



Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

1ª Fase - Primitivas gráficas

Grupo XX



Gonçalo Esteves
.(A85731).



João Araújo
.(A84306).



Mário Real
.(A72620).



Rui Oliveira
.(A83610).

7 de Março de 2020

Conteúdo

1	Introdução	2
2	Gerador	2
2.1	Utilização	2
2.2	Plano	4
2.3	Caixa	5
2.4	Esfera	9
2.5	Cone	12
2.6	Cilindro	15
3	Parser XML	17
3.1	<i>Parsing</i> do ficheiro de configuração	17
3.2	Carregamento dos ficheiros	17
4	Motor	18
4.1	Desenho dos Modelos	18
4.2	Câmara	19
4.3	Movimento da câmara	20
5	Conclusão	22

Resumo

Primeira fase do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica da Universidade do Minho. Esta fase consiste na realização de duas aplicações, um gerador de vértices, que permite criar todos os pontos referentes aos triângulos necessários para gerar o respetivo modelo e um *engine* que lê um ficheiro de configuração XML, onde se encontram todos os ficheiros relevantes criados pelo gerador e que os carrega em memória. Depois de carregados o *engine* é também responsável por desenhar os modelos.

1 Introdução

Neste documento vamos esclarecer os aspetos mais importantes relativamente à primeira fase do nosso trabalho, onde faremos uma síntese e explicação do código elaborado e resultados obtidos.

Iremos também descrever e explicar as abordagens que tomamos quanto à realização do gerador e do *engine* assim como o seu funcionamento apresentando algumas imagens de exemplo para facilitar a compreensão. Iremos também especificar os diversos raciocínios utilizados para ultrapassar as barreiras encontradas.

2 Gerador

Nesta secção iremos explicar como funciona o nosso gerador de vértices e quais as funções que permitem gerar o output desejado.

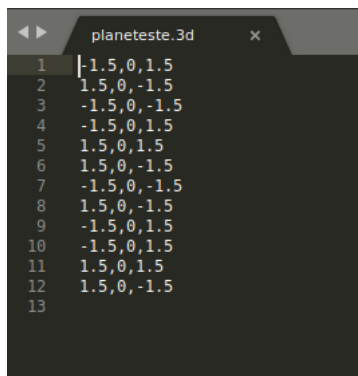
2.1 Utilização

Para utilizar o gerador é necessário passar-lhe como argumento o tipo da primitiva a gerar, as diferentes variáveis necessárias a cada uma das primitivas e por último o nome do ficheiro para o qual serão escritos os pontos.

Veja-se o exemplo: `$ generator box 5 5 5 3 box.3d`

Com estes argumentos o gerador irá criar, caso não exista, o ficheiro 'box.3d' e irá gerar os vértices necessários para desenhar uma caixa com largura, altura, e comprimento iguais a 5, 3 fatias e 3 pilhas que são respectivamente as subdivisões passadas no input.

O ficheiro criado tem a seguinte estrutura:



```
1 -1.5,0,1.5
2 1.5,0,-1.5
3 -1.5,0,-1.5
4 -1.5,0,1.5
5 1.5,0,1.5
6 1.5,0,-1.5
7 -1.5,0,-1.5
8 1.5,0,-1.5
9 -1.5,0,1.5
10 -1.5,0,1.5
11 1.5,0,1.5
12 1.5,0,-1.5
13
```

Figura 1: Exemplo do formato de ficheiro

As linhas correspondem aos pontos dos triângulos. As coordenadas de cada ponto são separadas por vírgulas, e cada ponto por um ”.

2.2 Plano

O plano é a primitiva mais simples de se desenhar. Recebe como parâmetros comprimento e largura.

O plano é desenhado em relação à origem paralelamente ao eixo do X e do Z, logo a sua coordenada Y será sempre 0.

Como podemos ver na imagem a seguir representada os cálculos para obter os dois triângulos que formam o plano são bastante intuitivas.

Sendo assim, para gerar o seguinte resultado final:

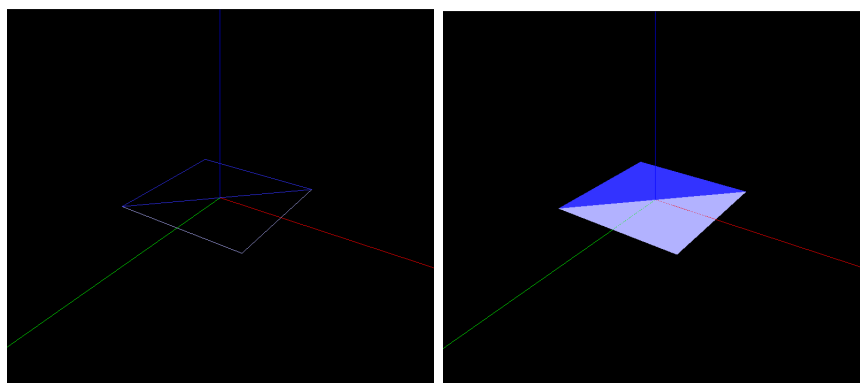


Figura 2: Plano no eixo XZ

Assim, para gerar um plano é necessário programar algo como:

```

1 // Triângulo de Cima (X,Y,Z)
  write -(x / 2) + ",0.0," + (z / 2)
3 write (x / 2) + ",0.0," + -(z / 2)
  write -(x / 2) + ",0.0," + -(z / 2)
5 // Triângulo de Baixo (X,Y,Z)
  write -(x / 2) + ",0.0," + (z / 2)
7 write (x / 2) + ",0.0," + (z / 2)
  write (x / 2) + ",0.0," + -(z / 2)

```

Listing 1: Código para gerar o plano

2.3 Caixa

Para desenhar uma caixa são necessárias 6 faces, ou seja, 6 planos. No entanto, não é possível reutilizar a função descrita anteriormente pois, esta necessita de mais argumentos: comprimento, largura, altura, e *divisões*.

Para 1 divisão, dada a altura (y), a largura (x) e comprimento (z) da caixa, é possível, da mesma forma que o plano, inferir as coordenadas dos vértices dos triângulos como mostra a figura seguinte:

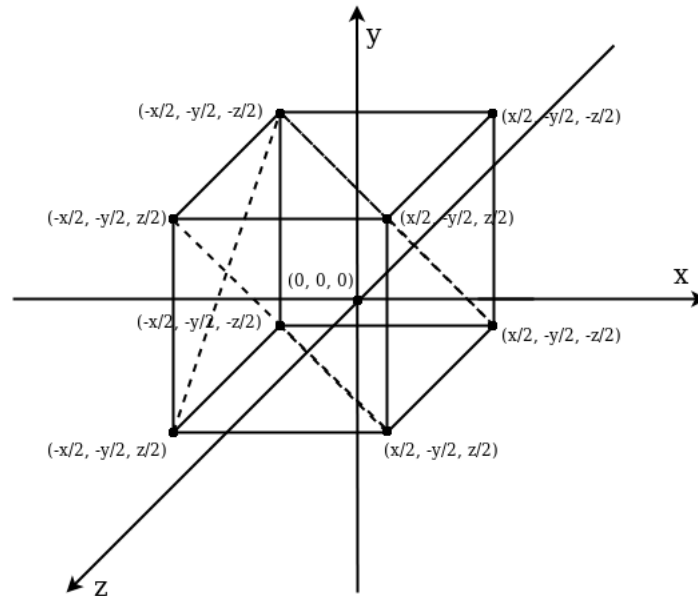


Figura 3: Caixa resultante de 1 divisão (*slices* e *stacks*)

No entanto, como é possível que seja passado um número variável de divisões (para converter em *slices* e *stacks*, ou "subcaixas") foi implementado um algoritmo que possui duas variáveis, *stepx*, *stepy*, ou *stepz*, *stepy*, ou *stepx*, *stepz*, dependendo das faces a desenhar, que representam a fração entre o comprimento do eixo total e o número de divisões a desenhar.

Sendo assim são utilizados dois ciclos *for*. O primeiro que percorre as linhas e, o segundo que percorre as colunas, desenhando os seis vértices de cada sub-plano, incrementando, de cada vez, o respetivo *step*.

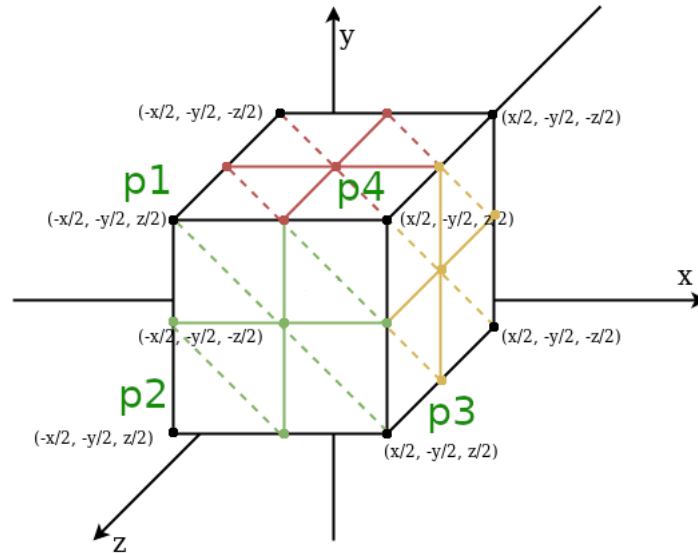


Figura 4: Caixa resultante de 1 divisões (*slices* e *stacks*)

Como podemos confirmar pelas figura anteriores que é possível desenhar duas faces de cada vez fazendo apenas variar uma coordenada, tendo em conta o eixo pretendido, para o simétrico. O algoritmo implementado gera as duas faces de cada vez, no entanto foi decidido separar a função geradora em 3 (uma para cada duas faces) de modo a facilitar a escrita e leitura do código como demonstramos num exemplo simplificado de estrutura que é seguido pelo nosso código.

```

void generateBoxFront(float x, float y, float z, int divis, string
filename){
2   for (float i = y/2; i > -y/2; i += stepy) {
      for (float j = -x/2; j < x/2; j += stepx) {
4         /*
          * p1—p4  (-p1)---(-p4)
          * | \ |      | / |
          * p2—p3  (-p2)---(-p3)
          * Front plane points.
          */
10        //faces frontais 1,2,3 \ 3,4,1 || -4 -3 -2 \ -2 -1 -4
        write j + i + z2 ; //p1
12        write j + (i-stepy) + z2 ; //p2
        write (j+stepx) + (i-stepy) + z2 ; //p3

14        write (j+stepx) + (i-stepy) + z2 ; //p3
16        write (j+stepx) + i + z2 ; //p4
        write j + i + z2 ; //p1
18        //—
        write (j+stepx) + i + -z2 ; //-p4
20        write (j+stepx) + (i-stepy) + -z2 ; //-p3
        write j + (i-stepy) + -z2 ; //-p2

22        write j + (i-stepy) + -z2 ; //-p2
24        write j + i << " ," + -z2 ; //-p1
        write (j+stepx) + i + -z2 ; //-p4
26    }
  }
28}

```

Listing 2: Código para gerar a face da frente e de trás da caixa

É, então, apresentado o resultado final:

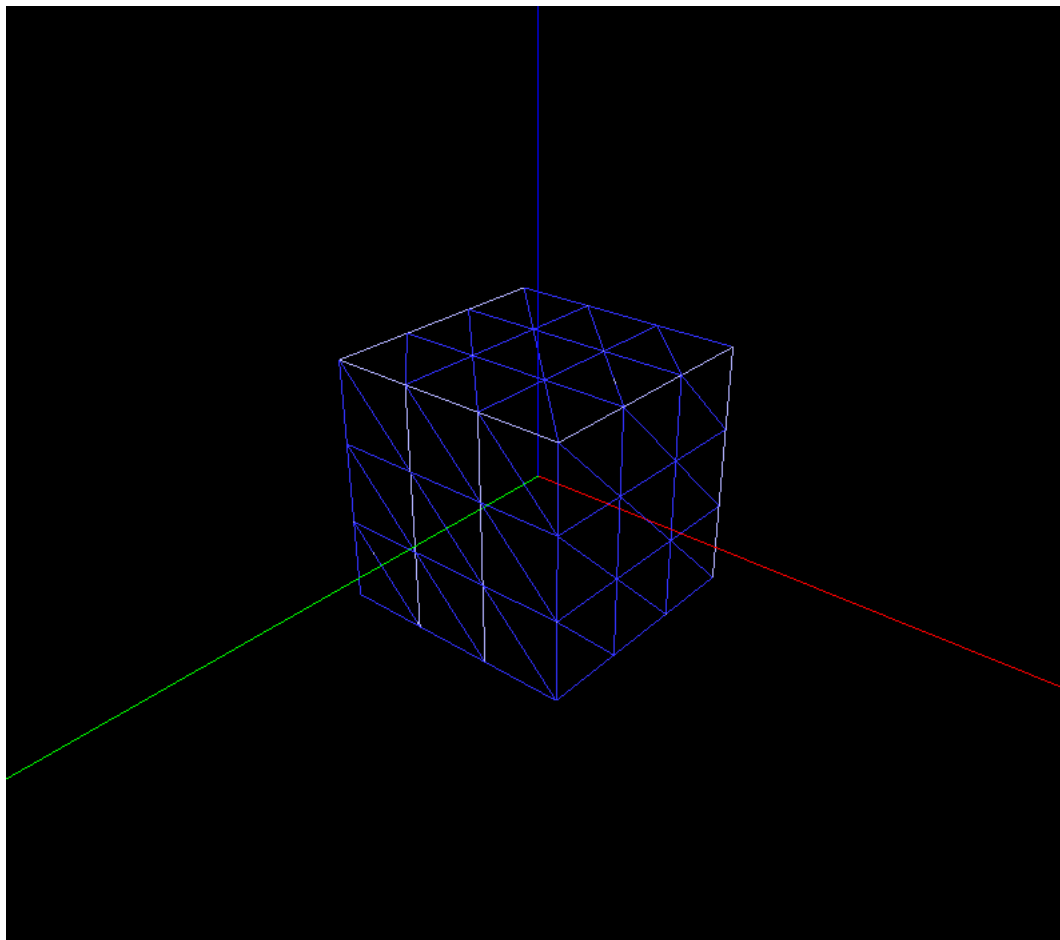


Figura 5: Resultado final para 3 subdivisões (3 *stacks* e 3 *slices*)

2.4 Esfera

Para gerar a nossa esfera utilizamos 3 variáveis: raio, *slices* (número de divisões verticais), e *stacks* (número de divisões horizontais). Os nossos pontos da superfície da esfera são os pontos de intersecção entre as nossas *slices* e *stacks*, sendo estes calculados a partir de coordenadas esféricas com auxílio do valor do raio e de dois ângulos *alpha* e *beta*.

Sabendo o número de *slices* e *stacks* recebido nos parâmetros da nossa função variá-mos os ângulos *alpha* e *beta*, respetivamente por cada *slice* e *stack*. Sendo que o *alpha* que está ligado ao número de *slices* irá variar de 0 a 2π . E o *beta* que está ligado ao número de *stacks* irá variar de $-\pi/2$ a $\pi/2$.

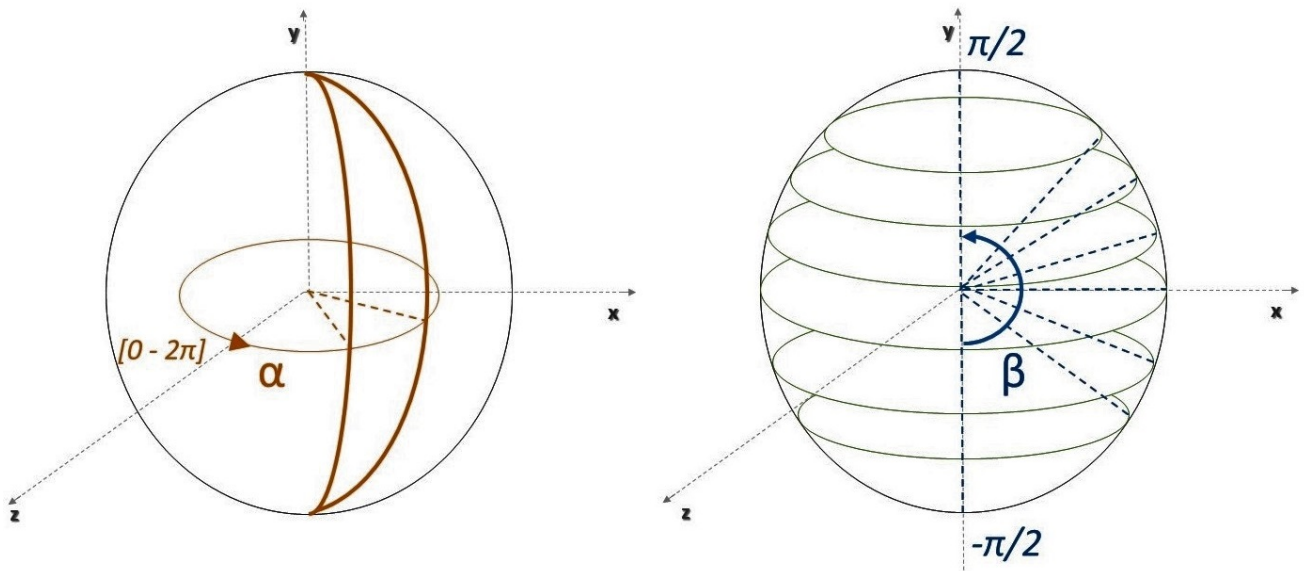


Figura 6: Variação dos ângulos *alpha* e *beta*

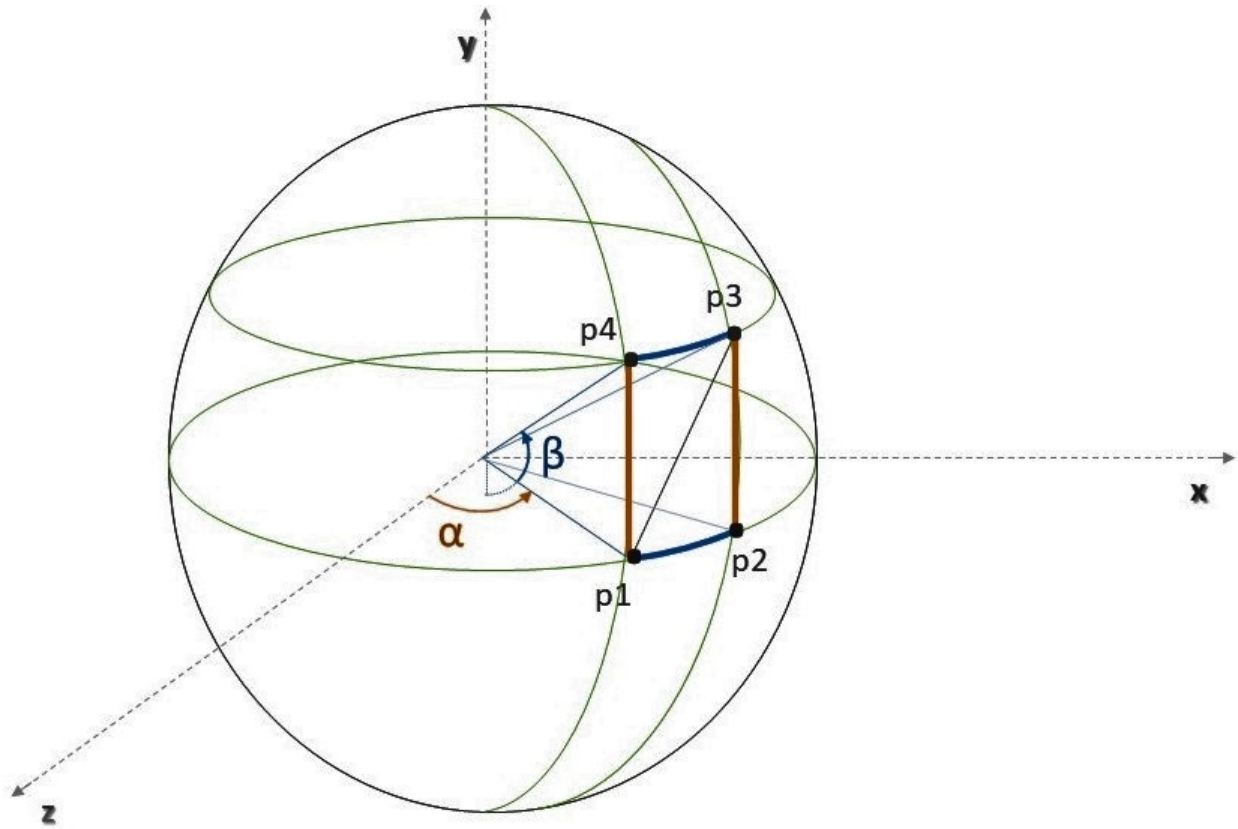


Figura 7: Diagrama da Esfera

Para obtermos todos os pontos da esfera, usamos dois ciclos encadeados que irão então percorrer, através dos ângulos *alpha* e *beta*, todas as *stacks* e *slices*, sendo que a cada iteração iremos ter a intersecção entre duas *slices* consecutivas com duas *stacks* também estas consecutivas. E assim, como observamos na imagem, das intersecções obtemos 4 pontos, e é a partir deles que iremos poder criar os 2 triângulos que irão formar um dos vários retângulos que moldam o formato da nossa esfera.

As coordenadas cartesianas de cada ponto são então calculadas com as seguintes fórmulas das coordenadas esféricas:

$$X = \text{raio} * \cos(\beta) * \sin(\alpha)$$

$$Y = \text{raio} * \sin(\beta)$$

$$Z = \text{raio} * \cos(\beta) * \cos(\alpha)$$

O código que gera a esfera segue o seguinte modelo:

```

1 //ciclo das stacks (variancia do beta por stack de baixo para cima da esfera)
for (int i = 0; i < stacks; i++){
3 // incremento dos betas
  beta1 = i * (M_PI / stacks) - M_PI_2;
5  beta2 = (i + 1) * (M_PI / stacks) - M_PI_2;
  //ciclo das slices (variancia do alpha por slice a volta da esfera)
7  for (int j = 0; j < slices; j++) {
    //incremento dos alphas
9    alpha1 = j * 2 * M_PI / slices;
    alpha2 = (j + 1) * 2 * M_PI / slices;
11    //geracao dos pontos dos triangulos atraves das coordenadas esfericas
    alpha1/2 e beta1/2
/*p1*/write radius * cos(beta1)*sin(alpha1) ++ radius * sin(beta1) ++ radius *
    cos(beta1)*cos(alpha1);
13 /*p2*/write radius * cos(beta1)*sin(alpha2) ++ radius * sin(beta1) ++ radius *
    cos(beta1)*cos(alpha2);
/*p3*/write radius * cos(beta2)*sin(alpha2) ++ radius * sin(beta2) ++ radius *
    cos(beta2)*cos(alpha2);
15
/*p1*/write radius * cos(beta1)*sin(alpha1) ++ radius * sin(beta1) ++ radius *
    cos(beta1)*cos(alpha1);
17 /*p3*/write radius * cos(beta2)*sin(alpha2) ++ radius * sin(beta2) ++ radius *
    cos(beta2)*cos(alpha2);
/*p4*/write radius * cos(beta2)*sin(alpha1) ++ radius * sin(beta2) ++ radius *
    cos(beta2)*cos(alpha1);
19  }
}

```

Listing 3: Código para gerar a esfera

Resultado final:

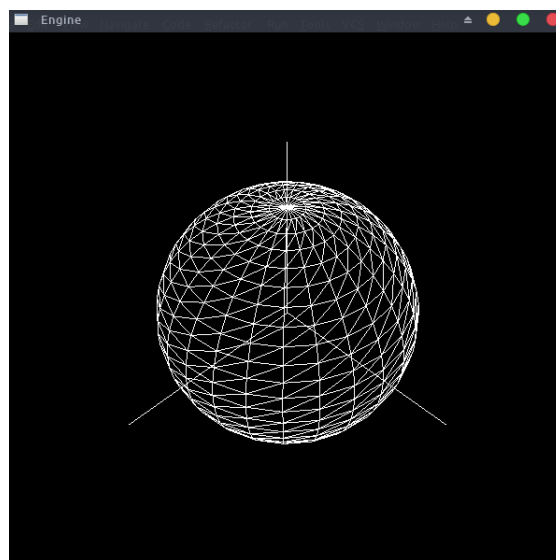


Figura 8: Esfera 2 25 25

2.5 Cone

Para gerar o cone utilizamos 4 variáveis: raio, altura, *slices* e *stacks*. Usamos coordenadas polares para determinar um ponto através de um ângulo, a que chamamos *alpha*, e do raio.

Para desenhar a base do cone utilizamos um ciclo para cada fatia com as respectivas variações de *alpha* de maneira a formar uma circunferência em torno da origem (0,0,0) de raio *radius*, a partir de um triângulo com coordenadas polares, como mostra a figura:

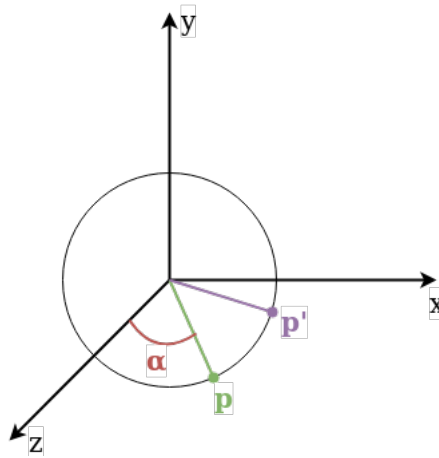


Figura 9: Base do Cone

A cada iteração do ciclo o *alpha* aumenta ($2\pi/slices$) vezes, o valor de Y é sempre 0 pois a circunferência encontra-se no plano XOZ.

Para desenhar o corpo do cone, tendo em conta o número de *stacks* e de *slices* são utilizados dois ciclos *for*. Um ciclo exterior para cada *stack* que percorre as colunas e um ciclo interior para cada *slice* que percorre as linhas. A cada iteração do ciclo interior desenhamos 2 triângulos, cada um com a sua altura e raio, formando um retângulo (como podemos verificar na figura 12). Cada triângulo tem pontos com coordenadas que diferem de altura e/ou raio. À medida que vamos subindo na *stack*, o raio vai diminuindo e a altura aumenta.

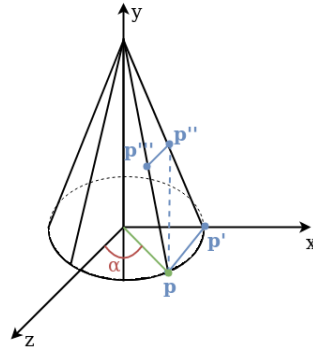


Figura 10: Corpo do Cone

A cada iteração do ciclo interior o *alpha* aumenta ($2\pi/slices$) vezes.
 A cada iteração do ciclo exterior o raio decrementa ($radius/stacks$) vezes.
 A cada iteração do ciclo exterior a altura aumenta ($height/stacks$) vezes.

O código que gera o cone segue o seguinte modelo:

```

1 alpha = (2 * M_PI) / slices;
2 scaleH = height/stacks;
3 scaleR = radius/stacks;

5 //BASE DO CONE
6 for (int i=0; i<slices;i++){
7     teta = i * alpha;
8     tetaNext = (i+1) * alpha;
9
10    write radius * sin(tetaNext) ++ 0.0 ++ radius * cos(tetaNext);
11    write radius * sin(teta) ++ 0.0 ++ radius * cos(teta) ;
12    write 0.0 ++ 0.0 ++ 0.0;
13 }

15 //CORPO DO CONE
16 for (int i = 0; i < stacks; i++){
17     heightNow = (float)i*scaleH;
18     heightNext = heightNow + scaleH;
19     radiusNow = radius - (float)i * scaleR;
20     radiusNext = radius - (float)(i + 1) * scaleR;
21     for (int j = 0; j < slices; j++) {
22         teta = (float)j * alpha;
23         tetaNext = (float)(j + 1) * alpha;
24
25         /*p1*/ write radiusNow * sin(teta) ++ heightNow ++ radiusNow * cos(teta);
26         /*p2*/ write radiusNow * sin(tetaNext) ++ heightNow ++ radiusNow * cos(tetaNext);
27         /*p3*/ write radiusNext * sin(tetaNext) ++ heightNext ++ radiusNext * cos(tetaNext);
28
29         /*p1*/ write radiusNow * sin(teta) ++ heightNow ++ radiusNow * cos(teta);
30         /*p3*/ write radiusNext * sin(tetaNext) ++ heightNext ++ radiusNext * cos(tetaNext);
31         /*p4*/ write radiusNext * sin(teta) ++ heightNext ++ radiusNext * cos(teta);
32     }
33 }

```

Listing 4: Código para gerar o cone

Resultado final:

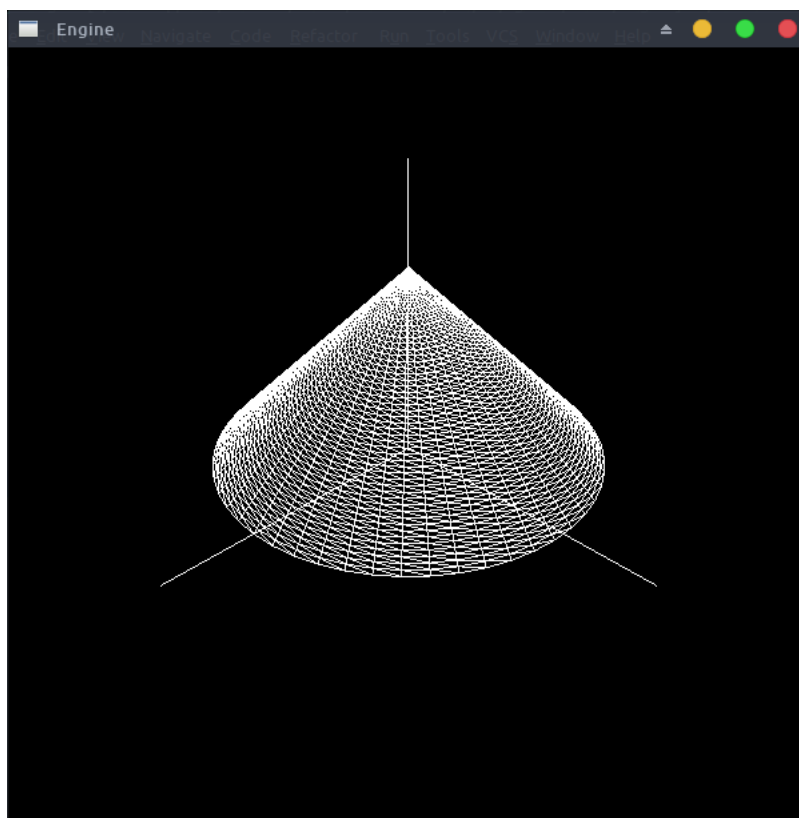


Figura 11: Cone 2 2 50 50

2.6 Cilindro

Como extra, utilizamos os conhecimentos adquiridos anteriormente para criar mos uma figura geométrica adicional, o cilindro. Para gerar o cilindro utilizamos 4 variáveis: raio, altura, *slices* e *stacks*. Usamos coordenadas polares para determinar um ponto através de um ângulo, a que chamamos *alpha*, e do raio.

Para desenhar as bases do cilindro utilizamos um ciclo para cada fatia com as respectivas variações de *alpha* de maneira a formar uma circunferência em torno da origem (0,0,0) de raio *radius*, a partir de um triângulo com coordenadas polares, como mostra a figura:

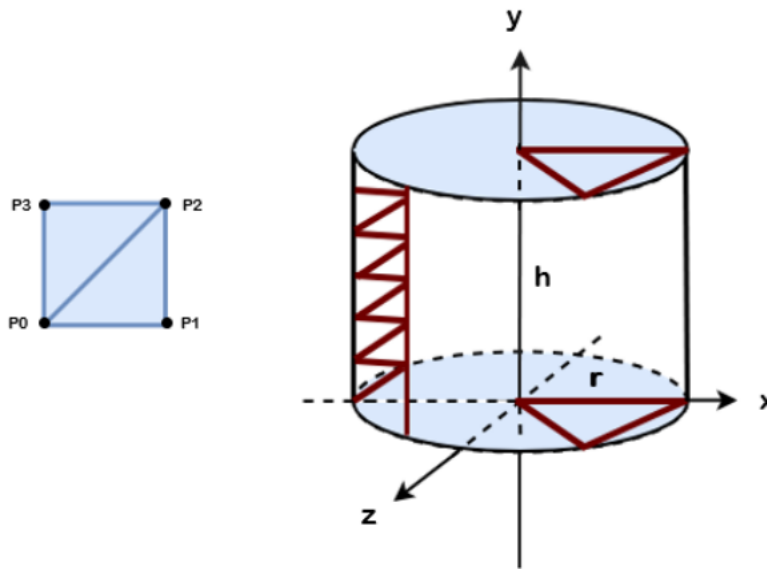


Figura 12: Cilindro

A cada iteração do ciclo o *alpha* aumenta ($2\pi/slices$) vezes, o valor de Y é sempre 0 pois a circunferência encontra-se no plano XOZ.

Para desenhar o corpo do cilindro, tendo em conta o número de *stacks* e de *slices* são utilizados dois ciclos *for*. Um ciclo exterior para cada *stack* que percorre as colunas e um ciclo interior para cada *slice* que percorre as linhas. A cada iteração do ciclo interior desenhamos 2 triângulos, cada um com a sua altura e raio, formando um retângulo (como podemos verificar na figura 12). Cada triângulo tem pontos com coordenadas que diferem de altura e/ou raio. À medida que vamos subindo na *stack*, o raio vai diminuindo e a altura aumenta.

A cada iteração do ciclo interior o α aumenta ($2\pi/slices$) vezes.
 A cada iteração do ciclo exterior o raio decrementa ($radius/stacks$) vezes.
 A cada iteração do ciclo exterior a altura aumenta ($height/stacks$) vezes.

O código que gera o cone segue o seguinte modelo:

```

1 alpha = (2 * M_PI) / slices;
2 scaleHeight = height / stacks;
3
4 for (int i = 0; i < stacks; i++) {
5     heightNow = -(height/2) + (i * scaleHeight);
6     heightNext = heightNow + scaleHeight;
7
8     for (int j = 0; j < slices; j++) {
9         teta = j * alpha;
10        tetaNext = (j + 1) * alpha;
11
12        //corpo do cilindro
13        /*p1*/ write radius * sin(teta) ++ heightNow ++ radius * cos(teta);
14        /*p2*/ write radius * sin(tetaNext) ++ heightNow ++ radius * cos(tetaNext);
15        /*p3*/ write radius * sin(tetaNext) ++ heightNext ++ radius * cos(tetaNext);
16
17        /*p1*/ write radius * sin(teta) ++ heightNow ++ radius * cos(teta);
18        /*p3*/ write radius * sin(tetaNext) ++ heightNext ++ radius * cos(tetaNext);
19        /*p4*/ write radius * sin(teta) ++ heightNext ++ radius * cos(teta);
20
21        //bases
22        write 0.0 ++ -height / 2 ++ 0.0;
23        write radius * sin(tetaNext) ++ -height / 2 ++ radius * cos(tetaNext);
24        write radius * sin(teta) ++ -height / 2 ++ radius * cos(teta);
25
26        write 0.0 ++ height / 2 ++ 0.0;
27        write radius * sin(teta) ++ height / 2 ++ radius * cos(teta);
28        write radius * sin(tetaNext) ++ height / 2 ++ radius * cos(tetaNext);
29    }
30 }
31

```

Listing 5: Código para gerar o cone

Resultado final:

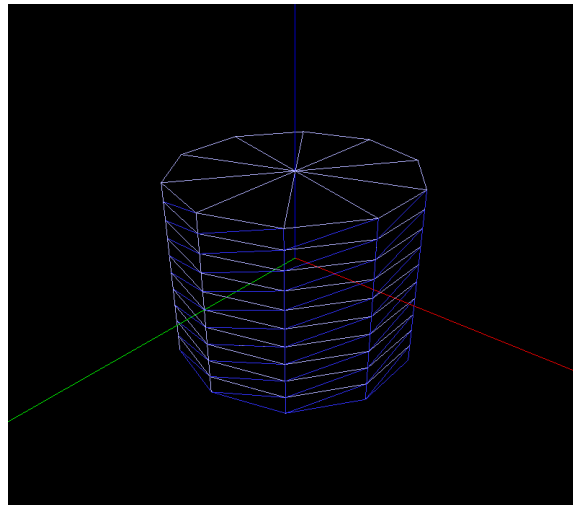


Figura 13: cylinder 3 4 10 10

3 Parser XML

Por forma a proceder ao *parsing* do ficheiro XML foi utilizada a biblioteca *tinyxml2* que, por ser uma biblioteca bastante completa, eficiente e rápida, torna o seu uso bastante adequado para o objetivo pretendido. Deste modo, poderemos construir o nosso *engine*, que será a aplicação responsável por realizar a leitura de um ficheiro XML, com o intuito de obter ficheiros .3d que contêm os pontos que permitem gerar as diferentes formas. Assim sendo, é efetuada a leitura deste, guardando todos os pontos, por forma a exibir as figuras. Será também possível interagir com estas através de diversos comandos.

3.1 *Parsing* do ficheiro de configuração

Para esta parte do trabalho foi utilizada a API do *tinyxml2*, pegando num ficheiro XML, verificando primeiro onde existe o primeiro elemento com a *tag* "model"e, de seguida, caso exista, procurar o atributo *file*.

```
1 < scene >
   < models >
3     < model file = " ../.. / files / exemplo .3 d " / >
   </ models >
5 </ scene >
```

Listing 6: Formato do ficheiro XML

3.2 Carregamento dos ficheiros

Para cada ficheiro encontrado no ficheiro de configuração XML, é invocada a função responsável por abrir o ficheiro. Esta é responsável por invocar uma segunda função que irá lê-lo e carregar os pontos do respetivo modelo para a estrutura.

Com o objetivo de evitar carregamentos repetidos, os vértices de cada modelo são carregados para uma estrutura que é depois adicionada à estrutura que será representada no final.

4 Motor

4.1 Desenho dos Modelos

O motor é responsável pelo carregamento de todas as informações trazidas pelos processos anteriormente explicados para finalmente executar as nossas funções de desenho aplicadas aos inputs recebidos. Entre outras, uma das principais é o desenho dos nossos pontos gerados e enviados para o motor como se pode observar no código a baixo.

```
1 void drawPrimitives(void){
3     glBegin(GL_TRIANGLES);
4     int i = 0;
5     bool cor = true;
7     for(const Point pt : points){
8         if(i == 3){
9             cor = !cor;
10            i = 0;
11        }
13        if(cor){
14            glColor3f(0.2, 0.2, 1);
15            glVertex3f(pt.x, pt.y, pt.z);
16        }
17        else{
18            glColor3f(0.7, 0.7, 1);
19            glVertex3f(pt.x, pt.y, pt.z);
20        }
21        i++;
23    }
24    glEnd();
25 }
```

Listing 7: Desenho dos Pontos recebidos e interpretados

4.2 Câmara

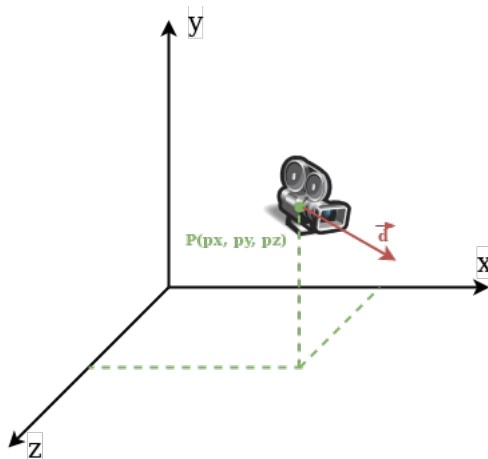


Figura 14: Diagrama da câmera.

Nesta fase foi implementada uma câmera capaz de se mover em torno dos modelos segundo um raio variável.

A câmera é capaz de se deslocar para cima e para baixo, para a esquerda e para a direita e aproximar-se e afastar-se, assemelhando-se ao movimento limitado pela superfície de uma esfera.

Os movimentos da câmera são implementados recorrendo ao uso das seguintes variáveis:

```

1 gluLookAt(radius*cos(beta)*sin(alpha), radius*sin(beta), radius*cos(
    beta)*cos(alpha),
    0.0, 0.0, 0.0,
3    0.0f, 1.0f, 0.0f);

```

Listing 8: Função da câmera

As variáveis **px**, **py** e **pz** representam a posição da câmera. A variável **radius** representa a distância que a câmera está da origem e as variáveis **pa** e **pb** são as variáveis que permitem o movimento da câmera segundo a superfície da esfera imaginária.

É de salientar que o vetor **d** está sempre a olhar para o ponto de origem (0, 0, 0).

4.3 Movimento da câmara

O movimento da câmara foi implementado utilizando coordenadas esféricas fazendo o valor de px , py e pz variar consoante um ângulo $alpha$ e $beta$ (pa e pb). Neste caso, conseguimos controlar a posição vertical somando ou subtraindo o ângulo $beta$ da seguinte maneira:

```

switch (key){
2   case GLUT_KEY_UP:
        if(beta < (M_PI/2 - step))
4           beta += step;
        break;

6   case GLUT_KEY_DOWN:
        if(beta > -(M_PI/2 - step))
8           beta -= step;
        break;

10  case GLUT_KEY_LEFT:
        alpha -= step;
14         break;

16  case GLUT_KEY_RIGHT:
        alpha += step;
18         break;

```

Listing 9: Variação positiva do ângulo beta

De referir que o código em cima apresentado é apenas um excerto dos cálculos efetuados e que o operador ternário serve para evitar que o valor do ângulo $beta$ tenha valores inadmissíveis no contexto das coordenadas esféricas.

O movimento da câmara dentro do motor é conseguido premindo uma das teclas:

- Seta para cima - Rodar a vista para cima
- Seta para baixo - Rodar a vista para baixo
- Seta para a esquerda - Rodar a vista para a esquerda
- Seta para a direita - Rodar a vista para a direita
- F1 - Aumentar tamanho da imagem
- F2 - Diminuir tamanho da imagem
- F3 - Preencher a figura
- F4 - Mudar a figura para linhas apenas
- F5 - Mudar a figura para pontos apenas

```
2      case GLUT_KEY_F1:
3          radius -= step;
4          break;
5
6      case GLUT_KEY_F2:
7          radius += step;
8          break;
9
10     case GLUT_KEY_F3:
11         line = GL_FILL;
12         break;
13
14     case GLUT_KEY_F4:
15         line = GL_LINE;
16         break;
17
18     case GLUT_KEY_F5:
19         line = GL_POINT;
20         break;
```

Listing 10: Implementação das Restantes funcionalidades de visualização

5 Conclusão

Para esta primeira fase podemos afirmar que cumprimos com os objetivos estabelecidos uma vez que criamos um programa que gera todas as primitivas pedidas, um motor 3D que processa e desenha os diferentes modelos gerados assim como implementamos o movimento da câmara e *menus* de visualização. No entanto verificamos que para um elevado número de *slices* e *stacks* o gerador da caixa gera pontos incorretos devido à acumulação do erro em números de vírgula flutuante.

Como correção deveríamos alterar o ciclo gerador da caixa forma a que a posição do vértice a desenhar *step* não seja incrementada usando somas, mas sim atualizada usando a multiplicação.