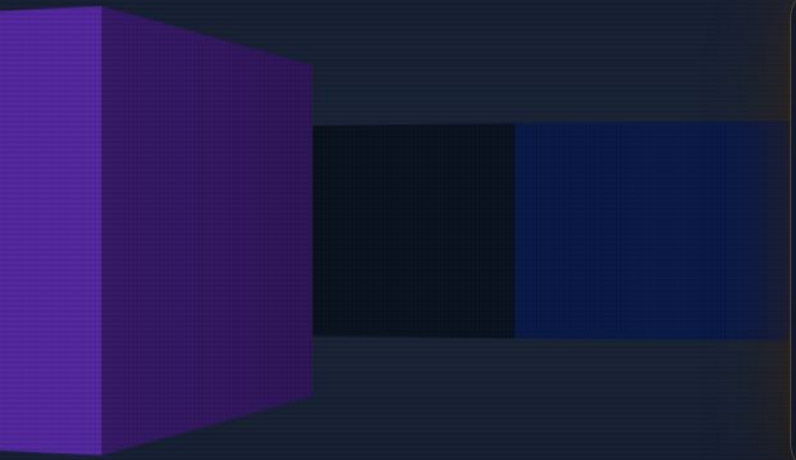


GAME ENGINE BLACK BOOK: WOLFENSTEIN 3D — FABIEN SANGLARD

Por Dentro da Engine

Capítulo 4 · Seções 4.1 a 4.3

Código-fonte, estrutura e arquitetura da engine que moldou os jogos em primeira pessoa

SEÇÃO 4.1

Getting the Source Code

Onde obter o código e o que isso significa para a história da computação

4.1 — O CÓDIGO ABERTO

O Arquivo Histórico

- Publicado pela **id Software** em **21 de julho de 1995**
- Distribuído pelo FTP oficial: **ftp.idsoftware.com**
- Arquivo: **wolfsrc.zip**
- Mais de **20 anos depois** — o mesmo endereço ainda funciona
- Raro dada a natureza mutável da web

Curiosidade: Quase nenhum link de 1995 sobrevive. Este sobreviveu.

4.1 — ACESSO MODERNO

FTP vs GitHub

- Em **2012** , a id Software migrou tudo para o **GitHub**
- Vantagens: mais rápido, mais confiável, histórico de commits visível
- Duas formas de obter o mesmo código

Hoje: `git clone git@github.com:id-Software/wolf3d.git`

SEÇÃO 4.2

First Contact

Abrindo o ZIP e medindo o que existe dentro

4.2 — ESTRUTURA DO CÓDIGO

O que está dentro

- ZIP contém **outro PKZIP auto-extraível** — prática comum em 1995
- Análise com **cloc.pl** — conta linhas por linguagem
- Total: **79 arquivos** , **27.223 linhas** de código
- Assembly com **TASM** (Borland) — notação Intel

Nota: id Software só adotou C++ "de verdade" a partir do **Doom 3** (~2000). Wolf3D é C com algumas extensões.

Linguagem	Arquivos	Blank	Comment	Código
C++	26	5.750	6.201	21.169
C/C++ Header	42	802	660	3.900
Assembly	10	669	732	2.150
DOS Batch	1	1	0	4
TOTAL	79	7.222	7.593	27.223

4.2 — COMPOSIÇÃO

90% C · 10% Assembly

- **C** para a lógica geral do jogo
- **Assembly** para gargalos de performance e I/O de baixo nível (vídeo, áudio)
- Assembly em **notação Intel** : destino antes da origem
- `instr dest, source`

> **Insight:** Assembly era o bisturi — usado só onde o compilador não gerava código rápido o suficiente.

C — 90% ASM — 10%

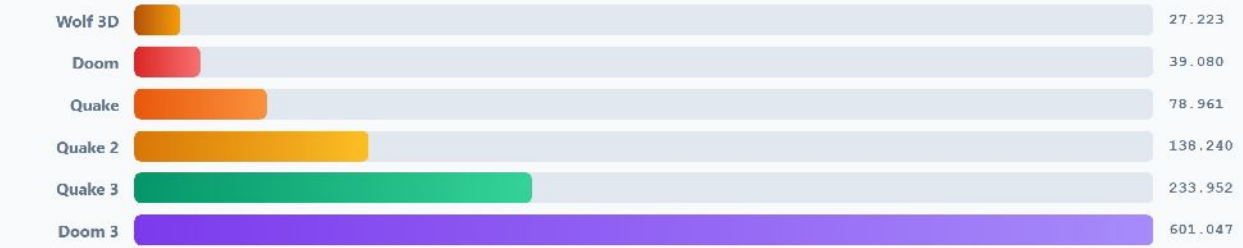
4.2 — ESCALA

SLOC em Perspectiva

Wolf3D cabe num final de semana de leitura. O Linux Kernel levaria anos.

4.2 — FIGURA 4.1

Evolução das Engines id Software



Wolf3D → Doom 3: crescimento ~22× em menos de 10 anos — quase exponencial.

4.2 — O LADO HUMANO

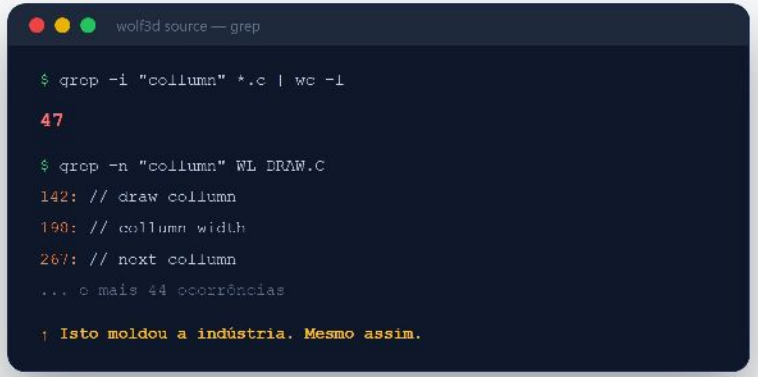
O Erro Histórico

- Editores de 1991 **não tinham corretor ortográfico**
- John Carmack escrevia mal — "collumn" aparece **47 vezes** no código
- O release foi muito bem recebido... mas alguns emails ficaram para a história

> *"It's **'COLUMN'**, you dumb FUCK!"*

— Email recebido após o release do código-fonte

O código que moldou a indústria. Com erro ortográfico e tudo.



SEÇÃO 4.3

Big Picture

A visão geral da engine: três subsistemas e como eles se articulam

4.3 — FIGURA 4.2 · BIG PICTURE

Arquitetura dos Três Subsistemas

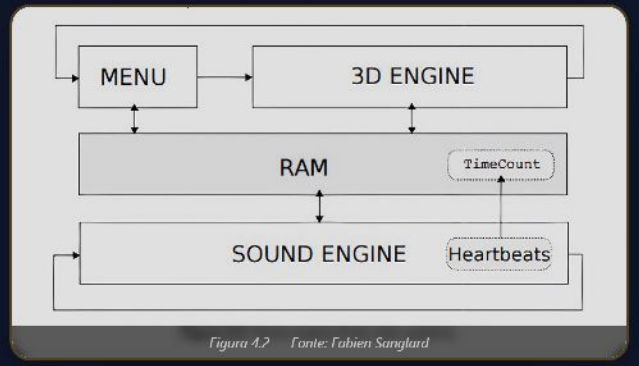
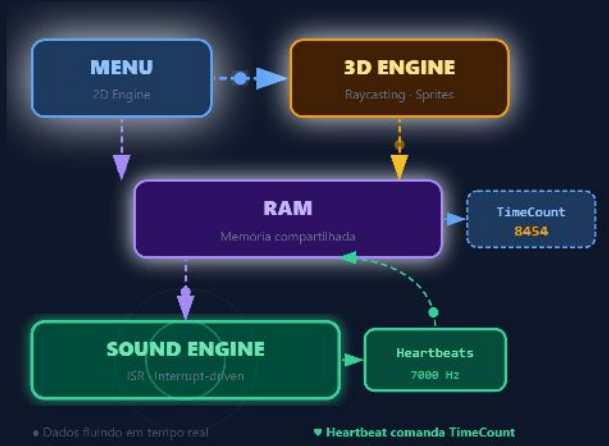


Figura 4.2 Fonte: Fabien Sanglard

- **MENU** → **3D ENGINE** via RAM
- **SOUND ENGINE** faz o heartbeat de toda a engine
- **TimeCount** sincroniza os renderers



SEÇÃO 4.3.1

Unrolled Loop

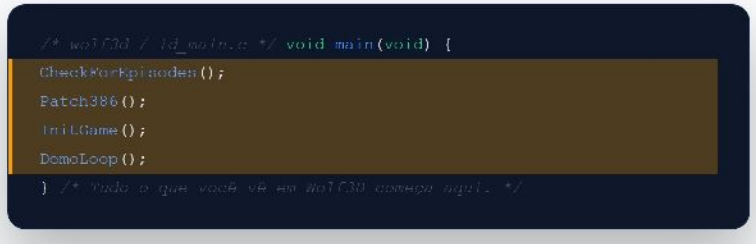
O loop principal da engine — ponto de entrada no código

4.3.1 — PONTO DE ENTRADA

void main()

- Toda a complexidade está **delegada** — main() tem apenas 4 linhas
- **CheckForEpisodes()** — verifica qual episódio está disponível
- **Patch386()** — auto-modifica o código em runtime
- **InitGame()** — sobe todos os managers e subsistemas
- **DemoLoop()** — o coração do jogo, loop infinito

main() real tem 4 linhas. O restante é abstração perfeita.



4.3.1 — ARMADILHA

Real Mode: 16-bit

- Target: **real mode** — código compilado com instruções **16-bit**
- `int` e `word` → **16 bits**
- `long` e `dword` → **32 bits**
- Operações com **long** usavam biblioteca Borland — lenta
- O 386 já era 32-bit, mas o compilador não explorava por padrão

Solução: Patch386() — o jogo reescreve o próprio código em runtime se detectar um 386!

4.3.1 — PATCH386()

Auto-Modificação em Runtime

- Wolf3D detecta em **runtime** se a CPU é um **386**
- Se sim: **reescreve o próprio código** na memória
- Substitui chamadas lentas da Borland por instruções 32-bit nativas
- Resultado: divisões muito mais rápidas

Hoje isso seria um red flag de segurança. Em 1992, era engenharia necessária.

```
// Código ANTES do Patch386()
; Lib Borland
push ax
push bx
call __ldiv ; lento
add sp, 4

// Código DEPOIS do Patch386() :
mov eax, [bp+8]
cdq
idiv [DWORD PTR bp+12] ; rápido
mov edx, eax
shr edx, 16
```

✓ Código Patchado!

4.3.1 — INITGAME()

Subindo os Managers

- `MM_Startup()` — Memory Manager
- `SignonScreen()` — Tela inicial
- `VW_Startup()` — Video Manager
- `IN_Startup()` — Input Manager
- `PM_Startup()` — Page Manager
- `SD_Startup()` — Sound Manager
- `CA_Startup()` — Cache Manager
- `US_Startup()` — Font Manager
- `BuildTables()` — sin/cos/view
- `SetupWalls()` — lookup texturas

Ordem importa: cada manager pode depender do anterior. Memory Manager é sempre o primeiro.

4.3.1 — DEMOLOOP()

O Coração do Jogo

- Loop **infinito** — só termina com mensagem de erro
- **GameLoop()** = renderer 2D (menus, configuração)
- **PlayLoop()** = renderer 3D (ação do jogo)
- Intercala telas de título, créditos, demos e highscores
- Última linha: `Quit("Demo loop exited???")` — nunca deveria ser atingida

```
void DemoLoop() { StartCPMusic(INTROSONG); PG13(); // tela Profound
Carnage while (1) { CA CacheScreen(TITLEPIC);
CA CacheScreen(CREDITSPIC); DrawHighScores(); PlayDemo(0);
GameLoop(); // 2D — menus
SetupGameLevel(); StartMusic(); PreloadGraphics();
PlayLoop(); // 3D — ação
StopMusic(); } Quit("Demo loop exited???"); }
```

4.3.1 — PLAYLOOP()

O Renderer 3D

- Padrão clássico: **Input → Update → Render**
- **PollControls()** — captura teclado/joystick
- **DoActor()** — IA de cada inimigo
- **ThreeDRefresh()** — raycasting: teto, paredes, sprites
- **UpdateSoundLoc()** — som estéreo posicional

4.3.1 — SOUND SYSTEM

SDL_ ≠ SDL!

- Sound Manager iniciado por **SDL_SetTimerSpeed()**
- **Atenção:** o prefixo SDL_ aqui **NÃO** é a Simple DirectMedia Layer
- Em 1991, essa biblioteca nem existia
- Aqui: **SDL_ = SounD Low level**
- Sem SO com threads — única saída: **interrupções de hardware**
- ISR instalada na **IVT (Interrupt Vector Table)**

Coincidência histórica: o prefixo SDL_ foi reutilizado décadas depois por outra biblioteca famosa, sem nenhuma relação.

Cadeia de Interrupções

Timer HW → IRQ 8 dispara

IVT[8] → aponta para a ISR instalada

ISR → processa som, atualiza TimeCount

```
setvect(8, isr);
```

4.3.1 — SDL SETTIMERSPEED()

Três Velocidades da ISR

TickBase × 2
= 140 Hz

TickBase × 10
= 700 Hz

TickBase × 100
= 7.000 Hz

FECHAMENTO

5 Ideias para Levar

01

27.223 SLOC — minúsculo para os padrões atuais. Compreensível em poucos dias de leitura.

02

90% C, 10% Assembly — a mistura foi inevitável pelas limitações do real mode 16-bit.

03

Três blocos (Menu, 3D, Som) comunicam-se exclusivamente via **memória compartilhada**.

04

Sound System comanda o heartbeat via interrupções — solução elegante para a ausência de threads em 1991.

05

O código faz coisas hoje "heréticas" com tranquilidade: **auto-modificação em runtime** (Patch386), **dados linkados no binário** (SIGNON, GAMEPAL), **prefixo SDL_ colidindo** com futura biblioteca famosa — e ainda assim moldou a indústria.