Wolfenstein 3D: Renderer 2D e 3D

Um resumo do Capítulo 4 (seções 4.6 a 4.7.4) de
Game Engine Black Book: Wolfenstein 3D

Enzo Zon
Arquitetura de Computadores II — Grupo 5

6 de maio de 2026

Abstract

Este artigo resume as seções 4.6 a 4.7.4 do livro *Game Engine Black Book: Wolfenstein 3D*, de Fabien Sanglard, que abrangem as páginas 132 a 154. O texto descreve a fase de menu (2D renderer), o início da fase de ação com o 3D renderer baseado em raycasting, o ciclo de vida de um frame e de um frame 3D (incluindo seus cinco estágios de renderização), a configuração inicial do HUD com o truque dos latches VGA, e a rotina de limpeza de tela com `REP STOSW`. O objetivo é oferecer uma visão concisa de como decisões de baixo nível em hardware e software definiram o desempenho do engine num contexto de hardware extremamente limitado.

Contents

1 Introdução

Com o sistema de triple buffering e o modo de vídeo VGA Mode-Y devidamente configurados (tópicos das seções anteriores do capítulo), o jogo Wolfenstein 3D entra em sua fase operacional. Duas etapas distintas se seguem: primeiro o **2D renderer**, que exibe os menus de configuração, e depois o **3D renderer**, que executa a renderização do mundo em perspectiva via raycasting.

As páginas 132–154 cobrem esses dois subsistemas e, ao longo do caminho, revelam técnicas engenhosas que extraíam o máximo do hardware disponível em 1992: CPUs Intel 386/486 sem unidade de ponto flutuante, 1 MB de RAM e placas VGA de 256 cores.

2 Seção 4.6 — Menu Phase: 2D Renderer

2.1 O truque da máscara de banco VGA

Ao entrar na fase de menu, o engine precisa pintar o fundo de vermelho – toda a tela de 320×200 pixels. De forma ingênua, isso exigiria 64.000 escritas na VRAM. O engine usa, porém, o recurso de *bank mask* (máscara de banco) do VGA.

O VGA organiza a VRAM em quatro bancos (Bank 0 a Bank 3). Com a máscara configurada para 15 (= 8+4+2+1), uma única escrita em endereço x atinge os quatro bancos simultaneamente, escrevendo quatro pixels de uma vez. Com registradores de 16 bits disponíveis na CPU, é possível escrever oito pixels por instrução, reduzindo a contagem de escritas para:

$$\frac{320 \times 200}{8} = 8.000 \text{ escritas} \tag{1}$$

ou seja, uma redução de **8**× em relação ao método ingênuo.

Limitação. Apenas bytes no *mesmo endereço* de cada banco podem ser escritos simultaneamente. O alinhamento de pixels com bancos deve ser cuidadosamente considerado: pixels 0, 1, 2 e 3 cabem numa escrita; pixels 3 e 4 exigem duas escritas; pixels 3–8 exigem três.

2.2 Estrutura dos menus em código

Os menus são armazenados como arrays de structs do tipo `CP_itemtype`, cada um contendo um flag (ativo/inativo), uma string de exibição e um ponteiro de função (callback). O código a seguir ilustra o menu principal:

Listing 1: Definição do menu principal em WL_MENU.C

```
CP_itemtype far MainMenu[] = {
  {1, "New Game",     CP_NewGame},
  {1, "Sound",        CP_Sound},
  {1, "Control",      CP_Control},
  {1, "Load Game",    CP_LoadGame},
  {0, "Save Game",    CP_SaveGame},
```

```
  {1, "Change View", CP_ChangeView},
};

void DrawMainMenu(void) {
  ClearMScreen();                          // pinta fundo
   vermelho
  VWB_DrawPic(112, 184, C_MOUSELBACKPIC); // imagem inferior
  DrawStripes(10);                         // faixa preta
  VWB_DrawPic(84, 0, C_OPTIONSPIC);       // logo OPTIONS
  DrawWindow(MENU_X-8, MENU_Y-3, MENU_W, MENU_H, BKGDCOLOR);
  DrawMenu(&MainItems, &MainMenu[0]);
  VW_UpdateScreen();
}
```

As macros `C_MOUSELBACKPIC` e `C_OPTIONSPIC` sao geradas automaticamente pelo compilador de assets `IGRAB`. O 2D renderer usa o *User Manager* para renderizar texto e o *Cache Manager* para buscar recursos do disco.

3 Seccao 4.7 — Action Phase: 3D Renderer

Apos a fase de menu, o engine 3D assume o controle. A tecnologia central e o **raycasting**: para cada coluna de pixels da tela (320 colunas), um raio e lancado a partir da posicao do jogador. A distancia d ate a parede mais proxima determina a altura h da coluna texturizada:

$$h = \frac{X}{d} \tag{2}$$

onde X e um fator de escala simples. Essa equacao garante perspectiva convincente sem nenhuma multiplicacao de matrizes ou hardware 3D dedicado.

4 Seccao 4.7.1 — Life of a Frame

4.1 O problema do timeslice variavel

O loop principal do engine de Wolf3D tem a seguinte estrutura:

Listing 2: Game loop de Wolf3D (timeslice variavel)

```
int lastTime = Timer_Gettime();
while (1) {
    int currentTime = Timer_Gettime();
    int timeSlice   = currentTime - lastTime;
    UpdateWorld(timeSlice);
    RenderWorld();
    lastTime = currentTime;
}
```

Cada iteracao tem duracao diferente, dependendo do tempo de renderizacao. Isso torna o jogo **nao determinístico**: em maquinas diferentes (286 vs. 486) ou ate entre duas execucoes na mesma maquina, um acesso a disco mais demorado pode alterar a duracao de um frame e, consequentemente, o estado do mundo.

Para os demos distribuidos com o jogo, o engine usou um hack: ignorar os heartbeats reais e simular eventos a intervalos fixos (`DEMOTICK = 4`). Isso resultava em playback mais lento em CPUs 286 e mais rapido em 486, pois o demo foi gravado num 386DX a aproximadamente 17 fps.

4.2 A solucao do Doom (1993)

O Doom resolveu o problema com um timeslice fixo de 28 ms (35 fps):

Listing 3: Game loop do Doom com timeslice fixo

```
int gameOn = 1, simulationTime = 0;
while (gameOn) {
    int realTime = Gettime();
    while (simulationTime < realTime) {
        simulationTime += 28;  // timeslice SEMPRE 28 ms
        UpdateWorld(28);
    }
    RenderWorld();
}
```

O renderer e desacoplado da simulacao: qualquer numero de passos de simulacao pode ocorrer por frame renderizado, e os inputs podem ser gravados e repetidos com perfeita sincronia.

5 Seccao 4.7.2 — Life of a 3D Frame

Um frame 3D completo passa por cinco estagios sequenciais:

1. **Limpar o framebuffer.** Teto e chao sao preenchidos com cores solidas usando o truque da mascara VGA (Secao 2).

2. **Raycasting e paredes.** Para cada coluna da tela, um raio e lancado. A distancia ate a parede determina a altura da coluna texturizada. Portas sao detectadas e tratadas com delta de distancia.

3. **Sprites (things).** Inimigos, lampadas e barris sao renderizados apos o raycasting, com clipping realizado contra paredes e portas.

4. **Arma.** Desenhada sobre a cena ja renderizada. Em 1992, acessar a memoria era tao lento que aceitar overdraw era mais eficiente do que usar um depth buffer.

5. **Flip de buffer.** O CRT Controller e instruido a usar o framebuffer recem-composto no proximo vsync, completando o ciclo de triple buffering.

6 Seccao 4.7.3 — 3D Setup

6.1 Configuracao do HUD

Antes de iniciar o loop de frames, o 3D renderer configura a VRAM com os elementos estaticos do HUD (*Heads-Up Display*): o fundo verde, a barra de status azul e os rotulos

`LEVEL`, `SCORE`, `LIVES`, `HEALTH` e `AMMO`. Este HUD e desenhado *uma unica vez*, em todas as tres paginas de video. A cada novo frame, apenas pequenas porcoes do HUD sao atualizadas.

6.2 O truque dos latches VGA

O Mode-Y nao precisa dos latches VGA para operar. Porem, esses latches — originalmente projetados para o Mode 12h de 16 cores — permanecem ativos no Mode-Y. Com uso criativo, e possivel reprogramar o ALU na frente de cada banco para usar somente o latch na escrita:

1. Uma **leitura** da VRAM carrega os quatro latches simultaneamente.
2. Uma unica **escrita** propaga o conteudo dos latches para os quatro bancos.

Isso permite transferencias VRAM→VRAM de 4 bytes por operacao, resultando num **aumento de 30% na velocidade geral** de renderizacao.

6.3 Armazenamento entrelaçado (*woven*)

Para que o truque funcione, os sprites de HUD sao armazenados de forma entrelaçada: todos os bytes do banco 0, depois os do banco 1, e assim por diante. Os 43 sprites de HUD totalizam:

$$48{\times}24{\times}4 + 14{\times}8{\times}16 + 24{\times}24{\times}32 + 224{\times}48 = 34.816 \text{ bytes}$$

deixando 35.324 bytes livres na quarta pagina VGA.

7 Seccao 4.7.4 — Clearing the Screen

7.1 O problema

A cada frame, o engine precisa limpar o canvas 3D de 304×152 pixels (= 46.208 pixels) com as cores do teto e do chao. Sem otimizacao, isso exigiria 46.208 escritas individuais.

7.2 A solucao: mascara + REP STOSW

Listing 4: Rotina VGAClearScreen (Assembly inline)

```
; Configura mascara: todos os bancos
asm   mov   dx, SC_INDEX
asm   mov   ax, SC_MAPMASK + 15*256
asm   out   dx, ax

toploop:                        ; Loop do teto
    asm   mov   cl, bl          ; bl = viewwidth/8
    asm   rep   stosw           ; escreve bl words (8 pixels cada)
    asm   add   di, dx          ; pula as margens laterais
    asm   dec   bh
    asm   jnz   toploop
```

```
12
13 bottomloop:                    ; Loop do chao
14     asm   mov   cl, bl
15     asm   rep   stosw
16     asm   add   di, dx
17     asm   dec   bh
18     asm   jnz   bottomloop
```

7.3 Contagem de instrucoes

A decomposicao do custo e:

$$\underbrace{16}_{\text{setup}} + \underbrace{5 \times 76}_{\text{teto}} + \underbrace{3}_{\text{inter-loop}} + \underbrace{5 \times 76}_{\text{chao}} = \mathbf{779} \text{ instrucoes}$$

para limpar 46.208 pixels — uma reducao de mais de **59**× em relacao ao metodo ingenuo. As cores do chao sao sempre `0x19`; as do teto sao hardcoded no array `vgaCeiling[]` por fase.

8 Consideracoes Finais

A leitura das paginas 132–154 revela um padrao consistente na engenharia do Wolfenstein 3D: *transformar limitacoes de hardware em oportunidades de otimizacao.* Quatro decisoes tecnicas se destacam:

1. **Mascara de banco VGA.** Transforma 64.000 escritas em 8.000 para limpar a tela — usada tanto no 2D quanto no 3D renderer.
2. **Raycasting como engine 3D.** A equacao $h = X/d$ entregou perspectiva convincente num 386 sem FPU, a cerca de 17 fps, sem hardware grafico especializado.
3. **Reaproveitamento dos latches VGA.** Circuito projetado para Mode 12h foi reutilizado no Mode-Y para copiar 4 bytes de VRAM por operacao, gerando um ganho global de 30% na renderizacao do HUD.
4. **REP STOSW + mascara.** Combinacao de instrucao de string x86 com mascara de banco: 46.208 pixels limpos com 779 instrucoes ($> 59\times$ de aceleracao).

Referencia

SANGLARD, Fabien. *Game Engine Black Book: Wolfenstein 3D*. 2. ed. Edicao independente, 2019. Capitulo 4, secoes 4.6 a 4.7.4 (paginas 132–154).