Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2022/2023

Parte 1 - Primitivas Gráficas

Filipa Gomes (A96556) Miguel Gomes (A93294) Pedro Pacheco (A61042)
Rita Lino (A93196)

10 de março de 2023

CG

Resumo

Neste relatório explicitaremos as nossas decisões na realização da primeira parte do trabalho da Unidade Curricular de Computação Gráfica. Deste modo, iniciamos por explicar como desenvolvemos o *generator* dos ficheiros - onde desenhamos as figuras pedidas pelo enunciado -, seguindo-se pela descrição do *engine*, que lê a configuração dos ficheiros e renderiza os modelos.

Área de Aplicação: Computação Gráfica.

Palavras-Chave: OpenGL, C++, XML, Transformações geométricas, Primitivas gráficas.

Índice

Lista de Figuras

1 Introdução

O objetivo desta tarefa prática é desenvolver um motor 3D baseado em grafos de cena e fornecer exemplos de uso que demonstrem o seu potencial. A tarefa está dividida em quatro fases, sendo esta a primeira. Para cada fase, serão fornecidos um conjunto de ficheiros XML de configuração para fins de teste e avaliação. Cada configuração é acompanhada pela respetiva saída visual.

Neste relatório fazemos uma descrição das funcionalidades pedidas para a primeira fase do projeto. Assim sendo, esta fase envolve a criação de primitivas gráficas usando duas aplicações independentes entre si: a primeira gera ficheiros com informações dos modelos (que até ao momento incluem apenas os vértices dos mesmos) e recebe o tipo de primitiva gráfica, os parâmetros necessários para a criação do modelo e o ficheiro de destino onde os vértices serão armazenados.
As primitivas desenvolvidas foram:

- Plano,
- Caixa,
- Esfera,
- Cone,
- Cilindro,
- Pirâmide quadrangular,
- Toro;

A segunda é o motor em si que lê um ficheiro de configuração XML e exibe os modelos num ambiente de apresentação gráfica.

Esta fase requer a criação de quatro primitivas gráficas: o plano, a caixa, a esfera e o cone. A estas acresceram outras que achamos serem um exercício interessante no desenvolvimento das nossas competências, e como tal ficaram também codificados os modelos de três figuras adicionais: um cilindro, uma pirâmide e uma *torus*.

Em resumo, a primeira fase deste projeto estabelece as bases para o motor 3D baseado em grafos de cena, criando primitivas gráficas usando ficheiros de configuração XML, com o objetivo final de exibir os modelos de forma visualmente atraente.

Para facilitar a execução das aplicações foram criados quatro *scripts* para compilação e execução *compile* e *engine* / *generator* respetivamente.

2 *Generator*

O *Generator* nada é mais que uma aplicação cujo objetivo é gerar a figura pretendida. Com este intuito ele terá que receber parâmetros que possam identificar qual o modelo que se quer produzir. Estes parâmetros alteram ligeiramente de modelo para modelo, mas de uma forma geral é necessário dizer qual o nome da figura a representar (*plane*, *box*, *sphere*...), as suas dimensões (raios, alturas, fatias - *slices*, pilhas - *stacks*...) e finalmente nome e localização do ficheiro de *output* (de extensão .3d) que irá conter os pontos que o motor usará para renderizar as figuras e qualquer parâmetro extra *eixo::(plane)*.

O formato escolhido para o ficheiro de *output* (de extensão .3d) corresponde à representação de 1 triângulo por linha, isto é cada linha do ficheiro contem os três pontos necessários para desenhar um triângulo, com exceção da primeira linha que representa o número de pontos totais a serem desenhados pelo *engine*. Cada grupo de três floats que geram um ponto encontram-se separados por *ponto e virgula* ***(";")*** e cada float responsável encontra-se separado por *virgula* ***(",")***

```
generator ↬ cat plane.3d
54
-0.5,0,-0.5;-0.5,0,-1.5;-1.5,0,-1.5;
-1.5,0,-0.5;-0.5,0,-0.5;-1.5,0,-1.5;
-0.5,0,0.5;-0.5,0,-0.5;-1.5,0,-0.5;
-1.5,0,0.5;-0.5,0,0.5;-1.5,0,-0.5;
-0.5,0,1.5;-0.5,0,0.5;-1.5,0,0.5;
-1.5,0,1.5;-0.5,0,1.5;-1.5,0,0.5;
0.5,0,-0.5;0.5,0,-1.5;-0.5,0,-1.5;
```

Figura 2.1: Formato do ficheiro gerado pelo *Generator*

2.1 Figuras Implementadas

De modo a mais facilmente organizar o código, cada modelo foi implementado num módulo diferente com nome próprio e que contém as funções necessárias ao cálculo dos pontos. Estas funções são chamadas pelo *Generator* e após a sua terminação são-lhe devolvidos os pontos calculados.

2.1.1 Plano

A função que desenha o plano, determina os seus pontos a partir de um dado comprimento do lado, o número de divisões e qual o eixo sobre o, qual o plano será desenhado. Como funcionalidade extra foi implementado a opção de escolher o eixo perpendicular ao plano.

```
 generator ↰ ./generator plane 1 3 test.3d x
Generating plane with 1 units in size, 3 splits per axis, perpendicular to the {x} axis and saving to test.3d
Generated 54 points
 generator ↰ ./generator plane 1 3 test.3d y
Generating plane with 1 units in size, 3 splits per axis, perpendicular to the {y} axis and saving to test.3d
Generated 54 points
 generator ↰ ./generator plane 1 3 test.3d z
Generating plane with 1 units in size, 3 splits per axis, perpendicular to the {z} axis and saving to test.3d
Generated 54 points
 generator ↰ ./generator plane 1 3 test.3d
Generating plane with 1 units in size, 3 splits per axis, perpendicular to the {y} axis and saving to test.3d
Generated 54 points
 generator ↰
```

Figura 2.2: Exemplo de utilização da funcionalidade extra.

Com este objetivo foi desenvolvido um algoritmo que armazena os triângulos em pares usando a seguinte lógica onde é possível confirmar que todos os pontos podem ser calculados a partir de um ponto inicial 'P' de coordenadas (i, j).

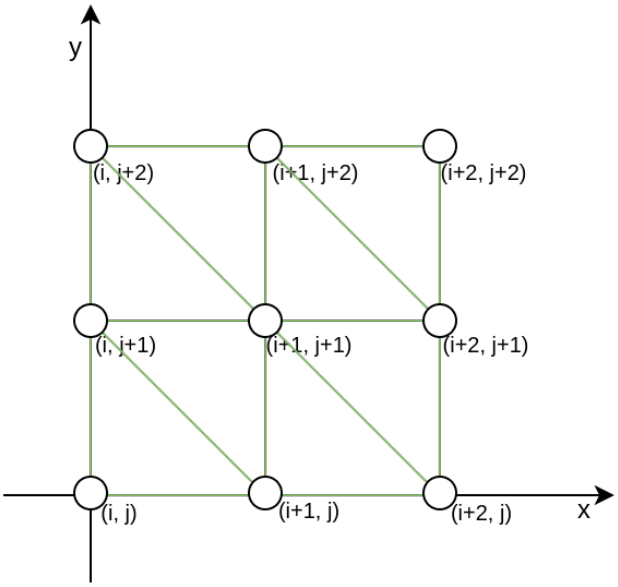

Figura 2.3: Lógica do Plano

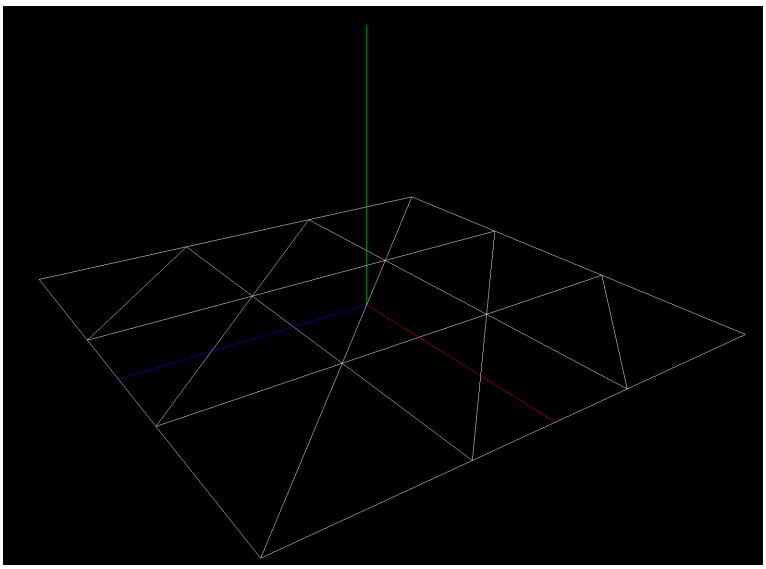

Figura 2.4: Representação de um plano no eixo XOZ

2.1.2 Caixa

Para desenhar uma caixa foi criada uma função `std::vector<Point> draw_box(double length, int divisions)`, que recebe dois parâmetros: a *length*, ou seja, o **comprimento**, que indica a altura de cada um dos quadrados do cubo e as *divisions*, que indicam o número de divisões a fazer a cada face da caixa; esta função retorna um vetor com todos os pontos da figura. Utiliza ainda a função auxiliar `std::vector<Point> draw_square(Point x1, Point x2, Point x3, Point x4)` que, a partir de quatro pontos, devolve um vetor de pontos com a ordem correta - seguindo a regra da mão direita - de forma a desenhar um par de triângulos que delineia um quadrado.

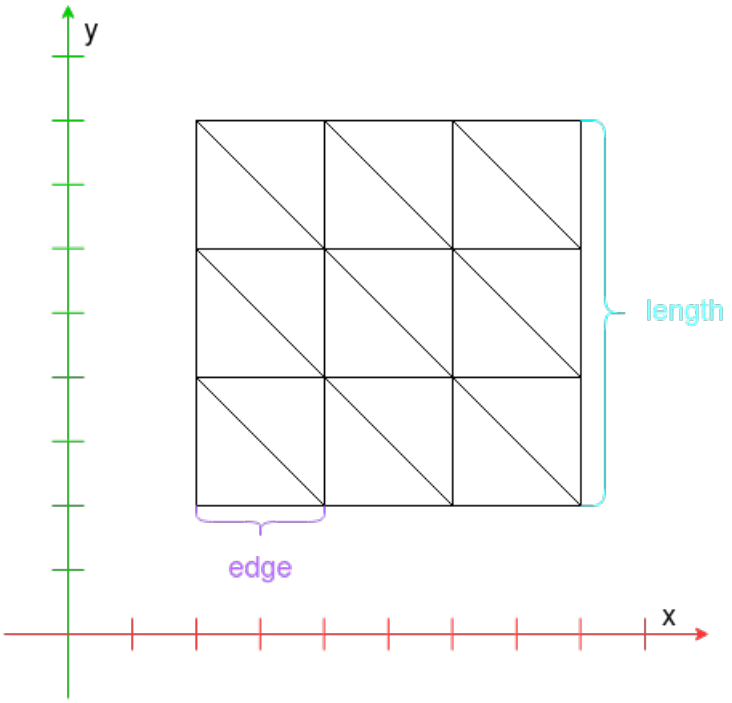

Figura 2.5: Representação das variáveis da função `draw_box` num quadrado com *length* = 3 e *divisions* = 2.

Uma face de um cubo é composta por $2^{divisions}$ quadrados. Deste modo, foi utilizado o seguinte algoritmo para inserir os pontos de cada face no vetor que a função retorna:

```
for (int i = 0; i < divisions; i++){
    for (int j = 0; j < divisions; j++){
    auto face = draw_square(Point(edge*j,edge*i,0),
                            Point(edge*(j + 1),edge*i,0),
                            Point(edge*j,edge*(i + 1),0),
                            Point(edge*(j + 1),edge*(i + 1),0)
                            );
        for(auto p : face){
            p.substract(Point(half_len, half_len, -half_len));
            points.push_back(p);
        }
    }
}
```

O uso da função `subtract` faz com que os pontos de cada face sejam colocados de modo que a caixa fique com o centro na origem do referencial.

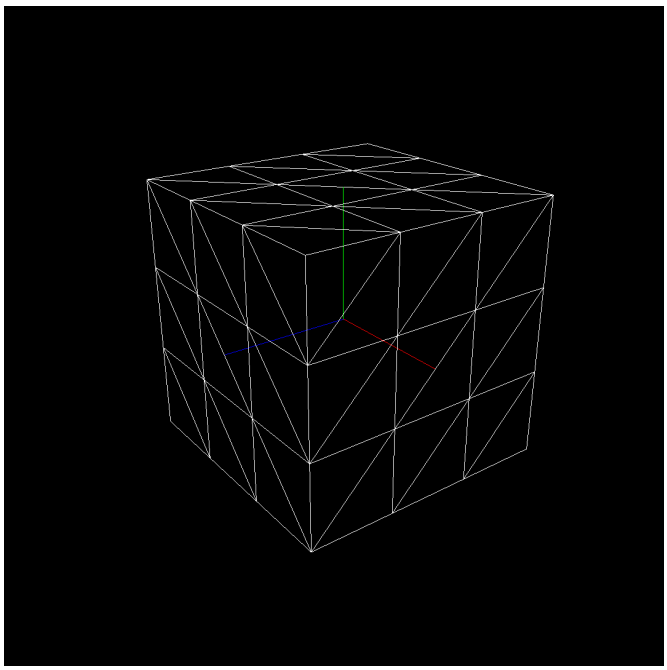

Figura 2.6: Representação de uma caixa

2.1.3 Esfera

A função da esfera recebe três argumentos, o raio da esfera e o seu número de *slices* e *stacks*. As *slices* são as subdivisões da esfera ao longo do eixo X, e as *stacks* são as suas subdivisões ao longo do eixo Y.
Começando no polo inferior e iterando por cada *slice* da esfera, construiu-se o sólido, utilizando coordenadas polares para ser possível descrever com precisão as coordenadas de cada ponto da esfera, tornando assim a representação o mais fiel possível.

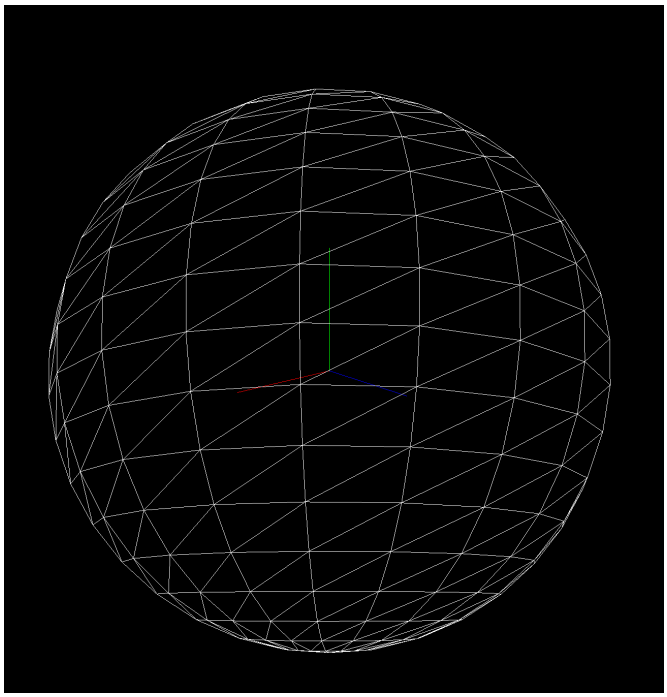

Figura 2.7: Representação de uma esfera

2.1.4 Cone

Para a possível construção de um cone a função de construção do mesmo (*draw_cone*), recebe quatro argumentos, sendo estes: o *radius* (raio da base), a *height* (altura do cone), número de *slices* e número de *stacks*.
Começando pela construção da base, vai se iterando as *stacks* do cone por cada *slice* de modo que as *stacks* e as *slices* sejam cada vez mais pequenas até ao pico do cone.

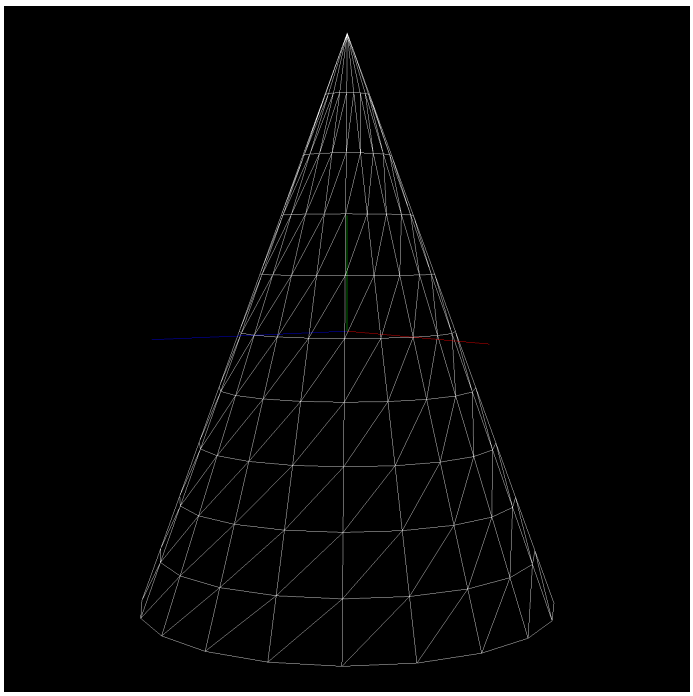

Figura 2.8: Representação de um Cone

2.1.5 Figuras extra

Cilindro

A função de construção de um cilindro recebe como parâmetros o *radius* (raio das bases), a *height* (altura do cilindro), número de *slices* e número de *stacks*.
Primeiro constrói-se a base de baixo e vai se iterando as *stacks* por *slice* de maneira que cada *stack* seja um cilindro sem tampas de altura *height/stacks*. Uma vez feitas todas as *stacks* finaliza-se com a construção da base de cima.

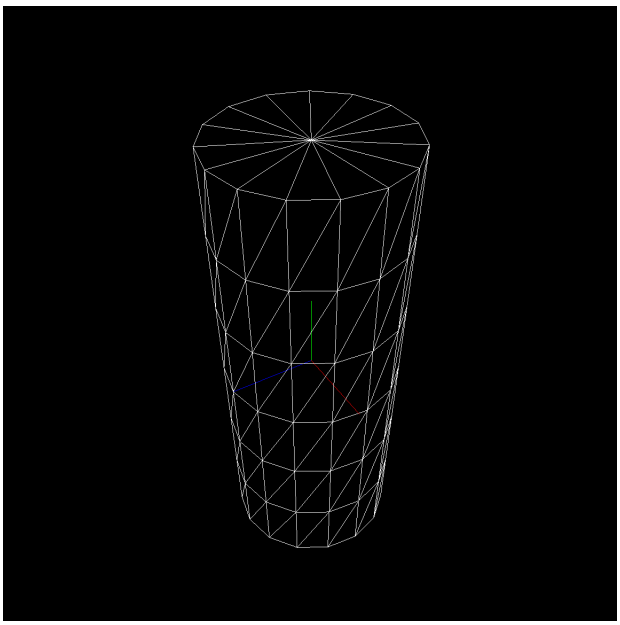

Figura 2.9: Representação de um Cilindro

Pirâmide quadrangular

Poderíamos construir a pirâmide com base no código de construção do cone, porem a função de construção da pirâmide apenas recebe os parâmetros: base (comprimento da aresta da base), *height* (altura da pirâmide) e número de *stacks*, logo a sua construção já terá que ser feita de maneira diferente da do cone. Iniciando a construção pela base, e passando então para as faces através do uso de pequenos triângulos (cada par forma uma face da *stack*), itera-se a construção das *stacks* por *slice* em função da altura.

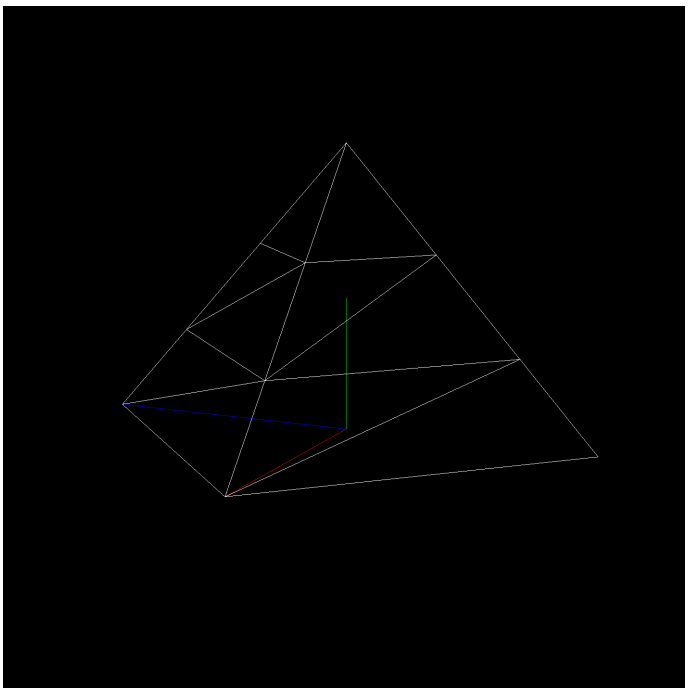

Figura 2.10: Representação de uma pirâmide

Toro

A partir das coordenadas paramétricas do toro:

$$\mathbf{x}(\theta,\varphi) = (\mathbf{R}+\mathbf{r}\cos\theta)\cos\varphi$$
$$\mathbf{y}(\theta,\varphi) = (\mathbf{R}+\mathbf{r}\cos\theta)\cos\varphi$$
$$\mathbf{z}(\theta,\varphi) = \mathbf{r}\sin\theta$$

e trocando os eixos **y** e **z** conseguiu-se implementar o código para a representação de um toro que será bastante útil numa fase seguinte quando quisermos implementar as órbitas no sistema solar.

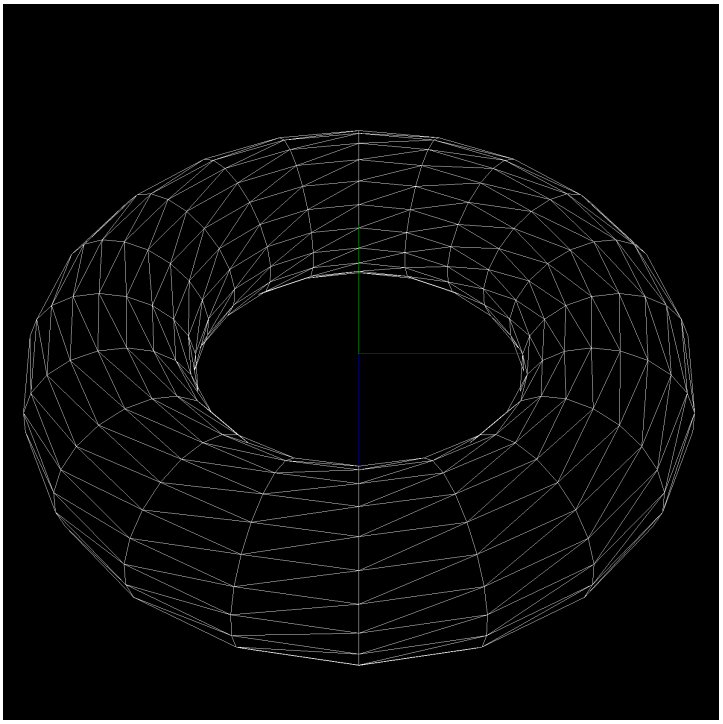

Figura 2.11: Representação de um Toro

2.2 XML

Os ficheiros XML são aqueles que serão lidos pelo *Engine* de modo à renderização dos pontos calculados pelo *Generator* ser possível. Contêm, portanto, algumas informações que são pertinentes à visualização. São elas:

- **Window**: informação relativa à resolução da janela na qual a renderização será feita;
- **Camera**

  position: posição inicial da câmera;

  lookAt: ponto inicial para onde a câmera está a olhar;

  up: inclinação da câmera;

  projection: valores relativos ao campo de visão da câmera;

- **Models**: nome dos ficheiros .3d a carregar;

```
<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="../models_generated/cylinder_1_6_15_6.3d" />
        </models>
    </group>
</world>
```

Figura 2.12: Exemplo de um ficheiro XML

3 *Engine*

O *Engine* foi a aplicação criada para ler e interpretar a informação da configuração das primitivas gráficas a partir de um ficheiro *XML* e a partir de aí gerar uma cena. Para esta funcionalidade ser de mais simples implementação, tomou-se partido do *parser* **tinyXML**. Desta forma, após o *parsing* do ficheiro, o *Engine* coloca a câmara na posição especificada e permite ainda o movimento da mesma de uma forma livre.

3.1 Câmera

Uma vez que o *OpenGL* trata das coordenadas da câmera como coordenadas cartesianas, foi criado um módulo responsável pela gestão da câmera, das suas variáveis e métodos. Sendo apenas necessário recalcular as coordenadas de posição do parâmetro *lookAt* quando o ângulo ou a distância da câmera relativamente ao *target* é alterada, evitando assim o seu cálculo a todos os *frames*. O valor lido pelo *XML* é para a posição da câmara, é usado como valor inicial e também é utilizado para calcular a distância ao *target*.

3.2 Funcionalidades Extras

Foram implementadas várias funcionalidades extra para melhor visualização das figuras renderizadas:

- ***View Mode*** - Carregando na tecla **M**, o modo de visualização da cena é alterado para um de três estados:

  GL_FILL - Visualizar as faces de cada triângulo preenchidas;

  GL_LINE - Visualizar os triângulos pelas suas arestas;

  GL_POINT - Visualizar todos os pontos gerados;

- ***Zoom*** - Carregando nas teclas + ou -

  \+ - Zoom in

  \- - Zoom out

- ***Rotação da câmera*** - Carregando nas **arrow keys**

  left arrow - Rotação da câmara para a esquerda

  right arrow - Rotação da câmara para a direita

  up arrow - Rotação da câmara para cima

  down arrow - Rotação da câmara para baixo

- ***Exit*** - Carregando na tecla **Q**, a janela de visualização fecha

4 Conclusões

A elaboração deste projeto permitiu-nos desenvolver competências e familiarizar-mo-nos com a linguagem de programação *C++* assim como com a interface *OpenGL* e e as ferramentas que este disponibiliza para a área de Computação Gráfica de modo a conseguirmos apresentar soluções aos problemas pedidos.

Muito embora seja uma pequena amostra das capacidades desta API, sentimos que estão implementadas as bases que nos permitirão desenvolver desafios mais complexos no futuro recorrendo ao *Generator* e *Engine* aqui mostrados.