Universidade do Minho

Licenciatura em Engenharia Informática

GitHub do Projeto

Computação Gráfica - Fase 1

Grupo 43

Eduardo Pereira (A94881)
Gonçalo Freitas (A96136)
Gonçalo Vale (A96923)
José Pereira (A89596)

Ano Letivo 2023/2024




Conteúdo

1 Introdução

Nesta fase, o objetivo principal era a implementação de 2 aplicações, um Generator e um Engine. O Generator tem como objetivo gerar um ficheiro com as informações do modelo requerido, isto é, criar um ficheiro com os vértices que constituem o modelo. Já o Engine, tem como objetivo ler um ficheiro de configuração, escrito em *XML*, e com essa informação e com os modelos previamente gerados, desenhá-los utilizando as configurações dadas pelo ficheiro *XML* com o auxílio das ferramentas *OpenGL* e *Glut*. De notar que este trabalho foi desenvolvido na liguagem de programação *C++*.

2 Generator

O Generator através de um conjunto de parâmetros efetua os cálculos necessários para produzir os pontos que constituem a forma geométrica desejada e assim gerar um ficheiro .3d para, posteriormente, ser utilizado no *Engine*.

O generator tem a capacidade de gerar as formas geométricas seguintes:

- Plano: Cria um plano paralelo ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o comprimento das arestas, o número de divisões e o nome do ficheiro 3d resultante.
- Box: Cria uma caixa, centrada na origem do referencial, recebendo como parâmetros o comprimento das arestas, o número de divisões em cada face e o nome do ficheiro 3d resultante.
- Esfera: Cria uma esfera, centrada na origem do referencial, recebendo como parâmetros o raio, as *slices* (divisões na vertical), as *stacks* (divisões na horizontal) e o nome do ficheiro 3d resultante.
- Cone: Cria um cone com a base paralela ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o raio da base, a altura, as slices (divisões na vertical), as stacks (divisões na horizontal) e o nome do ficheiro 3d resultante.
- Cilindro: Cria um cilindro com a base paralela ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o raio da base, a altura, os *slices* e o nome do ficheiro 3d resultante
- Torus: Cria uma figura similar a um "donut"com base paralela ao plano xOz, centrado na origem do referencial, recebendo como parâmetros o raio do tubo do *torus*, o raio do *torus*, o numero de *slices*, o numero de *stacks* e por ultimo o ficheiro onde se vai guardar os vértices gerados.

2.1 Plano

Para construir o plano, optamos por começar a desenhar o ponto no seu canto superior direito, o qual terá os valores de x e z iguais à metade do seu comprimento. Todos os outros pontos do plano serão então calculados a partir deste ponto original, subtraindo a x e z, iteração a iteração. O número de iterações é igual ao número de divisões para cada lado, ou seja $divisions^2$.

A ordem dos pontos é:

- *x,0,z*
- *x,0,z-(length/divisions)*
- *x-(length/divisions),0,z*

e

- *x,0,z*
- *x,0,z-(length/divisions)*
- *x-(length/divisions),0,z-(length/divisions)*

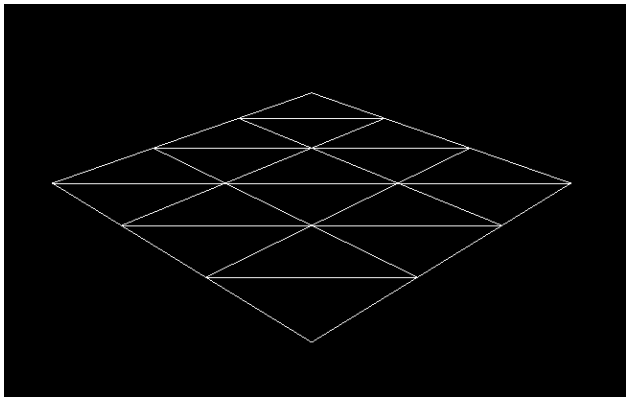

Figura 1: Exemplo de um plano desenhado pelo nosso programa

2.2 Esfera

Parâmetros:

- **r:** raio da esfera a desenhar;
- **slices:** numero de slices na esfera a desenhar;
- **stacks:** numero de stacks na esfera a desenhar;
- **filename:** nome do ficheiro para onde escrever os pontos da esfera.

Execução:

Inicialmente declaramos **alpha**, que contêm o ângulo interior de cada *slice*, e **beta**, que contém o ângulo interior da latitude da primeira *stack*.

Com estes ângulos, podemos calcular as coordenadas dos pontos através dos cálculos de coordenadas esféricas.

No programa, as iterações de i referem-se às *stacks*, enquanto que o j refere-se aos *slices*. Assim, em cada iteração de i são calculados todos os pontos em cada *slice* nessa *stack* usando os contadores como multiplicadores dos ângulos de forma a chegar aos valores das coordenadas necessárias, segundo os cálculos de coordenadas esféricas.

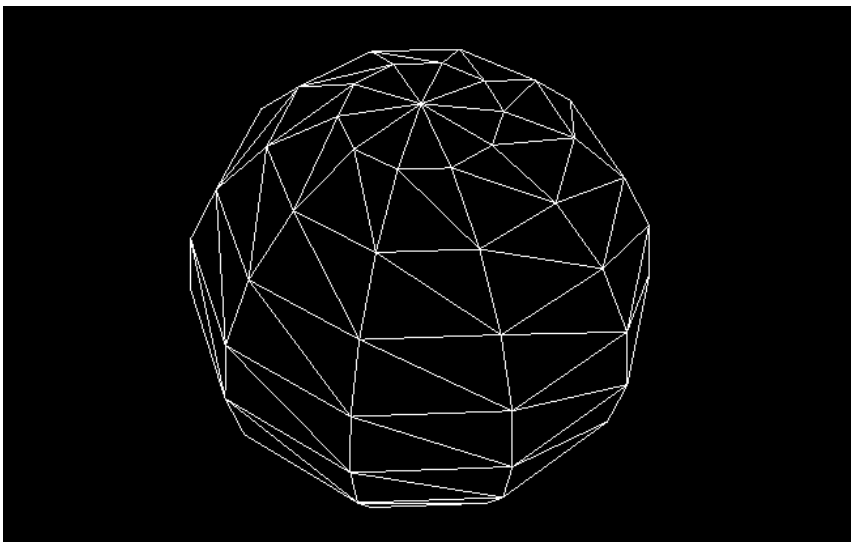

Figura 2: Exemplo de uma esfera desenhada pelo nosso programa

2.3 Cone

Parâmetros:

- **r:** raio da esfera a desenhar;
- **height:** a altura da esfera a desenhar;
- **slices:** numero de slices na esfera a desenhar;
- **stacks:** numero de stacks na esfera a desenhar;
- **filename:** nome do ficheiro para onde escrever os pontos da esfera.

Execução:

A criação do cone está separada em 2 partes: a base e o "corpo"do cone. Para desenhar a base, foi necessário desenhar um "círculo"no plano (XZ) (círculo entre aspas, pois na verdade não é um círculo, é apenas um polígono com lados, em que quantas mais *slices* tiver, mais fica aparentemente igual a um círculo).

Inicialmente declaramos **sliceAngle**, que contêm o ângulo interior de cada *slice*, **stackSize**, que contém a altura de cada *stack* e **rAux** que contém a diferença dos valores dos raios entre *stacks*. Com estes ângulos, podemos calcular as coordenadas dos pontos através dos cálculos de coordenadas esféricas.

Utilizamos um ciclo que, a cada iteração, calcula as coordenadas dos três vértices de um triângulo com um dos vértices no ponto de origem, comum a todos os triângulos. O tamanho de cada iteração será igual ao ângulo de cada *slice*. Para que a base fique no plano XZ, as coordenadas y dos três vértices fica igual a 0. As coordenadas dos outros dois vértices foram calculadas com base nos cálculos de coordenadas circulares.

Para desenhar o corpo, foi necessário utilizar dois ciclos, um para calcular os pontos por cada *stack* e outro para calcular as coordenadas por *slice*. Inicialmente, utilizando o *rAux* e o *stackSize*, calculamos os raios e latitudes de cada 2 *stacks* consecutivos. Para isso recorreu-se a seis pontos por iteração, para que cada *slice* de cada *stack* fosse dividida em dois triângulos. Para sabermos as coordenadas X e Z de cada ponto, recorremos aos cálculos de coordenadas circulares, usando os raios calculados anteriormente. Os Y são conseguidos com as latitudes calculadas anteriormente. O valor de cada iteração do ciclo das *slices* tem, tal como no cálculo da base, o valor do ângulo de uma *slice*, cujo valor é igual ao da base.

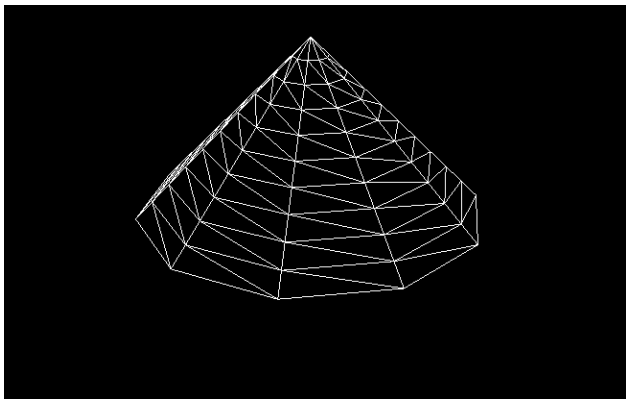

Figura 3: Exemplo de um cone desenhado pelo nosso programa

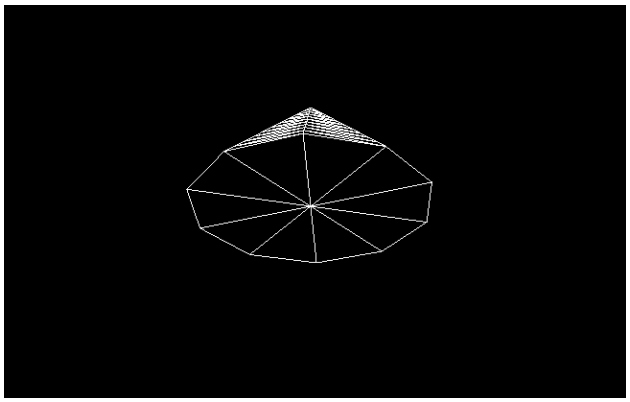

Figura 4: Exemplo um cone, visto por baixo, desenhado pelo nosso programa

2.4 Cubo

Para a criação de uma caixa, centrada na origem, limitamo-nos a criar 3 ciclos *for*, nos quais, em cada um, foram desenhados 2 planos, simétricos entre si pelo centro do referencial, criando assim as 6 faces da caixa.

Os cálculos usados na criação da caixa são iguais às usadas para o plano, repetidas, e atribuídas a coordenadas especificas de forma a garantir que as faces são desenhadas corretamente.

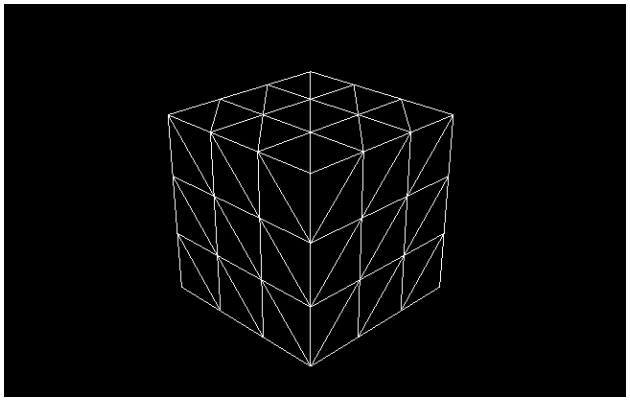

Figura 5: Exemplo de um cubo desenhado pelo nosso programa

2.5 Modelos Adicionais

2.5.1 Cilindro

Para fazer o cilindro, centrado na origem, dividiu-se a sua contrução em três partes, a base do topo, a base de baixo e a parte lateral. Para o topo, de forma a que a face fique voltada para cima, cada *slice* é calculado com os pontos:

- *0,height/2,0*
- *raio * sin(i * angle), height/2, raio * cos(i * angle)*
- *raio * sin((i+1) * angle), height/2, raio * cos((i+1) * angle)*

Para a base de baixo, o cálculo é semelhante, sendo o valor de y o simétrico, e a ordem dos vértices muda, de forma a que a face fique voltada para baixo.

Para a lateral, é necessário calcular um "quadrado"para cada slice, logo é necessário desenhar dois triângulos, com os vértices:

- *(raio * sin(i * angle), height/2, raio * cos(i * angle))*
- *(raio * sin(i * angle), -(height/2), raio * cos(i * angle))*
- *(raio * sin((i+1) * angle), height/2, raio * cos((i+1) * angle))*

e

- *(raio * sin((i+1) * angle), height/2, raio * cos((i+1) * angle))*
- *(raio * sin(i * angle), -(height/2), raio * cos(i * angle))*
- *(raio * sin((i+1) * angle), -(height/2), raio * cos((i+1) * angle))*

Estes vértices foram posicionados desta forma específica de forma a que os triângulos fiquem desenhados do lado de fora do sólido. As coordenadas x e z de cada um dos vértices é calculada com coordenadas polares, de forma a que o conjuntos de todos os *slices* formem uma aproximação ao círculo. A imagem a seguir representa a forma como estas são obtidas. O *angle* é calculado fazendo $2\pi/slices$, de forma a que a soma dos ângulos de todos os *slices* seja igual a 2π. A cada ângulo, é multiplicado (i) ou (i+1), de forma a criar uma transição suave entre os slices adjacentes. Para obter as coordenadas polares, é utilizado o raio do cilindro como a distância radial e o ângulo (i×angle) para determinar a posição ao longo do "círculo".

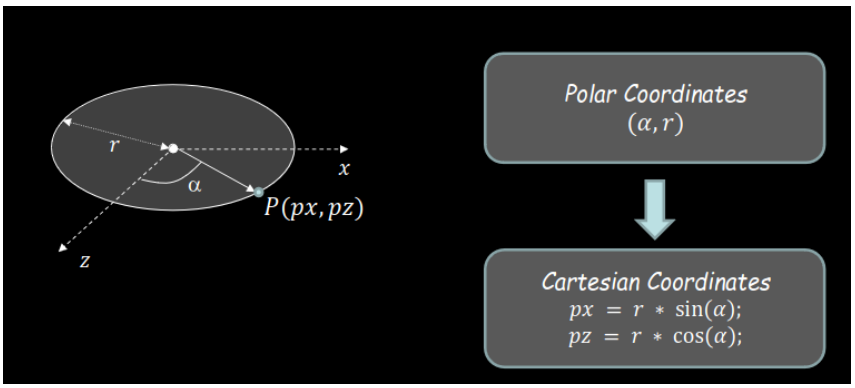

Figura 6: Coordenadas polares

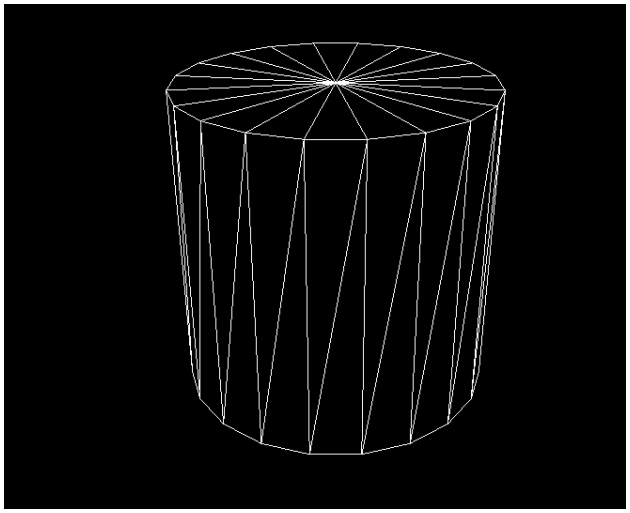

Figura 7: Exemplo de um cilindro desenhado pelo nosso programa

2.5.2 Torus

Um torus é definido pela rotação de um círculo em torno de um eixo que não intersecta o círculo. Os seguintes parâmetros definem como pode ser criado um torus:

- R: Raio do centro do toro até o centro do círculo que gira (distância do centro do toro até o centro do tubo).
- r: Raio do círculo que está a girar (raio do tubo).

- θ: Ângulo que define a posição de um ponto ao longo do círculo maior (circunferência do toro).
- ϕ: Ângulo que define a posição de um ponto ao longo do círculo menor (o círculo que gira).

Para cada ponto na superfície do toro, as suas coordenadas (x, y, z) podem ser calculadas utilizando as seguintes fórmulas:

$$x = (R + r \cdot \cos(\phi)) \cdot \cos(\theta) \tag{1}$$
$$y = (R + r \cdot \cos(\phi)) \cdot \sin(\theta) \tag{2}$$
$$z = r \cdot \sin(\phi) \tag{3}$$

Para gerar a malha do toro:

1. A superfície do toro é dividida em um número de 'slices' (fatias verticais ao longo do círculo maior) e 'stacks' (fatias horizontais ao longo do círculo menor). Estas divisões determinam a resolução do modelo do toro.
2. Para cada par de 'slices' e 'stacks', os pontos correspondentes são calculados utilizando as fórmulas acima. Estes pontos são utilizados para formar triângulos, que são as unidades básicas da malha. Cada 'quadrado' na malha é dividido em dois triângulos para simplificar a renderização.

Este processo gera um conjunto de triângulos que, quando unidos, criam uma representação visualmente contínua e suave da superfície do toro. A escolha dos valores para 'slices' e 'stacks' influencia a precisão e o detalhe do modelo, permitindo um equilíbrio entre performance e qualidade visual.

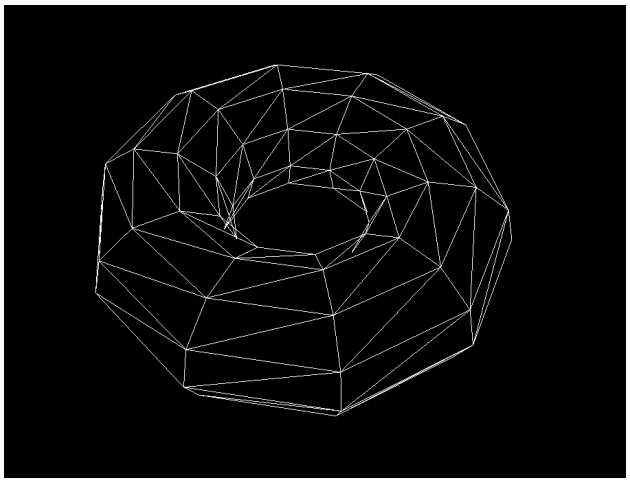

Figura 8: Exemplo de um torus desenhado pelo nosso programa

3 *Engine*

No *Engine*, é onde são utilizadas todas as funções que, tirando partido dos pontos calculados previamente, desenhará a figura através de vetores e recorrendo ás funções do Glut e OpenGL.

Antes de tais operações, o *Engine* lê o ficheiro XML de configuração, recorrendo ao módulo "tinyXML2"e atribui essas configurações à câmara e ao modelo.

3.1 Estruturas de dados

Para isso, tivemos de fazer uma estrutura de dados que permitisse armazenar as informações do xml. A estrutura desenvolvida foi a seguinte:

```
struct World {
    int windowWidth;
    int windowHeight;
    float cameraPosX;
    float cameraPosY;
    float cameraPosZ;
    float cameraLookAtX;
    float cameraLookAtY;
    float cameraLookAtZ;
    float cameraUpX;
    float cameraUpY;
    float cameraUpZ;
    float cameraFov;
    float cameraNear;
    float cameraFar;
    vector<Figure> models;
}xml;

class Figure{
    vector<Ponto> pontos;
    vector<string> modelFiles;
```

Decidimos utilizar estas estruturas de dados pois, além de serem de fácil utilização, a nosso ver, permitirão expandir a solução nas próximas fases.

De momento, o nosso *Engine* aceita inputs de largura e altura da janela, coordenadas cartesianas da posição da câmera, *lookat* (para que ponto esta "olha"), o *up* e, para a projeção, o *fox, near* e *far*. Em relação a modelos aceita que sejam dados como input vários, porém não permite transformações geométricas, logo são todos desenhados na origem. Para finalizar, permite alguns controlos com a câmera durante a execução. 'wasd' para rodar a camera e '+-' para aproximar/afastar a câmara do ponto 'lookat'.

4 Como utilizar o programa

Para começar, enquanto na pasta principal , utilize o **cmake** , seja por *cmake -B build -S .* ou pelo *cmake-gui*, de modo a criar o projeto. Após isso, para criar o executável, ir para a pasta *build* e correr o comando *make*.

Para criar uma das figuras implementadas no módulo generator.py, deverá realizar o comando **./cg generator [figura] [parametros] [nome do ficheiro 3D]** .

Para esta fase apenas estão disponíveis as seguintes variações do comando:

Plano: **./cg plane [tamanho do lado] [numero de divisoes] [nome do ficheiro]**

Esfera: **./cg sphere [raio] [número de slices] [número de stacks] [nome do ficheiro]**

Cone: **./cg cone [raio] [altura] [número de slices] [número de stacks] [nome do ficheiro]**

Caixa: **./cg box [tamanho do lado] [número de divisões] [nome do ficheiro]**

Cilindro: **./cg cilinder [raio] [altura] [número de slices] [nome do ficheiro]**

Torus: **./cg torus [raio interno] [raio externo] [número de slices] [número de stacks] [nome do ficheiro]**

Que irá criar um ficheiro **.3d** na pasta **3d_files**.

Crie um ou mais ficheiros **xml** na pasta **xml_files** seguindo o exemplo:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="4" y="4" z="4" />
        <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="plane.3d" />
            <model file="sphere.3d" />
            <model file="torus.3d" />
            <model file="cilinder.3d" />
            <model file="box.3d" />
            <model file="cone.3d" />
        </models>
    </group>
</world>
```

Com pelo menos um ficherio **3d** e um ficheiro **xml** já construidos, utilize o comando: **./cg engine [nome do ficheiro xml]**

5 Conclusão

Em suma, chegando ao período de entrega desta primeira etapa do trabalho prático, comprovamos e aplicamos os conhecimentos lecionados e praticados durante as aulas teóricas e teórico-práticas. Conseguimos também aperfeiçoar o nosso conhecimento sobre *C++*.

A criação do **generator** facultou-nos um maior conhecimento sobre algumas funções já existentes no **glut**. Após isso apenas necessitamos de recorrer ao *conversor* de XML em uma estrutura

de dados para que o *engine* consiga ler as configurações dos modelos.