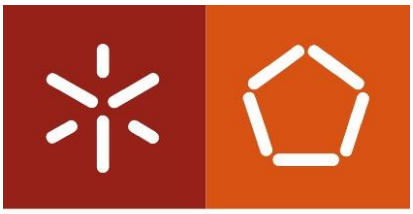


Universidade do Minho

Departamento de Informática

Relatório CG Fase 1
Grupo 31

Gustavo Lourenço a89561 João Machado a89510
Martim Almeida a89501

Ano Letivo 2020/21

Conteúdo

1 Introdução **3**

2 Classes Auxiliares **4**
2.1 Sistema Cartesiano 4
2.2 Sistema Esférico 5
2.3 Vetor 5

3 Generator **6**
3.1 Plano 6
3.2 Caixa 7
3.3 Esfera 10
3.4 Cone 11

4 Engine **13**
4.1 Estrutura de Dados e Desenho dos Modelos 13
4.2 Parsing do Ficheiro de Configuração 14
4.3 Outras Utilidades 15

5 Conclusão **16**

Capítulo 1

Introdução

Este relatório tem o objectivo de apresentar a fase inicial do Projecto da UC Computação Gráfica.

Neste relatório iremos apresentar os requisitos necessários presentes nesta fase:

1. Um *Generator* de ficheiros com a informação de um determinado Modelo;
2. Uma *Engine* que inicialmente lê um ficheiro escrito em XML que contém os modelos e posteriormente faz *display* dos mesmos.

Capítulo 2

Classes Auxiliares

Para facilitar o desenvolvimento desta fase, foi necessário a implementação de certas classes para representar o Sistema Cartesiano, o Sistema Esférico e Vetores nesses mesmos sistemas.

2.1 Sistema Cartesiano

Para o Sistema Cartesiano, implementámos a Classe *Point* que serve para representar um ponto num espaço em 3 Dimensões, sendo necessário 3 valores: x, y e z.

Para além disso, a Classe *Point* consta com a implementação de conversão de um Ponto do Sistema Esférico num Ponto do Sistema Cartesiano e com a implementação de somar um determinado Vetor a um Ponto.

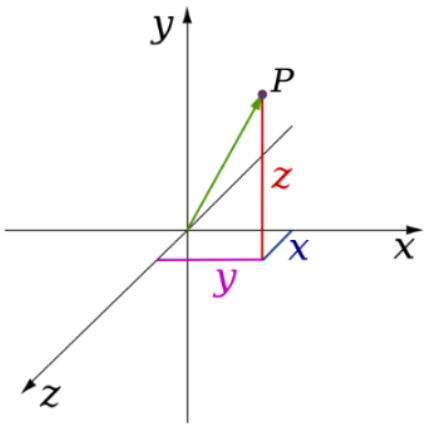


Figura 2.1: Sistema Cartesiano

2.2 Sistema Esférico

Para o Sistema Esférico, implementámos a Classe *SphericalPoint* cujo objetivo é representar um ponto no Espaço, sendo necessário também 3 valores: θ, ϕ e raio (no desenvolvimento deste trabalho decidimos renomear θ como α e ϕ como β).

Tal como afirmado acima, e como pode ser observado na Figura 2.2, é possível obter Coordenadas Cartesianas a partir de Coordenadas Esféricas a partir dos seguintes cálculos:

$$x = r \times \cos\beta \times \sin\alpha$$

$$y = r \times \sin\beta$$

$$z = r \times \cos\beta \times \cos\alpha$$

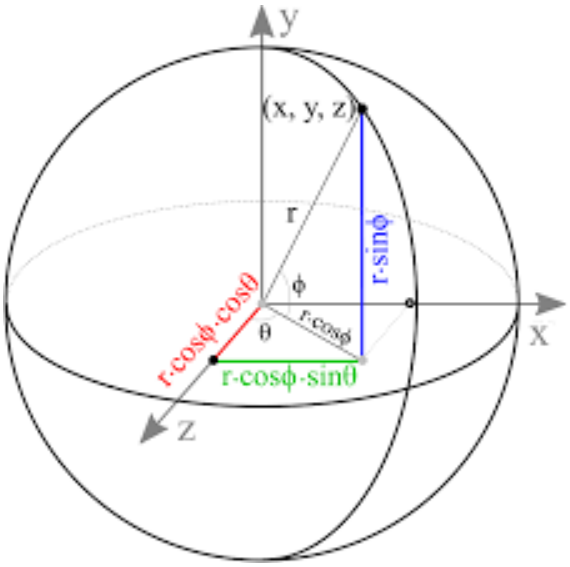


Figura 2.2: Sistema Esférico

2.3 Vetor

Para realizar translações dos pontos nos Sistemas, foi necessário a implementação da Classe *Vector*, que contém os vários métodos capazes para realizar essas mesmas operações, sendo que um Vetor necessita de 3 valores: x, y e z.

Capítulo 3

Generator

O *Generator* é responsável pela criação de ficheiros dos vários Modelos/Primitivas presentes no trabalho, sendo apresentadas no ficheiro os vários conjuntos de pontos necessários para gerar a Primitiva graficamente.

Por exemplo, os pontos (0,0,1), (0,1,0) e (1,0,0) ficariam representados no ficheiro:

```
0 0 1
0 1 0
1 0 0
```

As Primitivas que são possíveis de gerar são:

- Plano
- Caixa
- Esfera
- Cone

3.1 Plano

Como requerimento desta fase, o Plano gerado tem que estar no Plano XZ, centrado na Origem e é feito por dois triângulos.

Uma vez que passámos como argumento o tamanho do plano e sabemos que o Plano tem que estar no Plano XZ, conseguimos saber os 4 cantos do plano, necessários para desenhar os 2 triângulos.

Para desenhar um triângulo com o lado visível para cima, é necessário utilizar a Regra da Mão Direita de forma a decidir por qual ordem deve-se desenhar os pontos. Desta forma, os dois triângulos necessários são definidos:

Sendo $half$:

$$half = Tamanho\ do\ Plano/2$$

Primeiro Triângulo:

$$(-half, \quad 0, \quad -half)$$
$$(-half, \quad 0, \quad half)$$
$$(half, \quad 0, \quad -half)$$

Segundo Triângulo:

$$(half, \quad 0, \quad -half)$$
$$(-half, \quad 0, \quad half)$$
$$(half, \quad 0, \quad half)$$

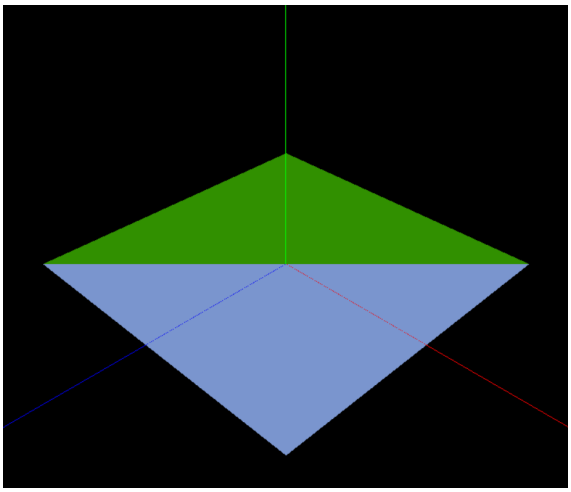


Figura 3.1: Plano, tamanho: 4

3.2 Caixa

Para desenhar uma caixa é necessário passar como argumentos as dimensões da Caixa (x,y e z) e, opcionalmente, o número de divisões por cada aresta. O número de divisões por aresta vai dividir o cubo em $div + 1$ partes.

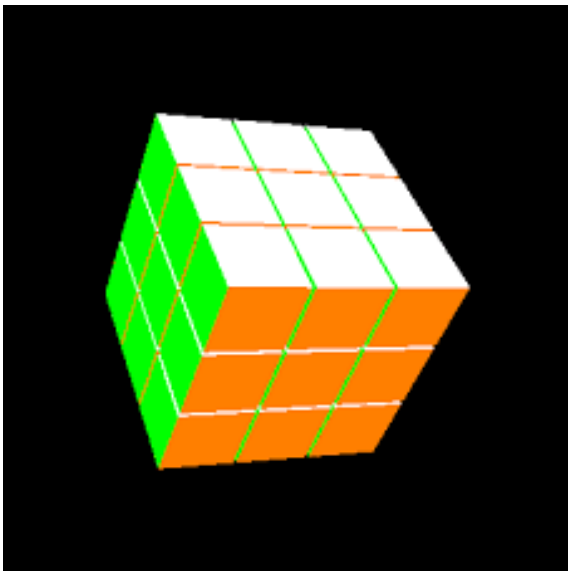


Figura 3.2: Caixa com div igual a 2

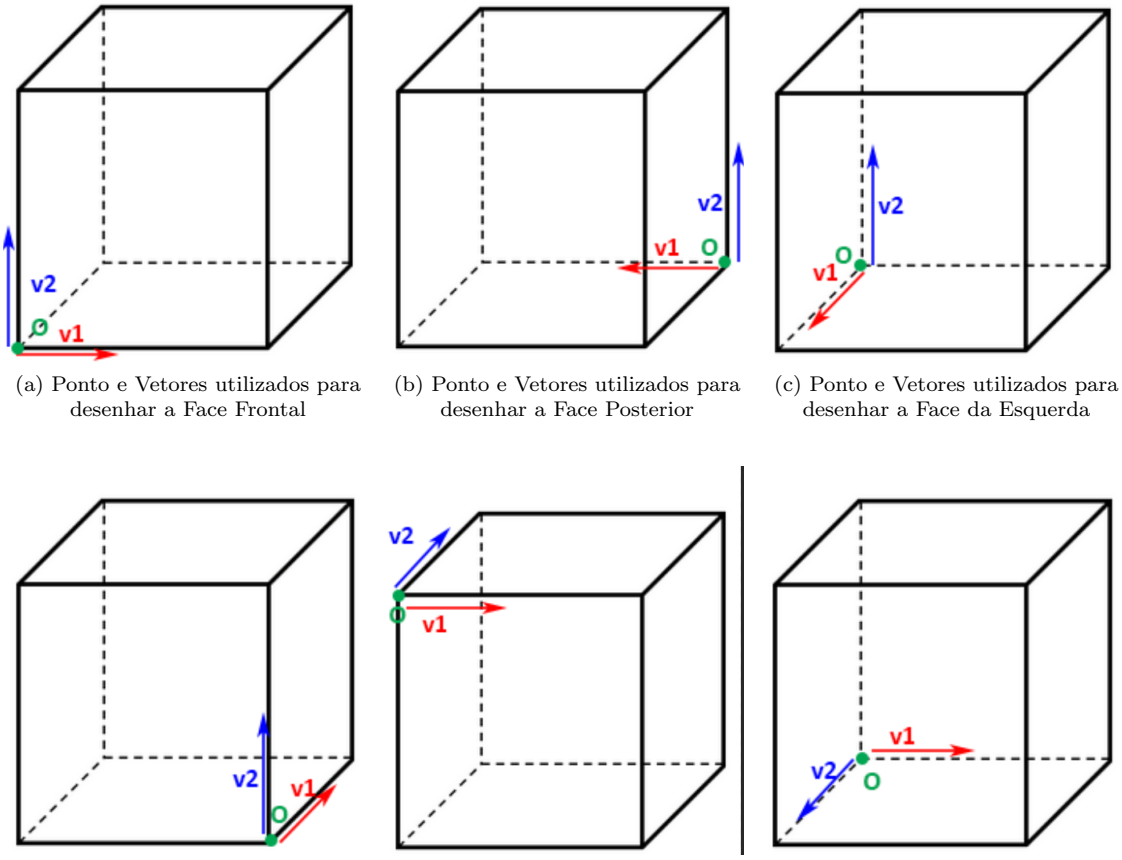
Para desenhar uma caixa, desenhámos cada uma das faces sequencialmente.

Para isso, utilizámos um método que desenha a face de um cabo dado um Ponto de origem *o* e dois Vetores *v1* e *v2*.

Através desses vetores e do ponto de origem, conseguimos obter quatro pontos, que representam os pontos constituintes do quadrado do canto inferior esquerdo da face desejada. Após isso, desenhámos esse quadrado e, dependendo do número de *divs*, fazemos $div + 1$ translações através do Vetor *v1*, e assim obtemos uma "linha"constituinte do cubo.

Para obter as restantes "linhas", nós fazemos $div + 1$ translações através do Vetor *v2*, selecionando assim um quadrado com uma posição superior comparado ao inicial. Depois de fazer isso, repetimos o passo anterior para desenhar todas as linhas a partir da primeira coluna.

```
void Box::draw_face(std::vector<Point>& points, Point o, Vector v1, Vector v2)
    const {
    //creating the points from the two triangles on the bottom left corner of the
        face
    Point p0 = Point(o.get_x(), o.get_y(), o.get_z());
    Point p1 = Point(p0.get_x(), p0.get_y(), p0.get_z());
    p1.add_vector(v1);
    Point p2 = Point(p0.get_x(), p0.get_y(), p0.get_z());
    p2.add_vector(v2);
    Point p3 = Point(p0.get_x(), p0.get_y(), p0.get_z());
    p3.add_vector(v1);
    p3.add_vector(v2);

    //itera sobre as "colunas"
    for (int i = 0; i < div; i++) {
        Point p0j = Point(p0.get_x(), p0.get_y(), p0.get_z());
        Point p1j = Point(p1.get_x(), p1.get_y(), p1.get_z());
        Point p2j = Point(p2.get_x(), p2.get_y(), p2.get_z());
        Point p3j = Point(p3.get_x(), p3.get_y(), p3.get_z());

        //itera sobre as "linhas"
        for (int j = 0; j < div; j++) {
            //1st triangle
            points.push_back(p0j);
            points.push_back(p1j);
            points.push_back(p2j);

            //front triangle
            points.push_back(p2j);
            points.push_back(p1j);
            points.push_back(p3j);

            //translate all points to the next slice ("horizontally")
            p0j.add_vector(v1);
            p1j.add_vector(v1);
            p2j.add_vector(v1);
            p3j.add_vector(v1);
        }

        //translate all points to the next slice ("vertically")
        p0.add_vector(v2);
        p1.add_vector(v2);
        p2.add_vector(v2);
        p3.add_vector(v2);
    }
}
```



(a) Ponto e Vetores utilizados para desenhar a Face Frontal

(b) Ponto e Vetores utilizados para desenhar a Face Posterior

(c) Ponto e Vetores utilizados para desenhar a Face da Esquerda

(d) Ponto e Vetores utilizados para desenhar a Face da Direita

(e) Ponto e Vetores utilizados para desenhar a Face Superior

(f) Ponto e Vetores utilizados para desenhar a Face Inferior

Figura 3.3: Desenho das Faces da Caixa

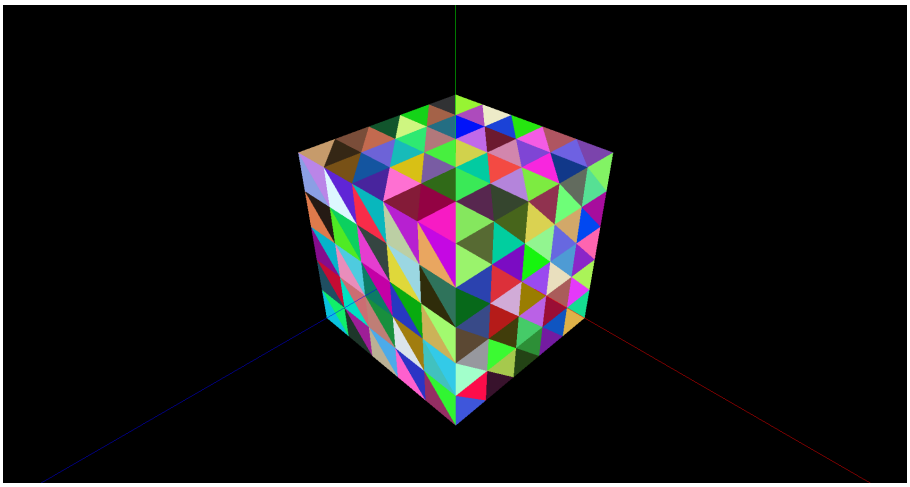


Figura 3.4: Caixa desenhada pela Engine com x = 3, y = 3, z = 3 e div = 4

3.3 Esfera

Para desenhar uma Esfera é necessário passar como argumentos o raio da Esfera, o número de *Slices* e o número de *Stacks*.

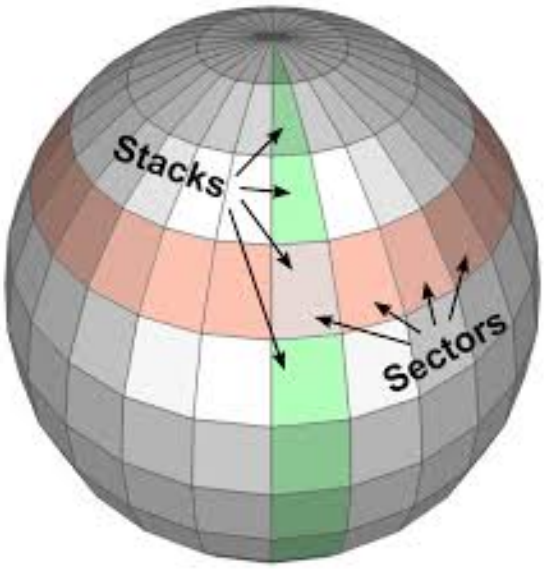


Figura 3.5: Representação gráfica de uma Esfera (denominamos Sectors como Slices)

Para desenhar uma Esfera começamos por iterar sobre as *Slices*. Desenhamos primeiro o Triângulo Superior da *Slice*, depois disso, desenhámos todos os quadrados contidos em todas as *Stacks* presentes nesta *Slice*, menos a útlima, a qual desenhamos no final da iteração, e assim, finalizámos o desenho da *Slice*.

Após iterarmos sobre todas as *Slices*, obtemos a Esfera.

```
std::vector<Point> Sphere::draw() {

    Point top = Point(0, radius, 0);
    Point bot = Point(0, -radius, 0);

    float step = 2 * M_PI / slices;
    float steph = M_PI / stacks;

    std::vector<Point> points;

    for (int i = 0; i < slices; i++) {
        Point p1 = Point(SphericalPoint(step * i, (M_PI / 2) - steph, radius));
        Point p2 = Point(SphericalPoint(step * (i+1), (M_PI / 2) - steph, radius));
        points.push_back(p1);
        points.push_back(p2);
        points.push_back(top);

        for (int j = 1; j < stacks-1; j++) {
            Point p3 = Point(SphericalPoint(step * i, (M_PI / 2) - steph * (j + 1),
                radius));
            Point p4 = Point(SphericalPoint(step * (i + 1), (M_PI / 2) - steph * (j
                + 1), radius));

            points.push_back(p1);
            points.push_back(p3);
            points.push_back(p2);

            points.push_back(p3);
```

```
27              points.push_back(p4);
28              points.push_back(p2);
29
30              p1 = Point(p3.get_x(), p3.get_y(), p3.get_z());
31              p2 = Point(p4.get_x(), p4.get_y(), p4.get_z());
32          }
33          points.push_back(p1);
34          points.push_back(bot);
35          points.push_back(p2);
36      }
37      return points;
```

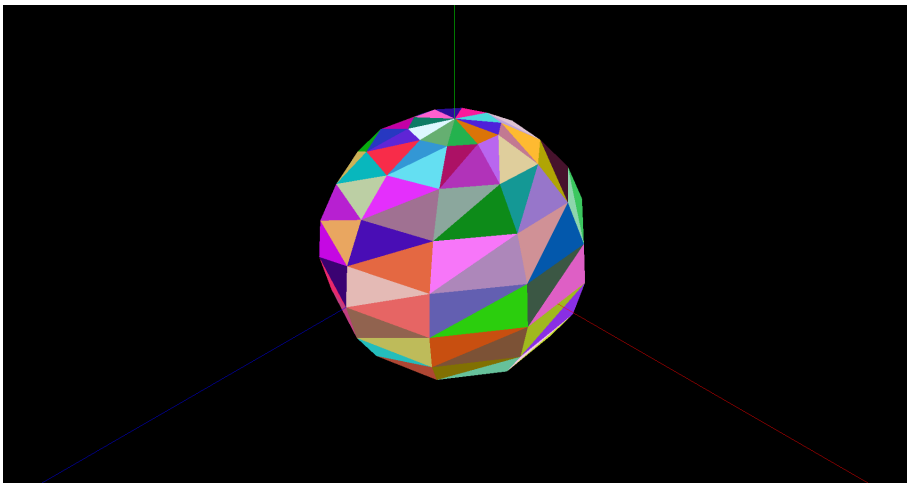


Figura 3.6: Esfera desenhada pela Engine com Raio = 2, Slices = 10 e Stacks = 10

3.4 Cone

Para desenhar um Cone é necessário passar como argumentos o raio da Base, a altura do Cone, o número de *Slices* e o número de *Stacks*. Para desenhar um Cone, iteramos por cada *Slice* e começamos por desenhar o Triângulo da Base do Cone correspondente à *Slice* e após isso, iteramos sobre as *Stacks* menos a última.

Para iterarmos sobre as *Stacks*, calculamos o Vetor necessário para a Translação dos dois Pontos que formam o Triângulo da base da *Slice* até ao Vértice Superior do Cone. Após obtermos esse Vetor, dividimos pelo número *Stacks*. Assim, para iterarmos sobre as *Stacks*, fazemos a translação dos Pontos iniciais pelo Vetor obtido através da Divisão e desenhámos a secção contida em cada uma das *Stacks* dessa mesma *Slice*.

Por fim, desenhámos o Triângulo Final que liga os dois Pontos da última *Stack* ao Vértice Superior do Cone.

```
1  std::vector<Point> Cone::draw() {
2      Point center = Point(0, 0, 0);
3
4      Point top = Point(0, height, 0);
5
6      float step = 2 * M_PI / slices;
7
8      std::vector<Point> points;
9
10     for (int i = 0; i < slices; i++) {
11         Point p1 = Point(SphericalPoint(step * i, 0, radius));
12         Point p2 = Point(SphericalPoint(step * (i+1), 0, radius));
13
14         points.push_back(p1);
```

```
15          points.push_back(center);
16          points.push_back(p2);
17
18          Vector step1 = Vector(top.get_x() - p1.get_x(), top.get_y() - p1.get_y(),
                top.get_z() - p1.get_z());
19          step1.divide(stacks);
20          Vector step2 = Vector(top.get_x() - p2.get_x(), top.get_y() - p2.get_y(),
                top.get_z() - p2.get_z());
21          step2.divide(stacks);
22
23
24          for (int j = 0; j < stacks - 1;j++) {
25              Point p3 = Point(p1.get_x(), p1.get_y(), p1.get_z());
26              p3.add_vector(step1);
27              Point p4 = Point(p2.get_x(), p2.get_y(), p2.get_z());
28              p4.add_vector(step2);
29
30              points.push_back(p1);
31              points.push_back(p2);
32              points.push_back(p3);
33
34              points.push_back(p2);
35              points.push_back(p4);
36              points.push_back(p3);
37
38              p1.set_x(p3.get_x());
39              p1.set_y(p3.get_y());
40              p1.set_z(p3.get_z());
41
42              p2.set_x(p4.get_x());
43              p2.set_y(p4.get_y());
44              p2.set_z(p4.get_z());
45          }
46
47          points.push_back(p2);
48          points.push_back(top);
49          points.push_back(p1);
50
51      }
52      return points;
53
54  }
```

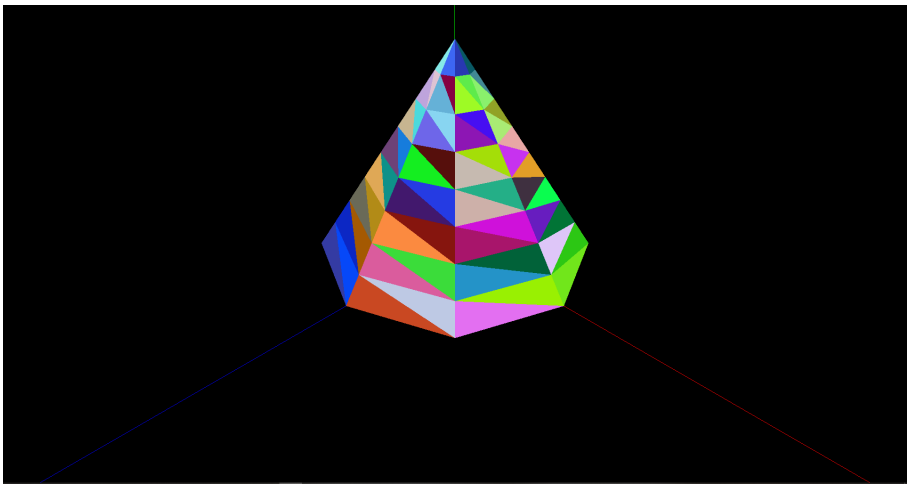


Figura 3.7: Cone desenhado pela Engine com Raio = 2, Altura = 3, Slices = 8 e Stacks = 5

Capítulo 4

Engine

O *Engine* é o módulo que contém um motor 3D.

Para executar o programa, é necessário passar o nome do Ficheiro de Configuração (escrito em XML) de forma a ser lido (sendo este ficheiro lido só uma vez) e toda a informação presente nesse ficheiro ser carregado para uma estrutura de dados.

Uma vez que nesta fase ainda não desenvolvemos sombras e luz, os vários triângulos que formam as várias primitivas presentes neste trabalho não são distinguíveis. Resolvemos este problemas fazendo com que os vários triângulos nas primitivas tivessem cores aleatórias, como podemos verificar, por exemplo, na Figura 3.7.

4.1 Estrutura de Dados e Desenho dos Modelos

Para auxiliar a Leitura do ficheiro e Armazenamento de toda a informação presente nele mesmo, foram criadas as Classes *Models* e *Model*.

```
class Model {
    private:
        std::vector<Point> points;
        void drawTriangles(Point p1, Point p2, Point p3);

    public:
        Model(const char *);
        void drawModel();
        void printValores();
};

class Models {
    private:
        std::vector<Model> models;

    public:
        Models();
        void drawModels();
        void readFile(char *);
};
```

Como podemos verificar pela porção de código acima, a Classe *Model* contém o Vetor de Pontos que são carregados dos Ficheiros presentes no Ficheiro de Configuração, sendo depois este *Model* guardado no Vetor presente na Classe *Models*.

Para além disso, podemos verificar os vários métodos criados para desenhar um modelo presente no Ficheiro de Configuração, sendo este o método *drawModel()*, que tem como método auxiliar o método *drawTriangles()*.

```
void Model::drawTriangles(Point p1, Point p2, Point p3){
    glColor3f((float) rand() / RAND_MAX, (float) rand() / RAND_MAX, (float) rand()
        / RAND_MAX);
    glBegin(GL_TRIANGLES);
        glVertex3f(p1.get_x(),p1.get_y(),p1.get_z());
        glVertex3f(p2.get_x(),p2.get_y(),p2.get_z());
        glVertex3f(p3.get_x(),p3.get_y(),p3.get_z());
    glEnd();
}

void Model::drawModel(){
    for (int i = 0; i < points.size(); i += 3){
        drawTriangles(points[i], points[i+1], points[i+2]);
    }
}
```

4.2 Parsing do Ficheiro de Configuração

Para realizar o Parsing do Ficheiro de Configuração XML, utilizamos a biblioteca *tinyxml2*.

Nesta Primeira Fase do Trabalho, a partir do Ficheiro XML apenas obtemos quais os Ficheiros que vão ser lidos e carregados para a Estrutura de Dados. Para garantir que o Ficheiro XML só é lido uma vez tal como é solicitado, o ficheiro é lido no início do programa e os vários ficheiros presentes no ficheiro XML são carregados para a Classe *Models*, através do método *readFile(char* fileName)*.

```
void Models::readFile(char * fileName){

    tinyxml2::XMLDocument xmlDOC;
    xmlDOC.LoadFile(fileName);
    if (xmlDOC.ErrorID()){
        printf("%s\n", xmlDOC.ErrorStr());
        exit(0);
    }

    tinyxml2::XMLNode* scene = xmlDOC.FirstChildElement("scene");
    if (scene == NULL){
        printf("Scene not found.\n");
        exit(0);
    }

    tinyxml2::XMLNode* model = scene->FirstChild();
    while(model){
        if(!strcmp(model->Value(), "model"))
            models.push_back(Model(model->ToElement()->Attribute("file")));
        model = model->NextSibling();
    }

}

Model::Model(const char* fileName){
    float x,y,z;
    std::ifstream file(fileName);
        while(file >> x >> y >> z){
            points.push_back(Point(x,y,z));
        }
}
```

No módulo *engine*, temos um Singleton de *Models* que guarda todos os resultados gerados pelo Parsing do ficheiro XML.

```
Models models;
```

4.3 Outras Utilidades

Para aquando da verificação dos modelos que geramos a partir do módulo *generator* estavam corretos, criamos o método *keyReaction* que juntamente com o método *glKeyboardFunc(keyReaction)* realizámos rotações da Primitiva que desenhamos no *engine*.

```
float angleX = 0;
float angleY = 0;
float angleZ = 0;
GLenum mode = GL_FILL;
void keyReaction(unsigned char key, int x, int y){
    switch(key){
        case '1':
            mode = GL_LINE;
            break;
        case '2':
            mode = GL_POINT;
            break;
        case '3':
            mode = GL_FILL;
            break;
        case 'z':
            angleX += 10;
            break;
        case 'x':
            angleX -= 10;
            break;
        case 'c':
            angleZ += 10;
            break;
        case 'v':
            angleZ -= 10;
            break;
        case 'a':
            angleY -= 10;
            break;
        case 'd':
            angleY += 10;
            break;
        case 'r':
            angleX = 0;
            angleY = 0;
            angleZ = 0;
            mode = GL_FILL;
            break;
    }
    glutPostRedisplay();
}
```

Capítulo 5

Conclusão

Com esta Fase aprendemos como aplicar diversos conteúdos lecionados nas aulas como *OpenGL* e o desenho de figuras através de Triângulos. Também aprendemos alguns conceitos sobre a linguagem C++, uma linguagem que nenhum dos integrantes não tinha um conhecimento vasto e como dar parsing a um ficheiro XML.

Futuramente, gostaríamos de implementar uma câmara e pretendemos aprender mais sobre a utilização de *CMake*/*CMakeLists* de forma a podermos melhorar a organização das várias diretorias do projeto e facilitar o uso do projeto entre os vários Sistemas Operativos, algo que tivemos um pouco de problemas.