Universidade do Minho

Departamento de Informática
Mestrado Integrado em Engenharia Informática
3º Ano, 2º Semestre

Computação Gráfica

Trabalho Prático - Fase IV

Grupo 21

Ana Pereira
A81712

Francisco Freitas
A81580

Maria Dias
A81611

Pedro Freitas
A80975

May 15, 2019

Índice

1 Introdução

Partindo do trabalho já desenvolvido nas três fases que se sucederam, passamos agora à quarta e última fase de construção de uma aplicação que nos permitirá representar o Sistema Solar.

Nesta última fase do projeto, pretende-se a inclusão de texturas e iluminação nas figuras de forma a produzirmos uma representação mais realista do desejado Sistema Solar.

Assim o ficheiro XML vai poder sofrer alterações e pode suportar a inserção de textura além das já implementadas *rotação*, *translação*, *cor* e *escala*.

Além disso vamos ter alterações quanto à câmara podendo agora usufruir da *First Person Camara*.

2 Arquitetura do Código

Nesta última fase, mantemos as classes definidas anteriormente, tendo alterado a classe *Shape* e *Group* de modo a introduzir as texturas e iluminação na nossa aplicação. Para além disso, foram criadas duas novas classes, *Material* e *Light*.

2.1 Classes inalteradas

2.1.1 Point.cpp

Classe que guarda as coordenadas x, y e z de um ponto, necessário para o desenho de cada vértice de um triângulo, uma vez que a base de todas as figuras desenhadas é o triângulo.

2.1.2 Action.cpp

Classe que representa as ações que podemos aplicar aos modelos. Para a realização desta etapa foi necessário criar a subclasse *Translate* que para além da tag, x, y e z herdados acrescenta ainda a variável **time**, que define o número de segundos que demora a percorrer a curva Catmull-Rom e ainda vetores que guardam os pontos dessa mesma curva.

A variável **time** também foi adicionada à subclasse *Rotate*.

2.1.3 Bezier.cpp

Classe encarregue de efetuar o parse do Bezier patch e conversão do mesmo num ficheiro de formato .3d, contendo os vértices da figura a desenhar.

2.2 Classes novas/alteradas

2.2.1 Material.cpp

Esta nova classe contém os parâmetros necessários de modo a representar as cores produzidas em cada modelo através dos vetores ou pontos de iluminação. Estes parâmetros são: **diffuse**, **ambient**, **specular**, **emission** e **shininess**, sendo que os quatros primeiros se tratam de arrays de floats, sendo representados através da primitiva *Action*.

2.2.2 Shape.cpp

Classe que armazena num vector o conjunto de todos os pontos pertences a um determinado modelo. Nesta última etapa foram adicionados mais dois vetores de pontos a esta classe, um que guarda os pontos referentes às normais e outro referente às coordenadas de textura. São ainda acrescentadas as variáveis **vertices**, **normals**, **textures**, **texID** e um Material **material**, com as componentes da iluminação definidas para o modelo.

2.2.3 Group.cpp

Classe que guarda num vetor todas as ações presentes num determinado group do ficheiro XML e noutro vetor o conjunto das figuras necessárias para o desenho desse mesmo group. Nesta fase, para a implementação da iluminação, foi adicionado um novo vetor que guarda as luzes do group em questão.

2.2.4 Light.cpp

Classe que representa as origens de iluminação que ou são feixes de luz, emitindo em uma única direção ou são pontos que emitem luz em todas as direções. É portanto formada por um ponto/vetor e por um booleano que distingue se é ponto ou vetor.

3 Generator

3.1 Aplicação das normais

Nesta fase um dos focos principais foi aplicar texturas às demais figuras geométricas que serão representadas. Para isso foi necessário compreender as faces e os pontos que constituem essas mesmas figuras.

O estudo das normais baseia-se em obter os vetores normais (perpendiculares) a cada ponto que constitui cada figura.

3.1.1 Plano

Obter as normais desta figura geométrica é um processo um pouco simples, pois baseia-se apenas em verificar em que plano geométrico é que esta figura geométrica está inserida. Como em fases anteriores tinha sido implementado, o plano está definido no plano xOz. Sabendo isto podemos concluir que todos os pontos têm um vetor normal comum entre si que pode ser representado pelo vetor: **(0,1,0)**.

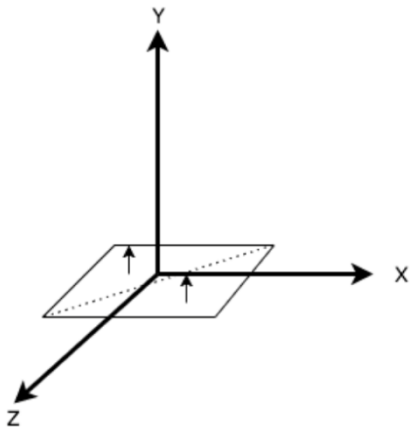

Figura 1: Vetores normais de um plano

3.1.2 Esfera

Para esta figura precisamos de saber a orientação da figura. Para isso temos um vetor desde a origem do figura até ao ponto em questão, sendo esta informação obtido quando originamos os pontos. Sabendo isso sabemos que para um dado vértice V(x,y,z):

(cos(dir), y/raio, sin(dir))

onde *dir* representa o desvio horizontal que é feito para se iterar sobre a circunferência.

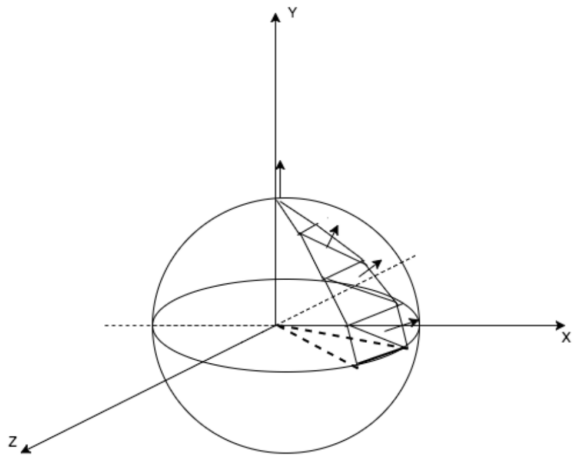

Figura 2: Vetores normais de uma esfera

3.1.3 Box

Para esta figura geométrica temos de seguir o mesmo raciocínio da figura geométrica do *Plano* e aplicar a cada face da caixa. Sendo assim podemos concluir facilmente os vetores de cada face:

- **Face Frontal:** (0,0,1)
- **Face Traseira:** (0,0,-1)
- **Face Direita:** (1,0,0)
- **Face Esquerda:** (-1,0,0)
- **Base:** (0,-1,0)
- **Topo:** (0,1,0)

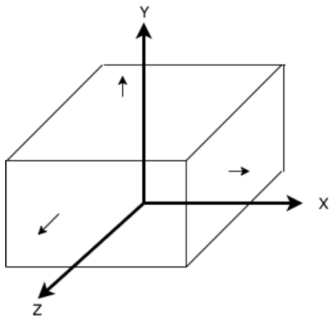

Figura 3: Vetores normais de um paralelepípedo

3.1.4 Cilindro

Para esta figura geométrica temos também dois tipos de comportamento diferentes. Quantos às bases podemos aplicar o raciocínio dos planos. Por isso temos que:

- **Base** : (0,-1,0)
- **Topo** : (0,1,0)

Quanto à **lateral** do Cilindro o vetor representante da normal é :

$$(cos(alfa), 0, sin(alfa))$$

, onde alfa é a amplitude a que se encontra o vértice.

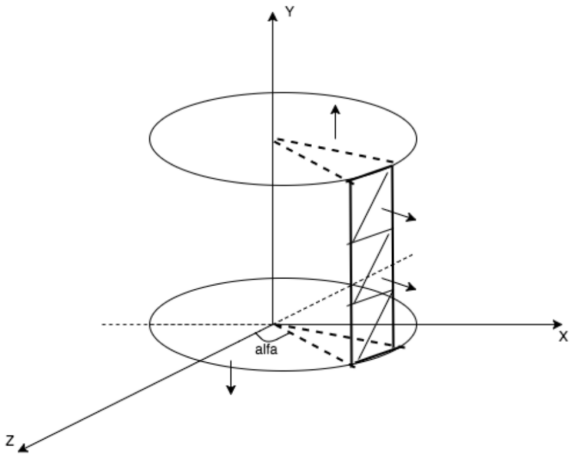

Figura 4: Vetores normais de um cilindro

3.1.5 Torus

Para obtermos os vetores normais desta figura geométrica temos de ter a noção da sua composição. Sendo o Torus uma figura geométrica comparada a um pneu sabemos que ele tem um raio interior(r) e um raio exterior(R). Assim para obtermos os vetores normais será necessário um processo iterativo e através da seguinte fórmula podemos ver como é que estres são obtidos:

$$(cos(DA), r * sin(DL), sin(DA))$$

, onde **DA** representa o desvio do anel e **DL** o desvio de cada lado que forma o anel.

3.1.6 Bezier Patches

De forma a obter o conjunto dos vetores normais correspondentes aos vértices gerados pelos ficheiros .patch, calculou-se a normal dos vértices do modelo, recorrendo à seguinte expressão, em que p representa um vértice da figura gerada através do ficheiro .patch:

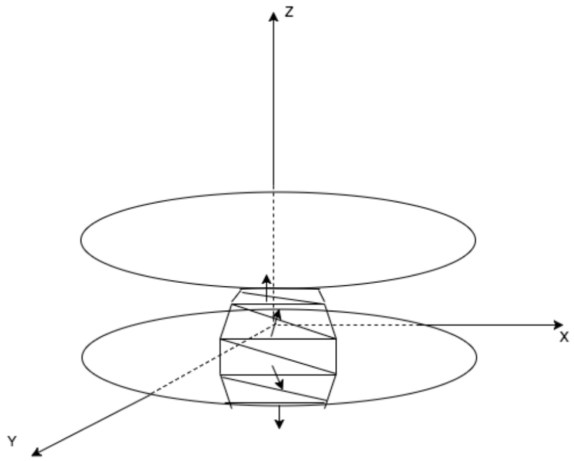

Figura 5: Vetores normais de um torus

$$magnitude = \sqrt{(p_x \times p_x) + (p_y \times p_y) + (p_z \times p_z)}$$
$$normal = (\frac{p_x}{magnitude}, \frac{p_y}{magnitude}, \frac{p_z}{magnitude})$$

3.2 Aplicação das texturas

3.2.1 Plano

Para mapear textura num modelo, temos de o "desdobrar" numa figura 2D com, no máximo, uma dimensão de 1x1. No plano, não precisamos de fazer cálculos, uma vez que encaixa perfeitamente na textura. O mapeamento é exemplificado na figura seguinte, num plano de lado $h \times 2$.

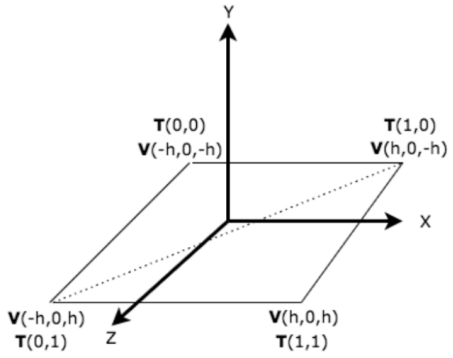

Figura 6: Mapeamento de vértices normais para vértices de textura (u,v)

3.2.2 Esfera

Para um dado ponto P numa esfera, sendo d o vetor unitário que liga o ponto P ao centro da esfera, as coordenadas uv podem ser calculadas através das expressões que se seguem.

$$d = normalize(P)$$

$$u = 0.5 + \frac{\arctan2(d_x, d_z)}{2\pi}$$

$$v = 0.5 - \frac{\arcsin(d_y)}{\pi}$$

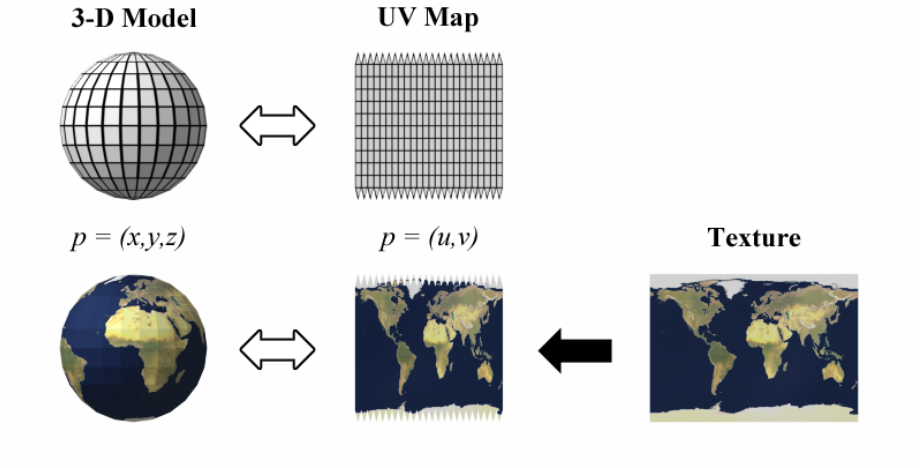

Figura 7: Aplicação de textura numa esfera

4 Engine

4.1 VBOs

Com a implementação da iluminação e da textura, foi necessário criar mais dois VBOs, uma vez que passamos a ter mais dois conjuntos de vértices, para além do conjunto relativo aos vértices que compõem cada modelo. Na figura seguinte, encontra-se o método que trata de criar os VBOs para cada modelo, definido na classe *Shape*.

```
void Shape::vbo() {
    float *vertex = (float*) malloc(sizeof(float)*3*this->points.size());
    float *n = (float*) malloc(sizeof(float)*3*this->normal.size());
    float *t = (float*) malloc(sizeof(float)*2*this->texture.size());
    int index = 0, index2 = 0, index3 = 0;

    for (int i = 0; i < this->points.size(); i++) {
```

```
        vertex[index] = this->points[i]->getX();
        vertex[index + 1] = this->points[i]->getY();
        vertex[index + 2] = this->points[i]->getZ();
        index += 3;
    }

    for (int i = 0; i < this->normal.size(); i++) {
        n[index2] = this->normal[i]->getX();
        n[index2 + 1] = this->normal[i]->getY();
        n[index2 + 2] = this->normal[i]->getZ();
        index2 += 3;
    }

    for (int i = 0; i < this->texture.size(); i++) {
        t[index3] = this->texture[i]->getX();
        t[index3 + 1] = this->texture[i]->getY();
        index3 += 2;
    }

    glGenBuffers(1,&vertices);
   glBindBuffer(GL_ARRAY_BUFFER,vertices);
   glBufferData(GL_ARRAY_BUFFER,sizeof(float)*index,vertex,
      GL_STATIC_DRAW);

   glGenBuffers(1, &normals);
   glBindBuffer(GL_ARRAY_BUFFER,normals);
   glBufferData(GL_ARRAY_BUFFER,sizeof(float)*index2,n, GL_STATIC_DRAW);

   glGenBuffers(1,&textures);
   glBindBuffer(GL_ARRAY_BUFFER,textures);
   glBufferData(GL_ARRAY_BUFFER,sizeof(float)*index3,t, GL_STATIC_DRAW);

    free(vertex);
    free(n);
    free(t);
}
```

4.2 Textura

O método *loadTexture* trata de fazer o carregamento da textura de uma *Shape*. Para isso, e como já vimos anteriormente, existe na classe *Shape* uma variável que identifica a textura que lhe é atribuída no momento do carregamento da imagem.

Na imagem que se segue, podemos ver o método responsável por ligar a textura ao respetivo modelo.

```
void Shape::loadTexture(string path) {
```

```
    unsigned int tw,th, t;
    unsigned char *texData;
    ILuint img;
    ilEnable(IL_ORIGIN_SET);
    ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
    ilGenImages(1, &img);
    ilBindImage(img);
    if(! ilLoadImage((ILstring) path.c_str()))
              cout << "Erro a ler imagem :'(\n";

    tw = ilGetInteger(IL_IMAGE_WIDTH);
    th = ilGetInteger(IL_IMAGE_HEIGHT);
    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
    texData = ilGetData();
    glGenTextures(1, &texID);
    glBindTexture(GL_TEXTURE_2D, texID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA,
        GL_UNSIGNED_BYTE, texData);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

4.3 Iluminação

A iluminação dos diferentes modelos é gerada através dos vetores normais desses modelos, sendo através das normais que conseguimos obter a intensidade da luz que atinge um dado triângulo da forma.

Para além da construção das normais dos modelos, que já foi explicada na secção anterior, é preciso ter em consideração aspetos como as componentes da cor, o direcionamento e posicionamento da luz.

A cor de um modelo depende do valor dos componente da cor, bem como da cor da luz que o atinge. Segue-se uma breve descrição de cada um destes componentes.

Diffuse colour: A cor que um objeto tem quando está a ser atingido por uma luz branca. No fundo, é a cor do próprio objeto, em vez da cor da reflexão da luz no mesmo.

Ambient colour: A cor de um objeto quando o mesmo se encontra "à sombra", ou seja, quando não está a ser atingido por luz.

Emissive colour: Relativa à cor da iluminação própria de um objeto.

Specular colour: A cor da luz da reflexão especular. Consiste no tipo de reflexão característica de uma superfície brilhante.

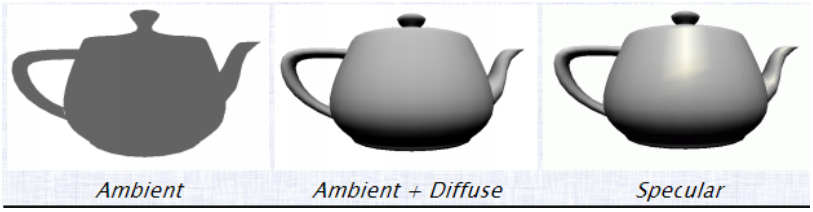

Figura 8: Exemplos de aplicação de alguns componentes da cor

Ao construir o nosso sistema solar, tivemos de ter em conta o posicionamento e direcionamento da luz. Para esse efeito, existem duas opções: a existência de um ponto de luz que que emite feixes de luz em todas as direções, ou a existência de um único feixe de luz, que apenas ilumina numa determinada direção. Para o nosso projeto, o tipo de iluminação que pretendíamos implementar é o primeiro, uma vez que o Sol é a principal fonte de iluminação do sistema solar, e o mesmo emite luz em todas as direções. Assim sendo, a posição do ponto de luz é (0,0,0) e pretende-se que seja um ponto, pelo que o array *pos* tem o valor {*0,0,0,1*}.

```
void Light::draw(){

   GLfloat amb[4] = {0.1,0.1,0.1, 1};
   GLfloat diff[4] = {1, 1, 1, 0};
   GLfloat pos[4] = {point->getX(), point->getY() , point->getZ(), (float)
       this->type};

   // POSICAO DA LUZ
   glLightfv(GL_LIGHT0, GL_POSITION, pos);

   // COLORACAO DA LUZ
   glLightfv(GL_LIGHT0, GL_AMBIENT, amb);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diff);
}
```

Em último lugar, ao aplicar a iluminação no nosso modelo, temos de considerar em que fase esta é aplicada. A aplicação da iluminação é feita pelo método **glLightfv(GL_LIGHT0,GL_POSITION,pos)**, em que *pos* é o array do posicionamento da luz que acabamos de referir. Este método pode ser aplicado em três momentos distintos, sendo que o comportamento da iluminação é completamente diferente para cada uma dessas alternativas.

Antes da câmara: Luz fixada no espaço da câmara.

Depois da câmara e antes das transformações: Luz fixada no espaço global.

Depois das transformações geométricas: Luz fixada para o objeto.

Para o desenho do sistema solar, escolhemos aplicar o segundo método de ilunação, uma vez que pretendemos uma luz globalmente fixada, representando a luz do Sol.

4.4 Câmara

Nesta fase decidimos alterar a nossa câmara para poder navegar entre os planetas e para poder ver o sistema solar de vários ângulos. Para esse efeito implementámos 2 câmaras, cada uma com um objetivo diferente, uma utiliza o cursor e a outras as teclas, sendo a câmara para navegar entre os planetas uma First Person Camera. A câmara é definida por 3 vetores. O vetor camera position(P), que tal como o nome indica é um vetor no espaço que indica a posição da nossa câmara, de forma a saber a posição da câmara em relação ao referencial. O outro vetor é designado por Look at point(L), que basicamente representa a direção em que câmara está a olhar. E, por fim, temos o up vector(), que representa o eixo superior positivo(y) da câmara. Assim na câmara First Person sempre que é pressionada uma das teclas AWSDEQ a câmara é atualizada consoante o movimento. Se queremos mover a câmara para a frente ou para trás (W e S) o L e o P são calculados com as seguintes fórmulas:

$$\vec{d} = L - P = (Lx - Px, 0, Lz - Lz)$$
$$P' = P + k\vec{d}$$
$$L' = L + k\vec{d}$$

sendo P' e L' a nova posição do utilizador e o local para onde este está a olhar, respetivamente, e k um número. Já no caso de querermos mover a câmara para a esquerda ou a direita (A e D), o L e o P serão calculados com as seguintes fórmulas:

$$\vec{d} = L - P$$
$$\vec{r} = \vec{d} \times \vec{up}$$
$$P' = P + k\vec{r}$$
$$L' = L + k\vec{r}$$

sendo P' e L' a nova posição do utilizador e o local para onde este está a olhar respetivamente, k um número e r um vetor lateral sobre o qual nos podemos mover.

Se pretendermos que a câmara faça rotação (E e Q) o P não é afetado e é recalculado o L:

$$L = (Px + sin(\alpha), Py, Pz - cos(\alpha))$$

sendo o α ângulo da rotação.

A câmara que utiliza o cursor tem como objetivo mostrar o sistema solar de vários ângulos, com o foco sempre no centro do sistema solar. Por este motivo o L não será afetado e só teremos que recalcular o P.

Como não é posível utilizar as 2 câmeras ao mesmo tempo, usámos a tecla Z para alternar entre as duas câmaras.

5 Resultados Obtidos

Depois de implementadas as funcionalidades foi hora de pormos em prática de forma a ir de encontro com o pedido: uma maqueta dinâmica do sistema solar. Segue nas próximas figuras o resultado obtido através do ficheiro "solarsystem.xml".

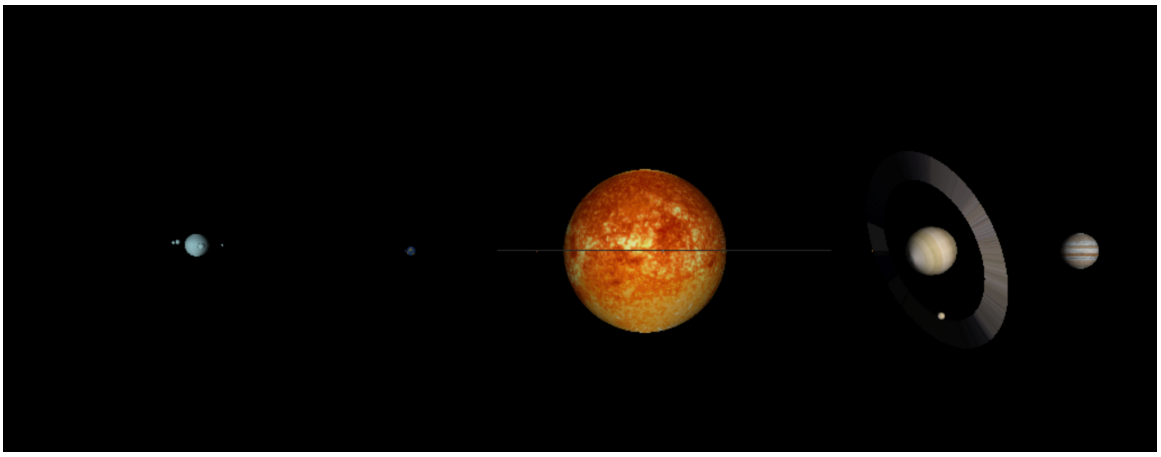

Figura 9: Vista lateral do Sistema Solar

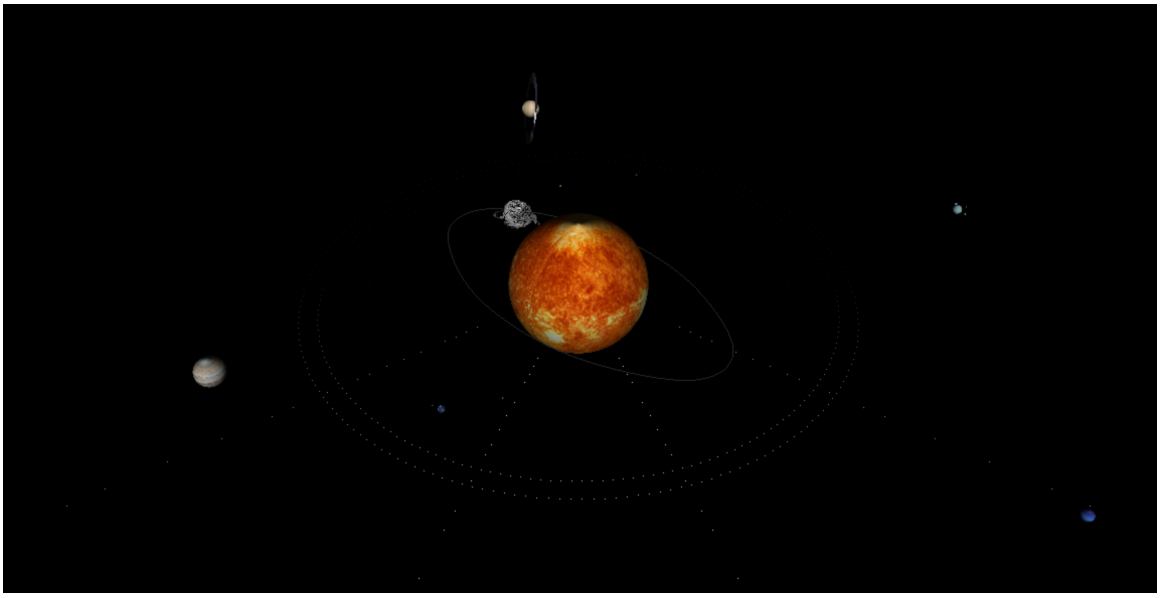

Figura 10: Vista de topo do Sistema Solar

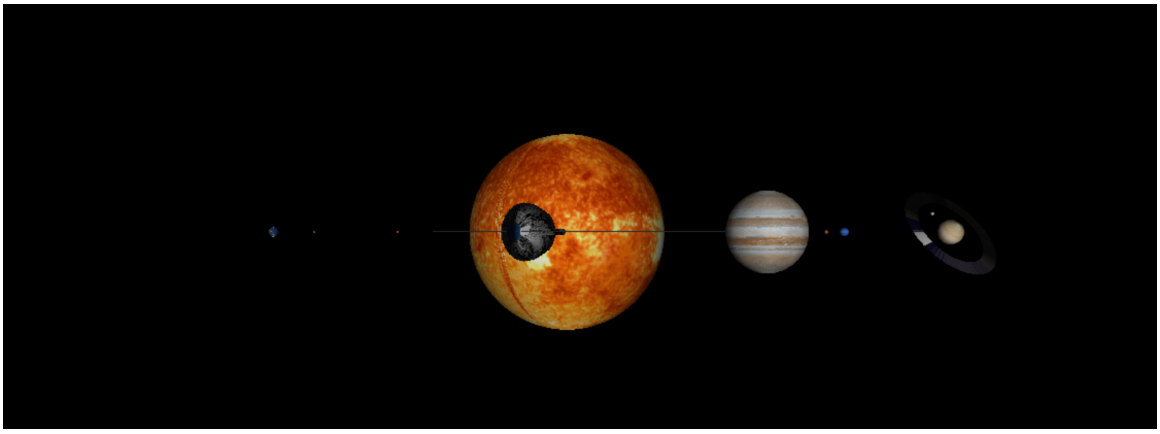

Figura 11: Vista lateral do Sistema Solar (2)

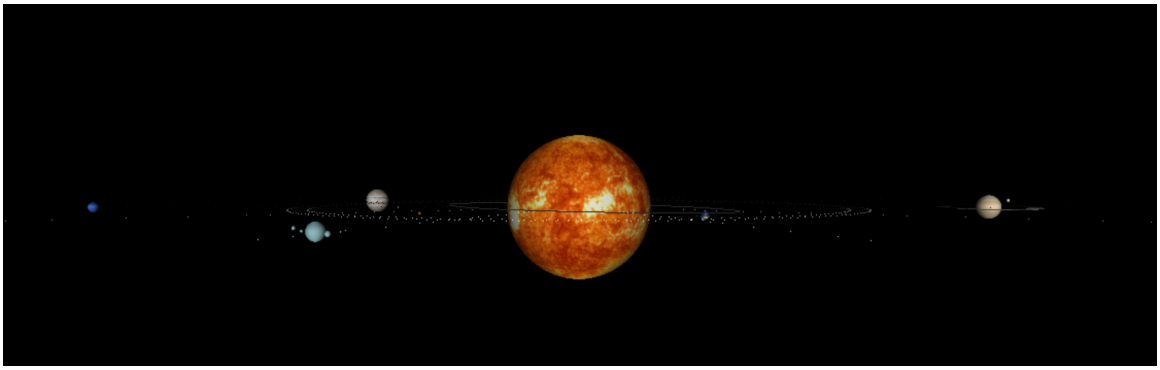

Figura 12: Vista lateral do Sistema Solar (3)

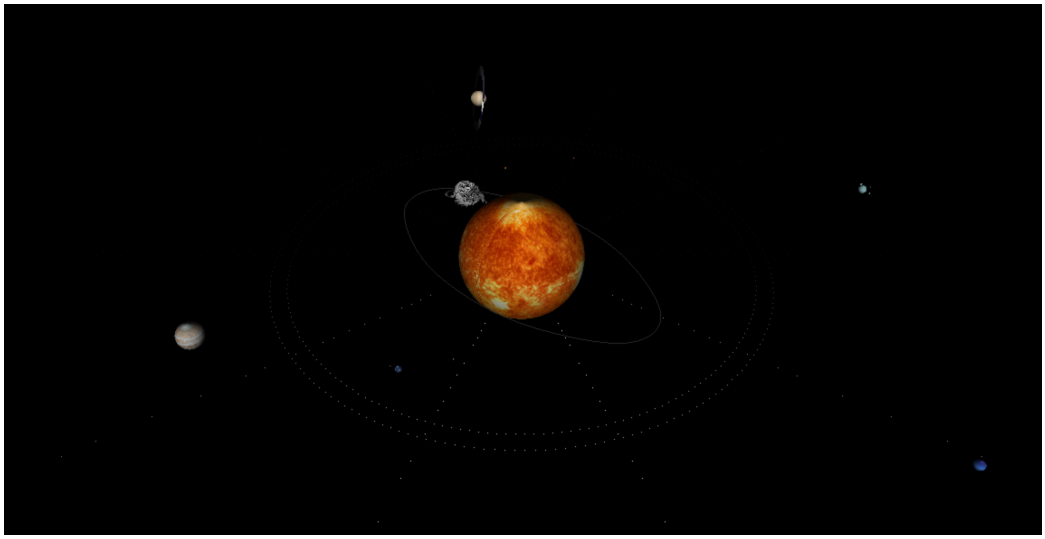

Figura 13: Vista de topo do Sistema Solar

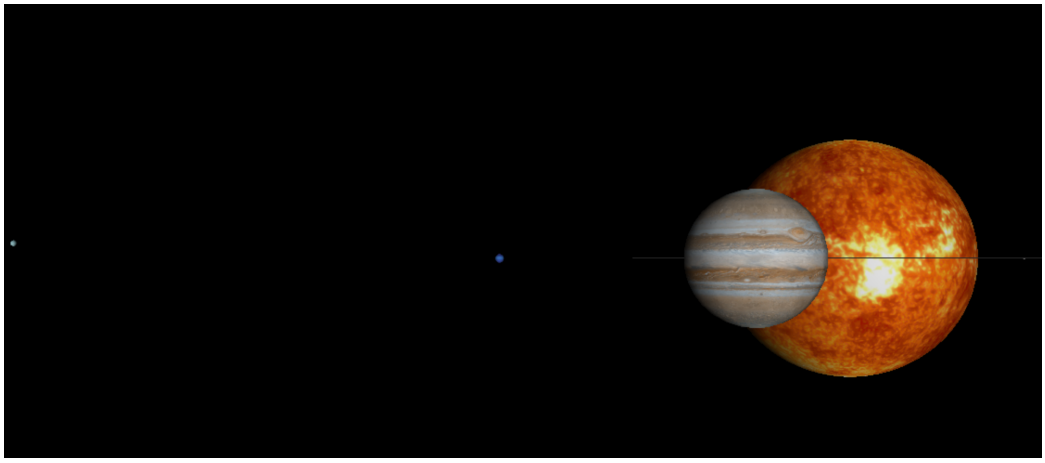

Figura 14: Vista aproximada

6 Conclusão

Esta última fase do trabalho permitiu-nos assim concluir todo o projeto realizado durante todo este semestre para a Unidade Curricular de *Computação Gráfica*.

Depois de termos passado por várias fases desde o desenho de simples figuras geométricas como planos, caixas ou esferas até à introdução de luzes, texturas, movimento simples e/ou com curvas, achamos que temos um trabalho muito bem conseguido. O Sistema Solar que nos foi desafiado a fazer teve assim uma boa abordagem por parte do grupo e obtemos um resultado muito fidedigno à realidade do Sistema Solar.

Dado o projeto como terminado temos a consciência que este trabalho foi muito importante tanto para consolidar como para adquirir novos conhecimentos nesta área da Computação sempre aproveitando as funcionalidades que o *OpenGL* e o *GLUT* têm para oferecer.