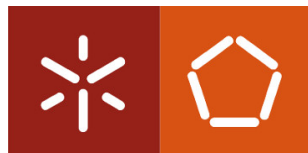


UNIVERSIDADE DO MINHO

ESCOLA DE ENGENHARIA



# Computação Gráfica

Licenciatura em Engenharia Informática

## Graphical primitives

Trabalho Prático - Fase 1 - Grupo 7

João Silva - [A89293]

Tiago Ribeiro - [A93203]

Francisco Novo - [A89567]

João Vieira - [A93170]

Março, 2022

# Conteúdo

<b>Introdução</b>	<b>2</b>
<b>Arquitetura do Programa</b>	<b>3</b>
<b>Structs</b>	<b>3</b>
<b>Generator</b>	<b>4</b>
Cálculo dos vértices . . . . .	4
Plane . . . . .	4
Box . . . . .	5
Sphere . . . . .	6
Cone . . . . .	7
Escrita do ficheiro . . . . .	8
<b>Engine</b>	<b>9</b>
Leitura do ficheiro . . . . .	9
Desenho das Primitivas . . . . .	9
Movimentação da Câmara . . . . .	10
<b>Conclusão</b>	<b>11</b>

# Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi desenvolvida a primeira fase do trabalho prático proposto pelos docentes.

O projeto será realizado na linguagem de programação C++, fazendo recurso à ferramenta *OpenGL*. O objetivo desta primeira fase é representar 4 primitivas gráficas: um cone, um plano, uma caixa e uma esfera, sendo fornecidos vários parâmetros como: altura, raio, largura, profundidade, slices e stacks. A representação gráfica irá recorrer ao desenho de vários triângulos, sendo necessário que se determine quais são os vértices que os compõem.

Adicionalmente, foram criadas duas aplicações: um "motor" e um "gerador", sendo o gerador encarregue de gerar ficheiros com a extensão *.3d*, onde estará contida toda a informação necessária para que se possam representar as suas respetivas primitivas gráficas. O motor fica encarregue de ler a informação contida no ficheiro *.3d* e representa a respetiva figura.

# Arquitetura do Programa

Conforme pedido, existem duas aplicações: o motor e o gerador. Assim, utiliza-se dois *packages*, em que cada um deles possui os ficheiros necessários para cada uma das aplicações funcionar.

Existe também o *package* "Structs", onde se encontram as estruturas de dados que serão necessárias, assim como funções que o motor e o gerador têm em comum, de forma a evitar a repetição de código. De forma a facilitar a leitura de ficheiros XML, foi utilizada a ferramenta *TinyXML*.

## Structs

Foi criado um *namespace* structs, onde se encontram as estruturas de dados utilizadas, assim como as funções comuns tanto ao gerador como ao motor.

Neste espaço, estão definidos os seguintes structs:

- **point**: Utilizado para representar um ponto, contém 3 floats que correspondem as coordenadas *xyz* de um ponto.
- **cameraSettings**: Corresponde às settings da câmara que foram obtidas através da leitura do ficheiro *XML*.
- **cameraPolar**: Corresponde às coordenadas polares relativas à posição da câmara.

Neste *namespace* está também definida a função **addPoint**, cuja função é adicionar um ponto numa *figure*.

# Generator

## Cálculo dos vértices

### Plane

A função que gera o plano recebe o comprimento dos lados e o número de divisões de cada um. Está centralizado na origem do referencial e é separado em  $2 \cdot (\text{número de divisões})^2$  triângulos.

Começou-se por desenhar o plano a partir do ponto  $(-x, 0, -x)$ , sendo  $x$  o comprimento do plano. Os triângulos são isósceles e os dois lados iguais têm tamanho  $x/(\text{número de divisões})$ . São realizados dois ciclos para a sua criação, sendo que um deles se encontra dentro do outro. Como é um plano e a sua altura é constante, não necessita de percorrer o eixo Y. Daí, o primeiro ciclo percorre o eixo Z, e o segundo o eixo X. Cada iteração do ciclo interior irá preencher um quadrado (dois triângulos) e cada iteração do ciclo exterior irá preencher uma fila de quadrados.

Na nossa implementação o plano está virado para as duas direções, ou seja, foram desenhados dois planos um virado para o lado positivo do eixo Y e outro para o lado negativo do mesmo eixo.

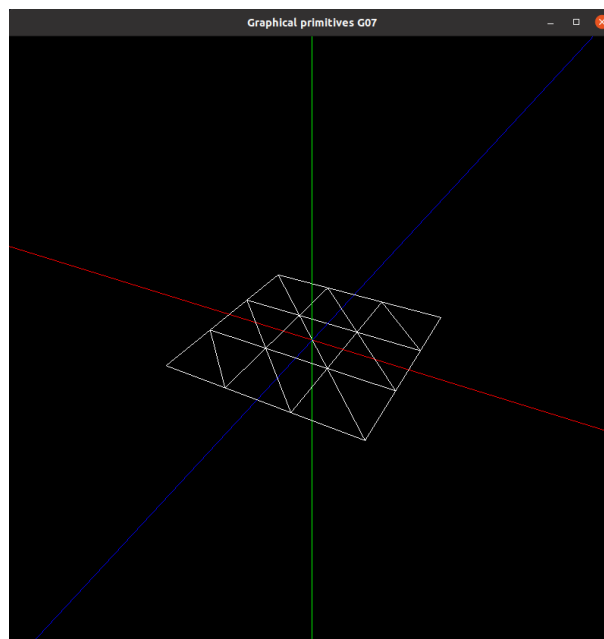


Figura 1: *Plane* com tamanho 2, com 3 divisões em cada eixo

## Box

A função que gera o cubo recebe, igualmente ao plano, o comprimento das arestas e o número de divisões de cada uma. O cubo desenhado irá estar centrado na origem do referencial, portanto ao desenhar as faces irá haver uma coordenada que se irá manter constante com o valor  $\pm(\text{comprimento das arestas})/2$ .

Para desenhar o cubo são realizados três ciclos em que cada um destes irá preencher duas faces opostas do cubo. A metodologia usada para desenhar uma face é a mesma utilizada para desenhar um plano, sendo que a única diferença é que as coordenadas que estão a ser utilizadas vão sendo alteradas dependendo da face que está a ser criada.

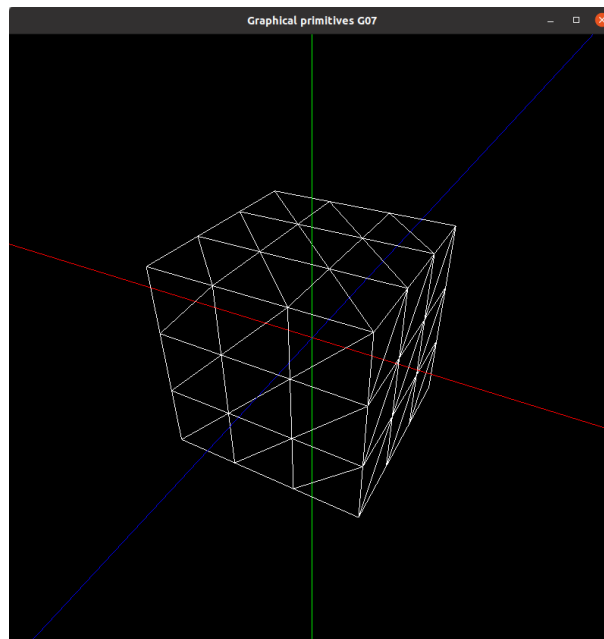


Figura 2: *Box* com tamanho 2, em que cada lado tem uma grelha 3x3

## Sphere

A função que gera a esfera recebe, o raio, as *slices* e as *stacks*. A esfera desenhada irá estar centrada na origem do referencial.

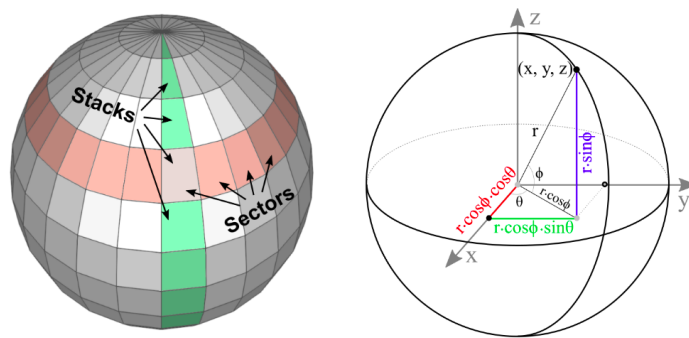


Figura 3: *Slices* (ou *sectors*) e *stacks* de uma esfera e como representar um ponto na esfera

Qualquer ponto da esfera pode ser obtido pelas seguintes equações paramétricas:

$$\begin{aligned}x &= r * \cos(\phi) * \sin(\theta) \\y &= r * \sin(\phi) * \sin(\theta) \\z &= r * \cos(\phi) * \cos(\theta)\end{aligned}$$

Para desenhar a esfera esta foi dividida ao meio na horizontal. Depois foram sendo feitas iterações pelas *slices*, sendo que a cada iteração são preenchidos os trapézios pertencentes às *stacks* com dois triângulos cada um, a partir da parte central para os polos da esfera. Nas *stacks* superiores e inferiores de cada *slice* apenas é necessário um triângulo, sendo este gerado no final de cada iteração, acabando assim por gerar cada *slice*. No final de todas estas iterações sobre as *slices* a esfera estará completa.

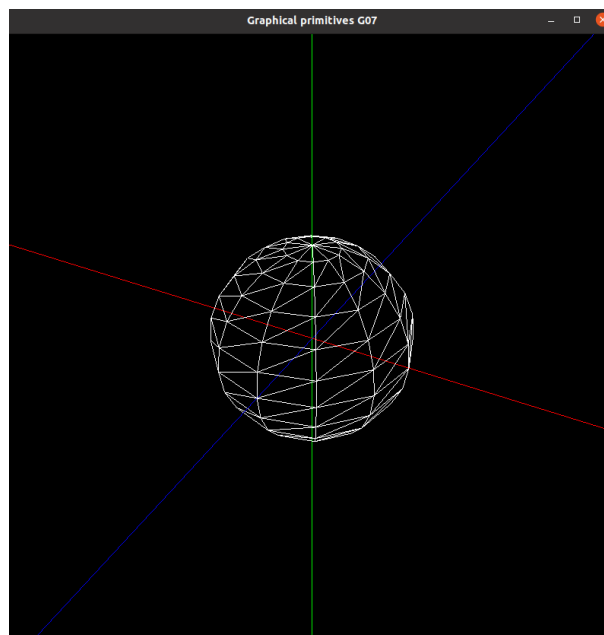


Figura 4: *Sphere* com raio 1, 12 *slices* e 12 *stacks*

## Cone

A função que gera o cone recebe, o raio, o tamanho, o número de *slices* e *stacks*. O cone desenhado irá ter a base centrada na origem do referencial.

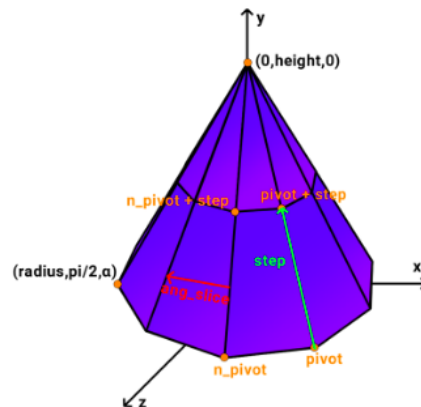


Figura 5: Como representar um ponto no cone

Para desenhar o cone são efetuadas iterações sobre as *slices*, onde inicialmente é desenhado o triângulo pertencente à base do cone. De seguida, são feitas iterações sobre as *stacks*, para serem desenhados todos os triângulos pertencentes às *stacks* de uma *slice* (dois triângulos em cada trapézio). Na última iteração sobre cada *stack* é desenhado o triângulo que une a penúltima *stack* ao vértice superior do cone. Depois de serem efetuadas todas as iterações sobre as *slices* irá ser obtido o cone.

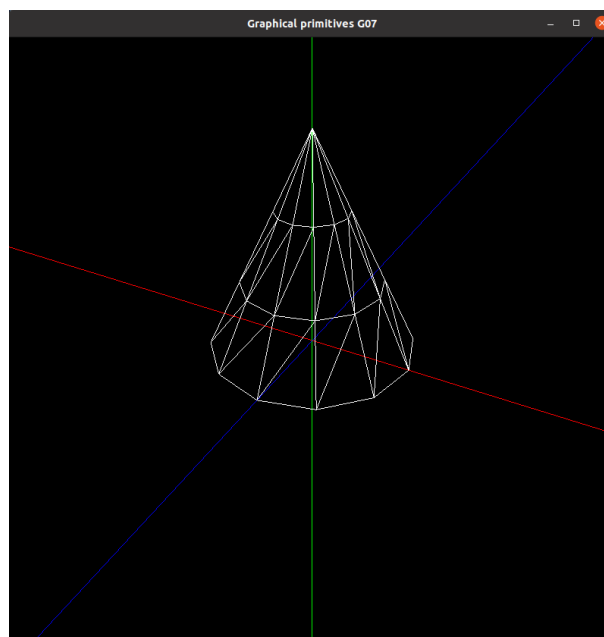


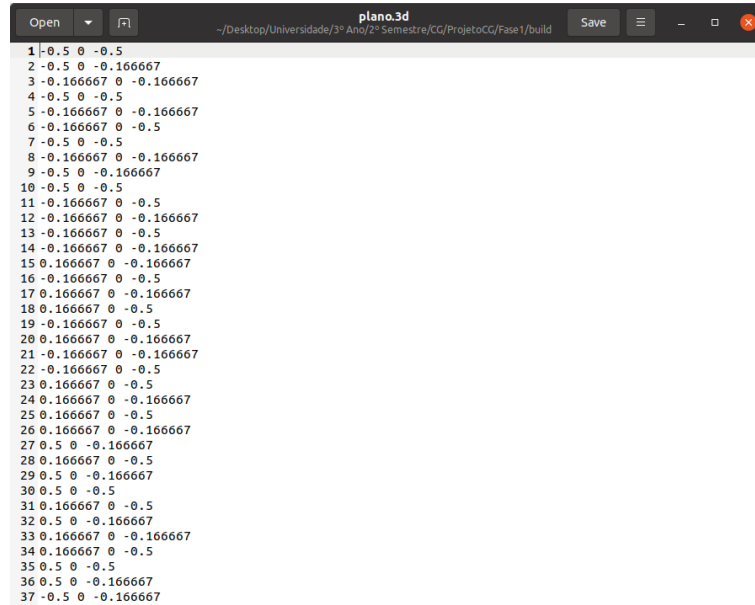
Figura 6: Cone com raio 1, tamanho 2, 12 *slices* e 3 *stacks*



## Escrita do ficheiro

Depois de serem calculados os vértices que fazem parte da figura e os mesmo serem guardados na struct *figure*, esta irá ser passada para um ficheiro *.3d* com o nome recebido no *input* do utilizador.

Este ficheiro contém todos os pontos necessários para desenhar uma determinada figura, sendo que em cada linha estão presentes as coordenadas de um dado ponto separadas por espaços.



```
1|-0.5 0 -0.5
2|-0.5 0 -0.166667
3|-0.166667 0 -0.166667
4|-0.5 0 -0.5
5|-0.166667 0 -0.166667
6|-0.166667 0 -0.5
7|-0.5 0 -0.5
8|-0.166667 0 -0.166667
9|-0.5 0 -0.166667
10|-0.5 0 -0.5
11|-0.166667 0 -0.5
12|-0.166667 0 -0.166667
13|-0.166667 0 -0.5
14|-0.166667 0 -0.166667
15|0.166667 0 -0.166667
16|-0.166667 0 -0.5
17|0.166667 0 -0.166667
18|0.166667 0 -0.5
19|-0.166667 0 -0.5
20|0.166667 0 -0.166667
21|-0.166667 0 -0.166667
22|-0.166667 0 -0.5
23|0.166667 0 -0.5
24|0.166667 0 -0.166667
25|0.166667 0 -0.5
26|0.166667 0 -0.166667
27|0.5 0 -0.166667
28|0.166667 0 -0.5
29|0.5 0 -0.166667
30|0.5 0 -0.5
31|0.166667 0 -0.5
32|0.5 0 -0.166667
33|0.166667 0 -0.166667
34|0.166667 0 -0.5
35|0.5 0 -0.5
36|0.5 0 -0.166667
37|-0.5 0 -0.166667
```

Figura 7: Exemplo de um ficheiro *.3d*

# Engine

## Leitura do ficheiro

Para a leitura do ficheiro é utilizado o *TinyXML*, conforme sugerido pelos docentes no enunciado do trabalho prático.

Começa-se pela leitura do campo *"camera"* do ficheiro, onde são obtidas todas as definições da câmara, sendo que o campo *"position"* corresponde às coordenadas *xyz* da posição inicial da mesma, e os restantes dois campos (*"lookAt"*, *"up"*) correspondem a definições adicionais da câmara. Por fim, obtém-se o conteúdo do campo *"projection"*, que corresponde aos parâmetros da função **gluPerspective**.

Após terem sido lidas as coordenadas da posição inicial da câmara, estas são convertidas para coordenadas polares e armazenadas numa estrutura de dados, de forma a que futuramente possam ser usadas para possibilitar a sua movimentação.

Finalizada a leitura das definições da câmara, passa-se à leitura do conteúdo do campo *"models"*. Neste campo encontram-se os nomes dos ficheiros *.3d* que terão de ser representados em uma figura. Ao obter o nome de um ficheiro, é feita também a leitura do mesmo, convertendo as instruções contidas no ficheiro em uma *figure*. No final da conversão esta é armazenada em um mapa, juntamente com as outras *figures*, no caso de ser pedida mais alguma no ficheiro *.xml*.

## Desenho das Primitivas

Após ter sido feita a leitura dos ficheiros, ficamos com um mapa *figurasMap*, onde estão contidas todas as figuras pedidas pelo ficheiro. A chave deste mapa é representada por um número inteiro pertencente ao intervalo  $[0, N]$ , sendo  $N-1$  o número total de figuras obtidas.

Inicialmente a variável *activarFig* estará inicializada como 0. Deste modo, na função **renderScene** vai ser desenhada a figure que no mapa é representada pela key 0. A partir do input do teclado (com as teclas "a" e "d"), é possível incrementar ou decrementar o valor desta variável, sendo redesenhada a cena com a figura cuja key corresponde ao valor de *activarFig*.

## Movimentação da Câmara

A movimentação da câmara é feita com o uso de coordenadas polares. Estas coordenadas poderão ser atualizadas a partir do uso das setas no teclado, ou do "+" e "-" para as ações de "zoom in" e "zoom out".

Conforme referido anteriormente, ao ler as coordenadas iniciais da posição da câmara, estas são convertidas para coordenadas polares e armazenadas em um struct que contém os campos "alpha", "beta" e "distance". Estes serão os campos a atualizar na eventualidade de se pretender mover a câmara. O campo beta corresponderá à altura da câmara, o campo alpha corresponderá à posição da câmara no plano xz e o campo distance corresponderá à distancia da câmara à origem.

Ao configurar a câmara na função **gluLookAt**, em vez de se colocarem as coordenadas *xyz* que foram lidas no ficheiro, são colocadas as conversões das coordenadas polares em coordenadas cartesianas. Assim, ao se modificarem as coordenadas polares que foram armazenadas (com recurso ao input do teclado), é também modificada a posição da câmara.

# Conclusão

O trabalho desenvolvido permitiu consolidar as noções em relação a primitivas gráficas, bem como expandir o nosso conhecimento sobre o GLUT e as suas funcionalidades, e ainda obter mais experiência com linguagens de marcação como a linguagem XML que foi utilizada na realização deste projeto.

De facto, a aplicação cumpre os requisitos fundamentais para o desenho gráfico de qualquer figura pedida pelos requisitos fornecidos no enunciado do trabalho.

Por fim, importa salientar que foram realizadas todas as funcionalidades obrigatórias e algumas adicionais, de forma a complementar e otimizar ainda mais o projeto em questão.