Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Computação Gráfica

Fase 1
Motor e Gerador

Grupo 18

Célia Figueiredo a67637

Luís Pedro Fonseca a60993

Braga, 3 de Março de 2017

Conteúdo

Resumo

O presente relatório documenta a 1ª fase do trabalho prático da Unidade Curricular de Computação Gráfica. Nesta primeira fase o objectivo foi construir duas aplicações: um gerador e um motor. Em termos gerais, o gerador é responsável por guardar num ficheiro, um conjunto de pontos correspondentes a uma figura. Esses ficheiros de pontos são posteriormente lidos pela aplicação motor que tem como tarefa desenhá-los numa janela. Neste relatório pretende-se apresentar a forma como estas duas aplicações foram construídas bem como explicar algumas decisões tomadas.

1. Gerador

1.1 Objetivos

O gerador é uma aplicação que é capaz de receber e interpretar pedidos do utilizador para desenho de figuras e gerar um ficheiro .3d com os pontos correspondentes à figura pedida. Nesta primeira etapa, o objetivo implementar o suporte à criação de um plano, uma caixa, um cone e uma esfera.

1.2 Programa principal

O pseudo-código do programa principal é o seguinte:

```
int main(int argc, char** argv) {
Declara figura onde vao ser guardados os pontos
figuraCriada = false;

if (figura pedida == plano) {
Lê e interpreta parametros
Chama funçao da figura para desenhar os pontos
de um plano
Grava pontos em ficheiro
figuraCriada = true;
}

if (figura pedida == caixa) {
Lê e interpreta parametros
Chama funçao da figura para desenhar os pontos
de uma caixa
Grava pontos em ficheiro
figuraCriada = true;
}

if (figura pedida == cone) {
Lê e interpreta parametros
Chama funçao da figura para desenhar os pontos
de um cone
Grava pontos em ficheiro
figuraCriada = true;
}
```

```
if (figura pedida == esfera) {
Lê e interpreta parametros
Chama funçao da figura para desenhar os pontos
de uma esfera
Grava pontos em ficheiro
figuraCriada = true;
}


if (!figuraCriada) {
if (o programa for corrido sem argumentos) {
Informar utilizador que programa
foi corrido sem argumentos
}
else {
Informar utilizador que nao foi possivel
criar a figura
}
Mostra mensagem de ajuda com sintaxe dos comandos
}


return 0;
}
```

1.3 Primitivas

Nesta secção apresentam-se os comandos correspondentes aos pedidos de figuras que o utilizador pode fazer e é explicada a sua implementação.

1.3.1 Gerar Planos

Para gerar um plano, o utilizador deve efetuar o comando com a seguinte sintaxe:

```
Gerador plane comprimento largura ficheiro
```

O resultado deste comando é a criação de um plano em XZ centrado no ponto (0,0,0) com o comprimento e largura indicados.

Plano em XZ (Y=constante)

Um plano é formado por 2 triângulos. Com a informação de 4 pontos é possível desenhar esses triângulos. A figura **??** representa um plano em XZ. De notar que é possível considerar dois triângulos: o triângulo formado por [ABC] e o triângulo formado por [CDA]

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura **??** são as seguintes:

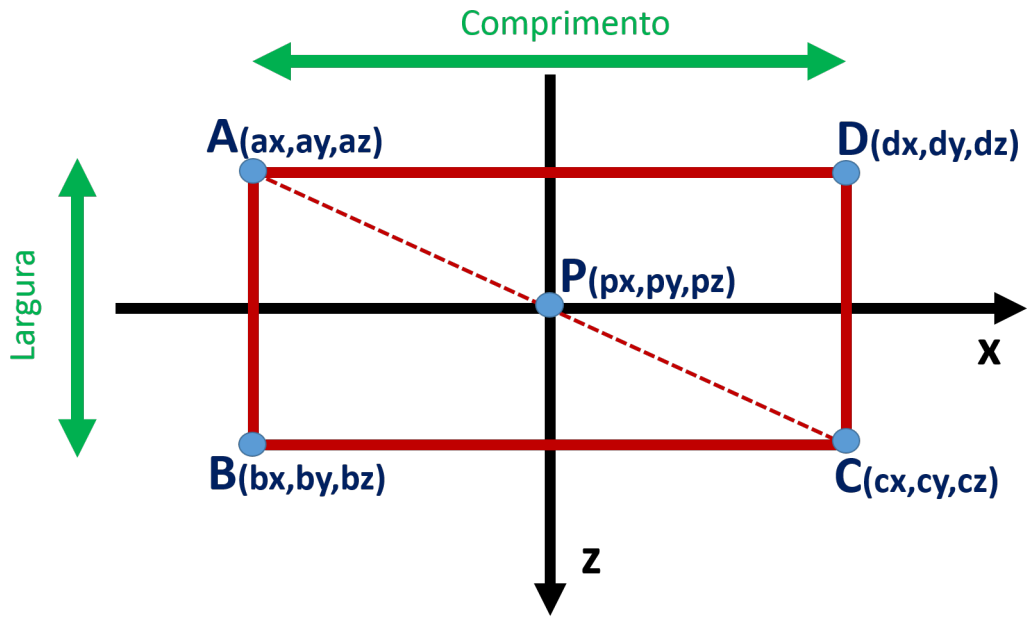

Figura 1.1: Plano em XZ centrado na origem

```
A(-comprimento/2, 0, -largura/2)
B(-comprimento/2, 0, largura/2)
C(comprimento/2, 0, largura/2)
P(0, 0, 0)
```

É também fácil exprimir as coordenadas de um plano não necessariamente centrado na origem, em função do seu centro P:

```
A(-comprimento/2 + px, 0 + py, -largura/2 + pz)
B(-comprimento/2 + px, 0 + py, largura/2 + pz)
C(comprimento/2 + px, 0 + py, largura/2 + pz)
P(px, py, pz)
```

A ordem pela qual se adiciona os pontos à figura determina para que lado ela fica virada. Se quisermos que o plano fique virado para o sentido positivo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-B-C (1º triângulo), seguido de C-D-A (2º triângulo). Se quisermos que o plano fique virado para o sentido negativo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-D-C (1º triângulo), seguido de C-B-A (2º triângulo).

A função responsável por implementar este algoritmo é a função *criaPlanoEmY()*, cujo pseudo-código se apresenta de seguida:

```
Figura& criaPlanoEmY(Ponto3D centroPlano, float comprimento,
float largura, int orientacao) {

Calcula coordendas dos pontos A,B,C e D

if (orientacao == 1) {
Coloca pontos pela ordem A-B-C-C-D-A
}
```

```
else {
Coloca pontos pela ordem A-D-C-C-B-A
}

return *this;
}
```

De notar que esta função além do comprimento e largura, recebe ainda o centro do plano e a orientação do mesmo.

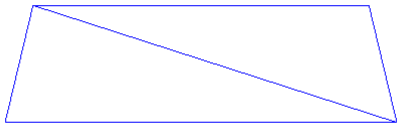

Figura 1.2: Exemplo de plano em XZ gerado, com 4 de comprimento e 2 de largura

Plano em XY (Z=constante)

Embora apenas fosse pedido que o gerador tivesse a capacidade de gerar um plano em XZ, considerou-se útil disponibilizar também uma primitiva para criar planos em XY. Um plano é formado por 2 triângulos. Com a informação de 4 pontos é possível desenhar esses triângulos. A figura 1.3 representa um plano em XY. De notar que é possível considerar dois triângulos: o triângulo formado por [ABC] e o triângulo formado por [CDA]

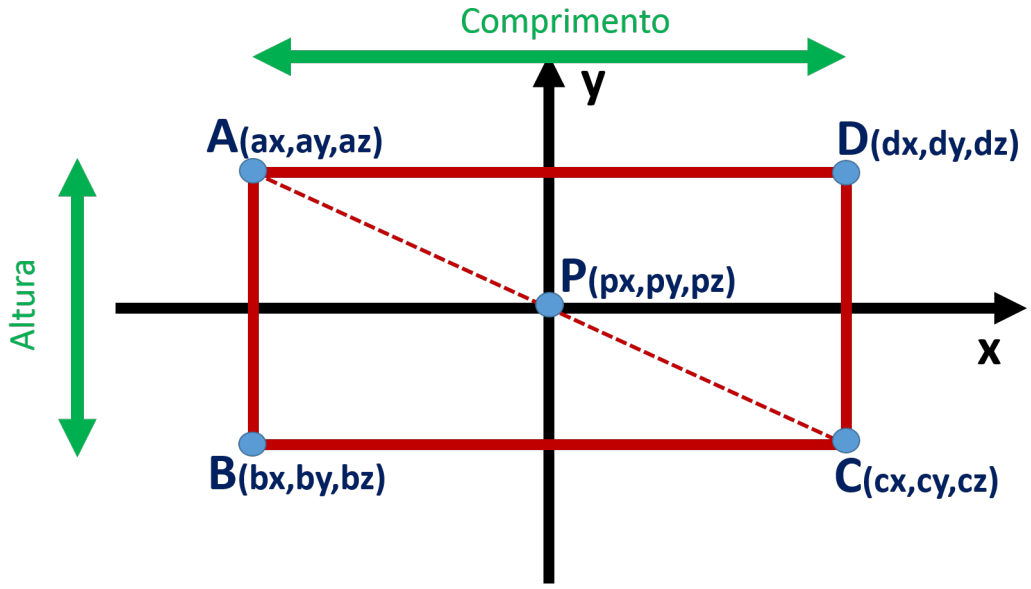

Figura 1.3: Plano em XZ centrado na origem

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura 1.3 são as seguintes:

```
A(-comprimento/2 , altura/2, 0)
B(-comprimento/2 , -altura/2, 0)
C(comprimento/2 , -altura/2, 0)
D(comprimento/2 , altura/2, 0)
P(0,0,0)
```

É também fácil exprimir as coordenadas de um plano não necessariamente centrado na origem, em função do seu centro P:

```
A(-comprimento/2 + px, altura/2 + py, 0 + pz)
B(-comprimento/2 + px, -altura/2 + py, 0 + pz)
C(comprimento/2 + px, -altura/2 + py, 0 + pz)
D(comprimento/2 +px, altura/2 + py, 0 + pz)
P(px, py, pz)
```

A ordem pela qual se adiciona os pontos à figura determina para que lado ela fica virada. Se quisermos que o plano fique virado para o sentido positivo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-B-C (1º triângulo), seguido de C-D-A (2º triângulo). Se quisermos que o plano fique virado para o sentido negativo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-D-C (1º triângulo), seguido de C-B-A (2º triângulo).

A função responsável por implementar este algoritmo é a função *criaPlanoEmZ()*, cujo pseudo-código se apresenta de seguida:

```
Figura& criaPlanoEmZ(Ponto3D centroPlano, float comprimento,
float altura, int orientacao) {

Calcula coordendas dos pontos A,B,C e D

if (orientacao == 1) {
Coloca pontos pela ordem A-B-C-C-D-A
}
else {
Coloca pontos pela ordem A-D-C-C-B-A
}

return *this;
}
```

De notar que esta função além do comprimento e altura, recebe ainda o centro do plano e a orientação do mesmo.

Plano em YZ (X=constante)

Embora apenas fosse pedido que o gerador tivesse a capacidade de gerar um plano em XZ, considerou-se útil disponibilizar também uma primitiva para criar planos em YZ. Um plano é formado por 2 triângulos. Com a informação de 4 pontos é possível desenhar esses triângulos. A figura 1.4 representa um plano em YZ. De notar que é possível considerar dois triângulos: o triângulo formado por [ABC] e o triângulo formado por [CDA]

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura 1.4 são as seguintes:

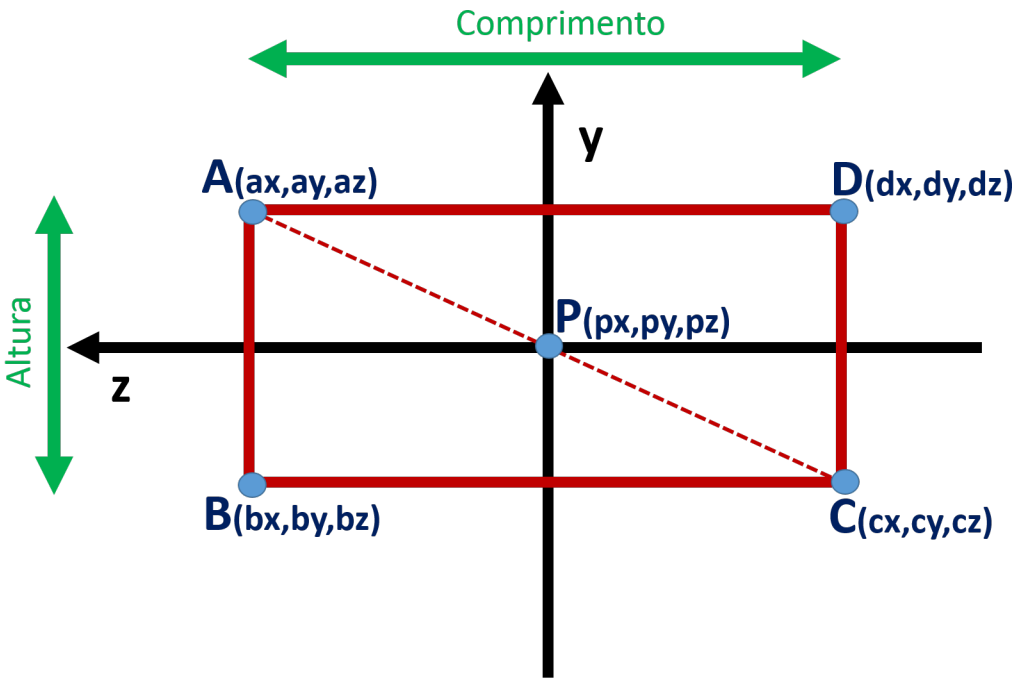

Figura 1.4: Plano em XZ centrado na origem

```
A(0, altura/2, comprimento/2)
B(0, -altura/2, comprimento/2)
C(0, -altura/2, -comprimento/2)
D(0, altura/2, -comprimento/2)
P(0,0,0)
```

É também fácil exprimir as coordenadas de um plano não necessariamente centrado na origem, em função do seu centro P:

```
A(0 + px, altura/2 + py, comprimento/2 + pz)
B(0 + px, -altura/2 + py, comprimento/2 + pz)
C(0 + px, -altura/2 + py, -comprimento/2 + pz)
D(0 + px, altura/2 + py, -comprimento/2 + pz)
P(px, py, pz)
```

A ordem pela qual se adiciona os pontos à figura determina para que lado ela fica virada. Se quisermos que o plano fique virado para o sentido positivo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-B-C (1º triângulo), seguido de C-D-A (2º triângulo). Se quisermos que o plano fique virado para o sentido negativo do eixo dos Y, deve-se colocar os pontos pela seguinte ordem: A-D-C (1º triângulo), seguido de C-B-A (2º triângulo).

A função responsável por implementar este algoritmo é a função *criaPlanoEmX()*, cujo pseudo-código se apresenta de seguida:

```
Figura& criaPlanoEmX(Ponto3D centroPlano, float comprimento,
float altura, int orientacao) {

Calcula coordendas dos pontos A,B,C e D
```

```
if (orientacao == 1) {
Coloca pontos pela ordem A-B-C-C-D-A
}
else {
Coloca pontos pela ordem A-D-C-C-B-A
}

return *this;
}
```

De notar que esta função além do comprimento e altura, recebe ainda o centro do plano e a orientação do mesmo.

1.3.2 Gerar Caixa

Para gerar uma caixa, o utilizador deve efetuar o comando com a seguinte sintaxe:

```
Gerador caixa comprimento largura altura ficheiro
```

O resultado deste comando é a criação de uma caixa centrada no ponto (0,0,0) com o comprimento, largura e altura indicados.

Na secção 1.3.1 foram apresentadas as primitivas que permitiam fazer planos. Nomeadamente viu-se que essas primitivas permitiam geral planos em XY, YZ e XZ dado um centro e uma orientação.

Uma caixa é apenas um conjunto de planos. Assim, para gerar a caixa o que se fez foi gerar os seus planos usando as primitivas da secção 1.3.1. No entanto, para usar tais primitivas é necessário indicar um centro do plano e uma orientação para o mesmo. É por isso necessário calcular esses parâmetros antes de usar as primitivas dos planos.

Considere-se a caixa com as faces visíveis identificadas pelas letras A, B e C, conforme mostra a figura 1.5

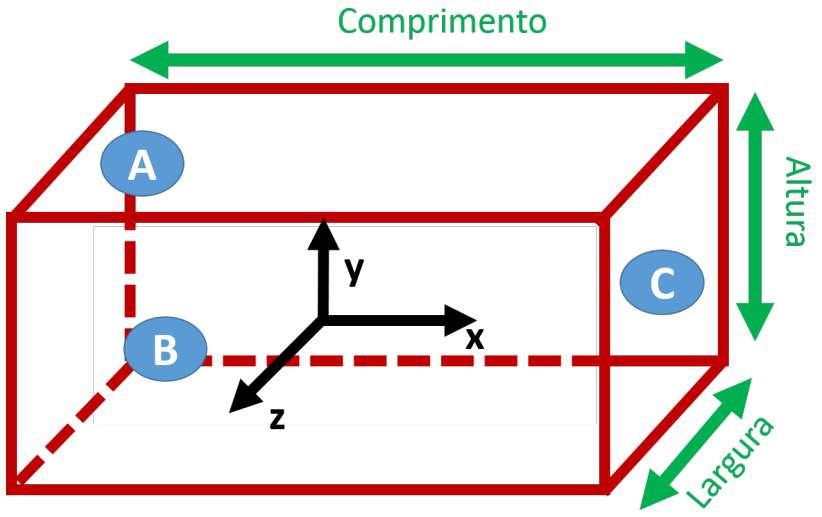

Figura 1.5: Esquema representativo da caixa

Considere-se ainda que a face oposta a A é designada por A', a face oposta a B por B' e a face oposta a C por C'. Interessa agora saber as coordenadas dos centros de cada uma destas faces. Considerando a caixa centrada na origem, temos o seguinte:

```
Centro A  = (0,altura/2,0)          Orientação = 1
Centro A' = (0,-altura/2,0)         Orientação = 0
Centro B  = (0,0,largura/2)         Orientação = 1
Centro B' = (0,0,-largura/2)        Orientação = 0
Centro C  = (comprimento/2,0,0)     Orientação = 1
Centro C' = (-comprimento/2,0,0)    Orientação = 0
```

O valor da orientação toma o valor 1 se o plano estiver virado no sentido positivo do eixo sobre o qual esta colocado.

É fácil verificar que caso a caixa não esteja centrada na origem mas esteja centrada num ponto P(px, py, pz) as coordenadas dos centros das faces serão:

```
Centro A  = (0 + px,altura/2 + py,0 + pz)          Orientação = 1
Centro A' = (0 + px,-altura/2 + py,0 + pz)         Orientação = 0
Centro B  = (0 + px,0 + py,largura/2 + pz)         Orientação = 1
Centro B' = (0 + px,0 + py,-largura/2 + pz)        Orientação = 0
Centro C  = (comprimento/2 + px,0 + py,0 + pz)     Orientação = 1
Centro C' = (-comprimento/2 + px,0 + py,0 + pz)    Orientação = 0
```

A função da classe *Figura* responsável por gerar os pontos de uma caixa é a função *criaCaixa()* cujo pseudo-código se apresenta a seguir:

```
Figura& criaCaixa(Ponto3D centroCaixa,
float dx, float dy, float dz) {

Calcula coordendas centro A
Cria plano com centro em A e orientacao = 1

Calcula coordendas centro A'
Cria plano com centro em A' e orientacao = 0

Calcula coordendas centro B
Cria plano com centro em B e orientacao = 1

Calcula coordendas centro B'
Cria plano com centro em B' e orientacao = 0

Calcula coordendas centro C
Cria plano com centro em C e orientacao = 1

Calcula coordendas centro C'
Cria plano com centro em C' e orientacao = 0

return *this;
}
```

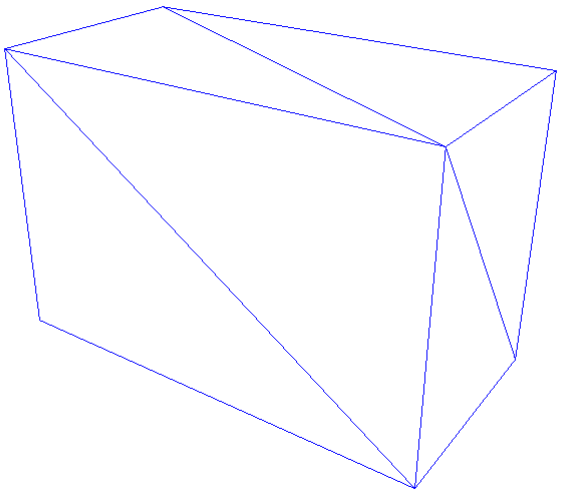

Figura 1.6: Exemplo de caixa gerada, com 6 de comprimento, 3 de largura e 4 de altura

1.3.3 Gerar Círculo

Um círculo corresponde apenas a um conjunto de triângulos em que o centro do círculo é um ponto comum a todos os triângulos. Na figura 1.7 destacado a vermelho mostra-se um desses triângulos:

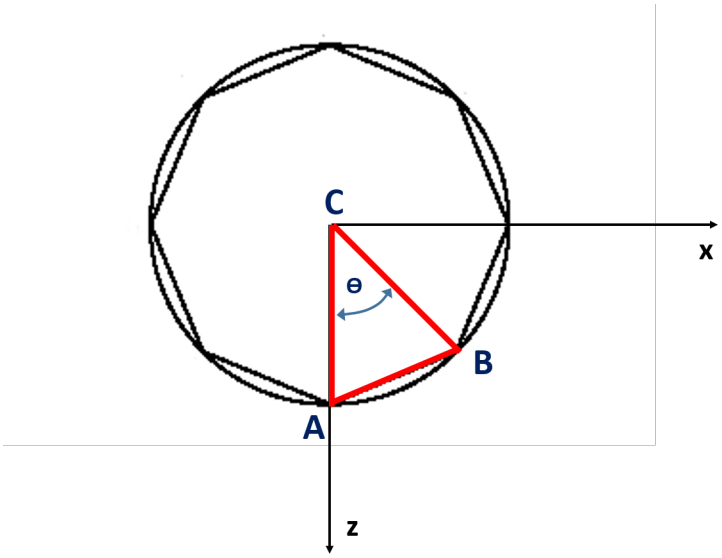

Figura 1.7: Esquema representativo do círculo

Para se saber os pontos de um círculo foram usadas coordenadas polares. Para tal a classe *CoordsPolares* foi usada (ver secção 2.2).

O número de triângulos de um círculo corresponde ao número de "fatias" que se pretende e com isso é possível calcular a variação do angulo θ em cada iteração para se obter os pontos.

Tal como o plano, um círculo também tem uma orientação. Caso se pretenda desenhar o círculo virado no sentido positivo do eixo dos Y, a ordem de colocação dos pontos deverá ser C-A-B. Caso se pretenda desenhar o círculo virado no sentido negativo do eixo dos Y, a ordem de colocação dos pontos deverá ser C-B-A.

A função responsável pelo desenho de círculos é a função *criaCirculo()*, cujo pseudo-código é o seguinte:

```
Figura& criaCirculo(Ponto3D C, float raio, int fatias,
int orientacao) {

CoordsPolares A, B;
float dAz = (2.0 * M_PI) / (fatias + 0.0f);

if (orientacao == 1) {
for (int i = 0; i < fatias; i++) {
A = CoordsPolares(C, raio, dAz*i);
B = CoordsPolares(C, raio, dAz*(i + 1));

Gera pontos pela ordem C-A-B
}
}
else {
for (int i = 0; i < fatias; i++) {
A = CoordsPolares(C, raio, dAz*i);
B = CoordsPolares(C, raio, dAz*(i + 1));

Gera pontos pela ordem C-A-B
}
}


return *this;
}
```

1.3.4 Gerar Cone

Para gerar um cone, o utilizador deve efetuar o comando com a seguinte sintaxe:

```
Gerador cone raio altura fatias camadas ficheiro
```

A base do cone corresponde a um círculo virado no sentido negativo dom eixo dos Y. Como apresentado na secção 1.3.3, a função *criaCirculo()* permite fazer isto. Resta apenas gerar os pontos para a superfície "curva" do cone. Para essa superfício, abordagem seguida foi a de fazer o cone camada a camada. As fatias partem cada camada numa espécie "rectângulos" como o que se mostra na figura 1.8.

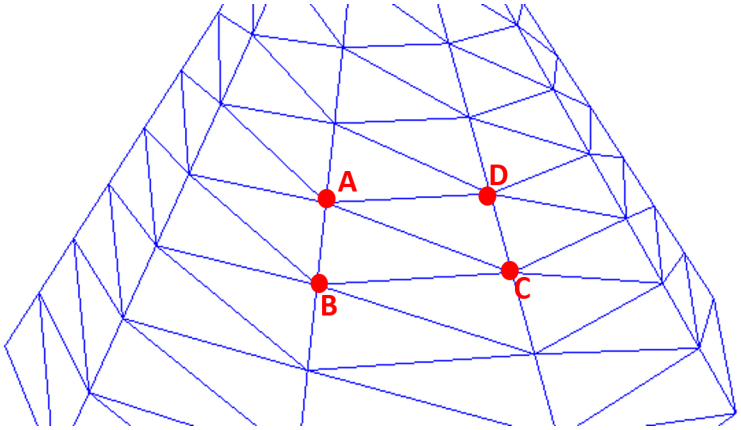

Figura 1.8: Esquema representativo de uma fatia de uma camada do cone (Pontos A,B,C e D)

Ou seja, é possível construir uma fatia de uma dada camada com os pontos A,B,C e D. É no entanto necessário saber as coordenadas destes pontos. Para tal, foram usadas coordenadas polares, com auxílio mais uma vez da classe *CoordsPolares*. Podemos ver os pontos A e D como pertencentes a um mesmo círculo, com centro no centro do cone. Da mesma forma, também os pontos B e C se encontram num mesmo círculo, com raio maior do que o círculo onde estão os pontos A e D. O raio dos dois círculos em que estes dois conjuntos de pontos se encontram difere, no entanto é fácil perceber que o raio do círculo depende da camada que se está a considerar. Por exemplo, se virmos a base do cone como um círculo de raio r, então o círculo correspondente à camada de cima terá raio $r - (r/camadas)$. A figura 1.9 pretende ilustrar tal situação para um exemplo de 3 camadas.

Saber o centro de cada um dos círculos do cone também é simples. Os centros dos círculos encontram-se todos no centro do cone, a única coisa que varia em cada um é a coordenada Y cuja diferença para a coordenada Y da camada anterior é $altura/camadas$.

Visto que a classe de *CoordsPolares* é capaz de dar coordenadas cartesianas sendo indicado um centro, um raio e um ângulo e tendo em conta o exposto anteriormente então a função criaCone() fica simplesmente:

```
Figura& criaCone(Ponto3D centroCone, float altura,
float raio, int camadas, int fatias) {

float dRaio = raio / (camadas + 0.0f);
float dAltura = altura / (camadas + 0.0f);
float dAz = (2.0 * M_PI) / (fatias + 0.0f);
float meiaAltura = altura / 2.0f;
```

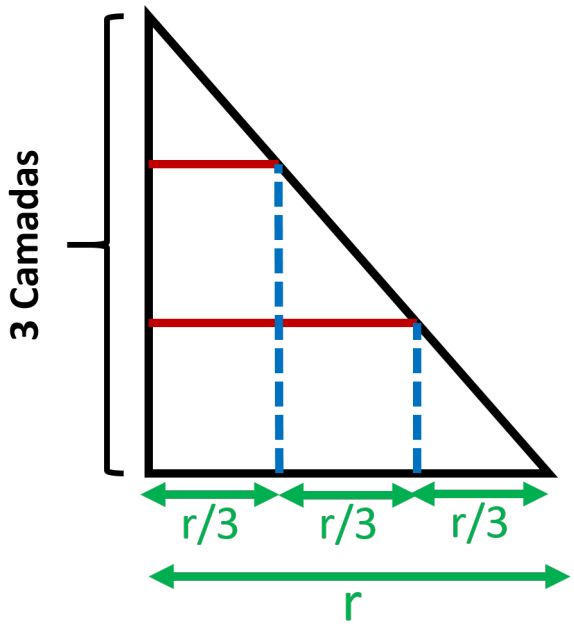

Figura 1.9: O círculo da camada superior à que se considera tem sempre raio $r - (r/camadas)$

```
Calcula coordendadas do centro da base

Cria circulo no centro da base virado no sentido
negativo do eixo dos Y.

for (int i = 0; i < camadas; i++) {
for (int j = 0; j < fatias; j++) {
Calcula o centro do circulo "de baixo" da
camada que se está a considerar

Calcula o centro do circulo "de cima" da
camada que se está a considerar

A = CoordsPolares(cCima,
raio - (dRaio*(i + 1.0)),
dAz * j);
B = CoordsPolares(cBaixo,
raio - (dRaio*(i + 0.0)),
dAz * j);
C = CoordsPolares(cBaixo,
raio - (dRaio*(i + 0.0)),
dAz * (j + 1.0));
D = CoordsPolares(cCima,
raio - (dRaio*(i + 1.0)),
dAz * (j + 1.0));

Gera os pontos pela ordem A-B-C-C-D-A
```

```
}
}

return *this;
}
```

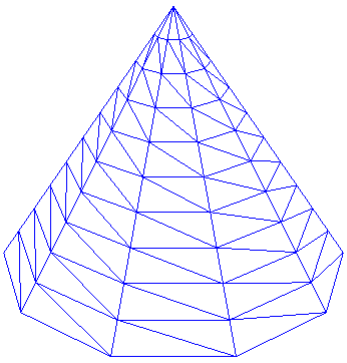

Figura 1.10: Exemplo de cone gerado, com 2 de raio de base, 3 de altura, 10 camadas e 10 fatias

1.3.5 Gerar Esfera

Para gerar uma esfera, o utilizador deve efetuar o comando com a seguinte sintaxe:

```
Gerador sphere raio fatias camadas ficheiro
```

Os pontos da esfera foram gerados à custa de coordenadas esféricas, nomeadamente com o auxílio da classe *CoordsEsfericas*.

À semehança do cone também para a esfera é possível gerar uma fatia de uma determinada camada usando 4 pontos conforme mostrado na figura 1.11.

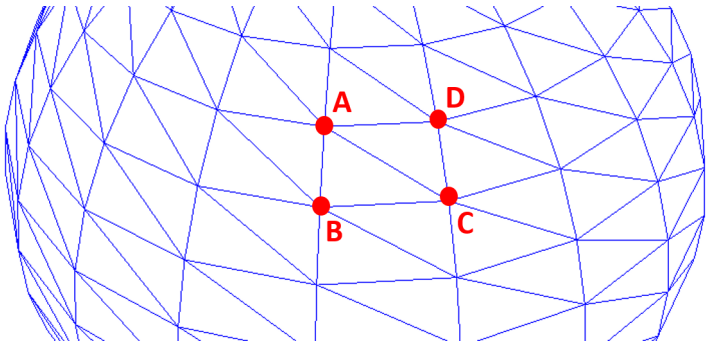

Figura 1.11: Esquema representativo de uma fatia de uma camada da esfera (Pontos A,B,C e D)

Também para a esfera a abordagem seguida foi a de construir camada a camada, começando pela camada de cima. A determinação das coordenadas dos pontos A, B, C e D é simples. Uma vez que se está a usar a classe *CoordsEsfericas*, a única coisa que se tem que perceber para usar a classe é de que forma variam os ângulos polar e azimuth entre camadas e fatias. Ora, tendo c camadas, sabe-se que a diferença em termos de ângulo polar de um ponto que esteja numa camada superior para outro que esteja numa camada imediatamente inferior é de π/c. Por outro lado, se considerarmos f fatias, a diferença do ângulo azimuth de um ponto para outro que esteja numa fatia imediatamente a seguir é de $(2 \times \pi)/f$. Assim temos o seguinte pseudo-código:

```
Figura& criaEsfera(int fatias, int camadas, float raio) {

float deltaPolar = M_PI / (camadas+0.0f);
float deltaAz = (2.0 * M_PI) / (fatias+0.0f);

for (int i = 0; i < camadas; i++) {
for (int j = 0; j < fatias; j++) {
A = CoordsEsfericas(raio,
deltaAz * (j + 0.0f),
deltaPolar * (i+0.0f));
B = CoordsEsfericas(raio,
deltaAz * (j + 0.0f),
deltaPolar * (i+1.0f));
C = CoordsEsfericas(raio,
deltaAz * (j + 1.0f),
deltaPolar * (i+1.0f));
D = CoordsEsfericas(raio,
```

```
deltaAz * (j + 1.0f),
deltaPolar * (i+0.0f));

Gera pontos pela ordem A-B-C-C-D-A

}
}

return *this;
}
```

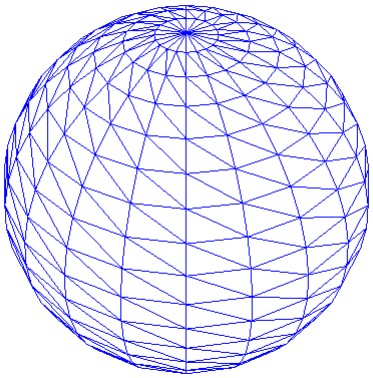

Figura 1.12: Exemplo de esfera gerada, com 2 de raio, 20 camadas e 20 fatias

2. Gerador1

Nesta secção apresentam-se as classes usadas. Tanto o motor como o gerador recorrem extensivamente a estas classes para desempenhar as suas funções. Algumas destas classes são recorrentes e usadas mais que uma vez em diferentes contextos. Assim, antes de introduzir como funciona o motor e o gerador é importante primeiro perceber quais são as classes que ambos usam, bem como a função de cada uma.

O objectivo de construção destas classes foi simplificar a construção do motor e gerador apenas. Nesta primeira etapa, foram deixadas para segundo plano questões como a modularidade e encapsulamento de dados, embora seja algo a considerar em etapas posteriores do trabalho.

2.1 Ponto3D

Tanto o gerador como o motor trabalham com pontos. Assim, foi criada uma classe que representa o conceito de um ponto num espaço 3D:

```
struct Ponto3D {
        float x, y, z;
};
```

Como se pode verificar, esta classe é bastante simples e apenas contém 3 variáveis do tipo *float*, correspondentes às coordenadas cartesianas de um ponto num espaço 3D.

Visto nenhuma das aplicações exigir operações complexas sobre pontos, esta classe foi mantida simples. De facto, tanto motor como gerador, apenas precisam de ler valores de coordenadas de pontos e ocasionalmente mudar os seus valores, o que pode ser feito directamente acedendo às variáveis desta classe.

O facto desta classe ser simples e não ter nenhuma função ou construtor faz com que seja considerada pelo C++ como uma *aggregate class*. Este tipo de classe tem propriedades específicas, nomeadamente uma inicialização fácil e intuitiva das instâncias, o que contribui também para código mais fácil de ler. Esse fator foi fundamental para a decisão de manter a classe simples.

2.2 CoordsPolares

Esta classe pretende abstrair os cálculos relacionados com coordenadas polares. A ideia é poderem ser criadas instâncias desta classe indicando os parâmetros correspondentes a coordenadas polares e as correspondentes coordenadas cartesianas poderem ser obtidas facilmente a partir da instância.

As coordenadas polares são constituídas por um ponto central, um raio e por um ângulo α (também designado por ângulo azimuth) e permitem localizar um ponto num plano conforme se ilustra na figura 2.1:

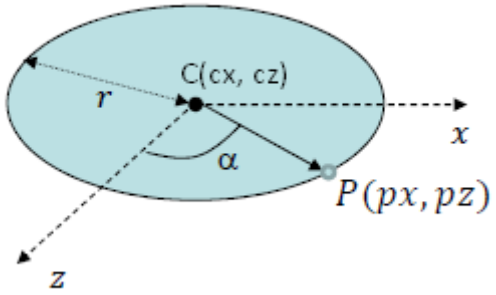

Figura 2.1: Localização de um ponto P(px,py,pz) segundo um centro C(cx,cy,cz), um ângulo azimuth (α) e um raio r

Designando o ponto central como C(cx,cy,cz) e um ponto P(px,py,pz) que se pretende localizar, temos que as coordenadas px e py podem ser obtidas da seguinte forma:

$$px = cx + r \times sin(\alpha)$$
$$py = cy$$
$$pz = cz + r \times cos(\alpha)$$

A classe *CoordsPolares* representa apenas uma forma fácil de trabalhar com este sistema de coordenadas. Esta classe possui variáveis de instância relacionadas com as suas coordenadas polares e cartesianas:

```
//Centro a partir do qual se
//considera as coordenadas polares
Ponto3D centro;
//Coordenadas polares
float raio, azimuth;
//Coordenadas rectangulares correspondentes
//às coordenadas polares
Ponto3D cCartesianas;
```

O ponto chave desta classe é que as variáveis das coordenadas polares e cartesianas referem-se sempre ao mesmo ponto. Sempre que um dos parâmetros é alterado (através de uma das funções disponibilizadas), todos os restantes são actualizados em conformidade.

A classe tem um único construtor:

```
CoordsPolares(Ponto3D c, float r, float az);
```

Neste construtor são pedidos os parâmetros referentes às coordenadas polares. O primeiro parâmetro *c* corresponde ao ponto central, o segundo parâmetro r corresponde ao raio e o último parâmetro corresponde ao ângulo azimuth (α). Ao receber estas coordenadas polares, o construtor seguidamente atualiza as variáveis de instância em conformidade, nomeadamente coloca na variável de instância *cCartesianas* as coordenadas cartesianas correspondentes às coordenadas polares passadas como argumento. Essa actualização é feita pela função *refreshCartesianas()*:

```
void refreshCartesianas(){
        cRectangulares.x = centro.x + raio * sin(azimuth);
        cRectangulares.y = centro.y;
        cRectangulares.z = centro.z + raio * cos(azimuth);
}
```

Como se pode ver, esta função implementa apenas as fórmulas apresentadas anteriormente. Esta função é privada à classe, por isso não pode ser chamada em qualquer parte do código. É a própria classe que se responsabiliza por chamar esta função sempre que necessário.

Assim, dada uma instância desta classe é sempre possível saber as coordenadas cartesianas correspondentes através da função *toCartesianas()*:

```
Ponto3D toCartesianas() {
        return cCartesianas;
}
```

2.3 CoordsEsfericas

Esta classe pretende abstrair os cálculos relacionados com coordenadas esféricas. A ideia é poderem ser criadas instâncias desta classe indicando os parâmetros correspondentes a coordenadas esféricas e as correspondentes coordenadas cartesianas poderem ser obtidas facilmente a partir da instância. É também possível fazer o inverso, ou seja, indicar um ponto com coordenadas cartesianas e obter as respectivas coordenadas esféricas.

As coordenadas esféricas são constituídas por um raio r, por um ângulo θ (também designado por ângulo azimuth) e um ângulo ϕ (também designado por ângulo polar), conforme apresentado na figura 2.2:

É possível saber as coordenadas cartesianas de um ponto P(px,py,pz) a partir das suas coordenadas esféricas através das seguintes fórmulas:

$$px = r \times cos(\theta) \times sin(\phi)$$
$$py = r \times sin(\theta) \times sin(\phi)$$
$$pz = r \times cos(\phi)$$

Por outro lado, consegue-se saber também as coordenas esféricas de um ponto a partir das suas coordenadas cartesianas de acordo com as seguintes fórmulas:

$$r = \sqrt{px^2 + py^2 + pz^2}$$
$$\theta = \arctan(py \backslash px)$$
$$\phi = \arccos(pz \backslash r)$$

De referir que ao contrário da classe *CoordsPolares*, nesta classe optou-se por não se considerar um centro. Assume-se o centro como sendo (0.0,0.0,0.0) sempre. Considerou-se esta

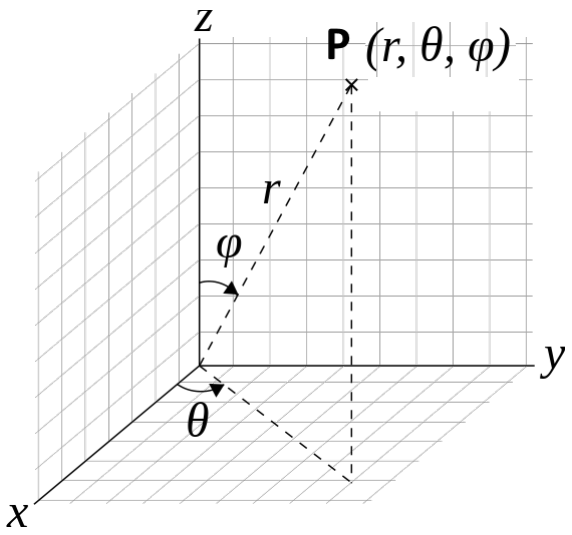

Figura 2.2: Localização de um ponto P segundo coordenadas esféricas

simplificação razoável na medida em que responde aos requisitos do motor e gerador nesta primeira fase.

A classe *CoordsEsfericas* tem as seguintes variáveis de instância:

```
// Coordenadas Esfericas
float raio, azimuth_ang, polar_ang;
// Coordenadas cartesianas
Ponto3D cCartesianas;
```

O ponto chave desta classe é que as variáveis das coordenadas esféricas e cartesianas referem-se sempre ao mesmo ponto. Sempre que um dos parâmetros é alterado (através de uma das funções disponibilizadas), todos os restantes são atualizados em conformidade.

Uma instância da classe *CoordsEsfericas* pode ser criada indicando os parâmetros das coordenadas esféricas (para se saber as suas coordenadas cartesianas), ou indicando um ponto em coordenadas cartesianas (do qual se pretende saber as coordenadas esféricas), através dos seguintes construtores:

```
CoordsEsfericas(float r, float az, float polar);
CoordsEsfericas(Ponto3D pto);
```

Caso a instância seja criada a partir de coordenadas polares, o construtor calcula as coordenadas cartesianas correspondentes através da função *refreshCartesianas()*:

```
void refreshCartesianas() {
        cCartesianas.z = raio * sin(polar_ang) * cos(azimuth_ang);
        cCartesianas.x = raio * sin(polar_ang) * sin(azimuth_ang);
        cCartesianas.y = raio * cos(polar_ang);
}
```

Caso a instância seja criada a partir de coordenadas cartesianas, o construtor calcula as coordenadas polares correspondentes através da função *refreshEsfericas()*:

```
void refreshEsfericas() {
        raio = sqrt(pow(cCartesianas.x, 2) +
                pow(cCartesianas.y, 2) +
                pow(cCartesianas.z, 2));
        polar_ang = acos(cCartesianas.y / raio);
        azimuth_ang = atan2(cCartesianas.x, cCartesianas.z);
}
```

Estas duas funções correspondem à implementação das fórmulas apresentadas anteriormente e garantem que as variáveis de instância correspondentes às coordenadas esféricas e cartesianas se referem sempre ao mesmo ponto. Ambas são funções privadas, pelo que é a própria classe que tem a responsabilidade de as chamar sempre que é necessário atualizar valores.

A qualquer momento, é possível saber as coordenadas cartesianas de uma instância pela função *toCartesianas()*

```
Ponto3D toCartesianas() {
        return cCartesianas;
}
```

Além destas funções, a classe possui ainda funções adicionais que permitem mudar a posição do ponto representado por cada instância. Sempre que uma destas funções é chamada tanto as variáveis de instância das coordenadas polares e cartesianas são atualizadas quer pela função refreshEsfericas() ou refreshCartesianas(). Estas funções que permitem mudar a localização do ponto, garantem ainda que $0 \leq \phi \leq \pi$ e que $0 \leq \theta \leq 2 \times \pi$

2.4 Figura

Esta classe representa uma figura que será desenhada pelo motor. Representa por isso apenas um conjunto de pontos numa determinada ordem que correspondem a triângulos, que por sua vez formam uma figura.

Por representar um conjunto de pontos, sem surpresa, a sua única variável de instância é um vector de pontos:

```
std::vector<Ponto3D> pontos;
```

A utilidade desta classe revela-se pelas funções que disponibiliza. Em primeiro lugar, disponibiliza um conjunto de funções que quando chamadas colocam no vector *pontos* os pontos necessários para o desenho de uma figura em concreto. Dessa forma, estas funções permitem criar planos, caixas, círculos, cilindros e esferas. Estas funções serão explicadas em mais detalhe quando for apresentado o gerador onde será mostrado de que forma estas funções podem ser chamadas bem como de que forma geram os pontos.

Além das funções que permitem criar figuras destaca-se ainda a função *toFicheiro()*, que permite guardar os pontos da figura num ficheiro cujo nome é passado como argumento.

```
void toFicheiro(std::string filePath)
```

É ainda possível obter os pontos da figura pela função *getPontos()*:

```
std::vector<Ponto3D> getPontos()
```

2.5 TinyXML-2

Esta biblioteca foi usada para auxílio à leitura de ficheiros XML por parte do motor e pode ser encontrada no endereço: http://www.grinninglizard.com/tinyxml2/

3. Motor

3.1 Objectivos

Com o motor pretende-se uma aplicação que seja capaz de ler um conjunto de pontos especificados em ficheiros XML e .3d e os desenhe numa janela. Além desse objetivo principal foi incluída uma câmara esférica em torno do objeto desenhado e um menu simples que permite alterar o modo de visualização dos pontos.

Nesta secção apresenta-se de que forma o motor foi desenvolvido, começando-se por explicar a leitura dos ficheiros

3.2 Leitura Ficheiros

3.2.1 Ficheiro XML

A estrutura do ficheiro XML contempla apenas dois tipos de elementos:

scene - elemento pai

model - elemento com 1 atributo *file* cujo valor corresponde ao nome de um ficheiro de pontos

A leitura do ficheiro XML é feita pela função:

```
vector<const char *> leXML()
```

O valor de retorno da função é um vector de *char* * que contém os nomes dos ficheiros a ser lidos.

O pseudo-codigo da função de leitura do XML pode ser expresso da seguinte forma:

```
vector<const char *> leXML() {
        Abre documento para leitura

        if (nao houve erro a abrir o ficheiro) {
                Acede ao primeiro elemento filho de "scene"
                (que é um elemento "model")

                while (houver elementos "model") {
                        Vê qual o valor do primeiro atributo
                        do elemento

                        Coloca o valor do primeiro atributo do
```

```
                        elemento no resultado

                    Avança para o proximo elemento
                }
        }
        else {
                Informa erro a abrir o ficheiro
        }

        Retorna resultado
}
```

3.2.2 Ficheiros .3d

Ns ficheiros .3d, cada linha representa um ponto. Em cada linha, existem 3 valores, separados por espaço, correspondentes às coordenadas cartesianas do ponto. É necessário por isso ter uma função que seja capaz de ler os pontos destes ficheiros para que o motor os possa desenhar.

Como referido anteriormente, a função de leitura do XML devolve um vector com os nomes dos ficheiros .3d onde se encontram os pontos a desenhar. Tendo esta lista de ficheiros, o main chama depois a função le3D para cada um dos ficheiros:

```
le3D(const char * ficheiro)
```

Esta função recebe como parâmetro o nome de um ficheiro e coloca os pontos do ficheiro num vector de pontos (que é uma variável global). Este vector de pontos será depois o que o motor vai utilizar para desenhar os pontos.

A função le3D() funciona da seguinte forma:

```
void le3D(const char * ficheiro) {
        Abre ficheiro para leitura

        while (houver linhas para ler) {
                Coloca numa string o conteudo da linha

                Lê os conteudos da linha

                Faz um Ponto3D correspondente à linha

                Coloca ponto no vector de pontos
        }

}
```

3.3 Desenho dos pontos

O resultado da leitura do XML e dos ficheiros .3d nele especificados é um conjunto de *Pontos3D* que são armazenados numa variável global. Este conjunto de pontos está implementado sobre a forma de um *vector<Ponto3D>* que é uma variável global ao programa.

A ordem pela qual os pontos aparecem no vector é a ordem pela qual apareceram nos ficheiros, que por sua vez é a ordem pela qual deverão ser desenhados. Nesta situação, desenhar os pontos corresponde simplesmente a percorrer todos os pontos colocados no vector e a pedir ao GLUT para os desenhar. Mostra-se um excerto de código da função renderScene() que corresponde ao desenho dos pontos:

```
glBegin(GL_TRIANGLES);
glColor3f(0.0, 0.0, 1.0);
for (auto it = pontos.begin(); it != pontos.end(); ++it) {
        glVertex3f(it->x, it->y, it->z);
}
glEnd();
```

Assume-se que os ficheiros .3d já contêm a ordem correta dos pontos a ser desenhados.

3.4 Câmara

Para se poder ver o objeto desenhado de vários ângulos foi implementada uma câmara colocada sobre uma esfera. Visto que a posição da câmara é um ponto numa esfera, foram usadas coordendas esféricas para gerir o movimento da câmara.

Na secção 2.3 foi apresentada a classe *CoordsEsfericas* que permite torna fácil operações sobre este tipo de coordenadas e respectiva conversão para coordenas cartesianas. Por este motivo, esta classe é ideal para auxiliar a implementação da câmara.

A câmara é representada por uma variável global declarada da seguinte forma:

```
CoordsEsfericas camara;
```

À função *gluLookAt()* são passadas as coordenadas cartesianas correspondentes às coordenadas esféricas:

```
gluLookAt(camara.cCartesianas.x,
        camara.cCartesianas.y,
        camara.cCartesianas.z,
        0.0, 0.0, 0.0,
        0.0f, 1.0f, 0.0f);
```

É possível ao utilizador mudar a posição da câmara. Os controlos são os seguintes:

Tecla 'W' permite ao utilizador ir para cima

Tecla 'S' permite ao utilizador ir para baixo

Tecla 'A' permite ao utilizador ir para a esquerda

Tecla 'D' permite ao utilizador ir para a direita

Tecla 'Q' permite ao utilizador afastar-se do objeto

Tecla 'E' permite ao utilizador aproximar-se do objeto

Estas operações traduzem-se em operações sobre a câmara disponibilizadas pela classe *CoordsEsfericas*:

```
void f_teclas_normais(unsigned char key, int x, int y) {
        switch (tolower(key)) {
                case 'w': camara.paraCima(2 * M_PI / 360.0);
                break;
                case 's': camara.paraBaixo(2 * M_PI / 360.0);
                break;
                case 'a': camara.paraEsquerda(2 * M_PI / 360.0);
                break;
                case 'd': camara.paraDireita(2 * M_PI / 360.0);
                break;
                case 'e': camara.aproximar(0.5);
                break;
                case 'q': camara.afastar(0.5);
                break;
        }
        glutPostRedisplay();
}
```

O efeito de aplicar uma destas operações é a classe *CoordsEsfericas* actualizar os valores das coordenadas esféricas e cartesianas da nova localização da câmara. Assim, quando a função *gluLookAt()* é chamada e acede aos valores de x, y e z da posição da câmara, estes estão atualizados.

3.5 Menu

Para proporcionar alguma flexibilidade na visualização das figuras desenhadas foi incluído um menu que permite ao utilizador alternar entre o modo de visualização de pontos, linhas, ou objeto preenchido:

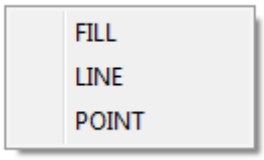

Figura 3.1: Menu de opções de visualização. Acessível com clique no botão direito do rato

O menu foi construído da seguinte forma:

```
glutCreateMenu(menuDrawing);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutAddMenuEntry("FILL", 0);
glutAddMenuEntry("LINE", 1);
glutAddMenuEntry("POINT", 2);
```

O modo de visualização é controlado por uma variável global:

```
GLenum modoPoligonos;
```

Quando o utilizador escolhe uma das opções, esta variável é modificada:

```
void menuDrawing(int opt) {
        switch (opt) {
                case 0: modoPoligonos = GL_FILL; break;
                case 1:  modoPoligonos = GL_LINE; break;
                case 2:  modoPoligonos = GL_POINT; break;
        }
        glutPostRedisplay();
}
```

O valor da variável é usado na chamada à função *glPolygonMode()* que se encontra na função *renderScene*:

```
glPolygonMode(GL_FRONT_AND_BACK, modoPoligonos);
```

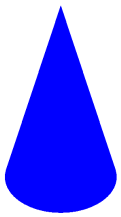

Figura 3.2: Exemplo de cone construído no modo GL_FILL

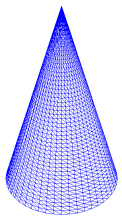

Figura 3.3: Exemplo de cone construído no modo GL_LINE

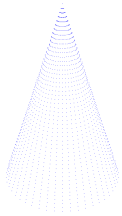

Figura 3.4: Exemplo de cone construído no modo GL_POINT

4. Conclusão

O trabalho apresentado cumpre todos os requisitos propostos. Foi feito um gerador de figuras com capacidade de gerar pontos para um plano, caixa, cone e esfera como pedido. Além destas figuras foram ainda desenvolvidas funções adicionais que permitem criar planos sem ser no eixo XZ bem como foram desenvolvidas funções para a criação de círculos e cilindros, algo que não era expressamente pedido.

No lado do motor, a aplicação consegue ler ficheiros XML e .3D e a partir dai desenhar as figuras com os pontos especificados nos ficheiros. Adicionalmente ao sugerido, incluiu-se também uma câmara colocada sobre uma esfera que permite ao utilizador ver a figura desenhada de vários ângulos. Incluiu-se ainda um pequeno menu para alterar o modo de visualização da figura.

O código produzido recorreu ao uso de classes, o que evitou bastantes repetições de código e sobretudo gerou um código fácil de ler, perceber e manter. Por estes motivos, considera-se como bastante sólido o trabalho desenvolvido.

Não obstante, existem aspectos em que o trabalho que poderiam ser melhorados e serão alvo de atenção no futuro.

Em primeiro lugar, destaca-se a questão da câmara. Nesta fase implementou-se a câmara usando coordenadas esféricas. Decidiu-se que a camâra seria por isso apenas uma instância da classe *CoordsEsfericas*. Deste modo mover a câmara corresponde apenas a chamar as funções definidas na classe. Este aspecto facilitou imenso a implementação da câmara, no entanto trouxe também algumas desvantagens. Sendo a câmara uma instância de *CoordsEsfericas* significa que a câmara só pode ter coordenadas esféricas. Isto dificulta a adição de funcionalidades extra à câmara. Além disso, enquanto que é perfeitamente válido que as coordenadas esféricas possam referir um ponto "no polo norte" da esfera, tal não é verdade para a câmara, pois nesse caso o objeto pode deixar de se tornar visível. Isto deixa a entender que no futuro a câmara terá que pertencer a uma classe própria e muito provavelmente será esse o caminho a seguir.

Em segundo lugar, refere-se a modularidade e encapsulamento de dados, aspectos que foram deixados um pouco para segundo plano. A prioridade desde cedo foi ter código simples, funcional e fácil de ler. Tal implicou que muitas vezes quando confrontados com a decisão de manter algumas variáveis como públicas ou privadas a decisão tenha sido manter públicas. Exemplos disso são as classes Ponto3D (note-se a falta de getters e setters) e as classes das coordenadas esféricas e polares. Enquanto que ter variáveis públicas em classes como a Ponto3D seja relativamente irrelevante, tal já não é verdade para as classes das coordenadas. Como o objetivo principal do projeto não é ter bons módulos de dados nem bom encapsulamento dos mesmos, estes aspectos foram deixados para segundo plano, no entanto serão alvo de uma atenção mais cuidada no futuro.