UNIVERSIDADE DO MINHO
MIEI

COMPUTAÇÃO GRÁFICA

Quarta Fase
Normals and Texture Coordinates

Grupo 33:
João Nunes - a82300
Shahzod Yusupov - a82617
Luís Braga - a82088
Luís Martins - a82298

Braga, Portugal
18 de Maio de 2019

Conteúdo

1 Introdução

Na quarta fase do projeto prático da UC de Computação Gráfica, foi proposta uma realização de uma cena gráfica 3D, em que foi proposta uma pequena demonstração de algumas funcionalidades propostas, tais como, a utilização de texturas, materiais e iluminação dos objetos.

De modo a demonstrar estas funcionalidades desenvolvidas, utilizou-se como prototipo o Sistema Solar, que se tem vindo a desenvolver nas fases anteriores.

2 Estrutura do Projecto

Tratando-se da última parte do projeto prático, é normal que algumas das funcionalidades desenvolvidas nas fases anteriores se mantenham inalteradas,enquanto outras não, de forma a cumprir os requisitos necessários.

Relativamente ao *generator*, este passará a ser capaz de conseguir obter as normais e coordenadas de textura para os vários vértices das primitivas geométricas anteriormente criadas.

O *engine*, por outro lado,para além de sofrer algumas alterações, receberá também novas funcionalidades. Os ficheiros XML passarão a conter as informações relativas à iluminação do cenário e o parser responsável pela leitura dos ficheiros assim como o modo como é processada toda a informação recebida com o intuito de gerar todo o cenário pretendido.

Por fim, a última modificação que o engine irá sofrer, está relacionada com a preparação dos VBOs e das texturas durante o processo de leitura de informação proveniente do ficheiro de configuração.

O propósito destas alterações é a geração eficaz de um Sistema Solar ainda mais realista, possuindo agora texturas e iluminação necessária.

Portanto, adivinha-se que com estas alterações seja possível gerar um sistema solar mais fidedigno e dinâmico em relação à realidade.

2.1 Aplicações Principais

2.1.1 Gerador

O gerador, tal como nas outras fases, é responsável por gerar os pontos necessários para representar uma figura, tal como, um plano, uma caixa, cone, esfera e o torus ou até mesmo um patch de bezier.

O gerador também guarda os pontos gerados num ficheiro 3d passado como argumento, podendo esses ser apenas os pontos necessários para formar a figura ou os pontos necessários para representar as normais ou pontos de textura.

2.1.2 Engine

É no *engine* que se encontram as funcionalidades relativas ao desenho de figuras/modelos. É através deste programa que as figuras são desenhadas e apresentadas numa janela e ,é também com ele, que podemos interagir com as mesmas.

Fizeram-se várias alterações neste módulo de modo a cumprir os objetivos desta fase. As alterações foram as seguintes: no ficheiro XML t alteraram-se/criaram-se novas classes listadas na seguinte secção.

2.2 Restantes Classes

Para além das duas classes principais, ou seja, o Gerador e o Engine, foram também adicionadas e modificadas algumas classes adicionais. Como tal, e em forma de sumarizar todo o trabalho elaborado até agora, serão apresentadas todas as classes adicionais criadas.

2.2.1 Point

```
#ifndef POINT_H__
#define POINT_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);
    void calcula_Normal();
    string to_String();
};

#endif
```

A classe Point foi elaborada com o intuito de representar um ponto segundo as coordenadas **(X,Y,Z)**, a classe portanto possui os construtores e os métodos *standard*, para além de uma função responsável por calcular a normal de um ponto, segundo o seguinte cálculo:

$$normal = \sqrt{x^2 + y^2 + z^2}$$
$$x = x/normal$$
$$y = y/normal$$
$$z = z/normal$$

Onde, primeiramente calcula-se a norma do ponto, e de seguida, de modo a normalizar o ponto divide-se a coordenada do ponto pelo valor da norma.

2.2.2 Struct

A classe Struct proporciona uma estrutura capaz de gerar e armazenar os pontos de modo a possibilitar o desenho das figuras primitivas. Como tal, foi necessário criar três arrays distintos,

um cujo intuito é guardar os pontos necessários para gerar a figura pretendida, de seguida um array para guardar os pontos necessários para construir as normais e outro para guardar os pontos de modo a construir as texturas para as figuras.

Portanto, para além de para cada figura apenas gerar os pontos necessários para a desenhar, agora envolve cálculos adicionais de modo a gerar os vetores normais a cada ponto da figura, de modo a aplicar a luz, e os pontos da textura.

```
#ifndef STRUCT_H__
#define STRUCT_H__

#define _USE_MATH_DEFINES

#include <math.h>
#include <iostream>
#include <vector>
#include "Point.h"

using namespace std;

class Struct {
  vector<Point*> LP;        // lista de pontos
  vector<Point*> normal;    // lista de pontos das normais
  vector<Point*> textura;   // lista de pontos das texturas

public:
  Struct();
  Struct(vector<Point*>, vector<Point*>, vector<Point*>);
  vector<Point*> getLP();
  vector<Point*> getNormal();
  vector<Point*> getTextura();
  void setLP(vector<Point*>);
  void setNormal(vector<Point*>);
  void setTextura(vector<Point*>);
  void genPlane(float);
  void genCylinder(float, float, int);
  void genSphere(float, int, int);
  void genCone(float , float , int , int );
  void genBox(float, float, float, int);
  void genTorus(float, float, int, int);
  void genCintura(float, float, int, int);
};

#endif
```

2.2.3 Bezier Patch

Esta classe, possui o intuito de proporcionar o conjunto de métodos necessários para gerar um patch de bezier, sendo que primeiramente é necessário calcular os pontos de modo a poder desenhar este mesmo patch.

```
#ifndef __BEZIERPATCH_H__
#define __BEZIERPATCH_H__

#include <vector>
#include "../../src/headers/Point.h"

```

```
using namespace std;

class BezierPatch {

  public:

    void Bezierpatch(int tesselacao, string input, string output);
    void BezierCurve(std::vector<Point> *vertices, std::vector<int> patch,
    std::vector<Point *> points, float u, float v, float intervalo);
    Point calculaPontos(std::vector<int> patch, std::vector<Point *> pontos,
    float u, float v);
};

#endif
```

2.2.4 Câmera

A classe câmera foi criada de modo a guardar toda a informação relativa à câmera e o seu respetivo movimento. Como tal, existem três vetores distintos, o primeiro destina-se a saber a localização da câmera, o segundo a posição para onde a câmera está a apontar, e o último a inclinação desta mesma. O *alfa* e o *beta*, representam os ângulos que são usados para determinar a posição da câmera. Por último, existe o *dist* que indica a distância entre a câmera e o ponto para o qual está a apontar.

```
#ifndef __CAMERA_H__
#define __CAMERA_H__

#include "../../../src/headers/Point.h"
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

class Camera {
  Point* camPosition;
  Point* lookPoint;
  Point* titl;
  float alfa;
  float beta;
  float dist;

public:
  Camera();
  void atualizaCamPosition();
  void atualizaDist();
  void camUp();
  void camDown();
  void camLeft();
  void camRight();
  void FocusUp();
  void FocusDown();
  void FocusLeft();
  void FocusRight();
  void maisZoom();
  void menosZoom();
  Point* getCamPosition();
```

```
33    Point* getLookPoint();
34    Point* getTitl();
35    void setCamPosition(float, float, float);
36    void setLookPoint(float, float, float);
37    void setTitl(float, float, float);
38
39 };
40
41 #endif
```

2.2.5 Colour

A classe colour é utilizada para atribuir cores aos objetos desenhados. Portanto, foram criados três floats, *rr*, *gg* e *bb* que indicam a primazia da coloração sobre o qual atribuirá ao objeto segundo as tonalidades de vermelho, verde e azul. Portanto estes três parâmetros representam o sistema de cores aditivas *RGB* em que estas três cores são combinadas de modo a reproduzir um largo espetro cromático.

```
1 #ifndef __COLOUR_H__
2 #define __COLOUR_H__
3
4 class Colour {
5   float rr;
6   float gg;
7   float bb;
8
9 public:
10   Colour();
11   Colour(float, float, float);
12   float getRR();
13   float getGG();
14   float getBB();
15   void setRR(float);
16   void setGG(float);
17   void setBB(float);
18 };
19
20 #endif
```

2.2.6 Group

Esta classe é responsável pelo armazenamento de toda a informação necessária à representação de um dado modelo.Sendo lida aquando da leitura dos ficheiros input,em XML,a cada modelo lido e interpretado corresponderá um objeto Group com informação relativa ao seu identificador*(id)*, às várias operações de rotação *(Rotate* rotation)* ,translação *(Translate* translation)* e escala *(*Scale* scale) associadas ao mesmo e também uma componente encarregue pelas cores produzidas através da iluminação *(Colour* colour)*. Por fim,apresentará também um vetor com filhos do tipo Group *(vector<Group*> childs)*, importante para estabelecer uma hierarquia no ficheiro de configuracção.

```
1
2 #ifndef __GROUP_H__
3 #define __GROUP_H__
4
```

```
#include <vector>
#include "scale.h"
#include "colour.h"
#include "pontos.h"
#include "rotate.h"
#include "translate.h"
#include "../../../src/headers/Point.h"

using namespace std;

class Group {
  int id;
  Rotate* rotation;       // rota   o associada ao group
  Translate* translation;     // transla   o associada ao group
  Scale* scale;          // escala associada ao group
  Colour* colour;          // cor associada ao group
  pontos LP;             // pontos a desenhar associado ao group
  vector<Group*> childs;       // vetor com filhos do tipo Group associado ao group

public:
  Group();
  Group(int);
  int getID();
  Rotate* getRotation();
  Translate* getTranslation();
  Scale* getScale();
  Colour* Group::getColour();
  pontos getLP();
  vector<Group*> getChilds();
  void setRotation(Rotate*);
  void setTranslation(Translate*);
  void setScale(Scale*);
  void setColour(Colour*);
  void setLP(vector<Point*>);
  void addChild(Group*);

};


#endif
```

2.2.7 Parser

A classe parser foi criada com o objetivo de ler os ficheiros XML presentes no projeto.

```

#ifndef __PARSER_H__
#define __PARSER_H__

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

using namespace tinyxml2;
```

```
13
14 void readFile(string, vector<Point*>*);
15 int readXML(string, vector<Group*>*);
16
17
18 #endif
```

2.2.8 Pontos

A classe Pontos foi criada com o intuito de armazenar e tratar das coordenadas, normais e texturas usadas por VBO's.

```
1  #ifndef __PONTOS_H__
2  #define __PONTOS_H__
3
4  #include <vector>
5  #include <string>
6  #include <stdlib.h>
7  #include <GL/glew.h>
8  #include <GL/glut.h>
9  #include <IL/il.h>
10 #include "../../../src/headers/Point.h"
11 #include "headers/material.h"
12
13 using namespace std;
14
15 class pontos{
16   GLuint buffer[3], size_buffer[3];
17   Material* colour;
18   GLuint texture;
19
20
21 public:
22
23   pontos();
24   pontos(string s, vector<Point*>, vector<Point*> , vector<Point*>);
25   Material* getColour();
26   void setColour(Material*);
27   void loadTexture(string s);
28   void prepare(vector<Point*>, vector<Point*>, vector<Point*>);
29   void draw();
30
31 };
32
33 #endif
```

2.2.9 Rotate

Classe usada para lidar com as rotações relativas aos objectos. Nesta classe são armazenadas informações importantes à execução de uma rotação, sendo assim, usamos uma variável tempo que guarda a quantidade de tempo necessário para percorrer uma rotação de 360º sobre o eixo. As restantes variáveis (x, y, z) são usadas para guardar o eixo sobre o qual a rotação irá decorrer.

```
#ifndef __ROTATE_H__
#define __ROTATE_H__

#include <GL/glut.h>

class Rotate {
  float time;
  float x;
  float y;
  float z;

public:
  Rotate();
  Rotate(float, float, float, float);
  float getTime();
  float getX();
  float getY();
  float getZ();
  void setTime(float);
  void setX(float);
  void setY(float);
  void setZ(float);
  void apply();
};

#endif
```

2.2.10 Scale

A classe Scale foi construída para armazenar informação relativas à execução de um redimensionamento, para tal, foram necessárias três variáveis que representam as dimensões cobre cada um dos diferentes eixos.

```
#ifndef __SCALE_H__
#define __SCALE_H__

class Scale {
  float x;
  float y;
  float z;

public:
  Scale();
  Scale(float, float, float);
  float getX();
  float getY();
  float getZ();
  void setX(float);
  void setY(float);
  void setZ(float);
};

#endif
```

2.2.11 Translate

Classe usada para translações feitas às figuras, sendo que para tal existe um array que armazena os pontos necessários para efetuar um vetor de translação, com o intuito de aplicar a tal transação, e um array com os pontos necessários para definir uma curva devido ao *Catmull-Rom* (de modo a se poder aplicar no modelo do sistema solar). Para além destes vetores de aplicação, existe também a variável tempo, que é aplicada sobre a curva, dando o número de segundos necessários para percorrer toda a curva.

```
#ifndef __TRANSLATE_H__
#define __TRANSLATE_H__

#include<vector>
#include "../../../src/headers/Point.h"
#include <GL/glut.h>

class Translate {
  float tempo;
  vector<Point*> trans;
  vector<Point*> curve;

public:
  Translate();
  Translate(float, vector<Point*>);
  float getTempo();
  vector<Point*> getCurve();
  vector<Point*> getTrans();
  void setTempo(float);
  void setTrans(vector<Point*>);
  void setCurve(vector<Point*>);
  void getCatmullRomPoint(float, int*, float*, vector<Point*>);
  void getGlobalCatmullRomPoint(float, float *, vector<Point*>);
  void drawCurve();
  void drawCatmullRomCurve();
  void draw();

};

#endif
```

2.2.12 Light

Uma vez que uma das funcionalidades base desta fase do projeto é a iluminação de uma cena montada pelo grupo (sistema solar), foi necessária a criação de uma classe que armazenasse toda a informação relativa à iluminação, neste caso, a origem da luz. Existem duas variáveis, uma que indica o tipo da luz aplicada, ou seja uma que indica se é aplicada num ponto, se é direcional ou spot, sendo para tal utilizada a variável point. É importante referir contudo que o grupo optou por apenas utilizar a luz aplicada ao ponto. A segunda variável guarda a posição da luz segundo o **(X,Y,Z)**. As restantes funções elaboradas limitam-se aos métodos *standard* tais como os *gets* e *sets* e os construtores.

```
#ifndef __LIGHT_H__
#define __LIGHT_H__

#include <GL/glut.h>
```

```
#include "../../../src/headers/Point.h"


class Light {
  bool point;           // 1 ->    ponto
  Point* p;       // posX, posY, posZ

public:
  Light();
  Light(bool, Point*);
  bool getType();
  Point* getPoint();
  void setType(bool type);
  void setPoint(Point* p);
  void draw();

};
#endif
```

2.2.13 Material

Para esta fase do projeto também é necessário aplicar materiais as figuras desenhadas, onde estes são aplicados consoante a iluminação sobre eles, mudando portante a reflexão que estes possuirão. Portanto, existem 4 tipos de materiais base, *diffuse*, *specular*, *emission* e *ambient* onde é necessário aplicar os gradientes existentes no *RGBA*. A última variável está reservada para o brilho do objeto. Portanto, estes materiais são aplicados à figura pretendida.

```
#ifndef __MATERIAL_H__
#define __MATERIAL_H__

#include <GL/glut.h>
#include "../../../src/headers/Point.h"

class Material {
  float diffuse[4];             // por omissao {0.8,0.8,0.8,1}
  float specular[4];        // por omissao {0,0,0,1}
  float emission[4];        // por omissao {0,0,0,1}
  float ambient[4];       // por omissao {0.2,0.2,0.2,1}
  float shini;            // por omissao 0

public:
  Material();
  Material(Point* diff, Point* specular, Point* emission,
  Point* ambient, float shin);
  void draw();

};

#endif
```

3 Generator

3.1 Aplicação de normais e pontos de textura

De maneira a obter as normais e os pontos de textura para as várias figuras geométricas desenvolvidas ao longo do projeto, o estudo das faces e os respetivos vértices que fazem parte da sua constituição, foi de elevada importância.

Nas secções seguintes serão abordados os raciocínios por detrás da geração das normais e texturas.

3.1.1 Plano

No desenho de um plano, espera-se que o vetor normal seja (0,1,0), ou seja, que tenha uma direção no sentido positivo do eixo dos Y.

Em relação às texturas, será apenas necessário fazer a correspondência direta de cada vértice do plano,ou seja, o valor h corresponde a 1 e o valor -h corresponde ao valor 0. Assim sendo, é possível percorrer toda a textura através de dois triângulos e aplicá-la corretamente ao plano.

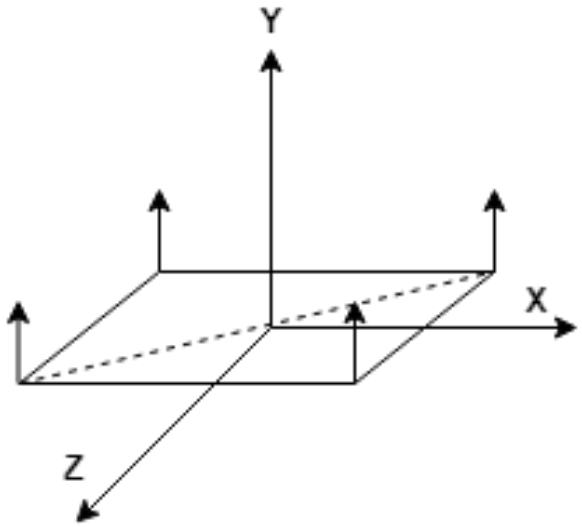

Figura 3.1: Normal do plano.

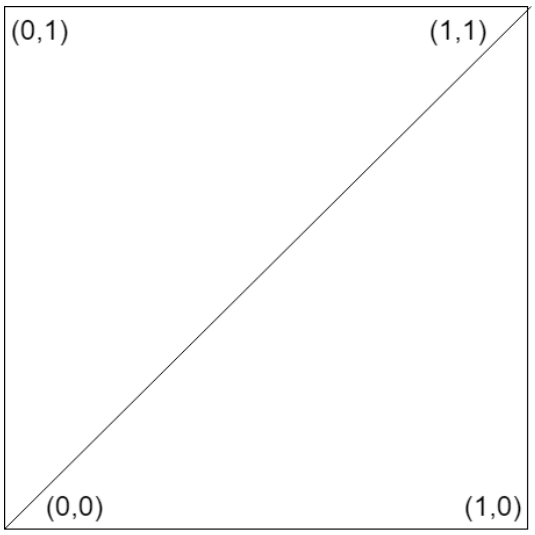

Figura 3.2: Cálculo das texturas.

```
1
2  void Struct::genPlane(float size)
3  {
4    float h = size/2;
5
6    LP.push_back(new Point(-h, 0, h));
7    LP.push_back(new Point(h, 0, h));
8    LP.push_back(new Point(-h, 0, -h));
9    LP.push_back(new Point(-h, 0, -h));
10   LP.push_back(new Point(h, 0, h));
11   LP.push_back(new Point(h, 0, -h));
12
13   normal.push_back(new Point(0, 1, 0));
14   normal.push_back(new Point(0, 1, 0));
15   normal.push_back(new Point(0, 1, 0));
16   normal.push_back(new Point(0, 1, 0));
17   normal.push_back(new Point(0, 1, 0));
18   normal.push_back(new Point(0, 1, 0));
19
20   textura.push_back(new Point(0, 1, 0));
21   textura.push_back(new Point(1, 1, 0));
22   textura.push_back(new Point(0, 0, 0));
23   textura.push_back(new Point(0, 0, 0));
24   textura.push_back(new Point(1, 1, 0));
25   textura.push_back(new Point(1, 0, 0));
26 }
```

3.1.2 Box

Na geração de uma box foi necessário ter em conta uma textura desdobrada desta figura:

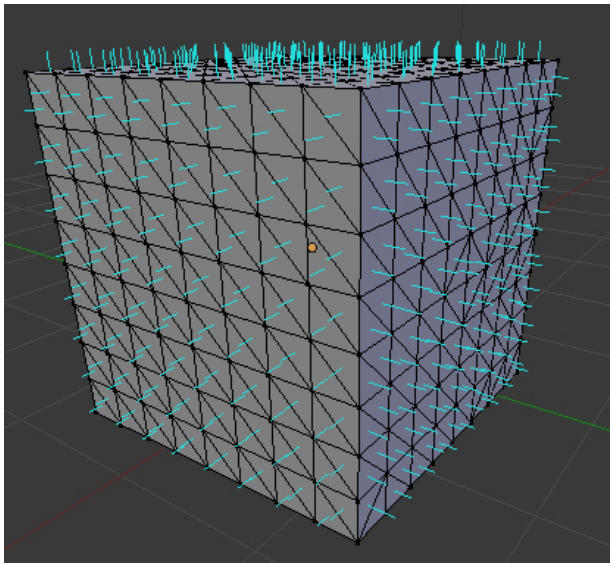

Figura 3.3: Normal da box.

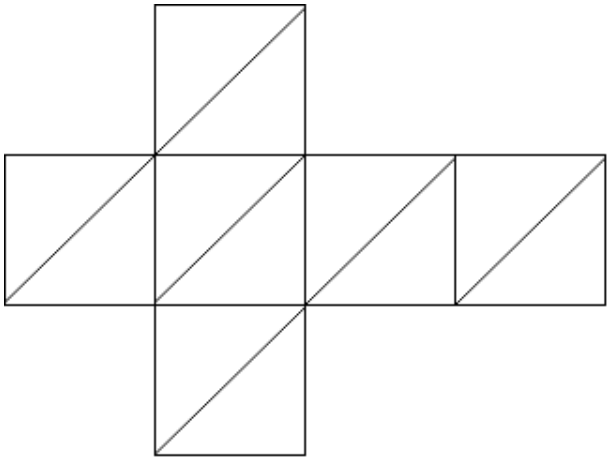

Figura 3.4: Planificação da box.

Assim,podemos verificar que se teve em conta 5 pontos base (texX1, texX2, texX3, texY1, texY2),essenciais para identificar as várias faces ao longo da construção da figura.

As divisões de cada aresta são representadas pelas variáveis texDivX, texDivY, texDivZ e a escolha destas é feita dependendo da medida da aresta a representar.

Assim obtemos o seguinte código:

```
1
2  void Struct::genBox(float cX, float cY, float cZ, int div){
3
4      float x = cX/2;
5      float y = cY/2;
6      float z = cZ/2;
7
8      float pX = (float)cX/div;
9      float pY = (float)cY/div;
10     float pZ = (float)cZ/div;
```

```
11
12  // Calculo dos pontos que identificam as faces da textura
13  float texY1 = cZ / ((cZ * 2) + cY);
14  float texY2 = (cZ + cY) / ((cZ * 2) + cY);
15
16  float texX1 = (cZ) / ((cZ * 2) + (cX * 2));
17  float texX2 = (cZ + x) / ((cZ * 2) + (cX * 2));
18  float texX3 = ((cZ * 2) + cX) / ((cZ * 2) + (cX * 2));
19
20  // Calculo das divisoes da textura
21  float texDivX = (cX / ((cZ * 2) + (cX * 2))) / float(div);
22  float texDivY = (cY / ((cZ * 2) + cY)) / float(div);
23  float texDivZ = (cZ / ((cZ * 2) + (cX * 2))) / float(div);
24
25
26    for(int i=0;i<div;i++){
27        for(int j=0;j<div;j++){
28            //Face da frente
29            LP.push_back(new Point(-x + (j*pX),-y + (i*pY),z));
30      normal.push_back(new Point(0, 0, 1));
31            LP.push_back(new Point((-x+pX) + (j*pX),-y + (i*pY),z));
32      normal.push_back(new Point(0, 0, 1));
33            LP.push_back(new Point(-x + (j*pX),(-y+pY) + (i*pY),z));
34      normal.push_back(new Point(0, 0, 1));
35
36
37            LP.push_back(new Point(-x + (j*pX),(-y+pY) + (i*pY),z));
38      normal.push_back(new Point(0, 0, 1));
39            LP.push_back(new Point((-x+pX) + (j*pX),-y + (i*pY),z));
40      normal.push_back(new Point(0, 0, 1));
41            LP.push_back(new Point((-x+pX) + (j*pX),(-y+pY) + (i*pY),z));
42      normal.push_back(new Point(0, 0, 1));
43
44        // Comecar na posicao de texX1 e ir aumentando um
45        //"passo" de cada vez.Por outro lado comeca - se
46        //  no texY2 e diminui - se um "passo"
47
48        textura.push_back(new Point(texX1 + (j*texDivX), texY2 - (i*texDivY), 0));
49        textura.push_back(new Point(texX1 + (j*texDivX), (texY2 - texDivY)
50       - (i*texDivY), 0));
51        textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
52        (texY2 - texDivY) - (i*texDivY), 0));
53
54        // Triangulo complementar ao anterior
55        textura.push_back(new Point(texX1 + (j*texDivX), texY2 - (i*texDivY), 0));
56        textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
57        (texY2 - texDivY) - (i*texDivY), 0));
58        textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
59        texY2 - (i*texDivY), 0));
60
61            //Face traseira
62
63            LP.push_back(new Point(-x + (j*pX),-y + (i*pY),-z));
64      normal.push_back(new Point(0, 0, -1));
65            LP.push_back(new Point(-x + (j*pX),(-y+pY) + (i*pY),-z));
66      normal.push_back(new Point(0, 0, -1));
67            LP.push_back(new Point((-x+pX) + (j*pX),-y + (i*pY),-z));
68      normal.push_back(new Point(0, 0, -1));
```

```
69
70
71              LP.push_back(new Point(-x + (j*pX),(-y+pY) + (i*pY),-z));
72          normal.push_back(new Point(0, 0, -1));
73              LP.push_back(new Point((-x+pX) + (j*pX),(-y+pY) + (i*pY),-z));
74          normal.push_back(new Point(0, 0, -1));
75              LP.push_back(new Point((-x+pX) + (j*pX),-y + (i*pY),-z));
76          normal.push_back(new Point(0, 0, -1));
77
78            // Comecar na posicao final, 1, e ir diminuindo um
79            //"passo" de cada vez.Por outro lado comeca - se
80            //  no texY2 e diminui - se um "passo"
81            textura.push_back(new Point((1 - texDivX) - (j*texDivX), (texY2
82            - texDivY) - (i*texDivY), 0));
83            textura.push_back(new Point(1 - (j*texDivX), (texY2 - texDivY)
84            - (i*texDivY), 0));
85            textura.push_back(new Point(1 - (j*texDivX), texY2 - (i*texDivY), 0));
86
87            // Triangulo complementar ao anterior
88            textura.push_back(new Point((1 - texDivX) - (j*texDivX), (texY2 - texDivY)
89            - (i*texDivY), 0));
90            textura.push_back(new Point(1 - (j*texDivX), texY2 - (i*texDivY), 0));
91            textura.push_back(new Point((1 - texDivX) - (j*texDivX),
92            texY2 - (i*texDivY), 0));
93
94                //Face direita
95              LP.push_back(new Point(x,-y + (i*pY),-z + (j*pZ)));
96          normal.push_back(new Point(1, 0, 0));
97              LP.push_back(new Point(x,(-y+pY) + (i*pY),-z +(j*pZ)));
98          normal.push_back(new Point(1, 0, 0));
99              LP.push_back(new Point(x,-y + (i*pY),(-z+pZ) + (j*pZ)));
100         normal.push_back(new Point(1, 0, 0));
101
102
103             LP.push_back(new Point(x,(-y+pY) + (i*pY),-z + (j*pZ)));
104         normal.push_back(new Point(1, 0, 0));
105             LP.push_back(new Point(x,(-y+pY) + (i*pY),(-z+pZ) + (j*pZ)));
106         normal.push_back(new Point(1, 0, 0));
107             LP.push_back(new Point(x,-y + (i*pY),(-z+pZ) + (j*pZ)));
108         normal.push_back(new Point(1, 0, 0));
109
110           // Comecar na posicao de texX2 e ir aumentando um
111           //"passo" de cada vez, em Z.Por outro lado
112           //  comeca - se no texY2 e diminui - se um "passo".
113           textura.push_back(new Point(texX2 + (j*texDivZ), (texY2 - texDivY)
114           - (i*texDivY), 0));
115           textura.push_back(new Point((texX2 + texDivZ) - (j*texDivX),
116           (texY2 - texDivY) - (i*texDivY), 0));
117           textura.push_back(new Point(texX2 + (j*texDivZ), texY2 - (i*texDivY), 0));
118
119           // Triangulo complementar ao anterior
120           textura.push_back(new Point((texX2 + texDivZ) + (j*texDivZ),
121           (texY2 - texDivY) - (i*texDivY), 0));
122           textura.push_back(new Point((texX2 + texDivZ) + (j*texDivX),
123           texY2 - (i*texDivY), 0));
124           textura.push_back(new Point(texX2 + (j*texDivZ), texY2 - (i*texDivY), 0));
125
126               //Face esquerda
```

```
127             LP.push_back(new Point(-x,-y + (i*pY),-z + (j*pZ)));
128         normal.push_back(new Point(-1, 0, 0));
129             LP.push_back(new Point(-x,-y + (i*pY),(-z+pZ) + (j*pZ)));
130         normal.push_back(new Point(-1, 0, 0));
131             LP.push_back(new Point(-x,(-y+pY) + (i*pY),-z +(j*pZ)));
132         normal.push_back(new Point(-1, 0, 0));
133
134
135             LP.push_back(new Point(-x,(-y+pY) + (i*pY),-z + (j*pZ)));
136         normal.push_back(new Point(-1, 0, 0));
137             LP.push_back(new Point(-x,-y + (i*pY),(-z+pZ) + (j*pZ)));
138         normal.push_back(new Point(-1, 0, 0));
139             LP.push_back(new Point(-x,(-y+pY) + (i*pY),(-z+pZ) + (j*pZ)));
140         normal.push_back(new Point(-1, 0, 0));
141
142           // Comecar na posicao inicial, 0, (tendo em conta
143           // que esta face utiliza a variavel Z) e ir
144           //aumentando um "passo" de cada vez.Por outro
145           //  lado comeca - se no texY2 e diminui - se um
146           //  "passo".
147
148           textura.push_back(new Point((j*texDivZ), texY2 - (i*texDivY), 0));
149           textura.push_back(new Point(j*texDivZ, (texY2 - texDivY) - (i*texDivY), 0));
150           textura.push_back(new Point(texDivZ + (j*texDivZ),
151           (texY2 - texDivY) - (i*texDivY), 0));
152
153           // Triangulo complementar ao anterior
154           textura.push_back(new Point((j*texDivZ), texY2 - (i*texDivY), 0));
155           textura.push_back(new Point(texDivZ + (j*texDivZ), (texY2 - texDivY)
156           - (i*texDivY), 0));
157           textura.push_back(new Point(texDivZ + (j*texDivZ), texY2 - (i*texDivY), 0));
158
159               //Topo
160             LP.push_back(new Point(-x + (j*pX),y,-z + (i*pZ)));
161         normal.push_back(new Point(1, 0, 0));
162             LP.push_back(new Point(-x + (j*pX),y,(-z+pZ) + (i*pZ)));
163         normal.push_back(new Point(1, 0, 0));
164             LP.push_back(new Point((-x+pX) + (j*pX),y,-z + (i*pZ)));
165         normal.push_back(new Point(1, 0, 0));
166
167
168             LP.push_back(new Point(-x + (j*pX),y,(-z+pZ) + (i*pZ)));
169         normal.push_back(new Point(1, 0, 0));
170             LP.push_back(new Point((-x+pX) + (j*pX),y,(-z+pZ) + (i*pZ)));
171         normal.push_back(new Point(1, 0, 0));
172             LP.push_back(new Point((-x+pX) + (j*pX),y,-z + (i*pZ)));
173         normal.push_back(new Point(1, 0, 0));
174
175           // Comecar na posicao de texX1 e ir aumentando um
176           //"passo" de cada vez.Por outro lado comeca - se
177           //  na posicao final e diminui - se um "passo", em
178           //  Z.
179           textura.push_back(new Point(texX1 + (j*texDivX), 1 - (i*texDivZ), 0));
180           textura.push_back(new Point(texX1 + (j*texDivX), (1 - texDivZ)
181           - (i*texDivZ), 0));
182           textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
183           (1 - texDivZ) - (i*texDivZ), 0));
184
```

```
185         // Triangulo complementar ao anterior
186         textura.push_back(new Point(texX1 + (j*texDivX), 1 - (i*texDivZ), 0));
187         textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
188         (1 - texDivZ) - (i*texDivZ), 0));
189         textura.push_back(new Point((texX1 + texDivX) + (j*texDivX),
190         1 - (i*texDivZ), 0));
191 
192             //Base
193             LP.push_back(new Point(-x + (j*pX),-y,-z + (i*pZ)));
194       normal.push_back(new Point(-1, 0, 0));
195             LP.push_back(new Point((-x+pX) + (j*pX),-y,-z + (i*pZ)));
196       normal.push_back(new Point(-1, 0, 0));
197             LP.push_back(new Point(-x + (j*pX),-y,(-z+pZ) + (i*pZ)));
198       normal.push_back(new Point(-1, 0, 0));
199 
200 
201             LP.push_back(new Point(-x + (j*pX),-y,(-z+pZ) + (i*pZ)));
202       normal.push_back(new Point(-1, 0, 0));
203             LP.push_back(new Point((-x+pX) + (j*pX),-y,-z + (i*pZ)));
204       normal.push_back(new Point(-1, 0, 0));
205             LP.push_back(new Point((-x+pX) + (j*pX),-y,(-z+pZ) + (i*pZ)));
206       normal.push_back(new Point(-1, 0, 0));
207 
208         // Comecar na posicao de texX1 e ir aumentando um
209         //"passo" de cada vez.Por outro lado comeca - se
210         //  no texY1 e diminui - se um "passo", em Z.
211         textura.push_back(new Point(texX1 + (j*texDivX), (texY1 - texDivZ)
212         - (i*texDivZ), 0));
213         textura.push_back(new Point(texX1 + (j*texDivX), texY1
214         - (i*texDivZ), 0));
215         textura.push_back(new Point((texX1 + texDivX) + (j*texDivX), texY1
216         - (i*texDivZ), 0));
217 
218         // Triangulo complementar ao anterior
219         textura.push_back(new Point(texX1 + (j*texDivX), (texY1 - texDivZ)
220         - (i*texDivZ), 0));
221         textura.push_back(new Point((texX1 + texDivX) + (j*texDivX), texY1
222         - (i*texDivZ), 0));
223         textura.push_back(new Point((texX1 + texDivX) + (j*texDivX), (texY1
224         - texDivZ) - (i*texDivZ), 0));
225         }
226     }
227 
228 }
```

3.1.3 Esfera

Em relação à esfera, para o cálculo das normais desta, teve-se em conta as fórmulas relativas às coordenadas cartesianas lecionadas na UC:

$x = r * cos(\beta) * sin(\alpha)$
$y = r * cos(\beta) * cos(\alpha)$
$z = r * sin(\beta)$

Como o que se quer calcular são as normais,apenas importa a direção do vetor,descartando-se a parte relativa ao raio da esfera.

O raciocínio por detrás do cálculo das texturas teve como base a seguinte imagem:

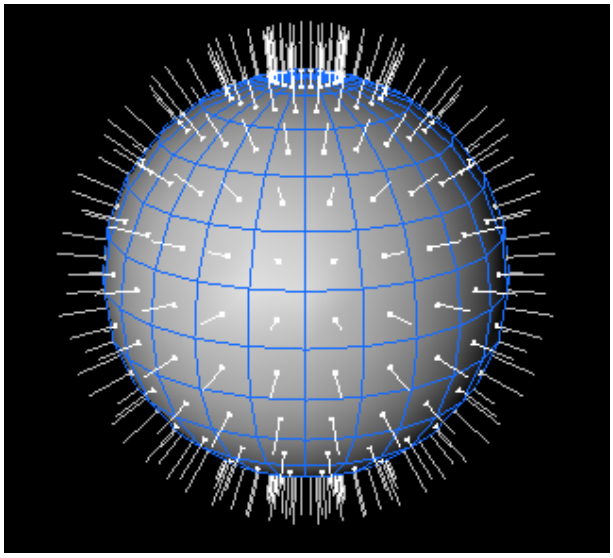

Figura 3.5: Normal do plano.

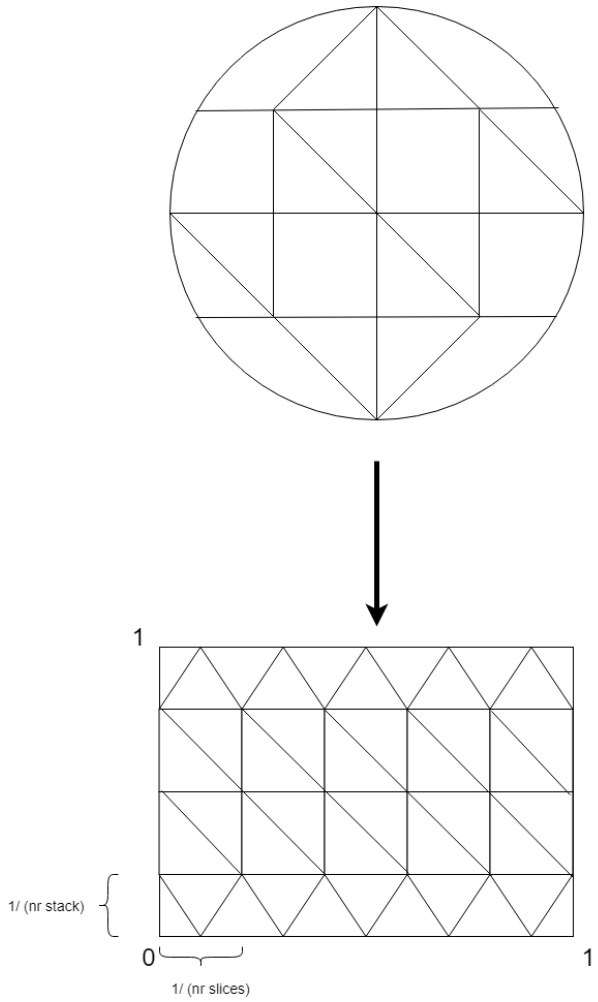

Figura 3.6: Algortimo de desenho de superficies esfericas

```
1
2  // Responsavel por gerar os pontos dos triangulos relativos a esfera.
3  void Struct::genSphere(float radius, int slices, int stacks) {
4
5    float beta = (float) (M_PI) / stacks; //angulo beta
6    float alfa = (float) (2 * M_PI) / slices; //angulo alfa
7
8    float texDivX = 1 / slices;
9    float texDivY = 1 / stacks;
10
11   float texX1 = 0;
12   float texX2 = texDivX;
13
14   float texY1, texY2;
15
16   //para cada slice percorrem-se as respectivas stacks
17   for (int i = 0; i < slices; i++, texX1 += texDivX, texX2 += texDivX) {
18
19     //calculam-se os pontos por slice
20     for (int j = 0; j < stacks; j++) {
21
22
23       //s   foram consideradas
24       // express es que se repitam mais do que uma
25       // vez por "if"
26       //porque, salvo certas excep  es,
27       // o mesmo "if"    executado uma vez por ciclo
28       float x1 = radius * cos((j + 1)*beta);
29       float x2 = radius * cos(i*alfa)*sin((j + 1)*beta);
30       float x3 = radius * sin(i*alfa)*sin((j + 1)*beta);
31       float x4 = radius * cos((j + 1)*beta);
32       float x5 = radius * cos((i + 1)*alfa)*sin((j + 2)*beta);
33       float x6 = radius * cos((j + 2)*beta);
34       float x7 = radius * sin((i + 1)*alfa)*sin((j + 2)*beta);
35
36       //come a-se a construir a esfera por cima
37       //j = 0 corresponde ao topo (polo norte) da esfera
38       //porque como podemos ver (0,radius,0)
39       //o que indica que estamos no topo
40       if (j == 0) {
41         texY1 = 1;              // como estamos no topo, o valor    m ximo
42         texY2 = 1 - texDivY;           // logo abaixo do topo, ser   este o valor
43
44         LP.push_back(new Point(0+xxx, radius+yyy, 0+zzz));
45         normal.push_back(new Point(0, 1, 0));
46         textura.push_back(new Point(texX1 + texDivX/2, texY1, 0));
47
48         LP.push_back(new Point(radius*cos((i + 1)*alfa)*sin((j + 1)*beta)+xxx,
49         x1+yyy, radius*sin((i + 1)*alfa)*sin((j + 1)*beta) +zzz));
50         normal.push_back( (new Point( radius*cos((i + 1)*alfa)*sin((j + 1)*beta)
   ,  x1    , radius*sin((i + 1)*alfa)*sin((j + 1)*beta)  ))->calcula_Normal);
51         textura.push_back(new Point(texX2, texY2, 0));
52
53
54         LP.push_back(new Point(x2+xxx, x1+yyy, x3+zzz));
55         normal.push_back((new Point(x2, x1, x3))->calcula_Normal);
56         textura.push_back(new Point(texX1, texY2, 0));
57       }
```

```
58
59          //estamos no polo sul da esfera
60          if (j == stacks - 1) {
61
62              //definicao dos pontos de referencia nesta
63              //regiao
64              texY1 = texDivY;
65              texY2 = 0;
66
67            LP.push_back(new Point(0+xxx, -radius+yyy, 0+zzz));
68            normal.push_back(new Point(0, -1, 0));
69
70            LP.push_back(new Point(x2 + radius*cos(i*alfa)*sin(beta) + xxx,
71            -(radius*cos(beta)) + yyy, x3 + radius*sin(i*alfa)*sin(beta) + zzz));
72            normal.push_back( (new Point( x2 + radius * cos(i*alfa)*sin(beta) ,
  -(radius*cos(beta))   ,  x3 + radius * sin(i*alfa)*sin(beta)  ))->calcula_Normal );
73
74            LP.push_back(new Point(radius*cos((i + 1)*alfa)*sin((j + 1)*beta) +
75            radius*cos((i + 1)*alfa)*sin(beta) + xxx,-(radius * cos(beta)) + yyy,
76            radius*sin((i + 1)*alfa)*sin((j + 1)*beta) + radius * sin((i + 1)*alfa)
77            *sin(beta) + zzz));
78            normal.push_back((new Point(radius*cos((i + 1)*alfa)*sin((j + 1)*beta)
79            + radius * cos((i + 1)*alfa)*sin(beta),-(radius * cos(beta)),
80              radius*sin((i + 1)*alfa)*sin((j + 1)*beta) + radius * sin((i + 1)*alfa)
81              *sin(beta))->calcula_Normal));
82
83              //raciocnio analogo a regi~ao topo
84              textura.push_back(new Point(texX1 + texDivX / 2, texY2, 0));
85              textura.push_back(new Point(texX1, texY1, 0));
86              textura.push_back(new Point(texX2, texY1, 0));
87
88          }//para os n veis intermedios entre os polos
89          else {
90
91            LP.push_back(new Point(radius*cos((i + 1)*alfa)*sin((j + 1)*beta) + xxx,
92            x4+yyy, radius*sin((i + 1)*alfa)*sin((j + 1)*beta)+zzz));
93            normal.push_back((new Point(radius*cos((i + 1)*alfa)*sin((j + 1)*beta), x4,
94            radius*sin((i + 1)*alfa)*sin((j + 1)*beta))->calcula_Normal));
95            LP.push_back(new Point(x5+xxx, x6+yyy, x7+zzz));
96            normal.push_back((new Point(x5, x6, x7))->calcula_Normal);
97            LP.push_back(new Point(x2+xxx, x4+yyy, x3+zzz));
98            normal.push_back((new Point(x2, x4, x3))->calcula_Normal);
99
100           LP.push_back(new Point(x2+xxx, x4+yyy, x3+zzz));
101           normal.push_back((new Point(x2, x4, x3))->calcula_Normal);
102           LP.push_back(new Point(x5+xxx, x6+yyy,x7+zzz));
103           normal.push_back((new Point(x5, x6, x7))->calcula_Normal);
104           LP.push_back(new Point(radius*cos(i*alfa)*sin((j + 2)*beta)+xxx, x6+yyy,
105           radius*sin(i*alfa)*sin((j + 2)*beta)+zzz));
106           normal.push_back((new Point(radius*cos(i*alfa)*sin((j + 2)*beta) , x6 ,
107           radius*sin(i*alfa)*sin((j + 2)*beta) ))->calcula_Normal);
108
109             //Triangulo lateral
110             textura.push_back(new Point(texX2, texY1, 0));
111             textura.push_back(new Point(texX2, texY2, 0));
112             textura.push_back(new Point(texX1, texY1, 0));
113
114             //Triangulo complementar ao anterior
```

```
115             textura.push_back(new Point(texX1, texY1, 0));
116             textura.push_back(new Point(texX2, texY2, 0));
117             textura.push_back(new Point(texX1, texY2, 0));
118         }
119     }
120 }
121 }
```

3.1.4 Cone

Tendo em conta que o raciocínio para a construção das normais e pontos de textura do cone é semelhante ao da esfera, foi usada a mesma figura como auxílio. É preciso ter em conta que nesta figura geométrica o cálculo dos vetores normais pode ser dividido em 2 partes, as da base cujo vetor é (0,0,1) e as do corpo

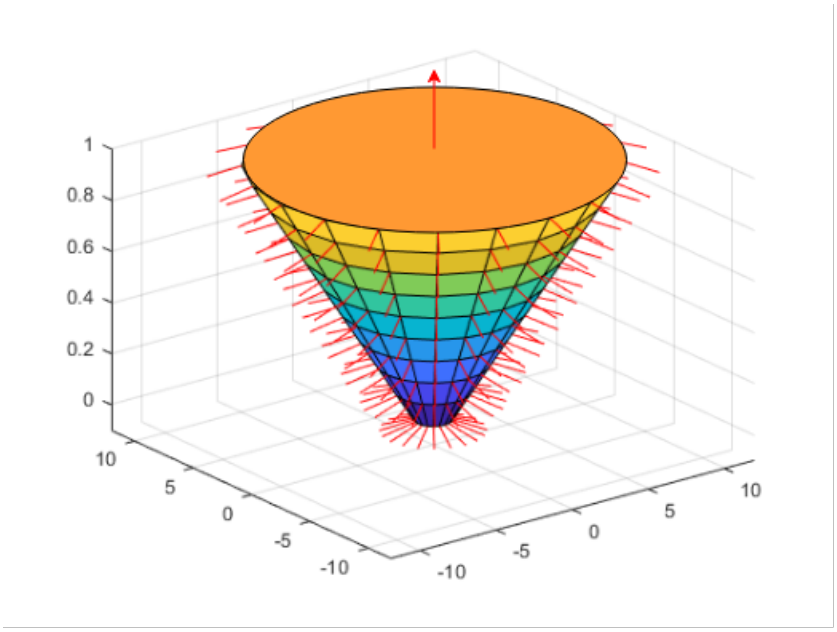

Figura 3.7: Normal do cone.

```
1  void Struct::genCone(float radius, float height, int slices, int stacks) {
2    float alfa = (float) (2 * M_PI) / slices; //angulo alfa
3    float alturaStack = height / stacks; // altura de uma stack
4    float tangenteBeta = height / radius;
5
6    float alturaBaixo = 0; //altura parte de baixo
7    float alturaCima; //altura parte de cima
8    float raio2; //raio do circulo superior
9
10   //um slice pode ser resumido em quatro pontos
11   //ligados entre a camada superior e inferior
12   //   __
13   //  /\/
14   //
15   float x1, x2, x3, x4, z1, z2, z3, z4;
16
```

```
17      //variaveis que representam um "passo" na textura
18      float texDivY = 1.0 / stacks, texDivX = 1.0 / slices;
19
20      //pontos de referencia da textura
21      float texYcima = 0 + texDivY, texYbaixo = 0;
22      float texXesq, texXdir;
23
24    float alfal;
25    float alfalMais1;
26
27    //este ciclo exterior trata das camadas
28    //o que faz com que o cone seja
29    //desenhado por "andares"
30    //que convergem no topo do cilindro
31    for (int j = 1; j <= stacks; j++, texXesq += texDivX, texXdir += texDivX) {
32      alturaCima = alturaStack * j;
33      raio2 = (height - alturaCima) / tangenteBeta;
34
35      //este ciclo foca-se num slice particular
36      //portanto, quando
37      for (int i = 1; i <= slices + 1; i++) {
38
39        alfal = alfa * i;
40        alfalMais1 = alfa * (i + 1);
41
42        x1 = radius * sin(alfal);
43        z1 = radius * cos(alfal);
44
45        x2 = radius * sin(alfalMais1);
46        z2 = radius * cos(alfalMais1);
47
48        x3 = raio2 * sin(alfal);
49        z3 = raio2 * cos(alfal);
50
51        x4 = raio2 * sin(alfalMais1);
52        z4 = raio2 * cos(alfalMais1);
53
54        if (j == 1) {
55
56          //triangulo da base
57          LP.push_back(new Point(0, 0, 0));
58          normal.push_back(new Point(0, -1, 0));
59          LP.push_back(new Point(x2, 0, z2));
60          normal.push_back(new Point(0, -1, 0));
61          LP.push_back(new Point(x1, 0, z1));
62          normal.push_back(new Point(0, -1, 0));
63
64          //triangulos do lado
65          LP.push_back(new Point(x1, 0, z1));
66          normal.push_back(new Point(sin(alfal), 0, cos(alfal)));
67          LP.push_back(new Point(x2, 0, z2));
68          normal.push_back(new Point(sin(alfalMais1), 0, cos(alfalMais1)));
69          LP.push_back(new Point(x3, alturaCima, z3));
70          normal.push_back(new Point( x3, alturaCima , z3)->calcula_Normal);
71
72          LP.push_back(new Point(x2, 0, z2));
73          normal.push_back(new Point(sin(alfalMais1), 0, cos(alfalMais1)));
74          LP.push_back(new Point(x4, alturaCima, z4));
```

```
75          normal.push_back(new Point(x4, alturaCima, z4)->calcula_Normal);
76          LP.push_back(new Point(x3, alturaCima, z3));
77          normal.push_back(new Point(x4, alturaCima, z4)->calcula_Normal);
78
79
80            //vertice inferior do triangulo (media das
81            //distancias)
82            textura.push_back(new Point(texXesq + texDivX / 2, texYbaixo, 0));
83            textura.push_back(new Point(texXdir, texYcima, 0));
84            textura.push_back(new Point(texXesq, texYcima, 0));
85
86            // lados
87            textura.push_back(new Point(texXesq, texYbaixo, 0));
88            textura.push_back(new Point(texXdir, texYbaixo, 0));
89            textura.push_back(new Point(texXesq, texYcima, 0));
90
91            textura.push_back(new Point(texXdir, texYbaixo, 0));
92            textura.push_back(new Point(texXdir, texYcima, 0));
93            textura.push_back(new Point(texXesq, texYcima, 0));
94
95        }// estamos no topo do cone
96        else if (j == stacks) {
97
98          //um unico triangulo no topo (por slice)
99          LP.push_back(new Point(x1, alturaBaixo, z1));
100         normal.push_back(new Point(x1, alturaBaixo, z1)->calcula_Normal);
101         LP.push_back(new Point(x2, alturaBaixo, z2));
102         normal.push_back(new Point(x2, alturaBaixo, z2)->calcula_Normal);
103         LP.push_back(new Point(0, height, 0));
104         normal.push_back(new Point(0, 1, 0));
105
106           texturesList.push_back(new Point(texXesq, texYbaixo, 0));
107           texturesList.push_back(new Point(texXdir, texYbaixo, 0))
108           texturesList.push_back(new Point(texXesq + texDivX / 2, texYcima, 0));
109       }
110       else {
111         //se nao estivermos no topo ou na base estamos a desenhar os lados do cone
112
113         //triangulos relativos aos lados do cone
114         LP.push_back(new Point(x1, alturaBaixo, z1));
115         normal.push_back(new Point(x1, alturaBaixo, z1)->calcula_Normal);
116         LP.push_back(new Point(x2, alturaBaixo, z2));
117           normal.push_back(new Point(x2, alturaBaixo, z2)->calcula_Normal);
118         LP.push_back(new Point(x3, alturaCima, z3));
119         normal.push_back(new Point(x3, alturaCima, z3)->calcula_Normal);
120
121         LP.push_back(new Point(x2, alturaBaixo, z2));
122         normal.push_back(new Point(x2, alturaBaixo, z2)->calcula_Normal);
123         LP.push_back(new Point(x4, alturaCima, z4));
124         normal.push_back(new Point(x4, alturaCima, z4)->calcula_Normal);
125         LP.push_back(new Point(x3, alturaCima, z3));
126         normal.push_back(new Point(x3, alturaCima, z3)->calcula_Normal);
127
128           textura.push_back(new Point(texXesq, texYbaixo, 0));
129           textura.push_back(new Point(texXdir, texYbaixo, 0));
130           textura.push_back(new Point(texXesq, texYcima, 0));
131
132           textura.push_back(new Point(texXdir, texYbaixo, 0));
```

```
133             textura.push_back(new Point(texXdir, texYcima, 0));
134             textura.push_back(new Point(texXesq, texYcima, 0));
135         }
136     }
137
138     //subimos de patamar
139     alturaBaixo = alturaCima;
140     //consequentemente tambem muda o circulo da base
141     radius = raio2;
142   }
143 }
```

3.1.5 Cilindro

3.1.6 Torus

Para a obtenção dos vetores normais do torus é fundamental estudar o processo de desenho do mesmo,processo esse que fornecerá as orientações dos vetores normais de cada vértice no momento do desenho deste.

Apenas sendo relevante a orientação da origem até cada vértice, os vetores normais são dados pelo cálculo dessa orientação.

O raciocínio por detrás do desenho das texturas teve como base, mais uma vez,a figura usada na esfera, sendo que desta vez só se utiliza a parte dos retângulos.

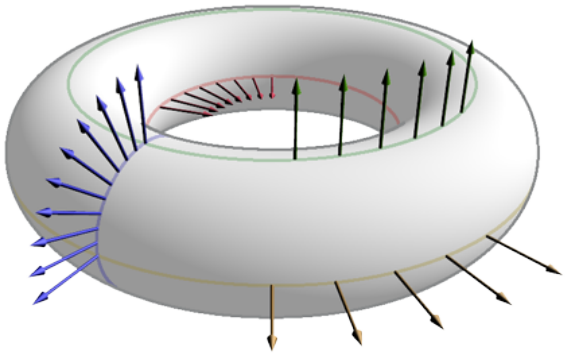

Figura 3.8: Normais do torus.

```
1
2 void Struct::genTorus(float tamanhoCoroa, float raio, int faces, int aneis) {
3
4   float lado = (float) (2 * M_PI) / faces;
5   float anel =  (float) (2 * M_PI) / aneis;
6
7     //variaveis que representam um "passo" na textura
8     float texX = 1.0 / slices;
9     float texY = 1.0 / stacks;
10
11
12  for (int i = 0; i < aneis; i++) {
13
```

```
14      float cosl = (float) cos(i * anel);
15      float sinl = (float) sin(i * anel);
16      float coslanel = (float) cos(i * anel + anel);
17      float sinlanel = (float) sin(i * anel + anel);
18
19      for (int j = 0; j < faces + 1; j++) {
20
21        float jj = j * lado;
22        float jjUm = (j + 1)*lado;
23
24        float tcos = tamanhoCoroa * cos(jj) + raio;
25        float tsin = tamanhoCoroa * sin(jj);
26
27        float tcosUm = tamanhoCoroa * (cos(jjUm)) + raio;
28        float tsinUm = tamanhoCoroa * sin(jjUm);
29
30        LP.push_back(new Point(cosl*tcos, sinl*tcos, tsin));
31        normal.push_back(new Point(cosl*tcos, sinl*tcos, tsin)->calcula_Normal);
32        LP.push_back(new Point(coslanel*tcos, sinlanel*tcos, tsin));
33        normal.push_back(new Point(coslanel*tcos, sinlanel*tcos, tsin)->calcula_Normal);
34        LP.push_back(new Point(cosl*tcosUm, sinl*tcosUm, tsinUm));
35        normal.push_back(new Point(cosl*tcosUm, sinl*tcosUm, tsinUm)->calcula_Normal);
36
37        LP.push_back(new Point(cosl*tcosUm, sinl*tcosUm, tsinUm));
38        normal.push_back(new Point(cosl*tcosUm, sinl*tcosUm, tsinUm)->calcula_Normal);
39        LP.push_back(new Point(coslanel*tcos, sinlanel*tcos, tsin));
40        normal.push_back(new Point(coslanel*tcos, sinlanel*tcos, tsin)->calcula_Normal);
41        LP.push_back(new Point(coslanel*tcosUm, sinlanel*tcosUm, tsinUm));
42        normal.push_back(new Point(coslanel*tcosUm, sinlanel*tcosUm, tsinUm)
43        ->calcula_Normal);
44
45          textura.push_back(new Point(texX*i, texY*j, 0));
46          textura.push_back(new Point(texX*(i + 1), texY*j, 0));
47          textura.push_back(new Point(texX*i, texY*(j + 1), 0));
48
49          //triangulo complementar ao anterior
50          textura.push_back(new Point(texX*i, texY*(j + 1), 0));
51          textura.push_back(new Point(texX*(i + 1), texY*j, 0));
52          textura.push_back(new Point(texX*(i + 1), texY*(j + 1), 0));
53
54      }
55    }
56  }
```

4 Engine

Tal como nas fases anteriores a ordem de execução das operações, presentes no ficheiro de configuração, segue a sequência:

Translation-> Rotation-> Scale.

Para além de todas as outras alterações levadas a cabo pelo grupo de trabalho de forma a implementar todas as funcionalidades necessárias, também se efectuaram mudanças no que toca à *performance* e optimização do código. Tais melhorias foram feitas recorrendo a *Vertex buffer object* (VBO) que é uma característica do OpenGL que oferece métodos para realizar o upload de dados relativos a vértices com bastantes aperfeiçoamentos relativamente a outros métodos existentes, nomeadamente no que toca ao armazenar os dados diretamente na memória gráfica ao invés de precisar de armazenar na memória do sistema, o que por consequência aumenta a rapidez de processamento.

O engine nesta fase também necessita de ser capaz de aplicar luz e texturas as figuras desenhadas, como tal foi necessário lidar com esta mudança de paradigma. Portanto, cada group, que contém informação retirada do XML, guarda um conjunto de luzes, luzes essas que são representadas por pontos, sendo estas aplicadas na função *draw* do group e o *draw* do light. O material é um componente do ponto, sendo que o ponto é também uma componente do group, o que permite portanto, que estes sejam aplicados aos planetas. Continuando com a textura, esta é aplicada na classe pontos, sendo esta desenhada na função *draw*.

5 Resultados

Conjugando todos os elementos explicados anteriormente, foi possível obter os seguintes resultados:

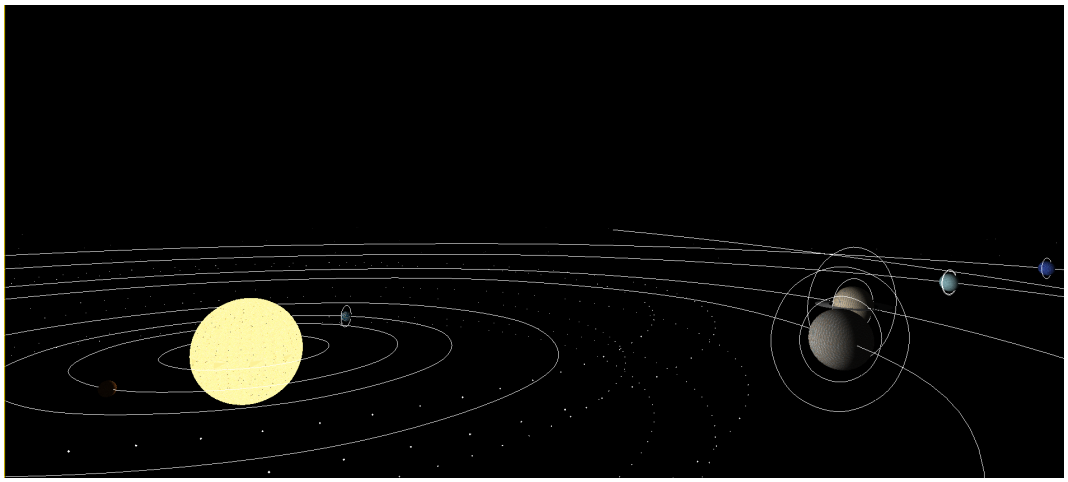

Figura 5.1: Representação do sistema solar.

6 Conclusão

Nesta quarta e última fase de uma sucessão de trabalhos práticos da unidade curricular de Computação Gráfica, continuou-se a desenvolver o nosso conhecimento sobre C++ e OpenGL, com mais ênfase no cálculo e aplicação de texturas, normais e iluminação.

Apesar de algumas dificuldades no cálculo das texturas, o grupo de trabalho superou esses impedimentos e cumpriu os objectivos propostos. Estas dificuldades, de forma semelhante às fases anteriores, surgiram devido à quantidade e a alguma complexidade dos cálculos que têm de ser feitos.

Assim, comparando a globalidade do trabalho ao que era pedido, afirmamos que foi gerado um modelo do sistema solar realista e o congruente.

Em conclusão, depois de tudo o que foi feito durante o desenvolvimento das consecutivas fases de trabalho, cremos que o conhecimento coletivo do grupo de trabalho cresceu no que toca à utilização das várias funcionalidades do OpenGL e do GLUT.