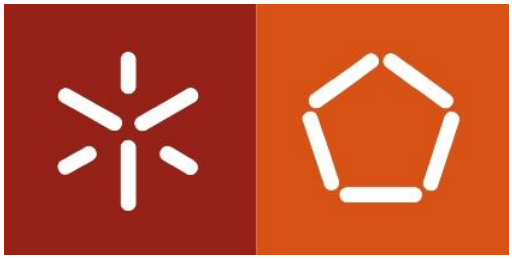
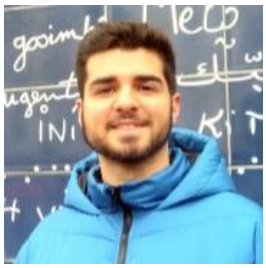


Computação Gráfica - CG 2019/2020

FASE 1

Mestrado Integrado em Engenharia Informática

José Pinto
A84590



Eduardo Costa
A85735

Luís Lopes
A85367

Ricardo Carvalho
A84261

Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de um mecanismo 3D baseado num cenário gráfico com o auxílio de duas ferramentas utilizadas, também, nas aulas práticas, sendo elas OpenGL e C++.

Nesta primeira fase do projeto, foi necessário o desenvolvimento de duas aplicações: um *generator* e um *engine* de forma a representar as primitivas gráficas do plano, caixa, esfera e cone. O *generator* é responsável por gerar vértices consoante as primitivas gráficas recebidas, enquanto que o *engine* é responsável pela leitura de ficheiros XML que servirão para desenhar os vértices das primitivas gráficas anteriormente gerados.

Assim sendo, neste relatório, tratamos de fazer uma descrição técnica do projeto, de forma a justificar as escolhas feitas ao longo da sua elaboração.

- ***Generator***

Para criar um executável da aplicação é necessário correr o comando:
"g++ generator.cpp -o generator".

Após a criação do executável, é possível criar ficheiros com os vértices das diferentes figuras referentes a esta fase (plano, caixa, esfera e cone), fazendo a invocação do executável e passando primitivas gráficas validas.

A aplicação é constituída por uma *main* que olha para os argumentos passados no terminal e invoca funções responsáveis por gerar cada uma das figuras ou, em casos de argumentos inválidos, dá display de uma mensagem de erro.

1. Plane

A unidade de constituição de todas as primitivas gráficas são triângulos. Desta forma, para gerar um plano é necessário dividi-lo em 2 triângulos.

Devido à regra da mão direita apenas um dos lados é representado ao desenharmos. Por esta razão desenhamos duas vezes o plano, mas alteramos a ordem em que os vértices são desenhados, para que, segundo a regra da mão direita, o plano seja visível dos dois lados.

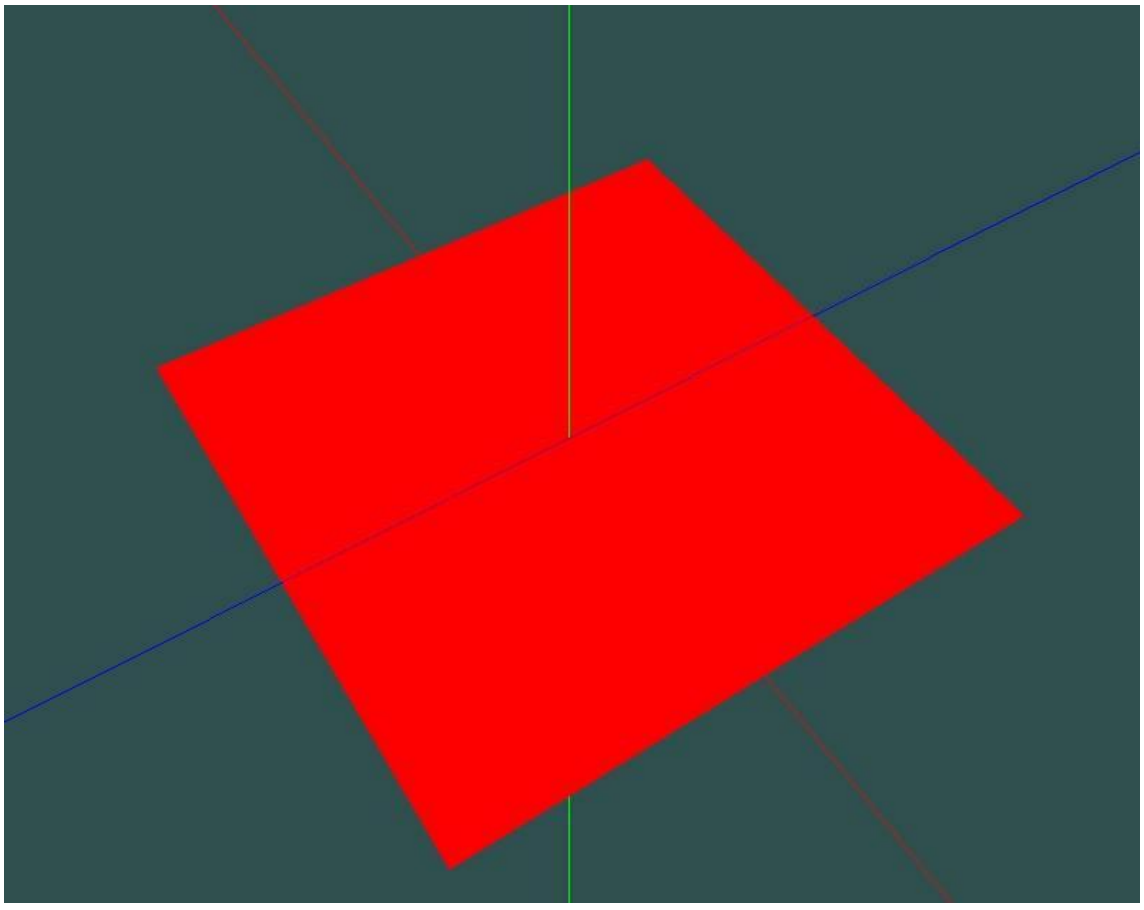


Figura 1- Plane

2. Box

A caixa é formada por seis faces, nas quais temos de ter em atenção o número de divisões.

O processo de desenhar uma face da caixa segue a mesma lógica do plano.

No entanto, devido à possibilidade de divisões, o espaçamento entre dois pontos pertencentes ao eixo X é dado pela divisão do comprimento pelo número de divisões. Já para o espaçamento entre dois pontos do eixo do Z e Y, substituímos o comprimento pela largura e altura, respetivamente.

Desta forma, são necessários 2 ciclos "*for*" que vão percorrendo os eixos divisão a divisão até a face estar completamente formada, sendo que, para cada face, temos um ciclo externo que percorre as divisões na vertical e um ciclo interno que percorre as divisões na horizontal.

Foi necessário, ainda, decidir qual a ordem correta para desenhar os vértices de cada face de forma a respeitar a regra da mão direita.

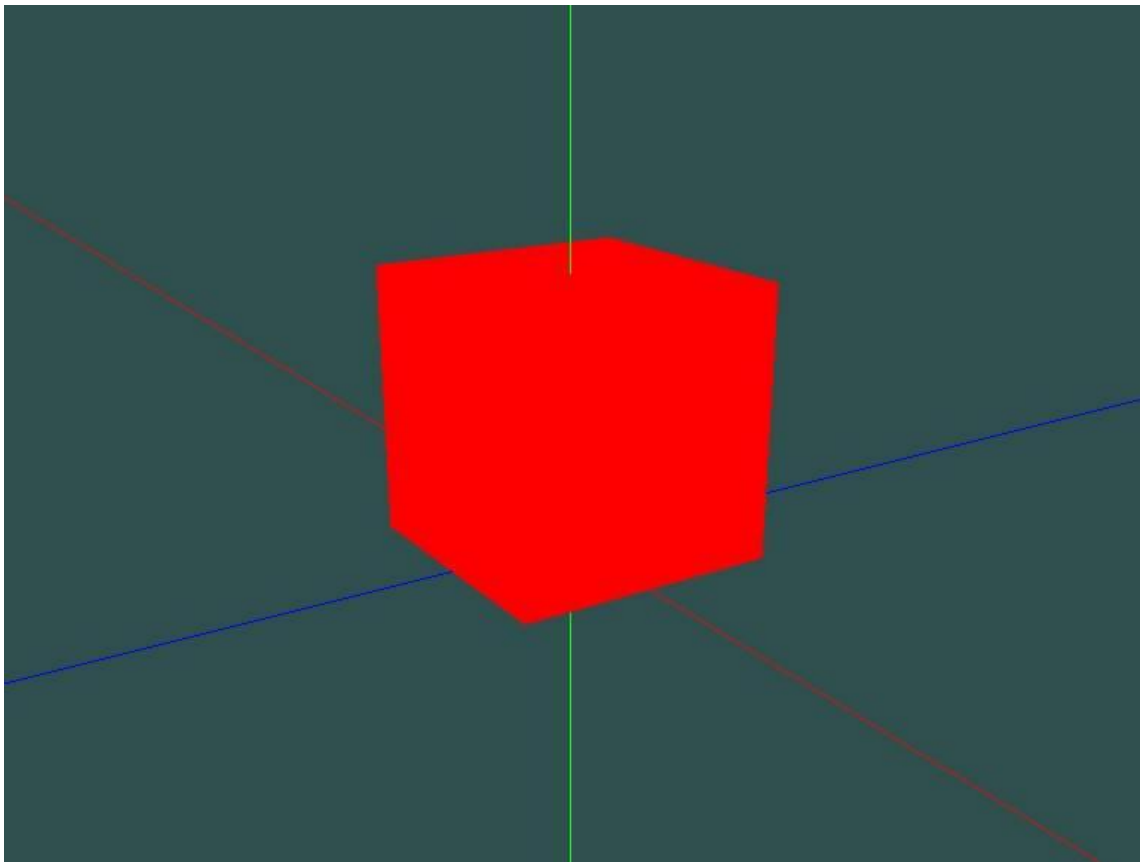


Figura 2 – Box

3. Sphere

Uma esfera é um sólido geométrico formado por uma superfície curva em que todos os pontos estão à mesma distância do centro.
Os vértices da esfera são formados através da função: void esfera(char* argv[])
Esta função requer 4 argumentos sendo um deles o nome do ficheiro onde os vértices são armazenados.
O primeiro argumento refere-se ao raio da esfera o segundo refere-se aos slices, que é o número de divisões da esfera na horizontal, já o terceiro refere-se às stacks, que representa o número de divisões pela vertical. Quanto maiores forem os valores das slices e stacks maior será a curvatura resultante.

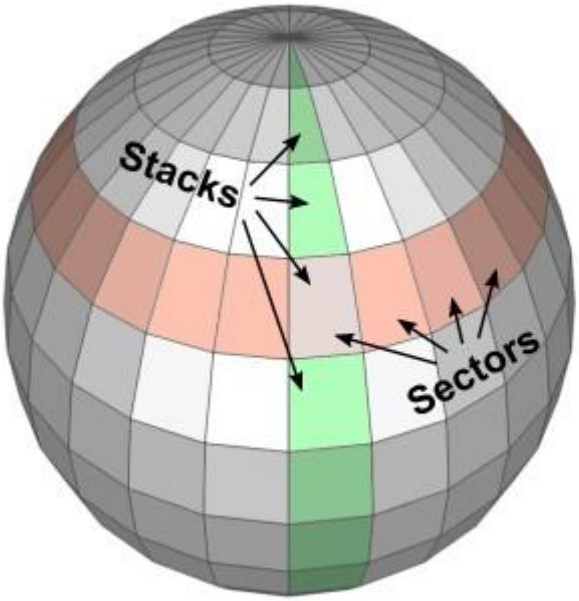


Figura 3 – Representação da esfera
(http://www.songho.ca/opengl/gl_sphere.html)

```
for (int i = 0; i < stacks; i++) {
        angle1 = 0; // reiniciar o alfa para cada stack.
        for (int j = 0; j < slices; j++)
```

O nosso algoritmo passa por percorrer todos os slices por cada stack através de dois ciclos "*for*" sendo o exterior para as stacks e o interior para as slices. Cada segmento obtém as suas coordenadas com recurso às coordenadas esféricas (r, θ, Ω):

- x = r * sin(θ) * sin(Ω)
- y = r * cos(θ)
- z = r * sin(θ) * cos(Ω)
- θ ∈ {α, α+ Δ (α)}
- Ω ∈ {β, β+ Δ (β)}

O espaçamento entre as fatias(slices) e as camadas(stacks) é dado pelas expressões:

Espaçamento entre fatias: angle2 = (2 * pi) / slices

Espaçamento entre camadas: angle4 = pi / stacks;

Concluindo, os valores das coordenas X,Y,Z são calculados através das expressões apresentadas a cima, sendo que no final de cada ciclo exterior e interior o valor do ângulo é incrementado pelo valor do espaçamento vertical e pelo valor do espaçamento horizontal respetivamente.

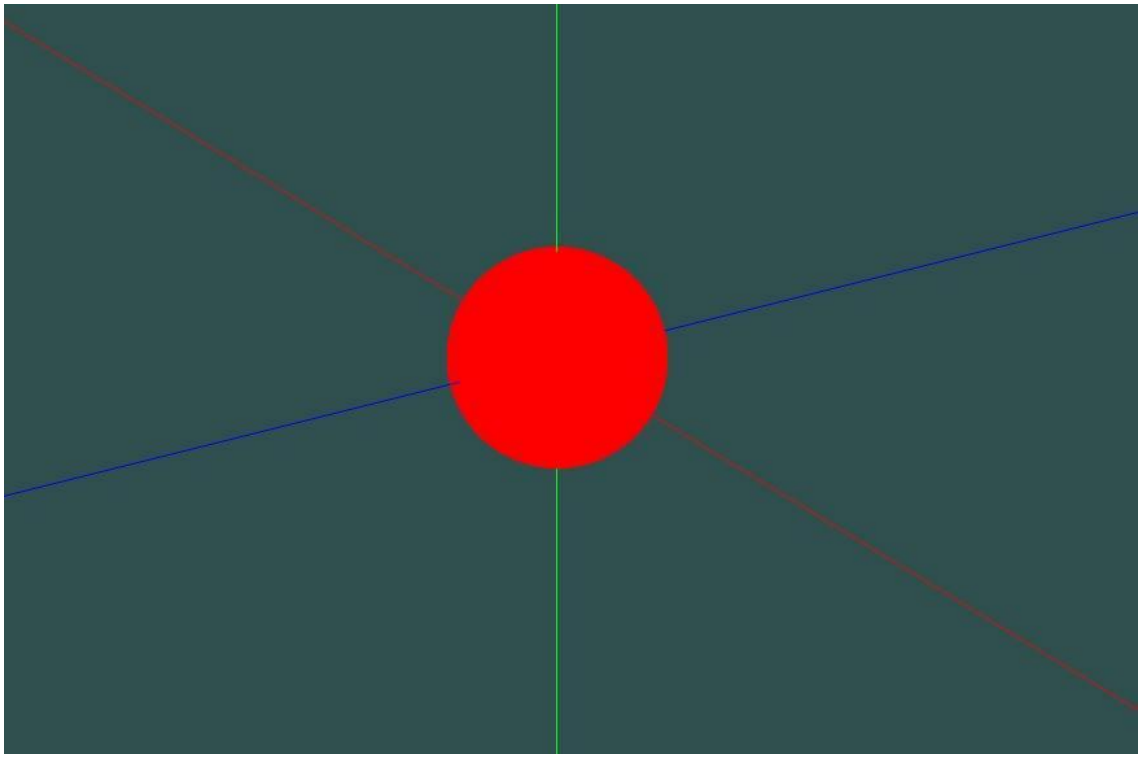


Figura 4 – Sphere

4. Cone

Para o cálculo do cone são necessários quatro parâmetros: raio, altura, slices e stacks. Tal como na esfera quanto maior o número de slices e stacks maior será a precisão no desenho da primitiva. A função responsável por formar o cone é void cone(char* argv[]).
O cálculo dos pontos do cone encontra-se dividido em duas partes: pontos que constituem a base e pontos da superfície lateral.
A base é um círculo que que se encontra no plano y=0, ou seja, é apenas necessário calcular as coordenadas em X e Z para estes pontos. Os triângulos da base partilham o ponto do centro do círculo (0,0,0). Tal como na esfera é necessário calcular o espaçamento entre estes triângulos que neste caso é dado por: angle2 = (2 * pi) / slices.
As coordenadas X e Z são calculadas através das expressões:

- $x = raio \times \sin(\alpha)$
- $z = raio \times \cos(\alpha)$

Já no caso dos vértices da superfície lateral todos os triângulos têm um ponto em comum que se trata do ponto (0,altura,0) e os outros dois encontram-se na base, ou seja, têm a coordenada y=0 e o cálculo das coordenadas X e Z é dada pelas expressões:

- $x = raio \times \sin(\alpha)$
- $z = raio \times \cos(\alpha)$

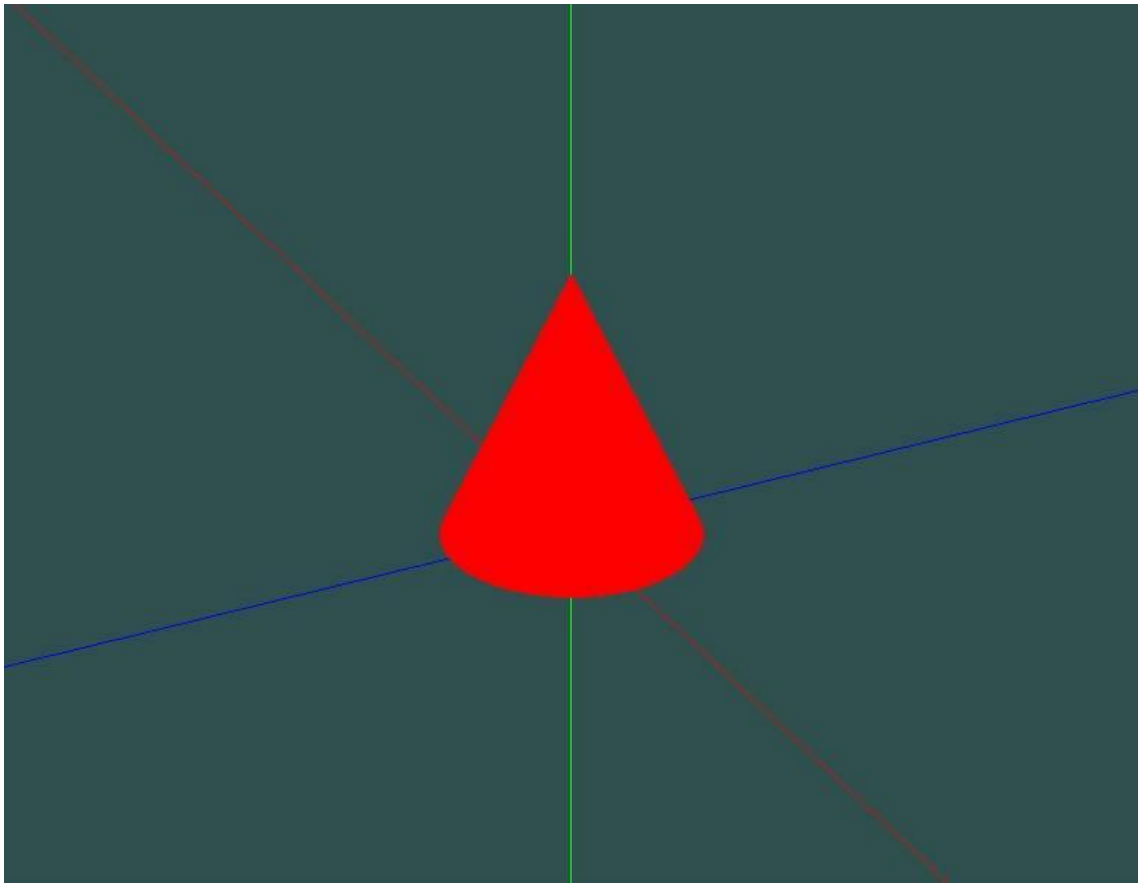


Figura 5 – Cone

- **Engine**

Nesta fase, o *engine* pedido deve ler o ficheiro de configurações, escrito em XML (no nosso caso, "*config.xml*"), que contem o nome dos ficheiros que se pretende que sejam exibidos, ficheiros estes gerados pelo *generator* anteriormente. É importante salientar, também, que o *parser* XML utilizado foi o tinyxml. Este *parser* lê o "*config.xml",* que deve ter um formato específico, exemplificado no ficheiro que se encontra na diretoria do projeto, armazenando num vetor esses ficheiros.

```
std::vector<std::string> Trabalho;
```

Figura 5 – Vetor que armazena nome dos *model files* a representar

Posteriormente, os vértices existentes em cada um dos ficheiros lidos também para um vetor de pontos, definidos como tuplos de três elementos, que representam as coordenadas dos vértices dos triângulos a representar, de modo a que seja possível o seu desenho.

```
std::vector<Ponto> triangles;
```

Figura 5 – Vetor que armazena coordenadas dos vértices

Conclusão

Consideramos esta primeira fase do projeto bastante importante, pois, além de ser crucial no desenvolvimento do resto do projeto, também nos permitiu consolidar o conhecimento em ferramentas associadas à computação gráfica, como o OpenGL e o GLUT. Para além disto, permitiu-nos adquirir, também, conhecimentos da linguagem C++, que não tínhamos e que será, de facto, útil para o futuro. Em relação à matéria referente à UC de Computação Gráfica, conseguimos consolidar e familiarizar-nos com os algoritmos que estão associados à criação de primitivas gráficas.

Finalmente, na nossa ótica, consideramos que o trabalho realizado nesta primeira fase do trabalho foi bastante positivo e que nos ajudará a continuar as seguintes fases do projeto. Logo, achamos que o objetivo desta fase inicial foi atingido, pois todas as funcionalidades propostas no enunciado foram concebidas.