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, 6 de Março de 2017

Conteúdo

1 Introdução **1**

2 Gerador **2**
2.1 Funcionamento 2
2.2 Algoritmo de Geração de Pontos 3
2.2.1 Gerar Planos 3
2.2.2 Gerar Caixa 6
2.2.3 Gerar Cone 7
2.2.4 Gerar Esfera 10

3 Motor **12**
3.1 Objectivos 12
3.2 Leitura Ficheiros 12
3.2.1 Ficheiro XML 12
3.2.2 Ficheiros .3d 13
3.3 Desenho dos pontos 13
3.4 Câmara 14
3.5 Menu 15

4 Conclusão **17**

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. Introdução

No âmbito da Unidade Curricular de Computação Gráfica pertencente ao plano de estudos do 3º ano do Mestrado Integrado em Engenharia Informática, foi proposto o desenvolvimento de um mini motor 3D.

Nesta primeira fase o motor 3D será o responsável pelo desenho dos modelos 3D armazenados em ficheiros diferentes. Para além do motor será criado um gerador que recebe como parâmetros o nome das primitivas gráficas e os dados necessários à sua criação. O gerador deverá escrever num ficheiro os pontos indispensáveis ao desenho da primitiva.

O gerador deverá ser capaz de gerar um plano, um cubo (caixa), um cone e uma esfera. Os ficheiros gerados deverão conter os pontos dos triângulos, em que cada linha corresponde a um ponto e a cada 3 linhas a um triângulo.

2. Gerador

O *generator* é um programa 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. A lista de pontos está ordenada de forma a formarem triângulos que juntos desenham a figura desejada.

Nesta primeira etapa, o objetivo implementar o suporte à criação de um plano, uma caixa, um cone e uma esfera.

2.1 Funcionamento

Nesta primeira fase o *generator* é capaz de criar diversas primitivas gráficas diferentes, entre as quais: *plane* (plano), *box* (paralelepípedo), *sphere* (esfera), *cone*. Para cada uma das primitivas geométricas é possível gerar o ficheiro correspondente através dos comandos:

- PLANE: generator plano/plane <comprimento largura> <ficheiro>, onde a largura e comprimento são opcionais
- BOX: generator box/caixa <sizeX sixeY sizeZ divisões> <ficheiro>, onde existe a possibilidade de as divisões serem opcionais
- SPHERE: generator sphere/esfera <raio fatias camadas> <ficheiro>
- CONE: generator cone <raio altura fatias camadas> <ficheiro>

Para a criação das formas geométricas é preciso obedecer a algumas restrições com a passagem de argumentos, o primeiro parâmetro a ser passado deverá ser o nome da figura a desenhar, o último será o nome do ficheiro onde será guardada a lista de pontos, sendo a extensão ".3d" adicionada automaticamente pelo generator. Sendo isto obedecido, o número de argumentos entre o nome da figura e o nome do ficheiro varia consoante o tipo de figura a ser criada, mas esses argumentos deverão ser numerais.

2.2 Algoritmo de Geração de Pontos

2.2.1 Gerar Planos

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 2.1 representa um plano em XZ. De notar que é possível considerar dois triângulos: o triângulo formado por [ABC] e o triângulo formado por [ACD].

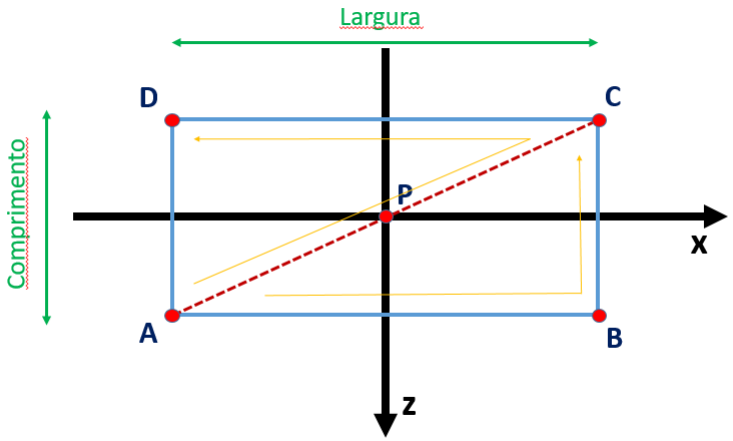

Figura 2.1: Plano em XZ centrado na origem

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura 2.1 são as seguintes:

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

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-C-D (1º triângulo), seguido de A-B-C (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: C-B-A (1º triângulo), seguido de D-C-A (2º triângulo).

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

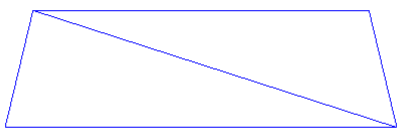

Figura 2.2: Exemplo de plano em XZ gerado, com 4 de comprimento e 2 de largura

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 2.3 representa um plano em XY. De notar que é possível considerar dois triângulos: o triângulo formado por [ABD] e o triângulo formado por [DBC]

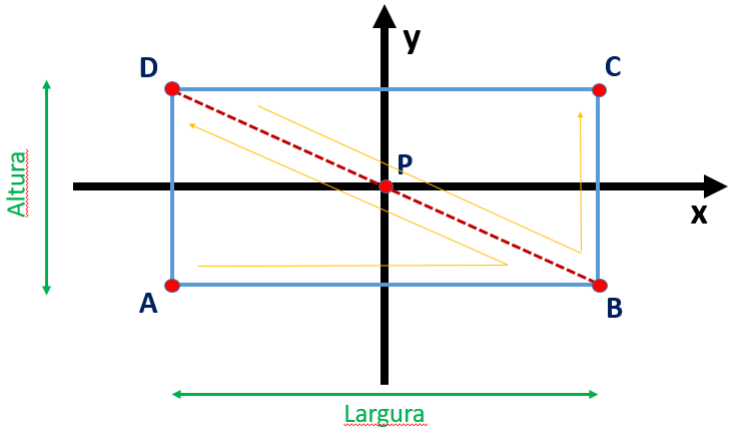

Figura 2.3: Plano em XY centrado na origem

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura 2.3 são as seguintes:

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

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-D (1º triângulo), seguido de D-B-C (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: C-B-D (1º triângulo), seguido de D-B-A (2º triângulo).

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 2.4 representa um plano em YZ. De notar que é possível considerar dois triângulos: o triângulo formado por [ABC] e o triângulo formado por [DAC].

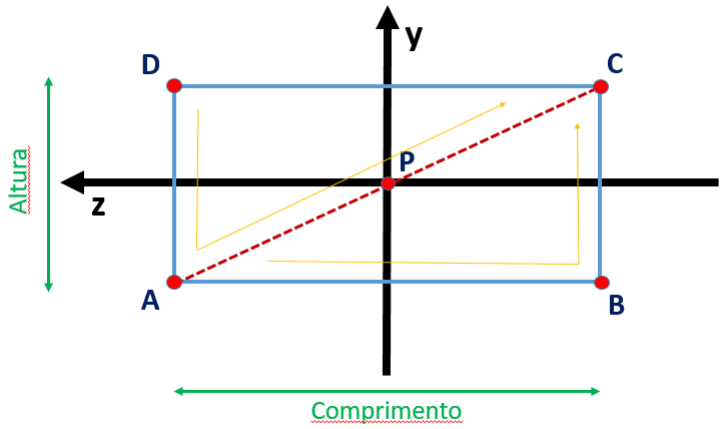

Figura 2.4: Plano em XZ centrado na origem

Estando o plano centrado na origem, e sabendo o comprimento e largura do plano, conclui-se que as coordenadas dos pontos da figura 2.4 são as seguintes:

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

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 D-A-C (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: C-A-D (1º triângulo), seguido de C-B-A (2º triângulo).

A função responsável por implementar estes algoritmos é a função *drawPlane_Points()*, isto é os quando queremos desenhar em ordem a um plano, esse plano tem valor zero, cujo pseudo-código se apresenta de seguida:

```
drawPlane_Points(float sizeX, float sizeY, float sizeZ,
float centerX, float centerY, float centerZ, int divisions,
bool rev, Ponto3D points){

Calcula coordendas dos pontos A,B,C e D

if (rev == 0) {
Desenha os pontos contra-relógio

}
else {
Desenha no sentido do relogio
```

```
}

return número de pontos calculados;
}
```

2.2.2 Gerar Caixa

Na secção 2.2 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 2.2. 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 2.5

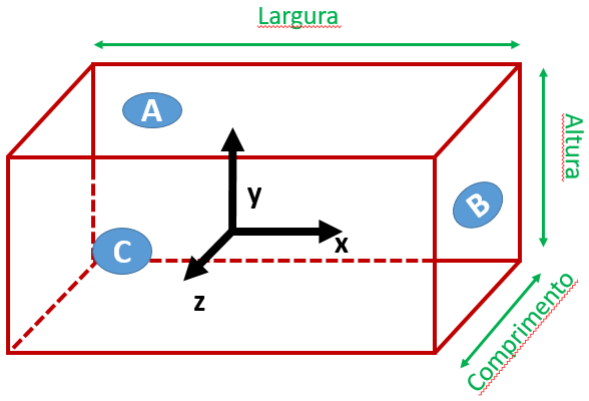

Figura 2.5: Esquema representativo da caixa

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

A função responsável por gerar os pontos de uma caixa é a função *drawBox_Points)* cujo pseudo-código se apresenta a seguir:

```
drawBox_Points(float sizeX, float sizeY, float sizeZ,
```

```
float centerX, float centerY, float centerZ, int divisions, Ponto3D pc

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

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

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

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

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

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

return número de ponto que calculou para fazer a caixa;
}
```

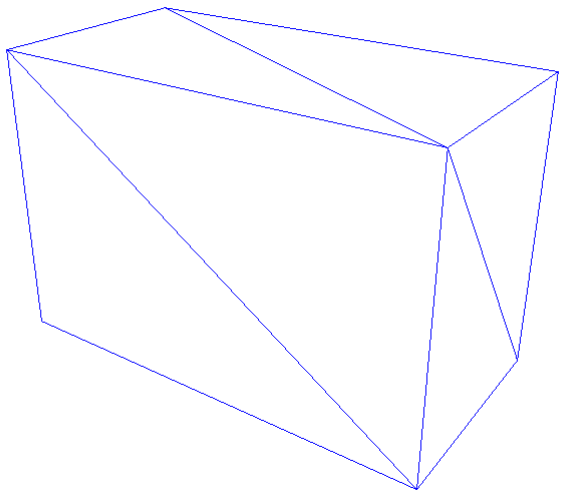

Figura 2.6: Exemplo de caixa gerada, com 6 de comprimento, 3 de largura e 4 de altura

2.2.3 Gerar Cone

A função *drawCone_Points()* 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 2.7.

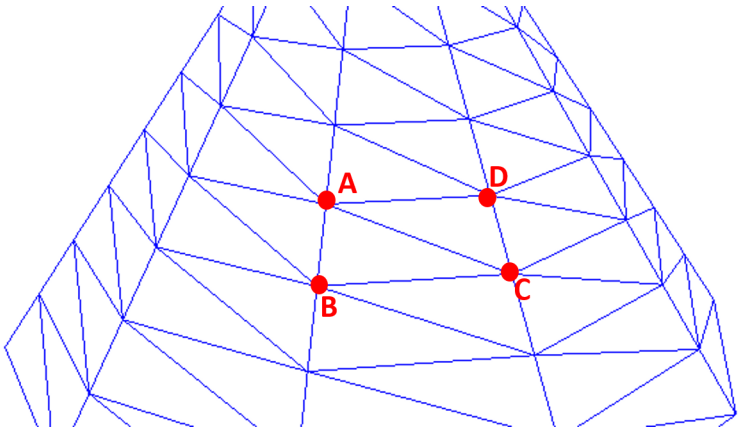

Figura 2.7: Esquema representativo de uma fatia de uma camada do cone (Pontos A,B,C e D)

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. 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 2.8 pretende ilustrar tal situação para um exemplo de 3 camadas.

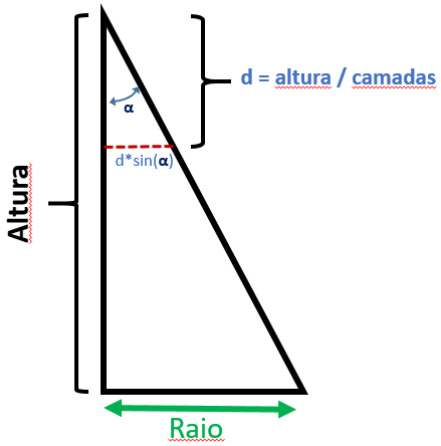

Figura 2.8: O círculo da camada superior à que se considera tem sempre raio $r - (r/camadas)$

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 $altura/camadas$.

```
int drawCone_Points(float radius, float height, float centerX,
```

```
float centerY, float centerZ, int slices, int stacks, Ponto3D points){

for(i = 0;i < slices; i++)
Desenhar os triangulos da cimo do cone

for(i = 2;i <= stacks;i++)
        for(j = 0;j < slices;j++){
                Desenha os quadradros até à base do cone
        }

        for(i = 0; i < slices; i++)
        Desenha o circulo (base do cone)
}
```

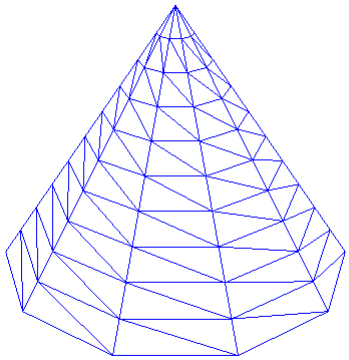

Figura 2.9: Exemplo de cone gerado, com 2 de raio de base, 3 de altura, 10 camadas e 10 fatias

2.2.4 Gerar Esfera

À semelhança do cone também para a esfera é possível gerar uma fatia de uma determinada camada usando 4 pontos conforme mostrado na figura 2.10.

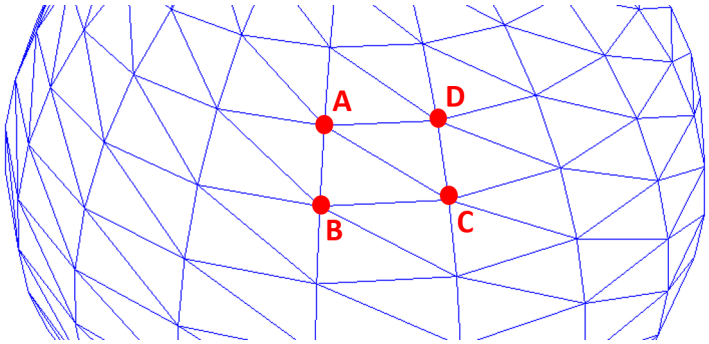

Figura 2.10: Esquema representativo de uma fatia de uma camada da esfera (Pontos A,B,C e D)

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.

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:

```
drawSphere(float radius, float centerX, float centerY, float centerZ,

// calculo do angulo de cada slice e stack, passado para radianos
divBeta = (360.0f / (float) slices) * M_PI / 180.0f;
divAlpha = M_PI / (float) stacks;


for(i = 0;i < slices; i++){
Desenha os triângulos da parte superior da esfera;
Desenha os triângulos da parte inferiro da esfera;
}

for(i = 2;i <= stacks;i++){
        for(j = 0;j < slices;j++){
Desenha quadrados para completar o resto da esfera
        }
}

}
```

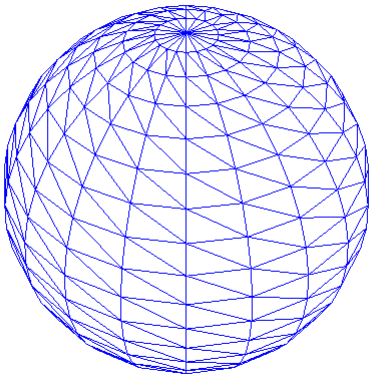

Figura 2.11: Exemplo de esfera gerada, com 2 de raio, 20 camadas e 20 fatias

3. Motor

O *engine* tem como principal função apresentar o modelo gráfico. Para tal utiliza-se a biblioteca *GLUT (OpenGL Utility Toolkit)*, em conjunto com a biblioteca gráfica *OpenGL*.

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.

No main está um pedaço de código que permite a leitura destes ficheiros, que 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 parte do código que lê o ficheiro XML funciona da seguinte forma:

```
if(xml = fopen(argv[1],"r")) {
lê a primeira linha, verifica abertura do scene
while(consegue ler do xml) {
procura os model files e guarda os nomes
}
        fclose(xml);

 } else {
 perror(Impossivel abrir XML);}
```

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 pe-

dir 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 **??** 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 para cima do objeto

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

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

```
void processNormalKeys(unsigned char key, int xx, int yy) {

float fraction = 0.1f;

switch (key) {
case 'd' :
angleX += 0.01f;
lx = sin(angleX);
lz = -cos(angleX);
break;

case 'a' :
angleX -= 0.01f;
lx = sin(angleX);
lz = -cos(angleX);
break;

case 'w' :
angleY += 0.01f;
ly = sin(angleY);
break;

case 's' :
angleY -= 0.01f;
ly = sin(angleY);
break;

case 'q' :
ycam = (ycam + 0.5);
break;

case 'e' :
ycam = (ycam - 0.5);
break;
}
glutPostRedisplay();
}
```

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:

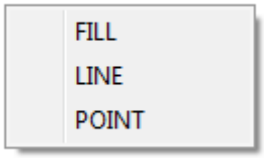

Figura 3.1: Menu de opções de visualização. Acessível com clique no botão direito do rato

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*:

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.

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.