Resumo Didático das Seções 3.2, 3.3 e 3.4

Wolfenstein 3D Game Engine Black Book

Gabriel Henrique Garces de Deus
Teo Giambarra Roseiro
Arthur Prates Pessoti
Arthur Solar de Castro Gomes
Rafael Fassina

Universidade Vila Velha
Grupo 4 — Turma CC5Mb
Disciplina: *Arquitetura e Organização de computadores 2*
Professor: *Abrantes Araujo Silva Filho*

15 de abril de 2026

Resumo

Este documento apresenta um resumo didático das seções 3.2, 3.3 e 3.4 do livro *Wolfenstein 3D Game Engine Black Book*. O foco está em explicar, de forma clara e direta, como a equipe da id Software programava o jogo, como os recursos gráficos eram produzidos e como esses arquivos eram organizados até chegarem ao motor do jogo.

Mais do que apenas listar ferramentas, essas seções mostram a parte prática do desenvolvimento: os programadores lidando com limitações reais de tela e depuração, o artista trabalhando com uma paleta rígida de 256 cores, e a equipe organizando um fluxo em que os assets podiam ser empacotados e referenciados pelo código de forma eficiente. Assim, o capítulo ajuda a entender que o desenvolvimento de *Wolfenstein 3D* dependia tanto de criatividade técnica quanto de organização de produção.

Sumário

1 Introdução

As seções 3.2, 3.3 e 3.4 tratam de uma parte muito importante do desenvolvimento de *Wolfenstein 3D*: o trabalho prático do dia a dia da equipe. Em vez de focar apenas no resultado final, o livro mostra como o jogo era realmente construído, desde a escrita do código até a produção e integração dos elementos visuais.

Essas seções deixam claro que o jogo não surgiu apenas de uma boa ideia de programação. Ele dependia de um conjunto de decisões concretas: quais ferramentas usar, como aproveitar o pouco espaço de tela disponível, como desenhar gráficos dentro das limitações da VGA e como transformar arquivos artísticos em dados que o motor pudesse carregar rapidamente.

Por isso, o valor dessas páginas está em mostrar o desenvolvimento como processo. O leitor entende não só *o que* foi feito, mas também *como* foi feito e *por que* esse método funcionava tão bem para a época.

2 Seção 3.2 — Programação

2.1 Máquinas e ambiente de desenvolvimento

O livro mostra que a equipe trabalhava em máquinas muito fortes para o padrão daquele momento: PCs 386-DX de 33 MHz com 4 MiB de RAM. Isso já indica um ponto importante: para criar um jogo tecnicamente avançado, não bastava ter boas ideias; também era necessário usar o melhor hardware disponível dentro do que a equipe podia comprar.

O desenvolvimento era feito com o *Borland C++ 3.1*. Apesar do nome da ferramenta, a linguagem usada no projeto era C. O ambiente da Borland reunia editor, compilador, linker e depurador em um pacote só, o que tornava o fluxo mais prático. Em vez de alternar entre muitas ferramentas separadas, boa parte do trabalho podia ser feita dentro do próprio ambiente integrado.

Na prática, isso significava que o programador podia editar, compilar, executar e depurar o projeto no mesmo ecossistema. Para um projeto complexo e iterativo como *Wolfenstein 3D*, isso acelerava bastante o desenvolvimento.

2.2 Divisão de responsabilidades

A seção também mostra que o trabalho de programação era dividido de forma clara. John Carmack cuidava do código principal de execução do jogo, ou seja, da parte mais diretamente ligada ao funcionamento do motor. John Romero programava várias ferramentas de suporte, como o editor de mapas TED5, o empacotador de assets IGRAB e o sistema MUSE, ligado ao áudio. Jason Blochowiak escreveu subsistemas importantes, como gerenciamento de entrada, páginas, som e interface de usuário.

Isso é importante porque mostra que o projeto não dependia de uma única pessoa fazendo tudo. Havia especialização. Enquanto um programador se concentrava no núcleo técnico do jogo, outros construíam ferramentas e sistemas auxiliares que permitiam o projeto avançar com mais velocidade e organização.

2.3 Programar com pouco espaço de tela

Um detalhe interessante dessa parte do livro é que programar naquela época também significava lidar com limitações físicas do ambiente de trabalho. Os monitores CRT eram pequenos, então o espaço visível para código e depuração era reduzido.

Para resolver isso, alguns desenvolvedores usavam duas telas. Uma delas ficava dedicada ao debugger em modo texto monocromático, enquanto a outra mantinha o jogo rodando em modo gráfico. Na prática, isso permitia acompanhar o funcionamento do jogo e, ao mesmo tempo, inspecionar o código passo a passo sem perder o contexto visual.

Além disso, outra solução usada era o modo texto 80x50, que dobrava a resolução vertical em relação ao modo texto tradicional. O ganho era simples, mas muito útil: mais linhas de código apareciam ao mesmo tempo, o que facilitava leitura, edição e navegação. O livro mostra que esse tipo de ajuste não era um luxo, mas parte do trabalho diário para ganhar produtividade.

2.4 Lição prática da seção

O ponto central da seção 3.2 é que programar *Wolfenstein 3D* não era apenas escrever algoritmos. Era também montar um ambiente de desenvolvimento eficiente, dividir tarefas corretamente e encontrar soluções práticas para contornar limitações do hardware disponível. Em outras palavras, a produtividade da equipe dependia tanto das ferramentas quanto da habilidade técnica dos programadores.

3 Seção 3.3 — Recursos Gráficos

3.1 Quem produzia os assets

Na parte gráfica, o livro destaca Adrian Carmack como o principal responsável pelos assets visuais do jogo. Todo esse trabalho era feito no *Deluxe Paint*, uma ferramenta muito conhecida da época, e os arquivos eram salvos em formato ILBM.

Esse ponto é importante porque mostra que os gráficos não eram gerados por pipelines modernos nem por ferramentas automatizadas. A maior parte do trabalho era manual. Os elementos visuais do jogo eram desenhados diretamente, pixel por pixel, com mouse.

3.2 A limitação da paleta VGA

A seção dá bastante ênfase a uma dificuldade prática: a VGA trabalhava com paleta indexada. Isso significa que os gráficos não eram definidos por qualquer cor RGB livremente escolhida, mas por índices apontando para uma tabela fixa de 256 cores.

Na prática, isso obrigava o artista a tomar uma decisão logo no início: quais cores fariam parte da paleta do jogo. Depois dessa escolha, todo o restante precisava ser desenhado usando apenas esse conjunto. Isso limitava bastante a liberdade artística, mas ao mesmo tempo exigia consistência visual.

O livro também destaca que a equipe da id Software optou por usar uma única paleta para o jogo inteiro, em vez de trocar paletas conforme a área ou a situação. Essa decisão simplificava o processo e mantinha uma identidade visual mais estável, embora aumentasse a responsabilidade de escolher bem essas 256 cores.

3.3 Transparência e desenho na resolução real

Outro detalhe prático importante é que a cor de índice `0xFF` era tratada pelo motor como transparente. Isso era essencial para sprites e objetos desenhados sobre o cenário, já que permitia ignorar o fundo de certas imagens durante a renderização.

Além disso, como a VGA esticava o framebuffer quando a imagem era exibida na tela, Adrian Carmack precisava desenhar já pensando na resolução real de execução do jogo, que era 320x200. Ou seja, o artista não criava apenas "uma imagem bonita"; ele precisava entender como essa imagem seria deformada ou apresentada no hardware final.

Esse ponto mostra uma integração muito forte entre arte e tecnologia. O artista

precisava conhecer o comportamento da máquina para que o resultado final funcionasse visualmente dentro do jogo.

3.4 Categorias de assets

A seção ainda divide os recursos gráficos em dois grandes grupos:

- itens 2D de menu, armazenados nos arquivos `VGAGRAPH`, `VGAHEAD` e `VGADICT`;
- itens da fase 3D, como paredes e sprites, armazenados no arquivo `VSWAP`.

Essa separação mostra que os dados visuais não eram todos tratados da mesma forma. O jogo já organizava os elementos conforme seu uso. Aquilo que aparecia em telas de menu seguia uma estrutura, e aquilo que participava da ação 3D seguia outra. Essa distinção facilitava o gerenciamento do conteúdo e combinava com as necessidades específicas de cada parte do jogo.

3.5 Lição prática da seção

A principal lição da seção 3.3 é que os gráficos de *Wolfenstein 3D* surgiam de um processo extremamente controlado. O artista precisava respeitar a paleta, desenhar manualmente, pensar na transparência e ainda considerar como a VGA exibiria o resultado. Portanto, a qualidade visual do jogo não vinha de liberdade total, mas de domínio técnico dentro de limites muito rígidos.

4 Seção 3.4 — Fluxo de Assets

4.1 Do arquivo artístico ao jogo

Depois de desenhados, os assets não iam direto para o jogo de forma solta. O livro explica que eles passavam por uma etapa de empacotamento feita pela ferramenta *IGRAB*. Esse programa reunia os arquivos ILBM em arquivos de dados usados pelo jogo e ainda gerava um arquivo de cabeçalho em C com os identificadores de cada asset.

Na prática, isso significa que o código do jogo não precisava conhecer diretamente o nome ou a posição física de cada imagem dentro do arquivo final. Em vez disso, o programador usava identificadores simbólicos, como constantes de enumeração, e o sistema resolvia onde aquele asset estava armazenado.

4.2 Enums e indireção no código

O livro mostra que o motor acessava os assets por meio de um `enum`. Esse `enum` funcionava como um índice na tabela `HEAD`, e essa tabela apontava para a posição correta do asset dentro do arquivo de dados.

Esse detalhe é muito importante do ponto de vista de engenharia. Com essa camada de indireção, os assets podiam ser reorganizados, regenerados e até alterados de ordem sem exigir mudanças no código-fonte principal. O código continuava pedindo o mesmo identificador, e o sistema de dados cuidava do restante.

Em termos práticos, isso dava flexibilidade. A equipe podia atualizar o pacote gráfico sem precisar revisar manualmente cada ponto do código que usava imagens. Para um projeto em evolução, isso era uma vantagem enorme.

4.3 Compressão e organização

Outro ponto citado na seção é o papel do arquivo `DICT`, que continha a árvore de Huffman usada para descompressão dos assets. Isso mostra que o fluxo não era apenas uma questão de organização lógica, mas também de economia de espaço e eficiência no carregamento.

Em uma época em que memória e armazenamento eram recursos muito limitados, empacotar bem os dados fazia diferença real. O pipeline de assets precisava servir tanto ao artista quanto ao programador e ainda respeitar as restrições da máquina.

4.4 Esboço, arte-final e integração

A seção também traz uma observação interessante sobre o processo criativo. O manual oficial do jogo mostrava esboços feitos por Tom Hall e depois transformados em arte final pixelada pela equipe gráfica. Isso revela que o pipeline visual começava antes do desenho final no *Deluxe Paint*. Existia primeiro uma ideia, depois um esboço e, por fim, a adaptação para o formato técnico que o jogo realmente usaria.

Ou seja, o fluxo dos assets não era só técnico, mas também criativo. A ferramenta IGRAB representava apenas a parte final de uma cadeia que começava na concepção visual e terminava na integração com o motor.

4.5 Lição prática da seção

A seção 3.4 ensina que um jogo não depende apenas de bons arquivos gráficos, mas de um bom sistema para transformar esses arquivos em conteúdo utilizável pelo código. O pipeline de assets de *Wolfenstein 3D* era eficiente porque conectava arte, organização de dados e programação em um fluxo só. Isso reduzia retrabalho, facilitava manutenção e deixava o projeto mais escalável para os padrões da época.

5 Conclusão

As seções 3.2, 3.3 e 3.4 mostram que o desenvolvimento de *Wolfenstein 3D* foi muito mais do que escrever um motor rápido. O jogo também dependia de um ambiente de programação bem pensado, de uma produção gráfica disciplinada e de um pipeline sólido para integrar os assets ao código.

Na parte da programação, o destaque está no uso inteligente das ferramentas e na adaptação do trabalho às limitações reais dos monitores e do hardware. Na parte gráfica, fica evidente que a arte era moldada diretamente pelas restrições da VGA. Já no fluxo de assets, o ponto forte era a organização: os dados visuais eram empacotados, identificados e acessados de maneira estruturada, permitindo que arte e código trabalhassem juntos sem confusão.

No fim, essas seções ajudam a entender uma lição importante: jogos marcantes não surgem apenas de uma tecnologia forte ou de uma direção artística boa isoladamente. Eles surgem quando programação, arte e organização técnica funcionam em conjunto.