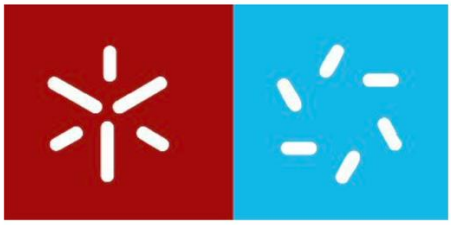

8 DE MARÇO DE 2024

TRABALHO PRÁTICO COMPUTAÇÃO GRÁFICA 2023/2024

LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO
UNIVERSIDADE DO MINHO

GONÇALO EMANUEL FERREIRA MAGALHÃES A100084
EDUARDO ANDRÉ SILVA CUNHA A98980

CONTEÚDO

1. Introdução 2
2. Generator 3
   - 2.1. Definição 3
   - 2.2. Execução do Programa 3
   - 2.3. Código 4
     - 2.3.1. Plano 4
     - 2.3.2. Box 4
     - 2.3.3. Cone 5
     - 2.3.4. Esfera 6
3. Engine 7
   - 3.1. Definição 7
   - 3.2. Código 7
     - 3.2.1. “RemoverLinhasComentario” 7
     - 3.2.2. “ReadFile” 7
     - 3.2.3. “Changesize e RenderScene” 7
     - 3.2.4. “Parser_xml” 8
     - 3.2.5. “Main” 8
4. Testes/resultados 9

1. Introdução

Na primeira etapa do trabalho prático de Computação Gráfica, o foco está na implementação inicial de um motor 3D e na criação de modelos gráficos.
Este processo envolve a utilização de dois códigos distintos: um gerador de modelos (generator.cpp) e um motor (engine.cpp).

O gerador é responsável por produzir os vértices necessários para a construção de um modelo específico, e armazená-los num ficheiro ".3d". O código criado permite a criação de diferentes formas geométricas, como planos, caixas, esferas e cones, utilizando parâmetros específicos, tais como dimensões, raio e subdivisões (slices e stacks). Os vértices resultantes são então guardados num ficheiro, juntamente com informação adicional.

Relativamente, ao motor (“engine”) temos um código desenvolvido para interpretar um arquivo “XML” que especifica diversos parâmetros como o posicionamento da câmara e outros detalhes relevantes, incluindo uma referência a um ficheiro ".3d" gerado pelo "generator.cpp". Com base nessas especificações, o motor é capaz de renderizar os modelos conforme descritos no arquivo “XML”.

2. Generator

2.1. Definição

O principal objetivo do "generator.cpp" é criar um ficheiro de texto que contenha as coordenadas dos vértices da forma geométrica passada como parâmetro no executável. Este código oferece a capacidade de gerar os vértices de diversas figuras distintas, como um cone, uma caixa, um plano ou uma esfera.
Posteriormente o ficheiro gerado será utilizado no "engine.cpp" para desenhar efetivamente as figuras.

2.2. Execução do Programa

Optamos começar por explicar como o programa é executado, pois na descrição do código são mencionadas várias variáveis que são parâmetros passados para o executável.

Para executar o programa "generator.cpp", é necessário fornecer como primeiro parâmetro o tipo de figura desejada. Em seguida, são passados valores numéricos que representam características como altura, comprimento, “slices” e/ou “stacks”, os quais variam conforme a figura geométrica escolhida. Por último, é necessário especificar o nome do arquivo que será criado para armazenar os dados da figura gerada.

```
(base) gugafm11@MacBook-Air-de-Goncalo generator % g++ generator.cpp
(base) gugafm11@MacBook-Air-de-Goncalo generator % ./a.out cone 1 2 4 3 cone_1_2_4_3.3d
A Rodar!
(base) gugafm11@MacBook-Air-de-Goncalo generator %
```

Figura 1: Criação do ficheiro “cone_1_2_4_3.3d”

O arquivo resultante será gerado na pasta "../engine/build", simplificando o código "engine.cpp" (que será discutido mais adiante), eliminando a necessidade de referências a caminhos no código futuro.

O arquivo final será composto apenas por pontos, um por linha, podendo conter informações adicionais desde que a linha seja iniciada por "#" como exemplificado abaixo.

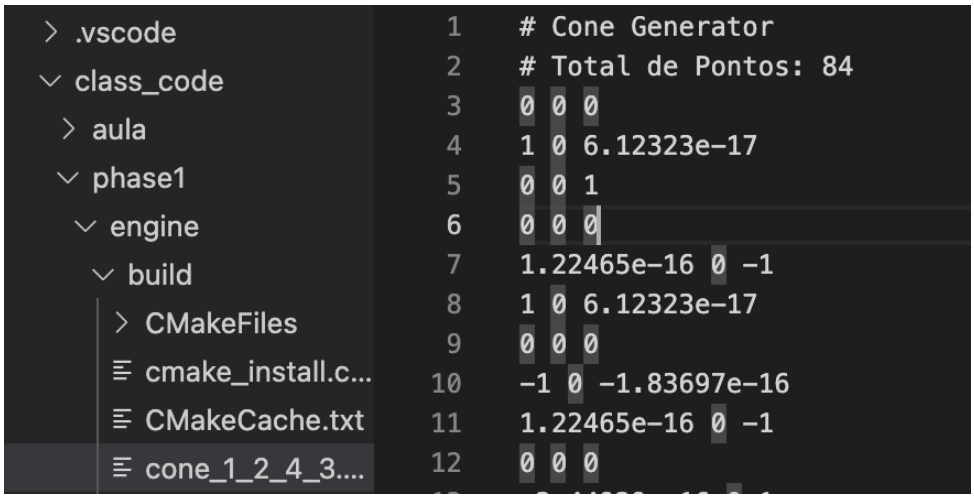

Figura 2: “cone_1_2_4_3.3d”, arquivo gerado pelo generator.cpp

2.3 Código

Na função "main" deste arquivo, os argumentos fornecidos através do terminal são analisados de forma a determinar qual forma geométrica será gerada, invocando a função auxiliar correspondente.
Para auxiliar no processo de escrita para ficheiro foi também definida uma função chamada "write_file", que recebe o nome do arquivo como parâmetro, juntamente com uma “string” de texto. Essa função cria o arquivo com o nome fornecido e insere a informação textual no arquivo.

2.3.1 Plano

Na função encarregada de gerar um plano, uma string chamada "str" é definida para armazenar os vértices dos triângulos necessários para desenhar a forma geométrica.

Variáveis auxiliares são definidas, como "x_inicial", que representa o ponto inicial do eixo x, situado na metade negativa do comprimento, garantindo que o plano esteja centrado na origem do referencial. Além disso, o comprimento de cada lado da subdivisão do plano é calculado, sendo igual ao quociente entre o comprimento total do plano e o número de divisões.

Posto isto, informações adicionais são adicionadas à “string”, sendo sempre iniciadas com "#". Em seguida, dois ciclos "for" são encadeados, iterando sobre o número de divisões fornecidas como parâmetro. Em cada iteração, os valores de “x” e “y” são atualizados para gerar os triângulos necessários para desenhar o plano, levando em consideração a regra da mão direita para garantir a orientação correta dos triângulos.

Finalmente, ao término da função, a função "write_file" é invocada com a “string” resultante.

2.3.2 Box

Para a geração de uma caixa ("box"), definimos uma função denominada de “boxGenerator”. Nesta função, mais uma vez, é definida uma “string” (“str”) para armazenar os vértices dos triângulos necessários para desenhar a forma geométrica.

Variáveis como a altura são armazenadas, representando metade do comprimento do lado, de forma à figura resultante estar centralizada na origem do referencial. O “x_inicial” é definido como a altura negativa, e o comprimento do lado é calculado como o resultado do quociente entre o comprimento do lado e o número de divisões da caixa.

Em seguida, é adicionada informação adicional à "str" para ser incluída no arquivo resultante. Posto isto, dois ciclos "for" são encadeados para determinar os pontos da face superior e inferior da caixa. Em seguida, outros dois ciclos "for" são encadeados para determinar os pontos das faces laterais, seguidos por mais dois ciclos "for" encadeados, para determinar os pontos da face frontal e traseira da caixa. Todos estes ciclos iteram o número de vezes correspondente ao número de divisões fornecido como parâmetro, atualizando os valores de “x”, “y” e/ou “z” a cada iteração.

Novamente, os vértices são colocados estrategicamente para garantir a correta orientação dos triângulos resultantes.

No final da função, é invocada a função "write_file" com a “string” resultante como parâmetro, de forma a guardar todos os pontos gerados no arquivo gerado.

2.3.3 Cone

Na criação de um cone, desenvolvemos uma função que recebe o raio da base, a altura do cone, o número de divisões horizontais (“slices”) e verticais (“stacks”), bem como o nome do arquivo final. Assim como nas outras funções de geração, uma “string” (“str”) é criada para armazenar os pontos gerados, que serão escritos posteriormente num arquivo.

Para calcular os pontos da base do cone, utilizamos um ciclo "for" com as seguintes equações:

$$x1 = raio * sin(i * (2 * M\_PI / slices));$$
$$x2 = raio * sin((i + 1) * (2 * M\_PI / slices));$$
$$z1 = raio * cos(i * (2 * M\_PI / slices));$$
$$z2 = raio * cos((i + 1) * (2 * M\_PI / slices));$$

sendo que o y irá ser sempre igual a 0, pois a base encontra-se no eixo “x0z”.

Para calcular os pontos das laterais do cone, encadeamos dois ciclos "for" com as seguintes equações:

Pontos referentes ao primeiro triângulo:

$$p1 == (raio\_atual * sin(angulo), y, raio\_atual * cos(angulo))$$
$$p2 == (raio\_atual * sin(angulo + (2 * M\_PI / slices)), y, raio\_atual * cos(angulo + (2 * M\_PI / slices)))$$
$$p3 == (nextraio * \sin(angulo), next_y, nextraio * \cos(angulo))$$

Pontos referentes ao segundo triângulo:

$$p4 == (next\_raio * sin(angulo), next\_y, next\_raio * cos(angulo))$$
$$p5 == (raio\_atual * sin(angulo + (2 * M\_PI / slices)), y, raio\_atual * cos(angulo + (2 * M\_PI / slices)))$$
$$p6 == (next\_raio * sin(angulo + (2 * M\_PI / slices)), next\_y, next\_raio * cos(angulo + (2 * M\_PI / slices)))$$

Após cada iteração o angulo é incrementado ($2 * M\_PI / slices$) e os valores de “y” e do “raio” são atualizados para os valores de “next_y” e “next_raio”, sendo:

$$next\_y = y + (altura / stacks);$$
$$next\_raio = raio\_atual - (raio / stacks);$$

Posto isto, é chamada a função "write_file" que trata de guardar todos os pontos gerados no ficheiro.

2.3.4 Esfera

No que diz respeito à geração dos pontos de uma esfera, desenvolvemos uma função que recebe o raio da esfera, o número de divisões verticais (“slices”) e horizontais (“stacks”) da esfera, assim como o nome do ficheiro final. Novamente, criamos uma “string” denominada "str" que irá armazenar toda a informação para posterior escrita em arquivo. Antes de determinar os pontos da esfera, calculamos o tamanho de cada divisão, que corresponde a: $2 * M\_PI \ / \ slices$. Da mesma forma, calculamos o tamanho de cada “stack”, que corresponde a: $M\_PI \ / \ stacks$.

Antes de calcular os pontos, é adicionada informação adicional à “string” e, em seguida, são encadeados dois ciclos "for", um para o número de "stacks" desejadas e outro para o número de "slices" desejadas. Os pontos são definidos pelas seguintes equações:

Pontos referentes ao primeiro triângulo:

$p1 = (raio * \sin(divisão) * \sin(pilha), raio * \cos(pilha), raio * \sin(pilha) * \cos(divisão))$
$p2 = (raio * \sin(pilha + tamanho_{pilha}) * \sin(divisão + tamanho_{divisão}), raio * \cos(pilha + tamanho_{pilha}), raio * \sin(pilha + tamanho_{pilha}) * \cos(divisão + tamanho_{divisão}))$
$p3 = (raio * sin(pilha) * sin(divisão + tamanho\_divisão), raio * cos(pilha), raio * sin(pilha) * cos(divisão + tamanho\_divisão))$

Pontos referentes ao segundo triângulo:

$p4 = (raio * \sin(divisão) * \sin(pilha), raio * \cos(pilha), raio * \sin(pilha) * \cos(divisão))$
$p5 = (raio * sin(pilha + tamanho\_pilha) * sin(divisão), raio * cos(pilha + tamanho\_pilha), raio * sin(pilha + tamanho\_pilha) * cos(divisão))$
$p6 = (raio * sin(pilha + tamanho\_pilha) * sin(divisão + tamanho\_divisão), raio * cos(pilha + tamanho\_pilha), raio * sin(pilha + tamanho\_pilha) * cos(divisão + tamanho\_divisão))$

De salientar que a ordem dos pontos importa para a boa orientação dos triângulos resultantes, seguindo a regra da mão direita.
Após isto, é chamada a função "write_file" que trata de passar toda a informação gerada para o arquivo.

3 Engine

3.1 Definição

A segunda parte do trabalho, o "engine" desempenha um papel crucial, sendo responsável por renderizar os objetos em questão. Ao executar o código, é selecionado um arquivo XML que descreve a cena, incluindo a altura, a posição da câmara, as dimensões da janela, entre outros detalhes. Além disso, o arquivo especifica o nome do arquivo que será aberto para ler os pontos e também inclui, em comentário, o comando para gerar esse arquivo através do código "generator.cpp" anteriormente especificado.
De forma a simplificar a análise e manipulação do "XML", o código faz uso da biblioteca tinyxml2, essa biblioteca oferece funcionalidades que tornam a leitura e a interpretação do arquivo XML mais acessíveis e eficientes.

3.2 Código

No código, definimos diversas variáveis globais que são essenciais para o funcionamento do programa. Entre elas, destacam-se a altura e a largura, que correspondem às dimensões da janela extraídas do arquivo "XML", assim como as coordenadas ("px", "py", "pz") que representam a posição da câmara e outras variáveis relacionadas à câmara. Além disso, criamos um vetor chamado "Pontos" responsável por armazenar todos os pontos lidos do arquivo de entrada, aos quais serão utilizados posteriormente para o desenho dos objetos.

3.2.1 "RemoverLinhasComentario"

Desenvolvemos uma função denominada "removerLinhasComentario" que recebe o arquivo de entrada e elimina todas as linhas iniciadas com "#" que, conforme definido no arquivo "generator.cpp", são linhas que referenciam informações adicionais não pertinentes ao processo atual.

3.2.2 "ReadFile"

Em seguida, criamos a função "readFile", responsável por processar um arquivo gerado pelo "generator.cpp". Primeiramente, ela utiliza a função anterior para remover as linhas de informação adicional. Em seguida, abre o arquivo resultante e, enquanto houver informações disponíveis, armazena esses dados no vetor "Pontos", registando grupos de três pontos consecutivos no final do "array".

3.2.3 "Changesize" e "RenderScene"

O código inclui as funções "changesize" e "renderScene", funções estas bastante usuais em código de geração de imagens e ambas são semelhantes ao código convencional. Porém na função "renderScene", além de definir a câmara com as variáveis globais previamente criadas (que são ajustadas com base no "XML", conforme veremos na próxima função), há também o desenho dos eixos com cores e o desenho da figura requisitada no arquivo. Isso é feito através de um ciclo "for" que itera sobre o vetor "Pontos", desenhando três pontos de cada vez (um triângulo).

3.2.4 “Parser_xml”

Em seguida, definimos a função "parser_xml", que está encarregue de interpretar o arquivo “XML” selecionado, garantindo que os valores especificados dentro dele sejam atribuídos às variáveis globais anteriormente definidas.

Inicialmente, a função abre o arquivo “XML” e, em seguida, procura pela “tag” "<world>". Dentro dessa “tag”, procura por outras “tags” como "<window>", "<camera>" (e dentro desta, mais “tags”) e/ou "<group>". Se encontrar alguma dessas “tags”, interpreta os valores contidos nelas e atribui-os às variáveis globais correspondentes. No caso da “tag” "<group>", localiza o nome do arquivo que contém os pontos a serem desenhados e chama a função "readFile", passando esse arquivo como parâmetro, que tratará de ler o ficheiro e passar os pontos nele contidos para o vetor “Pontos”, de forma a serem desenhados na função “RenderScene”.

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="5" y="-2" z="3" />
        <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_1_2_4_3.3d" />
        </models>
    </group>
</world>
```

Figura 3: Exemplo de um ficheiro XML.

3.2.5 “Main”

Na função "main", começamos por apresentar um menu no terminal, onde perguntamos ao utilizador qual o ficheiro “XML” que deseja testar. A função aguarda então por um “input”. Seguidamente, com base na entrada recebida, é invocada a função "parser_xml" com o ficheiro ".xml" escolhido como argumento. Posteriormente, são feitas chamadas a funções comuns em código de geração de imagens GLUT, destacando-se a função "glutInitWindowSize", que recebe como parâmetros a altura e largura já definidas no ficheiro “XML” selecionado.

4 Resultados obtidos:

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_1_1.xml":

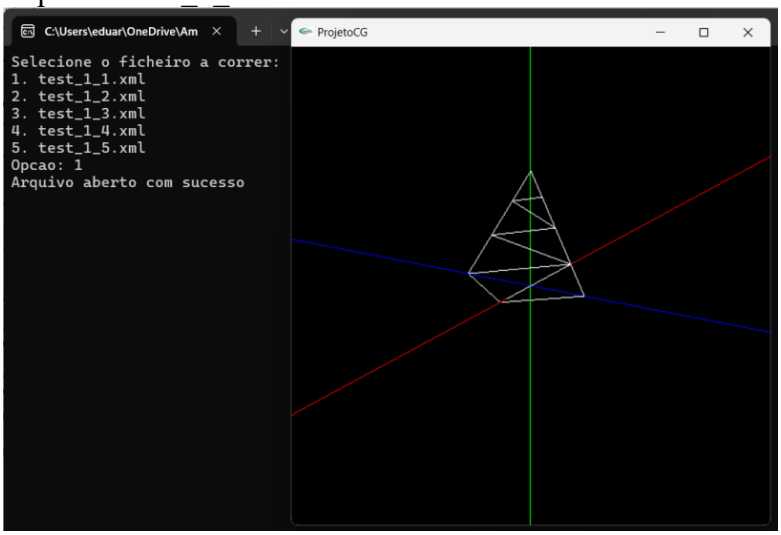

Figura 4: teste_1_1.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_1_2.xml":

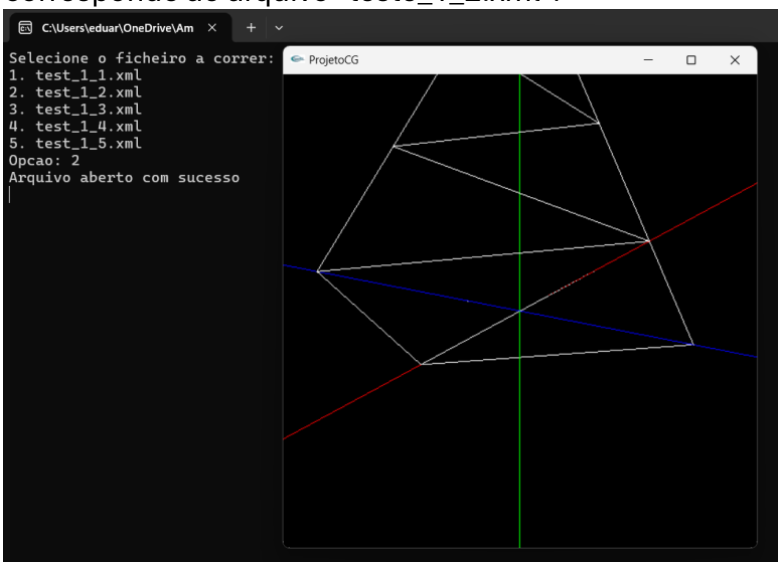

Figura 5: teste_1_2.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_1_3.xml":

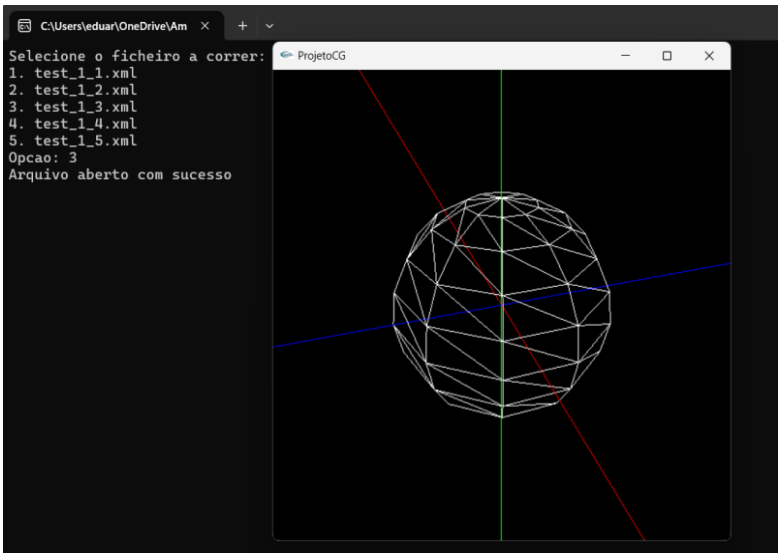

Figura 6: teste_1_3.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_1_4.xml"

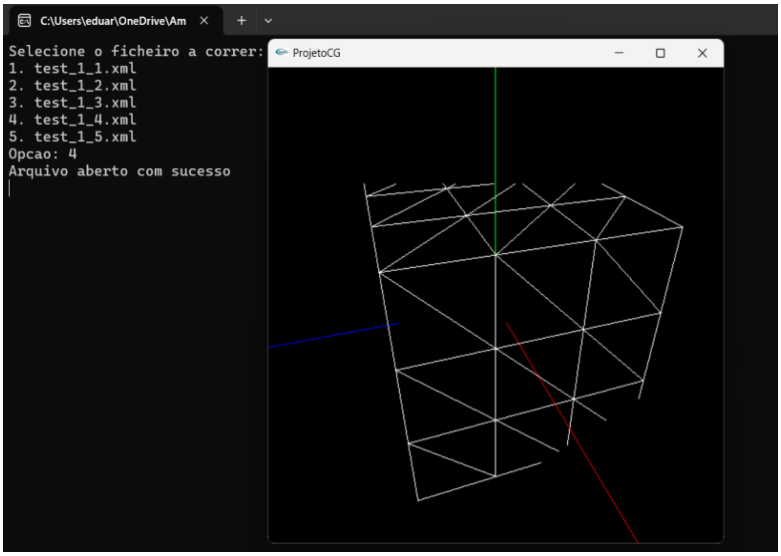

Figura 7: teste_1_4.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_1_5.xml"

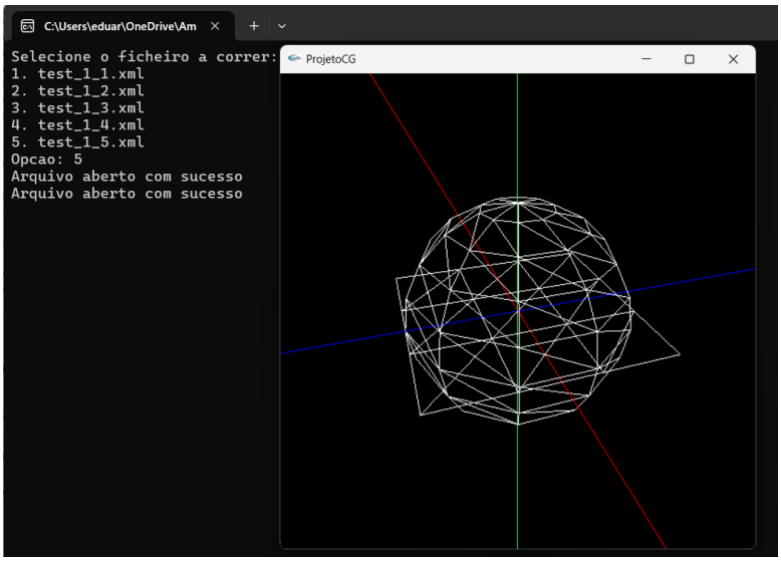

Figura 8: teste_1_5.xml