COMPUTAÇÃO GRÁFICA

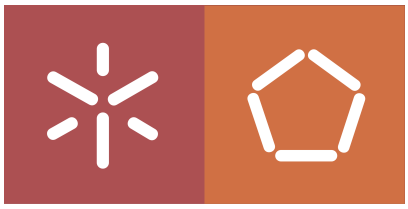
Fase 1 - Primitivas Gráficas

Diogo Esteves
a104004

Rodrigo Fernandes
a104175

Diogo Barros
a100751

Fevereiro 2025



Universidade do Minho
Escola de Engenharia

Conteúdo

1 Introdução

Este relatório descreve o processo de desenvolvimento e implementação de um motor 3D baseado em *scene graph*, realizado como parte do projeto prático da unidade curricular de **Computação Gráfica** da Universidade do Minho.

O objetivo central deste trabalho foi criar um sistema capaz de gerar primitivas gráficas, como planos, cubos, esferas, cones, etc, e renderizá-las numa cena 3D a partir de ficheiros de configuração *XML* e do OpenGL. O projeto foi dividido em fases, e este relatório é relativo à primeira fase, que engloba a criação das primitivas e a implementação inicial do motor gráfico.

Para esta fase, é proposta a criação de dois executáveis:

- **'generator':** gera ficheiros com a informação dos modelos (nesta fase contém apenas as coordenadas dos vértices do modelo).
- **'engine':** recebe a informação criada pelo *generator* e um ficheiro de configuração *XML* e desenha os modelos no ecrã.

2 *Generator*

O generator é a aplicação que gera os ficheiros '.3d' com os vértices de cada primitiva gráfica. Este tem em conta os argumentos passados pelo utilizador e, a partir deles, calcula precisamente os pontos das primitivas e escreve-os diretamente no ficheiro de output, com atenção à sua orientação, ou seja, a ordem pela qual os vértices são impressos está de acordo com a regra da mão direita.

Para que os vértices gerados cumpram a regra da mão direita, a ordem de desenho dos mesmos deve ter o sentido anti-horário. Assim, garantimos que todos os triângulos das primitivas estão voltados no corretamente segundo o eixo 'y', e por isso visiveis para a câmara.

A imagem abaixo exemplifica a ordem pela qual cada triângulo deve ser desenhado

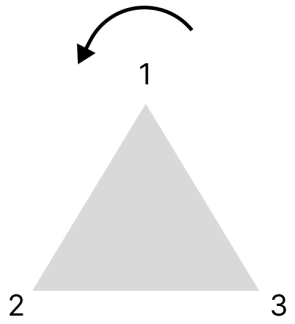


Figura 1: Exemplo de triângulo com face voltada para a câmara, segundo a regra da mão direita

Em cada ficheiro '.3d' gerado, é assinalada na primeira linha o nome da primitiva que foi gerada, para que a engine processe corretamente os dados facultados. Cada linha contém as coordenadas dos pontos no formato "X Y Z"e cada conjunto de 3 pontos (linhas) consecutivas corresponde a um triangulo que deve ser desenhado posteriormente pela *engine*.

2.1 Primitivas

2.1.1 Plano

O plano é um quadrado paralelo ao plano XZ, centrado na origem e subdividido na direção X e Z. Para que seja possível construir o plano com recurso a triângulos, começamos por cálcular dois parâmetros auxiliares, um *'step'* e um *'halfLength'*:

$$step = length/divisions \tag{1}$$

$$halfLength = length/2.0f \tag{2}$$

Após o cálculo dos parametros acima, podemos começar a iterar sobre os eixos relativos ao plano: no eixo X (variável i) e no eixo Z (variável j). A cada iteração, devemos cálcular as coordenadas dos pontos relativos ao triângulo que está a ser construido, segundo as seguintes fórmulas:

$$x1 = -halfLength + i * step \tag{3}$$

$$z1 = -halfLength + j * step \tag{4}$$

$$x2 = x1 + step \tag{5}$$

$$z2 = z1 + step \tag{6}$$

Como são necessários 3 pontos para a construção de um triangulo em openGL, vamos reciclar um dos pontos cálculados na iteração anterior. Assim, no ficheiro de output, um triângulo corresponde a uma sequência do tipo:

$x1$ 0.0 $z1$

$x1$ 0.0 $z2$

$x2$ 0.0 $z2$

$x1$ 0.0 $z1$

$x2$ 0.0 $z2$

$x2$ 0.0 $z1$

A sequência acima garante que a cada iteração, é construido um triângulo orientado em cada um dos eixos do plano e, assim, o output dos pontos gerados será um plano como:

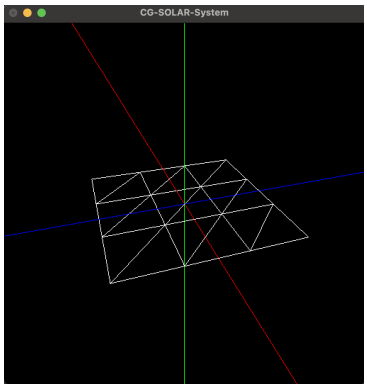


Figura 2: Plano

2.1.2 Caixa

A caixa (*box*) é um cubo centrado na origem do referêncial, e que varia de forma consoante parâmetros como a sua dimensão e o número de divisões por aresta.

Para começarmos a construção da *box*, com recurso a triângulos, calculamos dois parâmetros auxiliares:

$$halfDim = dimension/2.0f \tag{7}$$

$$step = dimension/divisions \tag{8}$$

Já com estes parâmetros auxiliares cálculados, podemos dar inicio à construção da *box*. Esta possui 6 faces, numeradas de 0 a 5, e cada face é subdividida em *divisions* × *divisions* quadrados. Para cada quadrado, são gerados dois triangulos, totalizando 6 vértices.

Para cada face, as coordenadas dos vértices são calculadas e escritas no ficheiro. A posição de cada vértice é definida em função das coordenadas (x1, y1), (x1, y2), (x2, y1), (x2, y2), onde:

- **x1 = -halfDim + j * step:** posição inicial do quadrado na direção x.
- **x2 = x1 + step:** posição final do quadrado na direção x.
- **y1 = -halfDim + i * step:** posição inicial do quadrado na direção y.
- **y2 = y1 + step:** posição final do quadrado na direção y.

Dependendo da face que está a ser processada, as coordenadas são ajustadas para corresponder à posição correta no espaço tridimensional.

Cada face é gerada utilizando um *switch(face)*, e os vértices são organizados para manter a orientação correta:

- **Face 0:** Frente (z = +halfDim)
- **Face 1:** Trás (z = -halfDim)
- **Face 2:** Esquerda (x = -halfDim)
- **Face 3:** Direita (x = +halfDim)
- **Face 4:** Topo (y = +halfDim)
- **Face 5:** Base (y = -halfDim)

Assim, após ler o ficheiro gerado, este deverá ser o output, por exemplo:

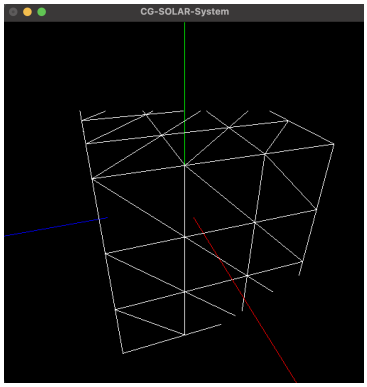


Figura 3: Box

2.1.3 Cone

O desenho do cone requer parâmetros como:

- **radius:** Define o raio da base do cone.
- **height:** Define a altura total do cone.
- **slices:** Especifica quantas subdivisões circulares (fatias) compõem a base do cone.
- **stacks:** Especifica quantas camadas verticais o cone terá.

Além dos parâmetros listados acima, é necessário que a base do cone esteja no plano XZ. Para fazermos o desenho do cone com recurso a triangulos, começamos por calcular parâmetros que nos vão auxiliar na criação desta primitiva:

$$sliceStep = (2.0f * \pi)/slices \tag{9}$$

$$stackStep = height/stacks \tag{10}$$

Cálculados estes parâmetros auxilares, começamos por gerar uma base para o cone, contida no plano XZ. A base do cone é gerada conectando o centro (0,0,0) aos vértices ao longo da circunferência:

- **x = radius * sin(theta)** e **z = radius * cos(theta)** determinam as coordenadas circulares.
- Cada fatia gera um triangulo formado pelo centro e dois pontos consecutivos da circunferência.

De seguida, passamos pela criação das laterais do cone. Este é subdividido em *stacks* camadas, gerado-se assim vértices intermédios:

- **r1 = radius * (1.0f - j / stacks)**, **r2 = radius * (1.0f - (j + 1) / stacks)** reduzem o raio conforme a altura aumenta.
- Para cada fatia, dois triangulos são gerados conectando os vértices das camadas adjacentes.

Assim, um cone gerado pelo e processado pela engine, dependendo do ficheiro de configuração, porderá ser representado da seguinte forma:

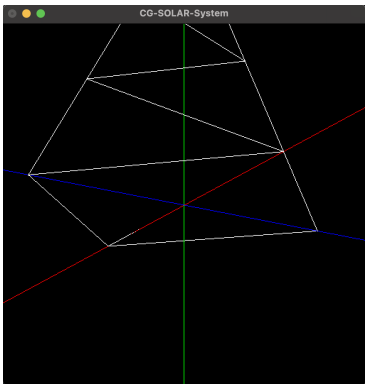


Figura 4: Cone

2.1.4 Esfera

Para conseguirmos fazer a criação de uma esfera utilizando triangulos, são necessários parâmetros tais como:

- **radius:** Define o raio da esfera.
- **slices:** Especifica o número de divisões ao longo da longitude.
- **stacks:** Especifica o número de divisões ao longo da latitude.

Além destes parâmetros, a esfera deve estar centrada na origem do referêncial. Assim, podemos começar a gerar as coordenadas da esfera,a partir da variação dos angulos theta (latitude) e phi (longitude):

- **theta**: varia de 0 a π, controlando a posição vertical.
- **phi:** varia de 0 a 2π, controlando a posição horizontal.

De seguida, passamos a cálcular as coordenadas dos vértices para cada divisão da esfera, da seguinte forma:

$$theta1 = i * \pi / stacks \tag{11}$$
$$theta2 = (i+1) * \pi / stacks \tag{12}$$
$$phi1 = j * 2\pi / stacks \tag{13}$$
$$phi1 = j * 2\pi / stacks \tag{14}$$

$$x = radius * sin(theta) * sin(phi) \tag{15}$$
$$y = radius * cos(theta) \tag{16}$$
$$z = radius * sin(theta) * cos(phi) \tag{17}$$

Por fim, é necessário escrever no ficheiro os triangulos relativos à esfera. Cada célula da malha da esfera é composta por dois triangulos, escritos no ficheiro na seguinte ordem:

$$x1\ y1\ z1$$
$$x4\ y4\ z4$$
$$x2\ y2\ z2$$

$$x1\ y1\ z1$$
$$x3\ y3\ z3$$
$$x4\ y4\ z4$$

Assim, um exemplo de uma esfera gerada pelo *generator* pode ser:

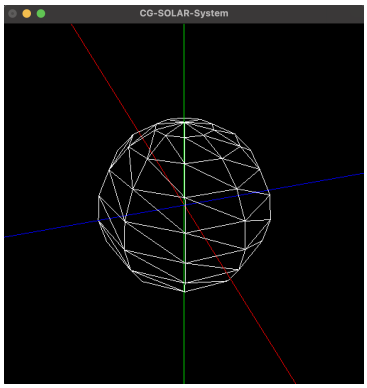


Figura 5: Esfera

2.1.5 Cilindro

O cilindro é um prisma, com duas bases circulares conectadas por uma superficie curva e centrado na origem do referêncial. As bases circulares são paralelas ao plano XZ e estão divididas em *´slices´*, para que seja possível fazer o seu desenho utilizando triângulos.

Assim, para o desenho das bases, é necessário dividir o ângulo da circunferência consoante o número de slices, segundo a seguinte fórmula:

$$\Delta = 2\pi / slices \tag{18}$$

Dado agora um Δ, é possível fazer o cálculo da posição dos pontos relativos à base da circunferência, utilizando sempre a origem do referêncial como ponto auxiliar, utilizando as seguintes fórmulas para o cálculo das coordenadas em x, y e z:

$$x1 = radius * sin(i * \Delta) \tag{19}$$

$$y1 = height/2 \tag{20}$$

$$z1 = radius * cos(i * \Delta) \tag{21}$$

Já calculado o primeiro ponto, como o openGL necessica de 3 pontos para formar o triangulo, é necessário cálcular o ponto seguinte e adicionar a origem:

$$x2 = radius * sin(i + 1 * \Delta) \tag{22}$$

$$y2 = (height/2) \tag{23}$$

$$z2 = radius * cos(i + 1 * \Delta) \tag{24}$$

$$ox = 0 \tag{25}$$

$$oy = +/- (height/2) \tag{26}$$

$$oz = 0 \tag{27}$$

Assim, no ficheiro, o output das 3 linhas correspondentes a um triangulo da base superior aparecem como:

$$x1\ y1\ z1$$

$$ox\ oy\ oz$$

$$x2\ y2\ z1$$

É necessário respeitar esta ordem, para que os triângulos cumpram a regra da mão direita e fiquem visiveis. Quanto à base do cilindro, o processo é idêntico, com a coordenada em y negativa. Em relação aos triângulos laterais, basta unir os triangulos das bases segundo a ordem correspondente, para que estes também fiquem visiveis.

Assim, o output do cilindro após a execução da engine será, por exemplo:

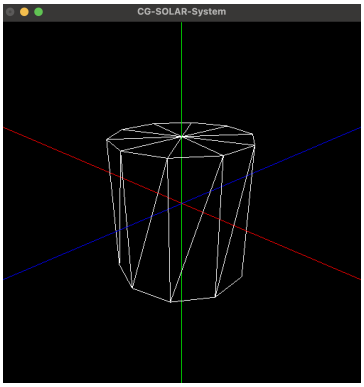


Figura 6: Cilindro

3 Engine

O principal objetivo da aplicação de *engine* é converter a informação gerada pelo *generator* e, com base em ficheiros de configuração XML, tornar essa informação visível ao utilizador. Para isso, utilizamos duas ferramentas para auxilio no desenvolvimento do programa:

- **Biblioteca GLut:** Biblioteca responsável pelo desenho dos modelos gerados anteriormente e criação de cenas 3D.
- **Biblioteca TinyXML:** Biblioteca auxiliar utilizada no processo de *parsing* dos ficheiros de configuração XML.

3.1 Organização da Engine

Na programação com a biblioteca Glut, é necessário antes de entrarmos no ciclo principal, fazermos o *callback* das funções que serão utilizadas para as funções básicas na renderização da cena: função de desenho, *resize*, interação, etc. Como estas funções têm assinaturas *standard*, que devem ser respeitadas, foram definidas para esta fase dois objetos 'globais': **'model'** e **'Config'**.

Com a definição dos objetos globais no inicio da execução da engine garantimos que:

- **Existe persistência dos dados durante a renderização da cena:** O loop de renderização do OpenGL precisa de aceder aos modelos frame a frame para desenhá-los no ecrã. Como a função de desenho (draw()) é chamada repetidamente dentro do loop principal, os modelos precisam estar dispiníveis num scope global.
- **Callbacks conseguem aceder a parametros:** Assim, mesmo respeitando a assinatura *standard* das funções do Glut, conseguimos aceder dentro do scope das mesmas a parametros necessários, como os presentes dentro dos ficheiros de configuração XML.

3.1.1 Config

É no objeto config que estão guardadas as infirmações presentes no ficheiro XML de configuração. Este está subdividido em diferentes parâmetros que são relevantes para armazenar a informação do ficheiro de configuração: **Window** (com as informações acerca do tamanho da janela); **Camera** (com as informações acerca ca camara que são relevantes para esta fase, como as coordenadas da sua posição, ponto de *lookAt*, vetor *up* e os parametros de projeção) e por fim, um vetor com estruturas de dados **ModelFile** (com a informação acerca do nome do ficheiro .3d a ser ligo e a flag correspondente à primitiva nele contida).

Como este é um objeto global e contém algumas das informações necessárias para renderizar os *frames*, será lido pelas funções de redenrização relativos ao objeto **Model**, que será abordado no próximo ponto.

3.1.2 Model

Este é o objeto responsável por armazenar em memória, após o *parse* dos ficheiros .3d, as informações relativas às coordenadas dos vértices dos triangulos que compõem as primitivas gráficas. Nele, estão contidos 5 vetores responsáveis por armazenar as diferentes primitivas suportadas pelo *generator*.

Além da função de armazenamento dos dados gerados, estão associadas ao objeto as funções de desenho dos frames (com base nos dados armazenados em memória), e, em fases futuras, de interação com os *input* do utilizador.

3.2 Parsing

Assim que é iniciada a execução do programa, o primeiro passo é fazer o parsing dos ficheiros de configuração XML, e de seguida dos ficheiros .3d gerados anteriormente.

Para o parsing dos ficheiros XML, utilizamos a biblioteca auxiliar tinyXML. Assim conseguimos preencher o objeto Config referido anteriormente.

Para o parsing dos ficheiros .3d, começamos por ler o tag presente na primeira linha, e associamos a mesma ao objeto de Config. Assim, quando estivermos a desenhar os frames, sabemos a que vetor do objeto Model ler a informação. Depois preenchemos os vetores do objeto Model com a informação das coordenadas dos pontos dos triangulos gerados pelo *generator*.

3.3 Desenho dos frames

Quando passamos para o desenho das frames, este é feito de forma imediata, ou seja, não utiliza recurso a VBOs nesta fase do projeto. A otimização através de VBOs será feita em fazes futura, sendo que esta fase tem como enfase o desenho e cálculo das primitivas gráficas.

Assim, a função de desenho apenas lê do objeto Config as informações relativas à camara e do objeto Model as informações relativas às coordenadas dos vértices dos triangulos e apresenta no ecrã esse mesmo output.

4 Conclusão

O principal objetivo desta fase do trabalho centra-se na criação das primitivas gráficas que servirão de base para o projeto nas fases furutas. Assim, apresentamos 5 primitivas para esta fase, com possibilidade de introdução de mais primitivas futuramente. Além da implementação de possivelmente mais primitivas, temos planeadas otimizações ao nível de performance, nomeadamente, a implementação das VBOs.

Além do referido acima, na próxima fase seão implementadas as transformações geonétricas, pelo que novas funções serão adicionadas à engine.

5 Referências

Referências utilizadas para a construção das primitivas:

- $https://www.songho.ca/opengl/gl_cylinder.html$
- $https://www.songho.ca/opengl/gl_sphere.html$
- $https://www.songho.ca/opengl/gl_cone.html$
- Slides das aulas TPs