Universidade do Minho
Escola de Engenharia

Computação Gráfica

4ª Fase

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601
Carlos Miguel Luzia Carvalho A89605
Bárbara Ferreira Teixeira A89610


A89588


A89601


A89605


A89610

30 de maio de 2021

Conteúdo

1 Introdução

1.1 Contextualização

No âmbito da Unidade Curricular de Computação Gráfica foi nos pedido o desenvolvimento de um projeto dividido em 4 fases, com o objetivo de, através de ferramentas como o C++ e o OpenGL criar um mini mecanismo 3D baseado num cenário gráfico.

O relatório presente é referente à quarta e última fase, na qual o objetivo é a inclusão de texturas e iluminação no trabalho anteriormente desenvolvido, tendo como finalidade a criação do modelo animado do Sistema Solar.

1.2 Resumo

Este relatório diz respeito à quarta fase do projeto prático desenvolvido na unidade curricular de Computação Gráfica. Tratando-se da quarta fase, várias funcionalidades das fases anteriores foram mantidas e outras alteradas, de modo a melhor cumprirem os requisitos correspondentes a esta fase. Assim sendo, esta quarta fase traz consigo novidades relativas ao generator e ao engine, uma vez que foi implementada a iluminação e o uso de texturas nas diferentes formas geométricas.

Começando pelo generator, nesta quarta fase, este torna-se capaz de, a partir de uma dada imagem, ser capaz de obter as normais e os pontos de textura para os vários vértices das primitivas geométricas anteriormente criadas.

Já o engine sofrerá algumas modificações e, ao mesmo tempo, receberá novas funcionalidades. Os ficheiros XML passarão a conter as informações relativas à iluminação e texturas do cenário. Assim sendo, terá que ser alterado não só o parser responsável por ler esses mesmos ficheiros, mas também o modo como é processada toda a informação recebida com o intuito de gerar todo o cenário pretendido.

Tudo isto tem como finalidade conseguirmos gerar efetivamente um modelo do Sistema Solar ainda mais realista do que o elaborado na fase anterior, pois este passará a ter texturas e a iluminação necessárias.

2 Arquitetura do Código

Sendo esta fase a continuação das três fases anteriores, grande parte do código se manteve inalterado, porém houve necessidade de modificar alguma parte deste e ou ainda acrescentar outro, tendo em vista os requisitos necessários a esta fase.

2.1 Aplicações

Nesta secção serão apresentadas as aplicações fundamentais que permitem gerar e exibir os cenários pretendidos. Na realização desta quarta fase existiram alterações significativas em ambas as aplicações do projeto (generator e engine) tendo em conta os requisitos propostos.

2.1.1 Generator

Na realização desta quarta fase é pretendido que o gerador seja capaz de gerar, para além dos pontos dos vértices já requisitados na primeira fase, as normais e coordenadas de textura a cada um destes.

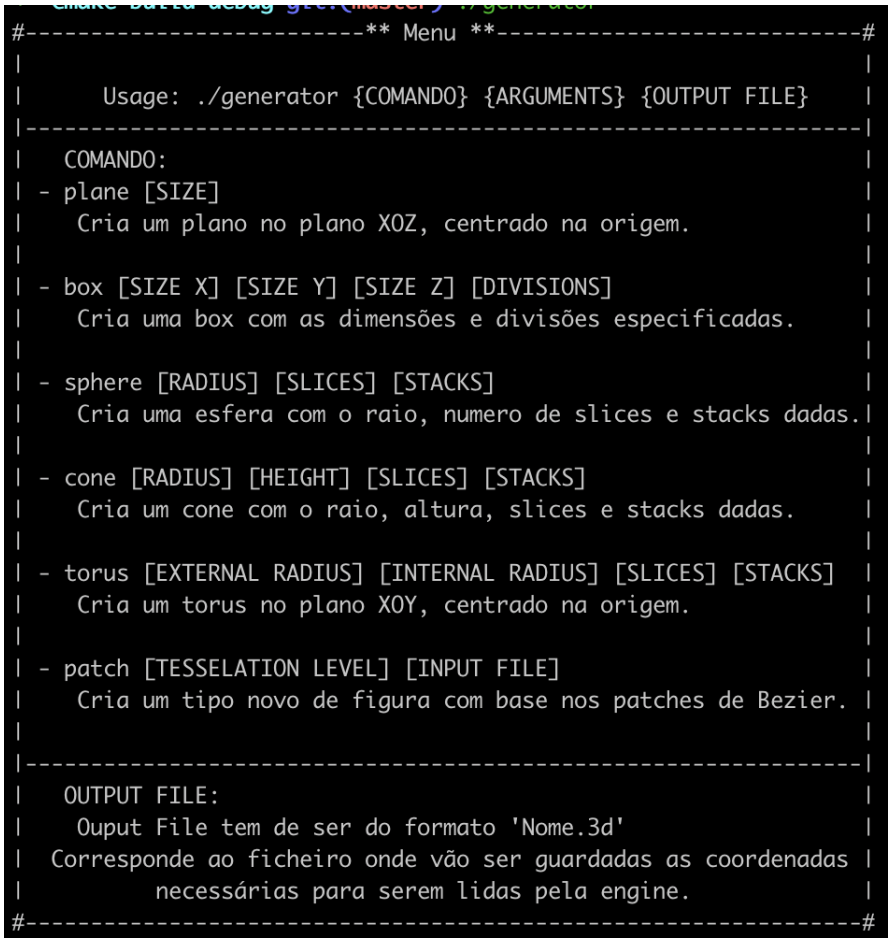

Figura 1: Menu Generator

2.1.2 Engine

Com as devidas alterações implementadas no Generator, identificamos facilmente a necessidade de aplicação de funcionalidades de iluminação e de representação de texturas, através das respetivas normais e coordenadas de textura.

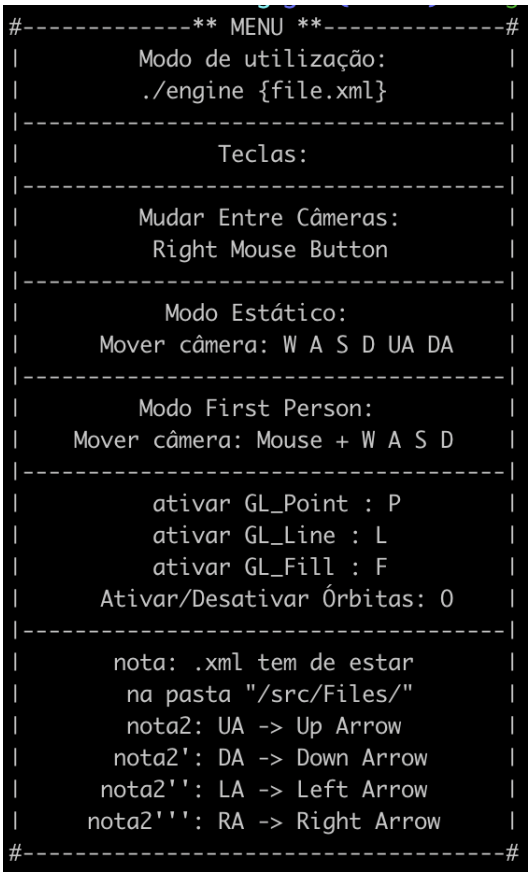

Figura 2: Menu Engine

2.2 Classes

A última fase deste projeto envolveu alterações a algumas das classes e a criação de uma nova de modo a cumprir com os requisitos da mesma.

2.2.1 Light

A criação desta classe deveu-se à necessidade de implementar iluminação. Ela armazena dados relativos ao ponto da iluminação e do tipo podendo ser "*Point*","*Directional*"ou "*Spot*".

Além disso também possui uma função **render()** que é utilizado para renderizar a iluminação. Esta função possui vários vetores com luzes default, isto é, utiliza vetores com a luz ambiente, luz difusa e luz especular já pré-definidos.

$$GLfloat\ ambiente[4] = 0.2, 0.2, 0.2, 1.0; \tag{1}$$

$$GLfloat\ diff[4] = 1.0, 1.0, 1.0, 0.0; \tag{2}$$

$$GLfloat\ spec[4] = 1.0, 1.0, 1.0, 1.0; \tag{3}$$

```
#ifndef GENERATOR_LIGHT_H
#define GENERATOR_LIGHT_H

#include "point.h"
enum Type {POINT,DIRECTIONAL,SPOT};
class Light {
    Point* point;
    Type type;

    public:
        Light();
        Light(Type,Point*);
        Type getType();
        Point* getPoint();
        void setPoint(Point* p);
        void render();
};


#endif //GENERATOR_LIGHT_H
```

Figura 3: light.h

2.2.2 Point

O Point.cpp é a classe que traduz a representação de um ponto no referencial em código, um ponto (X,Y,Z) representado com 3 floats.

Nesta fase foi acrescentado um método que permite obter a normal num vetor formado pela origem ao ponto.

```
#ifndef GENERATOR_POINT_H
#define GENERATOR_POINT_H

#include <math.h>
#include <string>

using namespace std;

class Point {
    float x;
    float y;
    float z;

    public:
        Point();
        Point(float, float, float);
        float getX();
        float getY();
        float getZ();
        void setX(float);
        void setY(float);
        void setZ(float);
        static Point* getNormal(Point* p);

};


#endif //GENERATOR_POINT_H
```

Figura 4: poin.h

2.2.3 Group

Classe cuja função é armazenar toda a informação referente a um determinado grupo. Esta é utilizada aquando da leitura dos ficheiros XML, na medida em que a cada grupo lido e interpretado, corresponde um objeto com informações relativas às transformações a que os modelos são sujeitos e ainda às mudanças de cor destes. Poderá também conter os grupos filhos de um determinado grupo, sendo estes grupos contidos neste grupo.

Na presente fase foi adicionada informação sobre a iluminação de cada grupo, bem como métodos de obtenção da mesma.

```
#ifndef GENERATOR_GROUP_H
#define GENERATOR_GROUP_H

#include <vector>
#include "figure.h"
#include "transforms.h"
#include "light.h"


class Group {
    std::vector<Transform*> transforms;
    std::vector<Figure*> figures;
    std::vector<Group*> childs;
    std::vector<Light*> lights;
    float R, G, B;

    public:
        Group();
        Group(std::vector<Transform*>, std::vector<Figure*>, std::vector<Group*>, std::vector<Light*>);
        std::vector<Transform*> getTransforms();
        std::vector<Figure*> getFigures();
        std::vector<Group*> getChilds();
        std::vector<Light*> getLights();

        void pushTransform(Transform* t);
        void pushFigure(Figure* f);
        void pushGroup(Group* g);
        void pushLight(Light* l);

        float getR();
        float getG();
        float getB();
        void setRGB(float,float,float);
};

#endif //GENERATOR_GROUP_H
```

Figura 5: group.h

2.2.4 Parser

Classe responsável por dar parse, com o auxílio do *tinyxml2*, aos ficheiros que servem de input ao engine, e que utiliza como forma de guardar os dados a classe group, figure e transforms.

Com a implementação da iluminação foi também necessário guardar informação sobre as normais de cada primitiva geométrica a ser representada, para uso posterior.

```
#ifndef GENERATOR_PARSER_H
#define GENERATOR_PARSER_H

#include "tinyxml2.h"
#include <string>
#include <regex>
#include <fstream>
#include <sstream>
#include "point.h"
#include <iostream>
#include "group.h"


int readXML(string file, Group* group, vector<Translation*>* orbits);

#endif //GENERATOR_PARSER_H
```

Figura 6: parser.h

2.2.5 Figure

Esta é a classe responsável por armazenar os dados relativos a cada primitiva geométrica, cada figura a representar, sendo estes posteriormente utilizados pelo *engine* para desenhar as figuras.

Sendo preciso acrescentar iluminação e texturas, esta foi uma classe fundamental a alterar e, por isso, foram acrescentadas várias variáveis fundamentais para a realização desta tarefa.

```
#ifndef GENERATOR_FIGURES_H
#define GENERATOR_FIGURES_H

#include <vector>
#include "point.h"
#ifdef __APPLE__
#define GL_SILENCE_DEPRECATION
#include <GLUT/glut.h>
#include "/usr/local/include/IL/il.h"
#else
#include <GL/glew.h>
#include <GL/glut.h>
#include <IL/il.h>
#endif

class Figure{
    private:
        float buffer_size[3];
        GLuint buffer1,buffer2,buffer3,texture;
        float ambient[4];
        float diffuse[4];
        float specular[4];
        float emissive[4];
        float shin;

    public:
        Figure();
        void setUp(vector<Point*>, vector<Point*>,vector<Point*>);
        void loadTexture(string file);
        void addMaterials(float diff[4], float spec[4], float emiss[4], float amb[4], float shine);
        void draw();
};

#endif //GENERATOR_FIGURES_H
```

Figura 7: figure.h

2.2.6 Bezier

Esta classe é responsável por armazenar informação sobre as curvas e patches de bezier a implementar para desenho das primitivas.

Nesta fase foram adicionados métodos para a obtenção de normais devido à utilização da iluminação.

```
#ifndef GENERATOR_BEZIER_H
#define GENERATOR_BEZIER_H

#include "point.h"
#include <fstream>
#include <string>
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

class Patch{
    vector<Point*> controlPoints;

public:
    Patch();
    Patch(vector<Point*>);
    vector<Point*> getCPoints();
    void addPoint(Point*);
};

int bezierParser(int tess,string filePath, string nameFile);

#endif
```

Figura 8: Bezier

3 Generator

3.1 Aplicação de normais e pontos de textura

Para a obtenção das normais e dos pontos de textura para as diferentes figuras geométricas foi necessário estudar as faces e vértices que as constituem.

O estudo das normais passa por conseguir obter um vetor normal para cada vértice (perpendicular a este), que constitui a figura geométrica.

Quanto aos pontos de textura, foi necessário estudar o mapeamento de um plano 2D (imagem) para as figuras geométricas 3D, servindo como revestimento das mesmas.

3.1.1 Plano

Para obter o conjunto dos vetores normais, de cada vértice, foi unicamente necessário verificar o plano cartesiano em que este plano geométrico é desenhado. Como desenhamos este no plano x0z, todos os vértices possuirão a mesma normal, como tal todos os vértices partilham como normal o vetor (0,1,0).

O processo de obtenção dos pontos de textura é simples no caso desta figura geométrica. Como é de esperar, o formato é o mesmo que o imagem 2D, sendo apenas necessário fazer a correspondência direta de cada vértice do plano com vértices da imagem 2D.

3.1.2 Box

Tal como foi feito para o plano, para obter o conjunto de vetores normais para cada vértice, é indispensável verificar o plano cartesiano de cada face da caixa.

Assim, facilmente reconhecemos que os correspondentes vetores normais a cada uma das suas faces são:

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

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

O cálculo dos pontos de textura passa por obter a posição da imagem a que corresponde à face em questão e iterar no mesmo sentido que a box é desenhada, atribuindo a cada vértice o ponto de textura correspondente.

Para tal usamos as seguintes equações matemáticas:

- textureY1 = c / (c * 2 + b)
- textureY2 = (c + b) / ((c * 2) + b)
- textureX1 = c / ((c * 2) + (a * 2))
- textureX2 = (c + a) / ((c * 2) + (x * 2))
- textureX3 = ((c * 2) + a) / ((c * 2) + (a * 2))
- textShiftX = (a / ((c * 2) + (a * 2))) / float(divisoes)
- textShiftY = (c / ((c * 2) + b)) / float(divisoes)
- textShiftZ = (c / ((c * 2) + (a * 2))) / float(divisoes)

Onde *a*, *b* e *c* são os valores introduzidos para definir o tamanho da box e divisoes o número de divisões opcional da box.

Tais equações são usadas nos cálculos dos pontos da seguinte forma:

- (textureX1 + (j * textShiftX), 1 - (i * textShiftZ), 0
- (textureX1 + (j * textShiftX), (1 - textShiftZ) - (i * textShiftZ), 0)
- (textureX1 + textShiftX) + (j * textShiftX), 1 - (i * textShiftZ), 0)
- (textureX1 + textShiftX) + (j * textShiftX), (1 - textShiftZ) - (i * textShiftZ), 0)

3.1.3 Esfera

Tratando-se de uma esfera, notamos desde logo que todos os pontos desenhados, além de terem a mesma distância à origem, coincidem também com a direção do vetor da normal a cada "face"desenhada. Assim, é preciso apenas e só fazer a normalização de cada vetor e acrescentar, assim, ao destinado vetor de normais.

Desta forma, ao adicionar o ponto

$$\{x, y, z\}$$

, e sendo

$$l = sqrt(x^2 + y^2 + z^2)$$

, podemos também adicionar ao vetor das normais o valor

$$\{x/l, y/l, z/l\}$$

No cálculo das **texturas** são calculados 2 valores: *textureU = 1/(slices/2)* e *textureV = 1/stacks*. Assim, a cada iteração, conseguimos saber em que momento do "desenho"estamos, e conseguimos portanto mapear a posição de cada vértice para a posição do pixel de uma imagem.

Consequentemente, a cada vértice inserido no vetor, a componente X e Y são dadas respetivamente por **textureV*j** e **(1+textureU*i)/2**, sendo j o iterador pelas stacks e i o iterador pelas slices.

Como os valores tanto do *textureU* ou do *textureV* nunca passam de 1, o valor da componente X de cada vértice nas texturas vai ser dado pelo valor calculado (*textureV*j*). No entanto o valor da componente Y, como -1 < textureU < 1, é preciso converter para um valor com textureU $\in$ [0,1], ficando finalmente com : **(1+textureU*i)/2**.

3.1.4 Torus

Para obter os vetores normais do torus é necessário estudar o processo de desenho do mesmo. Assim como a esfera, o processo de desenho do torus dá-nos as orientações dos vetores normais de cada vértice no momento do desenho deste. Como apenas nos interessa a orientação da origem até cada vértice, os vetores normais são dados pelo cálculo dessa orientação.

Para obter os vetores normais simplesmente fazemos uso do valor do desvio do anel (DL) e do desvio de cada lado que forma o anel (DA) chegando à seguinte expressão :

- N(cos(DA)*cos(DL), sin(DA)*cos(DL), sin(DL))

Desta forma, numa iteração sobre todos os pontos conseguimos obter os seus vetores normais.

Quanto aos pontos de textura, o mapeamento ocorre de forma simples, isto é, sabendo que a imagem padrão para textura do torus é, um retângulo, para o revestir, atribuímos uma tira da imagem 2D e envolvemos a mesma num anel do torus.

Ao aplicar este processo de forma iterativa conseguimos revestir o torus na sua totalidade.

A expressão utilizada para o cálculo dos pontos de textura é:

- T(Número do Anel / slice, Número do Lado / sides)

3.1.5 Bezier Patches

Para a obtenção dos vetores normais de um teapot (neste caso), decidimos seguir uma abordagem diferente. Ao invés de calcular as derivadas parciais de **B**, decidimos que:

- sejam *a,b* e *c* pontos pertencentes a um triângulo, desenhado de forma alfabética

  $$a- > b- > c$$

  , então conseguimos obter *Vab* = *b-a* e *Vac* = *c-a*. Ora, sabendo que, o cross de 2 vetores origina um vetor **perpendicular** aos 2, então conseguimos obter um vetor normal por : *Vnormal* = *Vab x Vac*, normalizando-o de seguida.

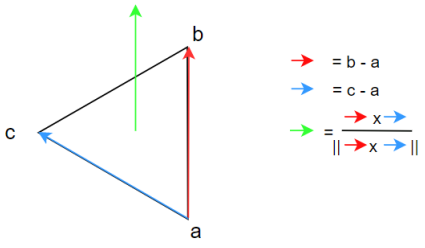

Figura 9: cross product

4 Engine

Na presente secção abordaremos os temas relativos ao *engine* que sofreram alterações desde a fase passada, sendo que as mesmas não foram tão significativas quanto em fases anteriores. Não obstante é sempre necessário mencionar as mesmas.

4.1 Menu

Relativamente ao Menu, decidimos enriquecer as possibilidades oferecidas ao utilizador de manipular o cenário em alguns sentidos tais como o modo de apresentação, a câmara que se encontra a utilizar e detalhes visuais.

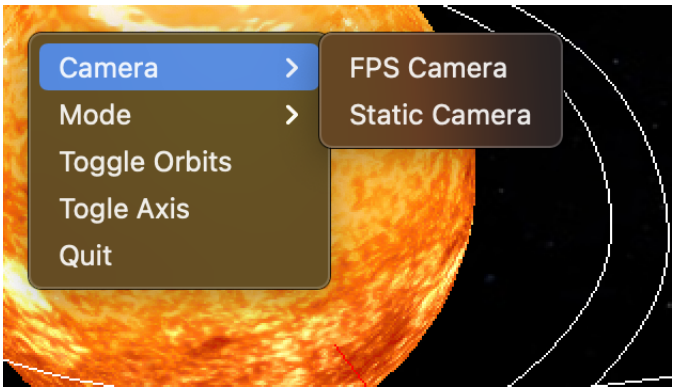

Figura 10: Menu De Auxílio

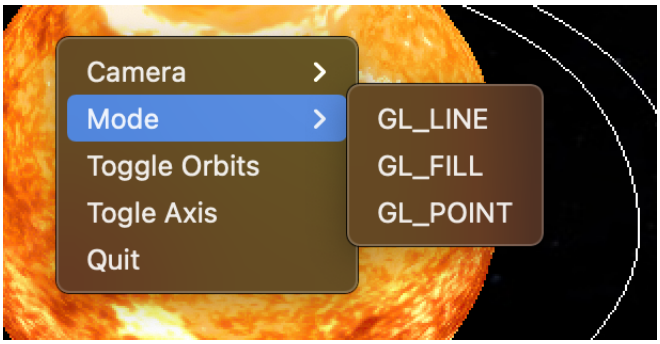

Figura 11: Menu de Auxilio

4.2 Renderização

O processo de renderização manteve-se basicamente idêntico ao que se encontra na fase anterior, sendo apenas efetuadas algumas alterações.

O principal método de renderização é o *renderScene()*, sendo através do mesmo que fazemos as devidas inicializações de modo a poder carregar e apresentar os modelos pretendidos.

Relativamente às fases anteriores foi somente adicionada a inicialização das bibliotecas necessárias para a iluminação e texturas.

5 Resultados Obtidos

O resultado final do Sistema Solar dinâmico desenvolvido em todas as fazes realizadas para a Unidade Curricular Computação Gráfica, correspondeu às expectativas do grupo sabendo que: todos os planetas foram representados o mais perto da escala possível, os seus movimentos de rotação e translação foram também aproximados ao máximo da realidade e as texturas dos planetas e respetivas luas foram selecionada de forma a tornar o Sistema Solar o mais próximo possível da realidade.

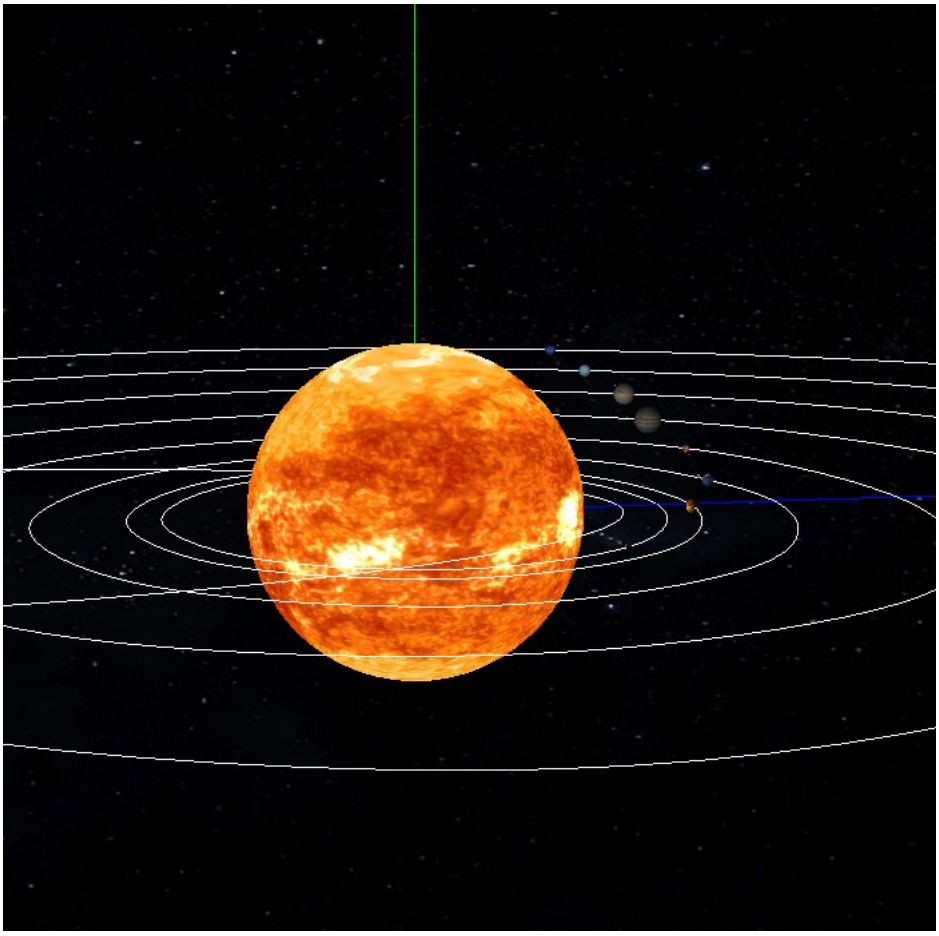

Figura 12: Representação do Sistema Solar - Exemplo 1

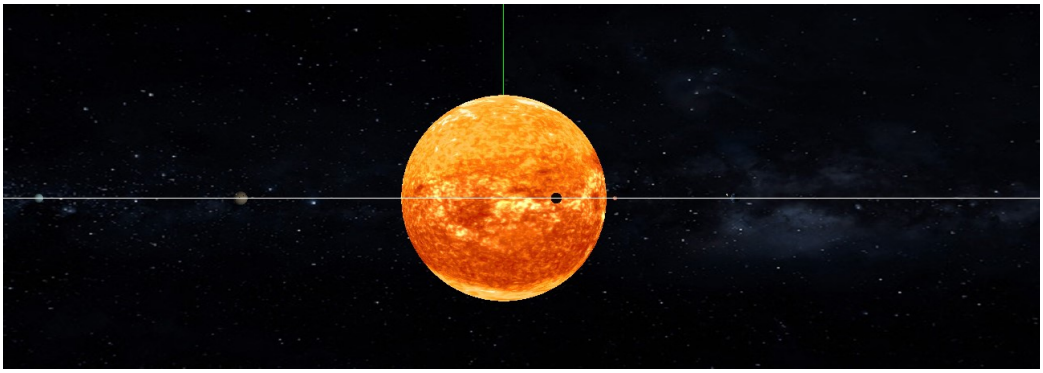

Figura 13: Representação do Sistema Solar - Exemplo 2

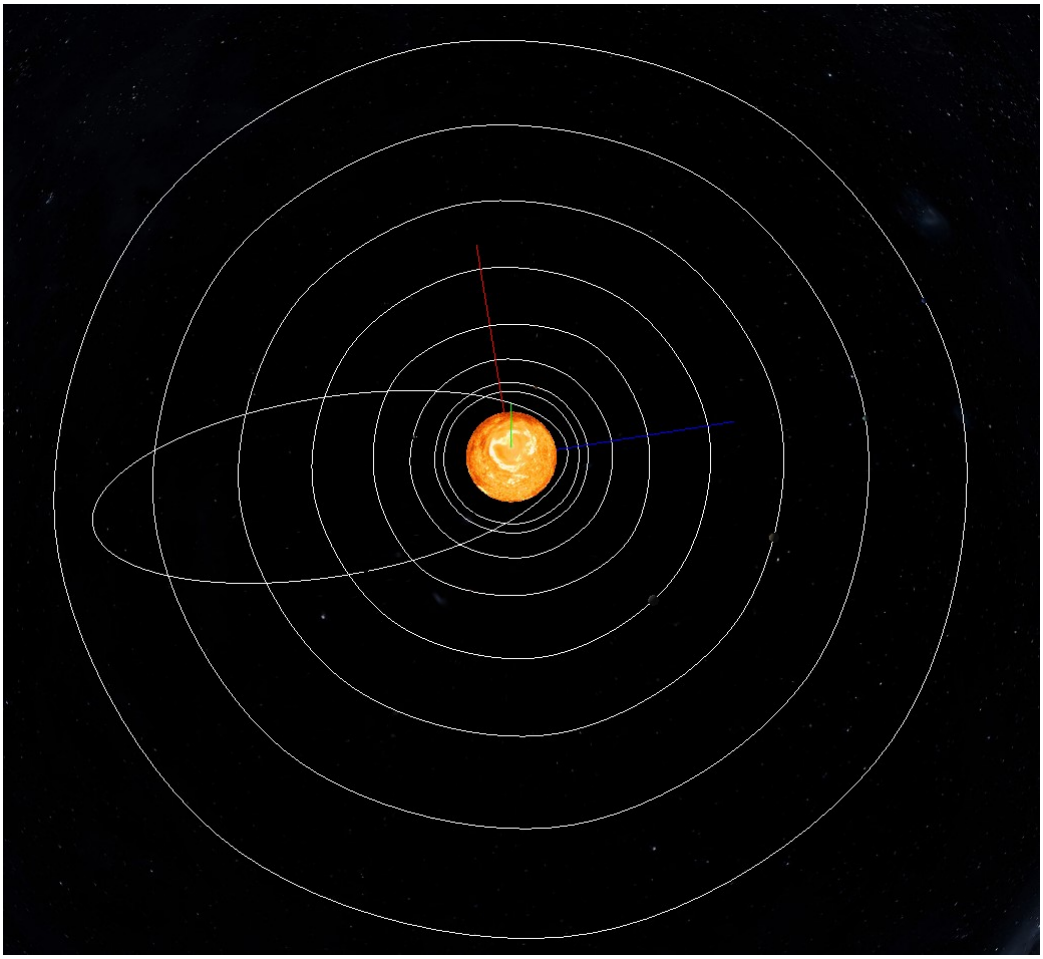

Figura 14: Representação do Sistema Solar - Exemplo 3

6 Conclusão

A elaboração desta última fase do trabalho foi um pouco menos trabalhosa, em relação às anteriores, uma vez que, não só o facto dos requisitos necessários serem menores, mas também as bases que fomos desenvolvendo durante a execução das outras fases que, de certa forma, nos ajudaram a ultrapassar algumas dificuldades que nos foram surgindo, visto já não sermos tão inexperientes na matéria.

Durante a elaboração desta fase deparamo-nos com algumas dificuldades que foram ultrapassadas com alguma dedicação e empenho por parte do grupo.

Assim, pensamos que o resultado final desta fase e, portanto, do projeto no seu todo, corresponde às expectativas, na medida em que conseguimos desenvolver um modelo deveras realista e visualmente apelativo do Sistema Solar, tal como nos era inicialmente pedido.

Desta forma, depois de tudo o que foi realizado durante as várias fases do trabalho, consideramos que o nosso conhecimento no que diz respeito à utilização do OpenGL e do GLUT cresceu exponencialmente, podendo então afirmar que nos consideramos deveras aptos e com bases sólidas para desenvolver eventuais projetos futuros.