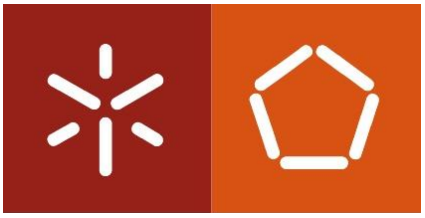

Universidade do Minho

Departamento de Informática

Computação Gráfica
Grupo 45
Trabalho Prático
Fase 4

17 de fevereiro de 2022

Ariana Lousada (A87998)

Rui Armada (A90468)

Carolina Vila Chã (A89495)

Sofia Santos (A89615)

Conteúdo

Capítulo 1

Introdução

Um dos principais objetivos desta quarta e última fase de projeto é aplicar luzes, sombras e texturas ao modelo do sistema solar desenvolvido ao longo das últimas fases do trabalho.

Ao longo deste documento vão ser expostos os vários processos desenvolvidos e aplicados até à conclusão do projeto.

Capítulo 2

Desenvolvimento

2.1 Primitivas (Normais & Texturas)

Para cada primitiva que foi implementada na primeira fase, foram adicionadas as **normais** de cada vértice gerado e também as coordenadas das **texturas**. Isto não implicou nenhuma alteração à sequência de vértices/índices gerada anteriormente. As alterações efetuadas são as que se apresentam de seguida.

Estes novos vetores/coordenadas são adicionados aos ficheiros dos modelos respetivos. O valor da primeira linha de cada ficheiro, o número de vértices, permite-nos saber que tipo de valores é que estamos a ler. Por outras palavras, se um dado modelo tiver N vértices, as primeiras N linhas do ficheiro (sem contar com a primeira) conterão as coordenadas do modelo, as linhas N a 2N conterão as normais e as linhas 2N a 3N as coordenadas de textura. Se o ficheiro terminar após N linhas, assumimos que o modelo não tem normais, e apenas carregamos as coordenadas normais. Acontece o mesmo se apenas tiver 2N linhas, não carregamos coordenadas de textura no *engine*.

De modo a evitar ter que criar novas estruturas de dados, tanto as normais como as coordenadas de textura são armazenadas na estrutura Point no nosso *engine*. Tecnicamente as normais são vetores, mas como apenas precisamos das coordenadas dos vetores, podemos "assumir"que são pontos, desde que tenhamos em conta depois ao ir buscar os valores que estamos de facto a manusear vetores. Nas coordenadas de textura não temos essa questão, mas como as coordenadas de textura são bidimensionais, ignoramos o campo `z` da estrutura Point.

Ao usar VBOs, estes vetores/coordenadas são carregados para buffers, da mesma forma que as coordenadas normais. A única diferença notável é que agora temos 3 conjuntos de buffers, por isso teremos que triplicar as operações que já efetuávamos para os buffers das coordenadas, mas de resto o funcionamento do *engine* não se altera.

Outro fator a notar é que, apesar das normais que calculamos estarem normalizadas, ao usá-las em modelos com escalas diferentes da escala "padrão", as escalas irão alterar as normais, sendo por isso preciso voltar a normalizá-las. O OpenGL permite-nos fazer isso de duas for-

mas. A primeira é ativar o modo `GL_NORMALIZE`, que tratará de normalizar todas as normais. Porém, as normais que fornecemos ao *engine* já estão normalizadas, por isso voltar a normalizá-las é um desperdício de recursos. Assim, o que fizemos foi ativar outra opção do OpenGL, `GL_RESCALE_NORMAL`, que irá apenas tentar normalizar as normais se a matriz *modelview* tiver valores de escala não unitários, poupando-nos assim vários cálculos desnecessários.

Iremos agora explicar como é que fizemos os cálculos das normais e coordenadas de textura para cada modelo.

2.1.1 Plano

As **normais** são obtidas de uma forma muito simples, uma vez que são iguais para os 4 vértices que constituem o plano. Serão elas, então, (0 , 1 , 0). Relativamente às coordenadas das **texturas**, para cada extremidade do plano, corresponde uma extremidade da textura a ser aplicada. Então, as coordenadas serão as seguintes:

- Point t1 = Point(0 , 0 , 0);
- Point t2 = Point(1 , 0 , 0);
- Point t3 = Point(0 , 1 , 0);
- Point t4 = Point(1 , 0 , 0);
- Point t5 = Point(1 , 1 , 0);
- Point t6 = Point(0 , 1 , 0);

2.1.2 Caixa

Relativamente às **normais** da caixa, para cada face as normais de todos os vértices são iguais, sendo elas as seguintes:

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

Quanto às texturas, estas serão replicadas igualmente por todas as faces da caixa, sendo que para cada face, a cada vértice (i , j) irá corresponder a coordenada de textura (i / n_divisões , j / n_divisões).

2.1.3 Esfera

Na esfera, as **normais** a cada vértice são facilmente obtidas através da normalização das componentes x, y e z de cada vértice. Quanto às coordenadas de **textura**, a coordenada x é simplesmente o valor da *slice* à qual o ponto pertence a dividir pelo total de *slices*. O mesmo se aplica à coordenada y, sendo esta o valor da *stack* a dividir pelo número de *stacks*.

2.1.4 Torus

Para o torus, usámos um raciocínio semelhante ao da esfera, visto que também usamos ângulos para construir um torus. Para cada vértice, assumimos que esse ponto pertence a uma esfera de raio unitário e calculamos o vetor normal como se fosse o vetor normal de um vértice de uma esfera, usando os ângulos que usámos para calcular as coordenadas dos vértices do torus. Para as texturas o método de cálculo é exatamente igual ao da esfera.

2.1.5 Cone

Calcular as normais da base do cone é muito simples, apontam sempre para baixo. A parte mais complicada é calcular as normais dos outros vértices. O método que arranjámos para o fazer consiste em formar um triângulo retângulo cujos vértices são o ponto cuja normal pretendemos obter (ponto I), o topo do cone (ponto P) e o ponto no eixo vertical do cone perpendicular ao ponto I (ponto A). A partir destes pontos, a normal é simplesmente I - A. Para obter A precisamos primeiro de saber a distância entre I e P, depois calcular a distância entre P e A através da lei dos senos e por fim obter a posição de A a partir desta distância.

Para as texturas, as coordenadas da base são $(0, 0)$ no centro e $(i, 1)$ na borda, em que i é o valor da *slice* do ponto. As coordenadas laterais são iguais às da esfera e do torus, em t ermos de cálculos.

2.1.6 Bézier Patch

Infelizmente não conseguimos implementar as texturas e normais sobre a superfície de Bézier a tempo da entrega. No entanto, indicamos o raciocínio que iríamos utilizar para fazê-las a seguir. Para calcular as **normais** de um dado ponto (i , j) de uma superfície de *Bézier*, recorreríamos às seguintes equações:

$$\frac{\partial B(i,j)}{\partial i} = [3i^2, 2i, 1, 0].M.P.M^T.V^T \tag{2.1}$$

$$\frac{\partial B(i,j)}{\partial j} = U.M.P.M^T.[3j^2, 2j, 1, 0]^T \tag{2.2}$$

Obtendo as derivadas parciais, o vetor normalizado a qualquer ponto na superfície será um

resultado normalizado do produto dos vetores tangentes ao ponto. Relativamente às **texturas**, a um ponto (por exemplo (i , j)) corresponde a textura (i , j), uma vez que i e j vão variar entre 0 e 1, tal como a textura.

2.2 Iluminação

O primeiro passo na implementação da iluminação é a colocação de luzes na cena. Para tal, se existir um elemento no ficheiro XML a definir as luzes na cena, ativamos a iluminação no OpenGL. Depois, para cada luz definida no ficheiro, criamos uma estrutura com o teu tipo (ponto, direcional ou *spot*) e respetivos componentes, e acrescentamos essa estrutura a um vetor, que após a leitura do ficheiro XML conterá todas as luzes presentes na nossa cena.

Em vez de ler os valores das componentes ambiente, difusa e especular de cada luz do ficheiro, como tal não é um requisito obrigatório para o trabalho prático, decidimos atribuir-lhes valores padrão, como é possível ver abaixo:

```
1 GLfloat dark[4] = {0.3,0.3,0.3,1.0};
2 GLfloat white[4] = {1.0,1.0,1.0,1.0};
3
4 glLightfv(CLight, GL_AMBIENT, dark);
5 glLightfv(CLight, GL_DIFFUSE, white);
6 glLightfv(CLight, GL_SPECULAR, white);
7
```

Neste pequeno pedaço de código, a variável `CLight` representa a luz que estamos a configurar num dado momento (por exemplo, `GL_LIGHT0`).

O segundo passo é, na função `renderScene`, antes de renderizar os modelos, colocar as luzes na nossa cena, tendo em conta a sua posição/direção/ângulo, tal como foi definido no ficheiro XML.

A outra componente da iluminação diz respeito aos materiais dos modelos. Para implementar esta funcionalidade, primeiro tentamos ler do ficheiro XML possíveis atributos dos materiais, como a sua componente difusa, ambiente, especular ou emissiva. Estes atributos são armazenados junto com os outros atributos de cada grupo da cena, na estrutura `group` que definimos nas fases anteriores, dentro de um vetor, onde a posição dos atributos diz respeito ao modelo com a mesma posição no vetor dos modelos.

Depois de obter estes valores, tal como para a luz, apenas temos de os aplicar aos modelos na função `renderScene`, imediatamente antes de os desenharmos.

2.3 Texturas

Para aplicar texturas aos nossos modelos, primeiro devemos especificar qual o ficheiro da textura que queremos aplicar a cada modelo. Podemos fazer isso no ficheiro XML, que será depois lido pelo *parser* do nosso *engine* e colocar o nome de cada ficheiro na estrutura que

guarda os outros atributos de cada modelo, que referimos na secção anterior.

Depois de ter lido o ficheiro XML, percorremos os atributos de cada modelo e, caso tenha um ficheiro de textura associado, carregamos o ficheiro para a memória, para além dos seus *mipmaps*, armazenando o apontador para cada textura num vetor, como temos feito.

Por fim, se as coordenadas de textura estiverem bem definidas, apenas nos resta fazer *"bind"* de cada textura ao desenhar o modelo respetivo na função `renderScene`.

Capítulo 3

Extras

Por defeito, o *engine* corre em modo VBO, mas é possível desativá-lo premindo o botão direito do rato durante a execução do *engine* e selecionando a opção adequada. Já tínhamos implementado esta funcionalidade na 3ª fase do trabalho prático, mas como esta fase apenas nos pedia que implementássemos iluminação e texturas em modo VBO, decidimos implementar também estas funcionalidades no modo "normal". Desta forma, ao desativar os VBOs, temos uma cena exatamente igual, mas com uma pior *performance*.

Usando as funcionalidades já presentes no *engine*, é possível ainda criar uma *skybox*. Para isso, apenas precisamos de desenhar uma esfera invertida (isto é, com escala negativa). Deste modo, a textura da esfera irá ficar do lado de dentro. Apenas temos de nos certificar que a esfera é grande o suficiente para englobar toda a cena.

Na segunda fase do trabalho prático implementámos a funcionalidade de definir cores para os modelos. Apesar do sistema solar não precisar dessa funcionalidade, visto que todos os astros têm a sua textura, para a outra cena que fizemos, o Olaf, as cores são perdidas ao ativar a iluminação. Para evitar ter que definir as componentes de todos os materiais para manter as suas cores, algo que seria tedioso e desnecessário, ativámos a opção `GL_COLOR_MATERIAL` do OpenGL, que calcula automaticamente as cores dos materiais com iluminação ativa, com base na cor dada aos vértices. Podemos assim reutilizar o ficheiro XML que definimos para o Olaf e manter as suas cores, ao mesmo tempo que colocamos luzes na cena.

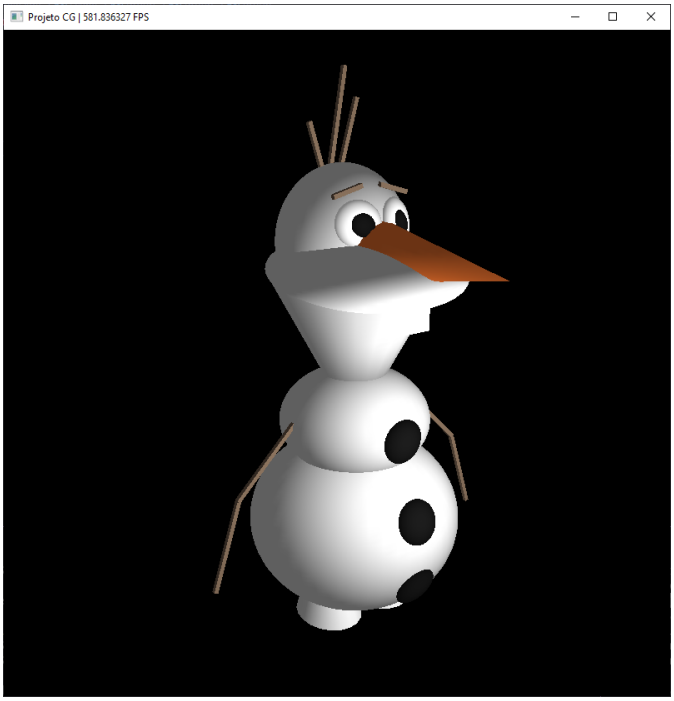

Figura 3.1: Olaf com iluminação e cores.

Capítulo 4

Sistema Solar

Implementando todas as funcionalidades obrigatórias e os extras desenvolvidos por nós, obtemos uma cena do sistema solar como a da imagem abaixo.

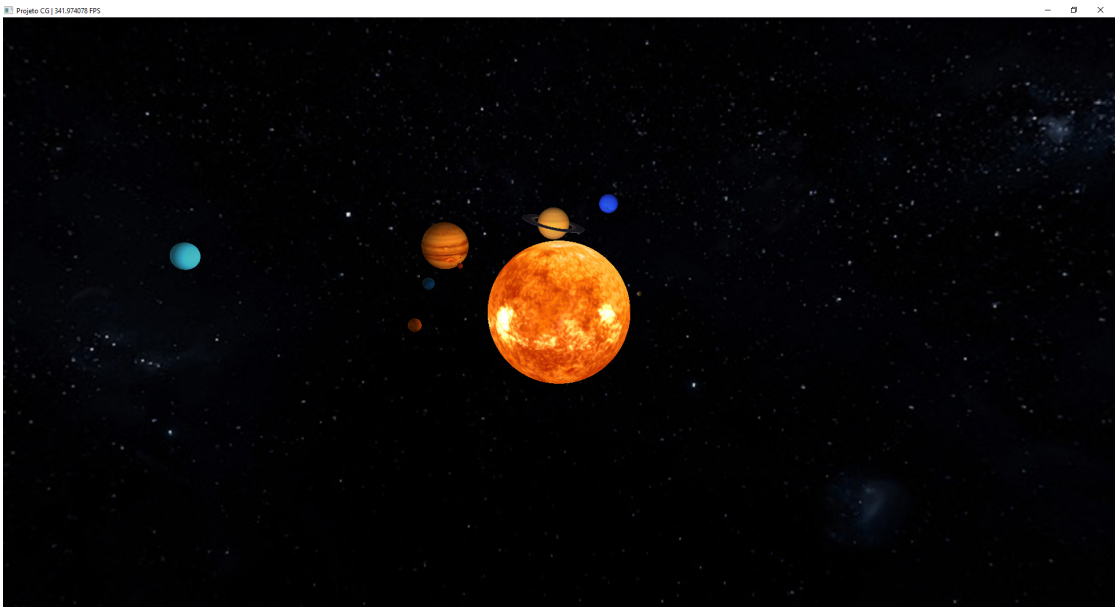

Figura 4.1: Sistema Solar

Capítulo 5

Conclusão

A realização desta fase do projeto permitiu-nos consolidar os conhecimentos adquiridos durante as aulas acerca de texturas e iluminação. Com isto, fomos capazes de não só definir e colocar fontes de iluminação no sistema solar previamente modelado, como também aplicar texturas aos planetas e outros corpos celestes.