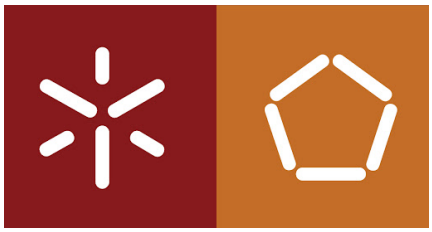

Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica - Fase 4
Grupo 38

Bernardo Emanuel Magalhães Saraiva - A93189
José João Cardoso Gonçalves - A93204
Mariana Rocha Marques - A93198 Rui Filipe Coelho Moreira - A93232

Ano Lectivo 2021/2022

Conteúdo

1 Introdução e Contextualização

1.1 Contextualização

O presente relatório foi elaborado no âmbito da Unidade Curricular de Computação Gráfica, com o objetivo de explicar e detalhar o trabalho desenvolvido na 4ª e última fase do Projeto.

Na sequência das fases anteriores, a presente fase do Projeto propõe a implementação de texturas e iluminação, sendo necessária a implementação e cálculo de normais, assim como o cálculo de coordenadas de textura.

1.2 Objetivos da 4ª fase

Nesta fase final do trabalho prático foi proposto ao alunos uma série de objetivos, pelo que se apresenta em seguida alguns dos trabalhos principais que foram divididos pelos elementos do grupo:

- Atualizar o leque de *tags* XML suportadas, de modo a acrescentar as funcionalidades de iluminação e textura (<lights>, <color>, <texture> e restantes tags correspondentes);
- Suportar utilização de texturas, que utilizar a imagem de textura a utilizar para um modelo, se desejado;
- Suportar a funcionalidade de utilizar e acrescentar iluminação e cores, que define a aparência de cada figura;
- Efetuar o cálculo das normais para cada vértice, assim como o cálculo das coordenadas de textura;

1.3 Alterações efectuadas na 4ª fase

Com vista a alcançar os objetivos propostos para a terceira fase, foram necessárias algumas alterações relativamente à fase anterior, sendo que foram necessárias algumas alterações em cada um dos ficheiros.

Para a implementação das coordenadas de textura e das normais, foram alterados cada um dos ficheiros que correspondem a uma primitiva geométrica, assim como o *generator*, especialmente na função *write3D* e nas funções *pointsGenerator* de cada primitiva.

De modo a suportar as novas *tags* (e as respetivas novas funcionalidades de texturas e iluminação), foi necessário efetuar algumas alterações na forma de leitura, parsing e armazenamento das informações constituintes dos ficheiros XML. Deste modo, o ficheiro *tree* foi alterado para suportar estas funcionalidades. De modo a implementá-las, foi também atualizado o *engine*, para que possa exprimir estas alterações de uma maneira gráfica.

2 Estrutura do código

2.1 Generator (Normais Texturas)

Nas fases anteriores, o *Generator* tinha como função gerar os vértices dos modelos num ficheiro *.3d*. Para esta fase, para permitir a aplicação das texturas, foi necessário alterar o *Generator* para que este gerasse também as coordenadas das normais e de textura de cada ponto. Foi definido que os ficheiros *.3d* teriam na primeira linha o número de vértices **(N)** para que seja possível distinguir que tipo de valores estamos a ler. Deste modo, se um dado modelo tiver N vértices, as primeiras N linhas do ficheiro *.3d* (sem contar a primeira onde tem o valor N) são as coordenadas dos vértices do modelo, as linhas N a 2N são as coordenadas das normais e as linhas 2N a 3N são as coordenadas de textura.

Para evitar a criação de novas estruturas, considerou-se mais eficaz armazenar as coordenadas das normais e de textura na estrutura *Point*. As normais são vetores, mas como apenas se precisa das coordenadas dos vetores, pode-se assumir que são "*Points*", desde que depois se trabalhe com estes valores tendo em conta que são com vetores. Já as coordenadas de textura, como são coordenadas bidimensionais, o campo z da estrutura *Point* tem o valor 0. Posteriormente, quando se estiver a trabalhar com estas coordenadas de textura, ignorar-se-á o campo z.

Para a presente fase, foram também implementados VBO's, sendo agora utilizados mais 2 *buffers*, respetivamente um para as normais e outro para as texturas. Estes permitiram melhorar a performance da ferramenta desenvolvida.

De notar que, apesar das normais que se calculou estarem normalizadas, ao usá-las em modelos com escalas diferentes, as normais irão ser alteradas. O *OpenGL* suporta efetuar esse processo de duas formas. A primeira seria ativar o modo GL_NORMALIZE que tratará de normalizar todas as normais. Visto que já se forneceu as normais normalizadas, não faz sentido usar esta opção, visto que seria um desperdício de recursos voltar a normalizá-las. Por este motivo, optou-se por ativar o modo GL_RESCALE_NORMAL do *OpenGL* que apenas irá tentar normalizar as normais se a matriz *modelview* tiver valores de escala não unitários.

Em seguida, passar-se-á a explicar como foram efetuados os cálculos das normais e das texturas para cada tipo de figura.

2.1.1 Plane

Coordenadas do vetor normal

As normais de um plano são muito simples de obter, uma vez que qualquer plano gerado está sempre no plano x0z e por isso as normais são iguais para todos os pontos do plano. **(0,1,0)**

Coordenadas de textura

Quanto às coordenadas de textura, como se tem o plano dividido numa espécie de uma grelha, é relativamente simples calcular as coordenadas de textura.

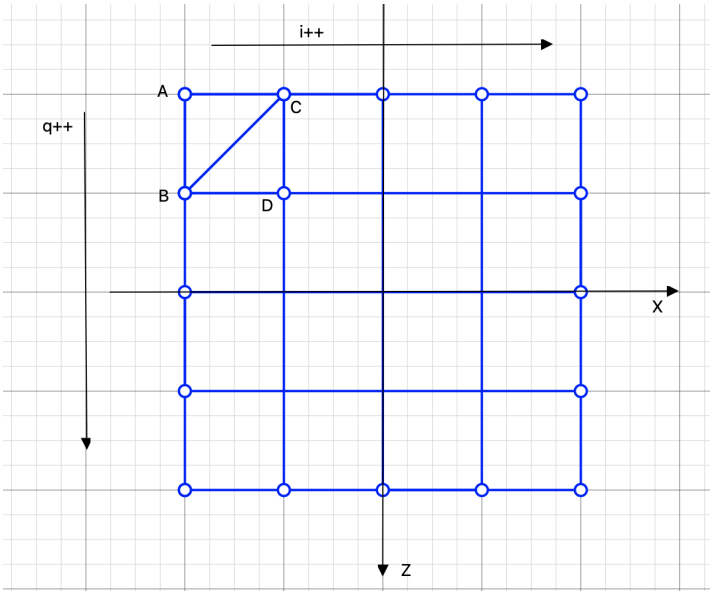

Figura 1: Plano no género de grelha

Deste modo, cada vértice *(i,j)* corresponde à coordenada de textura *(i/divisions,q/divisions)*. Assim, as coordenadas são as seguintes:

Triângulo superior:

- Ponto A $(i/divisions, q/divisions, 0)$
- Ponto B $(i/divisions, (q+1)/divisions, 0)$
- Ponto C $((i+1)/divisions, q/divisions, 0)$

Triângulo inferior:

- Ponto B $(i/divisions, (q+1)/divisions, 0)$
- Ponto D $((i+1)/divisions, (q+1)/divisions, 0)$
- Ponto C $((i+1)/divisions, q/divisions, 0)$

2.1.2 Box

Coordenadas do vetor normal

A *box* é muito idêntica ao plano, sendo que uma *box* são 6 planos, o que envolve um processo semelhante. Para a mesma face da *box*, as normais de todos vértices são iguais, sendo elas:

- Face de Frente : (0,0,1)
- Face da Esquerda : (-1,0,0)
- Face da Direita : (1,0,0)
- Face de Trás : (0,0,-1)
- Face de Cima : (0,1,0)
- Face de Baixo : (0,-1,0)

Coordenadas de textura

Já as coordenadas de textura serão replicadas pelas faces da *box* ficando assim muito idêntica ao *plane*. Deste modo cada vértice (i,j) corresponde à coordenada de textura (i/-grid,j/grid).

2.1.3 Sphere

Coordenadas do vetor normal

No caso da esfera, as normais são implementadas de uma maneira simples, visto que o vetor normal de cada ponto se inicia no centro da esfera até ao respetivo ponto ao qual queremos calcular. Deste modo, as coordenadas do próprio ponto são as coordenadas do vetor normal, resta apenas normalizar este vetor.

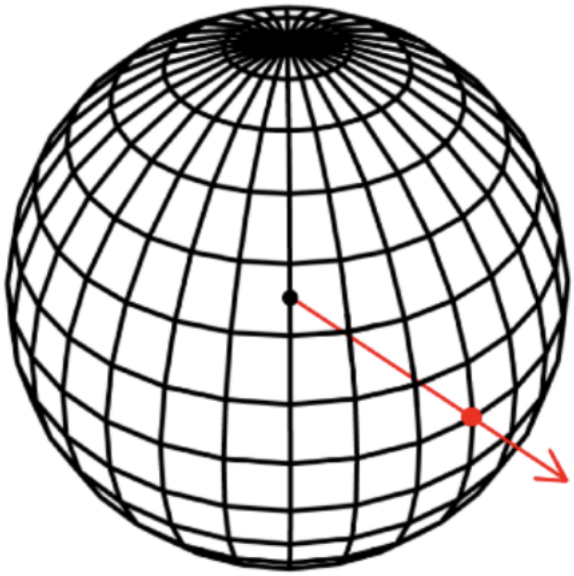

Figura 2: Calculo da normal de um ponto na esfera

Coordenadas de textura

Relativamente às coordenadas de textura, a coordenada x é o valor do *slice* ao qual pertence a dividir pelo número total de *slices*. Da mesma forma a coordenada y em que corresponde ao valor da *stack* a dividir pelo número total de *stacks*. Uma vez que ao desenhar a esfera se desenha dois triângulos da parte superior e dois triângulos da parte inferior, na parte superior efectua-se a soma (0.5 + j/stacks) e na parte inferior calcula-se (0.5 - j/stacks)

2.1.4 Cone

Coordenadas do vetor normal

No cone, calcular as normais dos vértices da base é muito simples, sabendo que apontam sempre para baixo (0,-1,0). O mesmo não se pode dizer dos outros vértices. Para estes, optou-se por formar um triângulo retângulo cujos vértices são: o ponto ao qual se quer calcular a normal (ponto A), o ponto do topo do cone (ponto P) e o ponto no eixo vertical do cone perpendicular ao ponto A (ponto P'). Feito isto, calcular a normal torna-se simples de obter. A normal é simplesmente A - P'.

Coordenadas de textura

Quanto às textura, as coordenadas da base são (0,0) no centro e (i/slices,1) na borda em que i é o valor da *slice*. Quanto às coordenadas laterais o cálculo das normais é idêntico à esfera.

2.1.5 Bezier

Coordenadas do vetor normal

Para calcular as coordenadas normais de um dado ponto de uma superfície de *Bezier*, começou-se por calcular as derivadas parciais dos pontos, em ordem a u e em ordem a v. Para tal, simplesmente recorreu-se ao cálculo por matrizes, derivando a matriz do U e a matriz do V respetivamente.

$\frac{\partial B(i,j)}{\partial u}$ = [3u², 2u, 1, 0] M P $M^T V^T$

$\frac{\partial B(i,j)}{\partial v}$ = U M P $M^T [3v^2, 2v, 1, 0]$

Assim, a normal será o produto externo destes vetores tangentes ao ponto.

Coordenadas de textura

Relativamente ao cálculo das coordenadas de textura, usa-se o mesmo método que se geram os pontos do modelo:

```
textures.push_back(Point(u + (1.0f/tessellation), v + (1.0f/tessellation),0));
textures.push_back(Point(u, v + (1.0f/tessellation),0));
textures.push_back(Point(u,v,0));
```

Figura 3: Pontos de textura da superfície de *Bezier*

2.2 Luzes

Com vista a permitir as funções de iluminação no sistema, foi necessário implementar a leitura, tratamento e aplicação de novas *tags*, que serão referidas em seguida, junto de exemplos e a explicação das funcionalidades que pretendem implementar.

2.2.1 Iluminação

Com a implementação desta nova tag, o programa desenvolvido permite associar determinadas opções de iluminação ao cenário, através de uma tag <Light> dentro da tag <Lights>, como se exemplifica em seguida:

```
<lights>
    <light type="point" posX="0" posY="10" posZ="0" />
    <light type="directional" dirX="1" dirY="1" dirZ="1"/>
    <light type="spotlight" posX="0" posY="10" posZ="0"
                            dirX="1" dirY="1" dirZ="1"
                            cutoff="45" />
</lights>
```

Como é possível verificar, foi implementada a iluminação *pontual*, *direcional* e *spotlight*, sendo que cada uma necessita de argumentos distintos.

Para proceder a esta leitura, criou-se uma classe Light no ficheiro *tree.cpp*, que armazena toda esta informação, de modo a poder ser usada mais tarde no *engine*, sendo guardado também o tipo de luz correspondente, para que se saiba qual o modo de iluminação.

Paralelamente, foi criada uma estrutura Lights, que é formada por um vetor de Light, uma vez que esta poderá englobar zero ou várias instâncias de Light.

Toda esta informação é obtida a partir da função *lightParsing()*, que localiza e trata a informação relativa a estas *tags*, guardando na respetiva estrutura para que possa ser utilizado mais tarde no *engine*.

Agora no engine, após se verificar se há a presença de luz, procede-se à ativação de cada luz para a propriedade expectável, recorrendo à função glLightfv, com a luz, tipo e cor correspondente. Este processo é iterado num ciclo, para que se efetue a iluminação para todas as luzes presentes.

Em seguida, passando para a função de *renderScene*, para cada uma das luzes iteradas, obtém-se a luz e, conforme seja do tipo *point*, *directional* ou *spotlight*, esta é desenhada.

2.2.2 Color

Tal como estudado, a luz tem várias componentes na sua função de iluminação, pelo que a presente fase do trabalho contempla esses tipos de iluminação e permite trabalhar com os mesmos. Para tal, recorreu-se à tag <Color>, que contém informação relativa às componentes da iluminação, nomeadamente a iluminação difusa, especular, emissiva, ambiente e o fator de *shininess*, cada uma contendo os seus valores de RGB na tag correspondente.

Para proceder a esta funcionalidade, criou-se a classe *ModLight* no ficheiro *tree*, no qual se recorreu a *arrays* de 4 elementos, contendo cada uma das componentes R, G e B da cor, assim como o indicador 0 ou 1 referente a um ponto ou um vetor, que é comummente usado em Computação Gráfica. Toda esta informação é obtida a partir das alterações efetuadas na função *modelsParser*, que guarda num ModColor (classe referida anteriormente) juntamente com o conjunto figuras (*Figure*) pertencentes à estrutura *Models*.

Para proceder à aplicação das cores, é na função *drawTriangle* que tudo é feito, uma vez que esta recebe a *Figure* (classe que detém as informações de uma figura), sendo usada as função de OpenGL *glMaterialfv* que conferem a cor ao objeto consoante o seu tipo, seja este especular, ambiente ou difusa, juntamente com o valor de *shininess*.

2.3 Extras

De modo a contribuir para uma melhor aparência do sistema solar desenvolvido, assim como tornar mais realista, decidiu-se implementar uma SkyBox. Para implementar esta funcionalidade, criou-se o ficheiro *boxUpsideDown*, que desenha uma box com os triângulos invertidos (relativamente à regra da mão direita), para que, de uma perspetiva interior (vendo o cubo pelo interior), essa seja definida como a face visível.

Posteriormente, acrescentou-se esta funcionalidade no código XML do sistema solar, e aplicou-se uma textura, demonstrando-se em seguida uma imagem do resultado.

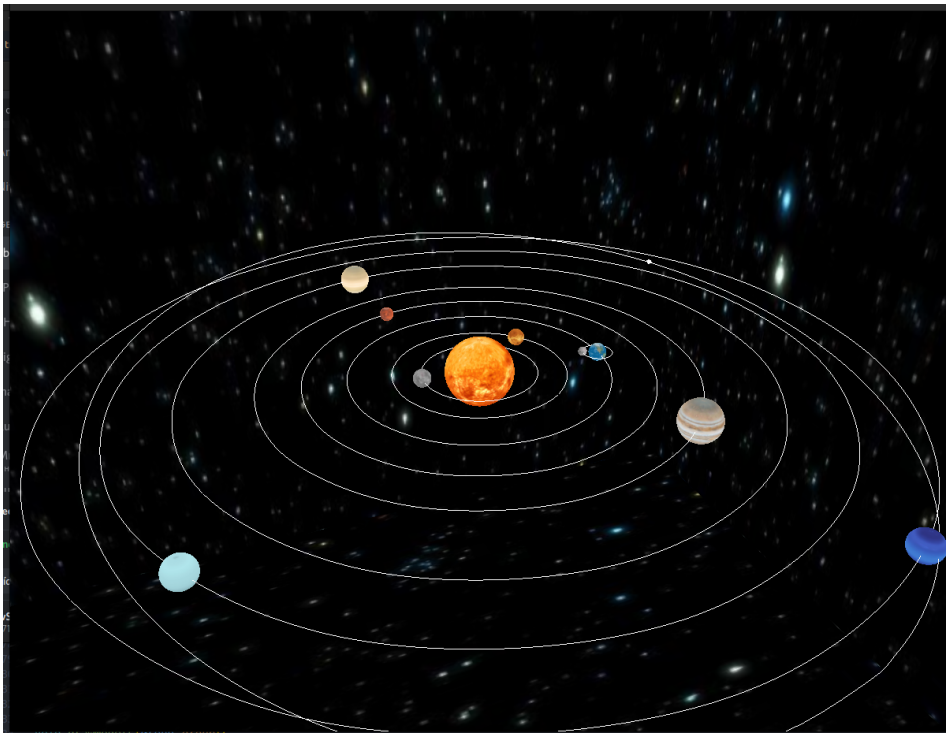

Figura 4: Sistema Solar desenvolvido com recurso a skybox e texturas aplicadas

3 Conclusão

Com o desenvolvimento do Projeto Prático, estando agora completo, considera-se que o mesmo foi extremamente enriquecedor, uma vez que forneceu uma perpetiva diferente das aulas teóricas, no sentido de ser um contexto prático e que permite analisar e verificar o que cada detalhe influencia e altera na concepção desejada do objetivo, para além de complementar os conteúdos lecionados nas aulas práticas.

Assim, apesar das principais dificuldades se terem prendido com o manuseio do OpenGL e as suas funções, considera-se que o trabalho desenvolvido cumpre os objetivos propostos, superando os mesmos através da adição de alguns extras, tal como referido.

Em suma, com a conclusão desta fase, foi possível definir e colocar as fontes de iluminação no sistema solar previamente modelado e aplicar texturas no mesmo, constituindo assim uma grande evolução e permitindo aumentar a complexidade do mesmo.