Relatório Projeto com OpenGL 3D

Allan Groisman Kusbick, Dhruv Babani
Computação Gráfica
Ciência da Computação — PUCRS

27 de novembro de 2023

Video Demonstrativo (Obs: Quando é falado parte “3D”, é sobre os objetos TRI):
https://youtu.be/7YtKji1Kk60

1. Introdução

Este trabalho tem como objetivo a elaboração de um pequeno jogo em ambiente 3D, com a utilização do OpenGL (Open Graphics Library), no qual o usuário controla um tanque que deve atirar contra os objetos no cenário. Para este desenvolvimento foi utilizado como base, um projeto disponibilizado pelo professor que já possui diversas implementações básicas e exemplos de algumas funcionalidades do OpenGL.

O cenário é composto por um piso que tem área de terreno 50m x 25m, um paredão com 15m de altura posicionado ao meio que divide este piso em duas partes iguais e o tanque controlado pelo jogador. O ambiente ainda deveria ter objetos 3D no lado contrário do paredão em relação ao tanque, para que sirvam de “amigos” e “inimigos”, alvos a serem atingidos ou evitados, porém por falta de tempo, ficarão por serem implementados no futuro e não estarão presentes neste trabalho.

2. Piso e Paredão

A modelagem do piso e do paredão se dão de formas parecidas, baseados em quadrados de tamanho 1m x 1m que são desenhados na posição e angulação correta. Há duas funções, a *DesenhaPiso()* que desenha o piso e a *DesenhaParede()* que desenha o paredão, estas duas que utilizam de um terceiro método em comum chamado *DesenhaLadrilho(int id)* que desenha os quadrados 1m x 1m.

A função *DesenhaLadrilho()* recebe como parâmetro a textura respectiva a ser utilizada, tema que será abordado mais adiante neste trabalho. A partir disto ela desenha um quadrado a partir da posição atual que tem como pontos (-0.5, 0, -0.5), (-0.5, 0, 0.5), (0.5, 0, -0.5) e (0.5, 0, 0.5), que acaba com arestas de tamanho 1. Para cada ponto destes há uma associação do respectivo ponto na imagem fornecida como textura, esta que é carregada no *InitTexture()* no início do programa.

O método *DesenhaPiso()*, a partir da origem, translada a cena -0,5 m em relação ao eixo Y para ficar de forma perpendicular ao paredão posteriormente. A partir daí existem dois *for()* aninhados que são respectivos ao comprimento (50m) e largura (25m), estes que partem do 0 e vão até seus tamanhos. Em cada iteração, há o desenho de um quadrado através do *DesenhaLadrilho()* e então o deslocamento no respectivo eixo, montando assim todo o piso (Figura 1).

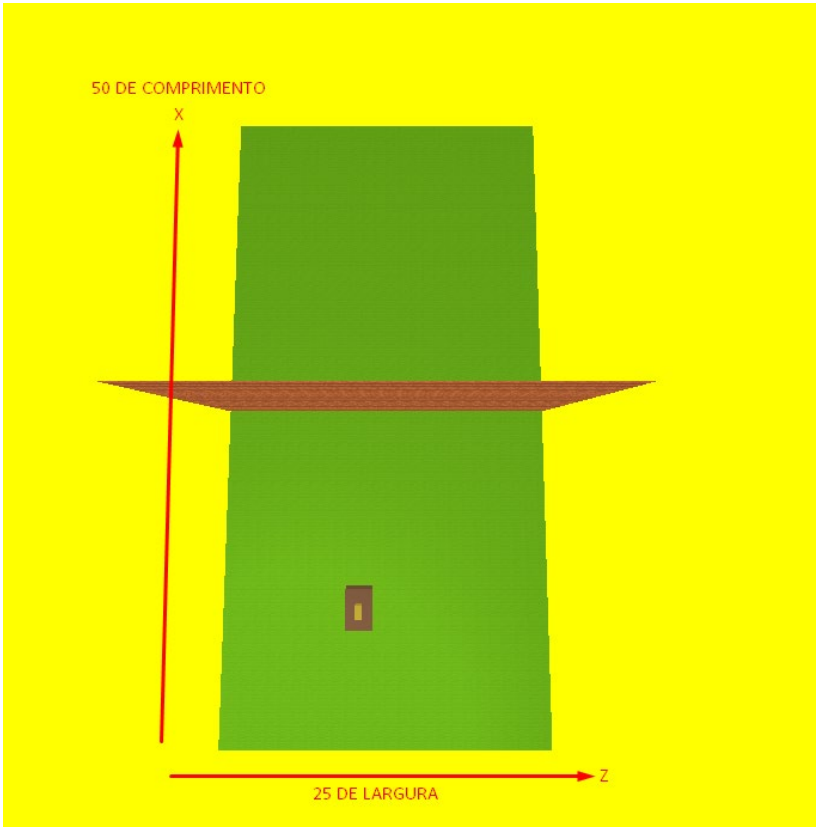

Figura 1: Captura do piso e suas dimensões.

Para montar o paredão, o processo é semelhante na função *DesenhaParede().* Há o deslocamento até o ponto (25, 0, 0) que corresponde ao meio do piso, além de uma rotação em 90 graus no eixo Z, isto que deixa a parede perpendicular.

Nesta função, porém, ao utilizar dos *for()* aninhados, tem integração com uma matriz chamada *matrizParede[][].* Esta matriz é booleana, tem tamanho proporcional ao paredão (25x15) e tem como função guardar quais partes da parede ainda estão "vivas" e não foram destruídas pelo jogador. Caso a posição respectiva no for, esteja *true* o quadrado é desenhado, ao contrário, uma pequena esfera é desenhada que no contexto geral, mostra onde a parede deveria ser desenhada antes de ser atingida.

Figura 2: Captura do paredão e suas dimensões.

3. Modelagem do Tanque

O tanque é separado em três partes, a base, o canhão e o projétil e é desenhado a partir da função *DesenhaCanhao().* Ele tem três variáveis que determinam o posicionamento do desenho e as rotações tanto da base, quanto do canhão. Desta forma, primeiramente o plano é deslocado para a posição atual do veículo através da variável *Ponto posVeiculo* e então rotacionado em relação ao eixo Y ao usar da *int rotacaoVeiculo*.

Com o ambiente posicionado corretamente, o primeiro desenho é da base, esta que é um cubo sólido de tamanho 1, de cor RGB (0.52,0.37,0.26), mas em escala (3,1,2) que resultam em um tanque de 3m de comprimento, 1m de altura e 2m de largura como segue na figura 4.

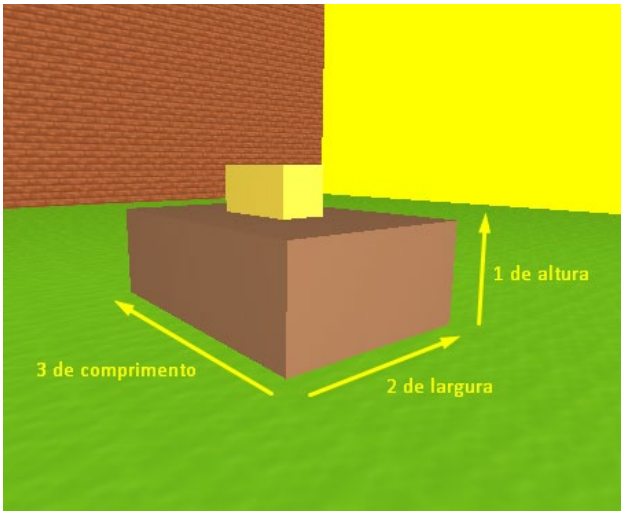

Figura 4: Captura do tanque e suas dimensões.

A partir do posicionamento inicial do tanque, para desenharmos o canhão, há um deslocamento em 0.75m no eixo Y, para posiciona-lo na parte superior e ainda uma rotação a partir do *int rotacaoCanhao* em relação ao eixo Z. O objeto é baseado em um cubo sólido de tamanho 1, mas de escala (1, 0.5, 0.5) e cor RGB (0.81,0.71,0.23).

Por fim, a partir da posição do canhão, há o desenho de uma esfera representativa de onde o projétil está antes de ser lançado, internamente no tanque, mas possível de ser visualizada apenas quando as faces estão desabilitadas nos desenhos.

4. Movimentação do Veículo

O veículo se movimenta a partir das entradas do usuário em estilo correspondente ao tanque, apenas para frente e para trás na direção em que aponta. Esta lógica está no método *movimentarVeiculo()* e tem como base a velocidade do veículo e a direção em que se encontra.

A velocidade é representada pela variável *double velocidadeReal,* esta que recebe uma outra *double velocidade* externa que neste trabalho é de 2,5, e converte-a, dividindo pelo número de frames por segundo do jogo, neste caso 30, obtendo uma velocidade prática de 2,5m/s.

Para adicionar a velocidade ao objeto, usa-se um vetor *Ponto DirecaoVeiculo* que serve, como o próprio nome sugere, para direcionar a velocidade. Ele começa como *Ponto(1,0,0),* direcionado para frente em relação ao eixo X, mas sofre uma rotação em Y, com base na mesma *rotacaoVeiculo* que o alinha com a direção real do veículo.

Com estas duas variáveis, a partir da entrada do usuário há a escolha de andar para frente ou para trás, a partir de duas variáveis *boolean andar* e *boolean voltar.* Caso andar esteja *true*, soma-se

na posição do veículo a direção multiplicada pela velocidade, que seria o deslocamento correspondente. Já se o caso for de voltar, andar para trás, esse deslocamento é subtraído.

5. Canhão e Tiro do Projétil

Atirar com o canhão requer duas ações do usuário: ativar o modo de mira e então atirar o projétil de fato. Para ativar a mira há uma variável *boolean miraViva*, esta que quando ativa, libera o desenho do projétil em sua posição atual, chama a função *Mirar()* e testa colisão contra a parede e contra o próprio tanque, colisões estas que terão tópico próprio adiante.

5.1 Função *Mirar()*

A função *Mirar()* é quem calcula a trajetória do projétil através da posição do tanque, as rotações verticais e horizontais, além da força com a qual o projétil será lançado, esta que pode variar com entradas do jogador.

Primeiramente o ambiente é deslocado para a posição onde se encontra o veículo, então há a criação do *Ponto DirecaoDoCanhao*, um vetor de início (1, 0, 0) em direção ao eixo X, que sofre as rotações do horizontal do veículo em relação ao eixo Y e vertical do canhão em relação ao eixo Z, que o transformam em um vetor de saída do projétil.

São calculados então dois pontos que compõem a trajetória total do projétil, o P*onto B*, ponto de altura máxima e o *Ponto C*, ponto de chegada do projétil. O ponto *B* tem origem na posição original do canhão, mas somado com o deslocamento do vetor multiplicado pela força escolhida.

Já o ponto *C* recebe também a posição do canhão, mas é somado uma distancia diferente calculada a partir da força em conjunto com o ângulo de rotação do canhão, esta que é aplicada apenas em seu eixo X. Para alinhar *C* com o veículo, no fim há a aplicação da rotação em seu eixo Y.

Com todos os pontos calculados, há o desenho de linhas entre os pontos e o canhão (figura 5), de forma que o usuário consegue visualizar toda a trajetória que o projétil irá percorrer. E por fim, caso o jogador atire, a uma variável *boolean tiroVivo* é ativa e chama a função *Atirar(),* esta que recebe como parâmetro a posição do canhão e os dois outros pontos calculados (B e C).

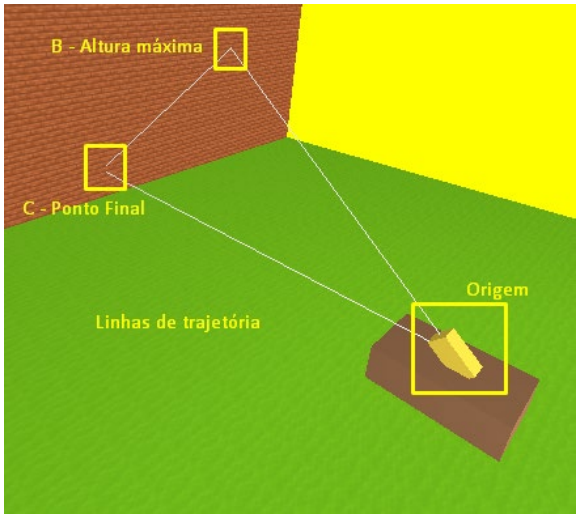

Figura 5: Linhas de trajetória da mira.

5.2 Função *Atirar(Ponto a, Ponto b, Ponto c)*

Este método tem como objetivo fazer com que o projétil percorra o caminho calculado pela função *Mirar(),* este que tem como origem os pontos informados como parâmetro. O ponto a é a

origem do canhão, b o ponto de altura máxima e c o ponto final. Desta forma, o caminho é dividido entre ab e bc, em que o projétil acaba partindo de a, subindo até b e então descendo à c.

Para fazer esta divisão, existe uma variável global *double caminhoProjetil* que guarda a que ponto do caminho o projétil está naquele momento, esta que é atualizada a cada frame com a soma da velocidade do projétil.

Caso este caminho seja menor que 1, é sinal que ainda não terminou o caminho entre a e b, desta forma, o calculo do vetor desta reta é calculado e somado na posição do projétil, porém multiplicado pelo valor atual do caminho, posicionando o projétil corretamente.

Se for o caso de o caminho ser 1 ou maior, é calculado o vetor da reta entre b e c, nos mesmos moldes da anterior, porém subtraindo 1 unidade do caminho projétil na hora de somar o vetor na posição, pois o caminho deve ser calculado apenas entre 0 e 1, mas a variável é a mesma. Por exemplo se o caminho for 1.2, o correspondente seria 0.2 da reta entre b e c.

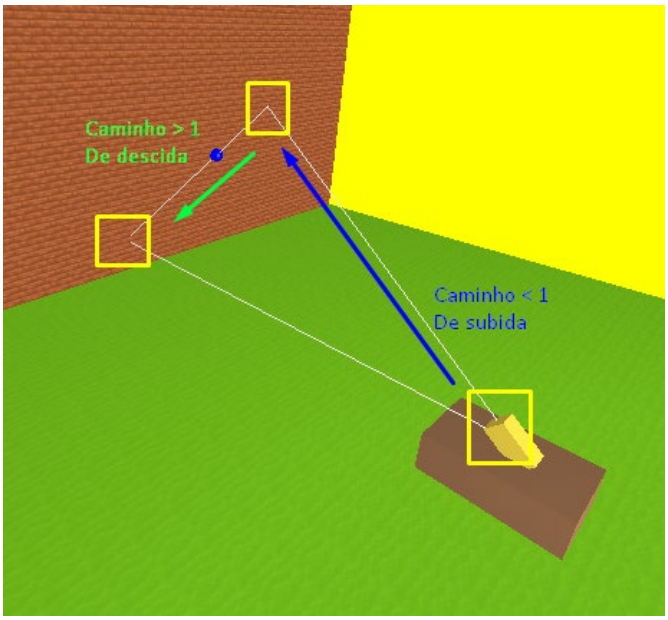

Figura 6: Trajetória do projétil.

Por fim, se o caminho do projétil alcançar 2.2, o desativa tiroVivo e chama a função *Pontuar(),* esta que neste contexto diminui a pontuação do jogador, pois o caminho se deu por completo e não atingiu ninguém em seu percurso.

6. Colisões

Neste trabalho foram implementadas duas ocasiões de colisões para o projétil, a colisão contra o próprio tanque que termina o jogo, além da colisão contra a parede, que soma pontos através de *Pontuar()*.

Para testar se o projétil tem colisão com o tanque, existe o seguinte *if()* (figura tal) dentro da função d*isplay()* que testa se a posição do projétil está dentro dos limites do tanque baseado na posição do veículo. Caso esteja, simplesmente fecha o programa.

Já a função de teste contra a parede se da através da função *TestarColisao()* que é chamada apenas caso a posição do projétil esteja dentro dos limites do paredão, como é possível ver na figura tal. Este método obtém a altura (Y) e a largura (Z) da posição atual do projétil para verificar as mesmas na *matrizParede[][]* utilizada para verificar se deveriam ser desenhadas na função, já demonstrada anteriormente, *desenhaParede().*

Caso a posição de contato entre o projétil e o paredão na matriz esteja falsa, quer dizer que já existe o buraco e o projétil pode continuar sua trajetória, não existe colisão. Porém, se a matriz retornar verdadeiro, não só o índice de contato é destruído, através da atribuição de falso na matriz, mas os 8 índices que formam seu contorno, também são.

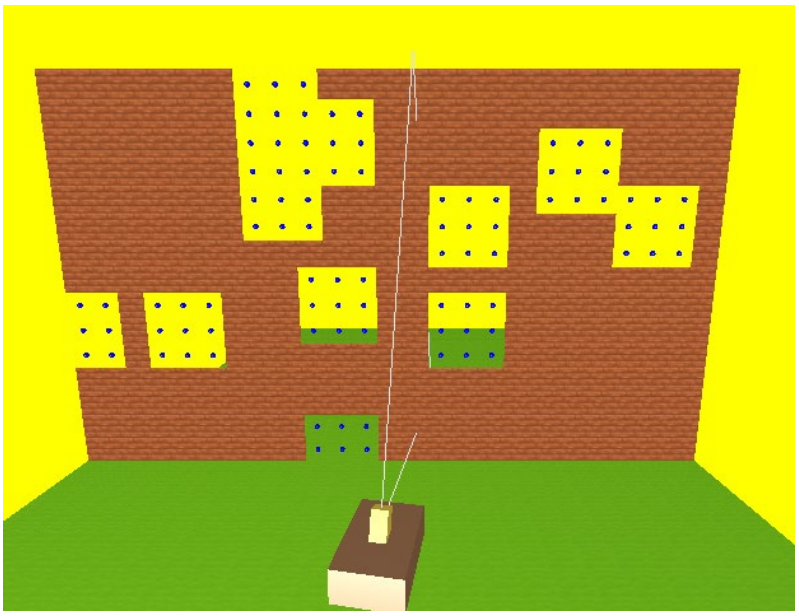

Figura 6: Paredão já com diversas colisões.

Se algum índice a ser destruído estiver fora dos limites, entre 0 e 15 de altura ou 0 e 25 de largura, nada acontece. Por fim, com a colisão confirmada, há a soma de pontuação através de *Pontuar()* e a variável *tiroVivo* se torna falsa, o que destrói o projétil.

7. Outras Funções e Comandos

Existem outros métodos mais simples:

- *display():* atua como centro principal e que chama todas as outras, é nela que se desenha o piso, a parede, o canhão e verifica se a função de mira e do tiro estão ativas para chamar as respectivas funções e desenhar o projétil.
- *Pontuar(id):* que soma ou subtrai de uma pontuação global do jogador, caso ele tenha acertado ou não na parede. Esta pontuação é atualizada no terminal quando alterada e a cada 5 segundos de jogo.
- *keyboard():* esta função que já estava implementada e ela trata das entradas do usuário. Os principais comandos estão na tabela abaixo:

COMANDO	**AÇÃO**
wasd	Movimenta o observador
v, barra de espaço	Desce e sobe o observador
ijkl	Movimenta o alvo
n,m	Desce e sobe o alvo
COMANDOS VEÍCULO NO NUMPAD	

8, 2	Mover para frente e para trás. Caso aperte mais de uma vez, aumenta a velocidade.
4, 6	Girar para esquerda e para direita.
COMANDOS CANHÃO NO NUMPAD	
7,1	Girar canhão para cima e para baixo.
3	Ativa/desativa modo de mira
*	Atira o projétil
+, -	Aumenta e diminui a força de lançamento
COMANDOS DE CÂMERA	
9	Fixar alvo no veículo
!, @, #, $, %, &	Estas teclas, são do 1 ao 5 clicadas com o shift. Elas tem alguns padrões de câmeras pré-definidos.