Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Relatório do projeto

Fase 1 - Primitivas gráficas

Computação gráfica
Grupo 37

Grupo de trabalho
André Santos, A61778
Diogo Machado, A75399
Lisandra Silva, A73559
Rui Leite, A75551

Braga, 6 de Março de 2017

Conteúdo

Resumo

O presente relatório documenta a 1.ª fase do trabalho prático da Unidade Curricular de Computação Gráfica.

O relatório encontra-se organizado em 3 partes. Na primeira parte apresentam-se as classes usadas, explicando qual o objetivo de cada uma e as suas respetivas funções. Na segunda parte apresenta-se o motor, nomeadamente a forma como os ficheiros de pontos são lidos, como são desenhados, bem como algumas considerações sobre a câmara e menus da aplicação. Na terceira parte é apresentado o gerador. São indicadas quais as figuras que é possível desenhar e a forma como os pontos de cada figura são gerados.

1. Classes

Neste capítulo serão apresentadas as classes utilizadas para a concretização do trabalho prático. Estas classes foram criadas de modo a auxiliar e a facilitar o desenvolvimento do motor e gerador. Visto que estas classes implementam alguma lógica (cálculos), a divisão do código em classes permitiu por um lado simplificar o código do motor e gerador. Por outro lado evitou a repetição de código, o que por sua vez tornou mais fácil detetar e evitar erros. Estes foram os dois principais motivos que levaram à criação das classes. De um ponto de vista da programação por objetos, as classes apresentam ainda a vantagem de poderem encapsular lógica de negócio e estruturas de dados. Visto se tratar de um trabalho de computação gráfica, esse aspeto foi ignorado. Como se vai poder ver, algumas classes permitem o acesso direto a variáveis e não há a preocupação de criar cópias de objetos, aspetos que seria necessário ter em conta caso se tratasse de um trabalho puramente orientado a objetos.

1.1 Ponto3D

Tanto o gerador como o motor trabalham com pontos. Assim, foi criada uma classe que representa um ponto num espaço 3D, a classe `Ponto3D`, com a seguinte declaração:

```
struct Ponto3D {
    float x, y, z;
    Ponto3D& operator+(const Ponto3D& p2);
    Ponto3D& operator-(const Ponto3D& p2);
}
```

Esta classe contém 3 variáveis de instância e tem duas funções que fazem *overload* a 2 operadores do C++. As variáveis de instância `x`, `y` e `z` representam as coordenadas cartesianas de um ponto no espaço a 3 dimensões. Estas variáveis podem ser acedidas diretamente, tanto para escrita como para leitura. As duas funções da classe fazem *overload* aos operadores $+$ e $-$ do C++ e permitem por isso que se possa aplicar estes dois operadores a objetos do tipo `Ponto3D`. Ou seja, tendo um ponto `a` e `b` definidos como sendo:

```
Ponto3D a = {1,2,3};
Ponto3D b = {4,2,1};
```

Podemos ter um ponto `c` definido como a soma de `a` com `b`:

```
Ponto3D c = a + b;
```

Como o operador + está definido na classe `Ponto3D`, o código de cima é válido e como esperado irá colocar no ponto `c` a soma das coordenadas cartesianas dos pontos `a` e `b`, ou seja `c` terá `c.x = 5`, `c.y = 4` e `c.z = 4`.

1.2 Coordenadas Polares

As coordenadas polares permitem representar um ponto a 3 dimensões através de um ponto central $C(cx, cy, cz)$, um ângulo α e uma distância ao ponto central r, também designada por raio, como mostra a figura 1.1.

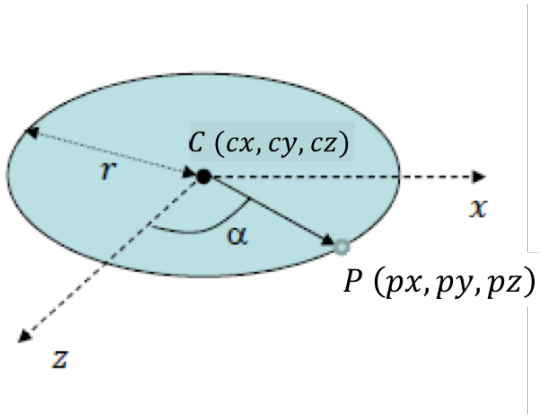


Figura 1.1: Localização de um ponto $P(px, py, pz)$ segundo um centro $C(cx, cy, cz)$, um ângulo *azimuth* (α) e um raio r

Através das seguintes expressões matemáticas é possível obter as coordenadas cartesianas do ponto $P(px, py, pz)$ a partir das coordenadas polares:

$$px = cx + r \times sin(\alpha)$$

$$py = cy$$

$$pz = cz + r \times cos(\alpha)$$

A representação dos pontos em coordenadas polares auxilia a criação dos pontos de algumas superfícies e figuras, como por exemplo o círculo, o cone e a esfera. Por esse motivo foi criada uma classe `CoordsPolares` com as seguintes variáveis de instância.

```
class CoordsPolares {
    Ponto3D centro;
    float raio, azimuth;
    Ponto3D cCartesianas;
}
```

Com esta classe consegue-se representar um ponto 3D pelas suas coordenadas polares e, a partir das suas instâncias, saber diretamente quais são as coordenadas cartesianas correspondentes, através da variável de instância `cCartesianas`. As outras 3 variáveis de instância (`centro`, `raio` e `azimuth`) correspondem aos parâmetros C, r e α das coordenadas polares, respetivamente.

Sempre que uma instância da classe `CoordsPolares` é criada passando C, r e α como parâmetros, as respetivas coordenadas cartesianas são atualizadas pela função `refreshCartesianas`:

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

Como se pode verificar, o código desta função limita-se a implementar diretamente as equações de transformação de coordenadas polares em coordenadas cartesianas apresentadas anteriormente.

1.3 Coordenadas Esféricas

As coordenadas esféricas permitem representar um ponto a 3 dimensões através de um centro $C(cx, cy, cz)$, uma distância r relativamente ao centro, um ângulo θ (*azimuth*) em relação ao eixo z e um ângulo polar φ em relação ao eixo OY, conforme representado na figura 1.2.

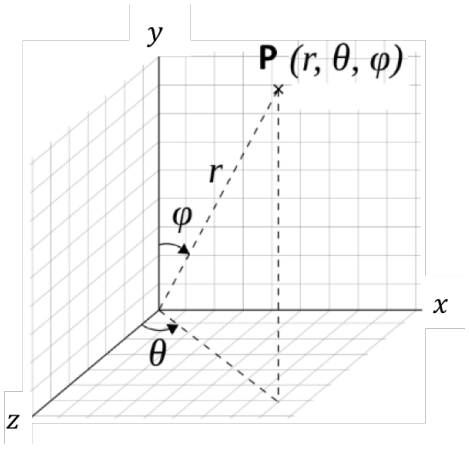


Figura 1.2: Localização de um ponto P segundo coordenadas esféricas

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

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

A conversão contrária também é possível, ou seja, a partir das coordenadas cartesianas de um dado ponto é possível obter as respetivas coordenadas esféricas através

das seguintes equações:

$$r = \sqrt{px^2 + py^2 + pz^2}$$
$$\varphi = \arctan\left(\frac{py}{px}\right)$$
$$\theta = \arccos\left(\frac{pz}{r}\right)$$

As coordenadas esféricas auxiliam a criação dos pontos da esfera e a controlar do movimento da câmara, o que levou à criação da classe `CoordsEsfericas` com as seguintes variáveis de instância:

```
class CoordsEsfericas {
    float raio, azimuth_ang, polar_ang;
    Ponto3D cCartesianas;
}
```

Com esta classe consegue-se representar um ponto 3D pelas suas coordenadas esféricas e, a partir das suas instâncias, saber diretamente quais são as coordenadas cartesianas correspondentes, através da variável de instância `cCartesianas`. As outras 3 variáveis de instância (`raio`, `azimuth_ang` e `polar_ang`) correspondem aos parâmetros r, θ e φ das coordenadas esféricas, respetivamente. Ao contrário das coordenadas polares, nesta classe não se representou o centro, que se assume ser $C(0, 0, 0)$.

Sempre que uma instância da classe `CoordsEsfericas` é criada passando r, φ e θ como parâmetros, as respetivas coordenadas cartesianas são atualizadas pela função `refreshCartesianas`:

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

Como se pode verificar, o código desta função limita-se a implementar diretamente as equações de transformação de coordenadas esféricas em coordenadas cartesianas apresentadas anteriormente.

1.4 Figura

A classe `Figura` foi criada com o objetivo de representar uma figura que será desenhada pelo motor. Para construir uma figura é desenhado um conjunto de pontos por uma determinada ordem. Cada três pontos formam um triângulo. A figura será então um conjunto de triângulos. Deste modo, esta classe possui apenas uma variável de instância, do tipo `vector`, que contêm os pontos da figura pela ordem com que devem ser desenhados.

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

Esta classe é útil na medida em que disponibiliza um conjunto de funções que coloca os pontos necessários ao desenho da figura pretendida no vetor `pontos`. As funções criadas permitem desenhar planos, círculos, caixas, cones, esferas, cilindros, elipsoides, Fitas de *Mobius* e *Seashells*. Estas funções serão explicadas com mais detalhe no capítulo 3 - "Gerador".

É possível obter os pontos da figura através da função `getPontos()`:

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

2. Motor

2.1 Objetivos

Com o motor pretende-se uma aplicação que seja capaz de desenhar uma cena especificada por ficheiros .3d e XML. Além de desenhar a cena, pretende-se que o utilizador possa interagir de forma básica com motor, nomeadamente rodar a câmara, fazer *zoom in*/*zoom out* e outras opções relativas à visualização da cena.

2.2 Leitura de Ficheiros

2.2.1 PugiXML

Para auxílio à leitura do XML foi usada uma biblioteca de parsing de XML chamada `pugiXML`. Foi considerada a utilização de outras bibliotecas como por exemplo `TinyXML2`, `RapidXML`, `LibXML2` e `Xerces`, no entanto a biblioteca escolhida pareceu-nos ser mais eficiente, tem uma API fácil de usar e com bastantes funcionalidades.

2.2.2 Ficheiro XML

A estrutura do XML é composta pelos seguintes elementos:

- **scene** - elemento pai
- **model** - elementos filhos de *scene* com um atributo *file* cujo valor corresponde ao nome de um ficheiro de pontos (ficheiro .3d).

Nos 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.

A leitura do ficheiro XML é feita pela função `leXML`. O pseudo-código desta função pode ser expresso da seguinte forma:

```
void leXML(){
    Abrir documento XML para leitura
    if (erro ao abrir ficheiro) {
        Imprimir mensagem de erro
        Sair da leitura
    }
```

```
    Aceder ao elemento root (<scene>)
    Para cada elemento <model> filho de <scene>:
        Ler nome do ficheiro de pontos
        Abrir ficheiro de pontos
        Para cada ponto no ficheiro de pontos:
            Adicionar à lista de pontos a desenhar
    }
}
```

O motor por defeito tenta sempre ler um ficheiro “scene.xml” colocado na pasta “Modelos”.

2.3 Desenho dos pontos

O resultado da leitura do ficheiro XML e dos ficheiros .3d é uma lista de `Ponto3D` ordenados pela ordem que devem ser desenhados. Para esta lista de pontos usou-se uma variável global ao motor: `vector<Ponto3D> pontos`, sendo que a função `leXML()` guarda os pontos neste vetor.

A função `renderScene`, além de fazer o *setup* gráfico, é responsável pelo desenho dos pontos, de acordo com o seguinte pseudo-código:

```
void renderScene() {
    Definir modo polígonos
    Limpar ecrã e definir cor
    Posicionar câmara
    Para cada ponto na lista de pontos a desenhar:
        Desenhar ponto
    Trocar buffers de desenho e visualização
}
```

Concretamente, o ciclo de desenho dos pontos fica escrito como:

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

2.4 Câmara

Com o objetivo de poder visualizar um objeto por diversos ângulos, foi implementada uma câmara. Esta câmara é pode ser vista como um ponto numa esfera com centro na origem do referencial, e por isso, para a implementação tirou-se partido da classe `CoordsEsfericas` apresentada anteriormente e declarou-se como sendo uma variável global ao motor:

```
CoordsEsfericas camara;
```

Na função `renderScene` é passado como argumento à função `gluLookAt` as coordenadas cartesianas da câmara, que podem ser obtidas diretamente a partir da variável de classe `Ponto3D cCartesianas`:

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

O utilizador pode mudar a posição da câmara recorrendo ao teclado e ao rato, como será explicado nas secções seguintes

A câmara têm ainda um fator de velocidade, também representado por uma variável global:

```
float cameraSpeed = 6.0f;
```

Esta variável será usada pelas funções do rato e do teclado para controlar a velocidade a que a câmara se movimenta, de cada vez que o utilizador carrega numa tecla ou mexe o rato, conforme será explicado a seguir.

2.5 Teclado

Foram associadas ações às teclas do teclado, dando ao utilizador oportunidade de interagir com o programa com as funcionalidades descritas abaixo.

- **Tecla 'W'** - Move a câmara para cima.
- **Tecla 'S'** - Move a câmara para baixo.
- **Tecla 'A'** - Move a câmara para a esquerda.
- **Tecla 'D'** - Move a câmara para a direita.
- **Tecla 'Q'** - Afasta objeto (*zoom-out*).
- **Tecla 'E'** - Aproxima objeto (*zoom-in*).
- **Tecla 'J'** - Reduz velocidade da câmara.
- **Tecla 'K'** - Repõe a velocidade "normal" da câmara (valor original).

- **Tecla 'L'** - Aumenta velocidade da câmara.

Para implementar estas operações foi criada uma função `teclas_normais_func` que consiste num `switch` que faz uma ação diferente consoante a tecla carregada.

```
void teclas_normais_func(unsigned char key, int x, int y) {
    switch (tolower(key)) {
        case 'w':
            camara.paraCima(cameraSpeed * M_PI/360.0);
            break;
        case 's':
            camara.paraBaixo(cameraSpeed * M_PI/360.0);
            break;
        case 'a':
            camara.paraEsquerda(cameraSpeed * M_PI/360.0);
            break;
        case 'd':
            camara.paraDireita(cameraSpeed * M_PI/360.0);
            break;
        case 'e':
            camara.aproximar(0.5);
            break;
        case 'q':
            camara.afastar(0.5);
            break;
        case 'j':
            --cameraSpeed;
            break;
        case 'k':
            cameraSpeed=6;
            break;
        case 'l':
            ++cameraSpeed;
            break;
    }
    glutPostRedisplay();
}
```

Como se pode verificar, as opções de movimentação da câmara consistem em operações disponibilizadas pela classe `CoordsEsfericas`. Para as teclas 'w' 's' 'a' e 'd' a câmara desloca-se um n.º de graus indexado pela velocidade da câmara (variável `cameraSpeed`) numa das direções determinadas pela tecla premida. Para as teclas 'e' e 'q' são usadas as operações de aproximar e afastar que diminuem/aumentam o raio da esfera na qual a câmara está colocada.

Os métodos de instância chamados na movimentação da câmara alteram os valores das coordenadas cartesianas correspondentes, e portanto, no *frame* seguinte, os parâmetros passados à função *gluLookAt*, relativos à posição da câmara, estarão atualizados com a nova posição.

As teclas ‘j’ ‘k’ e ‘l’ controlam a velocidade da câmara, o que corresponde a escritas na variável (`cameraSpeed`). O valor desta variável será usado para calcular o ângulo de deslocação de cada vez que uma das teclas de movimentação for carregada. Ou seja, uma maior velocidade da câmara corresponde a um movimento de número de graus maior em cada movimento.

Existem ainda teclas consideradas “especiais” pelo GLUT às quais também foram adicionadas ações de movimento da câmara:

- **Tecla ‘UP’** - Move câmara para cima.
- **Tecla ‘DOWN’** - Move câmara para baixo.
- **Tecla ‘LEFT’** - Move câmara para a esquerda.
- **Tecla ‘RIGHT’** - Move câmara para a direita.

Estas teclas são equivalentes às teclas ‘w’, ‘s’, ‘a’ e ‘d’, respetivamente.

Visto que o GLUT trata as teclas ‘UP’, ‘DOWN’, ‘LEFT’ e ‘RIGHT’ como teclas especiais, teve que ser criada uma função `teclas_especiais_func` para processar as ações nestas teclas:

```
void teclas_especiais_func(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_LEFT:
            camara.paraEsquerda(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_UP:
            camara.paraCima(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_RIGHT:
            camara.paraDireita(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_DOWN:
            camara.paraBaixo(cameraSpeed*M_PI/360.0);
            break;
        }
    }
```

2.6 Rato

Além de terem sido associadas ações ao teclado, foram também associadas ações aos botões direito e esquerdo do rato. O botão direito do rato permite ao utilizador aceder a um menu de contexto com diferentes opções de visualização. O botão esquerdo do rato permite movimentar a câmara.

2.6.1 Menus - Botão direito

Para proporcionar alguma flexibilidade na visualização das figuras e também de forma a aumentar a interação do utilizador com o programa, foi incluído o menu representado na figura 2.1. Este menu, que pode ser acedido em qualquer posição da janela, contém sub-menus e permite as seguintes funcionalidades:

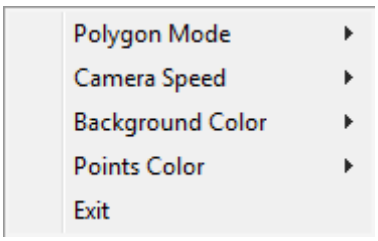


Figura 2.1: Menu de opções acessível com o clique do botão do lado direito do rato

O código de criação dos menus encontra-se na função `criaMenus` do motor, que chama as funções do GLUT necessárias. Esta função cria os sub-menus e o menu principal, associando os sub-menus ao menu principal.

Por exemplo, para a criação do sub-menu "Polygon Mode" foram usadas as seguintes instruções:

```
int polMode_menu = glutCreateMenu(polMode_menu_func);
glutAddMenuEntry("Fill", 1);
glutAddMenuEntry("Line", 2);
glutAddMenuEntry("Point", 3);
```

Cada sub-menu tem uma função responsável pelo tratamento da opção escolhida dentro desse sub-menu. O nome desta função é passada como argumento na criação do sub-menu, que no caso de cima é a função `polMode_menu_func`. Os restantes sub-menus foram criados de forma semelhante.

Por fim, é criado o menu principal, são associados os sub-menus e define-se o botão direito do rato como o botão de acesso ao menu.

```
int main_menu = glutCreateMenu(main_menu_func);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutAddSubMenu("Polygon Mode", polMode_menu);
glutAddSubMenu("Camera Speed", camSpeed_menu);
glutAddSubMenu("Background Color", bgColor_menu);
glutAddSubMenu("Points Color", ptColor_menu);
glutAddMenuEntry("Exit", 1);
```

Polygon Mode

Permite escolher o modo como os polígonos são desenhados: com preenchimento (*Fill*), apenas linhas/arestas (*Line*), apenas pontos (*Point*);

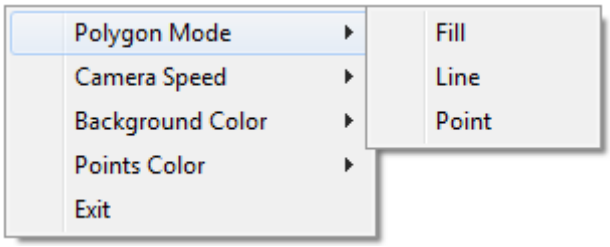


Figura 2.2: Sub-menu “Polygon Mode”

Para implementar esta funcionalidade criou-se uma variável global `modoPoligonos`

```
GLenum modoPoligonos = GL_LINE;
```

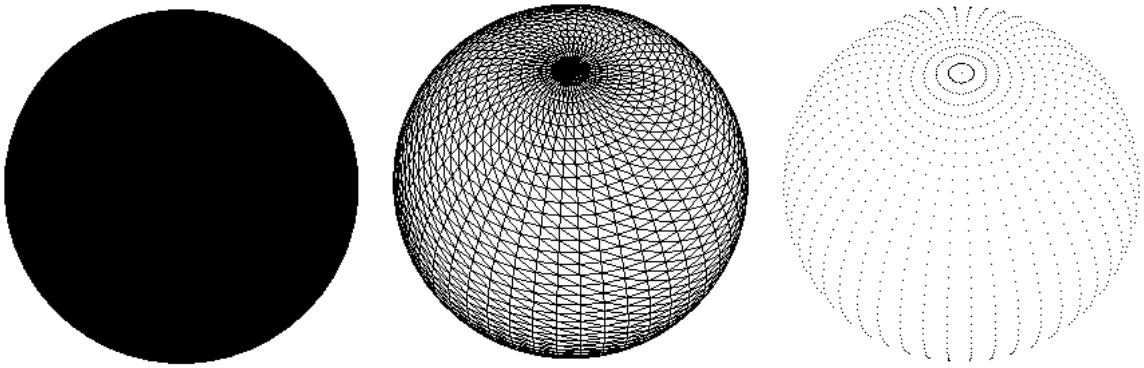
O modo de desenho é um dos parâmetros passados à função `glPolygonMode`, pelo que se passa o valor desta variável à função:

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

O valor desta variável é alterado consoante a escolha do utilizador:

```
void polMode_menu_func(int opt) {
    switch (opt) {
        case 1:  modoPoligonos = GL_FILL; break;
        case 2:  modoPoligonos = GL_LINE; break;
        case 3:  modoPoligonos = GL_POINT; break;
    }
    glutPostRedisplay();
}
```

Depois de alterado o valor da variável, é necessário pedir ao GLUT que volte a desenhar a cena para que a função `glPolygonMode` volte a ser chamada com o novo valor nesta variável.



(a) Modo “GL_FILL” (b) Modo “GL_LINE” (c) Modo “GL_POINT”

Figura 2.3: Diferentes formas de apresentação das figuras conforme a opção de “Polygon Mode”

Camera Speed

Este sub-menu permite alterar a velocidade do movimento da câmara. São mostradas ao utilizador 5 opções de velocidade, sendo que a opção "Normal" corresponde à velocidade da câmara por defeito quando se arranca a aplicação

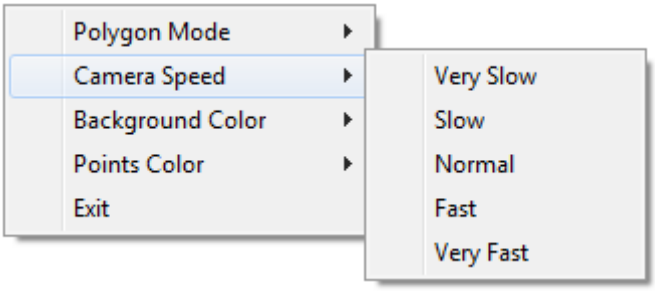


Figura 2.4: Sub-menu "Camera Speed"

Como foi referido anteriormente, a velocidade da câmara é controlada pela variável global `cameraSpeed`. Por isso, quando um utilizador escolhe uma opção de velocidade, a velocidade da câmara é alterada por alteração da variável `cameraSpeed`:

```
void camSpeed_menu_func(int opt) {
    switch (opt) {
        case 1:  cameraSpeed = 1.0; break;
        case 2:  cameraSpeed = 3.0; break;
        case 3:  cameraSpeed = 6.0; break;
        case 4:  cameraSpeed = 9.0; break;
        case 5:  cameraSpeed = 12.0; break;
    }
}
```

Background Color

Este sub-menu permite alterar a cor de fundo da cena para preto, branco, vermelho, verde ou azul.

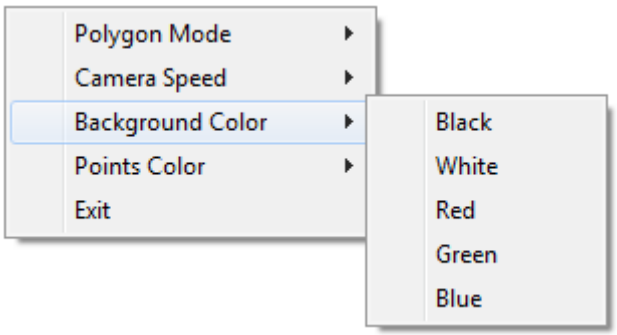


Figura 2.5: Sub-menu "Background Color"

Esta funcionalidade é implementada também com recurso a variáveis globais:

```
float bg_red = 0.0, bg_green = 0.0, bg_blue = 0.0;
```

Estas variáveis tomam valores entre 0.0 e 0.1 e definem uma cor no sistema RGB. Estes valores são passadas como parâmetro à função `glClearColor`:

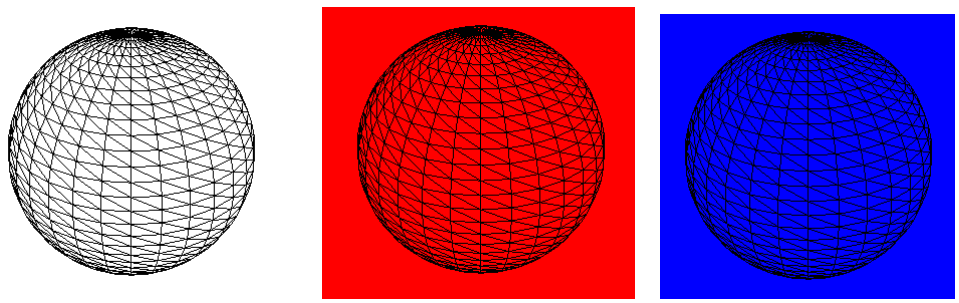
```
glClearColor(bg_red, bg_green, bg_blue,1);
```

Quando o utilizador seleciona uma cor, estas 3 variáveis são alteradas de modo a representarem a cor pretendida pelo utilizador:

```
void bgColor_menu_func(int opt) {
    switch (opt) {
        case 1:
            bg_red = 0; bg_green = 0; bg_blue = 0;
            break;
        case 2:
            bg_red = 1; bg_green = 1; bg_blue = 1;
            break;
        case 3:
            bg_red = 1; bg_green = 0; bg_blue = 0;
            break;
                case 4:
            bg_red = 0; bg_green = 1; bg_blue = 0;
            break;
        case 5:
            bg_red = 0; bg_green = 0; bg_blue = 1;
            break;
    }
    glutPostRedisplay();
}
```

Depois de alterados os valores da variável, é necessário pedir ao GLUT que volte a desenhar a cena para que a função `glClearColor` volte a ser chamada com os novos valores da cor.

De seguida apresenta-se a mesma esfera desenhada a preto com diferentes cor de fundo:



(a) Cor de fundo branca (b) Cor de fundo vermelha (c) Cor de fundo azul

Figura 2.6: Mesma esfera com diferentes cor de fundo

Point Color

Este sub-menu permite alterar a cor com que os pontos (e por isso os polígonos) são desenhados. Tal como na cor de fundo da cena, as opções para a cor dos pontos são preto, branco, vermelho, verde ou azul.

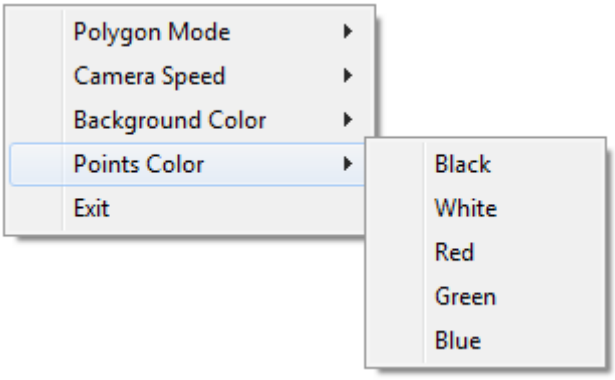


Figura 2.7: Sub-menu “Point Color”

Esta funcionalidade é implementada também com recurso a variáveis globais:

```
float pt_red = 1.0, pt_green = 1.0, pt_blue = 1.0;
```

Estas variáveis tomam valores entre 0.0 e 0.1 e definem uma cor no sistema RGB. Estes valores são passadas como parâmetro à função `glColor3f` que é chamada antes do ciclo de desenho dos pontos:

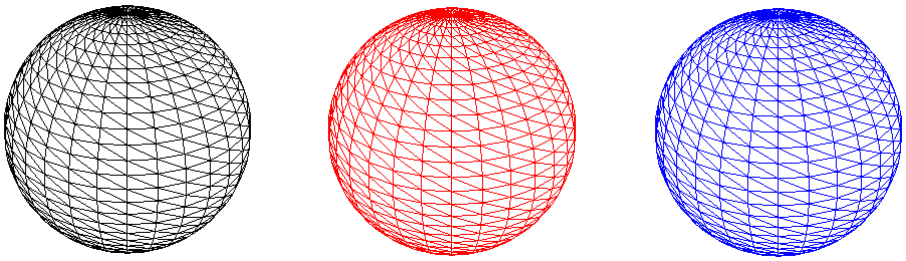
```
glColor3f(pt_red, pt_green, pt_blue);
```

Quando o utilizador seleciona uma cor, estas 3 variáveis são alteradas de modo a representarem a cor pretendida pelo utilizador:

```
void ptColor_menu_func(int opt) {
    switch (opt) {
        case 1:  pt_red = 0; pt_green = 0; pt_blue = 0;
            break;
        case 2:  pt_red = 1; pt_green = 1; pt_blue = 1;
            break;
        case 3:  pt_red = 1; pt_green = 0; pt_blue = 0;
            break;
        case 4:  pt_red = 0; pt_green = 1; pt_blue = 0;
            break;
        case 5:  pt_red = 0; pt_green = 0; pt_blue = 1;
            break;
    }
    glutPostRedisplay();
}
```

Depois de alterados os valores da variável, é necessário pedir ao GLUT que volte a desenhar a cena para que a função `glColor3f` volte a ser chamada com os novos valores da cor.

De seguida apresenta-se a mesma esfera desenhada a fundo branco com diferentes cores de pontos:



(a) Cor de fundo branca (b) Cor de fundo azul (c) Cor de fundo vermelha

Figura 2.8: Mesma esfera com diferentes cor de fundo

Exit

Permite ao utilizador sair da aplicação.

2.6.2 Movimento da câmara - Botão esquerdo

Como visto anteriormente a câmara pode ser controlada com as setas direcionais do teclado ou com as teclas 'w' 's' 'a' e 'd'. Além destas opções, também é possível movimentar a câmara com o botão esquerdo do rato.

Para arrastar a câmara, o utilizador deve carregar no botão esquerdo e arrastar o rato na direção que pretende movimentar a câmara, mantendo o botão pressionado.

Para implementar esta funcionalidade em primeiro lugar foi necessário criar 2 variáveis globais com a posição x e y do rato relativamente à janela:

```
int mouse_x, mouse_y;
```

Quando o utilizador carrega no botão esquerdo do rato, a função `mouse_events_func` é chamada e guarda a posição (x,y) do rato nas variáveis globais.

```
void mouse_events_func(int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            mouse_x = x;
            mouse_y = y;
        }
        break;
    }
}
```

Quando o utilizador arrasta o rato, é chamada a função `mouse_motion_func`. Esta função regista em `deltaX` e `deltaY` qual o deslocamento do rato. Este deslocamento do rato é depois usado como parâmetro funções de deslocação da câmara:

```
void mouse_motion_func(int x, int y) {
    int deltaX = x - mouse_x;
    int deltaY = y - mouse_y;
    float cf = 0.009;
    if (deltaX > 0) {
        camara.paraEsquerda(deltaX * cf *
                            cameraSpeed * M_PI / 360.0);
    } else {
        camara.paraDireita(abs(deltaX) * cf *
                           cameraSpeed * M_PI / 360.0);
    }
    if (deltaY > 0) {
        camara.paraCima(deltaY * cf *
                        cameraSpeed * M_PI / 360.0);
    } else {
        camara.paraBaixo(abs(deltaY) * cf *
                         cameraSpeed * M_PI / 360.0);
    }
}
```

Por fim, registou-se estas funções no GLUT:

```
glutMouseFunc(mouse_events_func);
glutMotionFunc(mouse_motion_func);
```

3. Gerador

3.1 Objetivos

O gerador foi criado com o propósito de interpretar os pedidos do utilizador e gerar um ficheiro .3d com os pontos correspondentes à figura solicitada.

3.2 Programa principal

Quando o utilizador chama o gerador indica a figura que pretende e parâmetros numéricos relativos a essa figura. O programa principal consiste numa série de if's que testam se o comando é válido e caso o seja, os pontos da figura são gerados e guardados num ficheiro indicado pelo utilizador. Um comando é considerado válido se a figura pedida é reconhecida pelo gerador e se o nº de parâmetros necessários ao desenho da figura está correto.

O pseudo-código do programa principal é o seguinte:

```
int main(int argc, char** argv) {
    if (comando valido) {
        Ler e interpretar parâmetros
        Desenhar os pontos da figura pedida
        figuraCriada = true;
    }

    if (figuraCriada == false) {
        Informar que o comando é inválido;
    }else{
        Criar ficheiro com os pontos.
    }
}
```

Na realidade, o teste de validade do comando é feito por vários `if`'s, um por cada figura possível. No pseudo-código acima, por questões de espaço optou-se por representar a validade apenas por um `if`. O ficheiro de pontos está a ser guardado na pasta "Modelos" do Motor.

3.3 Primitivas

Nesta secção são apresentadas as figuras que o programa é capaz de gerar, a forma como são geradas e os comandos correspondentes.

3.3.1 Planos

Para gerar um plano o utilizador deverá introduzir um comando com a seguinte sintaxe:

```
gerador plane comprimento largura divsx divsz ficheiro.3d
```

O resultado deste comando é um plano em XZ centrado na origem $C(0,0,0)$, com o comprimento, a largura, e o número de divisões no eixo X e em Z especificados.

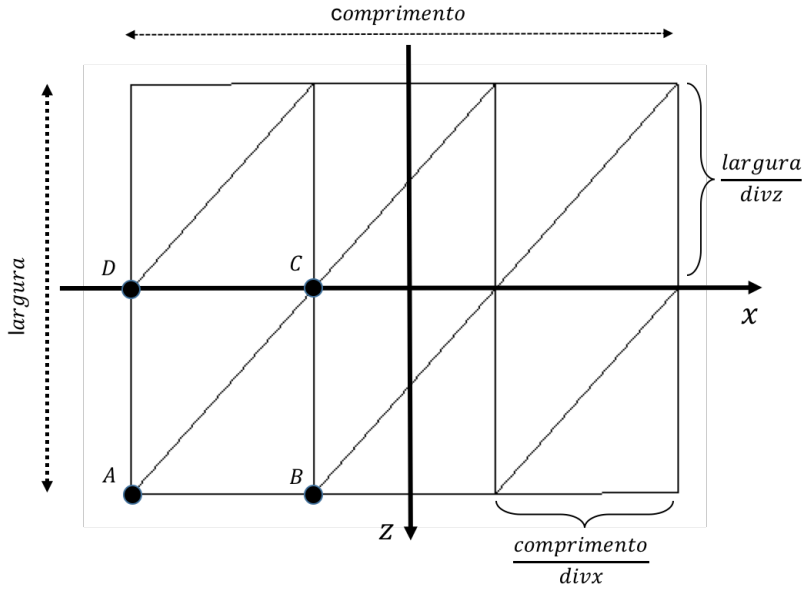


Figura 3.1: Plano em XOZ centrado na origem $C(0,0,0)$, com 3 divisões em X e duas em Z

Cada secção do retângulo é formada por dois triângulos: ABC e ACD, pela figura 3.1. Deste modo, de acordo com o número de divisões em X e em Z, é possível considerar um plano como um conjunto de retângulos menores numa quantidade igual a: $\#divisões\ X \times \#divisões\ Z$. Desta forma, pode-se olhar para um plano como estando dividido em diferentes colunas e linhas.

A ordem pela qual se adiciona os pontos à figura, pela regra da mão direita, determina para que lado ela fica virada. Se quisermos que o plano fique em XOZ voltado para cima, deve-se colocar os pontos pela seguinte ordem: A-B-C (1º triângulo), seguido de A-C-D (2º triângulo). Se quisermos que o plano fique voltada para baixo, 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 `geraPlanoY`, cujo pseudo-código se apresenta de seguida:

```
Figura& geraPlanoEmY(Ponto3D o, float comp, float larg,
                     int divsx, int divsz, int orientacao){
    Para cada linha
        Para cada coluna {
            Calcular coordenadas dos pontos A, B, C e D
            if (orientacao == 1) {
                Guardar pontos pela ordem A-B-C-C-D-A
            } else {
                Guardar pontos pela ordem A-D-C-C-B-A
            }
        }
    return *this;
}
```

3.3.2 Caixa

Para gerar uma caixa o utilizador deverá introduzir um comando com a seguinte sintaxe:

```
gerador box comprimento largura altura divsx divsz divsy
ficheiro.3d
```

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

Uma vez que uma caixa corresponde a 6 planos, para desenhar a caixa calculou-se o centro de cada uma das suas faces e utilizaram-se as primitivas da secção 3.3.1 que geram planos XY, YZ e XZ dado um centro e uma orientação.

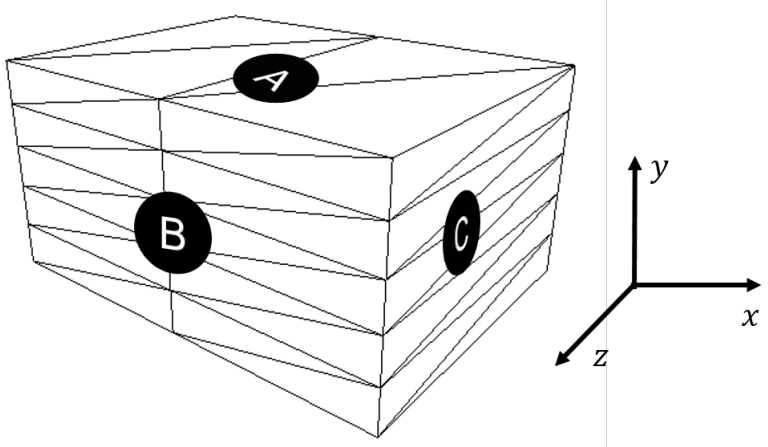


Figura 3.2: Caixa centrada no ponto $C(0,0,0)$ com 2 divisões em xy, uma divisão em zy e uma divisão em xz

Considere-se a caixa com as faces visíveis identificadas pelas letras A, B e C, conforme mostra a figura 3.2 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.

Se a caixa está centrada na origem, então os centros de cada uma das caixas são dados por:

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.

Assim a caixa é desenhada segundo o seguinte pseudo-código:

```
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;
}
```

3.3.3 Circulo

Um círculo corresponde a um conjunto de triângulos em que o centro do círculo é um ponto comum a todos os triângulos. O número de triângulos de um círculo corresponde ao número de "fatias" que se pretende e com isso é possível calcular o ângulo θ mostrado na figura 3.3. Sabendo o centro, o raio e o ângulo θ é possível saber as coordenadas de todos os pontos do círculo usando a classe CoordsPolares (ver secção 1.2).

Para gerar um círculo, o utilizador deverá introduzir um comando com a seguinte sintaxe:

```
gerador circle raio fatias ficheiro.3d
```

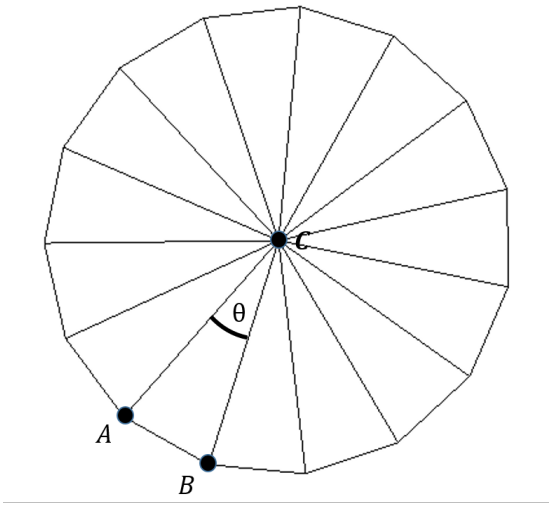


Figura 3.3: Esquema do desenho de um circulo com “fatias”

Tal como o plano, um círculo também tem uma orientação. Por exemplo, tendo ainda como referência a figura 3.3, caso se pretenda desenhar o círculo virado no sentido positivo do eixo dos Y, a ordem de colocação dos pontos para cada triângulo 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 `geraCirculo`, cujo pseudo-código é o seguinte:

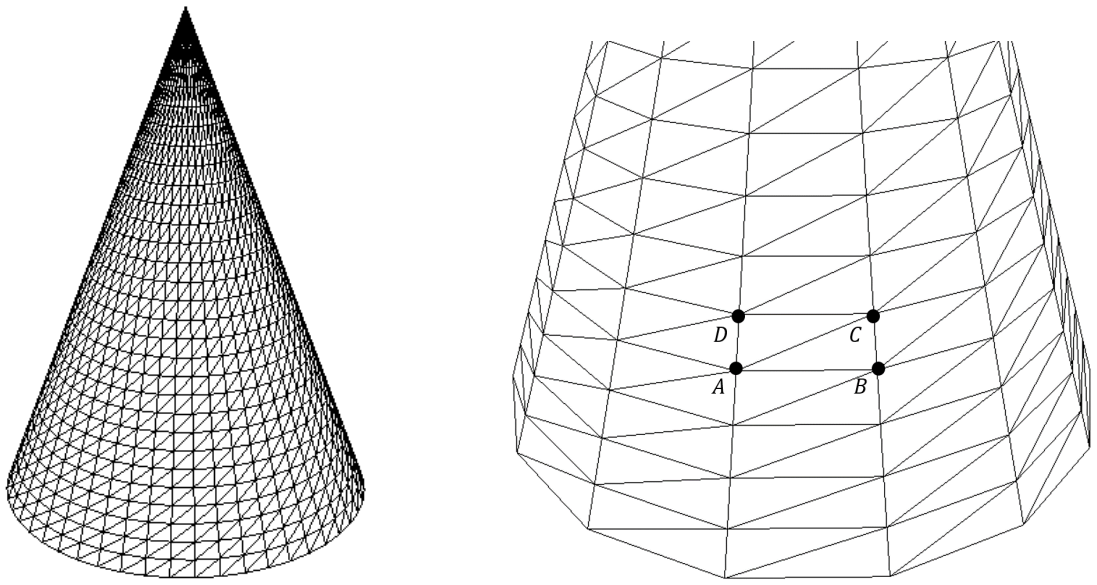
```
Figura& geraCirculo(Ponto3D C, float raio, int fatias,
int orientacao) {
    Para cada fatia{
        Calcula CoordsPolares A, B;
        if (orientacao == 1) {
            Coloca pontos no vetor pela ordem: A-B-C
        }
        else {
            Coloca pontos no vetor pela ordem: A-C-B
        }
    }
return *this;
}
```

3.3.4 Cone

Para gerar um cone, o utilizador deve inserir o seguinte comando

```
gerador cone raio altura fatias camadas ficheiro.3d
```

A base do cone corresponde a um círculo virado no sentido negativo dom eixo dos Y, que pode ser desenhado através da função `geraCirculo`. Resta apenas gerar os pontos para a superfície "curva" do cone. Para desenhar essa superfície, optou-se por desenhar o cone camada a camada, sendo que o circulo de cada camada é dividido em fatias. Seguindo este método obtemos a superfície apresentada na figura 3.4(b).



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.4: Cone

O retângulo de cada camada é formado pelos pontos A,B,C e D e as coordenadas destes pontos foram calculadas com auxílio mais uma vez da classe `CoordsPolares`. Podemos ver os pontos A e B como pertencentes a um mesmo círculo. Da mesma forma, também os pontos C e D se encontram num mesmo círculo, com raio menor do que o círculo onde estão os pontos A e B.

O raio destes dois círculos relaciona-se de acordo com a 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 $r - \frac{r}{\#camadas}$. A figura 3.5 pretende ilustrar tal situação para um exemplo de 3 camadas.

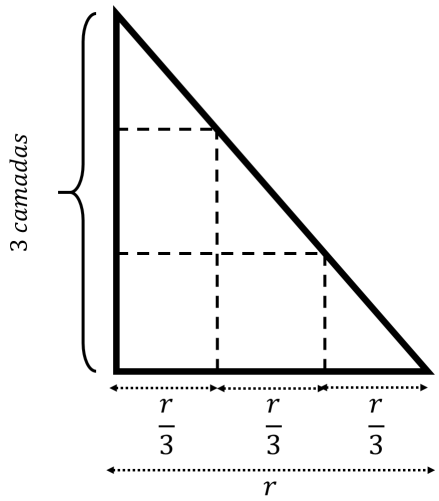


Figura 3.5: Relação do raio com o número de camadas de um cone

Os círculos têm todos como centro um ponto no eixo central 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 com a classe de `CoordsPolares` é possível saber coordenadas cartesianas sendo indicado um centro, um raio e um ângulo e tendo em conta o exposto anteriormente então a função `geraCone` pode ser representada pelo pseudo-código:

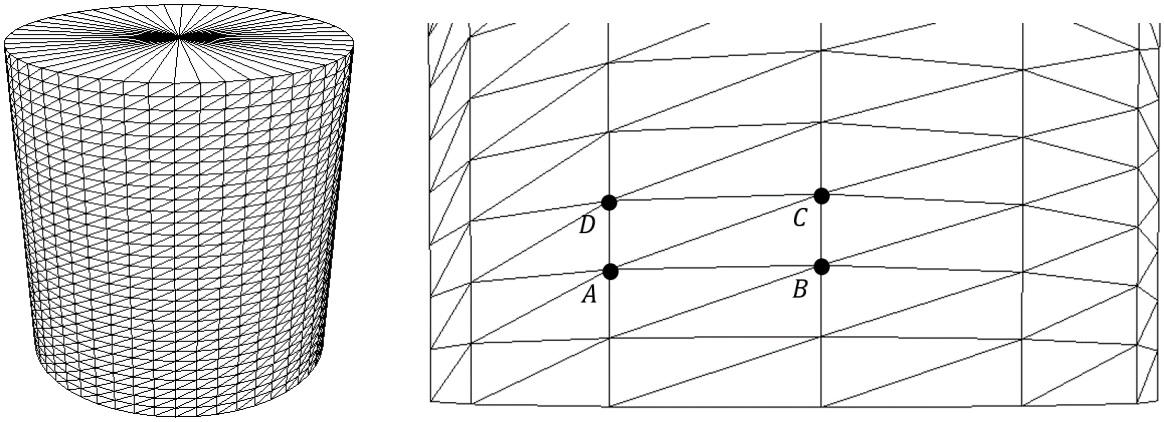
```
Figura& criaCone(Ponto3D centroCone, float altura,float raio,
                 int camadas, int fatias) {
     Criar base do cone voltada para baixo
     Para cada Camada
         Para cada Fatia
             Calcular coordenadas dos pontos A, B, C, D
             Guardar os pontos pela ordem A-C-D-A-D-B
     return *this;
 }
```

3.3.5 Cilindro

Para gerar um cilindro, o utilizador deve inserir o seguinte comando:

```
gerador cylinder raio altura fatias ficheiro.3d
```

A base inferior do cilindro corresponde a um círculo virado no sentido negativo do eixo dos Y e a base superior a um círculo virado no sentido positivo do eixo dos Y. Estes são desenhados através da função `geraCirculo`. Resta apenas gerar os pontos para a superfície "curva"do cilindro. Para desenhar essa superfície, optou-se por desenhar o cilindro camada a camada, sendo que o circulo de cada camada é dividido em fatias. Seguindo este método obtemos a superfície apresentada na figura 3.6(b).



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.6: Cilindro

O retângulo de cada camada é formado pelos pontos A,B,C e D e as coordenadas destes pontos foram calculadas com auxílio mais uma vez da classe `CoordsPolares`.

Podemos ver os pontos A e B como pertencentes a um mesmo círculo assim como os pontos C e D. Os círculos têm todos como centro um ponto no eixo central do cilindro, sendo que a coordenada Y é a única que varia de camada para camada, com uma diferença igual a $\frac{altura}{camadas}$.

A função `geraCilindro()` pode ser representada pelo pseudo-código:

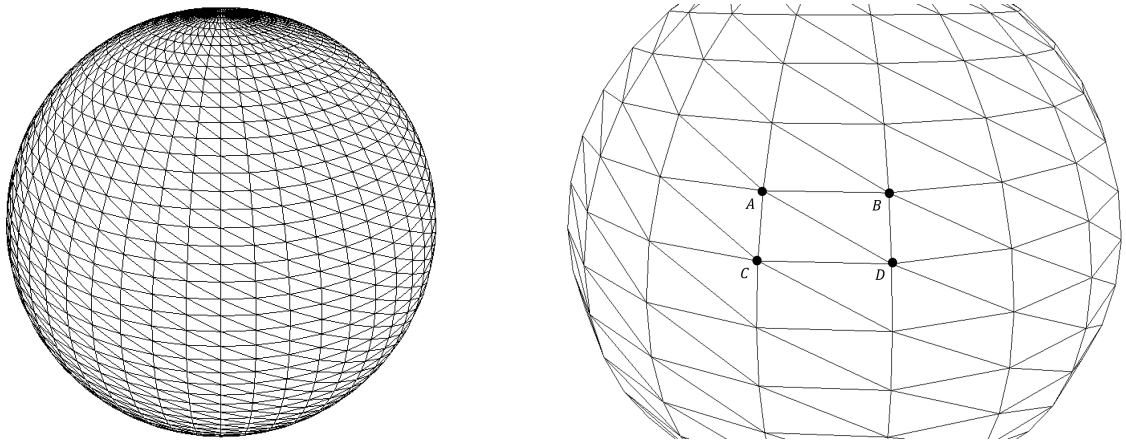
```
Figura& geraCilindro(Ponto3D o, float raio, float altura,
                     int fatias, int camadas) {
     Criar circulo da base inferior e da base superior
     Para cada camada
         Para cada fatia
             Calcula coordenadas dos pontos A,B,C,D.
             Guardar os pontos pela ordem A-B-C-A-C-D.
     return *this;
 }
```

3.3.6 Esfera

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

```
gerador esfera raio fatias camadas ficheiro.3d
```

Os pontos da esfera foram gerados à custa de coordenadas esféricas, nomeadamente com o auxílio da classe `CoordsEsfericas`. À semelhança do cone, também para a esfera é possível gerar uma "fatia" de uma determinada camada usando 4 pontos, conforme mostrado na figura 3.7(b) .



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.7: Esfera

A esfera é construída camada a camada a partir da parte superior. Para cada fatia é necessário calcular as coordenadas dos 4 pontos A, B, C e D, tarefa que se torna fácil recorrendo à classe `CoordsEsfericas`, uma vez que apenas variam os ângulos polar e $azimuth$ entre camadas e fatias.

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 $\frac{\pi}{c}$. Se considerarmos f fatias, a diferença do ângulo $azimuth$ de um ponto para outro que esteja numa fatia imediatamente a seguir é de $\frac{2\times\pi}{f}$.

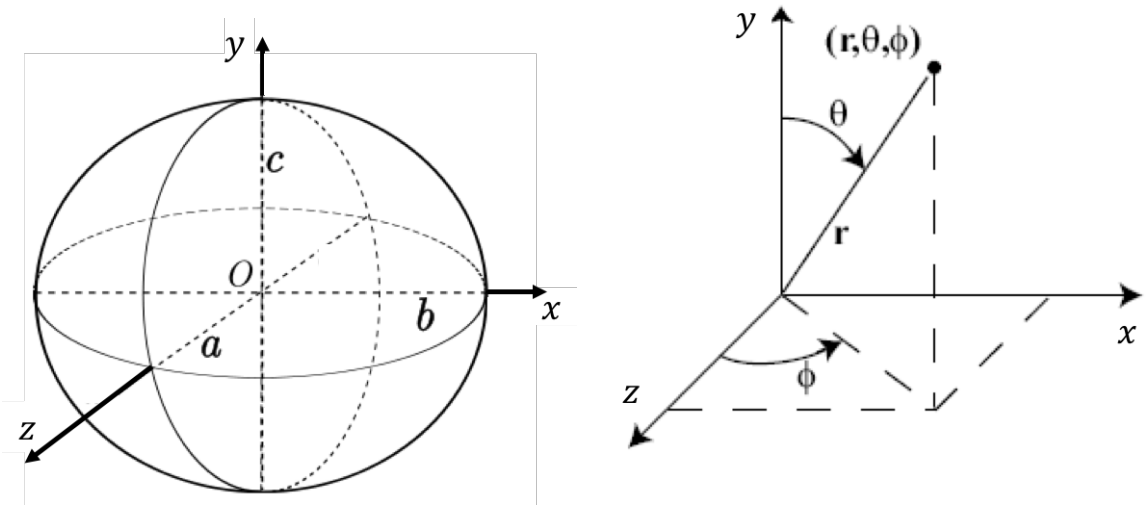
A função `geraEsfera` pode ser representada pelo pseudo-código:

```
Figura& geraEsfera(Ponto3D o, float r, int fat, int cam) {
    Para cada camada
        Para cada fatia
            Calcular coordenadas dos pontos A, B, C, D
            Guardar os pontos pela ordem A-C-D-A-D-B
    return *this;
}
```

3.3.7 Elipsoide

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

```
gerador ellipsoid a b c fatias camadas ficheiro.3d
```



(a) Eixos do elipsoide (b) Ângulos do Sistema de Coordenadas esfericas

Figura 3.8: Elipsoide - eixos

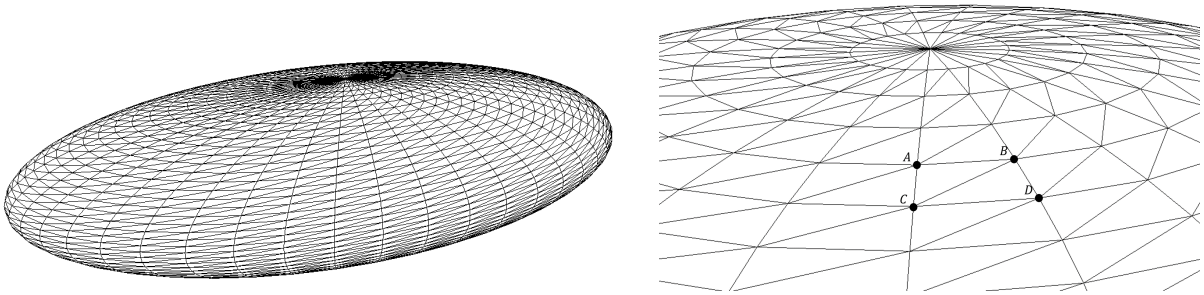
Considerando os eixos a, b e c do elipsóide representados na figura 3.8(a) e os eixos θ e ϕ representados na figura 3.8(b), os pontos do Elipsoide são gerados à custa das seguintes expressões matemáticas:

$$px = b \times cos\left(\frac{\pi}{2} - \theta\right) \times sen(\phi)$$

$$py = c \times sin\left(\frac{\pi}{2} - \theta\right)$$

$$pz = a \times cos\left(\frac{\pi}{2} - \theta\right) \times cos(\phi)$$

Similarmente à esfera, 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 $\frac{\pi}{c}$. Se considerarmos f fatias, o ângulo ϕ de um ponto para outro que esteja numa fatia imediatamente a seguir é de $\frac{2\times\pi}{f}$.



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.9: Elipsoide

Tendo em conta as expressões matemáticas supracitadas e o esquema da superficie lateral segundo o qual os pontos vão ser percorridos então a função `geraElipsoide` pode ser representada pelo seguinte pseudo-código:

```
Figura& geraElipsoide(Ponto3D o, float a, float b, float c,
                      int fatias, int camadas) {
    Para cada fatia
        Para cada camada
            Calcular coordenadas dos pontos A, B, C e D
            Guardar os pontos pela ordem A-C-B-B-C-D
    return *this;
}
```

3.3.8 Torus

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

```
gerador torus raioE raioI fatias camadas ficheiro.3d
```

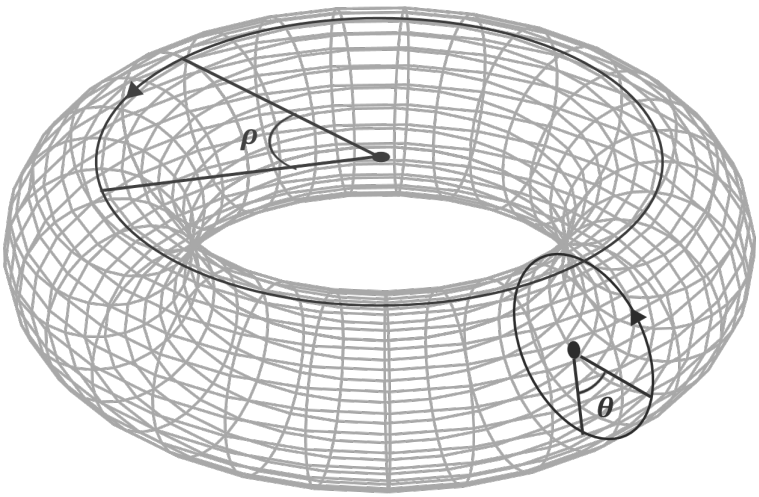


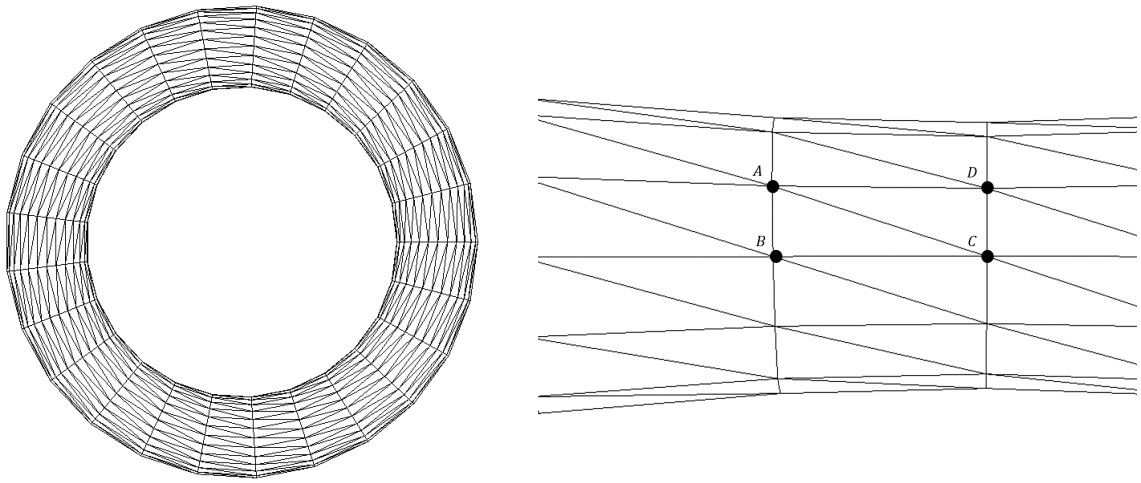
Figura 3.10: O torus é o produto de dois círculos (adaptado de Wikipedia)

Considerando R a distância do centro do tubo ao centro do torus (`raioE`), r o raio do tubo (`raioI`), ρ o ângulo toroidal e θ o ângulo poloidal (representados na figura

3.10), os pontos do torus podem ser definidos pelas seguintes expressões matemáticas:

$$px = (R + r \times cos(\theta)) \times sen(\rho)$$
$$py = r \times sen(\theta)$$
$$pz = (R + r \times cos(\theta)) \times cos(\rho)$$

Tendo c camadas, sabe-se que a diferença em termos de ângulo poloidal, θ, de um ponto que esteja numa camada superior para outro que esteja numa camada imediatamente inferior é de $\frac{2\times\pi}{c}$. Se considerarmos f fatias, o ângulo toroidal, ρ, de um ponto para outro que esteja numa fatia imediatamente a seguir é de $\frac{2\times\pi}{f}$.



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.11: Torus

Tendo em conta as expressões matemáticas supracitadas e o esquema da superficie lateral, na figura 3.11(b), segundo o qual os pontos vão ser percorridos então a função `geraTorus` pode ser representada pelo seguinte pseudo-código:

```
Figura& geraTorus(Ponto3D o, float R, float r,
                  int fatias, int camadas) {
    Para cada fatia
        Para cada camada
            Calcular coordenadas dos pontos A, B, C e D
            Guardar os pontos pela ordem A-B-C-A-C-D
    return *this;
}
```

3.3.9 Fita de Mobius

Para gerar uma Fita de Mobius, o utilizador deve efetuar o comando com a seguinte sintaxe:

```
gerador mobius raio largura divsl divscomp ficheiro.3d
```

A fita de Mobius é uma superfície não-orientável que tem apenas um lado. Possui um raio R, uma largura L e um comprimento C como mostra a figura 3.12 :

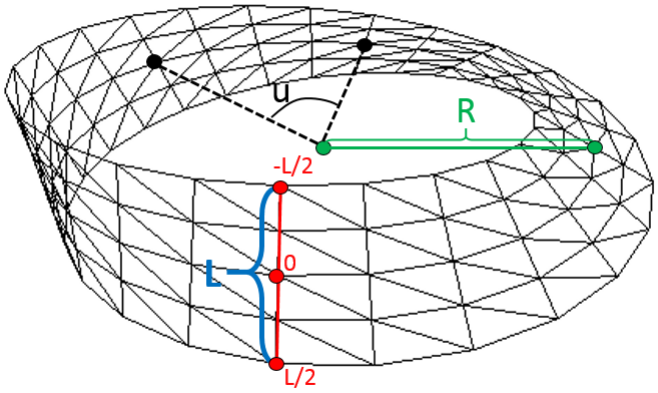
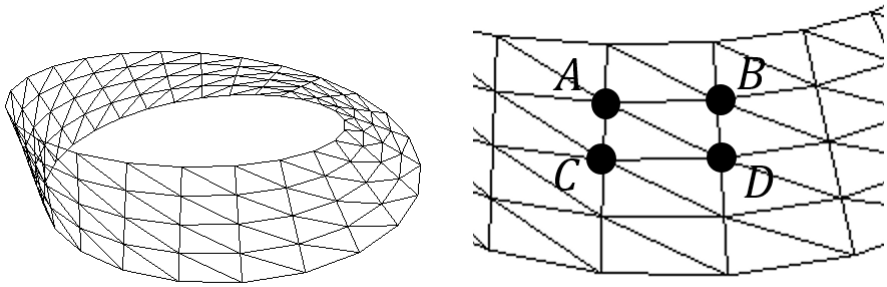


Figura 3.12: Parâmetros da fita de mobius.

Os parâmetros do comando `divsl` e `divscomp` correspondem às divisões na largura e no comprimento, respectivamente, que o utilizador pretende. De acordo com estes parâmetros as equações parmétricas da fita são:.

$$px = \left(1 + \frac{v}{2} \times cos(\frac{u}{2})\right) \times sen(u)$$
$$py = \frac{v}{2} \times sen\frac{u}{2}$$
$$pz = \left(1 + \frac{v}{2} \times cos(\frac{u}{2})\right) \times cos(u)$$
$$0 \leq u < 2\pi \ \wedge -\frac{L}{2} \leq v \leq \frac{L}{2}$$

Dividindo a fita pelo seu comprimento e pela sua largura, obtêm-se "Rectângulos" ACDB como exemplificado na figura 3.13(b).



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.13: Fita de Mobius

O algoritmo de geração de pontos para cada divisão da largura e para cada divisão do comprimento calcula as coordenadas dos pontos A,B,C,e D. As coordenadas do Ponto A podem ser calculadas usando como parâmetros $u = \frac{2\pi}{divscomp} \times divisaoComp$ e $v = -\frac{L}{2} + \frac{largura}{divslargura} \times divisaoLarg$, onde $divisaoComp$ e $divisaoLarg$ correspondem à divisão do comprimento e largura na qual o ponto A se encontra. Os parâmetros para os restantes pontos podem ser derivados destes, tendo em conta que o parâmetro u nos pontos B e D é acrescido de mais $\frac{2\pi}{divscomp}$ e nos pontos C e D o parâmetro v é acrescido em mais $\frac{largura}{divslargura}$.s

```
Figura& geraFitaMobius(Ponto3D o, float R, float                    l

    Para cada divisao da largura
        Para cada divisao do comprimento
            Calcular coordenadas dos pontos A, B, C e D
            Guardar os pontos pela ordem A-B-D-B-C-D
            Guardar os pontos pela ordem A-D-B-B-D-C
    return *this;
}
```

De notar que na fita de mobius, em cada iteração do ciclo interior, foram desenhadas as face da frente e a face de trás da superfície por forma a obter uma melhor perceção da figura ao ser visualizada.

3.3.10 Seashell

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

```
gerador seashell a b c n divsU divsV ficheiro.3d
```

Segundo Jorge Picado (Universidade de Coimbra), a superfície da concha é uma superfície tridimensional que pode ser vista como o resultado do deslocamento de uma curva C (a curva geratriz, que habitualmente é uma elipse) ao longo de uma espiral helicoidal H (a curva estrutural); o tamanho da curva C vai aumentando à medida que se desloca sobre H:

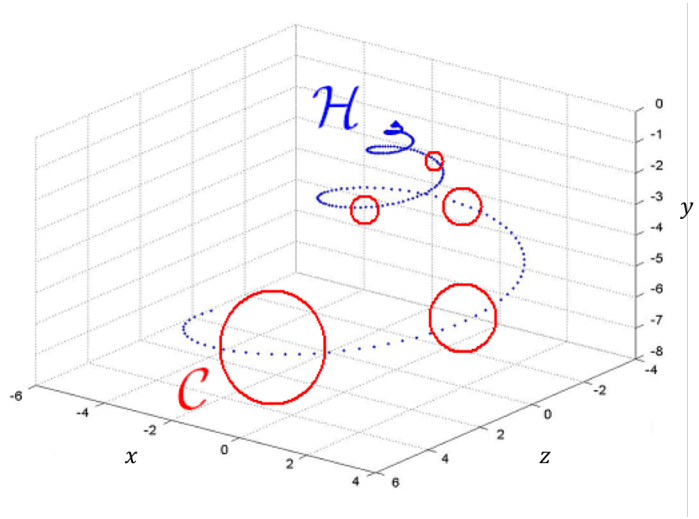


Figura 3.14: A forma de C descreve o perfil das secções da concha e da abertura da concha enquanto H determina a forma global da concha.

Qualquer ponto $P(px, py, pz)$ desta superfície pode ser definido pelas expressões matemáticas apresentadas abaixo, definidas por Seggern (2007), onde: u e v são os ângulos que definem a abertura e o alargamento da espiral, n o número de voltas da *Seashell*, a e b são o semi-eixo maior e menor da elipse, respetivamente, e c o comprimento da abertura.

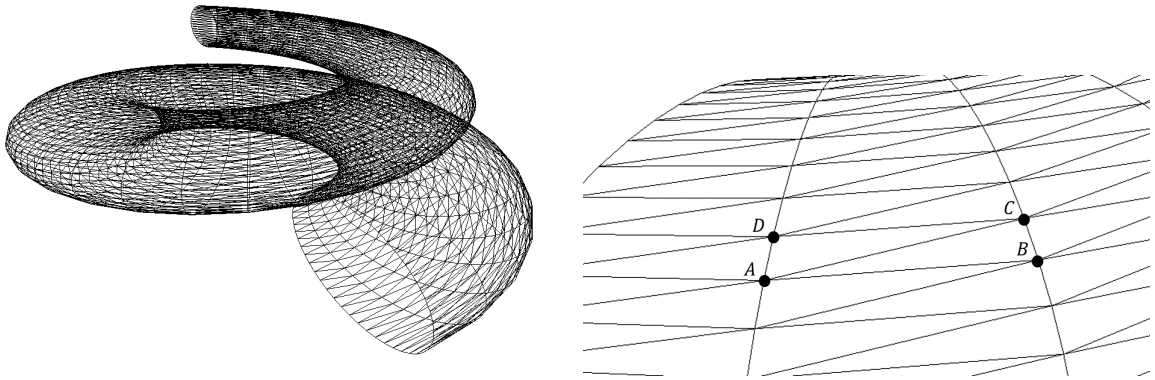
$$px = \left[\left(1 - \frac{v}{2 \times \pi}\right) \times (1 + cos(u)) + c\right] \times cos(n \times v)$$

$$py = \left[\left(1 - \frac{v}{2 \times \pi}\right) \times (1 + cos(u)) + c\right] \times sen(n \times v)$$

$$pz = \frac{b \times v}{2 \times \pi} + a \times sen(u) \times \left(1 - \frac{v}{2 \times \pi}\right)$$

Os parâmetros passados ao programa `divsU` e `divsV` podem ser vistos como as "fatias" e "camadas" da *Seashell*, respetivamente.

Tendo c camadas, sabe-se que a diferença em termos de ângulo v de um ponto que esteja numa camada superior para outro que esteja numa camada imediatamente inferior é de $\frac{2\times\pi}{c}$. Se considerarmos f fatias, o ângulo u de um ponto para outro que esteja numa fatia imediatamente a seguir é de $\frac{2\times\pi}{f}$.



(a) Representação geral (b) Esquema da superfície lateral

Figura 3.15: Seashell

Tendo em conta as expressões matemáticas supracitadas e o esquema da superfície lateral, na figura 3.15(b), segundo o qual os pontos vão ser percorridos, então a função `geraSeashell` pode ser representada pelo seguinte pseudo-código:

```
Figura& geraSeashell(Ponto3D o, float a, float b, float c,
                     int n, int divsU, int divsV) {
    Para cada fatia
        Para cada camada
            Calcular coordenadas dos pontos A, B, C e D
            Guardar os pontos pela ordem A-B-C-A-C-D
            Guardos os pontos pela ordem A-D-C-A-C-B
    return *this;
}
```

Pode-se notar pelo pseudo-código que os triângulos são desenhados segundo as duas orientações. Foi decidido fazer este trabalho para ser possivel visualizar o tubo da $Seashell$.

4. Conclusão

Os objetivos do trabalho foram cumpridos. Foi desenvolvido um gerador capaz de gerar pontos para as figuras pedidas e guarda-las num ficheiro, e um motor capaz de ler pontos a partir de ficheiros e de os desenhar. Além destes requisitos mínimos, foram realizados alguns extras, tanto no motor como no gerador. No gerador, além das figuras pedidas foram implementadas figuras extra: torus, fita de *Mobius*, elipsoide, *Seashell*, círculo e cilindro. No motor, foi implementada uma câmara sobre uma esfera que permite ao utilizador ver as figuras de diferentes ângulos. Além disso, foram adicionados menus e associadas ações ao rato e ao teclado que permitem controlar opções de visualização tais como o modo de visualização dos pontos, a cor de fundo e dos pontos da figura e a velocidade da câmara.

Como possíveis melhorias ao trabalho, destaca-se a possibilidade da mudança das funcionalidades disponibilizadas pela classe `CoordsPolares` e `CoordsEsfericas` para métodos de classe em vez de métodos de instância. O tratamento de argumentos por parte do gerador poderia ser tornado mais flexível e mais sólido com o recurso a bibliotecas de tratamento de argumentos como por exemplo a biblioteca *TCLAP* que permite, entre outras coisas, criar páginas de ajuda para os comandos e tornar alguns argumentos opcionais. Tal funcionalidade poderia ter sido implementada no projeto de forma a tornar alguns argumentos das figuras opcionais e as páginas de ajuda aos comandos seriam úteis para evitar a consulta de documentação (neste caso, este relatório) para utilização dos comandos. Por último, foi ainda considerada a implementação de uma câmara em modo primeira pessoa, no entanto decidiu-se implementar essa funcionalidade numa fase posterior do projeto. Esta decisão foi tomada por se considerar que nesta fase, o modo primeira pessoa pouco ajudaria o utilizador, visto todas as figuras estarem centradas na origem.

Como trabalho futuro, além da implementação da câmara em primeira pessoa e das melhorias referidas anteriormente, está prevista a expansão do motor de forma a aceitar XML's que além da especificação dos ficheiros de pontos das figuras, contenham instruções de colocação dessas mesmas figuras na cena, através de translações, rotações e escalas.

Achamos interessante o facto de integrarmos conhecimentos aprendidos noutras unidades curriculares, nomeadamente Análise, em que aprendemos as coordenadas esféricas e polares e as equações de transformação para cartesianas. Computação Gráfica permitiu-nos perceber que a sua aplicação prática era de extrema importância.