Seminário sobre o livro Wolfenstein 3D

Arthur Vieira Machado dos Santos
Davi Conde Garcia
Gabriel Oliveira de Souza Aguiar
Matheus Pastore Morgan
Thiago Gama Santoro Moreira

Vila Velha
2026

Universidade Vila Velha (UVV)
Ciência da Computação
Arquitetura de Computadores II
Prof. Abrantes Araújo

Resumo

Este trabalho apresenta um estudo baseado no livro sobre o desenvolvimento de *Wolfenstein 3D*, com foco na etapa chamada *Action Phase: 3D Renderer*. O objetivo é explicar como o jogo conseguia produzir uma sensação de ambiente tridimensional em computadores com hardware limitado, especialmente processadores 386 e placas VGA.

O conteúdo aborda conceitos como transformação de coordenadas, correção do efeito *fisheye*, desenho de paredes por colunas, uso de *compiled scalers*, renderização adiada de colunas, organização da memória VGA em bancos, texturização pré-iluminada, portas e paredes móveis. Esses temas mostram a relação direta entre arquitetura de computadores, desempenho, memória e técnicas de renderização em tempo real.

Sumário

1 Introdução

Wolfenstein 3D é considerado um dos jogos mais importantes da história dos jogos de tiro em primeira pessoa, também conhecidos como FPS. Apesar de não utilizar gráficos tridimensionais reais como os jogos modernos, ele criou uma forte ilusão de profundidade e movimento em primeira pessoa.

O jogo foi desenvolvido em uma época em que os computadores pessoais possuíam recursos bastante limitados. Processadores como o Intel 386 tinham pouca capacidade de processamento se comparados aos computadores atuais, e a memória de vídeo também exigia várias otimizações para que a imagem fosse desenhada rapidamente.

Por isso, o grande destaque técnico de *Wolfenstein 3D* não está apenas no resultado visual, mas nas soluções usadas para fazer o jogo rodar em tempo real. A engine precisava transformar um mapa bidimensional em uma cena aparentemente tridimensional, desenhar paredes texturizadas, corrigir distorções visuais e reduzir ao máximo o número de operações feitas pela CPU.

Este material serve como apoio para os alunos estudarem os principais pontos apresentados no seminário.

2 Contexto histórico

Na década de 1990, os jogos precisavam lidar diretamente com limitações de hardware. Diferente dos jogos atuais, que contam com placas de vídeo programáveis e APIs gráficas modernas, *Wolfenstein 3D* dependia fortemente da CPU para calcular e desenhar a cena.

O jogo utilizava uma técnica conhecida como *raycasting*. Essa técnica não renderiza um mundo 3D completo. Em vez disso, o mapa do jogo é representado como uma grade bidimensional, e raios são lançados a partir da posição do jogador para descobrir onde existem paredes. A partir da distância até essas paredes, a engine calcula a altura das colunas que devem aparecer na tela.

Essa abordagem era muito mais simples e rápida do que uma renderização 3D completa. Como consequência, o jogo conseguia atingir uma taxa de quadros jogável mesmo em computadores com baixo poder de processamento.

3 O renderizador 3D de Wolfenstein 3D

O renderizador de *Wolfenstein 3D* tinha a função de transformar informações do mapa em uma imagem na tela. O mapa era armazenado como uma grade 2D, onde cada célula podia representar espaço vazio, parede, porta ou outro elemento.

Durante a renderização, a engine calcula o que o jogador está enxergando. Para isso, ela considera:

- a posição do jogador no mapa;
- o ângulo de visão do jogador;
- os raios lançados para detectar paredes;
- a distância entre o jogador e os objetos;
- a altura da coluna a ser desenhada na tela;
- a textura correspondente à parede ou porta atingida.

A imagem final é formada por colunas verticais de pixels. Cada coluna da tela corresponde a um raio lançado no mapa. Se o raio encontra uma parede próxima, a coluna desenhada será alta. Se encontra uma parede distante, a coluna será menor.

4 Transformação de coordenadas: Map Space e Player Space

Um ponto importante do renderizador é a mudança do sistema de coordenadas. No mapa original, os pontos estão em *map space*, ou seja, no espaço do mapa. Porém, para desenhar corretamente a cena do ponto de vista do jogador, é necessário converter esses pontos para *player space*.

No *player space*, o jogador passa a ser considerado a origem do sistema de coordenadas, isto é, a posição $(0, 0)$. Dessa forma, fica mais fácil calcular o que está à frente, ao lado ou atrás do jogador.

A transformação pode ser entendida como uma rotação do sistema de coordenadas de acordo com o ângulo de visão do jogador. A fórmula geral pode ser representada por uma matriz de rotação:

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \cdot \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} dx \cdot \cos(\alpha) - dy \cdot \sin(\alpha) \\ dx \cdot \sin(\alpha) + dy \cdot \cos(\alpha) \end{bmatrix}$$

Nessa fórmula:

- dx representa a diferença de posição no eixo X;
- dy representa a diferença de posição no eixo Y;
- α representa o ângulo de visão do jogador;
- o resultado indica a posição do ponto em relação ao jogador.

Essa transformação é essencial porque a engine precisa renderizar o mundo a partir da perspectiva do jogador, e não apenas a partir das coordenadas absolutas do mapa.

5 Correção do efeito Fisheye

Um problema comum em renderizadores baseados em *raycasting* é o efeito *fisheye*. Esse efeito acontece quando a distância direta do raio é usada para calcular a altura da parede. Como os raios laterais percorrem uma distância maior do que o raio central, as paredes acabam ficando visualmente curvadas ou distorcidas.

Para corrigir isso, a engine não deve usar simplesmente a distância direta d entre o jogador e a parede. Em vez disso, deve usar a distância projetada na direção da visão do jogador. Essa distância projetada pode ser representada por z.

A ideia é que a altura da parede seja calculada com base na profundidade real em relação ao olhar do jogador, e não no comprimento inclinado do raio.

De forma simplificada:

$$z = dx \cdot \cos(\alpha) - dy \cdot \sin(\alpha)$$

Com essa correção, as paredes permanecem retas na tela, evitando a distorção visual causada pelo efeito *fisheye*.

5.1 Importância da correção

Sem essa correção, o cenário pareceria deformado, principalmente nas laterais da tela. A correção do *fisheye* melhora a percepção visual e torna a movimentação mais natural para o jogador.

6 Desenho de paredes

Depois que a engine calcula a distância correta até a parede, o próximo passo é desenhar a coluna correspondente na tela. Cada parede é formada por várias colunas verticais de pixels.

A altura de cada coluna depende da distância entre o jogador e a parede. Quanto mais perto a parede estiver, maior será a coluna desenhada. Quanto mais longe, menor será a coluna.

A dificuldade está no fato de que cada coluna possui uma textura. A engine precisa pegar uma coluna de textura, normalmente com 64 pixels de altura, e redimensioná-la para diferentes alturas na tela.

Esse processo de redimensionamento é chamado de *scaling*. Em computadores modernos, isso parece simples. Porém, em um processador 386, escalar milhares de colunas de pixels em tempo real era uma tarefa cara.

Por isso, *Wolfenstein 3D* usou duas grandes otimizações:

- *Compiled scalers*;
- *Deferred column rendering*.

7 Compiled Scalers

Os *compiled scalers* foram uma das principais otimizações usadas para desenhar paredes rapidamente. A ideia era trocar processamento por memória.

Em uma abordagem genérica, uma função de escala poderia receber qualquer altura e calcular, pixel por pixel, qual parte da textura deveria ser copiada para a tela.

Um exemplo simplificado seria:

```
void scaleTextureToHeight(int height, void* src, void* dst) {
    int src_cursor = 0;
    int dst_cursor = 0;
    int step = FixedDiv(64, height);

    while (height > 0) {
        dst[dst_cursor] = src[src_cursor >> 8];
        src_cursor += step;
        height--;
        dst_cursor++;
    }
}
```

Essa função é flexível, mas possui custo alto. Ela precisa executar um laço, atualizar variáveis intermediárias e calcular a posição correta da textura para cada pixel desenhado.

A solução usada em *Wolfenstein 3D* foi gerar funções específicas para determinadas alturas. Em vez de uma função genérica, a engine podia ter funções já preparadas para escalar a textura para alturas fixas.

Exemplo conceitual:

```
void scaleTextureTo2(void* src, void* dst) {
    dst[0] = src[16];
    dst[1] = src[48];
}

void scaleTextureTo4(void* src, void* dst) {
    dst[0] = src[0];
    dst[1] = src[16];
    dst[2] = src[32];
    dst[3] = src[63];
}
```

Essas funções são menos flexíveis, mas muito mais rápidas. Como a altura já está embutida no código, a CPU executa menos instruções.

7.1 Geração de código em tempo de execução

A engine gerava código de máquina durante a inicialização do jogo. Esse processo criava funções especializadas para escalar texturas em alturas específicas.

A vantagem era o ganho de desempenho. A desvantagem era o uso de memória RAM. Quanto mais funções pré-geradas, mais memória era consumida.

O livro mostra que gerar muitos *scalers* poderia ocupar uma quantidade alta de memória. Por isso, a engine escolhia apenas alguns tamanhos, evitando gerar todos os possíveis. Essa decisão representa um clássico compromisso em arquitetura de computadores:

usar mais memória para economizar processamento.

7.2 Relação com Arquitetura de Computadores

Essa técnica é importante para a disciplina porque mostra como o desempenho depende diretamente da arquitetura. Em vez de tratar a CPU como infinitamente rápida, os programadores precisavam pensar em:

- quantidade de instruções executadas;
- uso de memória RAM;
- custo de laços de repetição;
- acesso à memória de vídeo;
- redução de operações durante o jogo.

8 Deferred Column Drawing

Outra otimização importante foi a renderização adiada de colunas, chamada de *deferred column drawing*.

Em um renderizador simples, cada coluna seria calculada e desenhada imediatamente:

```
for (int x = 0; x < 320; x++) {
    castRay();
    height = CalculateWallHeight();
    drawColumn(x, height);
}
```

Porém, *Wolfenstein 3D* podia adiar o desenho de colunas parecidas. A engine comparava raios consecutivos. Se eles atingissem a mesma parede, com textura semelhante e altura parecida, as colunas eram agrupadas em um buffer.

A lógica simplificada seria:

```
for (int x = 0; x < 320; x++) {
    castRay();

    if (raySimilarToOnesInBuffer()) {
        AddColumnToBuffer();
        continue;
    } else {
        DrawBuffer();
        height = CalculateWallHeight();
        AddColumnToBuffer();
    }
}
```

Essa técnica reduzia o número de escritas na tela e permitia que a engine desenhasse várias colunas de maneira mais eficiente.

8.1 Por que isso melhora o desempenho?

A melhoria acontece porque raios consecutivos frequentemente atingem a mesma parede, principalmente quando o jogador está olhando para uma parede próxima. Nesses casos, várias colunas possuem características muito parecidas.

Agrupar essas colunas reduz trabalho repetido e aproveita melhor a organização da memória de vídeo VGA.

9 Bancos VGA e máscaras de escrita

A memória VGA usada pelo jogo não funcionava como uma memória linear simples. A tela era dividida em bancos de memória. Cada banco armazenava colunas específicas de pixels.

De maneira simplificada:

- o banco 0 armazenava as colunas 0, 4, 8, 12, e assim por diante;
- o banco 1 armazenava as colunas 1, 5, 9, 13, e assim por diante;
- o banco 2 armazenava as colunas 2, 6, 10, 14, e assim por diante;
- o banco 3 armazenava as colunas 3, 7, 11, 15, e assim por diante.

Para selecionar quais bancos seriam escritos, a engine usava uma máscara VGA. Cada bit representava um banco:

Banco	Valor da máscara
0	1
1	2
2	4
3	8

Assim, se a engine quisesse escrever nos bancos 0 e 1 ao mesmo tempo, usaria:

$$1 + 2 = 3$$

Se quisesse escrever em todos os bancos, usaria:

$$1 + 2 + 4 + 8 = 15$$

9.1 Exemplo de máscara

Se duas colunas estivessem alinhadas nos bancos 0 e 1, a engine poderia desenhá-las em uma única passagem usando a máscara 3.

Porém, se as colunas não estivessem bem alinhadas, seria necessário fazer mais passagens. Em alguns casos, para desenhar oito colunas consecutivas, a engine precisava de até três passagens diferentes, usando máscaras como 8, 15 e 7.

Essa técnica mostra como o conhecimento da memória de vídeo era essencial para ganhar desempenho.

10 Texturização

A texturização de paredes era outro ponto importante da renderização. Cada parede não era apenas uma cor sólida, mas sim uma imagem aplicada sobre a superfície.

Um truque usado pela engine era gerar versões diferentes da mesma textura: uma versão iluminada e outra versão menos iluminada. Isso é chamado de textura pré-iluminada ou *baked texture*.

Durante a execução, a engine escolhia qual textura usar dependendo da direção da parede atingida pelo raio:

- se o raio atingisse uma parede vertical, a engine poderia usar uma versão da textura;
- se o raio atingisse uma parede horizontal, poderia usar outra versão.

Esse efeito criava uma sensação visual de iluminação direcional, deixando o cenário mais realista sem precisar calcular luz dinamicamente.

10.1 Vantagem da textura pré-iluminada

Calcular iluminação real em tempo de execução seria caro demais para o hardware da época. A solução foi preparar as texturas antes e apenas escolher a versão correta durante o jogo.

Essa é outra demonstração de troca entre armazenamento e processamento:

guardar mais dados para evitar cálculos em tempo real.

11 Portas

As portas em *Wolfenstein 3D* eram tratadas de forma especial pelo raycaster. Diferente das paredes comuns, as portas podiam estar fechadas, parcialmente abertas ou totalmente abertas.

Por isso, a engine mantinha um vetor chamado `doorposition`, que indicava o quanto cada porta estava aberta.

Quando um raio atingia uma célula de porta, a engine verificava se o raio deveria parar na porta ou atravessá-la. De forma simplificada, o teste podia ser representado por:

$$A_x + \frac{ystep}{2} < doorposition[doorIndex]$$

Se o teste indicasse que a porta bloqueava o raio, a engine calculava o ponto de interseção e enviava essa informação para o renderizador.

As coordenadas de interseção podiam ser calculadas de forma semelhante a:

$$Intercept_x = A_x + \frac{ystep}{2}$$

$$Intercept_y = A_y + \frac{TILE\_SIZE}{2}$$

As portas não possuíam espessura real como um objeto 3D moderno. Elas eram renderizadas diretamente pelo raycaster, o que simplificava a implementação e mantinha o desempenho.

12 Push Walls

As *push walls*, ou paredes móveis, também eram implementadas por meio do ray-caster. Elas eram tratadas como casos especiais, assim como as portas.

Quando uma parede móvel era ativada, a engine ajustava as coordenadas de interseção do raio de acordo com o deslocamento da parede. Dessa forma, a parede parecia se mover dentro do ambiente.

Essa solução evitava a necessidade de um sistema físico complexo. O movimento era representado por ajustes matemáticos no momento em que o raio encontrava a parede.

13 Comparação com DOOM

O livro também menciona uma comparação com *DOOM*, outro jogo importante da id Software. Em *DOOM*, foi usada uma técnica chamada modo de baixa resolução, na qual colunas de pixels podiam ser duplicadas horizontalmente.

Essa técnica reduzia o número de operações de escrita na tela. Em sequências de ação, a diferença visual era pouco perceptível, mas o ganho de desempenho era significativo.

A ideia é parecida com outras otimizações vistas em *Wolfenstein 3D*: diminuir a quantidade de trabalho da CPU para manter uma taxa de quadros jogável.

14 Relação com Arquitetura de Computadores

O estudo do renderizador de *Wolfenstein 3D* é relevante para Arquitetura de Computadores porque mostra como o software precisa se adaptar ao hardware disponível.

As principais relações com a disciplina são:

- **CPU limitada:** muitas técnicas foram criadas para reduzir o número de instruções executadas.
- **Memória como recurso estratégico:** a engine armazenava dados e funções pré-geradas para evitar cálculos em tempo real.
- **Organização da memória de vídeo:** o uso de bancos VGA influenciava diretamente a forma como os pixels eram desenhados.
- **Otimização de baixo nível:** partes do código usavam instruções próximas da linguagem de máquina para aumentar o desempenho.
- **Compromisso entre qualidade e velocidade:** o jogo usava simplificações visuais para conseguir rodar bem.

Esses pontos mostram que desempenho não depende apenas de algoritmos abstratos, mas também da forma como o hardware executa instruções, acessa memória e organiza dados.

15 Principais conceitos para estudar

15.1 Raycasting

Técnica usada para lançar raios a partir da posição do jogador e detectar paredes no mapa. Cada raio gera uma coluna da imagem final.

15.2 Map Space

Sistema de coordenadas do mapa. Nele, os objetos possuem posição absoluta dentro da grade do jogo.

15.3 Player Space

Sistema de coordenadas em que o jogador é considerado a origem. Esse espaço facilita o cálculo da renderização a partir do ponto de vista do jogador.

15.4 Fisheye

Distorção visual causada pelo uso da distância direta do raio. É corrigida usando a distância projetada na direção da visão do jogador.

15.5 Compiled Scalers

Funções geradas para escalar colunas de textura rapidamente. Elas reduzem o trabalho da CPU, mas aumentam o uso de memória.

15.6 Deferred Column Drawing

Técnica que agrupa colunas parecidas antes de desenhá-las, reduzindo escritas repetidas na memória de vídeo.

15.7 VGA Banks

Organização da memória VGA em bancos. A engine usava máscaras para escrever em bancos específicos e acelerar o desenho das colunas.

15.8 Baked Textures

Texturas pré-iluminadas. Em vez de calcular luz em tempo real, o jogo usava versões já preparadas das texturas.

15.9 Doors e Push Walls

Portas e paredes móveis eram tratadas como casos especiais no raycaster, permitindo interação no cenário sem usar uma geometria 3D complexa.

16 Conclusão

O estudo do renderizador de *Wolfenstein 3D* mostra como grandes resultados visuais podem ser alcançados mesmo com hardware limitado, desde que o software seja cuidadosamente otimizado.

A engine do jogo usava diversas estratégias para reduzir o trabalho da CPU e aproveitar melhor a memória de vídeo. Entre elas estavam a correção do efeito *fisheye*, o desenho por colunas, os *compiled scalers*, a renderização adiada, o uso de máscaras VGA e as texturas pré-iluminadas.

Essas técnicas mostram uma forte relação entre programação, arquitetura de computadores e computação gráfica. O jogo é um exemplo importante de como conhecer o funcionamento do hardware permite criar soluções eficientes e inovadoras.

Portanto, *Wolfenstein 3D* não foi importante apenas para a história dos jogos, mas também como exemplo técnico de otimização, criatividade e uso inteligente dos recursos computacionais disponíveis na época.

Referências

SANGLARD, Fabien. *Game Engine Black Book: Wolfenstein 3D*. Estudo sobre a engine, renderização, otimizações e funcionamento interno do jogo *Wolfenstein 3D*.

Materiais apresentados em aula e imagens utilizadas como base para o seminário.