Computação Gráfica

Phase 1

GRUPO 6
A95917 - Eduardo Diogo Costa Soares - LCC
A95609 - Duarte Alexandre Oliveira Faria - LEI
A97941 - Diogo Filipe Oliveira da Silva - LEI
A98979 - Pedro Domingues Viana - LCC

Introdução

Para a primeira fase foram desenvolvidas duas aplicações: o generator, para gerar ficheiros XML com informação para criar vértices das figuras e um engine, que vai ler o ficheiro XML, e apresentar as figuras.

Generator

No generator nesta fase podemos criar 4 ficheiros .3d como pedido no enunciado, o plane , a box, a sphere e o cone. Cada um dos ficheiros é simplesmente o número de vértices total na primeira linha e todos os vértices linha a linha nas seguintes.

O **plane** é gerado no plano *xz* centrado na origem igualmente dividido entre o *x* e o *z* como pedido.
Para o gerar só precisamos do seu *length* e das *divisions*.
O número de vértices é independente do *length*, basicamente o número de vértices é

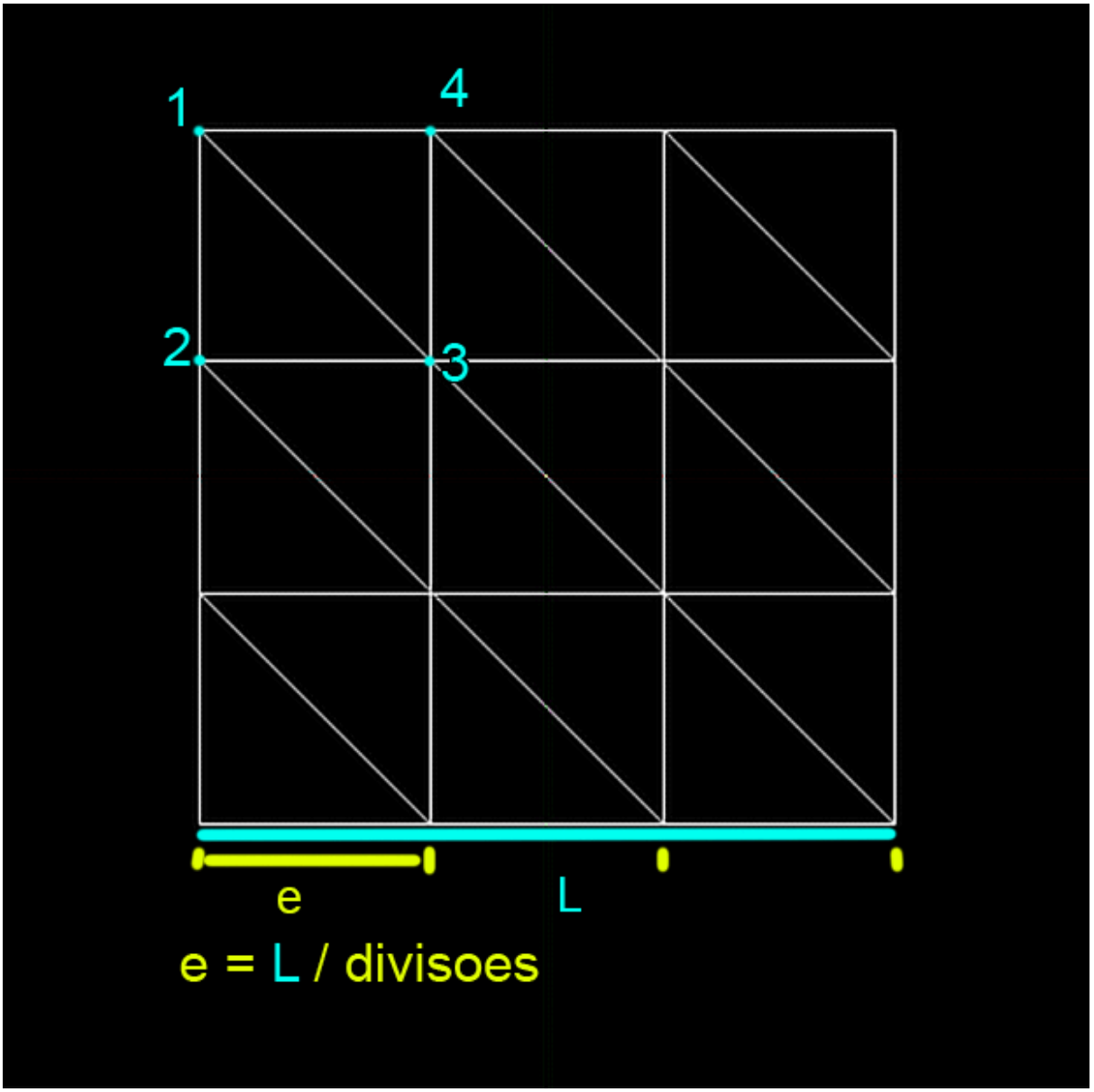
$$divisions * divisions * 2 * 3 * 2$$

Sendo que o número de divisões representa o número de quadrados por linha e por coluna, e cada quadrado tem 2 triângulos e cada triângulo tem 3 vértices, e, para podermos representar o plano frente e verso a usar o `glEnable(GL_CULL_FACE);` temos de escrever um plano virado para cima e um plano virado para baixo.
Para criar o plano, seguimos linha a linha, coluna a coluna, e criamos os triângulos virados para cima e depois os virados para baixo, para percorrer os vários pontos precisamos de decidir quanto alteramos os valores, que será o $espaçamento = length / divisions$.
Neste algoritmo repetimos vértices.

A **box** é centrada na origem e precisa apenas da dimension e das divisions.
O número de vértices é independente da dimension, basicamente o número de vértices é

$$divisions * divisions * 2 * 3 * 6$$

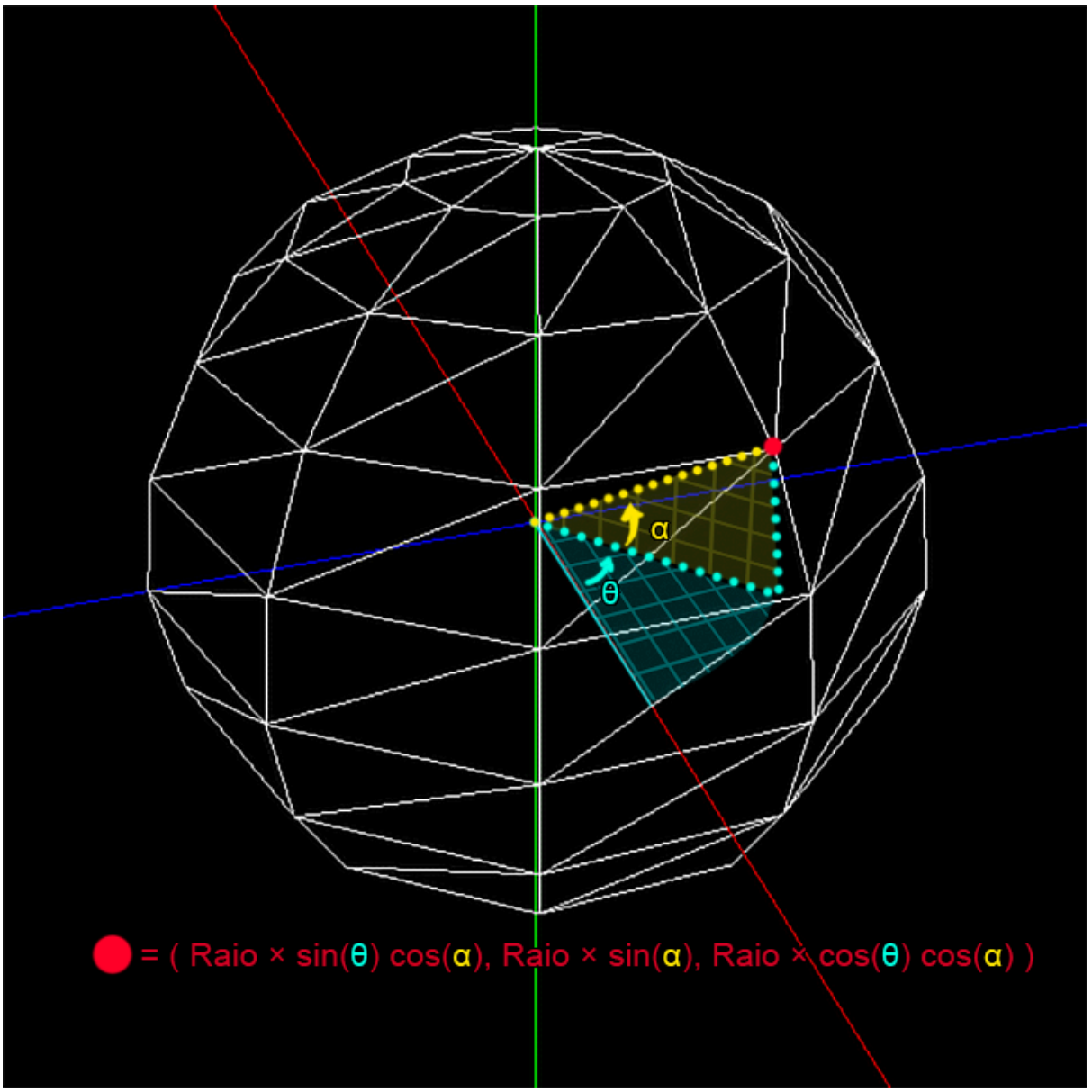
O nosso algoritmo neste caso foi apenas criar 6 planos, 2 no eixo xy, 2 no xz e 2 no yz distanciados pela dimension, só tivemos de ter em atenção a direção dos planos

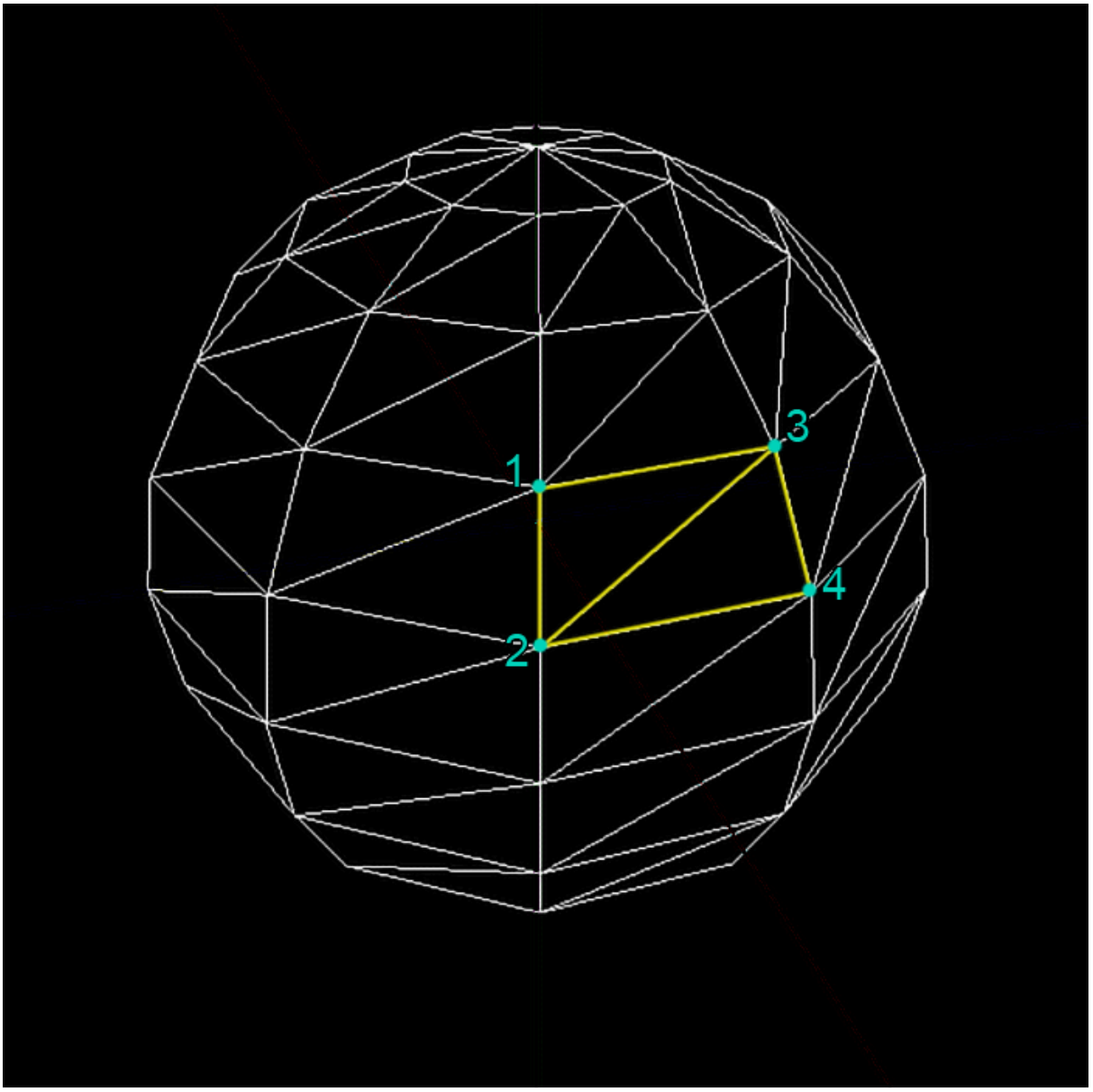


A **sphere** é centrada na origem e precisa de um radius, slices e stacks para ser gerada.
O número de vértices necessários para o nosso algoritmo é

$$slices * (6 + (stacks * 6))$$

Ou seja, não depende do radius, o algoritmo por cada slice escreve os vértices do topo e da base da esfera nessa slice, ou seja 1 triângulo para cada, e depois, por cada stack, escreve os vértices dos 2 triângulos necessários para a construir.
Para percorrer de ponto a ponto precisamos de saber o ângulo que fazer, neste caso dois ângulos porque estamos percorrer de duas maneiras a esfera, de cima para baixo e da esquerda para a direita, de cima para baixo temos a $longitude = pi / stacks$ que só precisa de rodar 180 graus e da esquerda para a direita temos a $latitude = 2 * pi / slices$ que precisa de rodar 360 graus.
Aqui também repetimos vértices.

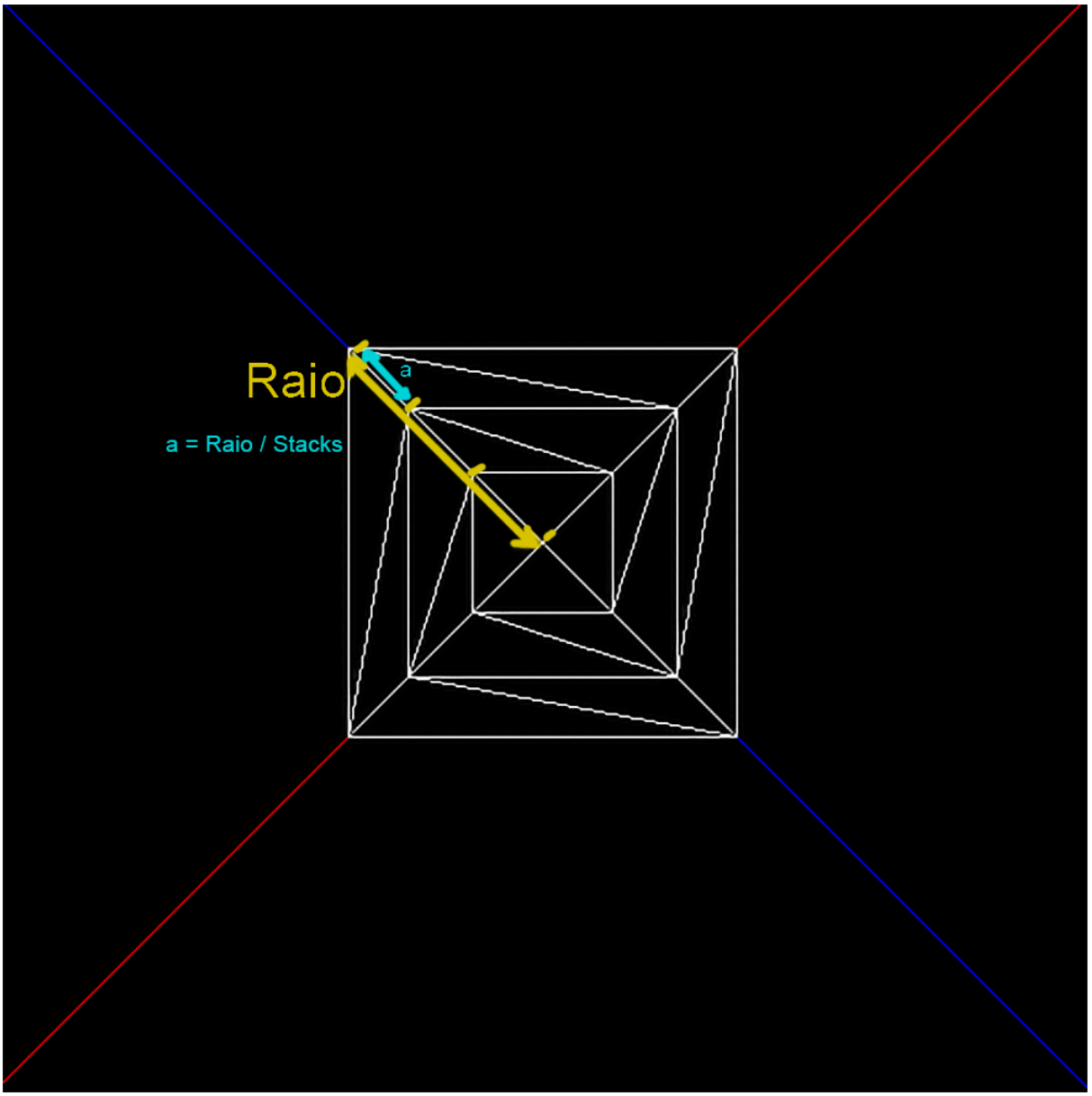

α
θ
= (Raio × sin(θ) cos(α), Raio × sin(α), Raio × cos(θ) cos(α))

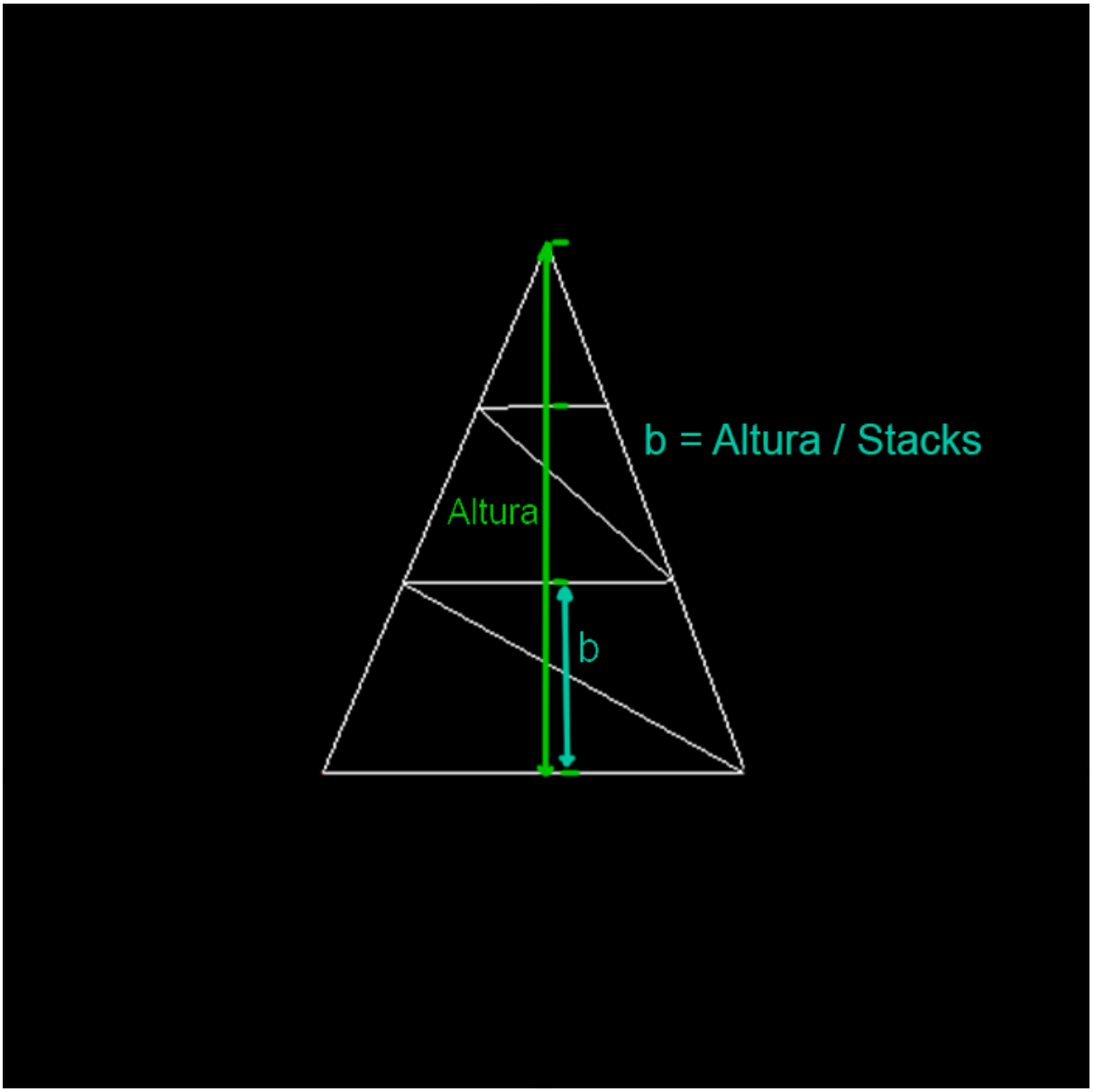


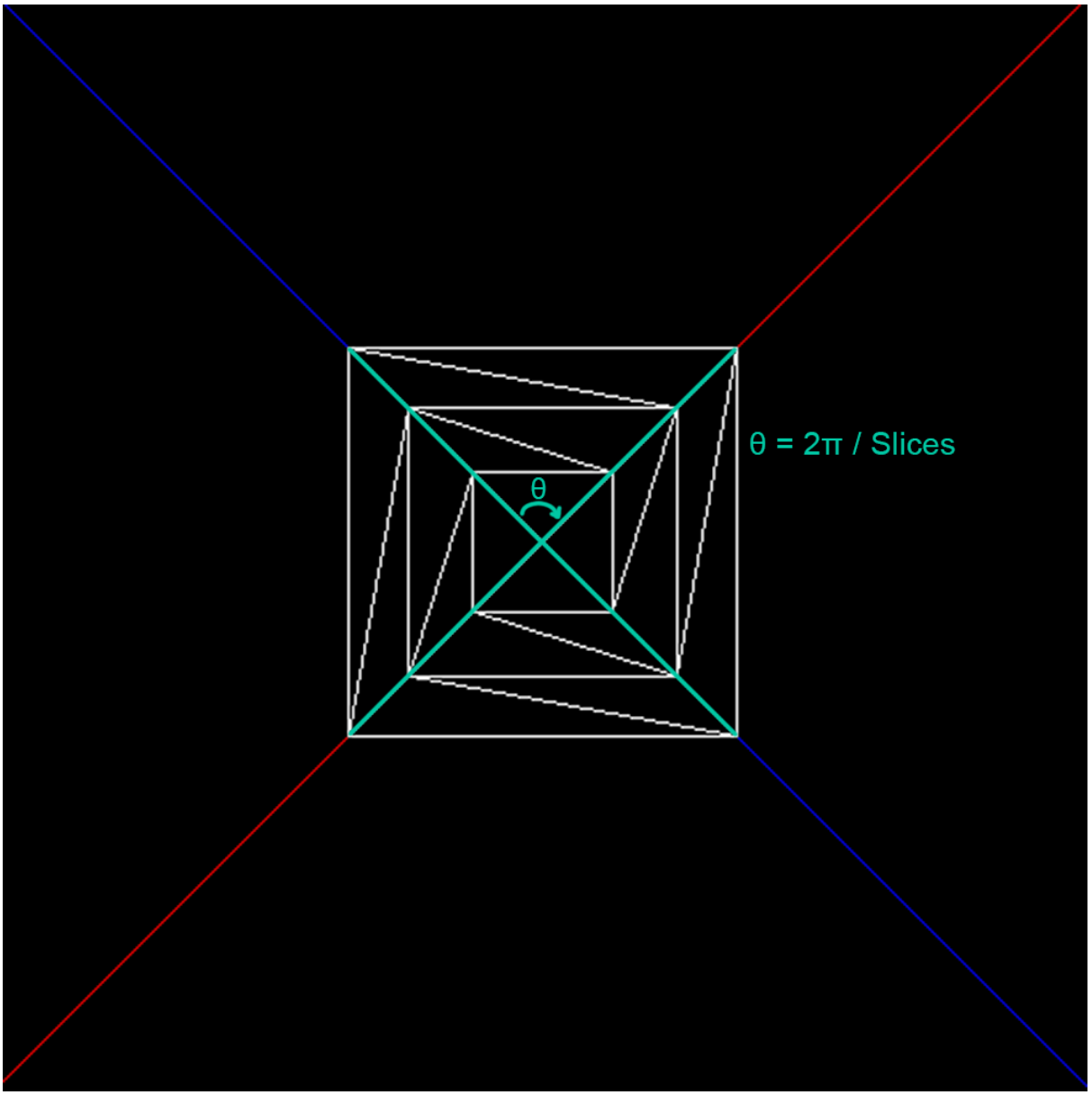
O **cone** tem a base centrada na origem e a altura segue o eixo do y.
Precisa do radius, da heigth, das slices e das stacks.
O número de vértices é

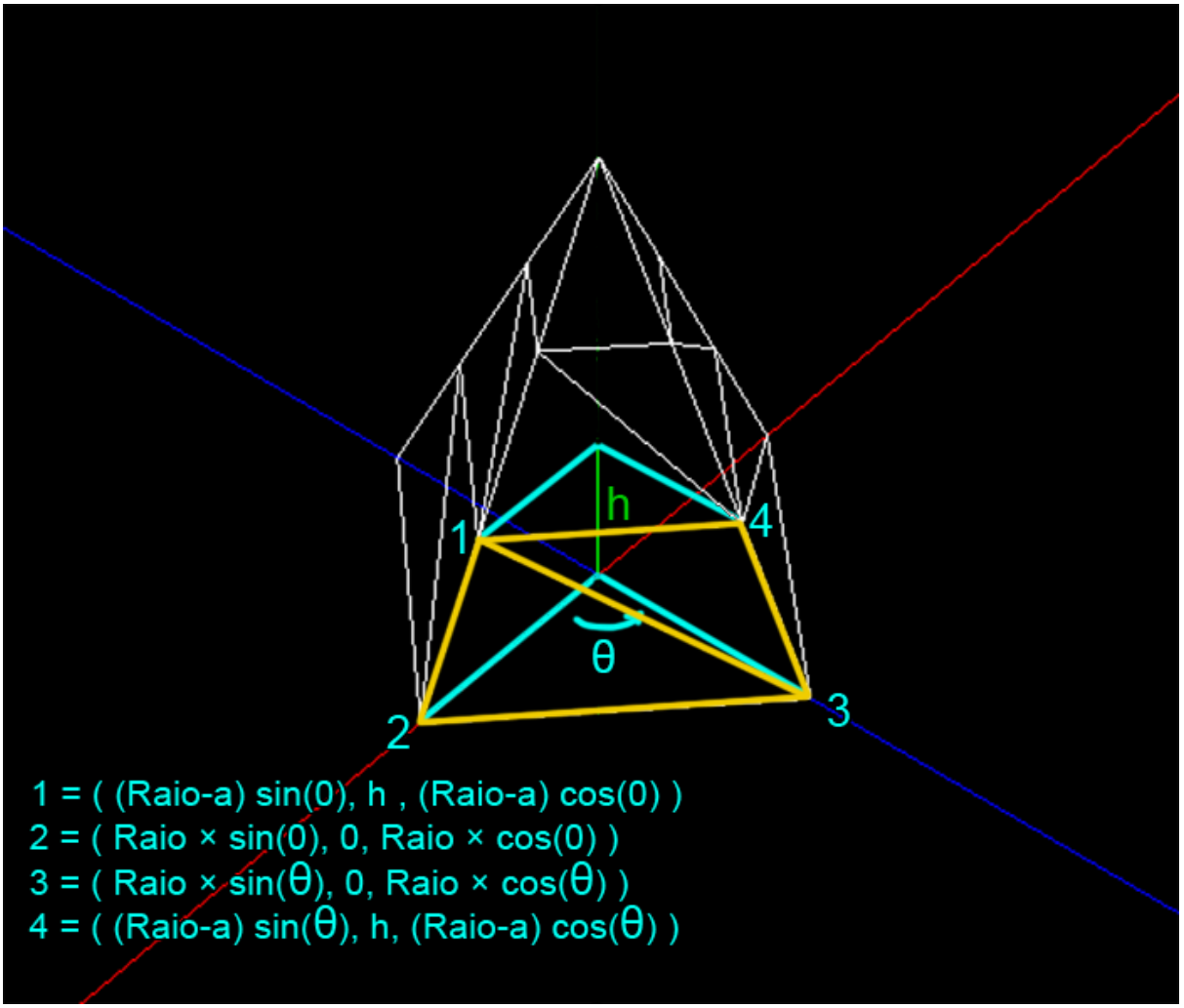
$$6 * stacks * slices + 3 * slices$$

A base precisa de tantos triângulos quantas slices têm, portanto 3 vértices por cada slice, depois por cada slice de cada stack precisamos de 2 triângulos, e portanto 6 vértices.
Ao contrário da esfera, o cone só precisa de mudar um ângulo que depende do número de slices, também precisamos da mudança de altura por stack e da mudança de raio por stack.
Depois por cada stack mudamos a altura e o raio e por cada slice escrevemos os 6 vértices.


Raio
a
a = Raio / Stacks


b = Altura / Stacks
Altura
b


θ = 2π / Slices
θ


h
1
4
θ
2
3
1 = ((Raio-a) sin(0), h , (Raio-a) cos(0))
2 = (Raio × sin(0), 0, Raio × cos(0))
3 = (Raio × sin(θ), 0, Raio × cos(θ))
4 = ((Raio-a) sin(θ), h, (Raio-a) cos(θ))

Engine

Ao iniciar o Engine podemos enviar como argumento o nome do ficheiro XML de onde são carregadas as definições da câmera e o nome dos ficheiros .3d dos modelos a ser gerados contidas nos nodes **<camera>** e **<models>**, respetivamente. A leitura destes ficheiros XML é feita pela função readXML(), utilizando a biblioteca tinyxml2 para fazer o parsing dos dados. De seguida é chamada a função readFile() para que esta leia os ficheiros .3d linha a linha e carregue todos os pontos para a memória na forma de um vetor de pontos.

Tendo os pontos todos guardados, passamos agora para a fase de desenho onde a cada renderScene a função draw() tira proveito da biblioteca glut para imprimir os modelos carregando três vértices de cada vez formando um triângulo.

Para modificar a forma de exibição, o **Engine** permite alternar entre diferentes modos de desenho, dependendo da tecla pressionada pelo utilizador:

- GL_FILL (Tecla F): triângulos são desenhados preenchidos.
- GL_POINT (Tecla P): apenas os pontos dos vértices são exibidos.
- GL_LINE (Tecla L): apenas as linhas das arestas.

Após a renderização inicial o Engine oferece ao utilizador um sistema de controlo de câmara permitindo a navegação em torno da origem. O movimento da câmera segue uma órbita em torno da origem, podendo este ser ajustado melhorando a experiência de visualização.

Definições do controlo da câmera

A câmera é inicialmente posicionada seguindo as definições especificadas no ficheiro XML correspondente ao modelo desenhado. Posteriormente, a interface permite que o utilizador mova a câmera livremente sobre uma órbita em torno do modelo, utilizando o rato. Esta funcionalidade é conseguida através das funções **processMouseButtons()** e **processMouseMotion(),** que processam a interação do utilizador com os botões e o movimento do rato.

A função **processMouseButtons()** é responsável por detectar quando os botões do rato são pressionados. O botão esquerdo (*GLUT_LEFT_BUTTON*) permite rodar a câmera enquanto que botão direito (*GLUT_RIGHT_BUTTON*) permite aproximar ou afastar a câmera Quando um desses botões é pressionada, a ação corresponde é registada e processada pela função **processMouseMotion()**.

A função **processMouseMotion()** é chamada sempre que o rato é movido enquanto um dos botões está a ser pressionado. Inicialmente calcula-se a diferença da posição inicial do rato quando o botão foi pressionado com a posição atual. Dependendo da ação em curso (rotação ou zoom), essa diferença é utilizada para atualizar os ângulos da câmera em torno do modelo.

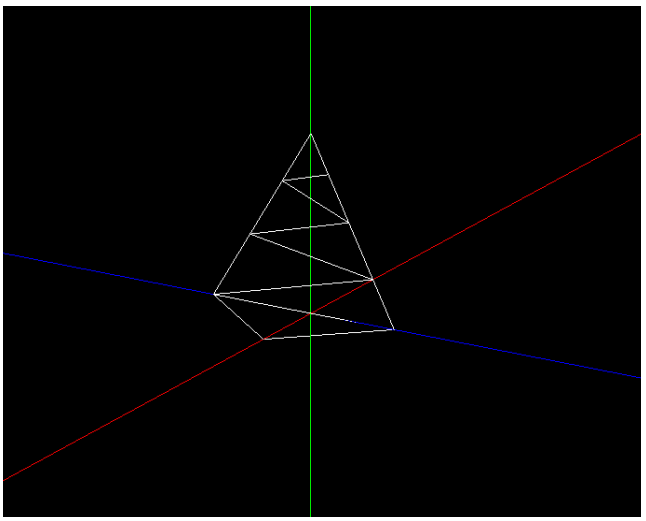
A posição da câmera é então calculada seguindo a equação:

$$(camX, camY, camZ) = (r * sin(\theta)cos(\phi)), rcos(\theta)cos(\phi), rsin(\phi))$$

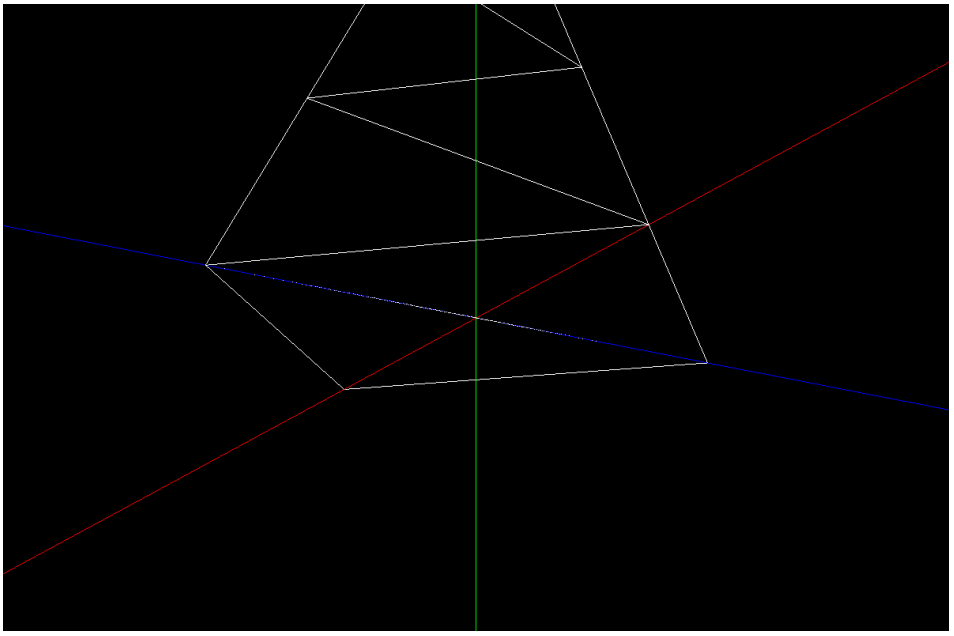
Desta forma, a câmera pode ser movida livremente em torno do modelo permitindo que o utilizador visualize o modelo de diferentes perspectivas.

Demos

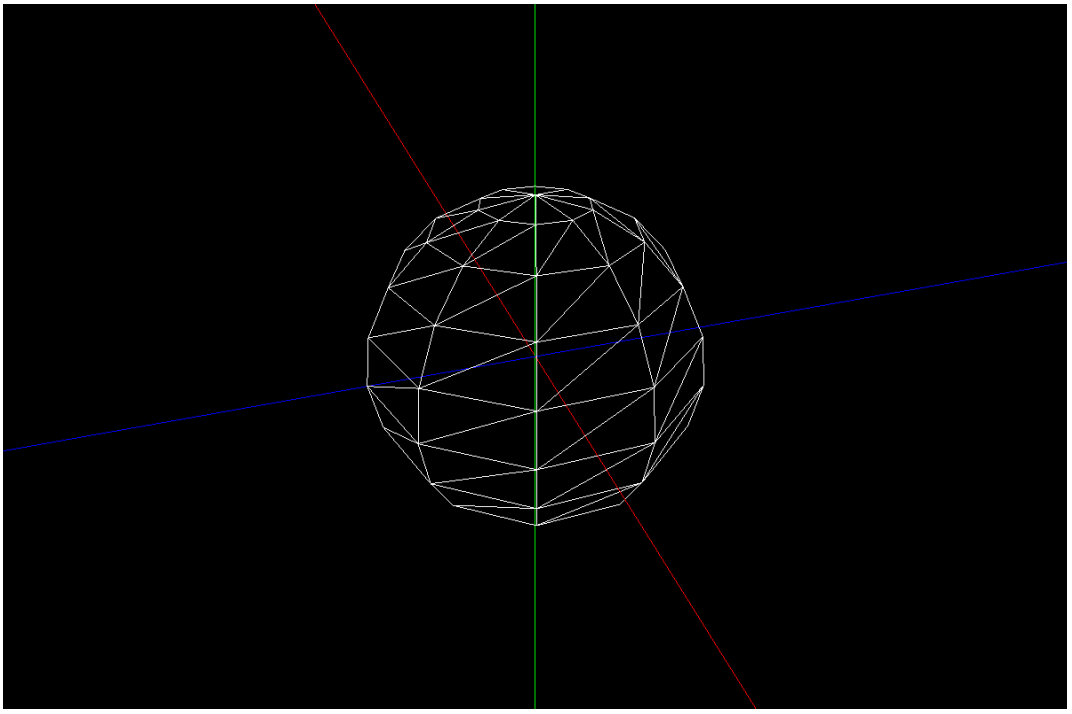
- Teste_1_1: Cone
  Raio = 1; Altura = 2; Slices = 4; Stacks = 3;
  Fov = 60; Near = 1; Far = 1000;
  Câmera (x,y,z) = (5,-2,3)



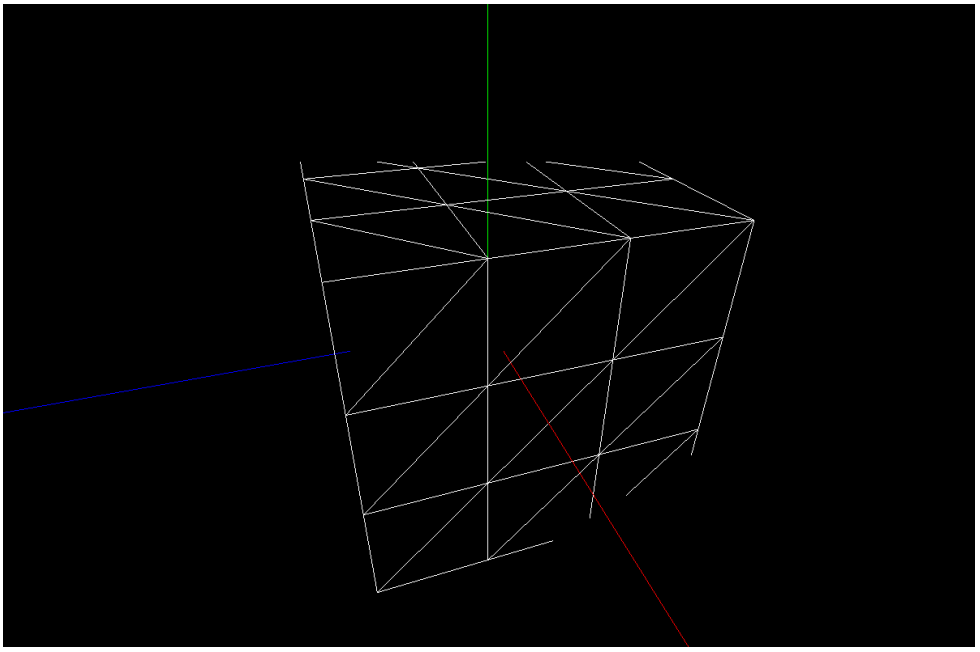
- Teste_1_2: Cone
  Raio = 1; Altura = 2; Slices = 4; Stacks = 3;
  Fov = 20; Near = 1; Far = 1000;
  Câmera (x,y,z) = (5,-2,3)



- Teste_1_3: Esfera
  Raio = 1; Slices = 10; Stacks = 10;
  Fov = 60; Near = 1; Far = 1000;
  Câmera (x,y,z) = (3,2,1)



- Teste_1_4: Box
  Comprimento = 2; Divisões = 3 ;
  Fov = 60; Near = 1; Far = 3.5;
  Câmera (x,y,z) = (3,2,1)



- Teste_1_5: Esfera + Plano
  Raio = 1; Slices = 10; Stacks = 10;
  Comprimento = 2; Divisões = 3 ;
  Fov = 60; Near = 1; Far =1000;
  Câmera (x,y,z) = (3,2,1)