
Computação Gráfica

Fase 1



Universidade do Minho
Escola de Engenharia

TRABALHO REALIZADO POR:

AFONSO LAUREANO BARROS AMORIM
GONÇALO MARTINS DOS SANTOS
JOÃO CARLOS FERNANDES NOVAIS
TELMO JOSÉ PEREIRA MACIEL

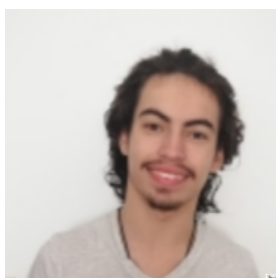
Grupo 26



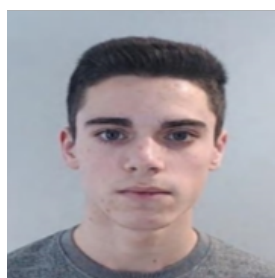
A97569
Afonso Amorim



A95354
Gonçalo Santos



A96626
João Novais



A96569
Telmo Maciel

Índice

1	Introdução	2
2	Arquitetura do projeto	3
2.1	Aplicações	3
2.1.1	<i>Generator</i>	3
2.1.2	<i>Engine</i>	3
2.2	Classes	3
2.2.1	<i>Point</i>	3
2.2.2	<i>Shape</i>	3
2.2.3	<i>Camera</i>	3
2.2.4	<i>Parser</i>	4
2.3	Ferramentas utilizadas	4
3	Primitivas geométricas	5
3.1	Plano	5
3.2	Box	6
3.3	Esfera	7
3.4	Cone	9
3.5	Cilindro	10
4	Generator	11
4.1	Descrição	11
4.2	Funcionalidades	11
4.3	Demonstração	11
5	Engine	12
5.1	Descrição	12
5.2	Funcionalidades	12
5.3	Demonstração	12
6	Apresentação dos modelos	13
6.1	Plano	13
6.2	Box	14
6.3	Esfera	15
6.4	Cone	16
6.5	Cilindro	17
7	Conclusão	18

1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi-nos pedido o desenvolvimento de um projeto de representação gráfica 3D.

O trabalho prático foi dividido em quatro fases de entrega sendo que esta primeira consiste em montar toda a arquitetura que vai servir como base para todo o nosso sistema. Deste modo, foram criadas duas aplicações: uma para gerar os ficheiros com as informações dos modelos, à qual chamamos de *Generator*, e outra que lê os ficheiros XML e representa um cenário recorrendo aos ficheiros gerados, à qual chamamos *Engine*.

Nesta primeira fase foi-nos pedida a interpretação e representação de quatro primitivas gráficas: plano, caixa, esfera, cone, sendo que decidimos adicionar uma quinta que é o cilindro.

O projeto é desenvolvido na linguagem C++ recorrendo à biblioteca OpenGL.

2 Arquitetura do projeto

Nesta secção iremos abordar a forma como o nosso projeto está estruturado, com base nas aplicações, classes e primitivas gráficas que desenvolvemos ao longo da primeira fase do trabalho prático.

2.1 Aplicações

Este projeto está dividido em duas aplicações que se complementam: gerador e motor. O gerador é responsável por fazer a tradução de primitivas gráficas para conjuntos de vértices armazenados num ficheiro. O motor vai ler o ficheiro XML contendo referências e ficheiros organizados pelo gerador, representando assim as figuras recorrendo à biblioteca OpenGL.

2.1.1 *Generator*

O módulo *generator.cpp* é responsável por converter as primitivas geométricas num conjunto de pontos, que são armazenados em ficheiros. Utilizamos as classes *Shape* e *Point* para auxiliar nesta tarefa. Os pontos que são armazenados em ficheiro são os vértices dos triângulos necessários para representar a primitiva respetiva. É de salientar que os ficheiros são guardados na pasta "files".

2.1.2 *Engine*

O módulo *engine.cpp* é responsável por ler um ficheiro XML de configuração, que contém nele informação acerca do tamanho da janela, da câmara e dos modelos a desenhar no ecrã, sendo isto tudo feito recorrendo à biblioteca OpenGL. Estes ficheiros devem estar na pasta "files", ou pelo menos deve ser explicado o caminho para eles a partir dessa pasta.

2.2 Classes

De modo a facilitar a implementação das funcionalidades anteriormente apresentadas decidimos criar algumas classes que servissem como suporte às mesmas.

2.2.1 *Point*

A classe *Point.cpp* traduz a representação de um ponto num referencial em código, sendo este representado por 3 floats (X,Y,Z).

2.2.2 *Shape*

A classe *Shape.cpp* é responsável pela criação de figuras, os pontos que representam uma figura são armazenados num vetor.

2.2.3 *Camera*

A classe *Camera.cpp* é responsável por todas as funcionalidades que podem ser implementadas numa câmara.

2.2.4 *Parser*

A classe ***Parser.cpp*** é utilizada para fazer o processamento do ficheiro de configuração XML, sendo auxiliado pela biblioteca *RapidXML* nesse aspeto. Nesta classe é guardada a informação contida no ficheiro XML, respetivamente nos atributos:

1. *camera*, do tipo *Camera**
2. *window*, do tipo *std::pair<int, int>*
3. *models*, do tipo *std::vector<*Shape>*

2.3 Ferramentas utilizadas

Para além do OpenGL onde assentam as funcionalidade gráficas do nosso projeto, recorreremos também ao **RapidXML**, uma ferramenta responsável por fazer o parsing dos ficheiros XML da forma mais rápida possível.

3 Primitivas geométricas

3.1 Plano

O plano ou **plane** foi a primeira primitiva que o nosso grupo decidiu desenvolver devido a ser a mais simples e que serviria como suporte para mais tarde implementar a **box**.

Para a implementação desta primitiva tivemos que ter em conta vários parâmetros: comprimento ou **length** e divisões ou **divisions**. O comprimento é o tamanho do lado do plano, e as divisões são o número de quadrados que cada lado do plano vai ter (se as divisões forem 3 então o plano vai ser formado por $3 \times 3 = 9$ quadrados). Para além destes parâmetros, ainda temos que ter em conta que o plano tem que estar contido no plano XZ (ou seja, o valor de Y é sempre 0) e que o plano está centrado na origem.

Com tudo isto em mente, prosseguimos para implementação da primitiva. Primeiro, reparamos que o lado de cada um dos quadrados que iria formar o plano dependia do comprimento e das divisões deste:

$$ladoQua = len/div$$

Depois, reparamos que precisávamos de ter um ponto de referência para poder começar a calcular os pontos que formam os triângulos que, no seu conjunto, vão formar o plano. Para isso reparamos que, um possível ponto de referência seria dividir o comprimento do plano por dois (já que este tem que estar centrado na origem):

$$ref = len/2$$

e assim teríamos o valor do X e Z de um dos pontos da extremidade do plano, a partir daí conseguíamos calcular todos os outros pontos.

Para calcular os restantes pontos o raciocínio passou por usar o ref para começar por uma extremidade e desenhar os triângulos usando o ladoQua (que representa o lado de um quadrado). Assim, começaríamos por gerar os primeiros dois triângulos (que formam o primeiro quadrado):

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

Agora com o primeiro quadrado formado basta calcular os outros quadrados incrementando os valores de referência usando o ladoQua.

Nota: No nosso trabalho usamos as variáveis de uma maneira um pouco diferente mas o raciocínio é o mesmo.

3.2 Box

Para desenvolver a primitiva do cubo ou da **box**, o nosso grupo baseou-se na primitiva anteriormente concebida, o plano. Decidimos seguir o mesmo raciocínio já que um cubo não é mais que um junção de 6 planos.

Para esta primitiva os parâmetros que tínhamos que ter em conta eram os mesmo que para o plano, o comprimento e as divisões para cada lado do cubo. Para além disto, agora temos que ter em atenção que o cubo está centrado na origem, ou seja, o Y não vai ser sempre 0 como era no plano.

Com tudo isto em mente, rapidamente percebemos que bastava gerar 6 planos para gerar o cubo que queríamos a única coisa que teria que mudar era o ponto de referência (ref) para cada plano.

3.3 Esfera

Para fazer a implementação da esfera, tivemos que ter em conta os parâmetros **radius** (raio da esfera), **slices** (fatias, ou seja, número de divisões verticais da esfera) e **stacks** (camadas, ou seja, o número de divisões horizontais da esfera). O raio influencia o tamanho da esfera, enquanto as camadas e as fatias influenciam o quão parecido a uma esfera perfeita ficará a que estamos a gerar.

Passando para a implementação da primitiva, tivemos que descobrir como gerar os triângulos sabendo os parâmetros da esfera. Decidimos usar coordenadas esféricas para isso. Olhando para a forma como a mesma devia ficar, notamos que apenas seria necessário unir os pontos de uma fatia aos da outra (não todos, apenas os que vamos explicar de seguida).

Considerando as seguintes variáveis:

$$\text{alphainc} = 2 * \pi / \text{slices}$$

$$\text{betainc} = \pi / \text{stacks}$$

Que representam o quanto devemos aumentar quer no α (neste caso, o ângulo contado desde o eixo Oz para o eixo Ox no sentido contrário aos ponteiros do relógio) quer no β (ângulo contado desde do eixo Ox para o eixo Oy, no sentido contrário aos ponteiros do relógio) a cada iteração, chegamos às seguintes combinações de pontos para desenhar a esfera:

- P1 : (radius * cos(beta) * cos(alpha), radius * sin(beta), radius * cos(beta) * cos(alpha))
- P2 : (radius * cos(beta) * sin(alpha + alphainc), radius * sin(beta), radius * cos(beta) * cos(alpha + alphainc))
- P3 : (radius * cos(beta + betainc) * sin(alpha), radius * sin(beta + betainc), radius * cos(beta + betainc) * cos(alpha))
- P4 : (radius * cos(beta + betainc) * sin(alpha + alphainc), radius * sin(beta + betainc), radius * cos(beta + betainc) * cos(alpha + alphainc))
- P5 : (radius * cos(beta + betainc) * sin(alpha), radius * sin(beta + betainc), radius * cos(beta + betainc) * cos(alpha))
- P6 : (radius * cos(beta) * sin(alpha + alphainc), radius * sin(beta), radius * cos(beta) * cos(alpha + alphainc))

Estes pontos são usados para desenhar uma das partes da esfera visível a rosa na seguinte figura:

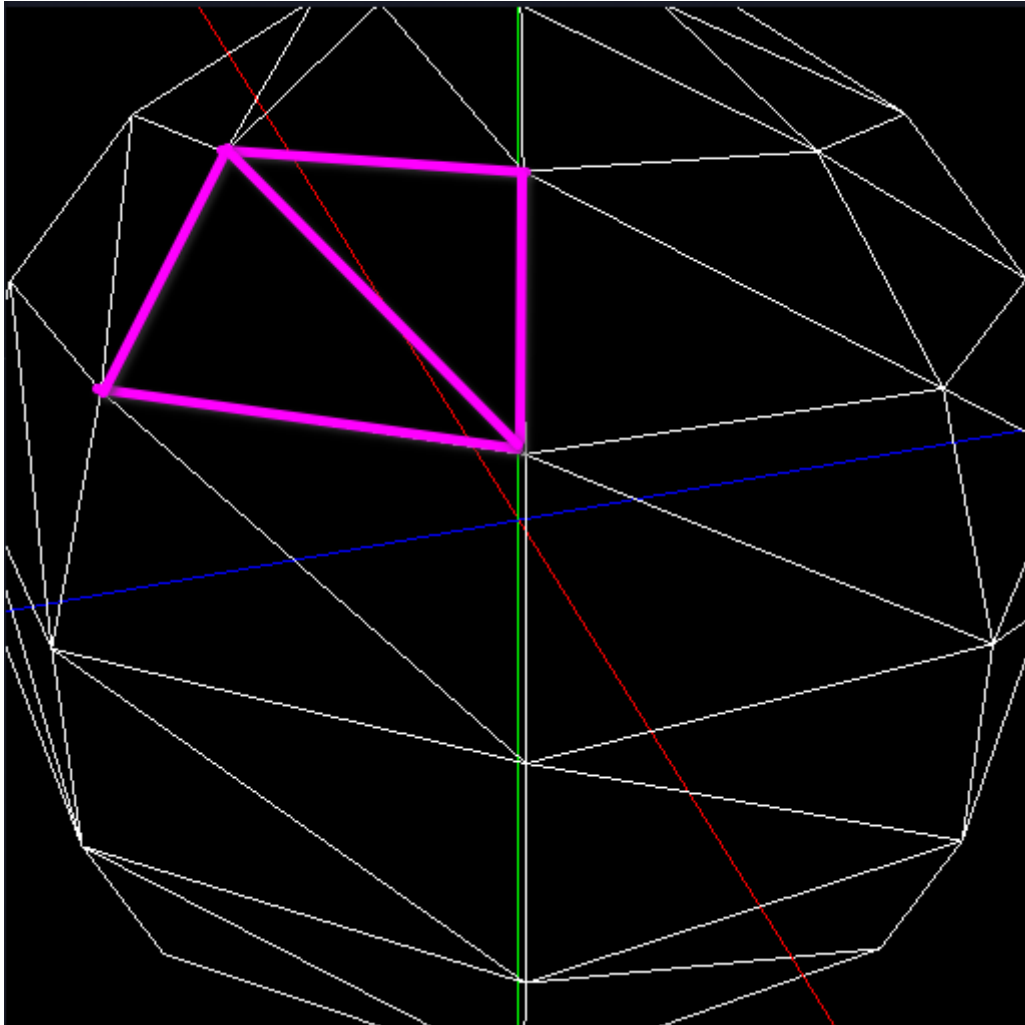


Figure 1: Parte da figura desenhada com um dos conjuntos de pontos acima referidos (destacado a rosa)

3.4 Cone

Para a implementação desta primitiva tivemos que ter em conta vários parâmetros: raio ou **radius**, altura ou **height**, fatias ou **slices** e camadas ou **stacks**. O raio é o parâmetro que especifica o raio da base do cone, a altura é o parâmetro que especifica a altura do cone. As fatias é o parâmetro que especifica o número de lados que o cone vai ter (tanto na sua base como nos seus lados). As camadas é o parâmetro que especifica o número de níveis que o cone vai ter nos seus lados. Para além destes parâmetros ainda tivemos em consideração que a base do cone tem que estar centrada no plano XZ.

Com tudo isto em mente, prosseguimos para a implementação da primitiva. Primeiro, reparamos que para gerar o cone tínhamos que recorrer a coordenadas polares e que, por isso, teríamos que calcular o ângulo para cada um dos triângulos que iam compor a base e os lados do cone. Reparamos que este ângulo é igual a 2π a dividir pelo número de fatias:

$$alfa = 2 * \pi / slices$$

Para além de termos de ter um ângulo, reparamos que teríamos de ter a altura para cada uma das camadas do cone e que a diferença de alturas entre duas camadas consecutivas do cone seria a altura do cone a dividir pelo número de camadas do mesmo:

$$alt = height / stacks$$

Por fim, ainda notamos que os raios de cada base de cada camada do cone seriam diferentes (à medida que subimos no cone os raios diminuem). Para isso teríamos que calcular a diferença de raios entre duas camadas consecutivas e concluimos que essa diferença seria igual ao raio da base do cone a dividir pelo número de camadas deste:

$$raio = radius / stacks$$

Agora, com todas estas variáveis de suporte conseguimos gerar todos os pontos do cone. A estratégia que seguimos para construir o cone foi gerá-lo por fatias construindo uma fatia de cada vez, ou seja, primeiro todas as camadas de uma determinada fatia e só depois continuar com a próxima fatia.

Sendo assim, para cada fatia começamos por gerar os pontos da base. Um deles sabemos sempre que é o ponto da origem (já que a base do cone está contida no plano XZ) os outros dois foram calculados recorrendo a coordenadas polares, ao raio da base (passado como argumento) e ao alfa (deduzido anteriormente). Depois, geramos todos os pontos de todos os triângulos de cada camada dos lados do cone para esta fatia, para isso recorremos ao raio, ao alt e ao alfa (todos deduzidos anteriormente).

Repetindo este processo para todas as fatias, temos o nosso cone gerado.

3.5 Cilindro

No que toca ao cilindro, à semelhança do cone, este terá que ter em conta os seguintes parâmetros: raio ou **radius**, altura ou **height**, fatias ou **slices**, apenas não contendo as camadas.

Para ser desenhado usamos coordenadas polares, gerando o cilindro fatia a fatia, começando pelo triângulo da base de cima, definido por estes 3 pontos (asumindo que teremos $step = 360.0/sides$):

- P1 : (0, height*0.5, 0)
- P2 : ($\cos(i * step * \pi / 180.0) * radius$, height*0.5, $-\sin(i * step * \pi / 180.0) * radius$)
- P3 : ($\cos((i+1) * step * \pi / 180.0) * radius$, height*0.5, $-\sin((i + 1) * step * \pi / 180.0) * radius$)

De seguida o triângulo da base de baixo:

- P4 : (0, -height*0.5, 0)
- P5 : ($\cos((i+1) * step * \pi / 180.0) * radius$, -height*0.5, $-\sin((i + 1) * step * \pi / 180.0) * radius$)
- P6 : ($\cos(i * step * \pi / 180.0) * radius$, -height*0.5, $-\sin(i * step * \pi / 180.0) * radius$)

E finalmente os dois triângulos da face lateral:

- P7 : ($\cos(i * step * \pi / 180.0) * radius$, height*0.5, $-\sin(i * step * \pi / 180.0) * radius$)
- P8 : ($\cos(i * step * \pi / 180.0) * radius$, -height*0.5, $-\sin(i * step * \pi / 180.0) * radius$)
- P9 : ($\cos((i+1) * step * \pi / 180.0) * radius$, height*0.5, $-\sin((i + 1) * step * \pi / 180.0) * radius$)
- P10 : ($\cos(i * step * \pi / 180.0) * radius$, -height*0.5, $-\sin(i * step * \pi / 180.0) * radius$)
- P11 : ($\cos((i+1) * step * \pi / 180.0) * radius$, -height*0.5, $-\sin((i + 1) * step * \pi / 180.0) * radius$)
- P12 : ($\cos((i+1) * step * \pi / 180.0) * radius$, height*0.5, $-\sin((i + 1) * step * \pi / 180.0) * radius$)

4 Generator

4.1 Descrição

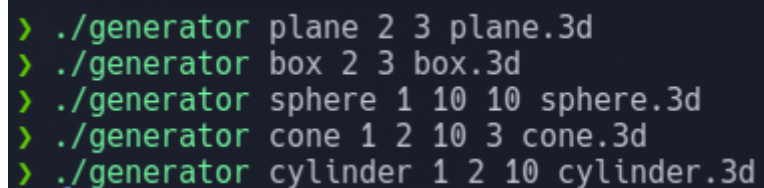
Para a representação das figuras geométricas apresentadas no projeto foram utilizados apenas triângulos. O *Generator* é responsável por efetuar todos os cálculos necessários para a concepção da forma pretendida, isto é, calcular os vértices de todos os triângulos que integram a constituição do nosso objeto.

4.2 Funcionalidades

Foram desenvolvidas várias formas geométricas. Estas podem começar a ser construídas através do *Generator*.

- **Plano** - plane tamanhoLado noDivisoes nomeFicheiro.3d
- **Box** - box tamanhoLado noDivisoes nomeFicheiro.3d
- **Esfera** - sphere raio noFatias noCamadas nomeFicheiro.3d
- **Cone** - cone raio altura noFatias noCamadas nomeFicheiro.3d
- **Cilindro** - cylinder raio altura noFatias nomeFicheiro.3d

4.3 Demonstração



```
> ./generator plane 2 3 plane.3d
> ./generator box 2 3 box.3d
> ./generator sphere 1 10 10 sphere.3d
> ./generator cone 1 2 10 3 cone.3d
> ./generator cylinder 1 2 10 cylinder.3d
```

Figure 2: Demonstração de invocação do generator para criar cada uma das primitivas

5 Engine

5.1 Descrição

Após o cálculo de todos os vértices por parte do *Generator*, o programa consegue iniciar a fase de representação 3D do objeto pretendido. O *Engine* efetua a leitura do ficheiro XML que lhe é apresentado como argumento (fornecendo o caminho para ele a partir da pasta "files") e, desta forma, recolhe a lista de ficheiros, analisa os mesmos e passa, assim, para a representação da sua forma no ecrã.

5.2 Funcionalidades

Como funcionalidades do Engine decidimos fazer com que fosse possível mover a câmara, usando coordenadas esféricas, estando estas centradas na origem, ou seja, no ponto (0,0,0), começando sempre na posição especificada no ficheiro XML. A movimentação é feita utilizando as setas do teclado, *PageUp* para diminuir o *zoom* e *PageDown* para aumentar o *zoom*.

5.3 Demonstração

```
> ./generator cone 1 2 10 3 cone.3d
> ./generator cylinder 1 2 10 cylinder.3d
> ./engine exemplo.xml
```

Figure 3: Invocação do Generator e do Engine por terminal

```
<world>
  <window width="512" height="512" />
  <camera>
    <position x="5" y="2" z="5" />
    <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="cylinder.3d"/>
      <model file="cone.3d"/>
    </models>
  </group>
</world>
```

Figure 4: Ficheiro XML lido pelo Engine

6 Apresentação dos modelos

6.1 Plano

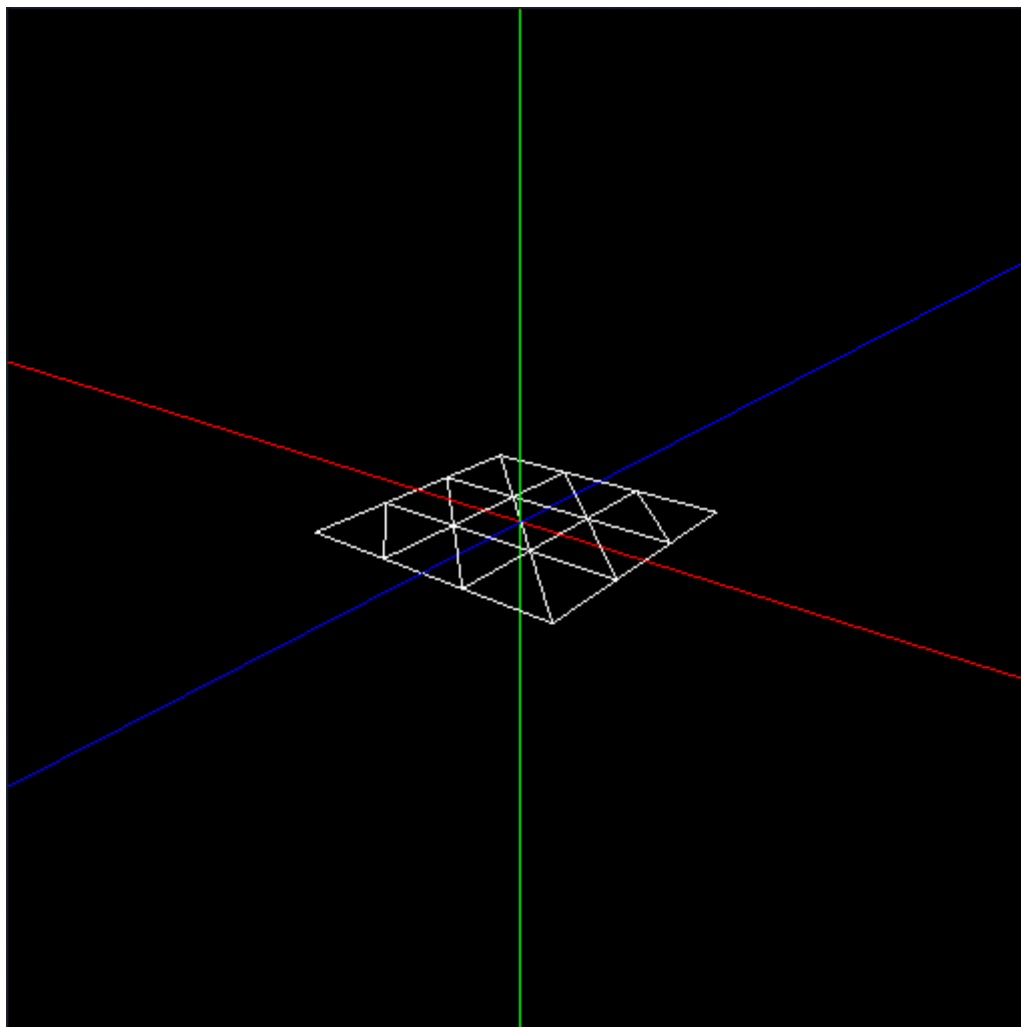


Figure 5: Plano com 2 de lado e 3 divisões

6.2 Box

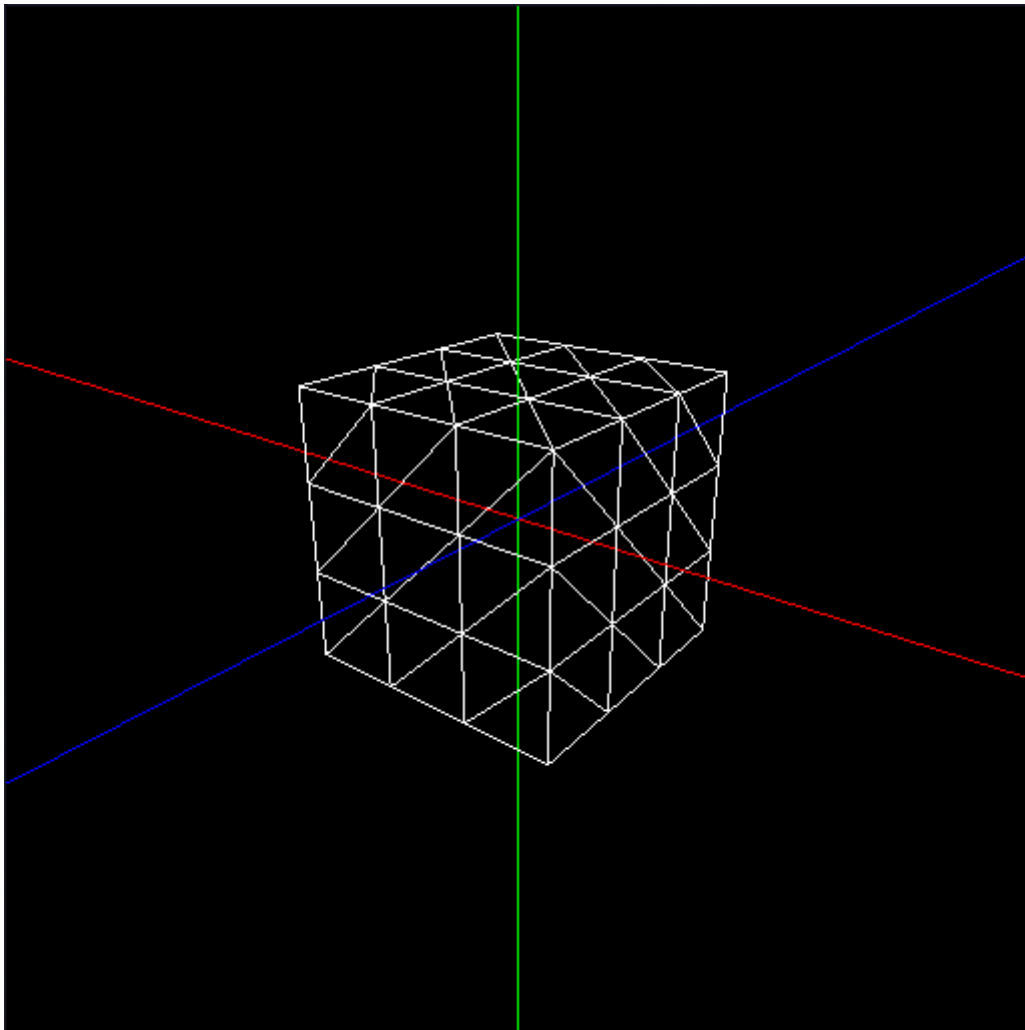


Figure 6: Box gerada com 2 de lado e 3 divisões

6.3 Esfera

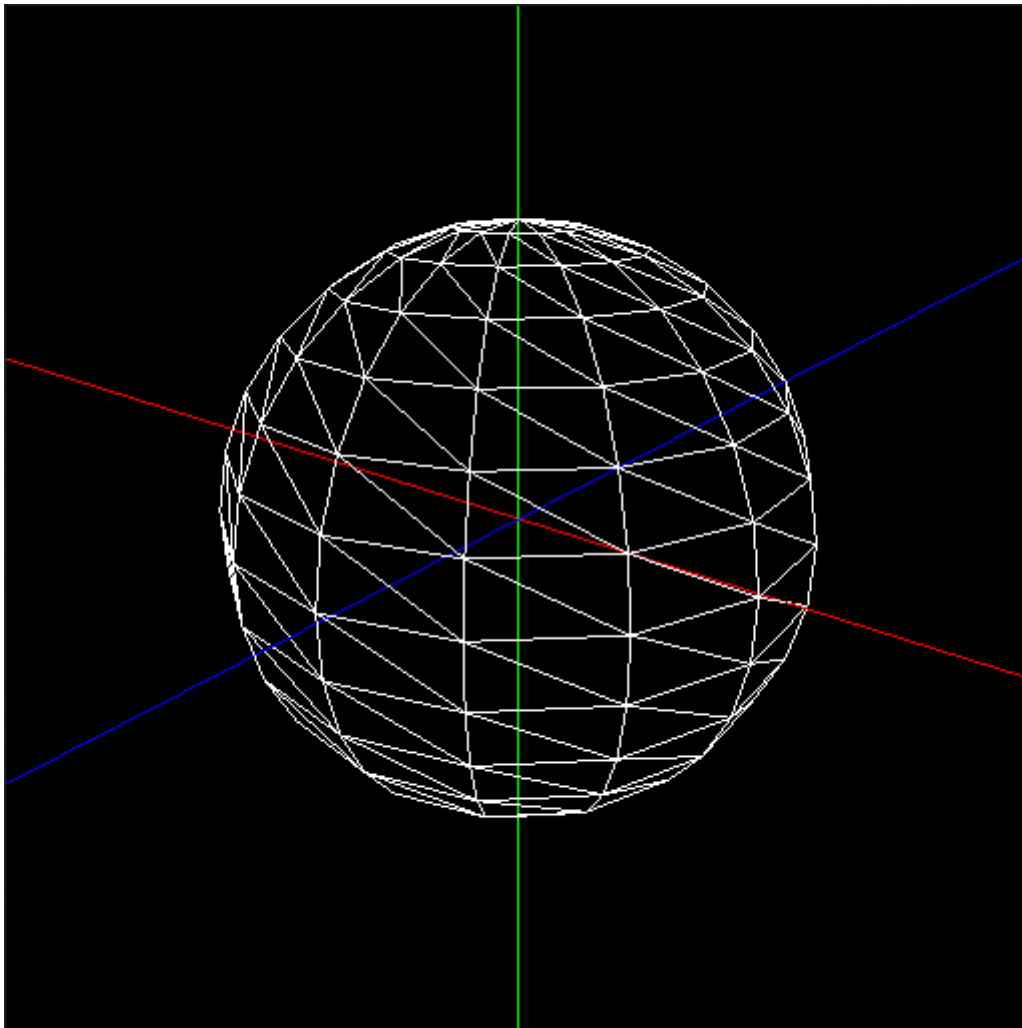


Figure 7: Esfera gerada com 1 de raio, 10 fatias e 10 camadas

6.4 Cone

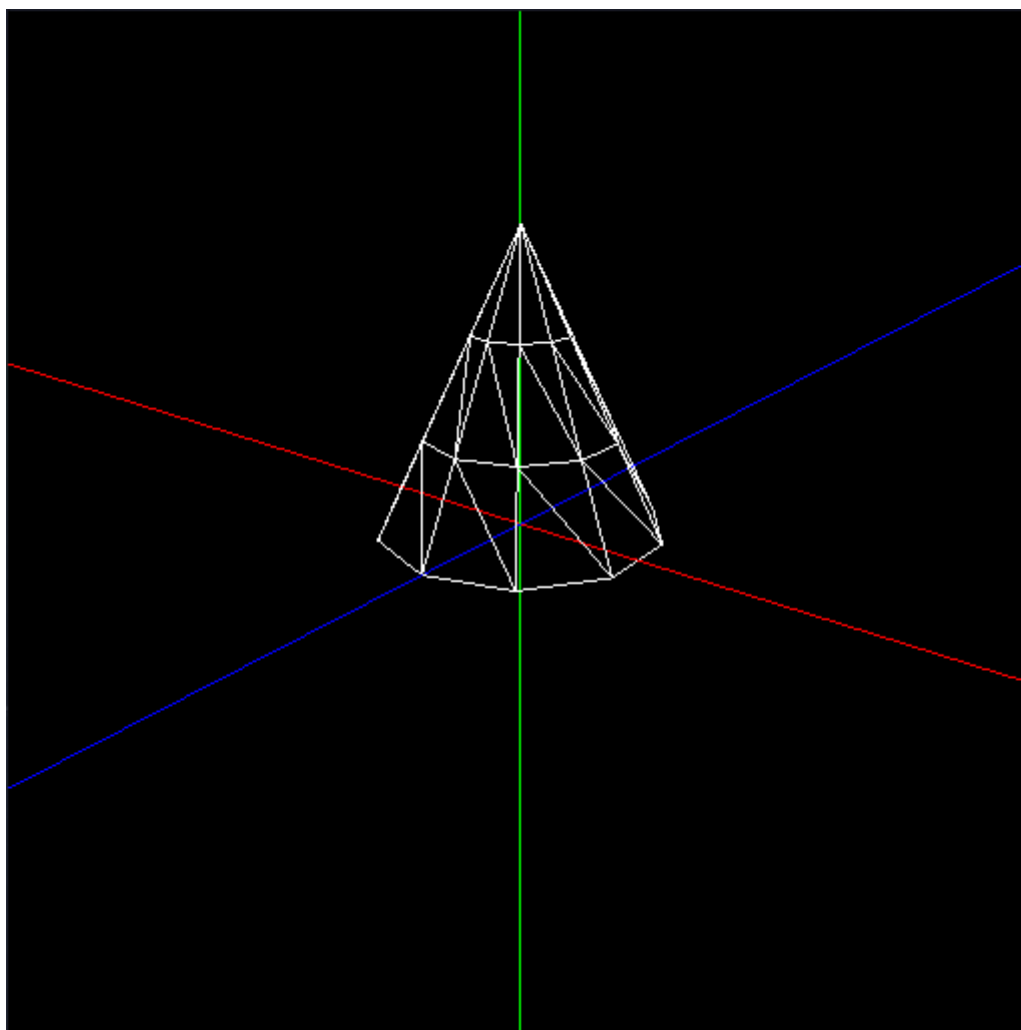


Figure 8: Cone gerado com 1 de raio, 2 de altura, 10 fatias e 3 camadas

6.5 Cilindro

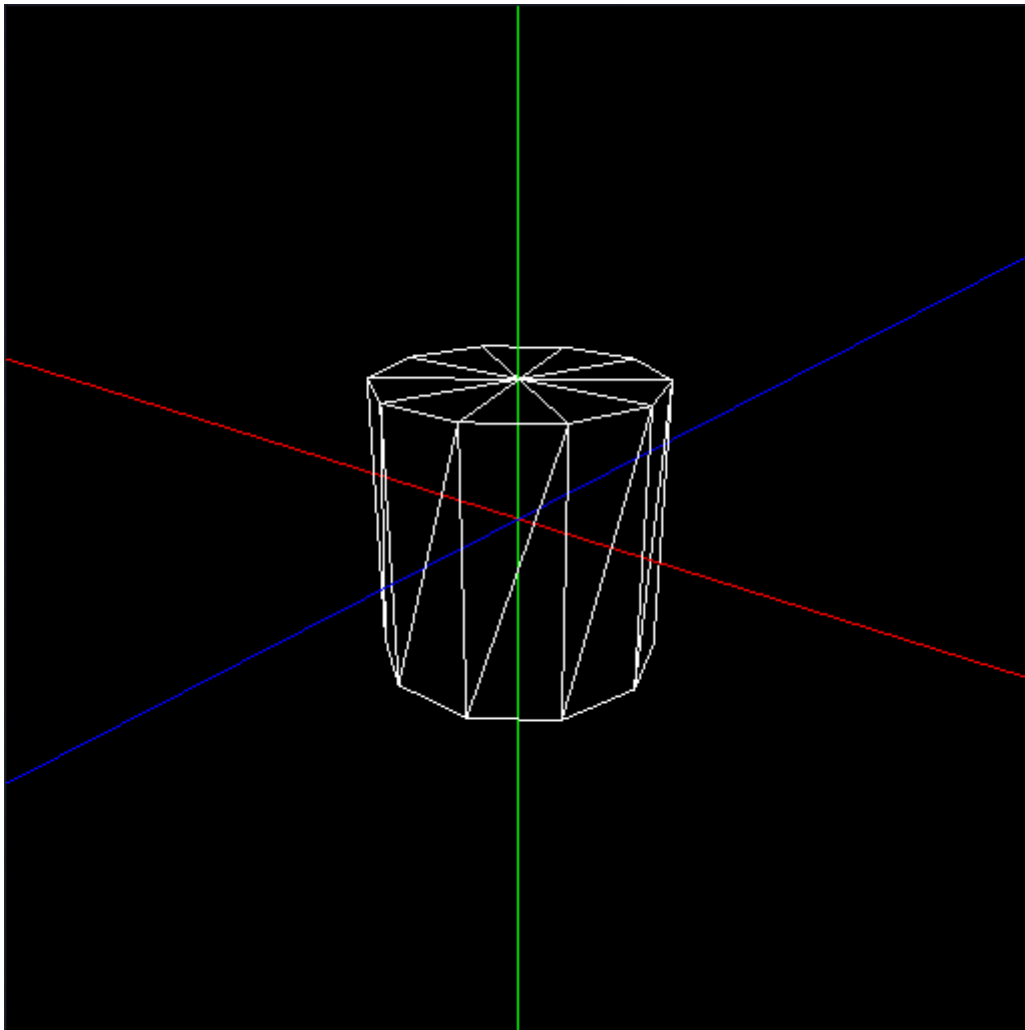


Figure 9: Cilindro gerado com 1 de raio, 2 de altura e 10 fatias

7 Conclusão

Com a conclusão da primeira fase do trabalho prático de Computação Gráfica reconhecemos que esta fase deu-nos a oportunidade de aprofundar conhecimentos em *GLUT* e *OPENGL*.

Esta fase foi ainda a primeira exposição à linguagem C++, ou seja, esta fase foi uma oportunidade de aprender e utilizar uma nova linguagem de programação.

Concluindo, esperamos que esta primeira fase do nosso trabalho sirva como uma sólida base para o desenvolvimento do resto do trabalho.