Computação Gráfica

Fase 1 - Graphical Primitives

LEI - 2022/2023

Martim Ribeiro
A96113

Afonso Bessa
A95225

João Silva
A91671

Luís Vilas
A91697

Grupo 40

Universidade do Minho

Índice

Lista de Figuras

1 Introdução

No âmbito da Unidade Curricular de **Computação Gráfica** foi proposto o desenvolvimento de *mini scene graph based 3D engine* e fornecimento de exemplos que mostrassem todo o seu potencial.

O presente relatório visa mostrar todos os passos e tomadas de decisão que levaram à conclusão da primeira fase deste projeto constituída por duas aplicações:

- **Generator** - responsável por gerar um Ficheiro Binário que guarda os vértices para as diferentes Primitivas Gráficas: Plano, Cubo, Cone e Esfera, tendo em conta diversos parâmetros como raio, altura, números de divisões, entre outros.
- **Engine** - tem como função ler um Ficheiro de Configuração, escrito em *XML*, que contém as configurações da câmara e a indicação de quais ficheiros, previamente criados, têm de carregar, de forma a exibir as Primitivas Gráficas pretendidas.

Siglas:

XML - *Extensible Markup Language*

Keywords: *OpenGL*, *Glut*, *Glew*, *Devil*, *XML*, *Generator*, *Engine*, Primitivas Gráficas, Cubo, Esfera, Cone, Plano, *Slices*, *Stacks*

2 Estrutura do Projeto

Para a primeira fase do trabalho prático, este encontra-se dividido num **CMakeLists.txt** que é usado para compilar os dois projetos, *Generator* e *Engine*, incluindo diferentes diretórios de bibliotecas e arquivos de cabeçalho para a compilação e quatro pastas:

1. **Build** - que contém todos os arquivos intermediários e/ou de saída gerados durante todo o processo de compilação.
2. **Fase1** - que se encontra subdividida em três pastas:
   (a) **Engine** - utiliza a biblioteca *OpenGL* para renderizar gráficos 3D, lê igualmente um arquivo *XML* que descreve um cenário com uma ou mais formas sólidas e as renderiza numa janela 3D interativa. As formas sólidas são definidas como um conjunto de triângulos num espaço 3D, e o programa utiliza funções matemáticas para calcular a posição da camera virtual e a perspectiva da renderização. O código também define outras funções auxiliares, como, por exemplo, uma função para ler pontos de um arquivo e outra para desenhar um sistema de coordenadas.
   (b) **Generator** - que possui todos os ficheiros relativos ao desenho das quatro figuras pedidas, bem como, um *main* que recebe os argumentos na linha de comandos para desenhar as figuras com diferentes parâmetros de entrada e salva as coordenadas dos pontos gerados num arquivo.
   (c) **Utils** - que têm as bibliotecas e/ou ficheiro que posteriormente servirão de auxílio para a construção e/ou representação das figuras.
3. **Include** - que engloba todos os *headers* dos ficheiros utilizados, como, por exemplo, *ponto.hpp*.
4. **Toolkits** - que engloba os arquivos *il.h (DevIL)*, *glew.h* e *glut.h*, ou seja, uma coleção de bibliotecas de ferramentas de desenvolvimento para criação de gráficos em 3D, permitindo manipulação de imagens, carregamento de texturas, extensões *OpenGL* e janelas em múltiplas plataformas.

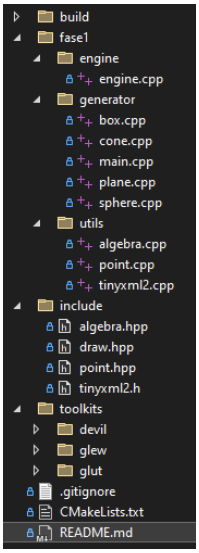


Figura 1: Estrutura do Projeto

3 Generator

O principal objetivo deste gerador é usar a informação introduzida pelo utilizador para criar ficheiros com pontos no espaço 3D. O utilizador poderá gerar uma das seguintes quatro figuras:

1. **Cubo**
2. **Cone**
3. **Esfera**
4. **Plano**

Figuras estas que são representadas usando triângulos e que serão mais exploradas a seguir.

De modo a facilitar a representação de dados, criamos uma estrutura *Point* que representa um ponto num referencial tridimensional com coordenadas cartesianas. Após a inserção do comando pelo utilizador, o *Generator* cria um ficheiro com o nome especificado, no qual consta o número total de pontos que constituem a figura, tal como as coordenadas de cada ponto, calculadas pela função *draw* da figura respetiva, que retorna um vetor de *Points*.

De seguida, vamos explicar sucintamente todo o processo que leva à construção das respetivas figuras.

3.1 Point

Tal como foi mencionado anteriormente, o propósito desta classe é a representação de um ponto no espaço tridimensional.

A classe possui um construtor que recebe três valores *double* correspondentes às coordenadas x, y e z do ponto, bem como, métodos para acessar e alterar cada coordenada individualmente, entre outros.

```cpp
#include <cmath>
#include <sstream>

#include "point.hpp"

Point::Point() {}

Point::Point(double x, double y, double z) {
    _x = x;
    _y = y;
    _z = z;
}

double Point::x() { return _x; }
double Point::y() { return _y; }
double Point::z() { return _z; }

Point Point::invertX() { return Point(-1 * _x, _y, _z); }

Point Point::invertY() { return Point(_x, -1 * _y, _z); }

Point Point::invertZ() { return Point(_x, _y, -1 * _z); }

void Point::setX(double x) { _x = x; }

void Point::setY(double y) { _y = y; }

void Point::setZ(double z) { _z = z; }

Point Point::normalize() {
    double N = sqrt(pow(_x,2)+pow(_y,2)+pow(_z,2));
    return N == 0 ? Point(_x,_y,_z) : Point(_x/N, _y/N, _z/N);
}
```

Figura 2: Ficheiro *Point.cpp*

3.2 Plane

O **Plane**, a figura mais simples, está no plano *xOz* do referencial, centrado em (0, 0), ou seja, na origem e as suas dimensões estão subdivididas pelos eixos X e Z.

Para a sua construção tem de se analisar primeiro os argumentos que lhe são passados.

```
std::vector<Point> drawPlane(double side, int div)
```

Figura 3: Cabeçalho *drawPlane*

Primeiramente, temos *side* que corresponde ao tamanho da aresta do plano e, de seguida, temos as *div*, ou seja, o número de divisões em cada eixo. Posto isto resta gerar o algoritmo para determinar a posição dos vértices necessários à construção do mesmo.

```
#include <vector>
#include "point.hpp"

std::vector<Point> drawPlane(double side, int div) {
    std::vector<Point> plane;
    double half = side / 2;

    Point p1 = Point(-half, 0, -half);
    Point p2 = p1.invertZ();
    Point p3 = p2.invertX();
    Point p4 = p1.invertX();

    double delta = side / div;

    for (int i = 0; i < div; i++) {
        for (int j = 0; j < div; j++) {
            double x1 = -half + i * delta;
            double x2 = x1 + delta;
            double z1 = -half + j * delta;
            double z2 = z1 + delta;

            Point p1(x1, 0, z1);
            Point p2(x1, 0, z2);
            Point p3(x2, 0, z2);
            Point p4(x2, 0, z1);

            plane.push_back(p1);
            plane.push_back(p2);
            plane.push_back(p4);

            plane.push_back(p2);
            plane.push_back(p3);
            plane.push_back(p4);
        }
    }

    return plane;
}
```

Figura 4: Ficheiro *Plane.cpp*

Tal como é possível verificar pelo código da Figura 4, inicialmente é definido:

1. uma variável **half** - que representa metade do lado do plano.
2. quatro pontos - **p1**, **p2**, **p3**, **p4** - que formam um quadrado no plano.
3. uma variável **delta** - que representa o tamanho da subdivisão em cada eixo.

A partir da iteração de dois *loops*, são calculados os vértices para cada subdivisão do plano, formando retângulos menores. Dentro dos *loops*, são definidas as coordenadas dos vértices do quadrado atual, a partir do índice de iteração e do tamanho da subdivisão. Com recurso a essas coordenadas, são criados quatro novos pontos que formam o quadrado, e esses pontos são adicionados ao vetor final *plane*.

Por fim, o vetor *plane* é retornado, contendo todos os pontos que formam o plano subdividido.

3.3 Cube

O **Cube** é uma versão mais complexa do *Plane*. Tal como a figura geométrica anterior, o cubo encontra-se centrado na origem e as suas dimensões estarão sub-divididas pelos respetivos eixos.

No entanto, o grupo de trabalho decidiu não usar a função *drawPlane* já criada, pois isso implicaria mudar os argumentos recebidos.

A função utilizada para criar a figura geométrica é a *drawCube* e necessita de duas informações adicionais: *length*, o comprimento da lateral e *div*, o número de divisões que esta terá.

```
std::vector<Point> drawBox(double length, int div)
```

Figura 5: Cabeçalho *drawBox*

O cubo é composto por seis quadrados iguais, cada um sendo uma das faces e subdividido da mesma maneira que o plano. Todas as subdivisões terão o mesmo tamanho e serão compostas por dois triângulos, tal como é possível ver na Figura 6.

```
#include <vector>
#include "point.hpp"

std::vector<Point> drawBox(double length, int div) {

    double mov = length / div;
    Point p1, p2, p3, p4;
    std::vector<Point> v;
    double offset = length / 2.0;

    // draw the top face
    for (double z = -offset; z < offset - 0.0001; z += mov) { ... }

    // draw the bottom face
    for (double z = -offset; z < offset - 0.0001; z += mov) { ... }

    // draw the front face
    for (double y = -offset; y < offset - 0.0001; y += mov) { ... }

    // draw the back face
    for (double y = -offset; y < offset - 0.0001; y += mov) { ... }

    // draw the left face
    for (double y = -offset; y < offset - 0.0001; y += mov) { ... }

    // draw the right face
    for (double x = -offset; x < offset - 0.0001; x += mov) { ... }

    return v;
}
```

Figura 6: Ficheiro *Box.cpp*

A abordagem seguida pelo grupo foi inicialmente criar:

1. **v** - um vetor vazio da classe *Point* que tem como objetivo armazenar todas as coordenadas do cubo.
2. **mov** - que é usada para o armazenar o valor do tamanho de cada movimento em cada direção ao desenhar um cubo.
3. quatro pontos - **p1**, **p2**, **p3**, **p4**.
4. **offset** - que é metade do valor do comprimento do cubo.

Posteriormente, para desenhar o cubo, são utilizados seis *loops for*, cada um deles responsável por desenhar cada uma das faces do cubo.

Cada face é desenhada dividindo-a em retângulos de tamanho *mov*, que são compostos por dois triângulos. Para cada retângulo, são calculados os quatro vértices que o compõem e são adicionados ao vetor de pontos *v*.

Os *loops* são organizados de tal forma que percorrem cada coordenada do espaço - X, Y, Z - uma vez, desenhando as faces correspondentes. As faces laterais, esquerda e direita, são desenhadas utilizando as coordenadas X e Z, enquanto as restantes faces são desenhadas utilizando as coordenadas X e Y ou Y e Z.

Nota:

No código fornecido, em todos os *loops* existe a seguinte linha:

```
for (double z = -offset; z < offset - 0.0001; z += mov)
```

Figura 7: *Loop*

A adição de **-0.0001** é uma forma de compensar erros de precisão de ponto flutuante que podem ocorrer durante a execução do programa. Quando se trata de valores de ponto flutuante, existem limitações na precisão que podem levar a diferenças em cálculos muito pequenos, como no caso dos *loops*.

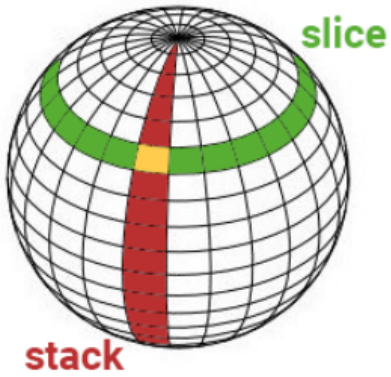
Esta pequena subtração é, portanto, uma maneira de lidar com tais imprecisões, garantindo que a iteração não ocorra em valores maiores do que o limite desejado. Caso contrário, sem essa correção, as iterações poderiam ocorrer em um valor ligeiramente maior que *offset*, o que poderia levar a resultados indesejados e potencialmente erros no programa.

3.4 Sphere

Centrada na origem, a **Sphere** é criada a partir da função *drawSphere*, que necessita de três informações adicionais:

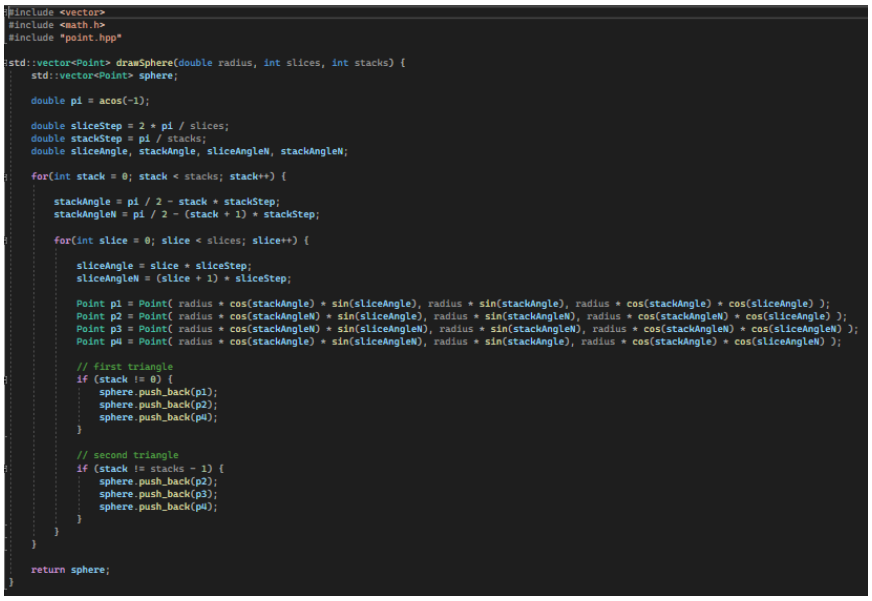
```
std::vector<Point> drawSphere(double radius, int slices, int stacks)
```

Figura 8: Cabeçalho *drawSphere*



1. Raio
2. Número de *slices* representam o número de divisões verticais necessárias
3. Número de *stacks* representam o número de divisões horizontais necessárias

Figura 9: Representação da Esfera



```cpp
#include <vector>
#include <math.h>
#include "point.hpp"

std::vector<Point> drawSphere(double radius, int slices, int stacks) {
    std::vector<Point> sphere;

    double pi = acos(-1);

    double sliceStep = 2 * pi / slices;
    double stackStep = pi / stacks;
    double sliceAngle, stackAngle, sliceAngleN, stackAngleN;

    for(int stack = 0; stack < stacks; stack++) {

        stackAngle = pi / 2 - stack * stackStep;
        stackAngleN = pi / 2 - (stack + 1) * stackStep;

        for(int slice = 0; slice < slices; slice++) {

            sliceAngle = slice * sliceStep;
            sliceAngleN = (slice + 1) * sliceStep;

            Point p1 = Point( radius * cos(stackAngle) * sin(sliceAngle), radius * sin(stackAngle), radius * cos(stackAngle) * cos(sliceAngle) );
            Point p2 = Point( radius * cos(stackAngleN) * sin(sliceAngle), radius * sin(stackAngleN), radius * cos(stackAngleN) * cos(sliceAngle) );
            Point p3 = Point( radius * cos(stackAngleN) * sin(sliceAngleN), radius * sin(stackAngleN), radius * cos(stackAngleN) * cos(sliceAngleN) );
            Point p4 = Point( radius * cos(stackAngle) * sin(sliceAngleN), radius * sin(stackAngle), radius * cos(stackAngle) * cos(sliceAngleN) );

            // first triangle
            if (stack != 0) {
                sphere.push_back(p1);
                sphere.push_back(p2);
                sphere.push_back(p4);
            }

            // second triangle
            if (stack != stacks - 1) {
                sphere.push_back(p2);
                sphere.push_back(p3);
                sphere.push_back(p4);
            }
        }
    }

    return sphere;
}
```

Figura 10: Ficheiro *Sphere.cpp*

O código da Figura 10 começa por inicializar um *array* vazio *sphere* que será, posteriormente, preenchido com os pontos que representam a esfera. Em seguida, o valor de pi é definido como o arco cosseno de -1, isso é feito porque *acos(-1)* retorna pi. Por fim, inicializa seis diferentes variáveis do tipo *double*:

1. **sliceStep** - corresponde ao ângulo em radianos entre cada *slice* horizontal da esfera. A fórmula usada para calcular *sliceStep* é *2 * pi*, ou seja, o comprimento da circunferência da esfera dividido pelo número de *slices* que foram especificadas como entrada para a função. Esta etapa é importante para determinar o espaçamento entre as fatias horizontais da esfera.
2. **stackStep** - corresponde ao ângulo em radianos entre cada *stack* vertical da esfera. A fórmula usada para calcular *stackStep* é pi, a metade do comprimento da circunferência da esfera, dividido pelo número de *stacks* que foram especificadas como entrada para a função. Esta etapa é, novamente, importante para determinar o espaçamento entre as *stacks* verticais da esfera.
3. As restantes variáveis de ângulo **sliceAngle**, **stackAngle**, **sliceAngleN**, **stackAngleN** são inicializadas posteriormente dentro do *loop* principal que gera os pontos da esfera. Essas variáveis são usadas para calcular as coordenadas de cada ponto da esfera e são atualizadas a cada iteração do *loop*, para se moverem ao longo da "grade"de *slices* e *stacks* da esfera.

Após tudo isto, a função usa dois *loops* for aninhados para gerar cada ponto da esfera. O *loop* externo itera pelas *stacks* da esfera, da base até o topo, e o *loop* interno itera pelas *slices*, da frente para trás.

Para cada combinação de *slice* e *stack*, a função usa algumas fórmulas matemáticas para calcular as coordenadas dos quatro pontos que formam os dois triângulos que representam a superfície da esfera naquela região. Essas fórmulas usam as funções trigonométricas seno e cosseno, que são usadas para calcular as coordenadas X, Y e Z de cada ponto.

No final de tudo, a função retorna o vetor *sphere* completo, que representa a esfera em 3D.

3.5 Cone

O Cone encontra-se com a base centrada a (0,0), ou seja, na origem e cresce segundo o eixo Y.

Para a sua construção é necessário, em primeiro lugar, analisar os argumentos que lhe são passados. Inicialmente temos o *radius*, que se refere ao raio da base, a *height*, que diz respeito à altura do cone, a *slices*, que representam o número de divisões da base e por fim temos a *stacks*, que correspondem ao número de divisões de cada face lateral do cone.

```
std::vector<Point> drawCone(double radius, double height, int slices, int stacks)
```

Figura 11: Cabeçalho *drawCone.cpp*

Posto isto, resta gerar o algoritmo para determinar a posição dos vértices necessários à construção da figura.

```
#include <vector>
#include "point.hpp"
#include "algebra.hpp"

#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

std::vector<Point> drawCone(double radius, double height, int slices, int stacks) {

    double slicedAlpha = (2 * M_PI) / slices;
    double stackHeight = height / stacks;
    Point p1, p2, p3, p4;

    std::vector<Point> v;

    p3 = Point(0.0f, 0.0f, 0.0f);
    for (int slice = 0; slice < slices; slice++) {
        p1 = Point(getX(radius, (slice + 1) * slicedAlpha, 0), 0.0f, getZ(radius, (slice + 1) * slicedAlpha, 0));
        p2 = Point(getX(radius, slice * slicedAlpha, 0), 0.0f, getZ(radius, slice * slicedAlpha, 0));

        v.push_back(p1);
        v.push_back(p2);
        v.push_back(p3);
    }

    for (int stack = 0; stack < stacks - 1; stack++) {
        for (int slice = 0; slice < slices; slice++) {

            p1 = Point(getX(radius - stack * radius / stacks, slice * slicedAlpha, 0), stackHeight * stack, getZ(radius - stack * radius / stacks, slice * slicedAlpha, 0));
            p2 = Point(getX(radius - stack * radius / stacks, (slice + 1) * slicedAlpha, 0), stackHeight * stack, getZ(radius - stack * radius / stacks, (slice + 1) * slicedAlpha, 0));
            p3 = Point(getX(radius - (stack + 1) * radius / stacks, (slice + 1) * slicedAlpha, 0), stackHeight * (stack + 1), getZ(radius - (stack + 1) * radius / stacks, (slice + 1) * slicedAlpha, 0));
            p4 = Point(getX(radius - (stack + 1) * radius / stacks, slice * slicedAlpha, 0), stackHeight * (stack + 1), getZ(radius - (stack + 1) * radius / stacks, slice * slicedAlpha, 0));

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

            v.push_back(p1);
            v.push_back(p3);
            v.push_back(p4);
        }
    }


    p3 = Point(0.0f, height, 0.0f);
    for (int slice = 0; slice < slices; slice++) {
        p1 = Point(getX(radius - ((stacks - 1) * radius / stacks), slice * slicedAlpha, 0), height - height / stacks, getZ(radius - ((stacks - 1) * radius / stacks), slice * slicedAlpha, 0));
        p2 = Point(getX(radius - ((stacks - 1) * radius / stacks), (slice + 1) * slicedAlpha, 0), height - height / stacks, getZ(radius - ((stacks - 1) * radius / stacks), (slice + 1) * slicedAlpha, 0));

        v.push_back(p1);
        v.push_back(p2);
        v.push_back(p3);
    }

    return v;
}
```

Figura 12: Ficheiro *Cone.cpp*

Como é possível observar pela Figura 12, a função começa por criar as coordenadas dos triângulos da base do cone. Para isso, cria três pontos, os pontos **p1** e **p2** são coordenadas num círculo com centro no ponto (0,0,0) e raio *radius* e o ponto **p3** que é simplesmente o ponto na origem. A distância angular entre esses pontos é dada por **slicedAlpha**, que é definida como *2 * M_PI / slices* e a distância vertical, dada por **stackHeight**, é calculada da seguinte maneira: *height / stacks*.

Em seguida, a função cria as *stacks* do cone. Para cada *stack*, a função cria *slices* que são conectadas à *stack* abaixo. Para cada *slice*, a função cria quatro pontos - p1, p2, p3 e p4. Os pontos p1 e p2 são as coordenadas de um ponto no círculo da *slice* atual, enquanto os pontos p3 e p4 são as coordenadas de um ponto no círculo da *slice* abaixo.

A distância angular entre p1 e p2 e entre p3 e p4 é dada por *slicedAlpha* enquanto que a distância vertical entre as *slices* é dada por *stackHeight*. O raio da *slice* atual é calculado: *radius - stack * radius / stacks*, onde stack é o índice da *stack* atual. Os pontos p1, p2, p3 e p4 são usados para criar dois triângulos, que são adicionados ao vetor v.

Finalmente, a função cria a ponta do cone. Para isso, ela cria novos pontos p1, p2 e p3. Os pontos p1 e p2 são as coordenadas de um ponto no círculo da última *slice*. O raio do círculo é calculado como *radius - ((stacks - 1) * radius / stacks)*. O ponto p3 é simplesmente o ponto (0, *height*, 0). Os pontos p1, p2 e p3 são usados da mesma maneira para criar um triângulo, que é adicionado ao vetor v.

No final, a função retorna o vetor v que contém as coordenadas dos triângulos que formam o modelo do cone.

4 Engine

O *Engine* é a aplicação responsável por renderizar as figuras construídas previamente pelo *Generator*. Para isso, recebe um ficheiro *XML*, colocado no primeiro argumento na execução do programa.

```
<world>
    <window width="512" height="512" />
    <camera>
            <position x="5" y="-2" z="3" />
            <lookAt x="0" y="0" z="0" />
            <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
        <group>
                <models>
                        <model file="cone_1_2_4_3.3d" /> <!-- generator cone 1 2 4 3 cone_1_2_4_3.3d -->
                </models>
        </group>
</world>
```

Figura 13: Exemplo de Ficheiro *XML*

Os ficheiros *XML* contêm diversas informações. Nesta primeira fase apenas focamos o programa na leitura do tamanho da janela (*window size*), da posição da camera e o *group* que dispõe da localização dos ficheiros 3D gerados pelo Generator.

De modo a conseguirmos gerir estes ficheiros *XML* foi utilizada a biblioteca *tinyXML2*. O código responsável pela implementação do *parse* dos ficheiros *XML* encontra-se no ficheiro *Engine.cpp* (Figura 14).

Inicialmente começamos por verificar se o caminho que o utilizador introduziu é valido. Caso não seja o programa termina com o respetivo erro.

Tendo conseguido identificar e carregar o ficheiro com sucesso podemos proceder com o *parser* propriamente dito.

Começamos por verificar se existe a *tag world* e, caso exista prosseguimos para identificar os ficheiros 3D, contidos na *tag group->models* e inserimos cada objeto no vetor para ser entregue à placa gráfica.

Após ler e interpretar todos os ficheiros 3D, são lidas as definições da camera (posição, *lookAt*, *up* e *projection*) e guardadas nas estruturas criadas para o efeito. Por fim e de igual forma, são registadas as informações do tamanho da janela.

Com todos os elementos necessários dos ficheiro *XML* guardados, para finalizar as tarefas da *engine* é desenhado modelo descrito no ficheiro com a ajuda do *OpenGL*.

```
XMLElement* world = doc.FirstChildElement("world");
if(world == NULL) {
    puts("<world> não foi encontrado.");
    return 1;
}

XMLElement* group = world->FirstChildElement("group");
if (group != NULL) {
    XMLElement* models = group->FirstChildElement("models");
    if (models != NULL) {
        XMLElement* model = models->FirstChildElement();

        while (model) {
            if (!strcmp(model->Name(), "model")) {
                solids.push_back(vectorize(model->Attribute("file")));
            }
            model = model->NextSiblingElement();
        }
    }

}

XMLElement* cam = world->FirstChildElement("camera");

if (cam) {
    XMLElement* pos = cam->FirstChildElement("position");

    if (pos) {
        parsePoint(pos, &camera.pos);
    }

    XMLElement* lookAtXml = cam->FirstChildElement("lookAt");
    if (lookAtXml) {
        Point p;
        parsePoint(lookAtXml, &p);
        double x = p.x() - camera.pos.x();
        double y = p.y() - camera.pos.y();
        double z = p.z() - camera.pos.z();
        double r = sqrt(x * x + y * y + z * z);
        camera.alpha = M_PI + atan(x / z);
        camera.beta = asin(y / r);
        if (camera.beta > M_PI_2)
            camera.beta -= M_PI;
    }

    XMLElement* upXml = cam->FirstChildElement("up");
    if (upXml) {
        parsePoint(upXml, &camera.up);
    }

    XMLElement* projectionXml = cam->FirstChildElement("projection");
    if (projectionXml) {
        //parseProjection(projXml, &camera.proj);
    }
}

XMLElement* window = world->FirstChildElement("window");

if (window) {
    windowSize.width = stoi(window->Attribute("width"));
    windowSize.height = stoi(window->Attribute("height"));
}
```

Figura 14: *Engine.cpp*

5 Keyboard Functions

Esta função é um *callback* do *GLUT* que é executado quando uma tecla do teclado é pressionada. Esta recebe três parâmetros:

1. Tecla pressionada - *key*
2. Coordenada X
3. Coordenada Y

A função começa por armazenar a posição atual da camera em três variáveis: *aX*, *aY* e *aZ*. Estas variáveis são usadas para atualizar a posição da camera com base na tecla pressionada.

O *switch case* testa a tecla pressionada e executa o código correspondente. Na seguinte tabela é possível verificar todos os casos e o respetivo movimento associado.

Tabela 1: *Keyboard Functions*

Key	**Movimento**
a / d	Camera move-se para a esquerda e para a direita, respetivamente, no plano XZ, mantendo a altura constante.
w / s	Camera move-se para frente e para trás no plano XZ, mantendo a altura constante, mas também ajustam a altura (*aY*) com base no ângulo de visão da camera (*camera.beta*).
r / f	Camera move-se para cima e para baixo, respetivamente, no eixo Y.
1 / 2	Giram a camera ao redor do ponto para onde ela está olhando, mais concretamente, giram a camera no plano XZ, em torno do eixo Y.
3 / 4	Giram a camera ao redor do ponto para onde ela está olhando, mais concretamente, giram a camera no plano YZ, em torno do eixo X.

Depois de atualizar a posição e a orientação da camera, a função chama *glutPostRedisplay()* para sinalizar ao *GLUT* que a cena precisa ser redesenhada com a nova posição da camera.

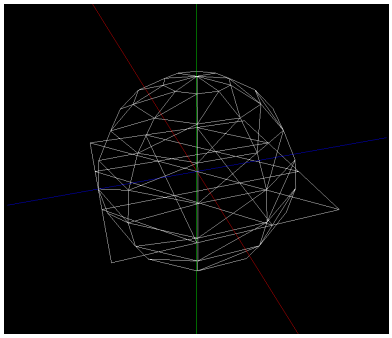
```
void keyboardFunc(unsigned char key, int x, int y) {
    double aX = camera.pos.x(), aY = camera.pos.y(), aZ = camera.pos.z();
    switch(key) {
        case 'a':
            aX += 0.6 * cos(camera.alpha);
            aZ -= 0.6 * sin(camera.alpha);
            break;
        case 'd':
            aX -= 0.6 * cos(camera.alpha);
            aZ += 0.6 * sin(camera.alpha);
            break;
        case 's':
            aX -= 0.6 * sin(camera.alpha) * cos(camera.beta);
            aZ -= 0.6 * cos(camera.alpha) * cos(camera.beta);
            aY -= 0.6 * sin(camera.beta);
            break;
        case 'w':
            aX += 0.6 * sin(camera.alpha) * cos(camera.beta);
            aZ += 0.6 * cos(camera.alpha) * cos(camera.beta);
            aY += 0.6 * sin(camera.beta);
            break;
        case 'r':
            aY += 0.6;
            break;
        case 'f':
            aY -= 0.6;
            break;
        case '1':
            camera.alpha -= M_PI / 64;
            break;
        case '2':
            camera.alpha += M_PI / 64;
            break;
        case '3':
            camera.beta -= M_PI / 64;
            break;
        case '4':
            camera.beta += M_PI / 64;
            break;
    }

    camera.pos = Point(aX, aY, aZ);
    if (camera.beta > M_PI_2)
        camera.up = Point(0, -camera.up.y(), 0);

    glutPostRedisplay();
}
```

Figura 15: Função *KeyboardFunc*

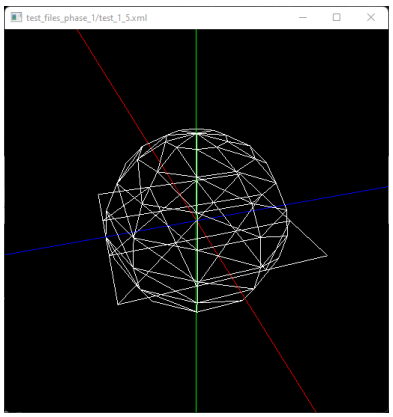
6 Resultados Obtidos

Neste tópico iremos mostrar a representação gráfica de cada uma das figuras geométricas faladas anteriormente, em comparação aos teste fornecidos pela equipa docente.

6.1 Plane

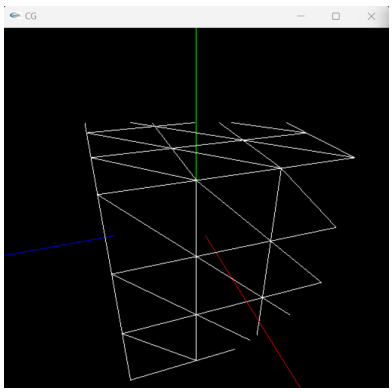


(a) Figura Geométrica - Plane

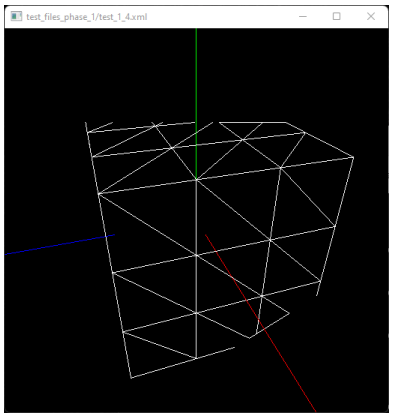


(b) Teste Plane

6.2 Cube

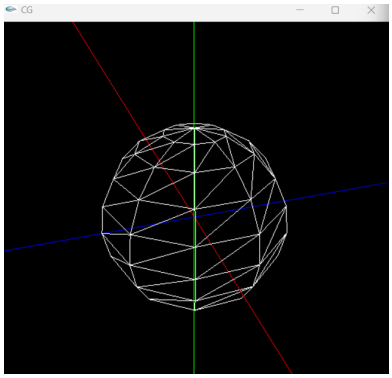


(a) Figura Geométrica - Box

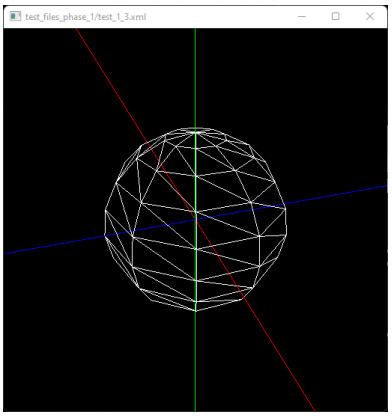


(b) Teste Box

6.3 Sphere

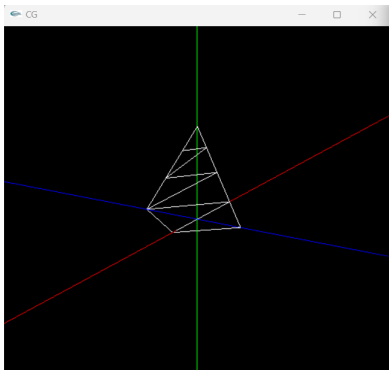


(a) Figura Geométrica - Sphere

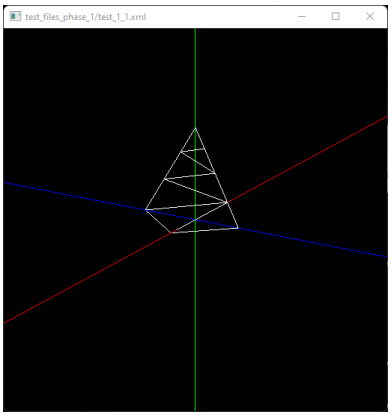


(b) Teste Sphere

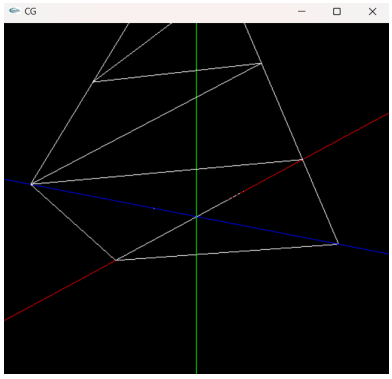
6.4 Cone



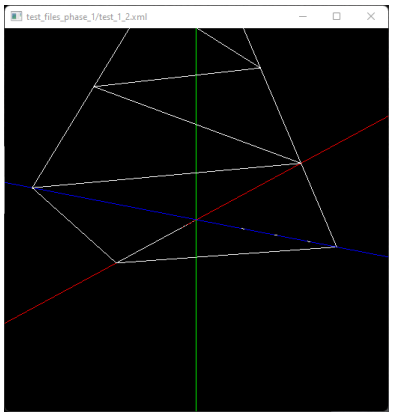
(a) Figura Geométrica - Cone 1



(b) Teste Cone 1



(a) Figura Geométrica - Cone 2



(b) Teste Cone 2

7 Conclusão

Nesta primeira fase do Trabalho Prático foi possível relembrar e consolidar alguns conceitos básicos da Linguagem C++, assim como conhecimentos adquiridos nas aulas, tanto a nível prático como teórico, nomeadamente, no que se refere à utilização de diversas funções da biblioteca *GLUT* do *OpenGL* e o processo por trás da construção de figuras simples tridimensionais.

Em suma, foram desenvolvidas as duas aplicações chave durante esta primeira fase, o **Generator** e o **Engine**, que por sua vez são capazes de construir e exibir todas as Primitivas Gráficas requisitadas. Com isto, é possível dar início à segunda fase do trabalho, que visa melhorar e adicionar conteúdo extra ao Engine.

8 Referências Bibliográficas / *Websites*

1. **OpenGL 2.1 Reference Pages**
   http://www.lighthouse3d.com/tutorials/glut-tutorial/
2. **Lighthouse 3D GLUT Tutorial**
   http://www.lighthouse3d.com/tutorials/glut-tutorial/
3. **StackOverFlow**
   https://stackoverflow.com/
4. **CPlusPlus**
   https://cplusplus.com/
5. **GeeksForGeeks**
   https://www.geeksforgeeks.org/