Gerardo Mikael Do Carmo Pereira
Fernando de Magalhães Toledo

Trabalho de Álgebra Linear
Raycasting

Rio de Janeiro
9 de Novembro de 2022

Gerardo Mikael Do Carmo Pereira
Fernando de Magalhães Toledo

Trabalho de Álgebra Linear
Raycasting

Relatório apresentado à matéria de Álgebra Linear para obtenção de nota parcial, Fundação Getúlio Vargas, Escola de Matemática Aplicada do Rio de Janeiro

Orientador: Yuri F Saporito

Rio de Janeiro
9 de Novembro de 2022

Resumo

Raycasting é uma técnica de renderização para criar uma perspectiva 3D em um mapa 2D, esse o algoritmo de Ray casting usa o lançamento de raios a partir do observador e calcula a distância que estão os objetos que compõem a cena. O Ray casting permite ignorar as superfícies escondidas numa imagem utilizando, para isso, as informações obtidas a partir das primeiras intersecções encontradas pelos raios lançados, economizando muito processamento. Pode-se considerar que o raycasting é uma versão abreviada muito mais simples e mais rápida do algoritmo de Raytracing. Muito antes da possibilidade do uso de engines que renderizam gráficos 3D em tempo real, os computadores eram muito mais lentos, logo não era possível executar esse tipo de renderização 3D. O raycasting foi a primeira solução pensada para fazer esse tipo de render, ele poderia ser utilizado na época por ter um processamento muito mais simples, e por fazer apenas um cálculo para cada linha vertical da tela.

Introdução

Wolfeinstein 3D, lançado em 5 de maio de 1992 foi um dos primeiros jogos a tentar introduzir os gráficos 3D ao mundo dos jogos, deixando de lado as plataformas onde o jogador se movimentava da esquerda para a direita para criar um mundo tridimensional onde o jogador adentrava os mapas. Utilizando o raycasting devido às limitações da época Wolfeinstein 3D foi uma grande tendencia e houve grandes jogos que nos anos seguintes utilizaram da mesma técnica em suas produções como Doom (1993) e ShadowCaster.

O objetivo desde projeto é gerar um mapa em '3D' usando raycasting no estilo de wolfeinstein.

Metodologia

O mecanismo de raycasting do Wolfeinstein era bastante limitado, mas foi uma revolução pra época, no sistema do jogo todas as paredes têm a mesma altura e são quadrados ortogonais em uma grade 2D. A partir desse mapa de grade quadrada, onde cada quadrado pode ser um valor inteiro positivo (cada número se refere a uma textura diferente) o mapa é gerado como no exemplo mapa exemplo.

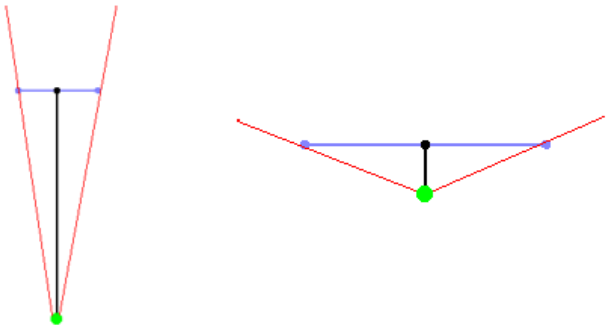
```
mapa_exemplo = ([1,1,2,2,1,1,1,1,2,2,1,1],
                [1,0,0,0,0,0,0,0,0,0,0,1],
                [1,0,1,2,0,0,0,2,1,0,0,1],
                [1,0,1,2,0,0,0,2,1,0,0,1],
                [1,0,1,2,0,0,0,2,1,0,0,1],
                [1,0,0,0,0,0,0,0,0,0,0,1],
                [1,0,0,0,0,0,0,0,0,0,0,1],
                [1,1,2,2,1,1,1,1,2,2,1,1])
```

Valores iniciais

O jogador terá uma posição inicial (x, y) no mapa além de uma direção inicial que ele estará visualizando. A partir disso para cada pixel horizontal na tela será enviado um raio que começa na posição do jogador e com uma direção que depende tanto da direção do olhar do jogador quanto da coordenada x da tela. O raio vai avançar no mapa 2D até atingir um quadrado que é uma parede. Nesse momento será calculada a distância desse ponto até o jogador e a distância será usada para calcular a altura que essa parede deve ser desenhada na tela. Logicamente quanto mais longe a parede, menor ela fica na tela e quanto mais próxima, mais alto ela é desenhada.

POV

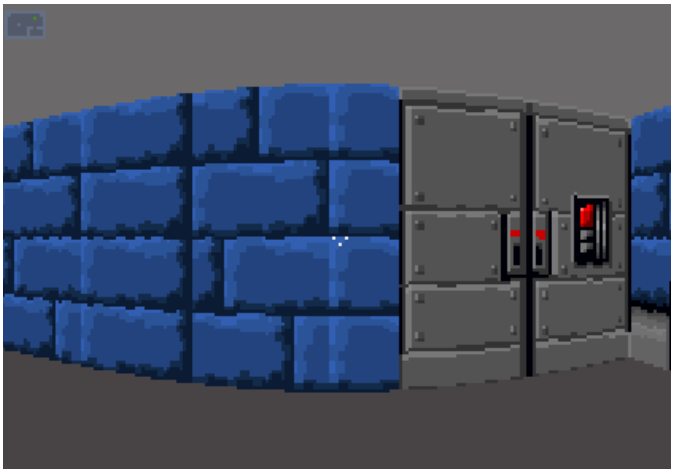
É necessário definir também o campo de visão do jogador que será o ângulo entre a primeira linha a ser calculada e a última. Definir esse campo de visão é responsável pelo 'zoom' da visão do jogador, quanto menor o FOV (Field of view) mais estreito o campo de visão. O jogador então verá mais detalhes e haverá menos profundidade, então isso é o mesmo que aumentar o zoom. No efeito contrário, quanto maior o FOV menos detalhes o jogador verá, mas haverá mais profundidade, como se diminuísse o zoom. Em nosso projeto usaremos um pov de 60°.



FOV menor (à esquerda) e FOV maior (à direita)

Fisheye

Nesse ponto encontramos um problema, pois ao usar a distância do raio do jogador à parede para calcular sua altura obtemos um efeito olho de peixe que distorce as imagens obtidas dando um aspecto arredondado a objetos que teoricamente são planos.



Fisheye

Isso é resolvido facilmente se dividirmos o tamanho da linha gerada pelo cosseno do ângulo do range da visão do jogador. O valor é menor nas laterais, logo ao fazer a divisão da distância pelo ângulo é feita a correção da distorção.

Movimentação do jogador

Nesse projeto o jogador tem apenas duas opções de movimentação, ir para frente ou para trás, e duas opções de rotação da câmera, para a esquerda e direita. Usaremos então as matrizes de transformação nas cordenadas do jogador.

$$T(t_x, t_y) * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

Isso só é possível ao adaptarmos a matriz da seguinte forma:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

Nesse primeiro caso a nova posição do jogador ao se movimentar para frente será obtida somando o cossenodo ângulo de rotação à posição x inicial, e o seno do ângulo de rotação à posição y inicial. Da mesma maneira ao se movimentar para trás a nova posição do jogador será obtida diminuindo o cosseno do ângulo de rotação à posição x inicial, e o seno do ângulo de rotação à posição y inicial.

$$\begin{bmatrix} 1 & 0 & \pm\alpha\cos\theta \\ 0 & 1 & \pm\alpha\sin\theta \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \pm \alpha\cos\theta \\ y \pm \alpha\sin\theta \\ 1 \end{bmatrix}$$

Quando o jogador gira, a câmera precisa girar, então tanto o vetor de direção quanto o vetor de plano da câmera devem ser girados. Então, todos os raios irão girar automaticamente também. Em relação a rotação da visão do jogador, o valor é obtido através da matriz de rotação:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + x\cos\theta \end{bmatrix}$$

Geração das imagens na tela

Quando fazemos a projeção em perspectiva usamos os pontos do modelo para formar a imagem que é exibida na tela. No rastreamento de raios, começamos do ponto de vista, enviamos raios através do plano de visão (a tela), e determinamos quais objetos são "atingidos" primeiro por cada raio. Temos então dispostos na tela uma resolução horizontal e uma resolução vertical, medidas em píxeis considerando cada pixel numa posição (j,i) onde j é sua linha e i sua coluna.

```
for i in range(res_horizontal):
    rot_i = rot + np.deg2rad(i/fov - 30)
    sin, cos, cos_correcao = np.sin(rot_i), np.cos(rot_i)
    , np.cos(np.deg2rad(i/fov-30))
    frame[i][:] = ceu[int(np.rad2deg(rot_i)%359)][:]/255

    for j in range(half_res_vertical):
        n = (half_res_vertical/(half_res_vertical-j))/cos_correcao
        x, y = posx + cos*n, posy + sin*n
```

```
xfloor, yfloor = int(x*2%1*299), int(y*2%1*299)

#Geracao das paredes textura 1
if mapa[int(x)%(tamanho-1)][int(y)%(tamanho-1)] == 1:
    h = half_res_vertical - j
    if x%1<0.019 or x%1>=0.981:
        xfloor = yfloor
    yfloor = np.linspace(0,598, h*2)%299

    for k in range(h*2):
        frame[i][half_res_vertical - h + k] = wall[xfloor]
        [int(yfloor[k])]/255
    break

#Geracao das paredes textura 2
elif mapa[int(x)%(tamanho-1)][int(y)%(tamanho-1)] == 2:
    h = half_res_vertical - j
    if x%1<0.015 or x%1>=0.985:
        xfloor = yfloor
    yfloor = np.linspace(0,598, h*2)%299

    for k in range(h*2):
        frame[i][half_res_vertical - h + k] = wall2[xfloor]
        [int(yfloor[k])]/255
    break

#Geracao das paredes textura 3
elif mapa[int(x)%(tamanho-1)][int(y)%(tamanho-1)] == 3:
    h = half_res_vertical - j
    if x%1<0.015 or x%1>=0.985:
        xfloor = yfloor
    yfloor = np.linspace(0,299, h*2)%299

    for k in range(h*2):
        frame[i][half_res_vertical - h + k] = wall3[xfloor]
        [int(yfloor[k])]/255
    break
```

```
#Geracao das paredes textura 4
if mapa[int(x)%(tamanho-1)][int(y)%(tamanho-1)] == 4:
    h = half_res_vertical - j
    if x%1<0.015 or x%1>0.985:
        xfloor = yfloor
    yfloor = np.linspace(0,299, h*2)%299

    for k in range(h*2):
        frame[i][half_res_vertical - h + k] = wall4[xfloor]
        [int(yfloor[k])]/255
    break
#Chao
else:
    frame[i][half_res_vertical*2-j-1] = floor[xfloor]
    [yfloor]/255
```

Conclusões

O desenvolvimento do presente estudo possibilitou uma análise de como algumas técnicas podem simular situações que são tecnicamente impossíveis de se reproduzir, é evidente que a técnica não tem o mesmo valor prático na atualidade onde as engines que renderizam gráficos tridimensionais em tempo real, porém o raycasting revolucionou os videogames sendo wolfeinstein, um sucesso de crítica e público, graças a suas inovações. Além disso, o projeto premitiu a elaboração de um sistema de raycasting próprio usando na prática o sistema.

Referências

Raycasting Foundations
"https://people.computing.clemson.edu/~dhouse/courses/405/notes/raycast.pdf",
[Online; accessed 07 November 2022] Raycasting Foundations
"https://lodev.org/cgtutor/raycasting.html", [Online; accessed 07 November 2022]
Ray casting
"https://en.wikipedia.org/wiki/Ray_casting", [Online; accessed 07 November 2022]