Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática 2021/2022

Computação Gráfica

Trabalho prático - Fase 1

- Graphical Primitives -

13 de março

Benjamim Miranda Costa (A87985)
Maria Sofia Rocha Gomes (A93314)
Marisa Ferreira Soares (A92926)
Miguel Rodrigues Santa Cruz (A93194)

Índice

Introdução

No presente relatório será explicado a estratégia da conceção desta primeira fase, a arquitetura do programa desenvolvido e os respetivos resultados obtidos, referentes ao trabalho prático da disciplina de Computação Gráfica.

O principal objetivo desta primeira fase do trabalho era realizar duas aplicações: uma que gerasse ficheiros "*.3d*" com as coordenadas dos vértices de acordo com as dimensões e divisões das figuras escolhidas e outra que lesse ficheiros *XML*, que contém informações sobre vários aspetos da câmara, como posição, *field of view*, etc..., bem como os respetivos ficheiros "*.3d*" que incluem os vértices, para que possam ser desenhados os modelos.

Arquitetura

A arquitetura desta primeira fase é constituída por duas aplicações:

- *Generator* - responsável pela geração de forma geométricas, sendo as coordenadas guardadas em ficheiros com a extensão ".3d".
- *Engine* - responsável pela visualização de ficheiros ".3d"

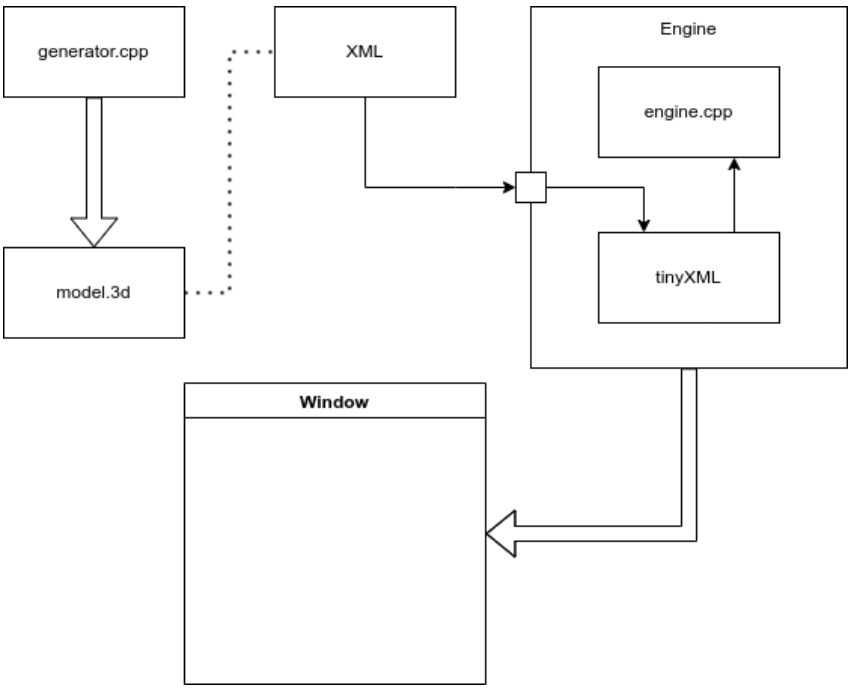

Figura 1. Arquitetura geral do funcionamento das duas aplicações

Generator

O *generator* é a aplicação que gera ficheiros com a extensão ".3d" que contém as coordenadas dos vértices dos modelos. Estas coordenadas são salvas linha a linha separadas por espaços em branco.
Exemplo de um ficheiro ".3d" gerado pela nossa aplicação:

```
0 2 0
0 4 1
1 7 1
1 0 9
```

Foram definidas várias funções que geram as coordenadas para figuras geométricas básicas:

- **Plane**: recebe como argumento o tamanho e o número de divisões
- **Box**: recebe como argumento o tamanho e o número de divisões
- **Sphere**: recebe como argumento o raio, o número de divisões horizontais e o número de divisões verticais
- **Cone**: recebe como argumento o raio, a altura, o número de divisões horizontais e o número de divisões verticais

No sentido de auxiliar o utilizador na utilização do *generator* é mostrado ao utilizador uma mensagem de sintaxe errado quando o número de argumentos não é o correto.

```
$ ./generator
Sintax error:
   Usage: ./generator [Shape] [Args] [Output File]
 Shapes available:
    Plane: [length] [divisions]
    Box: [size] [divisions]
    Sphere: [radious] [slices] [height]
    Cone: [radious] [heigth] [Slices] [stacks]
```

Figura 2. Erro de sintaxe no uso do *generator*

Abaixo descreve-se em detalhe a abordagem seguida no desenvolvimento das funções que geram as coordenadas dos vértices das diversas figuras.

Plane

A função que gera os pontos de cada triângulo que constituem o plano recebe como parâmetros o tamanho e o número de divisões dos triângulos. O plano está centrado na origem. Na função, é realizado dois ciclos `for` que é percorrido tantas vezes o número de divisões escolhidas, de modo a que desenhe o número de triângulos que é suposto.
Ao mesmo tempo que são calculados os vértices, estes são escritos para o ficheiro ".3d" respetivo.

Box

Para gerar o cubo são passados como argumentos o tamanho do cubo e o número de cortes em cada face do cubo, denominados *slices*.
Para a obtenção dos vértices dos triângulos que constituem as diversas faces, o cubo foi dividido em pares: base e topo, face esquerda e face direita e o par frente, trás. Esta estratégia é mais eficiente uma vez que cada par partilha as mesmas coordenadas, com a exceção de apenas uma coordenada.

Sphere

A função responsável pela geração das coordenadas da esfera é a função createSphere que tem como assinatura: `createSphere(float raio, int stacks, int slices);`
Esta função utiliza coordenadas polares para o cálculo das coordenadas, sendo para esse efeito utilizados dois ciclos `for` que percorrem verticalmente as *stacks* e horizontalmente as *slices*.

Cone

Os parâmetros que necessários para desenhar está primitiva gráfica podem ser divididos como os que estão relacionados com a representação da base, que será sempre um polígono regular centrado na origem com um dado *raio* (distância à origem) e determinado número de *slices* (número de divisões) e os que estão relacionados com as laterais com a altura (distância da origem até à ponta da figura) e *stacks* (número de divisões da altura) .

- Para obtenção das coordenadas da base:

A base da figura será representada à custa de triângulos isósceles com um vértice comum, a origem, e que giram em torno do eixo y, as equações seguintes demonstram as operações necessárias para o cálculo de todos os seus vértices.

$\Theta = 2 * \frac{\pi}{slices}$, ângulo de cada triângulo isósceles que constitui a base
$x = raio * sin(\theta)$
$y = 0$, visto que está no plano x0z
$z = raio * cos(\theta)$

- Para obtenção das coordenadas das laterais:

Para os vértices das laterais foi-se aumentando a componente y de acordo com a razão $altura/stacks$ e para as componentes x e z o raio foi diminuindo de acordo com a razão $raio/stacks$.

Engine

O engine é a aplicação que permite a visualização dos ficheiros ".3d" gerados pelo nosso *generator*.
O *engine* recebe como único argumento na linha de comando o nome do ficheiro XML que contém as configurações da câmera, bem como os modelos que serão necessários carregar. Ao ser executado é então invocada a função `readXMLConfigurationFile` que abre o ficheiro XML e cria uma instância da classe `cameraConfig`, previamente definida e que contêm informações acerca da posição da camera, o seu *field of view* entre outros atributos. Para a leitura do ficheiro XML foi utilizada a biblioteca *tinyXML* recomendada pelos docentes.
É então criada uma janela cujo o nome é o mesmo do ficheiro XML aberto e que mostra a cena gerada por esse mesmo ficheiro.
Para evitar leituras consecutivas dos modelos a serem mostrados, foi criada a classe *Model* que contêm em memória todos os vértices de um modelo e ainda permite invocar o método `drawModel` que desenha esse mesmo modelo.

Como funções adicionais e com o intuito de aumentar a funcionalidade da aplicação foram implementadas as seguintes funções:

- Utilizando as setas horizontais é possível rodar a cena que está a ser visualizada no momento em torno no eixo Oy;
- Utilizando as setas verticais é possível aumentar e diminuir o zoom da cena que está a ser visualizada no momento;
- Utilizando as teclas *f1* e *f2* é possível alterar o *culling*. Com *f1* são mostradas as faces que estão voltadas para a câmera (GL_FRONT), e com *f2* as faces que estão voltadas na direção contrária à câmera (GL_BACK);
- Utilizando as teclas *f3* e *f4* podemos alterar o *PolygonMode*, alternando entre GL_FILL e GL_LINE

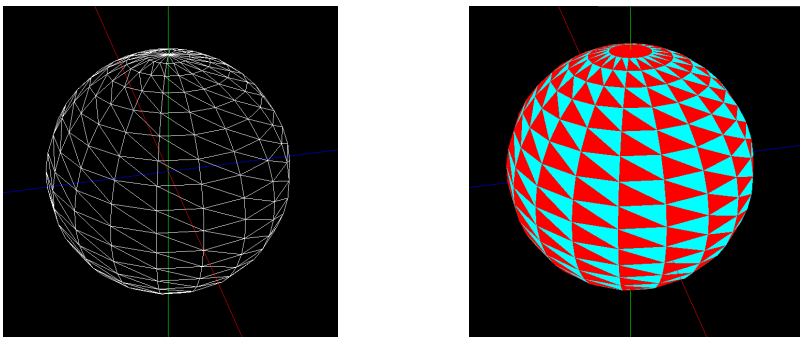

Figura 3. Exemplo do uso das teclas *f3* e *f4*

Para facilitar a visualização definiu-se o desenho das faces usando um sistema alternado de cores.

Resultados/Output

Plane

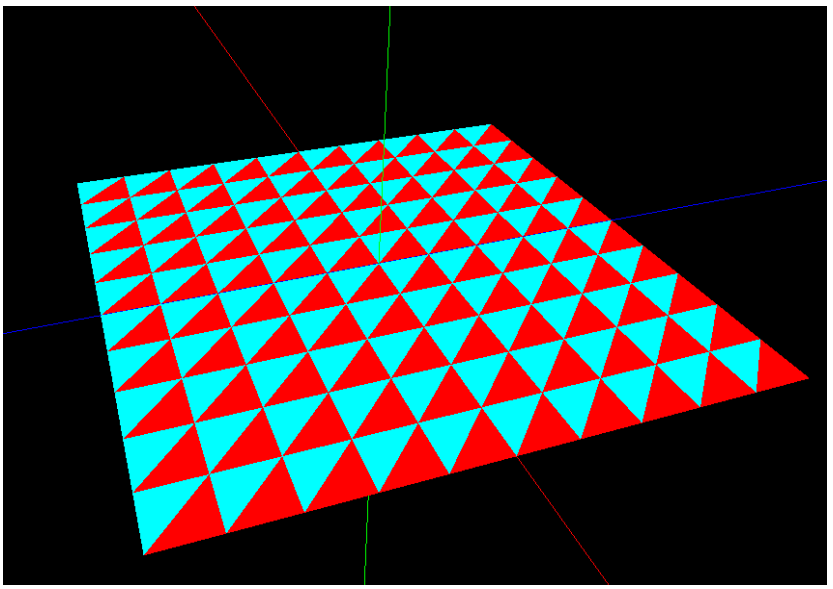

Figura 4. Exemplo de um plano gerado de dimensão 3 e 10 divisões.

Box

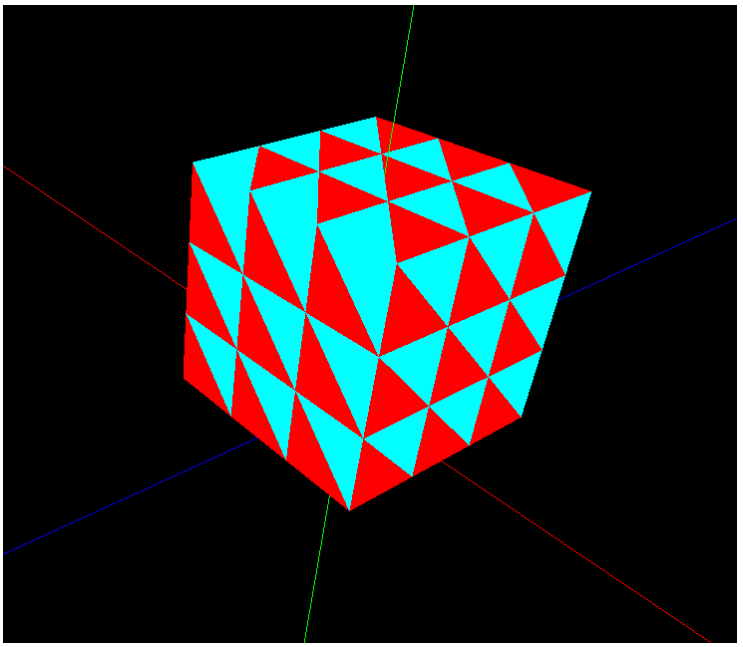

Figura 5. Exemplo de um cubo gerado de dimensão 3 e 3 divisões.

Cone

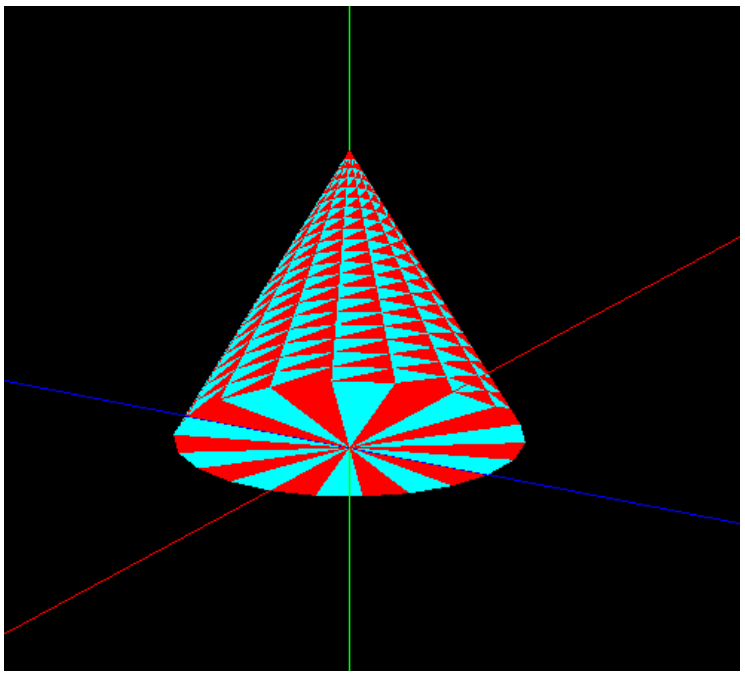

Figura 6. Exemplo de um cone de raio 1, altura 2, 20 divisões e 20 stacks

Sphere

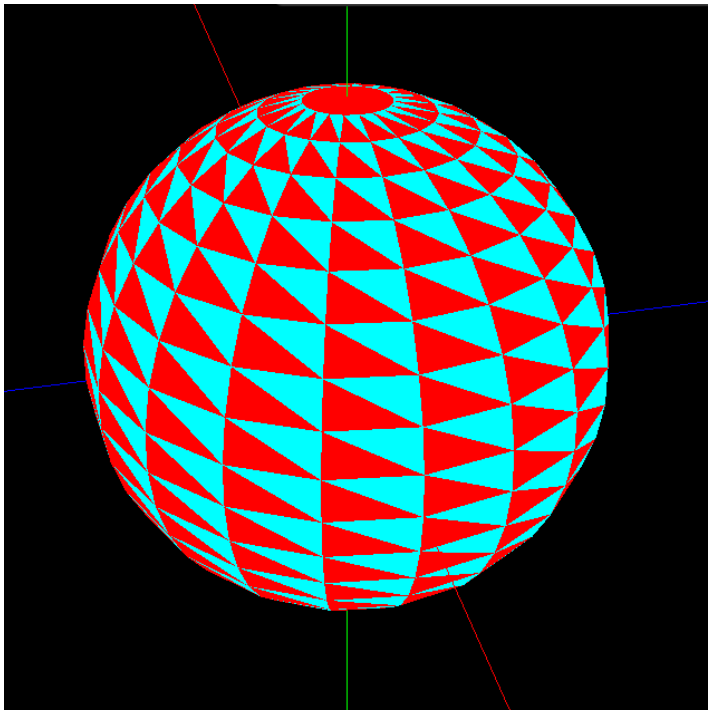

Figura 7. Exemplo de uma esfera gerada com raio 1, 10 slices e 10 stacks

Conclusão

Depois de realizada esta fase do trabalho prático, podemos afirmar que concluímos todos os objetivos que eram esperados e ainda implementamos alguns dos objetivos extra, como o movimento da câmara que permite rodar a cena que está a ser visualizada, fazer *zoom* entre outros que foram explorados ao longo deste relatório.

As principais dificuldades na realização do trabalho passaram principalmente pela inexperiência no uso da linguagem C++, dificuldades essas que conseguimos superar devido à familiaridade com a linguagem de programação C e com o paradigma de programação orientada aos objetos.