Instituto Superior Técnico 2021/2022

Computação Gráfica para Jogos

Relatório Técnico

Micromachines

GRUPO 12

Marina Martins
IST1104080

Miguel Mota
IST190964

TÉCNICO LISBOA

Vídeo https://streamable.com/weaq2b

Estrutura do Projeto

O ficheiro *MyObject.cpp* contém todas as definições dos objetos que são posteriormente renderizados no jogo juntamente com algumas classes auxiliares, tais como:

- ***My3DVector*** que representa vetores tridimensionais. Esta classe também tem um campo extra, o ângulo, utilizado para as rotações dos objetos.
- ***MyObject*** que contém a *mesh* do objeto e que contém todas as transformações que lhe são aplicadas quando são renderizados e a função de Render.
- ***MyAssimp*** similar ao *MyObject*, mas em vez de contar uma *mesh*, contém um conjunto de meshes que compõem um objeto Assimp, utilizado para o veículo.
- ***MyFlare*** que é responsável pelo *Lens Flare*, carrega o seu conjunto de texturas e renderiza-o.
- ***MyParticle*** que é responsável pelas partículas, definindo a sua velocidade, aceleração, posição inicial, entre outros e posteriormente renderiza então o objeto partículas.

O ficheiro *lightdemo.cpp* é onde o jogo corre. Este ficheiro é composto por um conjunto de funções que permitem o resultado final:

- ***Main***: Onde o programa é inicializado, as opções da biblioteca *Glut*(versão, *displaymode*, funções de *callback*, entre outros) são inicializadas, é realizado a configuração dos *shaders*, inicializado os objetos do jogo e o *glutMainLoop* para a execução do programa em *loop*.
- ***processKeysPressed*** / ***processKeysReleased***: São funções de *callback* que são chamadas quando uma tecla é pressionada e levantada, respetivamente, verificado a tecla em questão e procedendo a realização de uma funcionalidade.
- ***setupShaders***: Procede com a configuração dos *shaders* que são utilizados. Os *shaders* utilizados, no projeto, são os ficheiros *texture_demo.vert* e *texture_demo.frag*, que representam o *vertex shader* e o *fragment shader* respetivamente.
- ***processMouseButtons*** / ***processMouseMotion*** / ***mouseWheel***: Funções de callback que são chamadas quando os botões do rato são pressionados, quando o rato se move ou quando a roda do rato se move, utilizados para o movimento da câmera.
- ***CarCollisions***: Verifica as colisões do carro com as laranjas, cheerios e as moedas, em que posteriormente a sua funcionalidade é realizada em ***catchCoins***.
- ***Init***: Onde os objetos da cena são criados e onde as texturas são carregadas.
- ***restartGame***: Reinicia todas as variáveis do jogo quando o jogo é reiniciado sem necessidade de interromper o ciclo do programa.

- ***rearviewMirror***: Calcula os valores para o lookAt relacionado com a câmera de rearview.
- ***checkFinish***: Verifica se o veículo realizou uma volta completa, caso seja o caso, o jogo termina, as partículas são ativadas e é apresentado uma mensagem de vitória.
- ***changeSize***: Altera o tamanho do *display* da cena.
- ***changeProjection***: Utilizado para alterar a câmara entre ortogonal e perspetiva.

```
void changeProjection() {
    loadIdentity(PROJECTION);
    if (camType.compare("orthogonal") == 0) ortho(-orthoHeight * ratio,
    orthoHeight * ratio, -orthoHeight, orthoHeight, -0.1f, 1000.0f);
    else perspective(53.13f, ratio, 0.1f, 1000.0f);
}
```

- ***renderObjects***: Renderiza todos os objetos do jogo com a exceção da mesa onde se passa o jogo. A mesa é utilizada como reflexão de forma que se encontra excluída da função.
- ***Lights***: Realiza todos os cálculos relacionado com 3 tipos de luzes e posteriormente envia esses cálculos para os *Shaders*.
- ***Refresh***: Limita o número de *Frames Per Second* a 60.
- ***renderScene:*** Verificar quanto tempo passou desde a última chamada da função. Esta diferença de tempo é usada como *deltaTime* na equação do movimento. Realiza os cálculos relacionados com as 4 câmeras existentes. Envia as variáveis do jogo(ativação do fog e das luzes) aos *shaders*. As texturas são *binded* ao *TextureArray* e são indicados ao *sampler* do *GLSL* que *texture units* é que devem ser usados. É onde o HUD, sombras e luzes são calculadas e renderizadas, incluindo uma chamada da função *renderObjects* e a renderização da *SkyBox*. Caso o flare esteja ativado é renderizado.

Sistema de Partículas

As partículas no projeto foram utilizadas com o intuito de simular confettis quando o jogador ganha, gerando aleatoriamente as cores. Quando o jogador acaba uma volta, as suas coordenadas são usadas para dar render ao objeto.
As partículas possuem um conjunto de atributos que fazem a representação de um objeto como uma nuvem de partículas, tal como: como nascem, se movem, mudam de aparência, e como morrem. Utilizando o método de Euler é possível controlar a velocidade e posição das partículas.

Figura 1 – Sistema de Partículas

Câmeras

Foi implementado 4 tipos de câmeras: uma a simular um espelho retrovisor que segue os movimentos do veículo, um top-view de perspetiva fixa, top-view ortogonal fixa e uma de perspetiva que segue por trás o veículo.

Para definir a câmera correspondente ao pretendido foi utilizado a matriz LookAt que que recebe como valores [Figura 2] o vetor da sua posição no espaço do mundo, o ponto 3D para a direção para a qual ela está a olhar, um vetor apontando para a direita e um vetor Up apontando para cima a partir da câmera.

```
if (camType == "main" || camType
== "rearview") {
// set the camera position based
on its spherical coordinates
      (…)
//update camera position adding
the position of the vehicle
      (…)
      //update lookat
lookAt(x, y, z,
car.body.position.x,
car.body.position.y,
car.body.position.z, xUp, yUp,
zUp); }
```

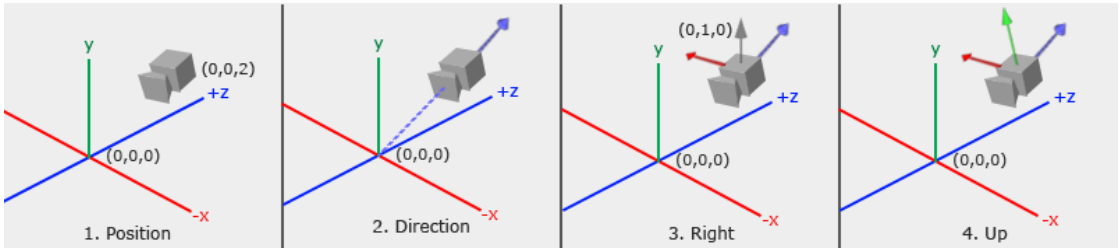

Figura 2 – LookAt coordenadas

HUD e pausa

O *Head Up Display* foi desenvolvido no render *scene* com o objetivo de dar informação ao utilizador sobre as vidas que lhe restam, os pontos que possui e quando se encontra com o jogo pausado [Figura 3]. Para isso é projetado, com as matrizes de *VIEW*, *PROJECTION*, *MODEL*, um conjunto de textos renderizados pela função *RenderText* [`void RenderText(VSShaderLib& shaderText, std::string text, float x, float y, float scale, float cR, float cG, float cB)`], disponibilizado pelos docentes, utilizando uma projeção ortogonal.

Figura 3 – HUD and pause

Sombras Planares

As sombras foram calculadas com a função *shadow_matrix*, fornecida pelos docentes, que recebe um *float** para guardar a informação, o plano do chão onde irão comparecer as sombras e a posição da luz que irá ser utilizado como a fonte de luz para as sombras. Posteriormente foi utilizado o *glBlendFunc* de forma a escurecer a cor dos objetos de forma a simular a sombra. Similar às reflexões é utilizado o *Stencil Buffer* para evitar que as sombras projetadas saiam do plano, o tabuleiro.

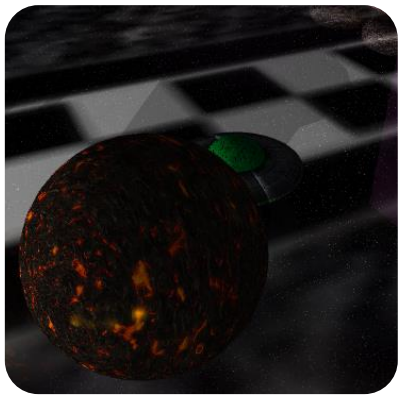

Figura 4 – Sombras planares

Reflexões Planares

Para a realização das reflexões planares foi necessário a utilização *do Stencil Buffer* para fazer *clipping* da zona onde pretendemos as reflexões, no projeto em questão optamos por utilizar o tabuleiro para realizar o *clipping*. Posteriormente todos os objetos das cenas foram renderizados no *stencil buffer* e invertidos. Tornando o tabuleiro translucido foi possível então "iludir" o jogador representando o mesmo objeto como uma reflexão.

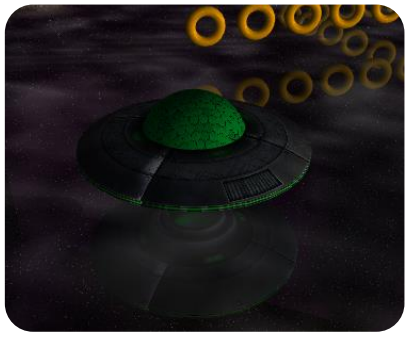

Figura 5 – Reflexões planares

Espelho retrovisor

Semelhante às reflexões é utilizado o *Stencil Buffer* de forma a visualizar numa pequena janela ortogonal o conteúdo que o *lookAt* reproduz. Assim, posicionando a câmera *rearview* na posição do veículo, a apontar para a direção oposta do mesmo, podemos simular um retrovisor.

Figura 6 – Retrovisor

Luzes

Todas as luzes deste projeto são calculadas de acordo com o modelo de iluminação *Blinn-Phong*. As coordenadas das luzes(posição no caso das luzes pontuais e *spotlights*, direção no caso da luz direcional) são mandadas para os *shaders* onde os cálculos do modelo de iluminação são feitos. Para cada fragmento, as contribuições de cada tipo de luz são somadas resultando na cor final de cada fragmento.

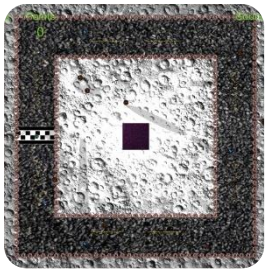

Figura 7 – Luz direcional

Luz Direcional

Uma luz direcional na direção (0,-1,0). Este tipo de luz emite uma luz em toda a cena numa determinada direção.

Seis Luzes Pontuais

Emite luz em todas as direções a partir de 6 posições.

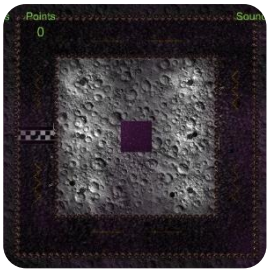

Figura 8 – Luzes Pontuais

Spotlights

Duas *spotlights* que seguem a posição da nave. São apenas luzes pontuais que só emitem luz num conjunto restrito de direções, um cone com um certo ângulo de abertura. A forma como as posições das luzes são calculadas é aplicando as transformações do objeto aos pontos associados às luzes.

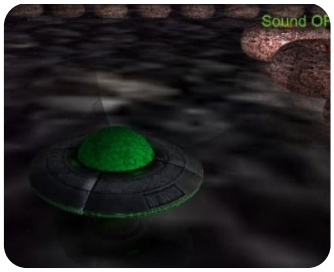

Figura 9 – Spotlights

Colisões

Para a deteção de colisões foram utilizados dois conceitos: *Bounding Sphere* e *Object Oriented Bounding Boxes (OOBB)*.
A técnica ***Bounding Sphere x OOBB*** é usada nas colisões entre o veículo e os meteoritos/laranjas. Para verificar a colisão verificou-se se a distância entre o centro da esfera e a *Bounding Box* do veículo é inferior ao raio do meteorito.

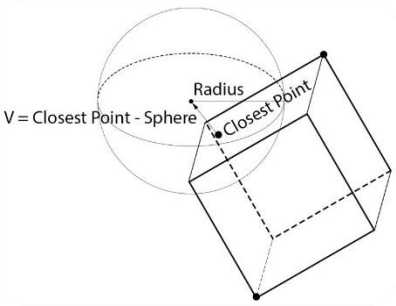

Figura 10 – Bounding Sphere x OOBB

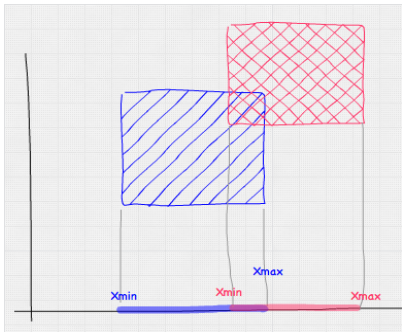

Figura 12 – OOBB x OOBB

A técnica ***OOBB x OOBB*** é usada em todas as outras colisões com o veículo.
Esta técnica utiliza uma Bounding Box para cada objeto que cujas transformações aplicadas ao objeto também são aplicadas. A colisão é verificada se houver uma interseção entre os limites da bounding boxes, isto é, se os limites A_{minX}–A_{maxX} e B_{minX}–B_{maxX} se sobrepõem.

Figura 11 – Objeto Assimp

Suporte OBJ Assimp

Ao contrário dos objetos criados com primitivas geométricas, os objetos Assimp necessitam de ser carregados a partir de ficheiros .obj. Para isto foi necessário passar como argumento o nome do ficheiro que pretendemos utilizar. É uma forma facilitada de renderizar objetos diversificados e com melhor qualidade sem necessidade de criar por nós próprios cada polígono. Existem ainda algumas diferenças no render, visto que os ficheiros .obj contêm várias meshes que necessitam de ser desenhadas

Painel

O uso dos painéis (*billboards*) são uma técnica que ajusta a orientação do objeto fixando a câmera. O painel foi criado como um retângulo, implementando uma textura e removendo o *background* utilizando o *Alpha Blending*, descartando fragmentos com alfas inferiores a 0.5.

```
if (texMode == 4) {
    //TRANSPARENCY
    if (texel.a < 0.5)
        discard;
}
```

Figura 13 - Fragment Alpha

Foi utilizado o *Spherical Billboarding, com esta versão o objeto encontra-se sempre de frente para a câmera. O alvo são os* parâmetros *camX, camY e camZ*, isto é, as coordenadas 3D, para onde o objeto aponta e o vetor 3D da posição do painel é influenciado.

2D lens flare effect worth

Lens flare é um efeito ótico adicionado ao projeto, criado por inter-reflexão entre os elementos de uma lente quando a câmera está apontada para uma das velas, renderizando-os ao longo de uma linha cruzando a posição da luz/vela e o centro da tela.

O tamanho e transparência dos elementos foi influenciado pela distância entre a fonte de luz e o centro. No exemplo da Figura 12 podemos determinar que a os elementos encontram-se mais transparentes e pequenos devido à distância da posição da luz.

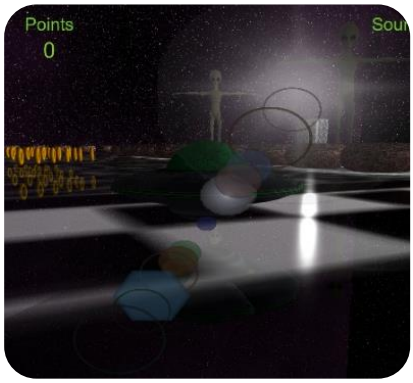

Figura 14 - Lens Flare

Bump-mapping

Bump-mapping é uma técnica/efeito utilizado para simular textura 3D sendo possível manter um objeto de *low-polygons.*

Para realizar o efeito, a normal é codificada com RBG, que posteriormente no *fragment shader* é descodificada da seguinte forma:

```
// lookup normal from normal map, move from [0,1] to [-1, 1] range, normalize
n = normalize(2.0 * texture(candleNormal, DataIn.tex_coord).rgb - 1.0);
```

Figura 15 - Resultado

Normais são utilizadas para a computação de *shading* sendo que ocorrem no espaço tangente, pois cada *texel* guarda um vetor normal no espaço tangente.

Figura 16 - Textura base

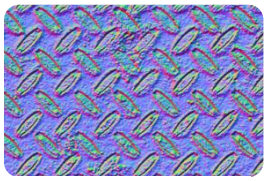

Figura 17 - Normal map