



UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Computação Gráfica
Fase I – Primitivas Gráficas

João Neves (a81366) Luís Manuel Pereira (a77667)
Rui Fernandes (a89138) Tiago Ribeiro (a76420)

Março 2021

Resumo

O presente relatório descreve o trabalho prático realizado no âmbito da disciplina de *Computação Gráfica*, ao longo do segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo da primeira fase do trabalho prático foi criar um gerador de vértices de quatro primitivas gráficas: plano, paralelepípedo, esfera e cone, tendo em consideração diferentes parâmetros, nomeadamente a altura, a largura, a profundidade e o número de divisões. Também foi necessário criar um motor capaz de ler um ficheiro XML e exibir a respetiva primitiva gráfica.

Neste documento descrevemos sucintamente a aplicação desenvolvida discutimos as decisões tomadas durante a realização do trabalho prático.

Conteúdo

1	Introdução	1
2	Arquitetura	1
2.1	<i>Generator</i>	2
2.2	<i>Engine</i>	2
3	Primitivas Gráficas	4
3.1	Plano	4
3.2	Paralelepípedo	5
3.3	Esfera	7
3.4	Cone	9
4	Conclusão	10

Lista de Figuras

1	Comando <i>help</i> do <i>generator</i>	1
2	Comando <i>help</i> do <i>engine</i>	2
3	<i>Output</i> gerado pelo <i>engine</i>	3
4	Representação de um plano centrado na origem	4
5	Plano com comprimento 8	5
6	Representação de um paralelepípedo centrado na origem	6
7	Paralelepípedo sem divisões, com dimensões $3 \times 3 \times 3$	6
8	Paralelepípedo de dimensões $4 \times 4 \times 4$, e 3 divisões por face	7
9	Geometria de uma esfera	7
10	Esfera de raio 8, construída com 50 <i>slices</i> e 50 <i>stacks</i>	9
11	Cone de raio 4 e altura 4, construído com 50 <i>slices</i> e 50 <i>stacks</i>	10

1 Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de *Computação Gráfica* e tem como objetivo desenvolver um motor gráfico genérico para representar objetos a 3 dimensões.

O projeto está dividido em várias fases, sendo que nesta primeira fase pretende-se a realização de dois sistemas – um primeiro que guarde num ficheiro informações relativas a uma primitiva gráfica que se pretende desenhar futuramente, usando o GLUT, e um segundo mecanismo, que, lendo de um ficheiro XML, desenhará o modelo segundo as indicações contidas no ficheiro e os modelos gerados anteriormente, obtendo-se assim uma figura.

2 Arquitetura

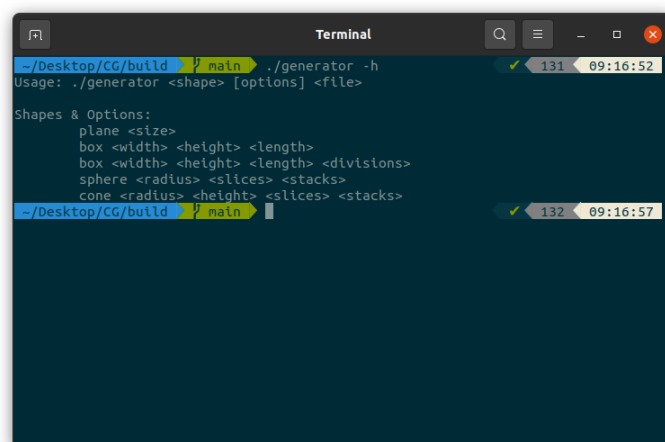
Após analisar o problema, foi decidido que o projeto estaria dividido em dois executáveis, o **generator** e o **engine**. Para além destes, foi também desenvolvido um módulo **vertex** comum aos dois, que representa um ponto no espaço tridimensional.

O *generator* destina-se a gerar os pontos que permitem, posteriormente, desenhar um conjunto de primitivas gráficas. Desta forma, interpreta os argumentos da linha de comandos, invocando as funções adequadas de modo a produzir um ficheiro com a representação destes pontos.

O *engine*, por sua vez, destina-se a representar as primitivas produzidas anteriormente. Para a leitura dos ficheiros XML, foi utilizada a biblioteca **tinyxml2**, que disponibiliza um vasto conjunto de funções para o tratamento deste tipo de ficheiros.

Adicionou-se, também, às funcionalidades um comando *help*, que poderá ser utilizado invocando o executável com a extensão **-h**. Este comando apresenta um manual que permite aos utilizadores visualizar quais os comandos disponíveis e os seus argumentos. Este comando está disponível tanto para o *generator* como para o *engine*, como se pode verificar nas seguintes figuras.

Por fim, com o propósito de melhorar a perspetiva sobre a própria figura, é possível mudar a posição da câmara, mudar o formato de desenho da figura para ponto, linha ou até mesmo sólido e ver o interior do modelo.

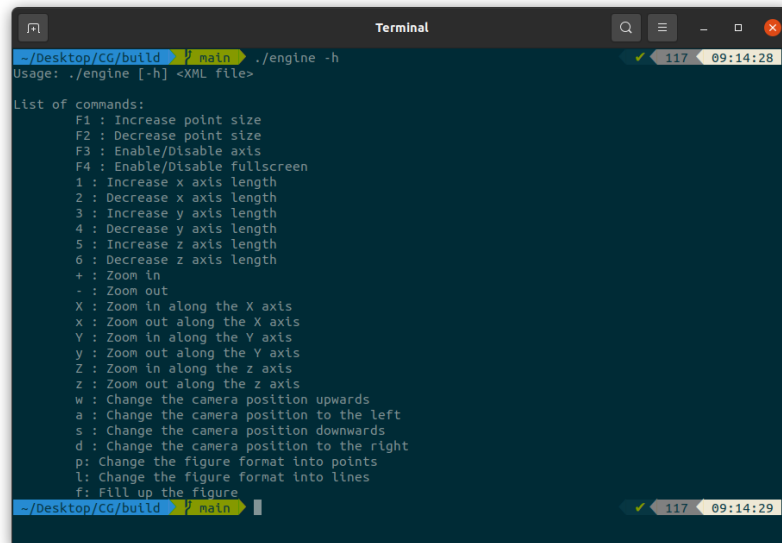


```
Terminal
~/Desktop/CG/build % main ./generator -h
Usage: ./generator <shape> [options] <file>

Shapes & Options:
plane <size>
box <width> <height> <length>
box <width> <height> <length> <divisions>
sphere <radius> <slices> <stacks>
cone <radius> <height> <slices> <stacks>

~/Desktop/CG/build % main
```

Figura 1: Comando *help* do *generator*

A terminal window titled "Terminal" with a dark blue background. The prompt is "~/Desktop/CG/build" and the command is "main ./engine -h". The output shows the usage and a list of commands. The prompt is now "~/Desktop/CG/build" and the command is "main".

```
~/Desktop/CG/build main ./engine -h
Usage: ./engine [-h] <XML file>

List of commands:
F1 : Increase point size
F2 : Decrease point size
F3 : Enable/Disable axis
F4 : Enable/Disable fullscreen
1 : Increase x axis length
2 : Decrease x axis length
3 : Increase y axis length
4 : Decrease y axis length
5 : Increase z axis length
6 : Decrease z axis length
+ : Zoom in
- : Zoom out
X : Zoom in along the X axis
x : Zoom out along the X axis
Y : Zoom in along the Y axis
y : Zoom out along the Y axis
Z : Zoom in along the z axis
z : Zoom out along the z axis
w : Change the camera position upwards
a : Change the camera position to the left
s : Change the camera position downwards
d : Change the camera position to the right
p : Change the figure format into points
l : Change the figure format into lines
f : Fill up the figure

~/Desktop/CG/build main
```

Figura 2: Comando *help* do *engine*

2.1 Generator

Tal como dito anteriormente, o *generator* é responsável pelo cálculo das coordenadas dos pontos necessários para o desenho dos triângulos que compõem as diversas primitivas gráficas.

Neste módulo foram definidas 4 primitivas gráficas: plano, paralelepípedo, esfera e cone. Como tal, existiu a necessidade adotar algoritmos distintos em função da primitiva em questão, algoritmos esses que serão discutidos a seguir.

O formato escolhido para representar cada um dos pontos gerados consiste de colocar um ponto por linha com as coordenadas x , y e z , por esta ordem e separadas por um espaço. Assim, o conjunto de pontos $(1, 0, 0)$, $(0, 1, 0)$ e $(0, 0, 1)$ seria representado como:

```
1 0 0
0 1 0
0 0 1
```

2.2 Engine

O *engine* é a aplicação responsável por receber o ficheiro de configuração de uma *scene* em XML com todos os ficheiros que contém os modelos a ser carregados para serem posteriormente gerados através do OpenGL. Cada ficheiro com modelos presentes no ficheiro de configuração é analisado pelo programa para dele serem extraídos os vários pontos que são depois adicionados a uma estrutura de dados adequada.

Assim, é necessário percorrer todas as linhas destes ficheiros e guardar cada entrada como um ponto (x, y, z) no espaço 3D. Para isso, foi criada uma classe **Vertex** que representa cada um destes pontos. Deste modo, cada entrada do ficheiro é instanciada como um objeto desta classe, sendo esta posteriormente adicionada à lista de pontos a desenhar, `std::vector<Vertex> vertices`.

Uma vez tendo os pontos necessários para o desenho dos triângulos que compõem diferentes primitivas, é, então, possível desenhar as figuras. Para tal, é necessário percorrer a estrutura supracitada e desenhar cada um dos pontos invocando, para isso, funções do OpenGL.

De seguida, cada modelo é renderizado no ecrã, chamando a função `glVertex3f` para cada um dos seus pontos previamente carregados para memória e, por fim, inicia-se o *main loop* do GLUT. Desta forma, os modelos são carregados para memória apenas uma vez.

```
class Vertex {  
    private:  
        float _x, _y, _z;  
        // (...)  
};
```

A título de exemplo, considerando os ficheiros `sphere.3d`, `coneup.3d` e `conedown.3d` gerados utilizando os comandos `./generator sphere 4 50 50`, `./generator cone 4 4 50 50`, e `./generator cone 4 -4 50 50`, respetivamente, e considerado o ficheiro XML `scene.xml` apresentado a seguir, o *output* gerado pelo *engine* é o seguinte:

```
<scene>  
    <model file="sphere.3d" />  
    <model file="coneup.3d" />  
    <model file="conedown.3d" />  
</scene>
```

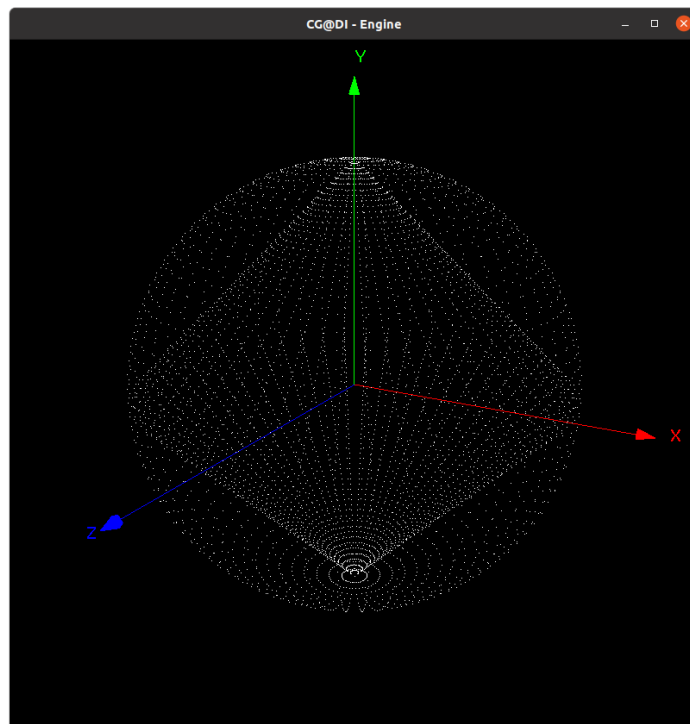


Figura 3: *Output* gerado pelo *engine*

3 Primitivas Gráficas

Para esta fase do trabalho prático foram desenhadas quatro primitivas gráficas, nomeadamente um plano, um cubo, uma esfera e um cone. Serão, em seguida, apresentados os algoritmos para o cálculo dos vértices necessários para o desenho de cada uma destas primitivas.

3.1 Plano

Para a construção do plano, que pode ser interpretado como 2 triângulos, é necessário um argumento: a sua dimensão (**size**).

Sabendo que um triângulo é definido por 3 vértices, verifica-se que 2 dos 4 vértices necessários à definição de um plano irão ser partilhados por ambos os triângulos. Sabendo também que o plano está centrado na origem e como este é desenhado sobre o plano XZ , o valor de y para todos os pontos é constante e igual 0. Por fim, uma vez que se pretende que este fique centrado na origem, é necessário dividir o comprimento passado como argumento (**size**) por dois de modo a obter as coordenadas em X e Z necessárias para formar ambos os triângulos.

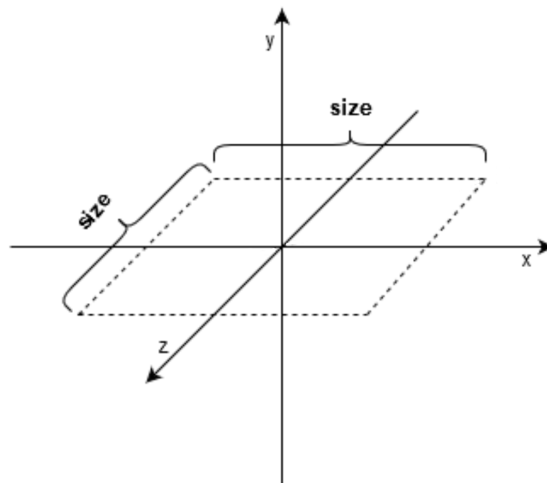


Figura 4: Representação de um plano centrado na origem

Assim, tal como ilustrado na figura anterior, os vértices dos triângulos terão as seguintes coordenadas:

$$\left(\frac{size}{2}, 0, \frac{size}{2}\right), \left(-\frac{size}{2}, 0, -\frac{size}{2}\right), \left(-\frac{size}{2}, 0, \frac{size}{2}\right)$$

e

$$\left(-\frac{size}{2}, 0, -\frac{size}{2}\right), \left(\frac{size}{2}, 0, \frac{size}{2}\right), \left(\frac{size}{2}, 0, -\frac{size}{2}\right)$$

O resultado da renderização pode ser visto na figura seguinte:

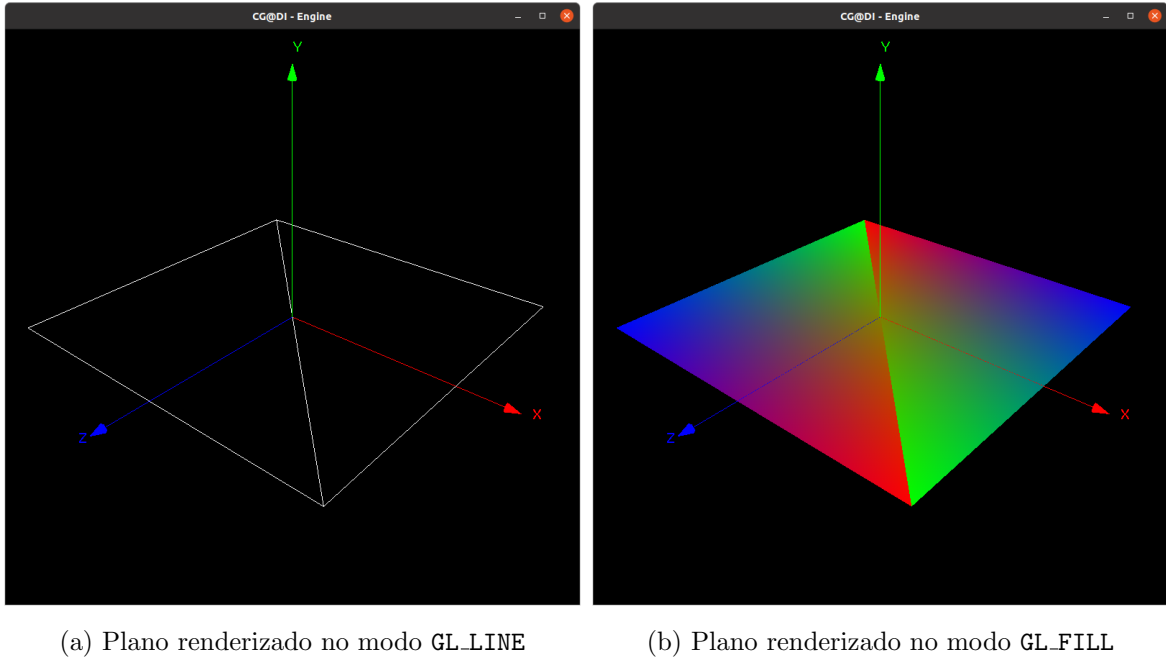


Figura 5: Plano com comprimento 8

3.2 Paralelepípedo

Um paralelepípedo é composto por 6 faces. Sendo assim, para a sua construção são necessários vários argumentos: o comprimento (**width**), a altura (**height**) e o comprimento (**length**). Pode, opcionalmente, ser também passado como argumento o número de divisões em cada face¹ (**divisions**).

Tendo o paralelepípedo centrado na origem de forma a facilitar os cálculos, e sabendo que cada uma das faces é constituída por *divisions*² quadriláteros, e sendo cada um deles formado por dois triângulos, é possível determinar o espaçamento entre cada um destes quadriláteros, espaçamento esse que será utilizado para iterar sobre todo o sólido:

$$d_x = \frac{width}{divisions}$$

$$d_y = \frac{height}{divisions}$$

$$d_z = \frac{length}{divisions}$$

Desta forma, torna-se, então, possível calcular as coordenadas de todos os vértices necessários à construção do sólido. Tal como ilustrado na figura seguinte, e usando como exemplo um triângulo da face frontal de um paralelepípedo, teríamos as seguintes coordenadas, onde *i* e *j* serão variáveis que permitem iterar ao longo das faces:

¹Um paralelepípedo sem divisões nas faces pode ser visto como um paralelepípedo com divisões tal que **divisions** = 1.

Vértice A:

$$\left(d_x \times i - \frac{width}{2}, d_y \times j - \frac{height}{2}, \frac{length}{2} \right)$$

Vértice P:

$$\left(d_x \times i - \frac{width}{2} + d_x, d_y \times j - \frac{height}{2}, \frac{length}{2} \right)$$

Vértice R:

$$\left(d_x \times i - \frac{width}{2}, d_y \times j - \frac{height}{2} + d_y, \frac{length}{2} \right)$$

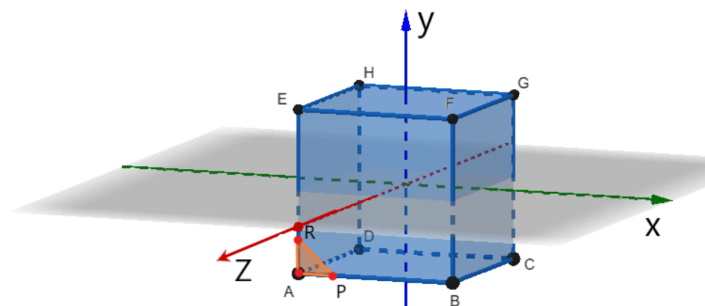
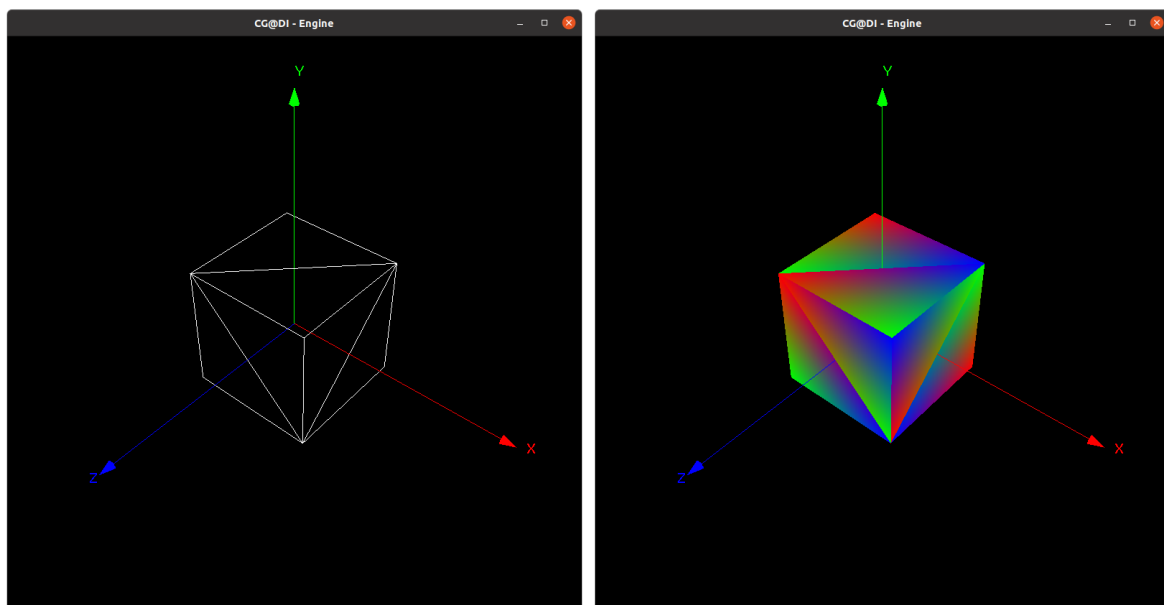


Figura 6: Representação de um paralelepípedo centrado na origem

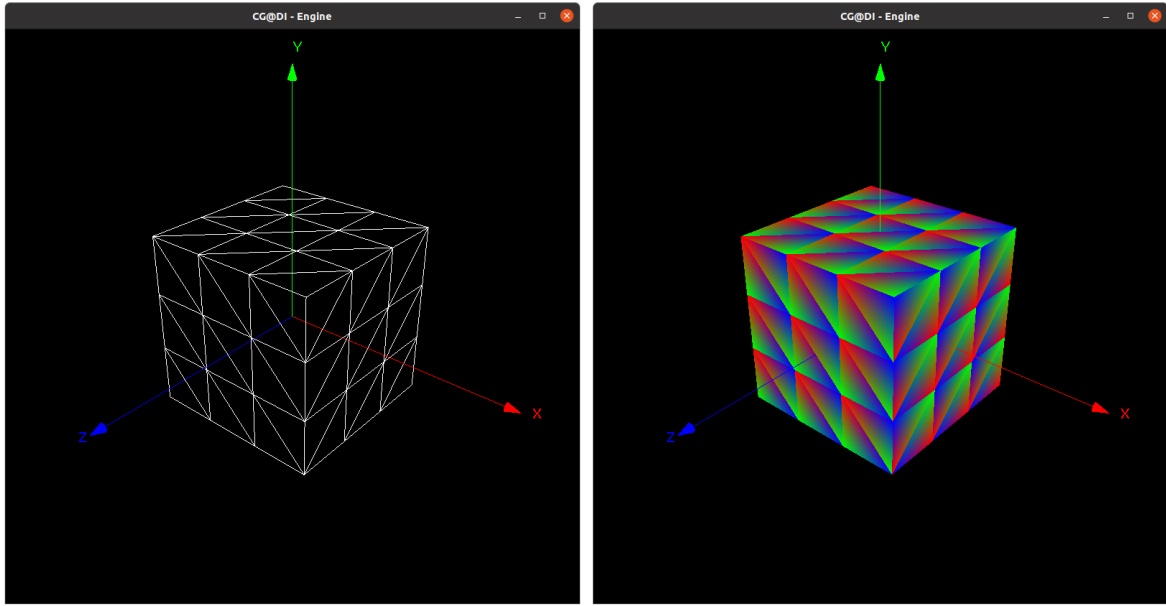
O resultado da renderização pode ser visto na figura seguinte:



(a) Paralelepípedo sem divisões renderizado no modo `GL_LINE`

(b) Paralelepípedo sem divisões renderizado no modo `GL_FILL`

Figura 7: Paralelepípedo sem divisões, com dimensões $3 \times 3 \times 3$



(a) Paralelepípedo com divisões nas faces renderizado no modo `GL_LINE`

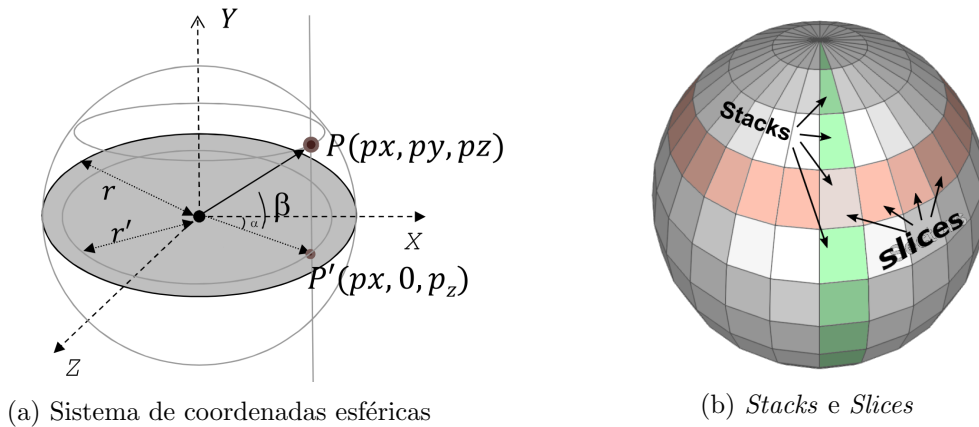
(b) Paralelepípedo com divisões nas faces renderizado no modo `GL_FILL`

Figura 8: Paralelepípedo de dimensões $4 \times 4 \times 4$, e 3 divisões por face

3.3 Esfera

Para a criação da esfera são utilizadas coordenadas esféricas de modo a facilitar o cálculo das coordenadas de cada um dos pontos. Deste modo, são necessárias duas variáveis para representar os dois ângulos α e β , representados na figura que se segue, em função dos quais serão expressas as coordenadas de cada ponto, e que serão atualizadas em cada iteração.

Uma esfera pode ser dividida em *stacks* e *slices*, que serão usadas para iterar à volta da mesma. A interseção entre uma *slice* e uma *stack* forma um retângulo, composto por dois triângulos.



(a) Sistema de coordenadas esféricas

(b) *Stacks* e *Slices*

Figura 9: Geometria de uma esfera

Usando coordenadas esféricas, podemos facilmente obter as coordenadas cartesianas para guardar posteriormente nas respectivas estruturas de dados. Analisado a figura 9a, e considerando os ângulos α e β , podemos obter as coordenadas de qualquer ponto sobre a esfera, sendo que para determinar as coordenadas do ponto P para o eixo dos X basta fazer o produto do cosseno do ângulo α com o cosseno do ângulo β e multiplicar estes dois fatores pelo raio da esfera. Para o eixo dos Z o raciocínio é semelhante, sendo que agora basta fazer o produto do seno do ângulo α com o cosseno do ângulo β e novamente multiplicando pelo o raio da esfera. Por fim, para obter o eixo dos Y , basta multiplicar o seno do ângulo β com o raio da esfera e obtemos diretamente a coordenada Y .

Como forma de exemplo podemos observar que para calcular o ponto P' bastava usar o cosseno de α para a coordenada em X , o seno de β para a coordenada em Z e simplesmente igualar a coordenada em Y a zero visto que estamos no plano XZ , no qual a coordenada y toma sempre o valor zero. Usando agora o primeiro cálculo para calcular este ponto P' :

$$\begin{cases} x = \cos \alpha \times \cos \beta \times r \\ y = \sin \beta \times r \\ z = \sin \alpha \times \cos \beta \times r \end{cases}$$

Como o ângulo β é igual a 0, anulando os senos e cossenos da fórmula, temos como resultado a previsão inicial:

$$\begin{cases} x = \cos \alpha \times 1 \times r = \cos \alpha \times r \\ y = 0 \times r = 0 \\ z = \sin \alpha \times 1 \times r = \sin \alpha \times r \end{cases}$$

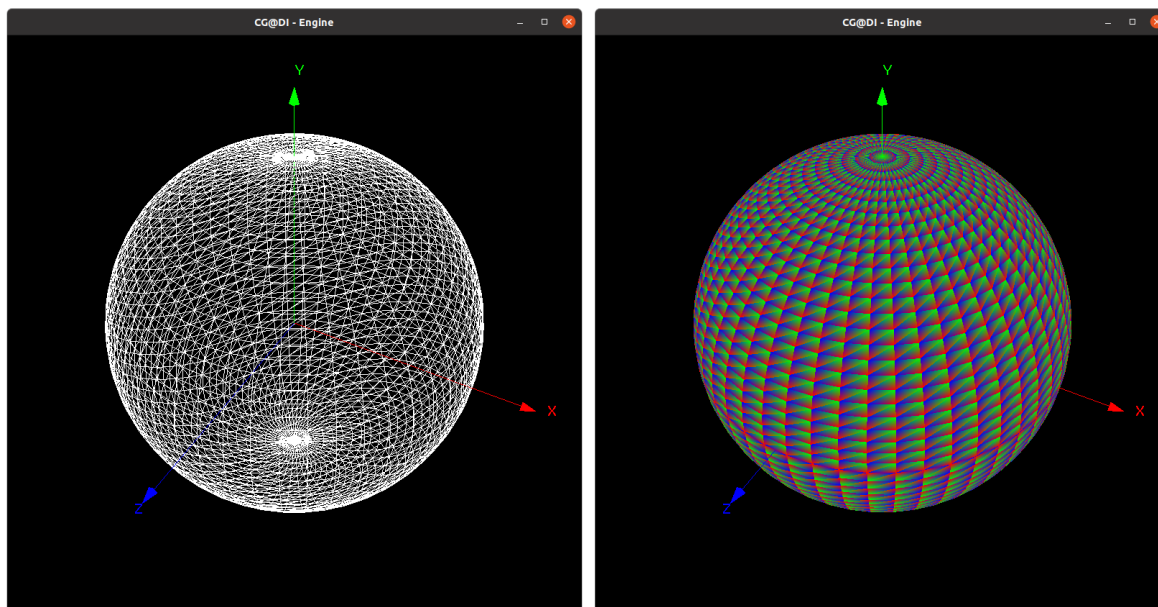
Atendendo ao número de *stacks* e *slices*, é possível calcular dois deslocamentos: o deslocamento horizontal, d_α , entre 0 rad e 2π rad, e o deslocamento vertical, d_β , entre 0 rad e π rad.

$$d_\alpha = \frac{2\pi}{slices}$$

$$d_\beta = \frac{\pi}{stacks}$$

Assim, e com estes deslocamentos, podemos proceder ao cálculo dos vertices, tendo em consideração que o ângulo α é incrementado a cada iteração no valor de d_α e, quando este atingir o valor de 2π rad (*i.e.* quando der uma volta completa em torno da esfera), é, então, incrementado o ângulo β , recorrendo ao valor d_β , e α é inicializado a partir do 0, novamente, e é mais uma vez percorrida a esfera horizontalmente, até que ângulo β atinja o valor de π rad.

O resultado da renderização pode ser visto na figura seguinte:



(a) Esfera renderizada no modo GL.LINE

(b) Esfera renderizada no modo GL.FILL

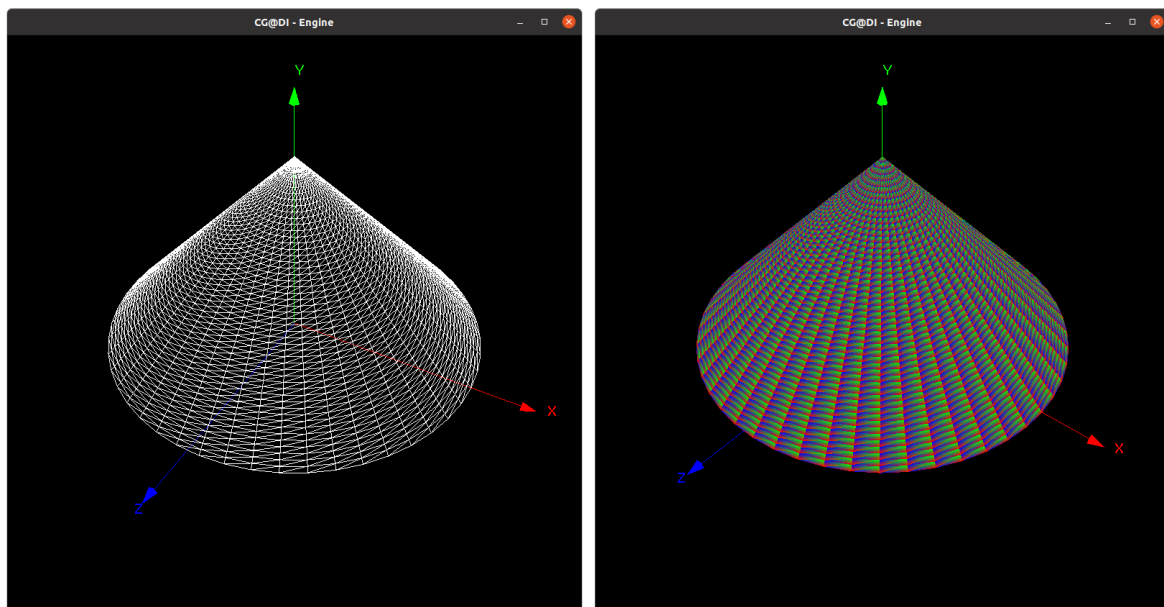
Figura 10: Esfera de raio 8, construída com 50 *slices* e 50 *stacks*

3.4 Cone

Para desenhar um cone é necessário saber o seu raio (**radius**), a sua altura (**height**), o número de *slices* (**slices**) e o número de *stacks* (**stacks**).

O algoritmo para o desenho de um cone introduz cálculos trigonométricos para o cálculo das coordenadas dos vértices, em função dos mesmos ângulos α e β referidos anteriormente. A diferença do cone para a esfera reside no facto de o ângulo α estar continuamente a decrescer à medida que percorremos o eixo Y , isto sabendo que o vetor da origem O ao vértice do cone está paralelo e com o mesmo sentido do eixo positivo Y . Dividindo então este ângulo inicial α pelo número de *slices*, obtemos o deslocamento radial $\Delta\alpha$ que permitirá decrementar o ângulo α a cada camada do cone. Ao decrementar este ângulo, podemos facilmente obter as coordenadas dos pontos utilizando um raciocínio análogo ao da construção da esfera.

O resultado da renderização pode ser visto na figura seguinte:



(a) Cone sem divisões renderizado no modo `GL_LINE`

(b) Paralelepípedo sem divisões renderizado no modo `GL_FILL`

Figura 11: Cone de raio 4 e altura 4, construído com 50 *slices* e 50 *stacks*

4 Conclusão

A elaboração desta primeira fase do trabalho prático permitiu a consolidação de conhecimentos relativos ao OpenGL e GLUT e, ao mesmo tempo, a linguagem de programação de C++, e até mesmo relembrar certos aspetos no âmbito da geometria.

Fazendo uma análise geral ao trabalho desenvolvido ao longo desta primeira fase, conclui-se que foram cumpridos todos os objetivos que foram propostos.

Como trabalho futuro gostaríamos de implementar outras primitivas no *generator*, como por exemplo o *torus*, assim como a implementação de uma câmara *FPS* – *First-Person Shooter* no *engine*.