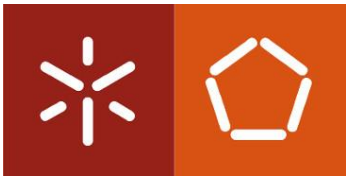

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

TRABALHO PRÁTICO – Fase 1

Graphical Primitives

Computação Gráfica
13 de março de 2021

Autores:

Grupo 8

Ana Filipa Pereira A89589

Carolina Santejo A89500

Raquel Costa A89464

Sara Marques A89477

Índice

Índice de Figuras

Introdução

Em âmbito da unidade curricular de Computação Gráfica, desenvolvemos a primeira fase do trabalho prático proposto. O grupo foi desafiado a demonstrar o conhecimento adquirido ao longo das aulas práticas e teóricas através da representação de algumas primitivas gráficas, considerando diferentes parâmetros.

Este projeto será desenvolvido ao longo de 4 fases, sendo que, para a realização desta primeira fase foi necessário utilizar a ferramenta *OpenGL*, utilizando a linguagem de programação C++.

Principais Objetivos

Nesta fase foi pedido ao grupo, que representassem 4 primitivas gráficas: um plano, uma caixa, uma esfera e um cone, de acordo com vários parâmetros, tais como: a altura, largura, raio, profundidade, e ainda, “slices” e “stacks”. A sua representação gráfica assenta no desenho de vários triângulos, sendo que para isto é necessário determinar os vários vértices que constituem esses mesmos triângulos.

Além disso, foi necessário criar duas aplicações distintas: um “motor” e um “gerador”. Sendo que o “motor” lê a informação (vértices) de uma primitiva de um dado ficheiro XML previamente gerado pela aplicação “gerador”, representando, de seguida, a respetiva figura. No caso do “gerador”, este cria um ficheiro XML que contêm uma “referência” ao nome de cada um dos ficheiros *.3d* criados que guardam os vértices calculados para um futuro desenho da respetiva figura.

Arquitetura do Programa

Tal como pedido, o programa encontra-se divido em duas aplicações: “Gerador” e “Motor”. Para tal temos dois packages, onde cada um deles contêm ficheiros responsáveis pelas funcionalidades de cada uma das aplicações, além do ficheiro *(Gerador/gerador.cpp* e *Motor/motor.cpp*) que contém a função *main* que se ocupa, com base nos argumentos dados pelo utilizador, de realizar as respetivas ações. Existe também o package “Utils” que está encarregue de struct, funções que o Motor e o Gerador têm comum. Desta forma, evitamos a repetição de código e redundância, obtendo assim um código mais organizado e “clean”. O package *tinyXML* foi descarregado do website https://sourceforge.net/projects/tinyxml/ , de forma a auxiliar a escrita e leitura de ficheiros XML. Estes packages encontram-se todos incluídos no package *src.*

Além disso, existe também o package *build*, que além de ter todos os executáveis criados pela *MakeFile* gerada pela *CMakeList.txt*, tem também a diretoria *3dFiles*, que guarda todos os ficheiros XML gerados e os ficheiros *.3d.*

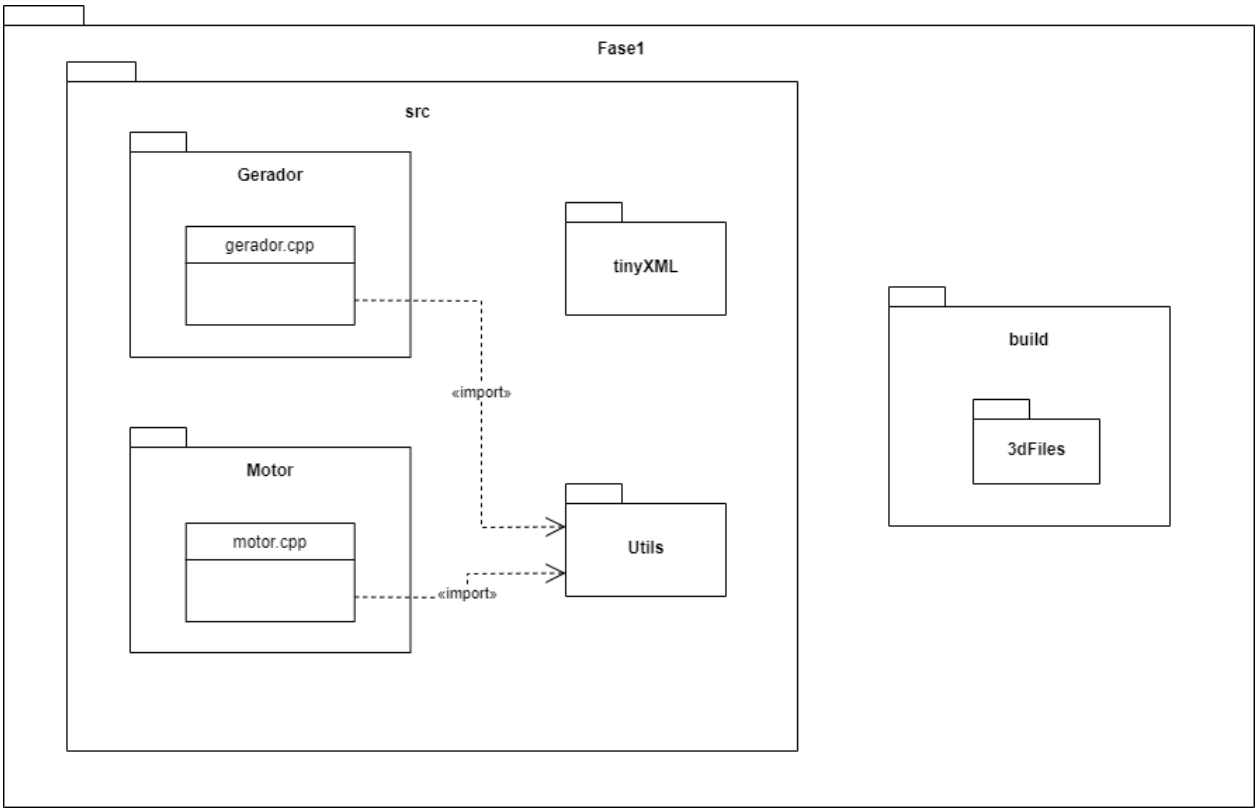

Figura 1 - Diagrama de Packages

Utils

De modo a facilitar a implementação das funções necessárias ao programa, foi criado um *namespace utils*, que contém estruturas e funções utilizadas tanto pelo motor (através do *header drawFunctions*) como pelo *engine* (através do *header calculaVertices*).

.hpp

No *header utils.hpp*, foi definida uma estrutura *point*, que contem as coordenadas *float* x, y e z que definem um dado ponto no espaço, e uma classe *figure* constituída por um vetor de *points*, sendo esta a estrutura a partir da qual será obtida a informação necessária para desenhar as várias figuras. O vetor nela contido é de acesso público, visto que o objetivo do uso da *figure* não é encapsular estes dados, mas sim facilitar a adição de novos pontos ao vetor através da função *addPoint*, instanciada também neste *header* e definida em *utils.ccp*.

Para além destas estruturas, é incluída a função *getPath*, também definida em *utils.ccp*. Esta função será usada pelo motor e pelo *engine* para aceder ou criar ficheiros *xml* e 3d numa pasta específica, sendo por isso necessária para identificar a diretoria relevante.

.cpp

O ficheiro *utils.cpp* contém apenas a definição das funções *addPoint* e *getPath* previamente mencionadas.

A função *addPoint* é uma função auxiliar que cria um *point* partindo das suas coordenadas x, y e z, e acrescenta-o ao vetor da classe *figure*. Trata-se de uma função auxiliar usada frequentemente na criação de novos vetores de pontos (*figures)*. Ao defini-la enquanto uma operação que atua dentro de uma classe, ao invés de receber e devolver um novo vetor, tornamos a sua implementação no código mais simples e levamos a que este seja muito menos extenso e repetitivo.

A função *getPath* apenas identifica a diretoria atual em que o programa está a correr, e devolve uma *string* correspondente ao *path* para uma pasta 3dFiles nela contida. Visto que a definição das diretorias em c++ é diferente dependendo do sistema operativo em que o programa é utilizado, é identificado se o programa está a correr em Windows ou Linux antes de definir os separadores do *path* ("\\"para Windows, "/" para Linux) e a função que identifica a diretoria atual (getcwd para Windows, _getcwd para Linux).

Generator/Gerador

Comandos

Através do seguinte comando:

```
./gerador -help
```

Temos acesso a um menu (Figura 2) que disponibiliza todos os comandos que o utilizador pode inserir.

```
Fase1/build$ ./gerador -help
Plane       [x] [y] [file.3d] [file.xml]
Box         [x] [y] [z] [divisions per edge] [file.3d] [file.xml]
Sphere      [radius] [slices] [stacks] [file.3d] [file.xml]
Cone        [radius] [height] [slices] [stacks] [file.3d] [file.xml]
```

Figura 2 - Menu -help

Cálculo de Vértices

Foi criado um ficheiro .cpp que contém todas as funções dedicadas ao cálculo dos vértices das figuras que o utilizador pretende representar. Após isto, esses mesmos vértices, são guardados numa *struct* apropriada (*figure*), para que depois possam ser armazenados num ficheiro *.3d*, que irá ser lido pela aplicação "Motor" e exibir as respetivas figuras.

▪ Plane/Plano

A função que gera os vértices do plano recebe como parâmetros a largura e comprimento do mesmo (x e z). Sendo que o plano será centrado na origem do referencial, e será formado por dois triângulos, as coordenadas dos seus pontos serão as seguintes.

Primeiro triângulo: $\left(\frac{x}{2}, 0, \frac{z}{2}\right)$ $\left(\frac{x}{2}, 0, -\frac{z}{2}\right)$ $\left(-\frac{x}{2}, 0, -\frac{z}{2}\right)$

Segundo triângulo: $\left(-\frac{x}{2}, 0, -\frac{z}{2}\right)$ $\left(-\frac{x}{2}, 0, \frac{z}{2}\right)$ $\left(-\frac{x}{2}, 0, -\frac{z}{2}\right)$

A partir da aplicação Motor será então possível desenhar figuras como as aqui exemplificadas:

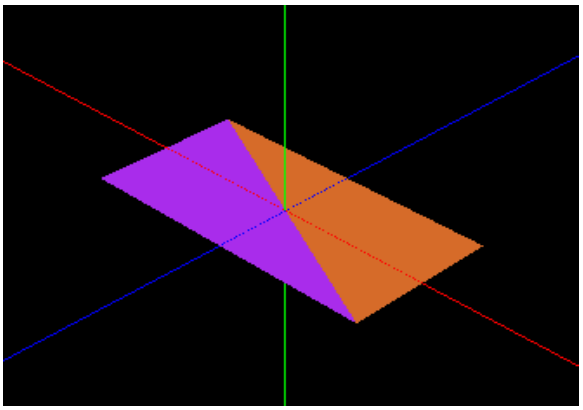

Figura 3 - Plano com x=2, z=1

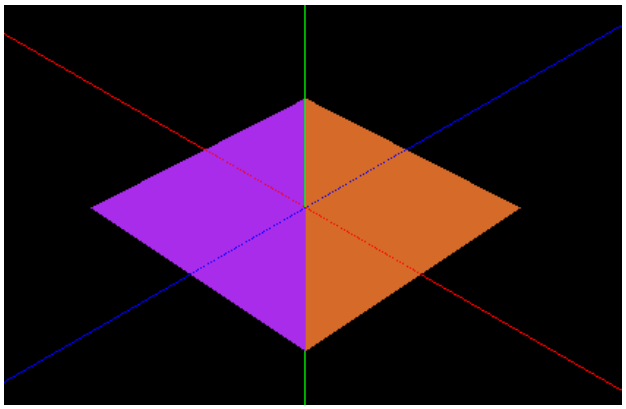

Figura 4 - Plano com x=2, z=2

Box/Caixa

A função *box* recebe como parâmetros a largura(*x*), o comprimento(*z*), a altura(*y*) e o número de divisões por aresta (*camadas*) e com estes valores calcula todos os vértices necessários para desenhar uma caixa com as medidas dadas. Por sua vez, o argumento *camadas* define em quantas partes iguais se tenciona dividir o sólido. Em relação à sua posição no referencial, foi definido que a sua base pertence ao plano xOz e que a caixa se encontra centrada na origem em relação aos eixos x e z.

Os planos das faces da caixa podem ser representados pelas seguintes equações gerais: (sendo x', y' e z' os parâmetros recebidos).

- Base: $y = 0$
- Teto: $y = y'$
- Laterais:
  $x = x'/2$; $x = -x'/2$; $z = z'/2$; $z = -z'/2$

Para desenhar o sólido, foram seguidos os seguintes passos:

- Divisão de cada medida da caixa pelo número de camadas para saber as medidas dos planos mais pequenos que definem as divisões de cada face (se o número de camadas for 1 teremos apenas um plano que é a face);
- Calcular os valores das coordenadas de um vértice inicial de cada face, sabendo as medidas e a posição do sólido no referencial;

- Calcular vértices de todos os planos pequenos de cada face. A ordem desta operação foi da esquerda para a direita e de cima para baixo (Figura 5) (se o observador estiver de "frente" para a parte visível da face). Como já foi referido, um plano constitui 6 vértices, logo cada face da caixa terá:

$$Nª\ vertices\ por\ face = 6 * Nº\ planos\ por\ face$$

$$Nº\ planos\ por\ face = camadas^2$$

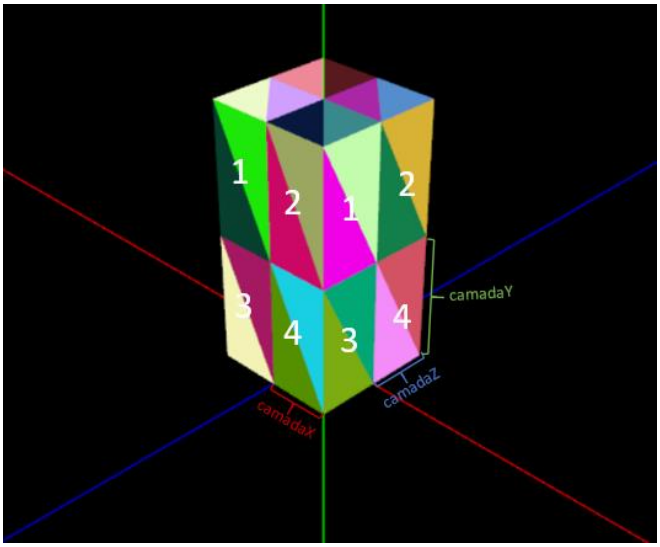

Figura 5 - Ordem do desenho dos planos de cada face da caixa

- Após o cálculo de todos os vértices necessários, estes são guardados no respetivo ficheiro para posteriormente serem lidos pelo motor que vai desenhar a figura respeitando a ordem fornecida.

Exemplo de caixas com diferentes dimensões e camadas:

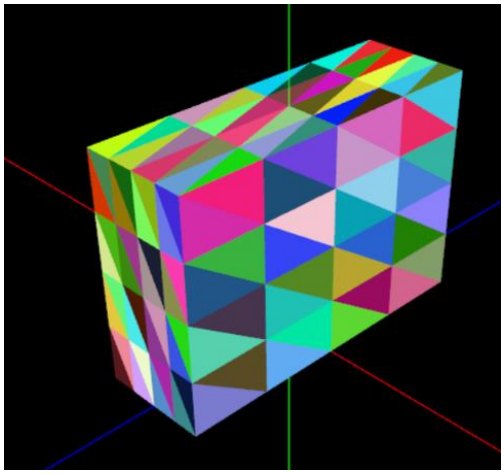

Figura 7 - Caixa 1x2x3

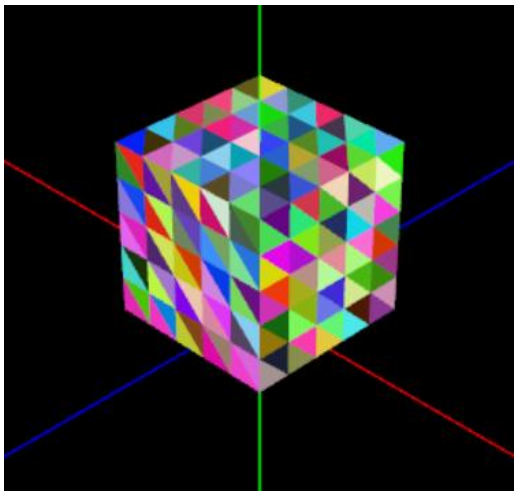

Figura 6 - Caixa 1x1x1

Sphere/Esfera

A função que gera os vértices, recebe como argumentos três parâmetros: o raio da esfera, o número de *stacks* e o número de "fatias" ou *slices* (na Figura 9 são designados *sectors*), sendo que quando maior for o número destas *slices* e *stacks*, mais redonda será a esfera (ver Figura 8).

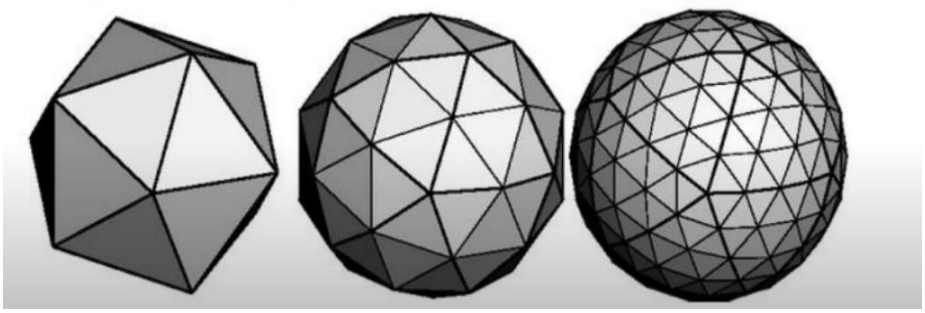

Figura 8 - Forma como o número de triângulos afeta o quão redonda a esfera irá parecer

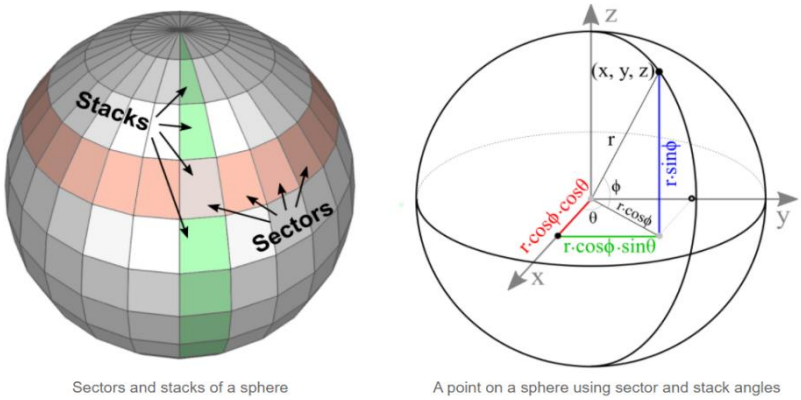

Figura 9 - Exemplificação dos vértices e ângulos da esfera com raio r

Tendo em conta a segunda esfera da Figura 9 conseguimos obter toda a informação necessária quanto aos vértices que constituem a esfera (cujo centro coincide com a origem do referencial).

Desta forma temos, inicialmente, o seguinte:

- $\frac{-\pi}{2} \leq \Phi \leq \frac{\pi}{2}$ → intervalo de tamanho π
- $0 \leq \theta < 2 * \pi$ → intervalo de tamanho 2*π

Com este intervalo, e o número de *stacks* e *slices* conseguimos saber a o valor da variação dos ângulos:

- $\Delta\Phi = \frac{\pi}{numStacks}$
- $\Delta\theta = \frac{\pi}{numSlices}$

Pela figura conseguimos também saber a expressão genérica de cada uma das coordenadas de um vértice. No entanto, é importante referir que em *OpenGL* os eixos são diferentes, ou seja, o eixo x da figura iremos considerar o eixo z, o eixo z será considerado o eixo y e o eixo y será considerado o x. Com isto sabemos que:

- $x = raio * cos(\Phi) * sin(\theta)$
- $y = raio * sin(\Phi)$
- $z = raio * cos(\Phi) * cos(\theta)$

Para tal, na função que calcula os vértices de uma esfera segundo os parâmetros dados pelo utilizador, foram feitos dois ciclos *for* pelos intervalos $[\frac{-\pi}{2} ; \frac{\pi}{2}]$ e $[0 ; 2\pi - \Delta\theta]$, de forma a que possamos iterar por todas as secções (trapézios) formadas pela interceção das *slacks* com as *slices*.

Assim, para cada secção é só criar dois triângulos (6 vértices) que formam esse pequeno plano, tendo em conta que a cada iteração de i, o ângulo Φ aumenta Δϕ (logo ϕ = $\frac{-\pi}{2} + i * \Delta\phi$). Já a cada iteração de j, o ângulo θ aumenta Δθ (logo ϕ = $j * \Delta\theta$).

Desta forma, conseguimos obter todos os pontos, cada um caracterizado pela *struct Point* que contém as coordenadas (x, y, z) do ponto em questão, incluídos num vetor da *struct Figure*. Quando executamos a outra aplicação ("Motor") obtemos por exemplo os seguintes desenhos:

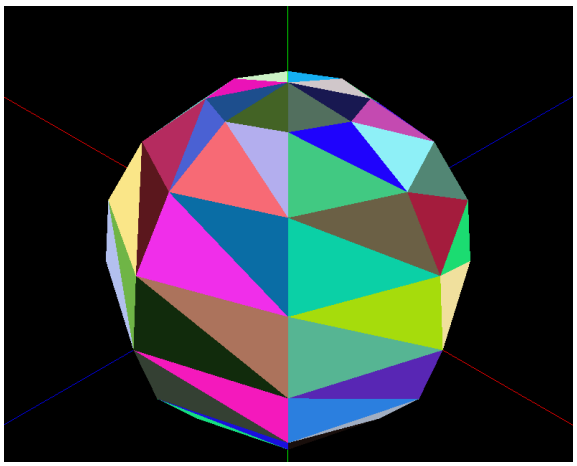

Figura 10 - Esfera com raio 2.5, slices 8, stacks 8

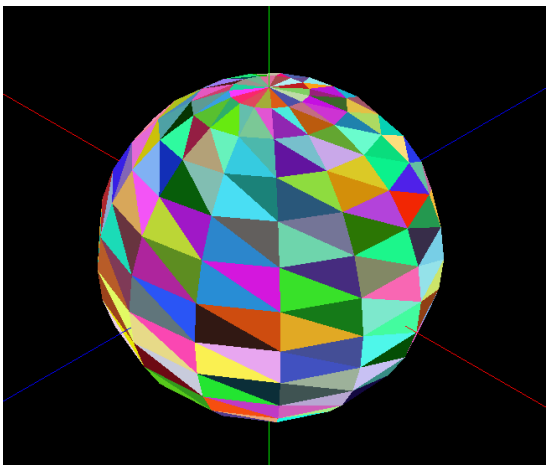

Figura 11 - Esfera com raio 2, slices 15, stacks 15

Cone

A função *cone* recebe como parâmetros o raio (*radius*), a altura (*height*), o número de fatias (*slices*) e o número de camadas (*stacks*) e com estes valores calcula todos os vértices necessários para desenhar um cone com as medidas dadas. O argumento *stacks* define o número de "cortes" horizontais da base até ao bico, enquanto que as *slices* vão determinar o numero de divisões da base em partes iguais que, por sua vez, estabelecem o nível de curvatura do seu "circulo". A figura que representa a base do cone será sempre um polígono regular com *slices* lados, ou seja, quanto maior for este valor mais parecido com um círculo se tornará ao observador.

Em relação à sua posição no referencial, foi definido que a sua base se situa no plano xOz e que o cone se encontra centrado na origem em relação aos eixos x e z.

Para desenhar o sólido, foram seguidos os seguintes passos:

- Como se verifica na Figura 12, a base do cone pode ser desenhada a partir de um triângulo isósceles, com um vértice na origem, que roda em torno do eixo y. Assim, a partir do cálculo do angulo Δ (angulo entre os dois pontos da base do triângulo) e do raio podem ser calculados todos os vértices do polígono pelas equações abaixo:

$$x = \text{radius} * \sin(\theta)$$
$$z = \text{radius} * \cos(\theta)$$
$$\theta = i * \Delta$$

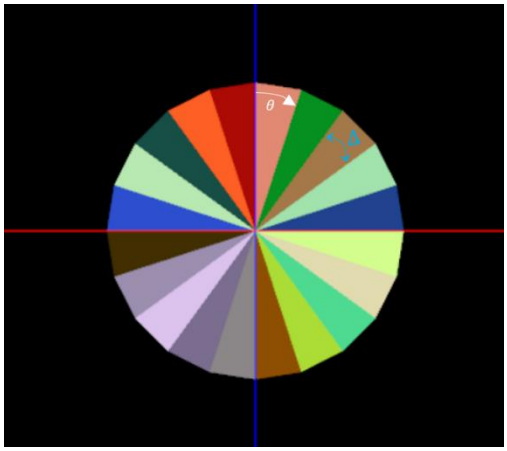

Figura 12 - Base do cone de raio 1 com 20 slices

- Para calcular todos os vértices das faces laterais do cone, para cada fatia foi-se diminuindo o raio e aumentado o valor de y de acordo com o número de *stacks (*Figura 13*)*.

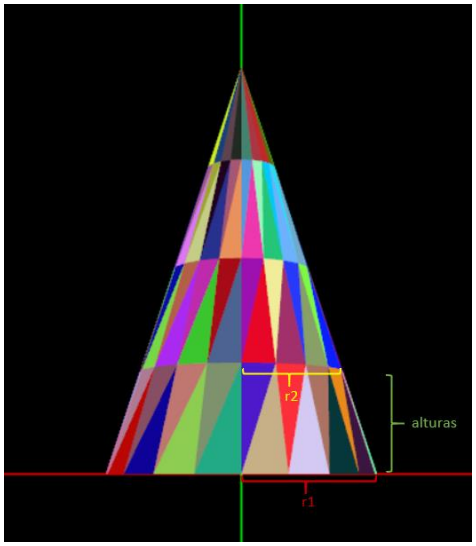

Figura 13 - Vista lateral do cone de raio 1, altura 3, 20 *slices* e 4 *stacks*

- Sabendo todos os vértices acima referidos é assim possível desenhar o cone pretendido.

Como se pode observar na Figura 14, quando o número de *slices* é substancialmente maior temos uma superfície praticamente circular.

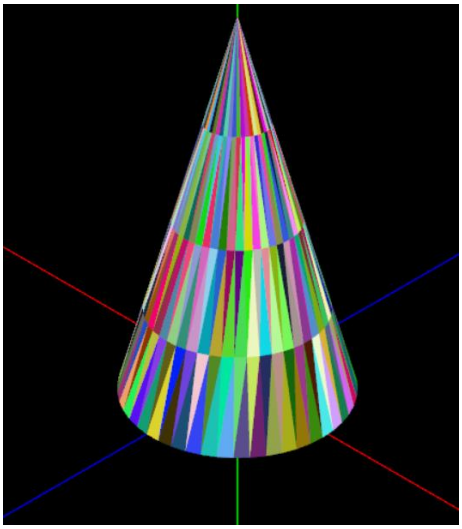

Figura 14 - Cone de raio 1 com 50 *slices*

Criação e Escrita nos ficheiros

Após os pontos serem calculados para representar a figura correspondente ao pedido do utilizador, estes são armazenados num ficheiro *.3d*, onde cada linha representa um ponto com as coordenadas $(x\ y\ z)$. Tal como podemos observar no seguinte exemplo:

```
1   0 -1.97538 0.312869
2   -0 -2 -8.74228e-08
3   -2.70151e-08 -2 -8.3144e-08
4   0 -1.97538 0.312869
5   -2.70151e-08 -2 -8.3144e-08
6   0.0966818 -1.97538 0.297556
7   0.0966818 -1.97538 0.297556
8   -2.70151e-08 -2 -8.3144e-08
9   -5.13858e-08 -2 -7.07265e-08
10  0.0966818 -1.97538 0.297556
11  -5.13858e-08 -2 -7.07265e-08
12  0.1839 -1.97538 0.253116
13  0.1839 -1.97538 0.253116
```

Figura 15 - Exemplo de um ficheiro .3d

De seguida, se o ficheiro tiver sido criado com sucesso, procedemos para a escrita da referência do seu nome no ficheiro *.xml*. Para tal usufruímos do *TinyXML* como auxiliar, como já foi referido. De acordo com a abordagem adotada, criamos um ficheiro *xml* com o nome escolhido pelo utilizador, criamos o "parentNode" que neste caso é o "Root element" <Scene> e adicionamos o nome do ficheiro .3d que será o "FirstChild", sendo o elemento <Model>. Mas caso esse ficheiro já exista, então nós fazemos "load" do mesmo, e adicionamos após o último filho do ficheiro *XML*, um novo <Model> que irá conter o nome do ficheiro *.3d* que queremos adicionar. Na Figura 16 podemos observar um exemplo de um ficheiro XML criado pelo Gerador.

```
<Scene>
    <Model file="cone.3d" />
    <Model file="sphere.3d" />
    <Model file="box.3d" />
    <Model file="plane.3d" />
</Scene>
```

Figura 16 - Exemplo de um ficheiro .xml

Engine/Motor

Comando

Após o utilizador executar a aplicação "Gerador", e desta gerar um ficheiro XML com o nome dado, poderá agora correr a aplicação "Motor" e passar como argumento o nome do ficheiro que este pretende que seja exibido.

```
./motor scenes.xml
```

Leitura do Ficheiro

Quando é feita a leitura do ficheiro XML que foi passado como argumento, é feita uma iteração de elemento a elemento, e abre-se cada um dos ficheiros *.3d*, e começa-se a fazer uma leitura de linha em linha. Sendo que cada linha representa um Ponto, guardamos todos esses pontos num vetor da *struct Figure*.

Estrutura de dados

Após a leitura do ficheiro, adicionamos cada Figure obtida a uma estrutura de dados. Essa estrutura consiste num *Map<Key, Value>*, onde a *key* é um número inteiro de acordo com a ordem que a primitiva se encontrava no ficheiro XML, e o *value* é a Figure.

Portanto, temos a estrutura: `map<int, figure> figurasMap;`

Esta estrutura foi feita de modo a conseguirmos representar uma figura de cada vez, sem esta estrutura então teríamos as figuras representadas todas em simultâneo. Para já nesta primeira fase o grupo decidiu manter esta funcionalidade de conseguir exibir uma figura de cada vez, sendo que com o decorrer das próximas fases o grupo irá avaliar se deverá retirar esta estrutura de modo a obter um só desenho que contenha todas as figuras existentes no ficheiro XML.

Desenho das figuras

O desenho das figuras será possibilitado pelo uso das funções definidas no *namespace draw* em drawFunctions (hpp e cpp). Para tal, é utilizada a função drawFigure, que lê o vetor de pontos contido numa dada *figure*, selecionando cada três pontos dados por ordem e aplicando a função auxiliar drawTriangle, que irá desenhar um triângulo de uma cor aleatória partindo das coordenadas desses mesmos pontos. Deste modo, todos os triângulos de uma dada figura terão cores diferentes, o que torna a forma obtida mais distinguível.

É de notar que o *namespace draw* contém, também, a função drawReferencial, responsável por desenhar as linhas do referencial em diferentes cores (RedGreenBlue - xyz).

Output

Gerador

O output do gerador serve apenas para confirmar ao utilizador se os ficheiros pretendidos foram criados com sucesso.

Figura 17 - Output para a criação bem sucedida dos ficheiros .3d e .xml

Motor

O output do motor indica ao utilizador quando o ficheiro .xml não foi lido com sucesso (Figura 18) ou, caso a leitura seja bem sucedida, é aberta uma nova janela do *OpenGL* com o desenho das figuras pretendidas. Para que as figuras não fossem mostradas todas ao mesmo tempo (originando sobreposições), o grupo decidiu implementar uma funcionalidade adicional, que a partir do map *figurasMap* que associa cada figura ao seu ID, desenha uma de cada vez. Para percorrer esta estrutura de dados foi utilizada a função *glutKeyboardFunc* que permite ao utilizador utilizando as teclas 'a' e 'd' escolher a figura a visualizar (Figura 19).

Figura 18 - Output quando ficheiro .xml não existe

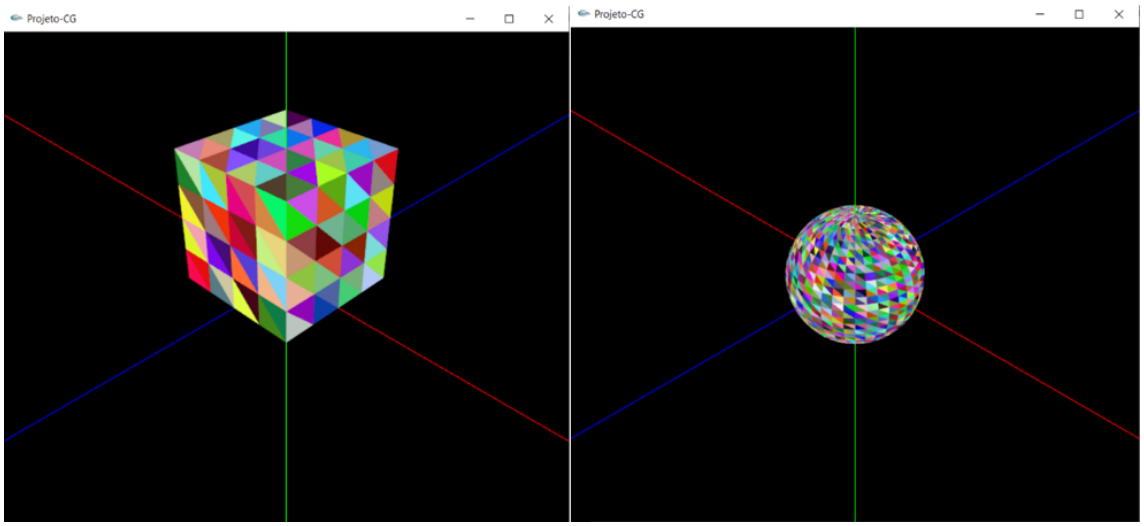

Figura 19- Outputs possíveis para o mesmo .xml com 2 figuras

Conclusão

Através do desenvolvimento deste trabalho pudemos aplicar os conhecimentos adquiridos nas aulas da unidade curricular de Computação Gráfica, e aprofundar a nossa compreensão do modo de funcionamento do OpenGL e da linguagem C++. Para além disto, permitiu introduzir-nos a "markup languages", neste caso o XML, e trabalhar com a leitura e escrita de ficheiros deste tipo.

Consideramos que desenvolvemos um bom trabalho, que implementa todas as funcionalidades exigidas nesta fase do trabalho. A abordagem utilizada poderá, no entanto, vir a sofrer alterações mediante os requisitos das fases seguintes.