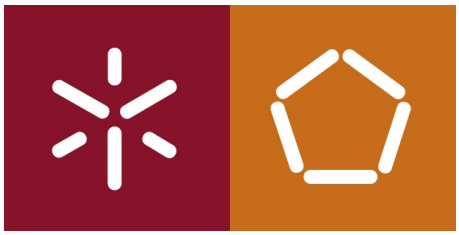

Universidade do Minho - Escola de Engenharia

Relatório do trabalho prático de Computação Gráfica

# Fase 1 – Primitivas Gráficas

*Autores :*
Daniel Maia (A77531)

Diogo Silva(A78034)

Marco Silva(A79607)

Versão 1.0
3 de Abril de 2018

**Resumo**

O objetivo desta fase do trabalho envolve desenvolver duas aplicações. Uma, designada de *Generator*, será responsável por gerar ficheiros com a informação relativa a um modelo, especificamente, os seus vértices. Para tal, receberá um conjunto de parâmetros necessários para a criação do modelo através da linha de comandos, incluindo o tipo de figura a ser gerada e as suas dimensões, bem como o nome e localização do ficheiro onde a informação será guardada. A outra aplicação, batizada de *Engine*, terá a responsabilidade de ler de um ficheiro de configuração, escrito em XML, e, a partir deste, demonstrar os modelos anteriormente gerados.

Para esta fase, foi definido um conjunto de figuras que o *Generator* suportará:

- Plano (*Plane*), representado por um quadrado centrado na origem, constituído por dois triângulos;
- Cubo (*Box*), centrado na origem, cujas faces podem ser divididas em partes menores pelo utilizador no momento de geração;
- Esfera (*Sphere*), centrado na origem, com um determinado raio e número de divisões horizontais e verticais;
- Cone, centrado na origem, com um determinado raio, altura e número de divisões horizontais e verticais.

# Conteúdo

# 1 Introdução

O objetivo deste projeto é desenvolver uma cena gráfica baseada num motor 3D. Para tal são necessárias duas componentes: o *generator* e o *engine*.

Primeiramente, é executado o *generator*. Este é responsável por produzir todos os pontos necessários para a construção de uma determinada primitiva gráfica. Após o cálculo dos pontos, estes são guardados num ficheiro com a extensão **3d**.

De seguida, a execução é encadeada com o componente *engine*. Este analisa um ficheiro XML e com base na informação presente no mesmo, recorre aos ficheiros **3d** para carregar os pontos, que são consequentemente usados para desenhar as primitivas gráficas.

Por último, para a realização do projeto foi utilizado o OpenGL Utility Toolkit (GLUT), que possibilita a escrita de programas em OpenGL independentemente do sistema de janelas. Adicionalmente, foi usada a linguagem C++ para a escrita do código e o *pugixml* como biblioteca de processamente de XML.

# 2 Descrição do Trabalho e Análise de Resultados

## 2.1 Generator

Antes de abordar qualquer pormenor mais técnico relacionado com a geração dos pontos para as diversas figuras a desenhar, é importante explicitar que foi definida uma norma para a escrita nos ficheiros que irão representar as figuras abaixo descritas. Desta forma, foi estabelecido que em cada uma das linhas estarão presentes as 3 coordenadas em formato cartesiano com a ordem x, y e z, separadas por espaços, ficando assim em cada linha toda a informação necessária para a representação de um ponto.

### 2.1.1 Plano

No contexto do trabalho, o plano trata-se de um quadrado com um comprimento N passado por argumento na linha de comandos. Como tal, é necessário apenas determinar quatro pontos com os quais se poderá definir os dois triângulos que o constituirão. O plano está centrado em $xz$, logo, os pontos seguem o formato $(x,\ 0,\ z)$, x e z sendo $\pm\ \frac{N}{2}$. Para manter a semântica do *OpenGL*, dois destes pontos terão um duplicado no ficheiro que guardará a figura.

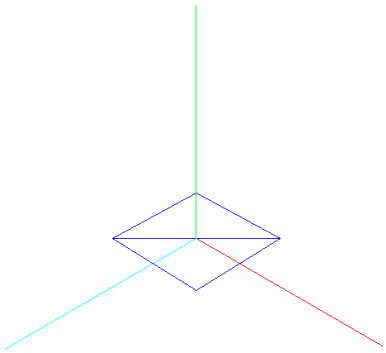

Figura 1: Representação gráfica de um plano.

### 2.1.2 Box

A primitiva gráfica *box* tem como características o comprimento (z), a largura (x), a altura (y) e opcionalmente o número de divisões de cada face.

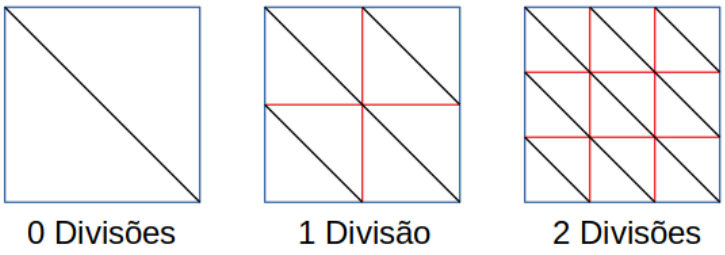


Figura 2: Aplicação do conceito de divisões.

Dado que as dimensões da *box*, isto é, o **x**, **y** e **z** caracterizam o comprimento de cada um dos lados foi necessário dividir cada uma das medidas por dois. Desta forma, o centro da *box* fica centrado na origem do referencial.

De seguida, partiu-se para a conceção de como seriam implementadas as divisões. Primeiramente, tentou-se perceber qual a relação entre o incremento, ou seja, a base do triângulo, o número de divisões e o comprimento do lado. No final surgiu a seguinte fórmula:

$$incremento_x = x/num\_divisions$$

Deste modo, o próximo passo seria arranjar um algoritmo para desenhar os triângulos numa determinada face, respeitando o número de divisões. Assim sendo, decidiu-se que o ponto (*master point* - MP) pelo qual nos guiaríamos seria, em toda e qualquer situação, o canto inferior esquerdo. Efetivamente, começaríamos a construir a face no quanto inferior esquerdo e os próprios triângulos teriam como ponto de referência o seu próprio canto inferior esquerdo.

A título exemplificativo, realizamos um protótipo do que seria realizar a face frontal com 5 divisões. [3]

De facto, seguindo a regra do MP começa-se a desenhar sempre do canto inferior esquerdo. De seguida, desenha-se a linha e no final desta, sobe-se para o início da linha seguinte e repete-se o processo até toda a face estar concluída. Para tal, é preciso conhecer os valores que se encontram na figura 3.

$$MP(-(x/2), -(y/2), z/2)$$

$$inc_x = x/divisions$$

$$inc_y = y/divisions$$

$$A'(MP_x, MP_y, MP_z)$$

$$B'(MP_x + inc_x, MP_y, MP_z)$$

$$C'(MP_x, MP_y + inc_y, MP_z)$$

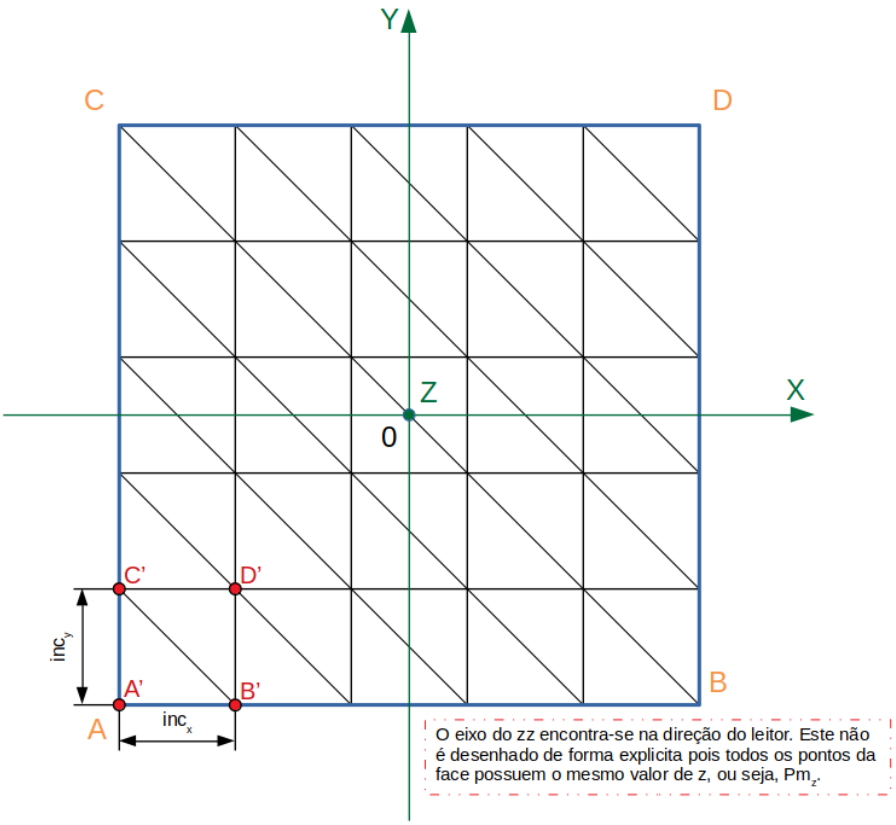


Figura 3: Face frontal com 5 divisões.

$$D'(MP_x + inc_x, MP_y + inc_y, MP_z)$$

De forma, a desenhar os primeiros dois triângulos deve-se inserir os pontos, na seguinte ordem, A', B', C' para o primeiro triângulo e C', B' e D' para o triângulo invertido. Após o desenho do primeiro quadrado (triângulo normal + triângulo invertido) é necessário atualizar o MP apenas numa das variáveis. Esta variável varia conforme a face que esteja a ser desenhada, contudo no exemplo em questão é o **x**.

$$MP_x = MP_x + inc_x$$

Contudo, chegando à última coluna é necessário subir para a linha seguinte. Neste caso, não basta atualizar uma das variáveis mas sim duas. No exemplo acima, são o **x** e o **y** que sofrem mudanças. O $MP_x$ volta à coordenada original e o $MP_y$ é simplesmente incrementado.

$$MP_x = -(x/2)$$

$$MP_y = MP_y + inc_y$$

Finalmente, basta correr o algoritmo para cada uma das faces. Como resultado teremos uma *box* com as dimensões x, y, z e com n divisões.

### 2.1.3 Cone

Para desenhar o cone de um modo mais detalhado, este será separado num número de cortes horizontais (*stacks*) e verticais (*slices*), permitindo-nos observar a sua face curva como um conjunto de retângulos, que por sua vez, serão divididos num par de triângulos, e observar a sua base como um conjunto de triângulos que partilham um vértice posicionado no centro da mesma. Deste modo, será possível desenhar o cone de modo a que tenha a aparência de ser realmente curvo, apesar de ser, na verdade, um conjunto de triângulos. Permitirá também colorir o cone de modo a dar uma ilusão de existir uma fonte de luz ao espetador.

Para facilitar o futuro uso de funções de transformação geométrica, decidiu-se desenhar o cone coincidindo o seu centro e alinhando o seu vértice com o eixo *yy*. Deste modo, sabe-se que o vértice estará posicionado no ponto $(0,\ \frac{h}{2},\ 0)$ e os pontos da base estarão posicionados em $(x,\ -\frac{h}{2},\ z)$ a uma distância $r$ do eixo *yy*, sendo $h$ a altura do cone, definida pelo utilizador. Portanto, é necessário assegurar que $r$ seja constante. No entanto, fazê-lo através de coordenadas cartesianas prova ser extremamente complicado, portanto, utilizou-se coordenadas polares, recorrendo aos fundamentos da matemática. Sabendo que $y$ se mantém constante, pode-se considerar momentaneamente que se depara com um plano alinhado a *xz*. No sistema de coordenadas cartesiano, $r = \sqrt{(x2 - x1)^2 + (z2 - z1)^2}$. A partir daqui, é possível definir três pontos não colineares, (x1,z1), (x2,z1) e (x2,z2), com os quais se pode definir um ângulo $\alpha$. Recorrendo ao processo inverso, para um dado ângulo $\alpha$, é possível determinar $x$ e $z$ através do Teorema Fundamental da Trigonometria: $\sqrt{sin^2(\alpha) + cos^2(\alpha)} = 1$. Aplicando o Teorema a $r$, $\sqrt{(r * sin(\alpha))^2 + (r * cos(\alpha))^2} = r$. Por convenção, define-se $x = r * sin(\alpha)$ e $z = r * cos(\alpha)$. No contexto do OpenGL, determina-se $\alpha$ através da fórmula $\alpha = i * \frac{2\pi}{slices}, 0 \leq i < slices$.

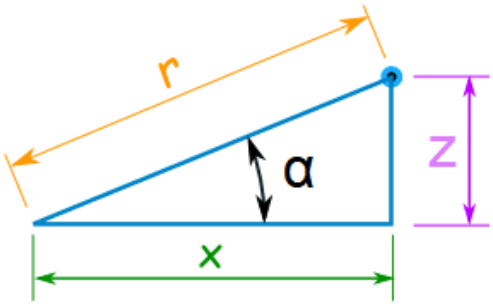


Figura 4: .

Expandir este conceito ao espaço requer considerar um segundo ângulo $\beta$. No entanto, no caso do cone prescinde-se deste graças ao facto de que se sabe que todos os pontos se encontrarão num ponto $(x, -\frac{h}{2} + j * \frac{h}{stacks}, z)$, distando $r - j * \frac{r}{stacks}$ do eixo *yy*, $0 \leq j \leq stacks$.

Deste modo, tem-se um método de determinar todos os pontos que serão introduzidos no ficheiro que guardará a figura. Para cada valor de $i$ e $j$, calcular-se-á um ponto, de coordenadas $((r - j\frac{r}{stacks}) * sin(i * \frac{2\pi}{slices}), -\frac{h}{2} + j * \frac{h}{stacks}, (r - j\frac{r}{stacks}) * cos(i * \frac{2\pi}{slices})), 0 \leq i < slices, 0 \leq j \leq stacks$. É necessário então aplicar o método num formato legível para o OpenGL. Isto obriga, no entanto, que cada ponto seja repetido no ficheiro seis vezes, resultando num ficheiro de tamanho maior, mas lido de um modo mais rápido pelo OpenGL.

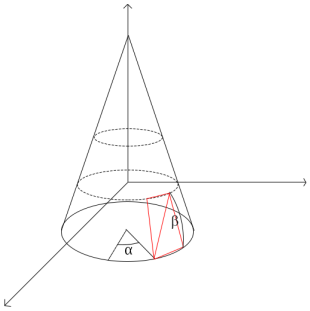


Figura 5: Representação de um cone utilizando coordenadas polares.

### 2.1.4 Esfera

Para a construção da esfera, inicialmente foram estudadas as várias possibilidades para a representação das coordenadas. O mais comum seria a utilização de coordenadas cartesianas mas, uma vez que para o desenho da esfera pretendemos representar superfícies planas foram utilizadas coordenadas polares, sendo mais adequadas ao problema.

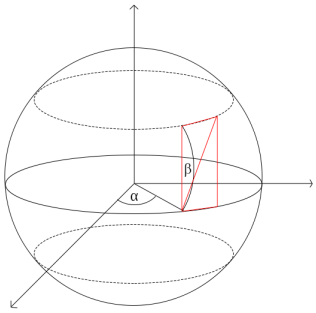


Figura 6: Representação da esfera utilizando coordenadas polares e demonstração do passo iterativo de desenho de triângulos.

Uma vez que, juntamente com a especificação do raio da esfera a desenhar, é também fornecido o número de *slices* (divisões verticais) e *stacks* (divisões horizontais), conclui-se que as variações dos ângulos $\alpha$ e $\beta$ que se podem ver representados na figura acima, irão estar diretamente relacionadas com o número de *slices* e *stacks* respetivamente.

Primeiramente, são calculados os incrementos quer em $\alpha$ quer em $\beta$ com base no valor de *slices* e *stacks* que são necessários. No caso das *slices* (correspondente ao $\alpha$ na figura),

$increment1 = (2\Pi)/slices$ uma vez que terão de ser desenhadas *slices* em redor de toda a esfera enquanto que para as *stacks*, $increment2 = \Pi/stacks$. Para as *stacks*, apenas é necessário percorrer $\Pi$ radianos visto que o desenho das *slices* já percorrem $2\Pi$ radianos cobrindo assim por completo a superfície da esfera.

Tendo este raciocínio consolidado, apenas é necessário explicitar os passos a tomar em cada uma das iterações. Tendo em atenção a imagem acima apresentada, podemos ver o desenho a vermelho de dois triângulos que juntos representam um retângulo. As coordenadas destes pontos são determinadas com base nos ângulos atuais quer de $\alpha$ e $\beta$ adicionando os incrementos respetivos calculados anteriormente.

Este raciocínio encontra-se dentro de dois ciclos, cada um com o número de *stacks* e *slices* indicados.

## 2.2 Engine

### 2.2.1 Formato XML

O formato XML surge neste projeto como "guião"da cena a ser desenhada uma vez que tem nele descrito os ficheiros de pontos que devem ser desenhados e numa fase mais avançada, instruções como *translate* ou *rotate*.

Nesta primeira fase, o ficheiro XML é bastante simples, sendo apenas constituído por uma *tag* scene que por sua vez alberga a *tag* model com o atributo *file* que representa o nome do ficheiro de pontos a desenhar.

Tendo assim bem definida a estruturação do ficheiro, apenas é necessário aceder aos conteúdos deste ficheiro e extrair a informação necessária, neste caso são apenas necessários os nomes dos ficheiros de pontos.

### 2.2.2 Leitura e Desenho

Uma vez extraída a informação necessária do ficheiro XML,utilizando a biblioteca pugixml [1], pode-se proceder ao tratamento dos ficheiros em causa. Primeiramente, estes são abertos apenas com permissões de escrita e é feita a leitura linha a linha dos mesmos.

Assim sendo, dando ênfase à estrutura definida anteriormente dos ficheiros **3d**, foi utilizada uma função *split* que dada uma string, um delimitador e um vetor, procede à divisão da mesma colocando o resultado no vetor indicado. Tendo a informação já devidamente tratada, basta proceder à conversão do formato *string* para *float* recorrendo à função *atof*. Assim, percorrendo todas as linhas do ficheiro e fornecendo os dados extraídos ao OpenGL, as figuras são construídas na sua totalidade.

# 3 Conclusões e Sugestões

Após experimentação, concluiu-se que a geração de ficheiros com alguma redundância, mas que podem ser lidos sequencialmente pelo *engine*, habilitam uma melhor performance num todo, quando comparados com ficheiros gerados com o intuito de serem mais eficientes em termos de armazenamento, o que consequentemente requereria uma maior computação no componente *engine* aquando da construção das primitivas gráficas, além de provocar uma pior legibilidade do código.

O projeto utiliza o *generator* para gerar os pontos que formam as primitivas gráficas e, consequentemente, executa o *engine* que transformará os pontos em objetos concretos, numa cena. Efetivamente, esta dualidade permite que o componente *engine* seja completamente independente da primitiva que se encontre a desenhar. Deste modo, este pode ser usado para desenhar qualquer figura, quer esta seja mais ou menos complexa, incluindo primitivas não suportadas pelo *generator*.

A modularidade com que o motor 3D foi concebido permitirá que, no futuro, facilmente se acrescente primitivas gráficas e funcionalidades mais exigentes a nível computacional.

# Referências

[1] Pugixml. Pugixml 1.8 manual. `https://pugixml.org/docs/manual.html`, 2017. [Online; acedido a 12-Março-2018].