Wolfenstein 3D

Capítulos 1 e 2.1

Brenno Gomes Breda

Daphne Rocha Amigo
Isabel Emília Sterim Saade
Maria Carla dos Santos Bellote
Rhuan Santos Wolfgramm

07-05-2026

Resumo

Este artigo resume os Capítulos 4.7.5 do livro sobre a engenharia por trás de Wolfenstein 3D. Explicamos como o *Wolfenstein 3D* contornou a limitação dos PCs de 1992, que não possuíam unidade de ponto flutuante rápida o suficiente para um motor 3D. Primeiro, o motor substituiu *floats* por aritmética de ponto fixo. Números fracionários são armazenados em inteiros de 32 bits divididos em parte inteira e fracionária. Adição, subtração e deslocamentos funcionam diretamente na ALU, enquanto a multiplicação exige rotinas dedicadas que aceitam pequena perda de precisão em troca de velocidade. Segundo, o sistema de coordenadas foi projetado para aritmética inteira. A origem fica no canto superior esquerdo, o mundo é uma grade de 64×64 blocos, e ângulos são inteiros de 0 a 3600 representando décimos de grau. Terceiro, o renderizador explora esse mundo quadrado com *ray casting* para resolver a determinação de superfície visível (VSD). Em vez de rasterizar polígonos arbitrários, o algoritmo percorre a grade e descarta rapidamente geometria não visível, abordagem destacada por Michael Abrash como o núcleo do desempenho. Juntas, essas três decisões permitiram trigonometria, movimento suave e projeção em tempo real usando apenas operações inteiras do 386, técnica que seria replicada em consoles dos anos 90 como PlayStation e Saturn.

Sumário

1 Resolvendo o Problema da CPU

Anteriormente nos deparamos com um problema: a máquina não consegue realizar Operações de Ponto Flutuante com rapidez suficiente. Isso representa um problema considerável para um motor gráfico 3D, dada toda a **trigonometria** envolvida. A solução encontrada foi enganar a ULA por meio de uma técnica chamada **"aritmética de ponto fixo"**.

2 Ponto-Fixo

O PC de 1992 (386/486 sem coprocessador matemático) não conseguia fazer contas com vírgula flutuante (*float*) rápido o suficiente para um motor 3D. A solução do id Software foi enganar a CPU: tratar números fracionários como inteiros, usando aritmética de ponto fixo.

2.1 Como funciona

Em vez de um `int` normal, onde todos os bits valem potências de 2 positivas, o programador divide o número em duas partes.

Exemplo do livro, formato 8:8 (8 bits inteiros, 8 bits fracionários):
O mesmo padrão de bits `0010001000100010`:

- Como inteiro puro = 8.738
- Como 8:8 = 34 (parte inteira) $+ 0,1328125$ (parte fracionária) = $34,1328125$

A CPU continua somando e subtraindo como se fossem inteiros, e o resultado continua correto. Deslocamentos também funcionam: << multiplica por 2, >> divide por 2.

A notação usada é `BITS_INTEIROS:BITS_FRACAO`. No *Wolfenstein 3D* a posição do jogador usa 16:16 (32 bits no total). Por isso, para saber em qual quadrado do mapa ele está, basta descartar a fração com um shift:

```
player->x >> 16
```

2.2 Multiplicação

Multiplicar dois valores 16:16 gera um resultado 32:32, que não cabe em 32 bits. Havia duas saídas:

- Multiplicar em 64 bits e depois jogar fora os 16 bits de cima e de baixo;
- Perder precisão antes, deslocando os operandos (o que o jogo fazia)

O livro mostra $98,7539 \times 1,5$ em 8:8. Os dois números são deslocados 4 bits à direita antes da multiplicação, o que apaga os bits menos significativos. O resultado sai $148,125$ em vez de $148,13085$. Era um *trade-off* aceitável: um pouco de erro, mas muito mais rápido.

No código, tudo é:

```
typedef long fixed;
```

Mas nem todo `fixed` é igual. O engine tem funções diferentes:

- `FixedMul(a,b)` – versão simples e rápida, usada quando o sinal é conhecido: `(a>>8)*(b>>8)`
- `FixedByFrac(a,b)` – versão em assembly que trata sinal, multiplica a parte alta e baixa separadamente e recoloca em complemento de dois.

2.3 Conclusão prática

O ponto fixo permitiu que *Wolfenstein 3D* fizesse **trigonometria, movimento suave e projeção 3D** usando só operações inteiras do 386. Não foi exclusividade do PC: PlayStation 1 e Sega Saturn, lançados em 1994, também não tinham unidade de ponto flutuante para redução de seu custo, e usaram a mesma técnica.

3 Sistema de Coordenadas

A game engine de Wolfeinstein 3D inicia seu sistema de coordenada no canto superior esquerdo do mapa, se expandido em uma grade de 64x64 blocos, onde cada unidade de bloco representa 2,4 metros. O sistema utiliza aritmética de ponto fixo em quase todas as variáveis, como a posição do jogador em 16:16, com exceção dos ângulos, que são números inteiros representando décimos de graus.

4 O Desafio da Superfície Visível (VSD)

Para Michael Abrash, renomado escritor de sua época pelas suas publicações sobre programação Assembly e Computação Gráfica, o desafio do VSD

5 Raycasting com Grid

O Raycasting com Grid foi a técnica utilizada para criar a sensação de ambiente tridimensional utilizando hardware limitado. O mapa do jogo era composto por uma grade de blocos quadrados, permitindo que os raios de visão do jogador verificassem colisões apenas ao atravessar as linhas do grid.

Essa abordagem reduzia drasticamente o processamento necessário, tornando o algoritmo rápido e preciso. Cada raio lançado pelo jogador percorria o mapa até encontrar uma parede, e a distância calculada era usada para definir o tamanho da parede renderizada na tela, criando a ilusão de profundidade já que quando mais longe a parede estivesse menor ela seria processada.

A utilização do grid também explicava as limitações do jogo, como paredes sempre retas e ambientes formados principalmente por corredores e salas quadradas similar a um grande labirinto.

Curiosidade: BSP

Durante a adaptação de Wolfenstein 3D para o Super Nintendo, o processador do console apresentou dificuldades para executar o raycasting de forma eficiente. Para melhorar o desempenho, John Carmack utilizou BSP (*Binary Space Partitioning*), uma técnica que dividia o mapa em regiões menores para acelerar a ordenação e renderização dos elementos da cena.

As paredes eram desenhadas ”de perto para longe”, utilizando uma matriz de oclusão para evitar o desenho de elementos escondidos, reduzindo o custo computacional e melhorando a taxa de quadros do jogo.

6 Call Apogee

O “Call Apogee” foi um segredo escondido no mapa E2M8 do Wolfenstein 3D. Os jogadores que encontrassem a área secreta veriam uma placa com a mensagem: ”Call Apogee and say Aardwolf”.

A ideia original era realizar um concurso promovido pela Apogee para recompensar os jogadores que descobrissem o segredo. No entanto, após o lançamento do jogo, muitos jogadores passaram a utilizar programas de trapaça e engenharia reversa que revelavam o mapa completo, tornando fácil localizar a sala secreta. Por esse motivo, o concurso acabou sendo cancelado.

> “A placa ainda estava no jogo, mas, em retrospectiva, provavelmente deveria ter sido removida. Até hoje, a Apogee recebe cartas e telefonemas perguntando o que é Aardwolf, frequentemente com a pergunta: ‘Alguém já viu isso?’ ”
>
> — Joe Siegler

O nome ”Aardwolf” foi escolhido por ser o primeiro arquivo de imagem presente no dicionário das máquinas NeXT utilizadas pela equipe da id Software, tornando-se posteriormente um mascote interno da empresa.

7 Raycasting e o Algoritmo DDA

O motor gráfico de *Wolfenstein 3D* utiliza a técnica de *raycasting* para projetar um ambiente tridimensional a partir de um mapa estritamente bidimensional. O processo consiste em projetar um raio para cada coluna de pixels da tela, partindo da posição do jogador, para localizar o ponto exato de colisão com as paredes do cenário.

Para otimizar essa busca, o motor implementa o algoritmo *Digital Differential Analyzer* (DDA). Em vez de verificar colisões passo a passo ao longo do raio, o DDA salta entre as linhas de grade (*grid lines*) horizontais e verticais do mapa. A implementação original de John Carmack foi escrita em aproximadamente 740 linhas de Assembly, utilizando uma estrutura de controle baseada em `gotos` que alternam entre verificações horizontais e verticais. Embora essa abordagem com múltiplos saltos (*jumps*) seja prejudicial em CPUs modernas devido ao *pipeline* e ao *instruction cache*, ela era extremamente eficiente na arquitetura Intel 386, que não possuía cache de instrução.

7.1 Funcionamento do DDA

Diferente de uma busca linear, o DDA decide a cada iteração qual é a próxima fronteira de célula mais próxima, reduzindo drasticamente o processamento:

```
Para cada coluna da tela:
    Calcular o ângulo do raio e vetores de direção
    Inicializar variáveis de passo (xstep, ystep)
    Enquanto parede não encontrada:
        Identificar qual interseção é a mais próxima
        Saltar para a respectiva fronteira da grade
        Verificar se a célula atingida é uma parede ou porta
```

7.2 Base Matemática

O deslocamento do raio baseia-se em trigonometria fundamental aplicada ao círculo unitário. Para calcular quanto o raio avança no eixo y ao percorrer uma unidade no eixo x, utiliza-se a função tangente:

- Ao avançar 1 unidade em x, o deslocamento em y é $\tan(\theta)$.
- Ao avançar 1 unidade em y, o deslocamento em x é $\tan(90^\circ - \theta)$.

Para evitar o custo computacional de funções trigonométricas em tempo real, o motor utiliza tabelas de consulta pré-computadas. Toda a lógica de visibilidade e projeção repousa sobre as relações da fácil técnica de memorização SOH-CAH-TOA:

- $\sin(\theta) = \frac{\text{Oposto}}{\text{Hipotenusa}}$

- $\cos(\theta) = \frac{\text{Adjacente}}{\text{Hipotenusa}}$
- $\tan(\theta) = \frac{\text{Oposto}}{\text{Adjacente}}$

8 Problema do Efeito Fish Eye em Raycasting e sua Solução Matemática

8.1 Cálculo esperado da distância

O uso da distância euclidiana direta pode levar à ocorrência do efeito fish eye. Ainda assim, essa abordagem representa o cálculo geométrico mais intuitivo para determinar a distância entre o jogador e o ponto de interseção do raio com a parede.

Essa distância é definida como:

$$d = \sqrt{dx^2 + dy^2} \tag{1}$$

em que dx e dy correspondem às diferenças nas coordenadas horizontal e vertical entre a posição do jogador e o ponto de colisão.

Embora esse cálculo represente corretamente a distância em linha reta no plano cartesiano, sua aplicação direta na renderização resulta em distorções visuais, pois não considera a projeção da cena no plano da câmera.

8.2 Problema do efeito Fish-eye

A utilização direta da distância euclidiana no algoritmo de raycasting resulta em uma distorção visual conhecida como efeito fish eye. Esse efeito ocorre porque os raios lançados nas extremidades do campo de visão percorrem distâncias maiores em comparação aos raios centrais.

Como consequência, paredes localizadas nas laterais da tela parecem mais distantes e, portanto, são renderizadas com menor altura, mesmo quando estão alinhadas com as paredes centrais. Isso gera uma percepção distorcida do ambiente, semelhante à produzida por lentes grande-angulares (por isso fish eye).

A intensidade dessa distorção aumenta conforme o jogador se aproxima das paredes, tornando-se mais perceptível em curtas distâncias.

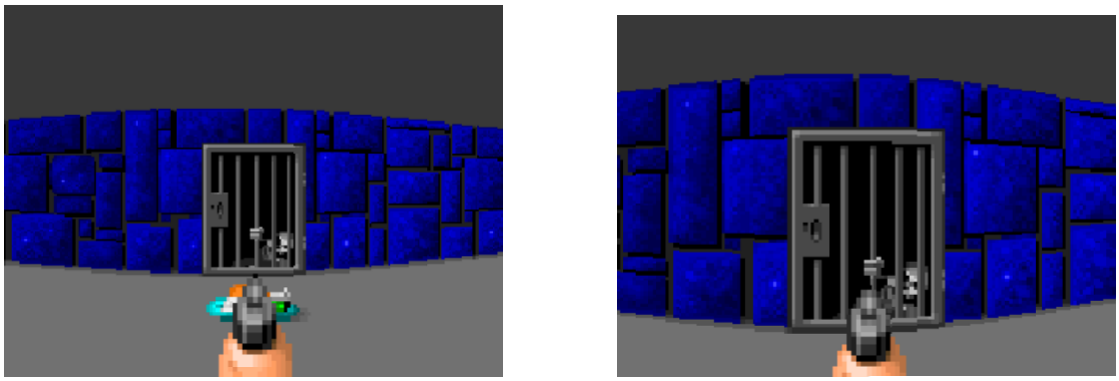

Figura 1: Distorção causada pelo efeito fish eye em diferentes distâncias

Nas imagens é possível notar a distorção das paredes,em que quanto mais perto o jogador está, mais distantes elas parecem nas extremiadades da visão.

8.3 Cálculo da distância projetada

O cálculo da distância projetada é realizado por meio da decomposição do vetor posição em dois componentes, utilizando relações trigonométricas básicas (SOH--CAH-TOA).

Inicialmente, definem-se os componentes:

$$A = dx \cdot \cos(\alpha) \tag{2}$$

$$B = dy \cdot \sin(\alpha) \tag{3}$$

A soma desses componentes resulta na distância projetada:

$$z = A + B = dx \cdot \cos(\alpha) + dy \cdot \sin(\alpha) \tag{4}$$

Considerando que dx e dy são grandezas direcionais (podendo assumir valores negativos), a equação é ajustada para:

$$z = dx \cdot \cos(\alpha) - dy \cdot \sin(\alpha) \tag{5}$$

Essa é a forma efetivamente utilizada pelo motor do jogo.

8.4 Interpretação geométrica

O cálculo da distância projetada pode ser interpretado como uma transformação de coordenadas, na qual o sistema de referência do mapa é rotacionado para o sistema de referência do jogador.

Matematicamente, essa transformação pode ser representada por uma matriz de rotação:

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \cdot \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} dx \cdot \cos(\alpha) - dy \cdot \sin(\alpha) \\ dx \cdot \sin(\alpha) + dy \cdot \cos(\alpha) \end{bmatrix} \tag{6}$$

O valor utilizado para a renderização corresponde ao primeiro componente do vetor resultante, que representa a distância projetada no plano da câmera.

A utilização dessa abordagem é suficiente para eliminar o efeito fish eye, resultando em paredes visualmente retas e proporcionalmente corretas.