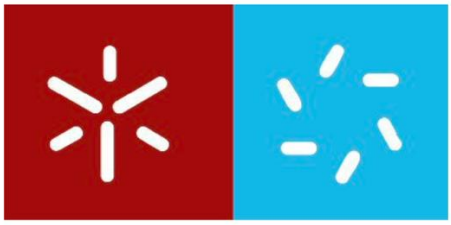

26 DE MAIO DE 2024

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

Na fase final do trabalho prático de Computação Gráfica, é requerido que o nosso ficheiro "Generator.cpp" seja capaz de produzir pontos para a construção de objetos, bem como as respetivas coordenadas normais e coordenadas de textura. No "Engine.cpp" desta fase, o objetivo é interpretar os ficheiros gerados pelo "Generator.cpp" e desenhar os objetos com a respetiva iluminação e texturas.

Primeiramente, no "Generator.cpp", adicionámos funcionalidades para gerar coordenadas normais e de textura para cada ponto, assegurando que cada objeto tem as informações necessárias para ser corretamente iluminado e texturizado. Estas alterações implicaram a reestruturação das funções de geração de pontos, incluindo cálculos adicionais para determinar as normais das superfícies e mapear as texturas de forma apropriada.

No "Engine.cpp", a leitura dos ficheiros gerados pelo "Generator.cpp" foi revista e expandida para interpretar corretamente as novas informações. Agora, o "Engine.cpp" processa não só as posições dos pontos, mas também as suas normais e coordenadas de textura. Com estes dados, implementámos algoritmos de iluminação que utilizam as normais para calcular a incidência de luz sobre os objetos, permitindo um efeito de luz mais realista. Além disso, integrámos o mapeamento de texturas, que aplica imagens aos objetos conforme as coordenadas de textura especificadas, adicionando um nível de detalhe visual significativo.

Deste modo, a estrutura geral do código foi mantida, mas o código sofreu grandes alterações, como veremos mais à frente.

2. Generator

A estrutura do "Engine.cpp" mantém-se semelhante ao ficheiro entregue na fase anterior. No entanto, em cada função de geração de pontos já existente, foi introduzido o cálculo dos vetores normais e das respetivas coordenadas de textura para cada ponto, além dos próprios pontos. No ficheiro de saída, foi também adicionado, inicialmente, o número total de pontos, um valor importante para a interpretação do ficheiro pelo "Engine.cpp".

2.1. planeGenerator

Agora, o código inclui mais três "strings" ("n", "t" e "res"), sendo uma para guardar as coordenadas das normais, outra para as coordenadas de textura e outra para a concatenação das três strings que armazenam os pontos, respetivamente.

No caso do plano, as normais para todos os pontos são (0, 1, 0), e as coordenadas de textura são calculadas multiplicando a posição da grelha pelo número de divisões.

2.2. boxGeneretor

De forma semelhante à função anterior (e a todas as funções alteradas), foram criadas mais três "strings" uma para armazenar os valores das normais, outra para as coordenadas de textura e outra para a concatenação dos pontos, normais e texturas.

Uma caixa tem seis faces, e as normais de cada face são as seguintes:

- Face superior: (0, 1, 0)
- Face inferior: (0, -1, 0)
- Face frontal: (0, 0, 1)
- Face traseira: (0, 0, -1)
- Face lateral direita: (1, 0, 0)
- Face lateral esquerda: (-1, 0, 0)

Os valores das coordenadas de textura (tal como as do plano) são calculados com base numa grelha, sendo que cada ponto corresponde a uma posição da grelha multiplicada pelo número de divisões.

2.3. coneGenerator

Inicialmente, temos a base do cone, onde a normal é (0,−1,0), e as coordenadas de textura são calculadas utilizando as funções “sin” e “cos” a partir do centro (0.5,0.5).

Relativamente à lateral do cone, as normais são calculadas com base na inclinação da superfície em relação ao eixo Y. As coordenadas de textura são calculadas proporcionalmente à posição dos vértices em relação às “slices” e “stacks”. Basicamente, é definida uma grelha onde a subdivisão horizontal é em função do número de “slices” e a subdivisão vertical é em função do número de “stacks”.

2.4. sphereGenerator

Normais da Esfera
As normais são calculadas através coordenadas esféricas: a componente X da normal é “cos(θ)×sin(ϕ)”, a componente Y é “sin(θ)”, e a componente Z é “cos(θ)×cos(ϕ)”, onde “θ” é o ângulo entre o vetor posição e o eixo Y, e “ϕ” é o ângulo horizontal ao redor do eixo Y.

Coordenadas de Textura da Esfera
As coordenadas de textura são mapeadas (novamente) sob a forma de grelha onde cada posição (i,j) é multiplicada pelo número de divisões, sendo horizontalmente relativa às “slices” e verticalmente relativa às “stacks”.

2.5. torusGenerator

O cálculo das normais do toroide foi efetuado com base na informação disponível na Wikipedia (https://pt.wikipedia.org/wiki/Toro_(topologia)), seguindo-se a normalização das mesmas. Para isso, as normais calculadas foram divididas pelo seu comprimento, garantindo assim a normalização.

Quanto às coordenadas de textura, elas seguem um padrão semelhante aos exemplos anteriores. São mapeadas em uma grade, onde cada posição (i,j) é multiplicada pelo número de divisões, sendo as subdivisões horizontais relativas às “slices” e as verticais relativas às “stacks”.

2.6. bezierGenerator

Na função que calcula os pontos com base num "patch" através das equações de "Bezier", não conseguimos calcular as normais nem as coordenadas de textura. Por isso, limitamo-nos a adicionar um "0" (zero) antes dos pontos calculados para que o "engine.cpp", ao interpretar o ficheiro, saiba que não irá receber os vetores das normais nem as coordenadas de textura.

3. Engine

Inicialmente, foram definidos mais dois "GLuint buffers", um para as normais e outro para as texturas. Estes, tal como os buffers definidos para os "VBOs", têm tamanho limitado ao número de objetos definidos, o que consideramos uma pequena limitação do nosso trabalho, dado que existe um número máximo de objetos que podem ser criados.

Para lidar com a iluminação, criámos uma "struct" chamada Cor, que armazena os parâmetros da luz relativos a cada objeto. Posto isto, à struct Figura que já existia foi acrescentado um parâmetro Cor e ainda um "GLuint id_textura" iniciado a 0, que, caso a figura possua textura, armazenará o ID da mesma.

Seguidamente, foi criada uma struct denominada Luz que armazena o tipo da luz e, para cada tipo, um vetor que guarda os respetivos valores. Depois, criámos também um vetor de luzes denominado Iluminação, que se encarrega de armazenar todas as luzes definidas.

```
// Struct que guarda os pontos para o desenho da figura e as transformações que estes vão sofrer
struct figura {
    vector <float> pontos;
    vector <float> normal;      // valores da normal
    vector <float> textura;     // coordenadas de textura
    vector <transformacao> transformacoes;
    cor color;
    GLuint idTextura = 0;       // == 0 Então não existe textura
};

struct luz {
    string tipo;
    float ponto[4];
    float direcional[4];
    float cutoff;
};

// Vetor que guarda todos as figuras
vector <figura> objetos;

// Vetor que guarda as luzes
vector <luz> iluminacao;
```

Figura 1 - Excerto do ficheiro "engine.cpp"

3.1 “readFile”

A função "readFile" agora retorna um tuplo com três vetores: um com os pontos da imagem, outro com os valores das normais e outro com as coordenadas de textura. Dentro desta função, inicialmente, é lida a primeira linha do ficheiro. Se esta for 0, então trata-se de um ficheiro relativo às superfícies de Bezier, para as quais, como vimos anteriormente, não conseguimos calcular os valores das normais e das coordenadas de textura. Nesse caso, são lidos todos os pontos e, a cada ponto, é adicionado o valor (0,1,0) ao vetor das normais e (0,0) ao vetor das coordenadas de textura, apenas para evitar erros na execução do código.
Caso o valor da primeira linha seja diferente de 0, então este representa o número total de pontos no ficheiro. Em seguida, são criados três ciclos: um do 0 ao número de pontos, outro do número de pontos ao dobro do número de pontos, e outro do dobro do número de pontos ao triplo do número de pontos, para interpretar os pontos, os vetores normais e as coordenadas de textura, respetivamente.

3.2 “Paser_group”

A função "parser_group" mantém-se semelhante, no entanto, na leitura do ficheiro agora, coloca na respetiva figura os valores dos pontos, das normais e as coordenadas de textura. Em seguida, verifica se existe um campo denominado "color" e, caso exista, coloca primeiramente o campo "existe" a 1 e armazena os parâmetros da cor na struct Color, que existe dentro da figura.

Depois disto, verifica se existe um campo "texture". Caso haja, abre a textura recebida com as respetivas funções de manuseamento de texturas, como foi visto nas aulas práticas, e guarda em "fig.id_textura" o ID da textura manuseada. Incluímos ainda alguns casos para verificação de erro ao carregar a textura.

3.3 “Parser_xml”

A função "parser_xml" agora recebe diretamente o conteúdo dentro da tag "<world>" do ficheiro, como explicaremos mais à frente. No restante, a função mantém-se idêntica, exceto que agora também reconhece o campo "lights" e guarda os valores das luzes, relativamente ao tipo de luz e aos seus parâmetros (sendo estes tipos: "point", "directional" e "spot"). A luz recebida é armazenada no vetor "iluminacao" previamente definido.

3.4 “RenderScene”

A função "renderscene" foi sujeita a várias alterações. A primeira, mais simples, consiste na necessidade de desligar as luzes durante o desenho dos eixos, para que estes não sejam iluminados. Em seguida, após o desenho dos eixos, é iterado um ciclo sobre o vetor iluminação, configurando as luzes existentes conforme o seu tipo.
Antes do desenho das imagens, caso não exista cor nem textura, então a iluminação é desativada e o modo de desenho é alterado. Além disso, a cor da linha é redefinida como branco.
Após a interpretação das transformações das figuras, se o parâmetro "existe" da cor do respetivo objeto estiver definido como 1, então as cores do objeto são ajustadas de acordo com as suas características quando iluminado.
Por fim, os buffers dos pontos e as coordenadas de textura são vinculados aos respetivos pontos e texturas. Após este processo, a imagem é desenhada.

3.5 “Main”

Na função main, para além de termos alterado os testes que são recebidos como input, interpretamos também o início do ficheiro XML de entrada para receber os valores da altura e largura do ficheiro, de forma a definir a janela. Além disso, foi feita a ativação da textura e a inicialização do GLEW e do IL antes do parser_xml e depois da definição da janela, pois estávamos a ter erros de acesso à memória na parte das texturas.
Em seguida, são configurados os buffers para o uso dos VBOs, sendo estes ativados, gerados e configurados para os pontos, as normais e as coordenadas de textura.
Para finalizar, caso existam luzes definidas, é configurada a luz ambiente e todas as luzes são ativadas e configuradas.

“sistema_solar.xml”

O ficheiro do sistema solar também foi alterado para tirar partido das novas funcionalidades do código. Foi definida uma luz na origem do referencial, representando a "luz do sol".
Para todos os objetos, foram atribuídas cores correspondentes às suas características quando iluminados e foram aplicadas texturas em todos os objetos.

```
<lights>
    <light type="point" posx="0" posy="0" posz="0" />       <!-- Luz é o sol -->
</lights>

<group>
    <group>
        <transform>
            <!-- Sol -->
            <scale x="25" y="25" z="25"/>
            <rotate time="50" x="0" y="1" z="0"/>
        </transform>
        <models>
            <model file="sphere_1_10_10.3d" >
                <texture file="sun.jpg" />
                <color>
                    <diffuse R="230" G="230" B="230" />
                    <ambient R="230" G="230" B="230" />
                    <specular R="0" G="0" B="0" />
                    <emissive R="230" G="230" B="230" />
                    <shininess value="100" />
                </color>
            </model>
        </models>
    </group>
    <group>
```

Figura 2 - Excerto do ficheiro sistema_solar.xml

Execução do ficheiro sistema_solar.xml criado:

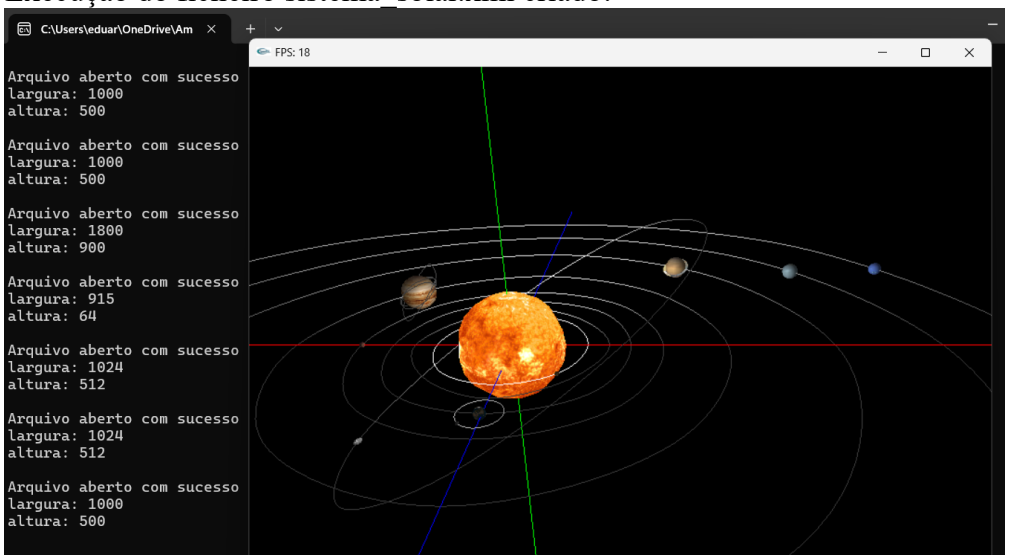

Figura 3 – execução do ficheiro sistema_solar.xml

4 Resultados obtidos:

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_1.xml":

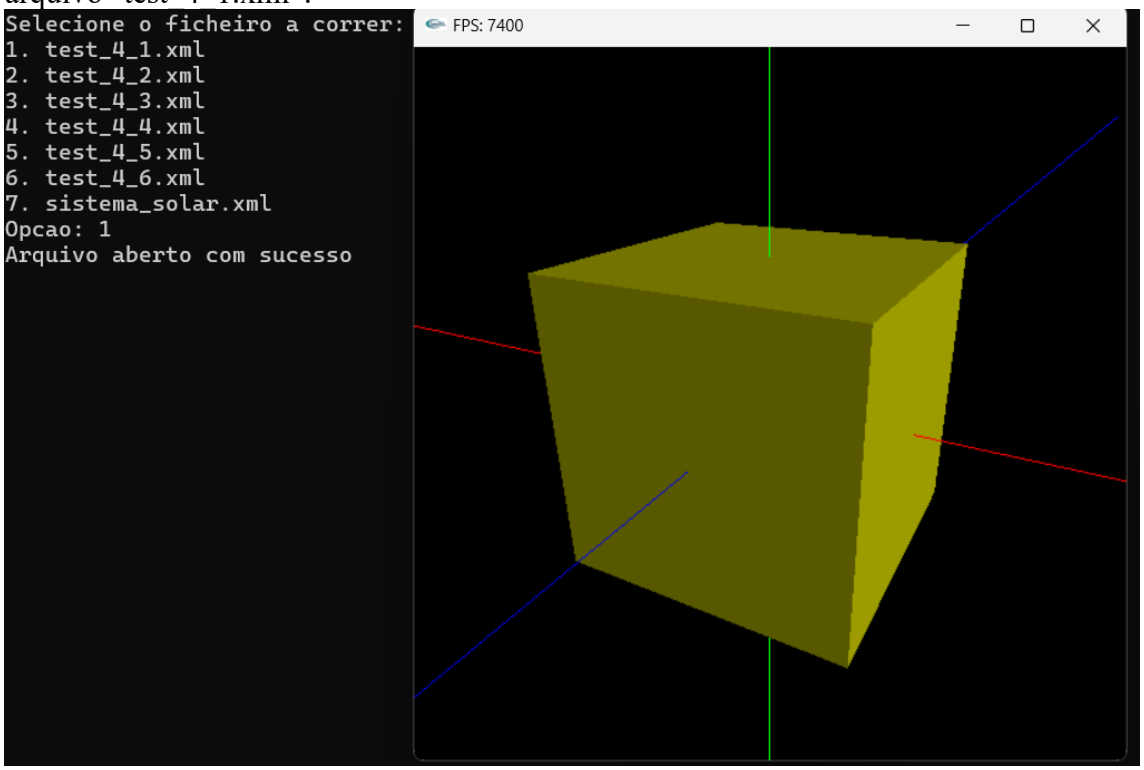

Figura 4: test_4_1.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_2.xml":

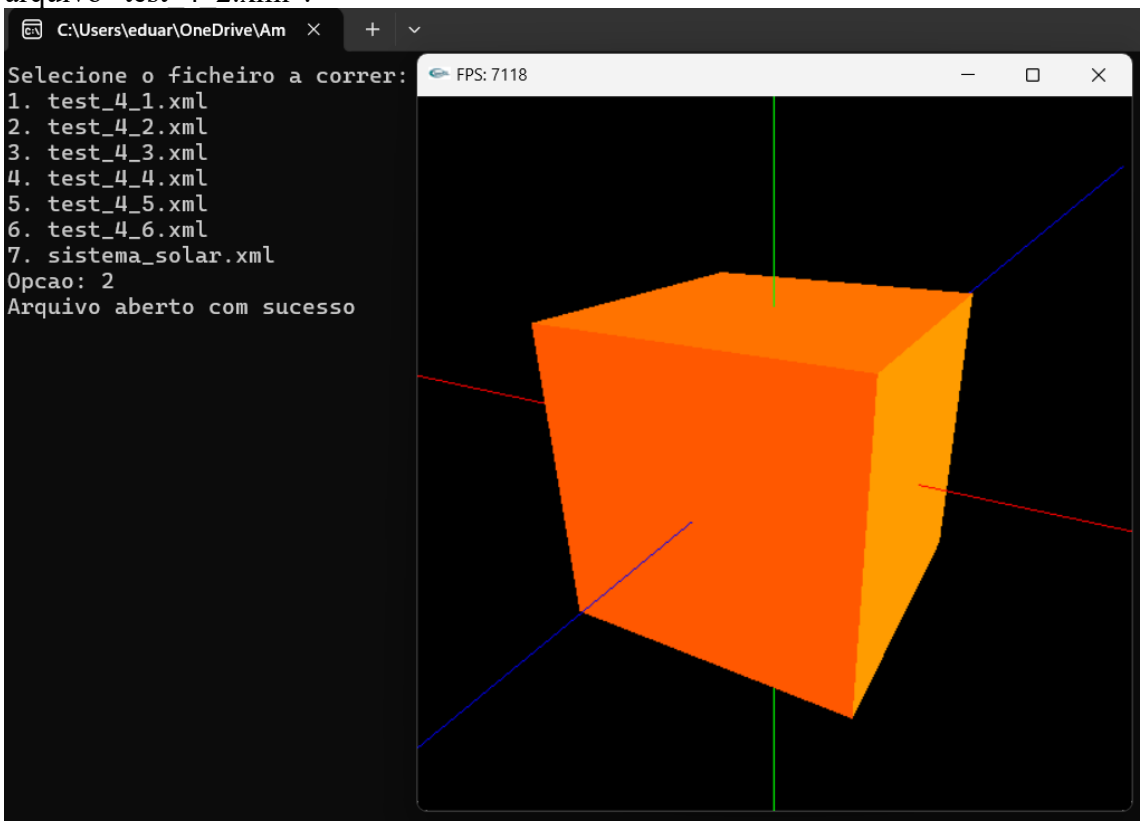

Figura 5: test_4_2.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_3.xml":

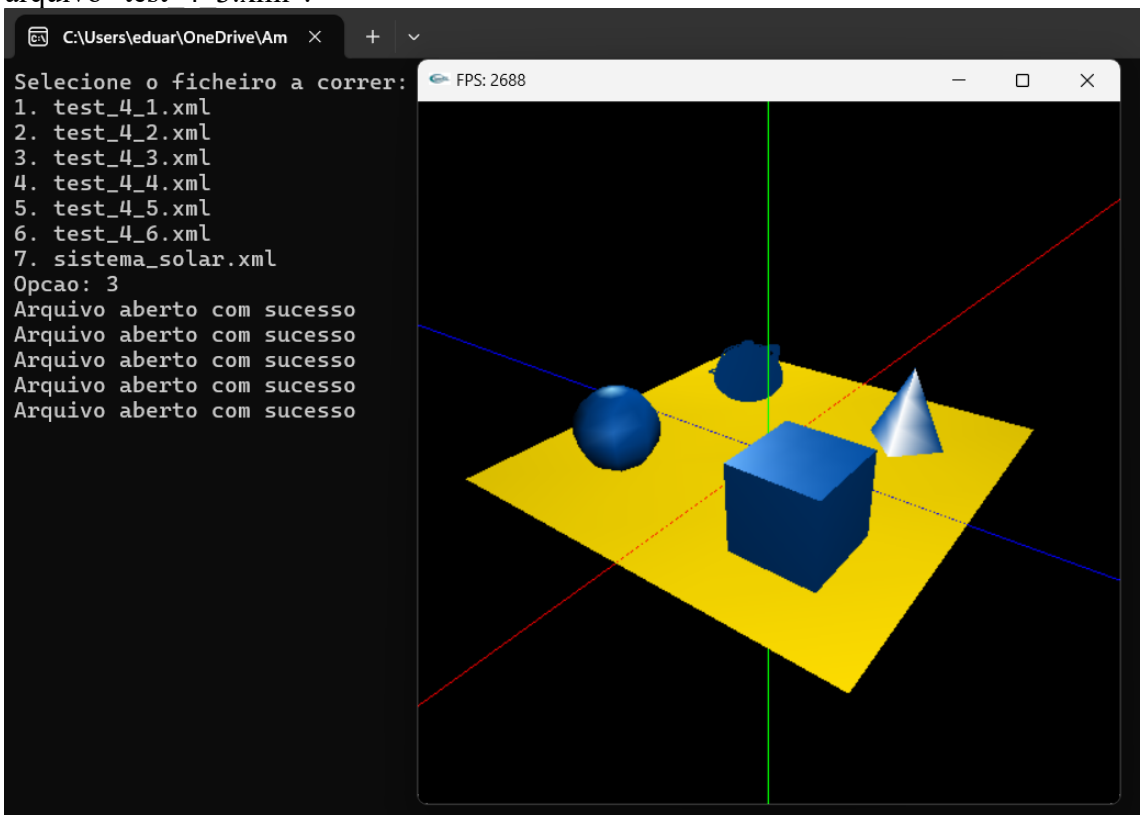

Figura 6: test_4_3.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_4.xml"

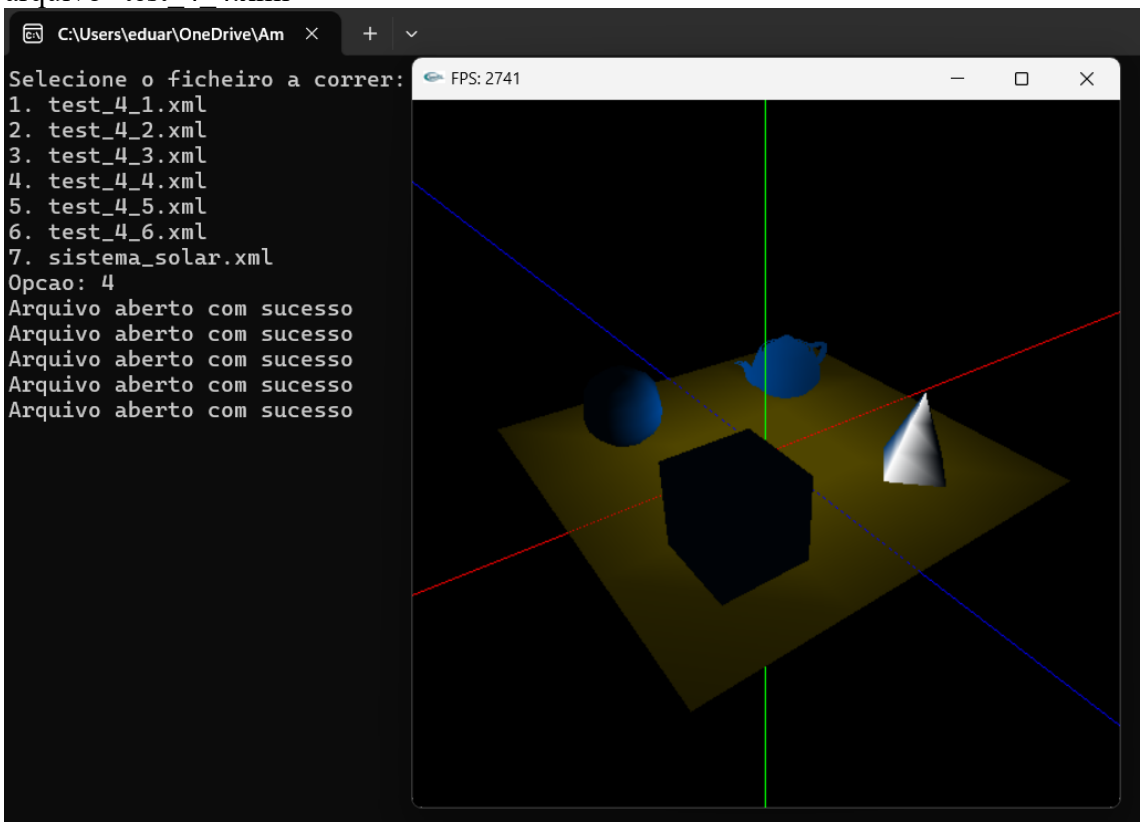

Figura 7: test_4_4.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_5.xml"

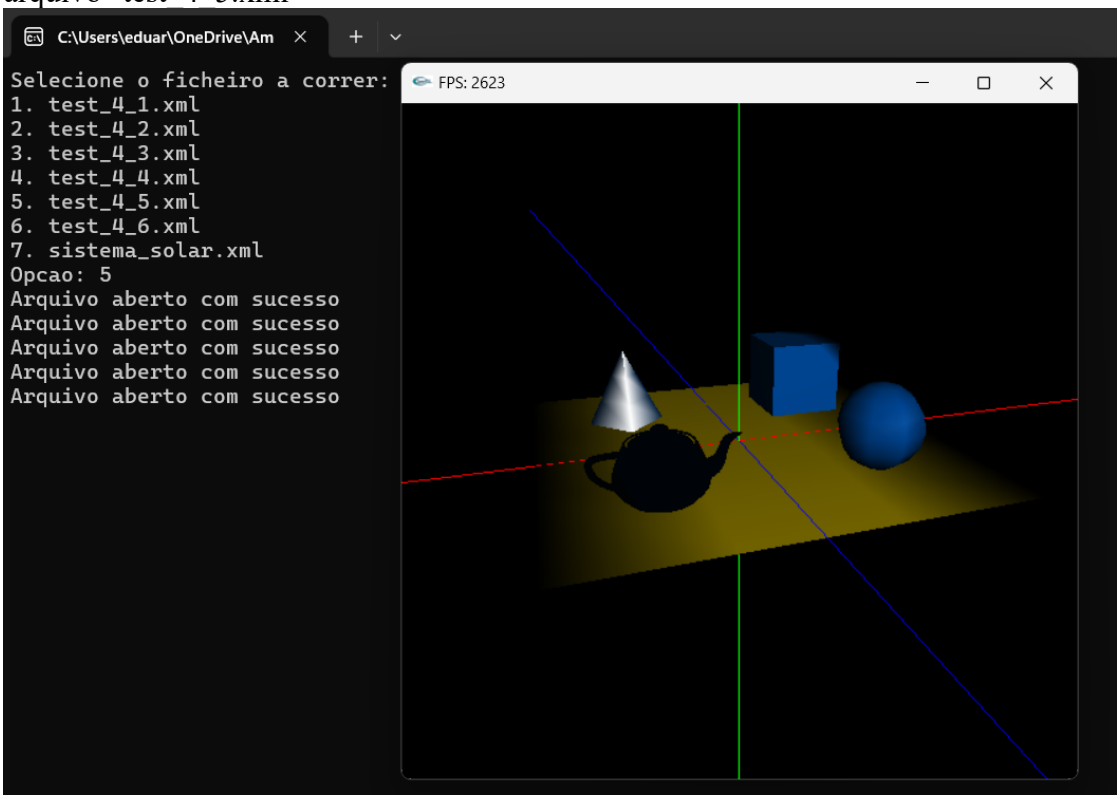

Figura 8: teste_4_5.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "test_4_6.xml"

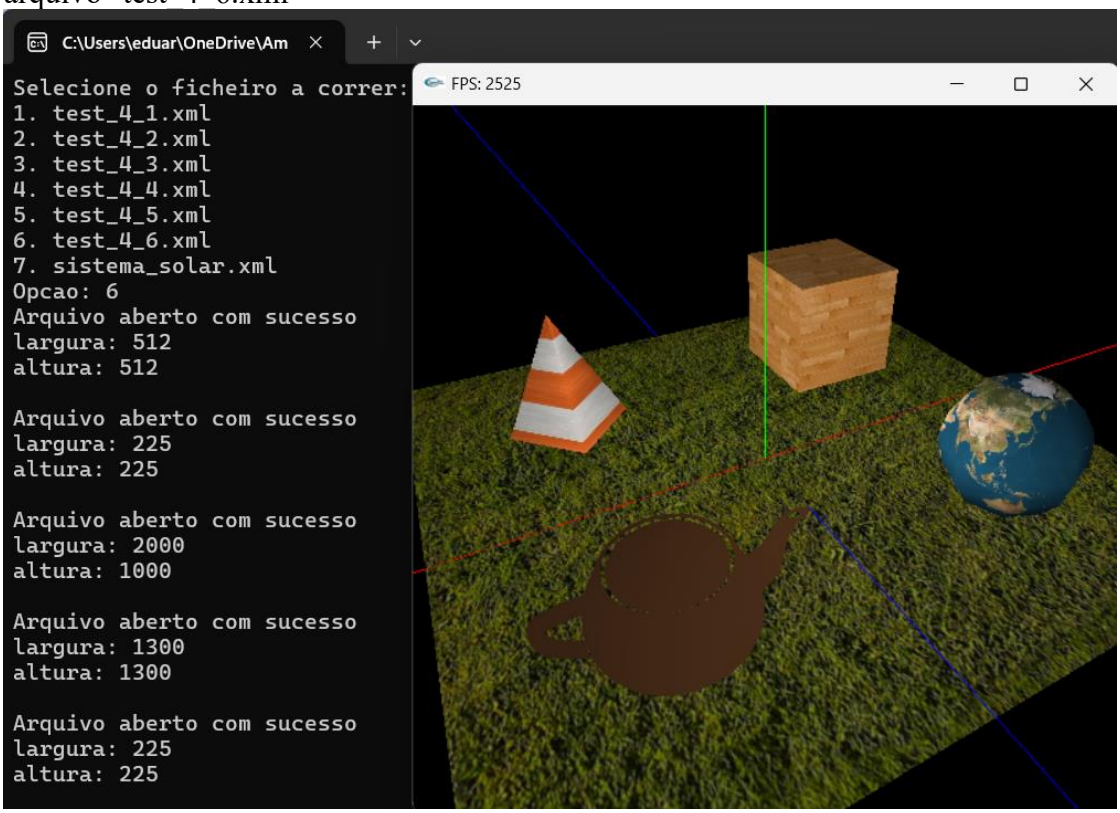

Figura 9: test_4_6.xml

4. Conclusão

Nesta fase, como mencionado anteriormente, o código foi alterado para incluir iluminação e texturas. Apesar de termos completado satisfatoriamente os objetivos para esta etapa, reconhecemos as limitações do trabalho, especialmente pela incapacidade de calcular as normais e coordenadas de textura para as superfícies de “Bezier”. Acreditamos que o facto de o grupo ser composto apenas por dois elementos foi um fator limitante neste aspeto, e agradecemos a extensão do prazo concedida pelo professor, pois sem isso teríamos definitivamente menos objetivos concluídos. Posto isto, consideramos ter um trabalho, de forma geral, completo.

Em suma, a realização deste trabalho foi crucial para a consolidação dos conhecimentos adquiridos nas aulas práticas, mantendo sempre presente a parte teórica lecionada.