Computação Gráfica (3º ano de MIEI)

Primeira Fase

Relatório de Desenvolvimento

André Gonçalves	João Queirós	Luís Alves	Rafaela Rodrigues
(a80368)	(a82422)	(a80165)	(a80516)

9 de Março de 2019

Resumo

Este relatório inicia-se com uma breve contextualização, seguindo-se a descrição dos problemas propostos e o que deve ser desenvolvido para os solucionar. Segue-se uma resolução formal e matemática do problema de geração dos vértices de primitivas, sendo essa resolução implementada no capítulo seguinte. Nesse capítulo são expostas interpretações muito similares ao código desenvolvido, mas simplificadas e com nomenclatura distinta para facilitar a sua interpretação sem conhecimento da linguagem de programação utilizada (C++). Por fim, são apresentadas conclusões sobre o trabalho desenvolvido, os seus pontos fortes e menos fortes.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização

O presente relatório foi elaborado no âmbito da Unidade Curricular de Computação Gráfica, que se insere no 2º semestre do 3º ano do primeiro ciclo de estudos do Mestrado Integrado em Engenharia Informática. Pretende-se com este relatório formalizar toda a análise e modelação envolvida na construção da solução ao problema proposto. Nesta primeira fase, foi proposta a criação de duas aplicações distintas: uma para gerar vértices de algumas primitivas, e outra para as mostrar no ecrã.

1.2 Descrição do trabalho proposto

Foi proposta para esta fase a elaboração de duas aplicações: um gerador de vértices de primitivas e um motor gráfico.

O gerador de vértices de primitivas deverá ser capaz de gerar as seguintes primitivas, de acordo com os argumentos fornecidos:

Primitiva	Argumentos
Plano	Dimensão
Caixa	Dimensão em X, Y e Z e nº de divisões (opcional)
Esfera	Raio, *slices* e *stacks*
Cone	Raio da base, altura, *slices* e *stacks*

Tabela 1.1: Tabela das Primitivas e Argumentos

Para além disso, a geração de vértices deverá ter como destino um ficheiro .3d, com um formato à escolha.

Relativamente ao motor gráfico, este deverá ser capaz de receber um ficheiro .xml, que referencia diversos modelos no formato .3d, e renderizá-los no ecrã.

1.3 Resumo do trabalho a desenvolver

Para o gerador de vértices de primitivas, deve ser desenvolvida uma função para cada primitiva em que dados os seus argumentos (ver 1.1), gere um conjunto de vértices que serão interpretados como triângulos, isto é, cada sequência de 3 vértices representará um triângulo de acordo com a regra da mão direita do *OpenGL*.

De forma a poderem armazenados os vértices gerados, terá de ser definido um formato para o ficheiro .3d gerado, que facilite a sua interpretação pelo motor gráfico.
Por fim, o motor gráfico deverá interpretar um ficheiro .xml, recorrer ao *OpenGL* para renderizar os triângulos produzidos pelo gerador, e interpretar corretamente os vértices armazenados nos ficheiros .3d. Deverá também armazenar a informação relativamente aos modelos importados em memória.

Capítulo 2

Resolução Formal do Problema

2.1 Plano

Para a criação do plano, é utilizado um parâmetro: o comprimento do lado, sendo que o plano se trata de um quadrado centrado na origem e é formado por dois triângulos.

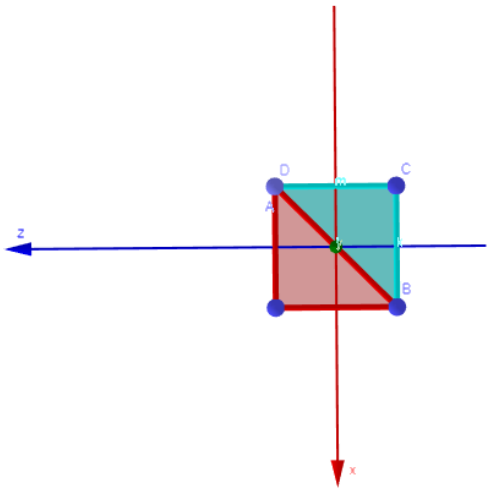

Figura 2.1: Triângulos do plano.

Assim sendo, as coordenadas de cada ponto tomarão os seguintes valores (sendo a dimensão d):

$$A = (d/2, 0, d/2) \tag{2.1}$$

$$B = (d/2, 0, -d/2) \tag{2.2}$$

$$C = (-d/2, 0, -d/2) \tag{2.3}$$

$$D = (-d/2, 0, d/2) \tag{2.4}$$

2.2 Caixa

Para a criação de uma caixa, foram utilizados 4 parâmetros: o número de *divisions*, o comprimento, a altura e a largura da caixa. As *divisions* representam o número de partes em que está divida cada face da caixa. Tendo isto em consideração, cada face será constituida pelo seguinte número de quadriláteros:

$$divisions^2$$

Cada quadrilátero tem, para cada lado, as seguintes dimensões:

$$nx = x/div; \tag{2.5}$$

$$ny = y/div; \tag{2.6}$$

$$nz = z/div; \tag{2.7}$$

Para além disso, o centro da caixa é a origem do referencial.

Para definir um triângulo numa das faces, utiliza-se a sequência de vértices representada na figura 2.2 e cujos pontos são calculados da seguinte forma (para a face da frente, como exemplo):

Ponto A

$$(nx * i - x/2, ny * t - y/2, z/2) \tag{2.8}$$

Ponto I

$$(nx * i - x/2 + nx, ny * t - y/2, z/2) \tag{2.9}$$

Ponto J

$$(nx * i - x/2, ny * t - y/2 + ny, z/2) \tag{2.10}$$

Variando i e t entre 0 e o número de *divisions*, conseguimos formar as diferentes faces.

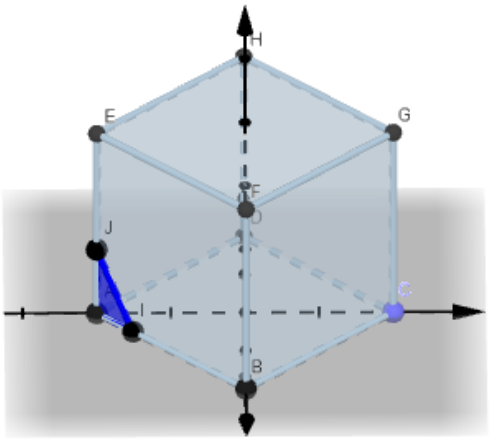

Figura 2.2: Triângulo pertencente à face da frente da caixa

2.3 Cone

Para a criação de um cone, foram utilizados 4 parâmetros: o raio da base, a altura do cone, o número de *slices* e o número de *stacks* que este irá conter. Tendo isto em consideração, dividiu-se a construção do cone em duas partes, que se detalham de seguida.

2.3.1 Base do cone

O número de *slices* fornecidos permite-nos calcular o número de vértices da base (exceto a origem). O cálculo do ângulo α entre 2 vértices consecutivos, relativamente ao eixo do **xx**, é dado pela expressão:

$$\alpha = \frac{2\pi}{slices} \tag{2.11}$$

Com o ângulo entre os vértices e o raio da circunferência, conseguimos definir as coordenadas de cada vértice, recorrendo às coordenadas polares.
O calculo da coordenada em x e em z de um vértice é dado pelas expressões:

$$x = raio \times \cos(\alpha) \tag{2.12}$$

$$z = raio \times \sin(\alpha) \tag{2.13}$$

Dado que a base do cone será definida no plano **xz**, a coordenada em y de todos os vértices é 0.
Para definir um triângulo da base, utilizamos a seguinte sequência de vértices, representados na figura 2.3:
Ponto O

$$(0, 0, 0) \tag{2.14}$$

Ponto A

$$(raio \times \cos(k\alpha), 0, raio \times \sin(k\alpha)) \tag{2.15}$$

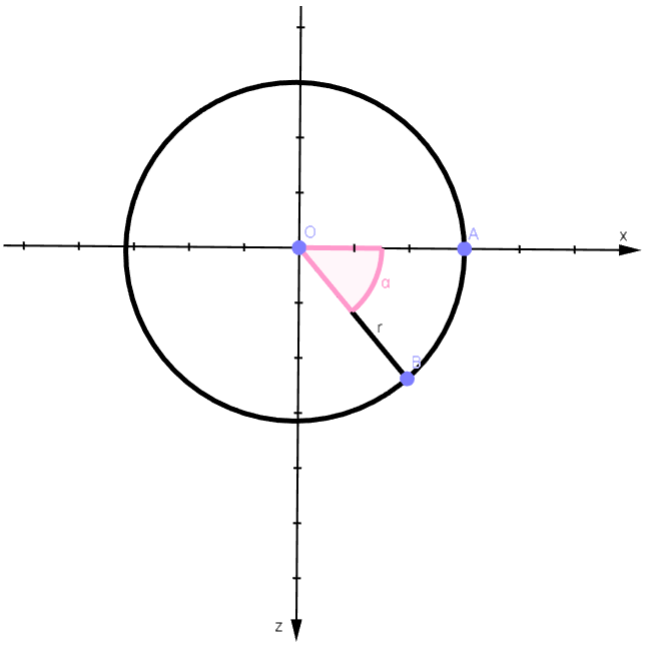

Figura 2.3: Ângulo das coordenadas polares

Ponto B

$$(raio \times \cos((k+1)\alpha), 0, raio \times \sin((k+1)\alpha)) \tag{2.16}$$

Variando o α entre 0 e o número de *slices*, conseguimos formar a base do nosso cone.

2.3.2 Superfície lateral do cone

A nossa abordagem para a criação da superfície do cone foi gerar uma sequência de triângulos (mesmo número que o número de *slices*). Para cada triângulo, as coordenadas dos vértices que se encontram no plano xoz são os mesmos que foram definidos para a base, respeitando assim o número de *slices* requeridas. No entanto, cada triângulo é limitado pelo número de *stacks*. Sendo uma *stack* um "corte"horizontal nos triângulos, estes passaram a ser formados por quadriláteros, como podemos comprovar pela figura 2.4

Conseguimos calcular o tamanho de cada stack (*division*) através da altura do cone (*height*) e do número de *stacks*, que é dado pela expressão:

$$division = \frac{height}{stacks} \tag{2.17}$$

Para os vértices de cada quadrilátero, conseguimos perceber que, para além da variação do valor de x, y e z, também varia o raio. Para o cálculo deste último, recorre-se à seguinte expressão:

$$novoRaio = (1 - \frac{stack}{stacks}) \times raio \tag{2.18}$$

Tendo o valor do novoRaio, para calcular as coordenadas usamos as equações para o cálculo dos vértices das base (com o raio diferente), passando a expressão a ser (para k entre 0 e *slices*):

$$x = novoRaio \times \cos(k \times \alpha) \tag{2.19}$$

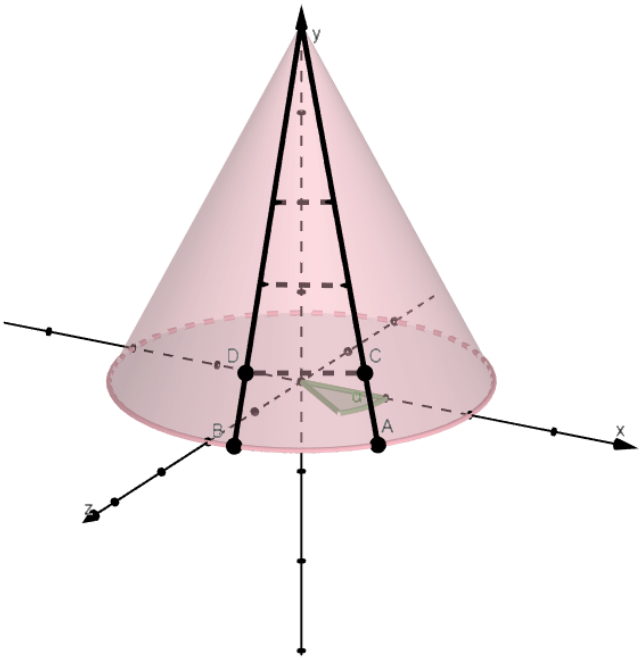

Figura 2.4: *Stacks* de um triângulo pertencente à superfície lateral do cone

$$y = k \times division \tag{2.20}$$

$$z = novoRaio \times \sin(k \times \alpha) \tag{2.21}$$

Tendo em conta estas equações, e sabendo que um quadrilátero é formado por dois triângulos, conseguimos definir os vértices para um dado quatrilátero, a k divisions do plano **xoy** com as seguintes equações (pontos da figura 2.4):

Ponto A

$$(novoRaioA \times \cos(k \times \alpha), k \times divisions, novoRaioA \times \sin(k \times \alpha)) \tag{2.22}$$

Ponto B

$$(novoRaioA \times \cos((k+1) \times \alpha), k \times divisions, novoRaioA \times \sin((k+1) \times \alpha)) \tag{2.23}$$

Ponto C

$$(novoRaioB \times \cos(k \times \alpha), (k+1) \times divisions, novoRaioB \times \sin(k \times \alpha)) \tag{2.24}$$

Ponto D

$$(novoRaioB \times \cos((k+1)\alpha), (k+1) \times divisions, novoRaioB \times \sin((k+1) \times \alpha) \tag{2.25}$$

Efetuando este processo para todas as *slices* do cone, temos um cone definido.

2.4 Esfera

Para obter as 3 coordenadas de um vértice na esfera, são necessários 3 parâmetros: raio da esfera, *slice* e *stack*. Uma *slice* representa um múltiplo de um ângulo em radianos, sendo *slices* o número total de *slices* da esfera. A expressão para se obter esse ângulo encontra-se abaixo:

$$slice = \frac{2\pi}{slices} \tag{2.26}$$

Uma *stack* representa um múltiplo de um ângulo em radianos, sendo *stacks* o número total de *stacks* da esfera. A expressão para se obter esse ângulo encontra-se abaixo:

$$stack = \frac{\pi}{stacks} \tag{2.27}$$

Tendo definidas as *slices* e as *stacks*, é necessário definir dois ângulos (α e β) para utilizarmos um sistema de coordenadas esféricas.

O ângulo α representa a rotação em radianos a partir do eixo positivo do x em relação ao eixo do y, no sentido contrário ao dos ponteiros do relógio. Toma valores entre 0 e 2π.

O ângulo β representa a rotação em radianos a partir do eixo positivo do y em relação ao eixo do z, no sentido dos ponteiros do relógio. Toma valores entre 0 e π.

Os dois ângulos encontram-se representados na Figura 2.5, para um ponto F. Para se obter as coordenadas cartesianas desse ponto F, recorre-se às seguintes equações:

$$x = r \times sin(\beta) \times sin(\alpha) \tag{2.28}$$

$$y = r \times cos(\beta) \tag{2.29}$$

$$z = r \times sin(\beta) \times cos(\alpha) \tag{2.30}$$

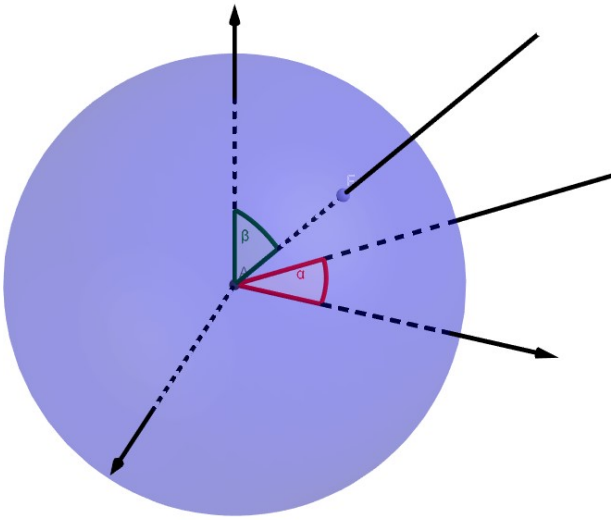

Figura 2.5: Ãngulos das coordenadas esféricas

Tendo estabelecidas as coordenadas de cada ponto na superfície da esfera, é necessário obter retângulos sequencialmente de forma a ser possível desenhar a esfera. Na Figura 2.6 estão representados os 4 pontos usados para cada iteração, tendo por base o ponto A e a sua *slice* (slA) e *stack* (stA). O ângulo base de uma *stack* é representada por β e uma *slice* por α. Assim, as coordenadas dos 4 pontos podem ser obtidas através das seguintes equações:

Para o Ponto A:

$$x = r \times sin(stA \times \beta) \times sin(slA \times \alpha) \tag{2.31}$$

$$y = r \times cos(stA \times \beta) \tag{2.32}$$

$$z = r \times sin(stA \times \beta) \times cos(slA \times \alpha) \tag{2.33}$$

Para o Ponto B:

$$x = r \times sin(stA \times \beta) \times sin(slA \times \alpha) \tag{2.34}$$

$$y = r \times cos((stA + 1) \times \beta) \tag{2.35}$$

$$z = r \times sin((stA + 1) \times \beta) \times cos(slA \times \alpha) \tag{2.36}$$

Para o Ponto C:

$$x = r \times sin((stA + 1) \times \beta) \times sin((slA + 1) \times \alpha) \tag{2.37}$$

$$y = r \times cos((stA + 1) \times \beta) \tag{2.38}$$

$$z = r \times sin((stA + 1) \times \beta) \times cos((slA + 1) \times \alpha) \tag{2.39}$$

Para o Ponto D:

$$x = r \times sin(stA \times \beta) \times sin((slA + 1) \times \alpha) \tag{2.40}$$

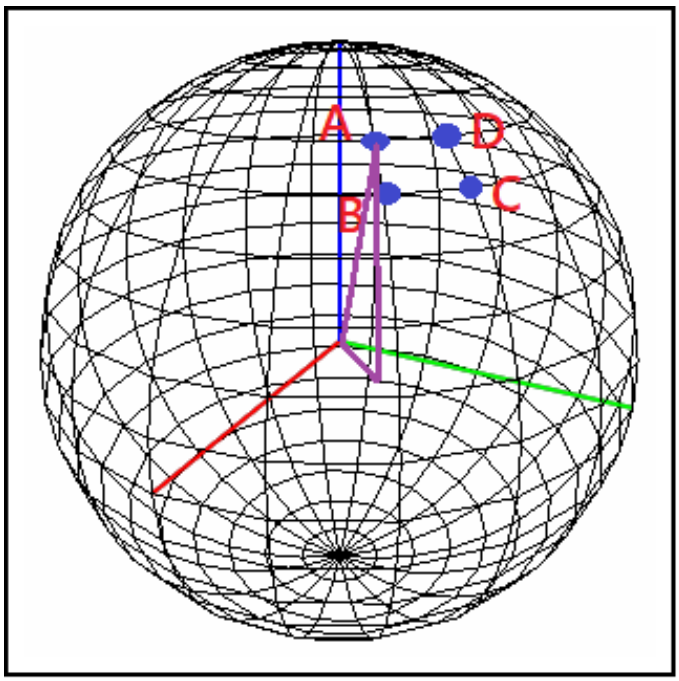

Figura 2.6: Retângulos na superfície esférica (adaptado de [1])

$$y = r \times cos(stA \times \beta) \tag{2.41}$$

$$z = r \times sin(stA \times \beta) \times cos((slA + 1) \times \alpha) \tag{2.42}$$

Estando definidos estes 4 pontos, é possível proceder à variação dos ângulos α e β, desde 0 até 2π e desde 0 até π respetivamente, para se obterem todos os vértices da superfície esférica. Em cada passo, é incrementado cada ângulo de acordo com as expressões em 2.26 e 2.27. Para a implementação em C++, ver secção 3.2.4.

Capítulo 3

Arquitetura do Código

Dada a inexperiência do grupo na linguagem C++, e a sua familiaridade com a linguagem C, este optou por desenvolver orientado a esta última, sendo que a única componente de C++ utilizada foi o `vector`, para armazenar estruturas. Assim, a noção de classe é representada por um ficheiro distinto e cada objeto dessa classe é um apontador para uma `struct classe`.

3.1 Motor

O Motor é composto por diferentes componentes, tendo eles por base duas estruturas: um Modelo e um Vértice. Estes componentes são a leitura do ficheiro de configuração e a renderização dos modelos.

3.1.1 Modelo

Um Modelo será um objeto que conterá um dado número de vértices, de acordo com um ficheiro nome.3d, cujo nome será também o nome atribuído ao Modelo. Por isso, deverá ser possível adicionar vértices, obter o n-ésimo vértice e obter o número de vértices do Modelo. A ordem dos vértices que contém será relevante, uma vez que cada sequência de 3 vértices representará um triângulo no espaço.

3.1.2 Vértice

Um Vértice será um objeto que conterá 3 coordenadas (x, y, z). Por isso, deverá ser possível obter cada uma delas e defini-las. Representa um ponto no espaço.

3.1.3 Leitura do ficheiro de configuração

O ficheiro de configuração está formatado em XML, pelo que para ser interpretado recorre-se à biblioteca externa *TinyXML2*. Assim, é procurada uma *scene* no ficheiro de configuração, que conterá os possíveis modelos representados em ficheiros .3d. Para cada um desses modelos, é interpretada a sua fonte .3d, sendo um exemplo de formato desse ficheiro:

```
1.2 3.5 6.3
3.7 5.7 8.9
5.3 4.9 3.2
```

Cada linha do ficheiro .3d representa um vértice, e em cada linha existem 3 *floats* separados por um espaço. O primeiro é a coordenada x, o segundo a coordenada y e o terceiro a coordenada z do vértice. Cada sequência de 3 vértices é representativa de um triângulo.
Cada um dos vértices presentes no ficheiro associado ao modelo é armazenado num `vector` de `Vertex`. Cada modelo é armazenado num `vector` de `Model`.

3.1.4 Renderização dos modelos

Para renderizar os modelos armazenados, na função `renderScene`, que é passada como argumento da função do *glut* `glutDisplayFunc()`, são iterados todos os modelos, e para cada modelo são iterados todos os seus vértices e renderizados com recurso à função `glVertex3f()`. Assim, como este ciclo está entre um `glBegin(GL_TRIANGLES)` e um `glEnd()`, cada 3 vértices são interpretados como um triângulo, como pretendido.
Tendo em conta as primitivas pedidas, o nosso motor gráfico renderiza-as de acordo com as figuras 3.1, 3.2, 3.3 e 3.4.

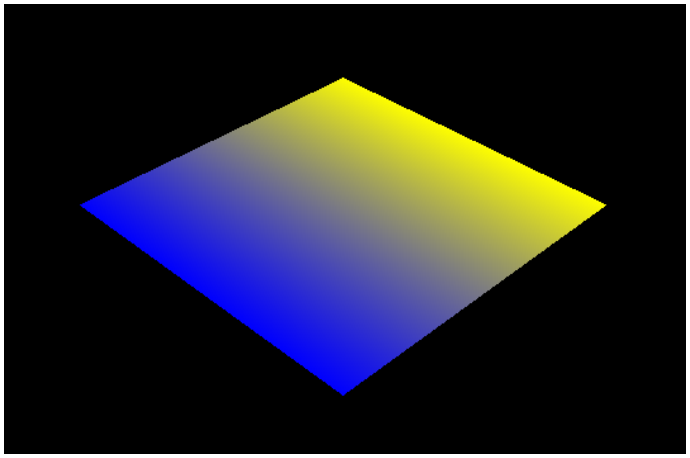

Figura 3.1: Plano de tamanho 3

3.2 Gerador

3.2.1 Plano

Sabendo que para gerar o plano apenas é necessário definir dois triângulos, foi criado um ficheiro *plane.cpp*, que contém uma função que, dado um tamanho e o nome dum ficheiro, armazena os vértices gerados nesse ficheiro (`void createPlane(float size, string fname)`). Demonstra-se de seguida o formato geral para a criação dos vértices:

```
float x, z;

x = size / 2;
z = size / 2;

// 1º Triângulo
writeToFile(x,0,z,file);
```

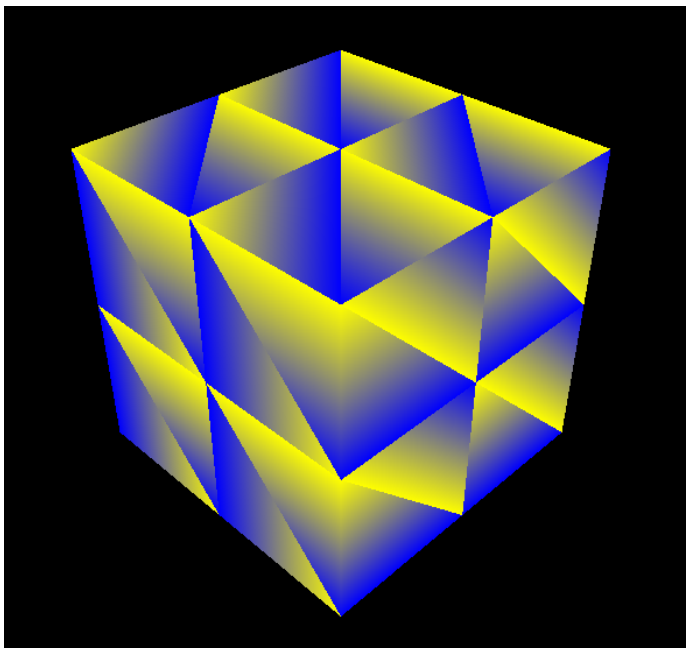

Figura 3.2: Caixa de tamanho 3 nos 3 eixos e 2 divisões

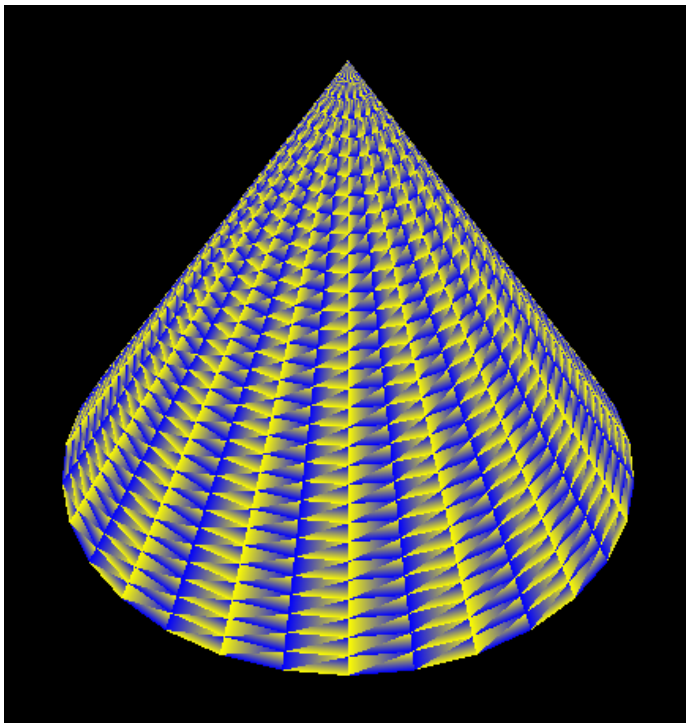

Figura 3.3: Cone de raio 2, altura 3, 32 *stacks* e 32 *slices*

```
writeToFile(-x,0,-z,file);
writeToFile(-x,0,z,file);

// 2º Triângulo
```

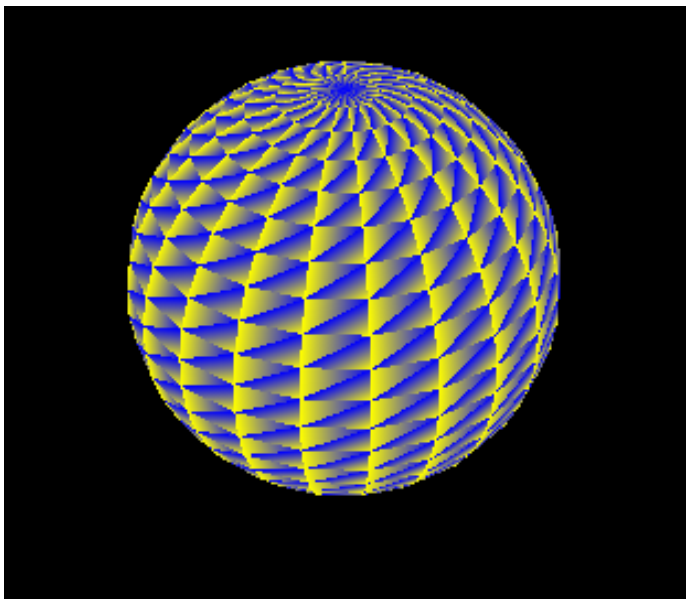

Figura 3.4: Esfera de raio 1, 21 *stacks* e 21 *slices*

```
writeToFile(-x,0,-z,file);
writeToFile(x,0,z,file);
writeToFile(x,0,-z,file);
```

A função `writeToFile` escreve os pontos no ficheiro, sendo que estes já estão ordenados de modo a que as faces dos triângulos estejam voltadas para Y positivo.

3.2.2 Caixa

Depois de calculado em termos matemáticos o desenho de uma caixa na secção 2.2, implementou-se a geração de vértices em C++. Assim, definiu-se um ficheiro *box.cpp* que contém uma função que, dadas 3 coordenadas (x, y, z) e nome de um ficheiro, armazena os vértices gerados nesse ficheiro (`int createBox(float x, float y, float z, int div, char * name)`.

Demonstra-se de seguida o formato geral para a criação dos vértices:

```
float x0=x/2; // center of box in x
float y0=y/2; // center of box in y
float z0=z/2; // center of box in z

float nx=x/div; // step in x
float ny=y/div; // step in y
float nz=z/div; // step in z

drawXZ(x0,y0,z0,nx,nz,div,file);
drawYZ(x0,y0,z0,ny,nz,div,file);
drawXY(x0,y0,z0,nx,ny,div,file);
```

As funções `drawXZ`,`drawYZ` e `drawXY` geram os vértices das faces em cada um destes planos, sendo que depois

são escritos sequencialmente para um ficheiro de modo a obter as faces visíveis para fora correctamente e de acordo com o *standard* do *OpenGL*[2] (regra da mão direita).

3.2.3 Cone

Para a geração dos vértices de um cone, criou-se um ficheiro *cone.cpp* que contém uma função que dado um raio, a altura do cone, número de *slices*, número de *stacks* e nome de um ficheiro, armazena os vértices gerados nesse ficheiro (`int createCone(float botRad, float height, int slices, int stacks, char * fname)`. Como foi referido acima, para a criação de um triângulo da base do cone, são usadas as expressões 2.14, 2.15 e 2.16. Por conseguinte, para criar todos os triângulos basta utilizar um ciclo que itera o número de slices. Relativamente à superfície lateral do cone, para cada slice é iterado o número de stacks, de acordo com as expressões 2.22, 2.23, 2.24 e 2.25.

```
float angle = 2*M_PI/slices;
float division = height/stacks;

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

    desenhaBase(botRad,angle,slices,i,file);

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

        float div = (float) j / stacks;
        float radiusD = (float) (1.0 - (div)) * botRad;

        div = (j + 1) / stacks;
        float radiusU = (1.0 - (div)) * botRad;

        desenhaSuperficieLateral(radioD, radiusU,i ,j, angle, division, file);

    }
}
```

Sendo a ordem em que os vértices são imprimidos relevante, para os vértices da base usamos a ordem [OAB] (ver figura 2.3), enquanto que para os vértices da superfície lateral do cone será [ACB] e [BCD] (ver figura 2.4).

3.2.4 Esfera

Após ter-se formalizado em termos matemáticos o desenho de uma esfera na secção 2.4, implementou-se a geração de vértices em C++. Assim, definiu-se um ficheiro *sphere.cpp* que contém uma função que dado um raio, número de *slices*, número de *stacks* e nome de um ficheiro, armazena os vértices gerados nesse ficheiro (`int createSphere(float rad, int slices, int stacks, char * fname)`.
Demonstra-se de seguida o formato geral para a criação dos vértices:

```
float slice = (pi*2)/slices;
float stack = pi/stacks;
```

```
for(int i = 0; i < stacks; i++) {
    float beta = i*stack;
    for(int j = 0; j < slices; j++) {
        float alfa = j*slice;
        desenhaRetangulo(raio,beta,alfa,slice,stack,ficheiro);
    }
}
```

A função `desenhaRetangulo` gera os 4 vértices definidos matematicamente, sendo que depois escreve para um ficheiro uma sequência relevante deles mesmos (representativa de 2 triângulos). A sequência é a seguinte:

1. `print(B)`
2. `print(C)`
3. `print(D)`
4. `print(B)`
5. `print(D)`
6. `print(A)`

Assim, os triângulos [BCD] e [BDA] estão com a sua face visível para fora de acordo com o *standard* do *OpenGL*[2] (regra da mão direita).

Capítulo 4

Conclusão

Para esta primeira fase limitamo-nos a fazer estritamente o que era pedido no enunciado, não nos tendo alongado em funcionalidades extra. Decidimos implementá-las nas próximas fases, uma vez que o conhecimento e experiência prática serão maiores e permitirão o desenvolvimento de um melhor produto final. Em relação à utilização da linguagem C++, foi a primeira vez que os membros do grupo estiveram a trabalhar nela, pelo que para não atrasar o desenvolvimento do projeto, recorreram à programação em C para modularizar o programa (linguagem na qual estavam mais familiarizados). Ainda assim, foi possível gerar todas as primitivas propostas e renderizá-las num motor gráfico, o que cumpre com as expectativas definidas para a realização desta primeira fase.

Bibliografia

[1] Prem Sasi Kumar Arivukalanjiam, 2012.

[2] Richard Wright. *OpenGL Super Bible*. Sams publishing.