Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Computação Gráfica

Mário Santos (a70697)
Pedro Costa (a85700)
Rui Azevedo (a80789)
6 de Março de 2020

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

O presente relatório diz respeito à primeira fase do trabalho da cadeira de Computação Gráfica. É pretendido com este trabalho desenvolver duas aplicações, um gerador de ficheiros que contêm uma série de vértices correspondentes a primitivas gráficas, e um motor capaz de ler esses vértices e desenhar a primitiva correspondente. A linguagem usada no desenvolvimento do projecto é o **C++** e tirou-se partido das bibliotecas do ***openGL*** para a representação gráfica das primitivas.

2 Arquitectura do projecto

O projecto é composto por duas aplicações, completamente isoladas uma da outra, designadas por ***generator*** e ***engine***.

O executável ***generator*** é responsável por toda a algoritmia capaz de desenhar vértices para uma determinada primitiva gráfica. Esse conjunto de vértices, uma vez gerados, são guardados em ficheiro para posterior leitura e representação gráfica dos mesmos. O programa recebe como parâmetros o nome da primitiva, uma lista de parâmetros característicos de cada primitiva e , por último, o nome do ficheiro onde serão guardados os pontos gerados.

./generator nome_ primitiva [p1,...,pn] nome_ primitiva.3d

Uma vez calculados os vértices, o programa ***engine*** é responsável por ler os ficheiros gerados pelo ***generate*** e fazer a representação gráfica dos mesmos. O ***engine*** é completamente independente da lógica de criação de pontos, o seu objectivo é simplesmente ler pontos de um ficheiro apontado por um ***XML***. O executável é gerado com o seguinte conjunto de comandos:

cmake -B build ; cd build ; make ; ./Engine

Nas secções seguintes, irá ser apresentado, com mais detalhe, a composição de cada um dos componentes deste sistema, bem como o *output* final.

3 Generator

Nesta secção é abordada em detalhe a arquitectura e componentes que compõem o gerador de vértices. Irá ser apresentada a arquitectura das classes desenvolvidas para a representação computacional das primitivas gráficas, bem como os algoritmos desenvolvidos para calcular, de forma sistemática, os vértices de cada figura.

3.1 Arquitectura de classes

Para melhor organizar e encapsular o código, tirou-se partido dos mecanismos de herança da linguagem ***C++***.

Para a representação de um vértice, foi criada uma classe designada por ***Vertex*** que contém a informação relativa a um ponto, isto é, as suas coordenadas.

De seguida, foi criada uma classe abstracta designada por ***primitive*** que contém um vector de vértices que irá ser usado para armazenar os vértices de uma determinada primitiva. Esta classe obriga a que as suas sub-classes implementam o método ***genVertex()*** responsável por gerar os vértices correspondentes a uma primitiva.

Por último, foram criadas as sub-classes ***plane***, ***box***, ***cone***, ***sphere***, onde estão contidos os algoritmos de geração.

Na imagem abaixo é apresentada o diagrama de classes desenvolvido para modelar o sistema a desenvolver.

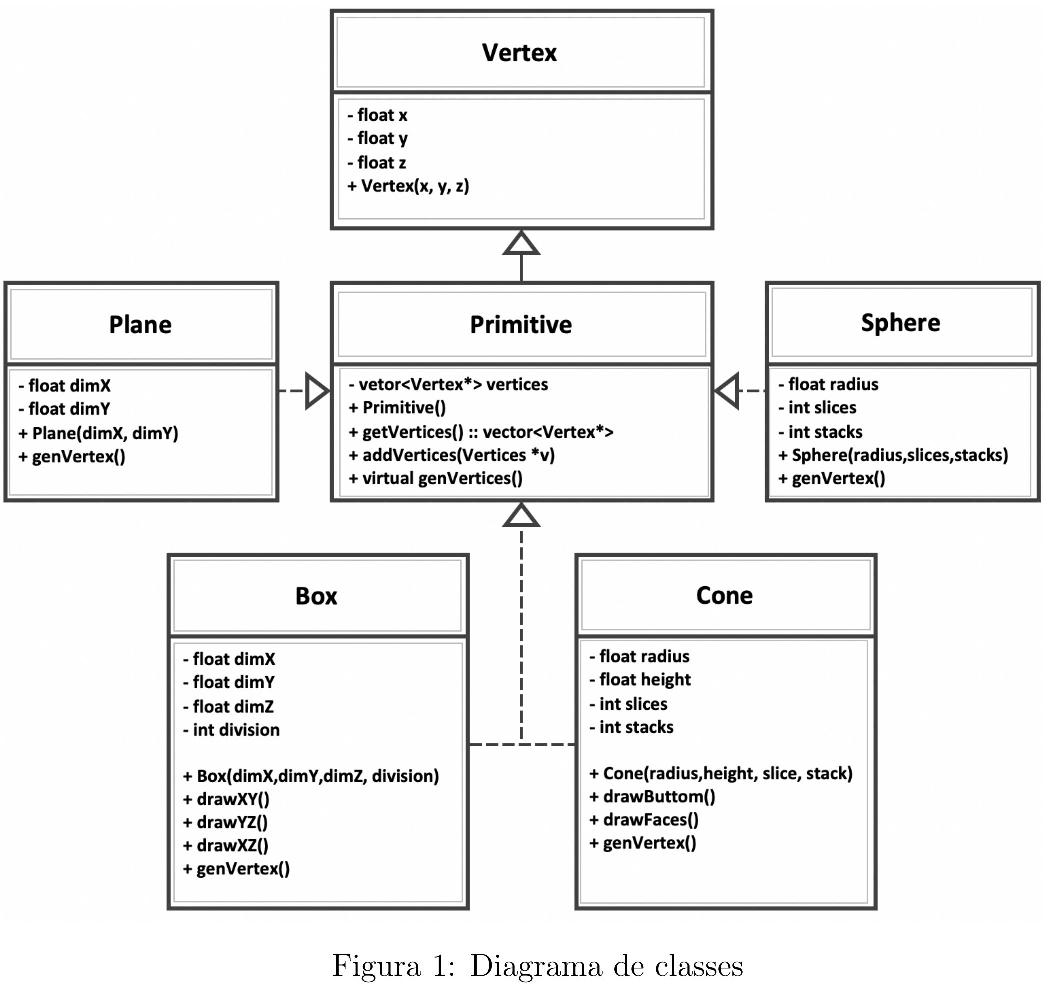

Figura 1: Diagrama de classes

3.2 Primitivas

Nesta secção são apresentados os algoritmos desenvolvidos para gerar cada uma das primitivas. É de notar que todas as primitivas foram criadas usando triângulos e que todos os vértices foram desenhados por uma ordem específica, ordem da mão direita, para que seja possível representar as faces que estão viradas para a câmara bem como as que, do ponto de vista da câmara, não são visíveis.

3.2.1 Plano

- **Parâmetros**: (dim_x, dim_z)
- **Algoritmo**: O plano é desenhado no centro do plano XZ. Para isso, calcula-se o ponto médio de cada um dos eixos e desenham-se dois triângulos para representar o plano. Um triângulo formado pelos vértices 1, 3 e 4 e o outro formado pelos vértices 1,2 e 3.
- **Cálculos auxiliares:**

$$p = (-dim_x/2, -dim_y/2, dim_z/2)$$

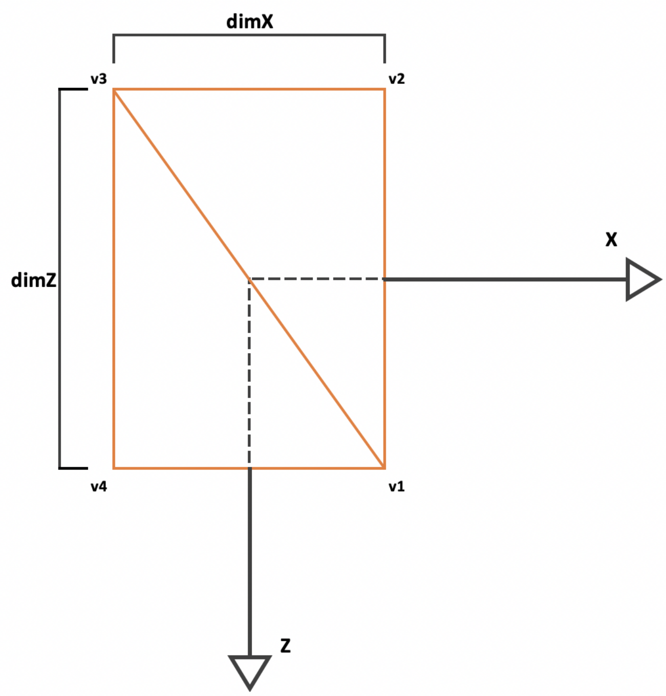

Figura 2: Esquema do plano

Algorithm 1 drawPlane

function DRAWPLANE(dim_x, dim_z)
    $c_x \leftarrow \mathrm{dim}_x/2$
    $c_z \leftarrow \mathrm{dim}_z/2$
vertices:
    $v_1 \leftarrow (\mathrm{c}_x, 0, c_z)$
    $v_2 \leftarrow (\mathrm{c}_x, 0, -c_z)$
    $v_3 \leftarrow (\text{-}\mathrm{c}_x, 0, -c_z)$
    $v_4 \leftarrow (\text{-}\mathrm{c}_x, 0, c_z)$
triangles:
    $t_l \leftarrow (\mathrm{v}_1, v_3, v_4)$
    $t_r \leftarrow (\mathrm{v}_1, v_2, v_3)$

3.2.2 Caixa

- **Parâmetros**: $(dim_x, dim_y, dim_z, divisions)$
- **Algoritmo**: O algoritmo para o cálculo dos vértices da caixa foi dividido em três funções, uma responsável por desenhar as faces da caixa paralelas ao plano XY, outra as faces paralelas ao plano YZ e, por último, uma responsável por desenhar as faces paralelas ao plano XZ. Cada um dos algoritmos toma como ponto de partida o ponto $A = (-dim_x/2, -dim_y/2, dim_z/2)$ e, a partir daí começa-se a desenhar a face correspondente. Os algoritmos começam por desenhar uma fileira da caixa, isto é, por exemplo, no caso das funções ***drawXY*** e ***drawYZ***, os triângulos são desenhados em altura, sendo que, ao fim do desenho de uma coluna de triângulos, é aplicada uma deslocação com a relação entre as dimensões respectivas da caixa e o número de divisões.
- **Cálculos auxiliares:**

$$
\begin{aligned}
&p = (-dim_x/2, -dim_y/2, dim_z/2)\\
&d_x = dim_x/divisions\\
&d_y = dim_y/divisions\\
&d_z = dim_z/divisions
\end{aligned}
$$

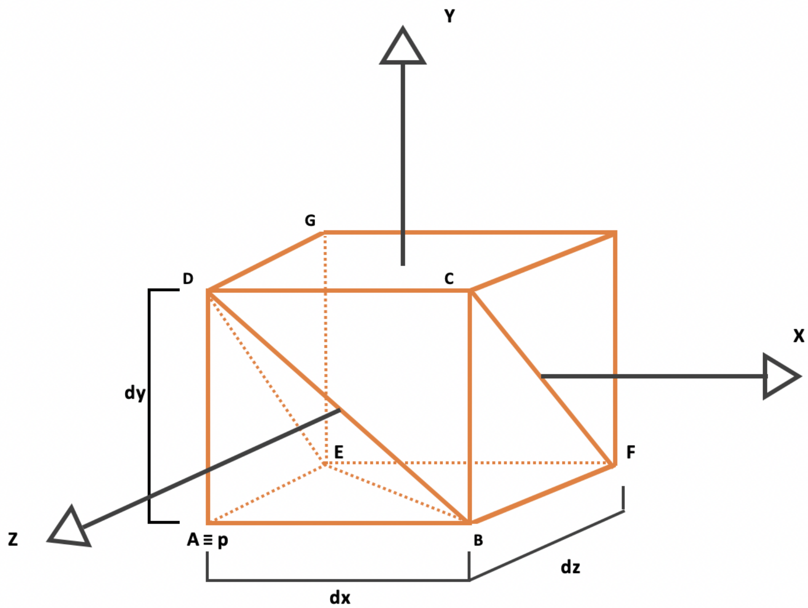

Figura 3: Esquema da caixa

Algorithm 2 drawBox

function DRAWBOX($dim_x, dim_y, dim_z, divisions$)
    p $\leftarrow$(-dim$_x/2, -dim_y/2, dim_z/2$)
    $d_x \leftarrow$dim$_x/divisions$
    $d_y \leftarrow$dim$_y/divisions$
    $d_z \leftarrow$dim$_z/divisions$
drawXY:
    for $i \leftarrow 0$ to $divisions$ **do**
        $p_y \leftarrow$-dim$_y/2$
        for $j \leftarrow 0$ to $divisions$ **do**
            //**Front face**
            $t_1 = ((p_x, p_y, p_z), (p_x + d_x, p_y, p_z), (p_x, p_y + d_y, p_z))$
            $t_2 = ((p_x + d_x, p_y, p_z), (p_x + d_x, p_y + d_y, p_z), (p_x, p_y + d_y, p_z))$
            //**Back face**
            $t_3 = ((p_x, p_y, p_z - dim_Z), (p_x, p_y + d_y, p_z - dim_Z), (p_x + d_x, p_y, p_z - dim_Z))$
            $t_4 = ((p_x + d_x, p_y, p_z - dim_Z), (p_x, p_y + d_y, p_z - dim_Z), (p_x + d_x, p_y + d_y, p_z - dim_Z))$
            $p_y \leftarrow$p$_y + d_y$
        $p_x \leftarrow$p$_x + d_x$

Algorithm 3 drawBox(continuação)

drawYZ:
for $i \leftarrow 0$ to $divisions$ **do**
  for $j \leftarrow 0$ to $divisions$ **do**
    //**Left face**
    $t_1 = ((p_x, p_y, p_z), (p_x, p_y + d_y, p_z), (p_x, p_y, p_z - d_z))$
    $t_2 = ((p_x, p_y + d_y, p_z), (p_x, p_y + d_y, p_z - d_z), (p_x, p_y, p_z - d_z))$
    //**Right face**
    $t_3 = ((px + dimX, p_y, p_z), (p_x + dim_X, p_y, p_z - d_z), (p_x + dim_X, p_y + d_y, p_z))$
    $t_4 = ((p_x + dim_X, p_y + d_y, p_z), (p_x + dim_X, p_y, p_z - d_z), (p_x + dim_X, p_y + d_y, p_z - d_z))$
    $p_y \leftarrow \mathrm{p}_y + d_y$
  $p_z \leftarrow \mathrm{p}_z - d_z$

drawXY:
  for $i \leftarrow 0$ to $divisions$ **do**
    $p_z = dim_z/2$
    for $j \leftarrow 0$ to $divisions$ **do**
      //**Bottom face**
      $t_1 = ((p_x, p_y, p_z), (p_x, p_y, p_z - d_z), (p_x + d_x, p_y, p_z - d_z))$
      $t_2 = ((p_x, p_y, p_z), (p_x + d_x, p_y, p_z - d_z), (p_x + d_x, p_y, p_z))$
      //**Upper face**
      $t_3 = ((p_x, p_y + dim_Y, p_z), (p_x + d_x, p_y + dim_Y, p_z - d_z), (p_x, p_y + dim_Y, p_z - d_z))$
      $t_4 = ((p_x, p_y + dim_Y, p_z)), (p_x + d_x, p_y + dim_Y, p_z), (p_x + d_x, p_y + dim_Y, p_z - d_z))$
      $p_z \leftarrow \mathrm{p}_z - d_z$
    $p_x \leftarrow \mathrm{p}_x + d_x$

3.2.3 Esfera

- **Parâmetros**: $(radius, slices, stacks)$
- **Algoritmo**: Para o cálculo dos vértices que compõem uma esfera foi usado um sistema de coordenadas esféricas por ter uma estrutura que torna a geração dos vértices muito mais intuitiva e menos complexa. Este sistema de coordenadas é definido por um ângulo em torno do eixo y (ângulo α), por outro em torno do eixo z (ângulo β) e por um raio que corresponde à distância de um ponto à origem. Sabe-se que todos vértices da esfera estão à mesma distância do centro dos eixos, por isso, pode-se tomar como garantido que o parâmetro ***radius*** é comum a todos os pontos. Dado isto, pode-se concluir que a única coisa que varia entre os diferentes pontos são os ângulos α e β. Pode-se ver a baixo os cálculos auxiliares para a construção da primitiva onde d_x e d_y correspondem, respectivamente, ao deslocamento do ângulo α e β em cada iteração. Uma vez que a esfera é desenhada *top-buttom*, pode-se também concluir os intervalos de valores dos respectivos ângulos.
- **Cálculos auxiliares:**

$$d_x = 2\pi/slices \qquad \alpha \in [0, 2\pi]$$
$$d_y = \pi/stacks \qquad \beta \in [-\pi/2, \pi/2]$$

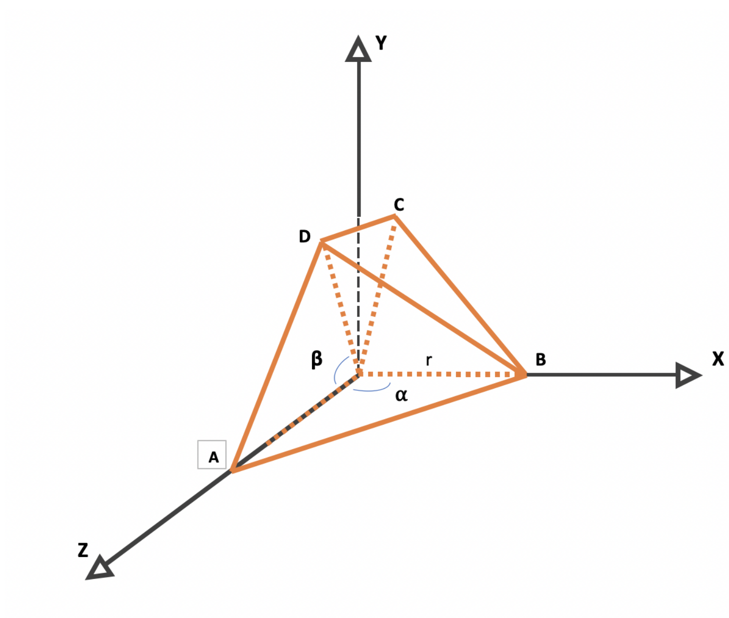

Figura 4: Esquema da esfera

Algorithm 4 drawCircle

```
function drawCircle(radius, slices, stacks)
    d_x = 2π/slices
    d_y = π/stacks

    for i ← 0 to stacks do
        β = π/2 − d_y ∗ (i + 1)
        for j ← 0 to slices do
            α = d_x ∗ j
            x_1 ← radius ∗ sin(α) ∗ cos(β);
            y_1 ← radius ∗ sin(β)
            z_1 ← radius ∗ cos(α) ∗ cos(β)

            x_2 ← radius ∗ sin(α + d_x) ∗ cos(β)
            y_2 ← radius ∗ sin(β)
            z_2 ← radius ∗ cos(α + d_x) ∗ cos(β)

            x_3 ← radius ∗ sin(α) ∗ cos(β + d_y)
            y_3 ← radius ∗ sin(β + d_y)
            z_3 ← radius ∗ cos(α) ∗ cos(β + d_y)

            x_4 ← radius ∗ sin(α + d_x) ∗ cos(β + d_y)
            y_4 ← radius ∗ sin(β + d_y);
            z_4 ← radius ∗ cos(α + d_x) ∗ cos(β + d_y)

            t_1 ← ((x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3))
            t_2 ← ((x_2, y_2, z_2), (x_4, y_4, z_4), (x_3, y_3, z_3))

```

3.2.4 Cone

- **Parâmetros**: $(radius, height, slices, stacks)$
- **Algoritmo**: O algoritmo de geração de pontos para o cone é ligeiramente parecido com o da esfera. Neste caso, foi adoptado o sistema de coordenadas polares, sendo este sistema definido por um ângulo α, correspondente à rotação em torno do eixo y, e um raio que representa a distância de um vértice ao centro dos eixos. Mais uma vez, o algoritmo foi dividido em duas partes, uma que desenha a base do cone e outra que desenha as suas faces. Para o cálculo da base, o valor da componente y dos diferentes vértices é sempre igual com o valor de 0, sendo que apenas é necessário variar o ângulo α. Para o cálculo

das faces do cone, são definidos os seguintes valores: h_i e h_s que representam a altura inferior e superior, respectivamente, associadas a uma *stack*, r_i e r_s que representam, respectivamente o raio inferior e superior dos vértices de uma *stack*.

- **Cálculos auxiliares:**

$$d_x = 2\pi/slices \qquad h = height/stacks$$
$$h_i = h * i, i \in [0, stacks[\qquad h_s = h * (i+1), i \in [0, stacks[$$
$$ri = radius - ((radius * hi)/height) \qquad rs = radius - ((radius * hs)/height);$$

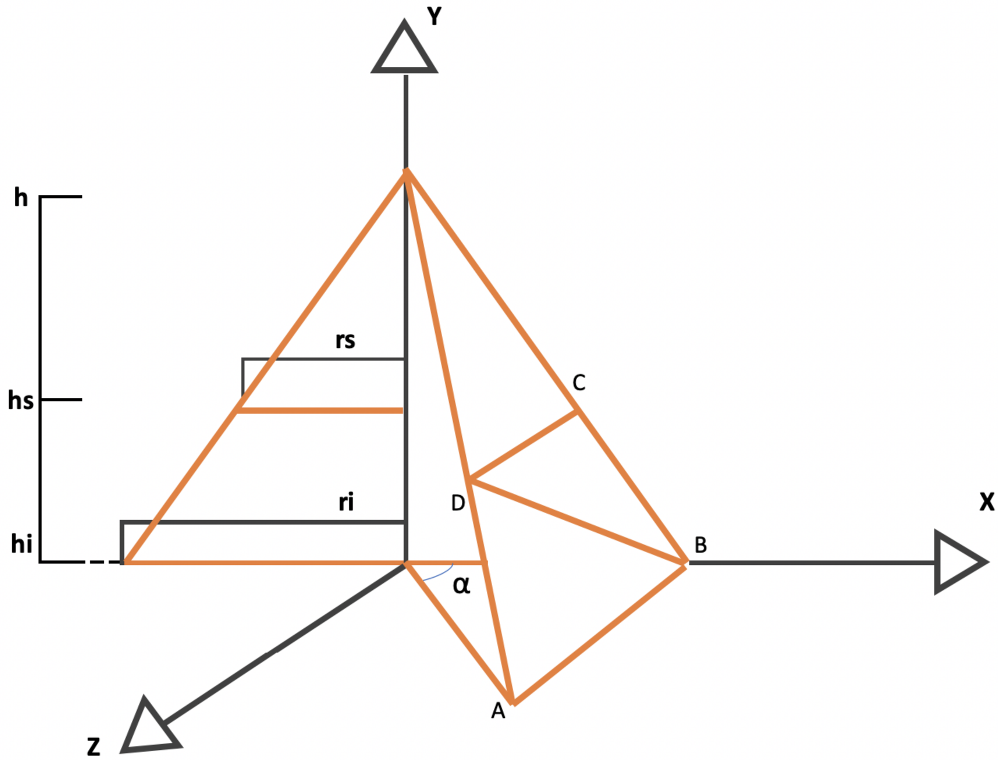

Figura 5: Esquema do cone

Algorithm 5 drawCone

```
function drawCone(radius, slices, stacks)
    d_x = 2π/slices
    h = height/stacks

drawButtom:
    for i ← 0 to slices do
        p_1 ← (radius * sin(alpha), 0, radius * cos(alpha))
        p_2 ← (radius * sin(alpha + dx), 0, radius * cos(alpha + dx))
        t_1 = (p_1, (0, 0, 0), p_2)

drawFaces:
        for i ← 0 to stacks do
            hi = h * i
            hs = h * (float)(i + 1)
            ri = radius − ((radius * hi)/height)
            rs = radius − ((radius * hs)/height)
            for j ← 0 to slices do
                α = d_x * j

                x_1 ← ri * sin(alpha);
                z_1 ← ri * cos(alpha)

                x_2 ← ri * sin(alpha + dx)
                z_2 ← ri * cos(alpha + dx)

                x_3 ← rs * sin(alpha)
                z_3 ← rs * cos(alpha)

                x_4 ← rs * sin(alpha + dx)
                z_4 ← rs * cos(alpha + dx)

                t_1 ← ((x1, hi, z1), (x2, hi, z2), (x3, hs, z3))
                t_2 ← ((x2, hi, z2), (x4, hs, z4), (x3, hs, z3))
```

4 Engine

Uma vez analisada em detalhe toda a algoritmia por trás da geração de vértices de primitivas, o passo seguinte é ler o ficheiro gerado pelo ***generator*** para que seja feita a representação gráfica das primitivas.

Como já tinha sido referido anteriormente, o ***engine*** tem que abrir os ficheiros apontados pelo documento ***XML*** e ler o conjunto de pontos. Para a leitura do ficheiro ***XML*** foi usada um módulo chamado ***tinyxml2*** que contém um conjunto enorme de funções para tratamento destes tipos de documento.

Para que haja uma melhor performance nos carregamentos dos vértices, foram usados ***VBO's*** (*vertex buffer objects*). Sem esta estrutura de dados, utilizando o modo normal de desenho imediato, os vértices ficariam definidos em memória *RAM* e, posteriormente, copiados um a um para a placa gráfica. De modo a aumentar a performance do programa, os ***VBO's*** permitem que os vértices sejam copiados de uma só vez e somente uma vez para a placa gráfica.

4.1 Representação gráfica em 3D das primitivas

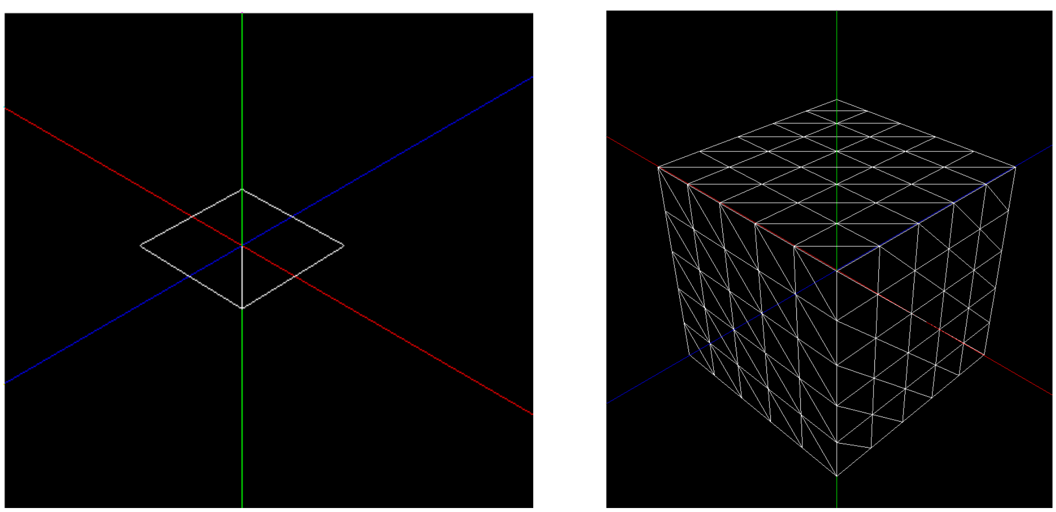

Figura 6: Plano e caixa

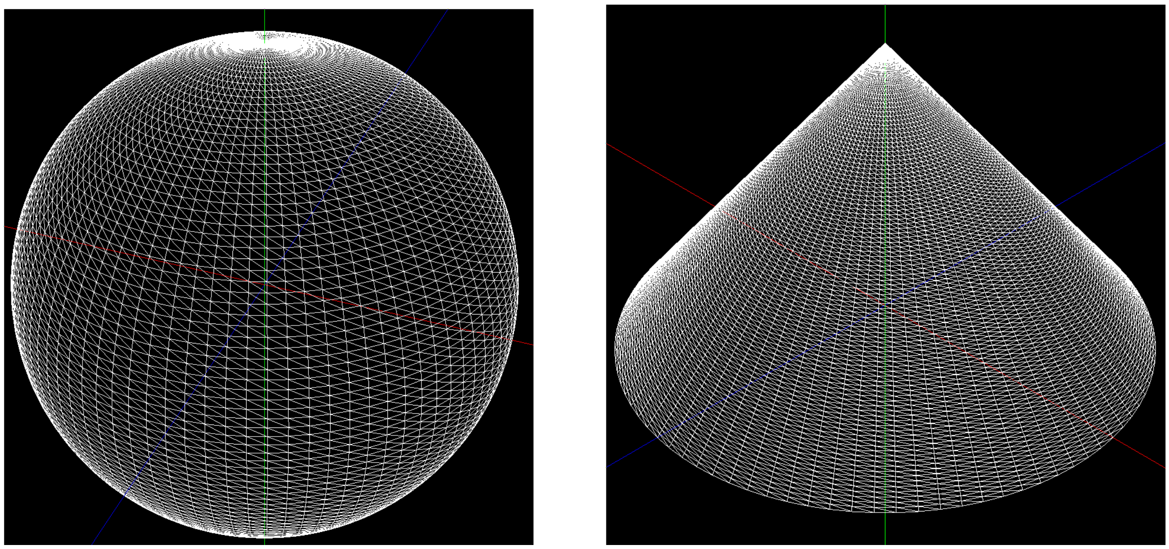

Figura 7: Esfera e cone

5 Conclusão

Após a conclusão desta primeira fase do trabalho, pode-se concluir que a componente com maior complexidade é o ***generator*** devido a toda a lógica necessária para a representação das primitivas apenas com triângulos. Dentro desta componente, o grupo achou que as primitivas mais complexas de desenvolver foram a esfera e cone por serem figuras menos regulares em comparação com o plano e a caixa.

Todos os requisitos impostos para esta fase do trabalho foram cumpridos, e foram desenvolvidos de forma modular para melhor organização do código.