CAPÍTULO 4 · SEÇÕES 4.6 A 4.7.4 · PÁGINAS 132–154

Wolfenstein 3D:
Renderer 2D e 3D

Um resumo das seções 4.6 a 4.7.4 de Game Engine Black Book: Wolfenstein 3D

Enzo Zon
Arquitetura de Computadores II · Grupo 5
2026

Agenda

01 **Contexto**
VGA, triple buffering e a estrutura do jogo

02 **Seção 4.6 — Renderer 2D**
Menu, banco VGA e o truque da máscara

03 **Seção 4.7 — Renderer 3D**
Raycasting e o pipeline do engine

04 **4.7.1 — Life of a Frame**
Timeslices variáveis e o problema de replay

05 **4.7.2 — Life of a 3D Frame**
Os 5 estágios de renderização

06 **4.7.3 — 3D Setup**
HUD, latches VGA e transferência VRAM

07 **4.7.4 — Clearing the Screen**
REP STOSW e otimização de limpeza

Contexto — O que são essas seções?

VGA inicializado

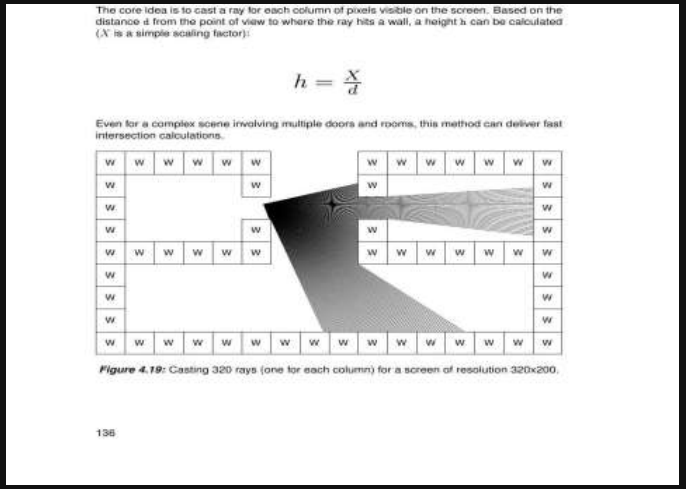
Com o VGA e o triple buffering prontos, o jogo entra na fase de menus (2D) e depois na ação (3D).

Duas fases

Fase 2D: menus de configuração. Fase 3D: o engine de ação com raycasting.

Hardware modesto

Intel 386/486, sem FPU, 1 MB de RAM, VGA de 256 cores — todo truque contava.


The core idea is to cast a ray for each column of pixels visible on the screen. Based on the distance d from the point of view to where the ray hits a wall, a height h can be calculated (X is a simple scaling factor):

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

Even for a complex scene involving multiple doors and rooms, this method can deliver fast intersection calculations.

Figure 4.19: Casting 320 rays (one for each column) for a screen of resolution 320x200.

136

Fig. 4.19 — 320 raios, um por coluna

4.6 · Menu Phase: 2D Renderer

4.6 Menu Phase: 2D Renderer

With the VGA up and running and a robust triple buffering system ready to operate, the game finally starts. The player enters the 2D renderer which displays menus to setup the 3D game. This is pretty simple yet features a nice VGA trick to turn the four banks' weak and cumbersome design into a strength.

Tela de menu em vermelho — 320×200 px

O Truque da Máscara VGA

- Fundo vermelho exige 320×200 = 64.000 escritas no modo ingênuo
- Máscara de banco = 15 (todos os bancos): uma escrita → 4 pixels simultâneos
- Com registradores de 16 bits: 8 pixels por instrução
- Resultado: 320×200 / 8 = apenas 8.000 escritas (redução de 8×)

Limitação: apenas bytes no mesmo endereço de banco podem ser escritos simultaneamente. Alinhamento de pixel com banco é crucial.

4.6 · Arquitetura dos Bancos VGA

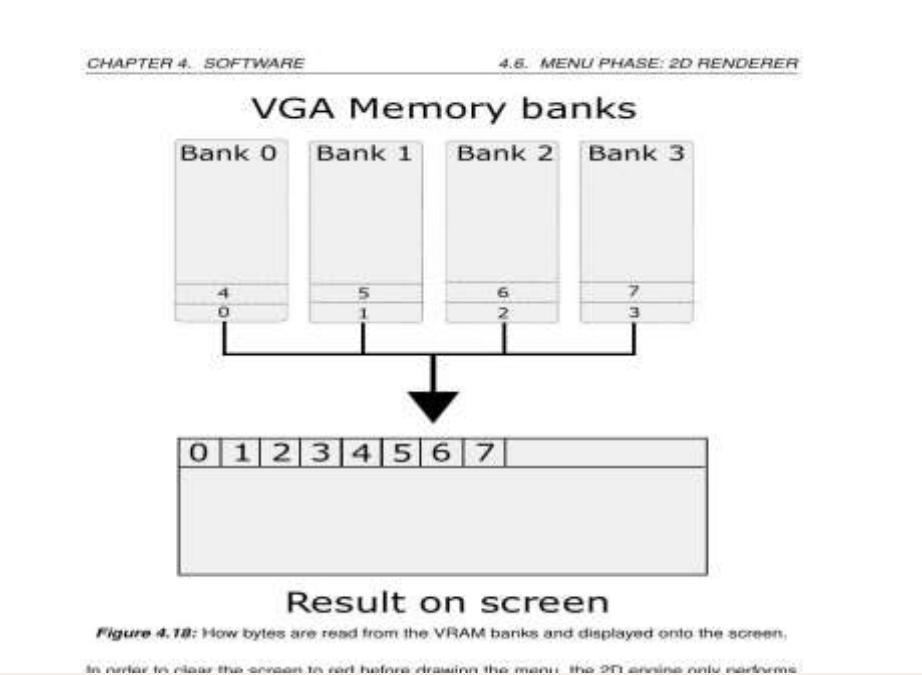
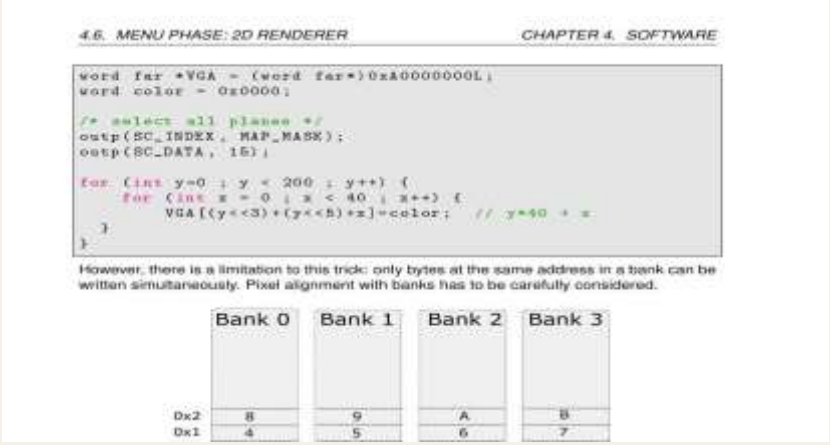


Fig. 4.18 — Leitura dos bancos VRAM e resultado na tela



Restrição de alinhamento: pixels em bancos diferentes requerem escritas separadas

Pixels 0,1,2,3 → 1 escrita ✓
Pixels 3 e 4 → 2 escritas • Pixels 3–8 → 3 escritas

4.6 · Estrutura do Menu em Código

```
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();          // fundo vermelho
  VWB_DrawPic(112,184,
    C_MOUSELBACKPIC);      // imagem inferior
  DrawStripes(10);         // faixa preta
  VWB_DrawPic(84,0,
    C_OPTIONSPIC);         // logo OPTION
  DrawWindow(...);
  DrawMenu(&MainItems,
           &MainMenu[0]);
  VW_UpdateScreen();
}
```

Menus como array de structs

Cada item tem flag (ativo/inativo), string e callback.

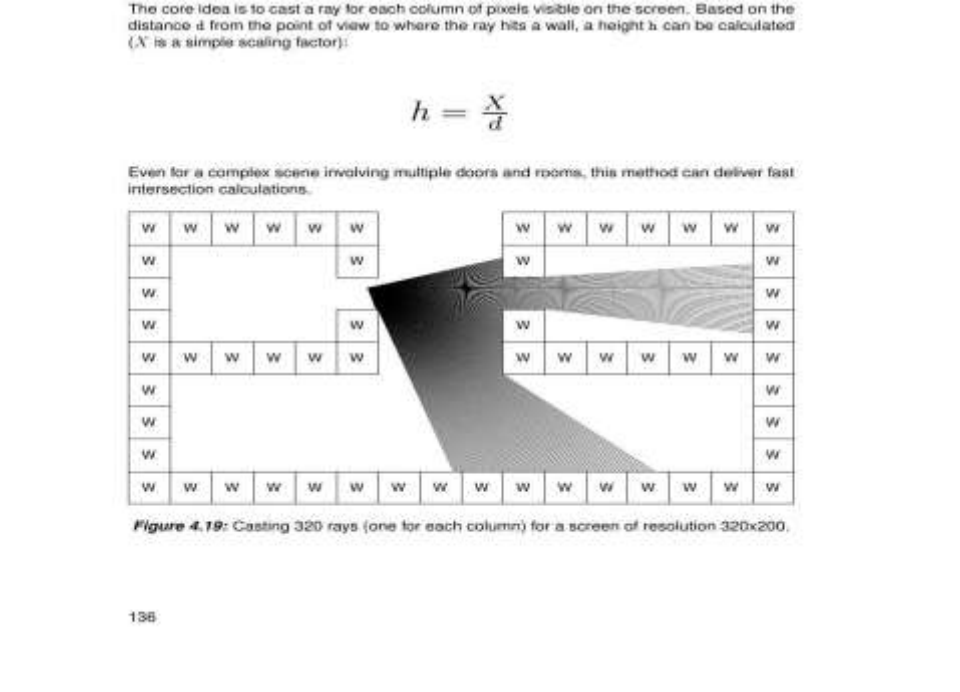
Assets via IGRAB

C_MOUSELBACKPIC e C_OPTIONSPIC são macros geradas pelo compilador de assets.

User Manager

O 2D renderer usa o User Manager para texto e o Cache Manager para buscar assets do disco.

4.7 · Action Phase: 3D Renderer



The core idea is to cast a ray for each column of pixels visible on the screen. Based on the distance d from the point of view to where the ray hits a wall, a height h can be calculated (X is a simple scaling factor):

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

Even for a complex scene involving multiple doors and rooms, this method can deliver fast intersection calculations.

Figure 4.19: Casting 320 rays (one for each column) for a screen of resolution 320x200.

136

Fig. 4.19 — 320 raios, um para cada coluna de pixels

Raycasting

$$h = X / d$$

altura da coluna ∝ 1 / distância

- Um raio por coluna da tela (320 colunas)
- Calcula distância até a parede mais próxima
- Desenha coluna texturizada com altura inversamente proporcional à distância
- Funciona bem mesmo com cenas complexas (portas, salas)

4.7.1 · Life of a Frame — O Problema do Timeslice

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

Loop do engine Wolf3D — timeslice variável

Solução do Doom (1993)

```
while (gameOn) {
  int real = Gettime();
  while (simTime < real) {
    simTime += 28; // fixo!
    UpdateWorld(28);
  }
  RenderWorld();
}
```

Problema: O timeslice varia conforme o tempo de render. Demos não podem ser repetidos deterministicamente — se um frame demorou mais (ex: acesso a disco), o replay fica fora de sincronia.

⚠ *Wolf3D usou DEMOTICK=4 para demos: mais lento em 286, mais rápido em 486!*

Timeslice sempre 28ms → simulação desacoplada do render → demos sincronizados

4.7.2 · Life of a 3D Frame — Os 5 Estágios

1. **Limpar framebuffer**
   Teto e chão com cores sólidas
2. **Raycasting + paredes**
   Ray por coluna, coluna texturizada proporcional à dist.
3. **Sprites**
   Inimigos, lâmpadas, barris — clipping contra paredes
4. **Arma**
   Desenhada sobre a cena (sem depth buffer, aceita overdraw)
5. **Flip de buffer**
   CRT Controller usa framebuffer novo no próximo vsync

Estágio 1: Teto + chão (Fig. 4.22)

Estágio 2: 15 raios (Fig. 4.23)

Estágio 3: Sprites (Fig. 4.26)

4.7.2 · Estágio 4 e 5 — Arma e Flip de Buffer

Fig. 4.27 — Estágio 4: Arma desenhada (sem depth buffer)

Estágio 4: Arma

Em engines modernos a arma seria renderizada primeiro com depth buffer. Em 1992, acesso à memória era tão lento que overdraw era mais rápido que ler o depth buffer.

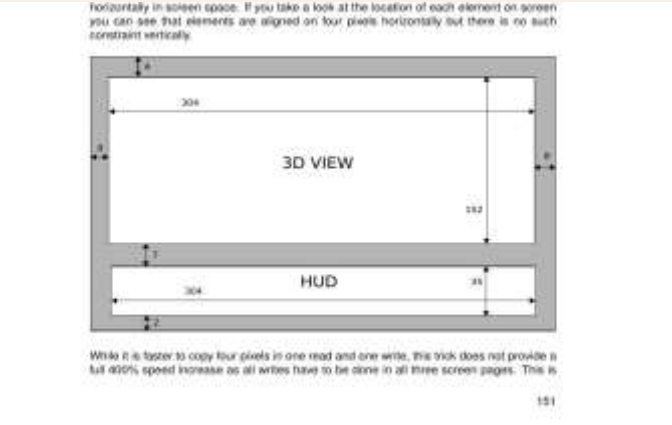
Estágio 5: Flip de Buffer

O CRT Controller é instruído a usar o framebuffer recém-composto no próximo vsync. Triple buffering garante que não haja tearing nem espera desnecessária.

"Wolf foi o último jogo da Id com vidas e score — ainda em modo arcade. Com o Doom finalmente você jogava sem parar."
— John Carmack

4.7.3 · 3D Setup — HUD e Latches VGA

HUD desenhado uma vez só — LEVEL, SCORE, LIVES, HEALTH, AMMO



Layout de pixels — elementos alinhados em 4px horizontalmente

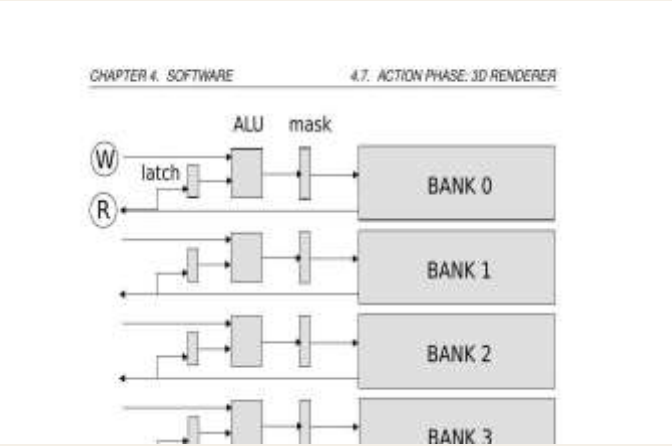
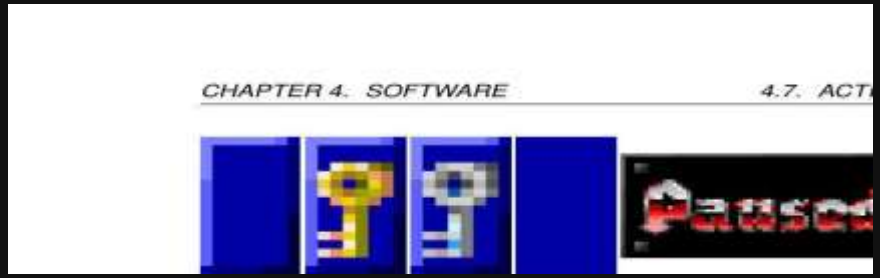


Fig. 4.28 — Latches: leitura memorizada para escrita futura

- HUD desenhado só uma vez (ao iniciar fase 3D) nas 3 páginas
- Latches VGA (ativos em Mode-Y) permitem cópia VRAM→VRAM com 1 read + 1 write
- 4 bytes transferidos de uma só vez: 30% de aumento geral de velocidade

4.7.3 · Armazenamento Entrelaçado (Woven)



Sprites do HUD: numerais, rosto e armas carregados no 4º página VGA

Truque de Armazenamento Entrelaçado

Todos os bytes do banco 0, depois banco 1, banco 2, banco 3.
Assim cada banco é carregado com apenas um memcpy (4 memcpy por imagem).
Elementos alinhados em 4px horizontalmente para o truque funcionar.

48×24×4 (armas) + 14×8×16 (numerais/chaves) + 24×24×32 (rostos) + 224×48 (paused+psyched) = 34.816 bytes → 35.324 bytes livres na 4ª página VGA

```
// Atualiza sprite do HUD em 3 páginas
void StatusDrawPic(int x, int y,
                   int picnum)
{
  unsigned temp;
  temp = bufferofs;

  bufferofs = PAGE1START +
    (200-STATUSLINES)*SCREENWIDTH;
  LatchDrawPic(x, y, picnum);

  bufferofs = PAGE2START +
    (200-STATUSLINES)*SCREENWIDTH;
  LatchDrawPic(x, y, picnum);

  bufferofs = PAGE3START +
    (200-STATUSLINES)*SCREENWIDTH;
  LatchDrawPic(x, y, picnum);

  bufferofs = temp;
}

// Resultado: +30% de velocidade geral
```

4.7.4 · Clearing the Screen — REP STOSW

304×152
pixels do canvas 3D
= 46.208 total

8.000
escritas para limpar
tela cheia (16-bit)

779
instruções reais
com REP STOSW

>59×
redução efetiva
de instruções

```
; Máscara = 15 (todos os bancos)
asm mov dx, SC_INDEX
asm mov ax, SC_MAPMASK+15*256
asm out dx, ax
; bl=viewwidth/8  bh=altura/2
toploop:     ; Loop do teto
  asm mov cl, bl
  asm rep stosw  ; REP STOSW!
  asm add di, dx
  asm dec bh
  asm jnz toploop
bottomloop:  ; Loop do chão
  asm mov cl, bl
  asm rep stosw
  asm add di, dx
  asm dec bh
  asm jnz bottomloop
```

Máscara = 15

Todos os bancos escritos simultaneamente — 8 pixels por instrução de 16 bits.

REP STOSW

Instrução x86 que repete STOSW (escrita em string) — carrega múltiplos pixels em um único fetch de instrução.

Decomposição

16 (setup) + 5×76 (teto) + 3 (inter) + 5×76 (chão) = 779 instruções para limpar 46.208 pixels.

Cores hardcoded

Chão: sempre 0x19. Teto: array vgaCeiling[] — um valor por fase, codificado no engine.

Considerações Finais

Tricks VGA no 2D

A máscara de banco transformou 64.000 escritas em 8.000 — truque simples de hardware com impacto enorme.

Otimização em Assembly

REP STOSW com máscara VGA limpou 46.208 pixels com apenas 779 instruções — engenharia de baixo nível decisiva.

Raycasting como solução elegante

Uma equação simples (h = X/d) entregou perspectiva convincente sem hardware 3D, rodando em 386 a ~17 fps.

Latches como bônus

Os latches do Mode-Y, pensados para Mode 12h, foram reaproveitados para copiar VRAM 4 bytes por vez: +30% geral.

Problema do timeslice

Wolf3D usou hack (DEMOTICK) para demos. Doom (1993) resolveu com timeslice fixo — lição aprendida.

SANGLARD, Fabien. Game Engine Black Book: Wolfenstein 3D. 2. ed. 2019. Cap. 4, seções 4.6–4.7.4 (pp. 132–154).