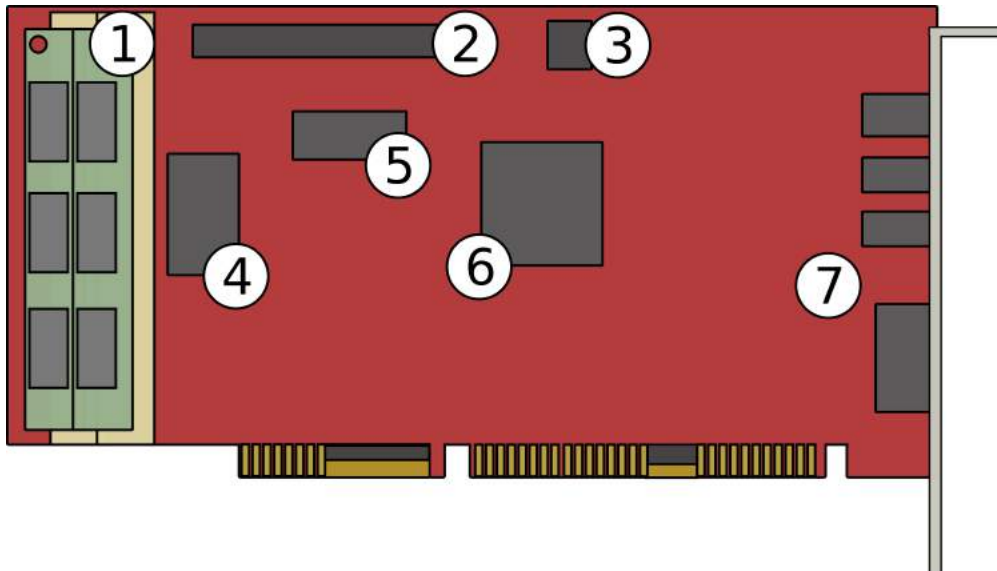
2.4.2 Gravis UltraSound

Gravis Computer Technology originally built what was universally accepted as the best PC joystick, the Gravis PC GamePad. With strong cash flow they decided to enter the sound card market with an audacious and innovative card. The Gravis UltraSound (nicknamed GUS) was released in 1992.

The GUS claimed Sound Blaster 2.0 music playback via TSR software emulation. On top of that the card had a capability like no other on the market. It was able to play back music not with FM synthesis but with digitized instrument samples. The technology, named "wavetable synthesis", achieved an audio quality far superior to its competitors.



The Gravis UltraSound Pro, ① 2 SIMM slots allowing up to 8 MiB RAM, ② IDE/AT-API Connector, ③ CD audio connector, ④ IW78C21M1 chip (1 MiB Flash ROM), ⑤ HM514260ALJ7 70ns DRAM, ⑥ Main CPU InterWave AM78C201KC and ⑦ from top to bottom: mic-in, line-in, line-out, MIDI/joystick port.



The concept was aggressive and so was the hardware that came out of the Gravis' factories. The red resin they used made their card unmistakably recognizable.

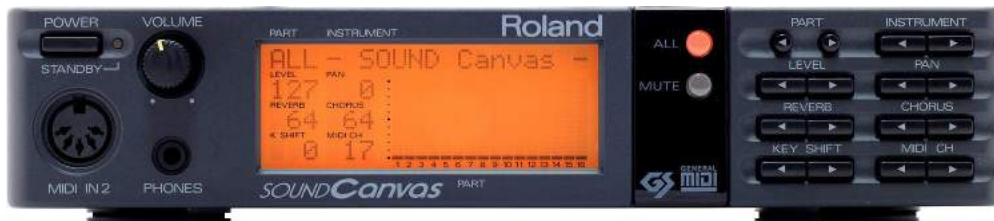
The cost of this technology was twofold. First, the card needed audio samples. This problem was "solved" via a Gravis driver that installed more than 12 MiB of sound samples⁴³. Second, the card had to be able to access the samples at runtime, meaning it had to have its own RAM. Since samples take up more space than sine equations, the original GUS shipped with 256 KiB, upgradeable to 1 MiB.

It rapidly developed a cult following among demo-makers who loved the high music quality it could achieve. For the gamer market however, things were more complicated. The GUS's GF1 main chip had difficulties emulating the OPL2 and setup was complicated (a mediocre TSR emulator had to be loaded manually by the user). The GUS also suffered from an unfortunate release date concurrent to the RAM shortage of 1993/1994. Players were reluctant to fork over the \$169 it cost, being \$40 more than a Sound Blaster 16. Initially selling well, sales slowed around 1995 and it was discontinued in 1996.

id Software was one of the few companies to support the Gravis UltraSound. DOOM included a mapping file that translated MIDI instrument IDs to Gravis .PAT instrument files⁴⁴. Listening to the electric guitars and drums of "At Doom's Gate" from DOOM's OST makes the SoundBlaster version pale in comparison. But all success stories must have the right timing and sadly the GUS was ahead of its time.

2.4.3 Roland

It would be a big omission to conclude without mentioning Roland's hardware. Established in 1972, Roland Corporation not only manufactured equipment for audio playback, it also provided the best hardware to author and record music. The breakthrough for DOS gaming was the Roland SC-55 (a.k.a SoundCanvas) released in 1991. Not only was it the very first General MIDI standard device (which defined 128 instruments that every device following the standard could adopt), it synthesized music using Roland's proprietary combination of prerecorded samples and subtractive synthesis which was far superior to Yamaha's OPL.



⁴³That was enormous at the time, when the full version of DOOM was 12 MiB as a matter of comparison.

⁴⁴It also controls which samples get loaded into RAM at various card configurations.

Roland's equipment was built entirely around the MIDI protocol which was carried via cables employing a special 5-pin circular connector.

The precursor of the SC-55, the MT-32 synthesizer, could be connected to a PC via an MPU-401 ISA MIDI adapter card. There was also a combo LAPC-I card which combined both the adapter and an MT-32 successor, the CM-32L, inside a single ISA card.



Figure 2.34: Roland LAPC-I

Roland also released the SCC-1 which combined the SC-55 and an MPU-401 onto a single ISA card.

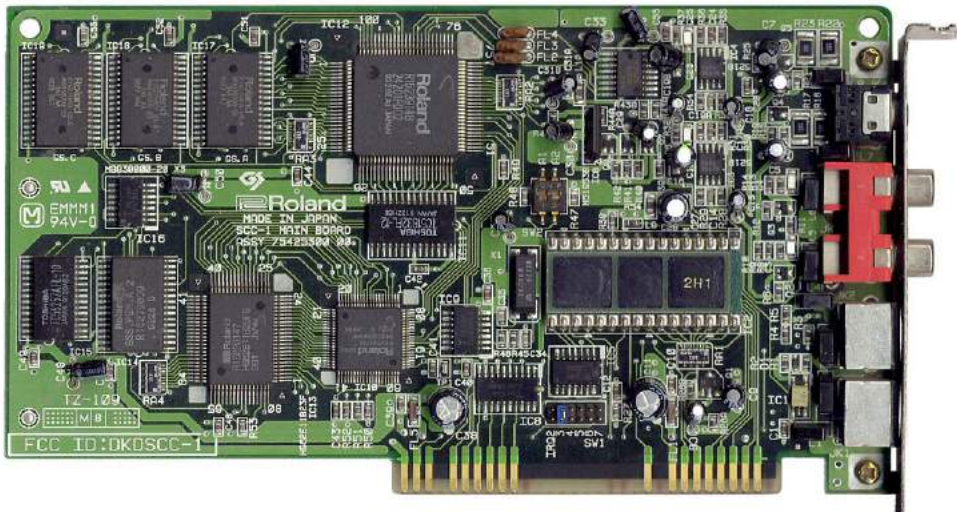
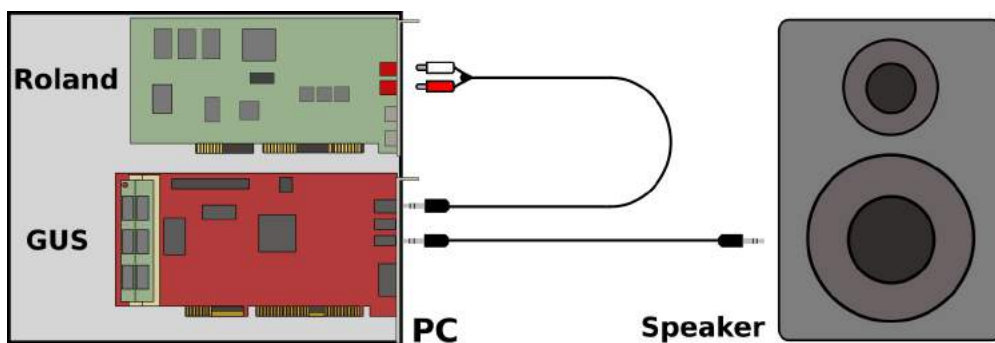


Figure 2.35: Roland SCC-1

For recording, an artist connected a musical keyboard to a "note recording program" called a MIDI sequencer. Once captured on the computer, the MIDI-based music could be tweaked and edited like any media.

For playback, things were a tiny bit more complicated. When Sierra On-Line pioneered support for Roland sound cards in 1988, the games leveraged the hardware to play beautiful music. The audio effects were either done via the PC Speaker or later using General MIDI stock audio effects⁴⁵. As games became increasingly elaborate with PCM digitized effects (which Roland cards could not play), gamers faced a dilemma where the best music needed a Roland but the best sound effects needed either a SoundBlaster or a GUS. The expensive (\$499 in 1991) solution was to buy both cards and mix the streams externally.



2.5 Network

The early 90s predated the wide adoption of the Internet and Wi-Fi. Connecting computers together was difficult and expensive⁴⁶. Even if you had the means, bandwidth and latency were abysmal. Most of the time, playing with friends meant getting all your computers into the same room (a LAN party). Playing from the comfort of your room was extremely uncommon. Amusingly, an unconnected computer is now deemed useless. Communication with other machines is something natural and the bare minimum for a machine to be useful.

But back in the early 90s, to pack your 50lb machine (including the CRT) on your bike, make it to a friend's place alive, plug in the cables, start DOOM and finally see your character move on the other computer's screen was an indescribable feeling. To witness machines actually communicating felt unreal and almost magical.

To achieve the impossible, players had three technologies available: Null-Modem cable, modems, and LAN via network cards.

⁴⁵ Another World in 1991 used the stock audio effects.

⁴⁶ Computer-to-computer games existed since the early 1980s. Some, like Battle Chess, were even cross-platform.

2.5.1 Null-Modem Cable

The cheapest way and what most people used was the \$20 cable known as a "Null-Modem" which was directly plugged in each PC's COM port. The cable offered no modulation at all (hence the name). For obvious reasons, only two players could participate.



Figure 2.36: Null-Modem cable

A two player game may sound lame by today's standards but back then it was so new and cool that it felt like the most amazing thing in the world.

2.5.2 BNC 10Base2 LAN (Local Area Network)

To play with more than one opponent was substantially more difficult. Besides the relatively easy financial burden of buying the equipment, you had to overcome the much more difficult task of convincing a parent to let four teenagers come to their house where they would scream all night. The famous saying, "fool me once, shame on you; fool me twice, shame on me" is rumored to have originated from betrayed mothers and fathers who had been *doomed* all night.

Leaving creative ways to ask for forgiveness aside, on the technical side a player had to plug in a 10Base2 network card via the ISA bus.



The card had a BNC connector upon which was to be plugged a T-shaped connector known as a T-piece. Each PC node was connected to up to two other nodes via 10Base2 coaxial cables. There was no central point in this type of networking; all machines involved in the network formed a chain. At both ends of the chain a signal terminator had to be connected to prevent an RF signal from being reflected back from each end, causing interference, or power loss.

The coaxial cables were bulky and so were the connectors. Connecting an end to a T-piece connector was fully achieved with a cool quarter turn of the coupling nut.

Once physically connected, games relied on the Internetwork Packet Exchange (IPX) which is a network level protocol like IP. There was no need to configure the host or the network since, contrary to IP, the IPX protocol was able to use the Ethernet MAC address as the machine's IPX address.

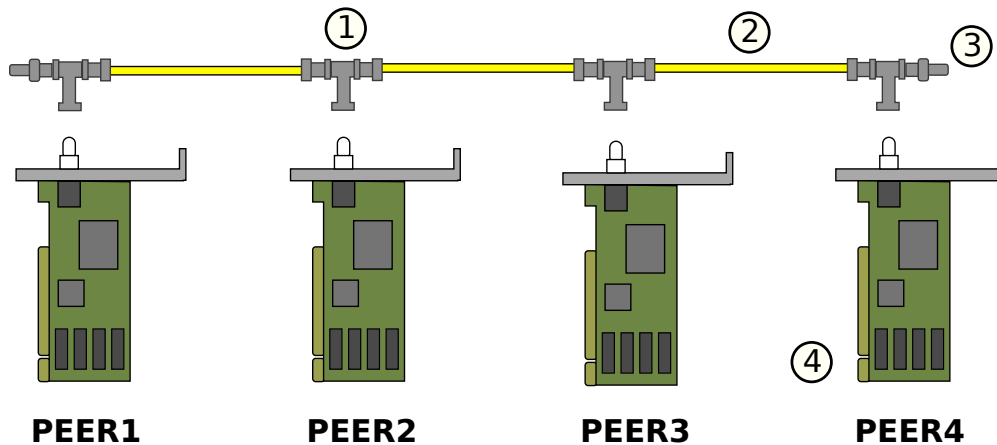


Figure 2.37: 10Base2 BNC based network.

Figure 2.37 shows the four elements of a 1994 LAN. ① the T-piece connector connecting two ② coaxial cables, forming a link. Each end of the chain must be closed with two load terminators ③. The network card connects to both the PC via its ISA bus extension slots ④ and the LAN T-base slots.

Trivia : Adding a new machine on the network meant either unplugging one of the T-piece connectors or unplugging a chain terminator. In both cases, the central bus was broken and all other machines lost connectivity. Everybody remembers the one friend who was always late to the LAN, forcing everybody to disconnect so he/she could join. The bandwidth was shared meaning the theoretical 10Mb/s was often closer to 5Mb/s. This does not account for friends who wanted to exchange a 30MiB song in .wav format (there was no MP3 at the time).

Trivia : Really fancy people could use a 10baseT network which required a "hub" central device resulting in a star-shaped network.

2.5.3 Modem

The most fortunate players were able to afford the luxury of networking from home. That was very expensive since they not only had to pay for a modem but they also had to pay for every single minute spent online. Before broadband, modems used phone landlines to connect to the Internet provider. This meant nobody could use the telephone while the connection was active. Anybody picking up the phone in the house created enough disturbance to kill the connection.

Internet was unattractive since gaming and accessing Bulletin Board Systems was done by calling phone numbers directly. Finding a cool BBS or a gaming partner phone number was an adventure of its own. If one really wanted to read the few HTML pages available, AOL (America OnLine) offered a package of five hours for \$9.95 with each extra hour billed \$3.50 per unit. An user averaging 2h/day was billed $9.95 + 55 * 3.5 = \$202$ for a month⁴⁷ not to forget a one-time fee of \$399⁴⁸ for a 9600 baud model .



Figure 2.38: US Robotics 28.8k baud modem. The top of the line in 1994.

While establishing the initial handshake, the modem speaker was kept open. An attuned ear could easily recognize the different phases of V.X bis transaction, speed negotiation, echo canceller disabling, and modulation mode selection, together making the unforgettable melody of a deathmatch in the making.

⁴⁷Adjusted to inflation: \$352 in 2018.

⁴⁸Adjusted to inflation: \$696 in 2018.

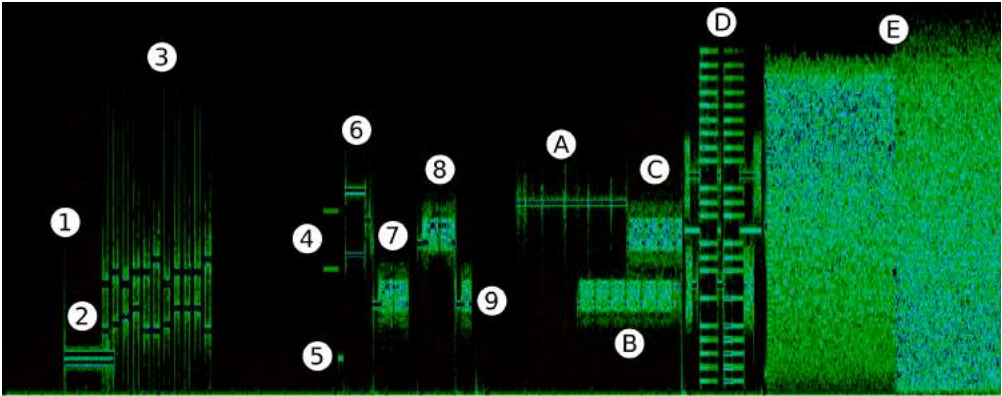


Figure 2.39: 18 second spectrogram of a V.34 handshake⁴⁹

Stage	Description
1	Modem goes off hook.
2	Telephone exchange sends a dial tone.
3	DTMF: Model dials 1-(570)-234-0001 a DOOM player in Pennsylvania, USA.
4	Answering modem initiates a V.8 bis transaction.
5	Answering modem asks caller for a list of its capabilities.
6	Caller responds to V8 bis initiation, agrees to list its capabilities and request to escape from telephony into information transfer mode.
7	FSK Data @ 300 bps: I'm capable of full V.8. I can transmit ACK. My country is US and I was made by Net2phone Inc.
8	FSK Data @ 300 bps: Why don't we use V8 then.
9	Ok, mode acknowledged. Terminating V.8 bis transaction.
A	Answering modem disables echo suppressors and cancellers in PSTN.
B	FSK Data @ 300 bps: Repeated 6x Here are my modulation modes: V.34, V.32, v.23 duplex ...
C	FSK Data @ 300 bps: Repeated 3x: I can do any of those.
D	Both modems send a wide-spectrum probing signal in both directions to do measurements on the line.
E	Both modems go to scrambled data

Figure 2.40

Throughout the '90s, bandwidth steadily improved. Upon DOOM's release most modems were capable of 14.4 Kbit/s. Those who downloaded the shareware version in December 1993 had to wait 25 minutes to retrieve the 2,166,955 bytes of the ZIP archive.

⁴⁹Source: "The sound of the dialup, pictured" by Oona Räisänen.

Year	Version	Bandwidth
1990	V.32	9.6 kbit/s
1991	V.32bis	14.4 kbit/s
1994	V.34	28.8 kbit/s
1995	V.34	33.6 kbit/s
1996	V.90	56.0/33.6 kbit/s
1999	V.92	56.0/48.0 kbit/s

Figure 2.41: Modem speeds through the 90s.⁵⁰

On top of the V.XX hardware communication layer, modems were driven using Hayes commands⁵¹. Notice how the command ATDT translated to DTMF in the previous spectrogram.

Modem A	Modem B	Comments
ATDT15551234		Modem A issues a dial command: AT-Get the modem's ATtention; D-Dial; T-Touch-Tone; 15551234-Call this number
	RING	Modem A begins dialing. Modem B's phone-line rings, and the modem reports the fact.
	ATA	Modem B issues answer command.
CONNECT	CONNECT	The modems connect, and both modems report "connect"..
abcdef	abcdef	When the modems are connected, any characters typed at either side will appear on the other side.
	+++	Modem B issues the modem escape command.
OK		The modem acknowledges it.
	ATH	Modem B issues a hang up command.
NO CARRIER	OK	Both modems report that the connection has ended. Modem B responds "OK" as the expected result of the command; modem A says NO CARRIER to report that the remote side interrupted the connection.

Figure 2.42: AT layer dialog between caller and callee.

Trivia : The fragility of these connections led to humorous ways to end a message. People would finish forum posts with "Hey! Wait! Don't pick up the ph{#{\$%&+'+'%NO CARRIER".

⁵⁰Bit rate increased at the expense of latency. A 9600 baud modem played DOOM better than the default configurations on 56kbit modems. Quake needed more bandwidth than DOOM's controller replication, so it became a different tradeoff.

⁵¹A nice abstraction layer, but DOOM still has a long file with initialization parameters for 49 modems.

2.6 RAM

With the price of RAM dropping, game developers now could count on 4 MiB. This increase should have been good news, resulting in video games with richer worlds, better assets, more characters and bigger maps. Due to the infamous way memory had to be managed it instead meant more complexity to handle and more headache.

The fault fell a little bit on Intel and a lot on Microsoft. In 1981, IBM released its first PC, the 5150, which was built around the Intel 8088. The CPU was limited to 16-bit registers, but Intel wanted it to be able to access a 20-bit address space. To reconcile both elements, Intel's designers came up with an abomination called segmented addressing where two 16-bit registers were combined to form a 20-bit address.

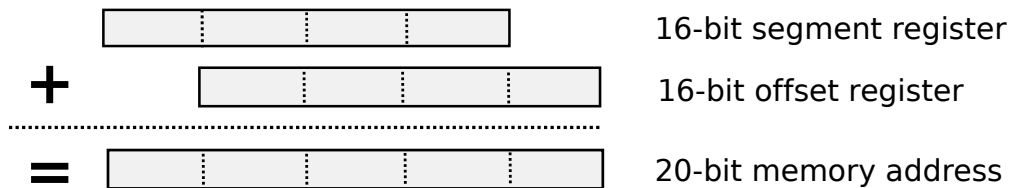


Figure 2.43

Pointer manipulation was error prone since different segment/offset combinations could point to the same RAM location. There were also issues related to pointer arithmetic where once the offset wrapped around, the segment was not automatically updated.

The RAM system became a mess with the Intel 286 and the 386SX which addressed 24 bits and worsened with the 386DX and the 486 which both had a 32-bit address bus. The address space was too much for what the 20-bit segmentation trick was capable of. The solution was to resort to memory managers such as `EMM386.EXE` and `HIMEM.SYS`⁵², which both provided the means to work with non-addressable RAM located beyond the 1 MiB barrier.

There would have been a simpler solution. Intel allowed its CPUs to function in two modes: the backward-compatible real mode which made the CPU behave like a very fast 8088, and protected mode which unleashed the full power of the CPU. In protected mode, 32-bit registers were large enough to address all RAM on board (this is known as flat addressing).

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

With the growing pain and frustration of DOS, some people saw an opportunity.

⁵²16-bit programming and memory managers were covered in *Game Engine Black Book: Wolfenstein 3D*.

While there were many products which could address this, two companies in particular stood out with a winning combination. Watcom International Corporation's C compiler and Rational Systems' DOS/4GW "DOS extender" together allowed programs to run in protected mode while still having access to 16-bit DOS functions.

2.6.1 DOS/4GW Extender

Under DOS the normal way to perform a "system call" is to use a software interrupt instruction with parameter 21h. In C programming, this was abstracted away by the header `DOS.H` which performed all the lower level work behind the scenes.

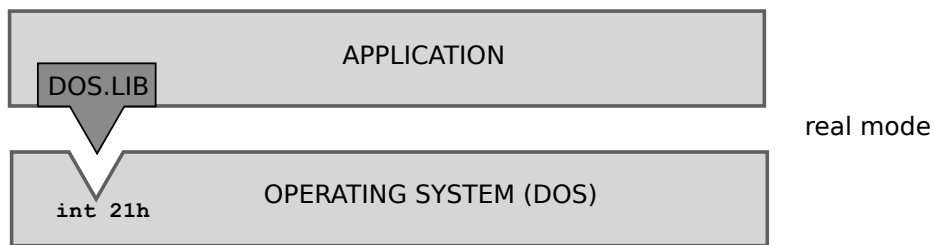
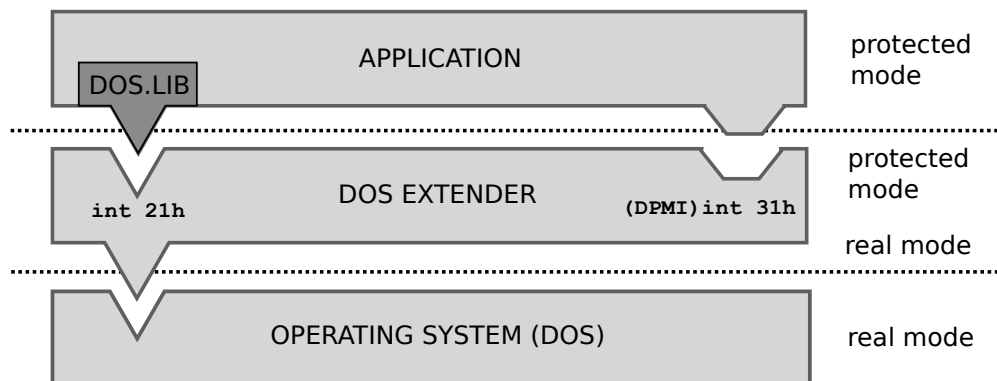


Figure 2.44

To allow the app to run in one mode and the OS to run in another, the two worlds had to be bridged. A middle layer called the "DOS extender" – able to run in both modes – inserted itself between the program and the operating system.



Upon startup, the DOS extender would place hooks into the OS's Interrupt Vector Table and place its own routines there. From an application's standing point everything was

transparent, the developer had no code to change. To perform a system call not hooked by the extender (e.g. `int 33h` to read mouse inputs), the extender offered a special interface called DPMI on interrupt `31h` which took care of translating 32-bit register requests to 16-bit so IVT routines would understand them.

Trivia : DPMI (DOS Protected Mode Interface) was originally created to allow Windows 3.0 to run 32-bit applications and to be compatible with a joint operating system project with IBM called OS/2.

When the extender intercepted an operating system call, it had a lot of work to do:

1. Perform all translation needed (e.g. a 32-bit address had to be expressed as a 16-bit offset with a 16-bit segment).
2. Switch the CPU to real mode.
3. Forward the call to DOS.
4. Retrieve the results and convert 16-bit register values back to 32-bit.
5. Switch the CPU back to protected mode.

The performance-sensitive operations were in switching between real mode and protected mode. Originally this was a problem on 286 CPUs since Intel never imagined a program might want to switch back to real mode from protected mode. Various tricks had to be used⁵³, among them faking a keyboard Ctrl-Alt-Del reboot to reset the CPU without actually rebooting.

On the other hand, switching from real mode to protected mode is simple. Setting the Control Register from bit 0 to 1 takes six instructions.

```
cli           ; disable interrupts.
lgdt [gdt_r] ; set Global Descriptor Table address.
mov eax, cr0
or al, 1      ; Prepare Protected Mode.
mov cr0, eax

; Flush of the pipeline via a far jump instruction.
JMP 08h:PModeMain

PModeMain:
; load DS, ES, FS, GS, SS, ESP.
```

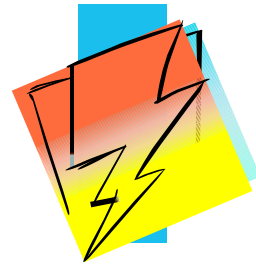
⁵³These are detailed in *Game Engine Black Book: Wolfenstein 3D*.

DOOM used the DOS/4GW extender by Rational Systems. Its presence could briefly be seen on startup. Executing `DOOM.EXE` triggered DOS to load the tiny extender. Once loaded, DOS/4GW switched the CPU into protected mode, loaded DOOM's code into memory and branched to the `main` function.

```
C:\DOOM>doom
DOS/4GW Professional Protected Mode Run-time Version 1.95
Copyright (c) Rational Systems, Inc. 1990-1993
```

2.7 Watcom

The DOS extender was magical but hard to set up in a standalone product. A bootstrap which would locate `DOS4GW.EXE` and the program to run, and set up both, required multiple steps and close to 100 lines of C code⁵⁴. The ramp-up time was significant and raised the barrier to entry. What was really needed was an integrated environment where the compiler and the linker would take care of bundling the extender and the application together into one executable. The solution would once again come from the Great White North.



The Watcom compiler project was started in 1979 at the University of Waterloo in Ontario, Canada. Initially only supporting BASIC, it was improved over the years by students with support for new OSes and languages. In 1987, three Ph.Ds (Fred Crigger, Ian McPhee, and Jack Schueler) made it the first C compiler to run on an IBM PC.

Sensing commercial potential, they incorporated Watcom International Corporation and picked a lightning bolt for their logo to advertise their focus on performance. Five years later, in 1993, Watcom C had considerably improved. The latest version (9.0), retailing for "only" \$639⁵⁵, was deemed the best available on MS-DOS⁵⁶.


Not only were they talented programmers, they also excelled at marketing their products. In the early 90s, a reader could not open a computer magazine without finding a full page advertising Watcom's compiler. Every ad underlined the presence of a DOS extender that freed programmers from the hated 16-bit mode and "unleashed 32-bit power".

⁵⁴Source: Watcom C/C++ Programmer's Guide, "7.1.1 The Stub Program"

⁵⁵Inflation adjusted, USD\$1,116 in 2018. Nowadays compilers are "free".

⁵⁶Editor's choice – PC Magazine, April 1995.

Delivering the Power: WATCOM C9.0/386



- ▶ **The Widest Range of 32-bit Intel x86 Platforms**
32-bit DOS, 32-bit Windows, OS/2 2.0, AutoCAD ADS
- ▶ **The Industry's Leading Code Optimizer**
Advanced global optimizer with new 486 optimizations
- ▶ **The Most Comprehensive Toolset**
Debugger, profiler, protected-mode compiler and linker, 32-bit DOS extender with royalty-free run-time, licensed components from Microsoft SDK, and more
- ▶ **The Best Value in 32-Bit Tools: \$895***

Unleash 32-bit Power!

WATCOM C9.0/386 lets you exploit the two key 32-bit performance benefits. The 32-bit flat memory model simplifies memory management and lets applications address beyond the 640K limit. Powerful 32-bit instruction processing delivers a significant speed advantage: typically at least a 2x speedup.

You Get:

- ▶ 100% ANSI and SAA compatible: C9.0/386 passes all Plum Hall Validation Suite tests
- ▶ Extensive Microsoft compatibility simplifies porting of 16-bit code
- ▶ Royalty-free run-time for 32-bit DOS, Windows and OS/2 apps
- ▶ Comprehensive toolset includes debugger, linker, profiler and more
- ▶ DOS extender support for Rational, Phar Lap and Ergo
- ▶ Run-time compatible with WATCOM FORTRAN 77/386

32-bit DOS support includes the DOS/4GW 32-bit DOS extender by Rational Systems with royalty-free runtime license

- ▶ Virtual Memory support up to 32Mb

32-bit Windows support enables development and debugging of true 32-bit GUI applications and DLLs

- ▶ Includes licensed Microsoft SDK components

32-bit OS/2 2.0 support includes development for multiple target environments including OS/2 2.0, 32-bit DOS and 32-bit Windows

- ▶ Access to full OS/2 2.0 API including Presentation Manager
- ▶ Integrated with IBM Workframe/2 Environment

AutoCAD ADS and **ADI** Development: Everything you need to develop and debug ADS and ADI applications for AutoCAD Release 11

Novell's Network C for NLM's SDK includes C/386

The Industry's Choice.

Autodesk, Robert Wenig, Manager, AutoCAD for Windows:

"At Autodesk, we're using WATCOM C/386 in the development of strategic new products since it gives us a competitive edge through early access to new technologies. We also highly recommend WATCOM C/386 to third party AutoCAD add-on (ADS and ADI) developers."

Fox Software, David Fulton, President: "FoxPro 2.0 itself is written in WATCOM C, and takes advantage of its many superior features. Optimizing for either speed or compactness is not uncommon, but to accomplish both was quite remarkable."

GO, Robert Carr, Vice President of Software: "After looking at the 32-bit Intel 80x86 tools available in the industry, WATCOM C was the best choice. Key factors in our decision were performance, functionality, reliability and technical support."

IBM, John Soyring, Director of OS/2 Software Developer Programs: "IBM and WATCOM are working together closely to integrate these compilers with the OS/2 2.0 Programmer's Workbench."

Lotus, David Reed, Chief Scientist and Vice President, Pen-Based Applications: "In new product development we're working with WATCOM C because of superior code optimization, responsive support, and timely delivery of technologies important to us like p-code and support for GO Corp's PenPoint."

Novell, Nancy Woodward, V.P. and G.M., Development Products: "We searched the industry for the best 386 C compiler technology to incorporate with our developer toolkits. Our choice was WATCOM."

WATCOM

1-800-265-4555

The Leader in 32-bit Development Tools

415 Pailin Street, Waterloo, Ontario, Canada. Telephone: (519) 886-2700. Fax: (519) 747-4871. *Price does not include freight and taxes where applicable. Authorized dealers may sell for less. WATCOM C and Lightning Developer are trademarks of WATCOM Systems, Inc. DOS/4G and DOS/IBM are trademarks of Rational Systems, Inc. Other trademarks are the properties of their respective owners. Copyright 1992 WATCOM Products, Inc.



Circle 124 on Inquiry Card.

Not only were they in the press, they also advertised online, such as on BBSes and Usenet.

“ WATCOM C/C++ will produce code which is at *least* twice as fast as your current 16-bit compiler, and more typically around five times as fast.

— **rec.games.programmer** ”

Trivia : One of the many marketing tricks up Watcom's sleeves was to never have released a Watcom v1.0 or even a Watcom v2.0. They started directly at "version 6". This was at least one version ahead of their competitors (Borland and Microsoft). A higher number unconsciously carried a notion of "more advanced than its competitors". Version one was also likely to feature many bugs whereas the sixth installment was likely to have been battle-hardened.

2.7.0.1 Popularity

id Software was not the only team to value Watcom's solution. Many other studios entrusted it with their code, and as a result much well-known software of the 90s was built with Watcom technology:

1. id Software
 - (a) DOOM (1993)
 - (b) DOOM II (1994)
2. Blizzard Entertainment
 - (a) Warcraft (1994)
 - (b) Warcraft II (1995)
3. Ken Silverman's BUILD Engine based games
 - (a) Duke Nukem 3D (1996)
 - (b) Shadow Warrior (1997)
 - (c) Blood (1997)
4. LucasArts Entertainment Company
 - (a) Full Throttle (1995)
 - (b) The Dig (1995)
 - (c) Dark Forces (1995)
 - (d) Rebel Assault II (1995)

2.7.1 ANSI C

The Watcom/extender combo made programming simpler and it also made programs run faster but the best has yet to be mentioned. There is a third aspect of protected-mode programming – less obvious but very important – that had a significant impact on DOOM.

To bring C to the world of PC/DOS's real mode and accommodate for segment manipulation, the language had been "augmented". An example from Wolfenstein 3D's memory manager shows what "C for DOS" looked like.

```
void MM_Startup (void) {
    int i;
    unsigned long length;
    void far *start;
    unsigned segstart, seglength, endfree;

    // get all available near conventional memory segments
    length=coreleft();
    start = (void far *) (nearheap = malloc(length));

    length -= 16-(FP_OFF(start)&15);
    length -= SAVENEARHEAP;
    seglength = length / 16;          // now in paragraphs
    segstart = FP_SEG(start)+(FP_OFF(start)+15)/16;
    MML_UseSpace (segstart, seglength);
    mminfo.nearheap = length;

    // get all available far conventional memory segments
    length=farcoreleft();
    start = farheap = farmalloc(length);
    length -= 16-(FP_OFF(start)&15);
    length -= SAVEFARHEAP;
    seglength = length / 16;          // now in paragraphs
    segstart = FP_SEG(start)+(FP_OFF(start)+15)/16;
    MML_UseSpace (segstart, seglength);
    mminfo.farheap = length;
    mminfo.mainmem = mminfo.nearheap + mminfo.farheap;
}
```

Notice the wart keywords such as `near`, `far`, macros like `FP_OFF` and `FP_SEG`, and the DOS.H library functions such as `farmalloc`, `coreleft`, and `farcoreleft`. Neither "C for DOS" nor the I/O functions were portable. As a result, it was impossible to take a UNIX program and compile it directly on DOS.

Using the Watcom compiler, C could be written using the ANSI standard, which opened the door to authoring programs on different machines running a different operating system.

One system in particular would end up catching id Software's attention. The name was NeXTSTEP, running on hardware manufactured by NeXT, Inc.

Chapter 3

NeXT

3.1 History

NeXT's history starts (and amusingly, also ends) at Apple. In May of 1985, the mediocre sales of the Macintosh painted a bleak future for the company. Steve Jobs, co-founder and then General Manager of the Mac department, wanted to lower the price and increase marketing in order to boost the Mac. John Sculley then CEO, wanted to abandon the Mac and refocus the company's resources on the Apple II, the only profitable product Apple had marketed until then.



A vote was called and the board of directors sided with Sculley. Steve Jobs found himself stripped of all responsibilities. A few months later, on September 13, 1985, he resigned and went on to work on his next project.

NeXT, Inc. was incorporated in February 1986 with \$7 million of Jobs' own money. Many members of the Mac division left Apple to join the newly formed company, among them Joanna Hoffman, Guy "Bud" Tribble (head of software division), George Crow, Rich Page, Susan Barnes, Susan Kare, and Dan'l Lewin.

With NeXT, Jobs went back to a project he had contemplated for Apple in August 1985. While touring universities to boost Mac sales, he had met Paul Berg, a Nobel Laureate in chemistry. Paul was frustrated with the cost¹ of teaching students about recombinant DNA in wet laboratories. It would have been cheaper to simulate them. It seemed there was a market for 3M² workstations targeted at universities and students³. NeXT set itself to build

¹\$100,000.

²One Megabyte of RAM, a Megapixel display and MegaFLOP performance.

³The Second Coming of Steve Jobs.

something powerful yet cheap enough that college students could afford it.

“

I want some kid at Stanford to be able to cure cancer in his dorm room.

— **Steve Jobs, 1987**

”

Steve Jobs spared no expenses. For \$100,000, Paul Rand was commissioned with a logo. An automated factory featuring automated surface-mount motherboard assembly⁴ capable of producing 10,000 units per month was built in Fremont, CA. The design firm Frogdesign, which had proven itself with the Apple IIc, was hired. The goal was to ship by the end of 1986.

The machine was to be perfect, following Alan Kay's concept of creating both the hardware and the software to run it.

“

People who are really serious about software should make their own hardware.

— **Alan Kay, 1980**

”

Using their experience from Apple and particularly their work on the Macintosh, the company defined the three pillars of the NeXT Computer: GUI, Networking and Object-Oriented programming.

“

I went to Xerox PARC. And they were very kind. They showed me what they are working on. And they showed me really three things. But I was so blinded by the first one that I didn't even really see the other two. One of the things they showed me was object oriented programming – they showed me that but I didn't even see that. The other one they showed me was a networked computer system... they had over a hundred Alto computers all networked using email etc., etc., I didn't even see that. I was so blinded by the first thing they showed me, which was the graphical user interface. I thought it was the best thing I'd ever seen in my life.

— **Steve Jobs, 1995**

”

⁴Source: "The Machine to Build The Machines" mini documentary.

3.2 The NeXT Computer

The first machine shipped in 1989 after three years of hard work. Not meeting the initial release target was not a problem according to Jobs who famously replied to a journalist inquiring about the delay: "Late? This computer is five years ahead of its time!".

Based on a Motorola 68030 25 Mhz with 8 MiB RAM and featuring powerful co-processors such as a DSP and a FPU, the high-performance hardware delivered. The machine also happened to be gorgeous. In an era where most computer cases were made of beige plastic, the elegance of the one foot perfect cube made of painted magnesium stood out.



Figure 3.1: The Next Computer

The monitor was a piece of art itself. The 17" MegaDisplay allowed a high⁵ resolution of 1120 x 832 pixels with a density of 92 DPI. The Cube's 256 KiB of VRAM allowed four shades of gray per pixel. At launch the supply chain was so tight that when ordering a NeXT Computer, the customer received two parcels – one from Fremont containing the central unit, and another directly from Sony containing the MegaDisplay.

⁵At the time, a 14" monitor delivering a resolution of 640x480 was high-end standard on PC.

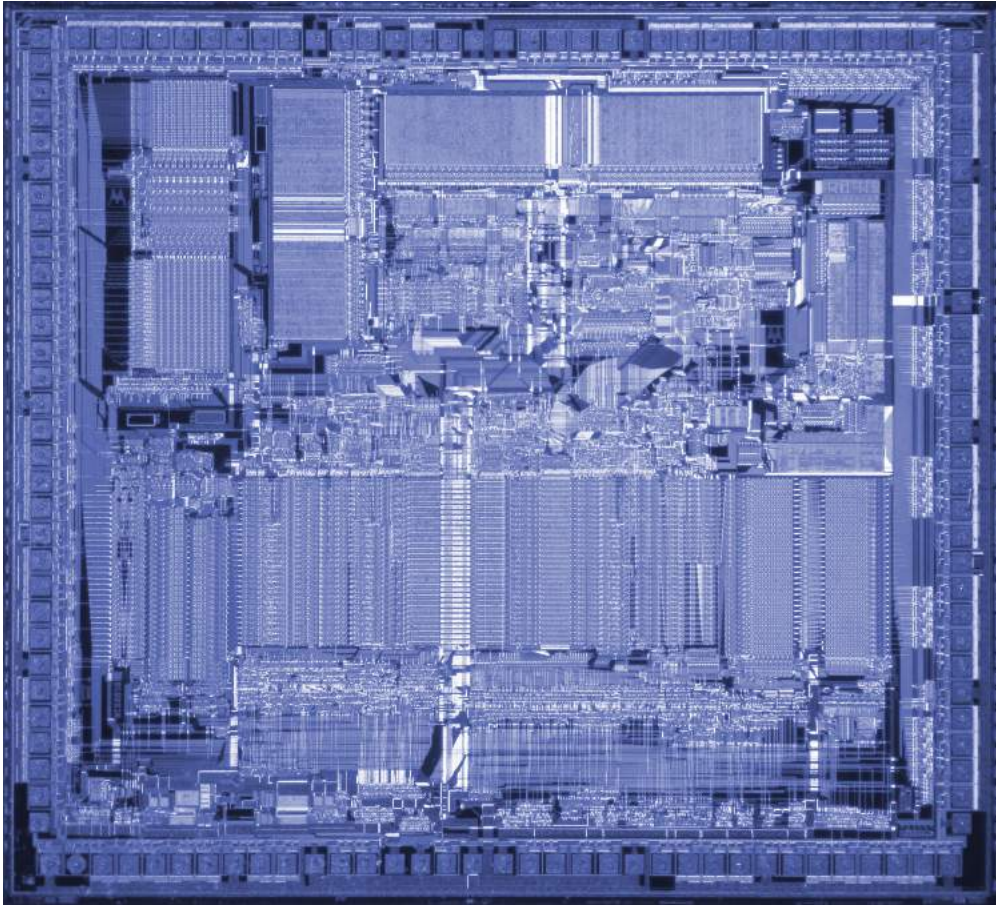


Figure 3.2: Motorola 68030

One of the many innovations of the NeXT Computer was its reliance on the 256 MiB magneto-optical drive, a hybrid between a HDD and floppy disk aimed at filling both use cases. According to Steve Jobs, it was supposed to allow users to "take their whole world in their backpacks".

At the heart of the machine, the 32-bit 68030 was the latest in Motorola's 68000 series. The choice was likely influenced by the experience NeXT hardware engineers had built while working on Apple's Macintosh and Lisa (both were powered by a 68000).

Running at a frequency of 25Mhz, it was able to execute nearly 5 MIPS. It did not feature a built-in FPU, so a Motorola 68882 was placed next to the CPU on the motherboard.

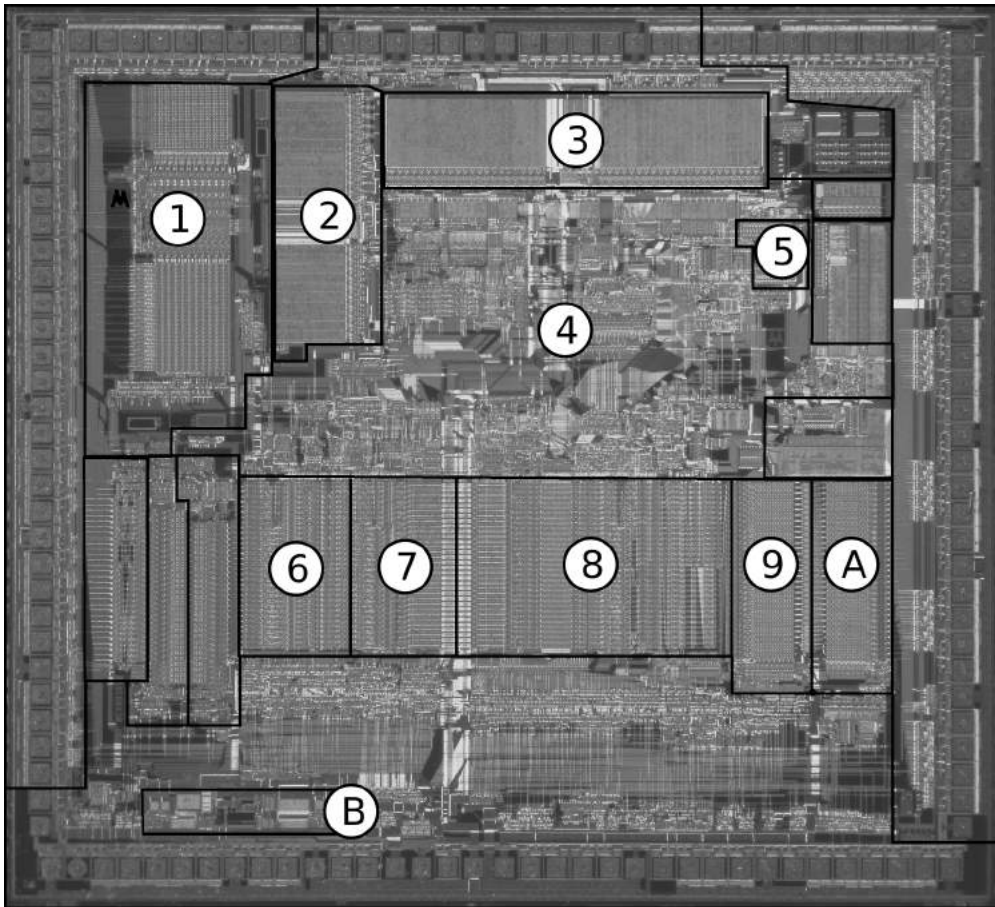


Figure 3.3: Motorola 68030 diagram⁶

Above, the 273,000 transistors of the 68030, made up of ① Memory Management Unit, ② μ ROM, ③ nROM, ④ Control Section, ⑤ Instruction Pipe, ⑥ Program Counter Execution Unit, ⑦ Address Execution Unit, ⑧ Data Execution Unit, ⑨ 256 bytes i-cache, ⑩ 256 bytes d-cache, and ⑪ Clock Generator.

It is unclear how much of a performance boost the two caches provided. Their small size of 256 bytes each would have meant a significant cache miss rate (Intel had discarded its on-die cache from their 386 for this very reason). Interestingly, the designer decided to use both micro-code and nano-code. Sixteen general-purpose registers were available which is pretty common for a RISC architecture where load and store have to be done manually⁷.

⁶Source: "The NeXT Book" by Bruce F. Webster.

⁷Intel's CISC-based 486 had eight.

Contrary to PCs which were a mess of wires, the NeXT Computers formed a chain. The mouse was connected to the keyboard, itself connected to the screen, connected to the Cube.



If initially the NeXT Computer was acclaimed for its specs, there was a serious issue with the price. Market studies showed that students and researchers wanted a workstation priced at \$3,000. The NeXT Computer started at more than twice the ideal price at \$6,500. To make it worse, the optical drive that powered the basic configuration would turn out to be great for backup but way too slow for runtime. Not only was it noisy and unreliable, it offered an access time of 90 ms, 10 times slower than a hard-drive and made the operating system crawl. This rendered the "optional" \$3,500 330 MiB SCSI hard-drive an absolute necessity, pushing the final price tag to \$10,000! A big price to pay for a machine not even able to output color.

3.3 Line of Products

Given the low sales of the NeXT Computer, the original machine was discontinued and the line of products refreshed. In 1991 NeXT released three new products⁸. The NeXTcube was the direct successor to the NeXT Computer. A smaller, flattened version of the NeXTcube called the NeXTstation offered built-in color capability but no expansion slots. Last but not least, there was a graphic and video processor expansion board called NeXTdimension.

Name	Year	CPU	Price	in 2018
NeXT Computer	1989	68030 25 Mhz	\$6,500	\$12,938
NeXTstation	1991	68040 25 Mhz	\$4,995	\$9,157
NeXTcube	1991	68040 25 Mhz	\$12 395	\$21,171
NeXTdimension	1991	i860 33 Mhz	\$3,995	\$7,552
NeXTstation Color	1991	68040 25 Mhz	\$7,995	\$14,656
NeXTcube Turbo	1992	68040 33 Mhz	\$10,000	\$18,121
NeXTstation Turbo	1992	68040 33 Mhz	\$5995	\$11,932
NeXTstation TurboColor	1992	68040 33 Mhz	\$8995	\$17,904

Figure 3.4: NeXT products from 1989 to 1993⁹.

⁸ Announced four months in advance on September 18, 1990.
⁹Source: kevra.org (Competing Hardware Comparisons), <https://simson.net/ref/NeXT/specifications.htm>, and "The Second Coming of Steve Jobs".

In 1992, they buffed up their entire line with Turbo versions and what would become the Gold Standard at id Software: The NeXTstation TurboColor.

3.4 NeXTcube

From the outside the NeXTcube's 12" cubic central unit looked exactly like its predecessor the NeXT Computer. However, the inside told a different story.

The CPU was bumped to a Motorola 68040 25Mhz, a chip capable of three times the 68030's throughput with 15 MIPS¹⁰. The machine's RAM capacity was doubled, with an out of factory 16 MiB, expandable to 64 MiB. The magneto-optical disc was abandoned in favor of a mandatory HDD, floppy disk reader, and an optional CD-ROM drive. The HDD capacity was augmented with a choice of 400 MiB, 1.4GiB or 2.8GiB SCSI drive. The floppy disks were twice the capacity of PCs at 2.88 MiB.



The NeXTcube Turbo released in 1992 was almost the same machine, except the 68040's frequency was bumped to 33 MHz and the max RAM capacity increased to 128 MiB.

The only weakness of the standard and turbo versions was the display. Shipping with 256 KiB of VRAM, the machine could only output four colors (white, black and two shades of gray). To bring color to the NeXTcube, customers had to invest in a NeXTdimension board.

Trivia : The NeXTcube Turbo expansion slot could welcome a Nitro board that replaced the 68040 33Mhz with a 68040 40 Mhz. Only 10 Nitro boards are known to exist, they are extremely rare and highly sought after by collectors.

¹⁰Source: "Fast New Systems from NeXT", B.Y.T.E Nov 1990



Figure 3.5: NeXTcube motherboard

Opening the machine revealed the minutiae NeXT had adopted as its standard. The NeXTcube motherboard above shows that aesthetics were not sacrificed for performance. Surface mounting allowed components to be placed much closer to others than usual.

Hidden under a heat-sink¹¹ and packing 1.2 million transistors, the CPU was a big step up. The Harvard architecture (separated storage and signal pathways for instructions and data), write-back capability, 8 KiB cache (4 KiB data and 4 KiB instructions), and integrated FPU tripled throughput compared to the 68030.

¹¹Heat dissipation was always a problem for the 68040 which prevented running at high frequencies, a handicap against Intel's 486 capable of 66 Mhz.

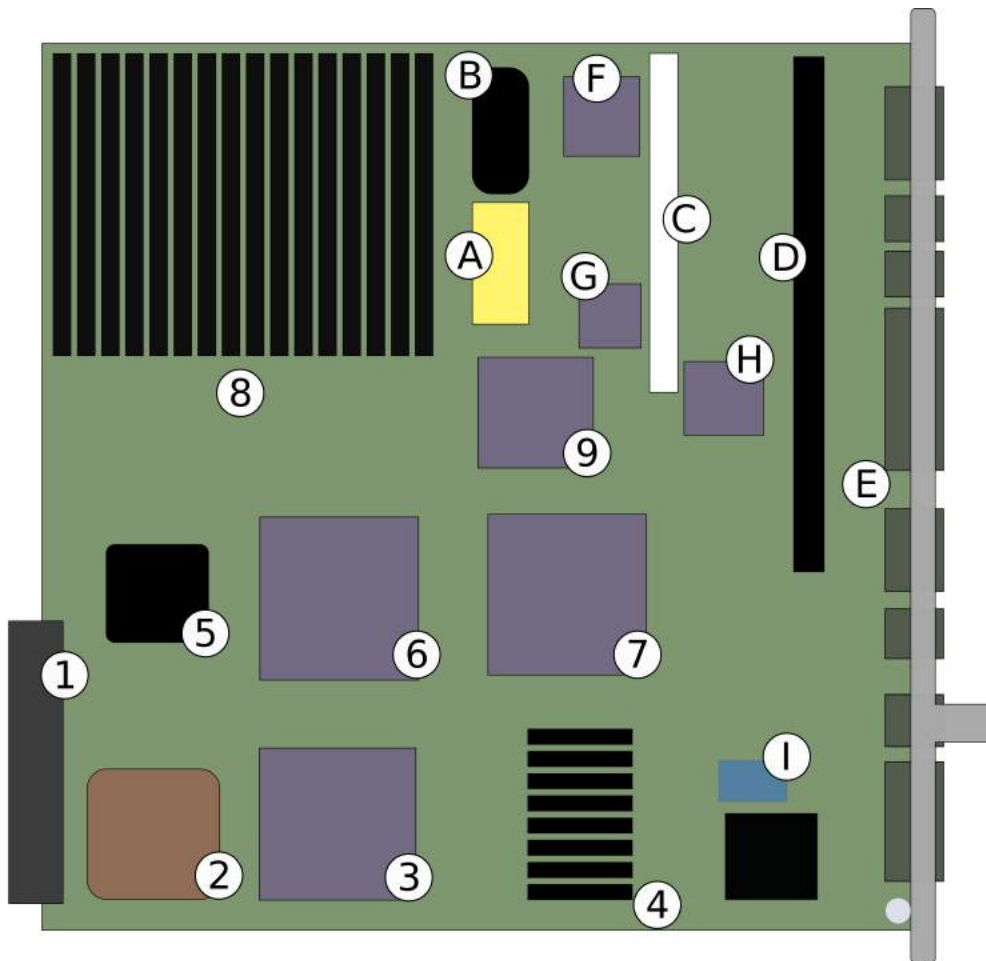


Figure 3.6: NeXTcube motherboard diagram

Chipsets and components of the NeXTcube motherboard:

① NeXTBus connector, ② VLSI NeXTBus Interface Chip, ③ CPU Motorola 68040, ④ 256 KiB VRAM, ⑤ DRAM Controller CS38PG017CG01, ⑥ Integrated Channel Processor (DMA Controller Fujitsu MB610313), ⑦ Optical Storage Processor (Fujitsu MB600310), ⑧ 16 SIMM slots max 4MiB each for total 64 MiB, ⑨ DSP-56001RC20, (A) Battery, (B) NeXT BIOS PROM, (C) DSP 768K Slot, (D) Hard-Drive and Floppy connectors. (E) Many connectors (top to bottom): 56001 DSP, Serial Port A&B, SCSI2, Printer, Ethernet RJ45&CoaxBNC, DB19 Monitor. (F) Intel n82077 Disk controller. (G) DSP SRAM (8KiB) MCM56824A. (H) SCSI Controller (NCR 53C90A) ① 100.000Mhz Oscillator K1149AA

3.5 NeXTstation

Since cost was the main issue with their product line, NeXT attempted to introduce a less expensive product. They designed something close to the NeXTcube but removed non-essential elements in order to produce a three times cheaper, all-in-one machine.

The NeXTstation's pricing and appearance made it a direct competitor to the SPARCstation. No longer a perfect cube, the case, nicknamed "the slab" (and also, banned by Steve Jobs, the "pizza box") was well-received by customers and became NeXT's most successful computer.

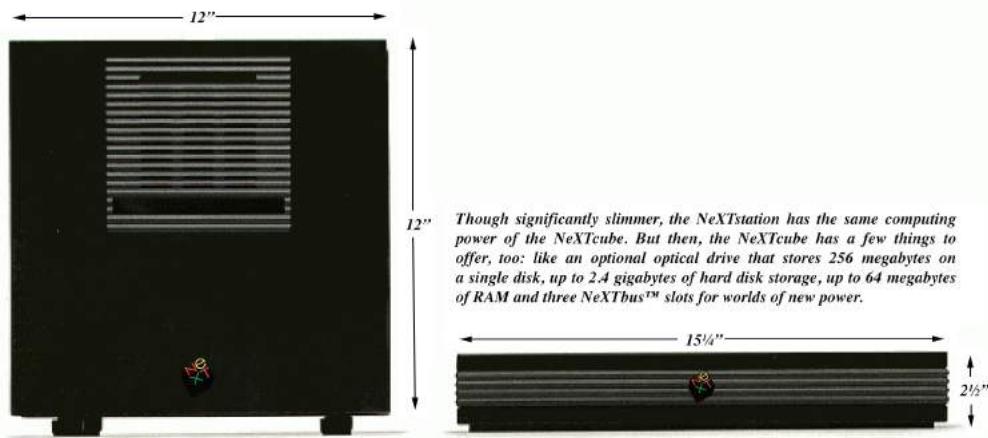


Figure 3.7: A NeXTstation ad from NeXTWorld 1991 magazine.

Designers picked elements from the NeXTcube and the NeXTdimension in order to produce an all-in-one, non-extensible machine. The three NeXTBus expansion slots were removed and so was the CD-ROM. A 2.88MiB floppy disk was added on the right side. The most notable difference came in the color version that had 2 MiB of VRAM, making the machine capable of 16-bit RGB colors. To accommodate the increased bandwidth requirements, the motherboard was redesigned to include a Bt463 RAMDAC.

“

The 16-bit color was only 4444, 1555 was not supported, which was unfortunate for us. It was also in a linear color space¹², as opposed to the non-linear PC standard that became sRGB. I didn't understand how to properly convert back then, so our graphics always looked washed out on the NeXT systems.

— John Carmack

”

¹²NeXT, SGI, and Apple had linear color space. PC of course did not.



Figure 3.8: A NeXTstation TurboColor (non-ADB)

In an unmistakable Steve Jobsian fashion, several components were renamed to emphasize subtle differences. The central unit fan was called a "whisper fan"¹³. The power supply was a 120-watt unit using a new technology called "parallel resonant switching" that allegedly allowed a much smaller form factor than conventional power supplies.

Despite its reduced size, the NeXTstation's performance didn't suffer compared to the more imposing NeXTcube. The four variants – NeXTstation, NeXTstation Color, NeXTstation Turbo, and NeXTstation TurboColor – all relied on a 68040 with 12 MiB of RAM.

Trivia : No button or switch are visible on the computer itself. The machine could only be turned on and off via the keyboard (a novelty at the time). An update later introduced the Apple Desktop Bus created by Steve Wozniak which bears many similarities to the USB standard released in 1996.

¹³Source: "Fast New Systems from NeXT", B.Y.T.E Nov 1990

3.6 NeXTdimension

The 256 KiB of VRAM on the NeXTcube only allowed a mediocre four shades of gray. The NeXTdimension was to take the workstation to a whole new level. Shipping with 4 MiB of VRAM and 8MiB of RAM (extensible to 32 MiB), it allowed a 24-bit color per pixel GUI and real-time recording/playback of video signals. Since the board was connected via a NeXTBus port, up to three NeXTdimensions could be connected, allowing the NeXTcube to drive four extended screens simultaneously.

On presentation day, Steve Jobs managed to demonstrate the groundbreaking capabilities in his signature spectacular fashion. A sequence from the black-and-white movie *Alice in Wonderland* was played live on a NeXTcube, already a tour-de-force at the time. The audience was impressed yet the best was to come. As *Alice* progressed through *Wonderland*, frames progressively turned to color. The audience went berserk.

At some point the NeXTdimension was even planned to feature real-time video compression, but problems prevented it.

“ NeXT has eliminated the C-Cube Microsystems CL-550 JPEG chip from NeXTdimension. This is because our supplier, C-Cube Microsystems, has failed to deliver chips that meet their specifications.

— Felipe_Fuster@NeXT.COM

”

The NeXTdimension was not a mere expansion board, but rather a full-featured computer within the computer. It had its own operating system, RAM, and clock generator which communicated with NeXTSTEP via Mach messages.

“ The NeXTdimension ran a custom kernel which was designed to do soft real-time management of multiple threads within a single address space, provide demand paged virtual memory, and provide a source-compatible Mach API subset and full Mach messaging interface, along with a minimal UNIX system call API, just enough to implement the RenderMan back end and the PostScript device layer. The kernel was called "Graphics aCcelerator Kernel, or "GaCK". Yes, this was a jape at the funny capitalization of the company name. It was not Mach, or BSD, or Minix, or Linux.

— M Paquette, NeXT Engineer

”

The card came with the `NeXTtv.app` which allowed video editing and frame capture.



Because of Steve Jobs's "hobby" venture with Pixar, the NeXTdimension had close ties with RenderMan. It had a built-in hardware acceleration module called Quick RenderMan.

“ Depending on the setup of the Renderman context, a RIB stream can be spooled to Photorealistic Renderman running on the host CPU (the m68K for black hardware), or to a Quick Renderman implementation loaded on demand into the Window Server. The Quick Renderman implementation in the Window Server may then, if the target window is on a NeXTdimension, forward the rendering operations to a Quick Renderman context running on the NeXTdimension board.

— M Paquette, NeXT Engineer

”

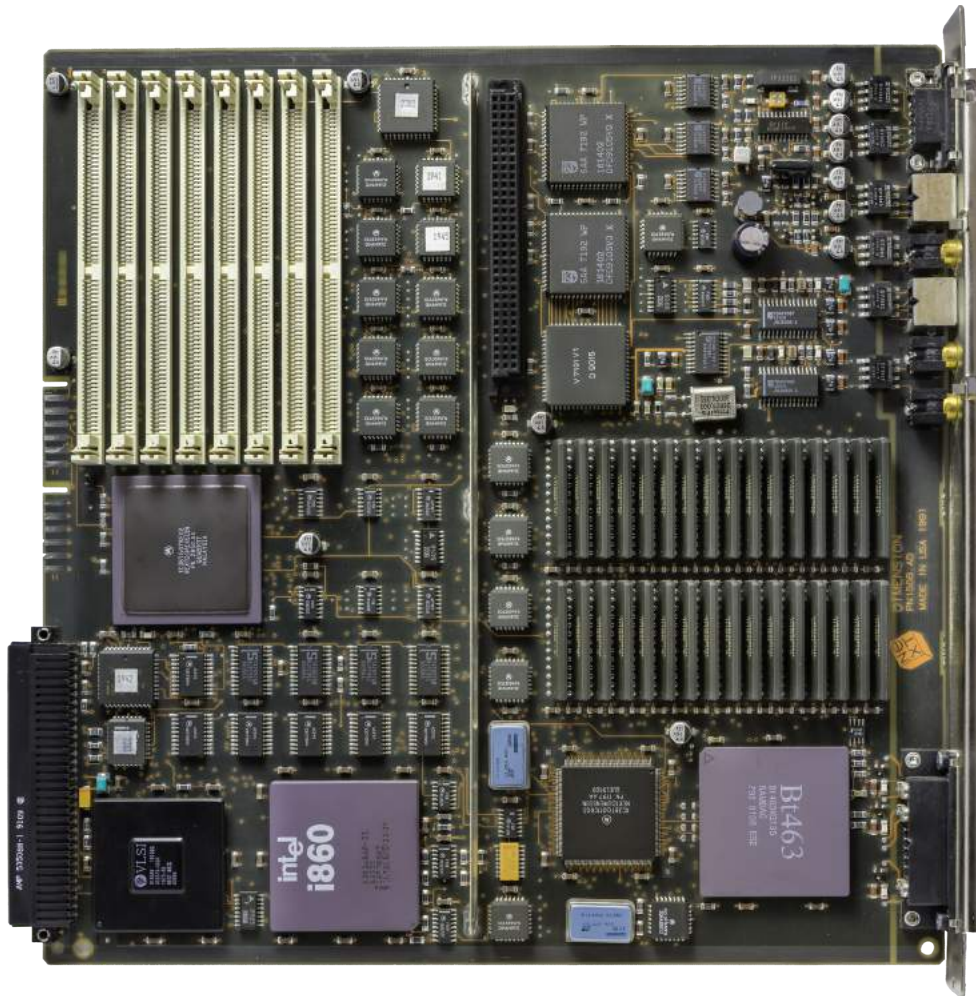


Figure 3.9: NeXTdimension board

Like the NeXTcube and NeXTstation motherboards, the NeXTdimension hardware was gorgeous and benefited from the same "surface mount" manufacturing process. The most prominent component is of course the Intel 860.

It had failed to beat Intel's 486 CPU in the market but its eagerness to participate in the DOOM phenomenon allowed it to land a gig on the tools team.

Trivia : Notice the Bt463 RAMDAC which would also be used on NeXTstations.

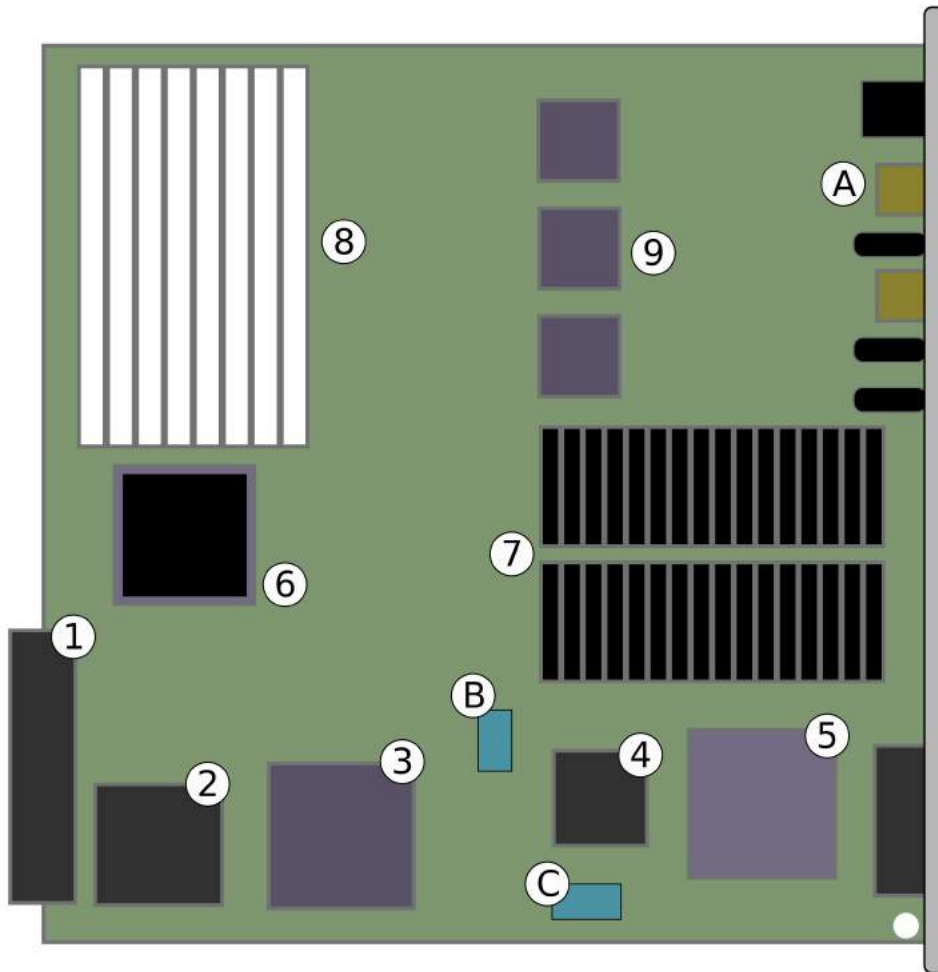


Figure 3.10: NeXTdimension board diagram

List of chipsets and components of the NeXTdimension motherboard:

① NeXTBus connector, ② VLSI NeXTBus Interface Chip (NBIC), ③ Intel i860 CPU 33MHz 64-bit RISC CPU, ④ Motorola U88 Memory Controller, ⑤ Bt463 Ramdac, ⑥ Motorola U52 Data Formatter, ⑦ 4 MiB VRAM, ⑧ SiMM RAM extension slots up to 8x4=32MiB, ⑨ Video color space conversion and video input (SAA 7191 WP & SAA 7192 WP), (A) Many connectors (top to bottom): Video Out(EGA/VGA, S-Video, Composite), Video In(S-Video, Composite, Composite), DB19 Monitor, (B) 33.000Mhz Oscillator K1100AA, (C) 100.000Mhz Oscillator K1149AA

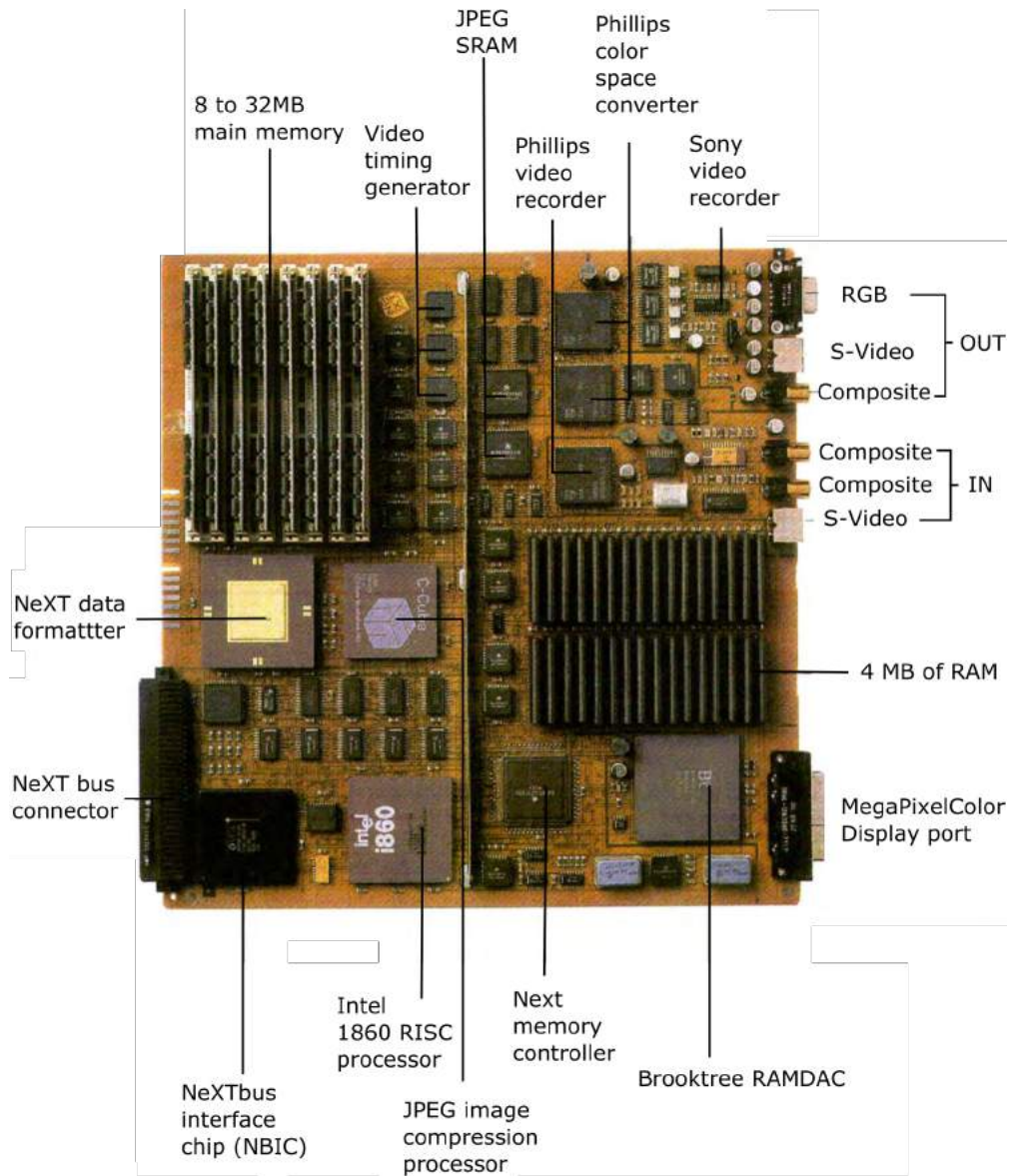


Figure 3.11: An early NeXTdimension ad from NeXT Magazine 1991.

Notice the C-Cube Microsystems JPEG chip which had to be cut from the final design, and the different resin, resulting in a different color for the motherboard. No doubt the color tone was the subject of much debates at NeXT headquarters. The "unlucky forever" Intel i860 was mislabeled "i860".

To select the i860 as the main chip for a graphics card could have been called overkill but it was a careful and ultimately sound decision. Video processing is an intensive CPU task. It benefits so much from the i860's SIMD pipeline that nothing else but Intel's "Cray on a Chip" could have done the job.

“

The Intel 80860 was an impressive chip, able at top speed to perform close to 66 MFLOPS at 33 MHz in real applications, compared to a more typical 5 or 10 MFLOPS for other CPUs of the time. Much of this was marketing hype, and it never became popular, lagging behind most newer CPUs and Digital Signal Processors in performance. The 860 has several modes, from regular scalar mode to a super-scalar mode that executes two instructions per cycle and a user visible pipeline mode (instructions using the result register of a multi-cycle op would take the current value instead of stalling and waiting for the result). It can use the 8K data cache in a limited way as a small vector register (like those in supercomputers). The unusual cache uses virtual addresses, instead of physical, so the cache has to be flushed any time the page tables change, even if the data is unchanged. Instruction and data buses are separate, with 4G of memory, using segments. It also includes a Memory Management Unit for virtual storage.

The 860 has thirty two 32 bit registers and thirty two 32 bit (or sixteen 64 bit) floating point registers. It was one of the first microprocessors to contain not only an FPU as well as an integer ALU, and also included a 3-D graphics unit (attached to the FPU) that supports lines drawing, Gouraud shading, Z-buffering for hidden line removal, and operations in conjunction with the FPU. It was also the first able to do an integer operation, and a (unique at the time) multiply and add floating point instruction, for the equivalent of three instructions, at the same time.

However actually getting the chip at top speed usually requires using assembly language - using standard compilers gives it a speed closer to other processors. Because of this, it was used as a co-processor, either for graphics, or floating point acceleration, like add in parallel units for workstations. Another problem with using the Intel 860 as a general purpose CPU is the difficulty handling interrupts. It is extensively pipelined, having as many as four pipes operating at once, and when an interrupt occurs, the pipes can spill and lose data unless complex code is used to clean up. Delays range from 62 cycles (best case) to 50 microseconds (almost 2000 cycles).

— John Bayko's "Great Microprocessors of the Past and Present"

”

3.7 NeXTSTEP

NeXT's 1990 24-pages brochure, "Welcome to the NeXT decade" introducing the NeXT Computer System, laid out the seven pillars of the system they were about to build.

“

Our collaboration with Higher Education provided the insight needed to visualize the seven breakthroughs that would ultimately define the NeXT Computer:

1. A new architecture optimized for total system throughput, not just individual component benchmarks.
2. A pioneering technology for vast and reliable storage, opening the door for new ways to access and use information.
3. Built-in CD-quality sound, allowing sound to be integrated into applications that are used every day.
4. A unified imaging system - Display PostScript - for both the display and the printer. So what you see on the screen is unequivocally what you get on paper.
5. An intuitive interface that gives everyone access to UNIX, with all of its power for networking and multitasking.
6. A multimedia mail system that enables communication combining text, graphics, and voice.
7. A new development environment that dramatically cuts the time it takes to create and customize software.

”

The first three points were the hardware team's responsibility. Everything else was on the software team and the amount of work ahead of them was overwhelming. The magical operating system described did not exist. To deliver their vision, they had to build it.

To create an OS from scratch would have been a humongous effort. To save time, the software team decided to reuse available components. They selected a microkernel called Mach from Carnegie Mellon University and combined it with elements from BSD (from University of California, Berkeley), such as the network stack, multi-user, multi-processing and filesystem. That was enough to bring the machine up to a command prompt.

3.7.1 GUI

To achieve the "unified imaging system", NeXT engineers started from PostScript – the language designed by Adobe for high-end printers – and modified it to meet the need of a GUI in terms of look-and-feel and performance. The result was called Display PostScript¹⁴.

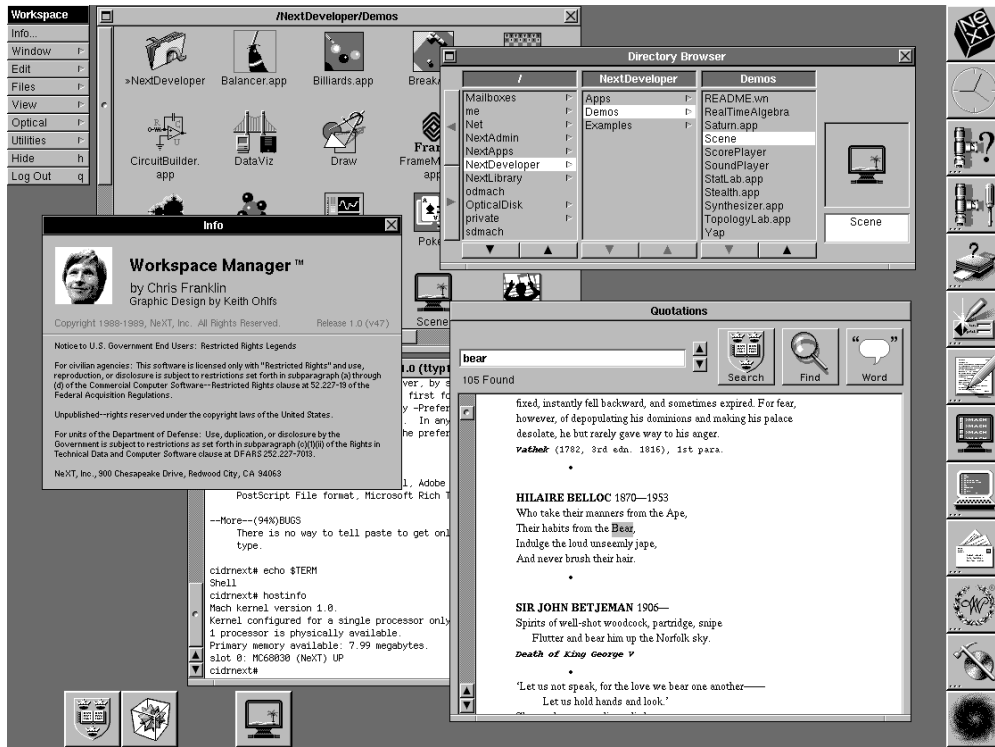


Figure 3.12: NeXTSTEP 1.0 running on a NeXT Computer in four color mode.

The resulting graphical system had many impressive features.

Some were raw power accomplishments, like for example the ability to see the content of a window while moving it (in all competing GUI-based OSes, windows had to be moved with only the outline visible because of graphic card and bandwidth limitations). Others were purely based on superior design and so solid that they remain in one form or another all the way up to Mac OS X running on 2018 Apple computers. The desktop metaphor, the unified titlebar scheme, the Dock, the bundles, and File Manager column view flow are only the most famous in a long list of GUI innovations.

¹⁴When OpenSTEP was used to build Rhapsody at Apple, the display system was changed to Portable Document Format (PDF) imaging model.

“

The Display PostScript system can be broken into two pieces, the PostScript interpreter and the device. The interpreter processes the language, and passes marking, imaging, and compositing directions to the device layer.

The device layer takes the high level marking, imaging, and compositing operations and (eventually) converts these to bitmap level operations. The Display PostScript system spends the majority of its time down here. In the case of the NeXTdimension board, the device layer is implemented on the NeXTdimension board. Marking, imaging, and compositing operations are asynchronously transmitted to the NeXTdimension for processing while additional PostScript is interpreted on the 68K processor. A good degree of parallelism is achieved in normal operation.

— M Paquette

”



Figure 3.13: NeXTSTEP running on system with a 24-bit NeXTdimension

Figure 3.13 shows NeXTSTEP running on a 24-bit color machine. The composition was more gorgeous than what any of its competitors was capable of. Notice `Mail.app`, which shipped with an email from Steve Jobs lauding the merits of object-oriented programming.

3.8 NeXT at id Software

To say NeXT polarized professionals would be an understatement. It is fair to say that everybody had a strong opinion about them. Some developers hated it.

“ Develop for it? I'll piss on it!
— **Bill Gates**¹⁵. ”

Some were interested in having access to a stable Unix system with a powerful GUI.

“ When id Software was stationed in Madison, Wisconsin during the winter of 1991, most of us were gone for the Christmas holiday - except John Carmack. John's present, which he bought with \$11,000 of his own money, procured by walking through the snow and ice to remove from the bank¹⁶, arrived during the holiday and he spent the whole time learning as much as he could about the computer and started working on vector quantization algorithms for compressing graphics. His test graphic was a 256-color screen from King's Quest 5.

After his research was done it was agreed that the entire company needed to develop our next game on NeXTSTEP.

— **John Romero, rome.ro, December 20, 2006** ”

Given the timing of his purchase, the NeXT was first used to produce the Wolfenstein 3D hint book. One of the best DTP applications at the time, `FrameMaker.app` proved perfect for the task. After that, NeXT rapidly conquered id Software for all other operations. PCs remained in use mostly for Deluxe Paint and to test the game they wanted to ship.

As the needs of the studio evolved with more and more power, id took advantage of NeXTSTEP's ability to run on various platforms from HP and Intel (called the "White hardware").

¹⁵Found in Walter Isaacson's book "Steve Jobs" but with no source. Bill Gates may have never said that.

¹⁶Emptying his bank account in the process.

The studio bought so much hardware over the years that twenty-five years later, accounts differ about which was the first kind of NeXT machine purchased. According to John Carmack it was a NeXTstation.

“

I bought our first NeXT (a ColorStation) just out of personal interest. Jason Blochowiak had talked to me about the advantages of Unix based systems from his time at college, and I was interested in seeing what Steve Job's next big thing was. It is funny to look back - I can remember honestly wondering what the advantages of a real multi process development environment would be over the DOS and older Apple environments we were using. I remember saying "What else would you do when the compiler was running?". Jason was ahead of the game when we first met; he was using an expensive Mac II system to cross develop for the lower end Apple IIGS. He was of course proven right on the value proposition. Actually using the NeXT was an eye opener, and it was quickly clear to me that it had a lot of tangible advantages for us, so we moved everything but pixel art (which was still done in Deluxe Paint on DOS) over. Using Interface Builder for our game editors was a NeXT unique advantage, but most Unix systems would have provided similar general purpose software development advantages (the debugger wasn't nearly as good as Turbo Debugger 386, though!). Kevin Cloud even did our game manuals, starting with Wolfenstein 3D, in Framemaker on a NeXT.

This was all in the context of DOS or Windows 3.x; it was revelatory to have a computer system that didn't crash all the time. By the time Quake 2 came around, Windows NT was in a similar didn't-crash-all-the-time state, it had hardware accelerated OpenGL, and Visual Studio was getting really good, so I didn't feel too bad moving over to it. At that transition point I did evaluate most of the other Unix workstations, and didn't find a strong enough reason not to go with Microsoft for our desktop systems.

Over the entire course of Doom and Quake 1's development we probably spent \$100,000 on NeXT computers, which isn't much at all in the larger scheme of development. We later spent more than that on Unix SMP server systems (first a quad Alpha, then an eventually 16-way SGI system) to run the time consuming lighting and visibility calculations for the Quake series. I remember one year looking at the Top 500 supercomputer list and thinking that if we had expanded our SGI to 32 processors, we would have just snuck in at the bottom.

— John Carmack, [quora.com](https://www.quora.com/John-Carmack)

”

John Romero remembers first buying a monochrome NeXTcube.

“ The NeXTCube was purchased in December 1991 and was the only NeXT Computer we had until December 1992 when we decided we would develop DOOM with them so we bought 3 NeXTStations: mine, John Carmack's new one, Tom Hall's. John C's original NeXTCube was the computer used to scan the clay models, gun toys, and latex models.

id's first NeXT hardware was all black - both Cubes and Stations. We upgraded through the years to the Turbo model then to other hardware like the HP Gecko and then Intel hardware at the end. We were building fat binaries of the tools for all 3 processors in the office - one .app file that had code for all 3 processors in it and executed the right code depending on which machine you ran it on. All our data was stored on a Novell 3.11 server and we constantly used the NeXTSTEP Novell gateway object to transparently copy our files to and from the server as if it was a local NTFS drive. This was back in 1993!

In fact, with the superpower of NeXTSTEP, one of the earliest incarnations of DoomEd had Carmack in his office, me in my office, DoomEd running on both our computers and both of us editing one map together at the same time. I could see John moving entities around on my screen as I drew new walls. Shared memory spaces and distributed objects. Pure magic.

— John Romero

”

John Romero in particular liked their production pipeline so much that he decided to champion it. He managed to successfully advertise it to another gaming studio.

“ DOOM, DOOM II and Quake weren't the only games developed on NeXTSTEP. When I got Raven Software to agree to develop Heretic for us I had them buy several Epson NeXT computers (Intel based) and I flew up to Madison, WI to get them all set up and teach them how to develop the game with our tools and engine. It was a great time I'll never forget - seeing their team get excited about the power of the new environment and that they got the game developed and released in under a year. They signed on for another title and developed Hexen on NeXTSTEP as well.

— John Romero, rome.ro, December 20, 2006

”

3.9 Roller coaster

During its seven years in business, NeXT lead a tumultuous life. It was sued by Apple within its first month of existence (the five people Steve Jobs had taken with him were not "minor people" as he had told Apple). Carried on by Steve Jobs' "reality distortion field", the company was praised for several years even though it had yet to produce anything. Glorified upon each release, owing a lot to marketing genius, the machines later struggled to find customers. Disappointing sales led to a near death experience before NeXT managed to re-invent itself.

3.9.1 Downfall

As elegant and powerful as the black hardware was, their high price tag made them a deal-breaker. Even the "cheaper" NeXTstations were well beyond most developers' budgets.

In 1988, the factory was building 400 units/month, well below its maximum 10,000/month capacity. Sales worsened in 1989 with only 360 units/month sold over the year. Production had to be slowed down to 100 units/month to avoid overflowing storage. By 1990 things improved slightly with \$28 million in revenue – still a far cry from the \$2.8 billions Sun Microsystems generated that same year. 1991 saw yet another improvement with 20,000 units sold and a revenue of \$127 million. That figure was still less than what Apple sold in a single week. By 1992, sales reached \$140 million¹⁷ and NeXT claimed its first profitable quarter¹⁸, seven years after its founding.

Despite the steady improvements, NeXT still lost \$40 million that year. With only 50,000 NeXT machines sold over the course of its short life¹⁹ (including thousands to the then secret National US Reconnaissance Office), Jobs decided that NeXT could not carry on as a hardware manufacturer. Struggling to close deals and hemorrhaging cash, it fired 300 out of its 500 employee workforce on a Black Tuesday of February 1993 to become a software-only company.

The purpose of NeXTSTEP was changed. From operating system in charge of making the black hardware sing, it was to become a white hardware enabler and the sole money maker. It was re-factored to be portable and capable of running on Intel, SPARC, and PA-RISC CPUs. Sold as a combination operating system and object-oriented development environment. NeXTSTEP for Intel became a popular product among large companies and especially financial institutions for rapidly developing and deploying custom software. NeXT also started to collaborate with Sun Microsystems on a light version of NeXTSTEP called OpenSTEP.

¹⁷Source: The Next Big Thing.

¹⁸In fact it was \$40 million from breaking even. Source: "The Next Big Thing"

¹⁹In 1993 Apple sold 50,000 units every six days.

Even in its darkest hours, NeXT curiously never capitalized on id Software's appreciation for the platform²⁰. Reportedly due to Steve Jobs disdain for video games, they even turned down an opportunity which could have helped them tremendously.

“ We loved our NeXTs, and we wanted to launch Doom with an explicit "Developed on NeXT computers" logo during the startup process²¹, but when we asked, the request was denied.

— John Carmack

”



Figure 3.14: DOOM alpha version credit screen featuring "NeXT Computers"

As DOOM's success grew and became a world-wide hit, NeXT backtracked and attempted to reverse its decision but by then, as recalls John Carmack, "that ship had sailed"²².

²⁰comp.sys.next.advocacy: "DOOM: NeXTstep's Most Successful App".

²¹Tom Hall's "DOOM Bible" mentioned designing maps with labs featuring "a lot of NeXT-looking computers".

²²"I showed up for him" by John Carmack. facebook.com May 14th, 2018.

3.9.2 Rebirth

The rest of NeXT's history is the envy of a Hollywood plot-twist. In 1995, Apple Computer's failed attempt at producing its own operating system under project "Copland" had placed the company in a precarious situation with nothing to replace its outdated System 7.

After briefly considering BeOS, Apple elected to buy NeXT in 1996 for \$429 million in cash. The timing could not have been better for NeXT which was on the verge of bankruptcy. Steve Jobs returned to the company he had co-founded twenty years earlier. After the sale, he first worked as an advisor but was later appointed acting-CEO, to finally become CEO of the company. This was not only a rebirth for NeXT, it was also a rebirth for Apple, which went from being ninety days away from insolvency²³ in 1996 to most valuable company in the world in December 2017.

NeXTSTEP was used as foundation for Apple's Rhapsody project which became Darwin, the core of Apple's OSes. The new operating system was met with enthusiasm by customers and professionals who praised the design, look and feel, and stability. Darwin was later used as a base for Apple's 2008 iPhone which took the world by storm. It is extremely likely some of the code that once ran on NeXTSTEP and contributed to DOOM now runs on the many millions of Apple computers and iOS phones across the world.

²³Source: "All Things Digital conference, Jun 1, 2010."

Chapter 4

Team and Tools

After shipping Wolfenstein 3D in May 1992, id Software went right back to work. The team of five (John Carmack, John Romero, Adrian Carmack, Tom Hall and Kevin Cloud) split in two. While part of the team would focus on the Wolfenstein 3D's Hint Manual and Spear of Destiny, another faction built the technology which would power the next title.

The development of DOOM really started in January 1993 with an impressive press release by Tom Hall (read it in Appendix D on page 391) promising ground-breaking technology and unprecedented gameplay. Within just eleven months they managed to have the shareware version ready in time for Christmas. Fourteen people would end up being involved.

Name	Age	Occupation
John Carmack	22	Programmer
John Romero	24	Programmer / Designer
Adrian Carmack ¹	23	Artist
Tom Hall ²	29	Creative Director
Jay Wilbur	31	Business
Kevin Cloud	28	Computer Artist
Donna Jackson	55	id's Mom
Dave Taylor	24	Programmer
Sandy Petersen	39	Designer
Shawn Green	28	Software Support
American McGee	22	Tech Support
Paul Radek	28	DMX Audio library
Gregor Punchatz	27	Artist, clay models
Bobby Prince	39	Music composer

¹The two "Carmacks" in the team are not related.

²Due to creative differences, Tom Hall left six months after completing the "Doom Bible" design doc. He went to Apogee/3D Realms to work on "Rise of the Triad" using an improved version of Wolfenstein 3D engine.

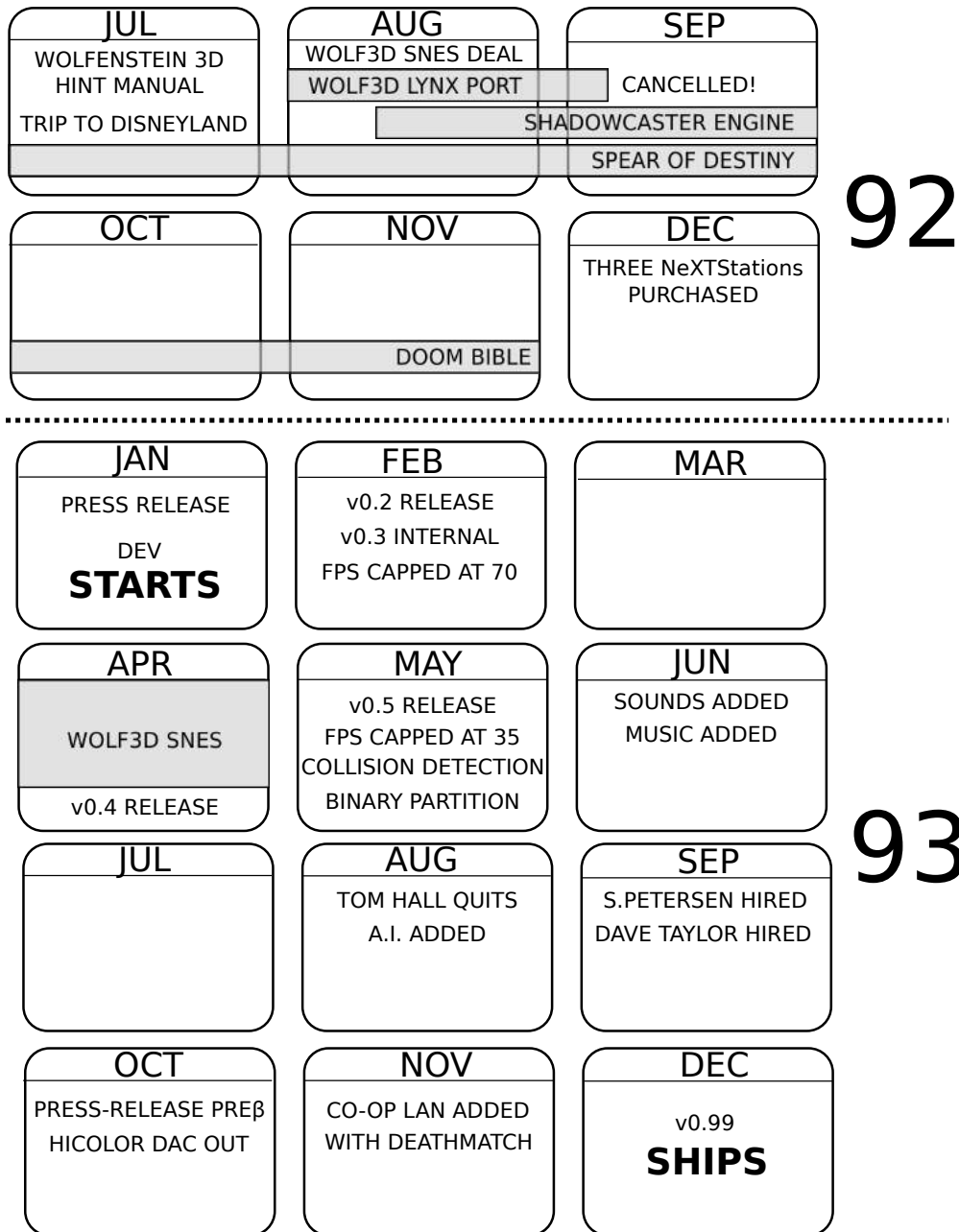


Figure 4.1: Making of DOOM timeline



In January of 1995, Electronic Games magazine ran a series of three articles for the release of DOOM II. This presented an opportunity to gather all members of the id team in a group photo.

In the back row, left to right: Kevin Cloud, American McGee, John Carmack, Adrian Carmack, and Sandy Petersen. Front row, left to right: Dave Taylor, John Romero, and Shawn Green.

The "plank" in the photo is John Romero's office door and the hole was the making of John Carmack.



“ Well, Romero's door jammed one day. He was in his office and was trapped in there, and we couldn't get the door open. It was after-hours, so we couldn't call building maintenance, and we were all standing around trying to figure out what to do, when it occurred to me and I said, "You know, I do have a battle axe in my office."

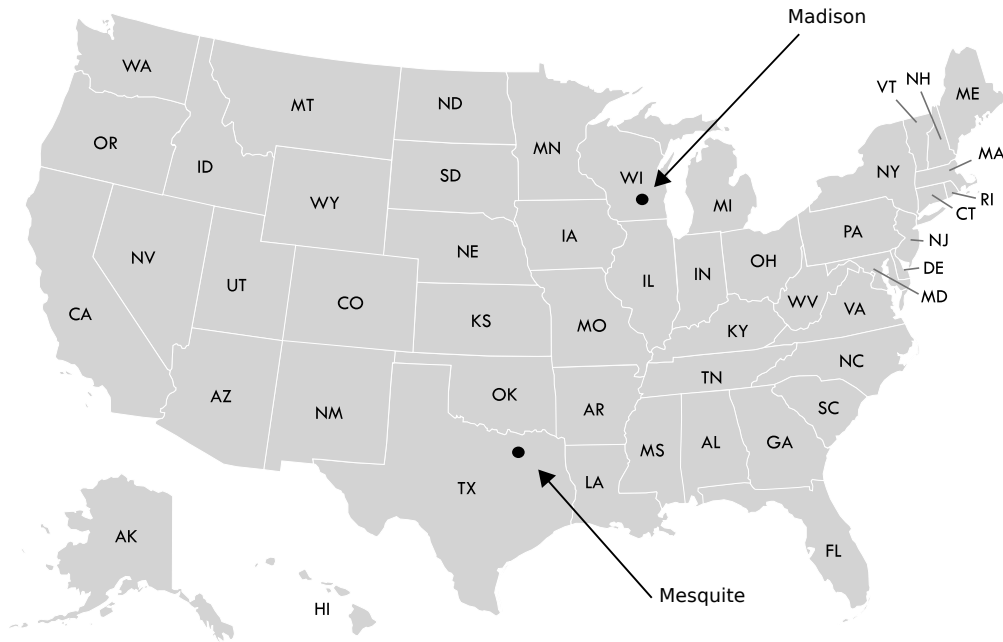
Yes, this thing really works.

— John Carmack

”

4.1 Location

Fed up with the winter of Wisconsin, the studio relocated on April 1st, 1992. They left behind the cold of Madison and settled in the Town East Tower in Mesquite TX, better known as the "Black Cube".



In an interview for the book "Masters of Doom", Tom Hall recounts how all the team members settled into the new environment.

“

On the first day, each guy chose his space. Carmack and Romero took side-by-side offices, while Adrian and Kevin, who were growing increasingly close, decided to share a space. Tom liked an open corner spot in a large room with a window. "This would be a great office area," he said, "we just need to put some walls up." The rest agreed. But the walls were slow to come. Whenever Tom asked Jay about it, Jay would say they were on their way. Out of humor and frustration, Tom put down two long strips of masking tape where the walls of what he called his creative corner would go.

— David Kushner, *Masters of Doom*

”

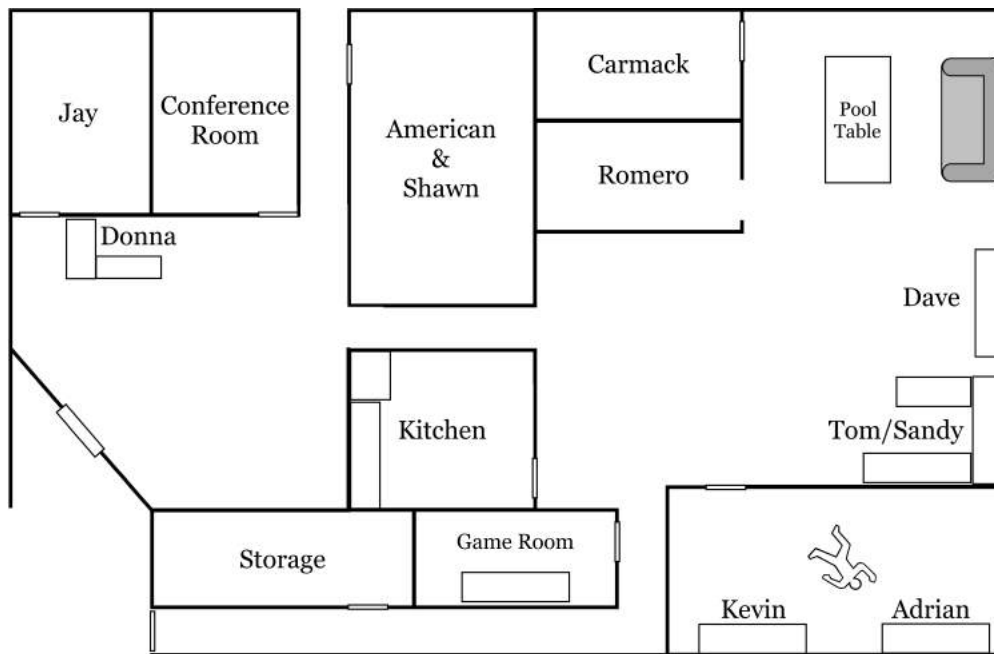


Figure 4.2: Id Software office layout as can be seen in John Romero's mini-documentary "A Visit to id Software (1993)".

Dave Taylor remembers id Software's unorthodox spirit at the time.

“ I fell asleep on the floors a lot, which is why they got the sofa and had me test-drive it for comfort, but I only remember them taping my outline once. I believe it stuck around for a while though.

— Dave Taylor

“ I remember companies would send us free sound cards all the time, so many that we took to using them as ninja throwing stars at one point and would throw them into the kitchen wall opposite my desk.

— Dave Taylor

4.2 Creative direction

The tone of the new title was to be much darker than the previous games. Initially intending to adapt the movie *Aliens* (1986), the team decided against it in order to retain total creative control³. They wanted to do something scary, inspired by movies such as *Evil Dead* or *The Thing*. They wanted it to be as aggressive as the metal music they occasionally listened to.

Even the name of the next title would be to inspire fear. The idea came to John Carmack while watching the movie, "The Color of Money".

Midway through the movie, Vincent, played by Tom Cruise, enters a pool bar carrying one of the best pool cues in the world, a Balabushka. Confident and committed on unleashing his skills upon his opponents, he is noticed by one of the locals, Moselle. Intrigued by the pool cue case he asks Vincent: "What you got in there?". "In Here?" exclaims Vincent, looking up and smiling maliciously: "DOOM!".



Figure 4.3: "In Here? DOOM!"

The game they envisioned would go way beyond what they had accomplished with *Wolfenstein 3D* and *Spear of Destiny*. *DOOM* would have eight weapons and seventeen types of opponent. The hero would fight countless demons. And there would be a lot of gibs⁴.

³Amusingly, one of the best mods of all time would be the total conversion "Aliens TC".

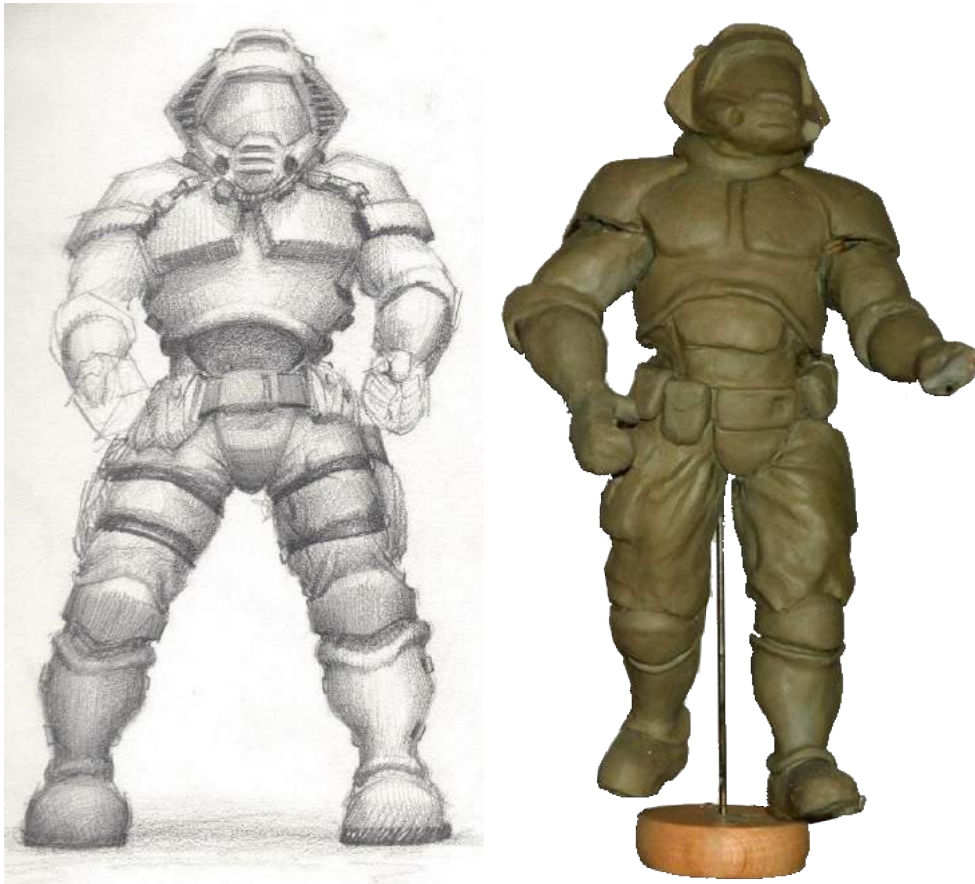
⁴"gibs" is short for "giblets" and there were often arguments about the correct pronunciation.

4.3 Graphic assets

4.3.1 Sprites

Given the ambitions, there was a lot of artwork to produce. Weapons were animated when firing. Monsters had to have eight poses depending on the viewing angle. Wall surfaces, ceilings, and floors were to be textured. And this is not to mention all the "utility" art for the menus, intermission and final screens. It was an immense undertaking for a team of only two artists working with just Deluxe Paint.

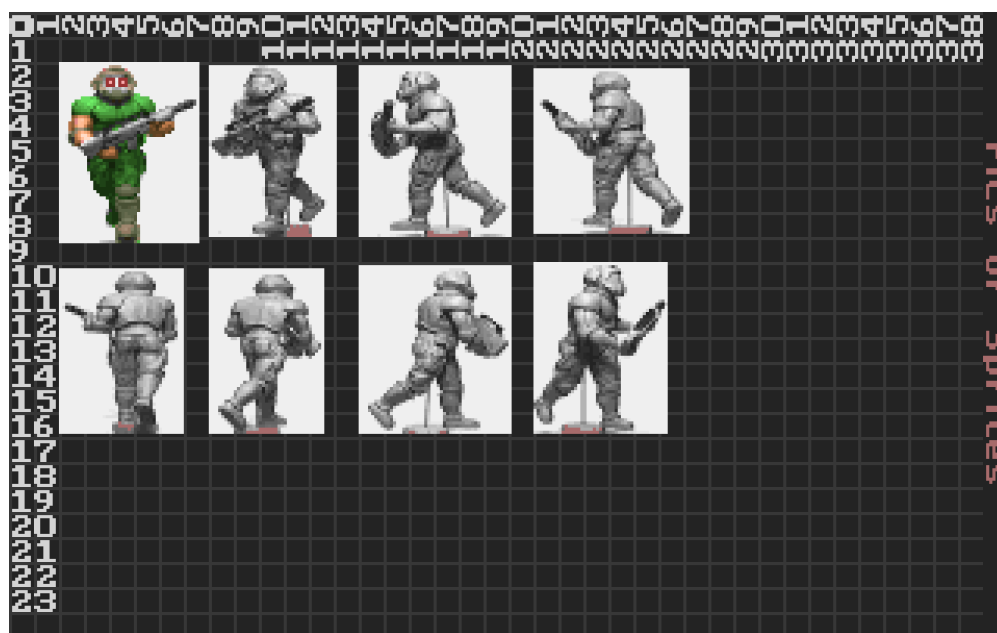
Monsters represented most of the workload. Drawing a single sprite while facing the player was easy. Drawing it seven more times at increasing angles (45°, 90°, 135°, 180°, 225°, 270°, and 315°) for each action was hell. To solve this problem they created a new process leveraging both their artistic talent and the technological power of the NeXTDimension. First they drew it on paper, then they applied clay on a small posable wooden mannequin.



Once the character was carved they could change the pose at will. They only had to connect a Handy-cam Hi8 Sony video camera to the NextDimension. Placed on a spinner, the clay model was lit and digitized from eight viewpoints. It was a much faster and a more fun process.



The output from the NeXTDimension was a 24-bit TrueColor image which had to be transformed to the DOOM palette⁵ of 256 colors via a tool called the "Fuzzy Pumper Palette Shop". To complete each sprite, artists performed touch ups and manual coloring with Deluxe Paint.



The process was not without its own flaws since the clay dried and had a tendency to break instead of folding. Nevertheless, seven Doom characters were built as sculptures for DOOM & DOOM II. The first models – the Doomguy, Baron of Hell and Cyberdemon – were all sculpted by Adrian Carmack. The iconic Imp, Zombieman, and Sergeant were all mouse-drawn by Kevin Cloud.

Trivia : Most models survived. Some are still in John Romero's possession while others are visible at id Software's headquarters. A few models managed to escape into the wild and are now highly-prized by collectors.

⁵See the DOOM palette on page 246.



Figure 4.4: A. Carmack sculpting the Baron of Hell, working from his preliminary drawing.



Using clay models was faster than drawing by hand but it was still not fast enough to produce the many monsters necessary. It also had limited capability in terms of textures since it was impossible to render specular material such as metal. There were also issues with intricate details which were way too fine to survive clay modeling. They needed better models, possibly stop-motion capable.

Don Ivan Punchatz, who had been commissioned for the DOOM package art and logo, mentioned he had a son doing exactly that. Greg had been successfully providing stop animation models for big Hollywood productions such as *A Nightmare on Elm Street 2*, *RoboCop* and its sequels, and *Coming to America*.

Kevin Cloud got in touch with Greg Punchatz and the young artist was promptly commissioned for the Arch-Vile, Mancubus, Revenant and Spiderdemon.

“ The spider creature was made out of parts I had literally just found at hardware and hobby stores, pieces of Tupperware and PVC pipes. The main body started out as a sculpture, then a plaster mold was pulled from that. Then we made the armature to fit that mold, and then foam latex was injected inside the mould and put into an oven.

Mastermind's legs pretty much only just moved, and his arms moved, but his mouth didn't move. As we went along, the other maquettes become full ball and socket armatures, so they had a full range of motion. In some ways, these stop-motion maquettes are easier to get right than they would be in CG. You don't have to worry about how your skin is weighted on stop-motion model because it just sticks to the metal armature.

— Greg Punchatz, Interview by develop-online.net Feb 16, 2016

”

Overall, Greg seems to have only fond memories except for one funny regret.

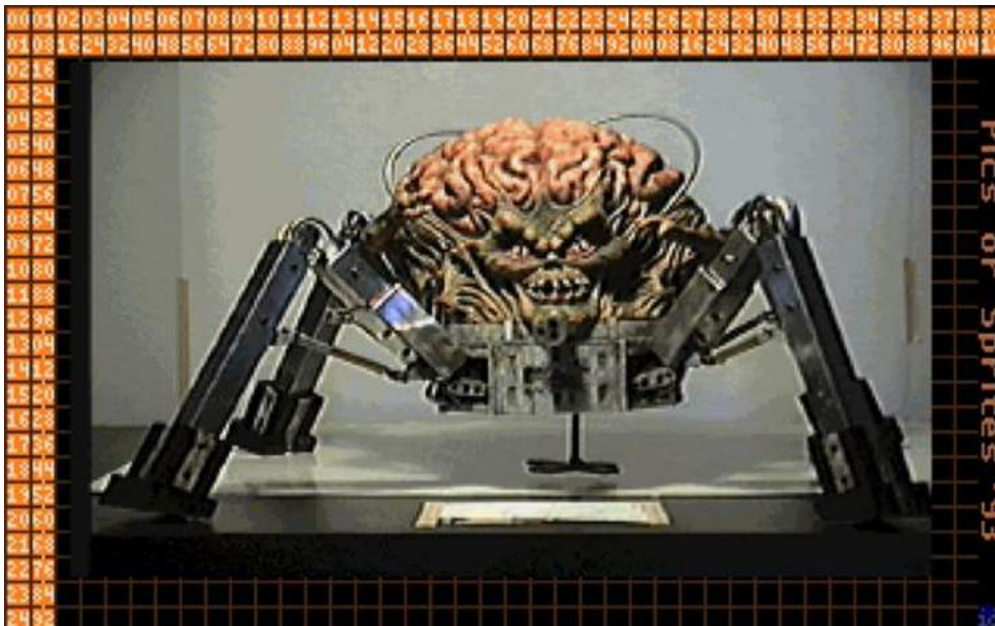
“ At one stage id offered me points on the backend to take \$500 off the price of one of the characters and I turned that down. It's a painful lesson. But to be part of something that has left a long-lasting impression on the world is kind of crazy – people find out that I worked on Doom and it's like I played on the Beatles' White Album.

— Greg Punchatz, Interview by develop-online.net Feb 16, 2016

”



Figure 4.5: Notice the spinner, camera, and a virgin wooden mannequin on the table



4.3.2 Weapons

The starting points for the weapons were mostly toys, digitized with the NeXTDimension and heavily cleaned up via Deluxe Paint.

The shotgun was in fact the "TootsieToy Dakota" cap shotgun, manufactured by the Strombecker Corporation of America.



Figure 4.6: TootsieToy Dakota. The thing had a rifle fire mode. (Courtesy of James Miller)

The chainsaw was a real, fully functional, McCulloch Eager Beaver. It was borrowed from Tom Hall's girlfriend.



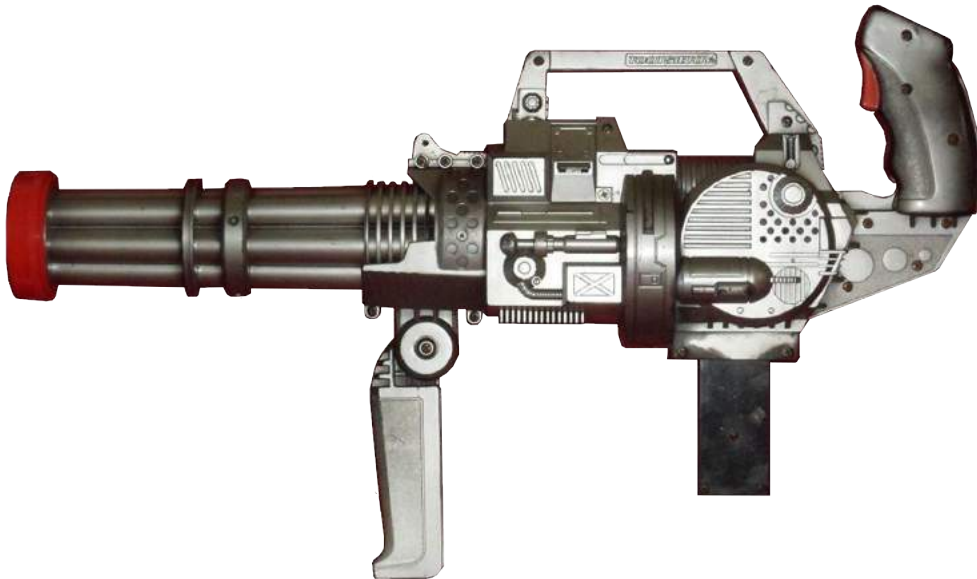
Figure 4.7: McCulloch Eager Beaver (Courtesy of James Miller)

Luckily it was only brought in after Romero locked himself up in his office. That could have been interesting.

Trivia : The chainsaw worked well but it leaked oil abundantly. They had to store it in a bowl on the ground.



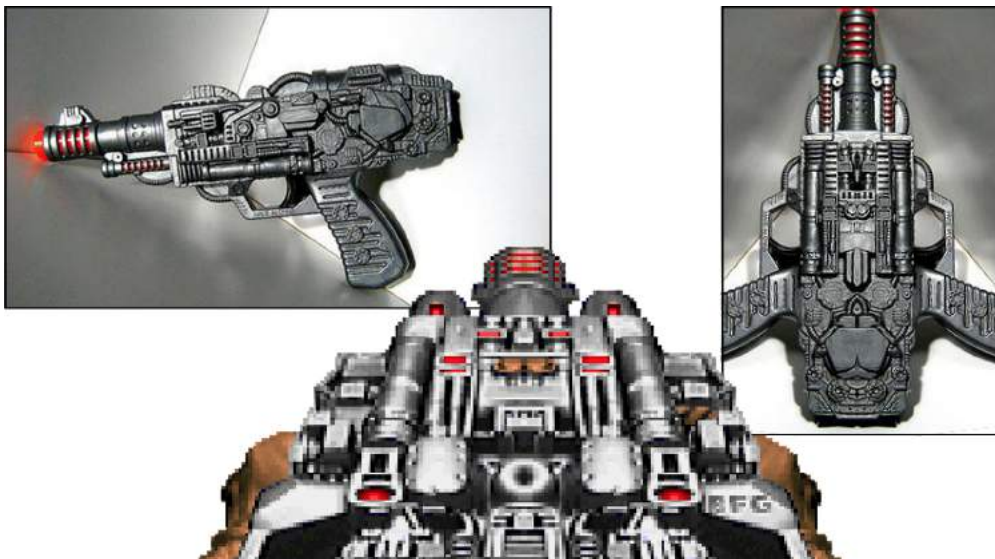
Below: the chaingun was another Tootsietoy toy called the Ol' Painless. To the delight of many parents it was able to produce loud firing sounds when fitted with a 9V battery.



The fist was actually Kevin Cloud's hands wearing a knuckle duster. The Plasma rifle was based on the grenade launcher of a Rambo III M-60 toy set (upper left element in the box).



The BFG 9000 was the RoarGun by Creatoy. It was photographed sideways, mirrored and inclined at an acute angle to give it more depth.



4.3.3 Skies

Since the player was to travel to several satellites of Mars during the game, matching SKY textures had to be produced.

In order to generate the "real" touch they wanted, Kevin and Adrian bought a set of 10 royalty free CD-ROMs called MediaClips. Each CD had a theme (Jets, Majestic Places, Props, Wild Places, Worldview) and the whopping 650MiB capacity allowed one hundred high-resolution (640x480) photos per CD.



Since they did not have much time for Episode I which had to ship with the shareware in December, they simply cropped Yangshuo Cavern from China to the sky standard 256x128 resolution. With more time for the other episodes they became more creative and composed images from numerous sources. The skylines of Phobos, Deimos and Hell ended up borrowing from places like China, Zion, and Hawaii.



Because the sky repeats four times in the engine, the texture had to be patched in order to wrap at the edges. Notice how the right and left edges connect without any discontinuity.

Trivia : Can you guess where the clouds in DOOM II Episode 2 skies are from?





DOOM, Episode II. Made of Zion's Watchman rock formation and a red tinted sunset.





DOOM, Episode I. Yangshuo Cavern in China. (All compositions courtesy of James Miller).





DOOM II, Map 1. Hawaii beach at sunset.





DOOM II, Map 13. Challenger rocket take-off used for burning city clouds.



4.4 Maps

Maps were designed in a top-down view. The one limitation was that walls were perpendicular to floors and both floors and ceilings were horizontal so maps were drawn in 2D. A designer worked with five types of element: VERTEX⁶, LINE, SIDEDEF, SECTOR, and THING.

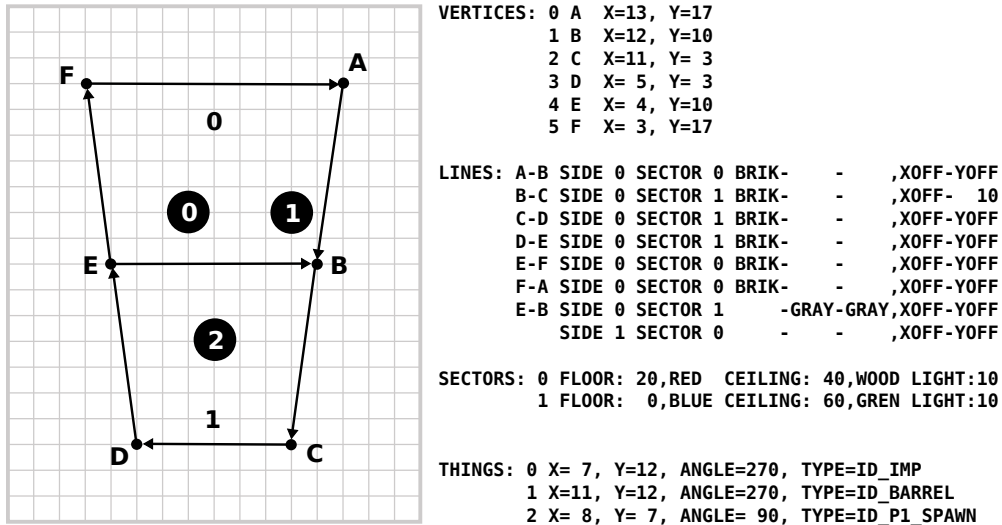


Figure 4.8: DOOM map and the resulting data authored via DoomED

A SECTOR is a closed area surrounded by LINES with a specified floor height, floor texture, ceiling height, ceiling texture, and light level. A sector can be concave, but lines cannot cross each other.

A LINE can either be a solid wall or a portal between two SECTORS. The difference is in the number of SIDEDEFs associated with it. A wall has only one SIDEDEF on its right side and is fully opaque. A portal has two SIDEDEFs and can usually be partially seen though.

A SIDEDEF describes one side of a LINE. To accommodate texturing of both the walls and portals, it can have up to three textures. The middle texture is used by walls for the full area they cover. A SIDEDEF can also have a lower and an upper texture for portals connecting SECTORS with different ceiling/floor heights. If the portal leads to a sector with higher floor, the lower texture is used to render the "step". If the SECTOR connects to a SECTOR with a lower ceiling, the upper texture is used to render the "down step". To help alignment of doors and buttons, SIDEDEF textures can have a vertical/horizontal offset.

⁶Vertex coordinates were expressed with signed short integers [-32768, 32767]. 32 units translate to roughly one meter (or 3.28 feet for poor souls who must use the imperial system.)

A THING is much simpler in comparison. It only features a 2D-coordinate X,Y, an angle, and an identifier controlling its type. At the bare minimum a map must have one player-spawning location THING.



Figure 4.9: Rendering of map shown in figure 4.8.

The resulting scene in figure 4.9 is ugly but the mismatched colors hopefully help to discern the different elements. All LINES are walls except for E-B which has two SIDEDEFs and is therefore a portal. All walls use the BRIK middle texture except for the portal which uses GRAY for both top and bottom.

SECTOR #0 uses a RED floor texture and a WOOD ceiling texture. The height of the floor is 20 and its ceiling is at 40. SECTOR #1 uses a BLUE floor texture and a GREEN ceiling texture. Its floor is at 0 and ceiling at 60. Both sectors have the same light level (10).

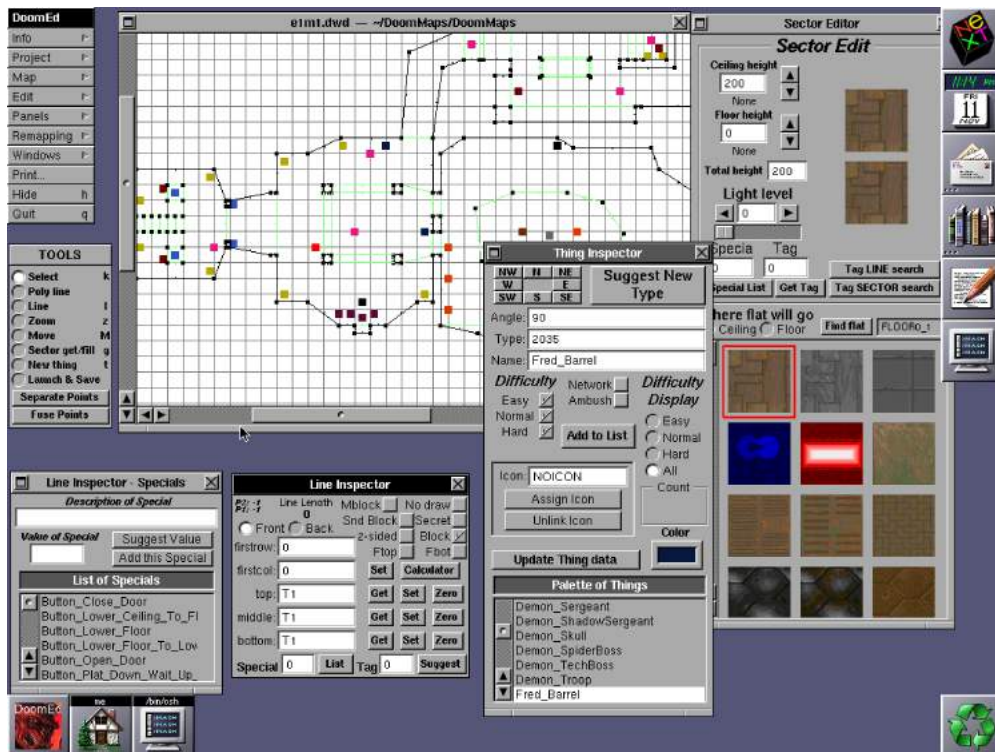
Notice the portal E-B which does not have a mid-texture but an upper and a lower texture. These were used to draw the up-step and down-step towards sector #0.

Also notice wall D-E on which the mid-texture vertical offset is not correctly set, resulting in a vertical tear when connecting with wall E-F. Wall B-C's vertical offset is properly set and has no visual artifact. None of the walls use a horizontal offset, but the corresponding field is labeled XOFF on figure 4.8 to show its location.

4.4.1 Map Editor (DoomED)

To harness the complexity of the map format, a new tool was created to replace TED⁷. The Doom map editor was to be called **DoomED**. This is where the NeXT solution gave the most impact. The high resolution of the display allowed a lot of real-estate showing small details and many widgets. The stability of NeXTSTEP allowed one to never lose work while writing DoomED or creating a map. The very design of Objective-C also had a tremendous influence. The language's message-dispatching system gracefully handled `nullptr`⁸ dereferences, resulting in a fault-forgiving environment where a faulty feature would not work but did not crash either.

The killer feature was Interface Builder which not only came with a full library of widgets but also allowed creating new ones and connecting them to the business logic instantly.



The release of the source code in April 2015 allowed programmers to peek inside. There is half as much code as in the game engine (doom:32kloc, DoomED:20kloc). Without the power of NeXT, the editor would have taken at least twice that amount of time to make.

⁷id Software map editor up to that point.

⁸"Understanding the Objective-C Runtime" by Colin Wheeler.


```
$ ./cloc.pl DoomEd
```

Language	files	blank	comment	code
Objective C	78	4806	5111	18638
C/C++ Header	72	638	222	2083
C	1	8	12	69
make	1	18	8	52
SUM:	152	5470	5353	20842

DoomED was designed to be the "Adobe Illustrator for World Maps" where the designer simply drew lines, selected sectors, and picked textures.

Trivia : DoomED's icon resembles a Baron of Hell. Upon startup an Imp growling sound is played.



DoomED did not output data usable by the game engine directly. Instead it generated a text format output called DWD. A header served as a magic number which was followed by a list of lines (including sidedefs) and a list of things. Sectors were inferred from a line's ceiling/floor textures, ceiling/floor height, and light properties.

```
WorldServer version 4
lines:475
(1088,-3680) to (1024,-3680) : 1 : 0 : 0
    0 (0 : - / - / DOOR3 )
    0 : FLOOR4_8 72 : CEIL3_5 255 1 0
[...]
```

```
things:138
(1056,-3616, 90) :1, 7
[...]
```

DWD was not designed with space efficiency in mind but rather to be easy to parse since it was post-processed by the node builder tool, `doombsp`.

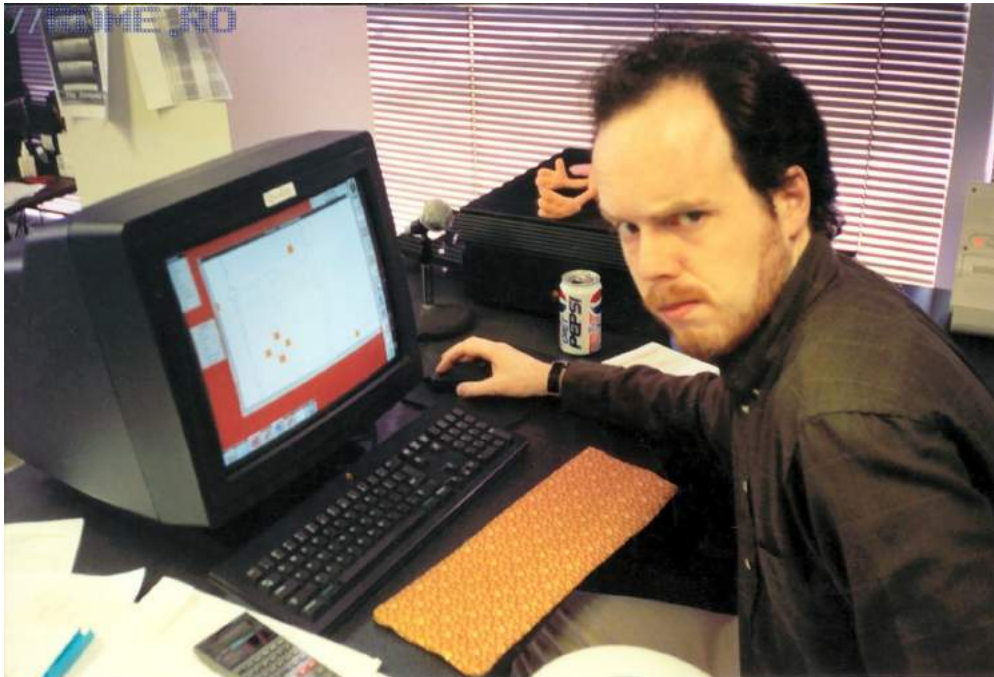


Figure 4.10: Tom Hall, seemingly delighted, working on what would later be called E2M7. The sticker on his monitor reads, "quality".

4.5 Map Preprocessor (Node Builder)

Map pre-processing was not something new at id. Since 1991 with Wolfenstein 3D, maps were already pre-processed to allow fast sound propagation. With DOOM, it was taken to a whole new level both in terms of complexity and processing time.

The main issue at hand was to maintain the same rendering speed despite relaxing the orthogonal grid constraints of Wolfenstein and losing the ability to use the DDA algorithm⁹. The solution chosen was to generate a multitude of accelerating data structures for each map, each dedicated to solving a particular problem.

The tool to do that was called `doombsp`. It took as input a `.DWD` map and outputted a `.WAD`. Not only was the map expressed in a space-efficient format (e.g. expressing vertices only once and referencing them via index), but three data structures were generated alongside it. A binary space-partitioned version of the map expressed a node tree to speed up rendering. A blockmap accelerated collision detection. Finally, a reject table accelerated A.I. processing.

Trivia : Map preprocessing took a significant amount of time. With a NextStation TurboColor, running `doombsp` on E1M1 took 10s. On E1M2, it took 30s. On E2M7, it took a full minute. The first nine maps of the shareware took 3m26s to process. The full twenty seven maps of the registered version required 11 minutes.

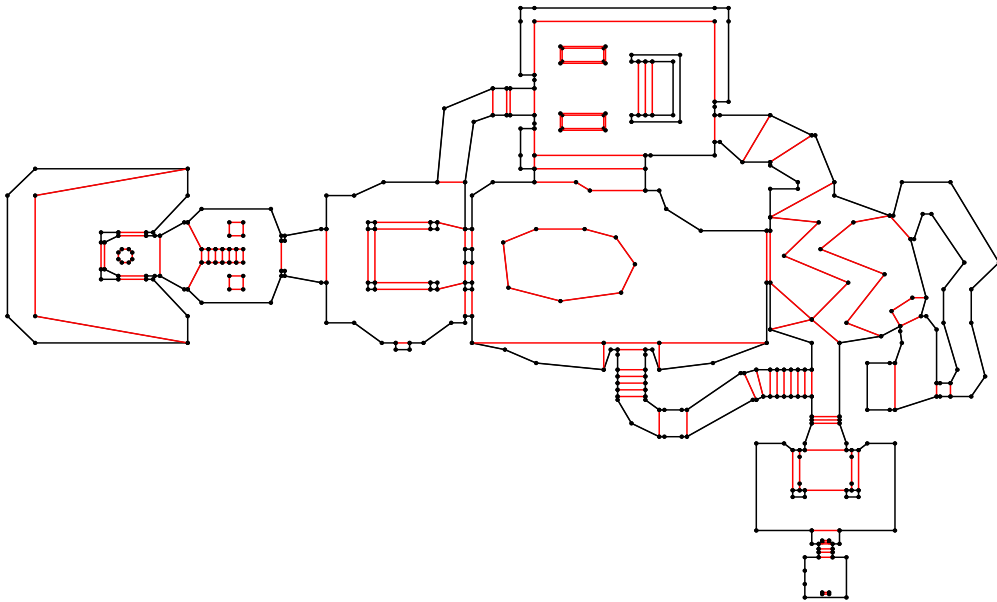
```
$ ./cloc.pl doombsp
```

```
38 text files.
36 unique files.
```

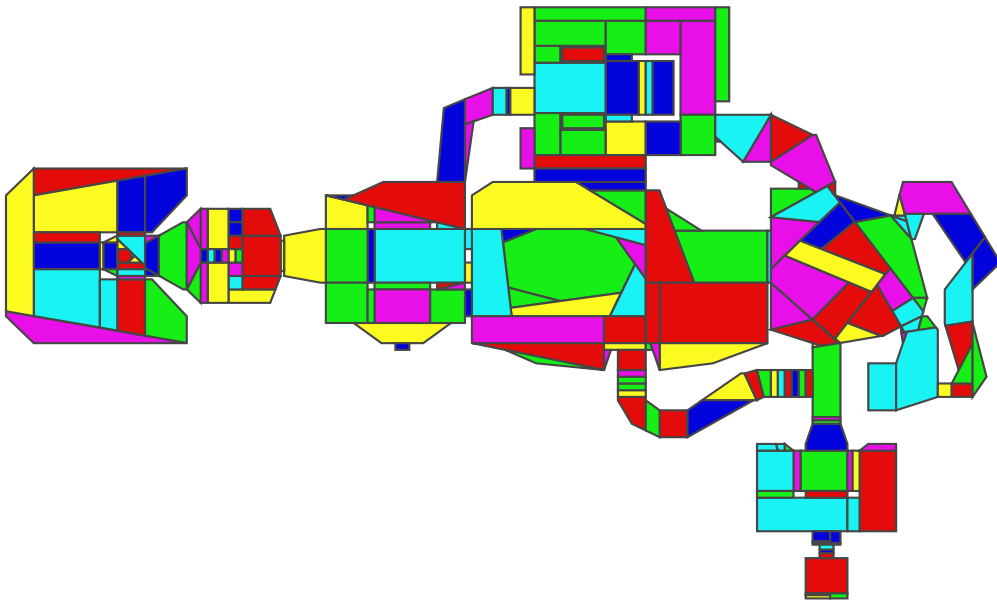
Language	files	blank	comment	code
Objective C	19	1112	1464	4529
C/C++ Header	9	285	190	613
C	2	244	184	603
make	1	16		20
SUM:	31	1657	1846	5765

Trivia : `DoomED.app`, `doom` and `doombsp` were tightly coupled. One button in the editor was enough to save the map, invoke the node builder and start the game with the WIP map.

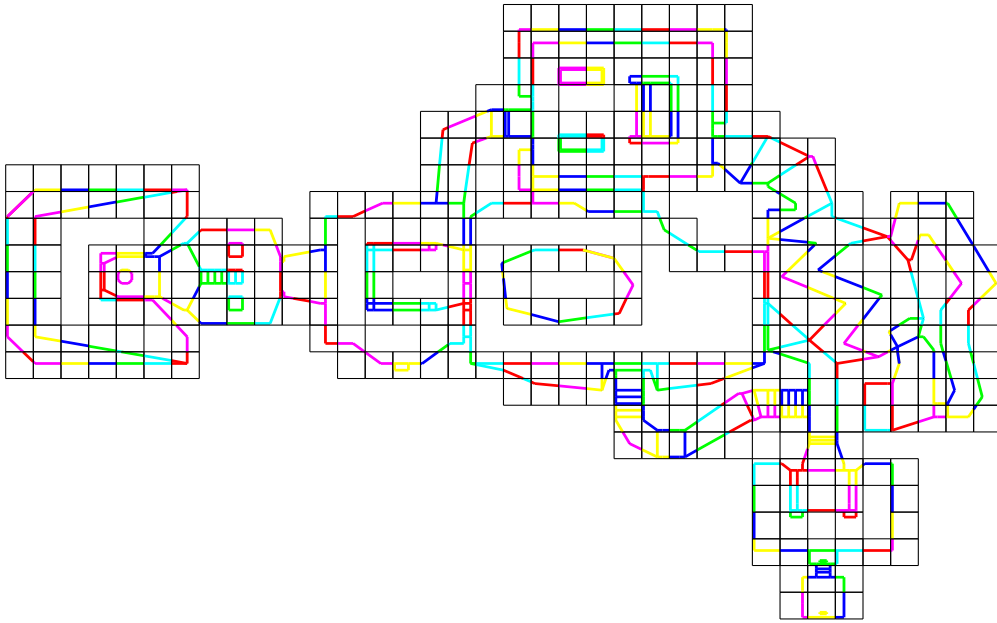
⁹Digital Differential Analyzer was used extensively for VSD (Visual Surface Determination), collision detection, and line-of-sight calculations. This was all gone with DOOM.



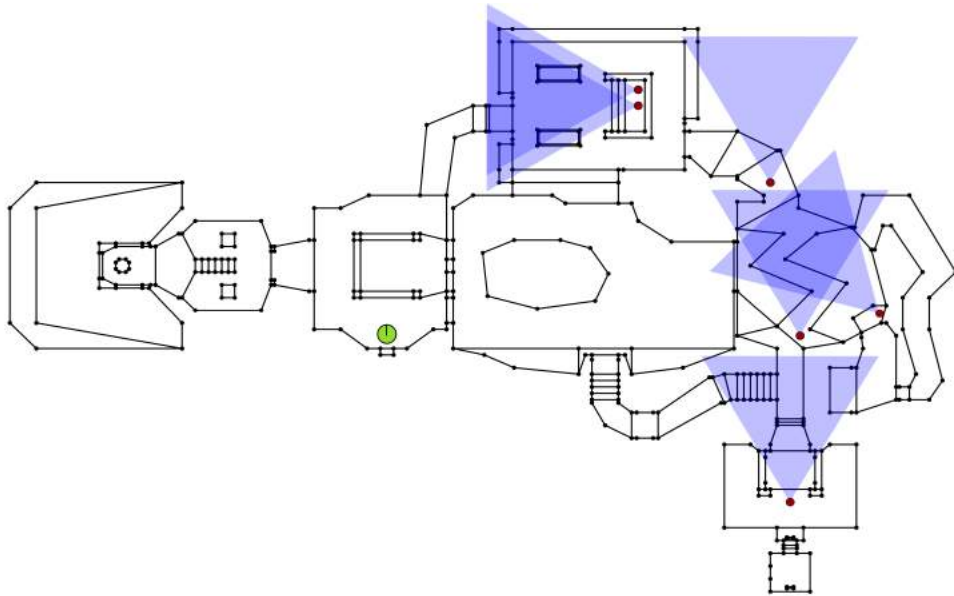
The map as it is generated via DoomED. Portals are red. Walls are black.



The BSP node tree where sectors are split into convex sub-spaces called sub-sectors.



Blockmap slicing where each block is 128x128 to accelerate collision detection.



The REJECT data structure to speed up enemies' and monsters' lines of sight calculations.

The source code of the node builder was released shortly after the game in May 1994. It was the NeXTSTEP version but it was quickly converted to DOS and released under the name IDBSP to the delight of a hoard of modders.

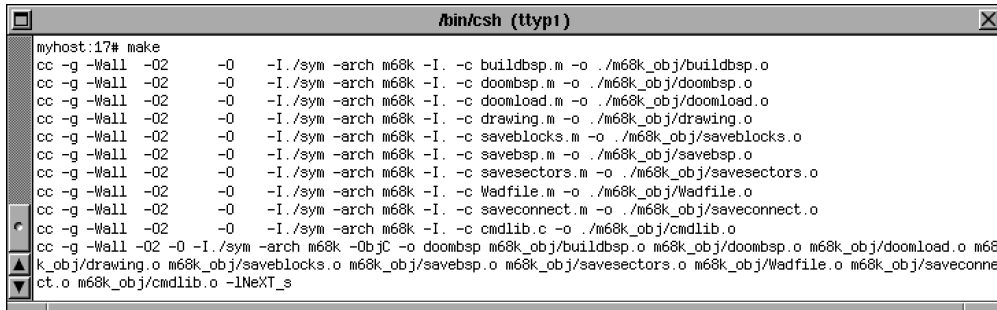


Figure 4.11: Building doombsp on NeXTSTEP.

For each .dwd, doombsp outputs a set of lumps and stores them in a .wad file (see p151).

Lump Name	Explanation
EXMY	Map start marker where X is the episode and Y the map number. All subsequent lumps are part of this map "block".
MAPXY	Same as EXMY but used in Doom II.
VERTEXES	An array of signed short X, Y pairs. All coordinates in this map block are indexes into this array.
LINEDEFS	An array of lines referencing two vertices. This is a direct translation of the lines used in DoomED. Also points to one or two SIDEDEFS depending on if this line is a wall or a portal.
SIDEDEFS	Defines upper, lower, and middle textures. Also defines texture horizontal and vertical offsets.
SECTORS	Area surrounded by lines, with set ceiling and floor textures/heights with light level.
THINGS	Position and angle for all monster, powerup and spawn location.
NODES	BSP with segs, nodes and sub-sector leaves.
SEGS	Portions of lines cut due to Binary Space Partitioning (see page 202).
SSECTORS	Set of SEGS representing a convex subspace.
REJECT	Sector-to-sector visibility matrix to speed-up line of sight calculations.
BLOCKMAP	128x128 grid partition of the map LINEDEFS to accelerate collision detection.

Figure 4.12: Map Data lumps as documented in "The Unofficial Doom Specs" v1.666.

Slicing the map via binary partitioning is non-trivial. doombsp's heuristic attempts to minimize the number of segments generated while creating a balanced tree and picking axis-aligned splitting lines. A debug flag `-draw` allowed monitoring what was happening. BSP trees and binary partitioning are explained in detail on page 202.

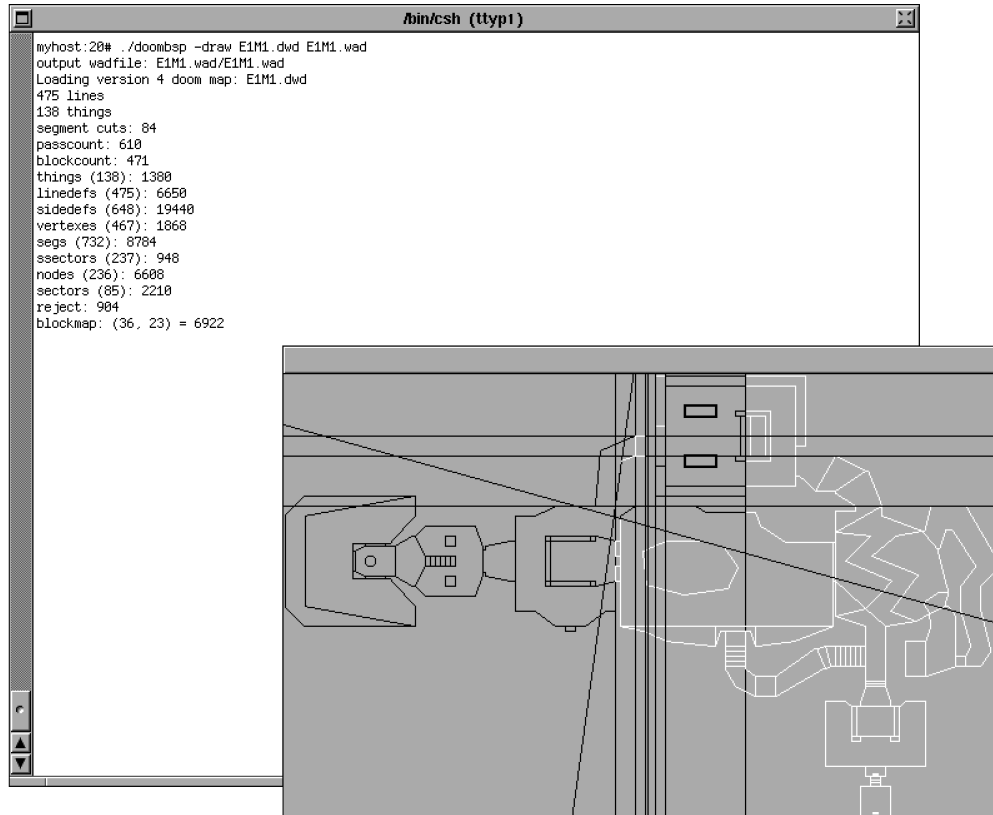


Figure 4.13: Running doombsp in debug mode shows splitter selection.

4.6 Public Relations

In an era before Facebook, Instagram, and Twitter, before social media and before a democratized Internet, studios had few ways to get directly in touch with their potential customers.

Most of the time, newspapers and magazines relayed information which was infrequent, slow, inaccurate and most of time left the reader with more questions than answers.

id Software quickly noticed the Unix system that came with their NeXT hardware featured a tool called `finger`. Entirely text-based, `finger` allowed remotely exploring a UNIX system. A `fingerd` daemon listened for incoming connections on TCP port 79, ready to respond to requests. Running `finger` on a domain name returned a directory of all the accounts registered on that machine.

```
$ finger idsoftware.com

[idsoftware.com]

Welcome to id Software's Finger Service V1.5!

The following people have information available:
```

User	Name	Description	Project
-----	-----	-----	-----
johnc	John Carmack	Programmer	
johnr	John Romero	Programmer	
ddt	Dave Taylor	Programmer	
adrianc	Adrian Carmack	Artist	
kevinc	Kevin Cloud	Artist	
donnaaj	Donna Jackson	id Mom	
.			
.			
.			
.			

Trivia : When it was invented, the term "finger" had, in the 70s, a connotation of "is a snitch". This imaginary "accusatory finger pointing" was a good reminder/mnemonic to the semantics of the UNIX command.

Each employee at id Software could create a `.plan` text file located in the home directory of their NeXT workstation. Anybody blessed with an Internet connection could consult the content of each `.plan`. This was done by prepending a username before the domain name. The result was a direct one-way connection from developer to consumers and a unique way of conveying information in the same way as what is known today as a blog.

When this system was first started, few people were aware of it and even fewer had the means to finger id. It had no update capability nor notifications. Some, like John Carmack, updated their `.plan` daily. Originally a bullet point list of bug fixes, the `.plan` morphed into blog-like content like John Carmack's famous "OpenGL vs Direct3D"¹⁰. The oldest plan to have been preserved was authored during the development of Quake on Feb 18, 1996.

¹⁰See page 423


```
$ finger johnc@idsoftware.com

[idsoftware.com]
Welcome to id Software's Finger Service V1.5!

Login name: johnc In real life: John Carmack
Directory: /raid/nardo/johnc Shell: /bin/csh
Never logged in.
Plan:

This is my daily work ...

When I accomplish something, I write a * line that day.

Whenever a bug / missing feature is mentioned during the
day and I don't fix it, I make a note of it. Some things
get noted many times before they get fixed.

Occasionally I go back through the old notes and mark
with a + the things I have since fixed.

--- John Carmack

= feb 18 =====
* page flip crap
* stretch console
* faster swimming speed
* damage direction protocol
* armor color flash
* gib death
* grenade tweaking
* brightened alias models
* nail gun lag
* dedicated server quit at game end
+ scoreboard
+ optional full size
+ view centering key
+ vid mode 15 crap
+ change ammo box on sbar
+ allow "restart" after a program error
+ respawn blood trail?
+ -1 ammo value on rockets
```

4.7 Music

Like in their previous games, id asked Bobby Prince Jr. to create the music and some of the many audio effects. He was contacted before the game was up and running. To get a grasp of the ambiance he was only given the "Doom" Bible, a document authored by Tom Hall which acted as a design doc describing the tone of the game.

“ Much of what was in Doom Bible never appeared in the game, but it set a mood for starting on the project. Within a few months of receiving that document, I had roughed out a lot of music and most of what turned out to be final sound effects.

— **Bobby Prince, Retro Gamer 44.**

”

Even though id Software set a general direction, it did not prevent Bobby from innovating and taking initiative.

“ The id Software development team originally wanted me to do nothing but metal songs for DOOM. I did not think that this type of music would be appropriate throughout the game, but I roughed out several original songs and also created MIDI sequences of some cover material. This was before any level design and was before most of the artwork had been created. As the game came together, the guys at id saw that this type of music was not appropriate for many of the levels in DOOM. Thinking that this would be the case, I had also roughed out a lot of ambient moody background music, much of which ended up in the game. This song ("At Doom's Gate") was one of the first of its type that I wrote. I heard it as being on a level that went by real fast. As it turns out, John Romero (who placed all of the songs on the levels) decided it was a perfect song for the first level.

— **Bobby**

”

Popular metal bands of the early nineties such as Metallica, Believer, Slayer, Alice in Chains, AC/DC, and especially Pantera served as sources of inspiration.

4.8 Sounds

For the audio effects, the "The General Series 6000" pack was purchased from Sound Ideas. The royalty-free set of 5 CDs was of such high quality that its samples are still used to this day. An attuned ear will recognize the Doom's door opening sound in the music

video of "Body Movin'" by the Beastie Boys¹¹ and in the Doctor Who TV series.

Trivia : If you ever wondered what kind of mixture of animal growling sounds could have possibly been the origin of the Imp dying sample, it was a plain and simple camel.

4.9 Programming

Migrating from the Borland C++ editor on DOS to TextEdit on NeXTSTEP was a trade-off. On the one hand, convenient features such as syntax highlighting were lost. On the other hand, apps never crashed. Precious hours of work were never lost. TextEdit also had markers (//) allowing to "fold" code to improve navigation and readability.

The higher resolution (1120 x 832) of the MegaDisplay allowed seeing much more code vertically and up to three DOS 80 column windows side by side. Notice how Borland's IDE (in its default mode¹²) shows 21 lines of code while TextEdit can show 57 lines.

```

1:1
void D_DoomMain (void)
{
    int          p;
    char          file[256];

    FindResponseFile ();

    IdentifyVersion ();

    setbuf (stdout, NULL);
    modifiedgame = false;

    nomonsters = M_CheckParm ("-nomonsters");
    respawnparm = M_CheckParm ("-respawn");
    fastparm = M_CheckParm ("-fast");
    devparm = M_CheckParm ("-devparm");
    if (M_CheckParm ("-altdeath"))
        deathmatch = 2;
    else if (M_CheckParm ("-deathmatch"))
        deathmatch = 1;

```

¹¹5:09 mark.

¹²Borland's C++ IDE could be set to use 80 column mode to get 50 lines but readability suffered greatly.

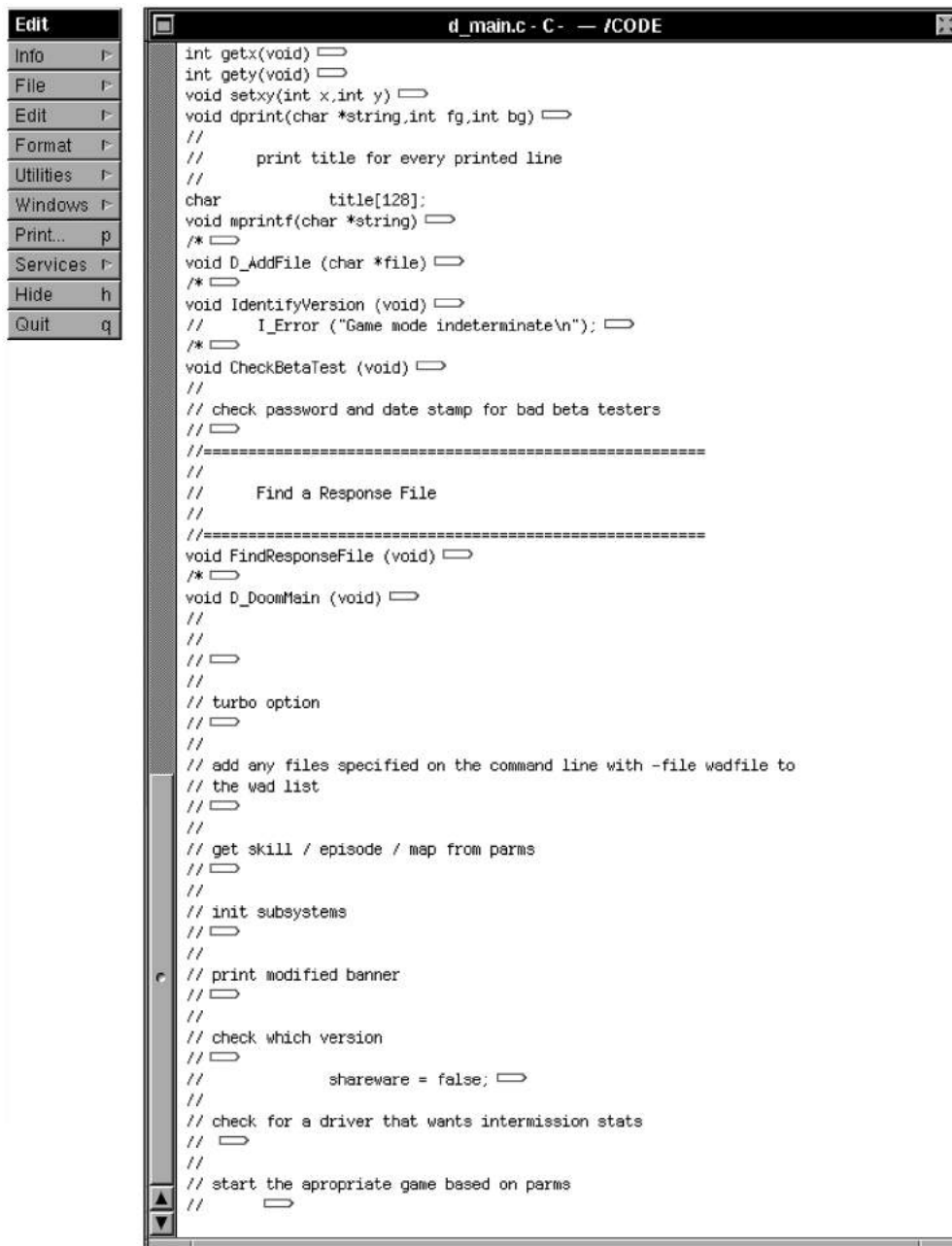


Figure 4.14: TextEdit.app allowed section folding via special markers

In figure 4.14, TextEdit tags and folding show close to 700 lines of `d_main.c`'s 1181 lines. This feature allowed developers to think in terms of systems instead of functions.

4.9.1 Interface Builder, OOP and Objective-C

The list of tools would not be complete without mentioning NeXT's crown jewel which many considered to be NeXTSTEP's killer app, Interface Builder.

"IB" was first written in Lisp by Jean-Marie Hullot in 1984 and commercialized in 1986 under the name "SOS Interface"¹³. Hullot was hired by NeXT, Inc. where along with a team he created a similar tool focused around Objective-C.

The NeXTSTEP version he managed to produce would reduce the construction cost of building GUIs by a factor of 5-10x¹⁴.



¹³Source: "A Brief History of Human Computer Interaction Technology".

¹⁴Source: "NeXT vs Sun: A world of a difference", 1991 promotional video.

With IB, all of the drudge involved in GUI authoring was done with a mouse in the blink of an eye. The "tedious" part involving writing code was only mandatory for the actual meat of the application in the business logic layer. Creating the GUI was a two-step process. First draw all the elements, then connect the UI elements to the Object models.

The first part was, as Steve Jobs qualified it, "insanely easy" since building an interface was done with a series of drag-and-drop operations from a palette of GUI elements onto a canvas. An inspector allowed seeing the properties of an element. Everything from min/-max of a slider to the default value of a text field could be adjusted easily.

The second part, connecting the visual elements to the business logic objects, was also done with the mouse, by connecting visual boxes to targets/actions. The target could be a boolean property in the case of a checkbox UI element or it could be a method in the case of a button UI element.

4.9.1.1 Object Oriented Programming

Beyond its revolutionary design, IB was nicely complemented by the OOP (Object Oriented Programming) design of a programmer-friendly language called Objective-C.

“ In my 20 years in this industry, I have never seen a revolution as profound as [object-oriented-programming]. You can build software literally 5 to 10 times faster, and that software is much more reliable, much easier to maintain and much more powerful... All software will be written using this object technology someday. No question about it.

— **Steve Jobs, Rolling Stone, June 16, 1994.**

”

OOP's encapsulation, inheritance and polymorphism allowed pushing back the limits of complexity a human programmer could deal with. A program would be conceptualized as a collection of potentially nested sub-systems. The mental image did not have to be a complex monolithic block. It could be decomposed in smaller, easier to summarize opaque systems.

4.9.1.2 Objective-C

Objective-C was developed around the same time as C++. However, as co-creator Brad Cox recalls, his creation and Bjarne Stroustrup's brainchild had opposing philosophies. Where C++ placed performance first, Objective-C valued the programmer's productivity first.

“ Back in 1980 when both our languages were under construction, I came down and met with Bjarne Stroustrup. We had radically different views on how our languages would be designed. The key concept was the relative importance of machine efficiency vs programmer efficiency. Eventually, we agreed to disagree.

— Brad Cox (Interview for "The NeXT Bible" book)

The version shipping with NeXT computers featured an important library called Foundation Kit. One of its components, `NSObject`, freed developers from the error-prone memory management burden by offering reference counting via its `retain` and `release` methods. A collection of swiss army knife containers – `NSArray`, `NSDictionary`, `NSSet` and `NSData` – further allowed focusing on the core functionality instead of losing time on infrastructure.

Trivia : Initially using the prefix `NX`, all objects in Foundation were renamed with a leading `NS` for OpenSTEP where "NS" stands for the alliance between NeXT and Sun Microsystems. All these objects are still at the core of macOS and iOS today; the prefix `NS` was never removed.

The very core of Objective-C architecture, based on messages routed via a dispatching method `objc_msgSend`, allowed programs to be more resilient to errors. By far the most amazing feat (at least to a C++ developer) was the ability to send a message to `nullptr`. When a developer sent a message to an object via the following syntax.

```
[obj message]
```

What really happened behind the scenes was a call to the dispatcher.

```
objc_msgSend(obj, @selector(message));
```

The highly-optimized¹⁵, hand written assembly was called millions of times by the time a NeXT had booted. Despite the complex mutable nature of ObjC objects (a method can be added at runtime, changing its duck type) `objc_msgSend` was able to follow an inheritance chain at runtime to find the proper target. More importantly, it was also able to detect a `nullptr` and simply return the value 0 instead of bringing down the entire process.

Altogether, the four pillars of NeXT development (Unix, IB, OOP, and Obj-C) sped up development of DoomED, doombsp, and wadlink to a level that DOS-based tools could not even compare¹⁶.

¹⁵Source: "Dissecting `objc_msgSend` on ARM64" by Mike Ash.

¹⁶Many years later, while evangelizing static code analysis, John Carmack would rant a bit about how the sloppy programming permitted by dynamic languages like ObjC was Not A Good Thing.

4.10 Distribution

To distribute DOOM, id software once again adopted the shareware model where a small part of the game was given away for free. Episode I "Knee-Deep in the Dead", consisting of nine maps, could be downloaded for free and players were encouraged to copy and give it away as much as possible. To this effect, id Software managed to cut and compress the game engine and the first episode down to only two 3¹/₂-inch floppy disks.

Players happy with what they saw could send id Software a payment and receive the two remaining episodes, "The Shores of Hell" and its sequel "Inferno", by mail.



Figure 4.15: "Advertisement Screen" shown at the end of the shareware episode, which left the player with instructions to follow in order to get more episodes.

This time though, id wanted to take things to another level. Not only did they want the game to be distributed via players, they also wanted to be in brick-and-mortar stores. But they did not want the painful logistics of boxing and inventory management tied to physical distribution.

As it turned out there was a way to achieve this seemingly impossible task, thanks to an idea from Jay Wilbur.

“ We told the retailers "we don't care if you make money off this shareware demo". "Move it. Move it in mass quantities." The retailers couldn't believe their ears, no one had ever told them not to pay royalties. But Jay was insistent. "Take DOOM for nothing, keep the profit". My goal is distribution. DOOM is going to be Wolfenstein on steroids, and I want it far and wide. I want you to stack DOOM high. In fact, I want you to do advertising for it, too, because you're going to make money off it. So take this money that you might have given me in royalties and use it to advertise the fact that you're selling DOOM.

— Jay

John Romero shares the same memory and even elaborates on the creativity he witnessed.

“ The challenge was: "How do we get Doom in the store? How do we get something free on shelves?

The idea was that the title screen of doom says "Suggested retail price \$9 dollars" on it and then we told the companies that were already in the stores "if you put DOOM in the store in a box on the shelf you just keep all the money. We don't want any of it just put it in a box and sell it".

Nutty, except that worked. It was everywhere. If you went into a CompUSA back then in 1994 you would see ten different boxes of doom and think they're all different games but they're all the same shareware game. Distributors ended up trying to make the best looking boxes to outperform their competitors because all they were allowed to sell was the shareware.

— John Romero

The success of the formula exceeded their wildest expectations. The shareware version would find its way everywhere, even into the most surprising packages.

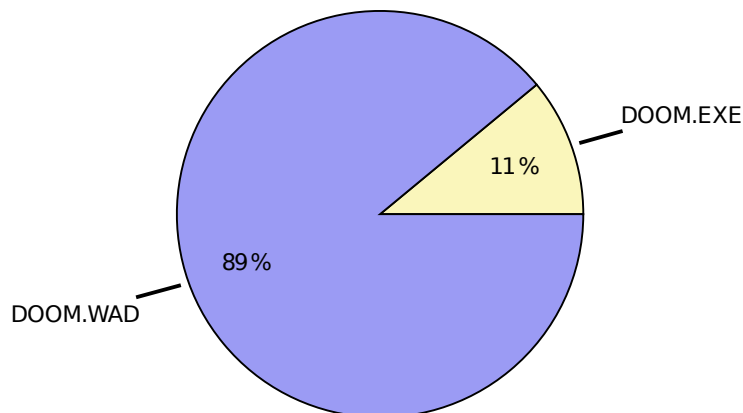
Despite manufacturing difficulties, the famous "DOOM Strategy guides" had the two floppies in an envelope on the back cover of books. Magazines also jumped on the opportunity even if they had to be wrapped in plastic to hold the floppies.



Figure 4.16: DOOM shareware floppy disks

Figure 4.16 shows two 3½" floppy disks containing DOOM shareware, bundled with the book "Survivor's Strategies and Secrets". The publisher paid no royalties for that.

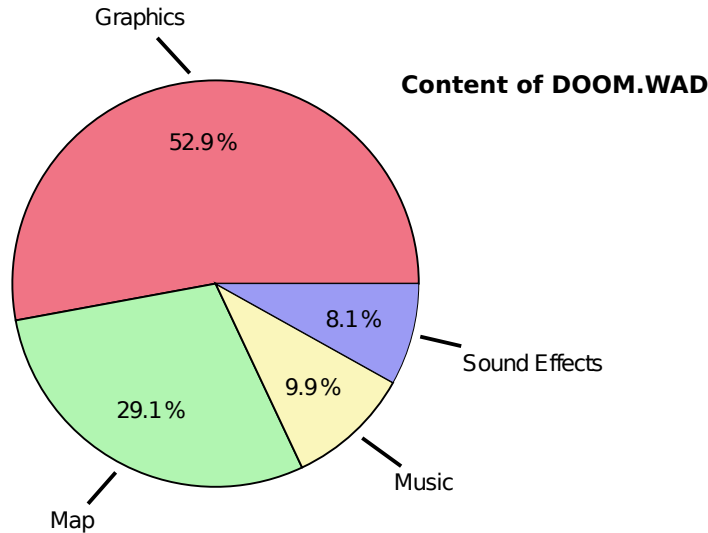
The binary packaging was a departure from their previous title. While Wolfenstein 3D shipped with a WOLF3D.EXE engine and a multitude of .WL6 files, DOOM had only two relevant files. After installation, besides a few TXT files and network drivers, the gaming experience was entirely contained in the engine DOOM.EXE and all assets contained in DOOM.WAD.



The registered version occupied 11,869,745 bytes on the HDD, with 709,905 bytes dedicated to DOOM.EXE and 11,159,840 bytes for DOOM.WAD.

Trivia : The game was also released on the Internet. On December 10th, 1993 they tried to seed it on `ftp://ftp.wisc.edu/` but they could not connect since gamers were permanently connected to the server in order to be the first to get it.

There was no difference between the two DOOM.EXEs from the registered (paid) and the unregistered (free) versions of the game. The engine scanned the current directory, recognized the filename of the WAD archive, and branched accordingly.



4.10.1 WAD archives: Where's All the Data?

The goal of the WAD format was partly to replace the OS filesystem but mostly to embrace the modding community. In a WAD, each asset is stored in a "lump". The WAD is made of three parts with a header, the lump content, and a directory at the end.

```
typedef struct {
    char magicNumber[4];           // "IWAD" or "PWAD"
    int32_t numDirectories;        // #lumps in directory
    int32_t directoryOffset;       // Offset to directory
} header;

typedef struct {
    int32_t offset;                // Offset to lump
    int32_t size;                  // Size of the lump
    char name[8];                  // Name of the lump
} directoryEntry;
```

Trivia : The extension WAD was coined by Tom Hall during an uncanny dialog. John Carmack was looking for a name for the archive format. Upon asking "How do you call a file Where's All the Data?", Tom responded immediately: "A WAD!"

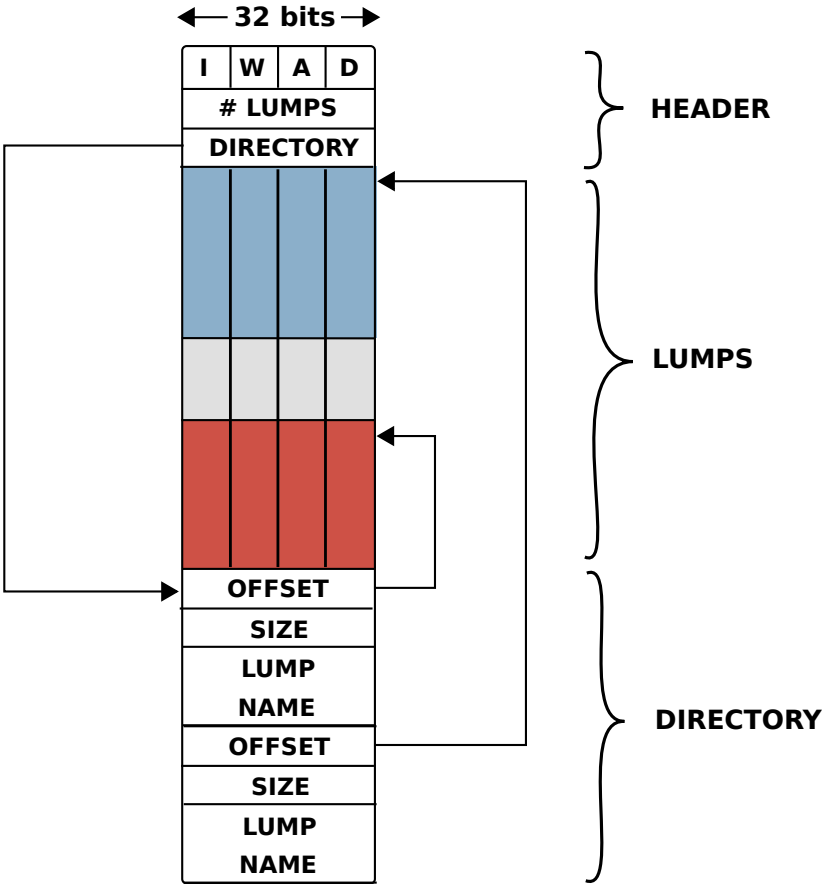


Figure 4.17: A wad file containing two lumps.

The archive format was manipulated via two tools. `lumpy` took a blob and packed it inside a lump, inside a WAD. `wadlink` took several WADs and created/appended them into a single WAD. The structure allows easily adding or removing lumps, since adding a lump only requires moving the small directory at the end and updating the header offset.

DOOM.EXE had a command-line parameter allowing modders to load their own WADs in order to overwrite DOOM.WAD lump entries. This mechanism permitted to customize almost everything. A custom WAD containing an E1M1 lump could be used via a simple `doom -file mylevel.wad` command (detailed on page 178).

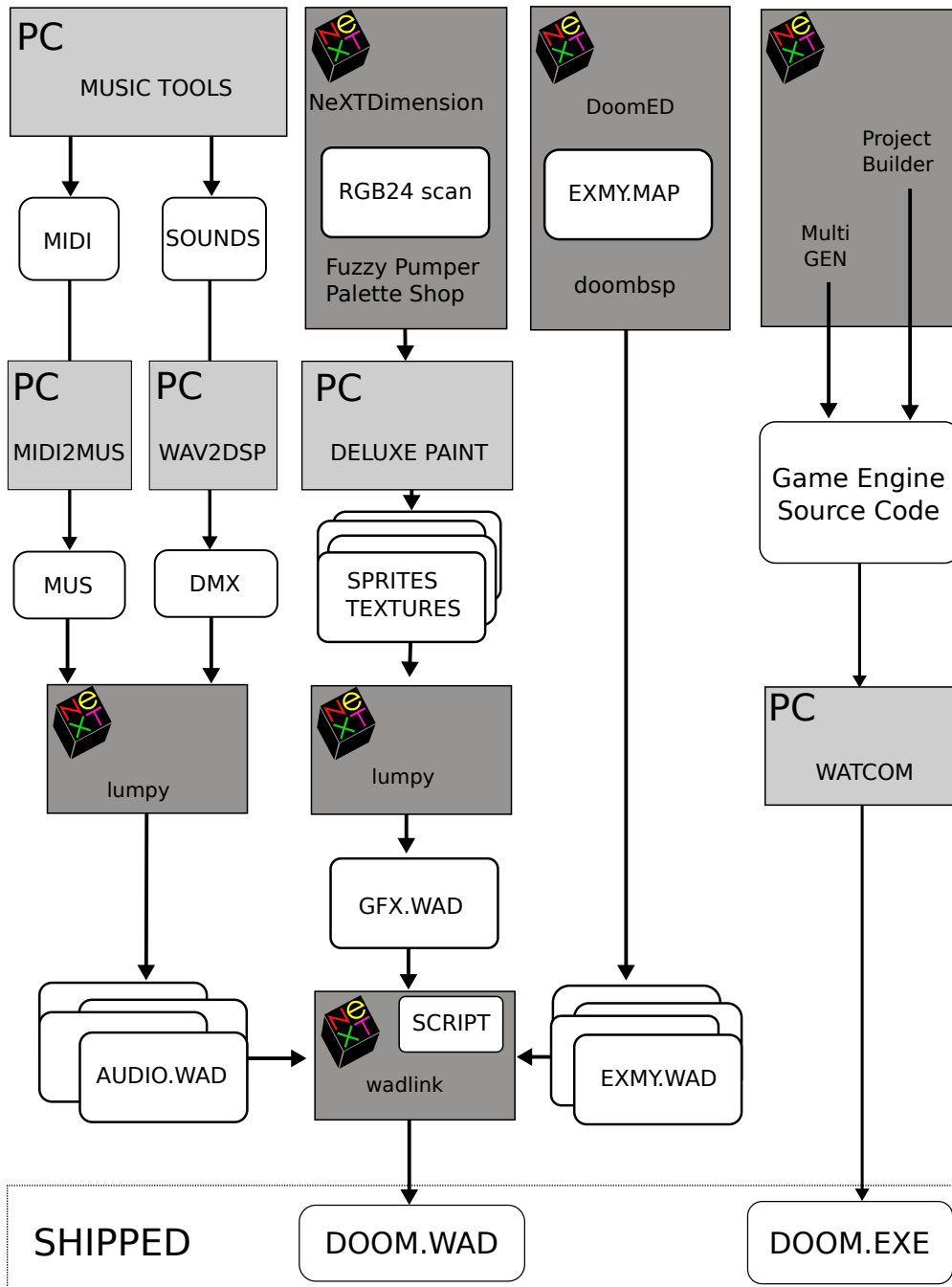


Figure 4.18: DOOM assets pipeline.

Chapter 5

Software: idTech 1

5.1 Source Code

The source code of DOOM was released on December 23, 1997, roughly four years after the commercial release of the game. Originally hosted on id Software's FTP server, the code was transitioned to `github.com` where it can still be found to this day.

```
git clone git@github.com:id-Software/DOOM.git
```

In the long series of id software source code releases¹, DOOM stands apart since what was released was not what was used to ship the game. There is a little bit of a back story.

In early 1997, Bernd Kreimeier approached id Software with a business proposition. He wanted to write a book explaining the internals of the game engine, how to compile, and how to modify it. The idea was to release the book along with the source code.

People at id, especially John Carmack thought it was the "Right Thing" to do². They promptly sent him the source code. Upon reviewing it Bernd realized he had to make a few important decisions. Between the development requirements and Dave Taylor's ports there was code specific to many platforms. The engine could be compiled on no less than five operating systems. Linux, NeXTSTEP, SGI IRIX, and of course MS-DOS were supported. To make the code easier to understand, Bernd decided to pick one platform and delete everything unrelated.

The ideal choice would have been the MS-DOS version. It was the dominant operating system and it was the version players had experienced the game with. However, there

¹From 1993 to 2012, id Software released the code for all games it produced.

²"The Right Thing" is a concept mentioned in the book "Hackers: Heroes of the Computer Revolution" by Steven Levy. It was often quoted in John Carmack's finger plans.

was a copyright issue. id Software had licensed an audio library, DMX, the code to which was proprietary and could not be included with the source. MS-DOS was a no-go.

Another option would have been to release the NeXSTEP version which had seen the most usage and therefore was the second most stable. However, since NeXT had stopped manufacturing workstations in 1994 and sold fewer than 50,000 units over its lifetime this was also a dead end. Few people would have had the software to enjoy it. There was a second problem with NeXTSTEP – the sound and music systems had never been implemented for this platform. NeXTSTEP was also a no-go.

The third available option was the Linux build, and that is what Bernd picked. As he was stripping the code of everything not related to Linux and writing the book, hardware and software kept on evolving. Ultimately, the world of gaming changed faster than he could write and before he was finished, interest in DOOM had decreased in favor of newer engines such as Quake and Duke Nukem 3D.

With the profitability of the venture compromised, the publisher withdrew and the book project was abandoned³. With the blessing of id Software, Kreimeier released the Linux code he had cleaned up. This port has become the base for hundreds of forks since.⁴

5.2 Architecture

Before diving into the code and its structure, let's take some time to understand how id's developers worked. Before DOOM, all work was done on a PC. Code was written and compiled, and the resulting executable started on the same machine. With the introduction of NeXT workstations into the mix, the process had to be different.

A developer had two machines, a NeXT workstation and a PC. All authoring work was done on the NeXT. Code was written with `TextEdit.app`, compiled with `gcc`, linked with `ld`, and the executable ran on the NeXTstation. The biggest advantage of working on a Unix system was stability. Developers never lost their work due to IDE crashes⁵.

Once the developer was happy with the result, he switched to his second machine. In order to do so, he literally rolled his chair over to the PC where the NeXT workstation's hard-drive was mounted over NFS. The PC compiled the same source code⁶ a second time, using

³What survived, "A Brief Summary of DOOM style Rendering by Robert Forsman and Bernd Kreimeier" was of high quality.

⁴The original MS-DOS code can largely be reconstructed thanks to Raven who were much less conservative about DMX-related code. Large portions of previously censored portions of the source, such as `i_sound.c` and `i_ibm.c` can be found in the Heretic and Hexen source code.

⁵With Borland C++, crashes were a daily occurrence (source: correspondence with John Carmack).

⁶With some platform specifics.

WATCOM.EXE and the WLINK.EXE linker which generated DOOM.EXE.

In this setup the PC was relegated to "only" running the game and assessing performance. The PC's hard-drive was actually used only to boot the machine and host the WATCOM compiler. Everything else including the DOOM.EXE executable was stored on the NeXT SCSI HDD.

There were significant obstacles to this methodology. First of all, DOS programs had direct access to the hardware whereas NeXT processes had to use "official" APIs. Second and perhaps most importantly, the two machines had different endianness. PCs ran on Intel CPUs which were little-endian whereas NeXT machines used the big endian Motorola 68040 CPU.

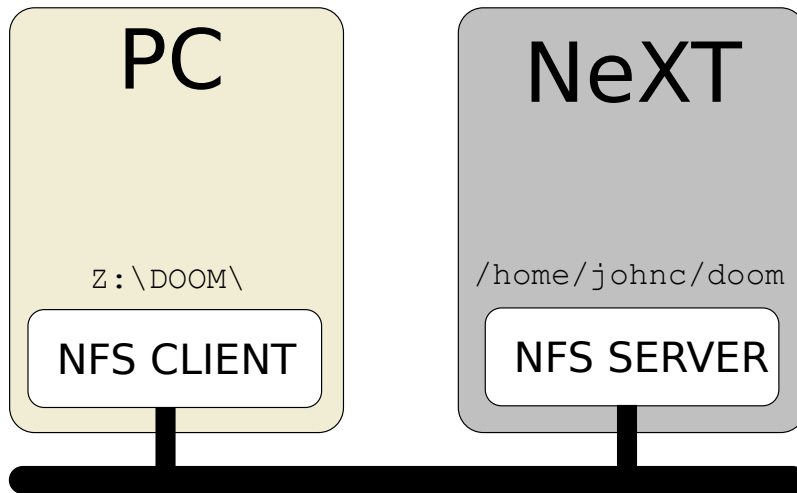


Figure 5.1: NeXT HDD user home folder is mounted on DOS machine as Z drive.

5.2.1 Solving Endianness

Endianness was a term introduced by Danny Cohen in his essay "On holy wars and a plea for peace". His satire, based on Gulliver's Travels in which civil war erupts over whether the big end or the little end of a boiled egg is the proper end to crack open, made for an analogy between two schools of thought among CPU manufacturers. Some wanted bytes organized in memory from left to right, some wanted them right to left. Each side viewed their way as the best one.

In a war where programmers paid the price, no side had any incentive toward peace.

The order of bits in a byte was in universal agreement, but the order of bytes in larger structure, such as shorts (16 bits) or ints (32 bits) was differently interpreted based on the vendor's internal architecture. The stream 0x12, 0x34, 0x56, 0x78 can be interpreted in two ways. On an Intel little-endian machine, it will become 0x78563412. On a Motorola big-endian machine, it will become 0x12345678.

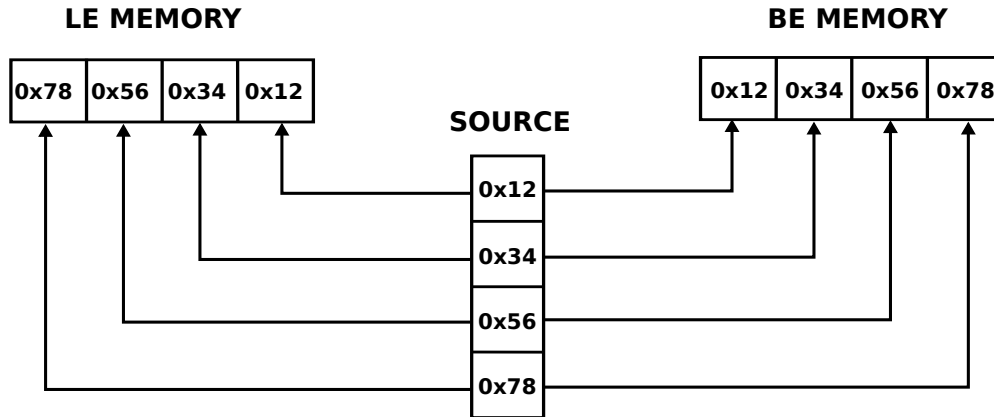


Figure 5.2: A tiny wiring difference results in a hard frontier between CPU words

At the game engine level, the problem was solved via a layer of indirection with a simple macro⁷. When reading from disk, the engine always uses either the LONG or SHORT macro to interpret data.

```
#ifdef __BIG_ENDIAN__
long LongSwap(long);
#define LONG(x)      LongSwap(x)
#else
#define LONG(x)      (x)
#endif
```

```
long LongSwap (long dat) {
    return  ( dat>>24)      |
           ((dat>>8) & 0xff00) |
           ((dat<<8) & 0xff0000) |
           ( dat<<24);
}
```

Even though DOOM was written first on a NeXT, the platform intentionally placed itself at a disadvantage. Because players would use MS-DOS, data was stored in little-endian so

⁷As of 2018 it seems the holy war is finally over. The little-endian tribe of Intel, AMD, and ARM has won.

LONG and SHORT macros translated to zero instructions on consumer Intel based hardware.

5.2.2 Solving APIs

Accommodating the need to run on different operating systems was more challenging. The solution was to have a common "core" that was platform agnostic. To perform I/O, the core would tap into sub-systems specific to the platform they targeted.

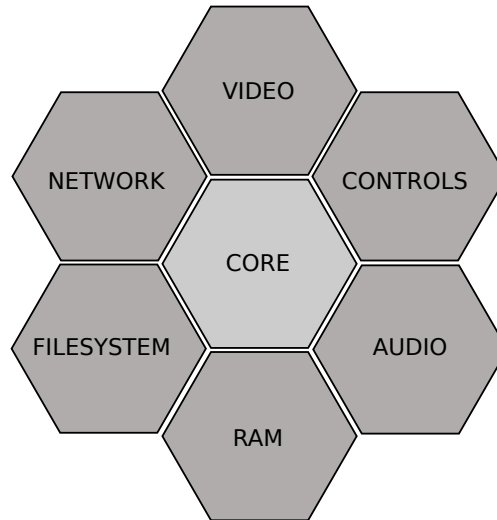


Figure 5.3: DOOM's core and its I/O platform-dependent systems. Notice the similarities with the design of an operating system.

In the case of the video system, it would be using the VGA hardware on MS-DOS but the `NSWindow` API on NeXT. A naive implementation would have required a function pointer acting as a layer of indirection to dispatch each I/O call. A better solution leveraged C's linking stage.

While building a C program, all compilation units (`.c` files) are compiled independently. At the end of the compilation step, all `.c` files have been transformed into object (`.o`) files. Object files may reference each other but because they were created independently, they have "holes" called "unresolved symbols". To generate an executable, all objects are given to a linker which will recognize unresolved symbols from all objects and patch the holes.

Taking the example of `s_sound.c` which is part of the core and looking at `s_sound.o`, we can see this translation unit uses functions such as `I_PlaySong` and `I_StartSound` which are defined in the platform-specific sound system.

Asking `nm` for undefined symbols shows an object file's "holes".

```
$ clang -c -o s_sound.o s_sound.c

$ nm -u s_sound.o | grep I_
U _I_Error
U _I_GetSfxLumpNum
U _I_PauseSong
U _I_PlaySong
U _I_RegisterSong
U _I_ResumeSong
U _I_SetChannels
U _I_SetMusicVolume
U _I_SoundIsPlaying
U _I_StartSound
U _I_StopSong
U _I_StopSound
U _I_UnRegisterSong
U _I_UpdateSoundParams
```

After the linker is done, there are no more unresolved symbols. The final executable is ready to run.

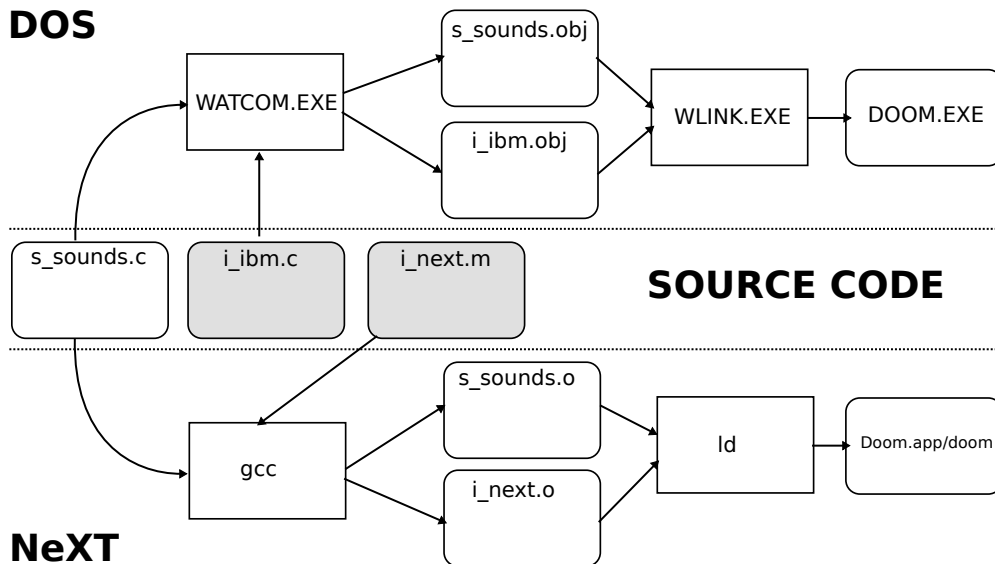
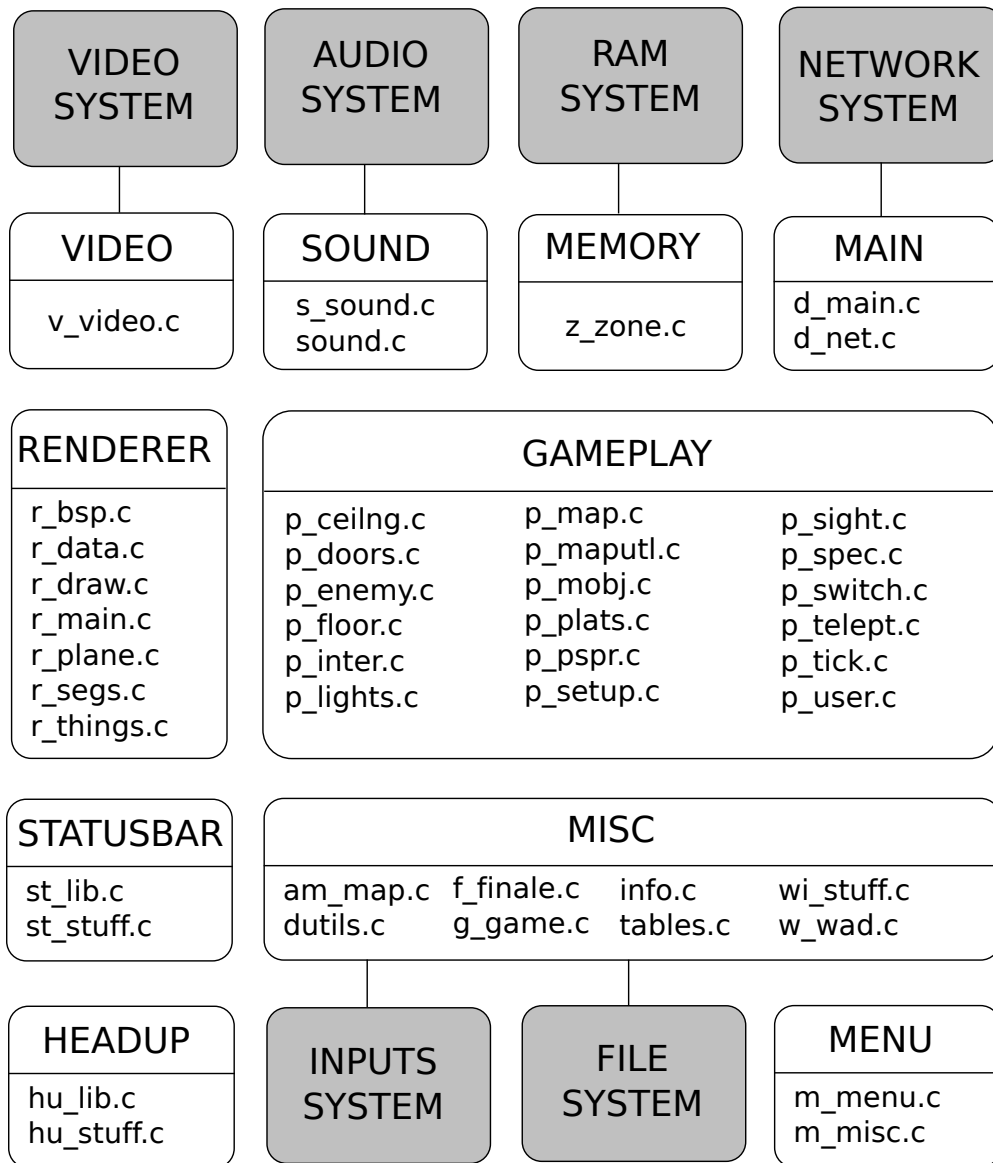


Figure 5.4: Most of the DOOM code is shared. Only a few files are platform specific.

**Figure 5.5:** DOOM source code architecture

In white are the core components. In grey are the I/O systems which require platform-specific code. On DOS these are provided by six extra files: `i_main.c`, `i_ibm.c`, `planar.asm`, `i_ibm_a.asm`, `i_sound.c`, and `i_cyber.c`.

```

/bin/csh (ttyt1)
myhost:78# make
cc -g -Wall -O2 -I./sym -arch m68k -I. -c DRCoord.m -o ./m68k_obj/DRCoord.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c VGAView.m -o ./m68k_obj/VGAView.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c Doom_main.m -o ./m68k_obj/Doom_main.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c i_next.m -o ./m68k_obj/i_next.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_debug.m -o ./m68k_obj/r_debug.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c am_map.c -o ./m68k_obj/am_map.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c dutils.c -o ./m68k_obj/dutils.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c d_main.c -o ./m68k_obj/d_main.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c f_finale.c -o ./m68k_obj/f_finale.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c g_game.c -o ./m68k_obj/g_game.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c hu_lib.c -o ./m68k_obj/hu_lib.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c hu_stuff.c -o ./m68k_obj/hu_stuff.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c info.c -o ./m68k_obj/info.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c m_menu.c -o ./m68k_obj/m_menu.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c m_misc.c -o ./m68k_obj/m_misc.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_ceiling.c -o ./m68k_obj/p_ceiling.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_doors.c -o ./m68k_obj/p_doors.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_enemy.c -o ./m68k_obj/p_enemy.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_floor.c -o ./m68k_obj/p_floor.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_inter.c -o ./m68k_obj/p_inter.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c pLights.c -o ./m68k_obj/pLights.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_map.c -o ./m68k_obj/p_map.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_maputl.c -o ./m68k_obj/p_maputl.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_mobj.c -o ./m68k_obj/p_mobj.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_plats.c -o ./m68k_obj/p_plats.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_pspr.c -o ./m68k_obj/p_pspr.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_setup.c -o ./m68k_obj/p_setup.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_sight.c -o ./m68k_obj/p_sight.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_spec.c -o ./m68k_obj/p_spec.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_switch.c -o ./m68k_obj/p_switch.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_telept.c -o ./m68k_obj/p_telept.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_tick.c -o ./m68k_obj/p_tick.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c p_user.c -o ./m68k_obj/p_user.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_bsp.c -o ./m68k_obj/r_bsp.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_data.c -o ./m68k_obj/r_data.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_draw.c -o ./m68k_obj/r_draw.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_main.c -o ./m68k_obj/r_main.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_plane.c -o ./m68k_obj/r_plane.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_segs.c -o ./m68k_obj/r_segs.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c r_things.c -o ./m68k_obj/r_things.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c sounds.c -o ./m68k_obj/sounds.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c st_lib.c -o ./m68k_obj/st_lib.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c st_stuff.c -o ./m68k_obj/st_stuff.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c s_sound.c -o ./m68k_obj/s_sound.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c tables.c -o ./m68k_obj/tables.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c v_video.c -o ./m68k_obj/v_video.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c wi_stuff.c -o ./m68k_obj/wi_stuff.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c w_wad.c -o ./m68k_obj/w_wad.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c z_zone.c -o ./m68k_obj/z_zone.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c d_net.c -o ./m68k_obj/d_net.o
cc -g -Wall -O2 -I./sym -arch m68k -I. -c fpfunc.s -o ./m68k_obj/fpfunc.o
cc -g -Wall -O2 -I./sym -arch m68k -ObjC -sectcreate __ICON __header Doom.iconheader -segprot __
_ICON r r -sectcreate __ICON app doom.tiff -o Doom.app/m68k_obj/* -linterceptor_s -lMedia_
s -lNeXT_s
myhost:79#

```

```

wcc386p %CFLAGS% i_main.c /fo=pc_obj\i_main.obj
wcc386p %CFLAGS% i_ibm.c /fo=pc_obj\i_ibm.obj
tasm /mx i_ibm_a.asm
wcc386p %CFLAGS% i_sound.c /fo=pc_obj\i_sound.obj
wcc386p %CFLAGS% i_cyber.c /fo=pc_obj\i_cyber.obj
tasm /mx planar.asm
wcc386p %CFLAGS% tables.c /fo=pc_obj\tables.obj
wcc386p %CFLAGS% f_finale.c /fo=pc_obj\f_finale.obj
wcc386p %CFLAGS% d_main.c /fo=pc_obj\d_main.obj
wcc386p %CFLAGS% d_net.c /fo=pc_obj\d_net.obj
wcc386p %CFLAGS% g_game.c /fo=pc_obj\g_game.obj
wcc386p %CFLAGS% m_menu.c /fo=pc_obj\m_menu.obj
wcc386p %CFLAGS% m_misc.c /fo=pc_obj\m_misc.obj
wcc386p %CFLAGS% am_map.c /fo=pc_obj\am_map.obj
wcc386p %CFLAGS% p_ceilng.c /fo=pc_obj\p_ceilng.obj
wcc386p %CFLAGS% p_doors.c /fo=pc_obj\p_doors.obj
wcc386p %CFLAGS% p_enemy.c /fo=pc_obj\p_enemy.obj
wcc386p %CFLAGS% p_floor.c /fo=pc_obj\p_floor.obj
wcc386p %CFLAGS% p_inter.c /fo=pc_obj\p_inter.obj
wcc386p %CFLAGS% pLights.c /fo=pc_obj\pLights.obj
wcc386p %CFLAGS% p_map.c /fo=pc_obj\p_map.obj
wcc386p %CFLAGS% p_maputl.c /fo=pc_obj\p_maputl.obj
wcc386p %CFLAGS% p_plats.c /fo=pc_obj\p_plats.obj
wcc386p %CFLAGS% p_pspr.c /fo=pc_obj\p_pspr.obj
wcc386p %CFLAGS% p_setup.c /fo=pc_obj\p_setup.obj
wcc386p %CFLAGS% p_sight.c /fo=pc_obj\p_sight.obj
wcc386p %CFLAGS% p_spec.c /fo=pc_obj\p_spec.obj
wcc386p %CFLAGS% p_switch.c /fo=pc_obj\p_switch.obj
wcc386p %CFLAGS% p_mobj.c /fo=pc_obj\p_mobj.obj
wcc386p %CFLAGS% p_telept.c /fo=pc_obj\p_telept.obj
wcc386p %CFLAGS% p_tick.c /fo=pc_obj\p_tick.obj
wcc386p %CFLAGS% p_user.c /fo=pc_obj\p_user.obj
wcc386p %CFLAGS% r_bsp.c /fo=pc_obj\r_bsp.obj
wcc386p %CFLAGS% r_data.c /fo=pc_obj\r_data.obj
wcc386p %CFLAGS% r_draw.c /fo=pc_obj\r_draw.obj
wcc386p %CFLAGS% r_main.c /fo=pc_obj\r_main.obj
wcc386p %CFLAGS% r_plane.c /fo=pc_obj\r_plane.obj
wcc386p %CFLAGS% r_segs.c /fo=pc_obj\r_segs.obj
wcc386p %CFLAGS% r_things.c /fo=pc_obj\r_things.obj
wcc386p %CFLAGS% w_wad.c /fo=pc_obj\w_wad.obj
wcc386p %CFLAGS% wi_stuff.c /fo=pc_obj\wi_stuff.obj
wcc386p %CFLAGS% v_video.c /fo=pc_obj\v_video.obj
wcc386p %CFLAGS% st_lib.c /fo=pc_obj\st_lib.obj
wcc386p %CFLAGS% st_stuff.c /fo=pc_obj\st_stuff.obj
wcc386p %CFLAGS% hu_stuff.c /fo=pc_obj\hu_stuff.obj
wcc386p %CFLAGS% hu_lib.c /fo=pc_obj\hu_lib.obj
wcc386p %CFLAGS% s_sound.c /fo=pc_obj\s_sound.obj
wcc386p %CFLAGS% z_zone.c /fo=pc_obj\z_zone.obj
wcc386p %CFLAGS% info.c /fo=pc_obj\info.obj
wcc386p %CFLAGS% sounds.c /fo=pc_obj\sounds.obj
wcc386p %CFLAGS% dutils.c /fo=pc_obj\dutils.obj
wlink @newdoom.lnk ..\dmx\lib\dmx_r.lib
wstrip newdoom
c:\4gwpro95\4gwbind c:\4gwpro95\4gwpro.exe newdoom.exe doom.exe -v

```

Trivia : A full DOS build took an average of 3m19s. Linking alone took 19 seconds. Even incremental builds were time-consuming (e.g: change `r_sky.c` = 27s).

Notice the prefixed file name in figure 5.5. Since C has no namespaces, these prefixes are also applied to function names. `I_` stands for "implementation-specific", `P_` gameplay, `R_` is for renderer and so on.

The beauty of this architecture is that once the platform-specific systems are written, there is zero overhead to writing code that runs on multiple platforms. Most of the code goes into the core and the platform-specific code needs not be touched any more.

Because portability was not an afterthought but an integral part of the development process, DOOM's code layering is never violated. This rigorous design partly explains why DOOM has been ported to so many systems: there is very little code to write⁸.

Additionally, working with multiple compilers such as gcc and Watcom not only surfaced many bugs, it also ensured the code would be ANSI standard compliant.

5.3 Diving In!

Just before finally jumping in, here are a few stats gathered with the `cloc` tool, just to know what volume of code to expect. There is almost twice as much code as in Wolfenstein 3D.

```
$ ./cloc.pl doom-dos-src
  167 text files.
  163 unique files.
   54 files ignored.
```

Language	files	blank	comment	code
C	63	6069	6034	31226
C/C++ Header	36	1022	760	4665
Objective C	5	354	310	1061
Assembly	3	167	151	668
make	1	20	8	34
C Shell	4	14	0	23
DOS Batch	2	2	4	9
SUM:	114	7648	7267	37686

⁸"I ported DOOM to the Nintendo Switch in 45 minutes" by Modern Vintage Gamer on [youtube.com](https://www.youtube.com/watch?v=73LzWzG8Y54).

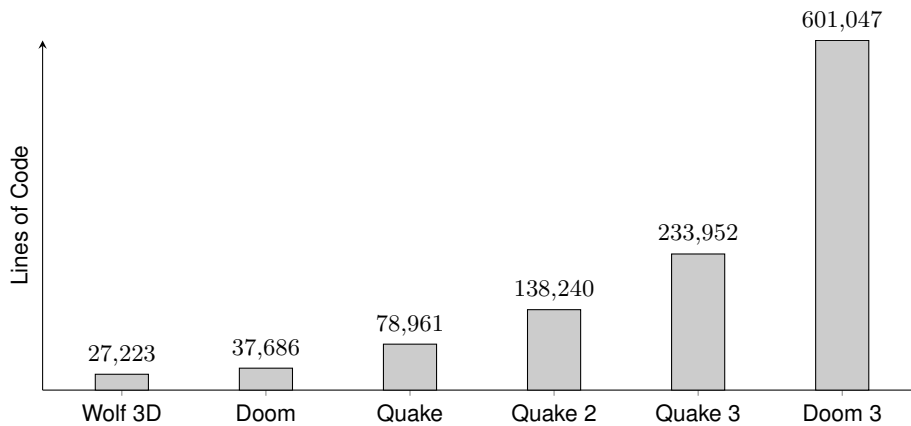


Figure 5.6: Lines of code from id Software game engines.

5.3.1 Where Is My Main?

Exploring a source code repository always starts with finding out what the OS will select as the entry point. 99% of the time it means finding the `int main(int, char**)` function⁹. In the case of DOOM there is one entry point per OS and they are all in implementation-specific (`I_*`) files. For DOS it is in `i_main.c`¹⁰.

Regardless of the platform, all entry points converge on the core main function named `D_DoomMain` located in `d_main.c`.

System	DOS Implementation	NeXT Implementation
Video System	VGA	NSWindow/libinterceptor
Audio System	DMX	Not Implemented
Control System	DPMI Interrupts	NSWindow/NXEvent
File System	WAD/ endian macros	WAD/ endian macros
Network System	Direct interrupts	BSD socket
RAM System	greedy malloc	tight 4 MiB malloc

Figure 5.7: Platform code specific.

Trivia : NeXT platform-specific code was written in Objective-C. All code was in five files named `DRCoord.m`, `VGAView.m`, `Doom_main.m`, `i_next.m`, and `r_debug.m`.

⁹Of course, it is different on Microsoft Windows and you have to search for `WinMain()`.

¹⁰On NeXT the `main` function is located in `Doom_main.m` and all it does is load "DoomRef.nib" into the window system.

```

#include "doomdef.h"

void main (int argc, char **argv) {
    myargc = argc;
    myargv = argv;
    D_DoomMain ();
}

/* i_main.c */

void D_DoomMain (void) {
    FindResponseFile (); // Search doom.wad, doom1.wad...
    IdentifyVersion ();  // shareware or registered?

    V_Init ();           // Video system.
    M_LoadDefaults ();   // Load params from default.cfg
    Z_Init ();           // Zone Memory Allocator
    M_Init ();           // Menu
    R_Init ();           // Renderer
    P_Init ();           // gamePlay
    I_Init ();           // Implementation dependant
    D_CheckNetGame ();   //
    S_Init ();           // Sound
    HU_Init ();          // HUD
    ST_Init ();          // Status Bar

    D_DoomLoop ();       // never returns
}

/* d_main.c */

```

No surprises in `D_DoomMain`, the engine begins by initializing all its sub-systems before jumping to a loop which will never return (`D_DoomLoop`).

Prefixes on translation units (and function names inside them) help to build a mental map of the various sub-systems. `V_` is for Video, `M_` is for Menu, `Z_` is for Zone Memory Allocator, `R_` is for Renderer, `P_` is for gamePlay, `I_` is for Implementation dependent, `D_` is for main Doom, `S_` is for Sound, `HU_` is for HUD, and `ST_` is for Status bar.

The developers did not try to hide what was going on during startup. DOOM's openness needed no splash screen. The text mode messages show the player what is going on behind the scenes (figure 5.8). For each initializer a line is output to the extent that the DOS screen on the right closely mirrors the source code in `D_DoomMain`.

The startup step was fairly fast except for the excruciating renderer initialization in `R_Init` which took forever to complete. It featured not only a line but also a progress bar made up of dots which often required up to a minute to complete on the full version depending

on the machine's hard drive access time. The details and meaning behind each dot are explained in the appendix on page 379.

Trivia : The `-devparm` command line parameter shows more text output. One of these is "I_StartupSound: Hope you hear a pop" which refers to the sound old speakers emitted when they were turned on. In an era before Plug&Play and the Internet, it was an achievement itself to configure the sound card's mysterious IRQ and DMA parameters.

```

                                DOOM System Startup v1.9
P_Init: Checking cmd-line parameters...
V_Init: allocate screens.
M_LoadDefaults: Load system defaults.
Z_Init: Init zone memory allocation daemon.
DPMI memory: 0xd59000, 0x800000 allocated for zone
W_Init: Init WADfiles.
        adding ./doom.wad
        registered version.
=====
        This version is NOT SHAREWARE, do not distribute!
        Please report software piracy to the SPA: 1-800-388-PIR8
=====
M_Init: Init miscellaneous info.
R_Init: Init DOOM refresh daemon - [.....]
P_Init: Init Playloop state.
I_Init: Setting on machine state.
I_StartupDPMI
I_StartupJoystick
I_StartupSound
I_StartupTimer()
        calling DMX_Init
D_CheckNetGame: Checking network game status.
startskill 2 deathmatch: 0 startmap: 1 startepisode: 1
player 1 of 1 (1 nodes)
S_Init: Setting up sound.
HU_Init: Setting up heads up display.
ST_Init: Init status bar.
```

Figure 5.8: DOS output upon starting DOOM.EXE

The game engine executable which shipped with the registered version of the game was the same that shipped with the shareware version. Two functions, `FindResponseFile` and `IdentifyVersion`, simply looked for the asset file and switched a flag depending on what WAD had been found (`DOOM1.WAD` or `DOOM.WAD`).

5.4 Fixed Time Steps

Peeking inside `D_DoomLoop` reveals a standard loop where the machine runs as fast as possible to update the game simulation according to user input and A.I., and then generate visual and audio output.

```
void D_DoomLoop (void) {
    I_InitGraphics ();
    while (1) {
        I_StartFrame ();    // frame synchronous IO operations.
        TryRunTics ();      // Simulate based on I/O and A.I.
        S_UpdateSounds ();  // Generate audio.
        D_Display ();       // Generate video.
    }
}
```

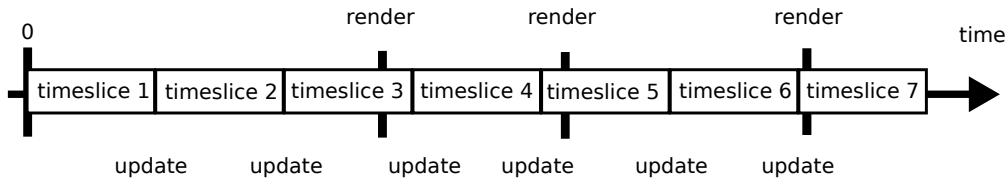
The game simulation happens in `TryRunTics` and uses fixed time steps. The body of the function is summarized as follow:

```
void TryRunTics (void) {
    int      availbletics;
    int      entertic;
    static int oldentertics;

    // decide how many tics to run.
    entertic = I_GetTime ();
    realtics = entertic - oldentertics;
    int counts = realtics;

    // Run as many tics as necessary.
    while (counts--){
        M_Ticker ();
        G_Ticker ();
        gametic++;
        NetUpdate ();
    }
}
```

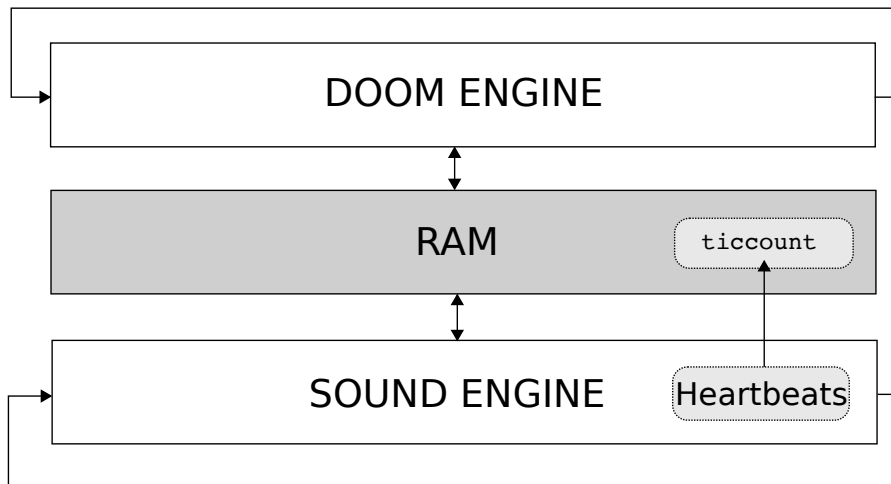
DOOM's unit of time is the `tic`. There are 35 tics in a second (which means a tic is 28ms) and `I_GetTime`'s clock unit is also the `tic`. On each iteration of `D_DoomLoop` the engine calculates how many tics have elapsed and advances the simulation by that amount. Only fully-elapsed tics are simulated. The value of 35 was not random; it is half the frequency of the VGA mode-Y. Once the game state has been updated, video and audio are generated.

**Figure 5.9**

This design choice would end up being controversial. On the one hand it solved the issue of recording a game session and being able to play back on any machine without desync. It also enabled network play and multiscreen play. On the other hand, it meant that no matter how fast the renderer could run, the game would only update at 35Hz which capped the visible framerate on the next generation of PCs based on Pentium CPUs.

5.5 Game Thread/Sound Thread

MS-DOS did not support processes or threads, yet video and audio had to happen in parallel. To make this happen, the audio system is based on interrupts generated at a regular interval. This is explained in detail in the audio section on page 252.

**Figure 5.10**

To disable register caching (resulting in infinite loops), the variable written by the sound engine and read by the DOOM engine is declared `volatile int ticcount`.

5.6 Fixed-point arithmetic

Since the Intel 486 CPU was unable to execute floating-point instructions fast enough, the programmers had to find a way to manipulate and store fractional values during calculations. The answer to this problem was to use fixed-point arithmetic.

Designers at Intel had designed their CPUs to manipulate two types of 32-bit integers. In unsigned integers each bit represents a value from 0 to $2^{32} - 1$ for a decimal range of [0 to 4,294,967,295].

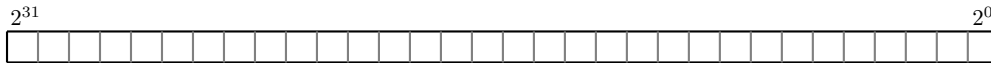


Figure 5.11: 32-bit unsigned integer bit values

Signed integers use two's complement where all bits represent a positive value except for the last one which is a negative value. Signed integers are able to represent values from -2^{31} to $2^{31} - 1$ yielding a decimal range of [-2,147,483,648 to 2,147,483,647].

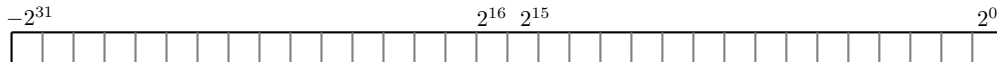


Figure 5.12: 32-bit signed two's complement bit values

For certain types of calculations, DOOM resorts to using a different layout. The signed fixed-point format used is 16:16 where bit 31 encodes a negative integer (-2^{15}), bits [30-16] store a positive integer, and bits [15-0] are used to store the fractional part. The decimal range is [-32768.0 to 32767.9999847]¹¹.

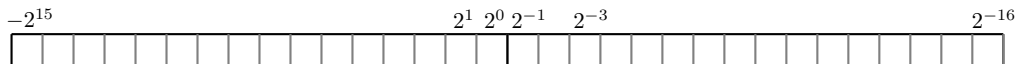


Figure 5.13: 32-bit DOOM fixed-point bit values

To help differentiate "regular" variables from "fixed-point" variables, a simple typedef `fixed_t` is used.

¹¹All 32 bits set to 1 is the maximum value where the negative bit (-2^{15}) = -32768 is added to the positive values: $(2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-9} + 2^{-10} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14} + 2^{-15} + 2^{-16}) = 32767.9999847$.

```
#define FRACBITS      16
#define FRACUNIT      (1<<FRACBITS)
typedef int fixed_t;
```

The beauty of the fixed-point system is that it "just works" at the ALU level – the CPU executes instructions the same way. To convert from one type to another is very simple. From integer to fixed point is a cheap "<< 16" bitwise left shift. From fixed point to integer is the reverse operation, a ">> 16" bitwise right shift.

As an example, $0.5 + 3.75 = 4.25$ yields the correct result bitwise.

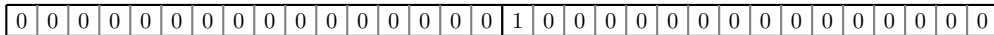


Figure 5.14: Fixed-point representation of 0.5

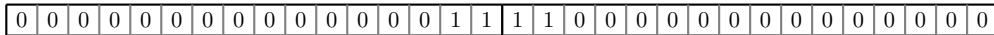


Figure 5.15: Fixed-point representation of 3.75

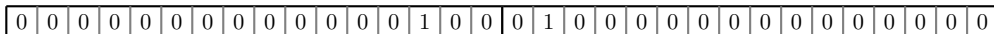


Figure 5.16: Fixed Point representation of 4.25

Even bitwise operations work such as fast divide/multiply with left shift (" \ll ") and right shift (" \gg ").

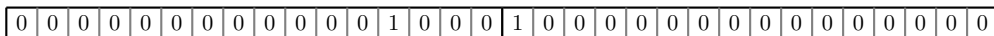


Figure 5.17: Fixed Point representation of $4.25 \ll 1 = 8.5$

There are only two limitations to this system:

- Contrary to floating point, there is no sliding window compensation to avoid overflow and adjust precision. Overflows have to be avoided otherwise information is lost (there are display bugs with extremely large maps caused by this specific problem).
- Integers and fixed-point variables cannot be mixed during operations. The programmer has to manually convert from one to the other since the C language will not "promote" types automatically.

5.7 Zone Memory Manager

Like all game engines of the era, DOOM did not trust stock `malloc`, not even the one provided by Watcom with `libc`. Because it could lead to memory fragmentation, a standard allocator would have jeopardized the stability of the engine. They were also wasteful for small allocations since they were optimized for big chunk allocations which is not what the engine does. They lacked good debugging tools to track leaks and buffer overflows. Finally none of them were portable. So DOOM uses its own memory manager.

Trivia : The engine runs on a clearly-established memory budget. Upon starting up on NeXT the memory manager allocates 4 MiB of RAM and not a byte more. This is done in order to make sure the advertised minimum 4 MiB configuration is sufficient. On DOS the memory manager checks that the machine has at least 4 MiB but will use up to 8 MiB if available in order to improve its cache retention.

The first incarnation of the memory manager was based on zones. Each zone had a memory pool from which RAM could be allocated. This design gave the allocator the name "zone allocator", with the prefix `Z_`, and filename `z_zone.c`. Later the multi-zone idea was abandoned (maybe thanks to DOS/4GW which unified the RAM) in favor of a design featuring one zone containing a chain of blocks. The Zone name was still cool, so it remained.

Looking at the code's structs sheds light on how the memory manager works.

```
typedef struct memblock_s {
    int             size;           // header and fragments
    void            **user;         // NULL if a free block
    int             tag;           // purgelevel
    int             id;            // should be ZONEID
    struct memblock_s *next, *prev; // Double linked list
} memblock_t;

typedef struct {
    int             size;           // total bytes malloced,
    memblock_t      blocklist;      // start/end linked list
    memblock_t      *rover;
} memzone_t;

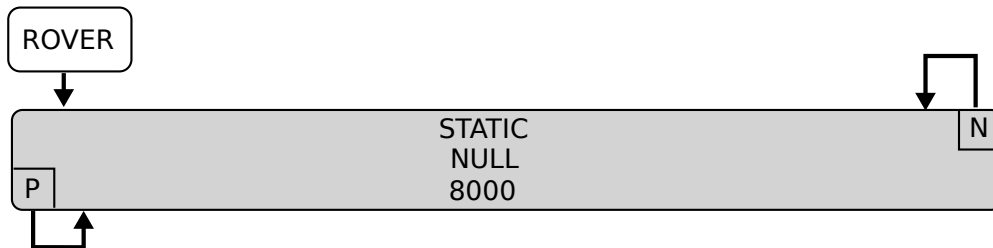
memzone_t *mainzone;
```

The memory allocator uses only one zone for the entire available RAM. This "main" zone is a doubly-linked circular list made of blocks. A block can represent in-use RAM if it has a `**user` value or it can represent free RAM if `user` is `NULL`. At all times the block chain tracks all RAM on the machine.

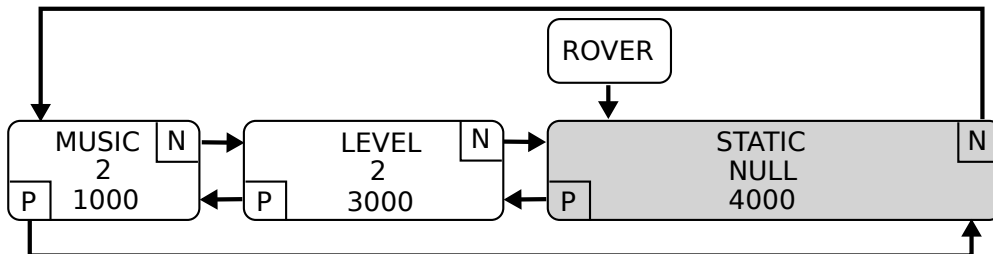
Each block is tagged with a purge hint so the allocator can know whether it can free this block when responding to an allocation request.

```
//-----
//MEMORY ZONE
//-----
// tags < 100 are not overwritten until freed
#define PU_STATIC      1  // static entire execution time
#define PU_SOUND       2  // NEVER USED
#define PU_MUSIC       3  // static while playing
#define PU_DAVE        4  // Dave's static NEVER USED
#define PU_LEVEL       50  // static until level exited
#define PU_LEVSPEC     51  // a special thinker in a level
// tags >= 100 are purgable whenever needed
#define PU_PURGELEVEL 100
#define PU_CACHE       101
```

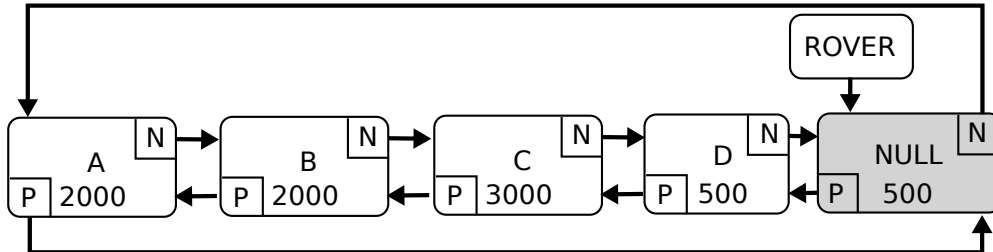
In its initial state (assuming the entire RAM is 8000 bytes) all RAM is in a single block which is marked STATIC. It has a NULL user which means this is a free block. Its size is 8000 bytes. Both next and prev point to itself. The rover points to the only block in existence.



For each call to `Z_Malloc`, the rover searches for a free block big enough. Once found, it creates a block and shrinks the free block. Two calls with sizes of 1000 and 3000 result in three blocks in the chain. Notice the default user value (2) which will be explained later.

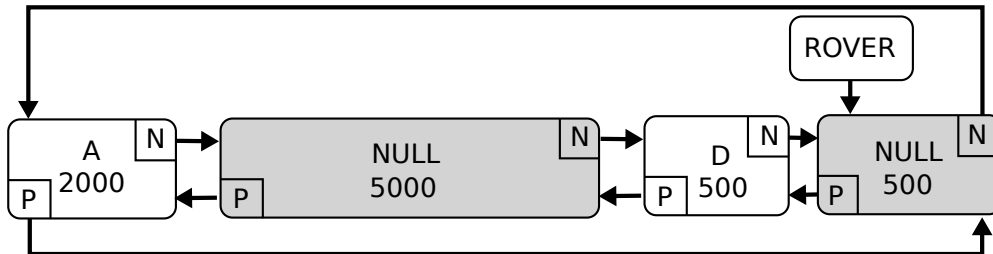


Eventually the allocator will receive a request for an amount of RAM that the "free" block pointed to by the rover doesn't have. In the following configuration, the free block has only 500 bytes. Any request asking for more than that amount will fail.

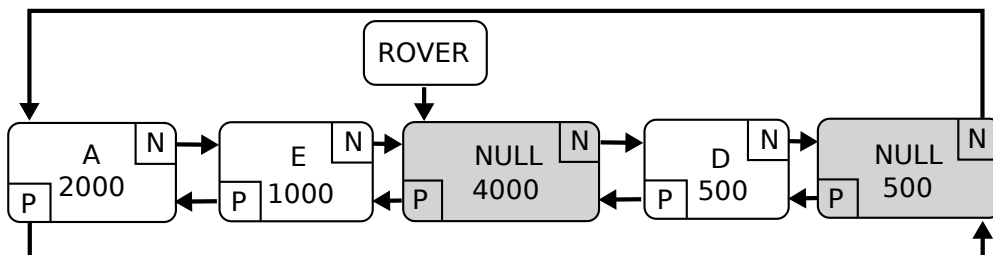


To fulfill a memory request, the rover will start by marking its current position and scan for a free block big enough. If the rover comes back to the same position, there is no free block big enough to satisfy the request. Here the engine will throw an error and terminate.

What is likely to have happened is that some blocks will have been freed via `Z_Free` in the meantime. When a block is freed, its `user` is set back to `NULL` and both neighboring free blocks are merged. Let's assume block B and block C have been freed. They would have been both merged into one free (`owner=NULL`) 5000 bytes block.



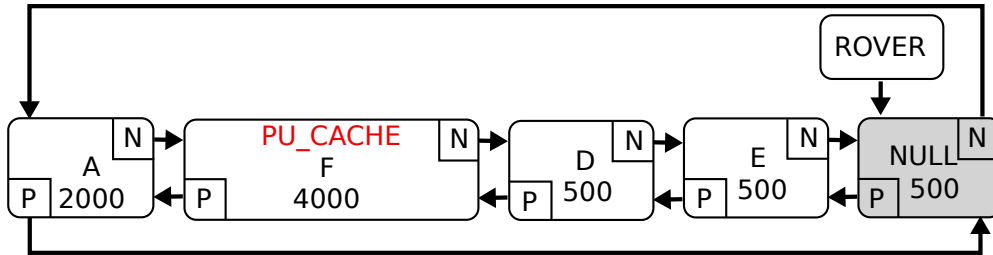
To allocate 1KB, the rover will "roll over", discover the free block and use it for block E.



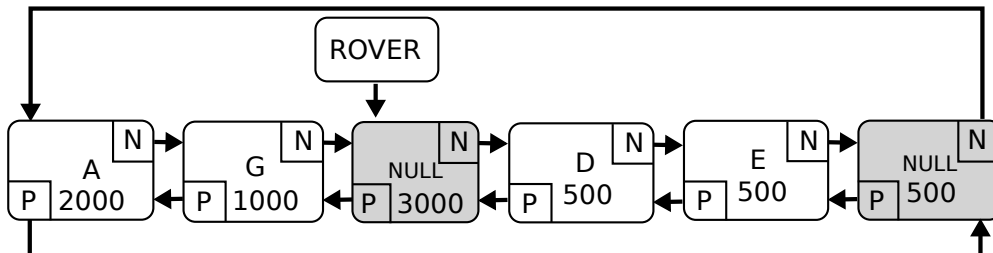
There is a third case which is vastly more interesting. So far we have only talked about

statically-allocated blocks such as `STATIC`, `MUSIC`, or `LEVEL`. But there is a third kind of tag which belongs to the "purgeable" category.

It is mostly used by the WAD/lump manager described in the next section. If a block is marked `PU_CACHE` it means the engine doesn't need the data now but it may in the future. However, the memory allocator is allowed to free it. In the following configuration there is not enough space in any block to successfully allocate 1000 bytes.



The rover will follow the chain and find block F, deallocate it (even though it has an owner) and use it. The result will be the new block G followed by a free block of size 3000.



The memory manager has many other features over `libc's malloc`. The field `id` is used as a canary marker to detect memory overflow (the value should always be `ZONEID`). `Z_Free` is able to detect mismanagement such as double frees. A dump system accessed via `Z_DumpHeap` allows memory inspection. There is even an integrity checker, `Z_CheckHeap`, which verifies that each block "touches" the next and that there are no two consecutive free blocks.

Trivia : Blocks can be "retagged" via the function `Z_ChangeTag`. It is frequent for the engine to allocate memory blocks with a `PU_STATIC` tag while working with assets, only to change the tag to `PU_CACHE` after it is done with it. This allows memory to be freed if needed while potentially avoiding a round trip to the HDD if the block is needed again and is still in RAM.

5.8 Filesystem

DOOM barely interacts with the OS's file system. During a typical gaming session, the engine DOOM.EXE only needs to open DOOM.WAD in order to access its assets. Therefore, the engine does not deal with files but rather with something called lumps which are the atomic unit of a .wad archive and its associated caching system.

While DOOM.EXE remained relatively stable, the asset archive kept on growing, with each new game being more elaborate than the last, as seen in the following list of .wads.

Game	Archive name	# Lumps	Size in bytes
Doom Shareware	DOOM1.WAD	1264	4,196,020
Doom Registered	DOOM.WAD	2194	11,159,840
Doom II: Hell on Earth	DOOM2.WAD	2918	14,604,584
Ultimate Doom	UDOOM.WAD	2306	12,408,292
The Plutonia Experiment	PLUTONIA.WAD	2984	17,420,824
TNT: Evilution	TNT.WAD	3101	18,195,736
French Doom II	DOOM2F.WAD	2913	14,607,420
Heretic Shareware	HERETIC1.WAD	1374	5,120,920
Heretic Registered	HERETIC.WAD	2633	14,189,976
Hexen Demo	HEXENDEMO.WAD	2856	10,644,136
Hexen Registered	HEXEN.WAD	4270	20,083,672
Hexen Deathkings of DC	HEXDD.WAD	326	4,440,584

Figure 5.18: WAD files in the many versions of DOOM¹²

Trivia : There was no need for an I_* abstraction layer to access the OS filesystem. Luckily, all systems now offered standard functions such as `open`, `lseek`, `read`, and `close`.

Lumps are identified by a unique name made of up to eight characters (which conveniently matches DOS's filename length limitation).

```
typedef struct {
    char    name[8];
    int     handle, position, size;
} lumpinfo_t;
```

There were more than thirty types of lump. Those associated with maps and music followed a naming convention so they could be grouped together. Not all lumps had content, some were only used to mark the beginning and end of groups of lumps.

¹²Source: doomgod.com "Internal War Allocation Daemons"

Lump Name	Usage
PLAYPAL	The fourteen palettes used at runtime. Detailed on page 246.
COLORMAP	Translation tables to simulate 32 shades of each of the 256 colors. Detailed on page 230.
DEMO?	Recordings of game sessions by id Software members. Played on game startup as "arcade style" demo.
EXMY	Zero-sized lump serving as marker for the beginning of a series of map lumps. X is the episode number and Y is the map id. The MAPXY variant was used later for the same purpose.
THINGS	All monsters, weapons, ammo, and sprites in the current map.
LINEDEFS	All lines referenced by SECTORS.
SIDEDEFS	All sides referenced by LINEDEFS. A line can have 1 or 2 sides.
VERTEXES	All vertices in the current map.
NODES	A binary tree allowing efficient sorting of segments.
SSECTORS	The sub-sectors, leaves of the binary tree in NODES.
SEGS	The segments pointed to by the SSECTORS lumps.
SECTORS	Referenced by SSECTORS. Specifies ceiling/floor height, texture, and lighting properties.
BLOCKMAP	A collision detection acceleration structure slicing the map into 128x128 blocks. Provides fast access to all LINEDEFS neighboring any point on the map. Detailed on page 262.
REJECT	Line of sight acceleration data structure.
DP.*	Sound effects in PC Speaker format.
DS.*	Sound effects in PCM Mono, 8-bit 11kHz (22kHz supported but used only for DOOM II's Super Shotgun).
D_.*	Music in MUS format (a slightly altered MIDI format).
ENDOOM	Text-mode exit screen to entice players to buy the full version.
DMXGUS	Translation table to match a MIDI instrument with a Gravis Ultra Sound sample file.
GENMIDI	Bank of instrument data to play MIDI music with an OPL audio chip.
PNames	Lists all lump names used as wall patches.
TEXTURE1	A dictionary of all wall texture lumps referenced by SIDEDEFS. Used to speed up access and allocation at runtime.
F_START	Zero-sized lump marking the beginning of flat textures.
F_END	Zero-sized lump marking the end of flat textures.
S_START	Zero-sized lump marking start of item/monster "sprites" section.
S_END	Zero-sized lump marking end of item/monster "sprites" section.
P_START	Zero-sized lump marking the beginning of wall textures.
P_END	Zero-sized lump marking the end of wall textures.
.*	Many others, fonts, TITLEPIC, HELP screens, intermission screens, VICTORY screen...

5.8.1 Lumps

The lump system is the least glamorous part of the engine but one of the coolest in its implementation and what it had to offer to players.

Upon starting up, it looks at every WAD archive provided and indexes every lump found into a gigantic array of `lumpinfo_ts` cunningly named `lumpinfo`. If additional archives are provided via the `-file` command-line parameter, lumps belonging to official id Software WADs (`DOOM.WAD`, `DOOM2.WAD`,...) are added to the `lumpinfo` array first.

In the next example, DOOM was started with the command-line:

```
C:\DOOM>DOOM -file MYMUSIC.WAD -file MYSPRITE.WAD
```

In this illustration, `DOOM.WAD` is represented with only three lumps and the two additional WAD archives have only one lump each. Notice how `DOOM.WAD` are listed first and how a lump name can appear several times (there are two lumps named `MUSIC1`) in the index.

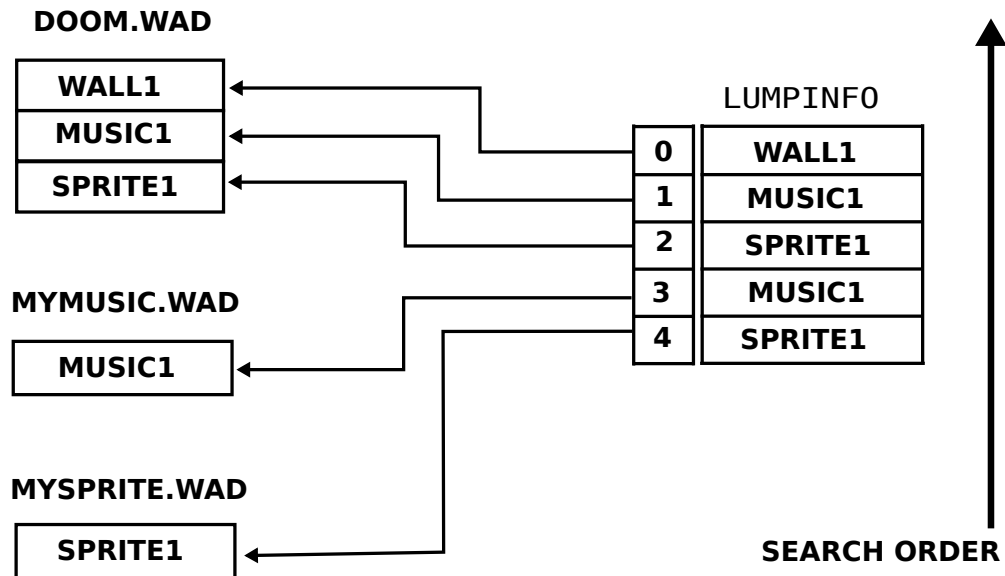


Figure 5.19: Lump system index

Requests to the lump system are sent via a `char[8]` name. The first task is to associate the lump name with an `int` lump index ID. The lump array is searched sequentially in the function `W_CheckNumForName` which in order to speed up comparison uses a cool trick. Instead of comparing eight characters each time, it compares two 32-bit integers.

```

int W_CheckNumForName (char *name) {
    union {
        char s[9];
        int x[2];
    } name8;
    int v1,v2;
    lumpinfo_t *lump_p;

    // make the name into two integers for easy comparison
    strncpy (name8.s,name,8);
    name8.s[8] = 0; // in case the name was a full 8 chars
    strupr (name8.s); // case insensitive

    v1 = name8.x[0];
    v2 = name8.x[1];

    // scan backwards so patch lump files take precedence
    lump_p = lumpinfo + numlumps;

    while (lump_p-- != lumpinfo)
        if (*(int *) lump_p->name == v1 &&
            *(int *)&lump_p->name[4] == v2)
            return lump_p - lumpinfo;
    return -1;
}

```

In the code listing above, you will notice that the index is searched starting from the end. This is done intentionally to allow modders to provide their own assets in WAD archives to overwrite id Software's lumps.

The beauty of this system means that the original DOOM.WAD never had to be modified or patched. Any of the assets could be overwritten – from maps, music, sfx, to graphics¹³ – with a simple command-line.

Trivia : To differentiate official WADs from fan-made ones, the magic number at the beginning of a WAD archive was different. IWAD was reserved for id Software while fan-made WADs were requested to use PWAD.

Once a lump location has been found, a memory block is requested from the memory allocator. The content of the lump is copied from HDD to RAM and returned to the caller.

¹³With the exception of A.I. and map names which are hard-coded in the executable.

The lumpinfo array is mirrored by a lump caching system. When a lump is requested, the index ID is used to look up a lumpcache array first (in `W_CacheLumpNum`). A non-null pointer means the lump is already in a zone block. The lump cache slot assigns itself as the user of the memory block, meaning the cache is automatically invalidated if the block is freed. This mechanism explains the default `owner` value set to 2 which means a memory block is owned but not cached (so there is no cache to invalidate upon deallocation). In figure 5.20, lumps 0 & 2 are not in the cache and will request WAD access.

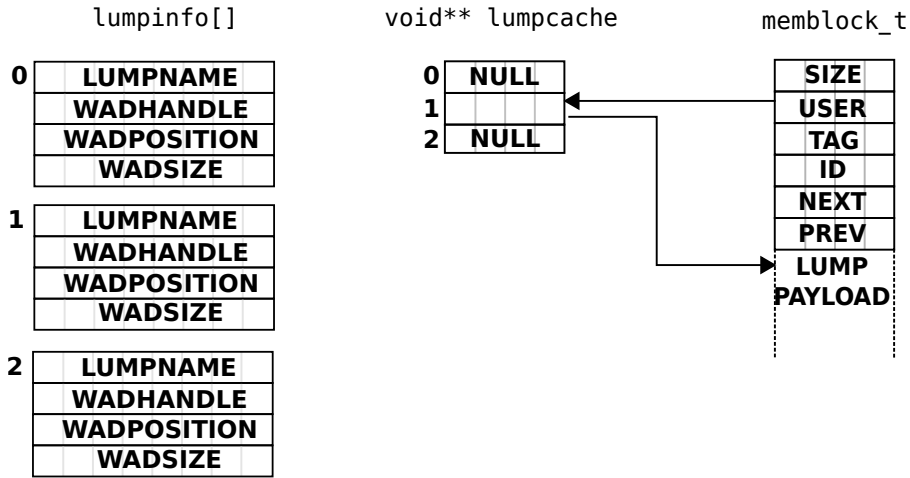


Figure 5.20: Lump 1 is in the cache. Lumps 0 and 2 are not.

```
void* W_CacheLumpNum(int lump, int tag) {
    byte *ptr;
    if ((unsigned)lump >= numlumps)
        I_Error ("W_CacheLumpNum: %i >= numlumps", lump);

    if (!lumpcache[lump]) {
        //printf ("cache miss on lump %i\n", lump);
        ptr = Z_Malloc(W_LumpLength (lump), tag, &lumpcache[
lump]);
        W_ReadLump(lump, lumpcache[lump]);
    } else {
        //printf ("cache hit on lump %i\n", lump);
        Z_ChangeTag(lumpcache[lump], tag);
    }

    return lumpcache[lump];
}
```


id Software explained the assets file format to a few individuals in the community. Within a month, the "Unofficial DOOM specs", describing the WAD format inside out, was online. With the format known and a way to inject new lumps, the modding community flourished.

Some fans used the system to replace almost every aspect of the original game. These mods were known as "Total Conversions" (TCs). The most notorious of them all was named Aliens Total Conversion.

Released in December 1994, it took Justin Fisher one year of hard work to complete. Amusingly the result revisited the Aliens motion picture theme id Software briefly considered before going for demons.



Many sound effects such as doors, weaponry, and explosions were straight from the movie. Actors' lines ("let's rock!") and screams had been digitized. All demons were replaced with aliens, eggs, facehuggers and even the alien queen. The Pulse Rifle, Grenade Launcher and Smart-Gun are there and even the chainsaw was replaced with the "Caterpillar P-5000 Work Loader". Map design wasn't neglected either. Aliens TC replicated the paranoid and scary atmosphere of the movie with the first level entirely devoid of enemies!

5.9 Video Manager

Before continuing our trip through the code and looking at the rendering in `D_Display`, let's take a few pages to study the graphics stack where each frame is stored and manipulated. The video system is located in the core. It maintains and exposes two data structures: a set of five framebuffers, and one dirtybox (used as a "dirty rectangles"). All write operations update the framebuffers and the box.

```
// doomdef.h
#define SCREENWIDTH      320
#define SCREENHEIGHT     200
extern byte *screens[5];

// v_video.c
byte* base = I_AllocLow (SCREENWIDTH*SCREENHEIGHT*4);
for (i=0 ; i<4 ; i++)
    screens[i] = base + i*SCREENWIDTH*SCREENHEIGHT;

// st_stuff.c
screens[4] = (byte *) Z_Malloc(ST_WIDTH*ST_HEIGHT,
    PU_STATIC, 0);
```

Figure 5.21: Five framebuffers.

```
// doomdef.h
enum {BOXTOP, BOXBOTTOM, BOXLEFT, BOXRIGHT};
extern int dirtybox[4];
```

Figure 5.22: The dirtybox.

The video system implementation must provide four functions to the core. `I_InitGraphics` is to be told when to initialize itself. `I_UpdateNoBlit` is to be called when a portion of the framebuffer has been modified. `I_FinishUpdate` is called when the framebuffer is fully composed and should be presented to the screen. `I_WaitVBL` blocks and returns on next V-Sync.

Method	DOS Implementation
<code>I_InitGraphics</code>	Set VGA in Mode-Y (320x200 256 colors stretched to 4:3).
<code>I_UpdateNoBlit</code>	Send updated area of the screen to the VGA hardware.
<code>I_FinishUpdate</code>	Flip buffer (update CRTC start scan address).
<code>I_WaitVBL</code>	Wait for V-Sync (Used to wait before updating palette).

To host the framebuffer in the core and not in the video system itself was an audacious trade-off: it was a performance hit since data had to be copied twice before reaching the screen. But it tremendously improved portability since the framebuffer had been abstracted. It also opened the door to reading back from the core framebuffer. This capability allowed new effects such as the "predator" transparency seen with the "spectre" demon.

Trivia : Because DOOM renders a full screen on every frame, there is no need to "clear" the framebuffer like Wolfenstein 3D did. If we were to interrupt the engine while rendering a frame to peek inside framebuffer #0, it would look like a composition of a previous frame and the unfinished new one.

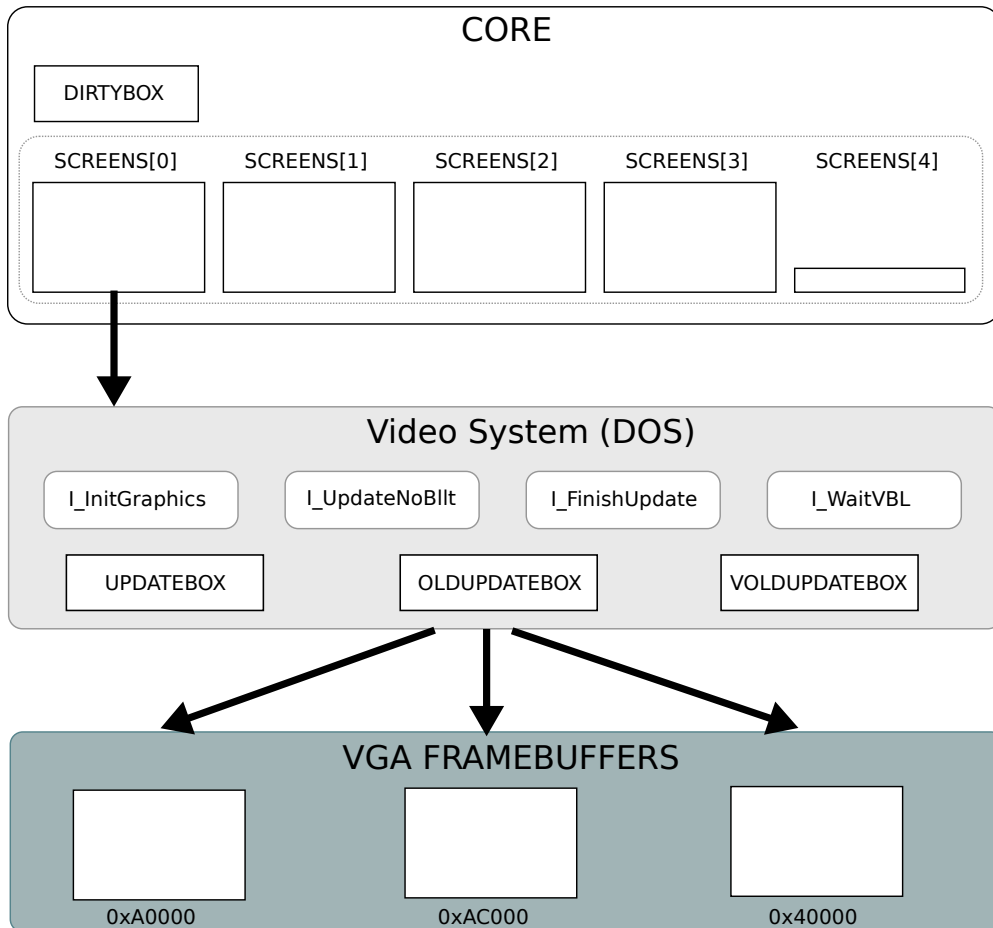


Figure 5.23

The life of a frame is always the same, regardless of what the game is rendering:

1. All write operations are done in framebuffer #0. Each time the engine updates this buffer, it also updates the dirtybox to mark which area has been touched.
2. After a major write sequence, the engine calls `I_UpdateNoBlit` which triggers the video system to read framebuffer #0, optimizing for the smallest data transfer possible via the dirtybox. On DOS the VGA subsystem copies content to the VGA hardware in one of its three VRAM framebuffers. Each VGA buffer has its own dirtybox updatebox to speed up blitting but this remains an expensive operation.
3. Once a frame has been completed, the engine calls `I_FinishUpdate` which signals to the video system that framebuffer #0 will no longer be updated. On DOS, the implementation of `I_FinishUpdate` simply updates the start address of the CRTC which is a lightweight operation.

```
void I_FinishUpdate (void) {
    static int      lasttic;
    int             tics, i;

    // page flip
    outpw(CRTC_INDEX, CRTC_STARTHIGH+(destscreen&0xff00);

    destscreen += 0x4000;
    if ( (int)destscreen == 0xac000)
        destscreen = (byte *)0xa0000;
}
```

The four other framebuffers in the core are used intermittently for temporary storage.

- Framebuffer #1 is used when taking a screenshot but also to store the background when the 3D view is not full screen (`R_FillBackScreen`).
- Framebuffers #2 and #3 are used during the wipe animation to store the start and end screens while compositing into framebuffer #1 (see detailed wipe system in the appendix on page 193).
- Framebuffer #4 is smaller and only stores a virgin status bar used when content cannot be delta updated and a full redraw is needed.

The content of function `I_UpdateBox` to transfer data from RAM to VRAM may look surprising. After all the hardware discussion about 32-bit CPUs and the 32-bit VL-Bus it is uncanny to see the transfer loop do it 16 bits at a time (`short *dest;`). It turns out this was a deliberate choice.

“ Our artwork was done in 8x8 blocks, or "ebes" as Tom called them. A 32-bit loop would have needed more code to handle widths that were an odd number of ebes. It might have been a speedup, but the bus and video card would have to have handled full 32-bit writes, and I don't recall that being common back then. Many VL-Bus cards were still the same basic chipset used on the ISA cards, and often still 16 bit, which meant that a 32-bit write just took 2x as long.

— John Carmack

”

```
void I_UpdateBox (int x, int y, int width, int height) {
    int      ofs;
    byte      *source;
    short     *dest;
    int       p,x1, x2;
    int       srcdelta, destdelta;
    int       wwide;

    x1 = x>>3;
    x2 = (x+width)>>3;
    wwide = x2-x1+1;

    ofs = y*SCREENWIDTH+(x1<<3);
    srcdelta = SCREENWIDTH - (wwide<<3);
    destdelta = PLANEWIDTH/2 - wwide;
    outp (SC_INDEX, SC_MAPMASK);

    for (p = 0 ; p < 4 ; p++) {          // For each VGA bank
        outp (SC_INDEX+1, 1<<p);         // Select bank
        source = screens[0] + ofs + p;   // Read from FB #0
        dest = (short *) (destscreen + (ofs>>2));
        for (y=0 ; y<height ; y++) {
            for (x=wwide ; x ; x--) {
                *dest++ = *source + (source[4]<<8);
                source += 8;
            }
            source+=srcdelta;
            dest+=destdelta;
        }
    }
}
```

5.10 Renderers

With the graphic stack in mind, it is now time to dive into the rendering routine, `D_Display`, which contains every renderer in the game.

```
void D_Display (void) {
    R_ExecuteSetViewSize ();

    // do 2D drawing
    switch (gamestate) {
        case GS_LEVEL:
            if (automapactive)
                AM_Drawer ();
            ST_Drawer (); break;
        case GS_INTERMISSION: WI_Drawer (); break;
        case GS_FINALE: F_Drawer (); break;
        case GS_DEMOSCREEN: D_PageDrawer (); break;
    }

    // signal to video that some stuff has been updated.
    I_UpdateNoBlit ();

    // draw 3D view
    if (gamestate == GS_LEVEL && !automapactive && gametic)
        R_RenderPlayerView (&players[displayplayer]);

    HU_Drawer ();

    I_SetPalette (W_CacheLumpName ("PLAYPAL", PU_CACHE));

    R_FillBackScreen (); // Background when not full screen

    M_Drawer (); // menu is drawn even on top of everything
    NetUpdate (); // send out any new accumulation

    I_FinishUpdate (); // page flip or blit buffer
}
```

There are several 2D renderers (called "drawers" in the code) and one 3D renderer. Different portions of `D_Display` are enabled via a switch case depending on `gamestate`.

Most drawers are requested to draw before the 3D view, then comes the HUD (the text indicating what was picked up) and finally the menu on top of everything else if it is visible.

5.11 2D Renderers (Drawers)

All "drawers" in DOOM were the work of Dave Taylor. The self-proclaimed "spackle coder"¹⁴ was hired three months before the game shipped but he managed to produce many systems. His code style and variable naming conventions differ from John Carmack's and therefore ownership is immediately recognizable.

- Intermission (WI_Drawer())
- Status Bar (ST_Drawer())
- Menus (M_Drawer())
- HUD (HU_Drawer())
- Automap (AM_Drawer())
- Transition screens (wipe_StartScreen())

5.11.1 Intermission

The intermission screen is simple and fully contained in `wi_stuff.c`. It starts by loading a virgin background map from the WAD archive into RAM and placing it in framebuffer #1.



¹⁴Source: "Dave Taylor Interview" by blankmaninc.com.

When a screen refresh is needed, the intermission code does a memcpy (called a "slam" in the code) from framebuffer #1 to framebuffer #0, then draws sprites and text on top of it.

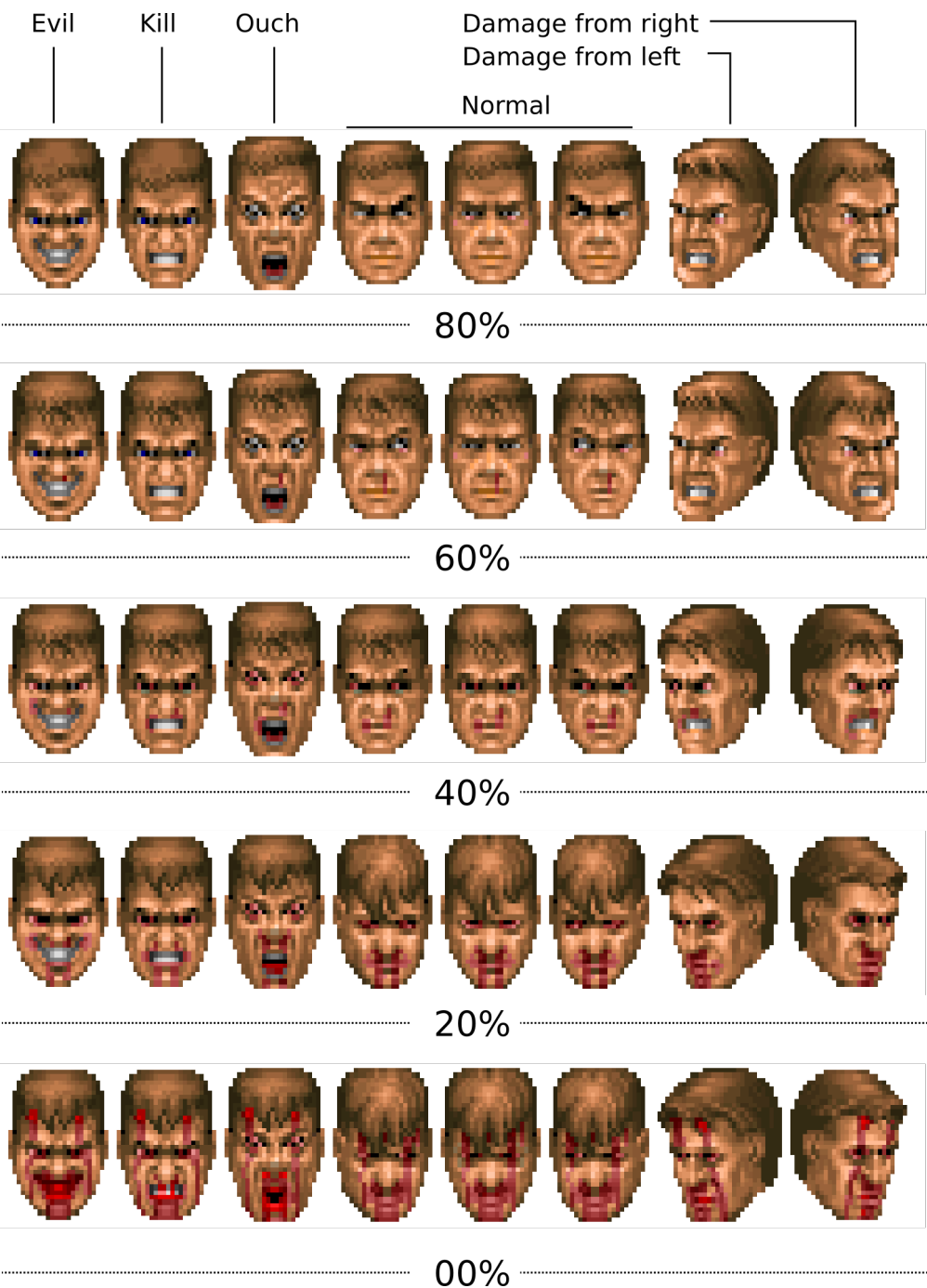


Once the intermission is finished, method `WI_unloadData` does not free the RAM, it just marks all elements it needs as `PU_CACHE`.

5.11.2 Status Bar

The design of the status bar is similar to the Intermission Drawer. Contained in `st_lib.c` and `st_stuff.c`, the code calls its elements "widgets". There are seven widgets, from left to right: ready weapon ammo, health percentage, arms, faces, armor percentage, key-boxes, and all four ammo counts.





Most of the time, the face widget shows the "normal" animation where the marine's eyes look left and right. Notice how there is a set of states for each of the five health brackets.

Perhaps because of the stress incurred during combat, many players never knew this widget showed the origin (left/right) of damage incurred. "Evil" is shown when picking up a weapon. "Kill" is displayed when the player takes head-on damage, while the player holds down the fire button, or upon "getting hurt because of your own damn stupidity" (verbatim code comment found in `st_stuff.c`).

Even to players who spent a lot of time on DOOM, the "ouch" face is likely to be something they have never seen. It was intended to show up when the player suffers an enormous amount of damage (more than 20 hit points), enough to move two brackets down. A bug in the code prevented this from happening:

```
#define ST_MUCHPAIN 20

void ST_updateFaceWidget(void) {
    // being attacked
    if (plyr->damagecount && plyr->attacker && plyr->attacker
        != plyr->mo) {
        if (plyr->health - st_oldhealth > ST_MUCHPAIN) {
            // Show Ouch face
            ...
        }
    }
}
```

The test is the opposite of what it should have been. The ST module shows the ouch face when 20 life is gained which almost never happens. The test should be reversed.

```
if (st_oldhealth - plyr->health > ST_MUCHPAIN) {
    // Show Ouch face
    ...
}
```

With regards to interaction with the video system, the status bar is similar to the intermission module. Upon startup, it draws a virgin status bar into framebuffer #4. When the status bar needs a refresh, it does a `memcpy` from framebuffer #4 to framebuffer #0 and draws all the widgets on top of it.



5.11.3 Menus

There are ten menus and they are all hard-coded in `m_menu.c` and `m_misc.c`. The design is simple with `menu_ts` containing lists of `menuitem_ts`. A name field for each menuitem is used to retrieve the appropriate sprite.



```
typedef struct {
    short    status;           // {no cursor, ok, arrow ok}
    char     name[10];
    void     (*routine)(int choice); // choice = menu item #.
    char     alphaKey;         // hotkey in menu
} menuitem_t;

typedef struct menu_s {
    short    numitems;         // # of menu items
    struct menu_s *prevMenu;   // previous menu
    menuitem_t *menuitems;     // menu items
    void     (*routine)();     // draw routine
    short    x,y;              // x,y of menu
    short    lastOn;           // last menu menuitem index
} menu_t;
```

A `menu_t*` is consumed by the menu drawing routine and a skull is drawn on the left of the currently selected `menuitem_t`.

```
menuitem_t MainMenu[] = {
    {1, "M_NGAME", M_NewGame, 'n'},
    {1, "M_OPTION", M_Options, 'o'},
    {1, "M_LOADG", M_LoadGame, 'l'},
    {1, "M_SAVEG", M_SaveGame, 's'},
    {1, "M_RDTHIS", M_ReadThis, 'r'},
    {1, "M_QUITG", M_QuitDOOM, 'q'}
};

menu_t MainDef = {
    main_end,
    NULL,
    MainMenu,
    M_DrawMainMenu,
    97,64,
    0
};
```



```
void M_DrawMainMenu(void) {
    V_DrawPatchDirect (94,2,0, W_CacheLumpName("M_DOOM",
        PU_CACHE));
}
```

In the previous code sample describing the main menu, notice how the strings `M_NGAME`, `M_OPTION`, `M_DOOM` are all names of lumps found in the WAD archive.

5.11.4 HUD (Head-Up Display)

In its early instances, DOOM featured a Head-Up Display mimicking Doomguy's helmet.



Over time the design changed and the HUD shrank to just lines of text. The small code is contained in `hu_lib.c` and `hu_stuff.c`.

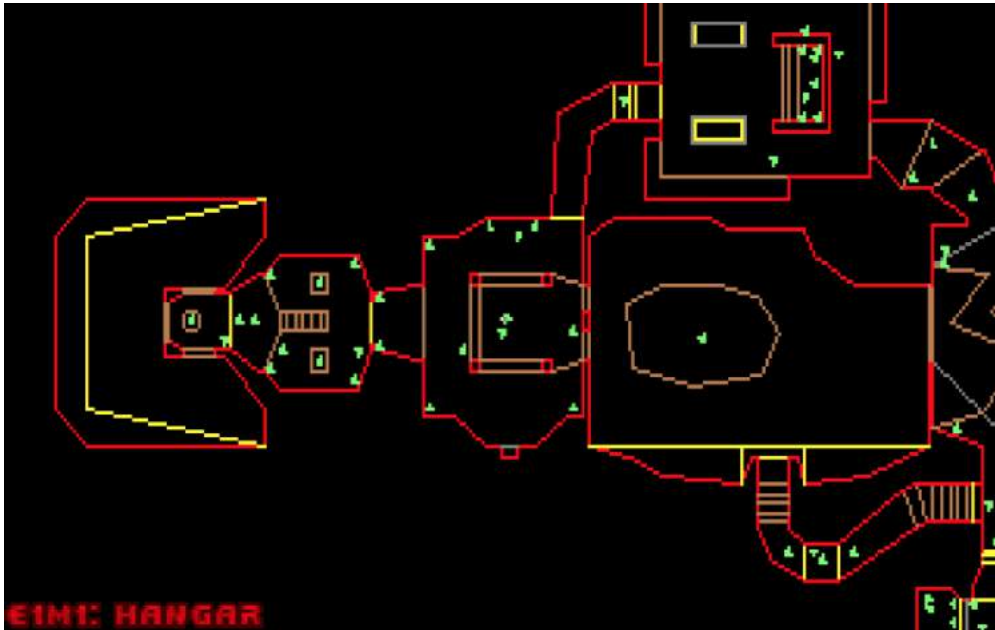
PICKED UP 4 SHOTGUN SHELLS.

5.11.5 Automap

The automap is a small and simple component contained in `am_map.c`. As the player discovers the level, the map keeps track of lines which have been seen. Red lines indicate

solid walls. Yellow lines indicate changes in ceiling height (e.g. doors). Brown lines indicate changes in floor height.

Sadly, it is not possible to play the game in this mode (we all have tried to, let's not kid ourselves) since marking of visited lines is done by the 3D renderer which is disabled when the automap is active.



Trivia : The automap almost featured an easter egg. The files `am_oids.h/c` were to allow the player to play a remake of Asteroids. Unfortunately the easter egg was left unfinished.

“

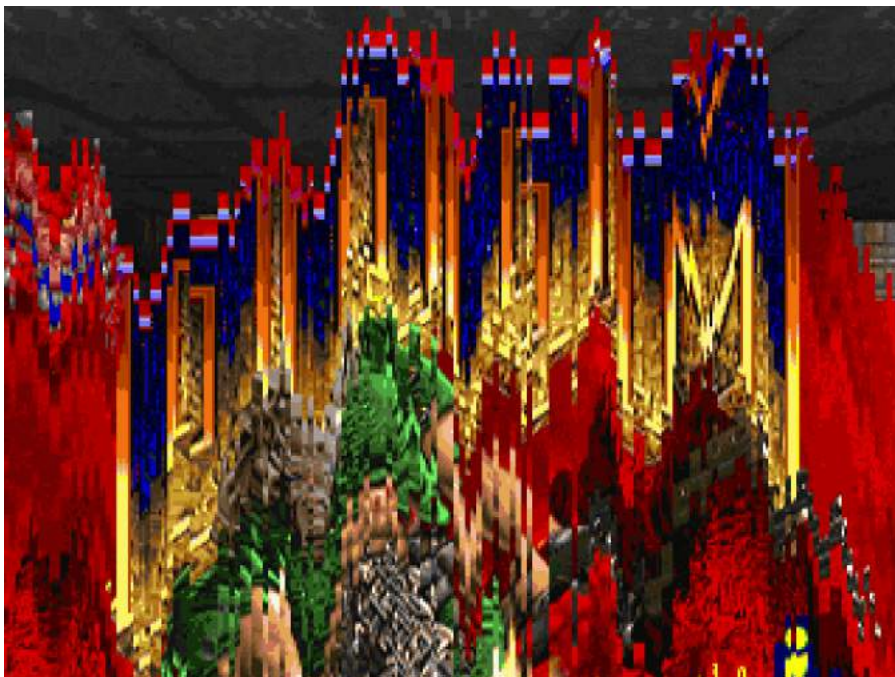
I can't recall whose idea it was, but it was probably mine. I was taken by the vector art style of the automap, so Asteroids would have been a good fit, but Doom was behind, and the pace of development at id was incredibly fast.

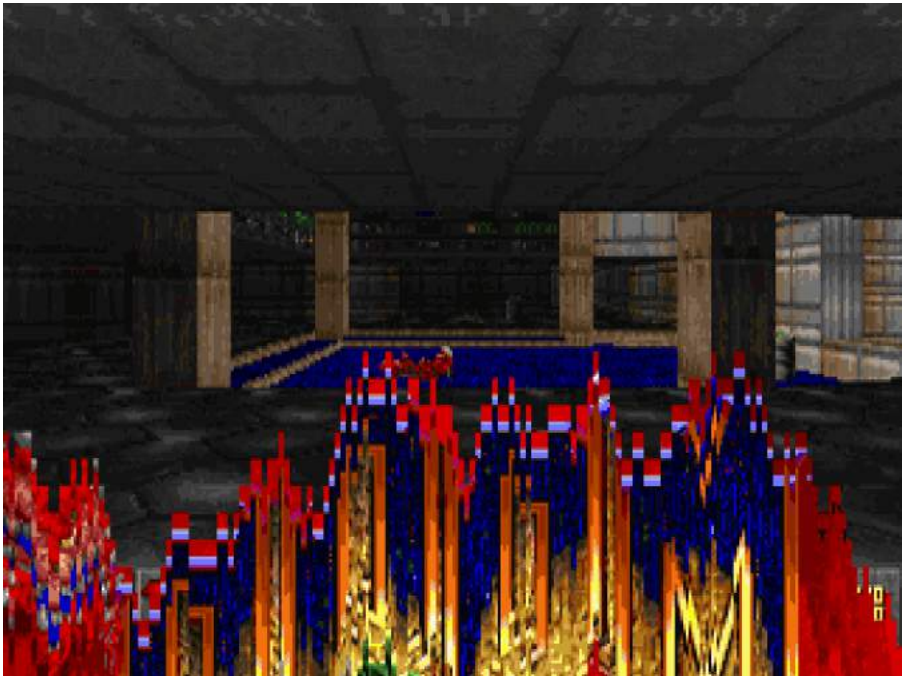
— Dave Taylor

”

5.11.6 Wipe

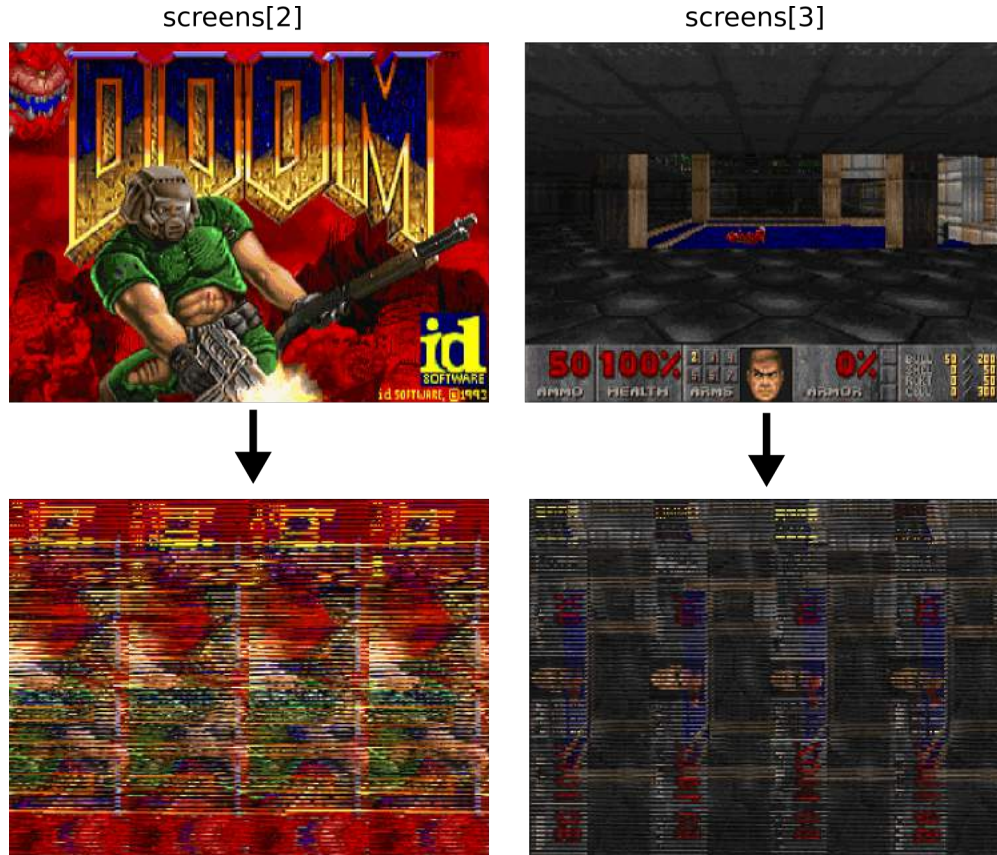
Also known as the "screen melt", wipe is used to transition between sections of the game.





"Wipe" takes whatever is in framebuffer #2 and progressively transforms it into what is stored in framebuffer #3, writing the output into framebuffer #0.

The first step is to reorganize the two sources of data from row major to column major. This is done so the read operations on vertical strips play nicely with the 486 cachelines.



After that, a random sequence of 160 numbers is generated in an array y where each value is within 16 units of its two neighbors. These are used to form the top "wave".

The animation is made so columns of pixels from the source are falling down, letting the destination show behind, as if the source image had been wiped off the screen.

During the animation, columns two pixels wide and 200 pixels tall are copied repeatedly from src and dest to framebuffer #0. All columns fall at the same speed but they are offset by values in y , hence producing the wipe illusion.

5.12 3D Renderer

Doom's 3D renderer is an uncanny combination of proper 3D techniques and screen space tricks. A summary of its main functions (`R_RenderPlayerView`) reveals it is capable of rendering three things.

```
void R_RenderPlayerView (player_t *player) {  
    R_RenderBSPNode (numnodes-1); // root node is last  
    R_DrawPlanes ();               // Draw visplanes  
    R_DrawMasked ();  
}
```

- Segments (walls and portals which are always vertical).
- Flats (ceilings and floors which are always horizontal).
- Things (also called "masked") which are not only monsters, weapons, ammo, and sprites but also partially-transparent walls.

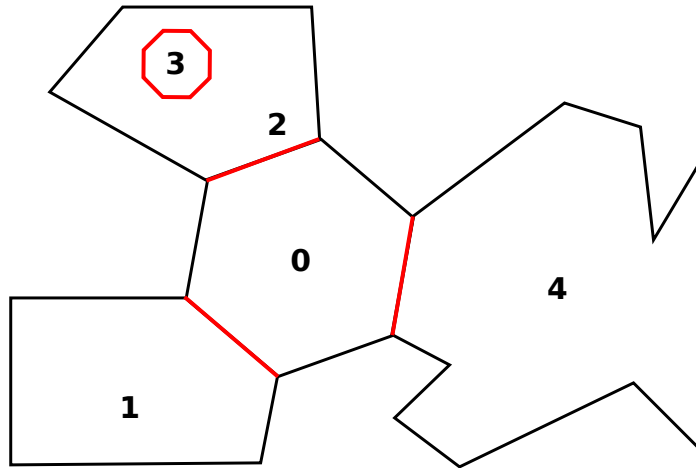
Its most breathtaking characteristic is its ability to render walls and flats with zero overdraw (each pixel is written exactly once). Sprites and transparent walls do introduce a little bit of overdraw but it is minimal.

The life of a 3D frame can be summarized as follows:

- Render wall segments, sorted front to back from the player's point of view. Both wall ends are projected into screen-space axis X. Based on the distance and floor/ceiling heights of the sectors the wall belongs to, calculate a column Y offset and a height.
- To render a full wall, generate a set of columns to make ends meet. Interpolate height and Y vertical columns. While rendering:
 - Record screen-space vertical gaps between walls or between wall and screen boundaries. Infer ceiling (if above mid-screen) and floor areas (if below mid-screen) and store the area into an array of structures called "visplanes".
 - Store sprites to be drawn into an array of structures called "vissprites".
- Render all ceilings and floors from the visplanes.
- Render transparent elements and sprites in back-to-front order.
- Render player sprite (the weapon the Doomguy is holding).

The most important part (and what DOOM's engine is most famous for) is the ability to sort walls and things extremely efficiently thanks to its Binary Space Partitioning tree. Interestingly, there is a back story about how the BSP came to be a central part of the game engine.

In its earlier versions the engine operated on exactly what the designer produced, namely lines and sectors. Starting in the sector containing the player, the engine would look for double-sided lines and treat them as portals, traversing the map in front-to-back order. Each portal lead to adjacent sectors where the process was repeated recursively.



In the map above, with the player located in sector 0, the renderer will flood into other sectors using the red portals. A convex sector 1 is relatively easy to deal with. Things get considerably more complicated when encountering a concave sector like 4. An even worse case is when nesting occurs like where sector 2 contains another sector 3.

Trivia : The design described is exactly what Ken Silverman's build engine would settle on to power Duke Nukem 3D in 1996. By then the Pentium had taken over the world and was more than able to deal with complex polygons.

“ Doom engine was built out of "sectors" – complex polygonal regions with a common floor / ceiling texture and height, but it didn't have the BSP-chopped "subsectors". It started in the view sector and recursively flowed into the adjoining sectors, but because they could all be complex polygons it was a lot of record keeping to know what parts you had already visited or were in the stack somewhere. It worked, and simple areas were fast, but it slowed down precipitously with complexity.

— John Carmack

”

Things indeed slowed down significantly with a particular map of John Romero's creation.

“

I was working on E1M2 around April 1993, and I created a set of circular stairs. John C. wrote the renderer with a sector list to know what should be rendered. The problem is that this set of stairs made his sector list building code take a really long amount of time to execute because the same sectors needed to be put into the list over and over due to how the algorithm worked.

— John Romero

”



With the news that their 3D technology was not good enough to ship already a concern, another serious issue arose.

Back in August 1992, id Software had landed a contract with Nintendo to port Wolfenstein 3D to SNES. With a release scheduled for May 1st, 1994, they had subcontracted the project and forgotten about it to focus on DOOM. In April 1994, the contractor was nowhere to be seen. They had nothing to deliver to Nintendo. It was a big deal involving a huge penalty.

Development for DOOM stopped immediately as the team desperately banged their old game together into a machine not remotely built to do what they wanted. While Tom Hall dusted off his 6502 assembly skills, John Carmack had a different kind of problem at hand: the raycasting technology which Wolfenstein relied on was too much for the Nintendo con-

sole. The SNES and its 6502 on steroids simply did not have enough juice for the DDA algorithm¹⁵.

“

John started searching around for 3D research papers. He had several VHS tapes of math conferences, and compendiums of graphics papers from conferences because game books were a rare thing back then, and there was nothing printed that could help us create the engine we were building – he had to figure out where to get information that was not directly applicable to games and figure out how to adapt it to his problem.

Bruce Naylor's May 1993 AT&T Bell Labs paper was titled "Constructing Good Partitioning Trees" and was published in the proceedings of Graphics Interface '93. John had this book in his collection. Bruce's explanation of BSPs was mostly to cull backfaces from 3D models, but the algorithm seemed like the right direction, so John adapted it for Wolfenstein 3D.

— John Romero

”

“

I do remember clearly that I first used BSP for the SNES version of Wolfenstein, which was a gentle introduction with everything being axial and easier to visualize, which gave me more confidence I would be able to make it work when I went back to working on Doom.

— John Carmack

”

With a visual surface determination not based on raycasting but rather on BSP, and using a low resolution (112x96 scaled up to 224x192 with Mode 7), Wolfenstein 3D SNES managed to reach an acceptable framerate.

Due to Nintendo's strict non-violence policy the game had to be heavily censored to reach a child-friendly quality. Blood was replaced with sweat, guard dogs were replaced with mutant rats, and Hitler was renamed "Staatmeister" (which translates to State Master).

With that problem solved, Wolfenstein 3D for Super Nintendo was released on schedule. The whole team resumed cramming for DOOM and the renderer was changed to also use the power of BSPs.

¹⁵You can read everything about DDA in Game Engine Black Book: Wolfenstein 3D.

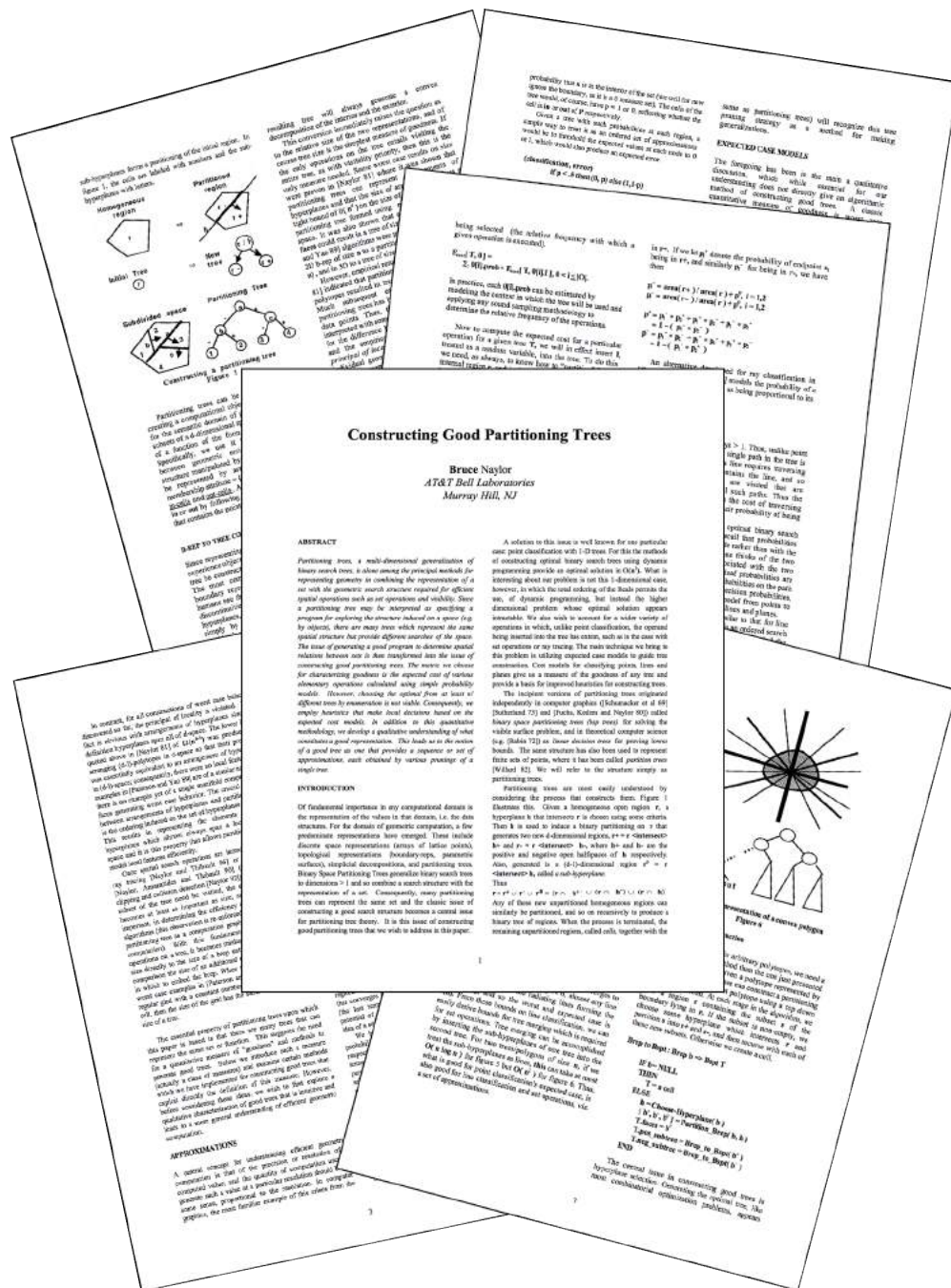


Figure 5.24: Bruce Naylor's paper: "Constructing Good Partitioning Trees"

5.12.1 Binary Space Partitioning: Theory

Binary Space Partitioning trees have many applications. The one we are interested in is how DOOM uses them to sort walls rapidly and consistently. Bruce Naylor's thesis paper, "On visible surface generation by a priori tree structures" features a pretty good summary.

“ In order to determine the visible surface at each pixel, traditionally tile distance from the viewing position to each polygon which maps onto that pixel is calculated. Most methods attempt to minimize the number of polygons to be so considered. Our approach eliminates these distance calculations entirely. Rather, it transforms the polygonal data base (splitting polygons when necessary) into a binary tree which can be traversed at image generation time to yield a visible priority z value for each polygon.

”

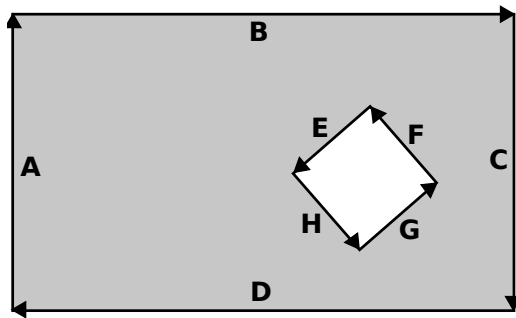
“ When I did the early work on BSPs¹⁶, Bruce Naylor came down and visited here and gave me copies of a bunch of his papers. It's interesting to talk to people about the old days. Of course, you've got the Internet now. You can find anything nowadays. But back then, it was really something to get reprints of old academic papers. There were some clearinghouses I used to use: you'd pay twenty-five dollars or whatever, and they'd mail you xeroxes of old research papers. It was just a very, very different world. I learned most of my programming when I had a grand total of like three reference books. You had to figure everything else yourself. So I was finding I was reinventing a lot of classic things, like Huffman encoding or LZW encoding. So I'd be all proud of myself for having figured something out, and then I'd find it was just classic method and they did it better than I did.

— John Carmack, Interview for Scarydarkfast

”

To study BSPs, let's take the example of a map created with DoomED. For simplicity the map we will be working with is made of eight vertices, four linked together to form a room made of four lines (A, B, C, and D). Inside the room is a pillar which is also made of four lines (E, F, G, and H). The map is made of only one complex sector (it has a hole in it). Notice that all lines have a direction and all lines have only one side (on their right side). Despite its simplicity it is obvious how it is a difficult problem to solve for a renderer since, depending on the point of view, the order in which the lines/walls must be drawn will vary. A naive solution would require a complex sorting algorithm.

¹⁶This was during development of Quake; John and Bruce met only after DOOM had shipped.

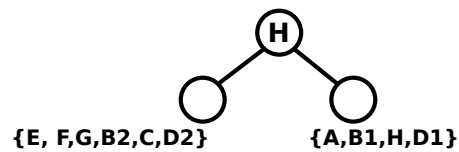
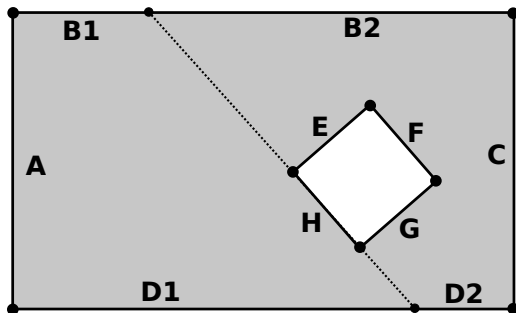
**8 VERTICES****8 LINES***Figure 5.25*

To build the BSP tree from the map, the core idea is to repeatedly select a line to split the map in two. Split lines become SEGMENTS and split sectors become SUB-SECTORS.

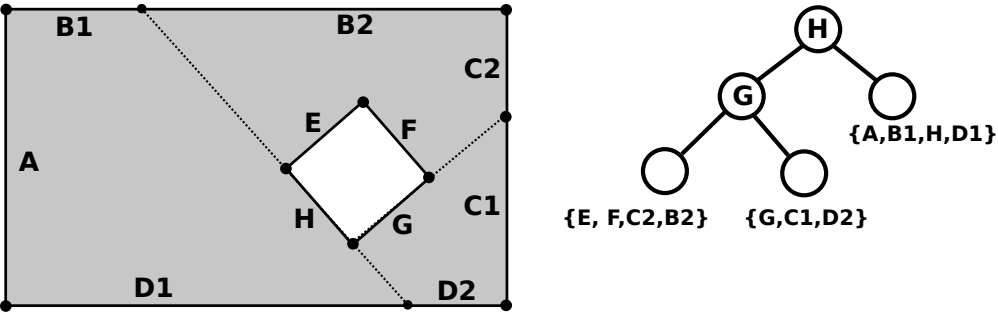
The choice of the splitter is extremely important. There are good splitters and bad splitters. A list of poor choices for the first splitter would be A, then B, C, and D since they would not divide the map evenly.

Let's say our heuristic selected line H which conveniently cuts the room in half. Some lines are entirely to the left of H and some are entirely to its right. Lines on both sides must be split into segments. After the split, the two leaves in the BSP contain two sub-sectors. One is convex ($\{A, B1, H, D1\}$) and therefore will not be touched anymore. The other one is concave ($\{E, F, G, B2, C, D2\}$) and will need further splitting.

The process is repeated until all subsectors are convex.

*Figure 5.26*

Let's follow the process, step by step, until we have a BSP decomposing the space into a set of convex sub-sectors. Notice how, as the binary tree grows, splitter lines are stored in the nodes and segments in the leaves. In the next step, line G is selected as the splitter.



At this point in the splitting we are still not done. The area between B2, C2, E, and F is concave. We need one last split where F is selected as splitter.

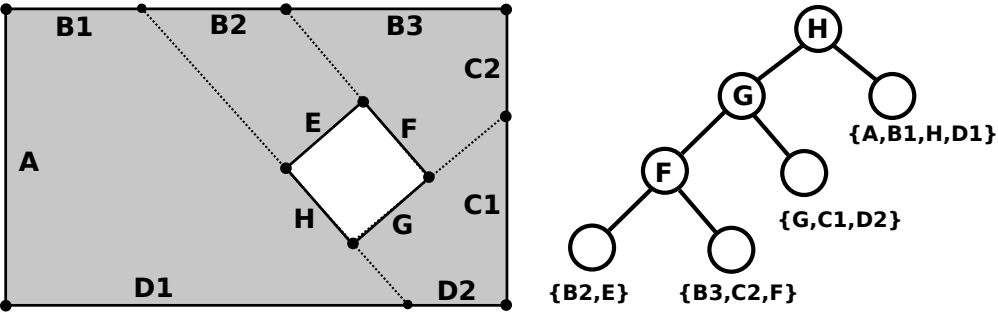
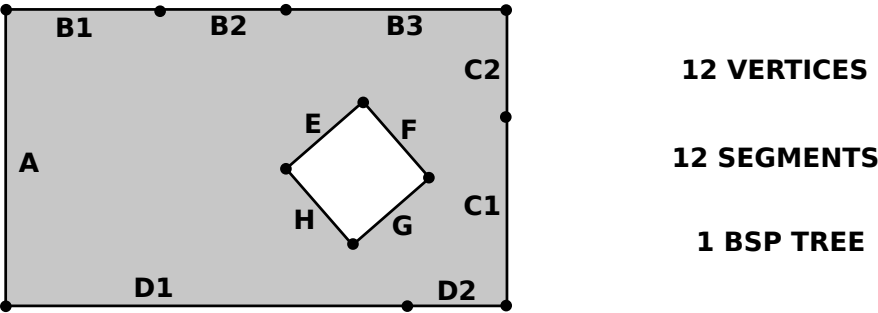


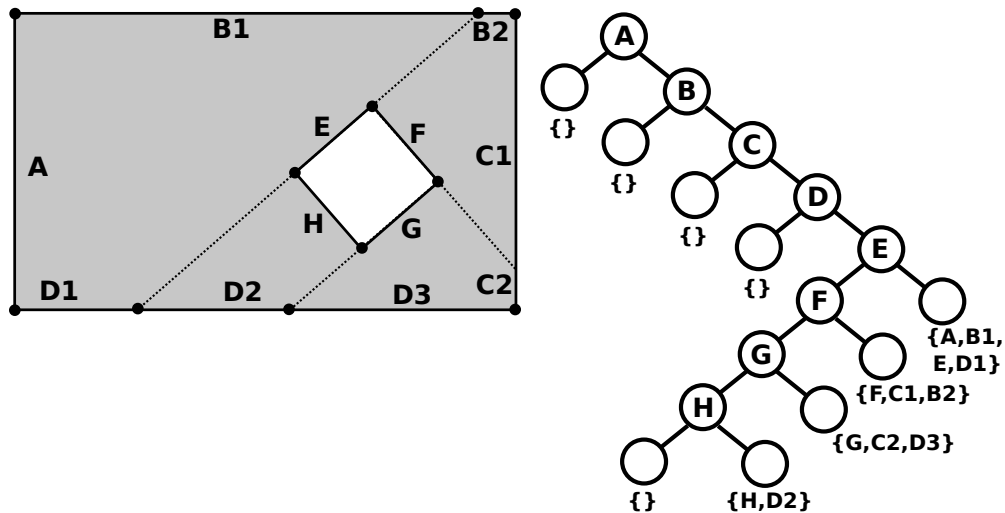
Figure 5.27

With all sub-sectors in the leaves now convex, the BSP construction ends. The number of vertices and segments to deal with has increased by 50% but we now have a data structure capable of sorting all segments, from any point of view, at the cost of only three comparisons.



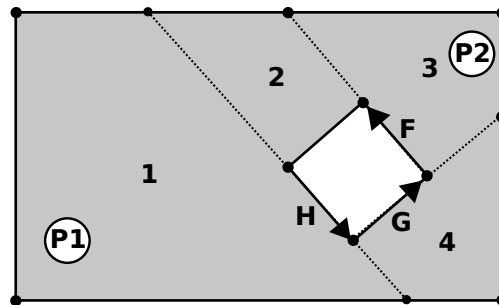
This is only one of the many possible trees which could have been generated from the

map. Choosing splitters in an alphabetical order would have produced an inefficient BSP.



5.12.1.1 Usage

To use the BSP, we only need to traverse it depth first and choose a branch based on our position in the map. Let's take two examples using the BSP in figure 5.27 on page 204. For convenience of notation, sub-sectors have been labeled 1 to 4 and only the splitting lines are marked.



From point of view (P1), traversing the BSP takes three tests. P1 is on the right¹⁷ of H, on the left of G, and on the left of F. This gives the front to back order: 1, 2, 3, 4. Notice that it doesn't matter what order segments within a subsector are drawn since all subsectors are convex.

From point of view (P2), traversing the BSP also takes three tests. P2 is on the left of H, on the left of G, and on the right of F. This gives the near to far order: 3, 2, 4, 1.

The beauty of a binary trees is that traversing it always require the same amount of computation. No matter where we try to place the player on this map, it will always take three tests to sort all subsectors and their segments.

¹⁷On the drawing it is on the left but remember that splitting plans have direction (shown via an arrow head). If you turn the page upside down, sub-sector 1 indeed is on the right of H

5.12.2 Binary Space Partitioning: Practice

Maps were preprocessed on a NeXTstation Turbo via the in-house tool node builder named `doombsp`.

For the tool to build the best BSP possible a splitter selection heuristic had to be established. A good splitter divides the map as evenly as possible (limiting the depth of the tree), and prefers axis-aligned lines (since they are easier to debug and side tests are faster).

`doombsp` recursively inspects all lines in a subspace and gives a splitting score to each of them. At the end of the evaluation, the highest-scoring line is selected. The map is split in two and the process is repeated until only convex subsectors remain. This is a CPU-intensive task which took eight seconds for E1M1. All thirty maps of `DOOM.WAD` took eleven minutes.

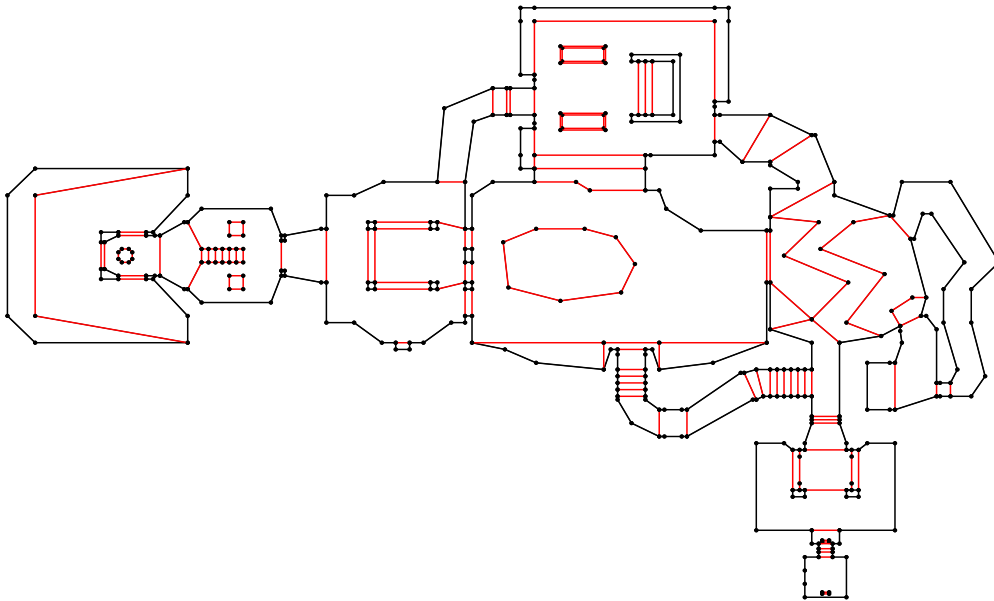


Figure 5.28: E1M1, a.k.a Episode 1 Map 1.

Figure 5.29 shows the seven first splitters selected on E1M1. The first level is in red, second level in blue and the third level in thick black. Notice how AA splitters are favored.

Switching from a sector flooding algorithm to a Binary Space Partitioning algorithm not only added preprocessing time and latency to map designers – there was another side effect that was more of an issue since it affected players. Because the BSP created new vertices, wall positions are set in stone, there is no way to move walls at runtime.

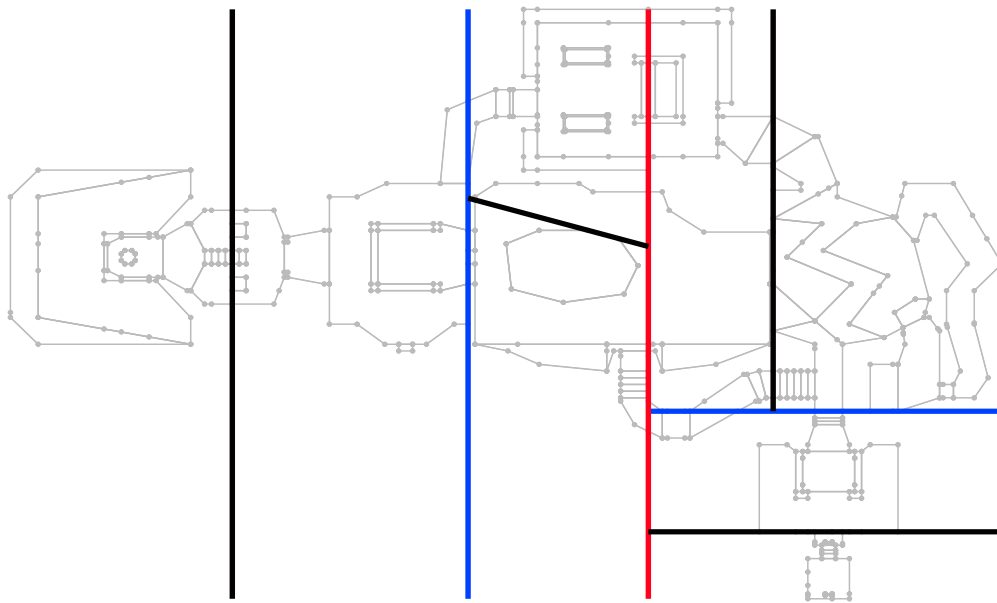


Figure 5.29: Seven first splitters (three BSP branches) in the E1M1 BSP

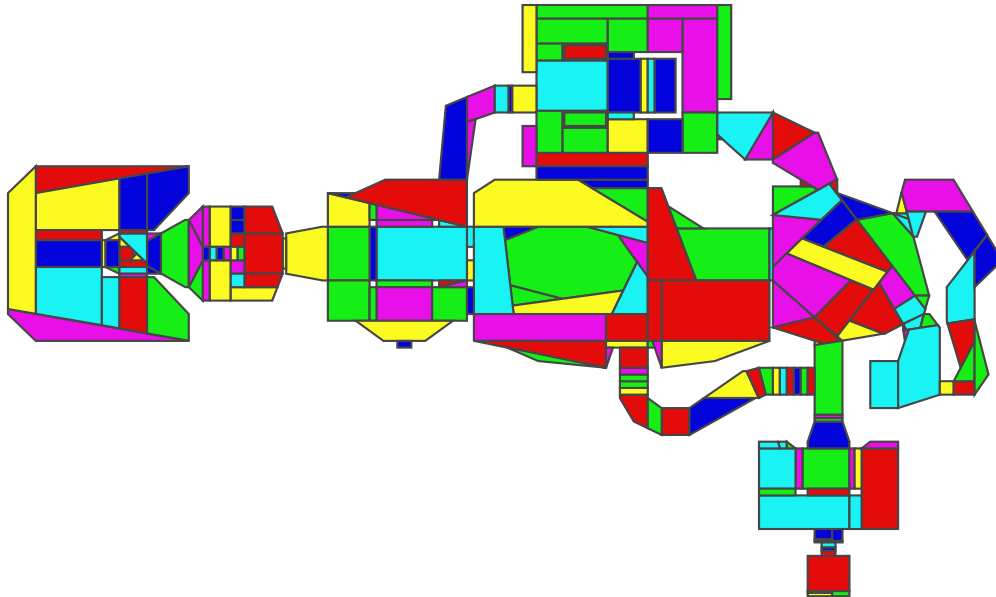


Figure 5.30: All subsectors at the end of E1M1 tree construction; each is a convex leaf.

5.12.3 Drawing Walls

With the expert knowledge of BSP in mind, let's take a look at the first step of rendering a scene, namely, wall rendering. As expected, `R_RenderBSPNode` traverses the binary tree front-to-back. Each sub-sector leaf is sent down to the renderer via `R_Subsector`.

```
void R_RenderBSPNode (int bspnum)
{
    node_t*    bsp;
    int        side;

    // Found a subsector?
    if (bspnum & NF_SUBSECTOR)
    {
        if (bspnum == -1)
            R_Subsector (0);
        else
            R_Subsector (bspnum & (~NF_SUBSECTOR));
        return;
    }

    bsp = &nodes[bspnum];

    // Decide which side the view point is on.
    side = R_PointOnSide (viewx, viewy, bsp);

    // Recursively divide front space.
    R_RenderBSPNode (bsp->children[side]);

    // Possibly divide back space.
    if (R_CheckBBox (bsp->bbox[side^1]))
        R_RenderBSPNode (bsp->children[side^1]);
}
```

To perform the side test (`R_PointOnSide`), any geometry book will describe how to represent the plane in general form with a vector (a, b) combined to a distance d :

$$ax + by + d = 0$$

Using a dot product operation, the coordinate of the test point $P = (x, y)$ is injected into the plane equation, essentially projecting P onto a line perpendicular to the plane. The sign of the result reveals whether P is in front of or behind the plane (zero = on the plane).

This method is far from being optimal. There is a better way, involving neither floating-point nor fixed-point arithmetic, which uses the awesome power of the cross product.

```
typedef struct {
    fixed_t      x,y,dx,dy;          // partition line
    fixed_t      bbox[2][4];         // child bounding box
    unsigned short children[2];      // NF_SUBSECTOR = subsector
} node_t;
```

```
int R_PointOnSide(fixed_t x, fixed_t y, node_t* node){
    fixed_t      dx, dy, left, right;

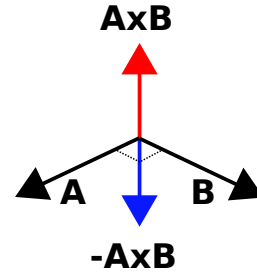
    if (!node->dx) { // Shortcut if node is vertical.
        if (x <= node->x) {
            return node->dy > 0;
        }
        return node->dy < 0;
    }

    if (!node->dy) { // Shortcut if node is horizontal.
        if (y <= node->y) {
            return node->dx < 0;
        }
        return node->dx > 0;
    }
    // Calculate node to POV vector
    dx = (x - node->x);
    dy = (y - node->y);

    if ( (node->dy ^ node->dx ^ dx ^ dy) & 0x80000000 ) {
        if ( (node->dy ^ dx) & 0x80000000 ) {
            // (left is negative)
            return 1;
        }
        return 0;
    }
    // Cross product here
    left = FixedMul ( node->dy>>FRACBITS , dx );
    right = FixedMul ( dy , node->dx>>FRACBITS );

    if (right < left) { // front side
        return 0;
    }
    return 1; // back side
}
```

Notice in the previous code listing how nodes are not stored as coordinates of two vertices (Point 1, Point 2) but rather as (Point 1, Vector to Point 2). This storage technique made the cross-product faster to generate since one of the vectors (from the node) was already calculated.



5.12.3.1 Wall Projection

We have now reached the `R_Subsector` function where all segments in a subsector are rendered in order. This part of the pipeline relies heavily on BAM (Binary Angular Measurement) where degrees in the interval $[0, 360]$ are mapped to the full range of a 32-bit integer.

```
// Binary Angle Measurement, BAM.
#define ANG45 0x20000000
#define ANG90 0x40000000
#define ANG180 0x80000000
#define ANG270 0xc0000000
typedef unsigned angle_t;
```

First, both ends of the segment are converted to an angle with respect to the player's position (via high-school level $angle = \arctan(\frac{O}{A})$). Segments with a negative angle ($angle1 - angle2 < 0$) are culled since they are not facing the camera. Segments passing the angle test are then reduced from 32 bits to 13 bits with a simple right-shift. Next, the angle is injected into a lookup table `viewangletox[4096]` to give a screen-space X coordinate.

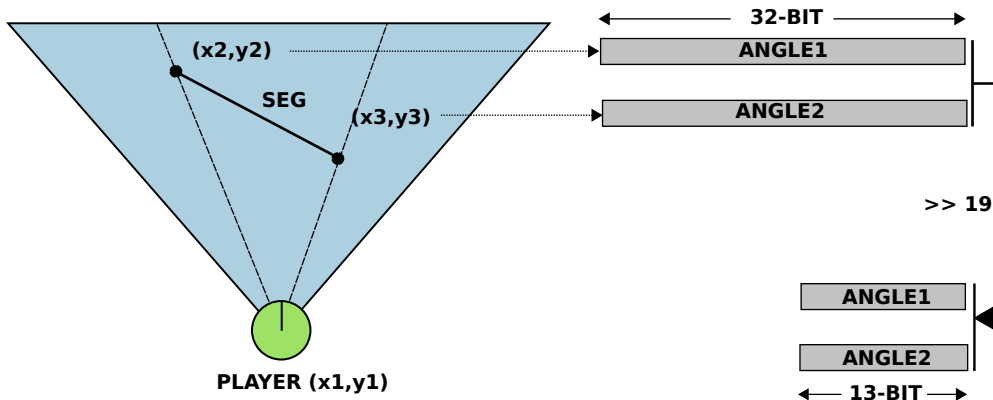


Figure 5.31

Values in `viewangletox` are generated at startup in order to give the player a 90 degree

field of view. The table is built such that anything not within 90 degrees is projected onto the edges of the screen.

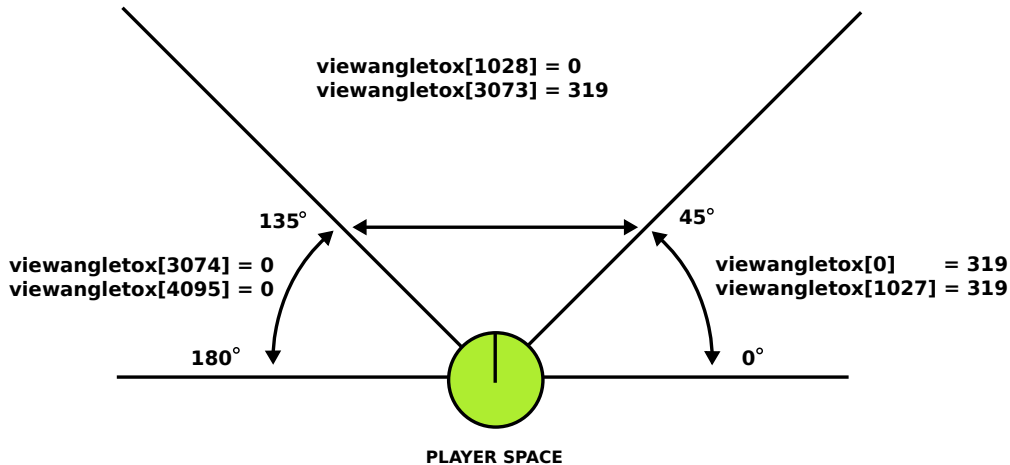


Figure 5.32

At this point the engine has calculated the screen-space X coordinates of both ends of a segment and the distance z from the player. But it is not time to draw yet. A little bit of clipping must occur.

5.12.3.2 Wall Clipping

Here the code branches depending on the two types of segment that can be encountered in a subsector. There are segments with only one side (connected with only one sector) which are opaque and have only a "middle texture". I call these "walls". There are segments with two sides (connecting two sectors) which are often transparent with no middle texture but with an "upper texture" and a "lower texture". I call these "portals".

Clipping is a two step process happening in screen space. The first, crude, step is horizontal-based. Only walls affect the horizontal occlusion array but all segments are clipped against it.

The second, finer, step is vertically-based. Both walls and portals affect the vertical occlusion array and both are clipped against it.

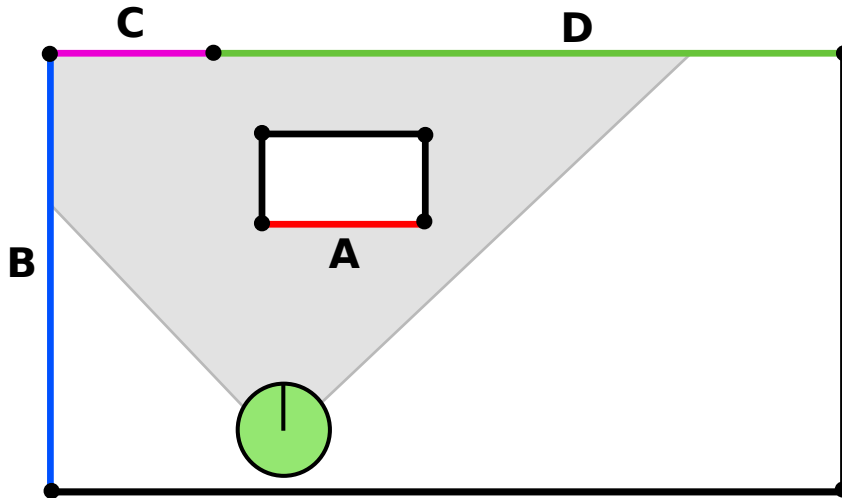
5.12.3.3 Horizontal Crude Wall Clipping

The first clipping pass is crude and only cares about horizontal occlusion. It maintains a `solidsegs` array which keeps track of the screen-space horizontal occlusion. Since portals can be partially seen through, they have no impact on `solidsegs`. Segments entering this step come out as "fragments" since they may be split due to occlusion.

```
typedef struct {
    int first;
    int last;
} cliprange_t;

cliprange_t* newend;
cliprange_t solidsegs[32];
```

Let's take a simple example and proceed step-by-step. In the room below, the player is facing north and four walls (A, B, C, and D) need to be rendered.



Initially the occlusion array has two entries, one representing what is on the left of the screen, from infinity to -1 and one on the right of the screen from 320 to infinity.

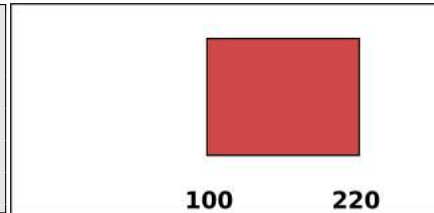
```
solidsegs[0] first = -0x7fffffff
              last  = -1
solidsegs[1] first = 320
              last  = 0x7fffffff
```

0

319

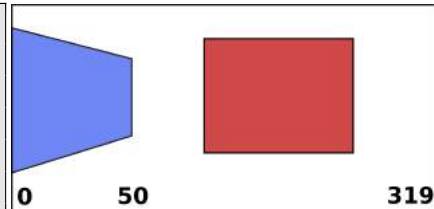
The first wall (A) is rendered. There is nothing to occlude it and it is in the middle of the screen. An entry is added to represent the occlusion state.

```
solidsegs[0] first = -0x7fffffff
              last  =          -1
solidsegs[1] first =          100
              last  =          220
solidsegs[2] first =          320
              last  = 0x7fffffff
```



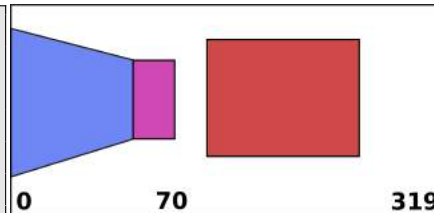
The second wall (B) is rendered. Its left side is clamped via angle adjustment and the right side is not occluded. Since it touches the left edge of the screen no entry in the occlusion array is added; only the entry boundaries need to be adjusted.

```
solidsegs[0] first = -0x7fffffff
              last  =           50
solidsegs[1] first =          100
              last  =          220
solidsegs[2] first =          320
              last  = 0x7fffffff
```



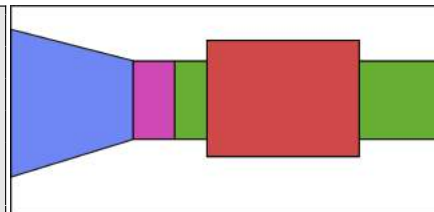
The third wall (C) is rendered. It happens to lie just next to the second wall. It is fully converted to a wall fragment and nothing is discarded. The occlusion array is updated.

```
solidsegs[0] first = -0x7fffffff
              last  =           70
solidsegs[1] first =          100
              last  =          220
solidsegs[2] first =          320
              last  = 0x7fffffff
```



Finally, the last wall (D) is rendered. While occluded against the array, it is split into two fragments. The occlusion array is updated. All segments end up touching each other.

```
solidsegs[0] first = -0x7fffffff
              last  = 0x7fffffff
```



Notice how little RAM was used to maintain the occlusion state and how fast it is to check if the full screen is occluded. All the engine has to do is to check the array is of size 1 and that the range goes from -infinity to +infinity.

5.12.3.4 Vertical Fine Wall Clipping

The second pass is finer and clips vertical segment fragments emanating from the first pass. A data structure based on two arrays as wide as the 3D canvas is maintained. It tracks how much vertical screen-space remains available for each column.

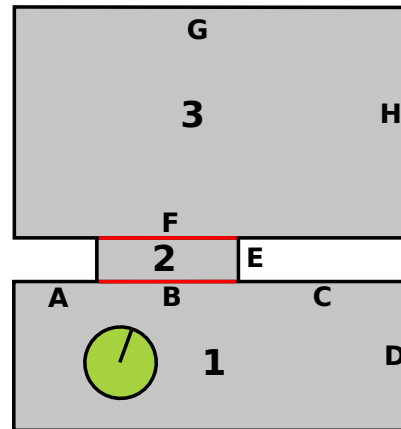
Each rendered segment updates the structure by making increases to `ceilingclip` and decreases to `floorclip`. A column is considered fully opaque when `ceilingclip`'s height and `floorclip`'s height are equal to each other. Wall fragments will mark a column as completely occluded while portal fragments will only update the occlusion columns with what they actually cover in screen-space.

```
#define SCREENWIDTH  320
#define SCREENHEIGHT 200
// clip values are the solid pixel bounding the range
// floorclip starts out SCREENHEIGHT
// ceilingclip starts out -1
short floorclip[SCREENWIDTH];
short ceilingclip[SCREENWIDTH];
```

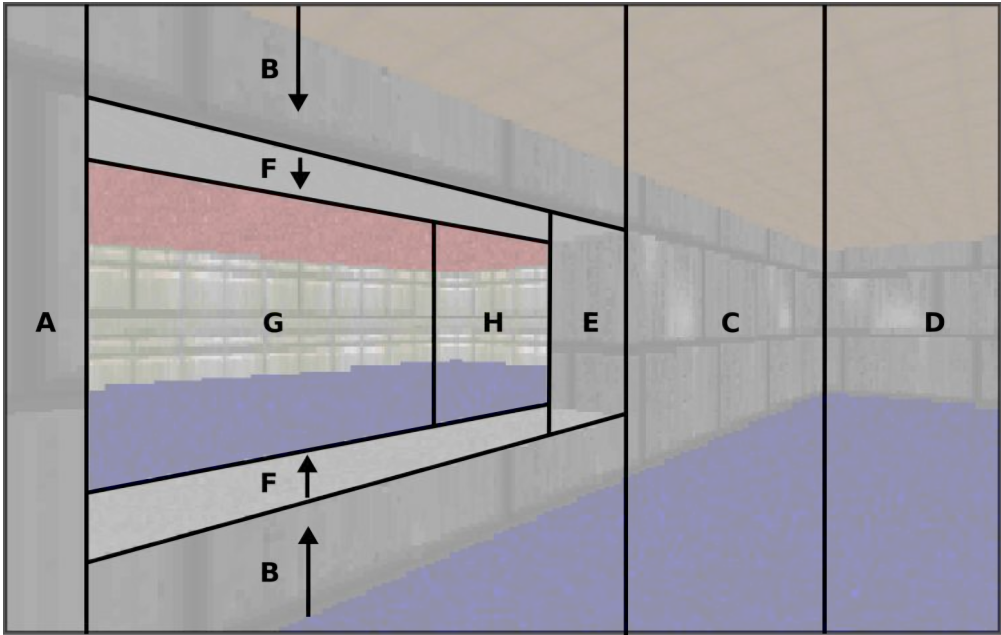
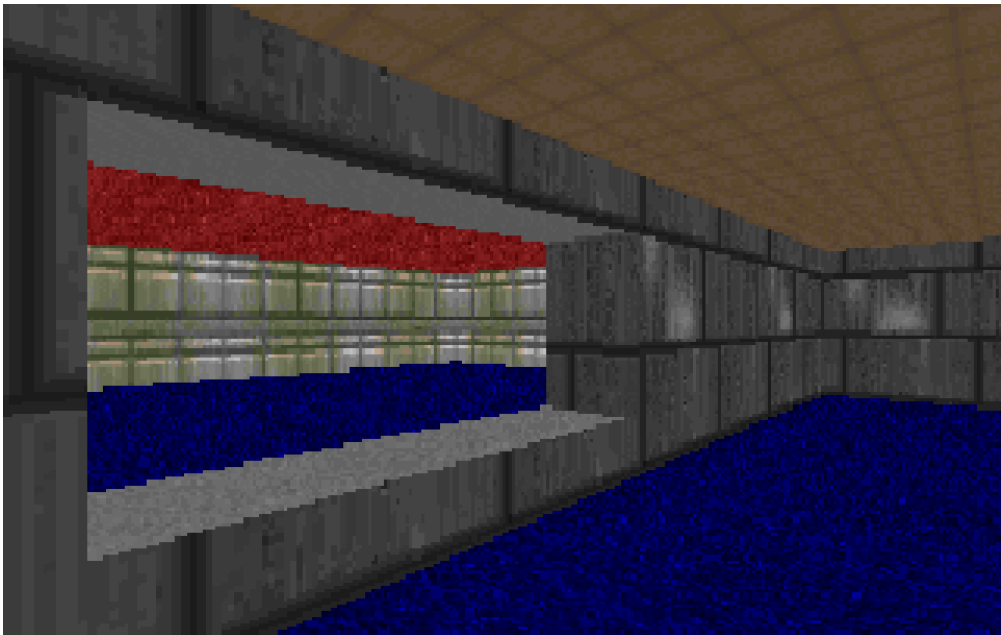
Let's take the example of a simple room made of three sectors (1, 2, and 3) connected by two portals (B and F) and surrounded by walls. Sectors 1 and 3 have the same ceiling and floor height but 2 is set to have a higher floor and lower ceiling so that it looks like a window.

Subsectors are rendered near-to-far in the order 1, 2, and 3. This means segments {A, B, C, D}, {E, F} and {G, H} (assuming the map was split on lines B and F). On the opposite page you can see the effect of each wall and portal on the vertical occlusion double array.

Walls A, C, and D mark the full height opaque for each of their columns. The first portal B has no middle texture. It is therefore rendered with its upper texture (to accommodate for subsector 2's lower ceiling) and its lower texture (to accommodate for subsector 2 higher floor) and the occlusion array is adjusted accordingly. Wall E marks all columns it covers as fully opaque.



Notice how portal F's lower and upper textures are not rendered but the occlusion array is still updated. Walls G and H finish marking the full screen opaque (but were only clipped during the crude pass since they are smaller than the visible window space).



After all this clipping, walls and portal are finally rendered. At the ends of each fragment, a screenspace Y offset is calculated based on sector floor, and a column height is generated based on floor/ceiling and distance. These are interpolated to generate a full set of columns of pixels (portals are drawn as a combination of columns according to their upper texture, middle texture, and lower texture while walls only have a middle texture). Rendering is done vertically via the `colfunc` function pointer (detailed in "performance" on page 286).

5.12.4 Subpixel Accuracy

It is worth mentioning that the engine is subpixel-accurate when calculating the screen coordinates of a wall's top and bottom edges. Subpixel accuracy is a subtle concept, the advantages of which are best demonstrated with an animated screen. Hopefully a few static drawings will do anyway.

Let's take the example of two points, $A = (0.7, 0.7)$ and $B = (5.3, 3.6)$ which are to appear on the screen.

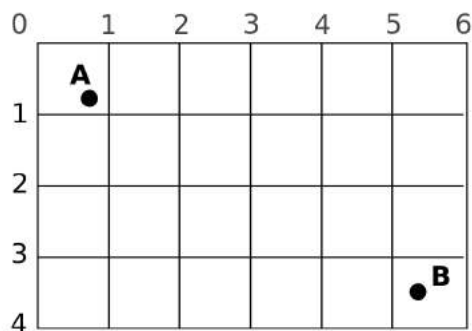


Figure 5.33:

The problem to solve here is: What pixel do you select between A and B? There are many solutions available here, offering different trade-offs. At the time of DOOM's engine, most games discarded the fractional part of a point and then navigated from $\text{floor}(A) = (0, 0)$ to $\text{floor}(B) = (5, 3)$. This is called "pixel-accurate".

Subpixel accuracy is slightly more difficult to perform. Here the fractional part is not discarded but used while navigating from A to B.

Figure 5.34 shows the two methods side by side. The difference looks negligible but it is one of the most important features of the engine which made the world feel "solid".

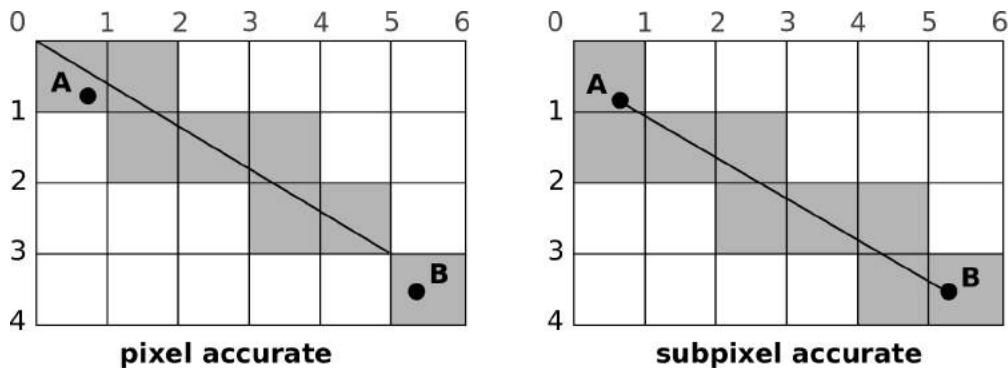


Figure 5.34:

At first sight it seems the two methods are different yet equivalent, but look at what happens when things start to move, such as in the case where A moves 0.3 units down.

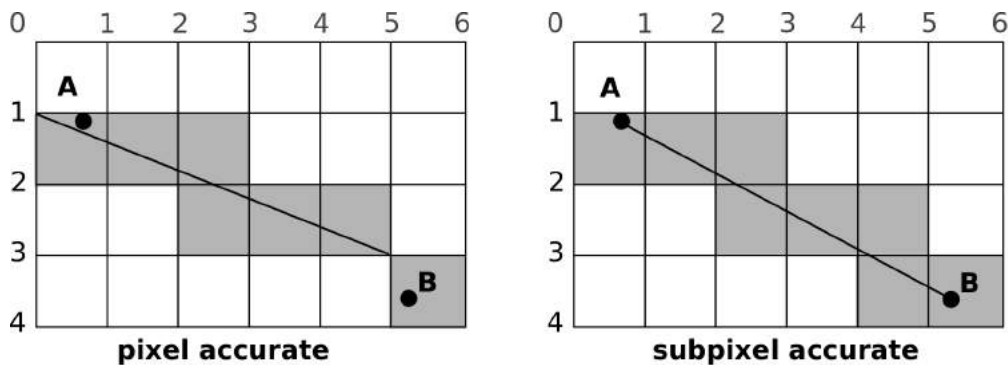


Figure 5.35:

The pixel-accurate method results in five pixels selected differently. The subpixel-accurate way results in one pixel difference. With subpixel accuracy lines tend to be more stable.

“

Almost every other texture mapped game back then snapped triangle vertices to integral pixel values, which meant that the individual texels in a surface would constantly be jumping around by up to a pixel from even tiny movements. Basically everything only feels loosely connected, and wiggles around a bit. DOOM did not have that problem.

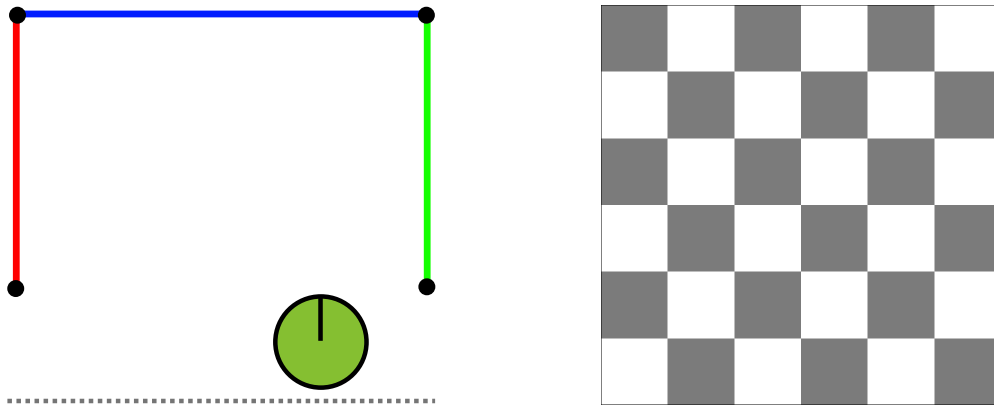
— John Carmack

”

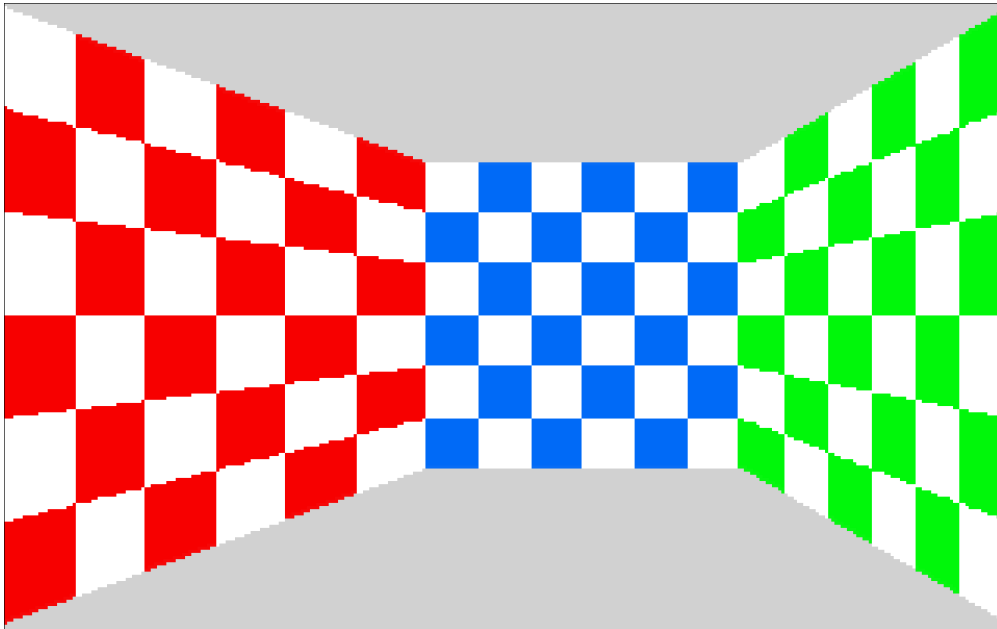
5.12.5 Perspective-Correct Texture Mapping

Before we move on to how DOOM drew flats, it is worth noticing that despite using affine texturing to draw columns, the visual result is still perspective-correct.

To illustrate the concept, let's first study what affine texture mapping looks like. We'll use a three-wall-room, each textured with a pattern of white and colored squares.



In order to focus on walls, the floor and ceiling are intentionally rendered in light gray.



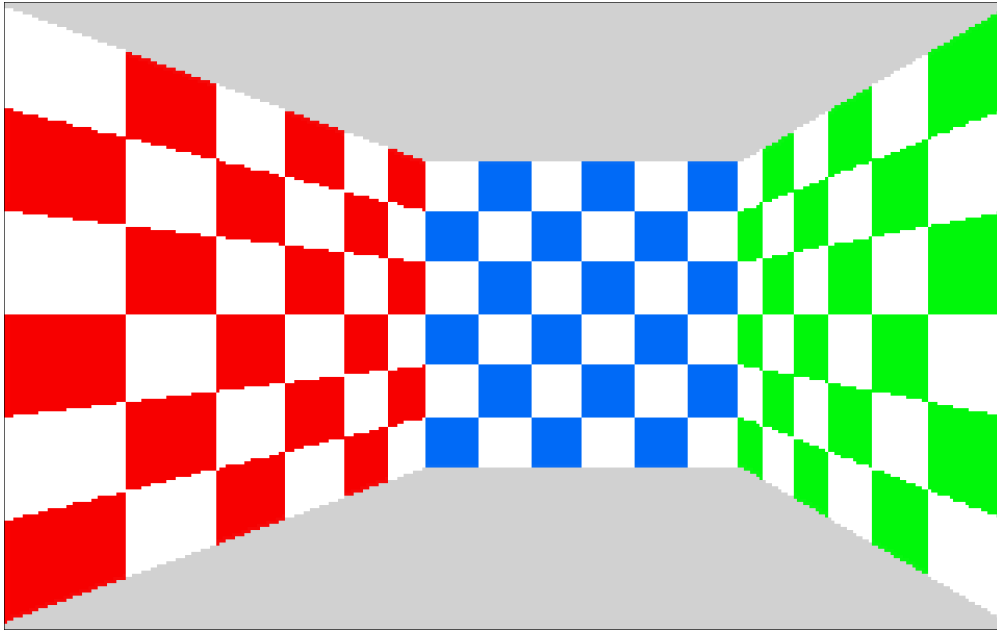
In this scene, wall vertices are projected into screen space coordinates (x, y) . A texture coordinate u is generated for each column to draw, based on linear interpolation of screenspace wall width and the column's x coordinate. The texture coordinate v is interpolated linearly based on column height and y position relative to the column. As a reminder, the formula for linear interpolation is as follows:

$$u_{\alpha} = (1 - \alpha)u_0 + \alpha u_1 \quad \text{where} \quad 0 \leq \alpha \leq 1$$

This texturing technique has the advantage of being fast but has the disadvantage of being visually incorrect. Notice in particular how the "width" of each square is constant even as the wall columns get further away. In order to generate correct texture coordinates, the u coordinate needs to factor in the distance from the player. To this effect, value $\frac{1}{z}$ (which is linear in screen space) is used:

$$u_{\alpha} = \frac{(1 - \alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1 - \alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}} \quad \text{where} \quad 0 \leq \alpha \leq 1$$

The calculation is much more expensive but now the result is perspective correct.



Had DOOM allowed sloped walls, texturing would have required six perspective correct computations per pixel (both u and v interpolated twice along vertical and horizontal edges of a quad) which would have been prohibitively CPU intensive.

By enforcing walls to be strictly vertical, DOOM was able to perform the expensive perspective correct computation only once per pixel column, use linear interpolation to draw each column, and still yield a visually correct result. It works because along a column, the distance from the player is constant. This trick effectively managed to produce perspective correct texture mapping at the computational cost of affine texturing.

Given the previous screenshots, it may not be clear why perspective correctness is so important and why the engine goes to such an extent just to be "correct". Keep in mind these were just examples to demonstrate the problem. As soon as real textures are used (using DOOM's marble textures featured below), the visual disturbance cannot be ignored.



Figure 5.36: MWALL4_1



Figure 5.37: MWALL4_2



Figure 5.38: MWALL5_1

On the opposite page, notice in particular how MWALL4_1's upper right pentagram edge appears arched instead of being straight. The same thing happens with MWALL5_1 where the left horn does not look right. MWALL4_2 suffers none of these issues since it is parallel to the viewing angle.

Several video game consoles of the era such as the PlayStation, the 3DO, and the Saturn¹⁸ featured hardware accelerated graphics but not perspective correctness. On these machines, affine texture mapping artifacts were supposed to be compensated for either via triangle sub-divisions or by avoiding texturing altogether. This is why PSX game *Crash Bandicoot*'s characters are devoid of texturing in favor of Gouraud shading.

John Carmack hated affine texturing so much he vetoed all ports attempting to be perspective incorrect which sometimes resulted in serious consequences (page 352).

¹⁸Thanks to a close collaboration with SGI, Nintendo gifted the Nintendo 64 with perspective correct texturing which made *Zelda* and *Mario* look amazing.



Above, distorted walls due to affine texturing. Below, perspective correct texturing.



5.12.6 Drawing Flats

If we were to take a look at the framebuffer at this point in the rendering of a frame, it would look like mashed potatoes (see opposite page where flats are in white to ease visualization). DOOM never clears the framebuffer so instead of white there would be whatever was drawn last frame.

To render the flats, the engine uses a data structure generated while the walls and portals were being rendered. These are called "visplanes".

```
// Now what is a visplane, anyway?
typedef struct {
    fixed_t      height;
    int          picnum;
    int          lightlevel;
    int          minx;
    int          maxx;
    // 4 padding bytes
    byte         top[SCREENWIDTH];
    byte         bottom[SCREENWIDTH];
} visplane_t;

// Here comes the obnoxious "visplane".
#define MAXVISPLANES      128
visplane_t      visplanes[MAXVISPLANES];
visplane_t*     lastvisplane;
```

The concept of visplanes is the most difficult aspect of DOOM to understand. The comments in the source code manifest their esoteric nature and also attest that even people closely related to id Software did not fully grasp what they were.

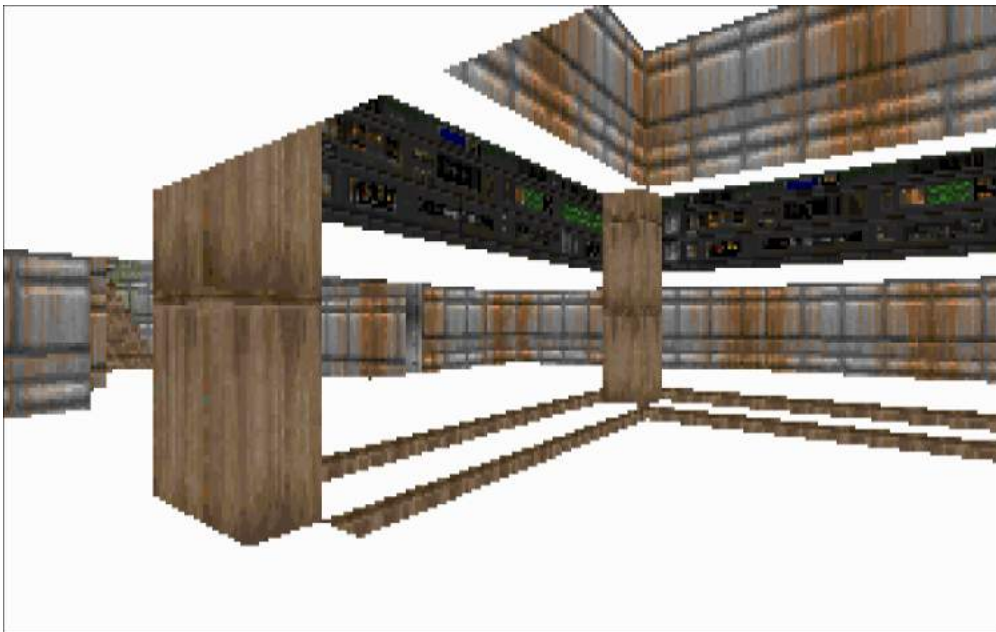
A visplane describes a screen-space area representing either a ceiling or a floor. It has a height, a texture (picnum), and a light level. To describe the limits of its area, it has two arrays as wide as the screen. Areas are represented as a set of columns with one column possible per X coordinate.

Trivia : The engine stores visplanes before drawing them. If storage runs out, the engine terminates execution and returns to DOS with a less than useful error message.

```
C:\DOOM>R_FindPlane: no more visplanes
```



Above is the final frame. Below is the current state of the frame with missing visplanes.

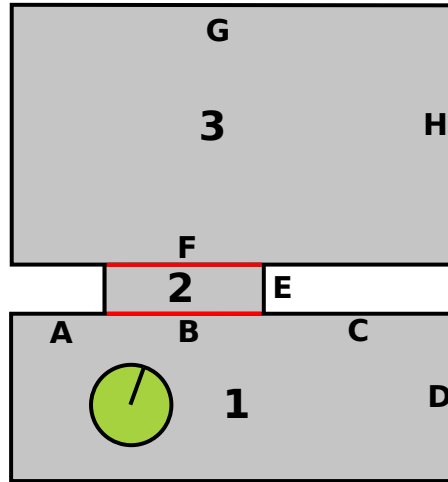


Visplanes represent the vertical "gaps" in screen-space between fragments and screen borders or between walls and portals. To better understand how they are generated, let's take an example and go back to the simple room we just studied. This time we will focus on how the `visplanes` array is populated.

As in the previous example, sectors are rendered near to far, resulting in segments in the following order: A, B, C, D, E, F, G, and H.

When wall A is rendered, it is clipped horizontally and vertically. Since it uses all the vertical screen space, no visplanes are created.

Things get slightly more interesting when walls C and D are rendered. Since they do not occupy the full height (there is a gap between the screen and the top/bottom edge of the walls), two visplanes are created per fragment – (1,2) and (3,4) – and stored in the `visplane` array.



Likewise, when portal B is rendered, there are gaps above its upper texture and below its lower texture, so two additional visplanes (5 and 6) are added.

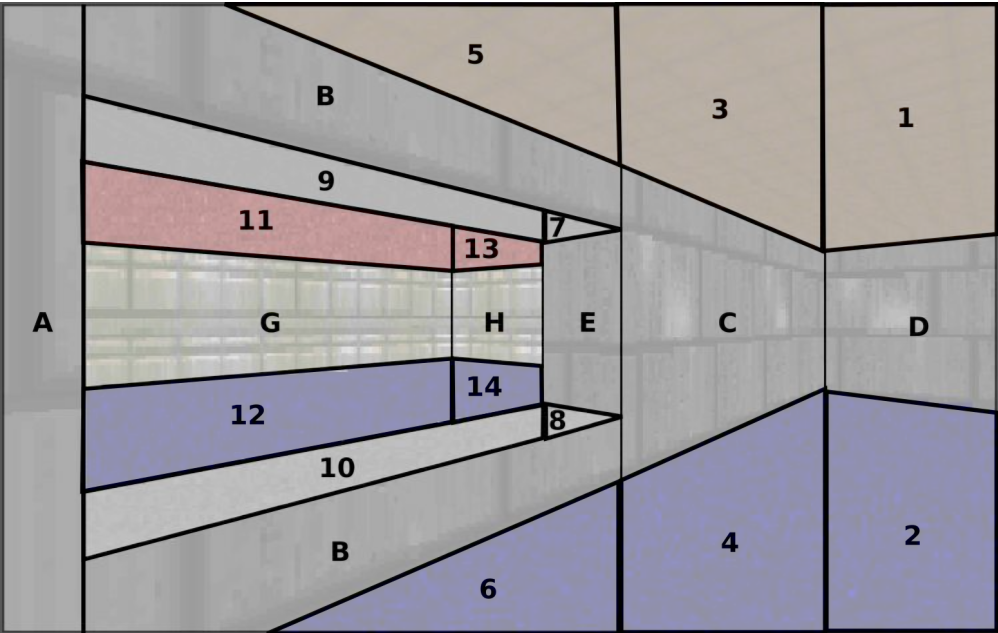
Wall E is another new case. The gaps above and below are not between a wall and the screen but between E and portal B's upper and lower parts. To detect the previous boundaries, the previously-seen vertical occlusion array is used to create visplanes 7 and 8.

Portal F is yet another special case. Since from this point of view it is connecting to a sector with a higher ceiling and a lower floor and has no middle texture, nothing is rendered. Yet the coordinates of its middle part are still generated in order to generate visplanes 9 and 10.

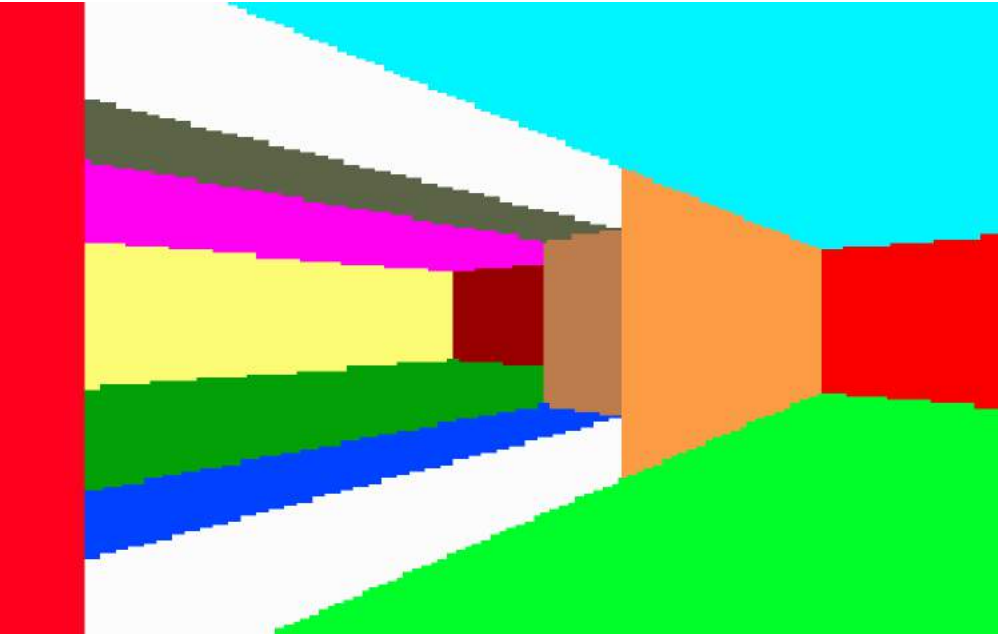
From this point on, the process is repeated. Wall G's rendering generates visplanes 11 and 12 and wall H's rendering generates visplanes 13 and 14.

The visplane generation algorithm is quite simple. However it generates many visplanes and therefore consumes a lot of RAM. To address this issue, DOOM merges visplanes.

Trivia : The `visplane_t` struct requires 664 bytes per visplane. The engine reserves space for 128 visplanes. The total represents a significant amount of RAM (84,992 bytes) accounting for roughly two percent of the minimum required (4MiB).



Below, the engine was modified to draw walls and flats in plain color to show merging.



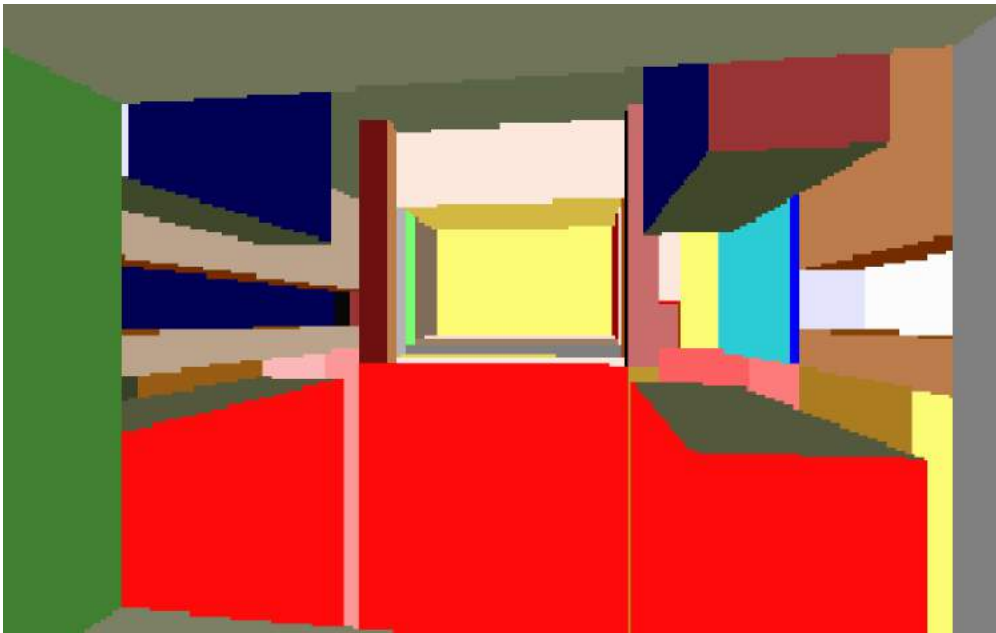


E1M1 benefits from merging. Reproduced below without diminished lighting for clarity.





Without merging (above) the frame requires 179 visplanes. With merging (below), only 28 are needed.



On the previous page, the modified engine shows how a visplane does not need to be horizontally continuous. The red floor's visplane for example is made of three parts yet uses only one entry in the visplane array, validating the judicious choice to represent visplanes as an array of columns.

There are several conditions required to merge two visplanes. Mergeable visplanes must have the same height, light and texture. Even if these conditions are met, the data structure has to be able to absorb others. This is only possible if visplanes are side-by-side. Even one pixel above or below with the same X coordinate makes two visplanes ineligible for merging. Function `R_FindPlane` looks for candidates; mergability is tested elsewhere.

```
visplane_t* R_FindPlane(fixed_t height, int picnum, int
    lightlevel) {
    visplane_t*    check;
    ...
    for (check=visplanes; check<lastvisplane; check++) {
        if (height == check->height &&
            picnum == check->picnum &&
            lightlevel == check->lightlevel)
            break;
    }

    if (check < lastvisplane)
        return check;

    if (lastvisplane - visplanes == MAXVISPLANES)
        I_Error ("R_FindPlane: no more visplanes");

    lastvisplane++;
    check->height = height;
    check->picnum = picnum;
    check->lightlevel = lightlevel;
    check->minx = SCREENWIDTH;
    check->maxx = -1;
    memset (check->top, 0xff, sizeof(check->top));

    return check;
}
```

Notice the overflow check which was mentioned earlier. This case was a map designer's worst nightmare since it meant abrupt termination without much information provided to fix the error. Many legends and theories circulated until the source code was released¹⁹.

¹⁹"The Facts about Visplane Overflows" by Lee Killough

5.12.7 Drawing Flats (For Real)

We can finally read the flat drawing routine which iterates over the array of visplanes.

```
void R_DrawPlanes (void) {
    visplane_t *pl;
    int light;
    int x, stop;
    int angle;

    for (pl = visplanes ; pl < lastvisplane ; pl++) {

        if (pl->minx > pl->maxx)
            continue;

        // sky flat
        [...] // Special case where perspective is disabled.

        // regular flat
        ds_source = W_CacheLumpNum(firstflat + flattranslation[
pl->picnum], PU_STATIC);
        planeheight = abs(pl->height-viewz);
        light = (pl->lightlevel >> LIGHTSEGSHIFT)+extralight;
        planezlight = zlight[light];

        pl->top[pl->maxx+1] = 0xff;
        pl->top[pl->minx-1] = 0xff;

        stop = pl->maxx + 1;
        for (x=pl->minx ; x<= stop ; x++)
            R_MakeSpans (x,pl->top[x-1],pl->bottom[x-1] ,
                pl->top[x],pl->bottom[x]);

        Z_ChangeTag (ds_source , PU_CACHE);
    }
}
```

Notice how the flat texture resource is not freed at the end of each iteration but rather marked as `PU_CACHE` according to what was described in the memory manager section.

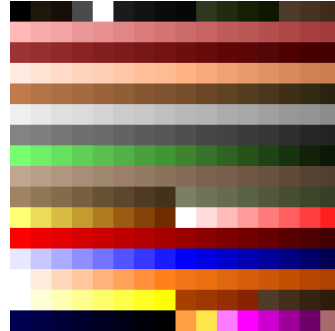
Despite being stored as columns, visplanes are converted to horizontal spans. Rendering this way allows for fast perspective correct texturing since each line is at a constant distance from the player. This choice also allowed fast rendering of something we have so far ignored: diminished lighting.

Trivia : The limitation of 128 visplanes was not only necessary to fit within the memory budget, it was also a runtime necessity. When attempting to merge visplanes, the engine searches linearly, making it a $O(n)$ operation which would become a bottleneck as the number of visplanes increased. In 1997, Lee Killough lifted this limitation, replacing linear search with a $O(1)$ chained hash table²⁰.

5.12.8 Diminishing Lighting

So far, in order to introduce complexity gradually, our trip down the rendering pipeline has completely ignored diminishing lighting. It was assumed that texel values from texture and sprite were used as-is and written directly to the framebuffer. Now it is time to introduce the concept of lightmaps.

In order to convey a scary atmosphere, à la *Aliens*, it was decided from day one that with increasing distance, colors would fade to black. Map designers also wanted to be able to switch the light in a room on and off and also to dim it if necessary. In response to this requirement, the engine had to be able to draw shades of color. With the VGA system limited to 256 colors from a palette, a possible implementation would have been to restrict artists to use 16 colors and use the 240 other slots to generate 15 shades of each "primary" color.



This would have severely impaired the work of the artists, not to mention it would have looked poor during outdoor scenes where light is not diminished with distance. Once again a clever trick allowed artists to use the full 256 colors for their assets and not 16 but 32 shades of the same color. On paper that would have meant $256 * 32 = 8192$ colors which the VGA hardware did not support.

The trick to fake more colors than available is to use an indirection "light table" where the other 255 colors are used to approximate a gradient for each of the 256 entries. The lightmap is 256 entries tall (one for each index) and 32 wide (one column for each shade). In figure 5.39 you can see how the original palette lines are unrolled vertically in the left-most column (notice the isolated white at the top and the pink at the bottom). Each row is a 32 value gradient toward black, using the same 256 colors. The right-most column is all black. The trick has its limits. It works well for red but not too well for crimson and yellow.

To use the lightmap, take the original texel value T which is between $[0, 255]$. This will be the Y coordinate. Take a light value L , with 0 being the brightest and 31 being the darkest. This will be the X coordinate. The value to write in the framebuffer is `lightmap[X][Y]`.

²⁰Source: "The Truth about Visplane Overflows".

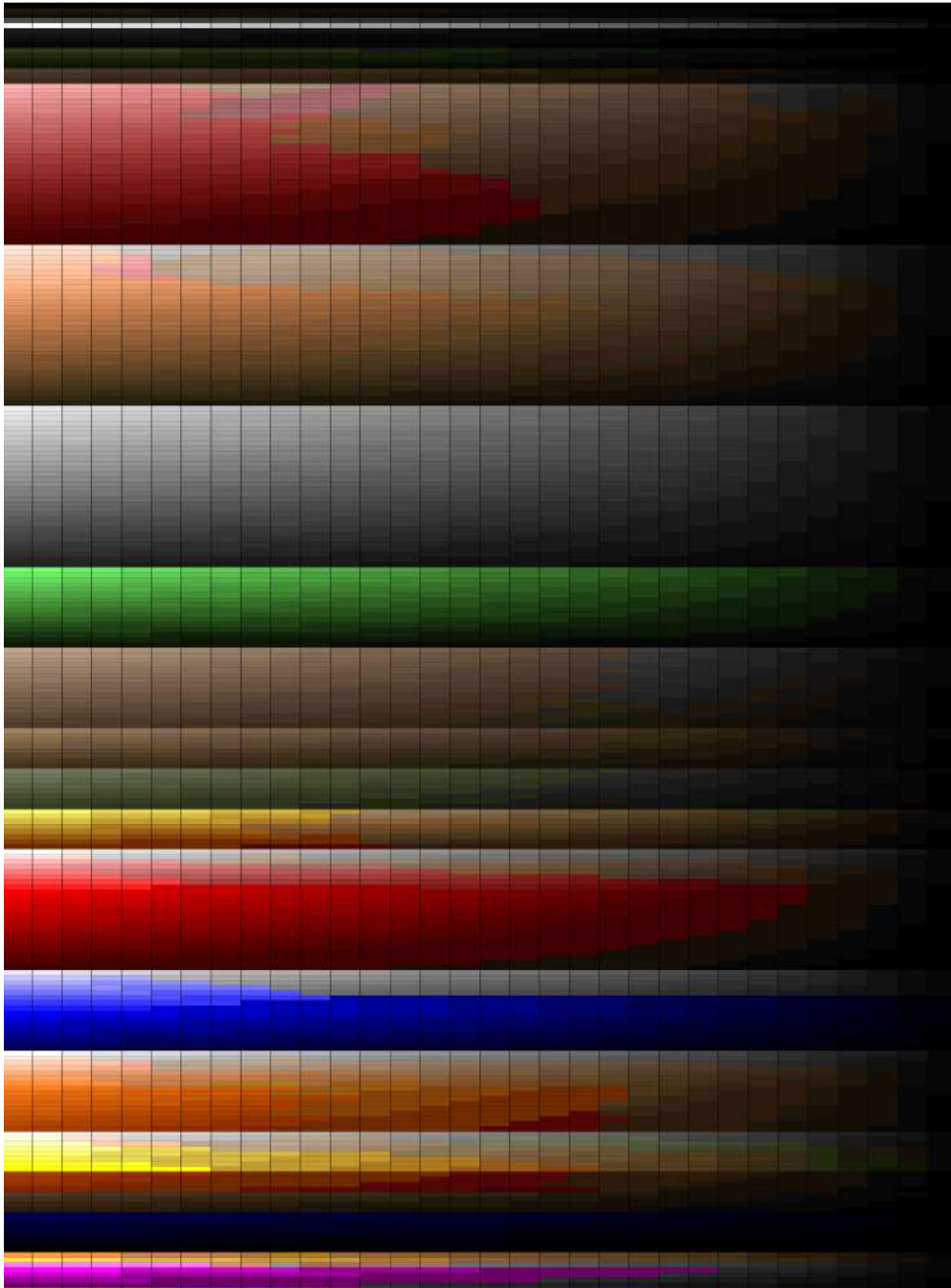


Figure 5.39: The COLORMAP contains 32 shades of 256 colors yet still 256 colors total.



Above shows the vanilla engine (with thing rendering disabled). Below is the same scene with a modified lightmap-less engine.



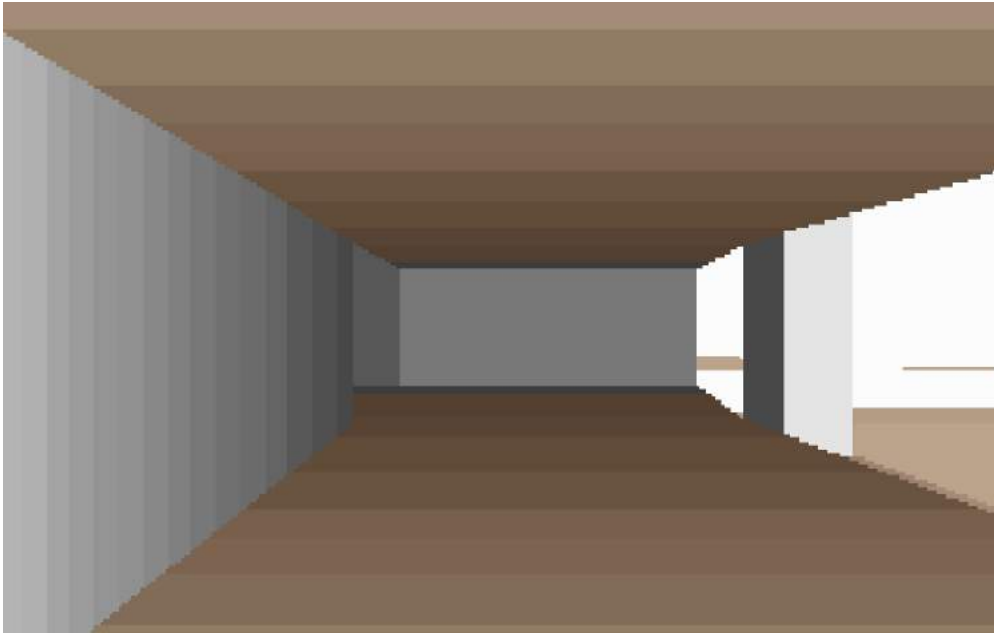


Figure 5.40: Same scene as opposite page, with lightmaps but with texturing disabled.

The visual effect of lightmaps can sometimes be subtle. On the opposite page, a normal scene (above) shows no banding. Disabling lightmaps altogether (opposite page, below) makes the colors appear washed out. Disabling texturing by rendering walls in white (0x04) and flats in brown (0x80) (as in figure 5.40) makes lightmaps and banding vividly apparent.

Because calculating which lightmap to use is based on the distance from the player and also on the sector light level, it is a slightly expensive operation.

$$lightmapId = sectorLightLevel + z * diminishingFactor$$

$$color = lightmapId[textureTexel]$$

In scene where a lot of visplanes are on screen, even selecting the proper lightmap ID was a performance hit. Therefore, a cache system keeping track of lightmap ID on a per screenspace line/sector ID mitigate how often the value is calculated.

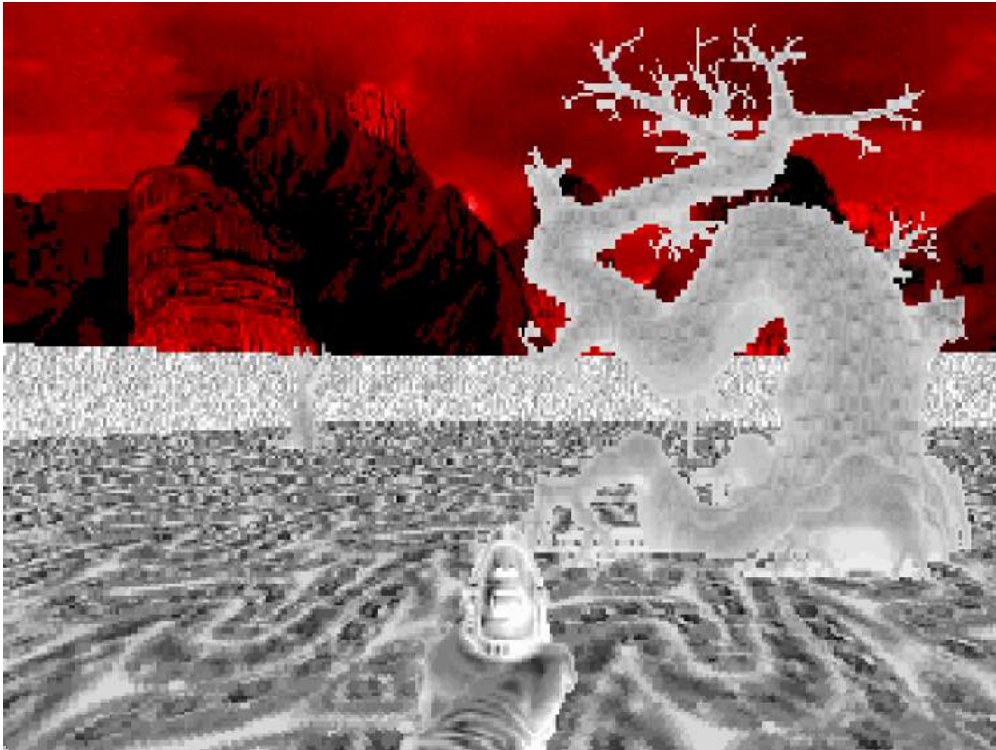
The engine uses a cool trick to embellish orthogonal surface rendering. Lightmap selection is tweaked based on world-space orientation. For north-south walls, the selection is

lowered by one unit, making them brighter. For east-west walls the lightmap selection is increased by one unit, making them darker. Lightmap selection on other walls is unaffected.

To render the sky, a sector has a special ceiling texture number which the engine recognizes as "sky", in which case its height is 0, lightmap is 0 and perspective is disabled. Each portion of the sky is stored as a visplane and drawn as a column of pixels (with `colfunc`).

The `COLORMAP` contains a 32nd lightmap made of 256 shades of grey. It is used when the player picks up the invulnerability bonus. There is a bug in the visplane rendering routine's special casing that handles skies. Skies are always rendered without diminished lighting and therefore ignore lightmaps.

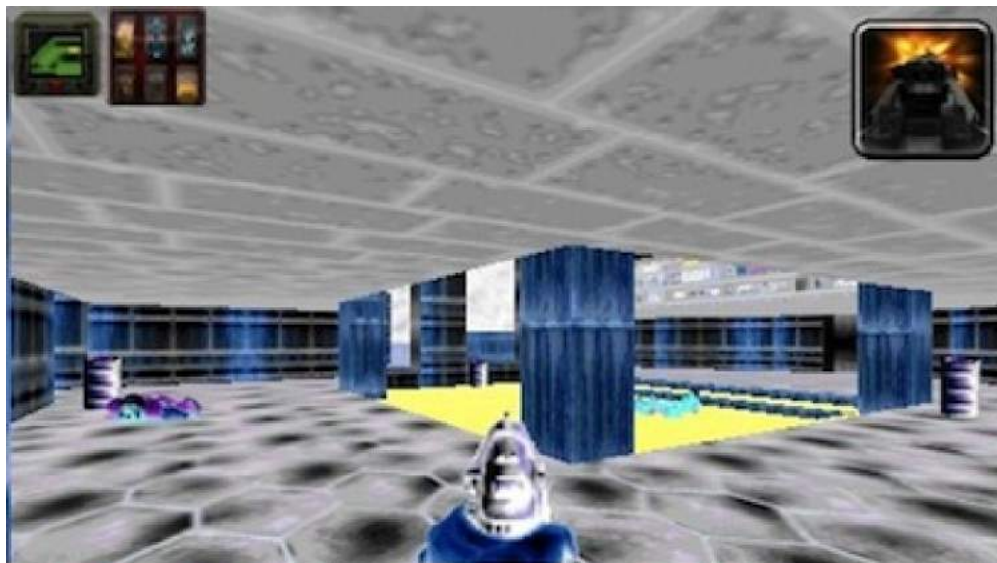
This causes a weird visual result when a player is outside and picks up the invulnerability orb, where everything except for the sky is rendered with a shade of gray.



The effect was difficult to replicate with seemingly "more powerful" hardware accelerated systems which use 24-bit colors. On iOS, `glBlendFunc(GL_ONE_MINUS_DST_COLOR, GL_ZERO)` was used to approximate the same visual with mixed results.



Above is the invulnerability effect as it appeared on the PC version. Below, how it was done on iOS via OpenGL ES 1.0 `glBlendFunc` function.



5.12.9 Drawing Masked

With the environment rendered, what remains are "masked" elements. This category encompasses not only all sprites but also partially-transparent walls and the player's weapon.

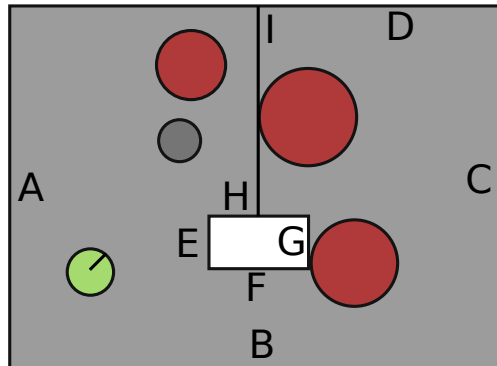


The function responsible for this task is called `R_DrawMasked`. It is the last step in the rendering pipeline. Contrary to the environment which is rendered in front-to-back order, this step proceeds in back-to-front order (which is the only way to get transparency right).



Before diving into `R_DrawMasked` function, let's take the example of a simple room containing all types of "masked" elements and reexamine what needs to be done.

The diagram shows a room with four walls (A, B, C, and D), a pillar also made of four walls (E, F, G, and H), a transparent wall I, a barrel (in grey), a player spawn point (in green) and three enemies (in red: a Baron of Hell, a Cacodemon, and a Demon).



The result as rendered by DOOM is visible in figure 5.41. Notice how the Baron is drawn on top of wall I, the Cacodemon is partially occluded by wall I, and parts of the demon are completely clipped behind the pillar. Also notice how the barrel must be drawn in front of the Baron for correctness. Last, wall I needs to be clipped against the pillar.



Figure 5.41: Scene with all Things and the transparent wall rendered

With the desired result in mind, let's go back to where we were with only walls and flats rendered in the framebuffer. In the case of the same room, it would look like figure 5.42.

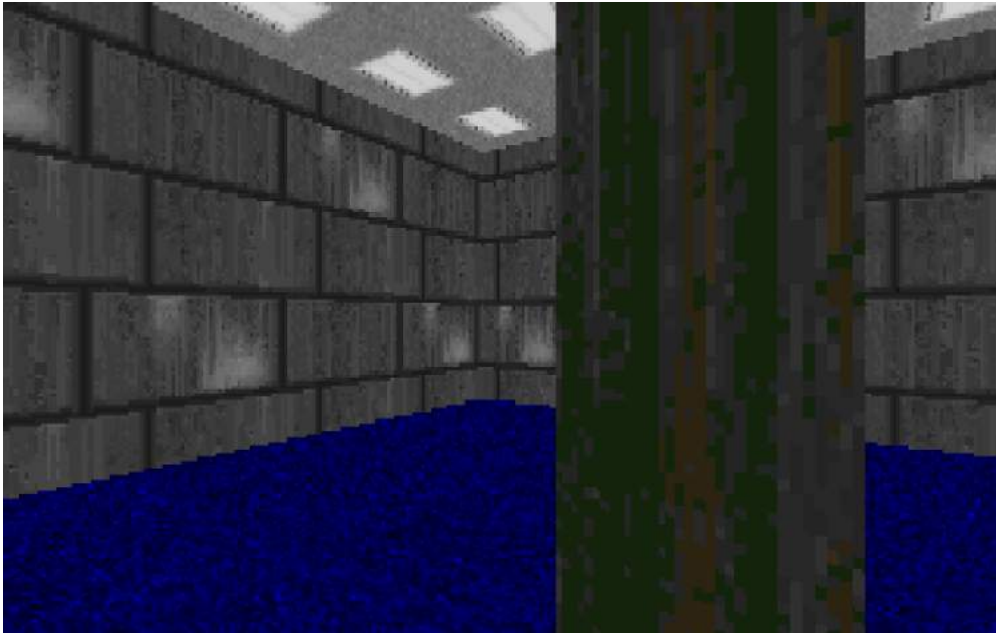


Figure 5.42: Same scene as it is currently stored in the framebuffer

To get from figure 5.42 to figure 5.41, the engine needs to generate a list of maskables (also simply called sprites in the code), to sort them into back-to-front order, iterate over the sorted list, perform clipping on each of them, and finally render them as columns of pixels.

5.12.9.1 List Of Things

The list of things was built while the BSP was traversed. Each time a subsector was sent to rendering (in `R_Subsector`) the list of things it contained were added to an array of `vissprite_ts`. Note that things are only pseudo-ordered and therefore the order is not usable. In our example the barrel and the Baron would be added based on the subsector they were in, but in no precise order (in the subsector's list of things, the baron could be returned before the barrel).

5.12.9.2 Clipping Information

The clipping information was also built while rendering walls. Ideally it would have been a depth buffer but RAM was expensive and not fast enough. The solution was to record

anything that could potentially appear on the screen in an optimized array of `drawseg_ts`.

The "drawn segments" array `drawsegs` is made of the following `drawseg_t` struct.

```
typedef struct drawseg_s {
    seg_t    *curline;
    int      x1, x2;
    fixed_t  scale1, scale2, scalestep;
    int      silhouette;    // 0=none, 1=bottom, 2=top, 3=both
    fixed_t  bsilheight;    // don't clip sprites above this
    fixed_t  tsilheight;    // don't clip sprites below this
    // pointers to lists for sprite clipping
    short    *sprtopclip;   // adjusted so [x1] is first value
    short    *sprbottomclip; // adjusted so [x1] is first value
    short    *maskedtexturecol; // adjusted so [x1] is first
                          value
} drawseg_t;

#define MAXDRAWSEGS      256
drawseg_t  drawsegs[MAXDRAWSEGS];
```

It is not easy to understand at first but it is much easier to think of it as a log of what occluders were rendered to the screen.

- For each wall that generated pixels in the framebuffer, a `drawseg_t` is added.
- For each portal, up to two `drawseg_ts` are added (one for the upper part and one for the lower part if the portal had no middle texture).
- For each masked segment (like the grid I in our example) that was skipped, one `drawseg_t` entry is added to record what should have been drawn.

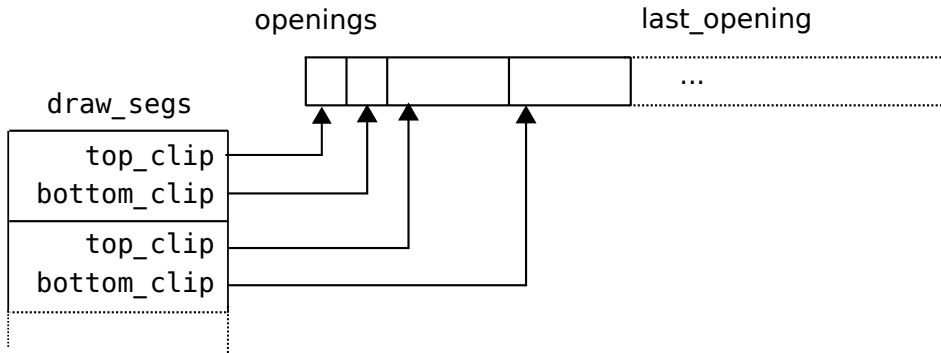
The log entries are ordered by distance from the player (since each entry was added while rendering the walls). The distance is not a `z` value but a `scale` value. The idea is to replay a portion of the log for each thing (everything in front of the thing) in order to build an accurate occlusion rectangle. Once the occlusion rectangle is obtained, the Thing is clipped against it. Each `drawseg_t` features the screen-space horizontal boundaries (`x1` and `x2`) as well as their respective scales (`scale1` and `scale2`). Since a `scale` value is generated for each Thing (based on its distance from the player), it is used to know what portion of the `drawsegs` array to replay.

The screen-space horizontal top and bottom edge of each wall are obtained from pointers `sprtopclip` and `sprbottomclip` which point to an array shared by all `drawseg_ts`.

```

#define MAXOPENINGS    SCREENWIDTH*64
short                 openings[MAXOPENINGS];
short*                lastopening;

```



Now that we know how they are clipped, let's take a look at how sprites are stored.

```

typedef struct vissprite_s {
    struct vissprite_s* prev; // Doubly linked list.
    struct vissprite_s* next;
    int x1;
    int x2;
    fixed_t scale;
    int patch;
    lighttable_t* colormap;
} vissprite_t;

#define MAXVISSPRITES 128
vissprite_t vissprites[MAXVISSPRITES];
vissprite_t* vissprite_p;
vissprite_t vsprsortedhead;

```

A vissprite entry contains everything needed to render a sprite on screen. It features information similar to the drawn segments such as the screen space horizontal boundaries ($x1$ and $x2$), its scale to compare distances with walls, the texture ID (`patch`) and the lightlevel (`colormap`) to be used for shading.

Notice the `prev` and `next` fields. Even though vissprites are stored in a linear array, they need to be sorted into back-to-front order. Instead of moving around array elements, the sorting method only updates the doubly-linked list. This way, elements remain at the same array position but a sorted list can be obtained by following the `next` pointers.

Let's finally look at how all this data is used to render the sprites and masked segments in `R_DrawMasked`.

```
void R_DrawMasked (void){
    vissprite_t  *spr;
    drawseg_t    *ds;

    R_SortVisSprites ();

    // draw all vissprites back to front
    if (vissprite_p > vissprites) {
        for (spr= vsprsortedhead.next ; spr != &vsprsortedhead;
             spr = spr->next)
            R_DrawSprite (spr);
    }

    // render any remaining masked mid textures
    for (ds=ds_p-1 ; ds >= drawsegs ; ds--)
        if (ds->maskedtexturecol)
            R_RenderMaskedSegRange (ds, ds->x1, ds->x2);

    // draw the psprites on top of everything
    if (!viewangleoffset) // don't draw on side views
        R_DrawPlayerSprites ();
}
```

As expected, the list of visible sprites is sorted based on their distances to the player (`R_SortVisSprites`). This process is fast since only scale has to be compared and no data is copied into the array, the doubly-linked list is only updated to move an element.

Next, all sprites are rendered one-by-one in a back-to-front fashion. The method taking care of this (`R_DrawSprite`) scans the array of `drawsegs` linearly to find what wall fragments were drawn in front of the sprite and clips it accordingly. Since the search is based on the scale of each `drawseg_t` and the screen-space X boundaries it is a fast operation.

The last step is to render the masked segments which were skipped during BSP traversal. This is done in function `R_RenderMaskedSegRange`.

As explained earlier (render all sprites then render all masked segments), there is no way the grid in figure 5.41 could be interleaved with the sprites. To address this, there is a little "hack" in `R_DrawSprite` during the linear scan for occluding `drawseg_t`s. It detects which segments were "masked" and renders them via `R_RenderMaskedSegRange`.

5.12.10 Drawing Masked Player

The last piece to render is the easiest of all. The sprite representing the Doomguy, a.k.a "the player sprite" (`psprite`), is drawn on top of everything. There is no clipping in effect here, only the need to account for the lightmap induced by the sector the player is currently standing in and making the hand bob left and right when moving/running. Like other masked elements this sprite is rendered vertically as columns of pixels.

You may have noticed in `R_DrawMasked` that the player hand is drawn only if the viewing angle offset is equal to zero (`viewangleoffset`). This is an artifact of a feature that was disabled in later versions. Until v1.2, DOOM supported a "three screen mode" where three computers with three monitors could be networked to render a wide field of view.



It is likely John Carmack was inspired to add this feature when he visited Alaska Airline training center, where he was able to see a multi-million dollar wide-screen flight simulator²¹. The feature was cut in later versions.

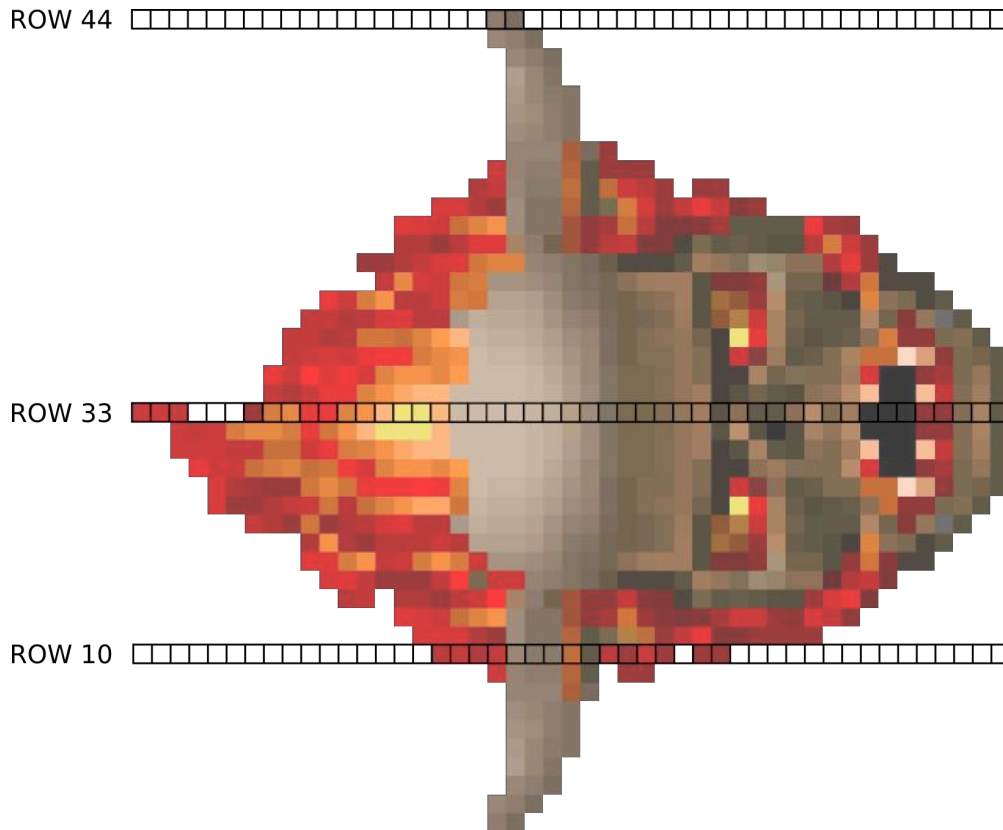
Trivia : Only the command-line code to enable/disable multiple screens was removed, the feature itself remained. Chocolate DOOM re-enabled it and even created a special mode allowing the "three monitor view" to be experienced on a single machine. This is how the above screenshot was created.

5.12.11 Picture format

Sprites are written one vertical column at a time but are stored in a way that cuddles the i486 cachelines during read operations. Each sprite is stored in its own lump and consists of a collection of lines describing the vertical columns in the sprite (in essence, sprites are stored rotated 90 degrees counterclockwise).

Each column (also called "post" in the code) is a set of "spans" with one byte giving the vertical offset where the span starts, then one byte giving the size of the payload and finally the texel payload.

²¹ Source: "<http://leeland.stores.yahoo.net/earlydoomstuff.html>"

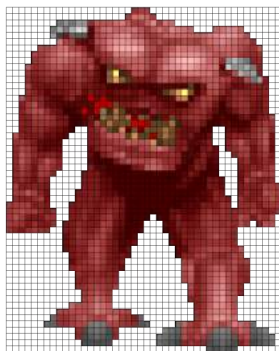
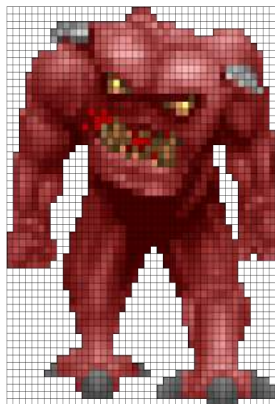


The Lost Soul sprite's row 44 is 4 bytes (one span with: 0x13 vertical offset, 0x02 data length, 0x54 and 0x59 two texels). Row 33 has two spans, accounting for 48 bytes and is a rare case where the encoding scheme is less favorable than "uncompressed" which would have been 47 bytes. Row 10 (also two spans) accounts for 19 bytes which is much less than the uncompressed length of 47 bytes. This entire Lost Soul sprite is 44x47 and fits in 1360 bytes for a surface of 2068 pixels, resulting in a rough 50% compression ratio.

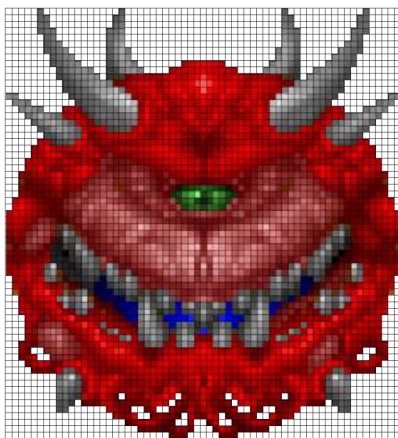
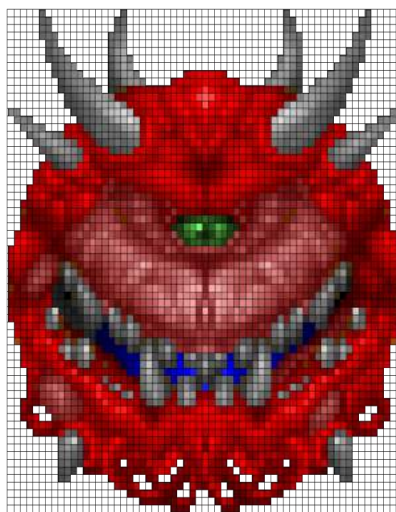
5.12.12 Sprite aspect ratio

The framebuffer was distorted when transferred from the VGA hardware to the CRT screen. The different aspect ratios make pixels taller than they are wide, stretching images vertically.

This distortion had not been an issue when working with Deluxe Paint since the tool could be set up in 320x200 (so artists did not have square pixels) but it was something to factor in when id switched to using scanned images from the NextDimension.

**LUMP CONTENT****AS RENDERED***Figure 5.43:* Demon storage vs rendered

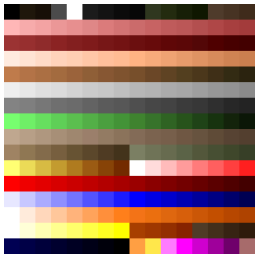
The Cacodemon was purposely drawn and rendered elliptically but effectively stored as a rounded shape. In the early days of tool authoring, programmers of asset extractors did not account for the CRT distortion, resulting in bulky monster visuals²².

**LUMP CONTENT****AS RENDERED***Figure 5.44:* Cacodemon storage vs rendered

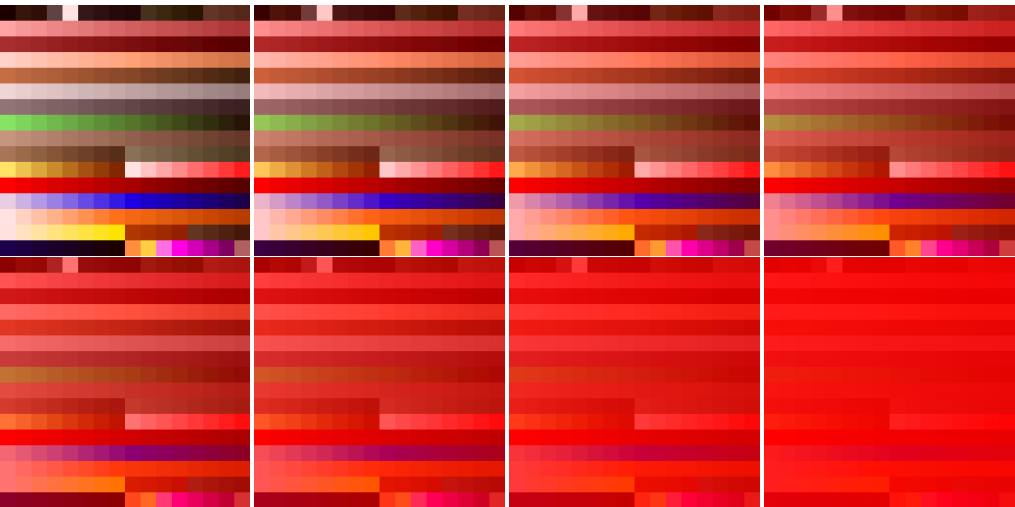
²²Even toy manufacturers made the mistake. Reaper Miniatures' Cacodemon figurine is round instead of elliptic.

5.13 Palette Effects

Despite its many weaknesses, the VGA hardware had one really cool feature, its palette. Only 768 bytes were required to make the entire screen fade toward a color. DOOM used fading for three effects – damage, item pickup, and the radiation suit. To implement these effects, palettes were precalculated and stored in the PLAYPAL lump. There are fourteen palettes in total. Palette #0 was the default and was used during most of the game.



Eight palettes (from #1 to #8) were used to emphasize damage and to let the player know how badly they were being injured. The effect switches the display to one of the seven palettes based on the amount of damage taken (the higher the damage the higher the starting palette). A fully red screen was usually very bad news. If no more damage is taken, the palette fades back to normal (Palette #1) at the rate of one palette unit every 1/2 second.



Due to an off-by-one bug, palette #1 is never used. Take a good look at the code selecting the damage palette in `ST_doPaletteStuff`; it can only generate values within the range [2,8].

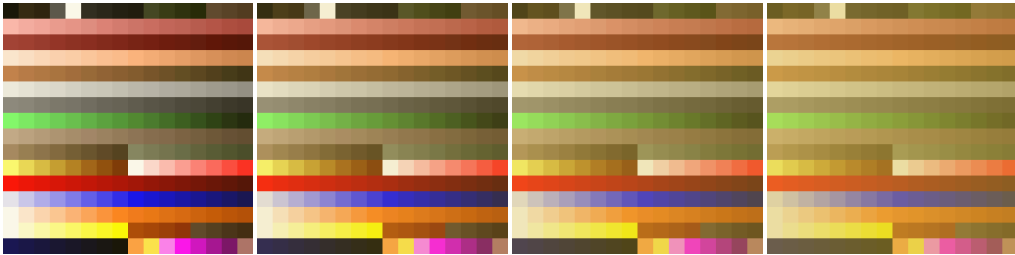
```
#define STARTREDPALS 1
#define NUMREDPALS 8
```

```

void ST_doPaletteStuff(void) {
    int palette;
    int cnt = plyr->damagecount;
    ...
    if (cnt) {
        palette = (cnt+7)>>3;
        if (palette >= NUMREDPALS) palette = NUMREDPALS-1;
        palette += STARTREDPALS;
    } else {
        ...
    }
    byte *pal = W_CacheLumpNum (lu_palette, PU_CACHE)+palette
        *768;
    I_SetPalette (pal);
    ...
}

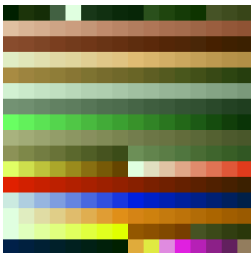
```

Palettes #9 to #12 are used briefly when an item is picked up.



Because of the same off-by-one miscalculation, palette #9 is never used. The palette selector can only generate values within range [10,12] whereas a [9,12] range was needed.

The last palette (#13) is not used for fading effects but as a temporary swap when the player is wearing the radiation suit.



5.14 Input

The input system is abstracted based on the notion of events – generated when devices are sampled in subsystems – and responders that consume these events in the core. Key strokes, joystick status and mouse inputs are stored in the `event_t` structure.

```
typedef enum {
    ev_keydown,
    ev_keyup,
    ev_mouse,
    ev_joystick
} evtype_t;

typedef struct {
    evtype_t    type;
    int         data1;    // keys/mouse/joystick buttons
    int         data2;    // mouse/joystick x move
    int         data3;    // mouse/joystick y move
} event_t;
```

The core system notifies the input system when a new frame starts or when a game tic starts, giving it an opportunity to sample devices for inputs, wrap them into an `event_t` and use a callback function to post it to the core event buffer.

Function	Usage
<code>I_StartFrame</code>	Called by the core before a visual frame is rendered.
<code>I_StartTick</code>	Called by the core when a game tic is rendered.
<code>D_PostEvent</code>	Called by the input system to send an event to the core.

Figure 5.45: DOOM's input system interface

Events are stored in the core via a circular buffer audaciously named `events`.

```
#define MAXEVENTS    64

event_t    events[MAXEVENTS];
int         eventhead, eventtail; // Circular buffer

void D_PostEvent (event_t *ev) {
    events[eventhead] = *ev;
    eventhead = (++eventhead) & (MAXEVENTS - 1);
}
```

On each game tic, the event queue is emptied. Events are sent one-by-one down a chain of responders²³. Each responder has the choice to ignore the event, in which case it is passed further down.

```
void D_ProcessEvents (void) {

    event_t      *ev;
    int          head = eventhead;
    int          tail = eventtail;

    for ( ; tail != head ; tail = (++tail)&(MAXEVENTS-1) ) {
        ev = &events[tail];
        if (M_Responder (ev))
            continue;      // menu ate the event
        G_Responder (ev);
    }
    eventhead = eventtail;
}

boolean G_Responder (event_t *ev) {
    ...

    if (HU_Responder (ev))
        return true;      // chat ate the event

    if (ST_Responder (ev))
        return true;      // status window ate it

    if (AM_Responder (ev))
        return true;      // automap ate it

    if (F_Responder (ev))
        return true;      // finale ate the event

    // 3D renderer consumes event here
    ...
}
```

If the event is consumed ("eaten" in the code) then it is not passed to subsequent responders. Notice that the 3D renderer is the last responder in the chain.

²³It is likely the "responder" architecture was influenced by NeXT's `NSResponder` elements found in the `AppKit` framework. It proved to be a robust design since it is still in use to this day.

Trivia : The file `i_cyber.c` has nothing to do with the enemy called the "Cyberdemon". It is a driver especially written to support a curious device manufactured by Logitech around 1992 called the "CyberMan". It was a hybrid input device providing six degrees of freedom. Think of it as a joystick upon which would be mounted a mouse. Support for its SWIFT API seems to have been added later since it doesn't generate events like the keyboard, mouse, and joystick but instead generates a tic command directly into the tic command stream.



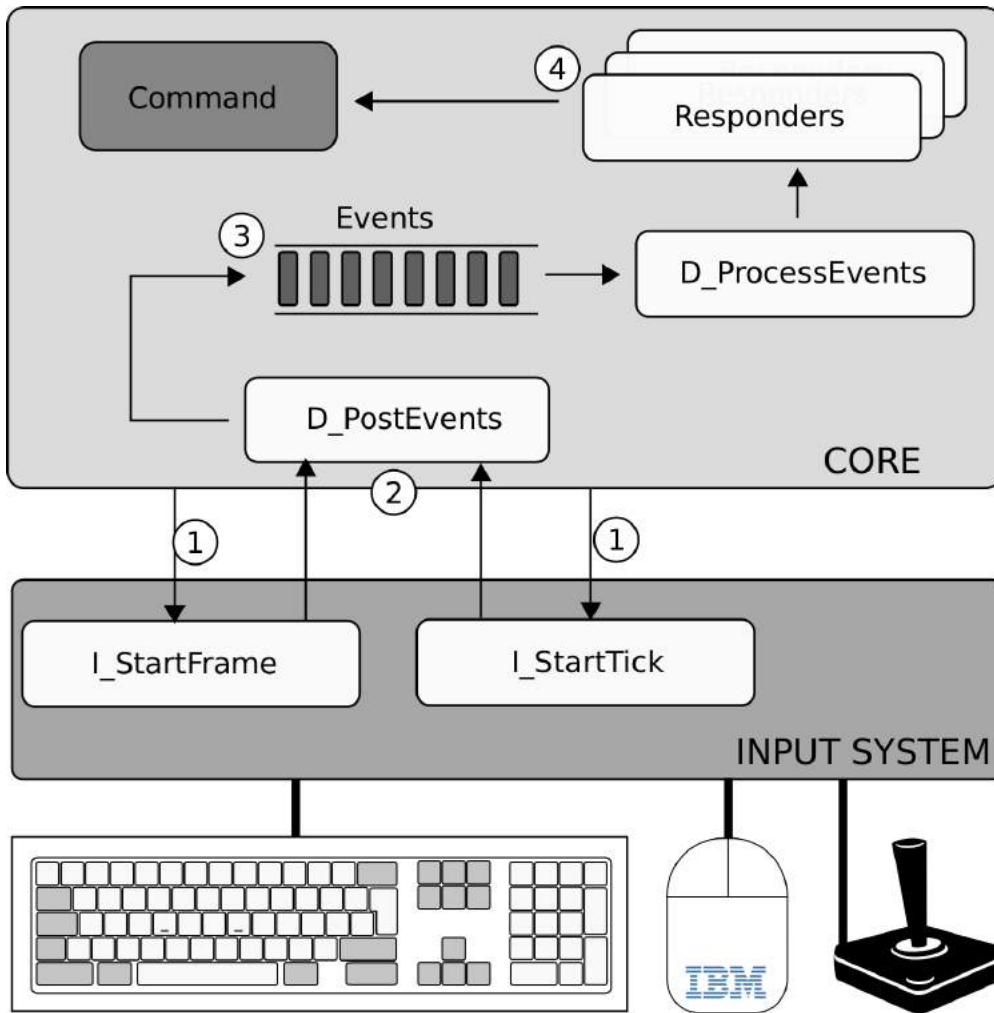
Most responders consume events in their raw (`event_t`) form but the 3D renderer normalizes them into a `ticcmd_t` containing not inputs but player actions. These "commands" as they are called have no timestamps since they are part of the game logic stream that runs at a fixed 35Hz frequency.

```
// The data sampled per tick (single player)
// and transmitted to other peers (multiplayer).
// Mainly movements/button commands per game tick,
// plus a checksum for internal state consistency.
typedef struct {
    char          forwardmove;    // *2048 for move
    char          sidemove;       // *2048 for move
    short         angleturn;      // <<16 for angle delta
    short         consistency;    // checks for net game
    unsigned char chatchar;
    unsigned char buttons;
} ticcmd_t;
```

The "commands" design pattern unlocks many features.

Commands are obviously consumed by the 3D engine but they can also be stored to disk when recording a demo. Later they can be re-injected into the engine to replay the exact same session. The beauty of this system is that players were able to exchange replays even if they had computers capable of different framerates.

This feature allowed DOOM to have demos like you could find in arcades. The DEM01, DEM02, and DEM03 lumps are streams of `ticcmd_ts` meant to be played at 35Hz. Since only commands are stored and the game is deterministic, demo files are very small, consuming only $8 * 35 = 280$ bytes/second.

**Figure 5.46**

Putting it all together, ① the core calls into the Input System once per frame and once per game tic to allow it to sample devices. ② Events are sent to the Core. ③ Received events are stored in a circular event buffer. ④ Events are dispatched to various responders. If the 3D renderer is active, the events are combined into a `ticcmd_t` which can be consumed, sent on the network, or stored on disk in the context of a demo recording.

During demo playback the input system is disabled; tics commands are read from disk and injected into the pipeline.

5.15 Audio System

Like every other I/O system in the engine, the audio is abstracted behind an interface. To fulfill DOOM's core expectations, such a system has to implement at least twenty functions covering sound effects, music, and also the timer. Here are a few.

Method	Usage
I_StartupSound	Initialize audio system, detect audio hardware
I_SetChannels	Set number of channels and sample rate
I_RegisterSong	Upload a music lump and get back an ID.
I_SetMusicVolume	Self explanatory.
I_PlaySong	Play Song
I_PauseSong	...hm, Pause Song
I_ResumeSong	Mysterious function with unknown effects.
I_StopSong	Maybe this table was not a good idea after all.
I_UnRegisterSong	Free music using ID obtained in I_RegisterSong.
I_GetSfxLumpNum	Upload audio sample from WAD and return an ID.
I_SetSfxVolume	If you read this, you are a real human being and a real hero.
I_StartSound	Start playing SFX sample.
I_StopSound	Stop playing SFX sample and free it.
I_SoundIsPlaying	Test if SFX is playing.
I_UpdateSoundParams	Set pitch, left/right position and volume.
I_StartupTimer	On DOS, triggers audio system to hook into the Intel 8259 PIC. No-op on other platforms.
I_ShutdownTimer	On DOS, remove the hook. No effect on other platforms.

Figure 5.47: DOOM's audio system interface

On NeXTSTEP this system was never a problem since only the timer function had to be implemented. On the PC side however, the game engine had to have sound effects and music. One difficulty was the fragmentation of the sound card market which had increased exponentially since Wolfenstein 3D. The previous title required tremendous effort just to support four types of sound card. Two years later there were more than fifteen available, all with different bugs, quirks and technologies.

To make things worse, the departure of Jason Blochowiak had left id Software with both a shortage of expertise and enthusiasm toward audio. They solved the problem by throwing money at it and licensed the DMX library. For the price of its license, the library authored by Paul J. Radek offered an all-in-one audio solution. With support for all major sound cards, a convenient means to detect them, support for many sound and music formats, and an easy way to integrate with any game engine, DMX was a perfect fit which undoubtedly saved id several months of development.

The abstraction layer provided by DMX was a colossal task. The comment before the function in charge of detecting the hardware leaves no ambiguity as to how much of a burden it was to tackle this problem.

```
//
// Why PC's Suck, Reason #8712
//

void I_sndArbitrateCards(void) {
    ...
}
```

With DMX backing the sound system, ten audio chipsets were supported.

```
typedef enum {
    snd_none,      //
    snd_PC,        // PC Speaker
    snd_Adlib,     // Adlib
    snd_SB,        // Sound Blaster
    snd_PAS,       // Media Vision (Pro AudioSpectrum)
    snd_GUS,       // Gravis UltraSound
    snd_MPU,       // Roland MPU-401
    snd_MPU2,
    snd_MPU3,
    snd_AWE,       // Sound Blaster AWE 32
    snd_ENS,       // ENSONIQ
    snd_CODEC,     // Compaq Business Audio
    NUM_SCARDS
} cardenum_t;
```

Running on an operating system supporting neither threads nor processes, there was seemingly no way to generate both the video and audio output simultaneously. Attentive readers will have noticed the hardware section mentioned two chipsets: the i8259 (Programmable Interrupt Controller: PIC) and i8254 (Programmable Interval Timer: PIT). Together they can be set up to interrupt the engine's execution and call into DMX's routines²⁴.

Upon initialization, DMX installs itself as an interrupt handler to be called by the PIC and PIT at 140Hz. Upon awakening, DMX's interrupt handler takes care of feeding the audio device with music and sound effect data provided by the engine.

Since it was tied to a timer, DMX was also in charge of the game engine's heartbeat via a variable called `ticcount`. Everything in DOOM uses that variable to pace itself.

²⁴The way the PIC and PIT interact is extensively detailed in "Game Engine Black Book: Wolfenstein 3D".

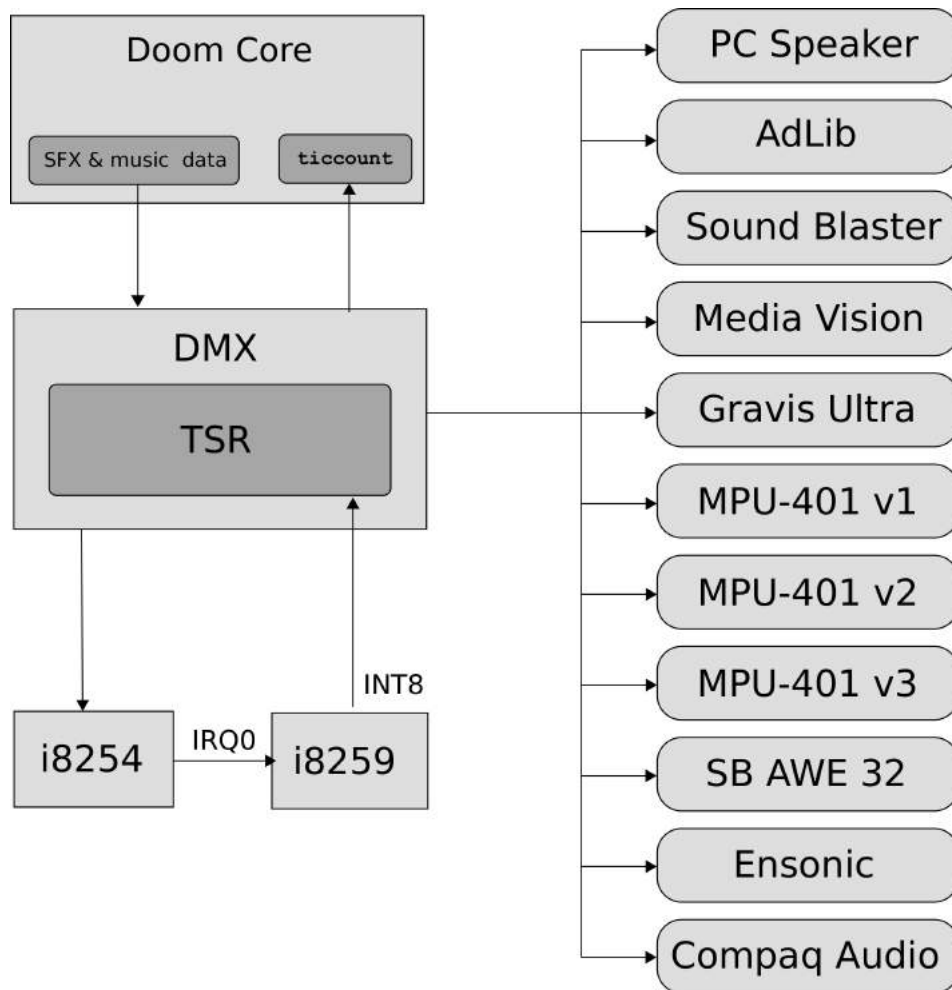


Figure 5.48: DMX architecture

With such an architecture there are two systems executing pseudo-concurrently but the audio has more constraints than the video. When an interrupt is triggered, DMX has only a few milliseconds to refill the sound card's buffers and go back to sleep. If DMX is too slow, it can delay video rendering or mask other interrupts.

This explains why allocation of audio assets gets special treatment in the zone allocator. The audio system has no time to recover from a memory miss (nor could it, since it has no access to the WAD or memory allocator). Audio data must be ready immediately whenever it is needed.

5.15.1 Audio Data: Formats and Lumps

The WAD archive contains hundreds of lumps of five types to feed to DMX. DP* lumps are for PC speaker sound effects, DS* lumps are for PCM sound effects, D_* lumps are for music tracks. There is also a GENMIDI lump, and one DMXGUS lump detailed later.

PC speaker audio was meant for PCs without a sound card. The PC speaker was capable of square waves and meant to emit boot-up diagnostic noises. DOOM found ways to make it generate something less irritating by changing the square wave frequency every 1/140th of a second²⁵. The data rate is one byte per 1/140th of a second, describing the square-wave frequency to set.

The digitized audio samples used in DOOM are 8-bit, 11025 Hz, mono PCM streams. PCM is a simple audio format that requires little explanation. Each byte in the stream is a moment in time that describes the amplitude of a waveform and indicates where the speaker cone should be positioned. The sound card simply translates the byte into a voltage to the magnet driving the cone position, which happens at a rate of 11025 times per second.

Music data is stored in MUS, a format similar to standard MIDI but slightly more compact. The format allows eight channels for instruments and a ninth for the drums. The MUS format describes a series of precisely timed events for each channel which specify which instrument should play a note and at what pitch. Note that MUS only describes what to do and when, not how to do it. Each channel refers to the note of an instrument; the instrument itself is described elsewhere in a data structure known as the "instrument bank".

The instrument bank is stored in the GENMIDI lump. It is aimed at SoundBlaster-compatible sound cards based on the OPL2 chip which synthesize music using Frequency Modulation. The lump describes how to play the note of each instrument in the MIDI instrument set. There are 175 entries in GENMIDI, one for each of the 128 standard General MIDI instruments and 47 percussion effects. Each entry describes how to set up a channel in order to emulate an instrument. A channel is made of two cells, where one cell is used as carrier and the other as modulator. For each cell, attack, decay, sustain, release, harmonic type, and waveforms can be selected by the musician. Instrument banks were 90s musicians' "secret sauce" and DOOM's General MIDI emulated FM patch set were known for being particularly good²⁶.

The DMXGUS lump played the same role as GENMIDI but for the Gravis Ultrasound card. It enabled the GUS to play MUS music notes, not with an FM synthesizer but with the PCM samples provided with GUS drivers. It is a simple lump mapping MIDI instruments to GUS instruments with special rules for RAM allocation depending on the amount of RAM installed on the board (256KiB, 512KiB, 768KiB, and 1024KiB).

²⁵The PC Speaker is detailed in "Game Engine Black Book: Wolfenstein 3D".

²⁶Source: "The dark and forgotten art of GENMIDI" in Freedoom's documentation.

Dealing with so much complexity proved difficult for DMX. Some of the craziness Paul Radek had to deal with surfaced in one of his post on usenet²⁷.

“ All of the sound effects are now mixed in software, rather than on the GUS hardware. Why, you ask? Because of several reasons. First, is that the GF1 chip has a minimal ramp time that is much to long for very sharp effects. Second, because loading of the MUSIC patches uses all of the GUS memory, I had to DMA all eight sound effects to the card when played. This intern exposed a bug in the GF1 chip that Gravis did not find until my code started to beat on it. The bug would cause the bus to freeze and any program with it. The workaround is to keep DMA activities to a minimum by mixing in software and transferring only 1 channel to the GUS. But since the GF1 can't support auto-initialize DMA, and because the only way to play interleaved data on the card is to set two voices pointing into a single patch and setting the frequency so the every other sample is skipped, you don't get the benefit of sample smoothing from the GF1.

Sorry, but that's the way it has to be :(

— Paul Radek, Digital Expressions, Inc.

”

The library evolved during the development of DOOM. Sometimes API changes introduced bugs. Gravis UltraSound support was broken with v1.666. Support for the Audio Spectrum was also broken so users of the card had to fall back on (poor) SoundBlaster emulation²⁸.

The problem was partly due to poor API practices on DMX's side and partly because of hasty adjustments on id Software's part. Most issues were ultimately fixed except for one major bug which made it to gamers. The engine was originally supposed to emit sounds at random pitches to avoid monotony. To this effect, DMX function SFX_PlayPatch was used.

```
int SFX_PlayPatch(  
    void    *patch,    // Patch to play  
    int     x,         // Left-Right Positioning  
    int     pitch,     // 0..128..255  -10ct..C..+10ct  
    int     volume,    // Volume Level 1..127  
    int     priority    // Priority, 0=Lowest  
);
```

²⁷Forum thread: "Gravis Ultrasound - Hardware Mixing Game List".

²⁸Source: John Romero post on alt.games.doom.

In early versions it worked as intended but then the DMX API was modified in an incompatible way. It may not be immediately apparent but the parameters to `SFX_PlayPatch` are swapped.

```
int SFX_PlayPatch(
    void    *patch,    // Patch to play
    int     pitch,     // 0..128..255  -10ct..C..+10ct
    int     x,         // Left-Right Positioning
    int     volume,    // Volume Level 1..127
    int     flags,     // Flags
    int     priority   // Priority, 0=Lowest
);
```

Invocation sites were never adjusted and the game shipped without the random pitch feature, instead randomly balancing sound between the left and right channel.

```
int I_StartSound (void *data, int vol, int sep, int pitch,
    int priority) {
    ...
    return SFX_PlayPatch(data, sep, pitch, vol, 0, 100);
}
```

In retrospect John Carmack regretted using DMX because it led to issues when open sourcing the game engine (it is unknown if Paul Radek was unwilling to open source DMX or if id software was unwilling to negotiate with him).

“ Our biggest mistake during DOOM development was the contracting of an outside party to do dos sound drivers. Because we had this black box functionality coming, I didn't simulate it under NS. BAAAAAD mistake. All future work will be entirely developed under NS, with only DMA buffer flipping being the hardware layer. We will probably also run midi under NS for music (which will be dynamically tuned to the game situation in Quake).

— John Carmack

”

One can still find archived Usenet posts from `alt.games.doom` where John Romero's comments hint that the relationship between Radek and id Software was suffering as the game neared its release date²⁹. Since the tone was less than cordial, it is left as an exercise to the reader if they want to dig these posts out. I do not recommend it.

²⁹Source: John Romero post on `alt.games.doom`.

5.16 Sound Propagation

How enemies react to sounds makes a tremendous impact on how smart the A.I. is perceived as being. id Software made sure sound propagation was realistic. When a shot is fired in a sector, a flood fill algorithm propagates the noise.

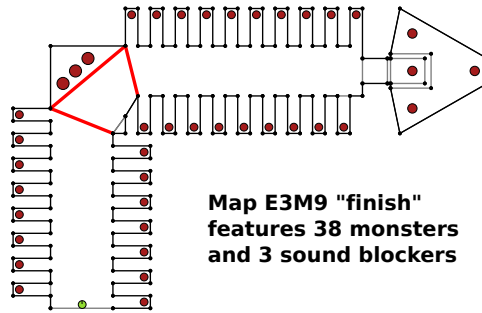
```
void P_RecursiveSound (sector_t *sec, int soundblocks)
{
    int      i;
    line_t    *check;
    sector_t  *other;

    // wake up all monsters in this sector
    if (sec->validcount == validcount &&
        sec->soundtraversed <= soundblocks+1)
        return;    // already flooded

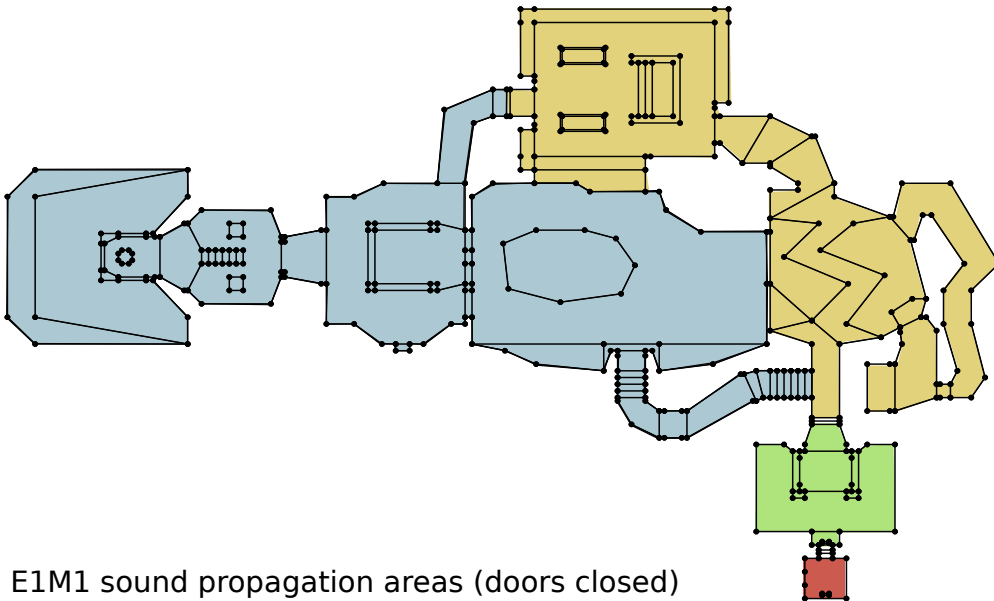
    sec->validcount = validcount;
    sec->soundtraversed = soundblocks+1;
    sec->soundtarget = soundtarget;

    for (i=0 ;i<sec->linecount ; i++) {
        check = sec->lines[i];
        if (! (check->flags & ML_TWOSIDED) )    // Portal?
            continue;    // No, this is a wall, skipping.
        P_LineOpening (check);    // fixed_t openrange = 0 if door
        closed.
        if (openrange <= 0)
            continue;    // closed door
        if ( sides[ check->sidenum[0] ].sector == sec)
            other = sides[ check->sidenum[1] ] .sector;
        else
            other = sides[ check->sidenum[0] ].sector;
        if (check->flags & ML_SOUNDBLOCK)
        {
            if (!soundblocks)    // Blocking noise?
                P_RecursiveSound (other, 1);
        }
        else    // Flooding to next sector !
            P_RecursiveSound (other, soundblocks);
    }
}
```

The sector/portal format is leveraged, starting from the player's sector and flood-filling into adjacent sectors via portals (two-sided lines). Sound is stopped when either a door is encountered or when two "sound blocker" lines have been crossed. Level E3M9 features a section with 38 monsters where the A.I. cost is lowered by three (red) blocker lines, ensuring some monsters remain dormant.



Notice how function `P_RecursiveSound` mentions "waking up all monsters" but never iterates over the list of things in the level. This is a speed-up trick aimed at avoiding an expensive search to find all monsters to be awakened in each sector. Monsters always look up for the current sector's `soundtarget` to pick a target. By simply assigning a value to `sec->soundtarget`, all monsters in `sec` automatically acquire the same target.



5.16.0.1 Ambushing

Map designers wanted to have sneaky monsters who could hide and wait to jump out at players. This is achieved via the `MF_AMBUSH` flag assigned to monsters. It doesn't make monsters deaf, rather it makes them not seek the player until they make visual contact.

5.16.0.2 Super Ambushing

Sound propagation was used in an inventive in level E1M9 for its super ambush. The designers wanted the player to be swarmed with monsters teleporting, seemingly from a hellish dimension, as soon as the player walked across the center of a pentagram.

Without a scripting language available that was next to impossible to implement. As we'll see in the A.I. section, monsters are state machines only able to do three things: stay dormant, pursue a target, and when they bump into walls and things, change direction.

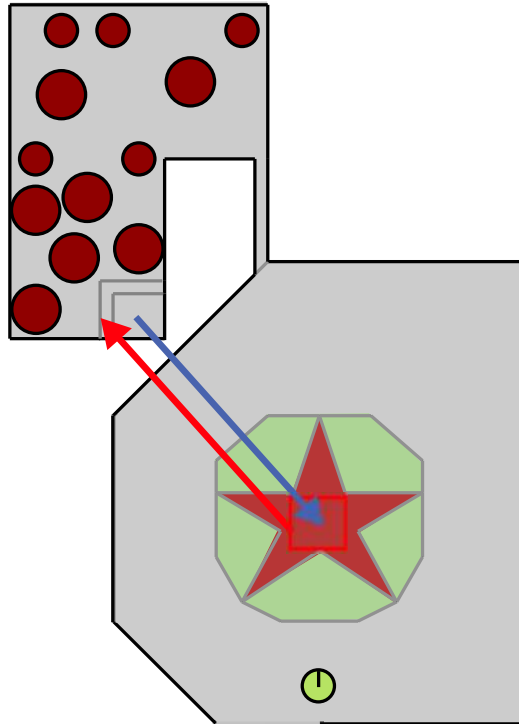
To achieve the effect, they created an inaccessible "monster pool" room next to where they wanted the super ambush and filled it with monsters (red circles in the diagram). In the south-east corner of the hidden room, they placed a teleporter, protected by walls.

Then they created a very tiny pipe to allow sound to flow between rooms, so that monsters from the "pool" room would wake up and try to reach the player. Without another way to get there (the pipe being at ceiling level and the monsters being too big to fit through it), monsters will wander in circles with a tendency to move towards the player's location (south-east).

The last piece of the super ambush trap was to have four tripwires around the center of the "pentagram" which lowered the walls around the teleporter. A tempting bait was placed on it (health and ammunition) to make sure the player would go there.

As soon as the player crosses the trigger in the "star" room, the walls around the teleporter lower, monsters move towards it and swarm the player.

Trivia : Comments in the code suggest that monsters screaming awaken other monsters but this is not the case. It is unknown why this feature was cut. Maybe it was too buggy or too expensive.



John Romero himself described how they called these sound conduits, "pipes".

“ We used sound zones in Wolfenstein 3D as another way to alert enemies to your presence. In DOOM, we did the same thing but used sectors as the conduits of audio travel. This was a really important part of making the game scary, as sound could leak all over the place and alert demons. You might see lots of little sector pipes that connect sectors together just to alert monsters-sectors that you'd never see because we put them way up high in the corner of a room. So, we paid a lot of attention to the sound flooding.

— John Romero, p29 in Scarydarkfast

”

The audio pipe for the E1M9 super ambush is so well-hidden in the ceiling that it is very hard to notice with the vanilla engine. The room is dark and in the distance, the tiny black rectangle blends in the obscurity. Opening the map with an editor such as SLADE that allows looking up and down reveals the mechanism.



Figure 5.49

In figure 5.49 the audio conduit is visible in the upper right corner.

5.17 Collision Detection

Collision detection is a significant part of the engine's activity. Each moving object (player, monster or projectile) must check for collisions before it moves. Line of sight also depends on an efficient collision system. Enemies which inflict direct hit damage also need to check if they have a clear line of fire.

Collisions could have been detected via the BSP. However it was only after Bruce Naylor visited id Software that John Carmack became aware this was possible. By then, DOOM had already shipped with its collision detection data structure called the blockmap.

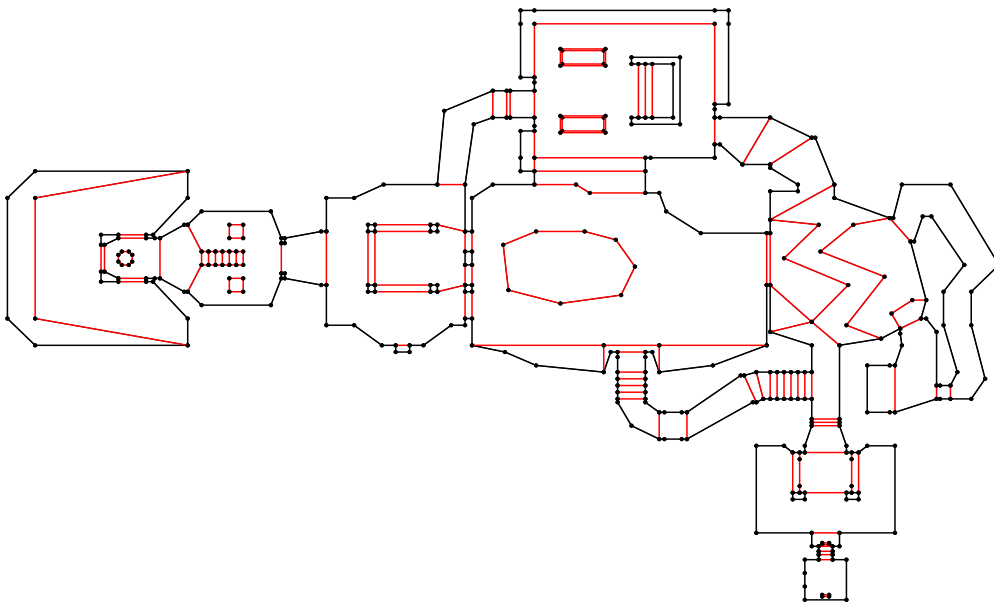


Figure 5.50: E1M1 sectors and lines

There is one blockmap per map which was generated via `doombsp` preprocessing on NeXTstations. Saved in a lump audaciously named `BLOCKMAP`, it is used at runtime to lower the number of lines to test intersections with.

The work done by `doombsp` is simple: divide the map into 128×128 axis-aligned blocks. For each block a list is made of every line that passes through it. At the end of the process, an index is constructed based on blockmap coordinates (in 128×128 units) pointing to the list of lines. Notice that a line can be present in multiple blocks. In the case of map E1M1, the result is visible in figure 5.51.

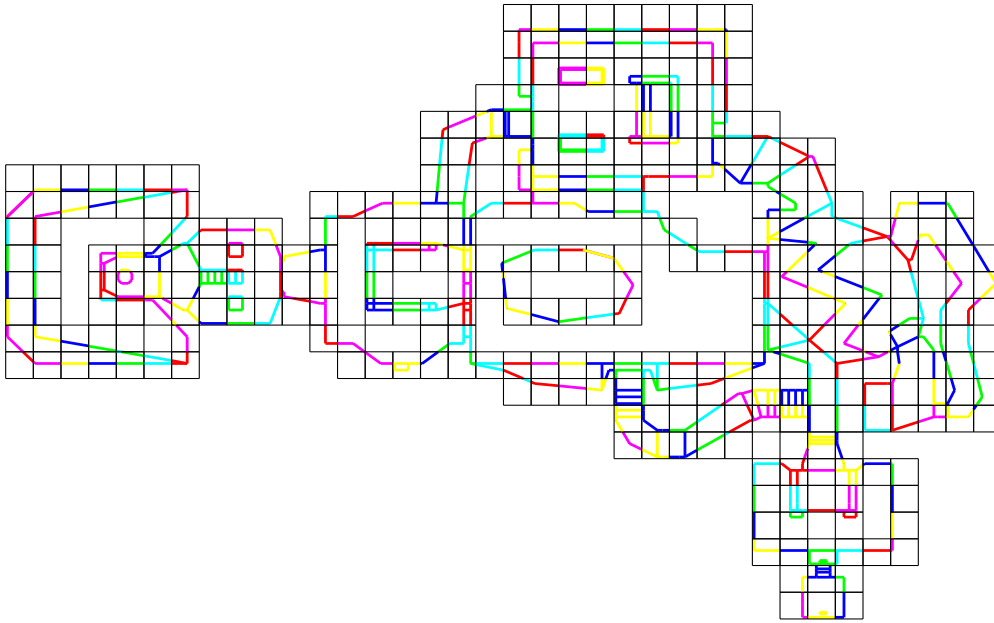


Figure 5.51: E1M1 lines indexed via blockmap. Note that empty blocks are not drawn

All map traversals are done with an abstract method `P_PathTraverse` which takes as arguments two coordinates making up a line to check collisions with and a function pointer to call when a hit is detected (a.k.a how to fake OOP with C).

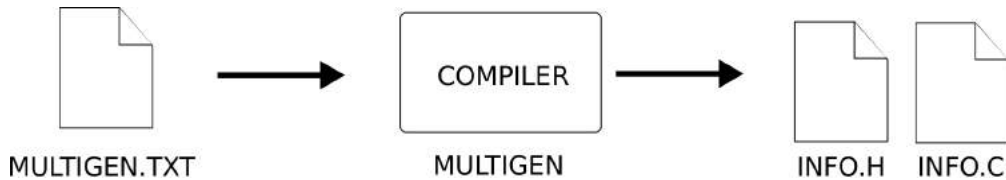
```
bool P_PathTraverse (fixed_t x1, fixed_t y1, // origin
                    fixed_t x2, fixed_t y2, // dest
                    int flags,                // targets
                    boolean (*trav) (intercept_t *));
```

Function `P_AimLineAttack` (used for punching and sawing) uses `P_PathTraverse` with flag = `PT_ADDLINES|PT_ADDTHINGS` so that only lines and things are considered during traversal. The block coordinates of any map coordinate are easy to obtain via a divide by 128 (optimized as `>> 7`). To detect collisions with things, their block coordinates are updated each time they change position.

```
fixed_t P_AimLineAttack (mobj_t *t1, angle_t angle, fixed_t
                        distance) {
    P_PathTraverse ( t1->x, t1->y, x2, y2,
                    PT_ADDLINES|PT_ADDTHINGS, PTR_AimTraverse );
}
```

5.18 Artificial Intelligence

As mentioned earlier, there is no scripting language in DOOM. The A.I. is based on a set of state machines for each enemy type baked into the engine binary. Designers did not have to learn C since they could entirely configure an opponent via the text file `multigen.txt`. This text file is parsed by a tool (cunningly named `multigen`) and compiled into C source code (`info.h` and `info.c`).



Let's dive into the hand-written state machine descriptor text file right away and take a look at the section of `multigen.txt` that controls the `imp` (known as `TROOP` internally).

```

; Imps
$ MT_TROOP

doomednum      3001
spawnhealth    60
speed          8
painchance     200
radius         20*FRACUNIT
height        56*FRACUNIT
flags          MF_SOLID | MF_SHOOTABLE | MF_COUNTKILL

spawnstate     S_TROO_STND
seestate      S_TROO_RUN1
meleestate     S_TROO_ATK1
missilestate   S_TROO_ATK1
deathstate     S_TROO_DIE1
xdeathstate    S_TROO_XDIE1
raisestate     S_TROO_RAISE1
painstate      S_TROO_PAIN

attacksound    0
activesound    sfx_bgact
deathsound     sfx_bgdth1
seesound       sfx_bgsit1
painsound      sfx_popain
  
```

S_TROO_STND	TROO	A	10	A_Look	S_TROO_STND2
S_TROO_STND2	TROO	B	10	A_Look	S_TROO_STND
S_TROO_RUN1	TROO	A	3	A_Chase	S_TROO_RUN2
S_TROO_RUN2	TROO	A	3	A_Chase	S_TROO_RUN3
S_TROO_RUN3	TROO	B	3	A_Chase	S_TROO_RUN4
S_TROO_RUN4	TROO	B	3	A_Chase	S_TROO_RUN5
S_TROO_RUN5	TROO	C	3	A_Chase	S_TROO_RUN6
S_TROO_RUN6	TROO	C	3	A_Chase	S_TROO_RUN7
S_TROO_RUN7	TROO	D	3	A_Chase	S_TROO_RUN8
S_TROO_RUN8	TROO	D	3	A_Chase	S_TROO_RUN1
S_TROO_ATK1	TROO	E	8	A_FaceTarget	S_TROO_ATK2
S_TROO_ATK2	TROO	F	8	A_FaceTarget	S_TROO_ATK3
S_TROO_ATK3	TROO	G	6	A_TroopAttack	S_TROO_RUN1
S_TROO_PAIN	TROO	H	2	NULL	S_TROO_PAIN2
S_TROO_PAIN2	TROO	H	2	A_Pain	S_TROO_RUN1
S_TROO_DIE1	TROO	I	8	NULL	S_TROO_DIE2
S_TROO_DIE2	TROO	J	8	A_Scream	S_TROO_DIE3
S_TROO_DIE3	TROO	K	6	NULL	S_TROO_DIE4
S_TROO_DIE4	TROO	L	6	A_FALL	S_TROO_DIE5
S_TROO_DIE5	TROO	M	-1	NULL	S_NULL
S_TROO_XDIE1	TROO	N	5	NULL	S_TROO_XDIE2
S_TROO_XDIE2	TROO	O	5	A_XScream	S_TROO_XDIE3
S_TROO_XDIE3	TROO	P	5	NULL	S_TROO_XDIE4
S_TROO_XDIE4	TROO	Q	5	A_FALL	S_TROO_XDIE5
S_TROO_XDIE5	TROO	R	5	NULL	S_TROO_XDIE6
S_TROO_XDIE6	TROO	S	5	NULL	S_TROO_XDIE7
S_TROO_XDIE7	TROO	T	5	NULL	S_TROO_XDIE8
S_TROO_XDIE8	TROO	U	-1	NULL	S_NULL
S_TROO_RAISE1	TROO	M	8	NULL	S_TROO_RAISE2
S_TROO_RAISE2	TROO	L	8	NULL	S_TROO_RAISE3
S_TROO_RAISE3	TROO	K	6	NULL	S_TROO_RAISE4
S_TROO_RAISE4	TROO	J	6	NULL	S_TROO_RAISE5
S_TROO_RAISE5	TROO	I	6	NULL	S_TROO_RUN1

There are four sections. Properties include the DoomED id, speed, height, radius, state names for state machine targets, sound strings, and finally the huge Action definition.

Property lists and sound names are self-explanatory, and there is no need to spend too much time on them. What is more difficult to understand is how the state machine is defined.

A thing's state machine is partly statically defined inside the engine (when a monster is attacked it goes directly to `seestate`; when it receives lethal damage, it goes directly to `diestate`) and partly defined in `multigen.txt`. Each line in the state definition follows a syntax:

- 1. State name.
- 2. Frame family.
- 3. Frame ID (sprite to render).
- 4. Duration in tics (engine runs 35 tics/second).
- 5. Function to call when in this state.
- 6. Next state.

Let's take the example of an imp that has just spawned in a level and therefore is in state `spawnstate`, which is `S_TROO_STND`. Upon simulating each game tic, the engine looks at what to do in this state. In this case, the imp will cycle between the states `S_TROO_STND` and `S_TROO_STND2`. In these sub-states, `A_Look` is called each tic trying to locate the player. If the player is found, the engine places the imp into the `seestate` (a.k.a `S_TROO_RUN1`)

Suppose this imp was unlucky, the player was fast and managed to hit it with a shotgun at point blank. In this case the engine places the imp into the `deathstate` (`S_TROO_DIE1`).

<code>S_TROO_DIE1</code>	<code>TROO</code>	<code>I</code>	<code>8</code>	<code>NULL</code>	<code>S_TROO_DIE2</code>
<code>S_TROO_DIE2</code>	<code>TROO</code>	<code>J</code>	<code>8</code>	<code>A_Scream</code>	<code>S_TROO_DIE3</code>
<code>S_TROO_DIE3</code>	<code>TROO</code>	<code>K</code>	<code>6</code>	<code>NULL</code>	<code>S_TROO_DIE4</code>
<code>S_TROO_DIE4</code>	<code>TROO</code>	<code>L</code>	<code>6</code>	<code>A_FALL</code>	<code>S_TROO_DIE5</code>
<code>S_TROO_DIE5</code>	<code>TROO</code>	<code>M</code>	<code>-1</code>	<code>NULL</code>	<code>S_NULL</code>

Notice how values `I`, `J`, `K`, `L`, and `M` translate to sprite names.



Let's follow the chain of state from here, where an imp dies in five steps:

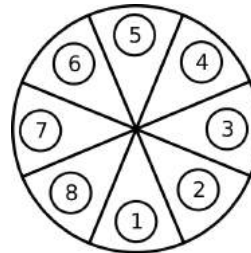
1. Show first death frame (I) for 8/35ths of a second.
2. Show second death frame (J) for 8/35ths of a second. Scream using `deathsound`.
3. Show third death frame (K) for 6/35ths of a second.
4. Show fourth death frame (L) for 6/35ths of a second. Mark itself as non-obstacle (`A_FALL`).
5. Show fifth death frame (M) forever (-1).

The total dying sequence lasts $8 + 8 + 6 + 6 = 24/35 = 0.68$ seconds. Note that this imp could have been even less lucky and been hit by a rocket. Enough damage would have caused it to gib, moved it to the `xdeathstate` (`S_TROO_XDIE1`) state and made it die in 1 second.

<code>S_TROO_XDIE1</code>	<code>TROO</code>	<code>N</code>	<code>5</code>	<code>NULL</code>	<code>S_TROO_XDIE2</code>
<code>S_TROO_XDIE2</code>	<code>TROO</code>	<code>O</code>	<code>5</code>	<code>A_XScream</code>	<code>S_TROO_XDIE3</code>
<code>S_TROO_XDIE3</code>	<code>TROO</code>	<code>P</code>	<code>5</code>	<code>NULL</code>	<code>S_TROO_XDIE4</code>
<code>S_TROO_XDIE4</code>	<code>TROO</code>	<code>Q</code>	<code>5</code>	<code>A_FALL</code>	<code>S_TROO_XDIE5</code>
<code>S_TROO_XDIE5</code>	<code>TROO</code>	<code>R</code>	<code>5</code>	<code>NULL</code>	<code>S_TROO_XDIE6</code>
<code>S_TROO_XDIE6</code>	<code>TROO</code>	<code>S</code>	<code>5</code>	<code>NULL</code>	<code>S_TROO_XDIE7</code>
<code>S_TROO_XDIE7</code>	<code>TROO</code>	<code>T</code>	<code>5</code>	<code>NULL</code>	<code>S_TROO_XDIE8</code>
<code>S_TROO_XDIE8</code>	<code>TROO</code>	<code>U</code>	<code>-1</code>	<code>NULL</code>	<code>S_NULL</code>



The engine uses a convention to find which sprite to use when rendering them. Because an enemy will not always be facing the player, it uses quantization where all orientations relative to the player's position fall into eight ranges (see diagram where 1 is facing the player, 5 facing away, and so on).



When in the `S_TROO_XDIE1` state, according to `multigen.txt`, the engine must use the sprite family `TROO` and frame `N`. Based on the orientation (lets say the imp has its back to the player), the engine should use `TROON5`. However, there is no such sprite in `DOOM.WAD` (exploding enemies always face the player) so the engine falls back to `TROON0` (0 being the "always facing" sprite).

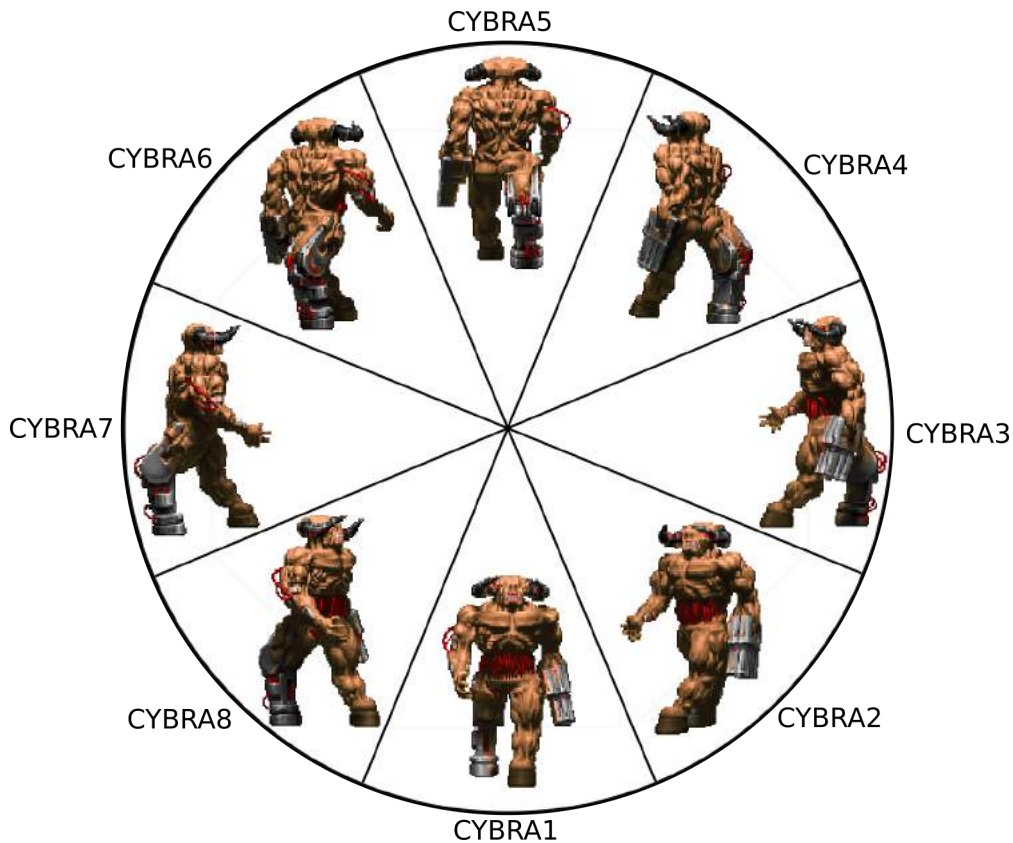
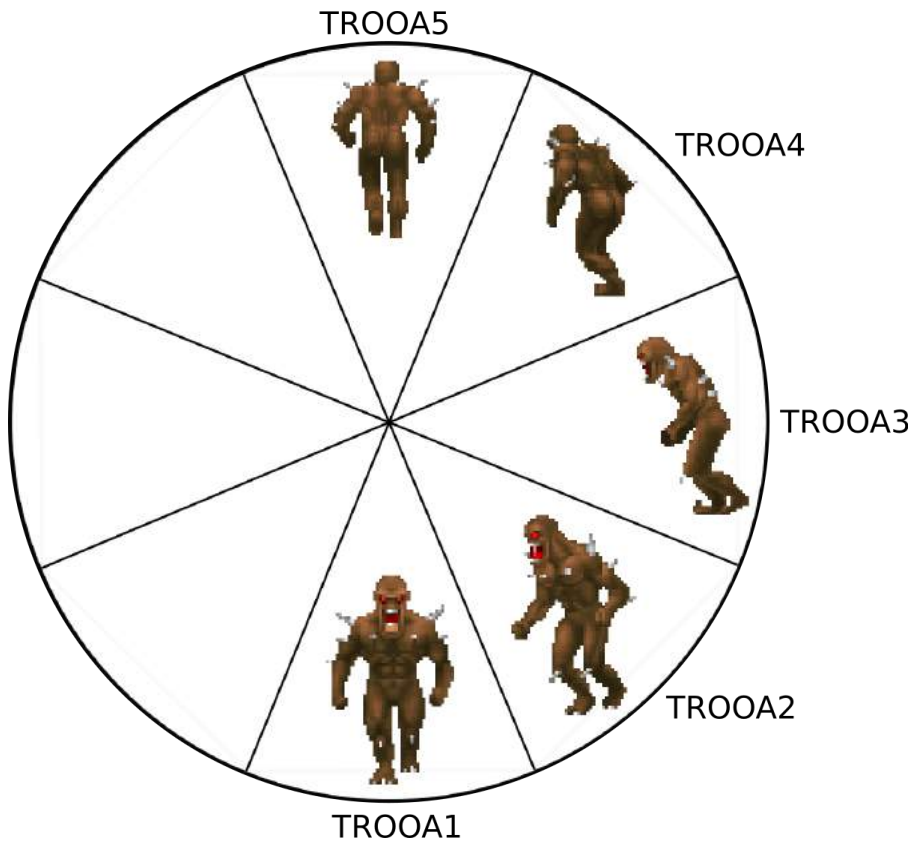


Figure 5.52: Cyberdemon poses in one of its two "walk" positions.

Taking a look at one of the frames for the Cyberdemon in figure 5.52 gives a good idea of the colossal work required from artists. Twelve monsters times eight states times an average of five frames per animation would have required close to 480 drawings (for the monsters only). The power of the NeXTdimension made a tremendous difference in this department.

The Cyberdemon however is an extreme case since it is not symmetrical. For the imp, storage is optimized to take advantage of its symmetry. If the engine needs TR00A6 but doesn't find it in the WAD, it uses its opposite (TR00A4) and draws it mirrored.

Trivia : You may have noticed that in the list of states there is a non-obvious one named RAISE. This is used when the Arch-Vile resurrects dead monsters. The animation plays the death animation in reverse. Note that there is no reverse gib sequence, but the Arch-Vile still revives gibbed monsters using a reverse normal death animation.

**Figure 5.53**

Trivia : When an entity receives more than spawnhealth damage (negative its spawning state; in the case of an imp that would be -60), the engine triggers not deathstate but the xdeathstate state that means the entity exploded.

```
void P_KillMobj (mobj_t *source, mobj_t *target) {
    [...]
    if (target->health < -target->info->spawnhealth
        && target->info->xdeathstate) {
        P_SetMobjState (target, target->info->xdeathstate);
    } else {
        P_SetMobjState (target, target->info->deathstate);
    }
    [...]
}
```

multigen.txt is compiled to the humongous 5000 line info.c containing an array of state_ts holding the state machine, and an array of mobjinfo_ts containing the thing properties.

```
typedef struct {
    spritenum_t  sprite;
    long         frame;
    long         tics;
    void         (*action) ();
    statenum_t   nextstate;
    long         misc1, misc2;
} state_t;

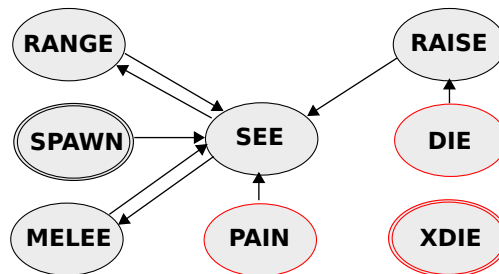
state_t  states[NUMSTATES] = {          // = NUMSTATES = 1109
// [...]
{SPR_TR00,0,10,A_Look,S_TR00_STND2,0,0} // S_TR00_STND
{SPR_TR00,1,10,A_Look,S_TR00_STND,0,0}, // S_TR00_STND2

{SPR_TR00,0,3,A_Chase,S_TR00_RUN2,0,0}, // S_TR00_RUN1
{SPR_TR00,0,3,A_Chase,S_TR00_RUN3,0,0}, // S_TR00_RUN2
{SPR_TR00,1,3,A_Chase,S_TR00_RUN4,0,0}, // S_TR00_RUN3
{SPR_TR00,1,3,A_Chase,S_TR00_RUN5,0,0}, // S_TR00_RUN4
{SPR_TR00,2,3,A_Chase,S_TR00_RUN6,0,0}, // S_TR00_RUN5
{SPR_TR00,2,3,A_Chase,S_TR00_RUN7,0,0}, // S_TR00_RUN6
{SPR_TR00,3,3,A_Chase,S_TR00_RUN8,0,0}, // S_TR00_RUN7
{SPR_TR00,3,3,A_Chase,S_TR00_RUN1,0,0}, // S_TR00_RUN8

{SPR_TR00,4,8,A_FaceTarget,S_TR00_ATK2,0,0}, // S_TR00_ATK1
{SPR_TR00,5,8,A_FaceTarget,S_TR00_ATK3,0,0}, // S_TR00_ATK2
{SPR_TR00,6,6,A_TroopAttack,S_TR00_RUN1,0,0}, // S_TR00_ATK3
// [...]
}
```

Not all automaton state changes can be inferred from the transition array. In the case of the imp, transitions are partly dictated by the logic in info.txt (black arrows) and partly dictated by the game engine (direct to red state).

In the state diagram, the engine-triggered transitions to pain, die, and xdie are not represented since these states can be reached directly from any other state.



5.18.1 Optimization

Monsters constantly request collision tests. Even the simple SPAWN state, for which a two frame animation cycles repeatedly (monsters do not have a standing script, they "walk on the spot"), calls the `A_Look` function to attempt to locate the player. Furthermore, once activated, monsters need path-finding and range attack tests. The calculations proved to be an unacceptable level of load on the CPU. Even using the blockmap structure to speed up collision detection it still meant hundreds of rays to cast, thirty five times per second which resulted in thousands of instructions.

To solve this problem, another data structure was introduced with `doombsp`. Each sector visibility is precomputed to allow impossible collisions to be rejected early. The visibility data is stored in a bit array³⁰. This dataset is packed into a matrix of size $num_sectors^2/8$ and stored in a `REJECT` lump. At runtime, the engine compares the player's current sector with a monster's sector to potentially bypass sight detections for this monster entirely.

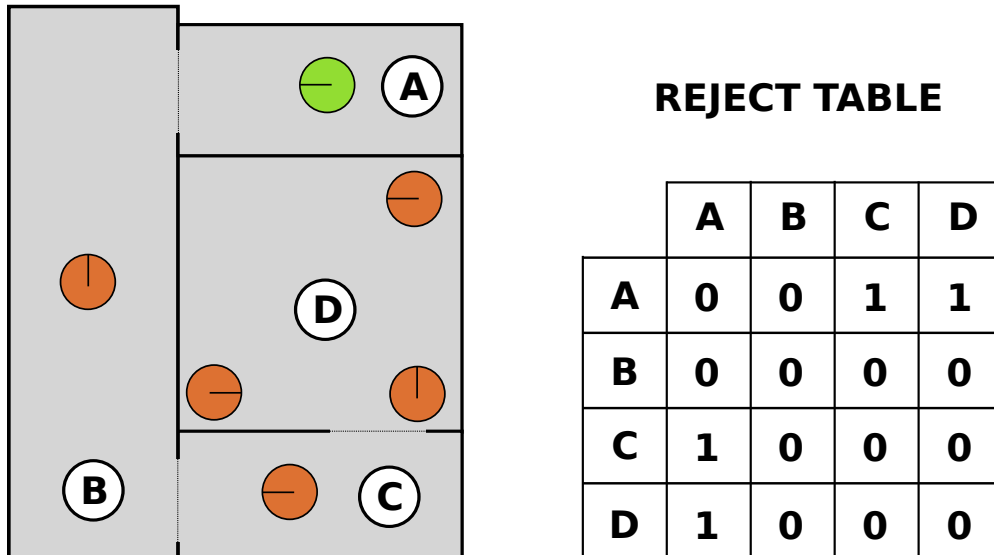


Figure 5.54

Figure 5.54 features four sectors with five monsters and one player in sector (A). The engine needs only to run expensive line of sight calculations for the monster in sector (B). All four others monsters are "rejected" via a cheap lookup. The table is only used for sight. Monsters will still hear the player since sound is cheaper to propagate.

Trivia : With "unlimited" CPU power, modern "node builders" don't build `REJECT` anymore.

³⁰This approach later morphed into the Potentially Visible Set which was instrumental to Quake engine.

5.19 Map Intelligence

Despite lacking a scripting system, maps still managed to offer a rich experience. They were full of surprises with numerous elements interacting with the player. Switches and tripwire-enabled doors, secret passages, elevators, crushing walls, traps and more.

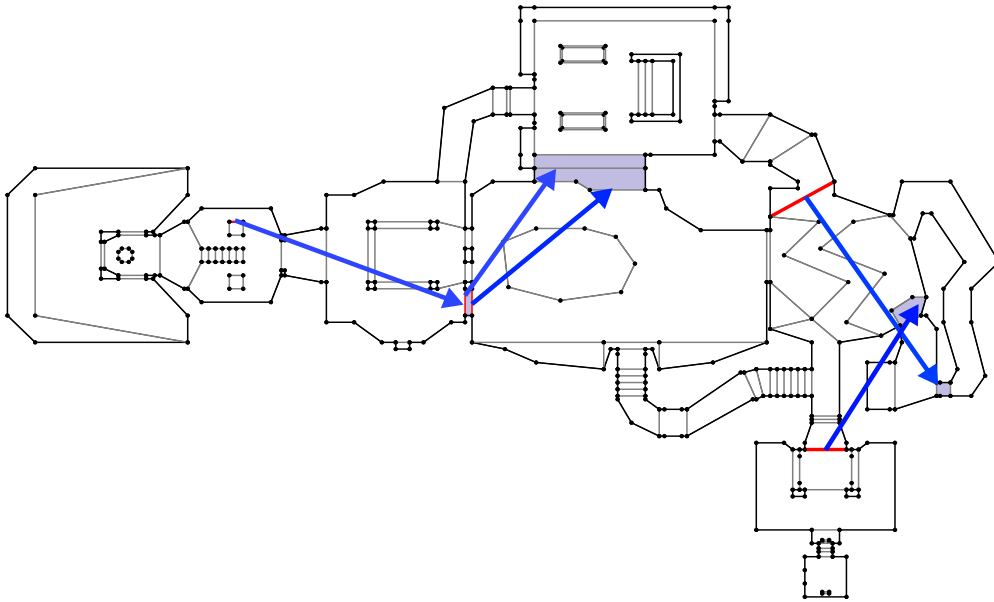


Figure 5.55: UDOOM's E1M1 features 21 special lines and five of them open secret areas.

All interactions are achieved via a simple association between a line's `special` attribute which designates one action to perform and a `tag` value that indicates the target sectors to act on.

The list of action types is impressive – there are more than 130 of them: open/close door at normal/turbo/blazing speeds; raise/lower floor and ceilings; fast ceiling crush & raise; stairbuilders; locked door so you are trapped with monsters; lighting level effects; raise floor to nearest height, texture changes, teleports, level normal and secret exits, and many others.

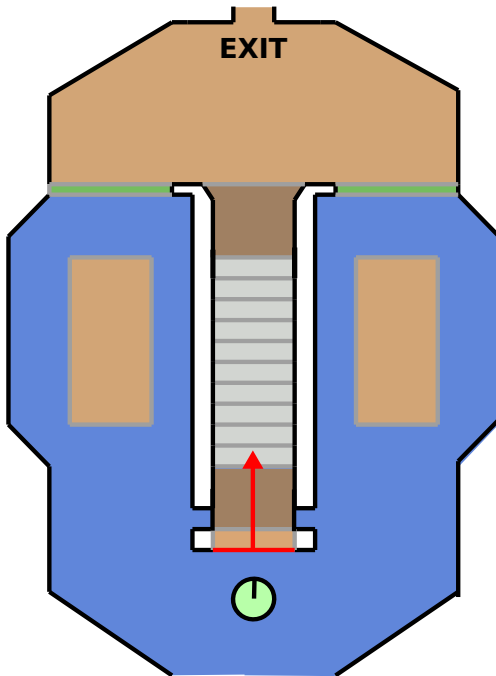
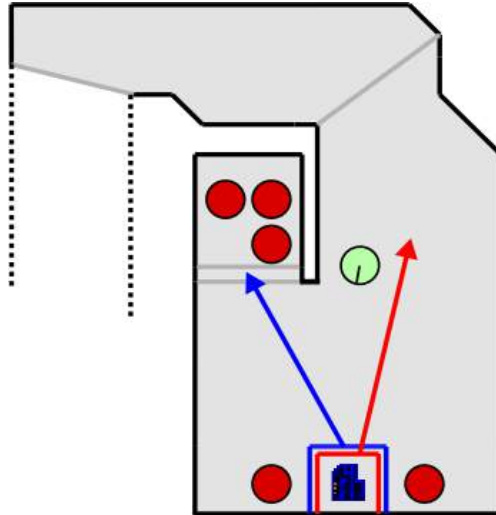
The tag designating the target is any number picked by the designer. Any lines the designer wants to be targeted by this action are tagged with the same number. With this system, a line can trigger only one action but target several sectors.

Trivia : The first maps designed had mostly orthogonal walls and military design. It took the team a few months to realize DOOM engine was capable of much more.

Who doesn't recognize E1M3's hunt for the blue key on page 274? After traversing the whole map, it is finally found sitting on a pedestal, guarded by only two low-level opponents which are easily dispatched.

But as soon as the player picks it up, the lights go out and the sound of a door opening and growling imps can be heard. It was a trap the player just walked into and imps are just behind (page 274)!

To implement this effect, two sets of lines were used. The first lines (in blue) target the door sector behind which the monsters were hiding and opens it. The second lines (in red) target the current sector and set its light level to "very dark".



Another interesting effect in E1M3 is the rising staircase that leads to a small room containing the exit to level 4. It is implemented by having a line with the "BuildStairs" (#8) type targeting the first step sector with its tag. The engine has a hardcoded `EV_BuildStairs` function that looks up the target sector via the tag then uses a flood-fill algorithm.

Adjacent sectors are repeatedly looked up and a bit more height is added with each iteration. To avoid raising the whole level, the algorithm stops when the next sector's texture is different from the last elevated sector, which explains why the stairs are gray, while the top and bottom surrounding them are dark brown.

Before and after screenshots can be seen on page 275.



Finally, the blue key (above)! Nooo, it's a TRAP (below)!





Above, all steps start at the same level. Below, after the "BuildStairs" trigger.



5.20 Game Tics Architecture

With the knowledge of how opponents and map elements work, we can pick up the code where we left it with regard to game simulation on page 168. `G_Ticker` is where all thinkers are run.

```
void G_Ticker (void) {
    [...]
    switch (gamestate) {
        case GS_LEVEL:
            P_Ticker (); // Update actors
            ST_Ticker (); // Status Bar
            AM_Ticker (); // Auto Map
            HU_Ticker (); // HUD
            break;
    }
    [...]
}
```

Most of the meat is in the 3D gameplay (`P_Ticker`) function.

```
void P_Ticker (void) {
    for (int i=0 ; i<MAXPLAYERS ; i++)
        if (playeringame[i])
            P_PlayerThink (&players[i]); // player actions

    P_RunThinkers (); // Monsters
    P_UpdateSpecials (); // Animate planes, scroll walls
    P_RespawnSpecials (); // respawn items in deathmatch
}
```

`P_PlayerThink` is where the player "thinks". This function consumes the `ticccmd_t` (which we already studied on page 250) and controls where the player moves and fires.

`P_RunThinkers` is how the map and monsters "think". Anything that must occur over more than one frame is placed in a thinker object and stored in a doubly-linked list. Thinkers are structs with a function pointer and some data for the function pointer parameters. Each gameplay tic, every thinker in the list "thinks". When thinkers are done thinking they set their function pointer to -1 and are dropped from the list. Note that doors have no feelings but they are nonetheless "thinkers" too.

`P_UpdateSpecials` takes care of animating special textures such as water, or animated walls. It also takes care of switching the texture when a button is pushed.

`P_RespawnSpecials` respawns medikits, weapons, and ammo in deathmatch.


```
typedef void (*think_t) ();

typedef struct thinker_s {
    struct thinker_s *prev, *next;
    think_t          function;
} thinker_t;
```

C has no OOP capabilities yet the engine managed to implement a polymorphism system. Semantically, `think_t` structs are stored in the linked list element (`thinker_t`) with no space for the payload.

```
typedef struct {
    thinker_t    thinker;
    floor_e      type;
    boolean      crush;
    sector_t     *sector;
    int          direction;
    int          newspecial;
    short        texture;
    fixed_t      floordestheight;
    fixed_t      speed;
} floormove_t;
```

`EV_BuildStairs`, which creates thinkers to build stairs, shows how to use the system.

```
void T_MoveFloor(floormove_t *floor);

int EV_BuildStairs(line_t *line, stair_e type) {
    floormove_t *floor;
    [...]
    floor = Z_Malloc (sizeof(*floor), PU_LEVSPEC, 0);
    P_AddThinker (&floor->thinker);
    floor->thinker.function = (think_t) T_MoveFloor;
    floor->direction = 1;
    floor->sector = sec;
    floor->speed = speed;
    floor->floordestheight = height;
    [...]
}
```

This is a pretty cool mechanism. While iterating over the list of `thinker_t`, the loop simply calls `thinker->function(thinker)` which has no knowledge of either the function called or the payload involved.

5.21 Networking

While DOOM's 3D rendering was breathtaking, it was the networking capability and its famous deathmatches that really took it to another level. The ability to connect two PCs and interact with human players was something most gamers had never seen before. Early during development it was apparent this aspect of the game was going to be amazing.

“ I still remember the day that multiplayer started just barely working in Doom. I had two DOS boxes set up in my office in addition to my NeXT workstation to test multiplayer. The IPX networking was forwarding user input between the systems, but there was no error recovery, so it was very fragile. Still, I could spawn two marines in a test level, and they could look at each other.

I was strafing back and forth on one system and looking over my shoulder at the other computer, watching the marine sprite slide side to side in front of the other player's pistol. I let it coast down, centered on the screen, and turned to the other computer. "Bang!" "Urgh!" Twitch. Shuffle. Big smile. :-)"Bang!" "Bang!" "Bang!" "Bang!" There was a consistency failure before the first frag was truly logged, but it was blindingly obvious that this was going to be awesome.

— John Carmack, kotaku.com "Memories Of Doom"

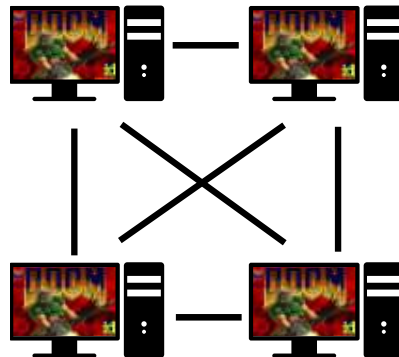
”

5.21.1 Architecture

Most modern FPS games are designed around client/server networking where there are several clients and one source of truth, the server. Clients can join a game any time, send their commands to the server, receive world updates, and perform predictions to minimize communications latency.

There is no central server in DOOM. All peers run their own copy of the game logic and remain in sync by performing no prediction and only running a game tic when all other peers' actions are known.

This means all peers must transmit their command to all other peers, resulting in significant communication overhead. All peers must be present when a game starts. A player can leave (and their avatar will remain in the game without performing any action) but new players cannot join.



On NeXT, the implementation used a simple UDP system with Berkeley sockets³¹. On the PC side, things were more complicated. Until v1.1 the game engine shipped with built-in support for LAN over IPX. To minimize communications, Dave Taylor suggested using IPX packet node number FF:FF:FF:FF:FF:FF to broadcast updates and reduce the amount of communications. Things did not work exactly as expected.

“ Doom used IPX broadcast packets to communicate between the players. This seemed like a good efficiency to me a four player game just involved four broadcast packets each frame. My knowledge of networking was limited to the couple of books I had read, and my naive understanding was that big networks were broken up into little segments connected by routers, and broadcast packets were contained to the little segments. I figured I would eventually extend things to allow playing across routers, but I could ignore the issue for the time being.

What I didn't realize was that there were some entire campuses that were built up out of bridged IPX networks, and a broadcast packet could be forwarded across many bridges until it had been seen by every single computer on the campus. At those sites, every person playing LAN Doom had an impact on every computer on the network, as each broadcast packet had to be examined to see if the local computer wanted it. A few dozen Doom players could cripple a network with a few thousand endpoints.

The day after release, I was awoken by a phone call. I blearily answered it and got chewed out by a network administrator who had found my phone number just to yell at me for my game breaking his entire network. I quickly changed the network protocol to only use broadcast packets for game discovery, and send all-to-all directed packets for gameplay (resulting in 3x the total number of packets for a four player game), but a lot of admins still had to add Doom-specific rules to their bridges (as well as stern warnings that nobody should play the game) to deal with the problems of the original release.

— John Carmack, kotaku.com "Memories Of Doom"

”

With the proliferation of networking, the embedded IPX support started to show its limits.

Instead of baking support for more types of network into the engine, the IPX code was removed and networking was refactored around the notion of network drivers.

³¹The Internet Assigned Numbers Authority (IANA) publishes a list, the Service Name and Transport Protocol Port Number Registry where UDP port 666 is reserved for DOOM!

5.21.2 PC Network drivers

In this model, the game engine deals with a data structure named `doomcom_t` (detailed on page 282) located in shared memory. Receiving or transmitting packets is done via interrupts. How this all worked together is a magnificent hack only possible on a system without proper memory protection.

A "loader" starts first and installs itself as an interrupt handler. It then starts `DOOM.EXE` with a special parameter `-net X` where `X` is the RAM address of the loader's `doomcom_t` variable. The engine literally casts the parameter to an address `((doomcom_t*)(atoi(param)))` to access the structure's fields. From then on the interrupt handler acts as a network driver.

```
C:\DOOM> DOOM.EXE -net 54359695
```

At this point, the engine has everything it needs to communicate in a generic way. It reads or writes to the `doomcom_t` and invokes the handler via the interrupt number (also provided in `doomcom_t`).

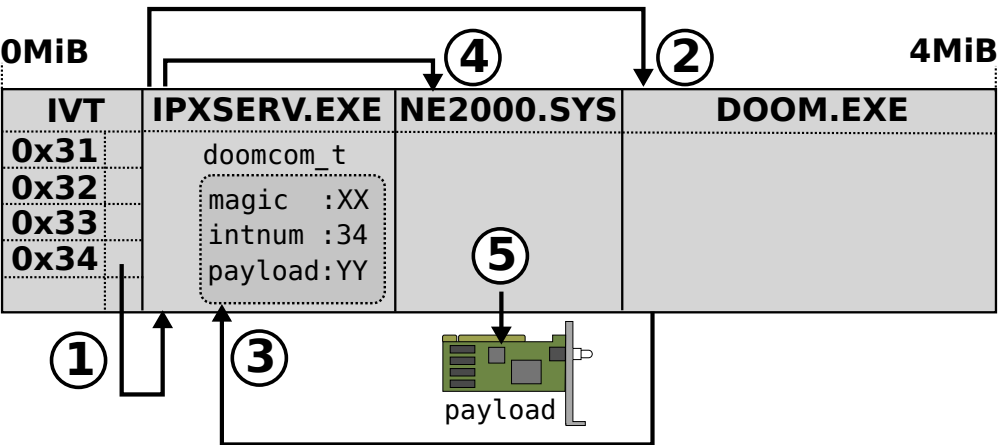


Figure 5.56

Figure 5.56 summarizes the steps. `IPXSETUP.EXE` is started up first ①. The loader registers itself in the software Interrupt Vector Table. Once registered, `IPXSETUP.EXE` starts `DOOM.EXE` and passes the address of its `doomcom_t` variable as an integer ②. Later, during gameplay ③, `DOOM.EXE` reads/writes to the `doomcom_t` payload field and triggers transfers via the interrupt found in the `intnum` field. This triggers ④ the DOOM network driver to communicate with the network card driver. The actual hardware interaction between the network card driver and the physical network card happens in step ⑤.

Two network drivers shipped with the game: IPXSETUP.EXE allowed up to four nodes over IPX, and SERSETUP.EXE allowed two players over serial cable or modem. A company called DWANGO provided their own driver (DWANGO.EXE) to enable 2+ nodes over modem.

5.21.3 Implementation

To perform network I/O, the core only deals with three elements provided by the network subsystem.

Element	Usage
I_InitNetwork	Initialize the network subsystem.
doomcom_t doomcom	Shared structure containing both in and out data.
I_NetCmd	Signal network to send/receive data based on doomcom.

Figure 5.57: DOOM network subsystem interface

On the PC, in the case of a multi-player session, the network initialization simply retrieves the address of the interrupt handler. Notice how the `atoi` return value is cast to a pointer in the last line, an oddly cavalier move nowadays.

```
doomcom_t doomcom;

void I_InitNetwork (void) {
    int i;

    i = M_CheckParm ("-net");
    if (!i) { // single player game
        doomcom = malloc (sizeof (*doomcom) );
        memset (doomcom, 0, sizeof (*doomcom) );
        netgame = false;
        doomcom->id = DOOMCOM_ID;
        doomcom->numplayers = doomcom->numnodes = 1;
        doomcom->deathmatch = false;
        doomcom->consoleplayer = 0;
        doomcom->tickedup = 1;
        doomcom->extratics = 0;
        return;
    }

    netgame = true; // multiplayer game
    doomcom = (doomcom_t *)atoi(myargv[i+1]);
}
```

With access to the `doomcom` variable comes access to the field `intnum` which contains

the software interrupt number that the DOOM network driver registered itself with. When called (via the `int` instruction), it interrupts DOOM to let the card's network driver copy network data in or out of `doomcom`.

```
#define BACKUPTICS 12

typedef struct {
    unsigned checksum;           // high bit=retransmit request
    byte    retransmitfrom;     // only valid if NCMD_RETRANSMIT
    byte    starttic;
    byte    player, numtics;    // player is player id.
    ticcmd_t cmds[BACKUPTICS];
} doomdata_t;

typedef struct {
    long    id;                 // MUST be = DOOMCOM_ID (0x123456781)
    short   intnum;             // DOOM interrupt to execute commands

    // communication between DOOM and the driver
    short   command;           // CMD_SEND or CMD_GET
    short   remotenode;        // dest for send
    short   datalength;        // bytes in doomdata to be sent

    // info common to all nodes
    short   numnodes;          // console is always node 0
    short   ticdup;             // 1 = no dup, 2-5 =dup for slow nets
    short   extratics;         // 1 = send a backup tic in packets
    short   deathmatch;        // 1 = deathmatch
    short   savegame;          // -1 = new game, 0-5 = load savegame
    short   episode;           // 1-3
    short   map;                // 1-9
    short   skill;             // 1-5

    // info specific to this node
    short   consoleplayer;
    short   numplayers;

    doomdata_t    data; // packet data to be sent
} doomcom_t;
```

The fields are self-explanatory but notice `command` which tells the driver if data should be sent or received. The `id` field allows DOOM to verify the address alleged to be a valid network driver. The `ticcmd_t` payload was described on page 250.

At a high level, DOOM's core uses a central function called `NetUpdate` to do all I/O. Notice how it is called in a loop until `ticcmds` for all peers are received, allowing only menus to keep on working. Except for running menus, nothing else will run.

```
void TryRunTics (void) {
    int    lowtic;
    ...
    // a gametic cannot be run until nettics[] > gametic
    while (lowtic < gametic) {
        NetUpdate ();
        lowtic = MAXINT;

        for (i=0 ; i<doomcom->numnodes ; i++)
            if (nodeingame[i] && nettics[i] < lowtic)
                lowtic = nettics[i];

        // don't stay in here forever
        if (I_GetTime ()/ticdup - entertic >= 20)
        { // give the menu a chance to work
            M_Ticker ();
            return;
        }
    }
    ...
}
```

The impossibility for the engine to extrapolate and carry on with gameplay was an issue since all machines ended up running at the lowest common framerate. To mitigate this issue, the engine performs an exotic form of "thread multiplexing" where `NetUpdate` is called several times during a frame. It is typically called no less than eight times.

```
void R_RenderPlayerView (player_t *player) {
    [...]
    NetUpdate ();           // check for new console commands
    R_RenderBSPNode (numnodes-1);
    NetUpdate ();           // check for new console commands
    R_DrawPlanes ();
    NetUpdate ();           // check for new console commands
    R_DrawMasked ();
    NetUpdate ();           // check for new console commands
}
```

Since there is no expectation the medium will guarantee packet delivery, the engine fea-

tures negative acknowledgments, where the packet sequence number is tracked on a per-peer (a.k.a. node) basis. If a packet is received but its sequence number indicates a previous packet was lost, the node requests the missing commands be sent again. This means each node cannot discard commands once they are sent on the wire.

This resend mechanism was a last resort to be avoided at all costs. To this effect, packets feature not only the current command but also the last command (if field `extratics` is set).

5.21.4 DeathManager

Given the complexity of the command-line parameters to set up a network game, several tools were provided. Originally players could use `SETUP.EXE`. In December 1994, id Software introduced `DM.EXE` (DeathManager) which was easier to use.



Figure 5.58: Death Manager 1.2 interface.

In "Old DeathMatch" mode, weapons did not disappear when picked up and ammunition & power-ups never respawned. In July '94, "Deathmatch 2.0" introduced rules changes where all items disappears when picked up and respawns after 50 seconds.

Trivia : DOOM was such a trailblazer that its UDP number (666) is still reserved on most routers and on Windows (file `C:\Windows\System32\drivers\etc`). The "official" IPX socket number (0x869C) is also part of Novell's "well-known static IPX sockets" tables.



Co-op play was fun (above) but so were incredibly bloody deathmatches (below).



5.22 Performance

There is a convenient and portable way to assess performance. Thanks to the fixed tic duration architecture, a demo can be recorded and played back exactly. With frame skipping disabled, a `-timedemo` will produce the exact same sequence of frames to render regardless of the machine's power. Only wall time will vary.

The gold standard of DOOM benchmarking was created by Anton Ertl around 1994. It consists of playing back DEMO3 from the unregistered version of DOOM shareware v1.9.

```
C:\DOOM> DOOM.EXE -timedemo demo3
```

Over the last twenty five years, Anton has gathered metrics for hundreds of configurations³² of the famous game session recorded by John Romero. The machines tested range from Amiga 1200s up to Core i5s. Because no frames are skipped, the playback duration varies. Upon completion two values are displayed.

```
2134 gametics in 1065 realtics
```

The first value, *gametics*, is the number of game tics rendered. For demo3 this is always equal to 2134 since it comes from the recorded lump. The second value, *realtics*, represents the wall time in tics that it took to render every frame.

The average frame per second is obtained with the following formula:

$$fps = \frac{gametics}{realtics} * 35$$

In the previous example³³, the game ran at an average of $\frac{2134}{4268} * 35 = 17.5$ fps.

This mechanism allows running benchmarks across varying configurations. Since machines from 1994 have become difficult to come by these days, a generous collector named Foone Turing kindly volunteered his impressive fleet of machines. The results of the archaeological-benchmarking session are visible in figure 5.59.

The results of the session demonstrate that none of the hardware of the time was able to max out the game. Remember that beyond 35fps there would be no visual improvement since the game logic is hard-coded to run at 35Hz. A machine able to render 70fps

³²Source: <https://www.complang.tuwien.ac.at/misc/doombench.html>

³³Benchmark machine was a miniPC Unisys CWD 4001 (486DX2-66/CirrusLogic-GD5424).

video/audio would render the same game frame twice.

CPU	Frequency	Graphic card	Bus	fps
386DX	33	Tseng Labs ET3000	ISA-8	4
386DX ³⁴	33	Cirrus Logic CL-GD5420	ISA-16	7
486SX	33	Tseng Labs ET3000	ISA-8	7
486SX	33	Cirrus Logic CL-GD5420	ISA-16	11
486SX	33	Diamond Stealth (Tseng ET4000)	VLB	15
486DX2	66	Tseng Labs ET3000	ISA-8	8
486DX2	66	Cirrus Logic CL-GD5420	ISA-16	13
486DX2	66	Diamond Stealth (Tseng ET4000)	VLB	24

Figure 5.59: Benchmark with out-of-the-box DOOM shareware.

Using out of the box settings for the game, a top of the line machine could barely reach 25 fps³⁵. Notice the importance of the bus, which yields a 3x performance hit/boost. At equal frequencies a 486 provides twice the framerate of a 386.

5.22.1 Profiling

Even without special tools, it is possible to gain insight into which parts of the engine are responsible for CPU cycle consumption, thanks to built-in command line parameters.

Parameter `-nodraw` skips rendering altogether (but does blit).

```
C:\DOOM> DOOM.EXE -nodraw -timedemo demo1
2134 gametics in 82 realtics
C:\DOOM>
```

Without drawing, the game's framerate improved from 17fps to 878fps.

Parameter `-noblit` renders to RAM but doesn't transfer the content to VRAM which is a good way to assess the impact of the bus speed. Because of optimizations described later this parameter was only available on non-DOS versions like on NeXTSTEP.

```
$ ./doom -noblit -timedemo demo1
2134 gametics in 6329 realtics
$
```

³⁴For reference, this configuration was able to run Wolfenstein 3D at 20 fps.

³⁵It would not be until the Pentium when PCI buses came out that DOOM could be maxed out at 35 fps.

5.22.2 Profiling With A Profiler

An excellent way of visualizing performance is with a flame graph. These are built by running a program, repeatedly interrupting it, and unwinding the stack starting from the Program Counter to generate a backtrace. This is repeated hundreds of times while playing back a demo. After completion, all backtraces are collected and merged together.

This produces a tree where the width represents 100% of the time and each level is a function call. It provides a visual breakdown of where the machine spends time during a frame³⁶. On NeXTSTEP, thanks to the multiprocessing capability of the OS it is easy to use `gdb` from a second terminal and gather backtraces. The result is as follows.

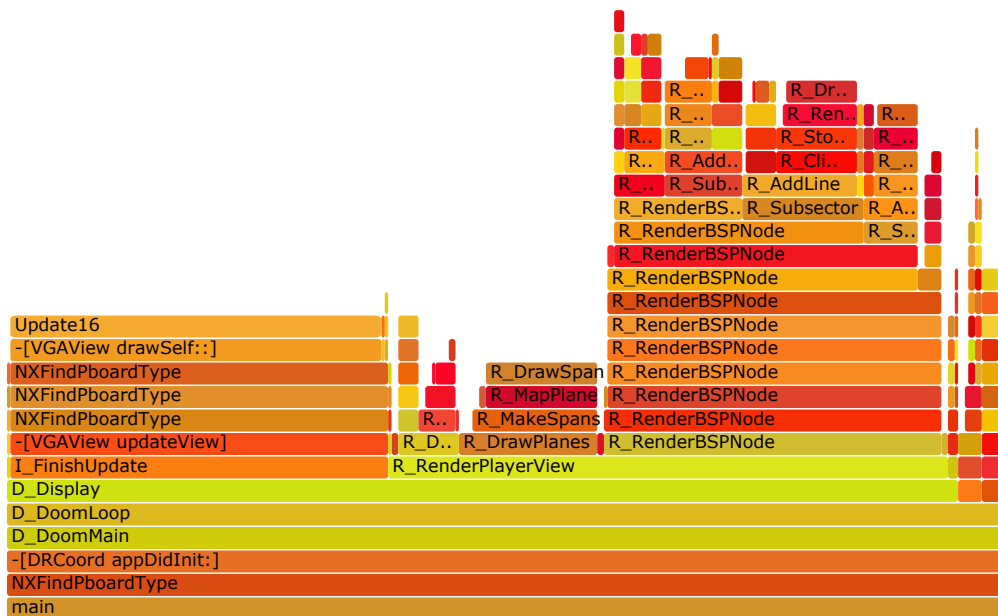


Figure 5.60: Flame graph of `./doom -timedemo demo1` on NeXTstation TurboColor

Obviously, `D_DoomLoop` accounts for 100% of the time, with `D_Display` overwhelmingly dominating³⁷. Gameplay (`TryRunTics`) (isolated column on the very right) is barely visible. In a flame graph, bottlenecks are identified by "mesas" which are high flat plateaus with no children. With that clue, notice the high cost of blitting from framebuffer #0 to the screen (`Update16`), the horizontal drawing routine (`R_DrawSpan`) rendering visplanes and the less obvious BSP traversal resulting in vertical drawing routines (`R_RenderBSPNode`) to render walls/sprites.

³⁶This is a wall time-based flamegraph but there are many other kinds, like CPU-cycle based for example.

³⁷But keep in mind the NeXTSTEP port did not implement the audio system.

On DOS, generating a flame graph is more difficult since it is a single-threaded operating system. However, with a little bit of instrumentation it is possible to get something similar. The following flame graph was generated by instrumenting ten functions.

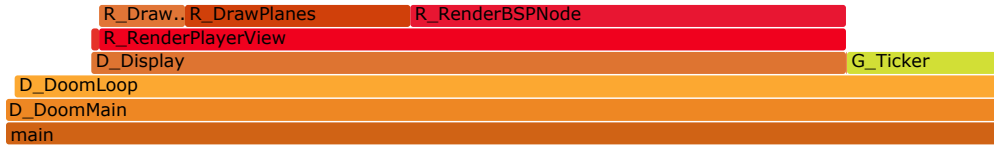


Figure 5.61: Flame graph of DOOM running on DOS

The DOS flame graph shows that A.I. run in `G_Ticker` consumes little CPU time. The audio routine `S_UpdateSounds` is even less taxing since it is barely visible as the tiny red span. This makes sense since the only work it does is to retrieve sound effects and music data from the `.wad` and place it into RAM for the audio card to use.

Something weirder to notice is that, despite being instrumented, `I_FinishUpdate` is not visible at all whereas it was a huge part of the loop on NeXT. How can the DOS version transfer a full framebuffer across the bus so fast? As it turns out, the DOS version benefited from heavy optimization, and as a result, did not have to blit at the end of a frame.

5.22.3 DOS Optimizations

Given that it was meant to be the money maker, the DOS version received special care. During optimization sessions, John Carmack identified three areas for improvement. One was in the math functions and two had to do with the 3D renderer.

“

An exercise that I try to do every once in a while is to "step a frame" in the game, starting at some major point like `common->Frame()`, `game->Frame()`, or `renderer->EndFrame()`, and step into every function to try and walk the complete code coverage.

This usually gets rather depressing long before you get to the end of the frame. Awareness of all the code that is actually executing is important, and it is too easy to have very large blocks of code that you just always skip over while debugging, even though they have performance and stability implications.

— John Carmack

”

5.22.3.1 Math Optimizations

Fixed-point operations are performed everywhere. The function `FixedMul` is found 124 times in the source code and is called over a thousand times per frame. Along with `FixedDiv2`, it was optimized with inline assembly³⁸. The C version `"return ((long long) a * (long long) b) >> 16"` was a function call in Watcom resulting in close to 30 instructions but the assembly version only uses two.

```
#ifdef __WATCOMC__
#pragma aux FixedMul = \
    "imul ebx", \
    "shrd eax,edx,16" \
    parm    [eax] [ebx] \
    value   [eax] \
    modify exact [eax edx]
#endif
```

5.22.3.2 Direct Framebuffer Access

A more substantial optimization had to do with layering. On DOS the rules separating the core and the video system were bent. Renderers like the status bar and the automap still function as originally designed but 3D drawing functions such as `R_DrawColumn` and `R_DrawSpan` are given direct access to the VGA banks. By bypassing framebuffer #0, one read and one write per pixel (plus bus transfer) are avoided. Since the menu renderer has to be able to draw on top of everything, it was also granted direct VGA VRAM access.

5.22.3.3 Assembly Renderer Optimization

Not only were `R_DrawColumn` and `R_DrawSpan` given direct access to the VGA bank, they were hand-optimized with gorgeous assembly using all the tricks in the book. Taking a look at `R_DrawColumn` from `planar.asm` is revealing. The function uses self-modifying code (see reserved value `12345678h` meant to contain the scaling factor, which is patched). We can also see the loop was unrolled to process two pixels at a time (and therefore avoid emptying the i486 prefetch queue because of the `jnz` instruction). Notice the cool trick where register `eax` is reused three times, once as a pointer to the texture source, then as `al` for texel storage (`al`), then as `al` for translated texel (lightmapped) storage again. Overall this method yields an amortized 7 instructions per pixel for drawing columns which is remarkable.

Altogether these three optimizations improved performance by a substantial 15%.

³⁸There is next to no assembly in DOOM. Around 1994, John Carmack even declared that "The days of assembly are numbered". This was without counting on Intel's Pentium, the super-scalar architecture of which would require extra care by Michael Abrash for Quake.

```

PROC    R_DrawColumn_
[...].  ; edi = destscreen + y*80 + x/4
[...].  ; set VGA mapmask register
[...].  ; ebp= texture delta 7:25 fix point
mov     esi,[_dc_source] ; esi = texture source
mov     ebx,[_dc_iscale]
shl     ebx,9
mov     eax,OFFSET patch1+2 ; patch scaling code
mov     [eax],ebx
mov     eax,OFFSET patch2+2 ; patch scaling code
mov     [eax],ebx

mov     ecx,ebp           ; begin calculating 1st pixel
add     ebp,ebx           ; advance frac pointer
shr     ecx,25            ; finish calculation for 1st pixel
mov     edx,ebp           ; begin calculating 2nd pixel
add     ebp,ebx           ; advance frac pointer
shr     edx,25            ; finish calculation for 2nd pixel
mov     eax,[_dc_colormap]
mov     ebx,eax
mov     al,[esi+ecx]      ; get first pixel
mov     bl,[esi+edx]      ; get second pixel
mov     al,[eax]          ; color translate 1st pixel
mov     bl,[ebx]          ; color translate 2nd pixel
doubleloop:
    mov     ecx,ebp           ; begin calculating 3rd pixel
patch1:
    add     ebp,12345678h    ; advance frac pointer
    mov     [edi],al         ; write first pixel
    shr     ecx,25          ; finish calculation for 3rd pixel
    mov     edx,ebp         ; begin calculating for 4th pixel
patch2:
    add     ebp,12345678h    ; advance frac pointer
    mov     [edi+PLANEWIDTH],bl ; write second pixel
    shr     edx,25          ; finish calculation for 4th pixel
    mov     al,[esi+ecx]     ; get third pixel
    add     edi,PLANEWIDTH*2 ; advance to 3rd pixel dest
    mov     bl,[esi+edx]     ; get fourth pixel
    dec     [loopcount]      ; done with loop?
    mov     al,[eax]         ; color translate 3rd pixel
    mov     bl,[ebx]         ; color translate 4th pixel
    jnz     doubleloop
ENDP

```

5.23 Performance Tuning

Despite all the care and optimization, most gamers could not get more than 10 fps with the game out of the box. To help reach a decent framerate, two tuning mechanisms helped to reduce the number of pixels written.

1. High detail/low detail toggle.
2. Adjust the size of the 3D canvas.

Trivia : These tradeoffs were not specific to the PC. All console ports – from the "weak" Super Nintendo to the "strong" PlayStation – used a combination of these two settings.

5.24 High/Low detail mode

The first tuning option was to lower the horizontal resolution using column doubling. In low resolution mode the engine only renders one out of every two columns but it writes them to the framebuffer twice. Given how the 3D renderer dominates runtime, this resulted in a tremendous performance improvement since fewer pixel values are generated but they also don't have to transit the bus. A performance gain confirmed as the following benchmark shows.

High detail resolution	High detail FPS	Low detail resolution	Low detail FPS
320x200	19	160x200	29
320x168	20	160x168	30
288x144	23	144x144	32
256x128	25	128x128	35
224x112	28	112x112	38
192x096	31	096x096	41
160x080	35	080x080	45
128x064	40	064x064	49
096x048	45	048x048	54

Figure 5.62: DOOM performance in low and high detail mode.

The benchmark above was conducted on a machine which would have been deemed "top of the line" in 1994, a Unisys CWD 4001 featuring a 486DX2-66 CPU with a Cirrus Logic VLB graphics card. Using low detail mode instead of high detail yields a variable 20%-50% performance improvement which brought DOOM up to a playable framerate on "weaker" PCs running on Intel 386 CPUs.



Above, the high resolution is 320x168. Below, in low resolution, it is dropped to 160x168 with odd columns duplicated. The resolution drop is particularly noticeable on the non-magnified sergeant and the door but only because diminished lighting and CRT scaling are disabled for demonstration purposes. In practice the difference was more subtle.

With direct access to the VGA banks, "low detail mode" is a completely free optimization without the need to write the same pixel column twice, since the VGA mask is set up to write the same pixel to two banks simultaneously.



5.25 3D Canvas size adjustment

Another option to improve the frame rate was to lower the size of the 3D canvas. The player had access to a sliding bar allowing eight sizes to be selected. The slider was a multiplier affecting the variable `numblocks` to produce a value in the range [3,11]. Value 11 was special and hard-coded to be recognized as "full-screen" 320x200.

```
void R_ExecuteSetViewSize (void) {
    viewwidth  = numblocks * 32;
    viewheight = (numblocks * 168/10)&~7;
}
```

It is unknown if anybody had the misfortune and courage to play the game in mode 3. This would have been an achievement in itself.

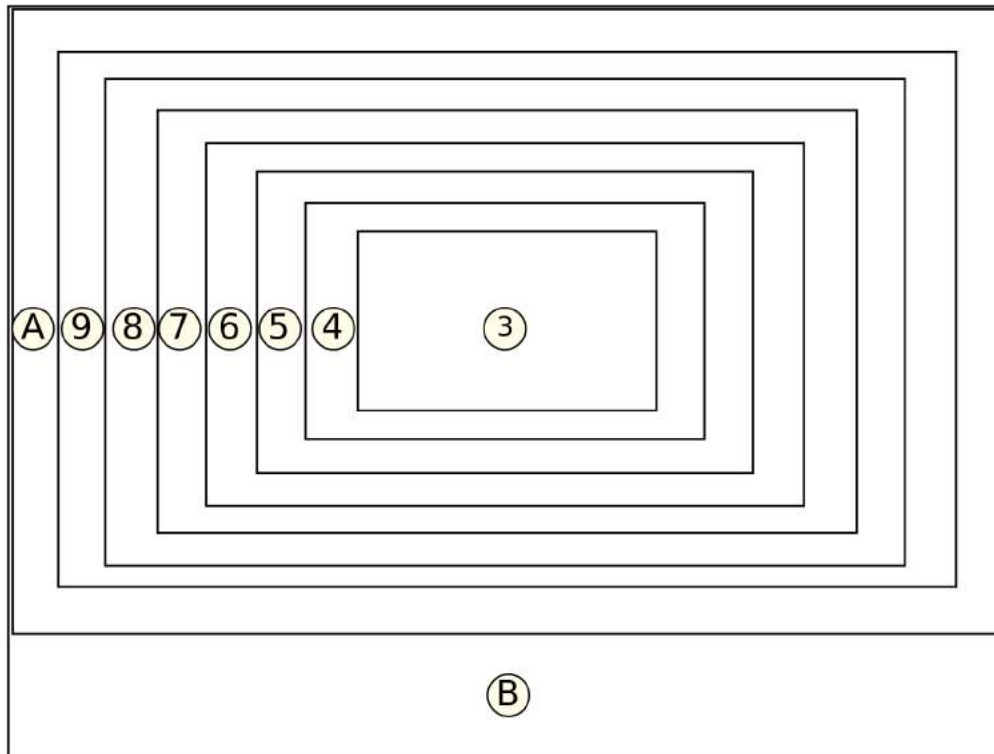


Figure 5.63: The nine 3D canvas configurations.

Trivia : id shipped DOOM by default in "high detail" with canvas size 9.

numblocks	Width	Height	# Pixels	# Pixels %	fps
0xB	320	200	64,000	100	19
0xA	320	168	53,760	84	20
0x9	288	144	41,472	64	23
0x8	256	128	32,768	52	25
0x7	224	112	25,088	39	28
0x6	192	96	18,432	28	31
0x5	160	80	12,800	20	35
0x4	128	64	8,192	12	40
0x3	96	48	4,608	7	45

Figure 5.64: Benchmark of performance gain vs 3D canvas size.³⁹

Gains are not as substantial as the detail level. Reducing the 3D canvas size by 50% yields only a 31% improvement yet the visible area is so small it is almost not worth it.



Figure 5.65: DOOM out-of-the-box configuration: Mode 9, High detail

³⁹Benchmarked on a miniPC Unisys CWD 4001 (486DX2-66/CirrusLogic-GD5424, 8MiB RAM).

Chapter 6

Game Console Ports

The success of the PC version and its mind-numbing sales figures made it an extremely desirable title for any game console publisher. From 1994 to 1997, DOOM was ported to the six major systems of the era with varying degrees of success.

At the time, the four year "console war" was raging, with consoles ranked by generation according to their "bitness". The third generation, 8-bit NES and Sega Master System had long since disappeared. The fourth, 16-bit generation, consisting of Nintendo's SNES, Sega's Genesis and the Turbografx-16 was reaching end of life. The fifth, 32-bit generation, was starting to appear, featuring Sony's Playstation and Sega's Saturn, with marketers trying to brand some systems as 64-bit, including the Nintendo 64 and Atari Jaguar¹. In retrospect, it was a rich period, prone to hardware innovation which contrasts compared to 2018's uniform world of Sony vs Microsoft where systems barely differ by more than their name.

The architecture of the DOOM engine, based on a core with system-specific subsystems, may suggest it would have been an easy task to port it to consoles. This intuition could not be further from the truth. From design trade-offs due to restricted resources, to crazy schedules, every version has a unique and rich story to tell.

Technically, the common problem to solve was to deal with smaller memory requirements. The PC minimum requirement was to have 4 MiB installed on the machine. It was of course not possible to ask customers to add more RAM to their console. Developers sometimes had to deal with as little as 512 KiB of RAM which was eight times less than the original version.

The second technical difficulty was in dealing with exotic hardware. The PC was designed around a single "big iron" CPU while consoles were made of a constellation of processors.

¹After this, consumers realized the silliness of the whole nomenclature. Bitness was soon forgotten.

6.1 Jaguar (1994)

Development of the Jaguar started in 1990 when Atari commissioned Cambridge-based Flare Technology to design not just one but two new game systems simultaneously: a fourth generation, 32-bit system called Panther, and an audacious 64-bit system called Jaguar².

Three years later, with the Jaguar project ahead of schedule, Atari decided to abandon the Panther and released the 64-bit Jaguar in November 1993.



Martin Brennan, Ben Cheese, and John Mathieson from Flare Technology made opinionated design decisions. Besides the 18-button controller, the machine had no fewer than five processors to juggle with.

On the audio side there was a 32-bit 27MHz RISC CPU nicknamed "Jerry". On the video side, three processors were all contained in a 32-bit 27MHz RISC chip nicknamed "Tom" with a GPU, blitter, and object processor. To orchestrate everything, there was a 16/32-bit 13Mhz 68000 with 2MiB of RAM. Connecting them all, to the delight of marketing, was a 64-bit data bus.

The bold ad, "Do the Math!" featuring the 64-bit claim triggered mostly suspicion from potential customers. No matter how John Mathieson attempted to explain it in interviews, the machine felt like an attempt to mislead by an already suspicious Atari marketing department. How Atari could have managed to manufacture something four times better than the 16-bit Super Nintendo and Sega Genesis was not clear.

²Atari named all its consoles after big cats. Besides the Jaguar and Panther, its handheld game system from 1989 was called "Lynx".

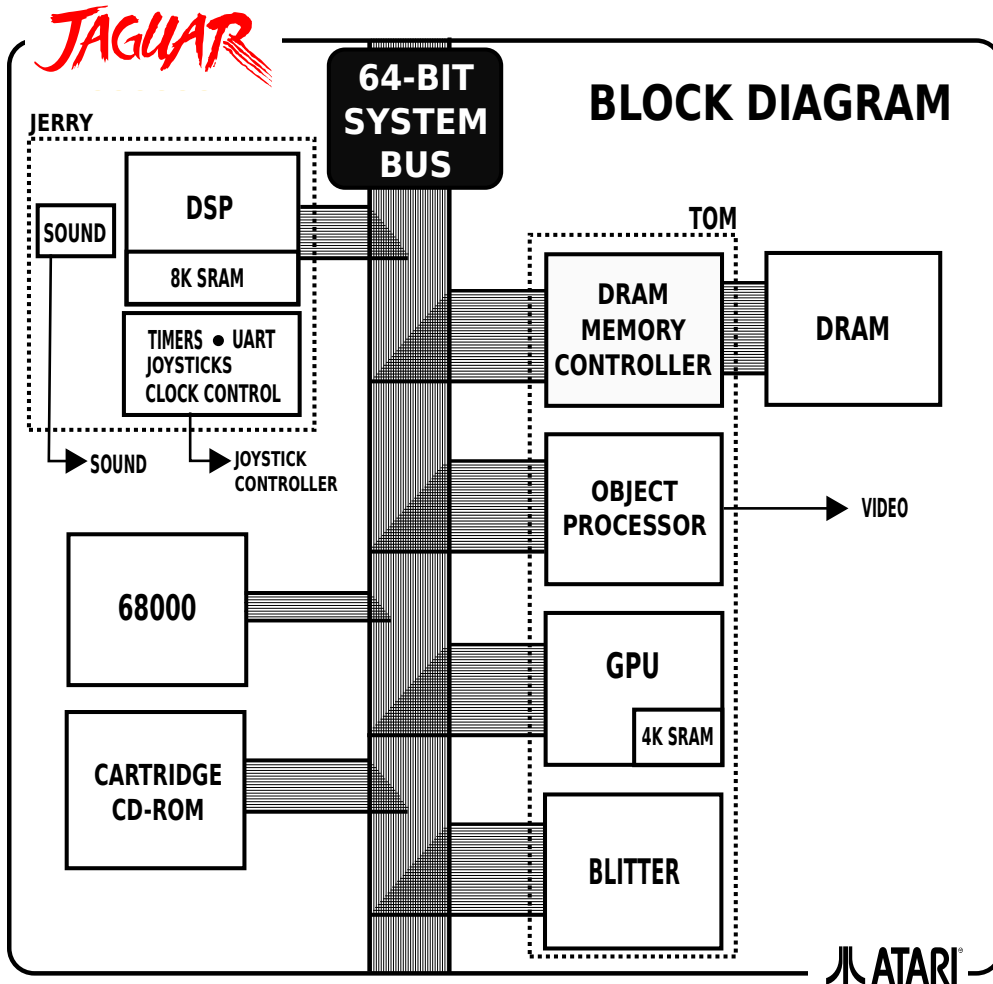
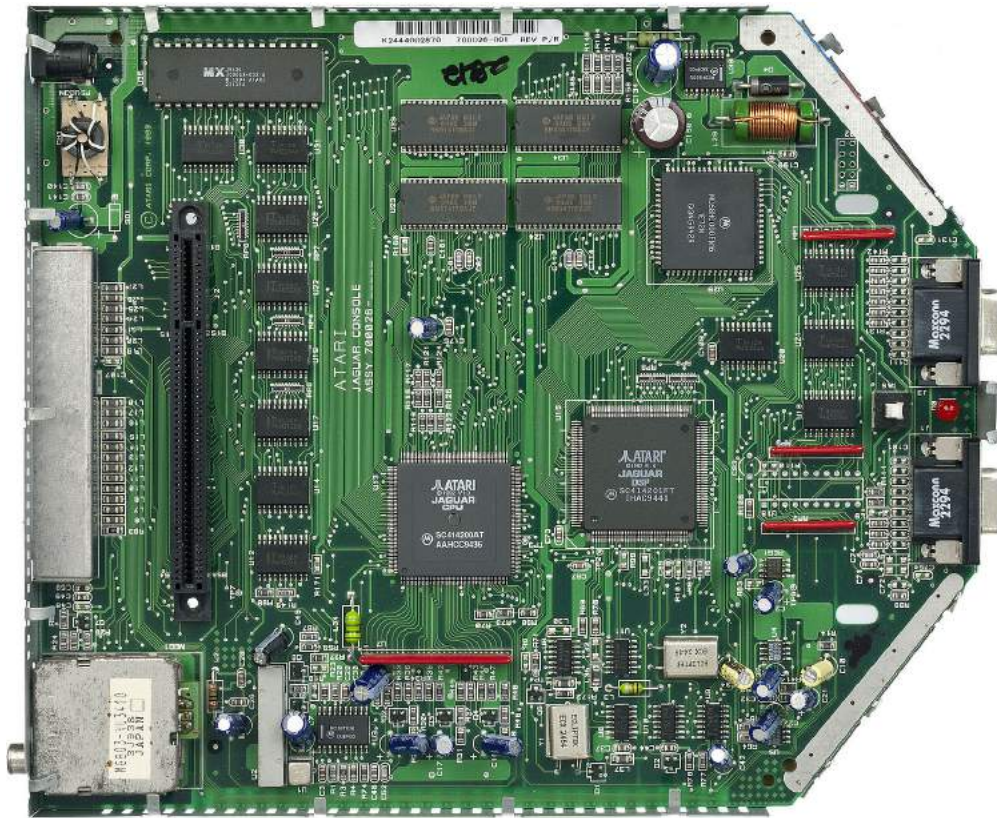


Figure 6.1: Architecture of the Jaguar. Notice the uneven buses.

Cautious purchasers on one side were met by incredulous game developers on the other side. The five-processor architecture was powerful but highly unusual for people accustomed to dealing with a single processor on the 8-bit Nintendo Entertainment System or Sega Master System. Programming the Jaguar was an art that few took the time to learn.

The limited library of games available at launch prevented the formation of a critical mass of customers. Low sales figures made developers less likely to invest in the Jaguar which in turn impacted sales. Over its three-year lifetime, Atari sold about 100,000 units.

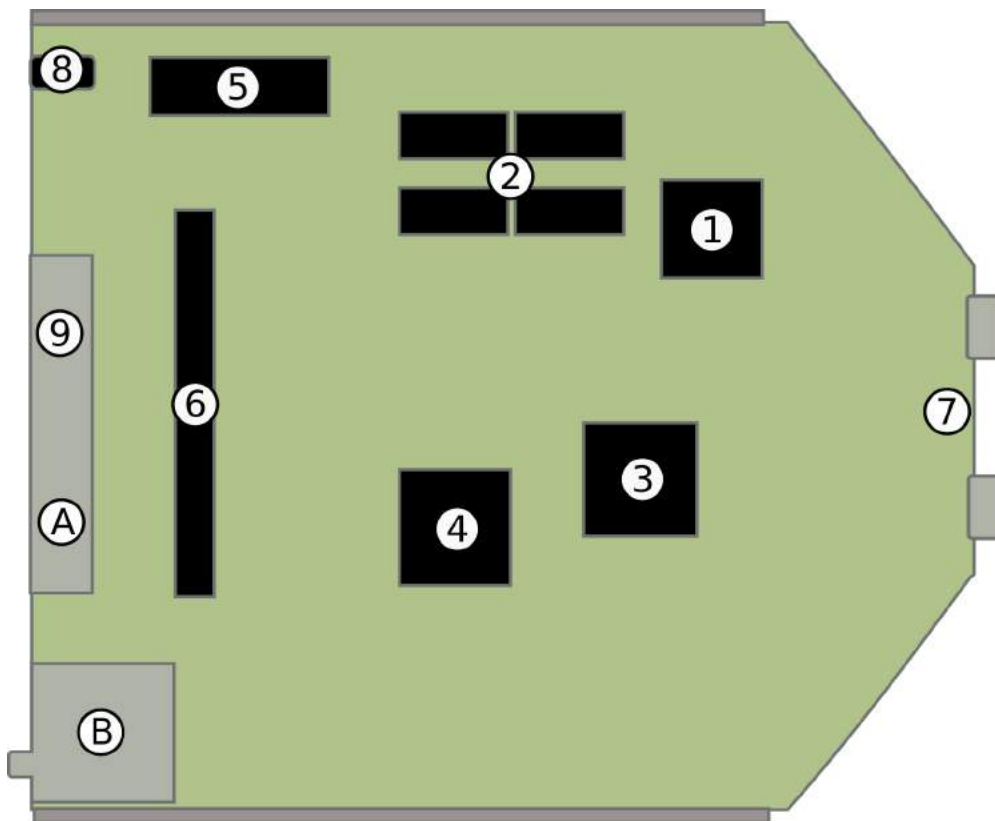


Inside the machine: ① Motorola 68000, ② 2MiB RAM, ③ JERRY, ④ TOM, ⑤ Operating System ROM, ⑥ Cartridge slot, ⑦ Joystick ports, ⑧ AC Adapter Jack, ⑨ DSP Port, ① Monitor (composite, component, and S-Video) Port, and ② Channel switch, TV port.

“Jaguar has a 64-bit memory interface to get a high bandwidth out of cheap DRAM. ... Where the system needs to be 64 bit then it is 64 bit, so the Object Processor, which takes data from DRAM and builds the display is 64 bit; and the blitter, which does all the 3D rendering, screen clearing, and pixel shuffling, is 64 bit. Where the system does not need to be 64 bit, it isn't. There is no point in a 64-bit address space in a games console! 3D calculations and audio processing do not generally use 64-bit numbers, so there would be no advantage to 64-bit processors for this.

— John Mathieson

”



John Mathieson granted many interviews giving more insight into the constraints a hardware engineer has to deal with, from cost-related pressures to mandates from Atari to use CPUs from Motorola. The grass on the hardware side was not greener than on the software side.

“ Atari were keen to use a 68K family device, and we looked closely at various members. We did actually build a couple of 68030 versions of the early beta developers systems, and for a while were going to use a 68020. However, this turned out too expensive. We also considered the possibility of no [Motorola 680x0 chip] at all. I always felt it was important to have some normal processor, to give developers a warm feeling when they start. The 68K is inexpensive and does that job well.

— John Mathieson

”

6.1.1 Programming The Jaguar

To unleash the beast meant having all five processors work in parallel³. It was complicated in theory and complicated in practice.

“ The 68000 may be the CPU in the sense that it's the center of operation, and boot-straps the machine, and starts everything else going; however, it is not the center of Jaguar's power. ... The 68000 is like a manager who does no real work, but tells everybody else what to do.

I maintain that it's only there to read the joysticks.

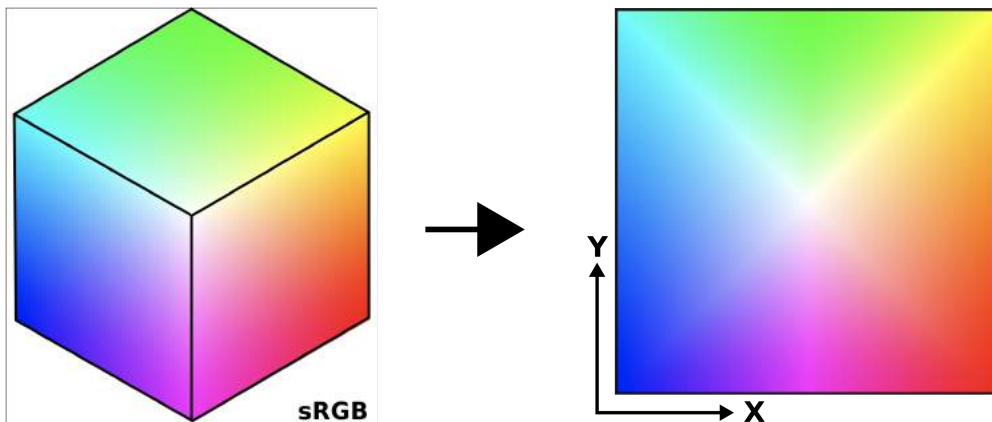
— John Mathieson

”

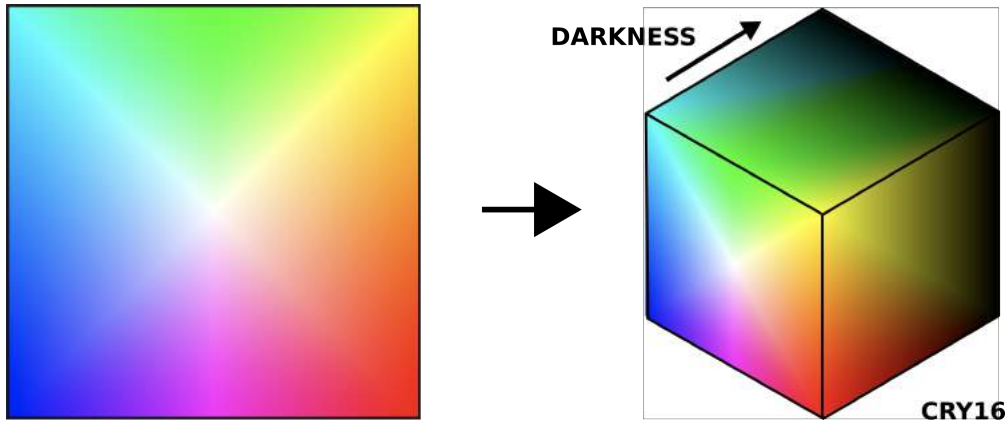
6.1.1.1 Theory

The Motorola 68000 is used as a manager. It deals with the outside world and manages resources for the other processors. It is at the highest control level and has complete control over the system.

The Object Processor is connected to the TV and is in charge of generating display lines. It reads an object list usually made of pixels (which can overlap). It performs all the functions of a traditional sprite engine. Its 16-bit per-pixel CRY (Cyan-Red-Yellow) color model was unconventional for consoles at that time. One byte is an (X,Y) coordinate in a sRGB cube flattened into a square which gives a color. The other byte gives the brightness, for a total of 65,536 colors.



³Source: "Jaguar Technical Reference Manual: Tom & Jerry".



The Graphics Processor is the powerhouse with a high instruction throughput, a powerful ALU with a parallel multiplier, barrel-shifter, and a divide unit, in addition to the normal arithmetic functions. Its internal 4KiB of SRAM is meant to store not only data to operate on but also its local program instructions.

The Blitter performs fast RAM block move and fill operations. It can generate strips of pixels for Gouraud-shaded Z-buffered polygons and is also capable of skipping pixels (based on Z-testing). It is capable of rotating bitmaps, line-drawing, character-painting, and a range of other effects. It is in charge of loading the SRAM for Tom & Jerry with local data and instructions

Jerry, the DSP, is the twin brother of the Graphics Processor. Its bigger local SRAM (8 KiB) and smaller 32-bit connection to the 64-bit main bus make it a perfect candidate for generating music and sound effects. However, it was at the programmer's discretion to make it perform other tasks, even graphical ones. The versatility is such that the Jag-Link connecting two Jaguars together is plugged directly in the "DSP port" on the back of the console and Jerry is in charge of networking.

All RAM (even SRAM in the RISC CPUs) is addressable by any component thanks to a flexible memory controller. At any point during execution of their programs, any processor can become the DMA bus master. Despite its status as governor, the 68000 has the lowest level of priority (were its DMA master request to conflict with another processor) while the DSP is king, in order to avoid extremely unpleasant audio glitches.

6.1.1.2 Practice

The hardware had several bugs, especially at the memory controller level, making multi-tasking hard to rely on and even harder to debug.

It may not be obvious at first sight but the Motorola 68000 and Tom/Jerry had different

architectures and different instruction sets. The intended workflow was to program the Motorola in C while the GPU/DSP RISC path was more convoluted. The programmer had to first learn the new instruction set, then write assembly code, use the provided assembler to generate machine code, and finally write a full pipeline to store and later deliver the machine code to local program-dedicated SRAM on Tom & Jerry.

6.1.2 Doom On Jaguar

John Carmack took an early interest in the Jaguar and did the port himself with Dave Taylor taking care of the sound and MIDI music. It was not John's first contact with the machine.

“

I converted Wolfenstein 3D on a whim. I was thinking about how the Jag's hardware could be applied to games other than Doom, and Wolfenstein seemed a pretty good utilization. I started programming one afternoon and 15 CDs later, when the other guys were coming in the next morning, I had a functional port of the Jaguar Wolfenstein code running. We sent it to Atari, and they gave us the go-ahead to stall Doom for a little while and get Wolfenstein out real quick.

— John Carmack for *EDGE Magazine*, June 1994

”

For DOOM, the duo had something up and running two weeks after they signed off on the port, even though it was running at a wretched rate⁴.

Trivia : To ease the task of generating RISC instructions for Tom & Jerry, John Carmack wrote his own `lcc` compiler backend. Output was optimized further by hand.⁵

To run on a machine with 50% less RAM was difficult. The Cyberdemon and Spidedemon were cut. Sprite and texture resolution were reduced. Maps were heavily modified to use fewer textures, have fewer segments, and create fewer visplanes. Take a look at E1M1 in Figure 6.2 and compare it with the PC version (on page 223). See how the blue floor texture is gone and there is only one step instead of two.

The 3D renderer had to be rewritten to fit in a small chunk of ASM that ran on the RISCs. Nine overlays were swapped in and out of the SRAM at runtime based on engine "phases".

⁴Source: *EDGE Magazine*, June 1994.

⁵It was not the last time id Software would use the excellent `lcc`. In 1999, it was used for Quake 3 to generate the VM bytecode.

**Figure 6.2**

The source code of DOOM for Jaguar was released in May, 2003 by Carl Forhan of Songbird Productions. Peeking inside reveals the details of how memory footprint was reduced. Visplane storage for example was reduced from 128 to 64.

```
#define MAXVISPLANES 64
extern visplane_t visplanes[MAXVISPLANES];
```

A big decision was to cut music playback during game. This was due to the poor performance of Tom which was not able to handle the game engine on its own. To resolve this problem, Jerry (the so-called DSP) was used to run collision detections. Thankfully, there was still enough juice in Jerry to play sound effects and the engine managed a solid 20 FPS.

The 3D rendering was done at a resolution of 160x180. Columns were doubled to reach 320x180 with 60 pixels at the bottom for the status bar, bringing the overall resolution to 320x240. In many ways the graphical result was better than the PC. With its tailor-made 16-bit CRY mode, the Jaguar really had 65,536 colors with no color banding in sight.



Further inspection shows the Jaguar code also contains a few artifacts from the Sega 32X version (MARS was the code name of the Sega 32X). We can also learn that development was done on NeXTSTEP in what was called "Simulator" mode.

```
#ifndef JAGUAR
#ifndef MARS
#define SIMULATOR
#endif
#endif

#ifndef MARS
#define SCREENWIDTH 160
#define SCREENHEIGHT 180
#else
#define SCREENWIDTH 128
#define SCREENHEIGHT 144
#endif
```

“

The memory, bus, blitter and video processor were 64 bits wide, but the processors (68k and two custom RISC processors) were 32 bit.

The blitter 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 useful 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 blitting. Still, the experience with the limitations and hard failure cases of that gave me good ammunition to rail against Microsoft's (thankfully aborted) talisman project.

The little RISC engines were decent processors. I was surprised that they didn't use off the shelf designs, but they basically worked ok. They had some design hazards (write after write) that didn't get fixed, but the only thing truly wrong with them was that they had scratchpad memory instead of caches, and couldn't execute code from main memory. I had to chunk the DOOM renderer into nine sequentially loaded overlays to get it working (with hindsight, I would have done it differently in about three...).

The 68k was slow. This was the primary problem of the system. Your options were either taking it easy, running everything on the 68k, and going slow, or sweating over lots of overlaid 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

”

6.2 Sega 32X (1994)

In January 1994, Sega was in a delicate position. The Genesis, its 16-bit moneymaker, was losing ground in Japan. By 1993 sales had placed the machine third, behind Nintendo's Super Famicom and NEC's PC Engine. To make things worse, Sega now had to deal with two new competitors who had entered the game console market in 1993, with Atari's Jaguar and Panasonic's 3DO. The consensus at Sega Of Japan (SOJ) was that the company should put all of its available resources into the 32-bit Saturn project.



While SOJ's work on the Saturn was moving forward, it was feared it would be a while before it would be finished. In the US, the Genesis had been selling well (32 million units as of late 1993) and Sega of America (SOA) was eager to take the financial opportunity to create a Genesis "booster".

During CES '94 in Las Vegas, then-CEO of SOJ Hayao Nakayama summoned Sega of America (SOA) executives Joe Miller – Head of R&D, Marty Franz – SOA Technical Director, and Scot Bayless – Senior Producer, into a conference call⁶.

They were given the green light for project "Mars" with the goal to release a Genesis Booster within nine months. Incredibly, they managed to reach their target. The Sega 32X was released in November 1994.



⁶Source: Retrogamer #77. All quotes in this section are also from Retrogamer interviews.

The 32X, as it would be marketed, was to be inserted like a cartridge. 32X games were inserted on top of the overall assembly. Games had access to everything including the Genesis's 7.6 MHz Motorola 68000 and the 3.58 MHz Zilog Z80.

“ After the call ended, Marty Franz grabbed one of those little hotel notepads and drew a couple of Hitachi SH2 processors, each with its own frame buffer. That's pretty much where the 32X started.

— Scot Bayless

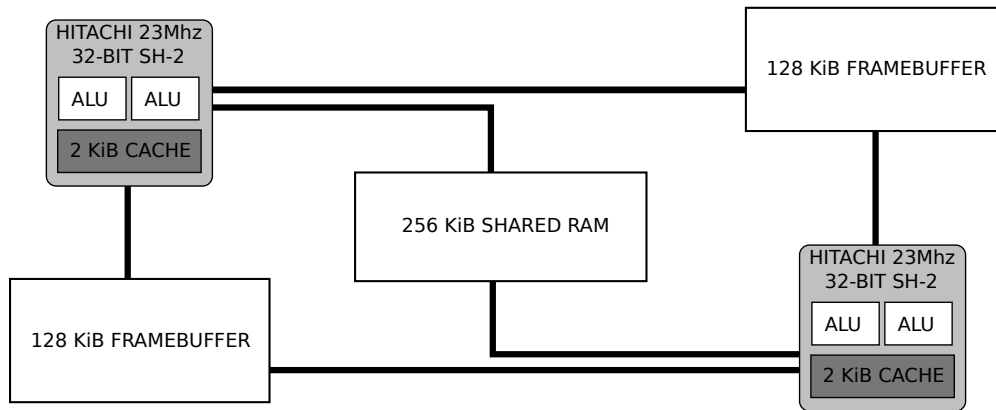


Figure 6.3: What the notepad may have looked like.

“ The design of the graphics subsystem was brilliantly simple; something of a coder's dream for the day. It was built around two central processors feeding independent frame buffers with twice the depth per pixel of anything else out there. It was a wonderful platform for doing 3D in ways that nobody else was attempting outside the workstation market.

— Scot Bayless

Besides the dual SH-2, the 32X was gifted an impressive audio chip from QSound. Capable of Pulse-Width Modulation, it added extra channels and even had multidimensional sound capability that allowed a regular stereo audio signal to approximate the 3D sounds heard in everyday life. Also present was a graphics chip named "VDP" that was in charge of double-buffering to avoid tearing and was also capable of clearing the framebuffers rapidly.

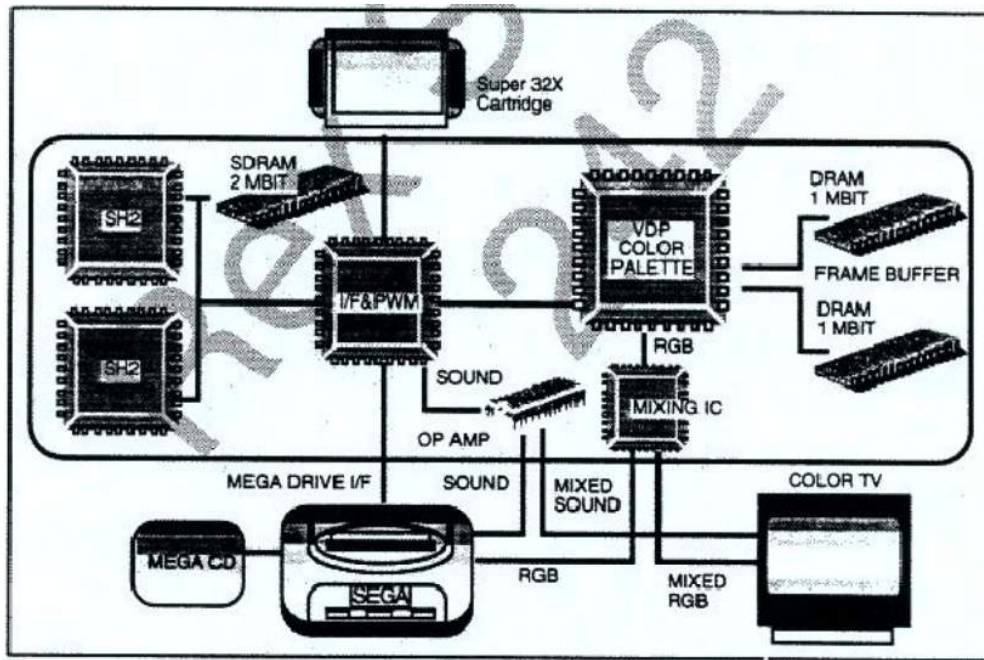


Figure 6.4: The 32X system as summarized in the developer documentation.

The design bore similarities to the Saturn (which also used two SuperH CPUs) but with a different philosophy.

“ The Saturn was essentially a 2D system with the ability to move the four corners of a sprite in a way that could simulate projection in 3D space. It had the advantage of doing the rendering in hardware, but the rendering scheme also tended to create a lot of problems, and the pixel overwrite rate was very high; much of the advantage of dedicated hardware was lost to memory access stalls. The 32X, on the other hand, did everything in software but gave two fast RISC chips tied to great big frame buffers and complete control to the programmer.

— Scot Bayless

With engineers pouring their hearts into the system and DevRel doing amazing work to have a decent portfolio of games for launch date, the 32X managed to sell 665,000 units by the end of 1994. This promising start was unfortunately followed by a sad story.

A story best summarized by Damien McFerran, reporter at RetroGamer.

“ How do you take half a decade’s worth of critical and commercial success and flush it down the toilet?

Easy: you release a device like the Sega 32X.

— **Reporter for RetroGamer #77**

”

What crippled the 32X was the Saturn. Throughout 1994, work had continued at SOJ on the 32-bit system. Enough progress was made that Sega decided to release it in Japan in November 1994, way ahead of its original schedule. This was the same month the 32X was to be released in USA.

“ Not surprisingly, word got out quickly in the West, US and EU consumers immediately started asking the obvious question: ‘Why should I buy 32X when Saturn is only a few months away?’ Sadly, the best answer Sega could come up with was that 32X was a ‘transitional device’ - that it would form a bridge from Mega Drive to Saturn.

It made us look greedy and dumb to consumers, something that a year earlier I couldn’t have imagined people thinking about us. We were the cool kids.

— **Scot Bayless**

”

This poor timing made the 32X almost dead on arrival. Not only was the Saturn just around the corner, the PlayStation was released one month after the 32X on December 3, 1994. By the end of 1995, inventory of the 32X was sadly liquidated at \$19.95 per unit.

Looking back on this era and reading interviews gives a bitter feeling when you keep in mind that, up until that point in history, Sega had been a colossal competitor to Nintendo. It had a cool image which had taken five years to build⁷.

From this point it looks like the company made one bad decision after another. Sega’s final console, the Dreamcast released in 1998, ended up being widely popular but sales were not enough to save the hardware business. Sega abandoned the market to focus on programming games instead.

⁷During 1993, Sega was the biggest advertiser on MTV (source: “RetroGamer #77”).

Looking back over his time trying to ship the 32X on schedule, Scot Bayless provided insightful memories.

“ 32X games in the queue were effectively jammed into a box as fast as possible, which meant massive cutting of corners in every conceivable way. Even from the outset, designs of those games were deliberately conservative because of the time crunch. By the time they shipped they were even more conservative; they did nothing to show off what the hardware was capable of.

— Scot Bayless

”

Deeper issues rooted in Sega, Inc.'s culture seem to explain later mistakes.

“ The 32X is a great case study in two things:

First, messaging: your number one job in marketing is to establish the value proposition. Even with all the rushed hardware and late software, if Sega had been able to convince people that the 32X was really worth having, it might have had a chance to succeed. But we never did that; we never managed to explain to anyone in any credible way what was so unique and worthy about the 32X. The result is exactly what you'd expect: Sony ate our lunch.

Second: honesty; not in the legal sense, nor in the public sense, but internally. I remember when I arrived at Microsoft in 1998 I attended an executive orientation briefing on my first day. The VP who met with us said: 'The one thing we demand of every one of you guys is to say what you think.' That attitude was what kept Microsoft vibrant, healthy and successful for more than 20 years. Sega, by contrast, lacked that ruthless honesty. Nobody wanted to hurt anyone's feelings. Even when everybody knew the 32X and Saturn were way behind the power curve, nobody was willing to stand up and say so. And it wasn't just the hardware; during the same period, Sega published some of the oddest games it ever released. Games that were deeply flawed. Games that completely failed to connect. And all the while everyone was smiling and saying, 'Gosh, aren't we great?' I wasn't able to articulate all this at the time, but I know I felt it intuitively. I knew there was something wrong, that we were losing our way.

— Scot Bayless

”

6.2.1 Doom On 32X

If porting DOOM to Jaguar had been a tour-de-force, repeating the feat on a system even less powerful was to require nothing short of a miracle. Once again John Carmack invested himself totally in the project.

“

I spent weeks working with Id Software's John Carmack, who literally camped out at the Sega of America building in Redwood City trying to get Doom ported. That guy worked his ass off and he still had to cut a third of the levels to get it done in time.

What amazes me now is that with all that going on, nobody at Sega was willing to say "Wait a minute, what are we doing? Why don't we just stop?" Sega should have killed the 32X in the spring of 1994, but we didn't. We stormed the hill, and when we got to the top we realized it was the wrong damn hill.

Looking back now I'd say that really was the beginning of the end for Sega's credibility as a hardware company.

— Scot Bayless

”

To make the game fit in the 512KiB RAM of the 32X, even more features than the Jaguar version had to be cut. Another enemy, the Spectre, had to be removed. Additional poses of monsters were removed except for the ones facing the player. Since they could no longer face each other, monster infighting was also removed. There were no savegames; players instead manually selected the starting level instead. The cartridge only had enough room for seventeen heavily-edited maps. Since none of them had the BFG9000, the weapon was unavailable (but could be obtained using cheat codes).

There was also a significant problem of performance. Even with its twin SuperHs, the machine was unable to render at the original resolution.

“

I liked the 32X – it was basically two decent 32-bit processors (SH2) and a framebuffer, so you programmed like on a PC, but with SMP long before it was mainstream on PC. It was still pretty underpowered compared to even a 386, so resolution was low.

— John Carmack

”



Figure 6.5: E1M1's legendary entrance hall

In figure 6.5 notice how, like on the Jaguar, E1M1's blue floor texture had to be replaced with a brown one to limit RAM consumption. Likewise, the number of steps on the stairs was also lowered in order to reduce the number of visplanes generated.

The game ran at a resolution of 320x224 but the CPUs struggled so much that the active window was reduced to 128x144 (column-doubled to reach 256x144), leaving 100 vertical pixels for the status bar and the brown border/background. With all these compromises the framerate managed to reach the 15-20 FPS⁸ range, which gave a pleasant experience.

Trivia : Sega named all its projects after planets of the solar system. Besides Saturn and Mars, two others are known. Neptune was a two-in-one Genesis and 32X console that Sega planned to release in the fall of 1995. It was canceled because of fears that it would dilute their marketing for the Saturn while being priced too close to the Saturn to be a viable competitor. Jupiter is rumored to have been a Saturn without a CD drive.

⁸Source: Digital Foundry YouTube channel, "DF Retro: Doom - Every Console Port Tested and Analysed!".



Map complexity was heavily reduced. Above, E1M1's main room was stripped of many textures (compare to the PC version on page 226). Below, the "pit" of E1M3 was flattened.



6.3 Super Nintendo (1995)

The Super Nintendo Entertainment System was released in 1990 in Japan and the following year in USA and Europe. It was the 16-bit successor to the 8-bit NES. In Japan, the Super Fami-Com ("FAMILY COMputer") was an instant success and the initial shipment of 300,000 units sold out within hours. The frenzy was such that the government requested Nintendo to release its future systems on weekends to avoid further disturbances.



Nintendo had established a merciless system to ensure quality of its games. Publishers were only allowed five games per year. To make sure this rule was enforced, only Nintendo was allowed to produce cartridges; publishers had to buy them from Nintendo. To make sure everybody played by the rules (and also to protect games from being copied) the SNES looked for a CIC lockout chip before a game was allowed to start. It was a powerful mechanism only cracked after the SNES reached its end of life.

During its nine-year lifespan⁹, 721 games were published, among them several critical and commercial successes such as Super Mario World, Zelda III, Mario Kart, F-Zero, Super Metroid, and Donkey Kong Country. Having sold close to 50 million units it is arguably one of the most popular consoles of all time¹⁰ both in terms of sales and catalog.



Figure 6.6: The Super Famicom (a.k.a SNES, Super Nintendo) by Nintendo.

From a technical standpoint, the SNES excelled at 2D. Its 16-bit 65C816 3.58 MHz CPU

⁹The Super Nintendo was discontinued in 1999.

¹⁰Source: "The SNES is the greatest console of all time" by Don Reisinger.

had 128 KiB RAM available. It piloted a PPU (Picture Processing Unit) with 64 KiB of RAM to manipulate large sprites, using up to 256 colors at a resolution of 256x240. On the audio side, the machine had the powerful combo of an 8-bit Sony SPC700 and a 16-bit DSP with 64 KiB of dedicated SRAM.

Despite its impressive 2D sprite engine and especially its "Mode 7" capability, the machine struggled with computationally-intensive operations such as 3D calculations. Nintendo was vividly aware that 3D would be the next big thing in gaming but was struggling to make it happen. As fate would have it, a small UK firm would hold the solution to the problem.

6.3.1 Argonaut Games

Back in 1982, Jez San was a lone game developer working exclusively on the C64, Atari ST, and Amiga computers. To sell his creations he needed a company. Seeing a similarity between his name (J.San) and the mythological story of Jason and the Argonauts, he named it Argonaut Games plc.

His venture did not remain a single-man project for long. By 1990 he had gathered talent in London offices and developed an interest in Nintendo's 1989 handset, the Game Boy. The team had managed two feats most deemed impossible: they had a 3D wireframe engine and they had cracked the CIC protection to install it on the Game Boy.

“

They had the Nintendo logo drop down from the top of the screen, and when it hit the middle the boot loader would check to see if it was in the right place.

The game would only start if the word was correctly in place in the ROM. If anyone wanted to produce a game without Nintendo's permission, they would be claiming to use the word 'Nintendo' without a licensed trademark, and therefore Nintendo would be in a position to sue them for trademark infringement. We figured out that with just a resistor and capacitor - around 1 cent's worth of components - we could find out how to beat the protection. The system read the word 'Nintendo' twice - once to print it on the screen at the boot up, and a second time to check if it was correct before starting the game cartridge. That was a fatal mistake, because the first time they read 'Nintendo' we got it to return 'Argonaut', so that was what dropped down the screen. On the second check, our resistor and capacitor powered up so the correct word 'Nintendo' was in there, and the game booted up perfectly.

— Jez San¹¹

”

¹¹Source: Jez San interview with Damien McFerran for "Born sloppy: the making of Star Fox" article.

At CES '90, his demo to the Nintendo booth of the engine on a hacked cartridge was relayed all the way up to the Kyoto headquarters. Unknown to Jez at the time, his timing could not have been better. Back in Japan, Nintendo was working on Super Famicom titles intended to showcase its superior technology upon launch. Super Mario World was in its infancy but the flight simulator "Pilotwings", was a bit more advanced.

The SNES PPU's Mode 7 (capable of affine transformation such as rotation, scaling, and shearing) was used along with the HDMA mode to simulate the terrain in Pilotwings. The planes however were still 2D hand-drawn sprites. This bugged producer Shigeru Miyamoto since it prevented the camera from smoothly rotating around the planes (quantized sprites were jaggy).



At the time, it was not in Nintendo's habit to deal with outsiders or even foreigners. This time they made an exception and flew Jez to their headquarters in Kyoto, along with Dylan Cuthbert who had done the 3D work.

The young pair¹² met all of Nintendo's VPs: Miyamoto, Gunpei Yokoi, Takehiro Izushi, Yasuhiro Minagawa and Genyo Takeda. They were shown everything from the secret SNES to the secret Mario/Pilotwings. Then they were asked if there was a way to draw the planes as full-faced polygon objects.

“

I told them that this is as good as it's going to get unless they let us design some hardware to make the SNES better at 3D. Amazingly, even though I had never done any hardware before, they said YES, and gave me a million bucks to make it happen.

— Jez San

”

Boldly promised a "10x" performance increase by Jez, Nintendo embraced the offer to get special hardware designed for their game. "Pilotwings" would ship with sprite planes so that it could be ready for the Super Famicom release, but the "Super FX" chip as it would be marketed later was to be used for another project Nintendo had up its sleeves.

The name was "Star Fox".

¹²Jez was 23 and Dylan 18.

6.3.1.1 Star Fox

The agreement was such that Nintendo would make all game design decisions while financing Argonaut Games to produce not only the hardware but also the 3D engine for the title. Jez-san lost no time hiring and contracting the best UK talent he knew.

For the hardware he contacted Flare Technology (the same people who designed the Atari Jaguar). Ben Cheese, Rob Macaulay, and James Hakewill's project was codenamed Mathematical Argonaut Rotation I/O, or "MARIO". What they ended up designing was so powerful they jokingly labeled the Super NES "just a box to hold the chip". Since there was no way to modify the console, the chip was soldered onto each new game cartridge which increased MSRP significantly.

“

We designed the Super FX chip in a way no one had designed hardware before - we built the software first, and designed our own instruction set to run our software as optimally as possible. No one did it that way around! Instead of designing a 3D chip, we actually designed a full RISC microprocessor that had math and pixel rendering functions, and the rest was run in software. It was the world's first Graphics Processing Unit, and we have the patents to prove it.

— Jez San

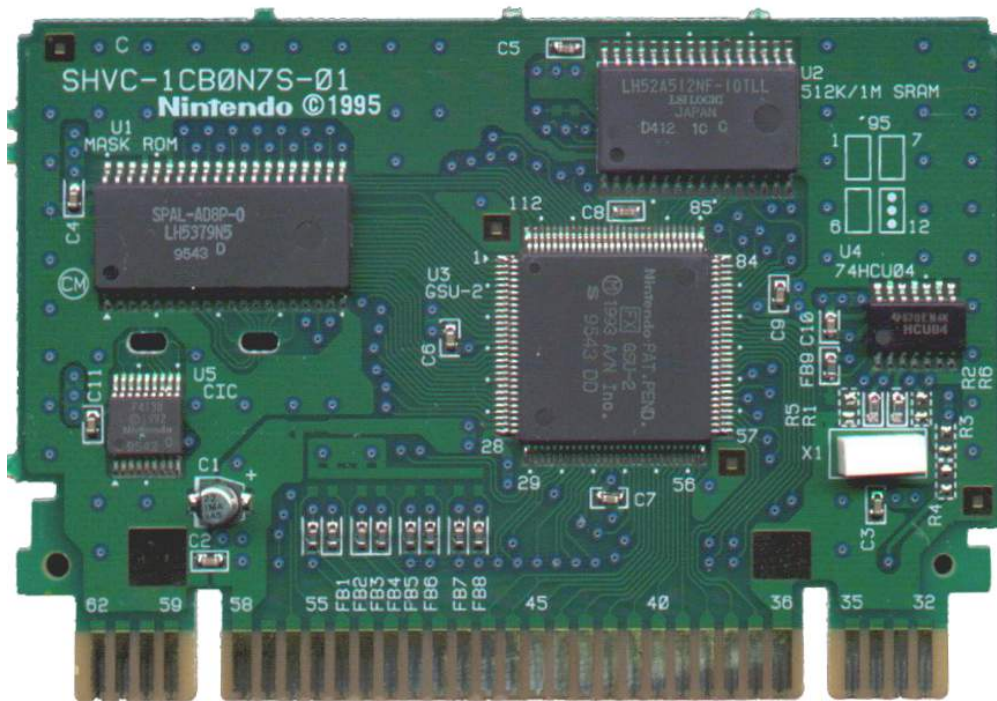
”

For the engine, Carl Graham and Pete Warnes worked in the London headquarters while Dylan Cuthbert, Krister Wombell and Giles Goddard (plus later Colin Reed) permanently relocated to Kyoto in Nintendo's offices to work in close collaboration with Miyamoto's team.

The project resulted in a critical, commercial, and engineering success. Star Fox shipped on February 21, 1993 and went on to sell four million copies worldwide

The rest of this idyllic story between the two companies is bitter. Star Fox 2, the sequel to their megahit, was completed by Argonauts and set to release in 1996 when Nintendo canceled it abruptly, fearing its impact on the launch of the Nintendo 64. Argonaut was not pleased and the relationship with Nintendo soured. Nintendo subsequently poached Goddard and Wombell. Dylan Cuthbert would have joined too, but he was prevented from doing so by a non-compete clause in his contract. He quit his position at Argonaut and joined Sony to work on the Playstation.

The divorce was finalized when Nintendo refused to let Argonaut use Yoshi for a platform game they were planning for the PS1. They ended up replacing Yoshi with a crocodile in "Croc: Legend of the Gobbos". Nintendo later released Mario 64 with mechanism seemingly inspired by "Croc" ... and even beat it to market by around a year.



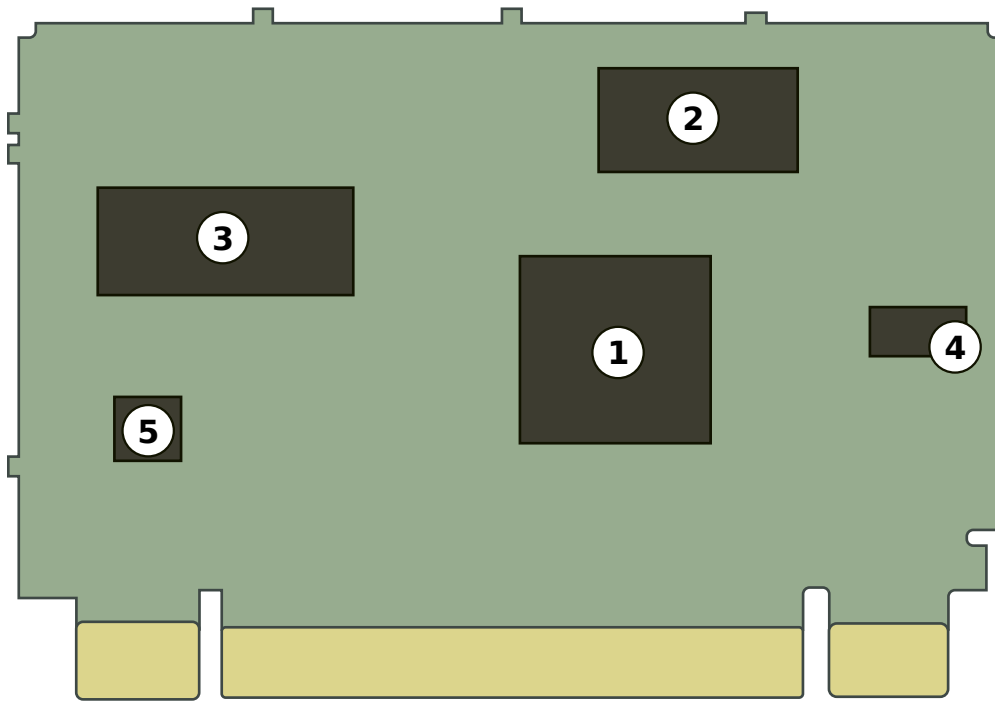
The MARIO chip had a simple design based on a 16-bit RISC processor running at 10.74 MHz with a 512 byte i-cache. It had its own instruction set optimized for math and its own framebuffer optimized for pixel plotting. Its mode of operation was to render to the frame buffer where the data would be periodically transfered to the SNES RAM via DMA. It was reportedly capable of rendering 76,458 polygons/s which meant about 15 fps for Starfox.

Upon witnessing Starfox's phenomenal success, other studios became interested in the technology. The chip was revised to be able to run at 21.4 Mhz and it was renamed "GSU"¹³. The first generation of GSUs powered four games: Dirt Racer, Dirt Trax FX, Stunt Race FX, and Vortex.

The second generation (GSU-2) was the same processor running at 21.4 Mhz with extra pins soldered to the bus to increase the size of supported ROM and framebuffer. It was used in three games: DOOM, Super Mario World 2: Yoshi's Island, and Winter Gold.

Opening a DOOM game cartridge reveals all the components previously discussed. ① The 16-bit GSU-2, ② 512 KiB framebuffer where the GSU writes, ③ 2MiB ROM where code and assets are stored, ④ Hex inverter, and ⑤ Copy protection CIC chip.

¹³Graphics Support Unit



“

The 'ten times' figure was a complete over-promise on my part. We didn't really know if that was even possible.

But it allowed us to over-promise and yet also over-deliver. Instead of achieving just 10x the 3D graphics performance, we actually made things about 40x times faster. In some areas - like 3D math - it was more like a 100x faster. It was not only capable of 3D math and vector graphics, but it was also able to do sprite rotation and scaling - something that Nintendo really wanted for their own games, like Super Mario World 2: Yoshi's Island.

— Jez San

”

Trivia : Some passionate fans have managed to collect all 791 games from the SNES catalog. Seeing them on a shelf is impressive. You can usually spot a DOOM cartridge from 20 feet away. Only three games were ever allowed not to be made in the standard gray. Two were red – DOOM and "Maximum Carnage" – while "Killer Instinct" was black.

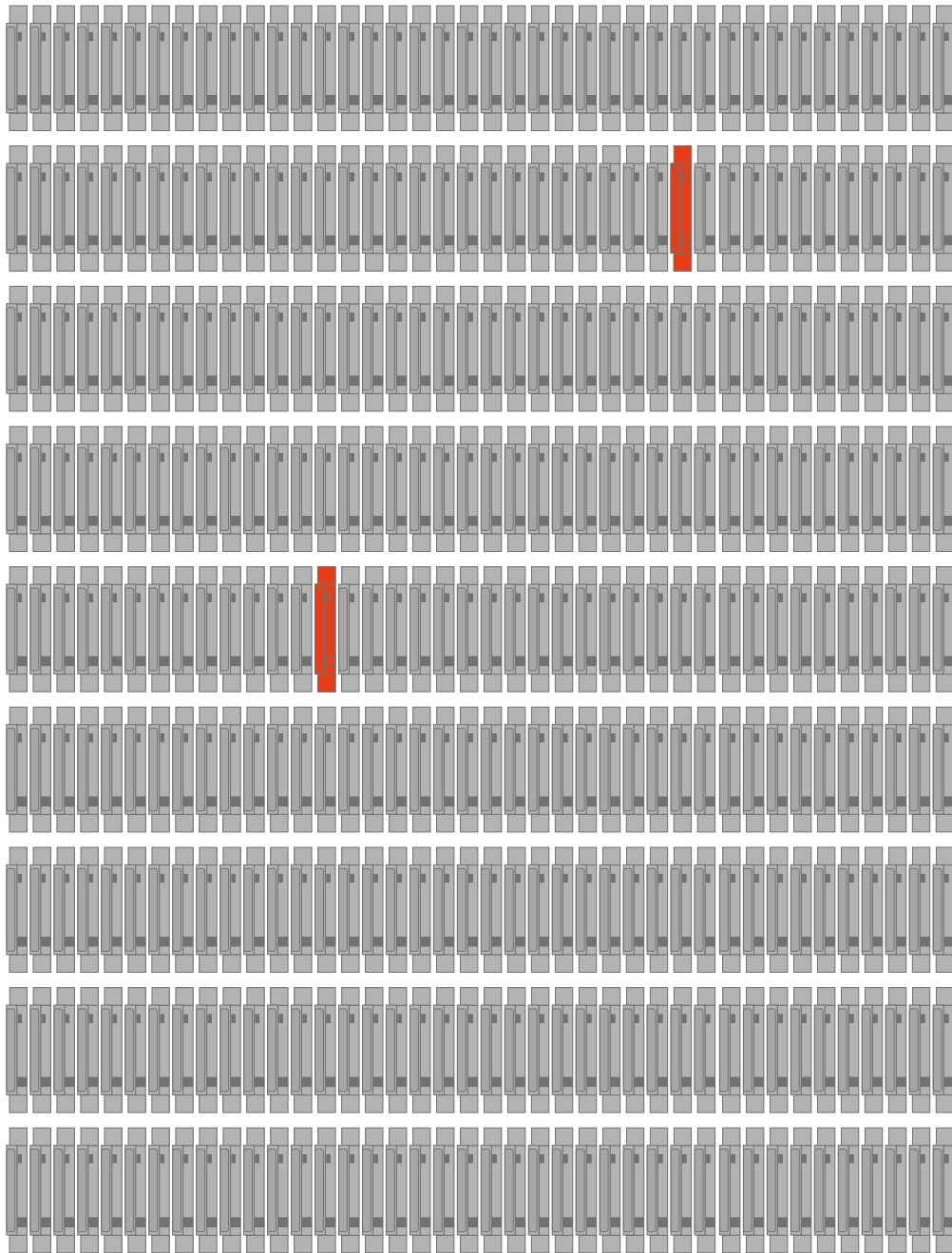
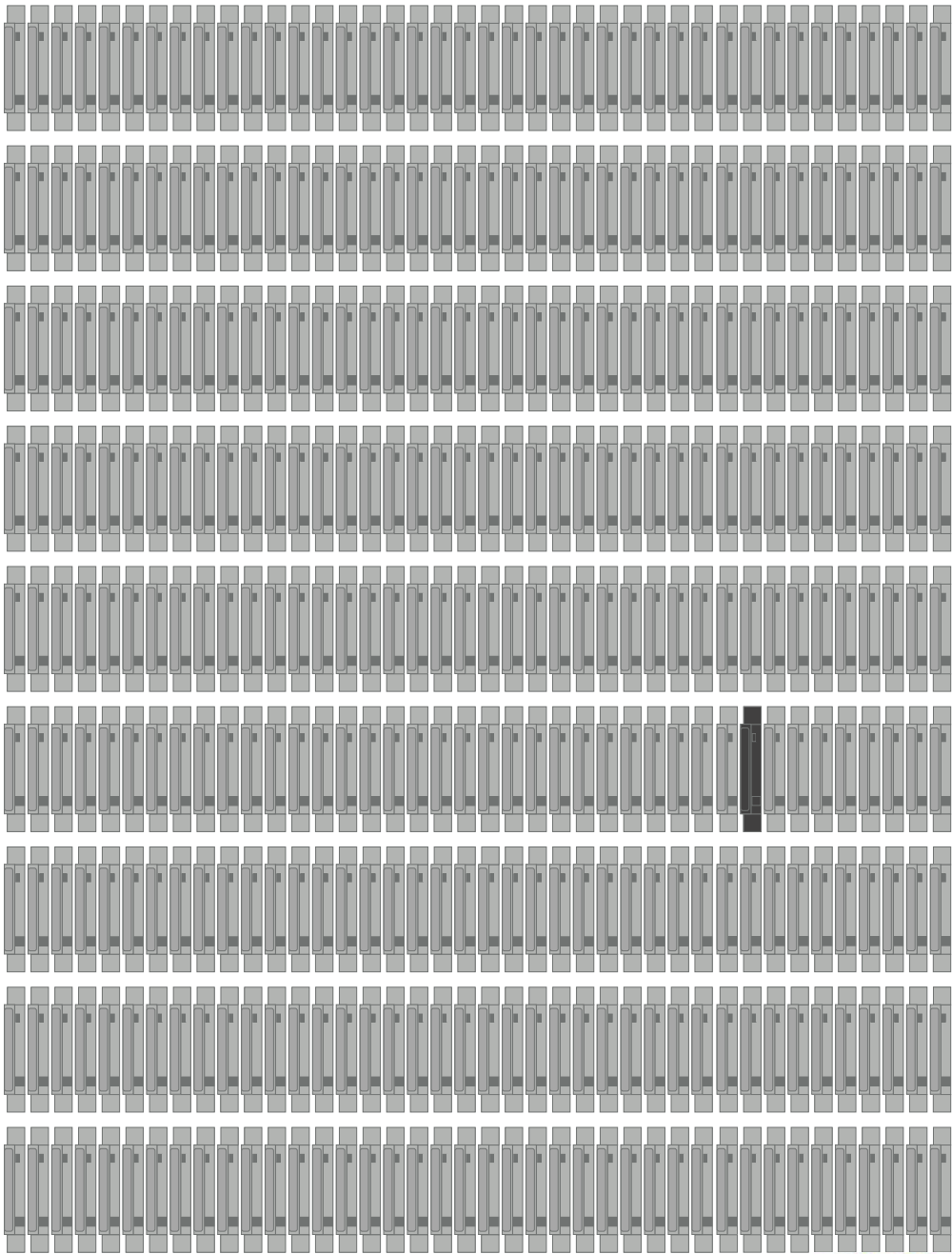


Figure 6.7: SNES 721 game library. Zelda stands apart. Because Zelda rules.



6.3.2 Doom On Super Nintendo

DOOM on SNES happened thanks to the genius and determination of a single man: Randy Linden. The man had an admiration for the game and decided to port it to a mass-market machine so more players could enjoy it. Randy never had access to the source code or the assets from either the PC or the console version. He started from nothing.

To retrieve the assets, he was able to leverage the "Unofficial Doom Specs" by Matthew Fell which explained the .wad lump layout in detail. The sprites, textures, music, sound effects and maps were extracted from DOOM.WAD. The engine was an entirely different story.

“

DOOM was a truly ground-breaking title and I wanted to make it possible for gamers without a PC to play the game, too. DOOM on the SNES was another one of those programming challenges that I knew could be accomplished.

I started the project independently and demo'd it to Sculptured Software when I had a fully operational prototype running. A bunch of people at Sculptured helped complete the game so it could be released in time for the holidays.

The development was challenging for a few reasons, notably there were no development systems for the SuperFX chip at the time. I wrote a complete set of tools – assembler, linker and debugger – before I could even start on the game itself.

The development hardware was a hacked-up Star Fox cartridge (because it included the SuperFX chip) and a modified pair of game controllers that were plugged into both SNES ports and connected to the Amiga's parallel port. A serial protocol was used to communicate between the two for downloading code, setting breakpoints, inspecting memory, etc.

I wish there could have been more levels but unfortunately the game used the largest capacity ROM available and filled it almost completely. I vaguely recall there were roughly 16 bytes free, so there wasn't any more space available anyway! However, I did manage to include support for the SuperScope, Mouse and XBand modem! – Yes, you could actually play against someone online!

— Randy Linden (Interview with gamingreinvented.com)

”

Trivia : Randy Linden later worked on an even more impressive reverse-engineering project. In 1999 he was co-author of a commercial PlayStation emulator called "Bleem!" – a gutsy move since the console was still being produced and sold by Sony at the time.

What is remarkable with this version is how Randy, with his engine capabilities and restrictions, had to cut different corners to other console ports.



Figure 6.8: SNES starting screen on map E1M1

In figure 6.8, despite having only 600 KiB RAM, see how the blue flooring remained (although as a plain color). Notice how the geometry was not altered¹⁴, E1M1's starting room has all its original steps (compare with the PC on page 223 and the Jaguar on page 305).

The Reality engine, as Randy named it, was able to deal with PC map geometry but must have had issues with fill rate or texture sampling because ceiling and floor textures were removed altogether.

¹⁴It would have been extremely difficult to alter the geometry since Randy had no access to DoomED or doombsp.



Figure 6.9: E1M1 exterior toxic pond

In the screenshot above we can see the window is not actually fullscreen. This was not an issue exclusive to DOOM since *Star Fox*, *Star Fox 2* and all games using the Super FX had to reduce the size of the active area. It was likely due to the SNES's limited bandwidth which did not permit a DMA transfer for fullscreen rendering¹⁵.

Out of the native 256x224 resolution, only 216x176 was actually drawn and only 216x144 for the 3D canvas (32 rows for the status bar). With vertical lines duplicated, the Reality Engine was actually rendering at 108x144. Even at this low resolution, the average framerate was around 10 FPS which was a remarkable achievement. The "low" framerate was not enough to discourage players from enjoying DOOM. According to Randy Linden the game sold very well.

¹⁵Although some like anthrofox.org theorized that the Super FX cannot render more than 192 lines.



Figure 6.10: E1M3, notice the dithering on the floor.

Amazingly, Reality was able to implement diminished lighting for both walls and floors using a dithering technique as the large floor "shade" in figure 6.10 demonstrates.

On the list of features sacrificed to the holy RAM, sprite resolution was lowered significantly to the point that they were sometimes hard to recognize (contrasting with the player weapon rendered at a higher resolution). Enemy poses were all removed except for the ones facing the player, monster infighting was also removed, there is no sound propagation (monsters are only awoken by visual contact), most SFX were cut and all monsters sound like imps.

Trivia : Nintendo originally forbade blood in SNES games. By the time DOOM came out, ESRB had come into the picture. Given the amount of blood and the pieces of flesh found behind every corner, Doom SNES understandably received an M rating.

6.4 Playstation 1 (1995)

The history of the PlayStation started in 1988 when Nintendo collaborated with Sony to produce a CD-ROM reader add-on for the SNES. Under the terms of the contract, Sony could develop independently for the platform and retained control over the "Super Disc" format – two unusual concessions on Nintendo's part.

The project moved forward until CES '91 when Sony announced the joint venture called "Play Station". The next day, during the same event, Nintendo announced it had instead partnered with Philips (with much more advantageous terms) much to Sony's surprise. Betrayed and publicly humiliated, Sony attempted to turn to Sega's Board of Directors who promptly vetoed the idea. In a 2013 interview then-Sega CEO Tom Kalinske remembered the board's conclusion.



“ That's a stupid idea, Sony doesn't know how to make hardware. They don't know how to make software either. Why would we want to do this? ”

They were not wrong. Sony had little experience with gaming. It also had almost no interest in trying either, since most of its involvement so far had relied on one man, Ken Kutaragi. Ever since he had witnessed his daughter play on a Nintendo Famicom, Ken had been advocating for Sony to enter the market. He had even designed Nintendo's audio chip (the SPC700) for their SNES against the advice of Sony VPs.

Despite being considered a risky gamble by other Sony executives, Kutaragi gained the support of Sony CEO Norio Ohga. In June 1992 Ken got the green light to build a gaming system from scratch¹⁶. The "Father of the PlayStation" as he would later be called had to be transferred to the financially separate Sony Music to appease the board but he could set himself to work on what would become the "PlayStation" (without space).

There was originally some uncertainty about the architecture which could focus either on 2D sprite graphics or 3D polygon graphics. The success of Sega's Oct 1993 Virtua Fighter in Japanese arcades cleared all doubts¹⁷: the PSX was going the 3D route.

The project would culminate two years later with the creation of Sony Computer Entertainment and a Japanese release on December 3, 1994. It was an instant success selling

¹⁶Playstation: Anthology by GREEKS-LINE.

¹⁷Source: "How Virtua Fighter Saved PlayStation's Bacon". WIRED. Sept 2012.

100,000 units on day one, 2 million after six months, and 102 million units over its lifetime.



Figure 6.11: Sony PlayStation

6.4.0.1 Keys to success

Among the numerous good choices, Sony listened to developer feedback and bumped the RAM from 1 to 2 MiB. They adopted a developer-centric attitude where the development cycle was easy, tools updated frequently and downloadable online, with third party technical support. The CD format allowed games to be priced lower and developers did not have to buy cartridges from Sony.

More importantly, Sony did not censor developers the way Nintendo and Sega did. On top of it all, royalties were lower which improved profitability¹⁸. "The PlayStation set us free" Kalisto Gaming's CEO would later testify.

Sony's stroke of luck was to land Psy-Q, which made the PSX's SDK a programmer's dream. Psygnosis was a UK game company working on the Atari ST, Amiga and SNES. Sony purchased them in early 1993 and tasked them with creating the then-still-secret PlayStation games *Wipeout* and *Destruction Derby* to showcase the PlayStation on launch.

Until then, Sony had envisioned development for the PlayStation as being based on dedicated Sony NEWS MW.2 workstations¹⁹. These were colossal, expensive machines based on the MIPS R4000 and manufactured by Sony. Psygnosis disliked that solution, especially when they compared the development experience to their regular tool (Psy-Q) which was produced by a company called SN Systems.

¹⁸Nintendo could sometimes take up to 20%.

¹⁹"The development system", Next Generation June 1995.

Around Christmas '93, SN Systems co-owner Andy Beveridge and Martin Day were given an MW.2 by Psygnosis with a request: Make Psy-Q run on it! The pair worked around the clock and managed to port the GNU toolchain (cc compiler, ld linker, lib builder ar, and gdb debugger) to a PC connected to a Sony's MW.2 box and demoed it at CES Las Vegas in early 1994.



Sony loved it and promptly ordered 700 dev kits. At the end of Spring 1994, the devkit hardware was shrunk to two ISA cards (called DTL-H2000) connected to a SCSI drive so there was no need to burn CDs for testing.

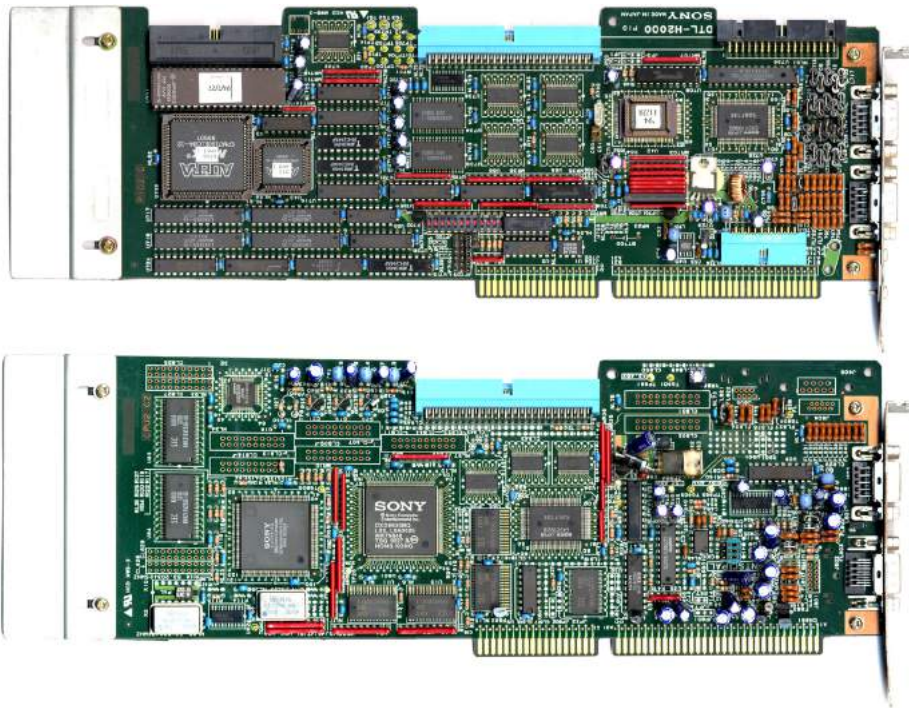


Figure 6.12: DTL-H2000 2x ISA card devkit. SCSI HDD and CD-ROM burner not shown

Leveraging PCs not only significantly reduced the financial cost to become a developer, it also lowered the barrier to entry since most developers were already familiar with Windows.

From September 1993 to June 1995, 500 licensees worldwide jumped on the opportunity to publish on Sony's dream machine²⁰. Devs bought both the PSX and its devkit.

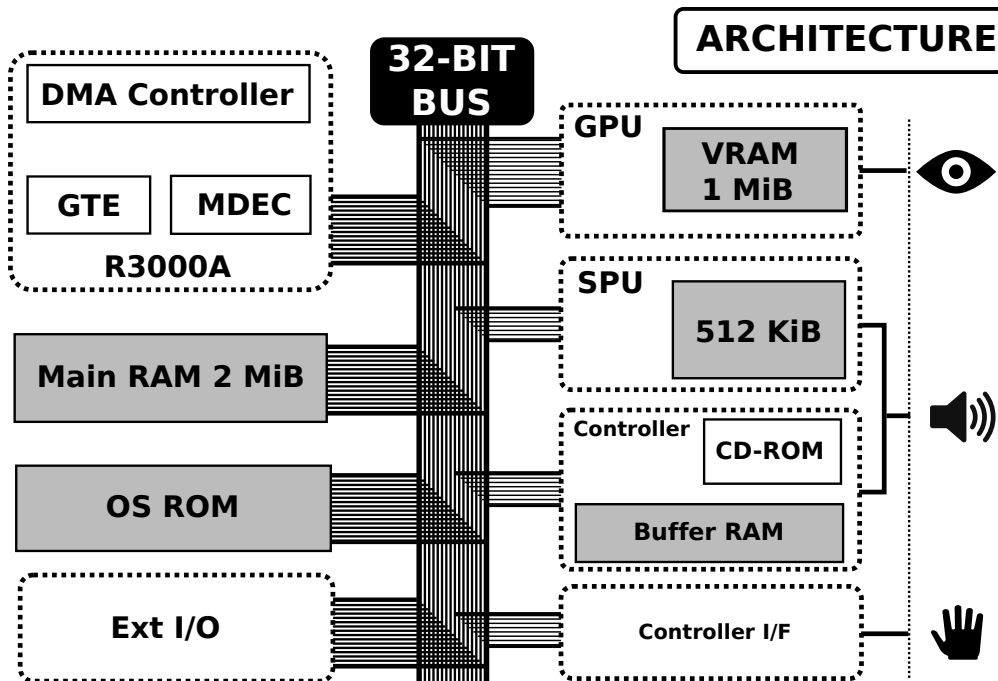
²⁰"Sony's PlayStation game plan", Next Generation June 1995.

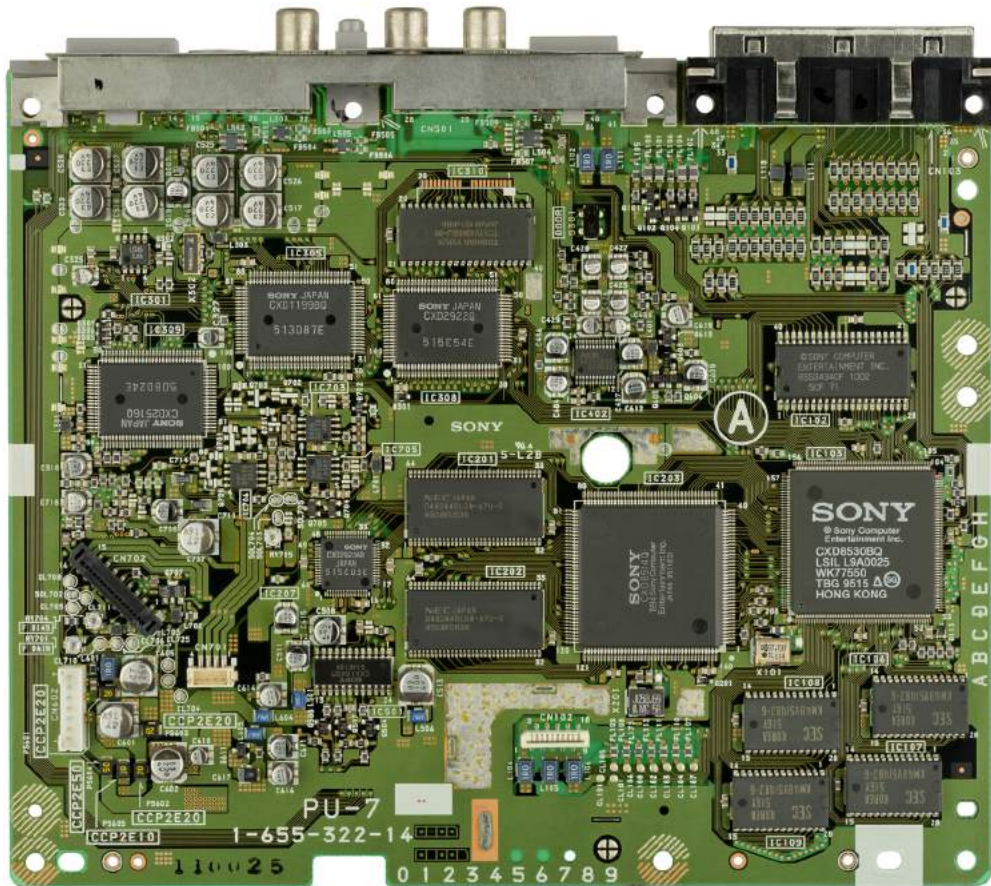
Programming the PlayStation was an unbelievably pleasant experience. Most programming was done in C while allowing handcrafted assembly if necessary. Psy-Q provided a compiler driver able to take a list of `.c/.obj` files and output a PlayStation executable in one keystroke.

The PSX programming philosophy was to not make developers juggle with multiple systems. The 1MiB video framebuffer for example could not be accessed by the programmer directly, delegation to the GPU was mandatory. This sample from the PSX Developer Tools summarizes well how much care was taken to lift the burden on programmers.

“ The CPU is only involved in giving the dedicated hardware very small amounts of data such as the display location and the start address for data transmission. Data is transferred via the DMA Controller and consumed by the GPU. The result of this parallel processing is that the CPU can devote almost all of its time to creating drawing command lists.

— PlayStation, Net Yaroze Manual

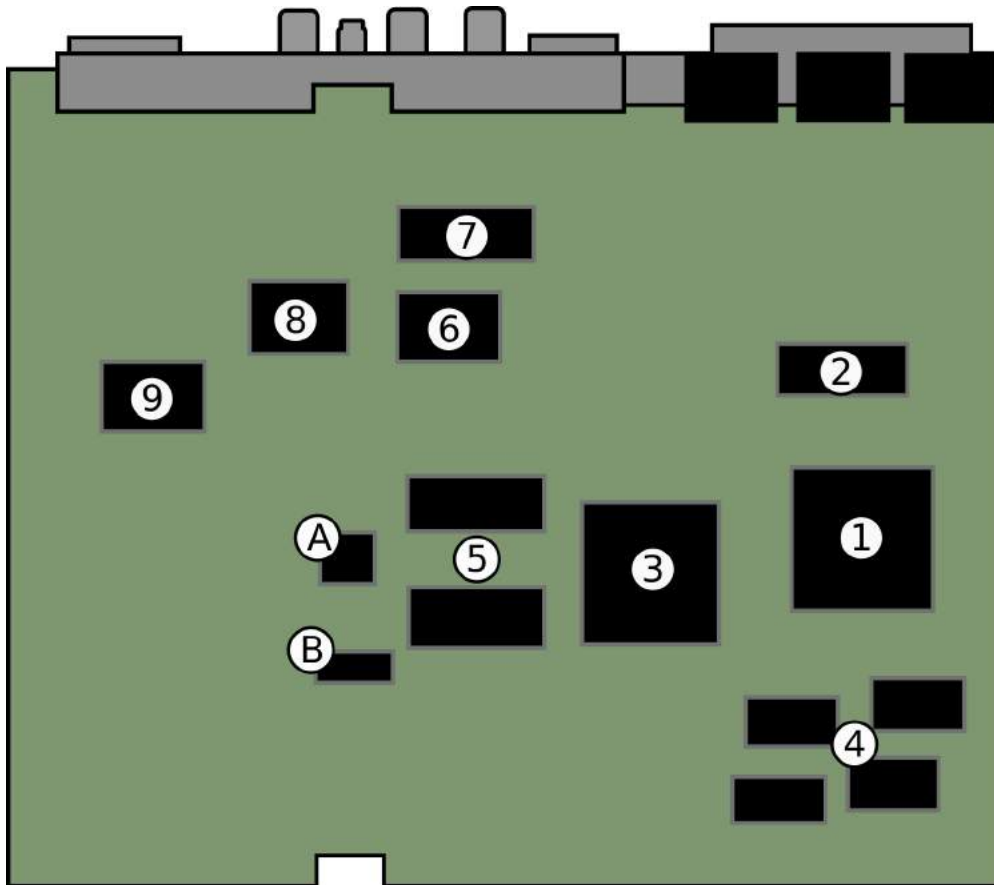




Opening a PlayStation and taking a look at the motherboard reveals no fewer than fifteen chips²¹

①, 32-bit 33MHz R3000 CPU (30 Mips) with 4KiB i-cache and 1KiB d-cache. Also contains the 88 Mips Geometry Transfer Engine (GTE), the DMA Controller and Sony's 80 Mips MDEC video decompression hardware. ② Operating System ROM. ③ GPU. ④ 2 MiB RAM. ⑤ 1 MiB VRAM. ⑥ DSP. ⑦ 512 KiB DSP RAM. ⑧ CD Controller: Contains a CD ROM-XA converter (allowing up to eight simultaneous streams of mixed audio and CD data) and a small amount of buffer RAM. ⑨ CD-drive DSP. A 16-bit video digital converter. B Video decoder and encoder (NTSC or PAL) to TV.

²¹Source: NEXT Generation Issue #6 June 1995, "Inside the Playstation".



It was initially difficult to convince developers to get on board and work with the PlayStation. On October 27, 1997, Sony gathered 300 developers representing 60 games publishers for a tour-de-force. They were shown the "dino demo"²² that featured a real-time controllable T-rex dinosaur.



The demo ran at 50 frames per second at a resolution of 512x256. It processed about 1800 polygons per frame, and drew up to 1300 polygons per frame. Jurassic Park, released in 1993, was still fresh in peoples' memory as a monumental engineering achievement. The breathtaking sight spread rapidly in the gaming community and orders for the SDK soon skyrocketed.

²²Source: "PlayStation: Anthology"

6.4.1 Doom on PlayStation

DOOM was ported to the PSX by Williams Entertainment. It took a little bit less than a year for a team of five²³ to port the engine, change the assets, and make everything work with "only" 3 MiB of RAM. The final result is universally considered the best console port with some aspects even outmatching the PC version.

Work did not start from scratch. The team leveraged work from the Jaguar version and in particular the simplified maps using fewer textures and walls.

“ The graphics were reduced: the textures chopped down in size, the sprites, monsters, and weapons reduced in size. [...] Sometimes animations had frames cut.

— **Harry Teasley**

”

The restrictions did not have to be as drastic as for Atari's console. Thanks to the CD-ROM capacity, 59 maps (33 from DOOM and 26 from DOOM II) shipped. To compensate for the slow access time and the restricted amount of RAM, each map is stored in its own WAD archive. On the other hand, most monsters are present except for the Arch-Vile.

“ The archvile had twice as many frames of animation as any other monster, and we just couldn't do him justice on the PSX. Couldn't lose his attack, and couldn't lose his resurrecting power. He was just too big to include.

— **Harry Teasley (Designer)** for doomworld.com

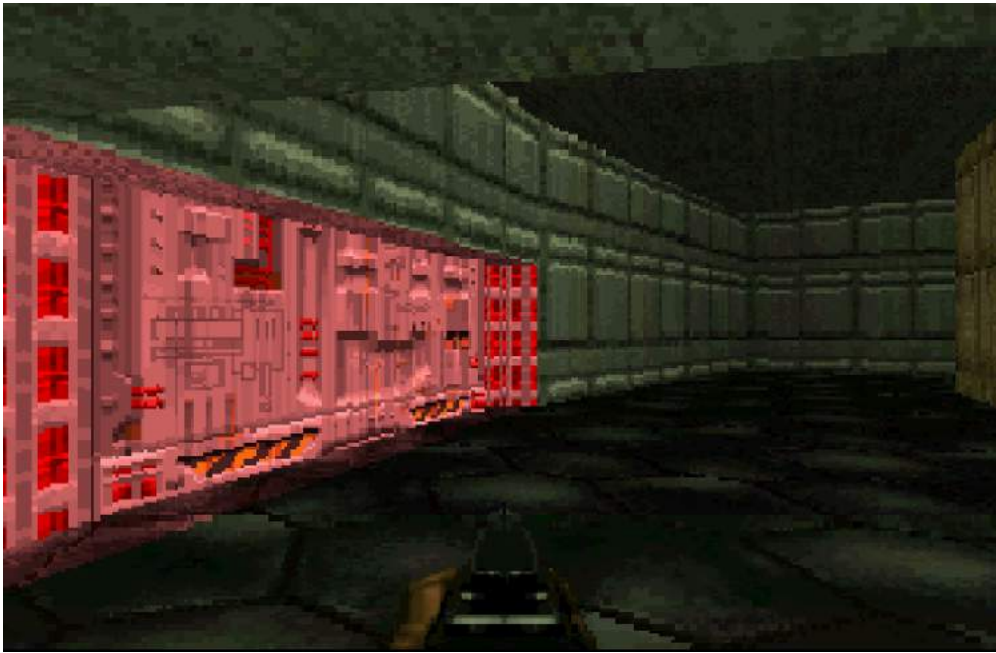
”

What may come as a surprise are the improvements over the PC version.

Sound was improved thanks to the SPU processor to render reverberation in small rooms. The spectre – which faked translucency with a Predator-like "shimmering" – was converted to subtractive blending. Musics were brought up to CD standard (44KHz, 16-bit, stereo).

The most impressive addition was the 16-bit colored lighting achieved by adding a color to sectors and 50/50 blending all textures with it. In some cases this feature was used to improve game mechanisms, as seen on the next page where a red light indicates a door that requires the red key. To perfect the illusion, the player hand is colored accordingly.

²³Three designers/artists and two programmers.



There were many other subtle additions such as animated skies. In the following screen-shot, the marine is in Hell and the sky contains gorgeous flame effects.



In many aspects, this port embodied the bittersweet deal of hardware rendering over software rendering.

On the one hand, the hardware acceleration allowed more complex worlds with many polygons and textured-mapped models with less code written. New effects such as the Nightmare Spectre subtractively blended against the background were added.

On the other hand, the freedom of software rendering allowed innovative tricks which were now impossible to achieve. A prime example is the red palette shift when damage occurs which could not be done and is absent from the PlayStation version. Another example is that sprites and textures had to have power of two dimensions to improve texture sampling (a 64×64 texture lookup at coordinate (u, v) can be optimized as $(u \ll 6) + v$).

In this instance the bitterness was taken to an extreme. It seems the technical difficulties were so considerable that some of the developers on the team doubted it could be done.

“

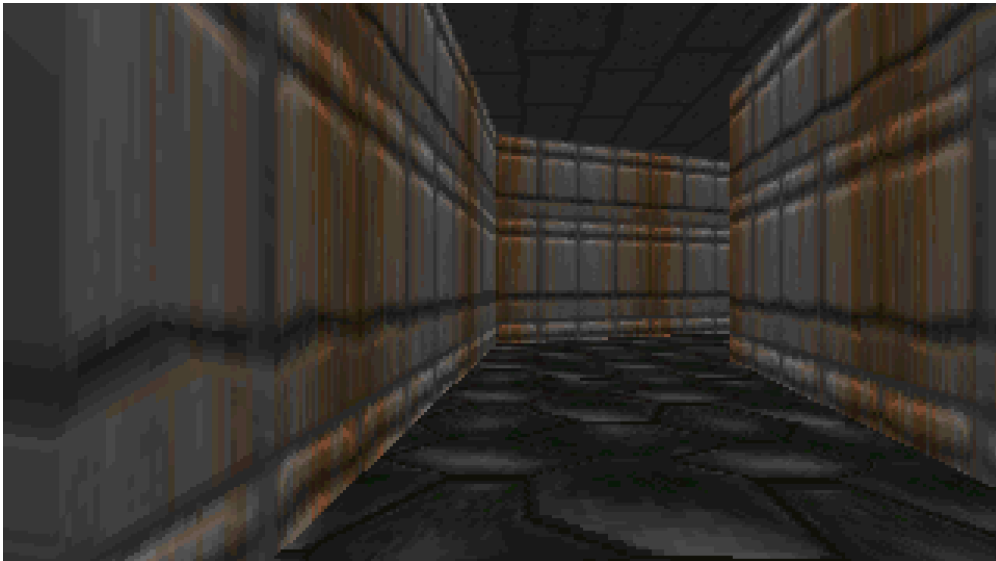
I worked with Aaron Seeler on the Nintendo 64 (which was a fairly different game) and Playstation versions. Those were the first versions that weren't written "to the metal", since both Sony and Nintendo were forcing (at least third party) developers to write to API instead of just handing them hardware register documentation. The SGI culture in particular cramped developers at the start, but Nintendo eventually walked it back a bit.

Funny story on Playstation development: Aaron and I started out with a different engine architecture that rendered the world with triangles, since they were fully hardware accelerated. That worked great on the N64, which had subpixel accurate, perspective correct rendering (that SGI influence), but Playstation had integer coordinate, affine texture mapping, and the big wall and floor triangles looked HORRIBLE.

— John Carmack

”

Affine Texture Mapping is the process of performing texture mapping in screen space without taking perspective into account. Thanks to user Lollie from doomworld.com we can take a look at what DOOM could have looked like with improper texturing.



The issue is particularly visible on the left wall where the black strip is not parallel to the ground any more but seems to zig-zag up and down.

From the previous screenshot we can see that something is wrong but it is not clear what. There is obviously a distortion but it is not exactly what we saw on page 218. The difference is that we were able to draw a quad directly whereas the Playstation's GPU is only capable of processing triangles.

Without the ability to draw quads, developers had to express everything as triangles. To draw a wall they had to place two triangles next to each other.

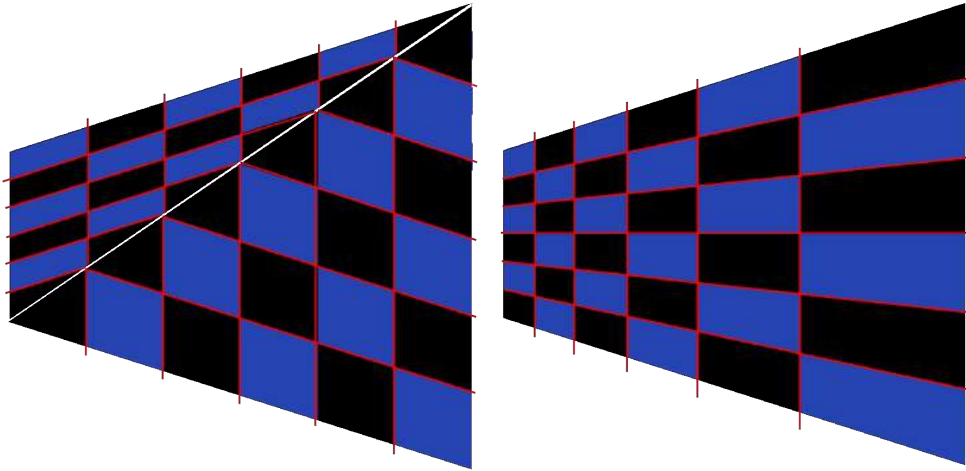


Figure 6.13: Left: Affine texturing. Right: Perspective texturing

In Figure 6.13, the left wall shows how the PSX received two triangles and performed screen space affine texturing without factoring in the distance from the point of view. The result differs significantly from what it should have been as the right wall shows.

To rasterize these triangles the GPU has no choice but to use a scanline algorithm. The process preserves line parallelism and moreover there seems to be no "agreement" between the two triangles resulting in an unpleasant "zigzag".

In figure 6.14, the visual artifacts become exacerbated as the angle of the wall increases. Also notice in the right column how the width of each square is always constant, a giveaway of affine texturing that contrasts with the perspective correct decreasing width seen in the left column.

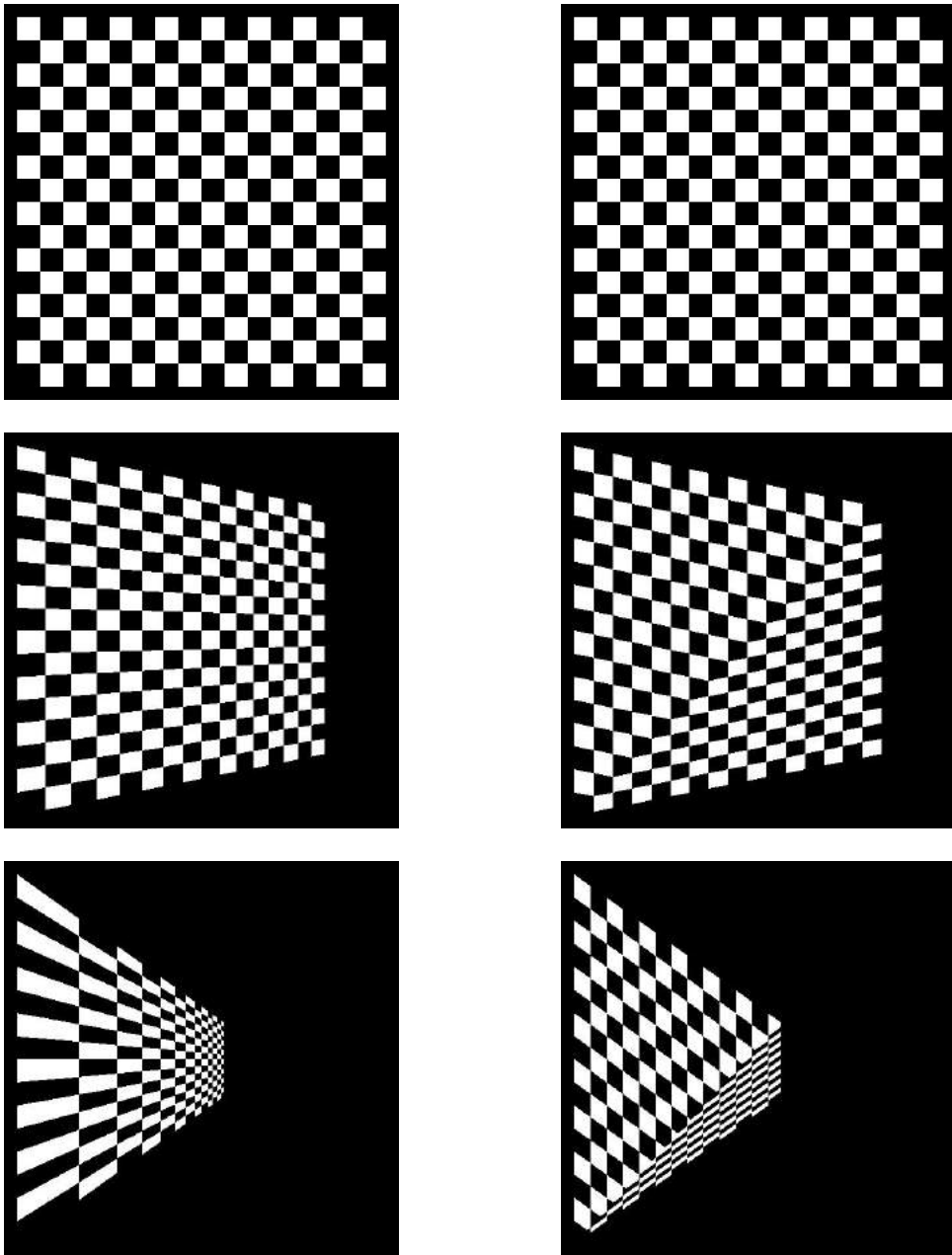


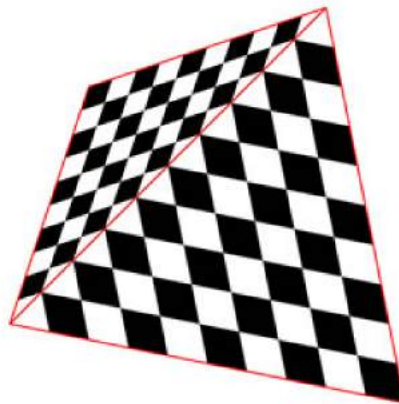
Figure 6.14: Perspective-correct texturing (left) vs affine texture mapping (right)

Sony was well aware of the problem but manufacturing cost prevented gifting the PlayStation with perspective correct hardware (they also lacked a partner with strong computer graphics experience such as SGI, who deeply influenced the Nintendo 64). The PSX's developer manual's recommended way to mitigate the problem was to subdivide triangles into more triangles.

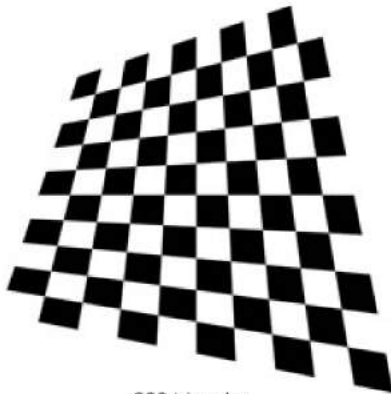
That may sound like running away from the problem but the PSX was capable of processing a rather high number of triangles for its time so it was not a bad suggestion. To get a satisfactory visual result however, the number of triangles had to be high.



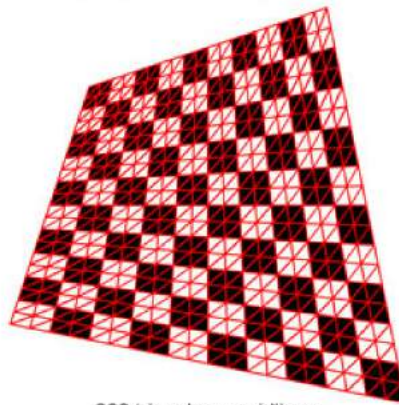
2 triangles (no subdivision)



2 triangles (no subdivision) w gridlines



800 triangles



800 triangles w gridlines

With this perspective issue, Williams Entertainment, John, and Harry had a big problem and some were really nervous about it.

“

Aaron was always a big ball of stress on the projects we worked together on, and this abject failure of the plan of record was giving him panicky visions of project failure. I sort of shrugged and said "back everything up (no source control back then!), we're going to do something completely different".

We wound up using the hardware to render triangles that were one pixel wide columns or rows, just like the PC asm code, and it worked well. The more common Playstation approach turned out to be tessellating geometry in two axis, but I was always pretty happy with how Doom felt less "wiggly" than most other Playstation games of the time.

— **John Carmack**

”

Switching from world-space triangles to screen-space pixel-wide triangles did the trick. The engine ran at 30Hz with game logic at 15Hz. The graphical result was exactly like the PC version. Running at a resolution of 256 x 240²⁴, the engine managed an impressive 20-30²⁵ frames per second in most instances.

DOOM on PSX achieved both critical and commercial success.

“

PlayStation version succeeded in "putting previous efforts for 32X, Jaguar, and especially Super NES, to shame.

— **Next Generation, 1995**

”

Even members of id Software stepped forward to assert the quality of the port.

“

This is the best DOOM yet!

— **John Romero**

”

Trivia : If the player took enough damage to become gibbed, the status bar head reflected the result in gory detail – not something that would have flown with Nintendo!

²⁴One of the lowest resolutions used by a game; even Ridge Racer rendered at a higher resolution.

²⁵Source: Digital foundry.

6.5 3DO (1996)

The 3DO Company was founded in 1991 by ex-Electronic Arts and ex-Apple employee Trip Hawkins. Without the means to actually produce the hardware, the goal was to develop not a machine but a standard. 3DO offered to license the specification of its machine. In exchange for fees a potential manufacturer received the blueprints which considerably cut the R&D cost. The company's business model was to collect a royalty on each console and each game sold. That was far from being a crazy idea since JVC had pulled it off with its hugely successful VHS Video Cassette system.

Sony briefly considered it for its PSX project. The Japanese company even visited the San Mateo office to see the prototype but they eventually declined. Several other companies did acquire the rights to build a 3DO (Samsung, Toshiba, and AT&T) but never built anything.

Finally in October 1993, Panasonic, Goldstar, and Sanyo each released their own machines, respectively the Panasonic 3DO GZ-1, Goldstar 3DO, and Sanyo TRY 3DO. Later, Creative released an ISA card you could plug into a PC.

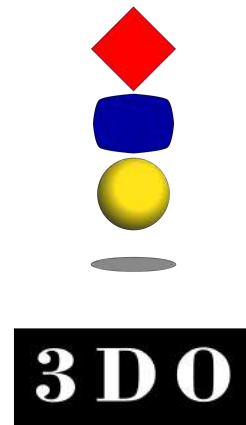


Figure 6.15: Panasonic FZ-1 implementation of 3DO specs

Trivia : The specs of the machine were originally written on the napkin of a restaurant in 1989 by Dave Needle and RJ Mical²⁶.

²⁶Source: RetroGamer #122 "Ahead of its time".

Announced during CES '92, the concept and specs made the 3DO an immediate sensation. It was the first of the fifth generation 32-bit era and there was nothing as powerful on the market.

“

Under the hood the 3DO used an ARM60 RISC Processor and had two powerful custom graphics chips and an animation processor. It also sported 3Mb RAM and a multitasking OS. Uniquely for a console, developers wrote games for the OS and not the hardware, ensuring backwards compatibility.

— **RetroGamer #122**

”

3DO had bigger plans than just gaming. Thanks to its CD format, it had the ambition to replace the VCR and enable movie streaming via the Internet.

But things would end up taking an ugly turn. In February 1993, WIRED magazine ran a full article "3DO: Hip or Hype?" that raised many concerns about the viability of the adventure.

One of the problems was the business model. To be profitable, 3DO had to make money on each console and each game sold. This was the opposite of Sega and Nintendo who would sell their machine at a loss and make it up with games. This pushed the MSRP of the 3DO up to \$699 which made it by far the most expensive console on the market²⁷. By comparison, the PlayStation's launch price was less than half the price at only \$299. Right off the bat, the machine gained itself a "rich kid" reputation²⁸.

The other aspect that poisoned the 3DO was its game library. At launch it was rather small with only six games of low quality besides "Crash 'N Burn". Firmware changes and devkit modifications until the last minute had hampered game studios in their efforts to have something ready for launch day.

Ironically, 3DO also suffered from its media, the CD-ROM. With a capacity more than 150 times more than what they were used to (650 MiB vs 4MiB), game studios experimented with lengthy pixelated cut-scenes and borderline interactive movies which turned out to be no fun at all.

More rushed poor-quality games and the release of the PSX in late 1994 annihilated any hopes of recovery. By 1995, with less than 700,000 consoles sold²⁹ the standard had lost momentum. It died soon after along with its creator, the 3DO Company.

²⁷Except for the Neo-Geo which was the same price and always remained a dream for most gamers.

²⁸The price was lowered shortly after release but by then its reputation had been set.

²⁹Source Next Generation magazine, Feb 1995.

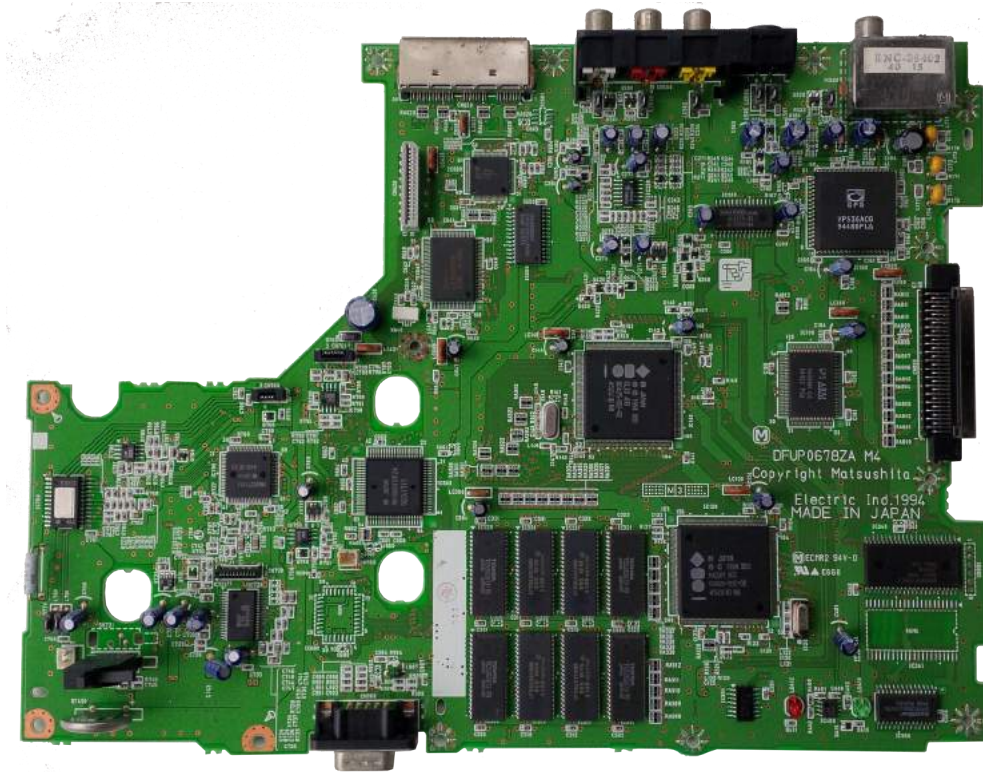


Figure 6.16: 3DO Panasonic model "FZ-10 R.E.A.L." System Board

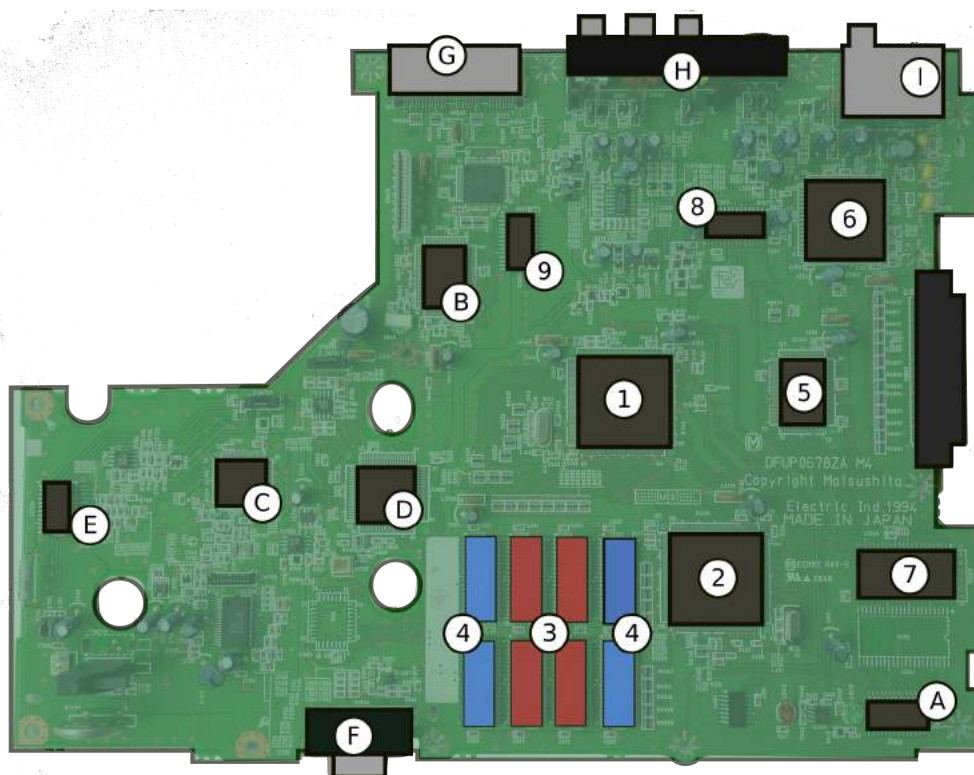
Panasonic, Goldstar, and Sanyo all designed their own motherboards but all 3DOs had the same functionality. Opening the most popular model, the "FZ-10", reveals seventeen chips!

Chips

① 50 MHz CEL Engine CLIO, ② 50 MHz CEL Engine Madam, ③ 2 MiB RAM, ④ 1 MiB VRAM (framebuffer), ⑤ 12.5 MHz ARM60 main CPU, ⑥ Corner engine (50Mhz Math co-pro), ⑦ 1 MiB OS ROM, ⑧ Digital Video Encoder (25Mhz VDLP), ⑨ 32 KiB Battery backed SRAM, ⑩ ARM CPU 32 KiB SRAM, ⑪ DSP (16-bit, 25Mhz), ⑫ CD signal processing LSI, ⑬ CD-ROM Controller MN1882410, ⑭ CD-ROM Firmware, All connected via 50 MiB/s and 36 DMA channels.

Connections

⑮ Gamepads, ⑯ Expansion port, ⑰ Composite/S-Video ports, ⑱ RF Out jack.



Developers for the 3DO were forbidden direct access to the hardware except for hand-crafted assembly for the ARM processor.

“

We were never given docs on the register set for the 3DO hardware. Using reverse engineering, we were able to get the I/O ports, but we were told by the 3DO Company our games would be rejected if they found we bypassed the OS.

— Rebecca Heineman

”

The M2 project – which began as an accelerator add-on for the 3DO – morphed into what could have been the 3DO 2. It was to feature dual PowerPC 602 processors in addition to newer 3D and video rendering technologies. It was never completed.

Trivia : Speculating on M2 machines, developers hid cheat codes in their games in order to improve graphics for the 3DO 2. In DOOM, the sequence "Up, Right, L, Up, Right, Right, R, A, Left" allowed increasing the active window size all the way to full screen.

6.5.1 3DO Programming

Graphics programming was done via the 3DO's "CEL engine", powered by the chips Clio and Madam, where CEL is a fancy name for "sprite". Each CEL can be drawn in screenspace with three associated vectors HD, VD, and HDD. Together they allow operations such as scaling, rotation, skewing, and even something called "perspective" in the programmer manual. The CEL engine was able to process several CELs simultaneously.

If HD and VD obviously set the horizontal/vertical vectors, the differential vector HDD deserves more explanation. Here is how it is described in the manual.

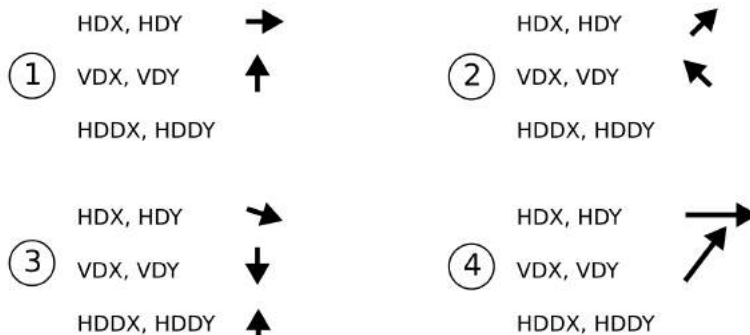
“

When fixed HDX and HDY values set the horizontal offset of a cel and VDX and VDY values set the vertical offset of a cel, the result is always a strict parallelogram—all the row edges are parallel as are all the column edges. Although you change the size and angles of the parallelogram, you cannot get any perspective effects with row edges converging or diverging. To add perspective, the projector uses the HDDX and HDDY offset pair. HDDX and HDDY change HDX and HDY values by a set amount at the beginning of each row edge. When one row edge is calculated, the projector adds HDDX to HDX and adds HDDY to HDY. It then uses the new HDX and HDY values to calculate the next row edge. Because HDDX and HDDY can change the row slope and pixel spacing from row edge to row edge, they can create converging or diverging row edges³⁰.

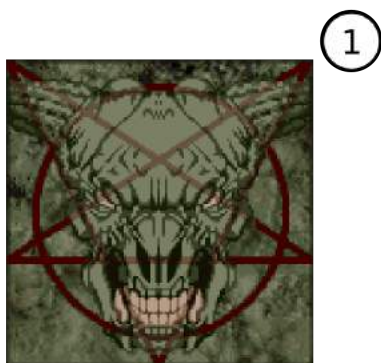
— 3DO Programmer Guide

”

① normal, ② rotation, ③ perspective (incorrect), ④ skewing with upscaling.



³⁰This is the same mechanism used for sprite distortion used in the Atari Lynx – and it should be, since RJ Mical and Dave Needle designed the Lynx too.



6.5.2 Doom on 3DO

Given the specs of the machine, the 3DO had the potential to be the best console host for DOOM. The Jaguar version had been positively received, so it would have been logical that with more RAM and superior graphic hardware, the result would make both gamers and publishers happy. Alas, in a crazy turn of events, what was released was a massacre of the original, universally accepted as the worst console version.

In January 1995, for \$250,000 (some articles even mentioned \$500,000) and an obligation to release before Christmas 1995, Art Data Interactive had landed the rights to DOOM on 3DO. To the people at the company it felt like "a license to print money". Many features were promised to the press, among them new weapons, new monsters, new maps, and Full Motion Video (FMV) sequences with real actors to build up the story.



Figure 6.17: The FMV shooting set (photo released by Rebecca Heineman)

The project was promptly subcontracted to a gaming company with actual experience in game development. Art Data Interactive quickly learned that the port was much more ex-

pensive than they thought. It would cost a million dollars and a year of development to which they agreed.

By July 1995, relations with the subcontractor had deteriorated beyond repair³¹. With a long-promised release date of October 1995 looming over their heads, Art Data Interactive reached out to a contractor that had previously done amazing work on porting Wolfenstein 3D to the 3DO. The name of the poor soul who accepted the project was Rebecca Ann Heineman from Logicware.

Having committed to the project, Rebecca asked ADI for the source code she assumed they had received upon signing the contract with id Software. Nothing came. Eventually, she received a floppy disk. It turned out to just contain the commercial version of DOOM with the compiled binary `DOOM.EXE` and `DOOM.WAD`! Rebecca had to explain to ADI what source code was and how she could not start working from the binary. After a few weeks of struggling, she finally emailed John Carmack, who sent her the source code for the Jaguar version.

With ten weeks before going gold, Rebecca worked heroically and managed to reach the deadline. The final product however had sustained extensive damage³².



Figure 6.18: Doom on 3DO

³¹Source: "The unfortunate tale of 3DO DOOM" by Matt Gander.

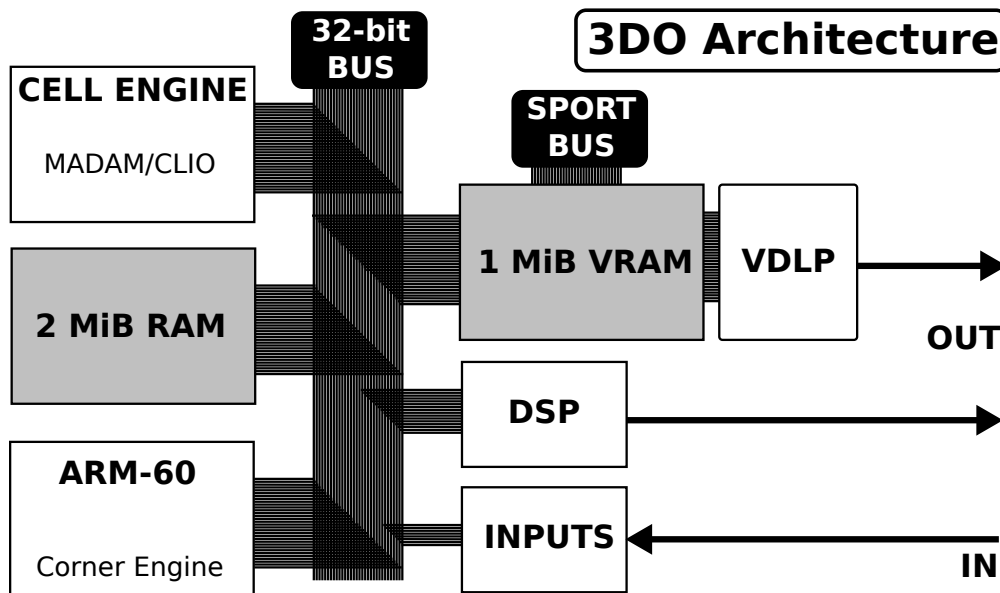
³²Burgertime 7/12/2015: DOOM 3DO.

The CEL engine was leveraged to render walls (one pixel wide columns like the PSX) but a bug had forced Rebecca to render flats in software. She had no time to write an audio driver for music playback so she recorded the PC version and sent them to the CEO of ADI who also happened to be a guitarist. A band was hired to re-record the music. Their covers were played directly from the CD.

Performance was terrible. Upon seeing the result, id Software demanded the active window size be reduced from fullscreen to 1/3 of the screen. Even with that adjustment, the frame rate was still an abysmal single digit most of the time.

ADI ordered 50,000 copies from 3DO at a price of \$150,000 in licensing and manufacturing fees. The only hope for the company to recover its expenses was to sell every single copy. With an estimated base of 250,000 users, and with AAA titles selling between 10,000 to 20,000 copies, that was a huge gamble. Unfortunately yet quite logically, players hated it, the gaming press destroyed it, and ADI declared bankruptcy soon after.

“ Although it's still Doom, it's a real duffer of a conversion.
— Ed Lomas for CVG - Rated 60% ”



Examining the 3DO hardware diagram on the previous page only makes us regret more that Rebecca didn't have more time on her hands to complete the project.

“

I was misled about the state of the port when I was offered the project. I was told that there was a version in existence with new levels, weapons and features and it only needed "polishing" and optimization to hit the market. After numerous requests for this version, I found out that there was no such thing and that Art Data Interactive was under the false impression that all anyone needed to do to port a game from one platform to another was just to compile the code and adding weapons was as simple as dropping in the art.

My friends at 3DO were begging for DOOM to be on their platform and with Christmas 1995 coming soon (I took this job in August of 1995, with a mid October golden master date), I literally lived in my office, only taking breaks to take a nap and got this port completed.

I had no time to port the music driver, so I had a band that Art Data hired to redo the music so all I needed to do is call a streaming audio function to play the music. This turned out to be an excellent call because while the graphics were lackluster, the music got rave reviews.

3DO's operating system was designed around running an app and purging, there was numerous bugs caused by memory leaks. So when I wanted to load the Logicware and id software logos on startup, the 3DO leaked the memory so to solve that, I created two apps, one to draw the 3do logo and the other to show the Logicware logo. After they executed, they were purged from memory and the main game could run without loss of memory.

There was a Electronic Arts logo movie in the data, because there was a time that EA was going to be distributing the game, however the deal fell through.

The vertical walls were drawn with strips using the cell engine. However, the cell engine can't handle 3D perspective so the floors and ceilings were drawn with software rendering. I simply ran out of time to translate the code to use the cell engine because the implementation I had caused texture tearing.

I had to write my own string.h ANSI C library because the one 3DO supplied with their compiler had bugs! string.h??? How can you screw that up!?!?! They did! I spent a day writing all of the functions I needed in ARM 6 assembly.

— Rebecca Ann Heineman

”

6.6 Saturn (1997)

Development of the Sega Saturn started in June 1992³³ as a replacement for the insanely popular yet aging Genesis. At this point in time, the Genesis had sold more than 30 million units and had a "cool" image among the 15-25 range - an image built with many good games and massive TV advertising campaigns. For Sega, it was a colossal yet mandatory undertaking to at least match its predecessor.

After two years of hard work, Sega demonstrated a prototype Saturn during the Tokyo Toy Show in June 1994. Unknown to them, it would cause even more damage than the 32X, ruin Sega International's image, and sell poorly.

During its development, Sega worked in partnership with Hitachi to develop a new CPU tailored to its needs. The joint venture resulted in the "SuperH RISC Engine" (a.k.a SH-2) at the end of 1993 which Sega used in dual configuration as foundation for the Saturn.

On the graphics side, one video display processor (VDP) was to do most of the job. However reports of the PlayStation's capabilities prompted Sega to add a second VDP to improve the system's 2D performance and texture-mapping.



³³Source: "Console Wars: Sega, Nintendo, and the Battle That Defined a Generation".

Sega managed to release its console before the dreaded PSX and sales in Japan were initially promising with games such as Daytona USA and especially Virtua Fighter being well received. The initial success had a lot to do with Virtua Fighter which was by far the most popular arcade game in Japan at the time³⁴.

Beating Sony came at a great price and the result seemed rushed. During E3 1995 in Los Angeles, Sega CEO Tom Kalinske surprise-announced that the Saturn would be available the very same day. Even their supplier did not know about this and the console was out of stock rapidly. Another consequence of the rush was that only six games were available upon launch. Panzer Dragoon, which could have made for a perfect flagship title, missed its deadline³⁵.

The machine was also difficult to program. 3D was achieved in a similar way to the 3DO. Programmers had to deal with 2D quads which could be distorted in screen space to poorly fake perspective. It was not a matter of not trying hard, the hardware was complex.

“

Currently we only use the Master SH2, the slave SH2 will be used when we get around to figuring out how.

— **Mick West 1995 Saturn development journal**

”

Worldwide, the platform had a lukewarm reception. And then the beast was unleashed.

Two weeks after release, the PSX came out with Ridge Racer and took the world by storm³⁶. Not only was the Saturn more expensive (\$399) than the PSX (\$299), games such as Daytona USA which used to look good now had blatant issues when put side-by-side with Ridge Racer. The lower framerate, polygon pop-up and letter-boxed presentation begged for mercy. To add more pressure, in June 1996 the market welcomed another competitor with the Nintendo 64.

Eventually, several good games were released but the damage had been done. More issues kept adding up. The copy protection was hacked early on. Electronic Arts refused to release its popular E.A. Sports games on the platform. With the release of the Xbox and the PS2, Sega found itself technically outgunned. With the failure of both the 32X and then the Saturn, Sega bled money for years. In late 2001, and on the verge of bankruptcy, Sega decided to withdraw from the hardware business and focus on producing games on its successful "Virtua" line of products. Ironically Sega's last console, the Dreamcast, was highly regarded by both programmers and players.

³⁴Source: "Virtua Fighter Mania". GamePro. No. 89. February 1996. p28.

³⁵Source: "The Making Of... Panzer Dragoon Saga", nowgamer.com.

³⁶The PlayStation outsold the Saturn by a factor of three.

6.6.1 Programming the Saturn

Programming the Saturn was a difficult task. The programmer's manual is broken down into eight voluminous manuals requiring repeat reading sessions to build a mental image of the flow of data. The diagram in figure 6.19 gives some idea of the daunting effort required to coordinate eight chips.

Main programming was done via the two SH-2 processors connected to 2.0MiB of shared RAM. One SH-2 was deemed the master and the other the slave. In the common configuration, the slave was intended to be used as a helper for parallelizable tasks. Communication between chips (to indicate what to execute) had to be done via a tedious interrupt system. To deal with static and global variables the programmer had to deal with mutexes and semaphores which were uncommon concepts for game console programmers. Because they were on the same bus, one had to wait for the other if both needed to access either to RAM or a peripheral on the system. Attempts to minimize the issue were made via a shared unified 4KiB cache that had little impact on the bottleneck.

Audio was done via the SCSP (Saturn Custom Sound Processor) that piloted a sound processor (a Motorola 68000). The SCSP was to be configured to perform sound mixing in the dedicated 512 KiB of RAM, which was then picked up by the Sound Processor. The combination of these made it a powerful system able to synthesize instruments, play PCM sound and perform 3D effects/distortion. The chip also polled control inputs from the player and stored them in internal registers to be polled by the SH-2s.

Graphics programming was done via two chips called VDP1 and VDP2. The VDP1 was a hardware-accelerated quad renderer. It had the particularity to use forward texture-mapping which is very efficient when rendering sprites (like in 2D games) but not so much when magnifying or minifying textures (like in 3D games). Rendering was done targeting layers which once ready were picked up by the the VDP2, composited according to their priority and transparency settings, and sends to the TV. Note that the two chips work in parallel. While the VDP1 works on the next frame, the VDP2 finishes the previous one.

Access to the CD-ROM was done via a driver piloting the SH-1 processor. The double-speed unit could read at 150 KiB/s but average access time was 300 ms. To compensate, the SH-1 stored data to a 512KiB buffer. Based on the abysmal access time, programmers were instructed to request data well in advance.

To control all these components and transfer data between systems, a seventh chip called the SCU (System Control Unit) acted as DMA controller, DSP and bus controller. The DSP was able to perform matrix transformations and write the result directly in the VDP RAM.

Trivia : Reading the programmer manual in detail reveals that each component can somehow interact with each others' RAM. This made debugging very difficult.

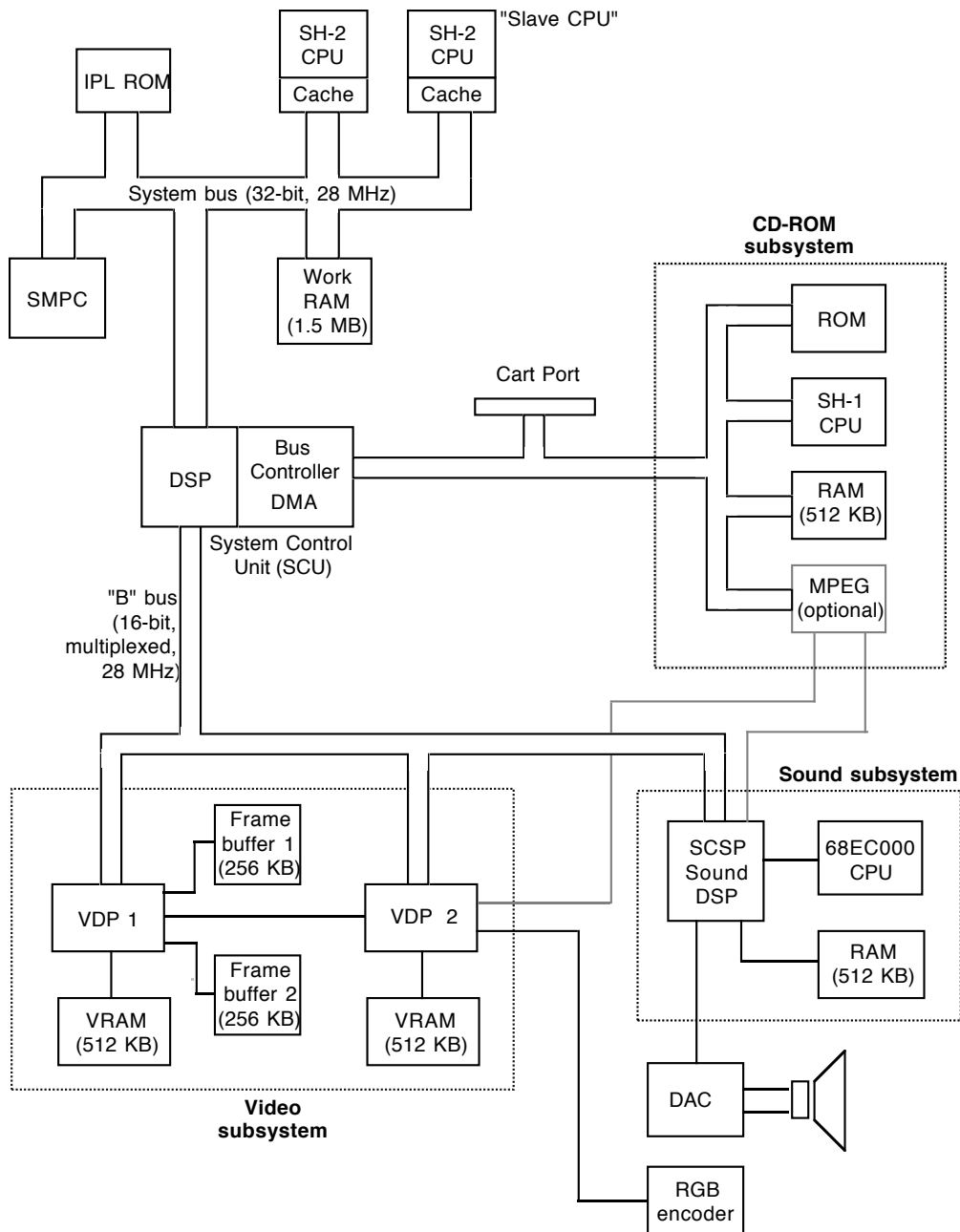


Figure 6.19: Sega manual: "Introduction to Saturn Game Development", April '94

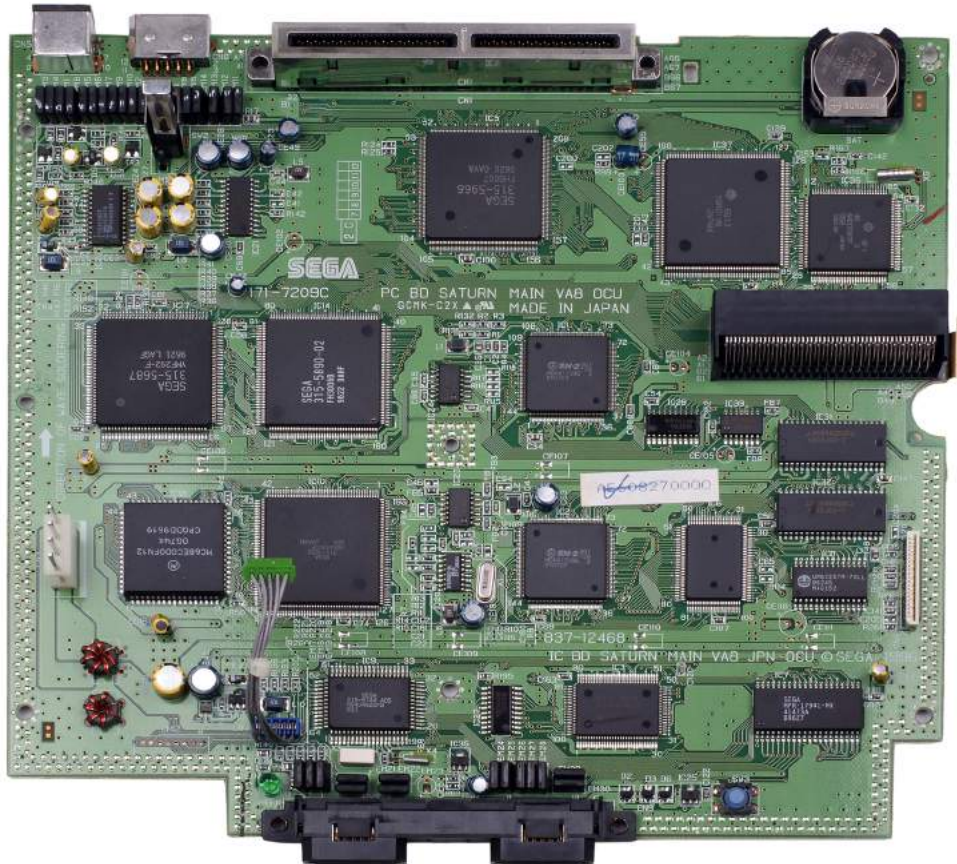
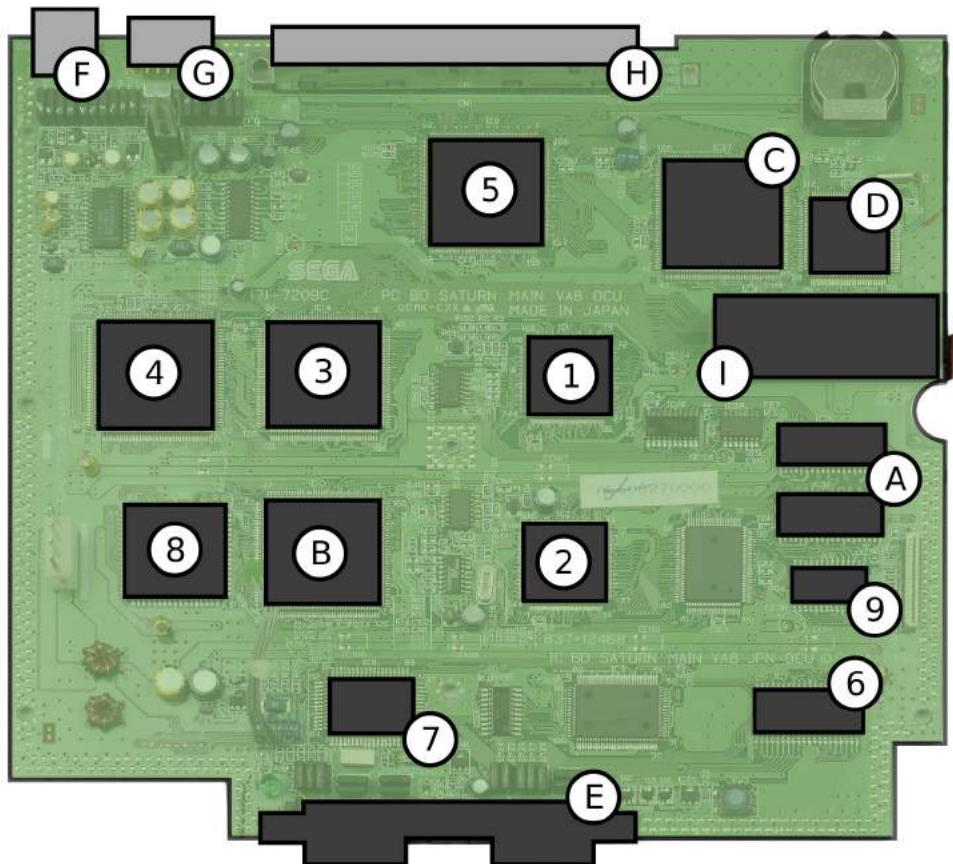


Figure 6.20: Sega Saturn motherboard

Opening a Sega Saturn and taking a look at the motherboard reveals close to twenty chips.

① 32-bit 28.6 MHz SH-2, ② 32-bit 28.6 MHz SH-2, ③ VDP2, ④ The YMF292, aka SCSP (Saturn Custom Sound Processor), ⑤ SCU DSP Math coprocessor @ 14.31818 MHz, ⑥ BIOS, ⑦ SMPC (System Management & Peripheral Control), ⑧ Motorola 68CE00, ⑨ 32 KiB Battery-backed SRAM, ⑩ 4 MiB RAM (2MiB RAM + 1.5MiB VRAM + 540KiB Audio RAM), ⑪ VDP1, ⑫ Hitachi CD-ROM I/O data controller, ⑬ 32-bit 20 Mhz SH1 microcontroller with 64k internal ROM, ⑭ Two controllers connectors, ⑮ A/V OUT socket, ⑯ Sega Communication socket, ⑰ Cart slot (RAM extender requested for "X-Men vs Street Fighter"), ⑱ CD-ROM connector.



Despite its issues and ill-timed release, it is a bitter feeling to see what happened to the Saturn and seeing it considered a failure. Over its four years of life, the platform managed to host amazing technical and entertaining games such as *Radiant Silvergun*, *Grandia*, *Sega Rally Championship*, *Virtua Fighter 2*, *Panzer Dragoon Saga*, *Guardian Heroes*, *NiGHTS into Dreams*, *Panzer Dragoon II Zwei* and *Virtua Cop*. Unfortunately, *DOOM* would not be part of the previous list.

“

After years of waiting, *Doom* finally arrives on Saturn. Unfortunately it is a breath-takingly bad conversion of a classic game.

— **Sega Saturn Magazine #16, February 1997**

”

6.6.2 Doom on Saturn

The port to the Saturn was done by Rage Software on a very tight schedule. The graphic part of the engine was done via the VDP1 writing quads into three separate layers (a floors, ceiling, and walls layer, a things layer, and a status bar layer) which were combined by the VDP2 and sent to the TV. The resulting framerate was outstanding compared to other ports but the lack of perspective correct texturing ended up disturbing Jim Bagley's plans³⁷:

“ When I started the project, I had to do a demo for id Software to approve. I started by extracting all the levels and audio and textures from the WAD files and made my own Saturn version of this, then got an early version of the renderer working using the 3D hardware. This got sent off and a couple days later I got a call from John Carmack, who stipulated that under no circumstances could I use the 3D hardware to draw the screen. I had to use the processor like the PC. Thankfully I enjoy challenges, so it turned out to be a really enjoyable project, using both SH2s to render the display like the PC did it, using the 68000 to orchestrate them both.

However, it kneecapped the game and the speed-framerate suffered greatly.

— **Jim Bagley for RetroGamer #134**

”

Years later, by 2014, Carmack had reconsidered.

“ I hated affine texture swim and integral quad verts, but in hindsight, I probably should have let experiment.

— **John Carmack**

”

In the end, the VDP1 hardware-accelerated 60 FPS-capable engine was tossed.

Due to time constraints, Jim did not have the time to change the renderer to work with pixel-wide triangles like the PlayStation. Upon shipping, the game managed a framerate that could reach 20 FPS but most of the time this dropped to the single digits at a full screen resolution of 281x235. To compensate for the low framerate, Jim Bagley made the decision to slow down all movements, a move that enraged the playing community.

³⁷RetroGamer #134.

Trivia : Another effect of the rushed schedule was a bug with the audio system that made all sound effects panned left. Players had to play in mono to hear from both speakers.³⁸



In the screenshot above, notice how E1M1 is the same as other console versions (all based on the initial work for the Jaguar). The status bar however welcomed a makeover.

What "knee-capped" the project was the walls, ceilings and floors rendition (accounting for most of the computing cost) which ended up being software-rendered via the SH-2s while the status bar and the things (such as monsters and walls featuring transparent parts) were hardware-accelerated via the VDP1³⁹. When all three layers were ready, the VDP2 composited the three layers toward the TV while the VDP1 started to render the next frame.

Translucency was done in a peculiar way for which the details constitute a testament to the complexity of the machine. Both the VDP1 and the VDP2 were capable of "half-

³⁸Digital Foundry: "Every Console Port Tested and Analysed!".

³⁹Digital Foundry: "Every Console Port Tested and Analysed!".

transparency", a term referring to equally blending source and target. However the VDP1 only supported transparency in 15-bit color while the VDP2 only supported transparency in indexed mode. As a result, you could either have sprites be transparent with regards to each other or transparent with regards to the VDP2 background layers, but not both.

This limitation was a big problem to render "Spectre" enemies. If the VDP1 marked the Spectre pixels "half-transparent", they would properly render over the background layer. However they would also have "swallowed" any other sprites possibly standing behind them, generating incorrect scenes⁴⁰.

Sega designers were well-aware of the limitation of the VDPs. To palliate the problem they introduced the concept of "mesh" sprites which were rendered opaque by the VDP1 but only every other pixels.



Figure 6.21: A Spectre enemy rendered as a "mesh".

The pixel-perfect screenshot, especially the zoomed-in version next page, may look crude at first sight. However, you have to keep in mind the composite interleaved display system

⁴⁰Source: "The Sega Saturn and Transparency" by Matt Greer.

which ended up blending everything together. Even though the visual result was remarkably convincing, this magic-trick did not survive the "HD" pixel-perfect era.



Figure 6.22: Same scene, zoomed-in to show skipped pixels in the Spectre.

Epilogue

The *Game Engine Black Book: DOOM* was first published on December 10th 2018, exactly twenty-five years after the December 10th 1993 release of the game. Within that timespan, it would be an understatement to say the world of DOOM has flourished.

DOOM I was a colossal success, beloved by critics and gamers. At \$9 per unit the game quickly found itself making \$100,000/day. According to Sandy Petersen, the game "sold a couple of hundred thousand copies during its first year". Experts estimate that the game sold approximately 2-3 million physical copies from its release through 1999. In 1995 it was estimated DOOM was installed on more computers than the Microsoft Windows operating system.

The sequel, DOOM II: Hell on Earth, was released in 1994. It was equally well received and managed to exceed sales expectations. More than 600,000 units were shipped to stores in preparation for the launch but it found itself sold out within a month. The game was the United States' highest-selling computer title of 1994. It placed 10th for 1996, with 322,671 units sold and \$12.6 million earned in the region that year alone.

id Software took a break to develop its Quake brand for a few years until they released DOOM III in August 2004. Featuring technological prowess such as dynamic shadows, the gameplay was slowed down to match the ambiance of a horror movie. Once again the title received favorable reviews from critics, and went on to become another successful title for id. In two years, 760,000 copies were sold for a total of \$32.4 million. By 2007, the title had gone on to sell over 3.5 million copies, making it id's most successful project to date.

DOOM 4 development started in 2007 but remained in limbo for several years. After a tumultuous development process and rumors of cancellation, id Software released *Doom* (named the same as the first title in the series) in 2016. It was praised for its return to a fast pace and innovative game mechanics. It was the second best-selling retail video game in the US in May 2016, reaching 500,000 copies. By July 2017, the game reached 2 million copies sold on PC.

The DOOM team from 1993 parted ways over the years, although the legacy and status

associated with the project continued to follow each of their careers. Before DOOM, John Romero famously shared his ambition to reach a level of success similar to Scott Miller and George Broussard from Apogee. He reportedly said to John Carmack:

“ They're driving bad-ass cars while we drive ass cars. It is time to kick ass.

”

After DOOM, *ass car* driving was no more. Romero's Ferrari Testarossa (modified with a COM port connected to the engine) and John Carmack's Ferrari F-40 were notorious in both the gaming and programming worlds.

Beyond sales and fame, DOOM reached a new dimension when its source code was open sourced on December 23, 1997. Hundreds of ports ensued, some still actively developed to this day, among them: LinuxDoom, DOOM 95, DosDoom, Chocolate Doom, ZDoom, BOOM, EDGE, Doom Legacy, Doom Retro, Crispy Doom, Doomsday Engine, GZDoom, csDoom, MBF, PrBoom, 3DGE, Risen3D, QZDoom, Skulltag, ZDaemon, Odamex, SMMU, PrBoom+, Zandronum and Eternity Engine – and these are only the most famous.

Personal Note:

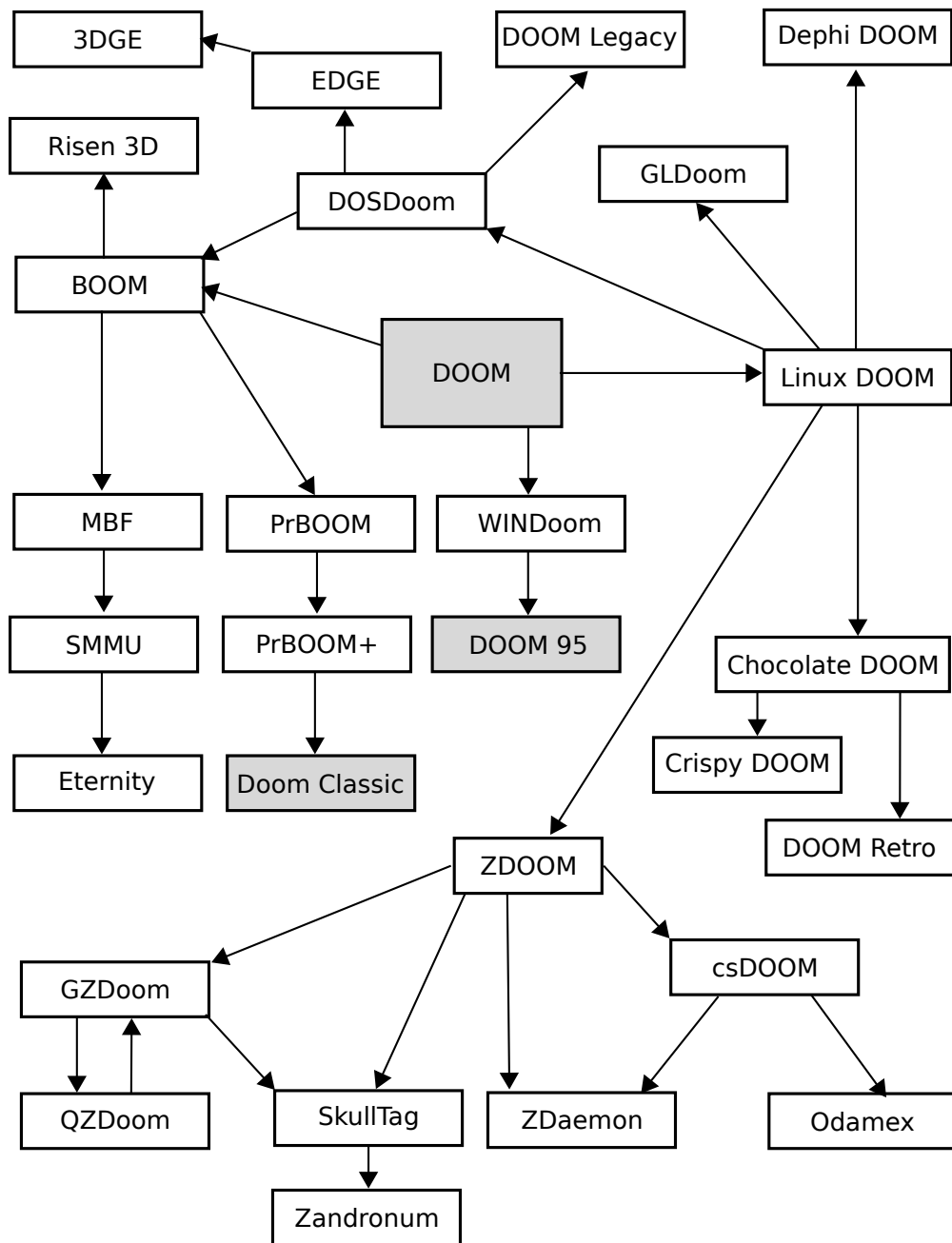
DOOM has a special place in my heart. As a 24-year old immigrant in Toronto knowing only Java, this is the codebase I used to learn C and build up my skills. It is the quality level I set myself to emulate. It is thanks to the "University of id Software" as I like to call it that I ended up being noticed by Google and eventually landed a job offer there. Something I once deemed impossible to achieve.

The title of the book *Game Engine Black Book* is an homage to Michael Abrash. The explanations in his *Graphics Programming Black Book* unlocked the most difficult parts of Quake. Michael's book features a quote which resonated with me. I have tried to live by it and so far it has served me well. Maybe you will also find it inspiring and it will guide you the same way it has guided me.

“ If you do what you love, and do it as well as you can, good things will eventually come of it. Not necessarily quickly or easily, but if you stick with it, they will come.

— Michael Abrash

”



Appendices

Appendix A

Bugs

A.1 Bugs

DOOM is known for its stability partly thanks to a development process using seven compilers and systems. Nonetheless it shipped with a few bugs.

A.1.1 Flawed collision detection

One rare collision detection bug was first brought to light (and subsequently explained in depth) by Colin "cph" Phipps in his article "Shooting Through Things".

“ A monster is getting too close for comfort. You shoot at it, and miss. If you are unlucky, the monster kills you. But you were so sure that you were pointing right into the monster, that so close as you were you couldn't have missed. Perhaps it was the chaingun playing up, shooting all the bullets off to one side. Perhaps the game was written by a bunch of losers who failed their high-school geometry. Or perhaps, in the heat of the moment, you really did miss; nobody's perfect.

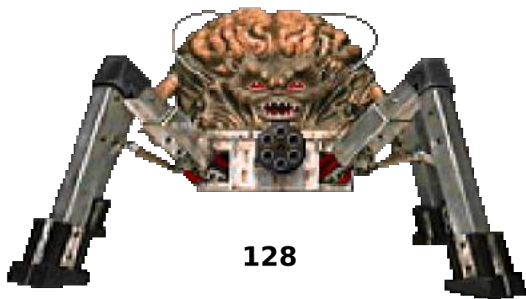
Well I have good news. You can blame your tools.

— Colin Phipps

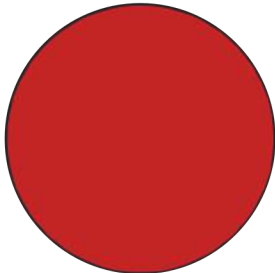
”

Some enemies can be really big, almost ten times bigger than the player (16 units), such as in the case of the Spider Mastermind (128 units). As we saw on page 263, DOOM uses blockmaps to speed up detection of intersections with things and walls. If a thing is on the edge of block and the player is a bit unlucky, what should have been a hit can end up as a miss.

SPIDER MASTERMIND



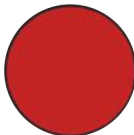
128



BABY SPIDER



64



MANCUBUS



48



CYBERDEMON



40



CACODEMON



31



**PAIN
ELEMENTAL**



31



DEMON



30



SPECTRE



30



BARON OF HELL



24



HELLKNIGHT



24



ARCH-VILE



20



REVENANT



20



CHAINGUNNER



20



IMP



20



TROOPER



20



SERGEANT



20



LOST SOUL



16

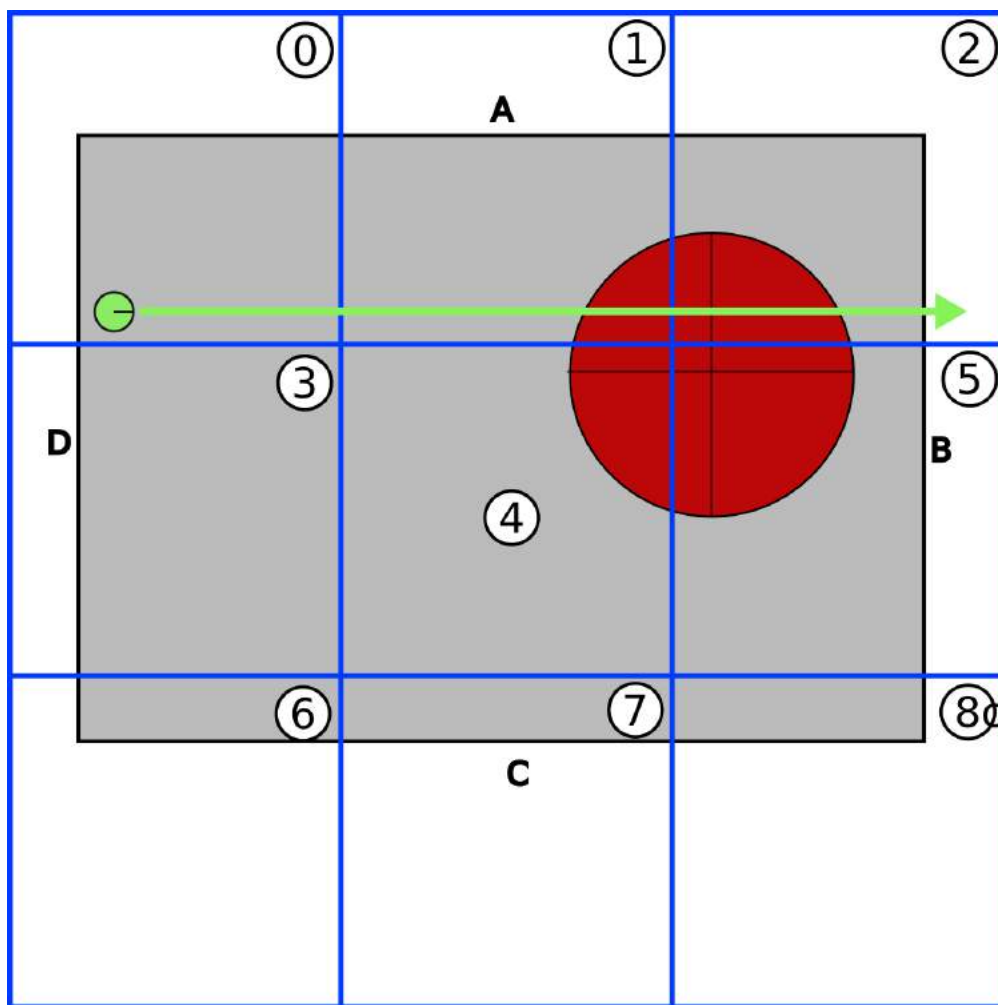


PLAYER



16





It is possible to miss a SpiderDemon in a hallway. In the single room map above, the green player on the left is firing at a red enemy 128 units wide on the right (probably a SpiderDemon). The blue grid shows the blockmap's alignment.

The line of the bullet and the radius of the monster clearly overlap. This should be a hit. But only the content of blockmaps 0, 1 and 2 will be checked, resulting in tests with walls D, A and B. Since the enemy is in block 5 and the bullet doesn't cross it, the hit is not registered.

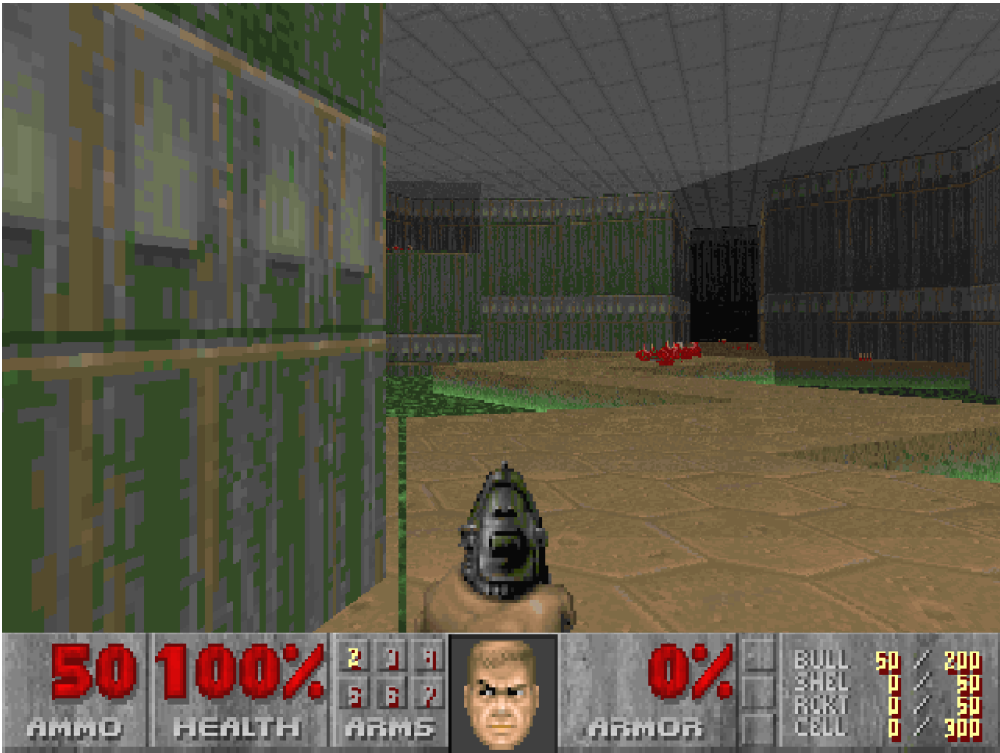
This bug is not limited to very large monsters. It can happen with any enemy depending on how close they are to the player and how the blockmaps align.

A.1.2 Slime trail

A slime trail happens when there is a horizontal screen space gap between two walls. Vis-planes "leak" between the space, resulting in graphical glitches. This was a known issue during development that was never fixed due to deadlines and the fact they only rarely occurred. John Carmack mentioned it when the `doombsp` source code was released.

“ There IS a bug in here that can cause up to a four pixel wide column to be drawn out of order, causing a more distant floor and ceiling plane to stream farther forward than it should. You can sometimes see this on E1M1 looking towards the imp up on the ledge at the entrance to the zig zag room. A few pixel wide column of slime streams down to the right of the walkway. It takes a bit of fidgeting with the mouse to find the spot. If someone out there tracks it down, let me know...

— John Carmack ”



The issue can actually be far wider than four pixels if one knows where to look.

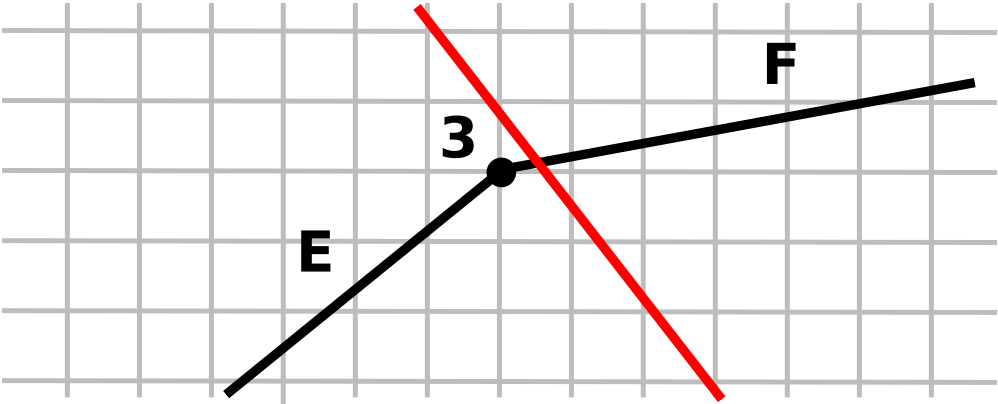
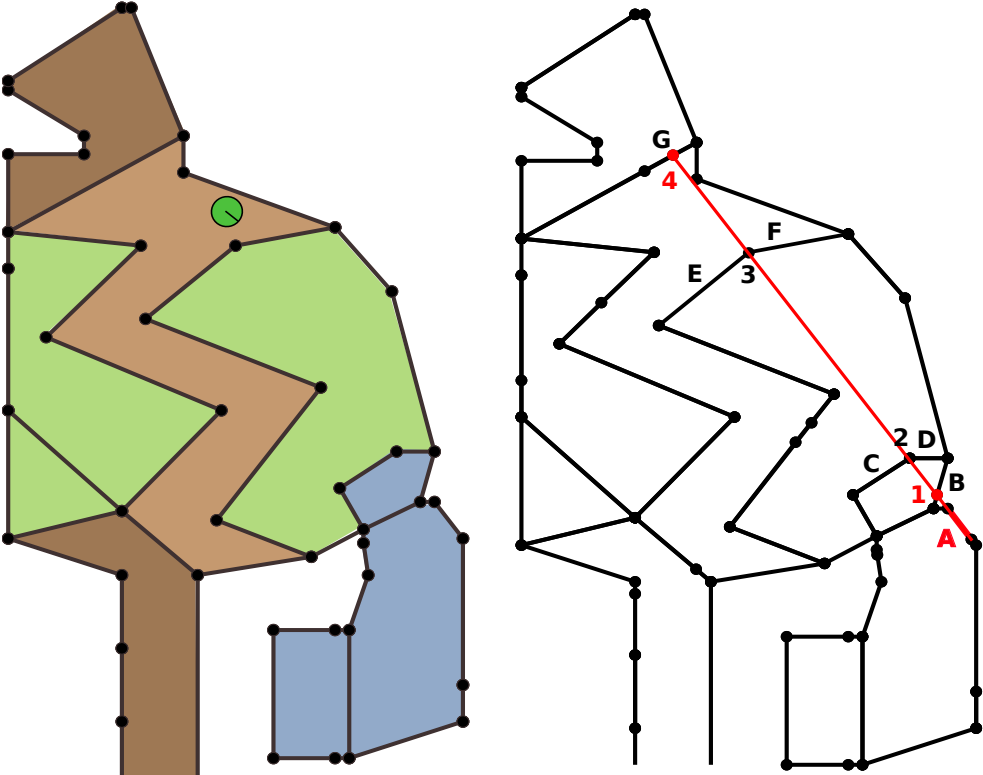


This particular instance of slime trail is the result of the engine's visplane inference system combined with the limited precision of integers when a map is sliced up by via `doombsp`.

Once again, let's take an example. The two previous screenshots were taken in the very first map of the game, E1M1, in a zigzag section surrounded by toxic green liquid. At first glance, there is nothing unusual here but when we take a look at how it was sliced during binary partitioning, an interesting special case appears.

Line A was selected as a splitter. As it crosses lines B and G, two segments are created as B1, B2, G1, and G2. The new vertices created are in red. Things are less clear for lines C and E (or is it F ?). We need to look closer.

If we zoom in on vertex 3, we see the splitter crossed F between integer coordinates. Since map vertex coordinates are stored as integers, a split is impossible. The error is small, so `doombsp` treats the vertex as if it was exactly on the splitting line.

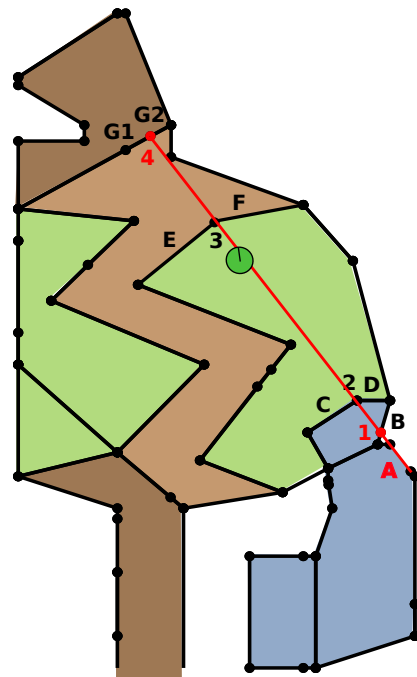




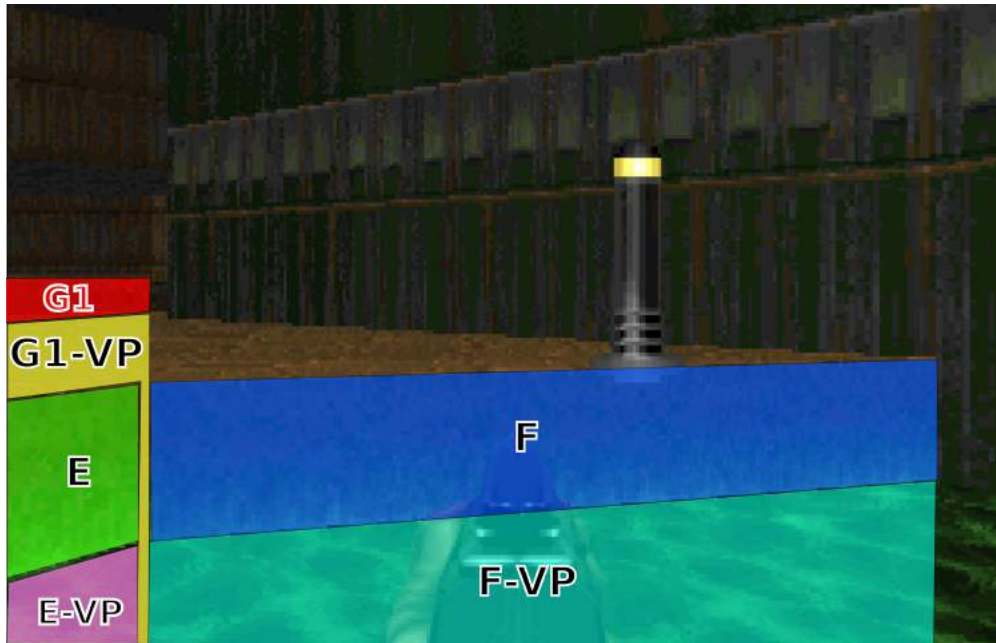
Let's position the player along the same split line we just studied, but facing the opposite direction (the scene is simpler with fewer walls). The player is very close to segments E and F (which is where the rendering error is). Further in the background are segments G1 and G2 where line G was split.

The frame starts with a blank canvas. Portal E is rendered first. It has no upper or middle texture but it does have a lower texture that is drawn (marked with a green overlay on page 377). As the lower texture is rendered, the screen space below is inferred to be a floor visplane E-VP (shown with a pink overlay). The engine then continues down the BSP and renders G1.

G1's lower texture is rendered (in red). Everything below that is inferred to be a floor and therefore visplane G1-VP is created. Notice that the visplane goes all the way to the bottom of the screen which is a rendering bug. Had the BSP been split properly, F1 would have been rendered before G1 and



stopped the visplane from flooding in screenspace improperly.



From there, the damage is done. The engine renders the other side of the BSP and hits F which is clipped against the vertical boundary set by G1 and then drawn (in blue). The space below F is inferred to be floor and visplane F-VP (turquoise) is generated¹.

A.2 Barrel suicide

DOOM monsters are capable of fighting against each other. If friendly fire was to occur during combat, the damaged monster will automatically attempt to retaliate instead of attacking the player. This is called monster infighting.

There is an amusing special case of infighting which involves exploding barrels. When a barrel is damaged, the engine "memorizes" which entity was responsible for it. When the barrel actually explodes the source of the damage is passed to all damaged entities so they can retaliate.

What can happen is that a monster triggers a barrel to explode and gets injured by the blast in the process. In this occurrence, monsters with a melee attack (Cacodemon, Imp, Hellknight) will "tear themselves apart" while those only capable of range attack (former

¹ The engine actually merges E-VP and F-VP but let's discard it for the sake of simplicity.

humans) will go nuts and fire blindly at random, possibly triggering further monster infighting.

Appendix B

Dots

B.1 Waiting for the Dots

For anybody who played DOOM on a PC in 1994, the most frustrating part was waiting for the game to load. One step in particular, the mysterious `R_Init`, seemed to take forever¹. An improvised progress bar made of dots informed the player that the game was loading and to just wait and be patient. Millions of hours were spent watching these tidy dots progress to the right. Probably a few more were spent trying to guess what `R_Init` actually did in the background.

```
R_Init: Init refresh daemon [.....]
```

With access to the source code it is possible to modify the engine to output a "label" matching the current phase performed instead of dots. It turns out there are eleven "phases".

```
R_Init: Init refresh daemon [000001233333333333333456789A]
```

Phase 00000 corresponds to `R_InitTextures`. Something that was not mentioned in the 3D renderer section of the book (for the sake of simplicity) is that textures are made of patches. In order to save space, textures reference all patches via an identifier. This is particularly powerful in the case of textures made of a repeating pattern. This phase is where all textures attributes such as dimension and patches definition are loaded to RAM (however the texels remain in the `.wad`). Because of the volume of data and the amount of `Z_Malloc` memory allocation, it is the most expensive phase.

The first step of this phase is to open a lump named `PNames` that contains all the patch names. From the order they appear, a mapping of patch name to ID is established.

¹In fact, it took anywhere from 15 to 30 seconds depending on the HDD speed.

The second step (which accounts for the bulk of the processing time) is to open lumps `TEXTURE1` and `TEXTURE2`. These contain all the texture entries. Each entry features a name and list of patch IDs, along with patch coordinate and offsets.

Phase 1 is just a marker showing when `R_InitTextures` returns.

Phase 2 corresponds to `R_InitFlats`. It looks for lumps `F_START` and `F_END` which are the markers surrounding flat textures. Only the number of flats is retrieved so a proper `malloc` of the flat array can be performed.

Phase 333333333333 is similar to phase 0 but this time it involves sprite lumps. Function `R_InitSpriteLumps` looks up lumps `S_START` and `S_END` to find the width and horizontal offset of all sprites in the WAD and saves that data to a sprite array. In the process it prints a dot every 64 sprites. This was the second slowest phase in `R_Init` (after `R_InitTextures`) since it had to perform a lot of I/O.

Phase five (4) is a simple marker to show the end of `R_InitSpriteLumps`.

Phase six (5) is also a marker to show the end of `R_InitData` which encompasses phases 00000, 1, 2, 333333333333, and 4.

Phase 6 matches function `R_InitPointToAngle` – when used – to build the tangent lookup table. This is now an empty function since it is pre-calculated and stored in `tables.c`

Phase 7 matches function `R_InitTables` which used to build lookup tables `finetangent` and `finesine`. Like the tangent lookup table, these are now pre-calculated in `tables.c` and baked into the executable.

Phase 8 matches function `R_InitPlanes` and does nothing. What a waste of a dot.

Phase 9 matches function `R_InitLightTables` and initializes the `zlight` table used to implement the lightmaps.

Phase A matches function `R_InitSkyMap` which initializes the static `skyflatnum`.

B.2 Reload Hack

While developing the game, the long startup time was a problem. Even if a designer made no change to the geometry of the map (and therefore did not need to run `doombsp`) it still took 30 seconds before he could see the result. To avoid breaking the creative flow, the

engine was gifted with a "reload hack".

By prefixing the path to a WAD with a tilde (~), the engine was set up to reload the WAD on every level start.

```
C:\>doom2.exe -file ~mycustom.wad
```

This allowed artists and designers to see the results of their work almost instantly.

Appendix C

NeXTstation TurboColor

As of 2018, it has been twenty five years since the last machine came out of NeXT 's Redwood City factory in 1993. It has become a rare occurrence to find one of these pieces of black hardware in working condition.

Since this book strives to be historically accurate, it was paramount to find an actual NeXTstation TurboColor – first and foremost to document the development condition of the time, but also to witness the full game pipeline in motion. Even though passionate and dedicated programmers have produced a gorgeous emulator called "Previous", the performance numbers would not have been accurate.

I lucked out on eBay and found exactly the configuration I needed. The machine was in working condition but the SCSI hard-drive was making clinking noises, a sign that it was about to die. Additionally, the MegaDisplay colors had faded out¹ and its 50 lbs (23 kg) made it difficult to move.

Thanks to Rob Blessin, owner and founder of Black Hole Inc., I was able to replace the HDD with a SD card SCSI2SD providing similar access time. It was difficult to find a screen compatible with NeXT's exotic "sync on green" but thanks to the wonderful people at www.nextcomputers.org I was pointed to a NEC MultiSync 1980SX which worked flawlessly.

Words cannot convey how it felt to hear the humming of the machine's fan. To witness `Doom.app`, `DoomED`, and `doombsp` compile flawlessly. To witness this NeXTstation TurboColor (serial #ABC0053943) come back to life. The machine did not cure cancer as Jobs wished for but it did provide happiness to countless developers.

¹This would have been easily fixable by replacing the capacitors on the monitor control board, but it would not have solved the issue of the weight.

C.1 Developing The Game

On this double page is recreated the typical developer desktop setup. Notice "Interceptor VGA Console" which gives away libinterceptor.a, a private library provided by NeXT's engineers to punch a hole in Display Postscript and bypass the "slow" compositor.

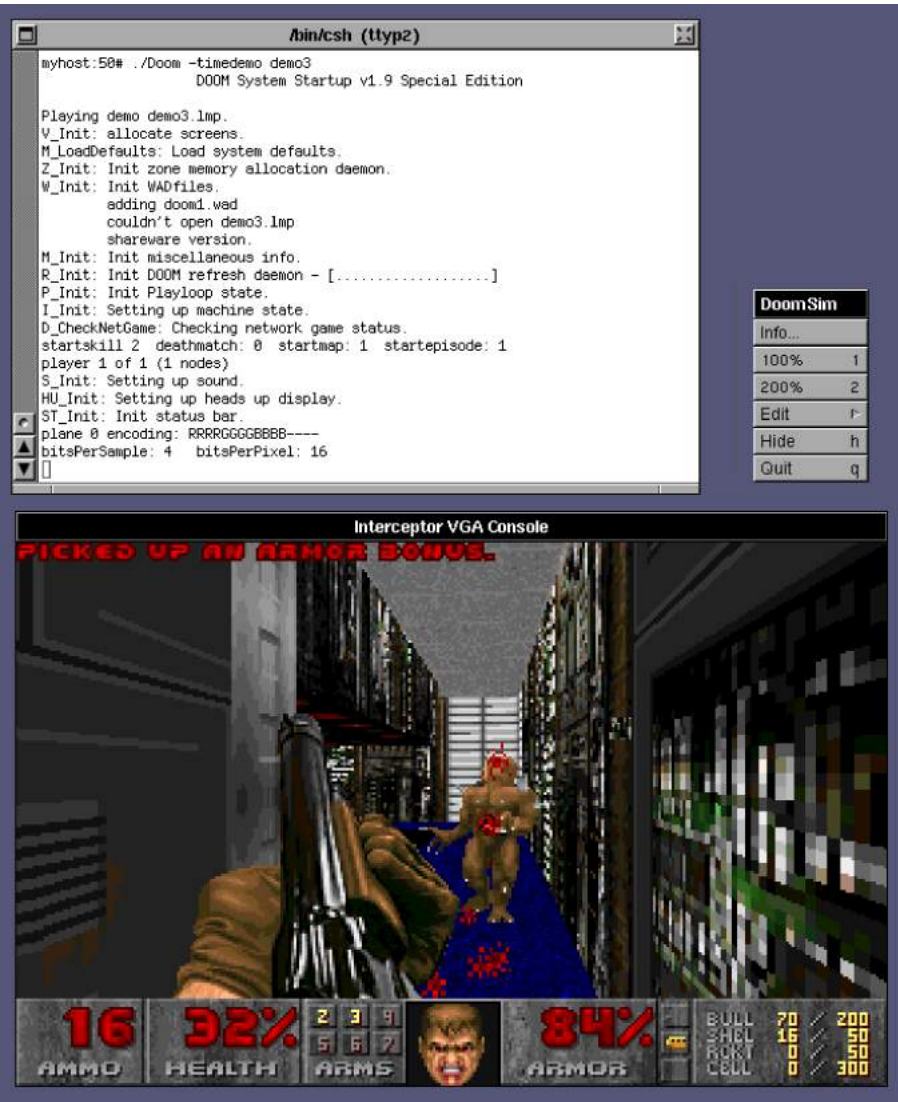


Figure C.1: NeXTSTEP development setup (left part of the screen)

The MegaDisplay resolution of 1120x832 was so high that id Software had to implement a 2x software zoom for the game window. Without it, the DOOM window looked like a tiny stamp with barely any pixels visible.

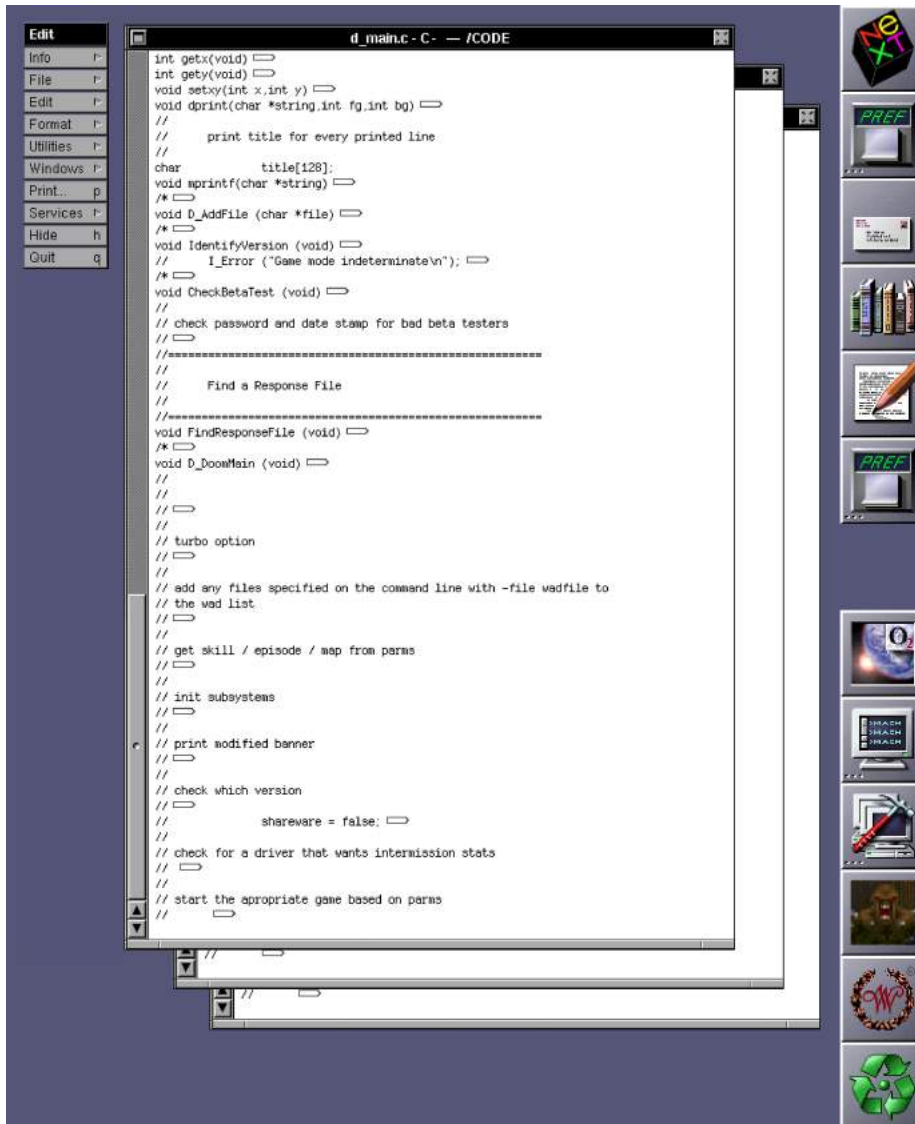


Figure C.2: NeXTSTEP development setup (right part of the screen)

C.2 Compiling Maps

Benchmarks for doombsp run time for each level in DOOM and DOOM II².

Map	doombsp runtime (s)	Map	doombsp runtime (s)
E1M1	8.2	MAP01	6.1
E1M2	32.0	MAP02	6.6
E1M3	26.2	MAP03	8.7
E1M4	18.4	MAP04	8.5
E1M5	19.9	MAP05	17.6
E1M6	44.0	MAP06	25.0
E1M7	22.3	MAP07	1.9
E1M8	6.9	MAP08	15.2
E1M9	15.4	MAP09	16.3
E2M1	6.0	MAP10	34.0
E2M2	55.4	MAP11	15.7
E2M3	19.6	MAP12	15.2
E2M4	36.0	MAP13	31.5
E2M5	46.8	MAP14	44.7
E2M6	32.5	MAP15	66.0
E2M7	60.8	MAP16	16.2
E2M8	2.5	MAP17	36.2
E2M9	1.5	MAP18	17.2
E3M1	2.5	MAP19	45.8
E3M2	9.2	MAP20	29.2
E3M3	38.1	MAP21	5.7
E3M4	23.7	MAP22	9.4
E3M5	34.5	MAP23	7.5
E3M6	22.5	MAP24	30.5
E3M7	23.4	MAP25	21.1
E3M8	1.9	MAP26	18.8
E3M9	8.9	MAP27	26.2
		MAP28	19.6
		MAP29	45.8
		MAP30	1.0
		MAP31	16.4
		MAP32	2.7
		MAP33	6.6
		MAP34	9.3
		MAP35	0.3

²Based on .map files released by John Romero on 2015-04-22.

C.3 Running The Game

Running DOOM on a NeXTstation TurboColor produced a surprisingly poor framerate.

Mode	Resolution	High Details FPS	Low Details FPS
B	320x200	9	13
A	320x168	9	14
9	288x144	11	15
8	256x128	12	16
7	224x112	13	17
6	192x096	15	19
5	160x080	17	20
4	128x064	19	22
3	096x048	21	23

Figure C.3: DOOM framerate on a NeXTstation TurboColor

It gets even worse when running the game with the 2x zoom used during development. In this mode, the same number of pixels are written to the core's framebuffer but four times more data must transit over the bus.

Mode	Resolution	High Details FPS	Low Details FPS
B	640x400	6	8
A	640x336	6	8
9	576x288	7	9
8	512x256	8	9
7	448x224	8	10
6	384x192	9	10
5	320x160	9	10
4	256x128	10	11
3	192x096	11	11

Figure C.4: DOOM 2x zoom framerate on a NeXTstation TurboColor

Lowering the resolution or the detail level helped a little bit but not as much as with the DOS version. That's because on NeXT, the video system is implemented differently.

The implementation disregards update signals from `I_UpdateNoBlit` and defers all work to `I_FinishUpdate` where the full content of framebuffer #0 is blitted to the `NSWindow`. There is no dirty box optimization and no direct access to the hardware like on DOS.

Trivia : The "low detail" mode was never properly implemented on NeXTSTEP. The engine writes only half the columns but there is no system to duplicate them like the VGA bank

mask did. As a result, only the left portion of the `NSWindow` is updated.

C.4 Framebuffer Non-distortion

The NeXTstation had a "clean" video system where the color space was linear and the pixels were "square" (the framebuffer had the same aspect ratio as the MegaDisplay monitor). As a result DOOM's³ framebuffer #0 suffers no distortion when presented by the window system.

The 320x200³ "Interceptor VGA Console" `NSWindow` is not stretched to 320x240 and therefore appears vertically squashed. It is particularly noticeable when the splash screen on NeXTSTEP (Figure C.5) is displayed next to the DOS version (Figure C.6).

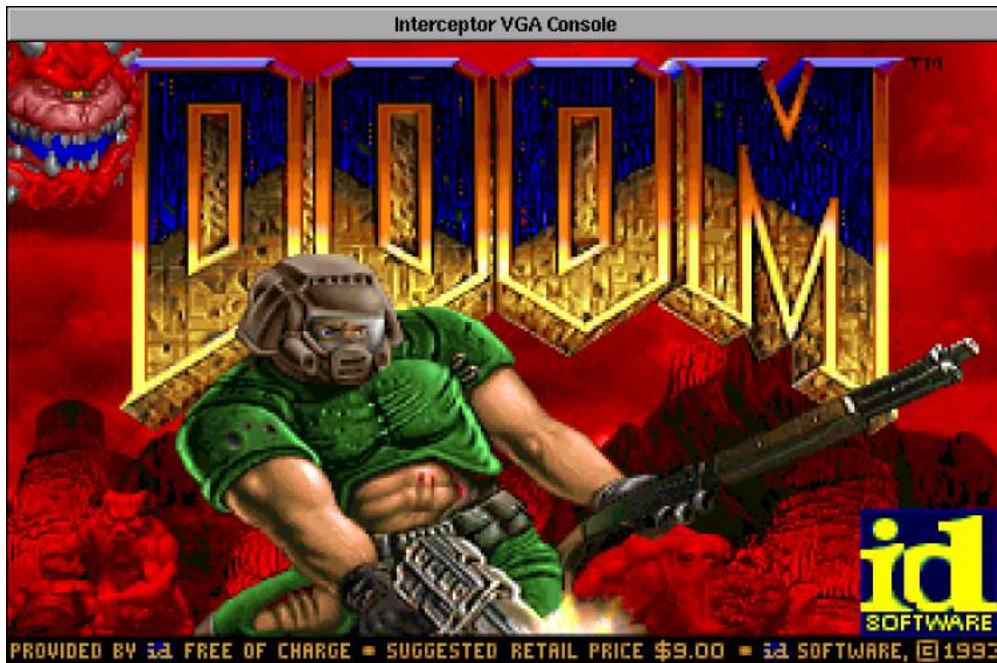


Figure C.5: DOOM on NeXTSTEP. Content appears vertically squashed

Trivia : Many ports got the aspect ratio wrong. DOOM 95 which was to showcase Microsoft's Windows 95 graphics drivers low overhead was among the guilty. In its default setting, the 320x200 hosting window directly maps DOOM's core framebuffer. As a result, enemies look shorter, rocket explosions are oval, and everything else is distorted.

³320x200 is the dimension of the active area, not including the title bar.



Figure C.6: DOOM in 4:3 aspect ratio as presented on a 1993 PC monitor

Appendix D

Press Release

In January 1993, DOOM development start was shared with the public through a press Release. The impressive list of graphics features, multiplayer modes, and an unseen before promise that DOOM would be an "open game" was warmly received.

Id Software
1515 N. Town East Blvd. #138-297, Mesquite, TX 75150

FOR IMMEDIATE RELEASE

Contact: Jay Wilbur
FAX: 1-214-686-9288
Email: jay@idsoftware.com (NeXTMail O.K.)
Anonymous FTP: ftp.uwp.edu (/pub/msdos/games/id)
CIS: 72600,1333

Id Software to Unleash DOOM on the PC

Revolutionary Programming and Advanced Design Make For Great
Gameplay

DALLAS, Texas, January 1, 1993-Heralding another technical revolution in PC programming, Id Software's DOOM promises to push back the boundaries of what was thought possible on a 386sx or better computer. The company plans to release DOOM for the PC in the third quarter of 1993, with versions planned for Windows, Windows NT, and a version for the NeXTall to be released later.

In DOOM, you play one of four off-duty soldiers suddenly thrown into the middle of an interdimensional war! Stationed at a scientific research facility, your days are filled with tedium and paperwork. Today is a bit different. Wave after wave of demonic creatures are spreading through the base, killing or possessing everyone in sight. As you stand knee-deep in the dead, your duty seems clear—you must eradicate the enemy and find out where they're coming from. When you find out the truth, your sense of reality may be shattered!

The first episode of DOOM will be shareware. When you register, you'll receive the next two episodes, which feature a journey into another dimension, filled to its hellish horizon with fire and flesh. Wage war against the infernal onslaught with machine guns, missile launchers, and mysterious supernatural weapons. Decide the fate of two universes as you battle to survive! Succeed and you will be humanity's heroes; fail and you will spell its doom.

The game takes up to four players through a futuristic world, where they may cooperate or compete to beat the invading creatures. It boasts a much more active environment than Id's previous effort, Wolfenstein 3-D, while retaining the pulse-pounding action and excitement. DOOM features a fantastic fully texture-mapped environment, a host of technical tour de forces to surprise the eyes, multiple player option, and smooth gameplay on any 386 or better.

John Carmack, Id's Technical Director, is very excited about DOOM: Wolfenstein is primitive compared to DOOM. We're doing DOOM the right way this time. I've had some very good insights and optimizations that will make the DOOM engine perform at a great frame rate. The game runs fine on a 386sx, and on a 486/33, we're talking 35 frames per second, fully texture-mapped at normal detail, for a large area of the screen. That's the fastest texture-mapping around-period.

Texture mapping, for those not following the game magazines, is a technique that allows the program to place fully-drawn art on the walls of a 3-D maze. Combined with other techniques, texture mapping looked realistic enough in Wolfenstein 3-D that people wrote Id complaining of motion sickness. In DOOM, the environment is going to look even more realistic. Please make

the necessary preparations.

A Convenient DOOM Blurp

DOOM (Requires 386sx, VGA, 2 Meg) Id Software's DOOM is real-time, three-dimensional, 256-color, fully texture-mapped, multi-player battle from the safe shores of our universe into the horrifying depths of the netherworld! Choose one of four characters and you're off to war with hideous hellish hulks bent on chaos and death! See your friends bite it! Cause your friends to bite it! Bite it yourself! And if you won't bite it, there are plenty of demonic denizens to bite it for you!

DOOM-where the sanest place is behind a trigger.

An Overview of DOOM Features:

Texture-Mapped Environment

DOOM offers the most realistic environment to date on the PC. Texture-mapping, the process of rendering fully-drawn art and scanned textures on the walls, floors, and ceilings of an environment, makes the world much more real, thus bringing the player more into the game experience. Others have attempted this, but DOOM's texture mapping is fast, accurate, and seamless. Texture-mapping the floors and ceilings is a big improvement over Wolfenstein. With their new advanced graphic development techniques, allowing game art to be generated five times faster, Id brings new meaning to "state-of-the-art".

Non-Orthogonal Walls

Wolfenstein's walls were always at ninety degrees to each other, and were always eight feet thick. DOOM's walls can be at any angle, and be of any thickness. Walls can have see-through areas, change shape, and animate. This allows more natural construction of levels. If you can draw it on paper, you can see it in the game.

Light Diminishing/Light Sourcing

Another touch adding realism is light diminishing. With

distance, your surroundings become enshrouded in darkness. This makes areas seem huge and intensifies the experience. Light sourcing allows lamps and lights to illuminate hallways, explosions to light up areas, and strobe lights to briefly reveal things near them. These two features will make the game frighteningly real.

Variable Height Floors and Ceilings

Floors and ceilings can be of any height, allowing for stairs, poles, altars, plus low hallways and high caves-allowing a great variety for rooms and halls.

Environment Animation and Morphing

Walls can move and transform in DOOM, which provides an active-and sometimes actively hostile-environment. Rooms can close in on you, ceilings can plunge down to crush you, and so on. Nothing is for certain in DOOM.

To this Id has added the ability to have animated messages on the walls, information terminals, access stations, and more. The environment can act on you, and you can act on the environment. If you shoot the walls, they get damaged, and stay damaged. Not only does this add realism, but provides a crude method for marking your path, like violent bread crumbs.

Palette Translation

Each creature and wall has its own palette which is translated to the game's palette. By changing palette colors, one can have monsters of many colors, players with different weapons, animating lights, infrared sensors that show monsters or hidden exits, and many other effects, like indicating monster damage.

Multiple Players

Up to four players can play over a local network, or two players can play by modem or serial link. You can see the other player in the environment, and in certain situations you can switch to their view. This feature, added to the 3-D realism, makes DOOM a very powerful cooperative game and its release a landmark event in the software industry.

This is the first game to really exploit the power of LANs and modems to their full potential. In 1993, we fully expect to be the number one cause of decreased productivity in businesses around the world.

Smooth, Seamless Gameplay

The environment in DOOM is frightening, but the player can be at ease when playing. Much effort has been spent on the development end to provide the smoothest control on the user end. And the frame rate (the rate at which the screen is updated) is high, so you move smoothly from room to room, turning and acting as you wish, unhampered by the slow jerky motion of most 3-D games. On a 386sx, the game runs well, and on a 486/33, the normal mode frame rate is faster than movies or television. This allows for the most important and enjoyable aspect of gameplay-immersion.

An Open Game

When our last hit, WOLFENSTEIN 3D was released the public responded with an almost immediate deluge of home-brewed utilities; map editors, sound editors, trainers, etc. All without any help on file formats or game layout from Id Software. DOOM will be release as an OPEN GAME. We will provide file formats and technical notes for anyone who wants them. People will be able to easily write and share anything from their own map editors to communications and network drivers.

DOOM will be available in the third quarter of 1993.

Appendix E

Source Code Release Notes

In December 1997, the much awaited source code of DOOM engine was finally released to the public. John Carmack wrote a few words for the event.

Here it is, at long last. The DOOM source code is released for your non-profit use. You still need real DOOM data to work with this code. If you don't actually own a real copy of one of the DOOMs, you should still be able to find them at software stores.

Many thanks to Bernd Kreimeier for taking the time to clean up the project and make sure that it actually works. Projects tends to rot if you leave it alone for a few years, and it takes effort for someone to deal with it again.

The bad news: this code only compiles and runs on linux. We couldn't release the dos code because of a copyrighted sound library we used (wow, was that a mistake -- I write my own sound code now), and I honestly don't even know what happened to the port that microsoft did to windows.

Still, the code is quite portable, and it should be straightforward to bring it up on just about any platform.

I wrote this code a long, long time ago, and there are plenty of things that seem downright silly in retrospect (using polar coordinates for clipping comes to mind), but overall it should still be a usefull base to experiment and build on.

The basic rendering concept -- horizontal and vertical lines of constant Z with fixed light shading per band was dead-on, but the implementation could be improved dramatically from the original code if it were revisited. The way the rendering proceeded from walls to floors to sprites could be collapsed into a single front-to-back walk of the bsp tree to collect information, then draw all the contents of a subsector on the way back up the tree. It requires treating floors and ceilings as polygons, rather than just the gaps between walls, and it requires clipping sprite billboards into subsector fragments, but it would be The Right Thing.

The movement and line of sight checking against the lines is one of the bigger misses that I look back on. It is messy code that had some failure cases, and there was a vastly simpler (and faster) solution sitting in front of my face. I used the BSP tree for rendering things, but I didn't realize at the time that it could also be used for environment testing. Replacing the line of sight test with a bsp line clip would be pretty easy. Sweeping volumes for movement gets a bit tougher, and touches on many of the challenges faced in quake / quake2 with edge bevels on polyhedrons.

Some project ideas:

Port it to your favorite operating system.

Add some rendering features -- transparency, look up / down, slopes, etc.

Add some game features -- weapons, jumping, ducking, flying, etc.

Create a packet server based internet game.

Create a client / server based internet game.

Do a 3D accelerated version. On modern hardware (fast pentium + 3DFX) you probably wouldn't even need to be clever -- you could just draw the entire level and get reasonable speed. With a touch of effort, it should easily lock at 60 fps (well, there are some issues with DOOM's 35 hz timebase...). The biggest issues would probably be the non-power of two texture sizes and the walls composed of multiple textures.

I don't have a real good guess at how many people are going to be

playing with this, but if significant projects are undertaken, it would be cool to see a level of community cooperation. I know that most early projects are going to be rough hacks done in isolation, but I would be very pleased to see a coordinated 'net release of an improved, backwards compatible version of DOOM on multiple platforms next year.

Have fun.

John Carmack
12-23-97

Appendix F

doombsp Release Note

In May 1994, the source code of doombsp was released. Like for other releases, John Carmack wrote a short note.

The source code for the binary space partitioner we used for DOOM is now available at `ftp.uwp.edu: /incoming/id/doombsp.zip`.

The catch is that the source has a few objective-c constructs in it, so it will take some work to port to dos. The only thing that will be a hassle is replacing the collection objects, the majority is just straight C.

This code was written and extended, not evolved, so it probably isn't the cleanest thing in the world. Please, PLEASE, PLEASE do not ask for support on this. I have far too much occupying my time as it is.

Our map editor does NOT work on wad files. It saves an ascii text representation of the file, then launches doombsp to process that into a wad file. I have included the input and output for E1M1, so you can verify any porting work you perform.

Having two programs allowed us to separate the tasks well under NEXTSTEP, but people working on dos editors will probably want to integrate a version of the bsp code directly into the editor.

If you are creating new DOOM maps for other people to use, we would appreciate it if the wadfiles you create use a PWAD identifier at the start of the file instead of the normal IWAD. This causes DOOM to tell

the user that they are playing a modified version, and no technical support will be given.

If you are creating a map editor for distribution to other people, contact Jay Wilbur (jayw@idsoftware.com) about getting a license agreement for the use of our trademark, etc. Its not a money issue, just some legal jazz.

BTW, there IS a bug in here that can cause up to a four pixel wide column to be drawn out of order, causing a more distant floor and ceiling plane to stream farther forward than it should. You can sometimes see this on E1M1 looking towards the imp up on the ledge at the entrance to the zig zag room. A few pixel wide column of slime streams down to the right of the walkway. It takes a bit of fidgeting with the mouse to find the spot. If someone out there tracks it down, let me know...

Have fun!

John Carmack
Id Software

Appendix G

Survivor's Strategies & Secrets

In 1994, for the publication of "Doom Survivor's Strategies & Secrets" by Jonathan Mendoza, three members of id Software wrote essays about their area of expertise. John Carmack wrote about the engine, Sandy Petersen about the map design, and Kevin Cloud wrote about the art. These are reproduced here with authorization from Jonathan Mendoza.

G.1 John Carmack

G.1.1 GOALS

High Speed Doom is an interactive game, so it should be played at a rate of ten frames per second or faster. Our target audience is people with 386/33 and faster machines. With the selectable detail and screen size, slower computers can trade visual fidelity for more usable speed. At the high end, a fast Pentium machine can run Doom at 35 fps, its maximum speed, under most circumstances.

Freeform World Geometry All of our previous games had been "tile-based," which means the world was divided up into fixed-sized blocks that are chosen from a palette of pre-created data. The advantage of tile-based worlds is that you can create them quickly and easily, by repeating simple tiles many times. But making levels with many unique areas or with angled corridors can require thousands of small tiles of geometry. We wanted the ability to design levels without being constrained to a block-based world.

Infinite View Distance Most 3-D games follow the principle that only the objects within a certain distance are considered for drawing. This simplifies the rendering algorithms, but it forces the viewing horizon to either fade out rapidly (Ultima Underworld/Shadowcaster) or

suffer the disconcerting effect of having objects pop into view (most flight simulators and driving games) rather than appear as a distant speck and grow larger.

G.1.2 IMPLEMENTATION

The work of programming Doom can be divided into four roughly equal parts:

- Developing the rendering engine to draw pictures of the world environment.
- Developing the utilities used to create data for the game.
- Developing the world model that governs the interaction of things in the game world.
- Tuning and modifying the code as new circumstances arise.

The main game code consists of just under 30,000 lines of C code. The DOS version has three functions in assembly language: horizontal texture map, vertical stretch, and read joystick. The sound code was developed by an outside contractor. A fundamental aspect of our development strategy is that we use NEXTSTEP systems for almost all programming work. The powerful, stable development environment has enabled us to do much richer work than if we had restricted ourselves to working under DOS.

The game is structured so that it can be run in a window under NEXTSTEP, where it can be easily debugged, or recompiled to run full-screen under DOS. The rendering engine was actually developed mostly on a black-and-white NeXTStation at my home. It was structured so that the graphics could be drawn in grayscale, eight-bit color, or twelve-bicolor (native to color NeXTstations). The refresh can also be used at any resolution, not limited to the PC screen size. Imposing the discipline of developing portable code has led me to some insights about better game architecture.

I usually categorize game rendering engines on three axes: speed, capabilities, and image fidelity. Speed is the relation of view window size to frame rate. Capabilities covers limitations to the world model, like 90 degree walls only, sloping floors, variable lighting, view height variability, etc. Fidelity includes the accuracy of texture mapping, any fudging done to improve speed, and things like anti-aliasing.

Our game design starts by selecting a speed for the game on our target platform, then trying to get as many capabilities and as high fidelity as possible. Doom's world geometry is limited to a two-dimensional arrangement of lines representing walls, and flat floors and ceiling of variable heights. Doom cannot draw sloping floors, overlapping walkways, or tilted walls. The viewpoint has four axes of freedom: forward/backwards, left/right, up/down, and clockwise/counterclockwise. These are significant limitations for, say, an architectural walk-through program, but they provided us with a great deal of freedom for our game design. We are still finding new ways to exploit these capabilities as work progresses on Doom II. I am proud of the fidelity of the Doom engine. The texture mapping is

subpixel-accurate, and there are no compromises with distance.

Because of the geometric limitations imposed on Doom, the hidden-surface removal problem can be reduced to a two-dimensional problem dealing only with the walls. The floors and ceilings are filled in to the remaining spaces after the walls have been properly drawn. This is a lot quicker than an arbitrary three-dimensional rendering scheme. The central algorithm Doom uses for the hidden-surface removal is a two-dimensional binary space-partition tree traversal. After a map has been drawn, it is passed to a separate utility which groups lines into sectors and recursively partitions the entire map into convex areas. This is a time-consuming task, but doing the work ahead of time lets the game perform less work at run time. The downside of this is that the lines that make up the world cannot have their endpoints adjusted during play. That's why there are no swinging doors in Doom.

Our map editor was used day-in, day-out for almost a year by our game designers, so the effort expended on making it productive to use was well spent. DoomEd is the NEXTSTEP application we created to build and modify worlds. It allows us to design the geometry of the world from a top-down perspective, and select the graphics to map onto the walls, floors, and ceilings.

The game world model was developed to support networking from the start. Each object in the world is processed through the same routines, regardless of whether it is a bonus item, a monster, or a networked player. Some of the world utility routines, like the bullet target and trace call, were actually more involved than the 3-D rendering routines.

The tuning of the entire project is the most important phase for making an enjoyable game. Subtle elements like the timing of an animation, the pitch of a sound effect, or the motion of an exploding body all impact the impression a user gets from the game. Proper tuning takes a long time, and a lot of testing, but the details really count. We experienced an interesting synergy in Doom, where several of the game elements regarding movement, combat, and the environment managed to complement each other so well that the game turned out better than our original vision of it. The normal process of game design starts with a glorious vision that is slowly torn down to reality as the project progresses. While our original plan was greatly changed, and some features were lost, the final product exceeded our early expectations.

G.1.2.1 AFTERTHOUGHTS

Doom is the first project I have worked on that I have still been proud of at its completion. I was not happy with Wolfenstein by the time it was released, and I was disappointed with my implementation of the Shadowcaster engine. I see a few warts, but I am still pleased with my work on Doom. There are a few remaining bugs in the refresh code that are unlikely to get fixed. Sometimes you will see a one-pixel-wide column stretching from the top

to the bottom of the screen. This is a result of drawing a line that has its two endpoints transformed to almost exactly the same polar angle. The fixed-point arithmetic that calculates the scale for the column sometimes overflows, and the column goes to the maximum possible scale of 64 times normal height. Cuts on the floor or ceiling that are nearly vertical will also sometimes show an error.

There is a roundoff error in the map partitioner that can pixel-wide segment of a line to be drawn in the wrong order in a narrow the strip of the floor and ceiling texture being drawn pas. line that should have stopped it. Some of the artwork was drawn wider than the real width of the game object, so it is possible to have a piece of a sprite be seen past a wall under certain circumstances. I could have made Doom about 15 percent faster by paying more attention to the low-level Intel architecture. We all learned a lot during the development of Doom, and there are many new things for us to bring on in the next project. Watch out!

G.2 Sandy Petersen

Doom's levels were not designed by one single person. John Romero created all the levels in episode 1, Knee Deep in the Dead, from scratch, except for level 1.8. All the remaining levels were done by me, either alone or sometimes by converting someone else's earlier work into a more polished form. The following paragraphs give full credit for the remaining levels.

Though a great deal of changes were made to Tom Hall and Shawn Green's levels (one a former id Software contributor, the other still with id Software), including placing monsters, repairing wall textures, and altering number-less small details, the basic architecture remains unchanged.

It is my belief that a perceptive player of Doom will sense a definite personality difference between the levels created by each of the designers. This effect may be slightly dimmed in the case of Tom Hall and Shawn Green, since their original distinctive style has been somewhat merged with my own through the heavy editing their levels underwent.

John Romero's particular lunacy appears to lie in flooding the player with seemingly unstoppable hordes of monsters, interspersed with long periods of tense quietude, as the player ponders what horrors are to be unleashed next. He often places monsters on distant vantage points, whence they can snipe at the player in relative safety. John's levels are riddled with special vantage points, cunning secret areas, and multilevel action.

John almost always starts out a level with a nightmarish bloodbath to get the player's adrenaline flowing. Only after you have survived this onslaught can you take a break and decide where to go next. Another tendency of John's is to make the level linear. if you

don't count the many secret passages, you pretty much have to go through John's levels in the order he prescribes.

On my own levels, I tend to present the player with a constant trickle of monsters, unlike John's episodic bursts of terror. Also, instead of John's diabolic secret tunnels and platforms, I tend to assault the player with booby traps and snares. The classic example is the false exit on E2M6. It looks like an exit, it smells like an exit, but it's not really an exit.

My levels start out kind of quiet, with the player left on his or her own. There's usually a monster or two right around the corner, but not the slavering horde you may have learned to expect from John. Some of my levels are quite linear (E3M1 or E3M4, for instance), but others, such as E3M2, E2M5 and E3M6, leave the area wide open for players to explore almost anywhere they want. I've found that some players really like this type of free-form experience, while others feel lost and confused until they manage to figure out the right way to go (which generally varies from player to player).

Three things must be kept in mind at all times while designing levels for the players:

1. How does it look?
2. Is it fun?
3. Did you remember to clean up?

G.2.1 How Does It Look?

This was the hardest part of level design to learn, at least for me. It seemed to come naturally to Romero, while I had to work and work. The basic problem is that in order to design a good-looking room for Doom, you must think architecturally. That is, you must see the room in terms of spaces rather than as a set of lines on a map. The exact wall textures used to give animation and color to a room often are secondary to the room's actual structural components. Some rooms end up looking very good indeed, while others are not as impressive, despite colors and structures. For instance, we've never been really happy with the large entry hall on E1M4. It does the job and is fun to play in, but it just doesn't seem to have that zip. An open hole in the roof and numerous alterations in the Chamber's decor didn't wholly fix it. In the end, we decided that it played just fine, so we'd leave it and move on to other things.

In the early design of Doom, there was a tendency to have lots of twisty little mazes. As playtest began, we discovered that these usually weren't too much fun, and most of them have been discarded (with a few exceptions, mostly in Tom Hall's old levels). Even the ones that remain have been altered and simplified in most cases, or serve a purpose by being claustrophobic and frightening (for instance, check out the excellent final maze in E1M4, the upstairs maze in E3M3, or the lava maze in E3M7).

G.2.2 Is It Fun?

This is, of course, even more important than making the game look good – if it doesn't PLAY well, it just doesn't matter how good it looks. Making a level fun, for me, was a combination of an initial overall plan and continual playtest.

When I began a level, I thought long and hard about the overall theme of that level – what the player was supposed to get from it. For example, on E3M5, I wanted to give the player an illusion of a vast fane or temple, with a symmetric and understandable architecture. At first, players on this level are puzzled, with the teleporters, released monsters, and so forth, but soon they understand the level's overall structure and are racing round it with ease. Once players comprehend the layout, they are able to approach E3M5 scientifically and rationally, which gives them an interesting contrast (I believe) between the emotionally laden nature of that level (a huge cathedral) and their own behavior.

On the other hand, on E3M1, the goal was simply to overawe the player with the wonders that await them in Hell. The level teems with ominous and frightening images from the start, where you find yourself outside under a glowering red sky, chased by Imps. When you open the promising door to escape, it releases a Cacodemon. The bridge leading to the shotgun collapses, etc. You are kept running around, seeing ever more ominous and weird sights and terrors that quickly teach you the different nature of Hell, as compared with the more rationally constructed levels of episodes 1 and 2.

Once I've got my theme worked out, I'll generally complete one small area of a level, then quickly playtest it. If it seems to work and looks fine I'll complete the next area, and test both completed areas out together, continuing to do this until the entire level is finished.

G.2.3 Did You Remember to Clean Up?

Just because a level looks good and plays well, doesn't mean it's done. Now I've got to make sure I've thought of everything. Is there enough ammunition and weapons for the players? How about bonus items? Players expect them, and they're easy to leave out. Did you remember to mark secret areas? And are there enough traps and tricks to keep the players amused?

After trying to cover these pathetic tiny details, I have to probe deeper. Is there some way that a clever player can bypass all the action of the whole level? If so, is that okay? (Sometimes it is – if you are smart you can skip almost the whole of E3M6, and I don't mind a bit; but you'll miss out on a lot of weapons and interesting combat.) How does the level mesh with the one preceding? The one following? If the start room of level B includes a single lonely Cacodemon coming at you down a long hall, but you were given the opportunity to pick up a rocket launcher in the exit room of level A, you aren't presented with much of a

problem. On the other hand, if that rocket launcher is at the very start of level A, so that you only have it available in level B if you carefully held onto your rockets, this might be okay – you should be rewarded for your stinginess.

When the final level is done, I play it a few more times, looking for flaws and mistakes (I find plenty of these), then I turn it over to those rotten excuses for human beings, the other id-iots at id Software. They quickly find all sorts of terrible things wrong with my poor baby level, and I fix these as rapidly as possible to avoid the rancorous comments and snide laughter that results when they expose flaws I've built into an area. If I sound like a bitter man, there are reasons.

G.3 Kevin Cloud

In games, good computer art is commonly referred to as "beautifully rendered or detailed" because most good game art looks meticulously hand drawn. Unfortunately, beautifully rendered worlds often begin to look staged. For Doom, we wanted to create a realistic and dark world that looked more dirty than pretty. There was nothing beautiful about Doom and we wanted its world to convey that concept – scary and dark.

To achieve the intended effect we used a combination of scanned and hand-rendered images. John Carmack created the program, Fuzzy Pumper Palette Shop, that would capture live video images and convert them into a PC graphic format. We then loaded the images directly into our PC art applications where we could edit, resize, colorize, and combine them – whatever it took to create an interesting graphic. The overall effect is somewhat distorted, but that's Doom.

The characters in Doom were created using a variety of methods – hand drawn, scanned clay models, and finally, latex and metal models. After working on Wolfenstein, we knew the frustration of creating the rotated views of every animation of a creature. Most characters are easy to draw from the front, but rotate them 45 degrees and things become a little more complex. Using small wooden mannequins and a couple of pounds of clay, we set out to make our own models. This technique wasn't perfect, but it enabled us to pose the creatures in stances we would normally not draw.

As the project neared its end, we wanted to create a monster that wasn't a biped. We came up with the idea of a large brainy creature with a chaingun embedded into its face, and its body attached by several large metal hooks to a four-legged metal machine. We couldn't create this guy using clay, and that's when we contacted Gregor Punchatz.

Gregor has an extensive background in creating models. Working on the sets of such movie classics as Nightmare on Elm Street and Robocop, Greg had the tools and the talent necessary for creating models. Within a few weeks, Gregor had turned our sketches into a

fully moveable monster. The process worked well. And although this version of Doom only features one of his creations, the retail version of Doom will fully utilize Gregor's talents.

G.3.1 HAPPY DOOMING

There is obviously a lot more to Doom than what you see on the screen, and there is much more the Doom creators could have shared with us. I hope the above discussions have at least given you an insight into the minds of the Doom creators and an appreciation for the art and science of Doom. For more technical information about playing and enjoying Doom, look for your specific topics in the appendices.

Appendix H

Interview with Dave Taylor

Dave Taylor was kind enough to allow an interview in June of 2017.

H.1 Q & A

Q: How old were you in 1993 when you started working at id?

A: I started studying the Sega Genesis tech docs in order to do a Sega Wolf3D port at the beginning of summer 1993, but by the end of summer when I started, I was assigned to Doom instead. I was 24 when I started.

Q: How did you get a job at id Software? It wasn't yet the powerhouse it became but they were already successful with Dave and Wolf3D games so I assume there must have been competition for the position?

A: I was studying electrical engineering at UT Austin and working as a journalist for an early electronic game magazine that came on floppies called Game Bytes. Wolf3D had come out, and I had interviewed the whole id team on a speakerphone call. There was a really friendly voice who would turn out to be Jay and a really knowledgeable voice with all the answers to my technical questions, who would turn out to be John Carmack. The summer before I took my senior lab, I emailed John to see if I could come up and interview.

I had been organizing very ambitious programming contests for the IEEE called the IEEE CS National Programming Contest, where we would develop a 3-on-3 multiplayer game in secret for Unix workstations, and then teams of 3 programmers would show up from 16 fancy schools (Stanford, MIT, Berkeley, Caltech, etc), we would reveal the game, and they'd have about 16 hours to write AI to play the game on their behalf.

At the end, we would do an exhibition game where all 16 teams of 3 players (48 players total) would do a deathmatch.

I had more Unix and network code experience than the rest of the id team, but I was a real noob at game development. Doom was my first commercial game.

Q: How advanced was Doom development when you joined?

A: The core 3D gameplay window was there, most of the art was in, the single-player gameplay was almost all there. I integrated the sound code/effects, the automap, status bar, screen wipes, level transitions, and cheat codes.

Q: Who did you report to, how did you know what to work on?

A: I reported to John Carmack, but I wasn't easy to manage and would often do my own thing.

Q: I can see¹ you had a NeXT workstation on your desk. What did you use it for?

A: We used them to make the whole game, and that's what the level editor ran on. DOS 3.3 was our target OS, and DOS wasn't really a complete operating system (no sound or video drivers, for example, and no debugger to speak of), so it was pretty painful to debug on. NeXTStep was much faster and easier to iterate on.

Q: You wrote ports for IRIX, AIX, Solaris and Linux. Was that for both Doom and Quake ?

A: For Doom, ya. For Quake, Linux for sure, but the others, I can't remember.

Q: What editor did you use, how did you compile?

A: I used vi for editing code. I made a Makefile and just typed make, as you'd think.

Q: Could you detail how it was to work with IRIX, AIX and Solaris ?

A: Once I got the basic code down for Linux (I had a system at home), it was pretty similar for the other Unix platforms. AIX, Irix, and Solaris were all kinda foreign to me but of course all still Unix variants, and I realized that by offering ports to Doom, I could get free workstations, so... :)

¹in "A Visit to id Software" 1994 video released by John Romero

Q: Did you get the sound to work on all of them?

A: I separated the sound code into its own server. It would load the files itself, and then the game would just tell it over a socket to fire off sounds and update volume/pitch/etc. The Linux code would eventually get really optimized, as Linus hooked me up with the XFree86 guys, who added an extension to give me direct access to the framebuffer, and then on Quake, Linus gave me a much faster way to get directly to the sound card DMA buffer and to get the current DMA transfer location down to a somewhat chunky granularity.

I know I got sound working on Irix and that Irix support was why Doom made the rounds with so many CG/VFX type people in the film industry. I can't remember whether I got it working on AIX and Solaris. Sound wasn't a priority for Sun/IBM at the time.

Q: You reportedly fell asleep on the floor and your coworkers taped the outline of your body on the ground. Did that happen a lot?

A: I fell asleep on the floors a lot, which is why they got the sofa and had me test-drive it for comfort, but I only remember them taping my outline once. I believe it stuck around for a while though.

Q: When did you leave id software?

A: I believe I left in early 1996, just after qtest1 shipped.

Q: Leaving was a courageous decision. Among many things you could have made more money. I assume you dreamed of making your own game. Do you regret leaving so "early"?

A: Actually, not brave. I asked when I hired on to get some ownership in id, and after the 6mo trial period, they decided against giving out more ownership, they said because it hadn't worked out with a couple of previous folks, and they had an expensive buy/sell agreement for anyone who left. When they said ownership wasn't going to happen, I asked if I could invest in my own game company on the side, and they said yes, as long as it wasn't 3D and I wasn't coding on it. So I invested in Crack dot Com, and produced Abuse. After it shipped, I was starting to get royalty checks that were bigger than what I got from id in bonus checks (and they were very generous). I was becoming more interested in producing for Crack and less interested in coding on Quake, which was starting to feel like a brown Doom with fewer monsters and a less relatable theme, so I was really slowing down. Carmack noticed and said we should prollly part ways after Quake shipped, and I countered that I'd prefer to leave after qtest1 shipped.

I don't know if I could have made more money. I wasn't very fulfilled when I left, and it was affecting my work. I've also never been much of a fan of money. It tends not to correlate

all that well to what I value.

Q: How do you feel looking back on this period of your life?

A: I don't look back much. My mind is usually dwelling on the far-enough future that I'm regarded as weird in the present. Back then was no different. I was trying to turn them onto networked games like netrek, which had persistent accounts, and I remember that being met with indifference. I started using the .plan files, sort of a blogging precursor, and would spend a lot more time on irc than the others. My fascination with the Unix ports was considered largely to be a waste of time, but they were very tolerant of me.

Q: Are you still in touch with anybody from the team of '93?

A: Not much. I exchange emails with John Carmack every once in a while.

Dave gave an interview to "blankmaninc.com" in 2013. His answer to "Why did you end up leaving id Software?" was so funny that I had to include it here.

“ Cocktail of reasons. It wasn't common knowledge that John Carmack was one of the best coders in the game industry. I just thought I had a very, very small penis. I had an electrical engineering degree, and I was one of the more capable coders in my class. So it was really demoralizing that no matter how hard I worked, I could never pull off anything a fraction as impressive as John, and of course he only seemed to be accelerating from his already stunning clip. This led to a pattern of me pushing really hard, burning out, and then limping along for a while until I made my next futile attempt to approach a fraction of his awesomeness. There was this pattern of him saying, "Hey, check this out," and I'd follow him to his office, and he'd be levitating in his chair while demonstrating his elegant solution to an intractable problem of computer science, and my version of "Hey, check this out" was usually motivated by needing his help to track down an issue.

”

Appendix I

Interview with Randy Linden

Q: Can you give a little bit of background about yourself, how old you were when you developed DOOM for SNES, etc.?

A: DOOM/FX was published in 1995 and I was 25 y/o. I wrote it from San Diego, California but I'm Canadian, born and raised in Toronto, Ontario. My first program was published when I was 13. Here is a list of what I have worked on:

- Bubbles – A Centipede-style game for the Commodore 64.
- The 64 Emulator – An Amiga program that emulated the Commodore 64.
- Dragon's Lair – Another Amiga program that was the first full-screen full-color animated game on any home computer – Dragon's Lair was also the first game which streamed data in real-time from six floppy discs during gameplay!
- DOOM – A version of the popular "2-1/2D" FPS for Super Nintendo running on an original game engine designed for the SuperFX RISC processor (aka "GSU2A") designed by Argonaut Software in the UK. DOOM for SNES was one of the very few titles which worked with most of the SNES hardware accessories including the mouse and light gun. It also supported the XBAND hardware modem and two players could complete head-to-head in one of the truly rare online games for SNES. I built a custom development system (Assembler, Linker, Debugger, etc.) which used a modified (ie. hacked apart) StarFox game cartridge to provide the CPU and communication to the Amiga-based toolchain by using a serial-based interface which plugged into both joystick ports on the SNES. Support for Nintendo's official hardware development system for the GSU2A was added later in the development because no games had been released at that point which could be "modified" for use (StarFox used the original GSU which ran at half the clock rate of the GSU2A and had less memory among other hardware changes and enhancements.)

- "Bleem!" was a PC-based emulator for Playstation games – the program was entirely in x86. I reverse-engineered Playstation software and hardware, wrote a run-time optimizing recompiler for the game logic and added PC enhancements like higher resolution and 3D graphics card support.
- "Bleem!" for Dreamcast was a Playstation emulator written entirely in SH4 for the Sega Dreamcast. I was awarded four patents as a sole inventor for the technologies and techniques I developed for bleem! and bleem! for Dreamcast.
- Cyboid is a high-speed 3D FPS (first-person shooter) which features single- and two-player split-screen gaming and eight-player online multiplayer across a variety of play modes – think "Quake" and that's pretty much it. The game features VR support, In-App purchasing, Achievements and Leaderboards, Advertising for multiple mediators and Multiplayer Internet gaming for up to eight players online – all of which runs on multiple platforms from Google and Amazon. The game is written in ARM assembly, native C/C++ and Java and the entire package (code, data, graphics, sound and music) is eight megabytes (... smaller than a single graphics texture in games/apps these days!) The custom engines have all been highly optimized to support the widest range of devices possible, including lower-powered hardware (such as the Amazon Fire TV Stick) or older devices running Android API 17 "Jellybean".

Q: It seems the GSU-2 was not able to render fullscreen. I have read several theories online. Some mention a hardware limitation limiting the processor to 192 lines. Some mention bandwidth issues related to DMA (to read from the GSU RAM). Do you have more insight about this?

A: As you know, the Super NES ("SNES") was a character-based graphics architecture which used a "font" that defined each character's image and "map" which specified what character to display in each position on the screen.

In effect, the the "Super/FX" (aka GSU "Graphics Support Unit") enabled bitmap emulation on the SNES using a combination of fast, custom hardware and a carefully designed instruction set that was optimized for graphics processing.

The GSU was a truly incredible RISC custom chip designed by Jez San and Argonaut Software – It is one of my favorite hardware architectures of all time for many reasons, in particular, the elegance and simplicity of the opcodes were both powerful and efficient and the entire design is very similar to the architecture of the ARM processor (which is one of my other favorite hardware systems!).

BTW, I agree 100000% with Jez's comments about the GSU – absolutely true!!!

The original GSU, used in StarFox, ran at 10.74 Mhz, the GSU/2A used in DOOM/FX ran at 21.48Mhz.

All standard ALU operations were supported (add, subtract, eor, etc.), plus a variety of fast multiplication operations, multiple memory load/store operations (based on whether you were accessing ROM or RAM), a "LOOP" command and the "PLOT" command. Code could run from ROM, RAM or an on-board 512 byte cache (which DOOM/FX used extensively!)

What made the GSU really special was the large register set (16 general purpose registers compared to the 65816) plus the opcode prefixes "FROM/TO/WITH" which allowed you to specify a source register ("FROM") and destination register ("TO") or operate on the same register using "WITH" – This type of operation is very standard these days, particularly on the ARM architecture, but back then it was both unique and powerful!

Bandwidth was never an issue for me – here are two examples:

1. The "PLOT" command writes to the emulated bitmap using "X" and "Y" coordinates and a "COLOUR" and can update single pixels at a time. I wrote code which timed how fast the plot command executed, and discovered that since the underlying memory is still effectively a character-based array, each time you "plotted" to a new character "line" (8-pixels wide), the hardware fetched that character's memory (basically eight bytes that form the 256 color "plane") into an internal cache so that subsequent plots to the same character "line" executed much faster.

I used this knowledge in DOOM/FX by "pre-writing" multiple pixels the first time a "new line" was written – in effect, instead of plotting a single pixel at a time, I wrote multiple pixels using the same color so the hardware quickly updated the internal "character line" cache and then overwrote the same pixels with the correct individual colors at high speed (typically 1 clock per pixel!)

2. The GSU ran SO much faster than the SNES 65816, the vast majority of the game (close to 95%) is all GSU code – the 65816 is basically halted waiting for various interrupts to update memory, swap screens, read the joysticks, etc. – but otherwise the 65816 is pretty much idle!

The GSU had four different rendering heights (128, 160 and 192 pixels) plus an "OBJ" mode for sprites as well as three options for pixel "depth" (4, 16 or 256 colors).

DOOM/FX used the 192 pixel / 256 color mode.

Here's something most people don't know: DOOM/FX was the second-most expensive cartridge to manufacture because it included the GSU/2A, had the largest ROM (2 Megabytes) and RAM configuration available and a custom red plastic cartridge ... the only option missing was the battery backup!

Q: I speculated that you used the Unofficial Doom Specs to extract all the assets, am I correct?

A: I used a variety of online resources to reverse-engineer the DOOM data formats – without the tremendous amount of work from all those other people, DOOM/FX wouldn't have been possible in the amount of time!

Q: Can you describe the Reality engine in its big lines. I assume you used the BSP but did you also create the equivalent of visplanes?

A: The Reality Engine is basically a highly-custom 2^{-1/2}D game engine that's almost all written in GSU code. All the code is designed to run in small blocks so they fit into the GSU's internal cache which runs at high-speed. The architecture is similar to DOOM in that it uses a 2D BSP, sectors, segments, etc. – there are a number of optimizations that minimize the processing required to figure out what to draw, etc.

Q: Did you get any help from id Software at all?

A: I demo'd the game to Sculptured Software when I had a fully operational prototype running – graphics, texturing, movement, enemies, etc. – it was obviously "DOOM" to anyone looking at what I had running on the hardware.

Id Software was shown the game a few weeks later and even though it was an early version, there wasn't much left for them to help with since it was much further along than a typical prototype or "proof of concept."

Of course, there was still a lot of work to do before the game could be published ... for example: sound, music, the rest of the levels, testing, etc. – and it was with the help of multiple people at Sculptured Software that the game was finally done.

Q: How long did it take you to build the toolchain, then how long to code the game?

A: IIRC, the game was developed in roughly eight months, give or take. There were multiple toolchains used to create the game, graphics, sound and data...

The code was developed using a custom development system I wrote that included an assembler, linker, source-level debugger, etc. – I had written multiple projects with the development system (NES, SNES among others) and it was built over a number of years. Adding support for the SuperFX only took a couple weeks, though. I also wrote a number of tools and utilities to extract and convert the original assets from the PC game into the optimized formats for the Reality Engine. Sound and Music used a toolchain from Sculptured Software so all I had to do was convert the extracted assets into the appropriate formats for processing.

Q: How did you get in touch with the publisher? Did you try several publishers before Sculptured Software accepted it?

A: Sculptured Software was the only publisher I considered – I worked at Sculptured for a few years and knew many people there – all of whom were awesome!

Q: Was it difficult to comply with Nintendo "no violence, no blood" policy at the time?

A: Nintendo was incredibly easy to work with – they had very few requirements, minimal changes and it was an easy submission process in general.

Q: Anything you want to mention, looking back at this game/part of your life? (Some people lament that programming is more complex nowadays but I feel it was harder back then, just to even get the programming manuals).

A: IMO, programming these days is very different than the "old days..."

It was so much easier to write something for a single piece of hardware like the SNES – you could spend time focusing on the game and improving performance, etc.

The same was true of the Commodore 64, Amiga and Sega Dreamcast – each of which had unique technical aspects and advances which provided opportunities to make something stand out in ways that is much harder to do these days...

In comparison, writing an Android app is a huge amount of work for many reasons, many of which are out of the developers' control...

There's a lot of hardware out there (with many differences in what is/isn't supported as far as graphics, sound, input, etc.), there are multiple versions of the OS (each of which have their own "quirks", limitations, differences and changes) and although it's truly wonderful that Google provides comprehensive components like Firebase and Google Play, the frequent updates, changes and differences require a huge amount of time and overhead just to keep your app "current"

These days it's fairly painless and even easy to create a simple "app" that works on a bunch of Android devices ... but writing something which takes advantage of the unique aspects of as many different devices as possible takes a lot of time, resources and effort – and I think that's why there are so many similar games, apps and titles out there – sure, there's uniqueness in many of them, but far less than there used to be – we don't see things like "Lode Runner", "Archon" or "Sword of Sodan" anymore.

A friend recently recommended a FPS that was over 1Gig to download! ... and then the game required a constant internet connection for updates, resources and even to play the game... we've sure come a long way from Dragon's Lair on the Amiga which was an 8K program and 5 Megs of data.

More about how the GSU-2 worked:

A: Basically the GSU generates a bitmap into its own RAM that is then transferred to the SNES PPU for display. I think there wasn't sufficient clock cycles and RAM to generate a full screen worth of data and transfer it to the PPU in time to avoid tearing/glitching, etc. The GSU RAM can be accessed by only one device at a time (either the GSU or the SNES), but not both ... so you have to generate data into the GSU RAM, then transfer it to the SNES-side PPU ("Picture Processing Unit") so it can be displayed on screen. During the transfer, the SNES-side has access to the RAM and the GSU does not, so you have to wait until the transfer is complete before you can use the GSU to generate more data. The GSU can still do many other things without accessing the internal RAM, though – and DOOM/FX used that feature extensively. BTW, the same is true of the ROM – only one device can access the ROM at a time – either the GSU or the SNES.

Anecdotes from DOOM SNES development:

A:

1. At one time a developer at Sculptured had modified a bunch of the basic levels – added some really nice changes like object placement, etc. – and after showing the new levels to id Software, they basically wanted the game to be as true to the original as possible... so we ended up reverting all the changes and keeping the original levels with as few modifications as possible.

One thing that id Software let me keep was the auto-rotating overhead map.

2. The engine had a number of configurable options – many of which weren't used in the final game ... some examples:
 - (a) The walls could be drawn single-pixel and not doubled, but there weren't enough clock cycles for a reasonable framerate
 - (b) Floors could also be drawn – as single or double pixels – but there wasn't enough ROM to store the textures and it was much faster to use colours/dithering instead.

3. The timing and logic to generate the display was fairly complicated... as you'll see:

I basically generate the display in "thirds" due to the limited amount of RAM available in the GSU and PPU.

The "PPU" is the SNES-side "Pixel Processing Unit" – it's basically the portion of the hardware which fetches memory and displays it on screen.

Each "third" is basically generated in three "steps":

- (a) Calculate all the necessary data/etc.
- (b) Generate the graphic output into GSU RAM
- (c) Transfer the GSU RAM to PPU.

There isn't enough PPU memory to store two complete game frames (one for the current game frame and a second for the next game frame) ...

To solve this, I divided the SNES-side PPU memory into five "portions", three of which are required to display a single frame (obviously) since the PPU has to be able to access that data to show it on screen.

The remaining two "portions" are where I store the first and second "thirds" of the next game frame... but what about the last third?

The final third of the next game frame is more complicated because it will be transferred overtop of one of the "active" thirds which the PPU is still using to show the current game frame!

When the GSU has finished generating the last third, I wait for a raster interrupt that ensures the PPU is NOT showing the area which is used by the "common" third and transfer the last portion overtop before the raster hits that area of the screen.

After the final transfer is complete, I reconfigure which of the five portions is used by the PPU to display the new game frame and start the whole process again, generating the next game frame into the two "portions" that are now available and updating the final third in the same way as before.

Of course, if the game was running at 60fps, I could just generate everything on-the-fly and wouldn't need all the complicated code!

Sound was also a bit tricky, for similar reasons ... but not nearly as complex. There wasn't enough RAM for all the sound effects, so for example, the weapon sound is transferred dynamically when you switch weapons – there's enough time to transfer the sound data for the weapon while the "weapon change" animation is running, but there's only one weapon sound in the SPU (sound processing unit) at a time.

4. Most of the graphics (textures, objects, etc.) are stored compressed internally and decompressed while drawing.

I used a simple encoding which reduced the memory requirements and since the GSU was fast enough, the tradeoff between processing cycles and storage was a good one.

OpenGL vs Direct3D .plan

John Carmack's .plan for Dec 23, 1996

I am going to use this installment of my .plan file to get up on a soapbox about an important issue to me: 3D API. I get asked for my opinions about this often enough that it is time I just made a public statement. So here it is, my current position as of december '96...

There are two viable contenders for low level 3D programming on win32: Direct-3D Immediate Mode, the new, designed for games API, and OpenGL, the workstation graphics API originally developed by SGI. They are both supported by microsoft, but D3D has been evangelized as the one true solution for games.

I have been using OpenGL for about six months now, and I have been very impressed by the design of the API, and especially it's ease of use. A month ago, I ported quake to OpenGL. It was an extremely pleasant experience. It didn't take long, the code was clean and simple, and it gave me a great testbed to rapidly try out new research ideas.

I started porting glquake to Direct-3D IM with the intent of learning the api and doing a fair comparison.

Well, I have learned enough about it. I'm not going to finish the port. I have better things to do with my time.

I am hoping that the vendors shipping second generation cards in the coming year can be convinced to support OpenGL. If this doesn't happen early on and there are capable cards that glquake does not run on, then I apologize, but I am taking a little stand in my little corner of the world with the hope of having some small influence on things that are going to effect us for many years to come.

Direct-3D IM is a horribly broken API. It inflicts great pain and suffering on the programmers using it, without returning any significant advantages. I don't think there is ANY market segment that D3D is appropriate for, OpenGL seems to work just fine for everything from quake to softimage. There is no good technical reason for the existance of D3D.

I'm sure D3D will suck less with each forthcoming version, but this is an oportunity to just bypass dragging the entire development community through the messy evolution of an ill-birthed API.

Best case: Microsoft integrates OpenGL with direct-x (probably calling it Direct-GL or something), ports D3D retained mode on top of GL, and tells everyone to forget they every heard of D3D immediate mode. Programmers have one good api, vendors have one driver to write, and the world is a better place.

To elaborate a bit:

"OpenGL" is either OpenGL 1.1 or OpenGL 1.0 with the common extensions. Raw OpenGL 1.0 has several holes in functionality.

"D3D" is Direct-3D Immediate Mode. D3D retained mode is a seperate issue. Retained mode has very valid reasons for existance. It is a good thing to have an api that lets you just load in model files and fly around without

sweating the polygon details. Retained mode is going to be used by at least ten times as many programmers as immediate mode. On the other hand, the world class applications that really step to new levels are going to be done in an immediate mode graphics API. D3D-RM doesn't even really have to be tied to D3D-IM. It could be implemented to emit OpenGL code instead.

I don't particularly care about the software only implementations of either D3D or OpenGL. I haven't done serious research here, but I think D3D has a real edge, because it was originally designed for software rendering and much optimization effort has been focused there. COSMO GL is attempting to compete there, but I feel the effort is misguided. Software rasterizers will still exist to support the lowest common denominator, but soon all game development will be targeted at hardware rasterization, so that's where effort should be focused.

The primary importance of a 3D API to game developers is as an interface to the wide variety of 3D hardware that is emerging. If there was one compatible line of hardware that did what we wanted and covered 90+ percent of the target market, I wouldn't even want a 3D API for production use, I would be writing straight to the metal, just like I always have with pure software schemes. I would still want a 3D API for research and tool development, but it wouldn't matter if it wasn't a mainstream solution.

Because I am expecting the 3D accelerator market to be fairly fragmented for the foreseeable future, I need an API to write to, with individual drivers for each brand of hardware. OpenGL has been maturing in the workstation market for many years now, always with a hardware focus. We have existing proof that it scales just great from a \$300 permedia card all the way to a \$250,000 loaded infinite reality system.

All of the game oriented PC 3D hardware basically came into existence in the last year. Because of the frantic nature of the PC world, we may be getting stuck with a first guess API and driver model which isn't all that good.

The things that matter with an API are: functionality, performance, driver coverage, and ease of use.

Both APIs cover the important functionality. There shouldn't be any real argument about that. GL supports some additional esoteric features that I am unlikely to use (or are unlikely to be supported by hardware --

same effect). D3D actually has a couple nice features that I would like to see moved to GL (specular blend at each vertex, color key transparency, and no clipping hints), which brings up the extensions issue. GL can be extended by the driver, but because D3D imposes a layer between the driver and the API, microsoft is the only one that can extend D3D.

My conclusion about performance is that there is not going to be any significant performance difference ($< 10\%$) between properly written OpenGL and D3D drivers for several years at least. There are some arguments that gl will scale better to very high end hardware because it doesn't need to build any intermediate structures, but you could use tiny sub cache sized execute buffers in d3d and acheive reasonably similar results (or build complex hardware just to suit D3D -- ack!). There are also arguments from the other side that the vertex pools in d3d will save work on geometry bound applications, but you can do the same thing with vertex arrays in GL.

Currently, there are more drivers available for D3D than OpenGL on the consumer level boards. I hope we can change this. A serious problem is that there are no D3D conformance tests, and the documentation is very poor, so the existing drivers aren't exactly uniform in their functionality. OpenGL has an established set of conformance tests, so there is no argument about exactly how things are supposed to work. OpenGL offers two levels of drivers that can be written: mini client drivers and installable client drivers. A MCD is a simple, robust exporting of hardware rasterization capabilities. An ICD is basically a full replacement for the API that lets hardware accelerate or extend any piece of GL without any overhead.

The overriding reason why GL is so much better than D3D has to do with ease of use. GL is easy to use and fun to experiment with. D3D is not (ahem). You can make sample GL programs with a single page of code. I think D3D has managed to make the worst possible interface choice at every opportunity. COM. Expandable structs passed to functions. Execute buffers. Some of these choices were made so that the API would be able to gracefully expand in the future, but who cares about having an API that can grow if you have forced it to be painful to use now and forever after? Many things that are a single line of GL code require half a page of D3D code to allocate a structure, set a size, fill something in, call a COM routine, then extract the result.

Ease of use is damn important. If you can program something in half the

time, you can ship earlier or explore more aproaches. A clean, readable coding interface also makes it easier to find / prevent bugs.

GL's interface is procedural: You perform operations by calling gl functions to pass vertex data and specify primitives.

```
glBegin (GL\_TRIANGLES);  
glVertex (0,0,0);  
glVertex (1,1,0);  
glVertex (2,0,0);  
glEnd ();
```

D3D's interface is by execute buffers: You build a structure containing vertex data and commands, and pass the entire thing with a single call. On the surface, this appears to be an efficiency improvement for D3D, because it gets rid of a lot of procedure call overhead. In reality, it is a gigantic pain-in-the-ass.

```
v = \&buffer.vertexes[0];  
v->x = 0; v->y = 0; v->z = 0;  
v++;  
v->x = 1; v->y = 1; v->z = 0;  
v++;  
v->x = 2; v->y = 0; v->z = 0;  
c = \&buffer.commands;  
c->operation = DRAW\_TRIANGLE;  
c->vertexes[0] = 0;  
c->vertexes[1] = 1;  
c->vertexes[2] = 2;  
IssueExecuteBuffer (buffer);
```

If I included the complete code to actually lock, build, and issue an execute buffer here, you would think I was choosing some pathologically slanted case to make D3D look bad.

You wouldn't actually make an execute buffer with a single triangle in it, or your performance would be dreadfull. The idea is to build up a large batch of commands so that you pass lots of work to D3D with a single procedure call.

A problem with that is that the optimal definition of "large" and "lots" varies depending on what hardware you are using, but instead of leaving that up to the driver, the application programmer has to know what is best for every hardware situation.

You can cover some of the messy work with macros, but that brings its own set of problems. The only way I can see to make D3D generally usable is to create your own procedural interface that buffers commands up into one or more execute buffers and flushes when needed. But why bother, when there is this other nifty procedural API already there...

With OpenGL, you can get something working with simple, straightforward code, then if it is warranted, you can convert to display lists or vertex arrays for max performance (although the difference usually isn't that large). This is the right way of doing things -- like converting your crucial functions to assembly language after doing all your development in C.

With D3D, you have to do everything the painful way from the beginning. Like writing a complete program in assembly language, taking many times longer, missing chances for algorithmic improvements, etc. And then finding out it doesn't even go faster.

I am going to be programming with a 3D API every day for many years to come. I want something that helps me, rather than gets in my way.

John Carmack
Id Software

Appendix K

Black Book Internals

It took ten months to write this book from October 2017 to July 2018. That was pleasant. Then it took three months to proof-read it from October 2018 to November 2018. That was very unpleasant.

I worked on it whenever I could find the time on evenings and weekends, but mostly favoring the 7am to 9am slot. The text was written in \LaTeX with Sublime Text 3.0 editor and compiled with `pdflatex`. The `.pdf` can be built with a single command.

```
pdflatex -o book.pdf book.tex
```

To improve compilation time, `book.tex` is a simple list of `\subfile` commands. Based on which section of the book was being worked on, other lines were commented out. This decreased the build time from several minutes to a few seconds and made \LaTeX bearable.

Drawings were made with Inkscape, saved to `.svg` and converted to `.eps` to incorporate nicely with `pdflatex` compiler. Screenshots were cleaned up with Adobe Photoshop CC. The font used is Computer Modern Sans Serif 10pts by Donald E. Knuth.

Description	Value
Number of files	880
Number of <code>.tex</code> files	95
Number of <code>.png</code> files	561
Number of <code>.svg</code> files	102
Build time (300 dpi)	2 minutes 20 seconds
Final PDF size (300 dpi)	387,054,465 bytes
Build time (150 dpi)	47 seconds
Final PDF size (150 dpi)	89,060,075 byte

