Wolfenstein 3D

Victor Samora, Daniel Frigini, Samuel Alves, Alyson godin

19 de abril de 2026

1 Introdução

O lançamento do código-fonte de Wolfenstein 3D em 21 de julho de 1995 marcou um ponto de inflexão na indústria de jogos. Disponibilizado inicialmente via servidores FTP da id Software e, posteriormente, migrado para o GitHub em 2012, o motor (engine) desenvolvido por John Carmack continua sendo um objeto de estudo fundamental para a compreensão da computação gráfica e engenharia de software.

2 Obtenção e Primeira Análise do Código-Fonte

2.1 Obtenção do Código-Fonte

O código-fonte do motor de jogo foi disponibilizado publicamente pela id Software em 21 de julho de 1995, por meio de seu servidor FTP oficial. Notavelmente, o arquivo original permanece acessível até os dias atuais, evidenciando a durabilidade de repositórios históricos na web.

Como alternativa mais moderna e confiável, o código também pode ser obtido via plataforma GitHub, para onde foi migrado por volta de 2012. Essa abordagem oferece maior estabilidade e facilidade de acesso, conforme ilustrado abaixo:

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

2.2 Primeiro Contato com o Código

Após o download, o arquivo `wolfsrc.zip` contém um segundo arquivo compactado no formato autoextraível (SFX), criado com a ferramenta PKZIP — uma prática comum na época, mas atualmente considerada obsoleta.

A extração pode ser realizada com ferramentas modernas, como:

```
unzip WOLFSRC.1
```

Para uma análise inicial da estrutura do projeto, pode-se utilizar ferramentas como o *cloc* (Count Lines of Code), que fornece estatísticas sobre a quantidade de arquivos e linhas de código, auxiliando na compreensão da dimensão e complexidade do software.

2.3 Métricas de Software (SLOC)

Utilizando a ferramenta *Count Lines of Code* (CLOC), é possível mensurar a magnitude do projeto. Embora as métricas de linhas de código (SLOC) não indiquem qualidade, elas fornecem uma escala de esforço e complexidade (Tabela 1).

Tabela 1: Estatísticas de Código-Fonte do Wolfenstein 3D

Linguagem	**Arquivos**	**Branco**	**Comentário**	**Código**
C/C++ (C)	75	4.521	3.892	22.145
C/C++ Header	32	1.204	1.105	4.230
Assembly	10	215	140	848
Total	**117**	**5.940**	**5.137**	**27.223**

O projeto é composto por aproximadamente 90% em linguagem C, reservando a Linguagem de Montagem (Assembly) para gargalos críticos de entrada e saída (E/S) e renderização de vídeo. Comparativamente, Wolfenstein 3D é um projeto conciso; enquanto ele possui cerca de 27 mil linhas, kernels modernos como o Linux ultrapassam a marca de dezenas de milhões de linhas.

3 Arquitetura do Motor (Big Picture)

A *engine* é dividida em três pilares fundamentais que operam de forma interdependente via memória compartilhada:

- **Motor de Menu 2D:** Responsável pela interface de configuração.
- **Renderizador 3D:** O núcleo de processamento onde o *Raycasting* é executado.
- **Sistema de Som:** Um módulo que opera de forma assíncrona, ditando o ritmo (*heartbeat*) do jogo.

4 Visão Geral da Arquitetura

O motor do jogo é estruturado em três componentes principais que operam de forma integrada:

- **Motor de Menu 2D:** responsável pela interface e configuração do jogo.
- **Renderizador 3D:** núcleo principal onde ocorre a ação e a maior parte do processamento gráfico.
- **Sistema de Som:** executado de forma concorrente, responsável pelo áudio e pelo controle de tempo (*heartbeat*) do sistema.

Esses três subsistemas comunicam-se por meio de memória compartilhada (RAM). O renderizador envia requisições de áudio, enquanto o sistema de som controla o ritmo global do jogo através da variável `TimeCount`, garantindo sincronização entre os componentes.

4.1 Estrutura de Execução

O fluxo principal do programa inicia-se na função `main()`, conforme ilustrado no Código 1.

```
void main ( void ) {
    CheckForEpisodes();
    Patch386();
    InitGame();
    DemoLoop();
}
```

Listing 1: Estrutura principal do programa

Inicialmente, o sistema verifica os recursos disponíveis e aplica otimizações específicas para processadores 386. Em seguida, inicializa todos os subsistemas e entra em um laço principal contínuo.

4.2 Modo Real e Otimizações

Devido ao uso do modo real da arquitetura x86, os tipos de dados possuem limitações específicas:

- `int` e `word`: 16 bits
- `long` e `dword`: 32 bits

Para melhorar o desempenho, o motor detecta CPUs mais avançadas (80386) e substitui rotinas matemáticas por instruções em Assembly utilizando registradores de 32 bits, como `EAX` e `EDX`.

4.3 Inicialização do Motor

A função `InitGame()` configura todos os gerenciadores do sistema, incluindo memória, vídeo, entrada, som e cache, além de preparar tabelas auxiliares para renderização e cálculos trigonométricos.

4.4 Laço Principal

O núcleo da execução ocorre em um laço infinito responsável por alternar entre menus e execução do jogo:

```
while (1) {
    GameLoop();       // Menu 2D
    SetupGameLevel();
    DrawLevel();
    PlayLoop();       // Execu   o 3D
}
```

Listing 2: Laço principal do jogo

4.5 Loop de Renderização 3D

O `PlayLoop()` segue o padrão clássico de jogos digitais:

1. Captura de entrada do jogador
2. Atualização do estado do mundo
3. Renderização da cena 3D

A renderização inclui a limpeza da tela, desenho de paredes, objetos e arma do jogador, utilizando técnicas de *raycasting* para simular tridimensionalidade.

4.6 Sistema de Som

O sistema de som opera de forma independente por meio de interrupções (*Interrupt Service Routines*), já que o ambiente MS-DOS não suporta multitarefa nativa. Dessa forma, o áudio é processado de maneira concorrente ao restante do jogo, garantindo sincronização e responsividade.

5 Implementação e Desafios do Modo Real

Diferente das arquiteturas modernas de 32 ou 64 bits, o Wolfenstein 3D foi projetado para o **Modo Real** dos processadores x86. Isso implica em:

1. **Tipagem Estrita:** `int` e `word` possuem 16 bits; `long` e `dword` possuem 32 bits.

2. **Aritmética de Ponteiros:** Limitação de endereçamento em segmentos de 64KB.

No *Patch386*, o motor detecta a presença de uma CPU Intel 80386 e realiza uma modificação dinâmica no código (*self-modifying code*), substituindo rotinas de divisão da Borland C++ por instruções nativas de 32 bits utilizando os registradores `EAX` e `EDX`.

5.1 Gerenciamento de Interrupções

Dado que o MS-DOS não é um sistema operacional multitarefa (não suporta threads nativas), a execução simultânea do som foi alcançada via *Interrupt Service Routines* (ISR). Através da função `SDL_SetTimerSpeed` (prefixo para *SounD Low level*), o motor instala uma ISR na Tabela de Vetores de Interrupção, permitindo que o áudio seja processado em frequências que variam de 140Hz a 7000Hz, independentemente do ciclo de renderização.

6 Fluxo de Execução do Motor

O fluxo principal de execução do Wolfenstein 3D segue uma estrutura sequencial clara, iniciando pela verificação de recursos disponíveis, aplicação de otimizações específicas de hardware, inicialização dos subsistemas e, por fim, entrada em um laço principal contínuo. O ponto de entrada do programa é definido conforme o Código 3.

```
1 void main ( void ) {
2     CheckForEpisodes();
3     Patch386();
4     InitGame();
5     DemoLoop();
6 }
```

Listing 3: Função principal do programa

A função `CheckForEpisodes()` verifica a disponibilidade de episódios do jogo, enquanto `Patch386()` aplica otimizações específicas para processadores

Intel 80386, substituindo rotinas de divisão por versões mais eficientes que utilizam registradores de 32 bits.

6.1 Inicialização dos Subsistemas

A função `InitGame()` é responsável por inicializar todos os gerenciadores do motor, incluindo memória, vídeo, entrada, som e cache, como ilustrado no Código 4.

```
void InitGame () {
    MM_Startup();
    SignonScreen();
    VW_Startup();
    IN_Startup();
    PM_Startup();
    PM_UnlockMainMem();
    SD_Startup();
    CA_Startup();
    US_Startup();
    InitDigiMap();
    ReadConfig();
    CA_CacheGrChunk(STARTFONT);
    MM_SetLock(&grsegs[STARTFONT], true);
    LoadLatchMem();
    BuildTables();
    SetupWalls();
}
```

Listing 4: Inicialização do motor do jogo

Além da inicialização dos sistemas, são criadas tabelas de consulta para operações trigonométricas (seno e cosseno) e estruturas auxiliares para renderização de paredes.

6.2 Laço Principal do Jogo

Após a inicialização, o programa entra em um laço infinito responsável por alternar entre menus, carregamento de níveis e execução do jogo propriamente dito (Código 5).

```
void DemoLoop () {
    StartCPMusic(INTROSONG);
    PG13();
    while (1) {
        CA_CacheScreen(TITLEPIC);
        CA_CacheScreen(CREDITSPIC);
        DrawHighScores();
        PlayDemo(0);
```

```
        GameLoop();
        SetupGameLevel();
        StartMusic();
        PM_CheckMainMem();
        PreloadGraphics();
        DrawLevel();
        PlayLoop();
        StopMusic();
    }
}
```

Listing 5: Laço principal de execução

Esse laço evidencia a separação entre o motor 2D (menus) e o motor 3D (ação).

6.3 Loop de Execução do Jogo

O núcleo da execução ocorre em `PlayLoop()`, que segue o padrão clássico de jogos digitais: entrada, atualização e renderização (Código 6).

```
void PlayLoop () {
    PollControls();
    MoveDoors();
    MovePWalls();
    for (obj = player; obj; obj = obj->next)
        DoActor(obj);
    ThreeDRefresh();
    UpdateSoundLoc();
}
```

Listing 6: Loop principal do gameplay

6.4 Renderização 3D

A função `ThreeDRefresh()` é responsável pela renderização da cena tridimensional, incluindo limpeza da tela, desenho de paredes, objetos e arma do jogador (Código 7).

```
ThreeDRefresh () {
    VGAClearScreen();
    WallRefresh();
    DrawScaleds();
    DrawPlayerWeapon();
}
```

Listing 7: Renderização da cena 3D

Essa abordagem utiliza técnicas de *raycasting*, permitindo simular um ambiente tridimensional com baixo custo computacional.

6.5 Sistema de Som e Interrupções

O sistema de som é implementado utilizando rotinas de interrupção (*Interrupt Service Routines* - ISR), possibilitando execução concorrente em um ambiente sem suporte a multitarefa. A frequência de atualização do áudio varia conforme a necessidade, como demonstrado no Código 8.

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

    if ((DigiMode == sds_PC) && DigiPlaying) {
        rate = TickBase * 100;
        isr = SDL_t0ExtremeAsmService;
    }
    else if (music || ((DigiMode == sds_SoundSource) &&
   DigiPlaying)) {
        rate = TickBase * 10;
        isr = SDL_t0FastAsmService;
    }
    else {
        rate = TickBase * 2;
        isr = SDL_t0SlowAsmService;
    }

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

Listing 8: Configuração do temporizador de áudio

Essa estratégia permite frequências de atualização que variam entre 140 Hz e 7000 Hz, garantindo responsividade sonora mesmo sem suporte a threads.

6.6 Considerações

A estrutura apresentada demonstra um padrão arquitetural que permanece relevante na indústria de jogos: um laço principal contínuo com separação clara entre entrada, atualização de estado e renderização. Essa abordagem, consolidada no Wolfenstein 3D, influenciou diretamente o desenvolvimento de motores gráficos modernos.

7 Conclusão

Wolfenstein 3D não foi apenas um sucesso comercial, mas um triunfo da engenharia de software sob restrições severas. A capacidade de John Carmack e sua equipe de extrair performance de sistemas de 16 bits, utilizando técnicas como o loop desenrolado (*unrolled loop*) e interrupções de hardware, estabeleceu os alicerces para o gênero de tiro em primeira pessoa (FPS).

Referências

[1] SANGLARD, Fabien. *Game Engine Black Book: Wolfenstein 3D*. 2nd Edition. 2019.

[2] ID SOFTWARE. *Wolfenstein 3D Source Code*. Disponível em: `https://github.com/id-Software/wolf3d`.