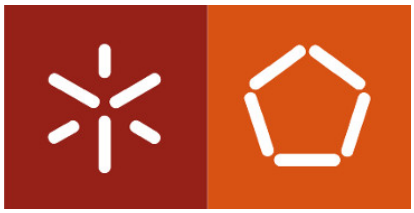

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica - Trabalho Prático - Fase 1
Ano Letivo 2021/2022
Grupo 6

Gonçalo Braz (a93178) Simão Cunha (a93262)
Tiago Silva (a93277) Gonçalo Pereira (a93168)

17 de março de 2022

Resumo

O presente relatório inicia-se introduzindo o contexto do trabalho, seguido de uma descrição do trabalho proposto e do trabalho desenvolvido. Sucede-se a resolução formal e matemática do problema de geração dos vértices que compõem as figuras, formalização essa que serviu como base de trabalho para a implementação da solução. Por fim, apresenta-se a estrutura da solução desenvolvida na linguagem de programação `C++`. Neste último capitulo, são descritos os diferentes componentes da solução e a ideia geral da sua implementação.

Conteúdo

Lista de Figuras

1 Introdução

1.1 Contextualização

Este relatório surge no âmbito da primeira fase do trabalho prático da UC de Computação Gráfica, onde foi proposto pela equipa docente o desenvolvimento de uma cena gráfica 3D.

Nesta primeira fase, era pedido que criássemos uma aplicação para gerar os diversos vértices que compõem um dado modelo (plano, cubo, cone e esfera) e outra aplicação que monta a cena, lendo os ficheiros de configuração em `XML`, onde estão explicitadas as definições da câmera e os modelos que são precisos desenhar.

1.2 Descrição do trabalho proposto

Nesta fase, tivemos que construir duas aplicações: o *generator* e o *engine*. A primeira é responsável pela criação dos vértices das primitivas (figuras a desenhar), de acordo com os argumentos referidos abaixo:

Figura	Argumento(s)
Plano	Dimensão e divisões
Cubo	Dimensão e nº de divisões por aresta
Cone	Raio, altura, *slices* e *stacks*
Esfera	Raio, *slices* e *stacks*

Com isto, será gerado um ficheiro `.3d` com a configuração pretendida. Já esta segunda aplicação estará responsável por ler ficheiros `XML` que refere ficheiros `.3d` e irá renderizá-los.

1.3 Resumo do trabalho a desenvolver

Para o *generator*, deve ser desenvolvida uma função para cada figura em que dados os seus argumentos, 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 `GLUT`.

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 *engine*.

Por fim, o motor gráfico deverá interpretar um ficheiro `.xml`, recorrer ao `GLUT` para renderizar os triângulos produzidos pelo *generator*, e interpretar corretamente os vértices armazenados nos ficheiros `.3d`.

2 Resolução formal do problema

Neste capítulo iremos explicar o caminho seguido para a resolução da renderização das diferentes primitivas ou seja, a lógica necessária para ser possível o cálculo das coordenadas de todos os pontos das mesmas.

2.1 Plano

Para a criação de um plano são utilizados dois parâmetros: o comprimento do lado e o número de divisões ao longo de cada eixo. Note-se ainda que o plano é sempre criado sobre os eixos xz e centrado na origem. Tendo em conta os parâmetros fornecidos, foram inferidas as coordenadas dos vértices que compõem os triângulos que, por sua vez, compõem o plano.

A = (-size/2 , size/2)
B = (-size/2 + size/div , size/2)
C = (-size/2 , size/2 - size/div)

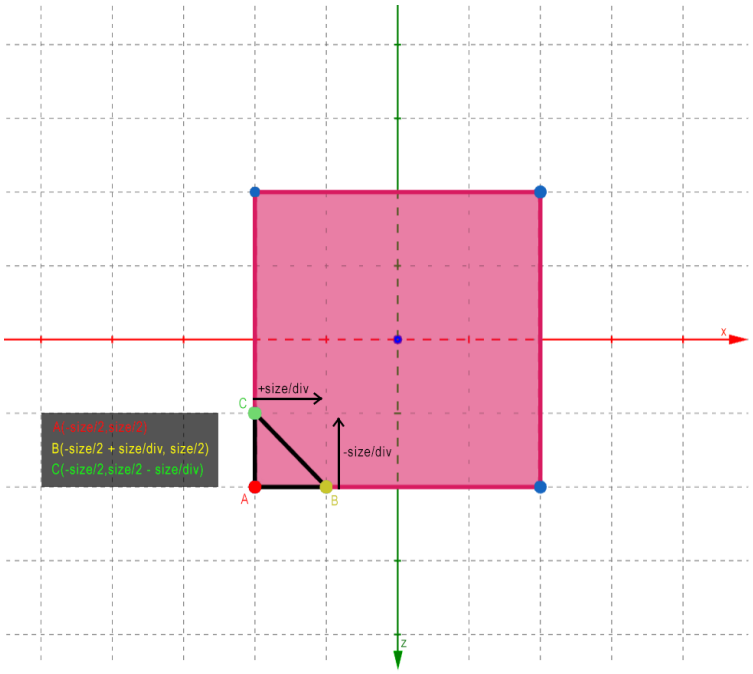

Figura 1: Plano

2.2 Cubo

Para a criação de um cubo são utilizados dois parâmetros: o comprimento do lado e o número de divisões de cada aresta. O cubo é centrado na origem.

Um cubo por natureza apresenta 6 faces quadradas de iguais dimensões, assim sendo, podemos considerar que este é completo por 6 planos de iguais dimensões, que seguem o mesmo raciocínio explicado anteriormente.

Considerando isto, desenhamos o cubo reutilizando o pensamento do plano, variando a posição inicial para cada face e dependendo da face que se tratava, as duas variáveis que variam durante a construção da mesma.

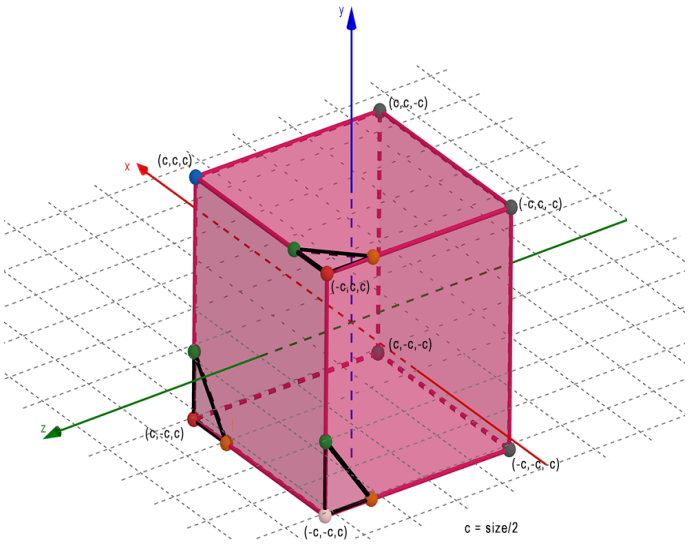

Figura 2: Cubo

2.3 Esfera

Para a criação da esfera são utilizados três parâmetros: o raio, as *slices* (divisões horizontais - podemos imaginar como sendo as fatias de um bolo) e o número de *stacks* (divisões verticais - as diferentes camadas que se sobrepõem verticalmente). De modo a resolver o problema, estabelecemos duas abordagens, primeiramente para tratar das diferentes *slices*, que irão definir as posições das coordenadas do x e z, tendo em conta o raio da *stack* que estamos a tratar no momento, e o tratamento das diferentes *stacks*, que irão originar a variação do y ou seja, a altura, e consequentemente o cálculo do raio para a respetiva *stack* (o raio é em relação ao eixo do y e não à origem da esfera, corte paralelo ao plano xOz).

Visto que o cálculo das coordenadas x e z numa determinada *stack* de uma *slice* são dependentes do raio dessa mesma *stack*, vamos começar pelo cálculo deste. Para isto, é necessário conhecer o ângulo β entre cada uma das *stacks*.

Este será igual a: $\beta = \frac{\pi}{nº\ stacks}$

Calculado β e tendo conhecimento do raio da esfera, conseguimos calcular os raios da *stack*, inferior e superior, assim com as suas alturas, valores da coordenada y, como se pode ver na figura 3.

Para calcular os raios necessitamos de conhecer dois ângulos: a e b do limite inferior e superior da *stack* respetivamente. Os cálculos para a sua obtenção são:

$$a = -\frac{\pi}{2} + \beta * stack$$
$$b = -\frac{\pi}{2} + \beta * (stack + 1)$$

NOTA: no programa desenhamos a esfera de baixo para cima, desse modo começamos a partir de $-\frac{\pi}{2}$ e vamos somando a este ângulo à medida em que subimos de *stack*.

Depois de calcular estes ângulos, conseguimos então calcular os raios inferior e superior, assim como as alturas dos limites correspondentes. Como se pode observar na figura 3, aplicando por exemplo o ângulo a ao raio, desenhamos um triângulo retângulo com base que tem como projeção o raio inferior da *stack* e altura correspondente ao módulo de y deste limite.

Seja r o raio da esfera,

$$rInferior = cos(a) * r$$
$$hInferior = sin(a) * r$$

$$rSuperior = cos(b) * r$$
$$hSuperior = sin(b) * r$$

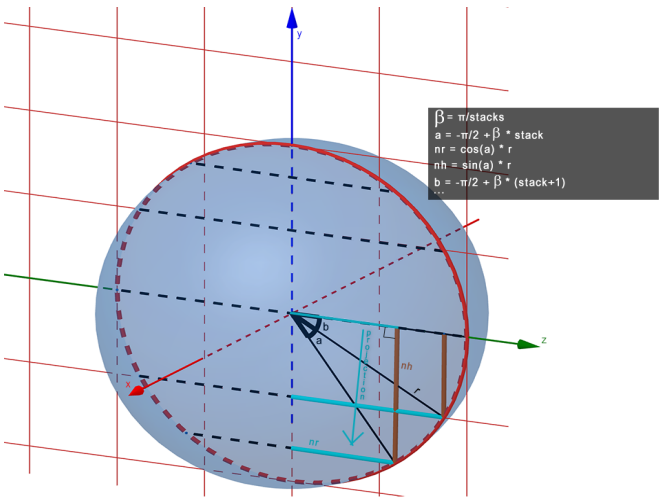

Figura 3: Esfera - divisão em *stacks*

Com estes dados somos capazes de calcular a posição y de todos os pontos de uma *stack* de uma *slice*. Além disso possuímos agora também o valor dos raios dos limites da *stack*.

Para calcular os valores de x e z dos pontos de uma *stack*, podemos novamente aplicar a lógica do *sohcahtoa* [2] como anteriormente, nos triângulos retângulos formados pelas *slices*, como por exemplo, o triângulo [AOB] na figura 4.

Com isto em mente, procedemos a descobrir o ângulo α.

$$\alpha = \frac{2\pi}{n^o\ slices}$$

Coordenadas de A:

$$x = cos(slice * \alpha) * r$$

$$r \equiv \text{obtido pelos cálculos anteriores}$$
$$slice \equiv \text{limite inferior da primeira slice, logo igual a 0 (slice - 1)}$$

$$\Rightarrow x = cos(0 * \alpha) * r$$

$$y \equiv \text{obtido pelos cálculos anteriores}$$

$$z = sin(slice * \alpha) * r$$
$$\Rightarrow z = sin(0 * \alpha) * r$$

Coordenadas de B:

$$x = cos(slice * \alpha) * r$$

$$slice \equiv \text{limite superior da primeira slice, logo igual a 1 (slice)}$$

$$\Rightarrow x = cos(1 * \alpha) * r$$

$$y \equiv \text{obtido pelos cálculos anteriores}$$

$$z = sin(slice * \alpha) * r$$
$$\Rightarrow z = sin(1 * \alpha) * r$$

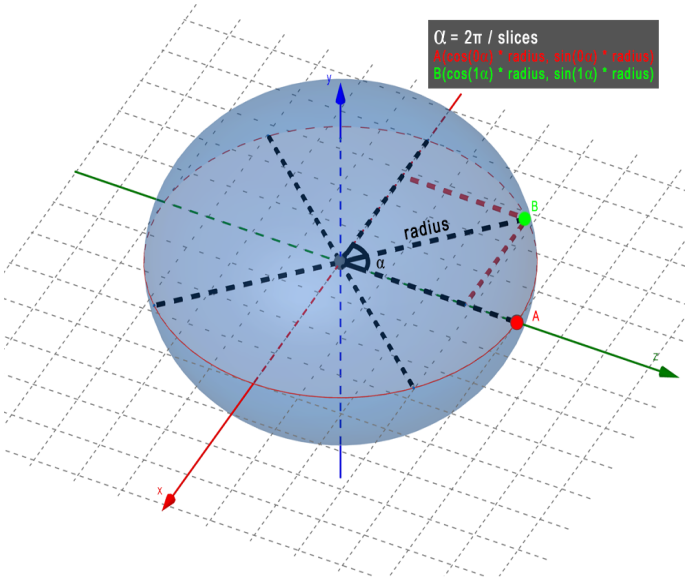

Figura 4: Esfera - divisão em *slices*

Conseguimos assim descobrir as coordenadas de todos os pontos necessários para desenhar uma esfera de raio r, com um dado número de *slices* e *stacks*.

2.4 Cone

Para a criação do cone são utilizados quatro parâmetros: o raio da base, a altura do cone, o número de *slices* e o número de *stacks*.

Para o tratamento das *slices*, ou seja, para descobrir as coordenadas do x e z, seguimos a mesma lógica utilizada no caso da esfera, visto que o corte realizado será semelhante (corte paralelo ao plano xOz, formará uma circunferência, tal como no caso da esfera).

De igual modo, as *stacks* seguem um raciocínio semelhante ao da esfera. Porém, ao contrário desta, que apresenta uma variação não linear do y à medida que o x e o z se aproxima de 0, fazendo com que as *stacks* que a formavam tivessem diferentes alturas, o cone apresenta uma variação linear do y logo, as *stacks* terão a mesma altura, sendo que conseguimos obter o y de um ponto sem necessitar do conhecimento de nenhum ângulo.

Assim sendo, considerando que o cone está centrado na origem do referencial, ou seja, o seu ponto mais alto terá y igual a $\frac{h}{2}$ e no mais baixo $-\frac{h}{2}$, numa dada *stack*, o valor de y será:

No limite inferior:

$$y = -\frac{h}{2} + hdiv * (stack - 1)$$

$$hdiv \equiv \text{altura de uma stack}, \frac{h}{\text{nº stacks}}$$

No limite superior:

$$y = -\frac{h}{2} + hdiv * (stack)$$

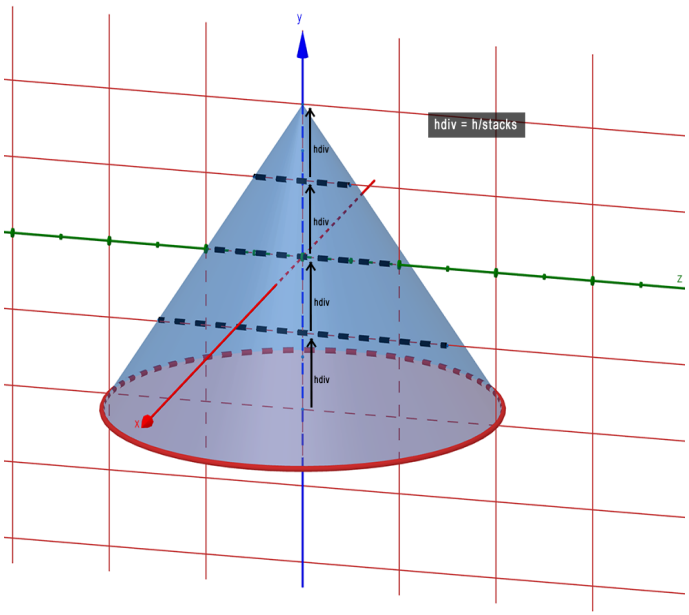

Figura 5: Cone

3 Arquitetura

3.1 Estruturas de dados

3.1.1 Point

A classe *Point* define os três campos de caracterizam um ponto. Além disso disponibiliza métodos para a sua manipulação e integra outras classes mais complexas

```
class point {
    private:
         float x;
         float y;
         float z;
}
```

3.1.2 Points

A classe *Points* mantêm uma *hashtable* que mapeia pontos únicos (não repetidos) associados a um índice, tem o campo `triangles` que é uma lista de *triangles_i* , o campo `ntriangles` que guarda o número de triângulos até ao momento e por fim o `buffer`, que é o espaço alocado para a lista. Quando `ntriangulos` igual ao tamanho do *buffer*, a lista é redimensionada com o dobro do espaço.

```
class points {
    private:
        map<size_t, point> map_points;
        triangle_i* triangles;
        int ntriangles;
        size_t buffer;
}
```

3.1.3 Triangles

A classe *Triangles* caracteriza os três pontos de um triângulo fazendo uso da classe *Point*. Esta classe tem a função de guardar três pontos que constituem um triângulo.

```
class triangle {
    private:
         point p1;
         point p2;
         point p3;
}
```

A classe *Triangles_i* guarda três índices, índices esses que mapeiam um ponto numa *HashTable*. Esta classe introduz uma otimização no que toca ao tamanho do ficheiro de uma figura, visto que, ao invés de repetirmos pontos, guardamos simplesmente os índices desses mesmos pontos, já que os índices ocupam menos espaço do que escrever um ponto. Deste modo, os índices funcionam como apontadores para locais na memória.

```
class triangle_i {
    public:
         size_t i1;
         size_t i2;
         size_t i3;
}
```

3.1.4 Models

A classe *Model* faz uso da classe *Points* para armazenar os pontos que constituem um modelo oriundos de um ficheiro.

```
class model {
    private:
        string filename;
        points ps;
}
```

A classe *Models* é essencialmente uma estrutura que guarda um conjunto de modelos. Esta classe é usada no momento da leitura do `XML` onde podem ser carregados vários modelos para compor uma cena.

```
class models {
    private:
         model* list_model;
         int nmodels;
         size_t buffer;
}
```

3.1.5 Camera

A classe *Camera* guarda as configurações da camera que são lidas do ficheiro `XML`.

```
class camera {
     public:
          float px, py, pz, lx, ly, lz, ux, uy, uz, fov, near, far;
}
```

3.2 *Generator*

3.2.1 Plano

No ficheiro `plane.cpp` está implementada a seguinte função:

```
int write_plane(float size, int div, char* fname);
```

A função recebe então o tamanho do lado do plano, o número de divisões de cada lado e por fim o ficheiro onde irá escrever a informação do plano (pontos únicos e índices).

Utilizando as fórmulas inferidas para os pontos do plano, a função começará a calcular os pontos dos triângulos do quadrilátero inferior esquerdo.

Partindo deste ponto, vão ser desenhados todos os quadriláteros de uma divisão, de baixo para cima, e só depois passa para a divisão seguinte, para a direita.

A cada três pontos calculados, um triângulo, seguindo a ordem da regra da mão direita, guardamos um triângulo na classe *points*.

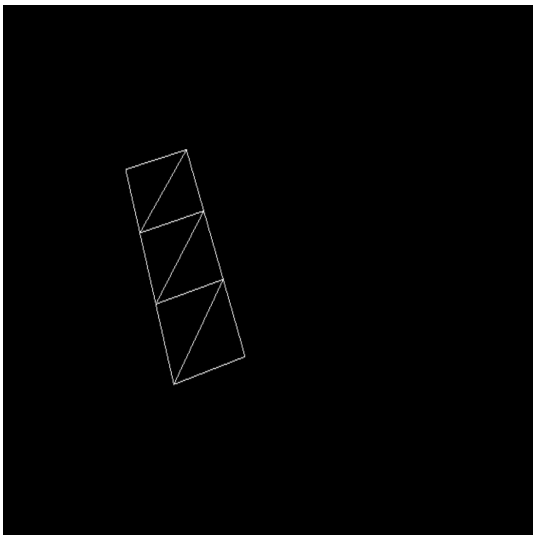

Figura 6: Divisão do plano

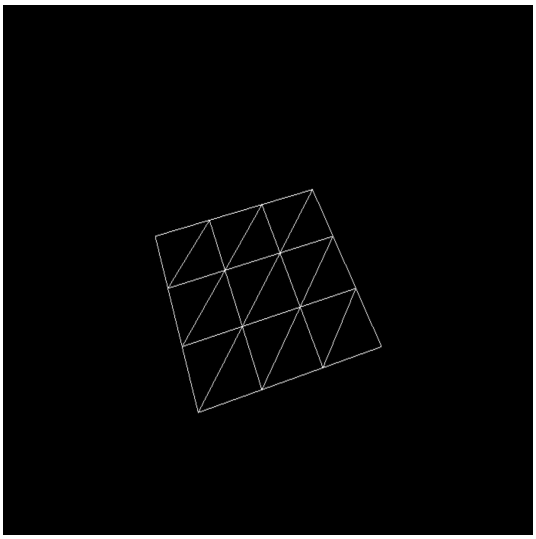

Figura 7: Plano completo

3.2.2 Cubo

No ficheiro `box.cpp` está implementada a seguinte função:

```
int write_box(float size, int div, char* fname)
```

A função recebe o tamanho do lado do cubo, o número de divisões de cada aresta e por fim o ficheiro onde irá escrever toda a informação do cubo.

Aqui, existe uma junção de 6 funções com o mesmo mecanismo adotado na função de escrita do plano, com o intuito de se desenhar as 6 faces que constituem um cubo. Todas as funções, como já referido, seguem o modelo da função que escreve o plano, sofrendo as devidas alterações para se encaixar nas faces correspondentes do cubo. Sendo assim, tem 2 funções para as faces no plano Oxz, 2 para as faces do plano Oxy e mais duas para o plano Ozy. As duas funções para cada plano são praticamente iguais, e apenas diferem na coordenada constante, que tem valores simétricas

entre elas, por exemplo para as funções do plano Oxy, uma tem o z como -size/2 e a outra como size/2. Chegamos a ponderar em juntar estas funções de cada face (dando o valor da constante como parâmetro ou então um booleano que controlasse o sinal), no entanto, acabamos por não o fazer, uma vez que achamos que seria mais legível e compreensível o código desta maneira.

3.2.3 Esfera

No ficheiro `sphere.cpp` está implementada a seguinte função:

```
int write_sphere(float radius, int slices, int stacks, char* fname)
```

Os argumentos da função são o raio da esfera, o número de *slices* e *stacks* da mesma e o ficheiro onde se vai escrever os dados da esfera, respetivamente.

Seguindo a lógica explicada no capítulo anterior para o cálculo das coordenadas dos pontos da esfera, começamos por calcular os ângulos α e β.

Os quadriláteros serão calculados de baixo para cima de uma *slice*, desenhando portanto todos as *stacks* dessa *slice* antes de se seguir para a próxima *slice*. Como as "tampas"inferior e superior da esfera, tal como no caso do cone são desenhadas utilizando simples triângulos invés de quadriláteros, estes serão desenhados separadamente do ciclo das *stacks*.

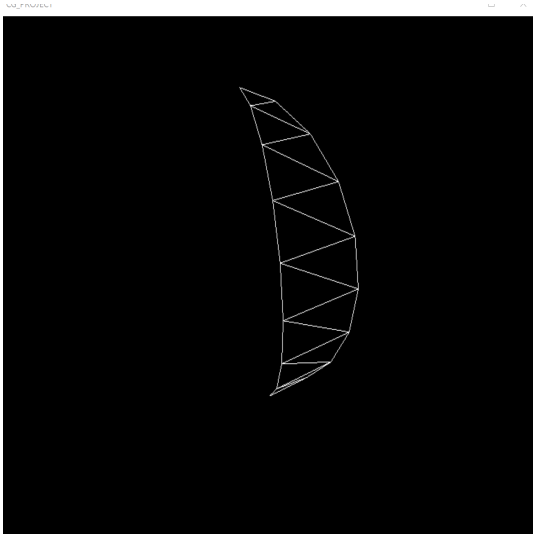

Figura 8: Slice da esfera

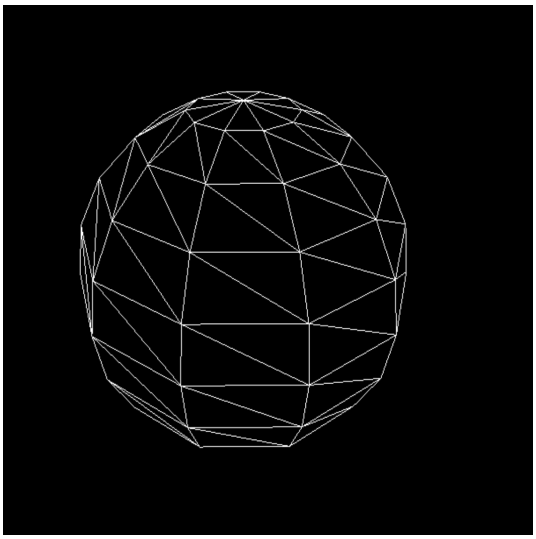

Figura 9: Esfera completa

3.2.4 Cone

No ficheiro `cone.cpp` está implementada a seguinte função:

```
int write_cone(float radius, float height, int slices, int stacks, char* fname)
```

A função recebe o raio da base do cone, número de *slices*, o número de *stacks* e, por fim, o ficheiro onde irá escrever toda a informação do cone.

De forma semelhante à esfera, irá desenhar uma *slice* completa antes de passar para a seguinte, novamente de baixo para cima. Seguindo também a mesma rotina, as "tampas"(base e topo) do cone serão calculadas separadamente do ciclo interior das *stacks*, começando portanto por desenhar um triângulo da base do cone e subindo *stack* a *stack* até chegar ao topo da *slice*.

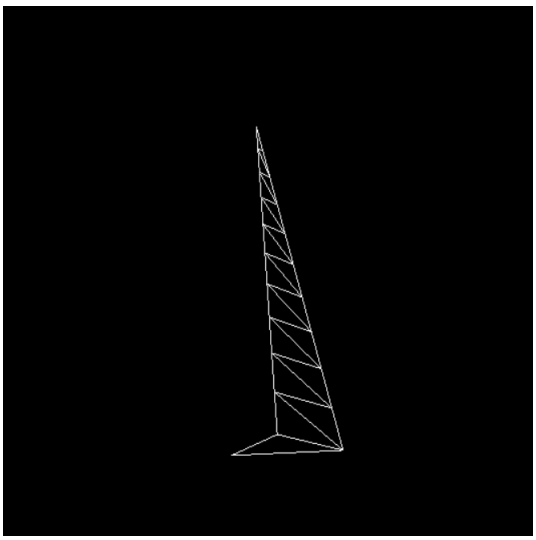

Figura 10: Slice do cone

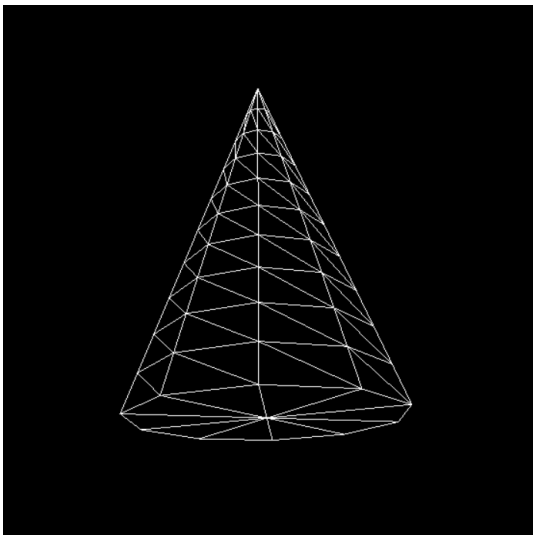

Figura 11: Cone completo

3.2.5 Writer

O ficheiro `writer.cpp` implementa as funções responsáveis pela escrita dos pontos calculados da primitiva dada como argumento no ficheiro especificado.

```
int write_points(points ps, FILE* file)
```

Para isto a função irá receber como argumento uma instância da classe *points*, onde estão guardados os pontos da primitiva, e o ficheiro onde serão escritos.

O *writer* utiliza o método da classe *points* para, em primeiro lugar, escrever no ficheiro todos os pontos que esta guarda na sua *hashtable* (pontos únicos) e, de seguida, utiliza o método da mesma classe para escrever todos os índices referentes aos pontos de 3 em 3, ou seja, triângulo em triângulo, de modo a ter uma partição mais fácil por parte do *parser* do *engine*.

3.3 *Engine*

3.3.1 Leitura do ficheiro de configuração

O *engine*, para saber a cena que tem de renderizar, tem que ler um ficheiro `XML` que obedece a uma estrutura bastante específica como demonstrado na imagem abaixo. Para fazer essa leitura, recorremos à biblioteca *TinyXml2*,

```xml
<world>
    <camera>
        <position x="10" y="10" z="10" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
         <projection fov="60" near="1" far="1000" />
     </camera>
    <group>
        <models>
            <model file="cone.3d" />
            <model file="plane.3d" />
        </models>
    </group>

</world>
```

Figura 12: Exemplo de ficheiro `XML`

A *tag* `world` é a *root tag* e explicita o contexto da cena. Dentro desta *tag* principal são introduzidas duas *tags*:

- `camera` que engloba dentro dela *tags* em que os seus atributos são valores para configurar a camera da cena;
- `group` que agrupa todos os modelos que participam na cena. Os modelos incluídos na cena têm que ter sido previamente gerados pelo *generator*.

3.3.2 Renderização dos modelos construídos

De modo a conseguirmos ver os modelos construídos no ecrã, será necessário renderizá-los. Para tal, recorremos à função `draw()` da classe *models* que irá a todos os modelos da cena (ou seja, aqueles que ler do ficheiro `XML`) e irá desenhar todos os triângulos constituintes. Por sua vez, esta função referida terá que estar incluída na função `renderScene()`, que será passada como argumento à função do *GLUT* `glutDisplayFunc(renderScene)`.

Assim, será possível observar todas os modelos do nosso trabalho prático, tal como é possível observar ao longo do nosso relatório.

4 Implementações extra

Além dos objetivos principais desta fase, tomámos a iniciativa de implementar alguns comandos de movimentação da câmera durante a renderização das cenas. Isto permite alguma interatividade e, adicionalmente, permitiu uma melhor análise das renderizações das primitivas.

O visualizador consegue movimentar a câmera em todas as direções no contexto 3D e ainda rodá-la.

Por defeito os triângulos são desenhados sem serem preenchidos ou seja, desenhando apenas as suas linhas, porém implementamos a opção de mudar para o preenchimento total de modo a permitir observar as primitivas pintadas na sua totalidade.

5 Conclusão

Nesta 1º fase do projeto implementámos um conjunto de primitivas num contexto 3D, através do estudo das suas formas e seguindo os princípios de renderização de cenas do `GLUT`.

Os pontos constituintes destas primitivas são guardados num ficheiro à parte pelo *generator*, de modo a que possam ser usadas pelo *engine*, de acordo com os cenários lidos por este vindos de ficheiros `XML`.

Além disso, implementamos uma otimização no modo de guardar pontos em ficheiro de modo a poupar recursos de espaço através da substituição da escrita de todos os pontos, inclusive pontos repetidos, por referências aos mesmos, que por norma ocupam menos espaço.

Assim, fomos capazes de realizar os objetivos propostos para esta primeira etapa que servirão como base para o trabalho futuro.

Referências

[1] https://www.cplusplus.com/reference/fstream/fstream/

[2] https://www.mathsisfun.com/algebra/sohcahtoa.html