Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase 1

Trabalho Prático

Grupo 9

João António Redondo Martins - a96215

João Pedro Antunes Gonçalves - a95019

Miguel de Sousa Braga - a97698

10 de março de 2023

Conteúdo

1 Introdução

Nesta primeira fase do trabalho prático de Computação Gráfica, foi-nos pedida a implementação de dois programas em C++. O primeiro, chamado *Generator*, seria responsável por gerar os vértices de uma dada figura, dados os argumentos que a caraterizam, através da linha de comandos. O resultado, na forma de uma lista de pontos escritos em ficheiro, seria lido por uma outra aplicação, a *Engine*, que os desenharia no ecrã. Para conhecer as configurações da câmara e o nome dos ficheiros a carregar, a *Engine* receberia ainda um ficheiro de configuração, escrito em XML.

2 Processo de desenvolvimento e decisões tomadas

2.1 *Generator*

Tal como referido acima, o programa *Generator* é responsável pela geração dos pontos que constituem as figuras a criar. De momento, a nossa aplicação é capaz de gerar 6 tipos de figuras:

- quadrado assente no plano $y = 0$ com x de lado e y divisões por lado
- cubo assente no plano $y = 0$ com x de lado e y divisões por lado
- cone assente no plano $y = 0$ com raio x, y stacks, z slices e w de altura
- esfera com raio x, y stacks e z slices centrada na origem
- cilindro de raio r, altura h e x slices centrado na origem
- torus de raio exterior re, raio interior ri, x slices e y stacks centrado na origem

2.1.1 Plano e Cubo

De forma a gerar os pontos que constituem cada um dos triângulos do grelha que forma o quadrado, definimos a função *generatePlane* que recebe uma direção e um ponto inicial e que devolve uma lista de triângulos a representar o quadrado gerado.

No caso do plano (quadrado), assumindo que o lado é l, o ponto inicial será $(-l/2, 0, -l/2)$, pois queremos que este seja centrado na origem, e a direção será 1 no sentido do eixo do x, 1 no sentido do eixo do z e 0 no sentido do eixo do y (pois queremos que este seja contruído na direção dos semieixos positivos do x e do z).

No caso do cubo, este é constituído por 6 faces, sendo cada uma delas um quadrado com lado n e m divisões. Entre cada uma das faces, distingue-se a orientação e o vértice inicial de geração do quadrado. Podemos assim reutilizar a função *generatePlane*, passando, a cada chamada da mesma, o ponto inicial e a direção adequadas.

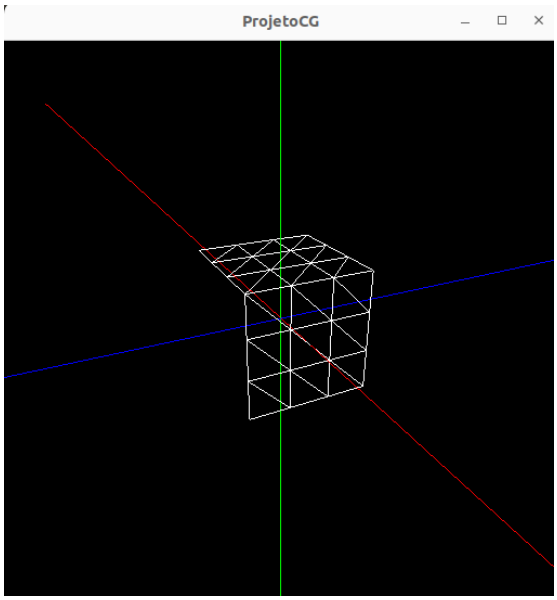


Figura 1: Faces do cubo

Por exemplo, na imagem acima podemos ver a face superior do cubo e uma face lateral do mesmo. Os eixos x, y e z encontram-se representados através das cores vermelho, verde e azul, respetivamente. Assumindo um cubo de lado 2, sabemos que a face lateral representada na figura tem como vértice superior esquerdo o ponto $(1, 1, 1)$. Para calcular os pontos desta face, passamos à função *generatePlane* a direção $(0, -1, -1)$. O quadrado será assim gerado, partindo do vértice $(1, 1, 1)$ tomando as direções negativas dos eixos y e z.

2.1.2 Esfera

Uma esfera é caraterizada por um determinado raio, um determinado número de stacks e um determinado número de slices.

Para gerar a esfera, começamos por definir um sistema de coordenadas polares, em que r é o raio da esfera, A é o ângulo com o eixo do z e B o ângulo com eixo do x. Com base em A e B, podemos caraterizar qualquer ponto na superfície da esfera com as coordenadas $(r * sin(A) * cos(B), r * sin(B), r * cos(A) * cos(B))$

Através do número de stacks, que nos dá o número de divisões verticais, calculamos o valor de B da seguinte forma: $pi/stacks$.

Da mesma forma, com o número de slices, podemos calcular o valor de A através de: pi/slices.

Começando a gerar a esfera de baixo para cima, começamos por desenhar os triângulos que incluem o vértice inferior, de coordenadas $(0, -r, 0)$. Os outros dois vértices podem ser gerados com o ângulo vertical igual a B e com o ângulo horizontal a variar entre 0 e 2 pi. Na imagem seguinte, podemos ver uma slice do cone, em que *alfa* e *beta* são ambos 30. Podemos assim calcular que a esfera terá 12 slices e 6 stacks.

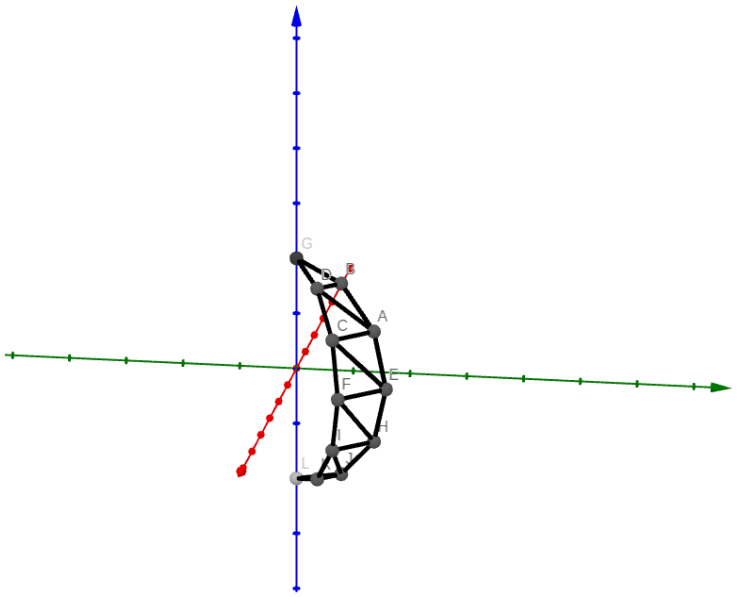


Figura 2: Esfera

2.1.3 Cone

No caso do cone, este recebe como parâmetros a altura, o raio da base, o número de fatias verticais (*slices*) e o número de fatias horizontais (*stacks*).

O cone é criado dividindo-se o círculo da base em fatias, iterando-se sobre as mesmas criando os triângulos da base. Para cada iteração deste ciclo também existe um ciclo interior para iterar sobre as stacks, que formam as geratrizes do cone.

Em cada iteração sobre as stacks é necessário criar dois triângulos, exceto na última iteração pois o cone converge num único ponto. Para os triângulos é preciso definir 4 pontos, com 2 alturas diferentes. A altura é definida a partir do número de stacks e da iteração do ciclo ($i * altura/stacks$). Já o raio também é importante para descobrir as coordenadas dos pontos a uma determinada altura no cone, visto que em cada nível usamos por base a circunferência paralela ao

plano da base com um determinado raio r, calculado em função do número de *stacks*. Na seguinte imagem ilustra-se o processo de desenho dos pontos de duas *stacks* do cone:

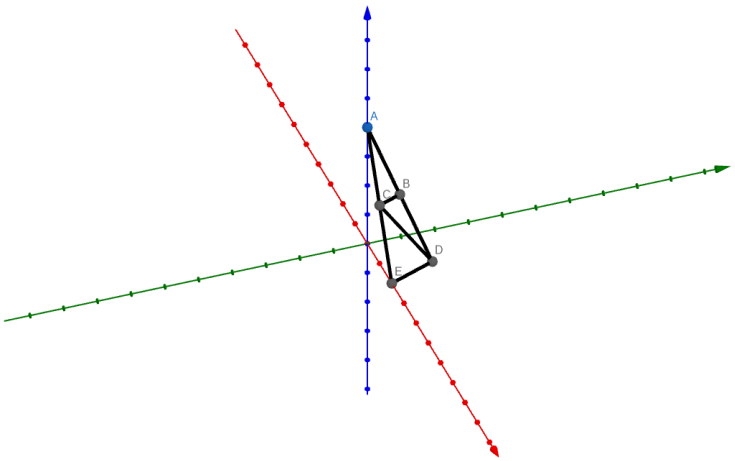


Figura 3: Cone

O eixo dos z está representado a vermelho, o eixo dos x a verde e o eixo dos y a azul. Neste caso, é representada uma fatia de um cone com 2 stacks, e cujo raio da base é 2 unidades. A partir do raio da base e do número de stacks, podemos calcular que o raio da circunferência à qual pertencem os pontos B e C é 1. Para cada altura, fazendo variar o *alfa* entre 0 e $360^{\underline{o}}$ encontramos os pontos que formam as geratrizes do cone.

Neste exemplo, os pontos C e E pertencem à geratriz que faz um ângulo de $0^{\underline{o}}$ com o eixo dos z. Já os pontos B e D formam uma geratriz que forma um ângulo de $45^{\underline{o}}$ com o eixo dos z. A partir destas informações criamos a tabela seguinte:

Ponto	**Coordenadas**
A	$(0, h, 0)$
B	$(1 * sin(0 + delta), 2, r * cos(0 + delta)$
C	$(1 * sin(0), 2, 1 * cos(0))$
D	$(2 * sin(0 + delta), 0, 2 * cos(0 + delta))$
E	$(2 * sin(0), 0, 2 * cos(0))$

Tabela 1: Coordenadas dos pontos da figura 3 (delta=$45^{\underline{o}}$)

2.1.4 Cilindro

Para além das primitivas referidas, também adicionamos outras primtivas, como é o caso do cilindro.

No caso do cilindro, este recebe como parâmetros o raio, a altura e o número de fatias verticais (*slices*). Para desenhar o cilindro usamos um ciclo para iterar sobre as slices. Em cada iteração desenhamos o triângulo da face inferior, da face superior e os dois triângulos da face lateral. Como o cilindro é centrado na origem, os vértices dos centros das faces inferior e superior encontram-se no eixo dos y e, assim, a sua coordenada no eixo dos y é metade da altura do cilindro, no caso da face superior, e o simétrico da metade da altura do cilindro, no caso da face inferior. Para os restantes vértices utilizamos coordenadas que dependem do ângulo *alfa* (que representa o ângulo relativamente ao semieixo positivo dos z, como podemos ver na seguinte imagem:

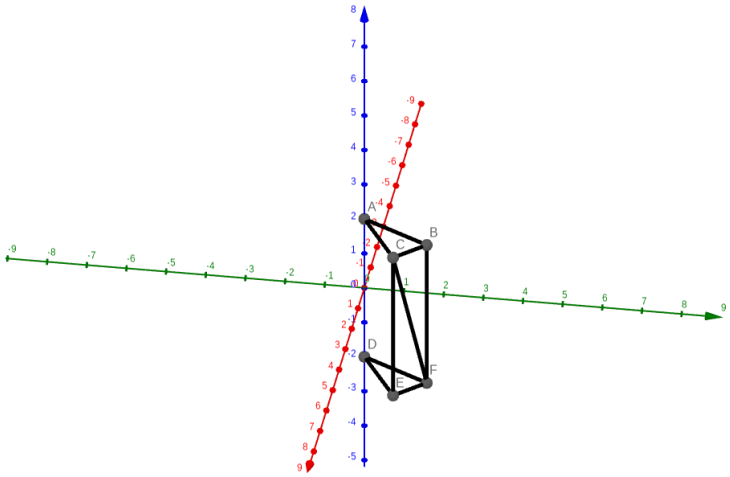


Figura 4: Cilindro

Neste exemplo de uma fatia do cilindro, assumindo que o eixo dos z é o eixo vermelho, e que o eixo verde é o eixo dos x e assumindo um determinado ângulo *alfa*, uma altura h e um raio r, podemos dizer que os pontos tem as seguintes coordenadas:

Ponto	**Coordenadas**
A	$(0, h/2, 0)$
B	$(r * sin(alfa + delta), h/2, r * cos(alfa + delta))$
C	$(r * sin(alfa), h/2, r * cos(alfa))$
D	$(0, -h/2, 0)$
E	$(r * sin(alfa), -h/2, r * cos(alfa))$
F	$(r * sin(alfa + delta), -h/2, r * cos(alfa + delta))$

Tabela 2: Coordenadas dos pontos da figura 4 (delta resulta da divsão de 360 pelo número de slices)

2.1.5 Torus

Uma outra primitiva adicionada foi a de criação de um *torus*. Esta figura é caraterizada por 4 valores: raio interior, raio exterior, número de *slices* (divisões verticais) e número de *stacks* (divisões horizontais).

Um torus pode ser gerado através da sobreposição de vários círculos com o mesmo raio - $(raioExterior - raioInterior)/2$ - cada um deles rodado em um ângulo *alfa* em torno do eixo dos z. Deste modo, o processo de geração dos vértices e das faces é muito semelhante ao da esfera. Começamos por iterar por cada uma das divisões verticais, e, para cada uma delas, geramos os triângulos que compõem as divisões horizontais. Cada uma das divisões horizontais será um quadrilátero, pelo que teremos de definir os 4 pontos que definem os vértices desse quadrilátero (que depois será dividido em 2 triângulos). De modo a calcular esses pontos, primeiro calculamos o centro de cada uma das circunferências às quais pertencem as duas arestas laterais - pontos A e B, como podemos ver na seguinte imagem:

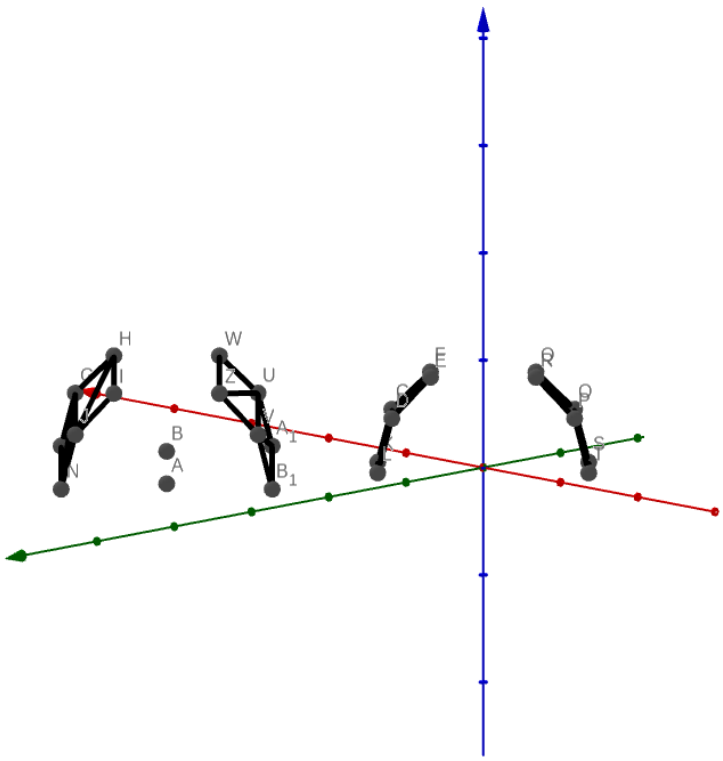


Figura 5: Torus

Assumindo um determinado ângulo *alfa* (ângulo com o eixo dos z) e um determinado ângulo *beta* (altura relativamente ao eixo dos x), e calculando

os próximos valores para *alfa* e *beta* (*nextAlfa* e *nextBeta*), podemos calcular (com recurso a coordenadas polares, os pontos respetivos pertencentes à esfera centrada na origem e com o mesmo raio da secção de corte do torus. Por fim, os pontos localizados no torus, resultam do deslocamento dos pontos desta esfera segundo o vetor definido pelos pontos A e B. Isto é válido, porque estes pontos têm exatamente os mesmos ângulos *alfa* e *beta* do que os pontos correspondentes na esfera centrada na origem.

2.1.6 Escrita dos dados em ficheiro

Depois de gerados, no *Generator*, os pontos constituíntes das figuras, estes são escritos em ficheiro com um formato próprio, para serem mais tarde lidos pelo programa *Engine*.

Numa primeira abordagem, os pontos seriam escritos no ficheiro em binário, numa lista. No entanto, decidimos optar por um outro formato, o formato obj do blender (um formato textual), dando maior flexibilidade na leitura de novos modelos, permitindo até mesmo importar outros modelos que não sejam gerados por nós.

No formato obj do blender, os vértices são apresentados numa lista sem uma ordem predefinida. Essa ordem define, contudo, os índices que mais tarde aparecerão como pontos de cada triângulo gerado. Uma linha que define um vértice começa pelo caracter 'v', seguindo-se as componentes x, y e z do mesmo separadas por um espaço. Por exemplo, a linha 'v 1 1 1' indica que estamos perante o ponto $(1, 1, 1)$.

As linhas que definem as faces da figura começam com a letra f à qual se seguem os índices dos vértices que definem essa mesma face. Vale mencionar que o formato do ficheiro não obriga a que as faces sejam triângulos. No entanto, na nossa aplicação forçamos a que isso aconteça, pelo que é necessário que antes de gerar o *obj*, as faces sejam transformadas em triângulos. É possível ainda definir, para cada vértice de cada face, o vetor normal a essa face, que define a ordem pela qual os pontos são colocados no triângulo no *glut* (e que tem impacto na orientação dessa face). Para cada vértice, esse vetor normal é indicado através de um índice da lista de vetores normais definidos no ficheiro anteriormente. É importante também referir que este campo é opcional. Se não forem definidos os vetores normais para cada uma das faces, o programa mantém a ordem dos pontos escritos no ficheiro.

Por fim, os vetores normais são definidos de forma semelhante aos vértices, com a exceção de começarem com os carateres 'vn'. A posição de cada vetor, escrito na forma 'vn vx vy vz', define o índice utilizado nas faces da figura. Tanto nos vértices como nos vetores normais do formato obj, os índices começam em 1 e não em 0, como habitual. No caso dos ficheiros gerados pelo *Generator*, os índices começam em 0, pelo que é necessário ter em conta essa situação.

Como exemplo, na figura seguinte podemos ver o resultado de gerar um plano (quadrado) de lado 2 e com 2 divisões em cada um dos lados.

```
v -1.000000 0.000000 -1.000000
v 0.000000 0.000000 0.000000
v 0.000000 0.000000 -1.000000
v -1.000000 0.000000 0.000000
v 1.000000 0.000000 0.000000
v 1.000000 0.000000 -1.000000
v 0.000000 0.000000 1.000000
v -1.000000 0.000000 1.000000
v 1.000000 0.000000 1.000000
f 0 1 2
f 0 3 1
f 2 4 5
f 2 1 4
f 3 6 1
f 3 7 6
f 1 8 4
f 1 6 8
```

Figura 6: Exemplo de serialização

2.1.7 Cálculo dos índices para serialização

Para cada figura gerada pelo *Generator*, de modo a não repetir a definição de vértices comuns entre triângulos, utilizamos índices, poupando assim a memória utilizada. Para isso, recorremos a um map, tanto na geração como na criação de figuras, de modo a guardar os índices dos pontos (vértices) conforme eles vão aparecendo na figura. Esta é uma solução temporária visto que apresenta alguns problemas de eficiência, devido principalmente à falta de localidade dos acessos à memória, pelo que o objetivo para as próximas fases passa por aumentar o desempenho tanto na geração dos pontos e faces como na leitura dos mesmos, a partir de ficheiro.

2.2 *Engine*

2.2.1 Leitura do ficheiro de configuração

O ficheiro de configuração, escrito em XML, contêm as definições da câmara e as referências para os ficheiros com os triângulos a carregar, gerados anteriormente pelo programa *Generator*.

O parser de XML utilizado foi o *tinyxml2*, disponível em `https://github.com/leethomason/tinyxml2`. O código disponibilizado (.h e .cpp) foi importado para uma pasta *libraries*, criada no projeto.

2.2.2 Leitura dos ficheiros de dados

O processo de leitura dos dados representativos dos pontos e dos triângulos escritos em ficheiro é o processo inverso do descrito na alínea 2.1.6. Os dados são lidos, portanto, em modo texto. Neste momento, é possível ler 3 tipos de linhas diferentes do ficheiro, tal como descrito na secção 2.1.6. A primeira, começada por um v, define os vértices da figura, segundo o formato 'v x y z'. A segunda define os vetores normais às faces, para efeitos de orientação. Tem o formato 'vn

vx vy vz'. Por fim, as linhas começadas por f definem as faces (triângulos) da figura que pretendemos gerar. O formato é 'f i1(/vn) i2(/vn) i3(/vn)', sendo que a componente i define o índice do vértice definido anteriormente e a componente vn define o índice do vetor normal a essa face.

Na imagem seguinte, podemos ver o exemplo de um ficheiro gerado pelo blender, que também é suportado no nosso trabalho:

```
v 2.638928 2.684652 -6.787258
v 2.638928 0.684652 -6.787258
v 2.638928 2.684652 -4.787258
v 2.638928 0.684652 -4.787258
v 0.638928 2.684652 -6.787258
v 0.638928 0.684652 -6.787258
v 0.638928 2.684652 -4.787258
v 0.638928 0.684652 -4.787258
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
f 5/1/1 3/2/1 1/3/1
f 3/2/2 8/4/2 4/5/2
f 7/6/3 6/7/3 8/8/3
f 2/9/4 8/10/4 6/11/4
f 1/3/5 4/5/5 2/9/5
f 5/12/6 2/9/6 6/7/6
f 5/1/1 7/13/1 3/2/1
f 3/2/2 7/14/2 8/4/2
f 7/6/3 5/12/3 6/7/3
f 2/9/4 4/5/4 8/10/4
f 1/3/5 3/2/5 4/5/5
f 5/12/6 1/3/6 2/9/6
```

Figura 7: Ficheiro gerado pelo blender

De modo a pré-estabelecer a ordem dos pontos para desenho da face com a orientação desejada recorremos a uma função que calcula o produto vetorial entre dois vetores. Os dois vetores que são passados à função são calculados com base em duas combinações de pontos do triângulo que define a face. Como exemplo, se tivermos os pontos P1, P2 e P3 a definir a face f, calculamos os vetores $P1P2$ e $P1P3$, por exemplo. Se neste caso, o produto vetorial $P1P2$ x $P1P3$ for colinear com o vetor normal e tiver o mesmo sentido, pela regra da mão direita sabemos que a ordem dos pontos é P1 P2 P3. Caso contrário, se o vetor calculado for colinear com o vetor normal, mas tiver o sentido oposto, então a ordem passa a ser P1 P3 P2.

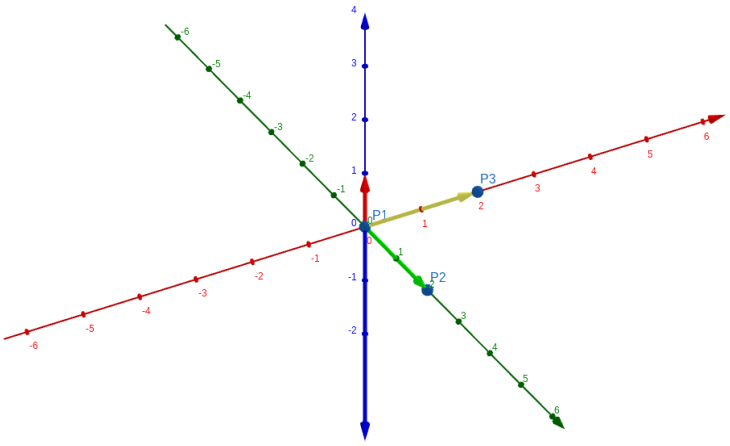


Figura 8: Exemplo de produto vetorial

Por exemplo, na figura acima, onde temos representados os eixos x (a vermelho), y (a verde) e z (a azul), podemos ver os pontos $P1(0,0,0)$, $P2(0,2,0)$ e $P3(2,0,0)$, com os quais definimos o vetor $u = P2 - P1$ (a verde), com coordenadas $(0,2,0)$ e o vetor $v = P3 - P1$ (a amarelo), com coordenadas $(2,0,0)$. O resultado do produto vetorial destes dois vetores (u x v) resulta no vetor $(0,0,-4)$ (a azul). Este vetor não tem o mesmo sentido do vetor normal, que é $(0,0,1)$ (a vermelho), pelo que a ordem correta dos pontos no triângulo será P1, P3, P2.

Lidos os vértices e calculados os triângulos que definem uma figura, criámos dois vetores, um com os pontos e outro com os triângulos a desenhar. São estas estruturas de dados que passamos à função que desenha a figura. Nesta fase, os triângulos ainda não são representados por 3 pontos (por uma questão de eficiência), mas sim por 3 índices.

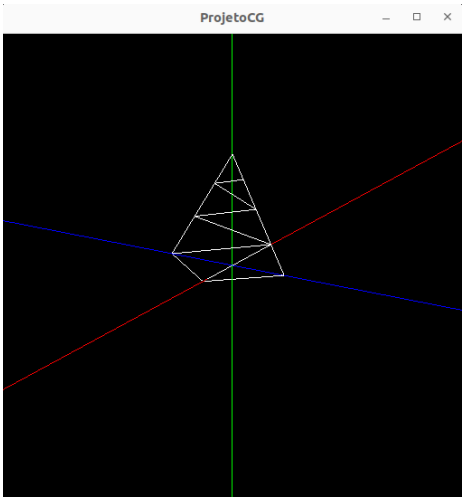
Nota: Os ficheiros com os dados das figuras devem estar numa pasta *figures* incluída no repositório.

2.2.3 Câmara

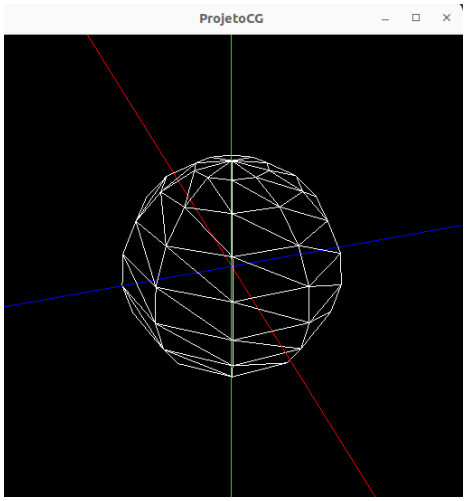
Nesta fase do trabalho, é possível utilizar a câmara em modo explorador, o que nos é útil para visualizar os diferentes lados das figuras geradas. O modo explorador permite que a câmara se mova numa superfície esférica de raio r, com o centro no ponto *lookAt*, disponibilizado no XML. Para movimentar-se na esfera, são utilizadas as setas (*arrows*) do teclado. Para além disso, é possível aumentar e diminuir o deslocamento pela superfície esférica e ainda aumentar e diminuir o raio da esfera por onde a câmara de movimenta.

3 Resultados obtidos

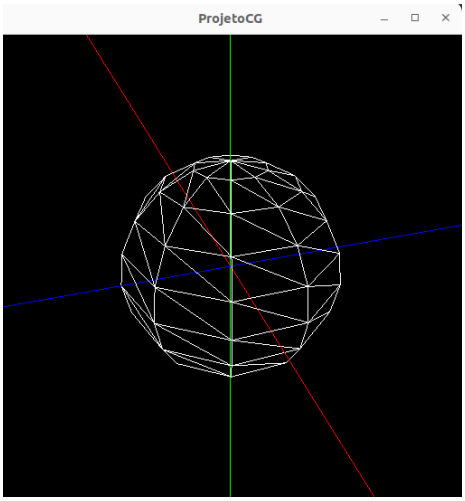
Com base nos ficheiros XML de teste e em outros testes por nós criados, obtivemos os seguintes resultados:



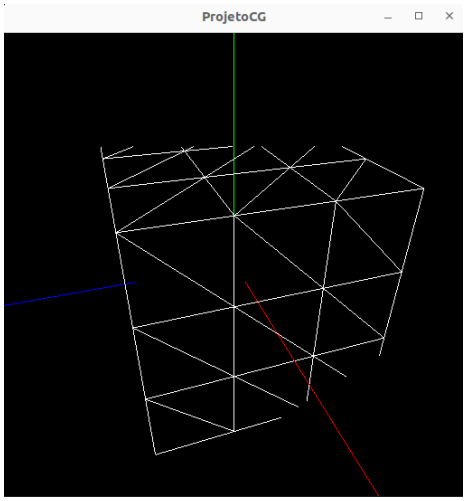
(a) Cone de raio 1, altura 2, 4 slices e 3 stacks



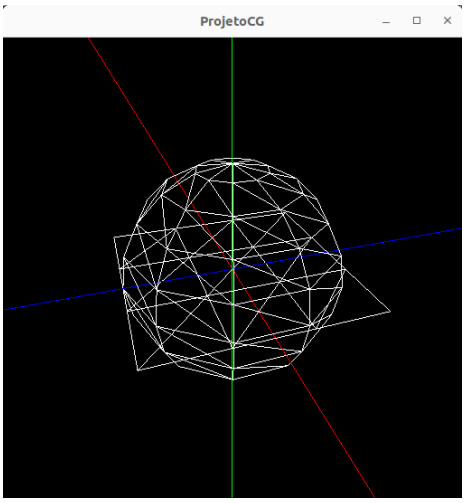
(b) Cone de raio 1, altura 2, 4 slices e 3 stacks com fov de 20



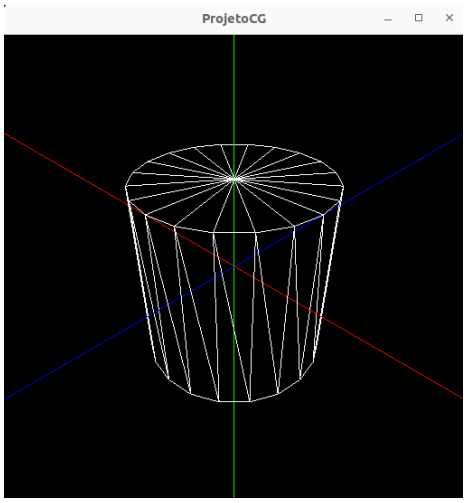
(c) Esfera com raio 1, 10 slices e 10 stacks



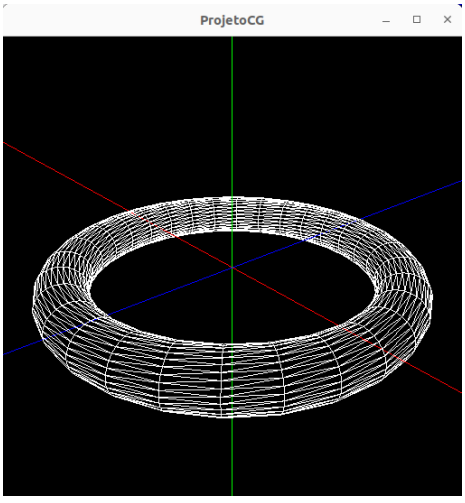
(d) Cubo com lado 2 e grid 3x3



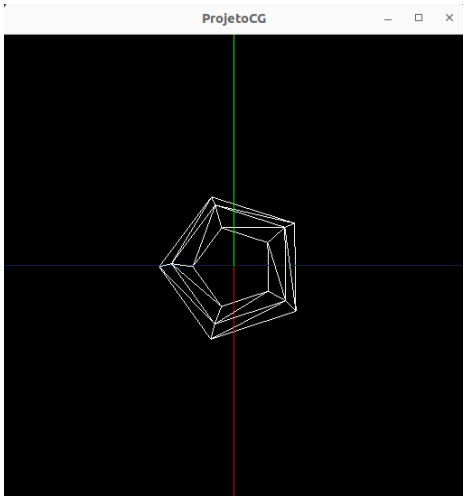
(e) Plano de lado 2 e grid 3x3 + esfera de raio 1, 10 slices e 10 stacks



(f) Cilindro de raio 2, 4 de altura e 10 slices

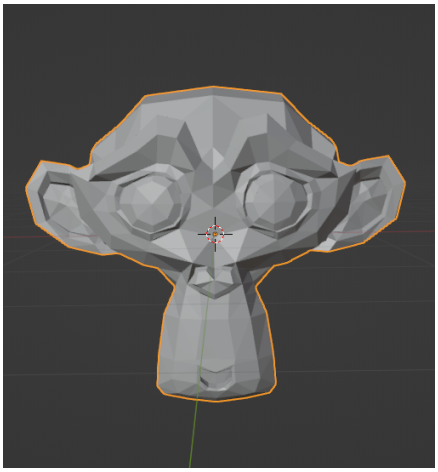


(g) Torus de raio exterior 4, raio interior 3, 30 slices e 30 stacks

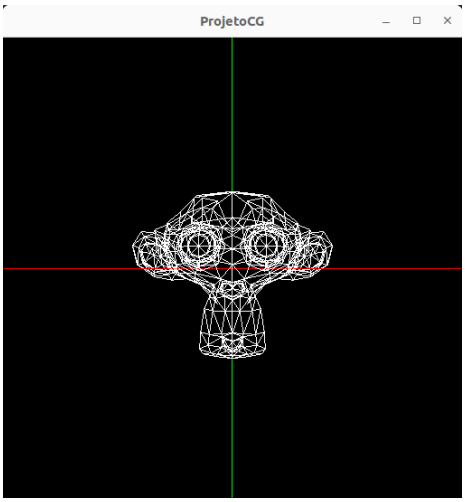


(h) Torus de raio exterior 2, raio interior 1, 5 slices e 30 stacks

De modo a testar a renderização dos objetos criados pelo blender, primeiro temos de os exportar num formato baseado apenas em triângulos. Para isso, utilizamos a opção do blender "Triangulate faces":



(i) Objeto gerado pelo blender



(j) Resultado na engine

4 Conclusão e balanço da primeira fase

Em suma, nesta primeira fase do trabalho começamos por nos debruçar na componente de geração de possíveis modelos para a cena que iremos criar num ambiente OpenGL. Para além das figuras propostas, procuramos gerar outras (cilindro e torus) porque acreditamos que estas nos poderão vir a ser úteis nas próximas fases do trabalho, adicionando complexidade desejável à nossa cena. Para além disso, de modo a aumentar ainda mais a flexibilidade da nossa *Engine*, seguimos o formato dos ficheiros obj gerados pelo blender, com o objetivo de poder vir a adicionar outro tipo de figuras no futuro, se desejado. O primeiro balanço que fazemos do trabalho realizado é, por isso, muito positivo, visto termos cumprido grande parte dos objetivos para esta fase. Para as próximas fases, um dos primeiros objetivos a atingir, passa por implementar outros modos de câmara, como a câmara em primeira pessoa.