Universidade do Minho
Escola de Ciências

Computação Gráfica

Ciências da Computação
Fase 1

Bruno Fernandes (A95972) | Tiago Silva (A97450) | Tomás Pereira (A97402)

10 de março de 2023

Índice

Capítulo 1

Introdução

Este relatório foi elaborado no âmbito da primeira fase do trabalho prático da unidade curricular de Computação Gráfica, no qual foi proposto o desenvolvimento de duas aplicações: uma para gerar ficheiros com a informação do vértices de cada modelo (*generator*) e outra que lê um ficheiro *XML* com as configurações e que reproduz os modelos (*engine*). No desenvolvimento foram utilizadas duas ferramentas, o *OpenGL* e o *C++*, à imagem do que é feito nas aulas práticas.

Capítulo 2

Generator

A função do *generator* é criar, a partir de um conjunto de parâmetros, as coordenadas de cada ponto que formam a figura, num ficheiro com extensão *.3d* que, posteriormente, será usado pela *engine*.

2.1 Arquitetura do Código

A nossa implementação do *generator* contém uma função *main* que faz a leitura do input. O input deve seguir um dos seguintes exemplos:

```
C:\>generator plane 1 3 plane.3d

C:\>generator box 2 3 box.3d

C:\>generator cone 1 2 4 3 cone.3d

C:\>generator sphere 1 10 10 sphere.3d
```

Figura 2.1: `Exemplos de comandos`

Esta função, seleciona qual é a primitiva que será calculada e verifica se o número de argumentos dado é o correto. Passando este teste, é executada a função que calcula as coordenadas correspondentes à primitiva. No início de cada uma das funções que calculam os pontos, é criada uma string para onde são copiados os pontos, os pontos sao constituidos por 3 coordenadas separadas por um espaço. No fim, esta string é escrita para um ficheiro *.3d* através da função *wrtfile*. A função *wrtfile* abre o ficheiro de escrita, caso exista, ou cria-o, caso não exista. De seguida, escreve no ficheiro todos os pontos que estão na string recebida como parâmetro. O ficheiro *.3d* gerado tem o seguinte formato:

```
-0.181636 0.951057 0.25
2.01381e-07 0.951057 0.309017
0 0.951057 0.309017
0.345491 0.809017 0.475528
0.181636 0.951057 0.25
0 0.951057 0.309017
0 0.809017 0.587785
```

Figura 2.2: `Exemplo do formato de ficheiro .3d`

2.2 Primitivas Gráficas

2.2.1 Plano

Para gerar o plano recebemos como *input* o tamanho (*length*) e o número de divisões (*divisions*). Através desses parâmetros, iteramos o comprimento e a largura do plano, com dois ciclos *for*, onde cada iteração gera dois triângulos. Como o plano tem de ficar centrado na origem, fixamos a variável y a 0 e variamos os valores de x e y de -*length/2* até *length/2*, incrementando *length/divisions* a cada iteração.

```
void drawplane(int length, int divisions, std::string name) {
    std::stringstream s;

    float init = -((float)length / 2.0f);
    float unidade = (float)length / (float)divisions;
    float x1, x2, z1, z2;
    for (int i = 0; i < divisions; i++) {
        for (int j = 0; j < divisions; j++) {
            x1 = init + i * unidade;
            z1 = init + j * unidade;
            x2 = init + (i + 1) * unidade;
            z2 = init + (j + 1) * unidade;

            // 1 triangulo
            s << x1 << ' ' << 0 << ' ' << z1 << '\n';
            s << x1 << ' ' << 0 << ' ' << z2 << '\n';
            s << x2 << ' ' << 0 << ' ' << z2 << '\n';

            //2 triangulo
            s << x2 << ' ' << 0 << ' ' << z1 << '\n';
            s << x1 << ' ' << 0 << ' ' << z1 << '\n';
            s << x2 << ' ' << 0 << ' ' << z2 << '\n';
        }
    }
    wrtfile(s.str(), name);
}
```

Figura 2.3: `Função que calcula as coordenadas do plano`

2.2.2 Cubo

Para gerar o cubo utilizamos um método idêntico ao anterior, uma vez que o cubo é composto por seis planos. À semalhança do que fizemos anteriormente, utilizamos dois ciclos *for* para iterar o comprimento e a largura para gerar dois planos paralelos. Repetimos o processo três vezes para concluir a construção do cubo. Para além disso, tivemos o cuidado ao definir a ordem dos pontos para respeitar a orientação de cada face, de forma a serem visíveis exteriormente.

```
void drawbox(int length, int divisions, std::string name) {
    std::stringstream s;

    float init = -((float)length / 2.0);
    float unidade = (float)length / (float)divisions;
    float x1, x2, y1, y2, z1, z2;
    float altura = (float)length / 2.0;

    for (int i = 0; i < divisions; i++) {
        for (int j = 0; j < divisions; j++) {
            x1 = init + i * unidade;
            z1 = init + j * unidade;
            x2 = init + (i + 1) * unidade;
            z2 = init + (j + 1) * unidade;

            //face de cima
            // 1 triangulo
            s << x1 << ' ' << altura << ' ' << z1 << '\n';
            s << x1 << ' ' << altura << ' ' << z2 << '\n';
            s << x2 << ' ' << altura << ' ' << z2 << '\n';

            //2 triangulo
            s << x2 << ' ' << altura << ' ' << z1 << '\n';
            s << x1 << ' ' << altura << ' ' << z1 << '\n';
            s << x2 << ' ' << altura << ' ' << z2 << '\n';

            //face de baixo
            //1 triangulo
            s << x1 << ' ' << -altura << ' ' << z1 << '\n';
            s << x2 << ' ' << -altura << ' ' << z2 << '\n';
            s << x1 << ' ' << -altura << ' ' << z2 << '\n';

            //2 triangulo
            s << x2 << ' ' << -altura << ' ' << z1 << '\n';
            s << x2 << ' ' << -altura << ' ' << z2 << '\n';
            s << x1 << ' ' << -altura << ' ' << z1 << '\n';
        }
    }
```

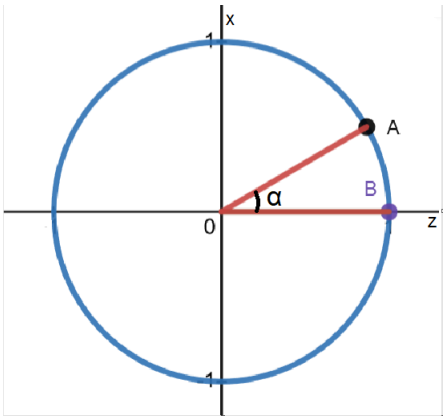
```
for (int i = 0; i < divisions; i++) {
    for (int j = 0; j < divisions; j++) {
        x1 = init + i * unidade;
        y1 = init + j * unidade;
        x2 = init + (i + 1) * unidade;
        y2 = init + (j + 1) * unidade;

        //face da esquerda
        // 1 triangulo
        s << x1 << ' ' << y2 << ' ' << altura << '\n';
        s << x1 << ' ' << y1 << ' ' << altura << '\n';
        s << x2 << ' ' << y1 << ' ' << altura << '\n';

        //2 triangulo
        s << x2 << ' ' << y2 << ' ' << altura << '\n';
        s << x1 << ' ' << y2 << ' ' << altura << '\n';
        s << x2 << ' ' << y1 << ' ' << altura << '\n';

        //face da direita
        // 1 triangulo
        s << x1 << ' ' << y2 << ' ' << -altura << '\n';
        s << x2 << ' ' << y1 << ' ' << -altura << '\n';
        s << x1 << ' ' << y1 << ' ' << -altura << '\n';

        //2 triangulo
        s << x2 << ' ' << y2 << ' ' << -altura << '\n';
        s << x2 << ' ' << y1 << ' ' << -altura << '\n';
        s << x1 << ' ' << y2 << ' ' << -altura << '\n';
    }
}
```

```
for (int i = 0; i < divisions; i++) {
    for (int j = 0; j < divisions; j++) {
        y1 = init + i * unidade;
        z1 = init + j * unidade;
        y2 = init + (i + 1) * unidade;
        z2 = init + (j + 1) * unidade;

        //face da frente
        // 1 triangulo
        s << altura << ' ' << y2 << ' ' << z2 << '\n';
        s << altura << ' ' << y1 << ' ' << z2 << '\n';
        s << altura << ' ' << y1 << ' ' << z1 << '\n';

        //2 triangulo
        s << altura << ' ' << y2 << ' ' << z1 << '\n';
        s << altura << ' ' << y2 << ' ' << z2 << '\n';
        s << altura << ' ' << y1 << ' ' << z1 << '\n';

        //face de tras
        // 1 triangulo
        s << -altura << ' ' << y2 << ' ' << z2 << '\n';
        s << -altura << ' ' << y1 << ' ' << z1 << '\n';
        s << -altura << ' ' << y1 << ' ' << z2 << '\n';

        //2 triangulo
        s << -altura << ' ' << y2 << ' ' << z1 << '\n';
        s << -altura << ' ' << y1 << ' ' << z1 << '\n';
        s << -altura << ' ' << y2 << ' ' << z2 << '\n';

    }
}
wrtfile(s.str(), name);
}
```

Figura 2.4: Função que calcula as coordenadas do cubo

2.2.3 Cone

Para gerar o cone, a nossa função recebe os parâmetros *radius*, *height*, *slices*, *stacks* e *name*. Começamos por definir a base. Para tal, utilizamos o seguinte método para cada triângulo:
Como temos um número certo de *slices*, começamos por calcular o ângulo que utilizamos para iterar as coordenadas ($\alpha = 2 * \pi / slices$).



Em seguida, fixamos um vértice na origem (0,0,0) e definimos os outros vértices, contidos na circunferência, para criar o triângulo:

$$(x1, y1, z1) = (raio * sin(angulo\ atual), 0, raio * cos(angulo\ atual))$$
$$(x2, y2, z2) = (raio * sin(angulo\ atual + 2 * \pi / slices), 0, raio * cos(angulo\ atual + 2 * \pi / slices))$$

Para gerar a parte lateral do cone, como está estava dividida em camadas (*stacks*), usamos o mesmo método para obter os vértices de cada uma das bases de cada camada. De forma a obtermos o raio e a altura (coordenada y) dos vértices da base superior utilizamos a altura do cone (*height*) e o número de camadas:

$$\text{proxH} = \text{atualH} + \text{height} \;/\; \text{stacks}$$
$$\text{proxR} = \text{atualR} - \text{radius} \;/\; \text{stacks}$$

Tendo os raios e as alturas superiores e inferiores, conseguimos, aplicando fórmulas identicas às anteriormente descritas, calcular os pontos que geram os triângulos da lateral. Agrupando estes com a orientação correta para serem visíveis exteriormente, obtemos os pontos que definem os triângulos que geram a figura pretendida.

```
void drawcone(int radius, int height, int slices, int stacks, std::string name) {
    std::stringstream s;
    float x1, x2, z1, z2;
    float atualH = 0;
    float proxH;
    float atualR = radius;
    float proxR;

    //desenhar a base
    for (int i = 0; i < slices; i++) {
        x1 = radius * sin(i * (2 * M_PI / (float)slices));
        x2 = radius * sin((i + 1) * (2 * M_PI / (float)slices));
        z1 = radius * cos(i * (2 * M_PI / (float)slices));
        z2 = radius * cos((i + 1) * (2 * M_PI / (float)slices));

        s << 0 << ' ' << 0 << ' ' << 0 << '\n';
        s << x2 << ' ' << 0 << ' ' << z2 << '\n';
        s << x1 << ' ' << 0 << ' ' << z1 << '\n';
    }

    //desenhar a lateral
    float angulo = 0;
    for (int i = 0; i < stacks; i++) {
        proxH = atualH + (float)height / (float)stacks;
        proxR = atualR - (float)radius / (float)stacks;

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

            //1 triangulo
            s << atualR * sin(angulo) << ' ' << atualH << ' ' << atualR * cos(angulo) << '\n';
            s << atualR * sin(angulo + (2 * M_PI / (float)slices)) << ' ' << atualH << ' ' << atualR * cos(angulo + (2 * M_PI / (float)slices)) << '\n';
            s << proxR * sin(angulo) << ' ' << proxH << ' ' << proxR * cos(angulo) << '\n';

            //2 triangulo
            s << proxR * sin(angulo) << ' ' << proxH << ' ' << proxR * cos(angulo) << '\n';
            s << atualR * sin(angulo + (2 * M_PI / (float)slices)) << ' ' << atualH << ' ' << atualR * cos(angulo + (2 * M_PI / (float)slices)) << '\n';
            s << proxR * sin(angulo + (2 * M_PI / (float)slices))<< ' ' << proxH << ' ' << proxR * cos(angulo + (2 * M_PI / (float)slices)) << '\n';

            angulo += (2 * M_PI / (float)slices);
        }
        atualH = proxH;
        atualR = proxR;

    }
    wrtfile(s.str(), name);

}
```

Figura 2.5: `Função que calcula as coordenadas do cone`

2.2.4 Esfera

Para gerar a esfera, temos os parâmetros *radius*, *slices*, *stacks*, *name*. De forma a percorrer toda a esfera para criar as coordenadas dos triângulos, criamos as variaveis *step_div* e *step_cam*.
A variável *step_div* percorre os graus de 0 a 2π (longitude) num intervalo de $2\pi/slices$.
A variável *step_cam* percorre os graus de 0 a π (latitude) num intervalo de $\pi/slices$.
Como esta vai ser dividida verticalmente em *stacks*, e horizontalmente em *slices*, fizemos dois ciclos, um que percorre as *stacks* e o outro que percorre as *slices*. Por cada ciclo são calculadas as coordenadas de dois triângulos.

```
void drawsphere(int radius, int slices, int stacks, std::string name) {
    std::stringstream s;
    float div;
    float camada;
    float step_div = 2 * M_PI / slices;
    float step_cam = M_PI / stacks;

    for(int i = 0; i < stacks; i++){
        camada = step_cam * i;
        for (int j=0; j < slices; j++){
            div = step_div * j;

            //fazer os dois triangulos que fazer a forma da face da esfera em cada iteração do circulo
            s << radius * sin(div) * sin(camada) << ' ' << radius * cos(camada) << ' ' << radius * sin(camada) * cos(div) << "\n";
            s << radius * sin(camada + step_cam) * sin(div + step_div) << ' ' << radius * cos(camada + step_cam) << ' ' << radius * sin(camada + step_cam) * cos(div + step_div) << "\n";
            s << radius * sin(camada) * sin(div + step_div) << ' ' << radius * cos(camada) << ' ' << radius * sin(camada) * cos(div + step_div) << "\n";

            s << radius * sin(div) * sin(camada) << ' ' << radius * cos(camada) << ' ' << radius * sin(camada) * cos(div) << "\n";
            s << radius * sin(camada + step_cam) * sin(div) << ' ' << radius * cos(camada + step_cam) << ' ' << radius * sin(camada + step_cam) * cos(div) << "\n";
            s << radius * sin(camada + step_cam) * sin(div + step_div) << ' ' << radius * cos(camada + step_cam) << ' ' << radius * sin(camada + step_cam) * cos(div + step_div) << "\n";
        }
    }
    wrtfile(s.str(), name);
}
```

Figura 2.6: `Função que calcula as coordenadas da esfera`

Capítulo 3

Engine

3.1 Arquitetura do Código

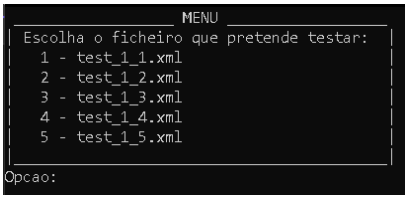
A *engine* gera uma janela a partir da informação contida num ficheiro *XML*. Para tal, utilizamos a função *readxml* que, usando o *parser tinyXML2*, abre o documento para leitura e realiza o *parsing* colocando em variáveis globais a seguinte informação:

- Janela
  - Largura (*width*)
  - Altura (*height*)
- Câmara
  - Posição da câmara (*position*)
  - Ponto para onde a câmara está a olhar (*lookAt*)
  - Inclinação da câmara (*up*)
  - Projeção (*projection*)
- Modelos
  - Nome do(s) ficheiro(s) *.3d*

A função *readxml* tem a capacidade de ler um ou mais modelos. Após o *parsing*, utilizamos a função *readFile*, que recebe como argumento o nome do modelo que contém os pontos que originarão as primitivas gráficas. Esta função lê as coordenadas contidas no ficheiro e coloca esta informação no vetor *Pontos*. Posto isto, a informação contida no vetor originará triângulos que, por sua vez, formarão a figura pretendida.

3.2 Resultados Gerados

Para facilitar a utilização da nossa implementação, decidimos criar um menu para o utilizador escolher a opção que prentende executar:



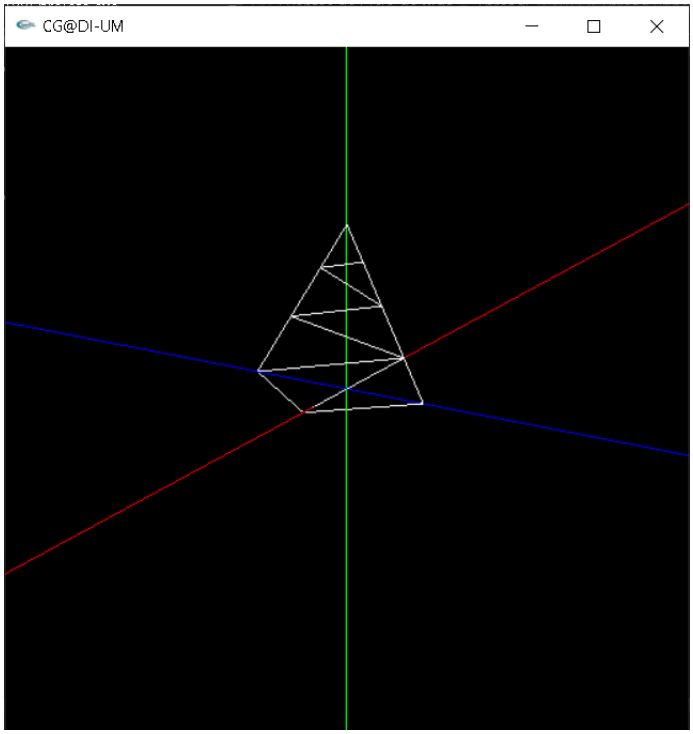


Figura 3.1: Resultado do *test_1_1.xml*

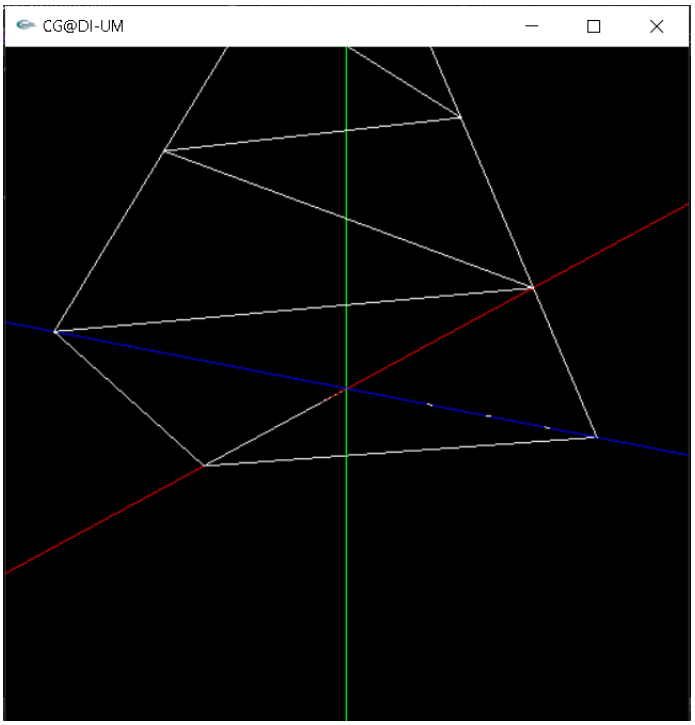


Figura 3.2: Resultado do *test_1_2.xml*

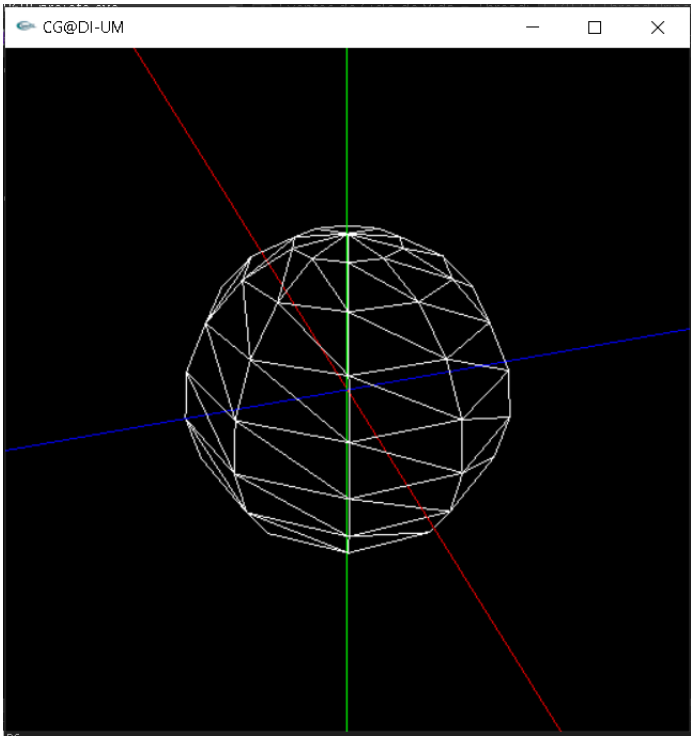


Figura 3.3: Resultado do *test_1_3.xml*

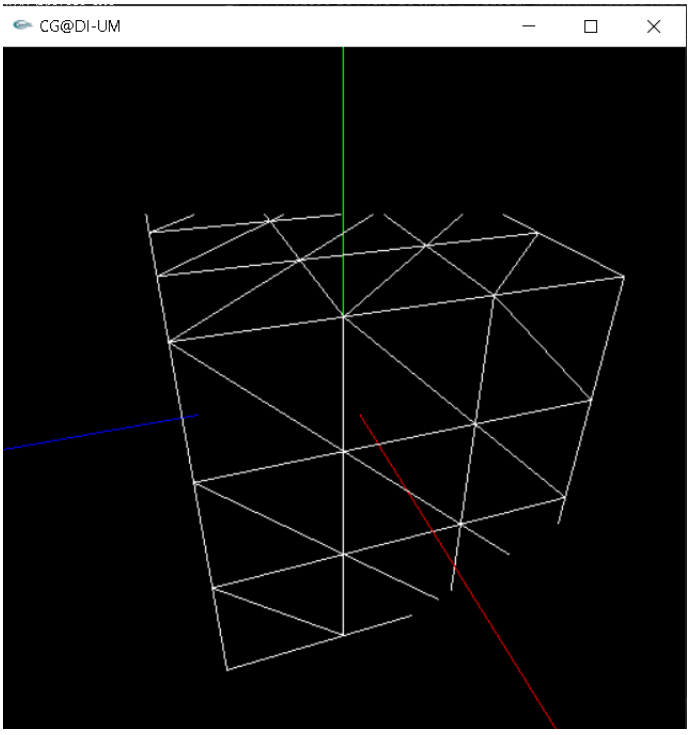


Figura 3.4: Resultado do *test_1_4.xml*

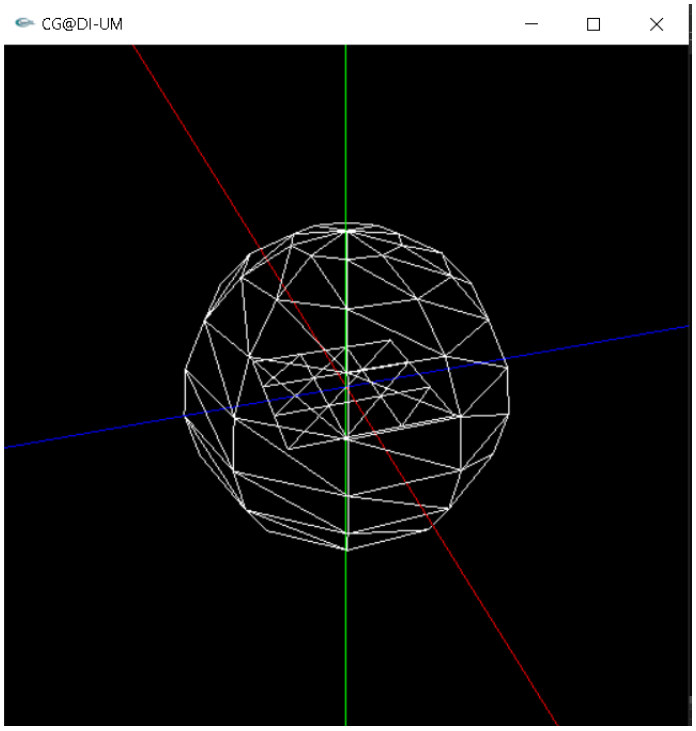


Figura 3.5: Resultado do *test_1_5.xml*

Capítulo 4

Conclusão

Após a realização da primeira fase do trabalho prático de Computação Gráfica, conseguimos conciliar e consolidar os conhecimentos adquiridos nas aulas relativos às primitivas gráficas e aos seus algoritmos. Ainda assim, surgiram algumas dificuldades, nomeadamente na realização da função geradora da esfera, e, para além disso, consideramos ter melhorias a efetuar.

Em suma, pensamos ter correspondido à exigência desta fase e com isto ter uma boa base para futuras implementações.