

---

# FASE 1

## Primitivas Gráficas

---



A95414  
Artur Luís



A95835  
Bianca Vale



A95454  
Lara Ferreira



A95111  
Luís Ferreira

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Estrutura do Projeto</b>	<b>2</b>
2.1	Aplicações . . . . .	2
2.1.1	<i>Engine</i> . . . . .	2
2.1.2	<i>Generator</i> . . . . .	2
2.2	Classes . . . . .	2
2.2.1	Ponto . . . . .	2
2.2.2	Forma . . . . .	2
2.2.3	Camera . . . . .	3
2.3	Ferramentas utilizadas . . . . .	3
<b>3</b>	<b>Primitivas Geométricas</b>	<b>4</b>
3.1	Plano . . . . .	4
3.2	Caixa . . . . .	4
3.3	Cone . . . . .	5
3.4	Esfera . . . . .	6
3.5	Cilindro . . . . .	7
<b>4</b>	<b><i>Generator</i></b>	<b>8</b>
4.1	Funcionalidades . . . . .	8
4.2	Demonstração . . . . .	8
<b>5</b>	<b><i>Engine</i></b>	<b>9</b>
5.1	Demonstração . . . . .	9
<b>6</b>	<b>Apresentação dos Modelos</b>	<b>10</b>
6.1	Plano . . . . .	10
6.2	Cubo . . . . .	11
6.3	Cone . . . . .	12
6.4	Esfera . . . . .	13
6.5	Esfera e Plano . . . . .	14
6.6	Cilindro . . . . .	15
<b>7</b>	<b>Conclusão</b>	<b>16</b>

---

## 1 Introdução

Foi-nos proposto, no âmbito da Unidade Curricular de *Computação Gráfica* o desenvolvimento de vários modelos em 3 dimensões, com a utilização do *GLUT* e do *OpenGL* em C++.

Este trabalho divide-se em quatro fases.

Nesta primeira etapa do trabalho, o objetivo é criar alguns modelos 3D, como um **Plano**, uma **Caixa**, um **Cone** e uma **Esfera**. Além dos modelos exigidos para esta fase, decidimos incluir a criação de um **Cilindro**.

Estes modelos não só refletem a aplicação dos conceitos aprendidos em sala de aula, como também proporcionam uma oportunidade para explorar e compreender os princípios fundamentais da Computação Gráfica na prática.

---

## 2 Estrutura do Projeto

Neste capítulo iremos explicar a nossa estratégia, o seu desenvolvimento e implementação dos modelos referidos anteriormente.

### 2.1 Aplicações

Esta fase requer duas aplicações: uma para gerar ficheiros com a informação dos modelos (*Generator*) e o motor que lê um ficheiro de configuração (*Engine*), escrito em *XML*, e apresenta os modelos.

#### 2.1.1 *Engine*

O ficheiro *engine.cpp* diz respeito ao motor que irá ler o ficheiro *XML*. Este motor tem a capacidade de carregar dados cruciais para a visualização, incluindo a posição da câmara, a perspetiva e os modelos das formas, todos provenientes do ficheiro mencionado.

O motor desenha as formas no ecrã e permite que o utilizador interaja com a cena usando o teclado.

#### 2.1.2 *Generator*

O gerador, presente no ficheiro *generator.cpp*, é responsável por criar as diferentes formas geométricas, convertendo as primitivas geométricas num conjunto de vértices dos triângulos necessários para representar essas formas, com auxílio das classes **ponto** e **forma**. Estes são guardados em ficheiros na pasta *files* para permitir o acesso e uso posterior por outras aplicações.

### 2.2 Classes

As seguintes classes auxiliam o *generator* e o *engine* de modo a que estes consigam cumprir as suas funções.

#### 2.2.1 Ponto

A classe *ponto* é crucial tanto para o *Generator* como para o *Engine*. Representa as coordenadas tridimensionais de um ponto e fornece métodos para aceder às suas coordenadas *x*, *y* e *z*. É fundamental para representar e manipular pontos no espaço tridimensional em ambas as aplicações.

#### 2.2.2 Forma

A classe **forma** é fundamental tanto para o *Engine* como para o *Generator*. Esta armazena os pontos que compõem as formas geométricas. No *Engine*, a função *escreverParaFicheiro* escreve os pontos das formas num arquivo de saída. No *Generator*, além da função *escreverParaFicheiro*, a classe **forma** possui a função *adicionarPonto*, que acrescenta um novo ponto à forma. Os pontos são escritos no arquivo com as suas coordenadas *x*, *y* e *z*.

---

### 2.2.3 Camera

A classe **camera** no *Engine* é essencial para controlar os parâmetros de visualização da cena. Inclui métodos para obter e definir a posição da câmara, o ponto de visualização, o vetor de orientação "up", o campo de visão (FOV), bem como os planos de corte próximo e distante (*near* e *far*). Estes métodos permitem ajustar a perspetiva da cena conforme necessário.

## 2.3 Ferramentas utilizadas

Em adição ao *OpenGL* onde se encontram as funcionalidades gráficas do nosso projeto, utilizamos também o **TinyXML-2** para fazer o *parsing* dos ficheiros *XML*.

---

## 3 Primitivas Geométricas

### 3.1 Plano

A primeira primitiva que decidimos criar foi o *plano*, pois é a primitiva mais simples e que serve para posteriormente desenvolver o *cubo*.

Para conseguirmos desenvolver o *plano* precisamos do **número de quadrados em cada lado** e do **tamanho do plano**.

Além disto, é necessário notar que o plano deve estar contido no plano XZ (logo todas as coordenadas terão Y=0).

Para definir o tamanho de cada quadrado, simplesmente dividimos o tamanho de cada lado pelo número de quadrados.

$$ladoCadaQua = tam\_lado/num\_divs$$

Também é necessário definir um ponto inicial que nos permita calcular os vértices dos triângulos que formam o plano. Verificamos que poderíamos utilizar a metade do lado do plano para definir um dos pontos da extremidade:

$$ref = tam\_lado/2$$

a partir do qual conseguimos calcular todos os outros pontos necessários para os primeiros dois triângulos (o primeiro quadrado)

- Ponto 1: (-ref,0,-ref)
- Ponto 2: (-ref,0,-ref+ladoCadaQua)
- Ponto 3: (-ref+ladoCadaQua,0,-ref+ladoCadaQua)
- Ponto 4: (-ref,0,-ref)
- Ponto 5: (-ref+ladoCadaQua,0,-ref+ladoCadaQua)
- Ponto 6: (-ref+ladoCadaQua,0,-ref)

Realizamos dois ciclos para preencher todo o plano com vértices, o ciclo externo itera sobre as divisões ao longo do eixo Z, enquanto o ciclo interno itera sobre as divisões ao longo do eixo X.

### 3.2 Caixa

Como mencionamos no ponto anterior, a forma como construímos o *Plano* é bastante útil para o desenvolvimento do cubo.

Como o cubo é composto por 6 faces, geramos o cubo tratando cada face como sendo um plano.

Desta forma, construímos 6 planos para representar todas as faces, gerando assim o *Cubo*

---

### 3.3 Cone

A geração do *Cone* pode ser separada em dois elementos, a *Base* e os *lados*. Para a gerar a base, são necessárias algumas variáveis:

- *raio*: O raio da circunferência da base
- *nr\_slices*: o número de fatias ( $n^\circ$  de triângulos da base)
- *angulo\_slices*: ângulo usado para a posição dos vértices
- *alpha*: diferença angular entre os vértices

Para todos os triângulos que compõe a base inicializamos um vértice na origem (0,0,0) e um vértice nas coordenadas

$$raio * \sin(anguloSlices), 0, raio * \cos(anguloSlices)$$

Para formar o triângulo basta-nos criar um terceiro vértice cujas coordenadas X e Z sejam equivalentes ao destino que o ponto anterior teria caso fizesse uma deslocação através da circunferência da base.

Fazemos o mesmo raciocínio até ser percorrida toda a circunferência da base, preenchendo-a de triângulos.

Para desenhar os lados do cone, precisamos de variáveis distintas:

- *altura*: altura do cone
- *altura\_niveis*: diferença de altura de cada nível do cone
- *altura\_aumento*: altura do próximo nível do cone
- *raio\_2*: valor da diferença de tamanho entre raios de diferentes consecutivos
- *raio\_niveis*: valor do raio a cada nível

Para desenhar a face lateral do cone, é necessário construir vários triângulos que conectam os pontos ao longo das fatias do cone. A cada iteração do ciclo, calculamos os pontos de uma fatia com base na altura atual (**altura\_aumento**) e na altura da próxima fatia (**altura\_aumento + altura\_niveis**), assim como nos raios correspondentes a essas fatias (**raio\_niveis** e **raio\_niveis - raio\_2**).

#### 1. Iteração sobre as slices

- O loop `for(int i = 0; i < nr_slices; i++)` percorre as fatias do cone, onde `nr_slices` representa o número de fatias do cone.
- A variável `anguloSlices` calcula o ângulo entre as fatias. Em cada iteração, `anguloSlices` é incrementado em `alpha`, onde  $\alpha = \frac{2\pi}{nr\_slices}$ .

#### 2. Definição dos pontos

- Dentro do loop, são adicionados pontos que formam os lados do cone.
- Cada iteração do loop adiciona pontos que compõem um triângulo.
- Os pontos são adicionados em pares de três, formando triângulos que compõem os lados do cone.
- Os pontos são calculados com base nas coordenadas polares, utilizando as funções trigonométricas `sin` e `cos`.

---

### 3. Atualização dos valores para a próxima iteração

- Após a adição dos pontos para uma fatia, os valores de altura e raio são atualizados para a próxima iteração.
- A altura e o raio são ajustados com base no número de *stacks* e *slices* definidos.
- Para calcular os pontos da próxima fatia, os valores de altura e raio são atualizados.
- A altura é incrementada em `altura_niveis` a cada iteração, onde  $\text{altura\_niveis} = \frac{\text{altura}}{\text{nr\_stacks}}$ . Isso garante que a altura seja dividida uniformemente em cada fatia.
- O raio também é ajustado para cada fatia. Inicialmente, `raio_niveis` é definido como `raio - raio_2`, onde  $\text{raio\_2} = \frac{\text{raio}}{\text{nr\_stacks}}$ .

### 3.4 Esfera

Para construir a *esfera* temos de entender que é uma Primitiva em que todos os pontos da sua superfície se encontram à mesma distância da origem. Assim sendo, para a construir precisamos de saber:

- *raio*
- *nr\_slices*: número de divisões da base
- *nr\_stacks*: número de fatias da altura
- *anguloSlices*: aumento no *alfa* a cada iteração
- *anguloStack*: aumento no *beta* a cada iteração

Para representar a *esfera* precisamos de conseguir representar as coordenadas da superfície esférica, pelo que optamos por usar coordenadas esféricas, fazendo uso de um *alpha* que vai fazer variar a posição dos pontos pelas *slices* da esfera, e um *beta* que vai controlar a posição dos pontos pelas *stacks* da mesma.

Agora que temos como percorrer todos os pontos da esfera, passamos a criá-la. dividindo a esfera em 3 partes: cima, baixo e meio.

Para cada parte, temos um *loop* que itera sobre as diferentes "fatias" horizontais da esfera.

Assim, para todas as partes, somos capazes de criar todos os pontos necessários para a geração dos triângulos que compõem a superfície da esfera.



---

### 3.5 Cilindro

À semelhança do cone, a construção do cilindro também está dividida na construção das bases e dos seus lados.

Para criar a base do cilindro, utilizamos as seguintes variáveis:

- *nr\_slices*: Número de fatias horizontais do cilindro.
- *raio*: Raio da base do cilindro.
- *altura*: Altura total do cilindro.
- *anguloSlices*: Ângulo entre cada fatia horizontal, calculado como  $(2 \times \pi) / nr\_slices$ .
- *halfHeight*: Metade da altura do cilindro.

1. Iteramos sobre as fatias horizontais do cilindro:

- Para cada fatia, calculamos os vértices da base inferior e superior do cilindro.
- Os vértices são calculados usando coordenadas polares, onde cada vértice é definido por um ângulo  $\theta$  que varia de 0 a  $2 \times \pi$ .
- Os vértices são adicionados à estrutura de dados que representa a forma do cilindro.

Para criar os lados do cilindro, dividimos a altura total do cilindro em secções verticais chamadas "stacks".

Utilizamos as seguintes variáveis:

- *nr\_stacks*: Número de fatias verticais do cilindro.
- *alturaStep*: Altura de cada stack, calculada como  $altura / nr\_stacks$ .

1. Iteramos sobre as stacks:

- Para cada stack, calculamos os vértices dos triângulos que compõem os lados do cilindro.
- Para cada fatia horizontal, conectamos os vértices da fatia atual com os da fatia seguinte para formar triângulos que representam os lados do cilindro.
- Os vértices são adicionados à estrutura de dados que representa a forma do cilindro.

Este procedimento resulta na criação de um ficheiro contendo os dados essenciais para representar o cilindro num ambiente gráfico tridimensional, utilizando as coordenadas dos vértices para o desenhar.

---

## 4 *Generator*

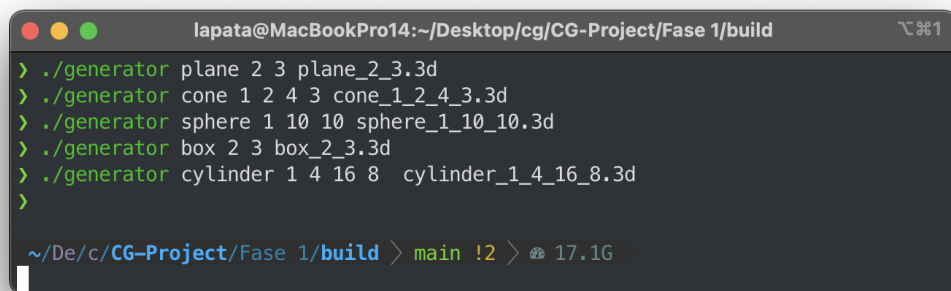
O gerador é encarregado de criar e armazenar informações para renderizar modelos tridimensionais. Gera as coordenadas dos vértices e outras propriedades essenciais, que são guardadas em ficheiros. Estes dados contêm detalhes sobre a geometria dos modelos, como a posição dos vértices e informações das faces, sendo fundamentais para a renderização dos modelos num ambiente gráfico.

### 4.1 Funcionalidades

Os modelos são construídos então através do *generator*:

- **Plano:** plane tam\_lado num\_divs nomedoficheiro.3d
- **Caixa:** box tam\_lado num\_divs nomedoficheiro.3d
- **Cone:** cone raio altura nr\_slices nr\_stacks nomedoficheiro.3d
- **Esfera:** sphere raio nr\_slices nr\_stacks nomedoficheiro.3d
- **Cilindro:** cylinder raio altura nr\_slices nr\_stacks nomedoficheiro.3d

### 4.2 Demonstração



```
lapata@MacBookPro14:~/Desktop/cg/CG-Project/Fase 1/build
> ./generator plane 2 3 plane_2_3.3d
> ./generator cone 1 2 4 3 cone_1_2_4_3.3d
> ./generator sphere 1 10 10 sphere_1_10_10.3d
> ./generator box 2 3 box_2_3.3d
> ./generator cylinder 1 4 16 8 cylinder_1_4_16_8.3d
>
~/De/c/CG-Project/Fase 1/build > main !2 > 17.1G
```

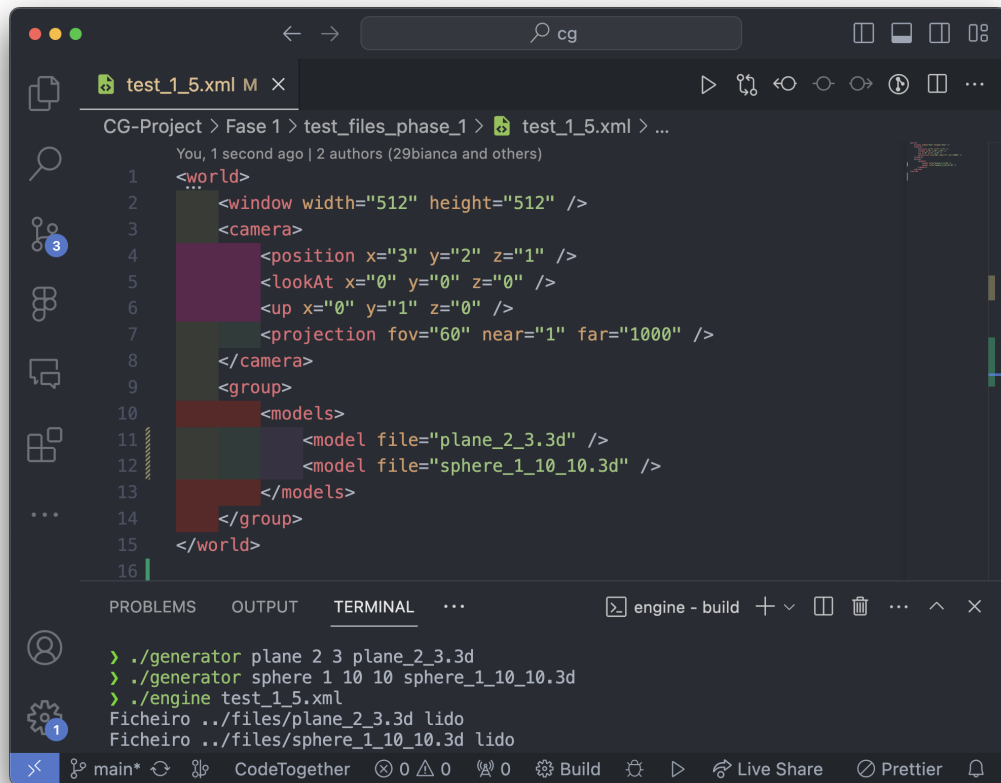
Figure 1: Invocação do generator para criação dos modelos

---

## 5 Engine

O motor lê ficheiros de configuração em *XML* para obter detalhes sobre os modelos a serem exibidos. Utiliza essas informações para apresentar os modelos no ecrã, controlando também as configurações de visualização.

### 5.1 Demonstração



The image shows a code editor window with a dark theme. The main editor displays an XML file named `test_1_5.xml`. The XML content is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<world>
  <window width="512" height="512" />
  <camera>
    <position x="3" y="2" z="1" />
    <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="plane_2_3.3d" />
      <model file="sphere_1_10_10.3d" />
    </models>
  </group>
</world>
```

Below the editor, there is a terminal window with the following commands and output:

```
> ./generator plane 2 3 plane_2_3.3d
> ./generator sphere 1 10 10 sphere_1_10_10.3d
> ./engine test_1_5.xml
Ficheiro ../files/plane_2_3.3d lido
Ficheiro ../files/sphere_1_10_10.3d lido
```

The terminal window also shows the status of the `engine - build` process, which is currently running.

Figure 2: Ficheiro XML e invocação das aplicações

---

## 6 Apresentação dos Modelos

### 6.1 Plano

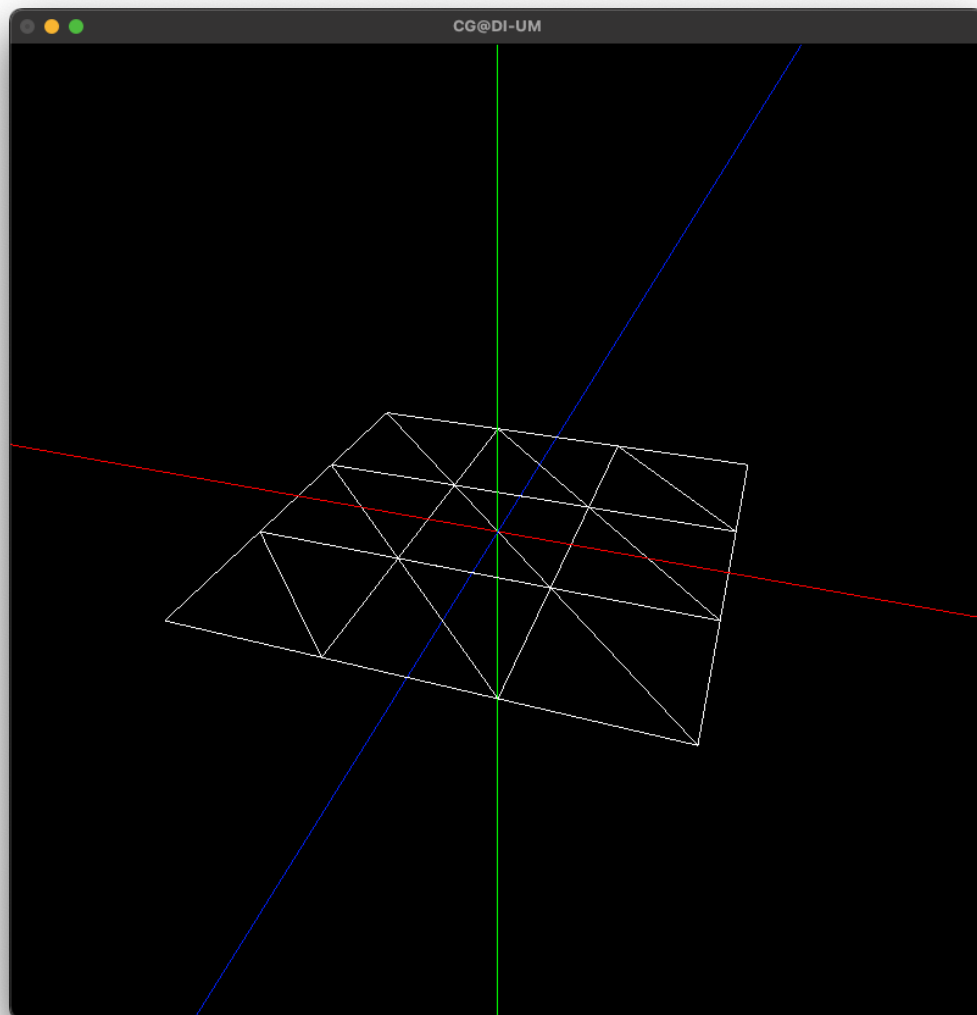


Figure 3: Plano com 2 de lado e 3 divisões

---

## 6.2 Cubo

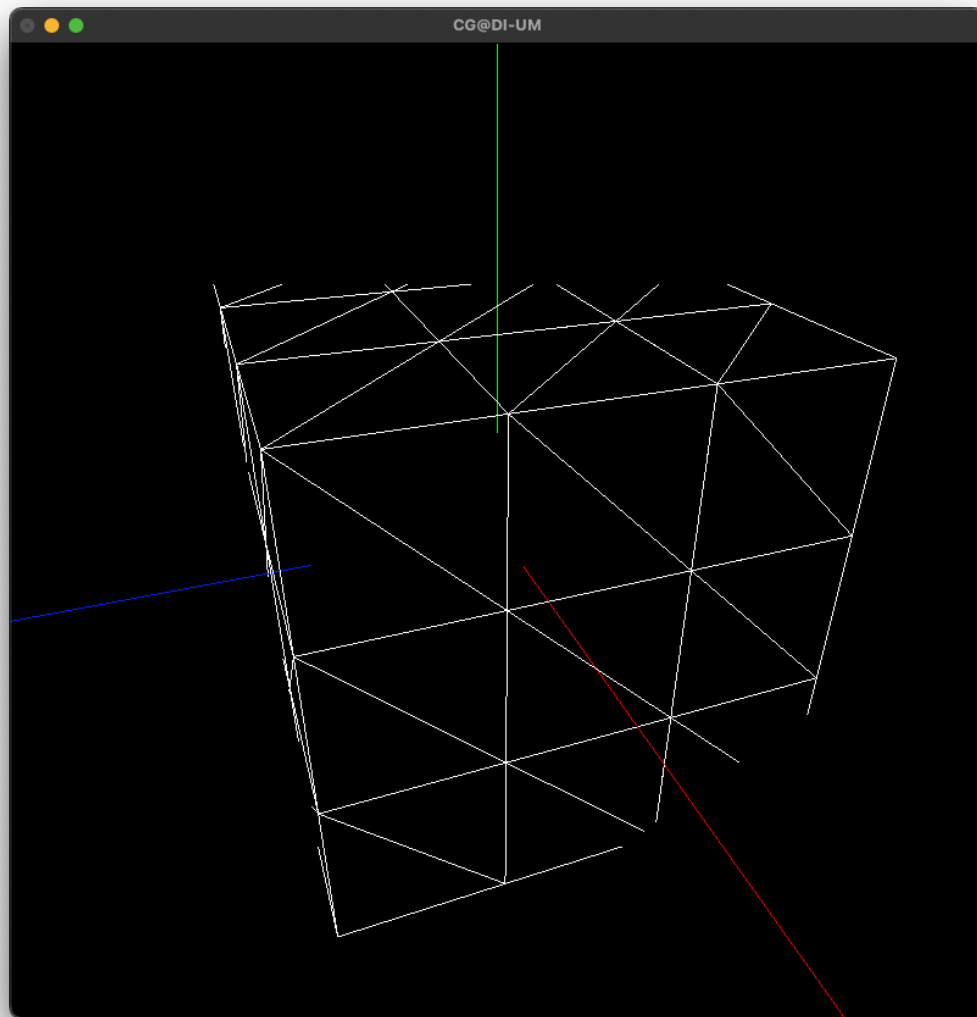


Figure 4: Cubo com 2 de lado e 3 divisões

---

### 6.3 Cone

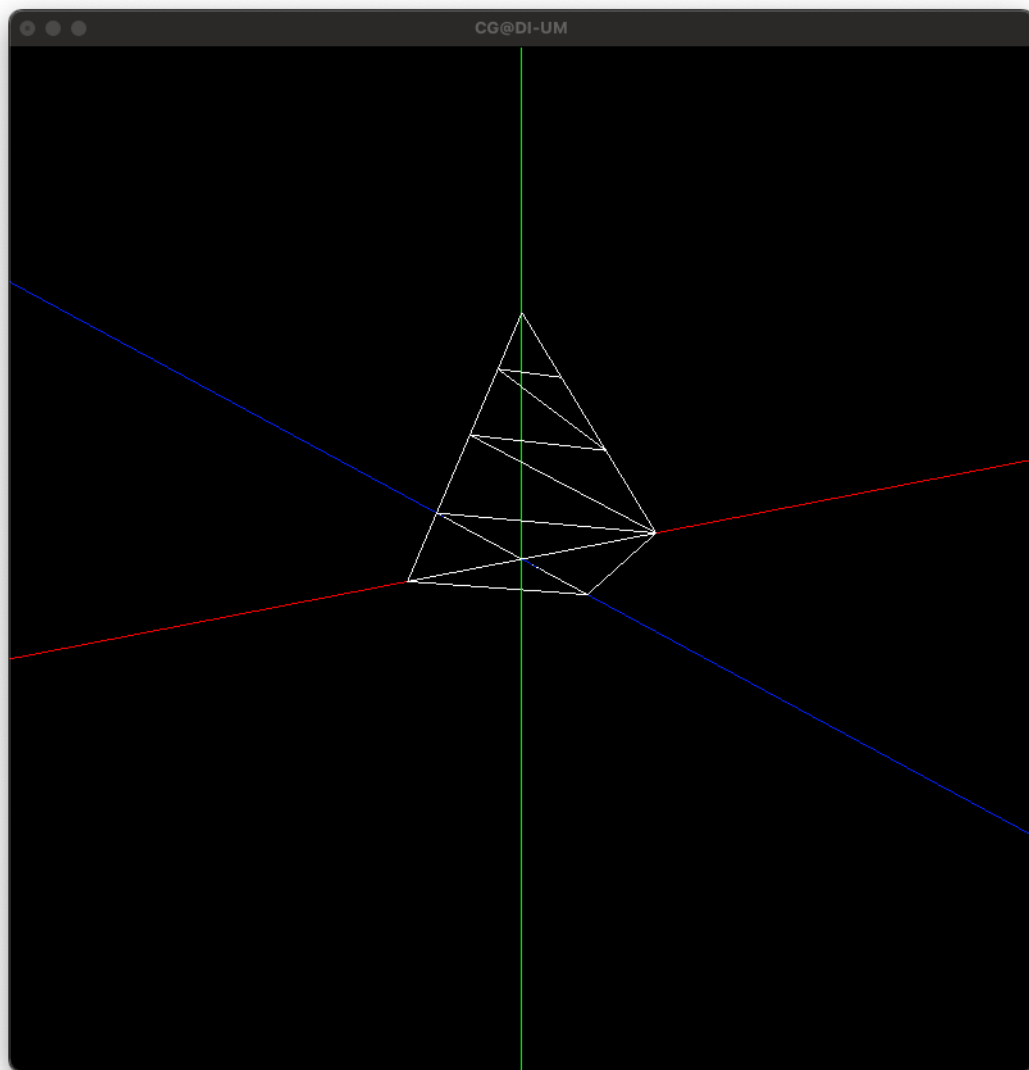


Figure 5: Cone com 1 de raio, 2 de altura, 10 fatias e 3 camadas

---

## 6.4 Esfera

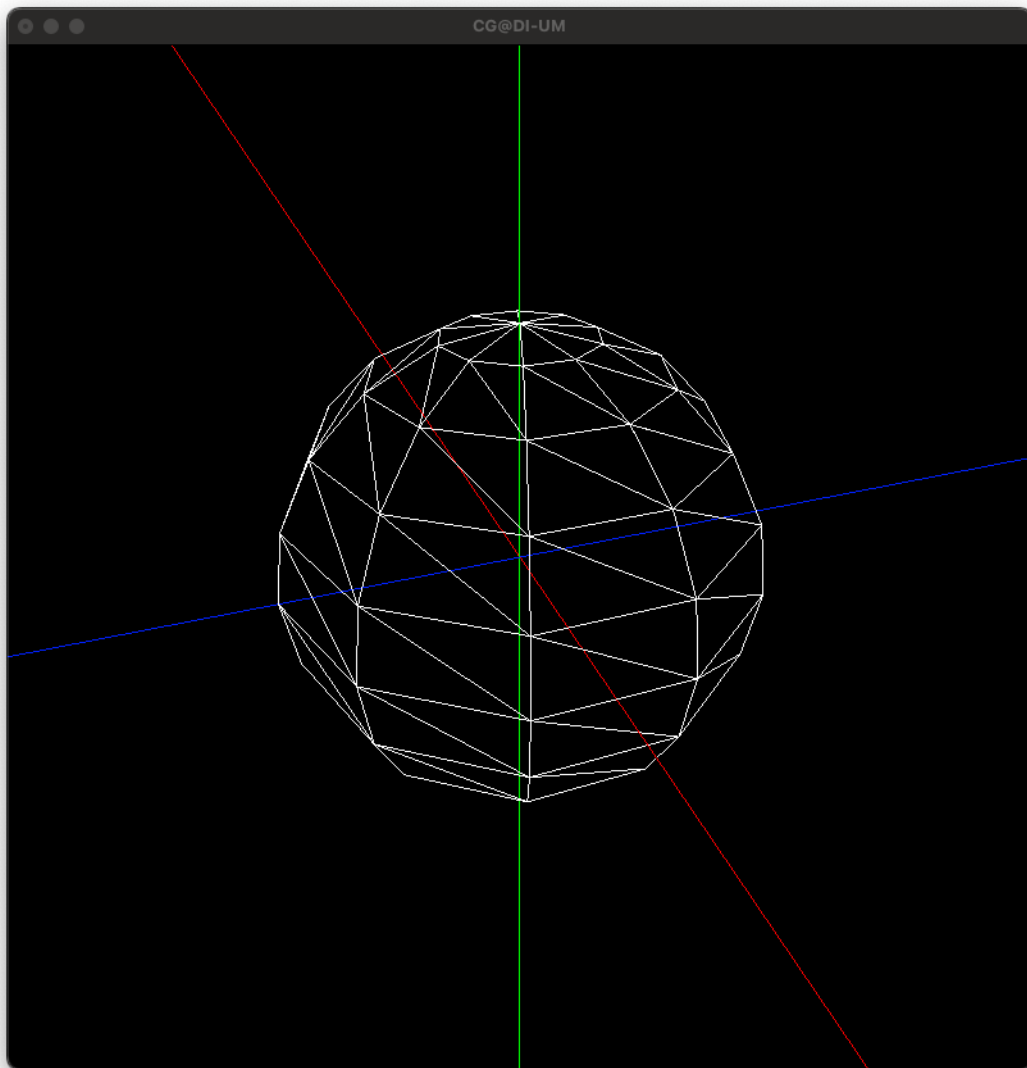


Figure 6: Esfera 1 de raio, 10 fatias e 10 camadas

---

## 6.5 Esfera e Plano

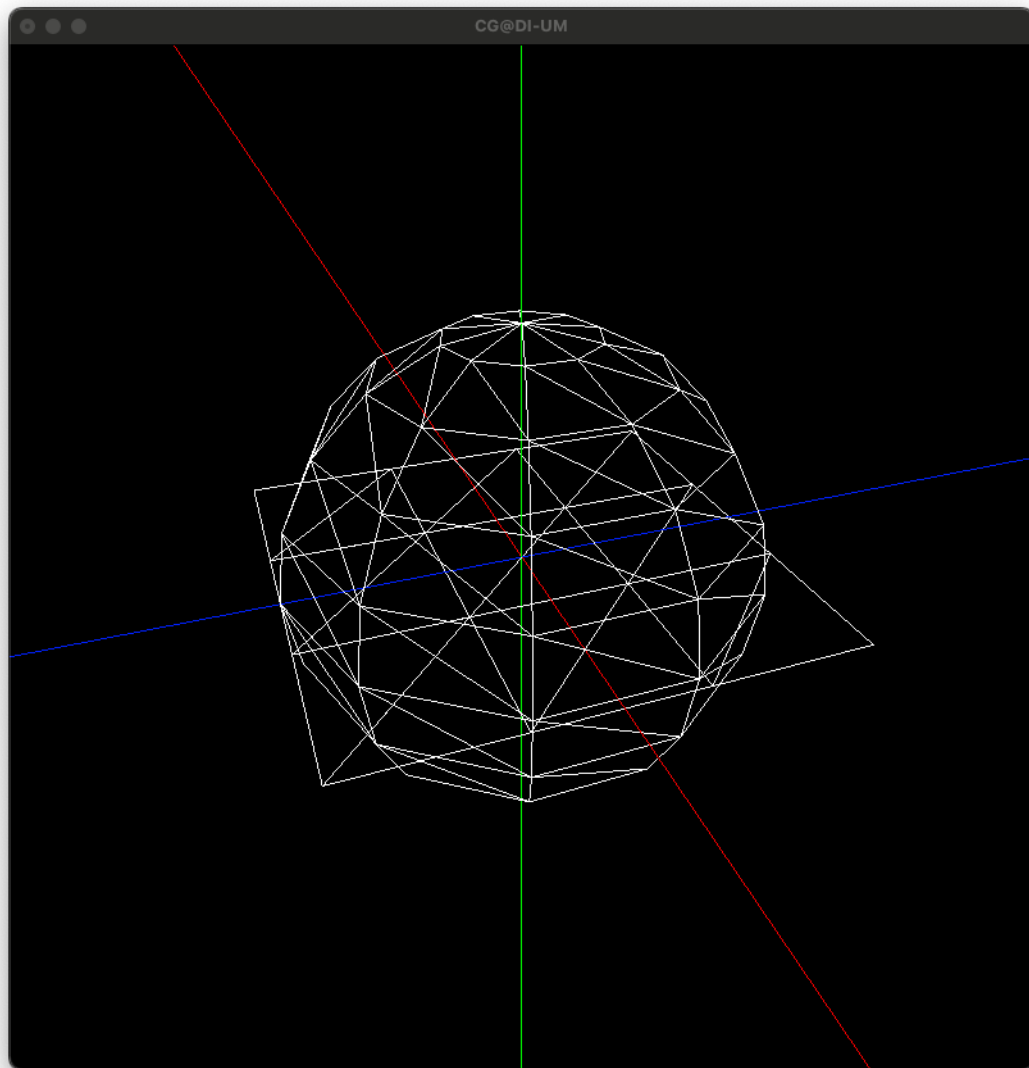


Figure 7: Esfera 1 de raio, 10 fatias e 10 camadas intersetada por um plano com 2 de lado e 3 divisões



---

## 6.6 Cilindro

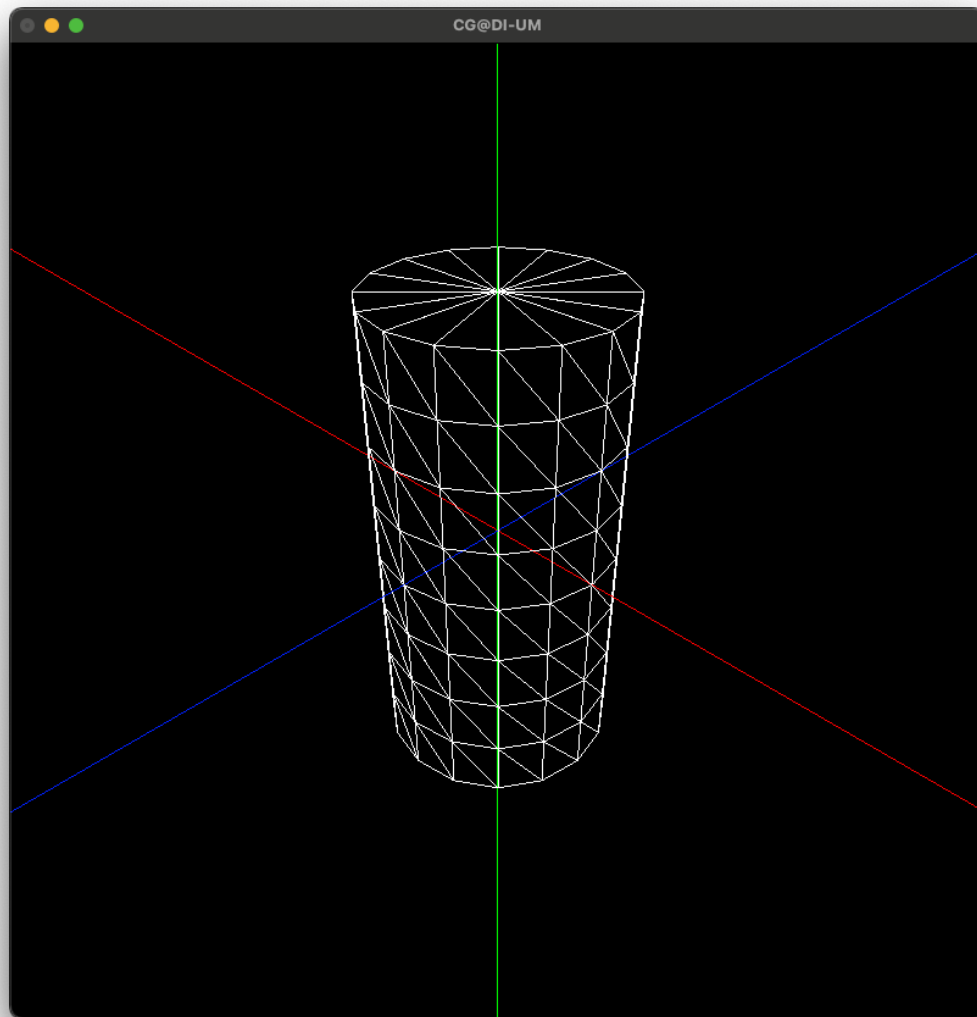


Figure 8: Cilindro com 1 de raio, 4 de altura, 16 fatias e 8 camadas

---

## 7 Conclusão

A primeira fase do projeto de computação gráfica marcou um passo inicial na compreensão e aplicação de conceitos fundamentais da área.

Consideramos que esta primeira fase foi bastante produtiva, e que nos permitiu ganhar experiência com **OpenGL** e **GLUT**, assim como um conhecimento mais geral de C++.

A compreensão dos algoritmos de geração de figuras geométricas proporcionou uma base sólida para futuros desenvolvimentos no projeto e este conhecimento inicial servirá como alicerce para explorar conceitos mais avançados de computação gráfica.