3ºAno

Computação Gráfica

Fase 1 - Primitivas Gráficas
LEI - 2023/2024

GRUPO 23

Ana Margarida Sousa Pimenta – A100830
Inês Gonzalez Perdigão Marques -
Miguel Tomás Antunes Pinto – A100815
Pedro Miguel Costa Azevedo – A100557

Índice

1 Introdução 2

2 Estruturação do Projeto 3

3 *Generator* 3

3.1 Plano (Plane) **4**

3.2 Caixa (Box) **4**

3.3 Esfera (Sphere) **5**

3.4 Cone **5**

4 *Engine* 6

5 Demonstração 7

5.1 Manual de utilização **7**

6 Resultado dos Testes 7

7 Conclusão 10

1 Introdução

O objetivo desta primeira fase do projeto, reside no processo de desenvolvimento de um sistema que gere ficheiros de modelo que contém informações sobre os vértices para diferentes primitivas gráficas tridimensionais.

Quanto ao sistema e à sua composição, este encontra-se dividido em duas partes, possuindo um gerador independente, que cria ficheiros de modelo com base em parâmetros específicos, e um mecanismo de renderização que lê um ficheiro de configuração *XML* e exibe os modelos correspondentes.

Nesta fase inicial, focamo-nos na geração de vértices para quatro primitivas gráficas, sendo elas o Plano, a Caixa, a Esfera e o Cone.

2 Estrutura do Projeto

Quanto à estrutura do projeto, de modo a permitir uma melhor organização e modularização do código, este foi estruturado de acordo com as seguintes diretorias:

- ***engine***: Contém o código necessário para a visualização *3D* dos modelos. Temos desde o *parseXML* a funções de renderização dos modelos e de *display*.
- ***generator***: Contém o código necessário para o cálculo das coordenadas dos pontos para a representação do plano, da caixa, da esfera e do cone, bem como a criação dos ficheiros *.3D* de forma a serem posteriormente utilizados pelo *Engine*.
- ***tinyXML***: contém o pacote *tinyXML*, auxiliando na leitura de ficheiros *XML*.

3 *Generator*

O gerador é a aplicação responsável por calcular os pontos dos triângulos que possibilitam a construção das diversas primitivas para um ficheiro *.3D*. A estrutura do ficheiro resultante inclui, na primeira linha, o número de pontos presentes no ficheiro. Nas linhas subsequentes, são apresentados os pontos, onde cada linha contém três valores decimais separados por vírgulas. Estes valores representam, respetivamente, as coordenadas *X*, *Y* e *Z*, sendo que, de três em três linhas, é então formado um triângulo.

Para construir os pontos para as diferentes primitivas, é necessário fornecer ao *generator* os seguintes argumentos, consoante a primitiva em causa, no Plano (*length*, *divisions*), na Caixa (*length*, *divisions*), na Esfera (*radius*, *slices*, *stacks*) e no Cone (*radius*, *height*, *slices*, *stacks*).

Struct Vector3: No *generator* definimos também uma estrutura chamada *Vector3* para armazenar coordenadas tridimensionais. A estrutura possui membros **x**, **y**, e **z**, representando as coordenadas nos eixos *x*, *y* e *z*, respectivamente.

void writeToFile: Esta função, recebe um nome de um ficheiro e um vetor de objetos *Vector3*. Abre o ficheiro especificado para escrita e, de seguida, escreve o número total de pontos (vértices) na primeira linha do ficheiro, seguido pelas coordenadas *x*, *y* e *z* de cada ponto em linhas subsequentes.

3.1 *Plane*

void generatePlane: Função que gera as coordenadas para um plano bidimensional em *3D*. Recebe o comprimento do plano (*length*), o número de divisões (*divisions*) e o nome do ficheiro de saída (*filename*).

A *half_length* é a metade do comprimento do plano, usado para centralizar o plano em torno da origem (0, 0, 0). O *step* é o tamanho do passo entre as divisões do plano. Dois *loops* , *i* e *j*, iteram sobre as divisões do plano, gerando coordenadas para dois triângulos em cada iteração (formando quadriláteros). As coordenadas dos vértices são calculadas de acordo com a posição atual no *loop*, ajustadas para centralizar o plano. As coordenadas são adicionadas ao vetor "vertices".

O plano é gerado subdividindo-o em quadriláteros, e cada quadrilátero é formado por dois triângulos. As coordenadas dos vértices são calculadas com base nas variáveis *i*, *j*, *half_length* e *step*. O resultado é um conjunto de coordenadas que representa um plano bidimensional em *3D*, e essas coordenadas são então escritas num ficheiro para uso posterior, por exemplo, em gráficos *3D*.

3.2 *Box*

A função ***void generateBox*** cria as coordenadas para representar um cubo tridimensional dividido em triângulos e, em seguida, escreve essas coordenadas num arquivo.

A função recebe o comprimento total do cubo (*length*), o número de subdivisões para cada face do cubo (*divisions*), e o nome do arquivo onde as coordenadas serão salvas (*filename*).

- ***vertices***: Vetor que armazenará as coordenadas dos vértices do cubo.
- ***half_length***: Metade do comprimento do cubo, usado para centralizar o cubo.
- ***step***: Tamanho do passo entre as subdivisões do cubo.
- ***Loop externo para cada face do cubo (for face)***: Este *loop* itera seis vezes, uma para cada face do cubo.
- ***Dois loops (for i e for j)***: Estes *loops* iteram sobre as subdivisões da face atual.
- ***Switch statement para determinar a face atual***: Dependendo do valor da *face*, as coordenadas dos vértices (*v0*, *v1*, *v2*, *v3*) são calculadas de maneira diferente para representar a face superior, inferior, frontal, traseira, esquerda ou direita do cubo.
- ***Cálculo das coordenadas dos vértices para formar quadrados***: As coordenadas dos vértices são calculadas em função da posição atual nos *loops i* e *j*, ajustadas para centralizar o cubo. Os vértices formam um quadrado para cada parte da subdivisão da face.
- ***Adição das coordenadas ao vetor de vértices***: As coordenadas dos vértices são adicionadas ao vetor *vertices* para formar dois triângulos, que, quando combinados, formam um quadrado.

O uso de *loop* dentro de *loop* e instruções de *switch* ajuda a automatizar o processo de geração de coordenadas para cada face do cubo. O resultado é um conjunto de coordenadas que representam um cubo tridimensional dividido em triângulos, e essas coordenadas são escritas num ficheiro para uso posterior em gráficos *3D*.

3.3 *Sphere*

Este código gera uma esfera tridimensional com base em parâmetros como raio, número de fatias (*slices*) e número de pilhas (*stacks*). A esfera é gerada criando triângulos a partir dos pontos obtidos.

O algoritmo utiliza coordenadas esféricas para calcular os pontos na superfície da esfera. A esfera é dividida em "*stacks*" e "*slices*", representando a latitude e a longitude, respetivamente.

A variável **vertices** é uma lista de vetores *3D* que armazenará os pontos da esfera.

O *loop* externo (***for (int i = 0; i < stacks; ++i)***) itera sobre as pilhas da esfera. O ângulo *phi* varia de *-π/2* a *π/2*, representando a latitude. O *loop* interno (***for (int j = 0; j < slices; ++j)***) itera sobre as fatias da esfera. O ângulo *theta* varia de 0 a *2π*, representando a longitude.

Dentro dos *loops*, são calculados os pontos correspondentes a quatro vértices de dois triângulos consecutivos. Cada triângulo é definido por três pontos na superfície da esfera. Esses pontos são convertidos de coordenadas esféricas para coordenadas cartesianas.

Os vértices calculados são então adicionados ao vetor *vertices*. O *loop* externo cria a esfera completa, e no final, a função retorna o vetor de vértices que representa a sua geometria.

3.4 *Cone*

O cone tridimensional é modelado com base em parâmetros como raio, altura, número de fatias (**slices**) e número de pilhas (*stacks*). A geometria do cone é formada por triângulos, incluindo uma base triangular e lados inclinados.

A primeira parte do código gera os triângulos que compõem a base do cone. Para cada fatia (*slices*), são calculados os pontos correspondentes à base utilizando coordenadas polares. A base é composta por triângulos formados por um vértice central (*v0*) e dois vértices no perímetro do círculo (*v1* e *v2*).

A segunda parte do código gera os triângulos que compõem os lados do cone. Para cada pilha (*stacks*) e para cada fatia (*slices*), são calculados os pontos correspondentes às *stacks*. As coordenadas dos vértices são ajustadas com base na altura da pilha e no raio proporcional à altura da *stack*.

A matemática envolve o uso de coordenadas polares para a base do cone e coordenadas cilíndricas para os lados. As fórmulas envolvem cálculos trigonométricos básicos.

Para a base do cone:

- *v0* é o vértice central na origem *(0, 0, 0)*.
- *v1* e *v2* são os vértices na circunferência da base, calculados usando as fórmulas *x = rcos(theta), y = rsin(theta)* onde *r* é o raio da base e *theta* é o ângulo em torno do centro da base.

Para os lados do cone:

- *v0,v1,v2* e *v3* são os vértices que compõem dois triângulos adjacentes, representando uma fatia da *stack*. As coordenadas são calculadas usando coordenadas cilíndricas, onde *r0* e *r1* são os raios nas alturas correspondentes, *i* é o índice da pilha, e *theta* é o ângulo em torno do cone.

Ambas as partes do código adicionam os vértices calculados ao vetor vertices, que representam a geometria final do cone.

4 *Engine*

O *Engine* é a aplicação responsável por ler a informação no ficheiro de configuração e a partir dos ficheiros *.3D* indicados nesse mesmo ficheiro, desenhar os triângulos que originam as primitivas que pretendemos.

Esse *engine* ajusta as configurações da câmara, define a janela de visualização e exibe os resultados gráficos dos modelos *3D* gerados pelo *generator*.

Nesta parte é também, como referimos previamente, o local onde fazemos a renderização do modelo, feita a partir da função ***void renderModel***. Esta função processa a renderização dos modelos *3D* em *OpenGL* em duas fases.

Primeiro, aplica transformações de rotação e depois desenha o modelo duas vezes, uma como sólido sem cor e outra em modo *wireframe* (linhas). Usamos também técnicas como *glColorMask* e *glPolygonOffset* para gerir a visualização e profundidade, garantindo que as linhas do *wireframe* se sobreponham corretamente ao modelo sólido.

Para além disso, é também neste ficheiro onde são desenhados os eixos, onde processamos o *display* e também onde fazemos o ***parseXML*** recorrendo á biblioteca *tinyxml2*.

Outras funcionalidades adicionadas foram o *reshape*, para gerir a consistência do modelo com aumento da janela, e *specialKeys*, sendo que estas permitem movimentar o objeto *3D* a partir do teclado em torno dos eixos *x* e *y*.

5 Demonstração

5.1 Manual de utilização

De modo a utilizar o nosso sistema, antes de proceder para a visualização, para corrigir um erro de *parse* associado à biblioteca *tinyxml2*, decidimos colocar os seus ficheiros na mesma diretoria que o *xml_part1.xml,* e dentro da pasta *engine*. Após isso, no ficheiro *CMakeLists.txt*, de modo a fazer o a leitura sem problemas, adicionamos à linha *add_executable* o ficheiro do *tinyxml2,* ficando então, ***add_executable(${PROJECT_NAME} main.cpp tinyxml2.cpp)***.

Após isso, basta correr, sem *debug,* e podemos visualizar os modelos que tivermos definidos no *xml_part1.xml*.

Ainda quanto ao *parse*, de modo a alcançar os ficheiros *3D,* estabelecemos um caminho base, que chamamos ***basePath***, que nos indica o caminho até aos ficheiros. No nosso caso, utilizamos ***basePath = "C:/Users/marga/OneDrive/Ambiente de Trabalho/CG/src/generator/build/release/"*** , para testar noutro dispositivo este caminho deve ser adaptado à máquina em questão.

Enquanto visualizamos o modelo, é possível interagir com o mesmo a partir do teclado, através dos seguintes comandos:

← Rotação da câmara para a esquerda

→ Rotação da câmara para a direita

↓ Rotação da câmara para baixo

↑ Rotação da câmara para cima

6 Resultado dos Testes

Neste ponto, demonstramos os resultados de todos os testes disponibilizados na *Blackboard*.

Começamos então por mostrar, na imagem à direita, o resultado do *Teste 1.1*.

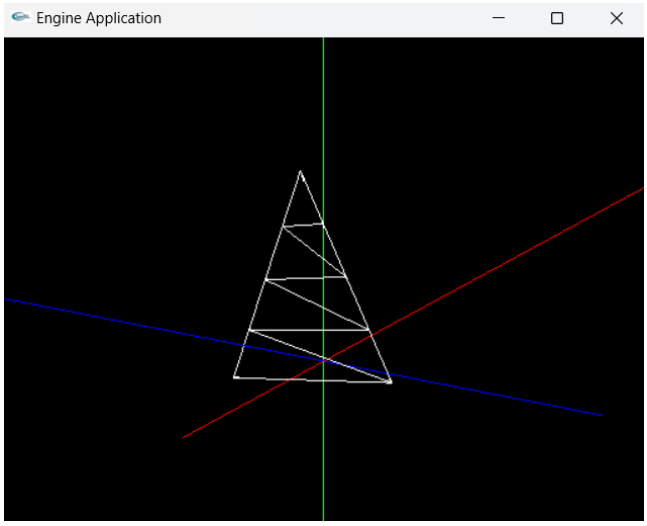

Figura 1 - Teste 1.1

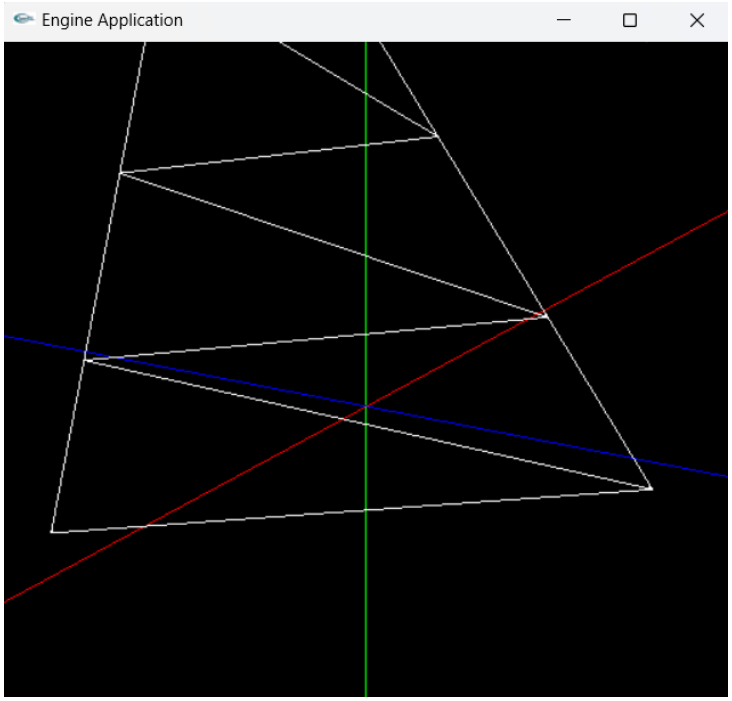

Figura 2 - Teste 1.2

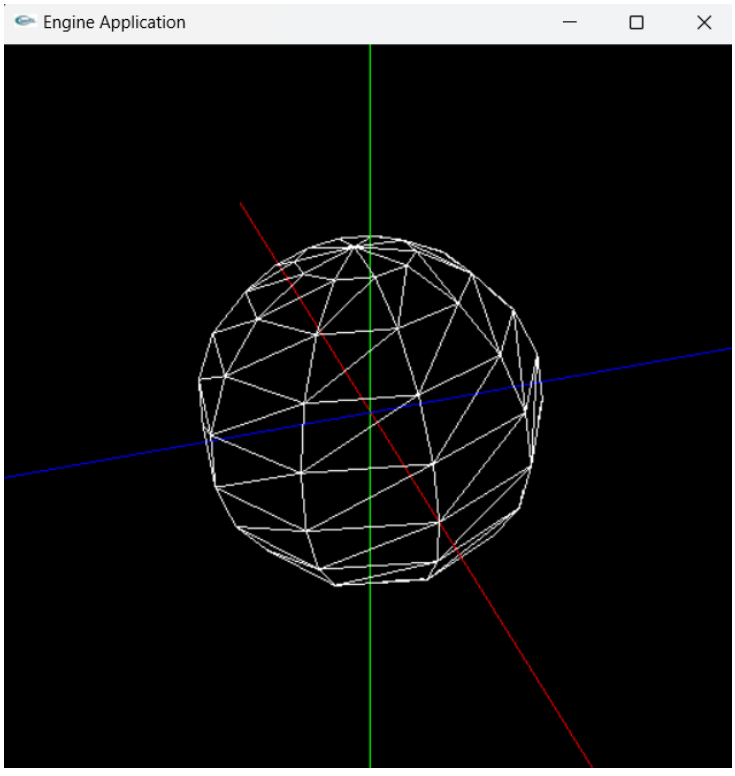

Figura 3 - Teste 1.3

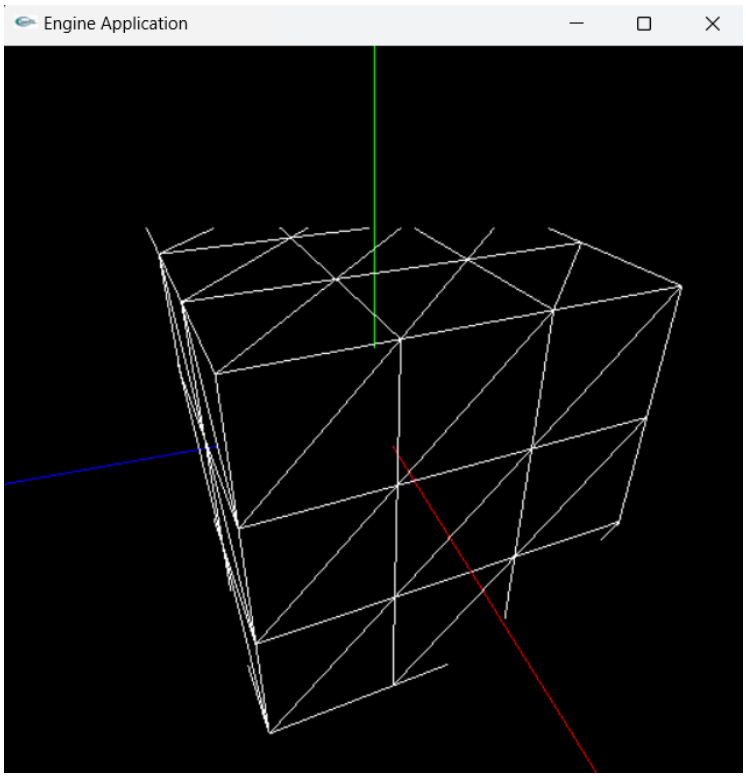

Figura 4 - Teste 1.4

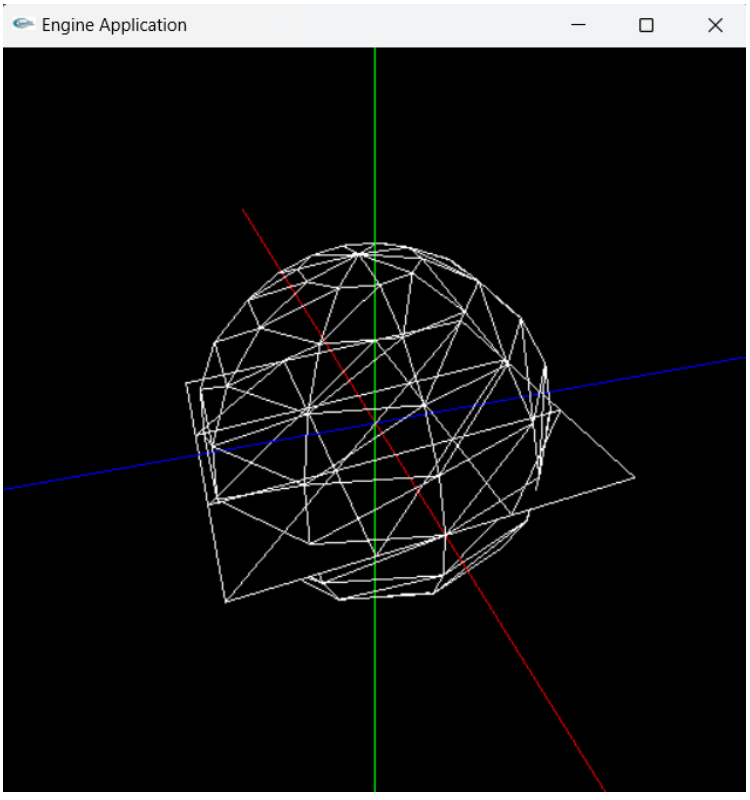

Figura 5 - Teste 1.5

7 Conclusão

Quanto à realização desta primeira fase do projeto, consideramos que foi um processo interessante e também bastante interativo.

Durante o trabalho desenvolvido encontramos algumas dificuldades, como por exemplo a nível do uso da biblioteca *tinyxml2*, pois sem a inclusão da mesma no ficheiro *CMakeLists.txt* esta não estava a ser lida e então os pontos não estavam a ser processados.

Após isso, outro problema que tivemos foi em ocultar as linhas de trás dos modelos. Este foi também uma dificuldade que levou a muita pesquisa pois estávamos a ter alguns problemas com os comandos básicos.

No entanto, conseguimos corrigir esses problemas e, consideramos que desenvolvemos um bom trabalho nesta primeira fase.

Em suma, gostamos de realizar este projeto e aprendemos bastante no seu processo de desenvolvimento, sobretudo com as dificuldades encontradas pois obrigaram-nos a procurar mais informação. Consideramos que cumprimos todos os objetivos com sucesso.