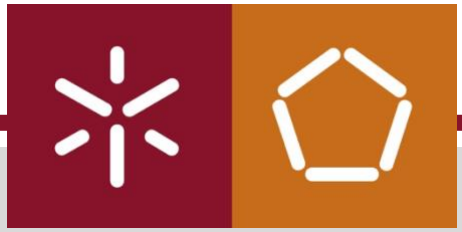
UNIVERSIDADE DO MINHO



Mestrado Integrado em Engenharia Informática

PROJETO DE COMPUTAÇÃO GRÁFICA
FASE 1 : PRIMITIVAS GRÁFICAS

março, 2020




GRUPO 10
Carolina Cunha, A80142 | Hugo Faria, A81283 | João Diogo Mota, A80791 | Rodolfo Silva, A81716

ÍNDICE

ÍNDICE DE FIGURAS

BREVE DESCRIÇÃO DO ENUNCIADO PROPOSTO

Prevê-se, com este trabalho, o desenvolvimento de um mecanismo 3D baseado em mini gráficos de cenas e fornecer exemplos de uso que mostrem o seu potencial. Este trabalho será dividido em quatro fases distintas, de modo a facilitar a elaboração do mesmo. A conceção e desenvolvimento do trabalho seguirá uma abordagem suportada pela ferramenta *OpenGL* e a linguagem C++.

A primeira fase do projeto consistirá na construção de algumas primitivas gráficas, nomeadamente um plano, uma caixa, uma esfera e um cone. Deste modo, são requeridas duas aplicações, sendo que a primeira consiste na criação de um gerador de ficheiros com as informações dos modelos (onde será guardada a informação relativa aos vértices de cada modelo) e a segunda na elaboração de um motor que permita ler o ficheiro configurado, escrito em XML, e apresentar os respetivos modelos em três dimensões (3D).

PRIMITIVAS GRÁFICAS

PLANO

A representação de um plano quadrado através de OpenGL requer apenas o desenho de dois triângulos, pelo que é apenas necessário ter conhecimento sobre o comprimento lateral do plano. De modo a obter o plano pretendido, vão ser originados dois triângulos retângulos (Figura 1), cujos vértices das hipotenusas irão coincidir, tendo por isso as mesmas coordenadas. Por este motivo, só será preciso identificar quatro pontos de forma a construir os triângulos.

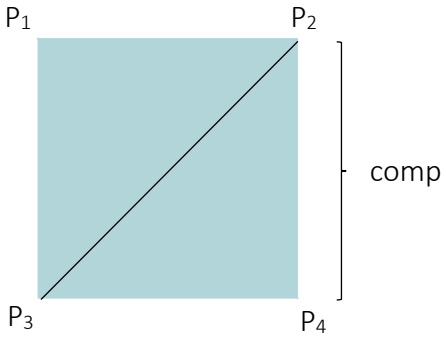


Figura 1: Representação do plano

O primeiro passo para a concretização da construção do plano passou por identificar a posição relativa de cada vértice no sistema de coordenadas cartesiano. Desta forma, o vértice P_1 encontra-se no lado negativo do eixo dos *xx* e no lado negativo do eixo dos *zz*; o vértice P_2 situa-se também no lado negativo do eixo dos *zz* mas tem coordenada x positiva; P_3, contrariamente ao vértice P_2, encontra-se no lado positivo do eixo dos *zz*, no entanto, no lado negativo do eixo dos *xx*. Por fim, o vértice P_4 tem ambas as coordenadas positivas. Uma vez que o plano desejado é o **plano XZ**, a coordenada y de todos os vértices é 0.

Tendo em consideração que se pretende centralizar o plano na origem do referencial O(0,0), os seus vértices irão estar situados em coordenadas de valor c = comp/2.

Assim, as coordenadas de cada vértice são as seguintes:

P_1 (-c, 0, -c); P_3 (-c, 0, c);

P_2 (c, 0, -c); P_4 (c, 0, c);

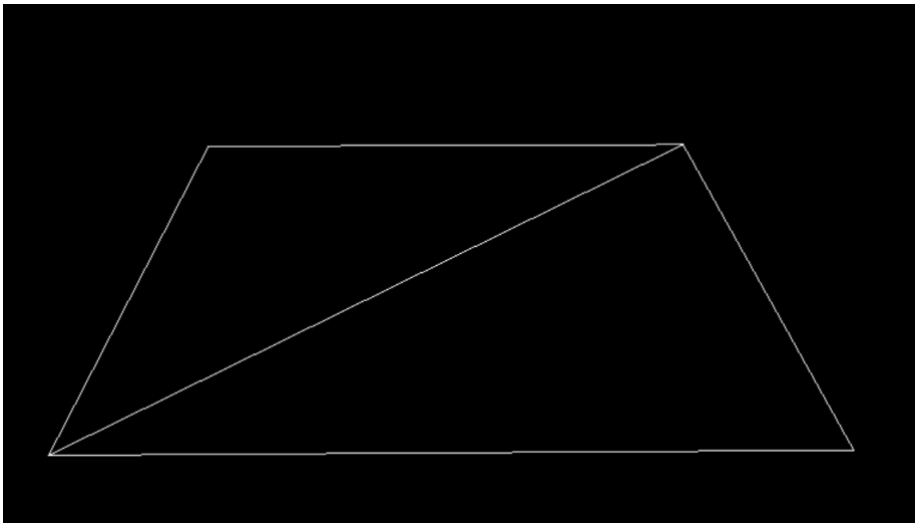


Figura 2: Representação do plano em OpenGL

De modo a tornar possível o desenho da caixa, é crucial ter conhecimento sobre os seguintes parâmetros:

i. Comprimento da caixa;
ii. Largura da caixa;
iii. Altura da caixa;
iv. Número de divisões de cada face (opcional).

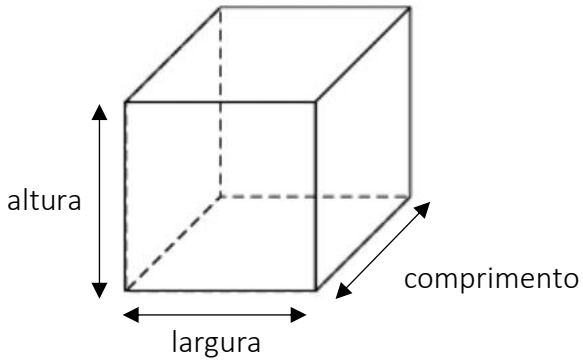


Figura 3: Representação da caixa

Cada face da caixa irá ser representada por triângulos. O número de triângulos de cada face depende do número de divisões passado como parâmetro, pelo que o número de triângulos é 2 x (divisões)^2.

Dado que a caixa estará centrada na origem O(0,0,0), as variáveis xA,yA e zA representam as coordenadas máximas em módulo de face, uma vez que correspondem a metade do valor de cada medida da caixa.

```
xA = largura / 2;
yA = altura / 2;
zA = comprimento / 2;
```

Nas variáveis sliceX, sliceY, sliceZ, estarão armazenadas as dimensões dos triângulos por cada divisão.

```
sliceX = xA / divisões;
sliceY = yA / divisões;
sliceZ = zA / divisões;
```

Assim, a utilização de dois ciclos permite construir esses mesmos triângulos nas direções vertical e horizontal, possibilitando a construção da caixa.

Tendo por exemplo a face frontal de uma caixa com duas divisões:

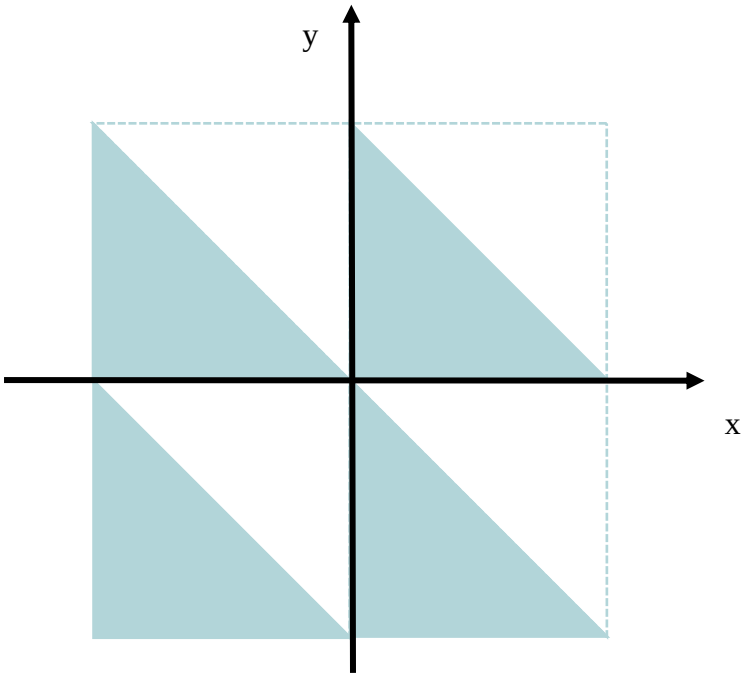
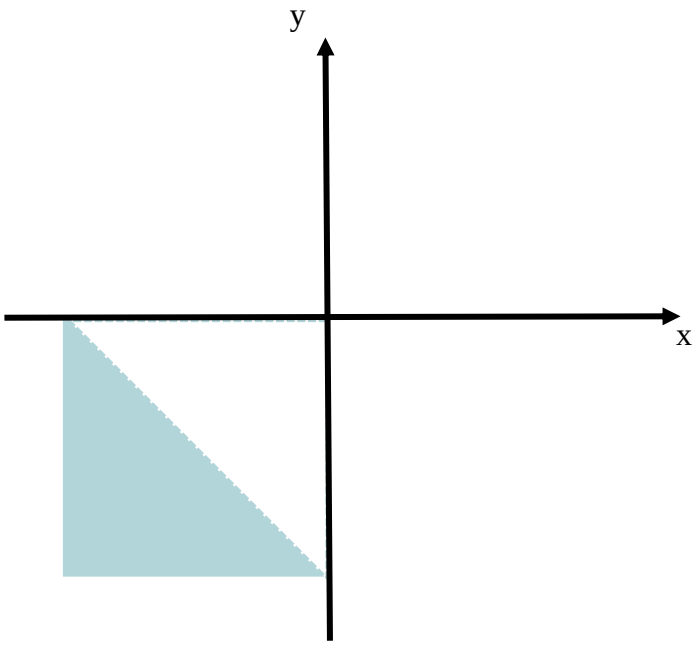
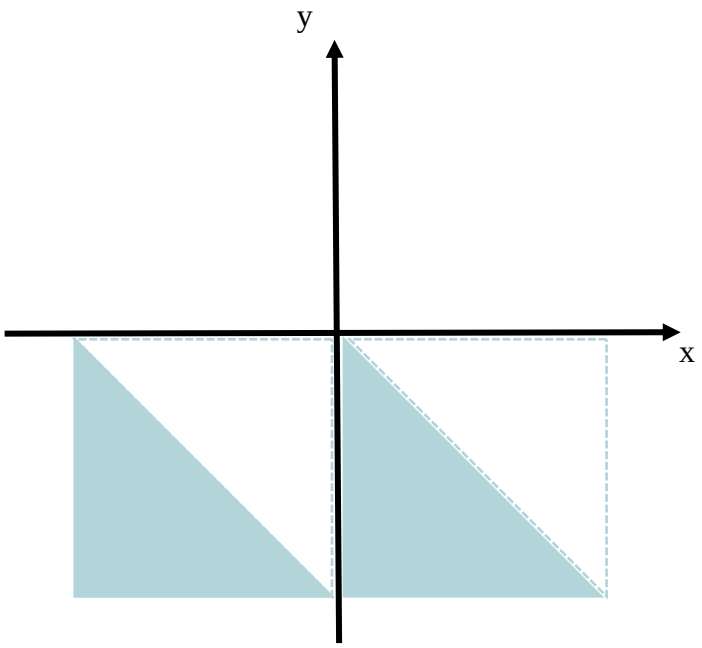


Figura 4: Exemplo de face frontal de caixa com 2 divisões

A primeira iteração dos ciclos permite a seguinte construção:

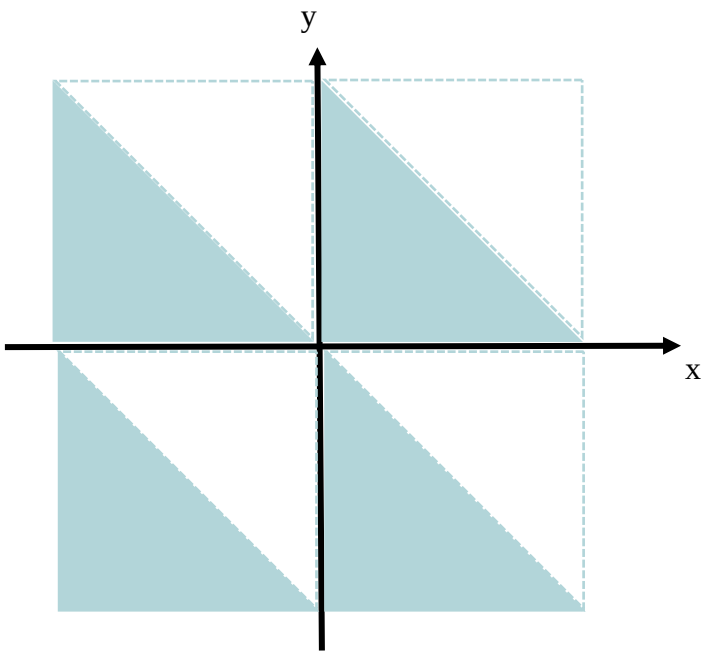


A segunda iteração do ciclo interior leva à construção de um segundo triângulo:



Deste modo, verifica-se que, para a face frontal, o ciclo interior é responsável por construir os triângulos nas faces em linhas horizontais.

Uma vez que a variável do ciclo alcançou o número de divisões, é incrementada a iteração do primeiro ciclo. Assim, a segunda iteração do ciclo externo será:



Posto isto, verifica-se que, para a face frontal, o ciclo externo está encarregue de construir os triângulos nas faces em linhas verticais.

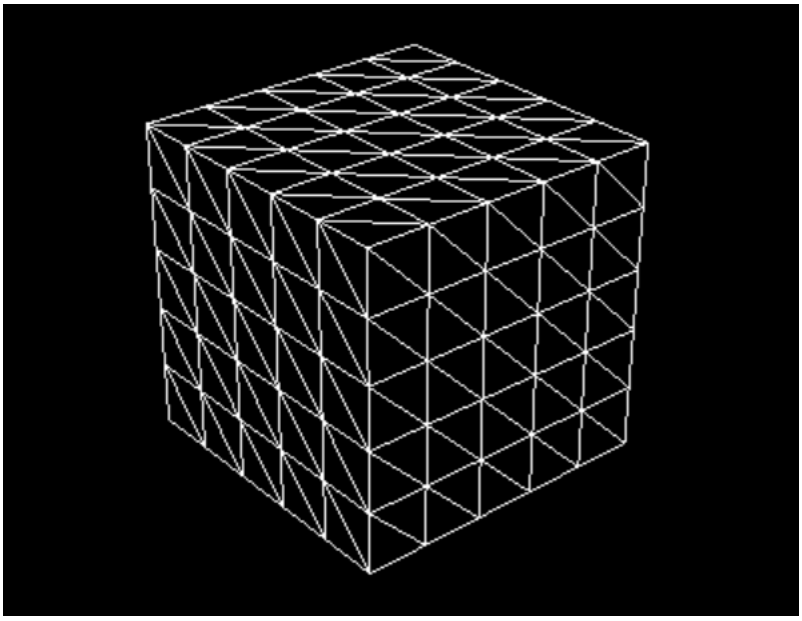


Figura 5: Representação da Caixa em OpenGL

Para que seja possível a construção da esfera, é necessário fornecer os dados sobre o seu raio, número de fatias da esfera e o valor de divisões a realizar.

Dado que uma esfera é uma superfície fechada de três dimensões (3D) onde todos os pontos que nela se encontram estão a uma mesma distância (raio) de um dado ponto, a sua equação na origem é $x_2 + y_2 + z_2 = r_2$.

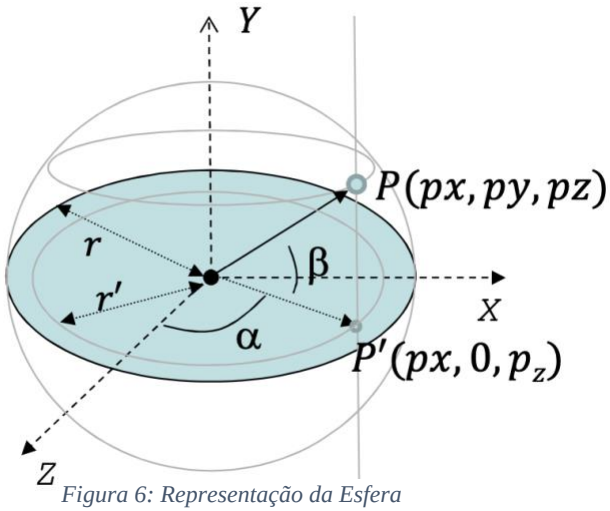


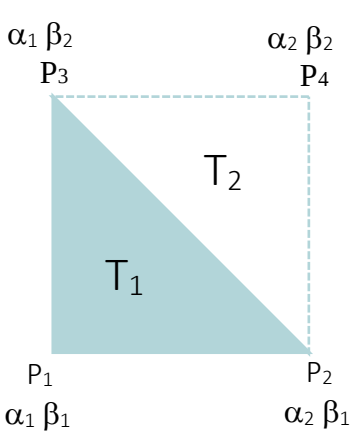
Figura 6: Representação da Esfera

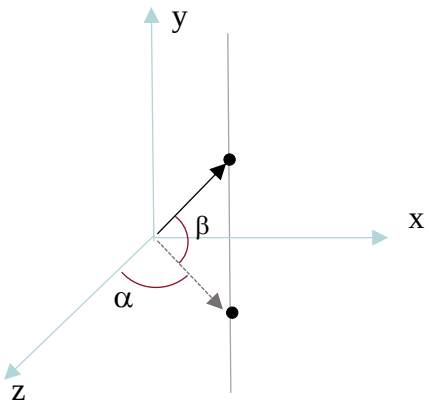
De modo a concretizar o desenvolvimento da esfera, é indispensável a utilização de coordenadas esféricas.

Através do teorema de Pitágoras e sabendo que r' = r x cos(β), obtém-se as seguintes coordenadas:

Coordenadas Esféricas
(α,β,r)
-90 < β < 90

Coordenadas Cartesianas
px = r x cos(β) x sin(α);
py = r x sin(β);
pz = r x cos(β) x cos(α);





Considerando $P_1(x,y,z)$; $P_2(x_2,y_2,z_2)$; $P_3(x_3,y_3,z_3)$; $P_2(x_4,y_4,z_4)$, o triângulo 1 é formado pelos vértices P_1,P_2,P_3 e o triângulo 2 pelos vértices P_3,P_2,P_4.

Assumindo que $0 < \alpha < 2\pi$, pois terá que percorrer todo o plano XZ de modo a construir uma esfera, e $-\pi/2 < \beta < \pi/2$, permitindo alcançar as extremidades da esfera, assim como tendo em consideração as variáveis sliceStep, que divide a circunferência em n fatias, e stackStep, que divide a circunferência em m *stacks* de modo a calcular o ângulo formando entre cada *stack*, tem-se:

$$\beta_1 + \text{stackStep} = \beta_2;$$
$$\alpha_1 + \text{sliceStep} = \alpha_2;$$

A construção da esfera na sua totalidade exige a criação de dois ciclos aninhados capazes de iterar pelo número de fatias e divisões indicadas como parâmetros. Assim, para cada iteração de divisões, é calculado um novo valor de β, stackAngle

```
stackAngle = M_PI / 2 - ((double)i * stackStep);
```

e a cada iteração do número de fatias, é calculado um novo valor de α, sliceAngle

```
sliceAngle = j * sliceStep;
```

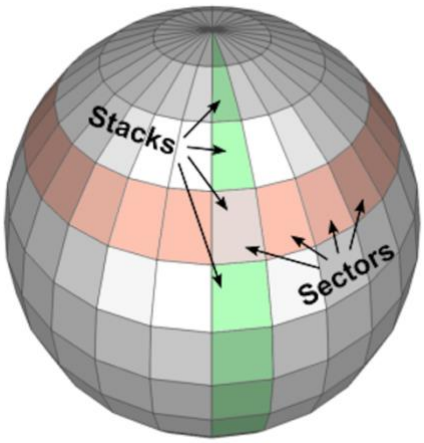


Figura 7: Exemplo de criação da esfera

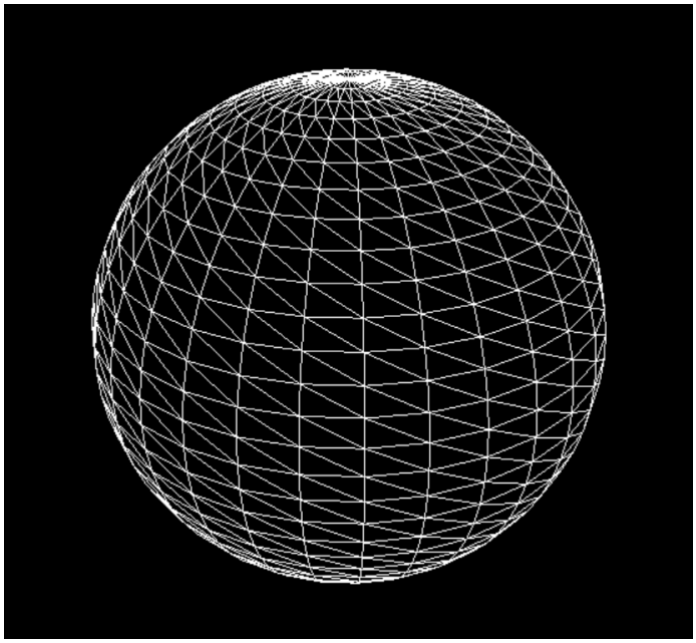


Figura 8: Representação da Esfera em OpenGL

A representação de um cone foi obtida através dos seguintes parâmetros passados como argumentos:

i. Raio da base;
ii. Altura do cone;
iii. Número de fatias;
iv. Número de divisões da lateral do cone;

A construção do cone teve como primeira etapa a elaboração da sua base. Desse modo, e uma vez que é preciso ter em consideração o número de fatias, é calculado o ângulo que cada fatia irá ter, de forma a repartir em fatias iguais a base do cone.

```
angle = ((2 * M_PI) / slices);
```

Após efetuada a divisão do número de fatias e recorrendo ao raio da base do cone, foram calculados os vértices dos triângulos que irão representar a sua base.

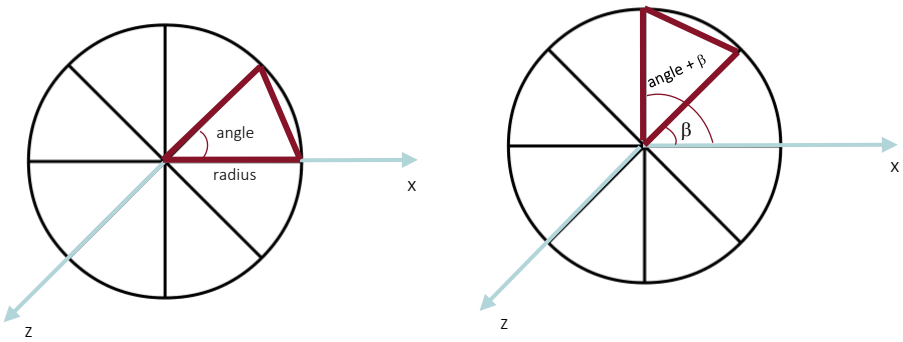


Figura 9: Representação cálculo dos triângulos

O desenho dos triângulos é efetuado através de um ciclo que, para cada número de fatias, vai calcular os vértices do triângulo seguinte. Este cálculo é conseguido devido à existência do ângulo β, dado pelo valor `angle * j`, em que j representa o número de fatias até então e β o valor do ângulo atual. Desta forma, o valor do ângulo do vértice seguinte é dado por `β = angle * (j+1)`.

Uma vez que estamos perante pontos baseados num raio e ângulo, é necessária a utilização de coordenadas polares.

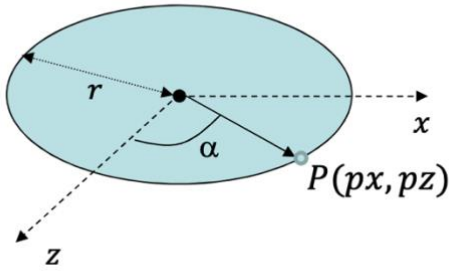
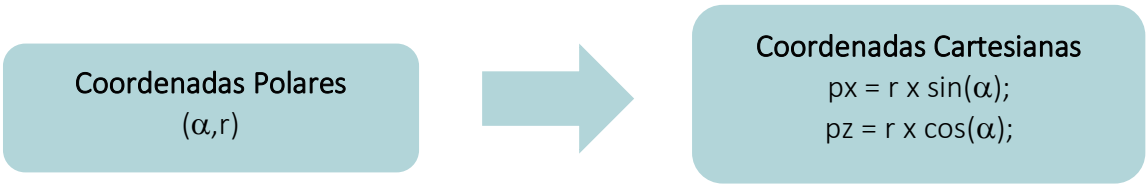


Figura 10: Representação da circunferência base para cálculos



Assim sendo, para cada iteração do ciclo tem-se os pontos

```
x = totRaio * sin(angle * j);
z = totRaio * cos(angle * j);
y = totAlt;

x1 = totRaio * sin((j + 1) * angle);
z1 = totRaio * cos((j + 1) * angle);
```

em que totRaio corresponde ao raio atual e totAltura à altura atual, dados pelas seguintes expressões, onde stackDiv representa a altura de cada divisão e stackRaio o raio de cada sub-cone:

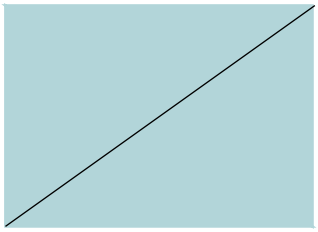
```
stackDiv = height / stacks;
stackRaio = radius / stacks;
```

```
totRaio = radius - (stackRaio * i);
totAlt = stackDiv * i;
```

Uma vez que o cone se encontra centrado na origem O(0,0,0) e a sua base está no plano XZ, totAlt tomará sempre o valor de 0 (zero) para as coordenadas dos pontos da base.

Concluída a construção da base do cone, iniciou-se o processo de criação da face lateral do mesmo. Para que tal fosse possível, foi necessário ter em consideração a relação inversa entre os valores tomados por y e pelo totRaio.

Uma vez que as divisões conferiam ao cone camadas retangulares, foram utilizados dois triângulos para construir cada porção correspondente a uma fatia de cada divisão.



Desta forma, através dos ciclos para cada fatia e cada divisão, foram calculados novos valores de raio, altura e ângulos. Seguindo o raciocínio utilizado para a construção da base do cone, foi possível a construção do mesmo.

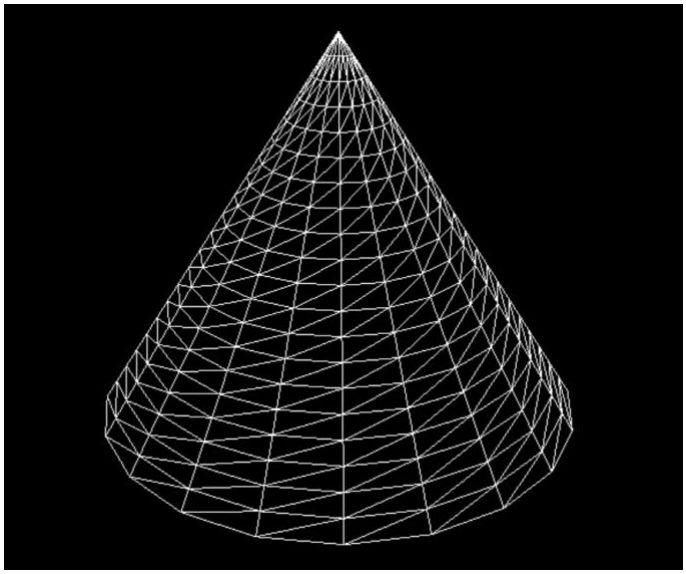


Figura 11: Representação do Cone em OpenGL

GENERATOR

Foi requerida a criação de uma aplicação geradora de ficheiros com as informações dos modelos (onde será guardada a informação relativa aos vértices de cada modelo).

Deste modo, foi criado um gerador, recorrendo à linguagem *C++*, que permitiu escrever em ficheiros na extensão *.3d* as coordenadas de cada vértice das primitivas, em que a cada linha corresponde um vértice.

Se, por exemplo, for pretendido desenhar um plano de comprimento = 3, o comando a introduzir no terminal será "`plano 3 plano.3d`".

Ao receber os parâmetros necessários, o gerador irá invocar a função capaz de escrever em ficheiro os vértices do plano.

```
std::string argument(argv[1]);


if (argument == "plano") {
        std::string comprimento(argv[2]);
        std::string name(argv[3]);


        drawPlan(std::stof(comprimento),name);
}



void drawPlan(float comp,std::string name) {
  ofstream myfile;
  myfile.open("../../SistemaSolar/"+name);

  float c = comp / 2;

  myfile << c << " " << "0.0" << " " << '-' << c << '\n';
  myfile << '-' << c << " " << "0.0" << " " << c << '\n';
  myfile << c << " " << "0.0" << " " << c << '\n';

  myfile << c << " " << "0.0" << " " << '-' << c << '\n';
  myfile << '-' << c << " " << "0.0" << " " << '-' << c << '\n';
  myfile << '-' << c << " " << "0.0" << " " << c << '\n';

  myfile.close();
}
```

O ficheiro gerado encontra-se na (Figura 12).

```
1.5 0.0 -1.5
-1.5 0.0 1.5
1.5 0.0 1.5
1.5 0.0 -1.5
-1.5 0.0 -1.5
-1.5 0.0 1.5

```

Figura 12: Ficheiro plano.3d

ENGINE

Para concluir a primeira fase deste projeto, foi ainda solicitada a elaboração de um motor que permita ler o ficheiro configurado, escrito em *XML*, e apresentar os respetivos modelos em três dimensões (3D).

Deste modo, foi implementada a função startParsing, responsável pelo *parsing* do ficheiro *XML* com o nome dos ficheiros a representar (ficheiros estes criados pelo generator). Com o auxílio da biblioteca *tinyxml2*, este ficheiro será iterado, guardando os nomes dos ficheiros em `std::vector<String>`.

De seguida, a função getVertices vai efetuar a leitura dos ficheiros devolvidos pela função anteriormente referida. A leitura vai ser efetuada em intervalos de nove valores, correspondentes às coordenadas de um triângulo. Estes valores serão depois armazenados numa estrutura de dados com o nome Triangulo, que contém também nove valores. Uma vez que vão ser lidos vários triângulos, vai ser criado um vetor onde vão ser armazenados todos os triângulos que pertence a uma figura (`std::vector<Triangulo>`). Dada a possibilidade de ser realizado o *parsing* a vários ficheiros, é utilizado um vetor de vetores, permitindo armazenar todas as figuras a implementar (`std::vector<std::vector<Triangulo>>`).

Por fim, aplicando a função startDrawing ao vetor de vetores devolvido pela função anterior, todos os triângulos vão ser desenhados, originando as figuras desejadas.

MOVIMENTO DA CÂMARA

De modo a facilitar a verificação da construção das primitivas gráficas, foram alterados os parâmetros relativos à posição da câmara.

Para este efeito, e uma vez que é necessária a utilização de coordenadas esféricas, foi criada a função spherical2cartesian.

```
camX = radius * sin(alfa) * cos(beta);
camZ = radius * cos(alfa) * cos(beta);
camY = radius * sin(beta);
```

Assim, tem-se

```
gluLookAt(camX, camY, camZ,
          0.0, 0.0, 0.0,
          0.0f,  1.0f, 0.0f);
```

Desta forma, é possível, através de teclas especiais, alterar a posição da câmara, modificando os valores de alfa e de beta.

```
case(GLUT_KEY_RIGHT):
      alfa -= 0.01; break;

case(GLUT_KEY_LEFT):
      alfa += 0.01; break;

case(GLUT_KEY_UP):
      beta += 0.01;
      if (beta > 1.5f)
            beta = 1.5f;
      break;

case(GLUT_KEY_DOWN):
      beta -= 0.01;
      if (beta < -1.5f)
            beta = -1.5f;
      break;
```

CONCLUSÕES

A realização da primeira fase do projeto do Sistema Solar permitiu que o grupo solidificasse conhecimento sobre primitivas gráficas, bem como adquirisse a capacidade de analisar ficheiros em *XML*. A construção de cada primitiva gráfica possibilitou o melhor entendimento sobre a relevância da ordem de construção de cada triângulo, uma vez que o não cumprimento da regra da mão direita para a definição de cada triângulo leva a resultados contrários aos esperados.

A criação das aplicações propostas possibilitou o que o grupo considera ser o sucesso da primeira fase deste projeto.