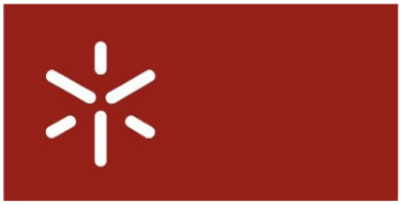

Computação Gráfica

Mestrado Integrado em Engenharia Informática

Fase 1

Primitivas gráficas

GRUPO 19 - PL2

A85227 João Pedro Rodrigues Azevedo

A85729 Paulo Jorge da Silva Araújo

A83719 Pedro Filipe Costa Machado

A89983 Paulo Jorge Moreira Lima

Braga
Fevereiro 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Este relatório é relativo ao trabalho prático da UC de Computação Gráfica que tem como objetivo final desenvolver uma *engine* para resolução de modelos **3D**, mostrando o seu potencial através da apresentação final de um sistema solar, na fase 4, com texturas e luzes.

1.2 Objetivos para a fase 1

A primeira fase teve como principais objetivos a criação de duas aplicações distintas: uma para gerar os ficheiros **.3d** com informação dos modelos, nomeadamente, os seus vértices, e uma *engine* que irá ler um **ficheiro de configuração XML** e apresentar os modelos.

Os modelos necessários para esta fase são os seguintes:

- **Plano**: dada a dimensão do lado;
- **Caixa**: dadas as dimensões X, Y e Z, e, opcionalmente, o nº de divisões;
- **Esfera**: dado o raio, *slices* e *stacks*;
- **Cone**: dado o raio, altura, *slices* e *stacks*.

Capítulo 2

Geração de modelos

2.1 Plano

Este modelo, sendo o mais simples de gerar, trata-se de um quadrado, no plano XZ, com uma dimensão do lado dada como parâmetro, sendo este, centrado na origem e formado por dois triângulos, como se pode ver na figura seguinte:

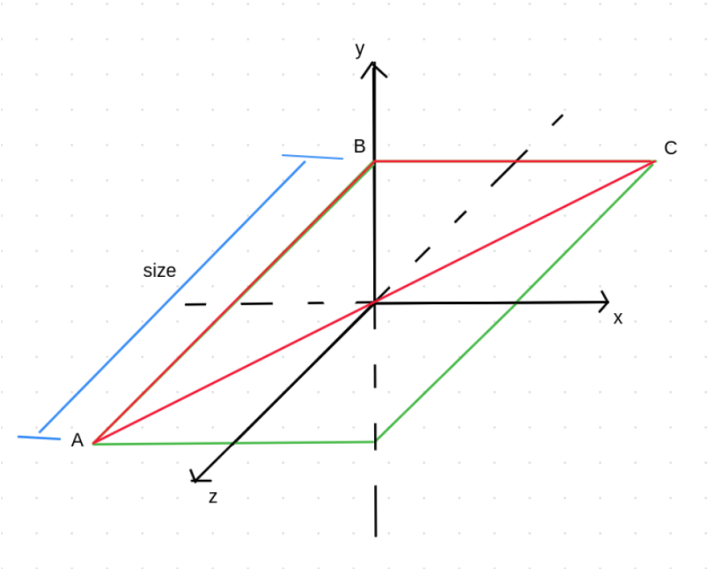

Figura 2.1: Representação do plano em 3 dimensões.

Como exemplo da criação do triângulo a vermelho, representado pelos vértices **A, B** e **C**, tomando a ordem C → B → A, temos que:

- C = (size / 2, 0, -size/2)
- B = (-size / 2, 0, -size/2)
- A = (-size / 2, 0, size/2)

O segundo triângulo, representado a verde, segue o mesmo raciocínio.

2.2 Caixa

A caixa recebe como parâmetros: as dimensões de X, Y e Z, e também, opcionalmente, o nº de divisões. Na nossa implementação, para **X** *divisions* teríamos em cada face $(1+X)^2$ rectângulos.

A figura seguinte ilustra uma caixa com uma divisão ($(1 + 1)^2 = 4$) e, deste modo, 4 rectângulos em cada face:

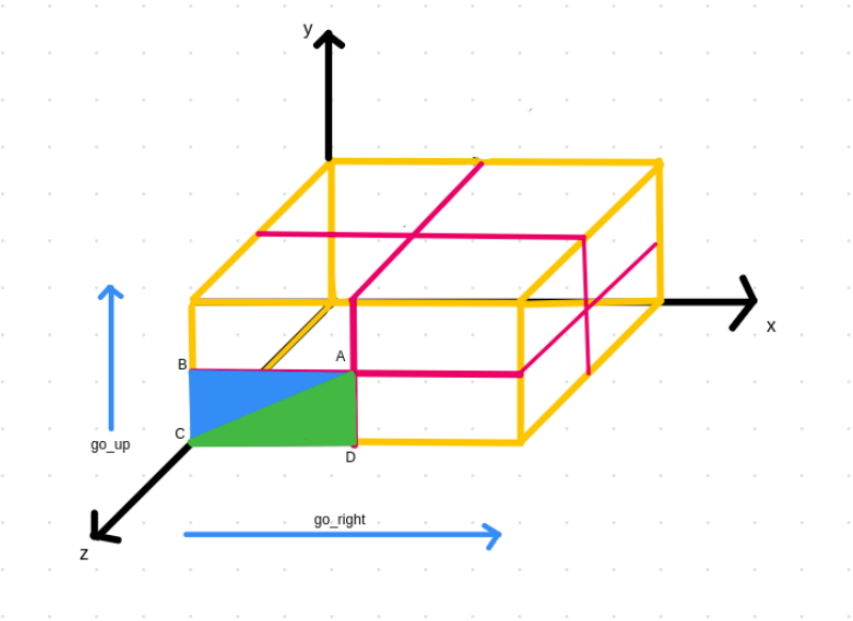

Figura 2.2: Representação de uma caixa com uma divisão.

Para gerar todos os rectângulos utilizámos duas variáveis (**go_up** e **go_down**) para percorrer a caixa a partir de um dos pontos da face principal para cada plano.

Neste caso em particular, com o eixo Z no seu valor máximo, começamos a gerar a partir do vértice C e concluímos que:

- A = (x/(divisions+1) * (go_right+1), y/(divisions+1) * (go_up+1), z)
- B = (x/(divisions+1) * (go_right), y/(divisions+1) * (go_up+1), z)
- C = (x/(divisions+1) * (go_right), y/(divisions+1) * (go_up), z)

Os rectângulos são constituídos por dois triângulos e são desenhados, segundo o nosso algoritmo, da esquerda para a direita, com o **go_right**, subindo, iterativamente, de nível, com o **go_up**.

Para cada iteração do ciclo mais interior, ou seja, o que utiliza a variável de controlo **go_right**, são desenhados 6 rectângulos, um para cada face, como é mostrado na figura seguinte:

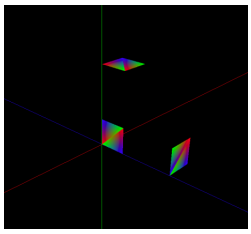

Figura 2.3: Modelo da caixa ao fim de uma iteração do **go_right**.

Para cada iteração do ciclo exterior, ou seja, o que utiliza a variável de controlo **go_up**, são desenhados 6 * (*divisions* + 1) rectângulos para cada face, como é mostrado na figura seguinte:

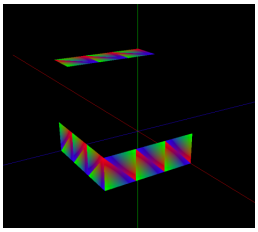

Figura 2.4: Modelo da caixa ao fim de uma iteração do **go_up**.

Nota: Não são visíveis metade dos rectângulos que são desenhados pois depende do ângulo em que a câmara se apresenta.

2.3 Esfera

A esfera recebe como parâmetros: o raio, os *slices* e os *stacks*. Para desenhar a esfera consideramos o ângulo α ($\alpha = (2 * \pi)/slices$) que nos permite obter as circunferências presentes em cada *stack*, com raio sucessivamente menor.

Na figura, $\alpha * i$ e $\alpha * (i + 1)$ representam as amplitudes com o ângulo atual e seguinte, somando-lhe α.

O β representa a amplitude da *stack*, nunca tomando valores superiores a 180 graus. No nosso algoritmo, consideramos uma variável adicional, o *next_beta_top* que representa o β da *stack* superior, de forma a calcular as coordenadas dos pontos que formam os triângulos com vértices nessa *stack*, nomeadamente, o triângulo amarelo (vértices A' e B').

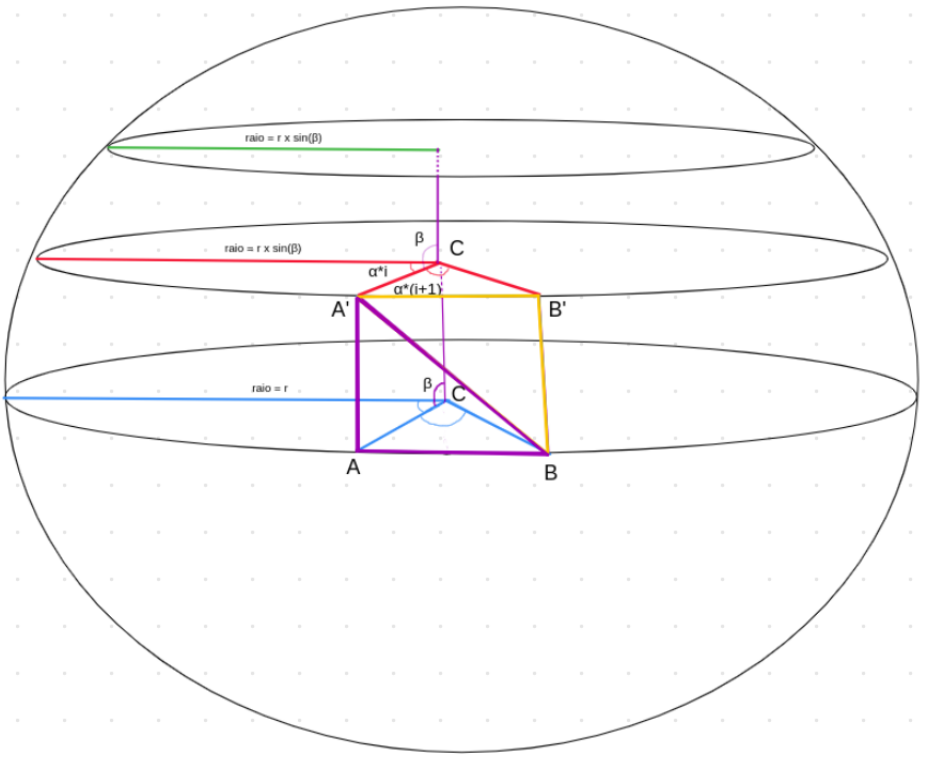

Figura 2.5: Representação do desenho da esfera.

Com isto, definimos os seguintes vértices do exemplo apresentado acima, para o triângulo a roxo:

- A' = (r*cos(next_beta_top) * sin(αi), r*sin(next_beta_top), r*cos(next_beta_top) * cos(αi))
- A = (r*cos(beta_top) * sin(αi), r*sin(beta_top), r*cos(beta_top) * cos(αi))
- B = (r*cos(beta_top) * sin(α(i+1)), r*sin(beta_top), r*cos(beta_top) * cos(α(i+1)))

Nota: A referência à variável **beta_top** é relativa à variação do β na semi-esfera superior. Da mesma forma, para os cálculos da semi-esfera inferior, temos a variável **beta_bottom**.

Desta forma, podemos proceder à variação dos ângulos definidos acima para desenhar os *slices*, que compõem cada *stack* da esfera, através de dois triângulos.

2.4 Cone

O Cone recebe como parâmetros: o raio, a altura, os *slices* e os *stacks*. Para desenhar o cone consideramos o ângulo α ($\alpha = (2 * \pi)/slices$) que nos permite obter as circunferências presentes em cada *stack*, com raio sucessivamente menor.

Na figura, $\alpha * i$ e $\alpha * (i + 1)$ representam as amplitudes com o ângulo atual e seguinte, somando-lhe α.

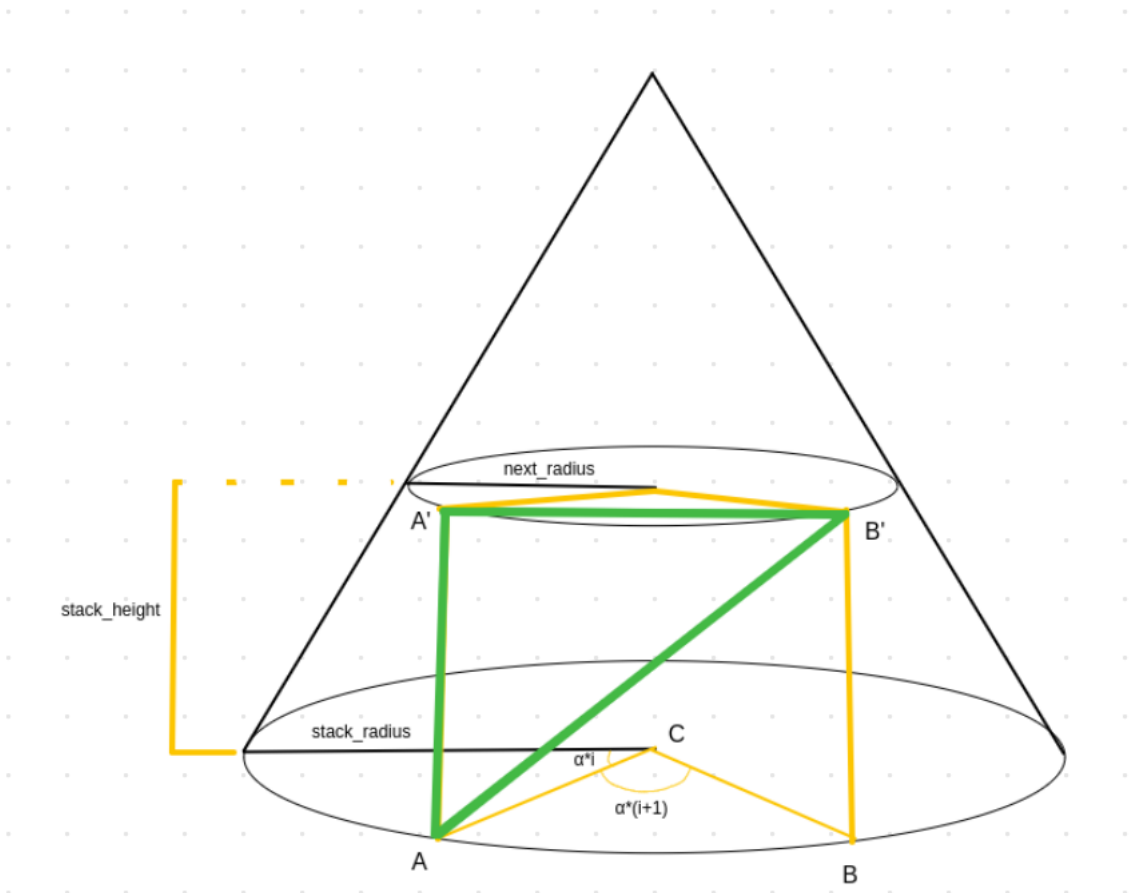

Figura 2.6: Representação do desenho do cone.

Na figura, o *stack_height* representa a distância vertical entre *stacks* e, para isso, fizemos o seguinte cálculo:

$$\text{stack\_height} = \text{height}/\text{stacks}$$

O cone, ao contrário da esfera, tem uma altura que evolui de forma linear e, deste modo, para obter as alturas das *stacks* subsequentes multiplicámos o fator *stack_height* pelo número da *stack* atual, que no código se traduz na variável de controlo de ciclo **j**.

Já o raio de cada *stack* evolui retirando ao raio inicial um fator constante, multiplicado pelo número da *stack* atual, sendo esse fator **radius/stacks**.

- A = (stack_radius * sin(αi), j * stack_height, stack_radius * cos(αi))
- B' = (next_radius * sin(α(i+1)), (j+1) * stack_height, next_radius * cos(α(i+1))
- A' = (next_radius * sin(α(i+1)), (j+1) * stack_height, next_radius * cos(αi)

Desta forma, podemos proceder à variação do α para desenhar os *slices*, que compõem cada *stack* do cone, através de dois triângulos.

Capítulo 3

Gerador, motor e arquitetura.

Na arquitetura da nossa aplicação temos 3 pastas principais:

- **generator**: contém todo o código do gerador de modelos (algoritmos e funções auxiliares);
- **engine**: contém todo o código do motor que inicializa os modelos a partir dos ficheiros *modelo.3d* e executa-os utilizando o OpenGL;
- **examples**: contém 2 pastas, uma com a localização dos modelos gerados pelo **generator** e outra com o ficheiro **xml** que será carregado pelo **engine**.

3.1 Gerador

Na arquitetura do **generator** decidimos criar um módulo contendo as funções auxiliares para a geração de cada modelo[1], recebendo também o ficheiro *output* como argumento (que será guardado em *"examples/Model-Read-Tests"*).

Em termos da sua **execução**, nas próximas secções serão apresentados os comandos possíveis para cada modelo, no entanto, em caso de dúvida, pode-se correr o **generator** sem argumentos de modo a visualizar os diferentes comandos, como por exemplo: "**./generator**".

Já os ficheiros **.3d** gerados por esta aplicação implementam a estrutura sugerida pelo enunciado, ou seja, com o número de vértices total na primeira linha seguido de todos os vértices do modelo, um por linha:

```
3600 //nr de vertices
0 0 0 //x y z, floats
1.54508 0 4.75528
0 0 5
1.46783 0.5 4.51752
...
```

3.1.1 Plano

O protótipo da função que recebe uma dimensão **size** e escreve para o ficheiro **file_name** encontra-se de seguida:

```
void generate_plane_3d (double size, string file_name)
```

[1]Encontrado em "Model-Generator/model-generator.h", dentro da pasta que contém o código do **generator**.

Para gerar o primeiro triângulo, cujo exemplo foi apresentado no capítulo 2, escrevemos o seguinte código:

```
outfile << size/2 << " " << 0 << " " << -size/2 << endl;
outfile << -size/2 << " " << 0 << " " << -size/2 << endl;
outfile << -size/2 << " " << 0 << " " << size/2 << endl;
```

Onde **outfile** representa a *ofstream* de C++ que escreve para o ficheiro *output* os vértices indicados. O segundo triângulo segue o mesmo raciocínio.

3.1.2 Caixa

O protótipo da função que recebe as dimensões **x, y e z**, o **número de divisões (divisions)** e escreve para o ficheiro **file_name** encontra-se de seguida:

```
void generate_box_3d (double x, double y, double z,
                      int divisions, string file_name)
```

Como explicado no capítulo 2, utilizamos duas variáveis auxiliares **go_up** e **go_right** para desenhar os rectângulos de cada face mediante o número de divisões:

```
for (int go_up = 0; go_up < (divisions + 1); go_up++) {

    for (int go_right = 0; go_right < (divisions + 1); go_right++) {

        ...
    }
}
```

Dentro do ciclo **for** mais interior, ou seja, o da variável **go_right** desenhamos, em cada iteração, um dos rectângulo que compõem cada face, para todas as faces, isto é, com **X, ou Y, ou Z** a tomar o valor **máximo** (dado como argumento) ou a tomar o valor **zero**, por exemplo:

```
/* PLANO Z max */

//First triangle
outfile << (x/(divisions+1)) * (go\_right + 1) << " "
        << (y/(divisions+1)) * (go\_up + 1) << " "
        << z << endl;
(...)

/* PLANO X = 0 */

//First triangle
outfile << 0 << " "
        << (y/(divisions+1)) * (go_up + 1) << " "
        << (z/(divisions+1)) * (go_right + 1) << endl;
(...)
```

Assim sendo, o modelo da nossa caixa será gerado com início no vértice da origem do referencial, tomando apenas valores positivos para cada um dos eixos, ou seja, não é centrado na origem.

3.1.3 Cone

O protótipo da função que recebe o **raio, a altura**, o **número de *stacks***, o **número de *slices*** e escreve para o ficheiro **file_name** encontra-se de seguida:

```
void generate_cone_3d (double radius, double height,
int slices, int stacks, string file_name)
```

Inicialmente definimos algumas das variáveis mais importantes para o cálculo dos vértices:

```
double alpha = 2*M_PI/slices;
//altura de cada stack
double stack_height = height/stacks;
//raio da stack atual e da proxima stack
double stack_radius, next_radius;

(...)

for(int j=0; j < stacks;j++) {

    stack_radius = radius - (j*radius/stacks);
    next_radius = radius - ((j+1)*radius/stacks);

    for(int i=0; i < slices;i++) {

        // Bottom of each stack
        outfile << 0.0 << " " << j*stack_height << " "
                << 0.0 << endl;
        (...)

        // Sides of each stack

        //triangle 1
        outfile << stack_radius * sin(alpha*i) << " "
                <<  j*stack_height << " "
                <<  stack_radius * cos(alpha*i) << endl;
        (...)
    }
}
```

Sendo que para calcular o raio da próxima *stack* (*next_radius*) utilizamos o raio principal (*radius*) retirando-lhe um fator que aumenta, de forma linear, à medida que vamos desenhando cada *stack*, diminuindo o valor desse raio. Os triângulos para cada *slice* de cada *stack* são gerados como se pode ver no excerto código acima e como foi explicado no capítulo 2.

3.1.4 Esfera

O protótipo da função que recebe o **raio**, o **número de *stacks***, o **número de *slices*** e escreve para o ficheiro **file_name** encontra-se de seguida:

```
void generate_sphere_3d (double radius, double slices,
int stacks, string file_name)
```

O primeiro passo foi criar as variáveis mais importantes, assim como no cone, na geração dos vértices:

```
double alpha = 2*M_PI/slices;
double beta_top = M_PI/2 + (M_PI / stacks),
next_beta_top = M_PI/2 + (M_PI / stacks);
double beta_bottom = -M_PI/2 - (M_PI / stacks),
next_beta_bot = -M_PI/2 - (M_PI / stacks);
```

Sendo que, as variáveis com sufixo "..._top"serão utilizadas na semi-esfera superior e as terminadas em "..._bot"serão utilizadas na semi-esfera inferior.

```
for(int j=0; j < stack_iter + 1;j++) {

        beta_bottom += M_PI / stacks;
        beta_top -= M_PI / stacks;

        for(int i=0; i < slices;i++) {

        (...)
        //Top stacks of the sphere
        outfile << 0.0 << " "<< radius * sin(beta_top) << " "
                << 0.0  << endl;
        (...)
        //Bottom stacks of the sphere
        outfile << 0.0 << " "<< radius * sin(beta_bottom)
                << " "<< 0.0  << endl;
        (...)
        //triangle 1 - top
        outfile << radius * cos(next_beta_top) * sin(alpha*i) << " "
                <<  radius * sin(next_beta_top) << " "
                <<  radius * cos(next_beta_top) * cos(alpha*i)  << endl;
        (...)
        //triangle 2 - bottom
        outfile << radius * cos(next_beta_bot) * sin(alpha*i) << " "
                <<  radius * sin(next_beta_bot) << " "
                <<  radius * cos(next_beta_bot) * cos(alpha*i)  << endl;
        (...)
    }
}
```

Deste modo, desenhamos os círculos de cada camada e os triângulos que compõem a superfície da esfera, em cada *slice* e *stack*, garantindo que, no fim da execução do ciclo *for* desenhámos ambas as semi-esferas, inferior e superior.

3.2 Engine

O motor que lê os modelos a partir dos ficheiros **.3d** indicados no ficheiro **XML** da nossa aplicação encontra-se na pasta **engine** da raiz do projeto.

Trata-se de um programa em C++ que lê os vértices dos modelos selecionados no ficheiro de configuração, guarda-os em estruturas na memória principal e projeta-os numa janela com a ajuda de OpenGL.

3.2.1 Ficheiro de configuração xml

O ficheiro de configuração **xml** possui um formato simples, visto que ainda estamos na fase 1. Um exemplo de configuração seria o seguinte:

```
<scene>
    <model file = "cone.3d" />
    <model file = "plane.3d" />
</scene>
```

Isto assumindo que os ficheiros "cone.3d"e "plane.3d"se encontram na pasta examples/Model-Read-Tests, tendo sido, previamente, gerados pelo programa **generator**.

A leitura deste ficheiro é um processo relativamente simples: procura-se por uma **root tag** chamada **scene** e a partir da mesma, num ciclo *for*, percorre-se todos os **XMLElements** cuja **tag** seja "**model**".

Definimos o caminho principal para o ficheiro xml no *main.cpp* como sendo:

```
#define xml_config_file "ex-config-1.xml"
```

Nota: para auxiliar o *parsing* do ficheiro xml utilizamos uma ferramenta sugerida pelo enunciado, denominada por **tinyxml2**, cujo código fonte se encontra na pasta do código do **engine**.

3.2.2 Modelos e estruturas de dados

Para facilitar a organização das ideias e otimizar o dinamismo da memória principal, criamos uma estrutura principal para guardar cada modelo, com 2 classes C++:

- **MODEL_INFO**: Guarda o nome do ficheiro que contém o modelo e os vértices do mesmo num **vector** de C++:

```
class MODEL_INFO {
    private:
        string name;
        vector<POINT_3D> vertices;
(...)
}
```

- **POINT_3D**: Guarda a representação de um ponto em 3 dimensões como composição de 3 floats numa classe C++:

```
class POINT_3D {
    public:
        POINT_3D(float x, float y, float z);
    private:
            float x; float y; float z;
    (...)
}
```

E carregamos os modelos presentes no ficheiro de configuração para estruturas próprias lendo, posteriormente, o conteúdo dos ficheiros por linha, separando os valores de cada linha em 3 *floats*.

Capítulo 4

Conclusão

Esta fase serviu como motor de desenvolvimento para as próximas fases, tendo sido desenvolvido tudo o que foi pedido de forma prática e, esperemos, simples de entender.

A utilização de C++ na arquitetura do nosso projeto foi importante, não só para conhecer mais uma língua, mas também na utilização das ferramentas que a mesma nos dá para otimizar o processo de inicialização de estruturas e renderização de modelos. Mais se refere à importância de ter passado por a experiência de interpretar xml visto que será importante para as próximas fases.

Foi portanto possível gerar todas as primitivas gráficas pedidas e carregar as mesmas no nosso **engine** gráfico o que cumpre os requisitos desta primeira fase.