



Universidade do Minho
Escola de Engenharia

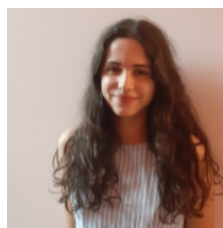
Computação Gráfica

Trabalho Prático - Fase 1

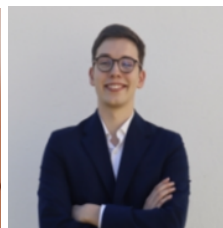
Grupo 02



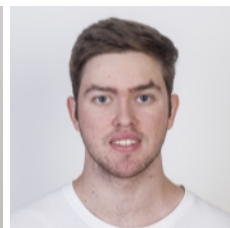
André Campos
a104618



Beatriz Peixoto
a104170



Diogo Neto
a98197



Luís Freitas
a104000

Índice

1. Introdução.....	2
2. Ficheiros .3d.....	2
3. Leitura de XML.....	3
4. Gerador.....	3
4.1. Como funciona.....	4
4.2. Como produzir as figuras.....	4
4.2.1. Plano.....	4
Plano 1.....	6
Plano 2.....	6
Plano 3.....	7
4.2.2. Esfera.....	7
Esfera 1.....	8
Esfera 2.....	9
Esfera 3.....	9
4.2.3. Caixa.....	10
Caixa 1.....	11
Caixa 2.....	11
Caixa 3.....	12
4.2.4. Cone.....	13
Cone 1.....	16
Cone 2.....	17
Cone 3.....	18
5. Engine.....	19
6. Como usar.....	20
6.1. CMake - Testado apenas em Linux/macOS.....	20
6.2. Geração de figuras.....	21
6.3. Desenhar o Cenário.....	21
6.3.1. Ficheiro test_1_1.xml.....	21
6.3.2. Ficheiro test_1_2.xml.....	22
6.3.3. Ficheiro test_1_3.xml.....	23
6.3.4. Ficheiro test_1_4.xml.....	24
6.3.5. Ficheiro test_1_5.xml.....	25
6.3.6. Ficheiro demo1.xml.....	26
6.3.7. Ficheiro demo2.xml.....	27
7. Conclusão.....	28

1. Introdução

O objetivo deste trabalho é desenvolver uma engine num mini cenário em 3D e usar exemplos que mostrem o seu potencial.

Nesta fase desenvolvemos duas aplicações:

- **Generator** : aplicação que gera ficheiros .3d que contêm modelos, indicados por nós, juntamente com os seus parâmetros, também indicados por nós
- **Engine** : aplicação que lê um ficheiro XML com as configurações que serão usadas, utiliza ficheiros .3d indicados pelo mesmo e apresentará os modelos indicados por esse ficheiro no ecrã, usando **OpenGL**

2. Ficheiros .3d

Os ficheiros .3d serão ficheiros binários onde:

- 1º número é a quantidade de vértices que este ficheiro .3d contém
- Os próximos números estão agrupados em 3, onde cada grupo corresponde aos vértices de um triângulo

Exemplo de um ficheiro .3d:

6
0 0 0
1 0 0
0 0 1
1 0 0
1 0 1
0 0 1

Neste exemplo, o ficheiro .3d indica que:

- O modelo tem 6 vértices
- Tem um triângulo cujos vértices são : (0,0,0) (1,0,0) (0,0,1)
- Tem outro triângulo cujos vértices são : (1,0,0) (1,0,1) (0,0,1)

O **gerador** tem uma opção chamada **text** que dado um ficheiro binário .3d, cria um ficheiro texto com as informações dentro do binário. Isto é útil, caso seja necessário descobrir o que tem exatamente nesse ficheiro .3d de uma forma legível.

```
andre@msi:~/Desktop/CG-2425/build$ ./generator plane 1 1 plano.3d
Ficheiro plano.3d foi criado!
andre@msi:~/Desktop/CG-2425/build$ ./generator text plano.3d plano_texto.txt
Ficheiro plano_texto.txt foi criado!
```

Figura 1. Criação de um ficheiro .3d e a sua conversão para ficheiro texto

```

1 2
2
3 0.5 0 0.5
4 0.5 0 -0.5
5 -0.5 0 -0.5
6
7 0.5 0 0.5
8 -0.5 0 -0.5
9 -0.5 0 0.5
10

```

Figura 2. Ficheiro texto resultante

3. Leitura de XML

Para a leitura de ficheiros XML, usamos uma biblioteca externa chamada **TinyXML**, que fornece métodos para a leitura de ficheiros em formato XML.

O TinyXML está inserido dentro do projeto e esta biblioteca é uma cópia (*mirror*) do projeto <http://www.grinninglizard.com/tinyxml/>. O github onde fomos buscar a biblioteca TinyXML é <https://github.com/lucasg/tinyxml/tree/master/>.

A leitura do XML acontece quando a **engine** é executada. Durante essa leitura, a aplicação cria um ficheiro de texto chamado **xml_logger.txt** que relata o que aconteceu durante a leitura do ficheiro XML e quais os valores que extraiu:

```

1 [JANELA::width] Definido! (512)
2 [JANELA::height] Definido! (512)
3 [CAMERA::position::x] Definido! (3.000000)
4 [CAMERA::position::y] Definido! (2.000000)
5 [CAMERA::position::z] Definido! (1.000000)
6 [CAMERA::lookAt::x] Definido! (0.000000)
7 [CAMERA::lookAt::y] Definido! (0.000000)
8 [CAMERA::lookAt::z] Definido! (0.000000)
9 [CAMERA::up::x] Definido! (0.000000)
10 [CAMERA::up::y] Definido! (1.000000)
11 [CAMERA::up::z] Definido! (0.000000)
12 [CAMERA::projection::fov] Definido! (60.000000)
13 [CAMERA::projection::near] Definido! (1.000000)
14 [CAMERA::projection::far] Definido! (1000.000000)
15 [GRUPO::models::model::file] Adicionado modelo plane_2_3.3d!
16 [GRUPO::models::model::file] Adicionado modelo sphere_1_10_10.3d!

```

Figura 3. Ficheiro xml_logger.txt após a leitura do ficheiro XML test_1_5.xml

4. Gerador

O **gerador** é uma aplicação que gera ficheiros .3d que indicam à **engine** como deve desenhar os modelos descritos nesses ficheiros.

4.1. Como funciona

O **gerador** recebe vários argumentos, que podem variar dependente do 1º argumento que ele recebe.

O primeiro argumento pode ser:

- **sphere** : para gerar uma esfera
- **box** : para gerar uma caixa
- **cone** : para gerar um cone
- **plane** : para gerar um plano
- **text** : para converter ficheiros .3d para ficheiro texto (como mencionado acima)

Se o modelo a gerar é:

- **Esfera** : ele deve receber como argumentos o seu raio, quantas divisões verticalmente (slices) e quantas divisões horizontalmente (stacks)
- **Caixa** : ele deve receber como argumentos o tamanho da caixa e a quantidade de divisões em cada lado
- **Cone** : ele deve receber como argumentos o seu raio, a sua altura, a quantidade de divisões verticais (slices) e a quantidade de divisões horizontais (stacks)
- **Plano** : ele deve receber como argumentos o seu comprimento e a quantidade de divisões

O último argumento é o nome do ficheiro .3d que queremos produzir.

4.2. Como produzir as figuras

Para a produção de figuras, usamos as classes **Ponto** e **Triangle**:

- **Ponto** : contém 3 números que correspondem às suas coordenadas num gráfico 3D. Os números são representados em **doubles** para ter uma ótima precisão
- **Triangle** : contém 3 Pontos que correspondem aos seus vértices num gráfico 3D

No **Gerador**, usamos algoritmos para conseguirmos obter todos os triângulos que serão desenhados pela engine, e após obtê-los, guardamos todos os triângulos num vector de triângulos, onde será usado para produzir o ficheiro .3d

4.2.1. Plano

O gerador cria um plano centrado na origem, que corresponde a um quadrado no plano XZ subdividido igualmente nos eixos X e Z. Deste modo, a função que constrói este plano é a função “geraPlano”. Esta função recebe como argumentos o comprimento do lado do plano (length) e o número de divisões deste plano (divisions).

Neste sentido, para gerar o plano, começamos por calcular as seguintes medidas:

- $\text{inicio} = \text{length} / 2$
- $\text{salto} = \text{length} / \text{divisions}$

O “início” corresponde a metade da medida do lado do plano e calculamo-lo para garantir que o plano fica corretamente subdividido nos eixos X e Z . O “salto” corresponde ao tamanho de cada subdivisão do plano (subdivisões estas criadas pelas “divisions” do plano).

Para determinar as coordenadas foram utilizados dois loops.

O loop externo (i) vai percorrer cada coluna ao longo do eixo X e o loop interno (j) vai percorrer cada linha ao longo do eixo Z.

Para cada iteração do loop externo, isto é, para cada coluna, o loop interno vai iterar por todas as linhas. Em cada iteração do loop interno, o “salto” permite avançar para a posição correta de modo a garantir que as coordenadas são calculadas de forma uniforme e corretamente subdividida. Para cada iteração do loop interno, são criadas as coordenadas dos dois triângulos que constituem um pequeno quadrado, que é uma subdivisão do plano.

Neste sentido, no seguimento do que foi dito acima, para cada iteração do loop interno, criamos quatro pontos necessários para os triângulos.

A imagem abaixo fundamenta e explica como é que criamos e orientamos os vértices dos triângulos. Deste modo, para cada subdivisão do plano, o ponto de partida para desenharmos os triângulos é o ponto P3, uma vez que corresponde ao ponto onde não há incrementos de “salto”, pelo que é a partir desse ponto que determinamos as coordenadas dos restantes pontos. Assim, as restantes coordenadas são determinadas da seguinte forma:

- Ponto P2: incrementa-se o “salto” à coordenada x.
- Ponto P4: incrementa-se o “salto” à coordenada z.
- Ponto P1: incrementa-se o “salto” à coordenada x e à coordenada z.

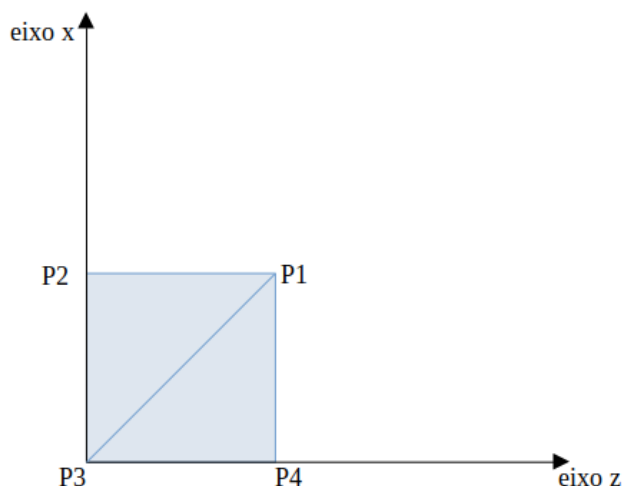


Figura 4. Representação de uma subdivisão do plano e os seus pontos

A forma como organizamos os pontos de modo a garantir a orientação correta dos triângulos foi a seguinte: no primeiro triângulo a ordem de pontos foi P1, P2 e P3, e no segundo triângulo foi P1, P3 e P4. Além disso, o método que seguimos para descobrir qual a ordem correta, foi utilizar a regra da mão direita

Por fim, nas imagens abaixo estão representados alguns resultados do plano gerado com a nossa função “geraPlano”. Os argumentos da função variam de imagem para imagem.

Plano 1

Length = 2

Divisions = 2

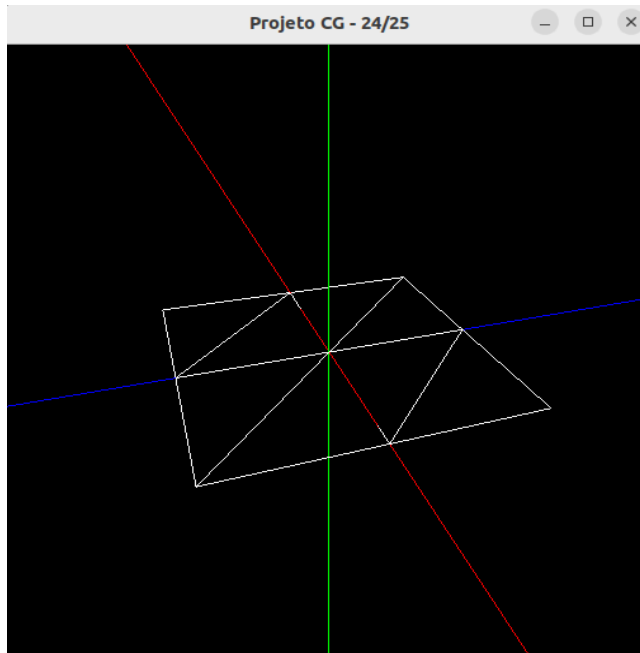


Figura 5. Plano exemplo 1

Plano 2

Length = 3

Divisions = 3

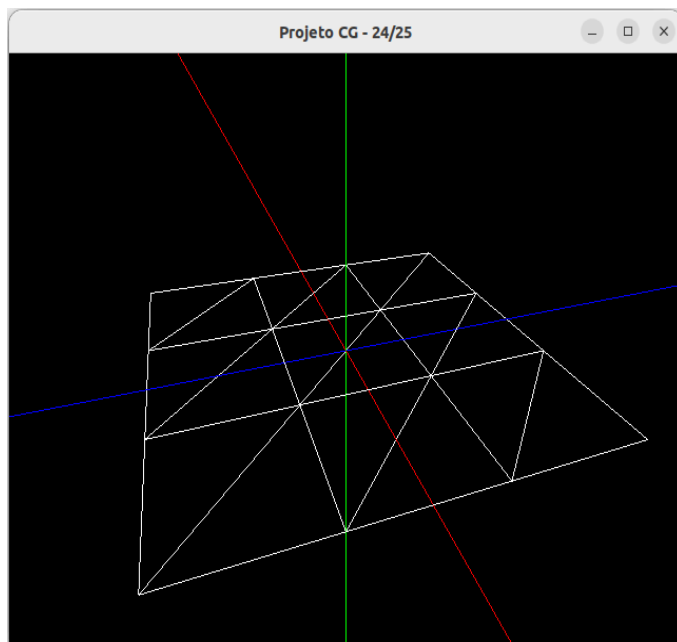


Figura 6. Plano exemplo 2

Plano 3

Length = 8

Divisions = 20

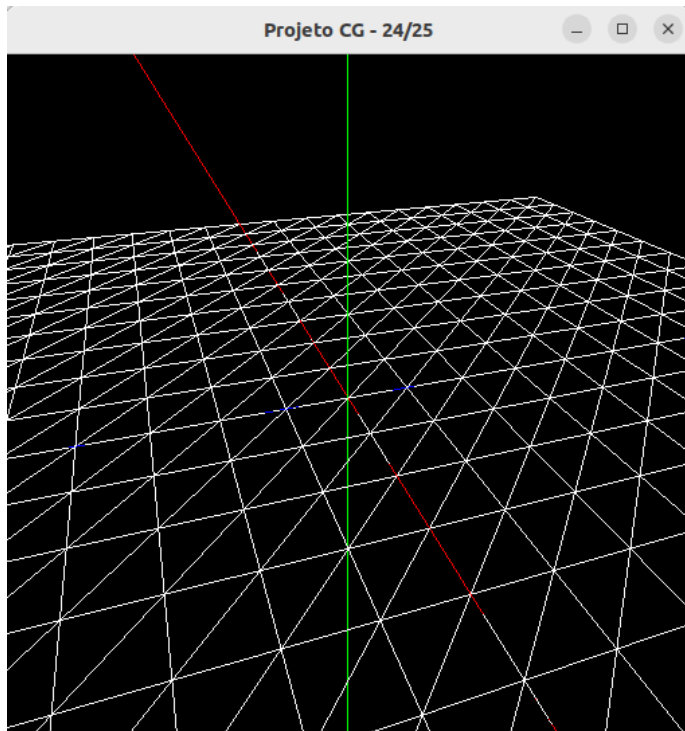


Figura 7. Plano exemplo 3

4.2.2. Esfera

A esfera é gerada dividindo-a numa série de "fatias" e "camadas" que formam os triângulos que compõem a sua superfície:

- **Stacks (Camadas):** Controlam o número de "faixas" horizontais ao longo da esfera, representando as divisões ao longo do eixo vertical (de cima para baixo).
- **Slices (Fatias):** Controlam o número de "segmentos" verticais ao redor da esfera, representando as divisões ao longo do eixo horizontal.

A função "geraEsfera" recebe três parâmetros: raio (radius), slices e stacks, e usa dois loops aninhados para percorrer cada camada e cada fatia da esfera:

1. **Loop Externo (i de 0 a número de stacks):**
 - Calcula dois ângulos α_1 e α_2 , que correspondem às latitudes das camadas superior e inferior de cada faixa. Isso é feito em relação ao número total de camadas (stacks).
2. **Loop Interno (j de 0 a número de slices):**
 - Calcula dois ângulos β_1 e β_2 , que são as longitudes das fatias dentro de cada camada.
 - Dentro deste loop, são calculadas as coordenadas de 4 pontos (que formam 2 triângulos) usando as fórmulas esféricas:

Com as coordenadas dos 4 pontos (p1, p2, p3, p4) calculadas, são criados dois triângulos com essas coordenadas:

- O primeiro triângulo é formado por p2, p1 e p3.
- O segundo triângulo é formado por p2, p3 e p4.

A ordem de chamada dos pontos na formação dos triângulos é crucial, dado que seguimos a **regra da mão direita** para definir a orientação correta. Uma vez criados os triângulos, estes são adicionados à lista de triângulos que representará a malha da esfera que, no final, a função “geraEsfera” vai retornar.

Por fim, nas imagens abaixo estão representados alguns resultados da esfera gerada com a nossa função “geraEsfera”. Os argumentos da função variam de imagem para imagem.

Esfera 1

Radius = 1

Slices = 10

Stacks = 10

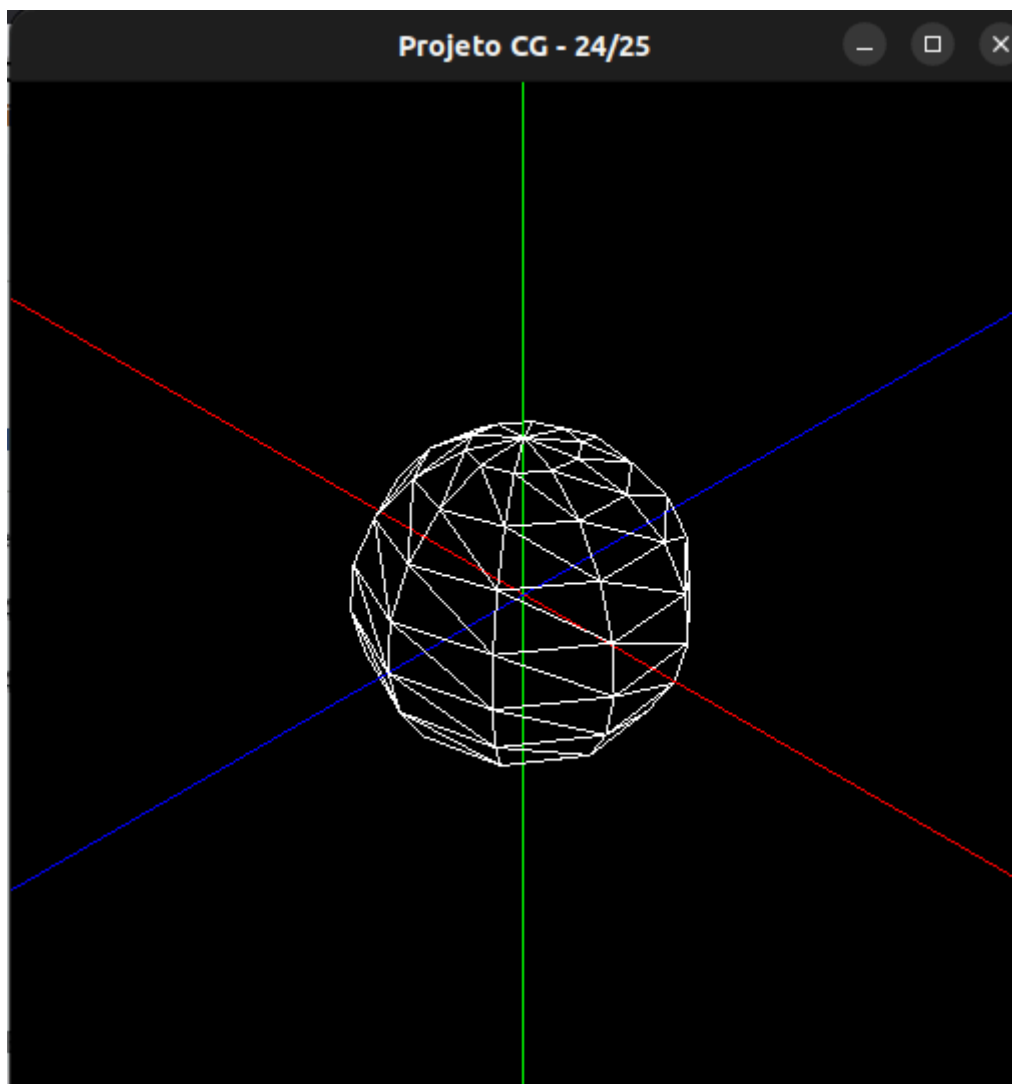


Figura 8. Esfera exemplo 1

Esfera 2

Radius=2

Slices=7

Stacks=7

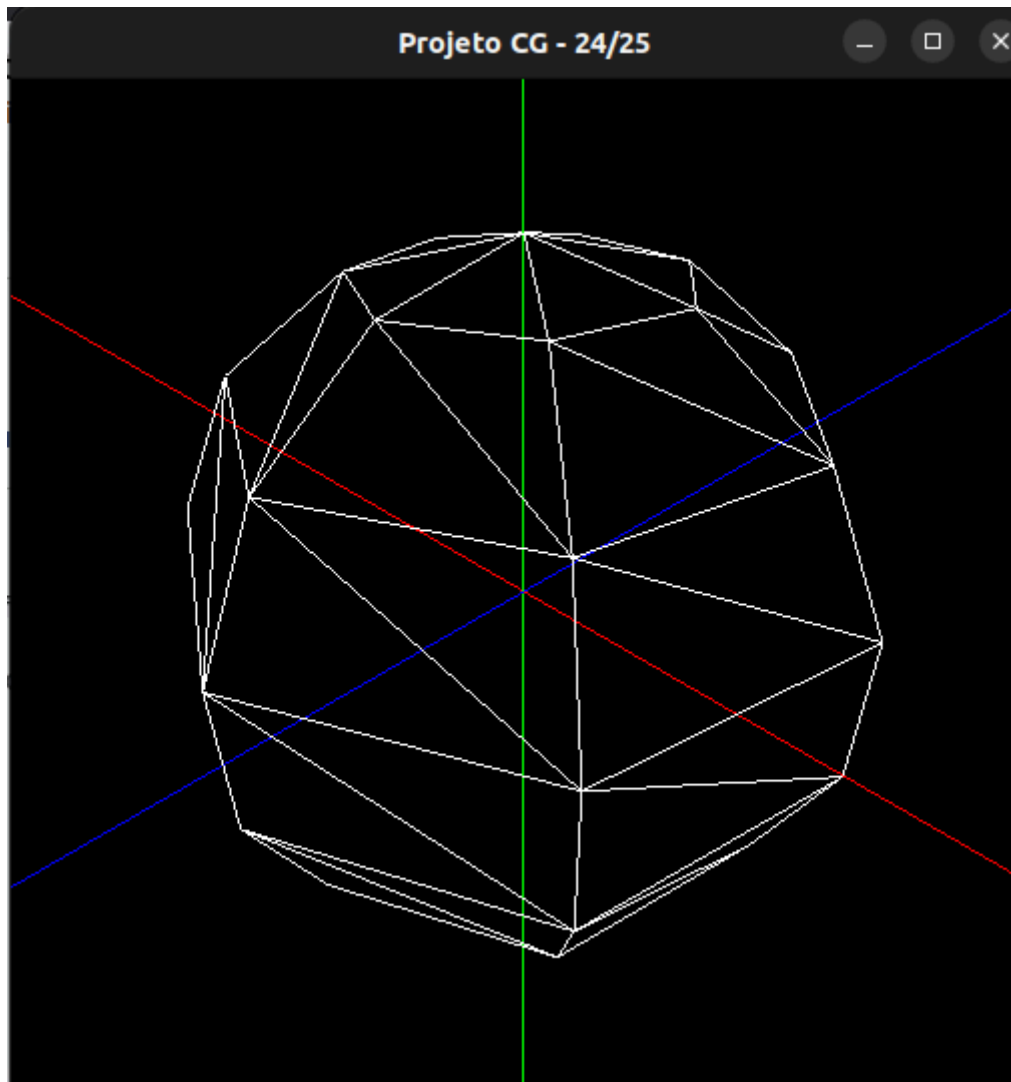


Figura 9. Plano exemplo 2

Esfera 3

Radius=2

Slices=100

Stacks=30

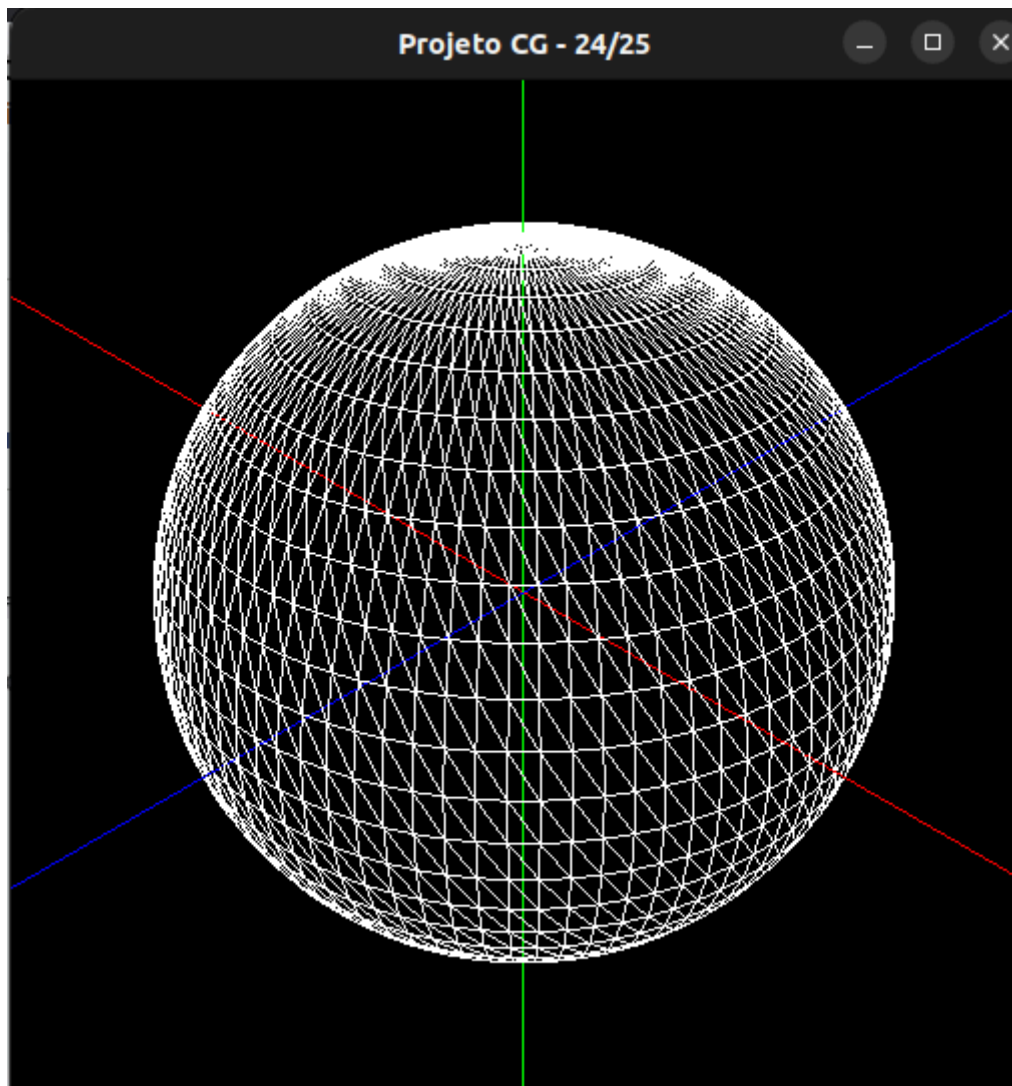


Figura 10. Esfera exemplo 3

4.2.3. Caixa

O gerador constrói uma caixa no plano XYZ dividida em pequenos triângulos, permitindo especificar o tamanho da caixa, definido por *length* e o número de subdivisões em cada face. A caixa é formada por seis faces, cada uma delas composta por vários quadrados, que são divididos em vários triângulos.

Cada subdivisão tem um tamanho determinado pelo cálculo:

tamanho da subdivisão = $length / divisions$

A metade do comprimento é calculada, facilitando o posicionamento dos vértices em relação ao centro da caixa que coincide com a origem.

Para cada subdivisão, o algoritmo itera sobre *i* e *j*, percorrendo a grade de subdivisões. Os vértices de cada face são calculados e utilizados para formar triângulos por subdivisão.

O algoritmo gera triângulos para as seis faces da caixa:

- **Face Inferior (-Y):** Gera triângulos orientados para baixo.

- **Face Superior (+Y):** Gera triângulos voltados para cima.
- **Face Frontal (+Z):** Define triângulos voltados para frente.
- **Face Traseira (-Z):** Define triângulos na direção oposta da frontal.
- **Face Esquerda (-X):** Gera triângulos apontando para o lado esquerdo.
- **Face Direita (+X):** Gera triângulos voltados para a direita.

Por fim, nas imagens abaixo estão representados alguns resultados da caixa gerada com a nossa função “geraCaixa”. Os argumentos da função variam de imagem para imagem.

Caixa 1

Length = 2

Divisions = 10

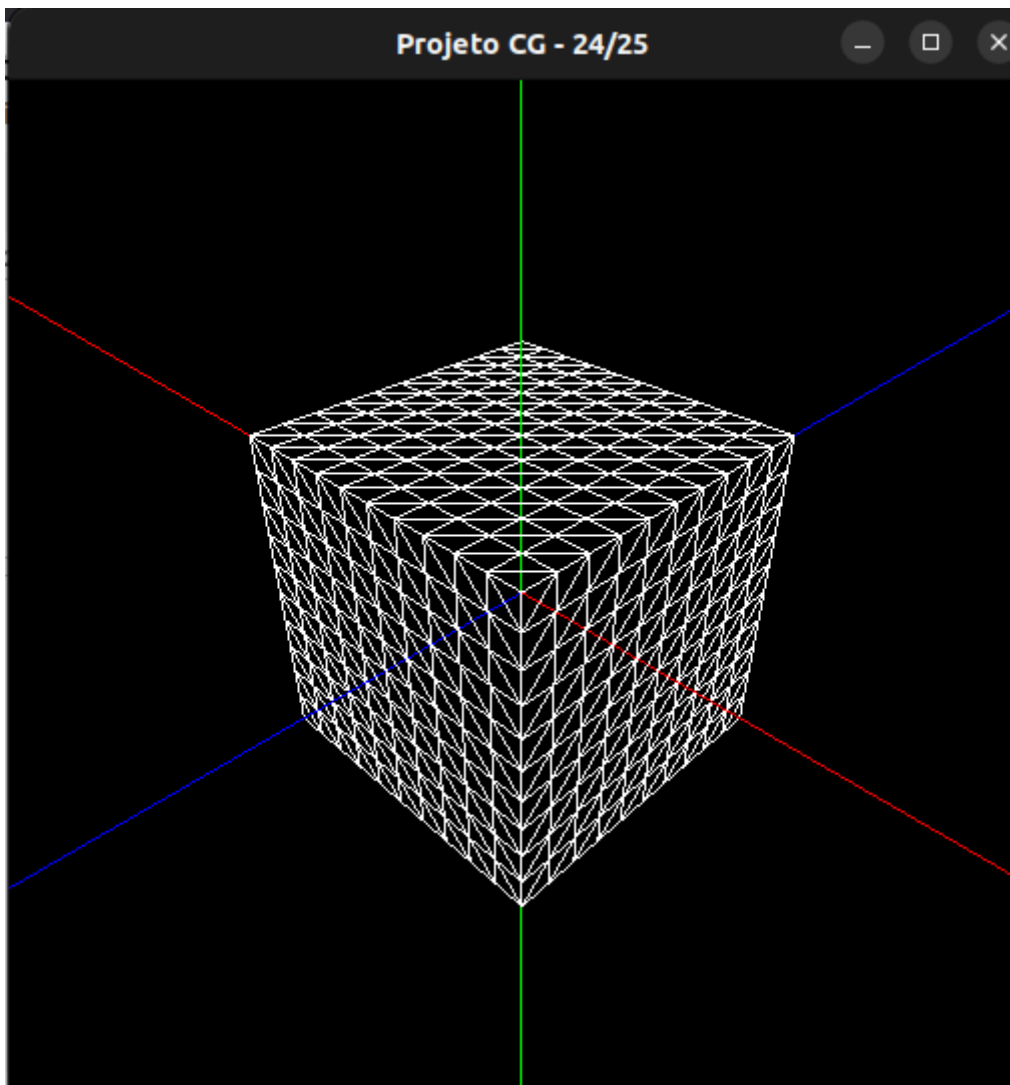


Figura 11. Caixa exemplo 1

Caixa 2

Length = 3

Divisions = 2

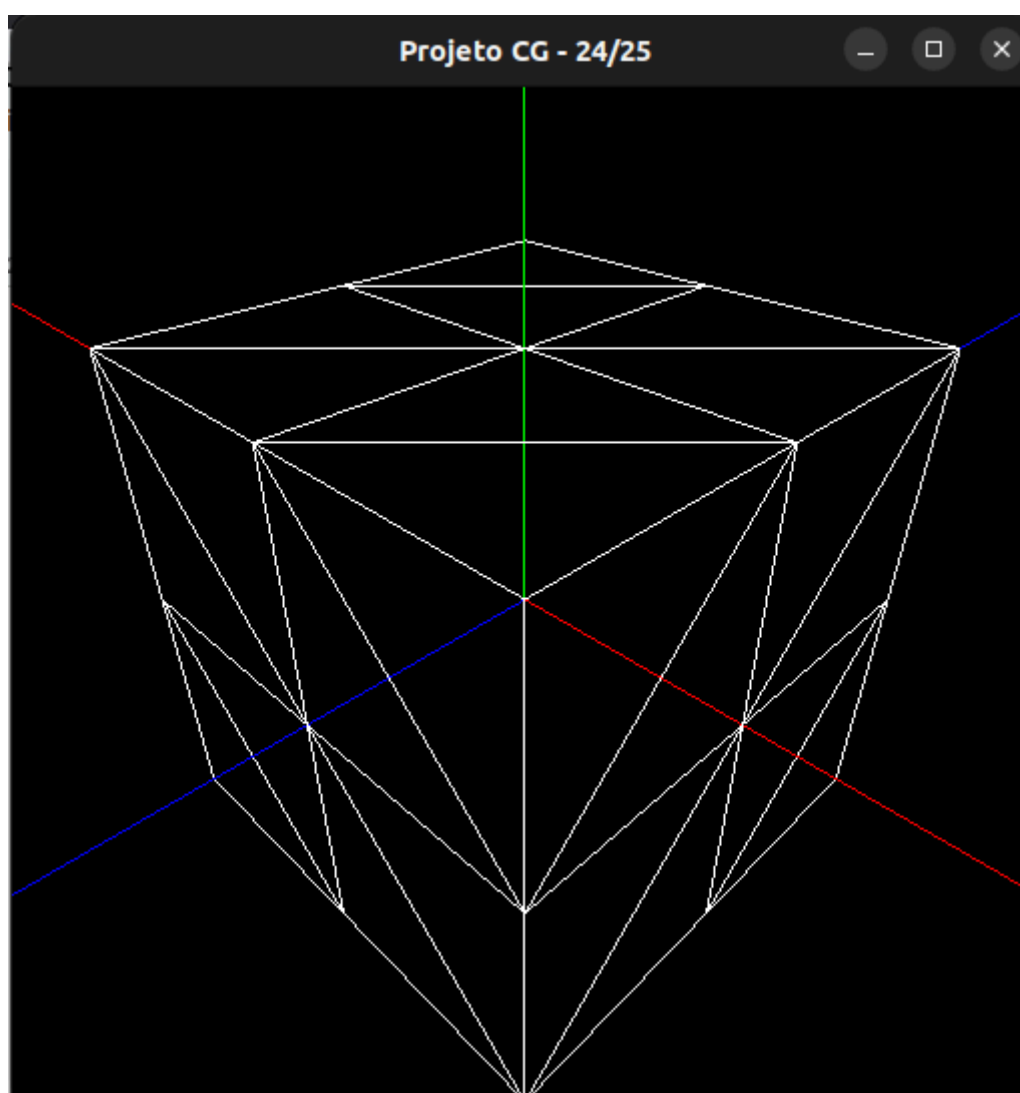


Figura 12. Caixa exemplo 2

Caixa 3

Length = 2

Divisions = 5

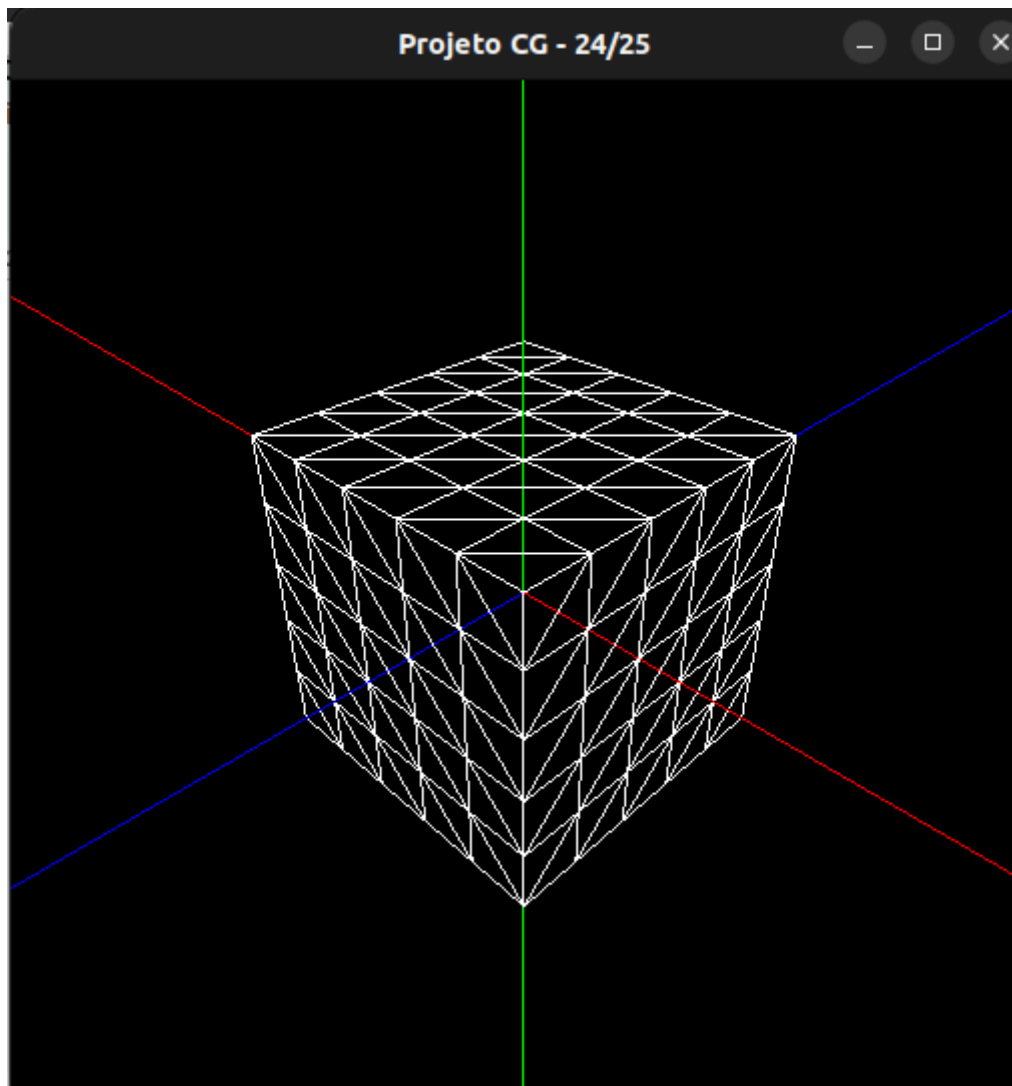


Figura 13. Caixa exemplo 3

4.2.4. Cone

O gerador cria um cone centrado na origem no plano XZ. A função que constrói este cone é a função “geraCone”. Esta função recebe como argumentos o raio (radius), a altura (height), divisões do círculo (slices) e divisões da altura (stacks).

Começamos a geração do cone gerando a base do cone. Para a criação do cone, usamos coordenadas polares, isto é, utilizamos um raio e um ângulo, pois é a forma mais simples para formarmos um modelo para a geração do cone. Após obtermos as coordenadas polares, convertemo-las em coordenadas cartesianas, usando trigonometria.

Na imagem a seguir mostra como conseguimos obter os pontos da base:

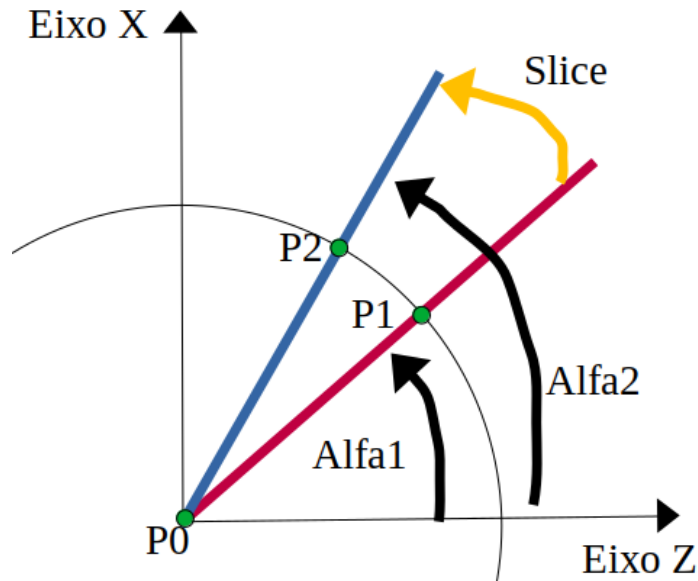


Figura 14. Fórmula para obter os pontos polares da base do cone

Slice : Ângulo resultante de dividir 360° pelo número de *slices*

- **P0** : Ponto de origem
- **P1** : Ponto obtido após aplicar um ângulo **alfa** (que no código chamamos de *alfa1*)
- **P2** : Ponto obtido após aplicar a soma de dois ângulos (ângulo alfa e ângulo slice), a que chamamos de **alfa2**

Geramos triângulos até que **alfa2** tenha dado uma volta de 360° . A forma como organizamos os pontos de modo a garantir a orientação correta dos triângulos da base do cone para que fique virado para baixo foi pela ordem P0, P2 e P1, usando a regra da mão direita.

Como já referido acima, as stacks são as divisões da altura que permitem dividir o cone em vários anéis. De modo a desenvolver o cone, para cada anel, geramos iterativamente, os triângulos que constituem cada anel da figura geométrica. Deste modo, começamos por determinar a altura de cada anel da seguinte forma:

- **delta_h** = height/stacks

Este **delta_h** representa também o “salto” que usamos para incrementar e subir progressivamente no cone.

Definimos uma altura “h” que representa a altura atual do cone, isto é, até onde já desenhamos do cone. *Este “h” inicia-se no plano XZ, isto é, tem início na base do cone.*

De seguida, utilizamos dois ciclos. O ciclo externo corresponde ao ciclo que itera pelas stacks do cone. Para cada iteração deste ciclo incrementamos o **delta_h** à altura atual do cone, “h”, de modo a determinar a altura, “h2”, do anel que estamos a desenhar nesse momento.

- **h2** = h + delta_h

Em semelhança ao que foi feito para a base do cone, utilizamos coordenadas polares para calcular as coordenadas dos vértices que constituem os triângulos do corpo do cone. Neste sentido, tivemos que calcular os raios das duas circunferências (uma em baixo e outra em cima) que constituem um anel do cone. Para determinar os raios imaginamos como seria uma fatia do cone e utilizamos semelhança de triângulos para os determinar.

A imagem abaixo fundamenta o nosso raciocínio.

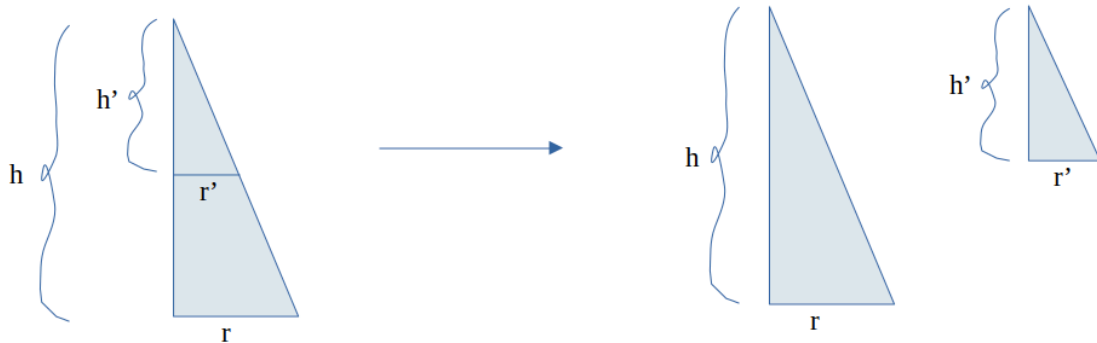


Figura 15. Imagem que representa a semelhança de triângulos

- **r** : Raio do cone
- **h** : Altura do cone
- **h'** : Comprimento que falta ao 'h' (não confundir com o 'h' da imagem) para ser igual à altura do cone
- **r'** : Raio do anel que queremos determinar

Portanto, a equação fica:

$$\frac{r}{h} = \frac{r'}{h'} \Leftrightarrow r' = \frac{r \cdot h'}{h}$$

Tendo os raios necessários para cada anel calculados, determinamos finalmente as coordenadas dos triângulos. Neste sentido, no ciclo interno, fizemos um raciocínio semelhante ao da base do cone, representado na *Figura 14*, ou seja, utilizamos dois ângulos consecutivos, respectivamente, alfa1 e alfa2. Com estes ângulos determinados foi possível definir as coordenadas de cada vértice do triângulo. A imagem abaixo representa a forma como calculamos essas coordenadas.

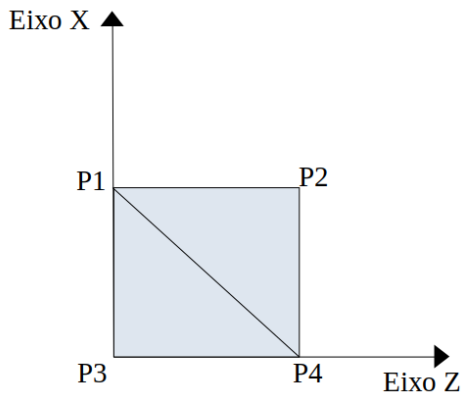


Figura 16. Como calculamos os triângulos que fazem o anel do cone

- **P1** : Ponto do nível de cima do anel que usa o ângulo *alfa1*
- **P2** : Ponto do nível de cima do anel que usa o ângulo *alfa2*
- **P3** : Ponto do nível de baixo do anel que usa o ângulo *alfa1*
- **P4** : Ponto do nível de baixo do anel que usa o ângulo *alfa2*

A forma como organizamos os pontos de modo a garantir a orientação correta dos triângulos no corpo do cone foi a seguinte: no primeiro triângulo a ordem de pontos foi P3, P4 e P1, no segundo triângulo foi P1, P4 e P2. Além disso, o método que seguimos para descobrir qual a ordem correta, foi utilizar a regra da mão direita

Por fim, nas imagens abaixo estão representados alguns resultados do cone gerado com a nossa função “geraCone”. Os argumentos da função variam de imagem para imagem.

Cone 1

Radius = 1

Height = 2

Slices = 5

Stacks = 5

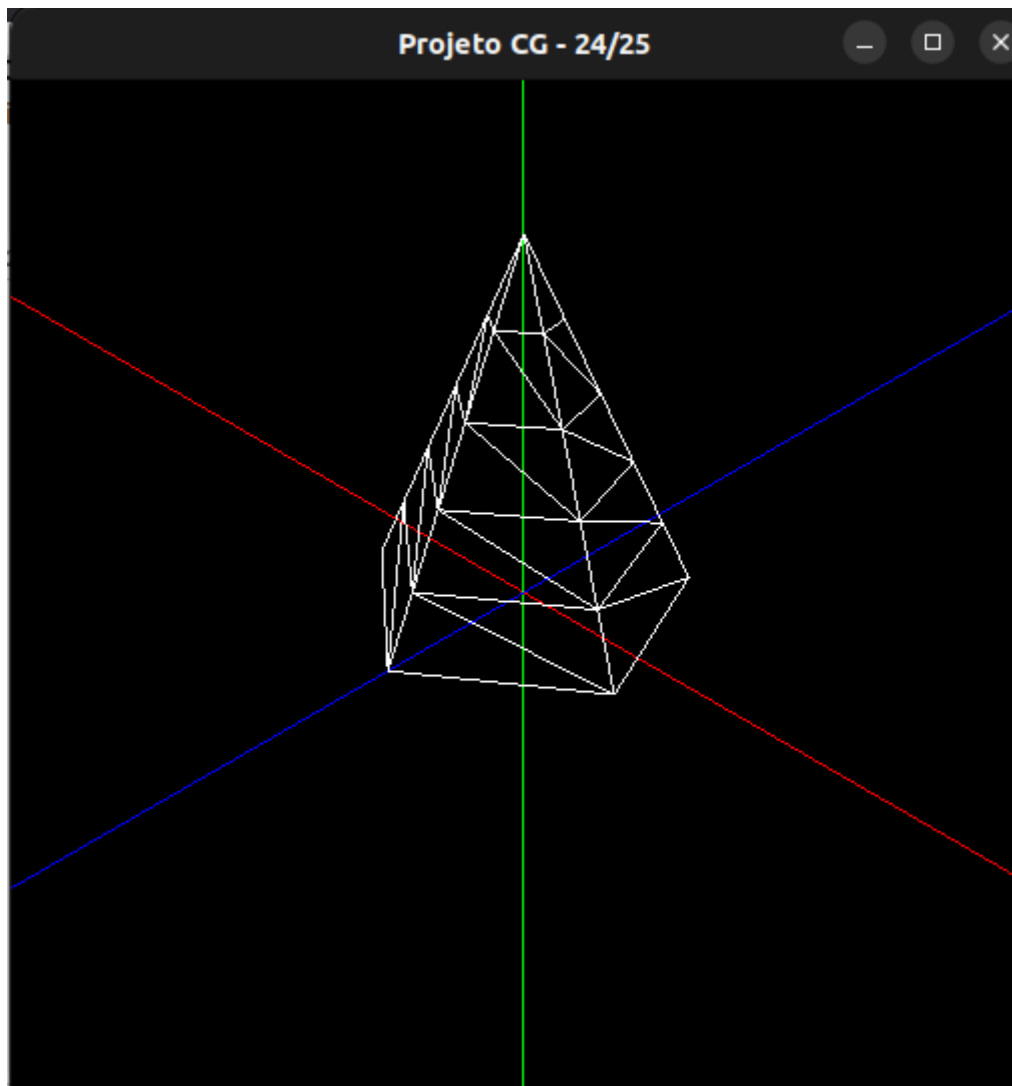


Figura 17. Cone exemplo 1

Cone 2

Radius = 1
Height = 2
Slices = 10
Stacks = 5

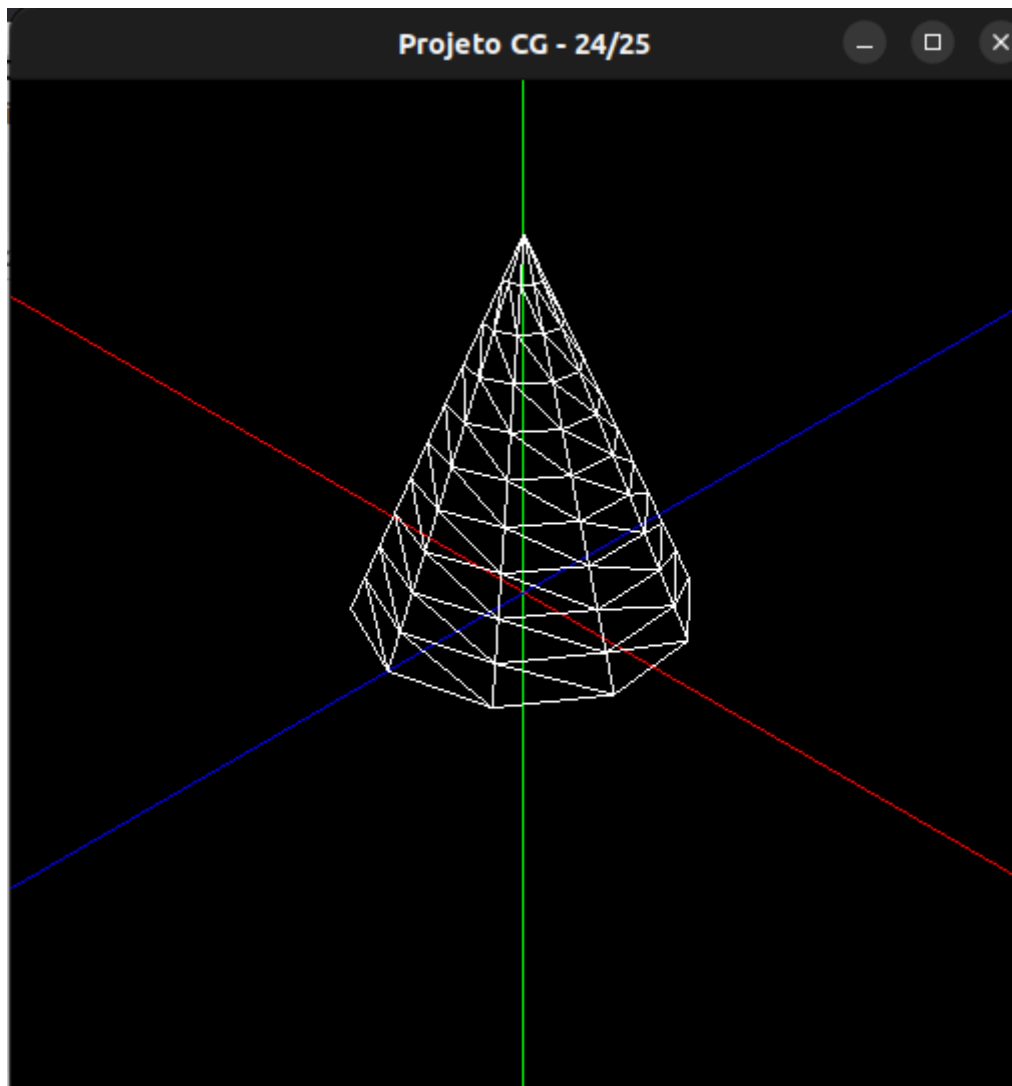


Figura 18. Cone exemplo 2

Cone 3

Radius=2

Height=2

Slices=100

Stacks=2

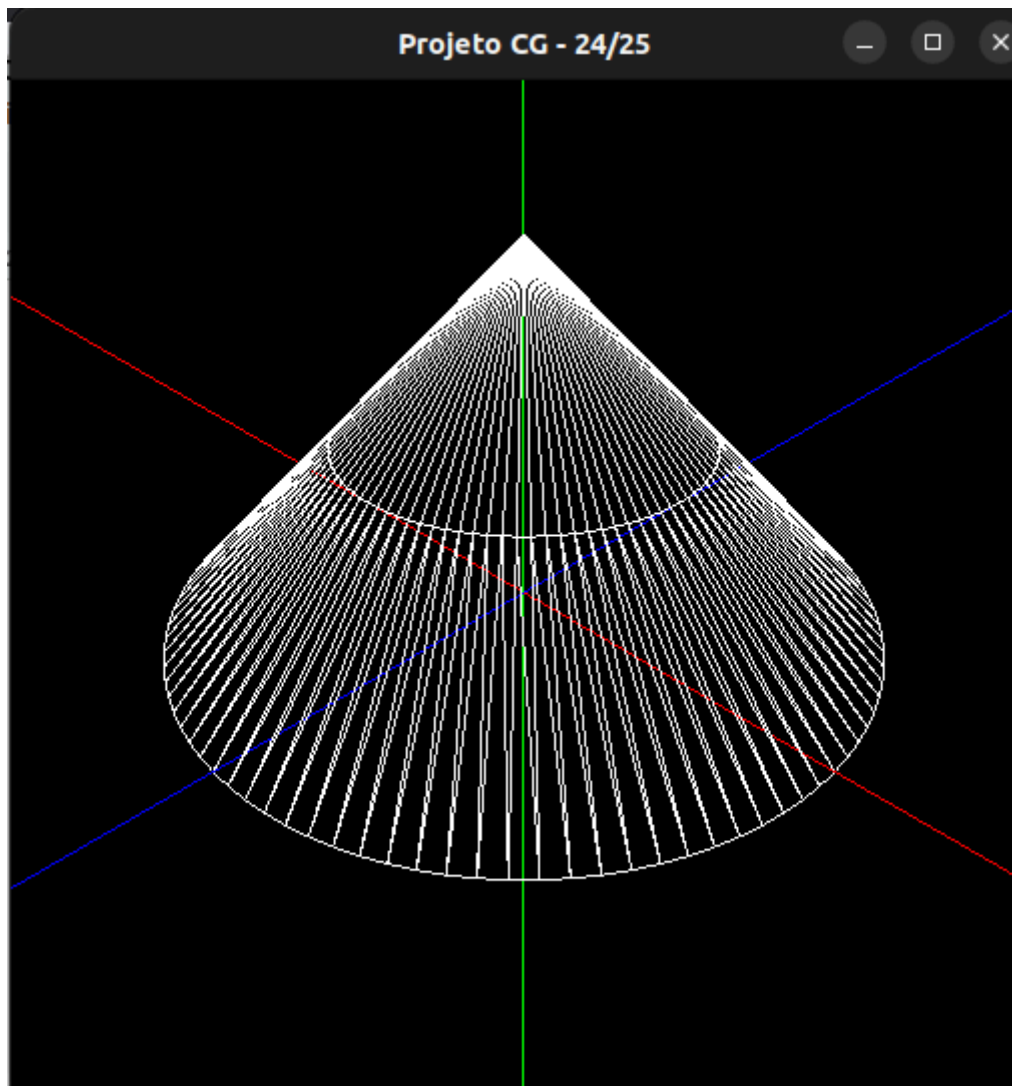


Figura 19. Cone exemplo 3

5. Engine

A aplicação **engine** é a aplicação responsável por desenhar os modelos no ecrã, usando **OpenGL**.

A aplicação recebe como único argumento o *path* para o **ficheiro XML** que irá ler. Neste ficheiro XML, está descrito como devemos desenhar o cenário, indicando quais os parâmetros da janela, da câmara e quais os modelos a desenhar (*indicando o path dos ficheiros .3d desses modelos*). Conseguimos visualizar como a leitura do ficheiro XML correu vendo o ficheiro de texto **xml_logger.txt**, como foi mencionado anteriormente.

Se a leitura do XML correu bem, então a **engine** cria a janela como foi indicada pelo XML, posiciona a câmara (também indicado pelo XML) e desenha todos os pontos de forma linear indicados pelos ficheiros .3d.

A nossa engine apenas desenha a face da frente dos triângulos e usamos um objeto chamado **World**, onde nele estão todas as configurações que deverão ser usadas pela engine, ditas no ficheiro XML lido anteriormente, como também uma lista de todos os modelos a desenhar.

A nossa engine apenas desenha o cenário uma vez após a inicialização do programa ou quando alguma alteração é efetuada. Isto garante que não há **DrawCalls** desnecessárias, aumentando o desempenho do programa.

Nota: Na engine é possível utilizar o teclado para manipular o cenário, mas como é algo temporário, decidimos não explicar essa funcionalidade neste relatório

6. Como usar

Nesta secção vamos explicar como usar o nosso projeto para desenhar os cenários descritos no ficheiro de testes dado pelo professor e nas nossas demos.

6.1. CMake - Testado apenas em Linux/MacOS

Devemos usar o **CMake** para executar o nosso ficheiro chamado **CMakeLists.txt**. A pasta **build** é a pasta onde o projeto deverá ser compilado:

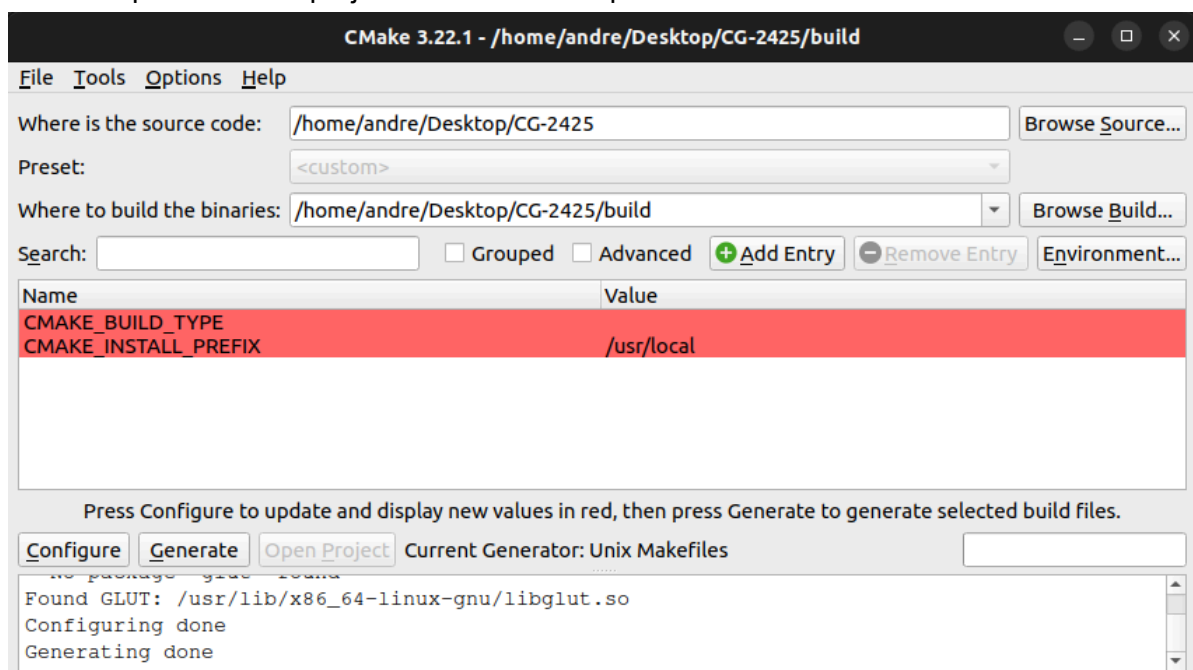
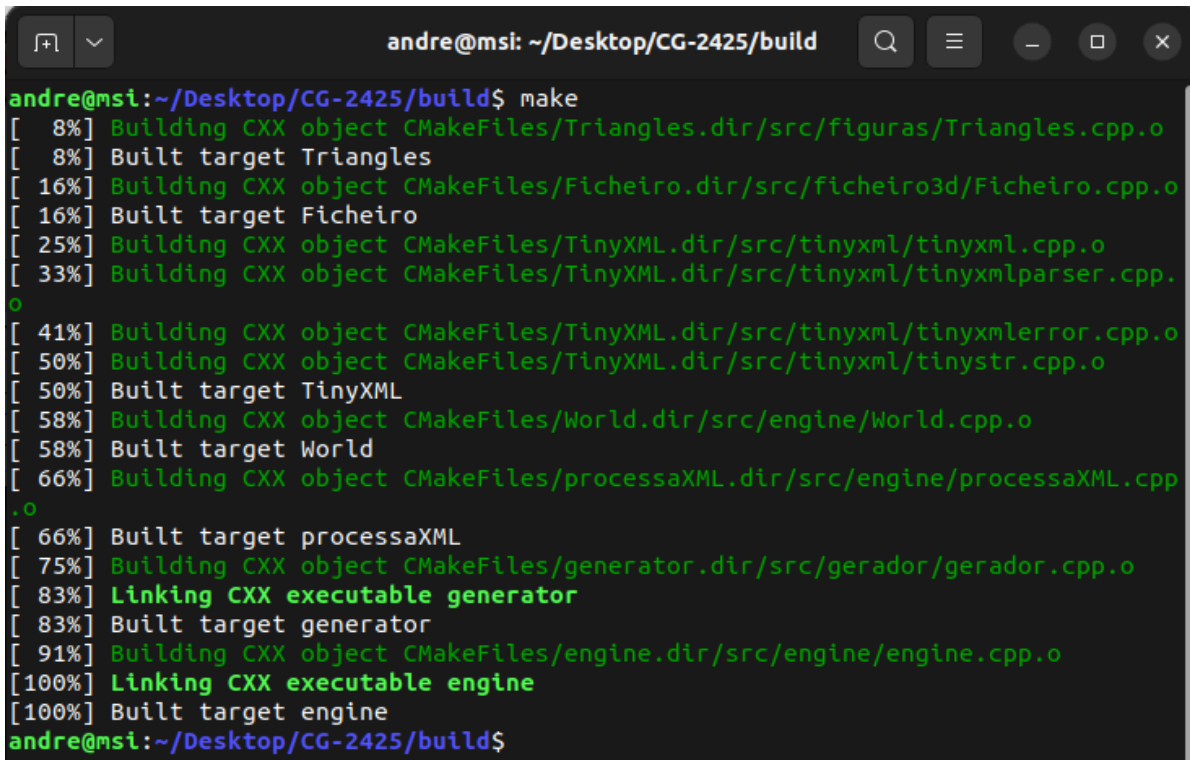


Figura 20. CMake

Após isso, devemos executar a **Makefile** situada dentro da pasta **build**:

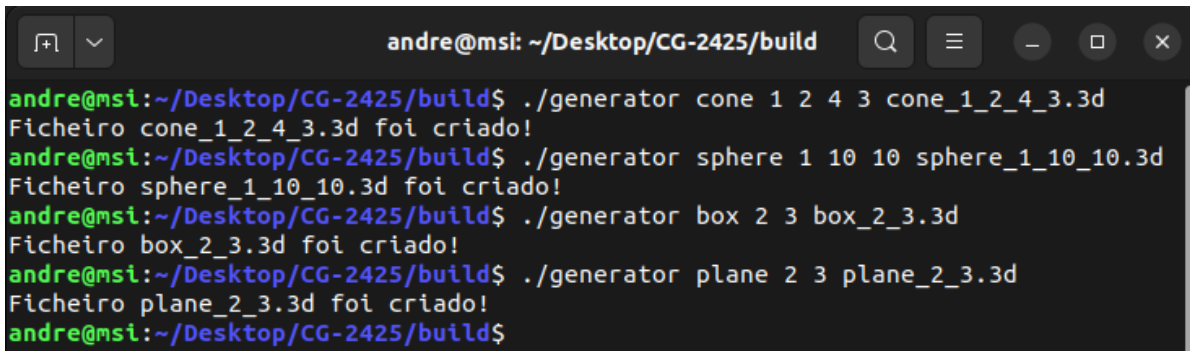


```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ make
[ 8%] Building CXX object CMakeFiles/Triangles.dir/src/figuras/Triangles.cpp.o
[ 8%] Built target Triangles
[ 16%] Building CXX object CMakeFiles/Ficheiro.dir/src/ficheiro3d/Ficheiro.cpp.o
[ 16%] Built target Ficheiro
[ 25%] Building CXX object CMakeFiles/TinyXML.dir/src/tinyxml/tinyxml.cpp.o
[ 33%] Building CXX object CMakeFiles/TinyXML.dir/src/tinyxml/tinyxmlparser.cpp.o
[ 41%] Building CXX object CMakeFiles/TinyXML.dir/src/tinyxml/tinyxmlerror.cpp.o
[ 50%] Building CXX object CMakeFiles/TinyXML.dir/src/tinyxml/tinystr.cpp.o
[ 50%] Built target TinyXML
[ 58%] Building CXX object CMakeFiles/World.dir/src/engine/World.cpp.o
[ 58%] Built target World
[ 66%] Building CXX object CMakeFiles/processaXML.dir/src/engine/processaXML.cpp.o
[ 66%] Built target processaXML
[ 75%] Building CXX object CMakeFiles/generator.dir/src/gerador/gerador.cpp.o
[ 83%] Linking CXX executable generator
[ 83%] Built target generator
[ 91%] Building CXX object CMakeFiles/engine.dir/src/engine/engine.cpp.o
[100%] Linking CXX executable engine
[100%] Built target engine
andre@msi:~/Desktop/CG-2425/build$
```

Figura 21. Execução da makefile

6.2. Geração de figuras

Para a geração de figuras, usamos o executável **generator**:



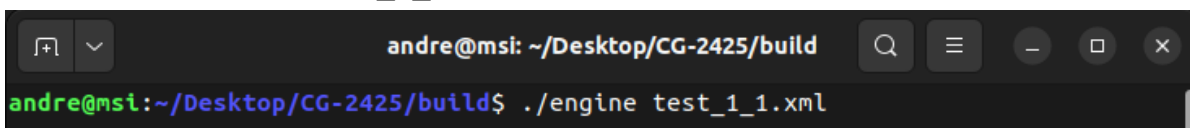
```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./generator cone 1 2 4 3 cone_1_2_4_3.3d
Ficheiro cone_1_2_4_3.3d foi criado!
andre@msi:~/Desktop/CG-2425/build$ ./generator sphere 1 10 10 sphere_1_10_10.3d
Ficheiro sphere_1_10_10.3d foi criado!
andre@msi:~/Desktop/CG-2425/build$ ./generator box 2 3 box_2_3.3d
Ficheiro box_2_3.3d foi criado!
andre@msi:~/Desktop/CG-2425/build$ ./generator plane 2 3 plane_2_3.3d
Ficheiro plane_2_3.3d foi criado!
andre@msi:~/Desktop/CG-2425/build$
```

Figura 22. Criação de figuras usando o Generator

6.3. Desenhar o Cenário

Para desenharmos o cenário, vamos usar o executável **engine** e vamos dar-lhe como argumento o ficheiro XML para cada cenário:

6.3.1. Ficheiro test_1_1.xml



```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./engine test_1_1.xml
```

Figura 23. Execução da Engine do Teste 1 da Fase 1

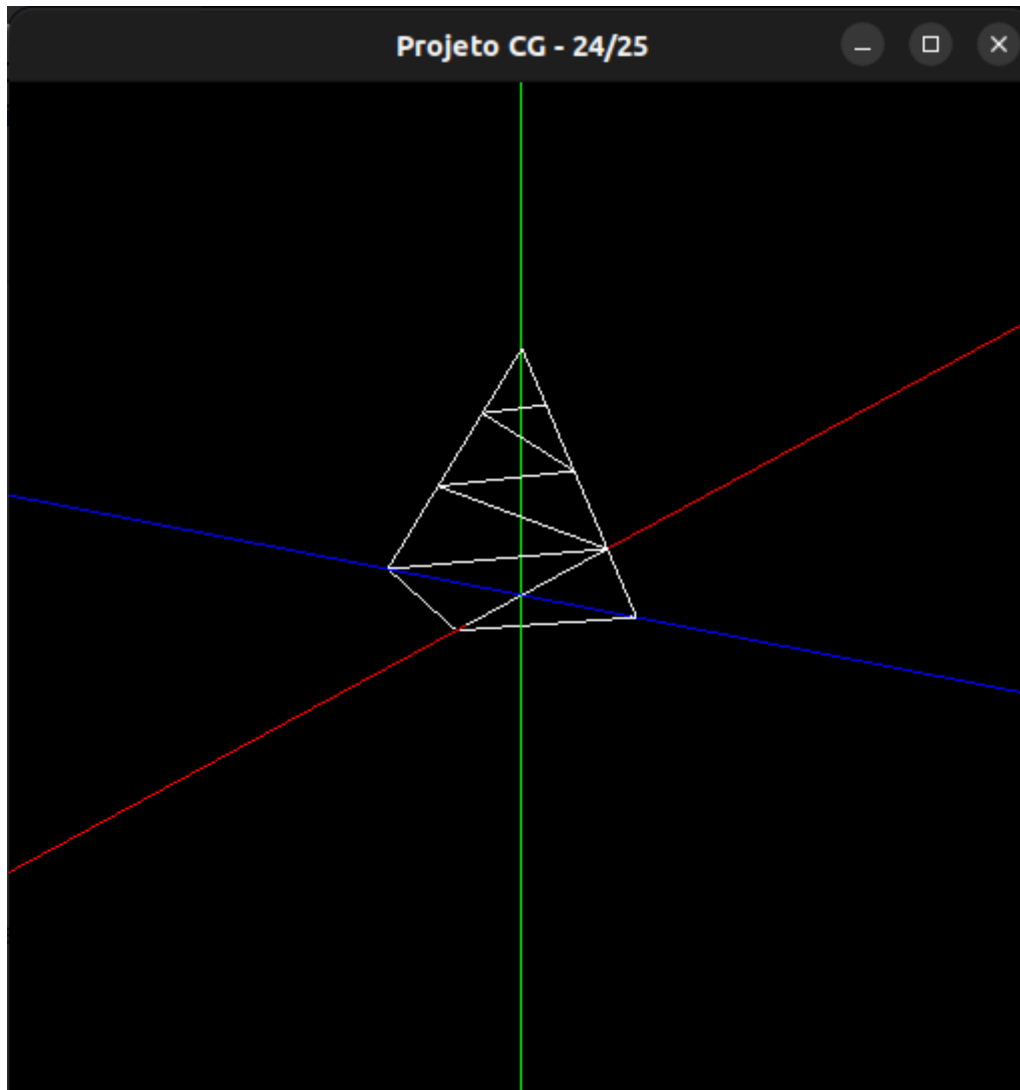


Figura 24. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 1 da Fase 1

6.3.2. Ficheiro test_1_2.xml

```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./engine test_1_2.xml
```

Figura 25. Execução da Engine do Teste 2 da Fase 1

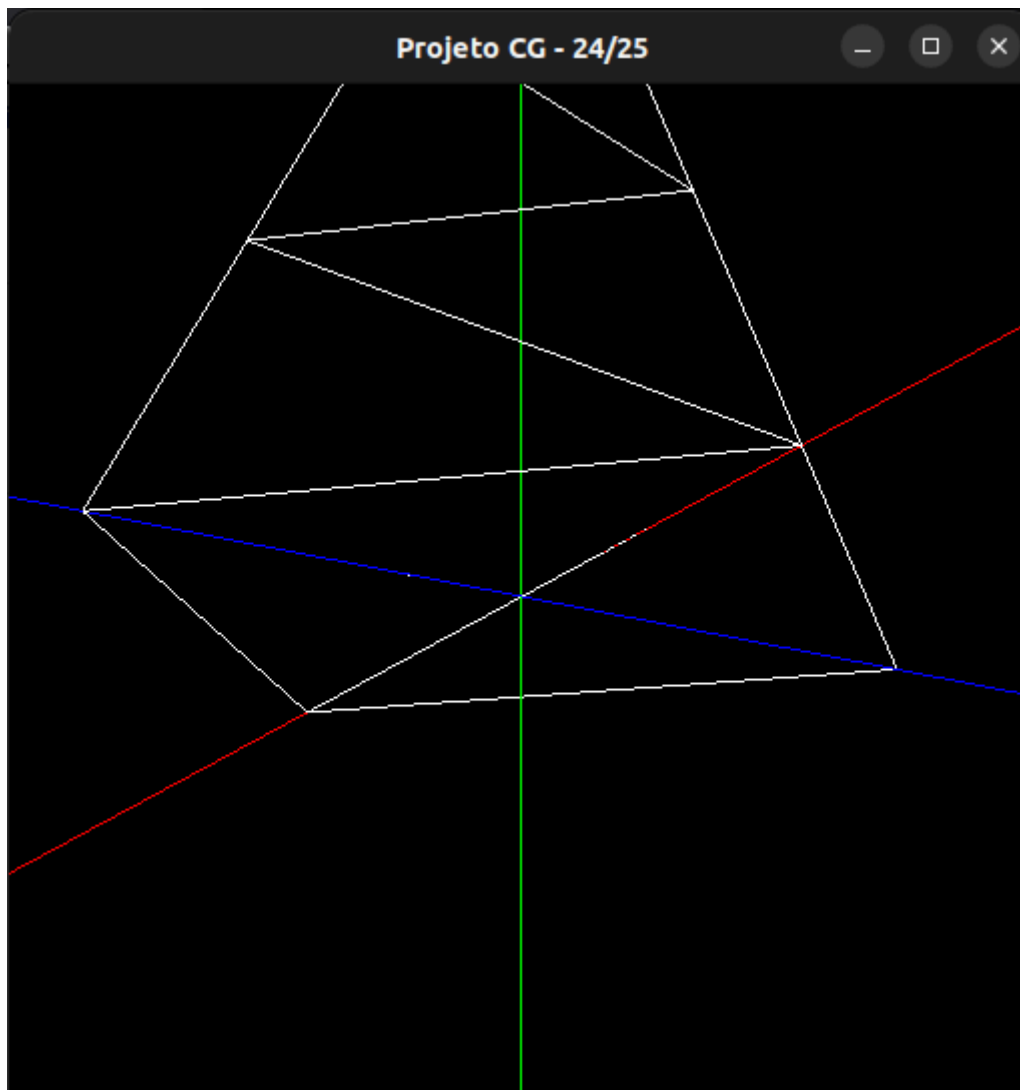


Figura 26. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 2 da Fase 1

6.3.3. Ficheiro test_1_3.xml

```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./engine test_1_3.xml
```

Figura 27. Execução da Engine do Teste 3 da Fase 1

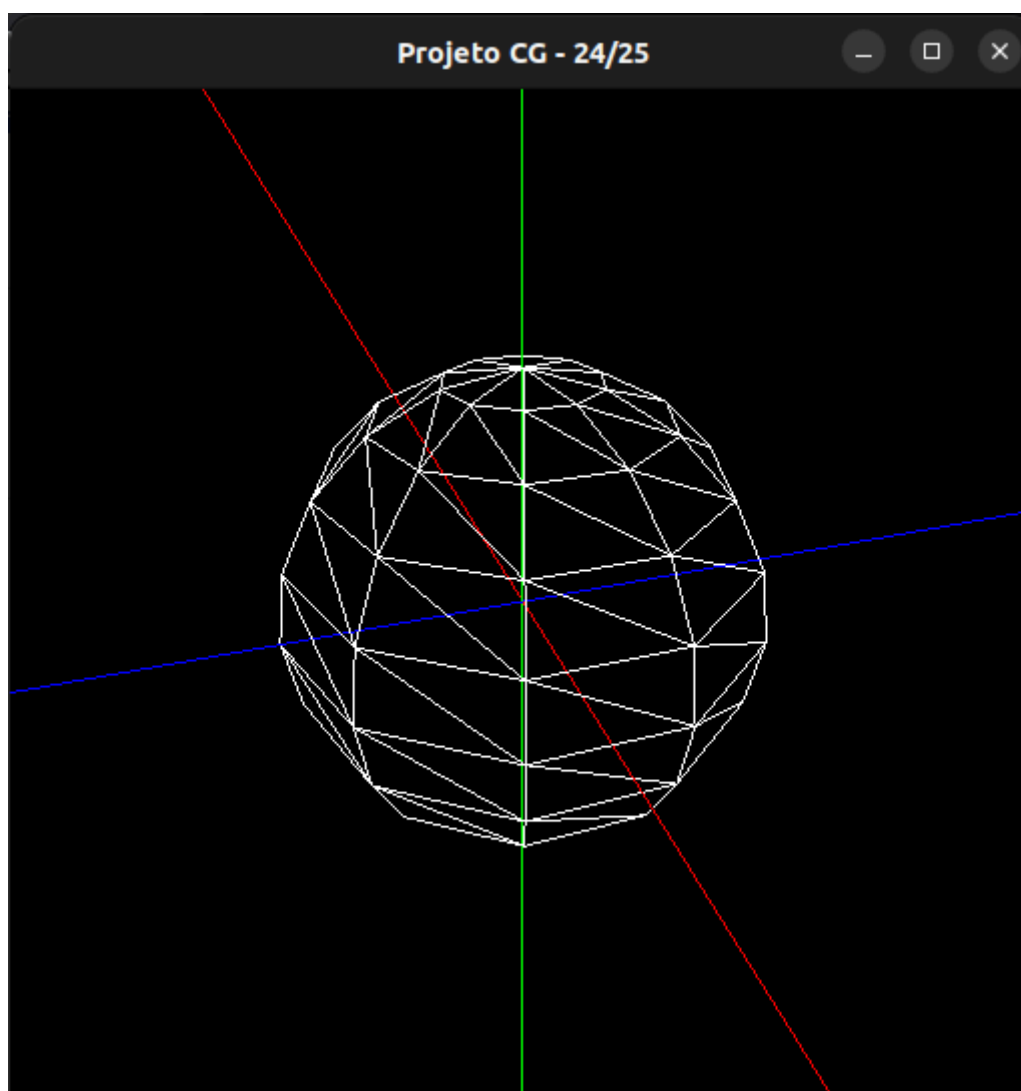


Figura 28. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 3 da Fase 1

6.3.4. Ficheiro test_1_4.xml

```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./engine test_1_4.xml
```

Figura 29. Execução da Engine do Teste 4 da Fase 1

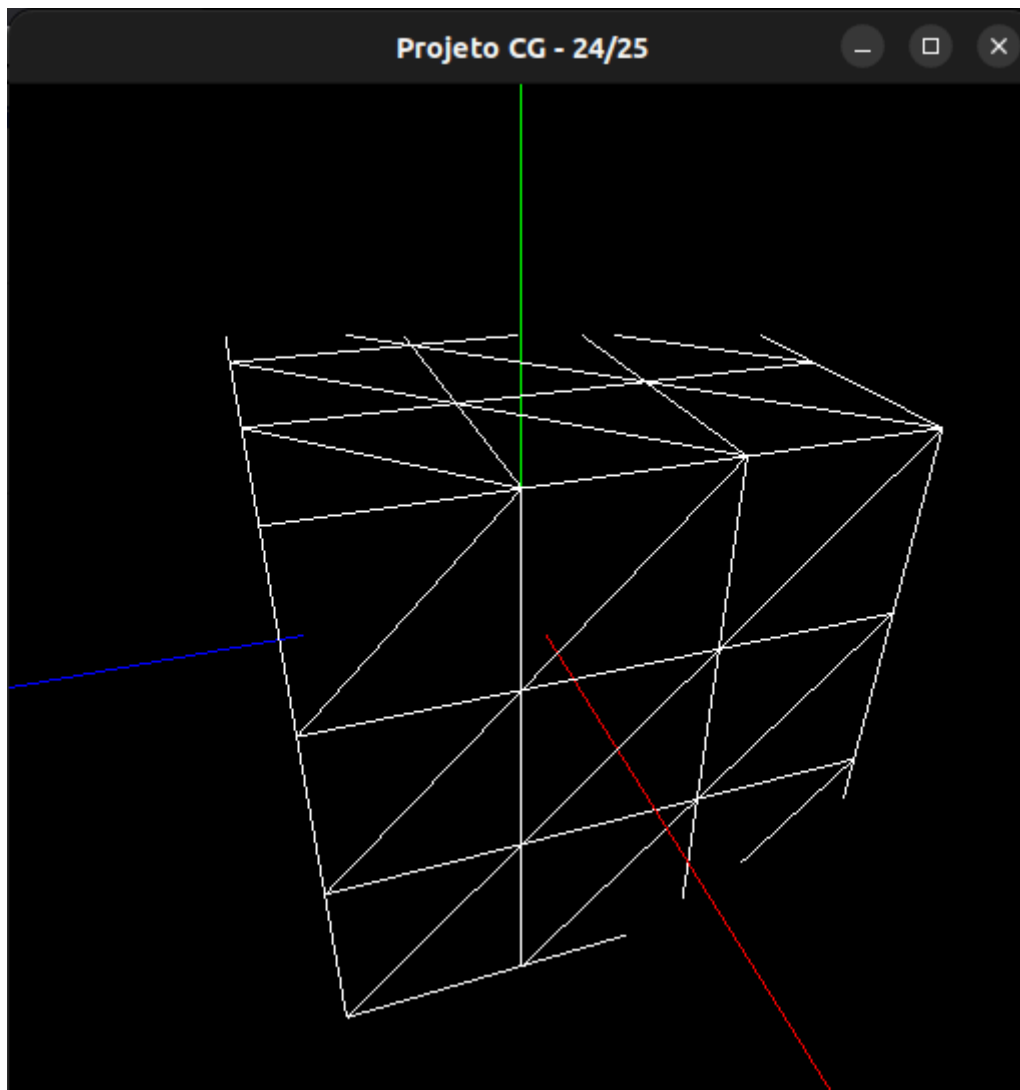


Figura 30. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 4 da Fase 1

6.3.5. Ficheiro test_1_5.xml

```
andre@msi: ~/Desktop/CG-2425/build
andre@msi:~/Desktop/CG-2425/build$ ./engine test_1_5.xml
```

Figura 31. Execução da Engine do Teste 5 da Fase 1

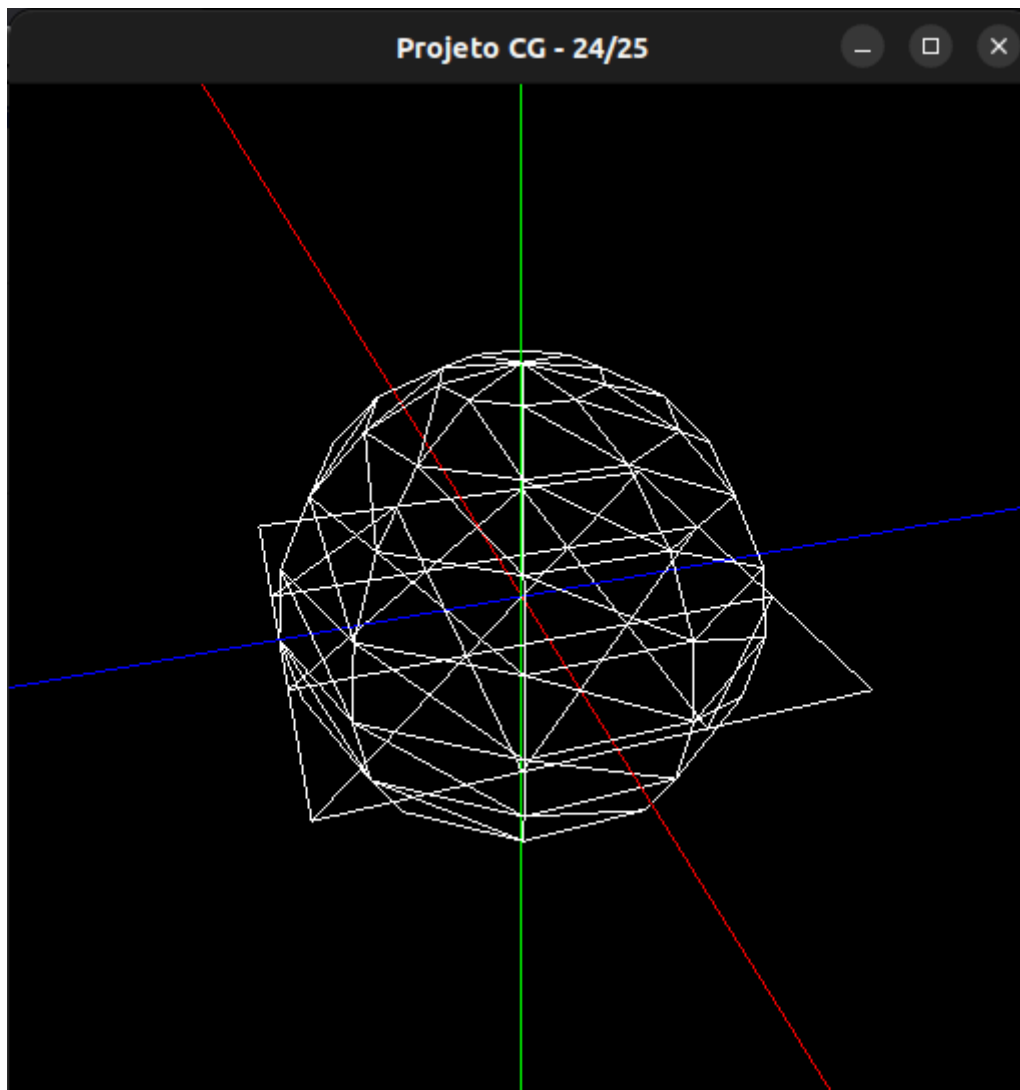


Figura 32. Cenário desenhado pela Engine descrito pelo ficheiro XML Teste 5 da Fase 1

6.3.6. Ficheiro demo1.xml

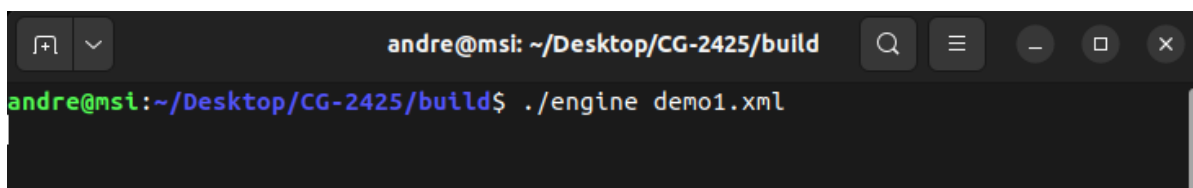


Figura 33. Execução da Engine da Demo 1 da Fase 1

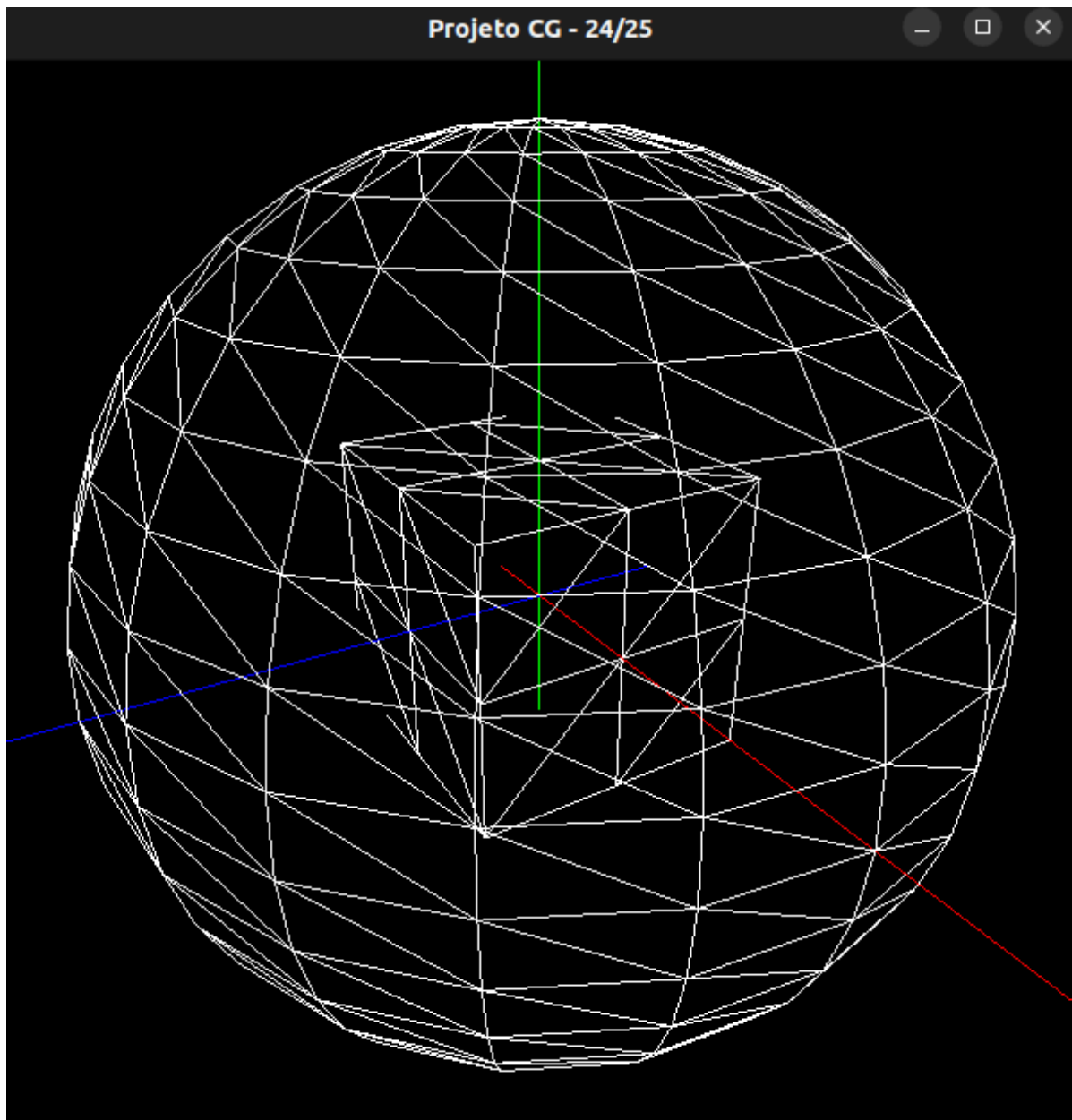


Figura 34. Cenário desenhado pela Engine descrito pelo nosso ficheiro XML Demo 1 da Fase 1

6.3.7. Ficheiro demo2.xml

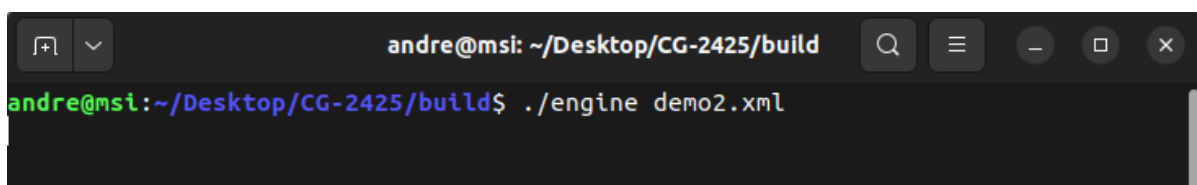


Figura 35. Execução da Engine da Demo 2 da Fase 1

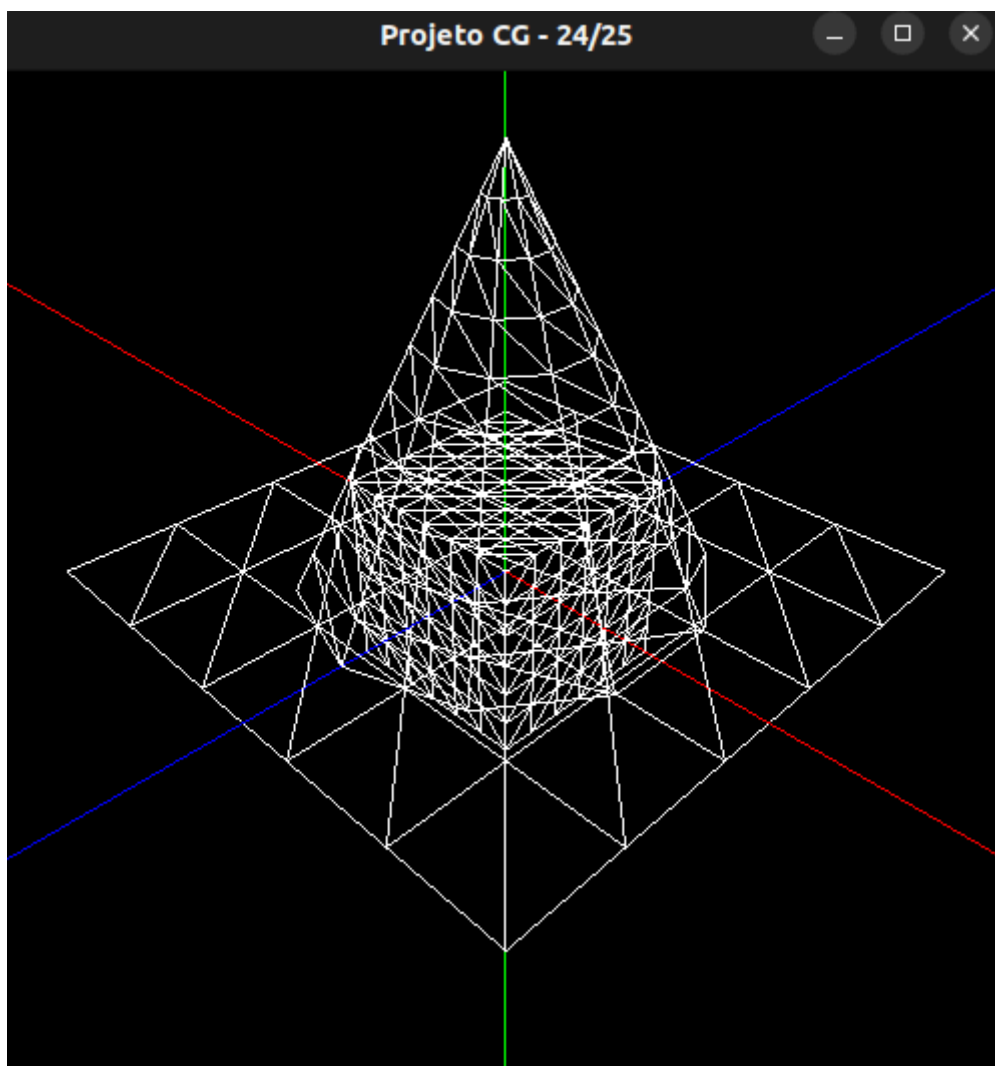


Figura 36. Cenário desenhado pela Engine descrito pelo nosso ficheiro XML Demo 2 da Fase 1

7. Conclusão

Com este projeto, conseguimos realizar e produzir as primitivas pedidas nesta fase, isto é, o plano, a esfera, a caixa e o cone. Produzimos uma aplicação chamada **generator** que cria um ficheiro .3d com o modelo desejado e produzimos uma aplicação chamada **engine** que lê um **ficheiro XML** e os **ficheiros .3d** referenciados por este e desenha todo o cenário descrito pelo XML.

Nesta fase conseguimos aprender como desenhar cenários com o OpenGL e como transformar modelos da vida real num conjunto de triângulos, que em harmonia, são capazes de representar esses modelos no ecrã, desenhando esses triângulos. Conseguimos aplicar técnicas de otimizações e melhorias de desempenho em OpenGL fazendo com que apenas sejam desenhadas as faces da frente dos triângulos e evitando drawcalls desnecessárias, executando-as apenas quando há alterações no cenário ou quando ele é desenhado pela primeira vez.

Futuramente queremos adicionar novos modelos mais complexos para enriquecer este trabalho.