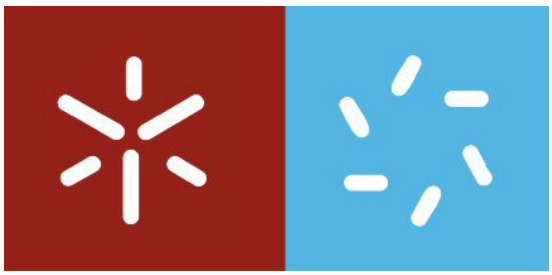

Trabalho Prático CG - 2022/2023
Fase 1 - Primitivas Gráficas

Licenciatura em Ciências da Computação
Universidade do Minho
Grupo 11

Gonçalo Silva	Nelson Almeida	Nuno Costa
(A95696)	(A95652)	(A97610)

12 de março de 2023

Conteúdo

Capítulo 1

Introdução

Nesta primeira frase do trabalho prático da UC de Computação Gráfica, foi-nos proposto o desenvolvimento de dois programas, sendo eles o *generator* e o *engine*. O *generator* tem a função de gerar um ficheiro com as informações do modelo requerido, isto é, criar um ficheiro com os vértices que constituem o modelo. Já o *engine*, é responsável pela leitura de um ficheiro de configuração (escrito em XML) e pela exibição dos modelos obtidos nessa mesma leitura. De notar que este trabalho foi desenvolvido na liguagem de programação C++, juntamente com a API de computação gráfica *OpenGL*.

Capítulo 2

Generator

2.1 Definição

O Generator através de um conjunto de parâmetros efetua os cálculos necessários para produzir a forma geométrica desejada e assim gerar um ficheiro .3d para, posteriormente, ser utilizado no Engine.

2.2 Funcionalidades

O generator tem a capacidade de gerar as formas geométricas seguintes:

- Plano:
  Cria um plano paralelo ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o comprimento das aretas, o número de divisões e o nome do ficheiro 3d resultante.
- Box:
  Cria uma caixa, centrada na origem do referencial, recebendo como parâmetros o comprimento das arestas, o número de divisões em cada face e o nome do ficheiro 3d resultante.
- Esfera:
  Cria uma esfera, centrada na orifem do referencial, recebendo como parâmetros o raio, as slices (divisões na vertical), as stacks (divisões na horizontal) e o nome do ficheiro 3d resultante.
- Cone:
  Cria um cone com a base paralela ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o raio da base, a altura, as slices (divisões na vertical), as stacks (divisões na horizontal) e o nome do ficheiro 3d resultante.

2.2.1 Plano

Para a criação de um plano, centrado na origem e paralelo ao plano xOz, começamos por criar uma variável "edgeIncrementer"com o comprimento de cada divisão e também as variáveis "initX"e "initY"que representam a posição por onde vamos começar a construir o plano.
Nessa posição inicial começamos por criar dois triângulos, de forma a fazer um quadrado, e através de dois ciclos for e da constante "edgeIncrementer"construir os restantes quadrados nas respetivas posições.

2.2.2 Caixa

Para a criação de uma caixa, centrada na origem, o processo foi similar ao da criação do plano, começamos por criar uma variável "edgeIncrementer"com o comprimento de cada divisão e também as variáveis "initX", "initY"e "initZ"que representam a posição por onde vamos começar a construir cada uma das faces. //Utilizando o processo usado para construir o plano, mas repetindo três vezes, porque construímos as faces duas a duas, umas vez que, por exemplo, a face superior da caixa em relação à face inferior, apenas difere da posição do Y, na superior o valor de Y é initY e na inferior é - initY.

2.2.3 Cone

Para o desenvolvimento do cone começamos por definir a base, esta é constituida por x slices divididas uniformemente por 360 graus, e por isso o ângulo entre elas é 360/slices.

Após termos a base podemos começar a idealizar uma forma de conectar a base com o vértice do cone. Isto é trivial se nos abstrairmos do facto de termos stacks, pois basta conectarmos as coordenadas da base com (0,0,altura).

Para concretizarmos a ligação entre pontos da stack utilizamos funções trigonométricas.

2.2.4 Esfera

Para gerar uma esfera, é necessário determinar as coordenadas dos pontos que formam a sua superfície. Para isso, podemos utilizar funções trigonométricas para calcular as coordenadas x, y e z dos pontos.

Vamos começar pelo loop externo que itera através das fatias da esfera. Cada fatia é uma seção vertical que percorre toda a circunferência da esfera. O ângulo da fatia é determinado pela divisão de 360^{o} através do número de slices. Após termos este ângulo entre as slices é necessário saber o ângulo entre stacks, para este efeito dividimos 90 pelo número de stacks.

Utilizamos o ângulo da base em conjunção com o ângulo entre stacks para calcularmos o valor de x e z para os 4 pontos necessários para definir uma "face"da stack, e o y é calculado unicamente com o ângulo entre stacks.

Capítulo 3

Engine

3.1 Leitura do ficheiro de configuração

Como sugerido, utilizou-se o *tinyxml2* no auxílio à leitura do XML proveniente do ficheiro de configuração. O código do *engine* possui duas funções responsáveis pela mesma, sendo elas:

1. void readXML(char* filename);
2. void readGroupXML(tinyxml2::XMLElement *group);

De notar que a segunda função é apenas uma auxiliar da primeira função.

3.2 Estruturação dos dados

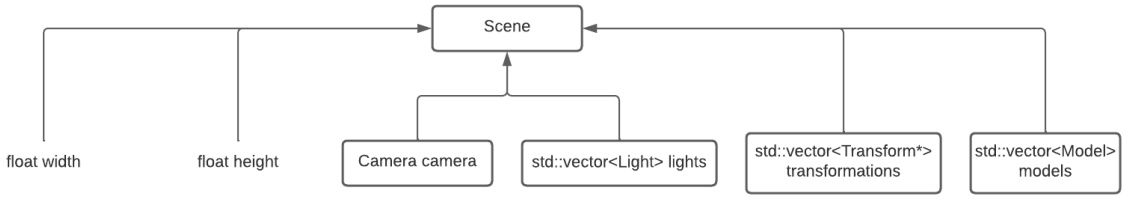

Figura 3.1: Estrutura de dados *Scene*

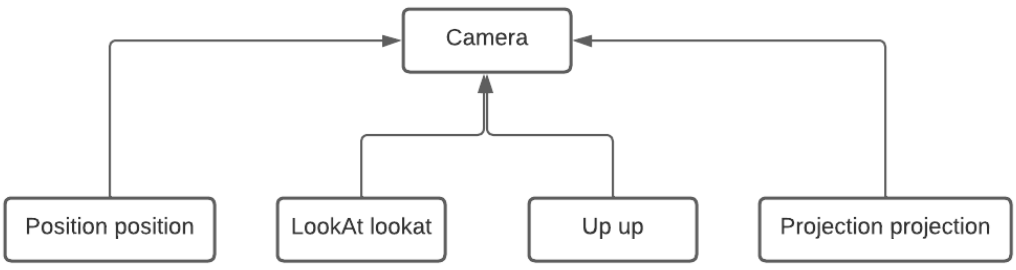

Figura 3.2: Estrutura de dados *Camera*

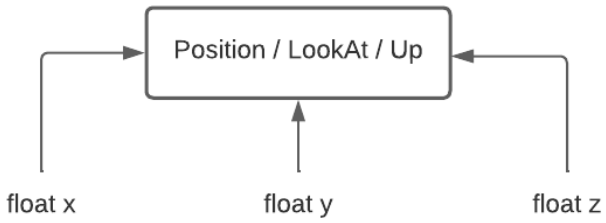

Figura 3.3: Estruturas de dados *Position, LookAt e Up*

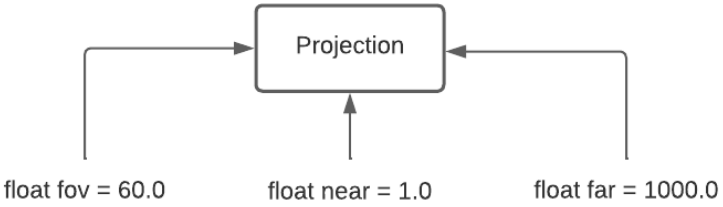

Figura 3.4: Estrutura de dados *Projection*

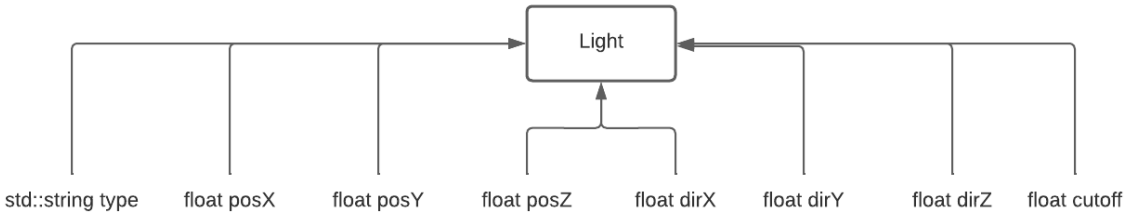

Figura 3.5: Estrutura de dados *Light*

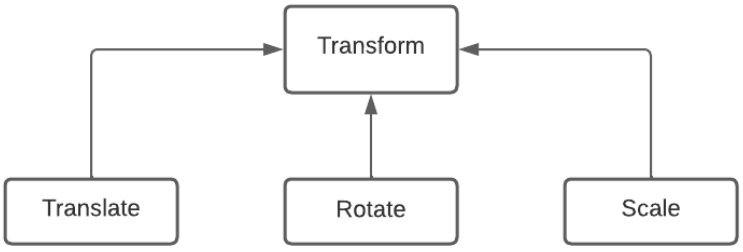

Figura 3.6: Classe *Transform*

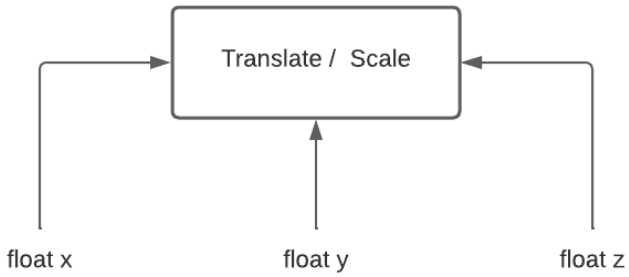

Figura 3.7: Classes *Translate* e *Scale*

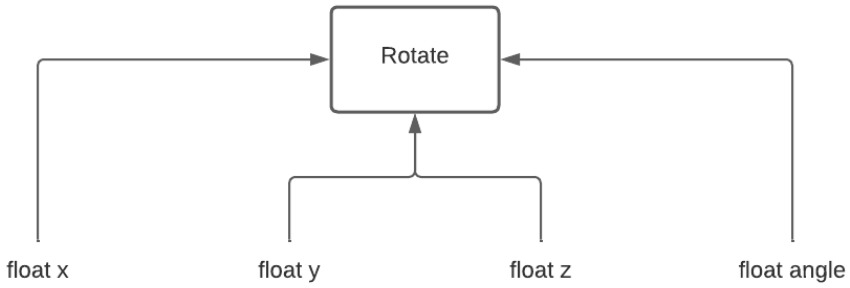

Figura 3.8: Classe *Rotate*

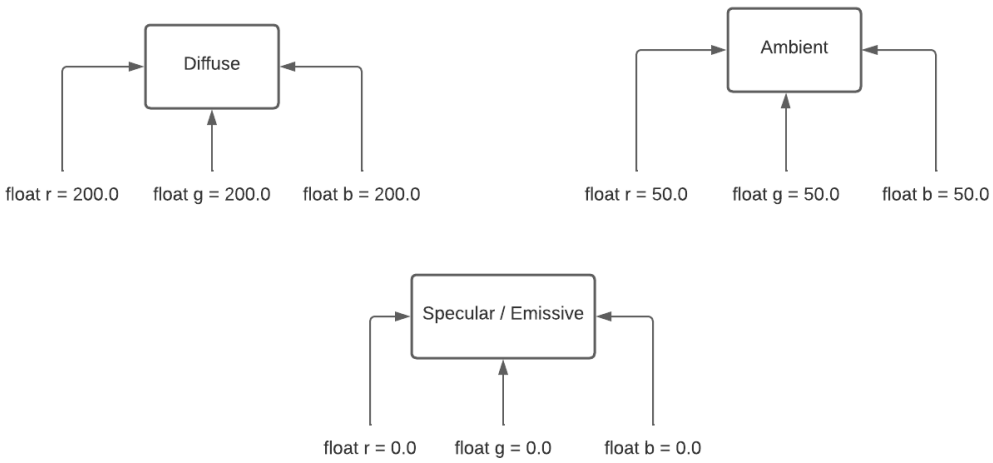

Figura 3.9: Estruturas de dados *Diffuse, Ambient, Specular* e *Emissive*

As figuras exibidas acima representam as classes e estruturas de dados utilizadas para a estruturação dos dados obtidos na leitura do ficheiro de configuração. De notar que os valores descritos nas figuras indicam os

valores padrão das variávies em questão, isto é, os valores que as variáveis tomam quando não especificadas no ficheiro de configuração.

3.3 Desenho dos modelos

No que toca ao desenho dos modelos, desenvolveu-se as seguintes funções:

- void drawAxis();
  Desenha os eixos x, y e z;
- void drawModels();
  Desenha os modelos descritos no ficheiro de configuração utilizando a função auxiliar drawFigure;
- void drawFigure(std::string filename);
  Desenha os pontos contidos no ficheiro passado como argumento, resultando na figura pretendida. Utiliza a função tokenize como auxiliar;
- void tokenize(std::string const &str, const char* delim, std::vector<float> &out);
  Recebe uma linha com coordenadas, separa-as e coloca-as num vetor, onde a posição 0, 1 e 2 do vetor correspondem às coordenadas x, y e z, respetivamente;

3.4 Execução do Engine

O comando utilizado para execução do programa *engine* é:

$: engine <configFile.xml>

O "configFile.xml"corresponde ao ficheiro de configuração que se pretende utilizar na execução.

Capítulo 4

Testes realizados e Resultados

4.1 Teste nº1

Listing 4.1: XML do Teste 1

```
<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="../../3d/cone_1_2_4_3.3d" /> <!-- generator cone 1 2 4 3
                    cone_1_2_4_3.3d -->
        </models>
    </group>
</world>
```

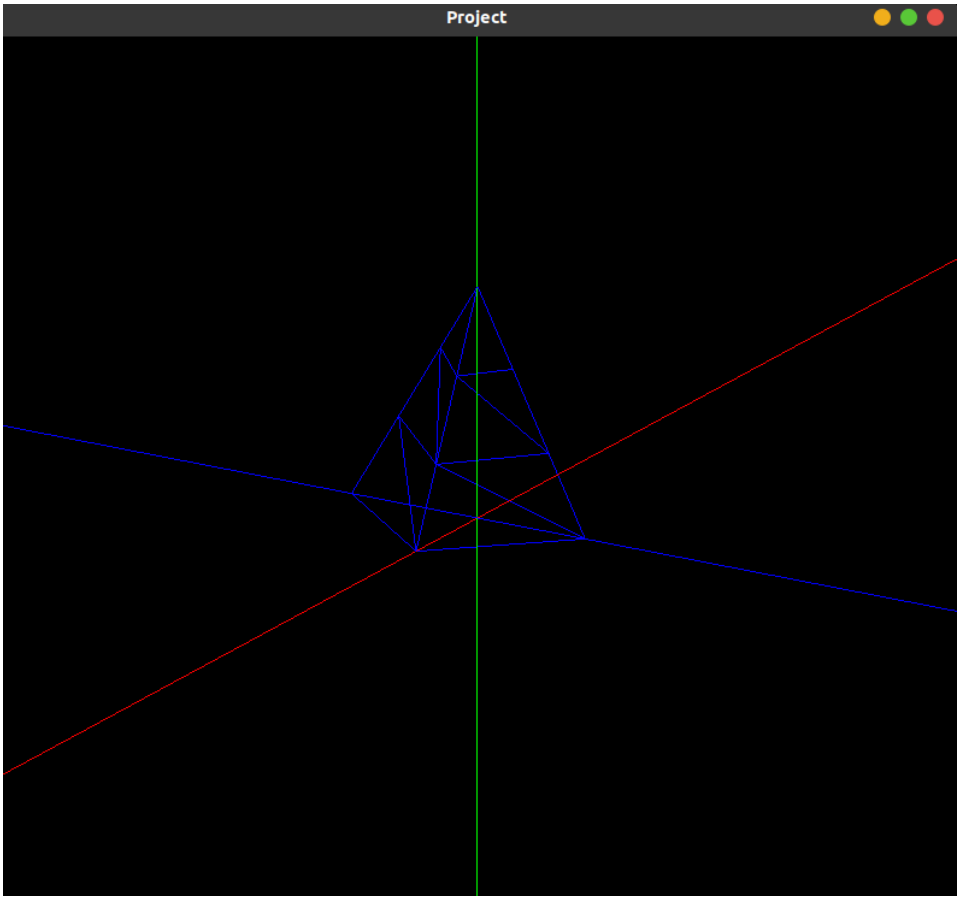

Figura 4.1: Resultado do Teste 1

4.2 Teste nº2

Listing 4.2: XML do Teste 2

```
<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="20" near="1" far="1000" />
    </camera>
    <group>
        <models>
            <model file="../../3d/cone_1_2_4_3.3d" /> <!-- generator cone 1 2 4 3
                cone_1_2_4_3.3d -->
        </models>
    </group>
</world>
```

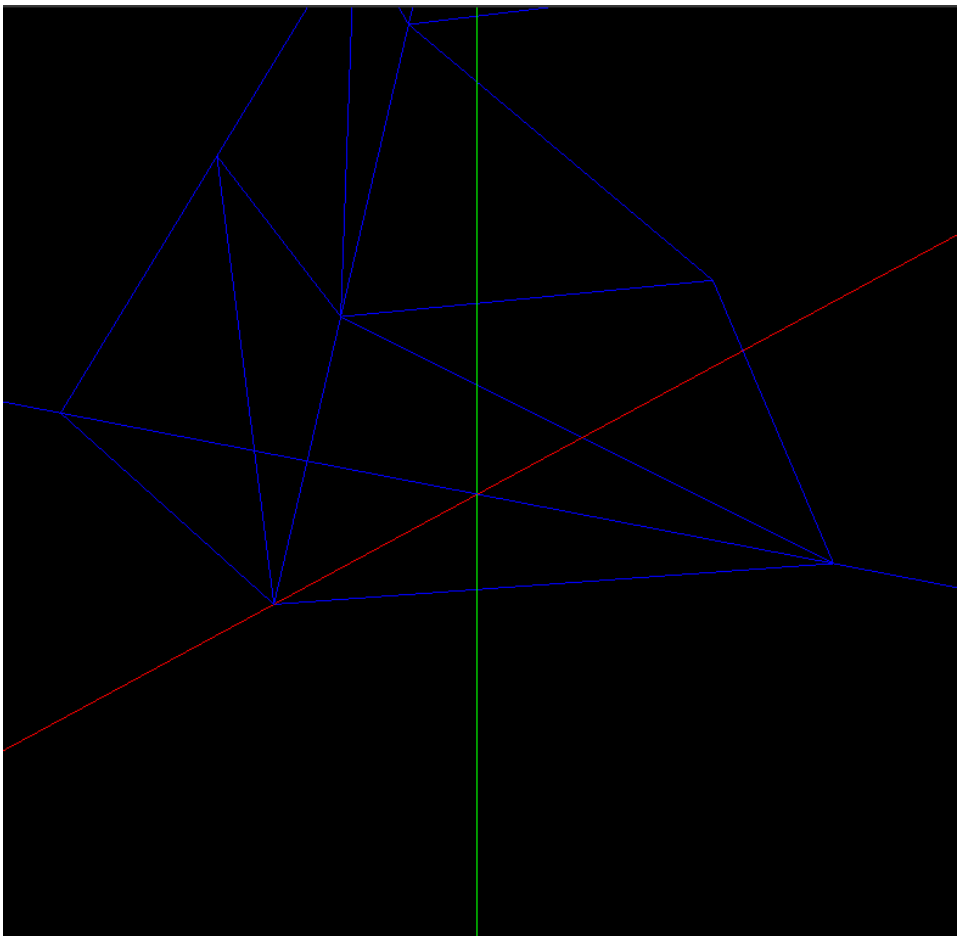

Figura 4.2: Resultado do Teste 2

4.3 Teste nº3

Listing 4.3: XML do Teste 3

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <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="../../3d/sphere_1_10_10.3d" /> <!-- generator sphere 1 10 10
                sphere_1_10_10.3d -->
        </models>
    </group>
</world>
```

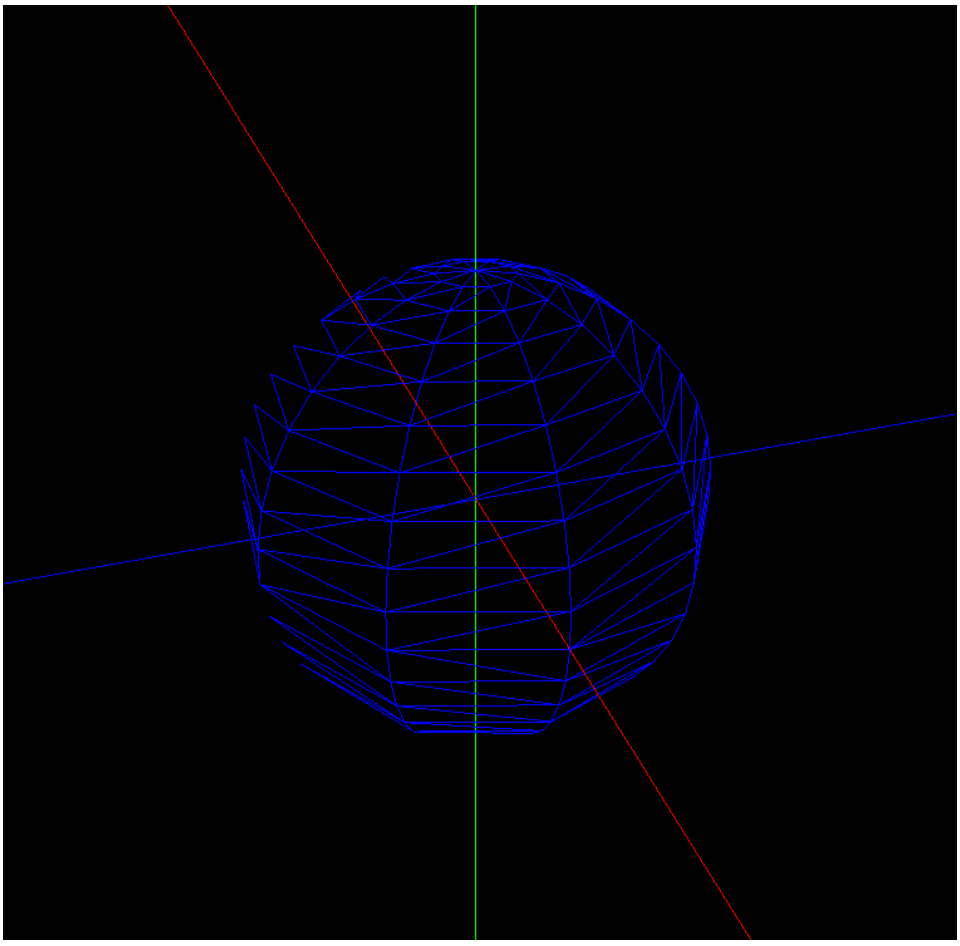

Figura 4.3: Resultado do Teste 3

4.4 Teste nº4

Listing 4.4: XML do Teste 4

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="3.5" />
    </camera>
    <group>
        <models>
            <model file="../../3d/box_2_3.3d" /> <!-- generator box 2 3 box_2_3.3d
                -->
        </models>
    </group>
</world>
```

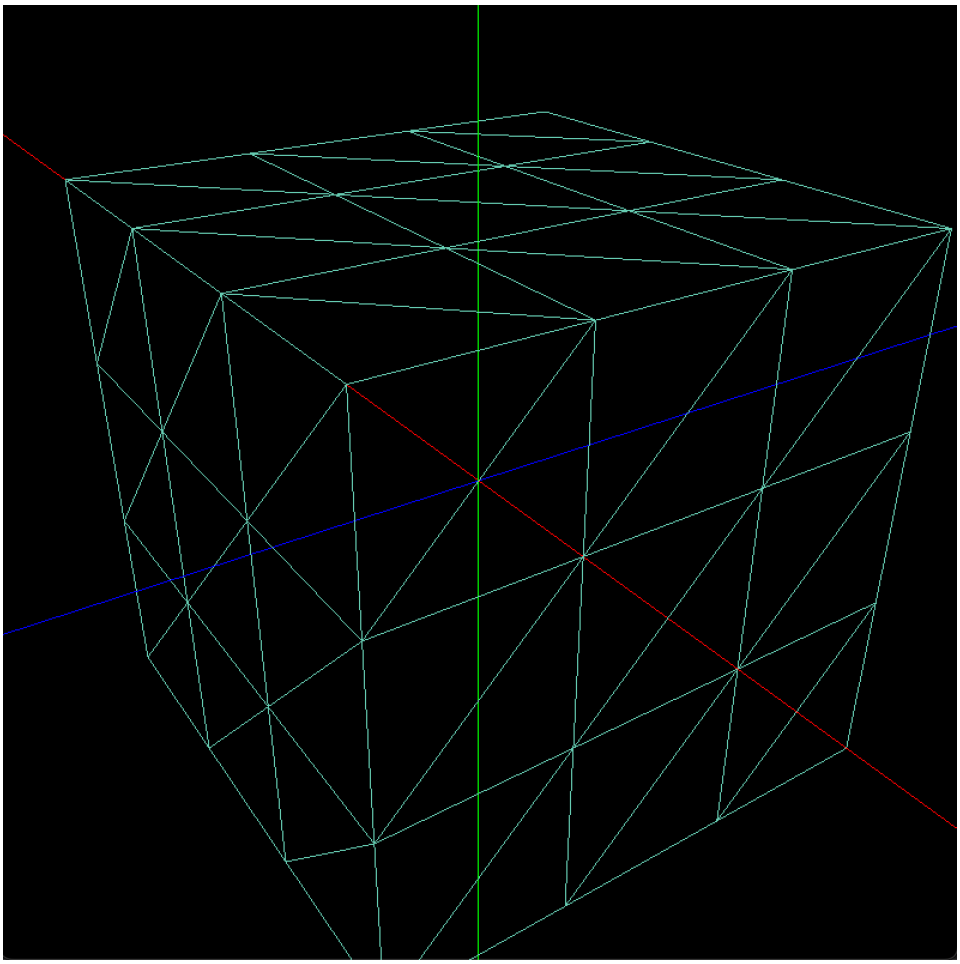

Figura 4.4: Resultado do Teste 4

4.5 Teste nº5

Listing 4.5: XML do Teste 5

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <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="../../3d/plane_2_3.3d" /> <!-- generator plane 2 3 plane_2_3.3d -->
            <model file="../../3d/sphere_1_10_10.3d" /> <!-- generator sphere 1 10 10 sphere_1_10_10.3d -->
        </models>
    </group>
</world>
```

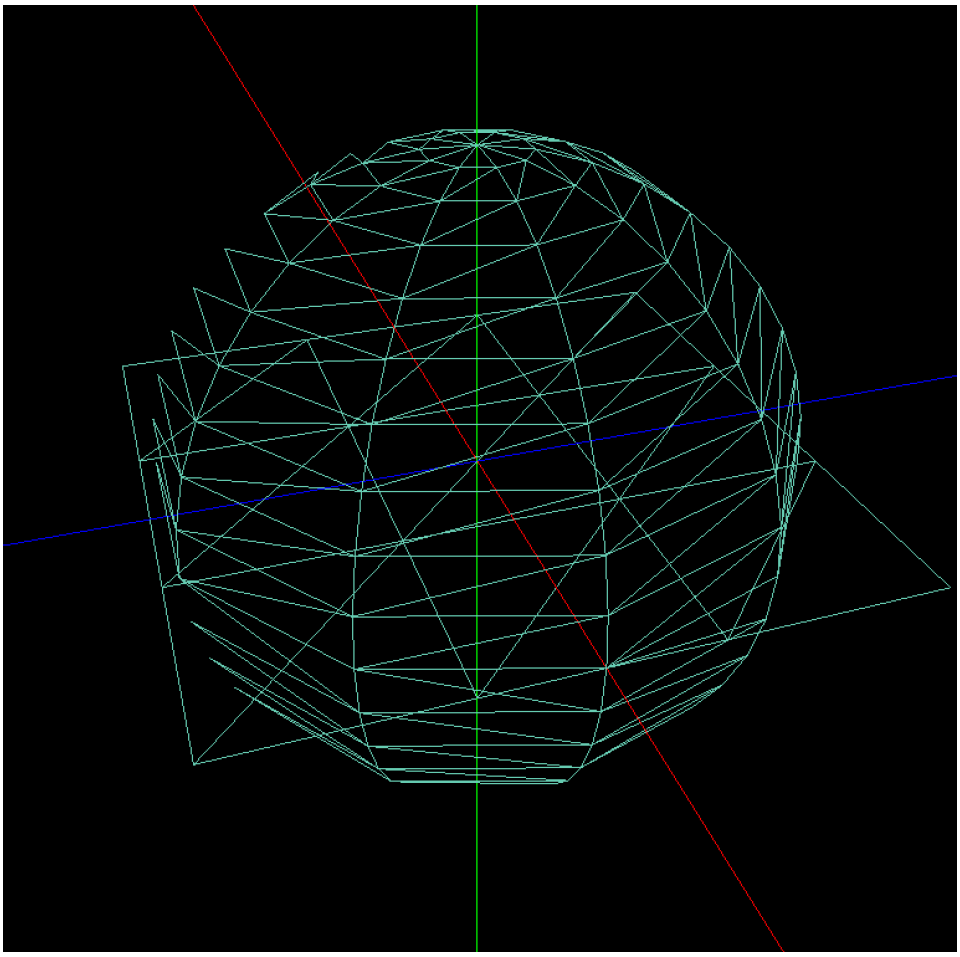

Figura 4.5: Resultado do Teste 5