Universidade do Minho
Escola de Ciências

Computação Gráfica (3ª ano de LCC)

Trabalho Prático

4ª Fase

Relatório

Grupo 15

Pedro Manuel Pereira dos Santos	(A100110)
João Manuel Franqueira da Silva	(A91638)
David Alberto Agra	(A95726)
João Pedro da Silva Faria	(A100062)

26 de maio de 2024

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

No âmbito da unidade curricular de Computação Gráfica da Licenciatura em Ciências da Computação, foi proposto o desenvolvimento de duas aplicações utilizando a linguagem C++, recorrendo à ferramenta *OpenGl*.
Nesta quarta e última fase do projeto, foram-nos pedidas mudanças em ambas as aplicações. No generator, para além do que já foi desenvolvido nas fases anteriores, temos que ser capazes de calcular as coordenadas de textura e as normais de cada vértice.
No caso da engine, teremos que deixar a possibilidade de uso de luz e texturas, como ler e aplicar as normais das coordenadas de textura dos ficheiros modelo.
Para demonstrar a implementação dessas novas funcionalidades, e conforme também nos foi solicitado, modificamos o arquivo XML que representa um modelo dinâmico representativo do nosso Sistema Solar.
Será fornecida uma análise detalhada das decisões e abordagens adotadas para viabilizar a implementação dessas aplicações propostas neste relatório.

Capítulo 2

Generator

Como já mencionado, a aplicação Generator terá que ser capaz de, para as suas primitivas, escrever para o ficheiro não apenas os valores que definem os vértices, mas também as coordenadas da normal, bem como as coordenadas de textura para cada vértice, de forma a introduzir a iluminação aos nossos modelos, e para mapear imagens sobre as superfícies desenhadas pela aplicação engine.
Exploraremos métodos para calcular automaticamente essas características, aprimorando a qualidade visual e a precisão dos objetos gerados.

2.1 Plano

2.1.1 Normais

Como o plano está nos eixos XZ, as normais de cada vértice têm sempre as coordenadas (0,1,0).

2.1.2 Coordenadas de Textura

A coordenada de textura de um ponto (x, y, z) pertencente ao plano terá os valores $(x/slices, z/slices)$.

2.2 Caixa

2.2.1 Normais

No caso da caixa, por ser constituída por 6 planos, consoante o plano cujos vértices estão a ser calculados, as coordenadas das normais variam entre (1, 0, 0), (-1, 0, 0), (0, 1, 0), (0, -1, 0), (0, 0, 1), (0, 0, -1).

2.2.2 Coordenadas de Textura

Para obter as coordenadas de textura de um dado ponto, assim como no plano, basta dividir o valor das coordenadas x e z do ponto pelo número de slices que definem o cubo.

2.3 Esfera

2.3.1 Normais

Iterando pelos slices e pelos stacks que definem a esfera, a coordenada da normal de um certo ponto P seria então (x, y, z), com $x = \cos(\pi - \beta) * \sin(\alpha)$, $y = \sin(\pi - \beta)$, $z = \cos(\pi - \beta) * \cos(\alpha)$, sendo β e α ângulos

que variam entre 90º e -90º, e 0º e 360º.

2.3.2 Coordenadas de Textura

Iterando sobre os slices e as stacks que definem a esfera, com variáveis i e j, as coordenadas de textura de um dado ponto serão $(i/slices, j/stacks)$.

2.4 Cone

2.4.1 Normais

No caso do cone, primeiro começamos por desenhar os vértices que constituem a sua base, tendo estes todos uma normal de coordenadas $(0, -1, 0)$.
Para o corpo, ao iterarmos com dois ciclos for sobre o número de stacks e slices que caracterizam a figura, dado um certo ponto P, este terá uma normal com coordenadas $(\sin(\alpha * j),\ \sin(atan(radius, height)),\ \cos(\alpha * j))$, sendo j a variável que itera as slices do cone, e $\alpha = 2 * \pi / slices$.

2.4.2 Coordenadas de Textura

Começando pela base, temos que o ponto central tem coordenadas de textura $(0.5, 0.5)$. Para os restantes vértices que constituem a base, apenas precisamos de somar o valor dos senos e cossenos dos ângulos α e β para obter a coordenada de textura, $(0.5 + \sin(\alpha), 0.5 + \cos(\beta))$.
Para o corpo do cone, as coordenadas de textura são obtidas com $(j/slices, i/stacks)$ sendo j e i as variáveis que iteram sobre os slices e stacks, respectivamente.

2.5 Torus

2.5.1 Normais

Para calcularmos os pontos de um toro, iteramos também sobre as slices e as suas stacks. Dado um ponto P, a normal deste terá coordenadas (x, y, z) em que $x = r * \cos(\alpha * i) * \cos(\beta * j)$, $y = r * \sin(\alpha * i)$, $z = r * \cos(\alpha * i) * \sin(\beta * j)$, sendo i e j as variáveis que iteram sobre as stacks e slices, respetivamente, r o raio inferior da figura, $\alpha = 360/stacks$ e $\beta = 360/slices$.

2.5.2 Coordenadas de Textura

Tal como no cone e na esfera, iterando sobre os slices e stacks do torus com variáveis i, j, as coordenadas de textura do Torus serão obtidas através de $(i/slices, j/stacks)$.

2.6 Bezier

Dado um certo ponto da superfície, $B(u, v)$, com a matriz de Bézier M, podemos calcular a normal e as coordenadas de textura do mesmo da seguinte forma:

2.6.1 Normais

$$u' = \begin{bmatrix} 3*u^2 & 2*u & 1 & 0 \end{bmatrix} \mathrm{M} \begin{bmatrix} P00 & P01 & P02 & P03 \\ P10 & P11 & P12 & P13 \\ P20 & P21 & P22 & P23 \\ P30 & P31 & P32 & P33 \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

$$v' = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P00 & P01 & P02 & P03 \\ P10 & P11 & P12 & P13 \\ P20 & P21 & P22 & P23 \\ P30 & P31 & P32 & P33 \end{bmatrix} M^T \begin{bmatrix} 3*v^2 \\ 2*v \\ 1 \\ 0 \end{bmatrix}$$

Por último, a normal do ponto será $\mathbf{u}' \times \mathbf{v}'$.

2.6.2 Coordenadas de Textura

Para o mesmo ponto P, as coordenadas de textura serão apenas (v', u').

Capítulo 3

Engine

Neste capítulo, focaremos na ativação das funcionalidades de iluminação e texturização no engine, além da leitura em ficheiro e aplicação de normais e coordenadas de textura.
A iluminação é essencial para simular a interação da luz com os objetos, criando realismo às cenas 3D. A texturização, por sua vez, permite aplicar imagens detalhadas às superfícies dos modelos, aprimorando a riqueza visual.
Exploraremos como ler as normais e coordenadas de textura dos arquivos XML, integrando essas informações ao engine para produzir renderizações mais precisas e visualmente atraentes.

3.1 Leitura do XML

De forma a otimizar o desenho das nossas cenas, foi alterada a função **get_attributes**, de forma a também recolher possível informação sobre as luzes do nosso mundo, guardando num vetor **luzglobal** o tipo de luz em questão, bem como os seus atributos.
Também foi definida a função **processGroup**, que percorre o ficheiro XML pretendido, que, por cada elemento **group** que encontra, armazenamos num vetor **childs**, na posição i, os seus elementos **group** filhos, apenas guardando os filhos de primeira ordem. Também armazenamos aqui nesta função o nome dos ficheiros num vetor **files**, e as transformações num vetor **ts**, na posição relativa à ordem do elemento **group** em questão.
Por último, na função **get_lights_textures** recolhemos informações relativas a cor e texturas de um ficheiro .3d, guardando estas nos vetores **textures** e **luzObj**, na posição relativa à ordem do ficheiro, respetivamente.
Desta forma, ao desenhar a nossa cena, conseguimos percorrer os filhos de cada grupo, empilhando na matriz as transformações sucessivas a serem efetuadas, e repor a matriz original quando for necessário renderizar a cena contida num grupo irmão.

3.2 Iluminação

Com a introdução da iluminação ao nosso meio, primeiramente, quando um ficheiro .3d é analisado, recolhemos agora depois dos valores de um dado vértice, os 3 valores referentes à normal do mesmo vértice, guardando estes num vetor de floats, **normalsArray**. No fim de completar a leitura de um ficheiro, este vetor contendo as normais dos vértices do objeto, é guardado em **normalsArrays**, inserido na posição do vetor relativa à ordem do ficheiro .3d. Caso o ficheiro já tenha sido analisado anteriormente, em **normalsArray** é guardado apenas um valor referente à posição do mesmo vetor em que as coordenadas das normais foram armazenadas em primeiro lugar.

3.3 Texturas

Semelhante ao processo anterior, depois de ler as normais de um vértice, prosseguimos a leitura de 2 valores referentes às coordenadas de textura do próprio vértice, guardando agora estes em **texturesArray**, que irá ser armazenado em **texturesArrays**. No caso de um arquivo já lido anteriormente, realizamos o mesmo processo, guardando na primeira posição do vetor um valor que indique onde se encontram as coordenadas de textura em **texturesArrays**.
Depois deste processo, temos a garantia de que para um vértice de uma dada figura, os valores das suas coordenadas, bem como da sua normal e coordenadas de textura, encontram-se armazenados em 3 vetores diferentes, porém na mesma posição.
Com os ficheiros das texturas armazenados, invocamos a função **loadTextures**, que percorre o vetor **textures**, e armazena a informação do ficheiro, no vetor de **GLuint**, **texturesglu**, pré computando então esta informação, não sendo necessário o cálculo duma textura referente a um sólido para cada vez que a cena seja desenhada.

3.4 VBOs

Tendo agora mais dois vetores onde armazenamos as normais e as coordenadas de textura, temos também mais dois vetores de **GLuint**, **vbo2** e **vbo3**, para armazenarem esses dois novos tipos de informação. Além disso, tal como para os pontos, quando numa dada posição do vetor **numbersArrays** encontramos um vetor com apenas um valor **i**, então sabemos que tanto os vértices, como as normais e as coordenadas de textura estarão armazenados na posição **i** dos três vetores. Dessa forma, não transferimos dados repetidos para os vetores de **vbos**, guardando num vetor global **mapa** a posição nos vetores de **vbos** em que se encontram as informações relativas ao ficheiro em questão, continuando assim com a otimização do nosso programa. Caso contrário, então geramos um ID de buffer, transferindo a informação nos três vetores de **float** para os três vetores de **GLuint**, também associando no **mapa** a posição nos vetores de **vbos** da figura em questão.

```
std::vector<GLuint> vbo(1);
std::vector<GLuint> vbo2(1);
std::vector<GLuint> vbo3(1);
```

Figura 3.1: Inicialização dos vbos usados

3.5 Renderização

Toda a atualização às estruturas de dados de armazenamento de informações, foi feita com o objetivo de melhorar e otimizar vastamente a forma como os objetos são desenhados.
A função **global_light** é primeiro chamada, percorre apenas o vetor **luzglobal** e para cada elemento, identifica o tipo de luz através de uma flag na primeira posição, obtendo os restantes atributos da mesma, ativando a luz em questão.
A função **draw**, previamente com a função de percorrer todos os ficheiros, aplicando as transformações necessárias, desenhando por último os pontos, tem agora uma utilidade diferente. É chamada no **RenderScene**, com o índice **0**, referente ao grupo inicial, começa por empilhar a matriz atual na stack, invoca a nova função **draw_group** para o grupo recebido como argumento, e percorre os filhos do mesmo, chamando-se recursivamente para estes grupos filhos. Por último, repõe a matriz anterior na stack.
Finalmente a nova função **draw_group** responsabiliza-se por em primeiramente, efetuar todas as trans-

formações.
De seguida, caso existam, são aplicados os cores dos sólidos em questão, e por último são desenhados os pontos dos ficheiros do grupo que recebe como argumento, tal como era feita na função **draw** nas fases anteriores. Porém, agora para um dado ficheiro, obtemos a posição em que se encontram os seus pontos, normais, e coordenadas de texturas, nos vetores de **GLuint**, através do vetor **mapa**. Aplicadas agora as normais e as texturas (caso necessárias), os pontos são por último desenhados.

Capítulo 4

Funcionalidades Extra

Como incentivado no enunciado do trabalho, tomamos a iniciativa de expandir o nossa câmara, agora podendo também aproximar ou distanciar a câmara do ponto para onde esta observa, com o propósito de melhorar a visualização do desenho de cenas que ocupem mais espaço, ou em que os objetos contidos nela se encontrem mais distantes.
Este efeito é obtido ao incrementar/decrementar a variável global **dist**, que referencia a distância da câmara ao centro. Ao reescrever o seu valor, a posição da câmara também irá ser alterada.

Capítulo 5

Sistema Solar

Com o propósito de demonstrar as funcionalidades implementadas nesta fase, foi-nos solicitado então um arquivo XML que modele dinamicamente o nosso Sistema Solar, fazendo uso de texturas, iluminação e luzes. Apresentamos então dois modelos do Sistema Solar.
No primeiro modelo, utilizamos apenas luzes para colorizar o nosso modelo, mantendo a antiga versão do XML, e atribuindo valores de iluminação aos ficheiros, de forma a gerar cores que respeitassem a cor verdadeira de cada astro.
Na segunda versão, continuamos com o modelo XML passado, atribuindo agora texturas para cada ficheiro, cada textura foi devidamente escolhida de acordo com o corpo celeste em questão.

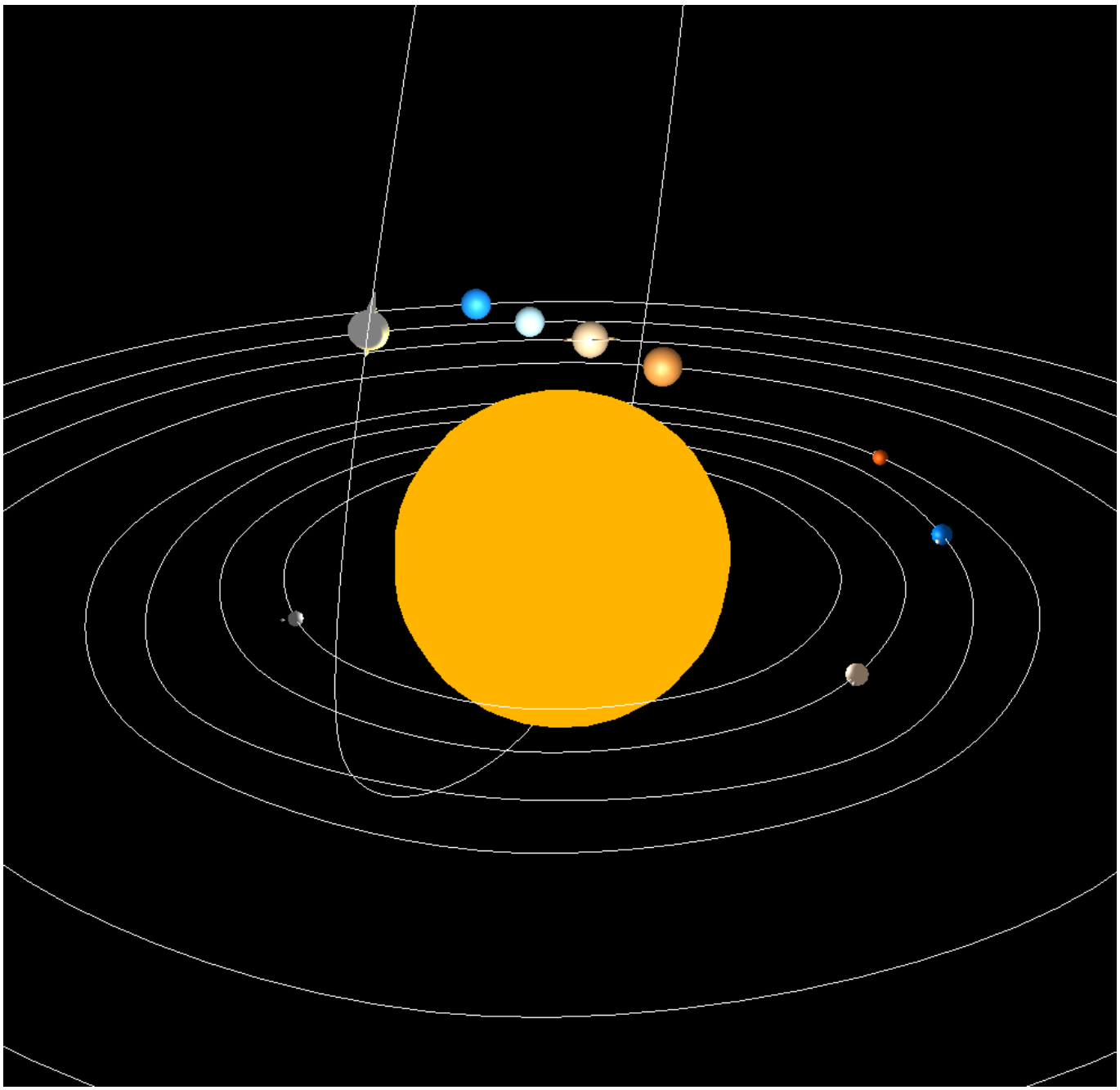

Figura 5.1: Desenho do Sistema Solar, 1ª versão

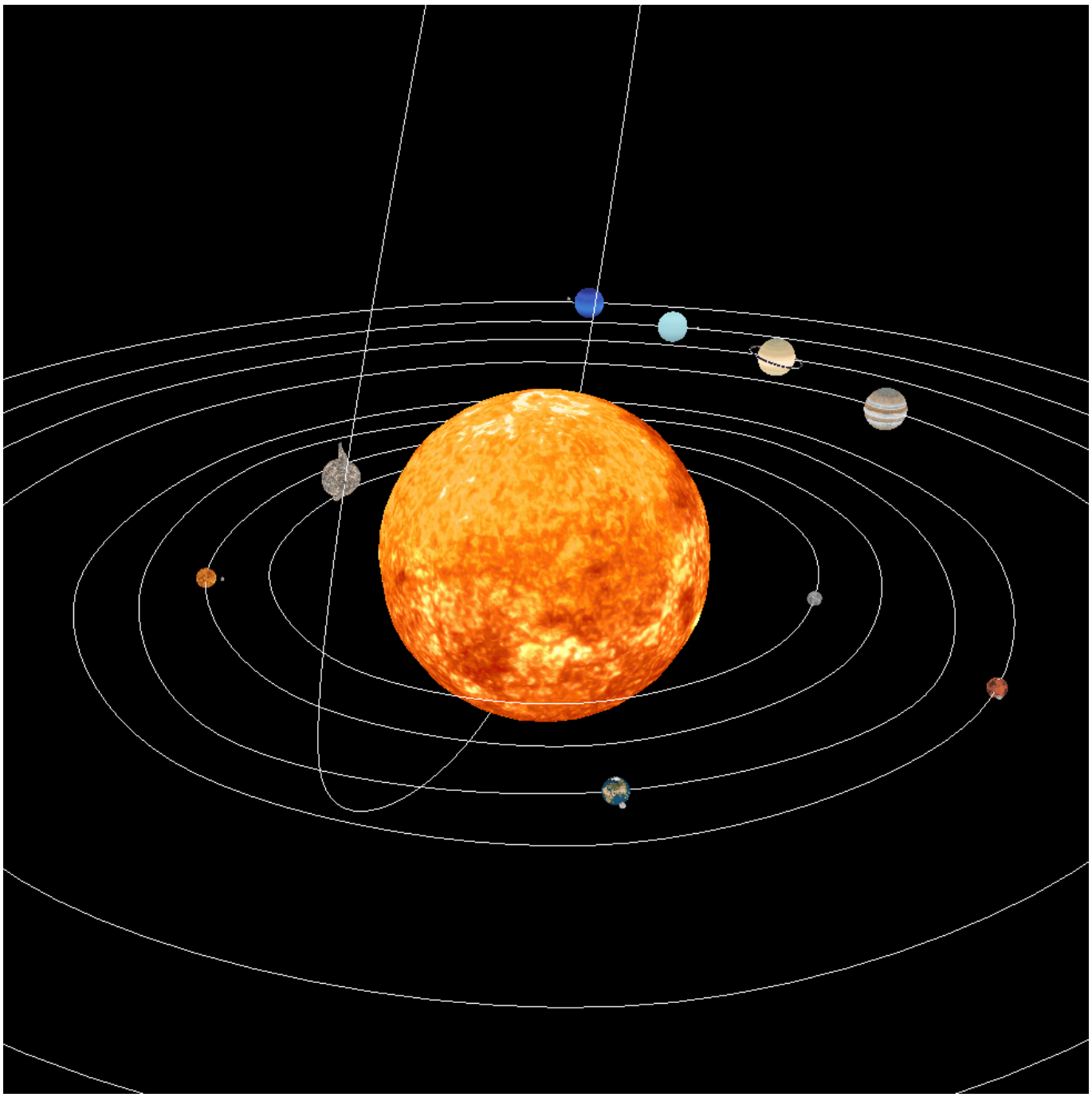

Figura 5.2: Desenho do Sistema Solar, 2ª versão

Capítulo 6

Conclusão

Concluindo então esta quarta e última fase do projeto, destacamos a implementação bem-sucedida das funcionalidades propostas, como armazenar e aplicar os diferentes tipos de iluminação, as várias cores dos objetos gerando uma junção de iluminação difusa, ambiente, especular, emissiva e o seu brilho. Tudo isto devido ao facto do sucesso na implementação do calculo e obtenção das coordenadas de textura, e normais, para cada vértice dos sólidos desenhados.
A cena dinâmica do sistema solar, com a aplicação de luzes e texturas aos corpos celestes, é uma boa demonstração de como os objetivos desta fase foram alcançados.
Reconhecemos também que poderiam ser realizadas várias melhorias ao trabalho, tal como cenas dinâmicas mais complexas, bem como uma abordagem diferente no calculo de texturas e normais.
Por último, o grupo atribui um balanço positivo ao trabalho final realizado, tendo cumprido na vasta maioria os requisitos pedidos ao longo das 4 fases.