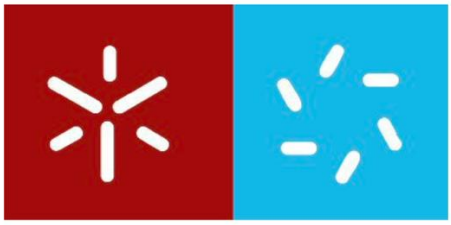

5 DE ABRIL DE 2024

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 segunda fase do trabalho prático de Computação Gráfica, é solicitado que o nosso "Engine" seja capaz de desenhar figuras que possam sofrer transformações (translações, escalas e rotações). Além disso, é solicitado o desenvolvimento de um ficheiro XML que, ao ser executado pelo "engine.cpp", criará uma representação do sistema solar.

Para isso, mantivemos a estrutura da primeira fase, com um ficheiro gerador (“Generator.cpp”) e outro motor (“Engine.cpp”).

O "Generator" continua responsável por produzir os vértices necessários para a construção de um modelo específico e armazená-los num ficheiro ".3d", permitindo a criação de diferentes formas geométricas.

O "Engine" interpreta um arquivo XML selecionado como "input". A partir deste arquivo, são extraídos diversos parâmetros, tais como o posicionamento da câmara e outros detalhes relevantes, juntamente com um ficheiro ".3d" gerado pelo "Generator". Além disso, nesta fase, são também obtidas algumas transformações por meio do arquivo XML. Com base nessas especificações, o motor é capaz de renderizar os modelos de acordo com as descrições contidas no ficheiro selecionado.

2. Generator

2.1. Código

O código do ficheiro "generator.cpp" manteve-se praticamente igual à fase anterior, mantendo a sua função de gerar ficheiros ".3d" que contêm pontos para o desenho de planos, caixas, cones e esferas conforme pedido. Contudo, face à proposta de desenho de um sistema solar como iremos ver a seguir, achamos por bem definir também uma função que trate de desenhar um "torus" ("donut") para o desenho do anel de saturno.

2.1.1. Função “TorusGenerator”

Na função encarregada da geração do "torus", são recebidos como argumentos o raio menor do "torus", o raio maior, o número de fatias ("slices"), o número de pilhas ("stacks") e o nome do ficheiro onde os pontos serão gravados.
É definida uma "string" ("str") que irá armazenar os vértices dos triângulos necessários para desenhar a figura.
As variáveis como raio_menor e raio_maior, recebidas como parâmetros, são redefinidas da seguinte forma:

```
raio_menor = (raio_menor + raio_maior) / 2;
raio_maior = raio_menor – raio maior;
```

O raio_menor é ajustado para a média entre os valores do raio menor e do raio maior. O raio_maior é atualizado para ser a diferença entre o novo raio_menor e o raio maior original. Os valores "ang1" e "ang2" são calculados para representar os incrementos angulares para as "stacks" e "slices" do "torus". Em seguida, é adicionada informação adicional no ficheiro (iniciada por "#") que indica o número de pontos gerados. Após isso, são encadeados dois ciclos "for": um que itera sobre o número de "stacks" e outro sobre o número de "slices" recebidas como parâmetro. Dentro de cada iteração, são calculados os 4 pontos para desenhar os 2 triângulos necessários para a construção do “torus” (2 triângulos por iteração). Os pontos são calculados através das seguintes expressões:

```
x1 = (raio_menor + raio_maior * cos (ang1 * i)) * cos (ang2 * j);
x2 = (raio_menor + raio_maior * cos (ang1 * (i+1))) * cos (ang2 * j);
x3 = (raio_menor+ raio_maior*cos (ang1*(i+1))) * cos (ang2 * (j+1));
x4 = (raio_menor+ raio_maior *cos (ang1 * i)) * cos (ang2 * (j+1));

y1 = raio_maior * sin (ang1 * i);
y2 = raio_maior * sin (ang1 * (i+1));
y3= raio_maior*sin (ang1 * (i+1));
y4 = raio_maior*sin (ang1 * i);

z1 = (raio_menor + raio_maior * cos (ang1 * i)) * sin (ang2 * j);
z2 = (raio_menor+ raio_maior*cos (ang1 * (i+1))) * sin (ang2 * j);
z3 = (raio_menor + raio_maior * cos (ang1 * (i+1))) * sin (ang2 * (j+1));
z4 = (raio_menor + raio_maior * cos (ang1 * i)) * sin (ang2 * (j+1));
```

Para cada triângulo, as coordenadas dos três vértices são adicionadas à “string” “str”.
Após a geração de todos os pontos, é chamada a função “write_file” que trata de guardar a “str” gerada arquivo recebido como argumento.

3. Engine

3.1 Definição

O código do ficheiro "engine.cpp" mantém a sua estrutura principal e função principal de receber um arquivo XML, interpretá-lo e desenhar os elementos conforme especificado no mesmo. No entanto, sofreu algumas alterações para permitir transformações (como translações, escalas e rotações). Além disso, agora também foi modificado de forma a permitir a movimentação da câmara e do ponto de vista.

3.1 Código

Uma mudança significativa inicial foi que, anteriormente, todos os pontos eram armazenados num vetor. Agora, temos um vetor de figuras chamado "objetos", representando uma "struct" que contém um vetor de pontos da figura específica e um vetor de transformações. Essas transformações são também representadas por uma "struct" que inclui o nome da transformação e os seus valores correspondentes.

```
// Struct que armazena as transformações
struct transformacao {
    string nome;
    float x;
    float y;
    float z;
    float angulo; // para rotates
};

// Struct que guarda os pontos para o desenho da figura e as transformações que estes vão sofrer
struct figura {
    vector <float> pontos;
    vector <transformacao> transformacoes;
};

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

Figura 1: "Struct" transformacao e figura e "vector" objetos

"Readfile"

A função "readfile", anteriormente, lia o ficheiro dos pontos e guardava-os no vetor de pontos global. Agora, a função retorna um vetor de pontos. Esta mudança foi importante porque permite armazenar cada figura com o seu respetivo conjunto de pontos.

A função "parser_xml" manteve-se maioritariamente semelhante à definida na fase anterior. Contudo, agora não lida diretamente com o parsing de grupos; em vez disso, chama uma nova função chamada "parser_group" e envia como parâmetros um vetor de transformações vazio e o grupo recém-lido.

O "parser_group", face ao grupo e ao vetor recebidos, começa por guardar o vetor inicial numa outra variável como backup. Em seguida, inicia um loop que continua enquanto existirem grupos. No início de cada iteração do loop, o vetor recebido como parâmetro é igualado ao backup, o que é importante para reiniciar as transformações entre grupos distintos.

Após este passo, o código procura por elementos "transform" dentro do grupo atual. Para cada elemento "transform", verifica o tipo de transformação (por exemplo, "translate", "scale" ou "rotate") e as suas propriedades ("x", "y", "z" e "angle"). As informações de cada transformação são então armazenadas num objeto "trans" da estrutura "transformacao", que é adicionado ao vetor "t". Em seguida, é procurada uma nova transformação caso exista.

Enquanto o código prossegue, procura por elementos "model" dentro do grupo atual. Para cada elemento "model", obtém o nome do respetivo arquivo e lê os seus pontos através da função "readFile". Os pontos lidos são armazenados no objeto "fig" da estrutura "figura", que é então adicionado ao vetor "objetos".
Depois disso, o código busca novamente por um novo grupo "model". Caso não seja encontrado, é feita uma chamada recursiva da função, passando como grupo o próximo grupo (dentro do grupo atual) e o vetor de translações atual, isto é, para o tratamento de subgrupos.

Em seguida, o grupo é atualizado para o seguinte e o ciclo continua a iterar, ou não, terminando assim a função.

```
// Função que trata do parse XML para os grupos
// Vetor t inicialmente encontra-se vazio
void parser_group(tinyxml2::XMLElement* group, vector <transformacao> t) {

    vector <transformacao> transf_conj = t; // Cópia do vetor inicial como backup
    transformacao trans;
    figura fig;

    while(group != NULL){
        t = transf_conj; // t igualado ao vetor inicial a cada iteração

        XMLElement* transform = group->FirstChildElement("transform");
        if(transform){
            XMLElement* tipo = transform->FirstChildElement(); // sem parametros retorna um qualquer
                                                               // pode ser "translate", "scale" ou "rotate"
            while(tipo != NULL){
                string nome = string(tipo->Name());
                trans.nome = nome;
                trans.x = atof(tipo->Attribute("x"));
                trans.y = atof(tipo->Attribute("y"));
                trans.z = atof(tipo->Attribute("z"));

                if (nome == "translate" || nome == "scale") {
                    trans.angulo = 0;   // translate e scale não recebem parametro angulo
                }
                else if (nome == "rotate") {
                    trans.angulo = atof(tipo->Attribute("angle")); // atof -> string para float
                }

                t.push_back(trans); // Armazena a translação lida
                tipo = tipo->NextSiblingElement(); // avança para a proxima transicao -> NextSibling
            }
        }

        XMLElement* models = group->FirstChildElement("models");
        if(models){
            XMLElement* model = models->FirstChildElement("model");
            while(model != NULL){
                char* filename = (char*)model->Attribute("file"); // buscar o nome do ficheiro e converter para (char*)
                fig.pontos = readFile(filename);
                model = model->NextSiblingElement(); // NextSibling
            }
            fig.transformacoes = t; // guarda o vetor das transformacoes na struct
            objetos.push_back(fig); // guarda a figura no vetor das figuras
        }

        // Chamada recursiva para processar os grupos filhos
        parser_group(group->FirstChildElement("group"), t); // passada com as transformacoes atuais pois são comulativas para subgrupos
        group = group->NextSiblingElement("group");
    }
}
```

Figura 2: Função "parser_group"

“RenderScene”

A função "renderScene" sofreu alterações. Primeiramente, o referencial foi alargado. Em vez de ser desenhado entre os valores -100 e 100, foi alterado para desenhar entre -500 e 500, por razões puramente estéticas no desenho do sistema solar.

No entanto, a mudança mais significativa ocorre na forma como as figuras são desenhadas. Agora, são aninhados dois ciclos: um que itera sobre os objetos a serem desenhados e outro sobre as respetivas transformações de cada objeto. Antes de qualquer tipo de desenho, as transformações são aplicadas, caso existam, e por ordem, uma vez que são guardadas com o comando "push_back". Posteriormente, os triângulos são desenhados através dos pontos armazenados para cada objeto. É importante destacar que cada desenho é realizado sobre uma matriz única, já que há um "push" antes das transformações e um "pop" após o desenho, para reiniciar as transformações realizadas entre cada objeto.

```
// Desenho das imagens
for (int i = 0; i < objetos.size(); i++) { // para cada objeto

glPushMatrix();
    for (int j = 0; j < objetos[i].transformacoes.size(); j++) { // sobre cada transformação
        if (objetos[i].transformacoes[j].nome == "translate") {
            glTranslatef(objetos[i].transformacoes[j].x, objetos[i].transformacoes[j].y, objetos[i].transformacoes[j].z);
        }
        if (objetos[i].transformacoes[j].nome == "scale") {
            glScalef(objetos[i].transformacoes[j].x, objetos[i].transformacoes[j].y, objetos[i].transformacoes[j].z);
        }
        if (objetos[i].transformacoes[j].nome == "rotate") {
            glRotatef(objetos[i].transformacoes[j].angulo, objetos[i].transformacoes[j].x, objetos[i].transformacoes[j].y, objetos[i].transformacoes[j].z);
        }
    }

    // após as transformacoes de cada objeto, o seu desenho
    glBegin(GL_TRIANGLES);

        for (int k = 0; k < objetos[i].pontos.size(); k += 9) {
            glColor3f(1.0f, 1.0f, 1.0f); // branco
            glVertex3f(objetos[i].pontos[k], objetos[i].pontos[k + 1], objetos[i].pontos[k + 2]);
            glVertex3f(objetos[i].pontos[k + 3], objetos[i].pontos[k + 4], objetos[i].pontos[k + 5]);
            glVertex3f(objetos[i].pontos[k + 6], objetos[i].pontos[k + 7], objetos[i].pontos[k + 8]);
        }

    glEnd();

glPopMatrix();
}
```

Figura 3: 2 Ciclos “for” alinhados para tratar dos objetos com as transformações

Movimento

Conforme mencionado anteriormente, também definimos funções que alteram a posição da câmara ("movimento") e o ponto para o qual a mesma está a olhar ("movimentospecialkeys").

```
// funcao q move a pos da camara
void movimento(unsigned char key, int x, int y){
    if (key == 'w') {
        px += 1;
    }
    else if(key == 's'){
        px -= 1;
    }
    else if(key == 'n'){
        py += 1;
    }
    else if(key == 'm'){
        py -= 1;
    }
    else if(key == 'd'){
        pz += 1;
    }
    else if(key == 'a'){
        pz -= 1;
    }
}

// funcao que altera o ponto que a camara esta a olhar
void movimentoSpecialKeys(int key, int x, int y) {
    if(key == GLUT_KEY_UP){
        ly += 1;
    }
    else if(key == GLUT_KEY_DOWN){
        ly -= 1;
    }
    else if (key == GLUT_KEY_RIGHT) {
        lx += 1;
    }
    else if(key == GLUT_KEY_LEFT){
        lx -= 1;
    }
}
```

Figura 4: Movimentos para a câmara

Main

A função main mantém-se semelhante, porém atribui de facto funcionalidade às funções de movimento definidas anteriormente. Além disso, os testes, que são redirecionados de acordo com o input recebido, agora envolvem os testes da fase 2.

4. “Sistema_solar.xml”

Conforme especificado no guião, desenvolvemos um ficheiro ".xml" que representa o sistema solar.
Similar aos ficheiros de teste anteriores, começamos por definir o tamanho da página e os atributos da câmara. Em seguida, criamos um elemento <group> que contém vários subgrupos, cada um representando um planeta. Dentro de cada subgrupo, é atribuída uma translação e, se necessário, uma escala. Todos os planetas são representados por esferas cujos pontos estão contidos no ficheiro gerado: "sphere_1_10_10.3d". Para os planetas que possuem luas ou anéis, dentro do subgrupo do respetivo planeta, definimos outro subgrupo contendo translações, escalas e rotações conforme necessário para construir essas figuras adicionais. Para as luas, usamos os pontos gerados do ficheiro "sphere_1_8_8.3d", enquanto para o anel de Saturno utilizamos o ficheiro seguinte: "torus_10_8_15_8.3d".

sistema_solar.xml

```
<world>
    <window width="800" height="600" />

    <camera>
        <position x="50" y="100" z="200" />
        <lookAt x="100" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="100" far="500" />
    </camera>

    <group>
        <group>                                                    <!-- Sol -->
            <transform>
                <scale x="25" y="25" z="25" />
            </transform>
            <models>
                <model file="sphere_1_10_10.3d" />
            </models>
        </group>
        <group>                                                    <!-- Mercurio -->
            <transform>
                <translate x="35" y="0" z="0" />
                <scale x="1" y="1" z="1" />
            </transform>
            <models>
                <model file="sphere_1_10_10.3d" />
            </models>
        </group>
        <group>                                                    <!-- Venus -->
            <transform>
                <translate x="45" y="0" z="0" />
                <scale x="1.5" y="1.5" z="1.5" />
            </transform>
            <models>
                <model file="sphere_1_10_10.3d" />
            </models>
        </group>
        <group>                                                    <!-- Terra -->
            <transform>
                <translate x="60" y="0" z="0" />
                <scale x="2.5" y="2.5" z="2.5" />
            </transform>
            <models>
                <model file="sphere_1_10_10.3d" />
```

Figura 5: Excerto do “sistema_solar.xml”

5. Resultados obtidos:

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_2_1.xml":

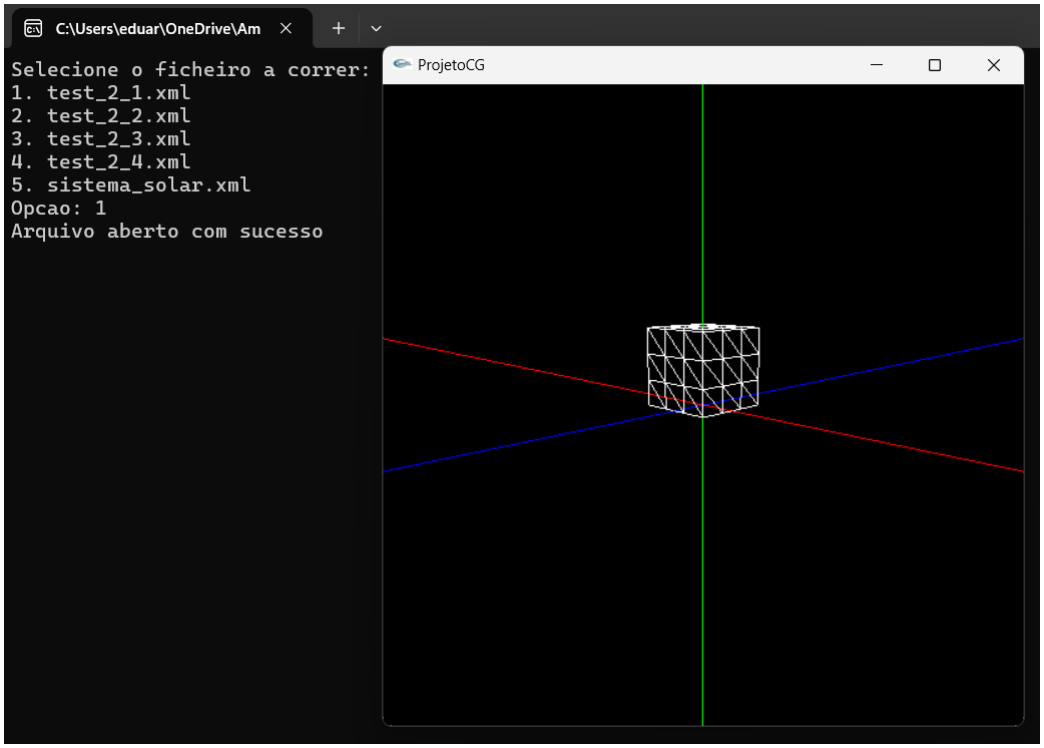

Figura 6: teste_2_1.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_2_2.xml":

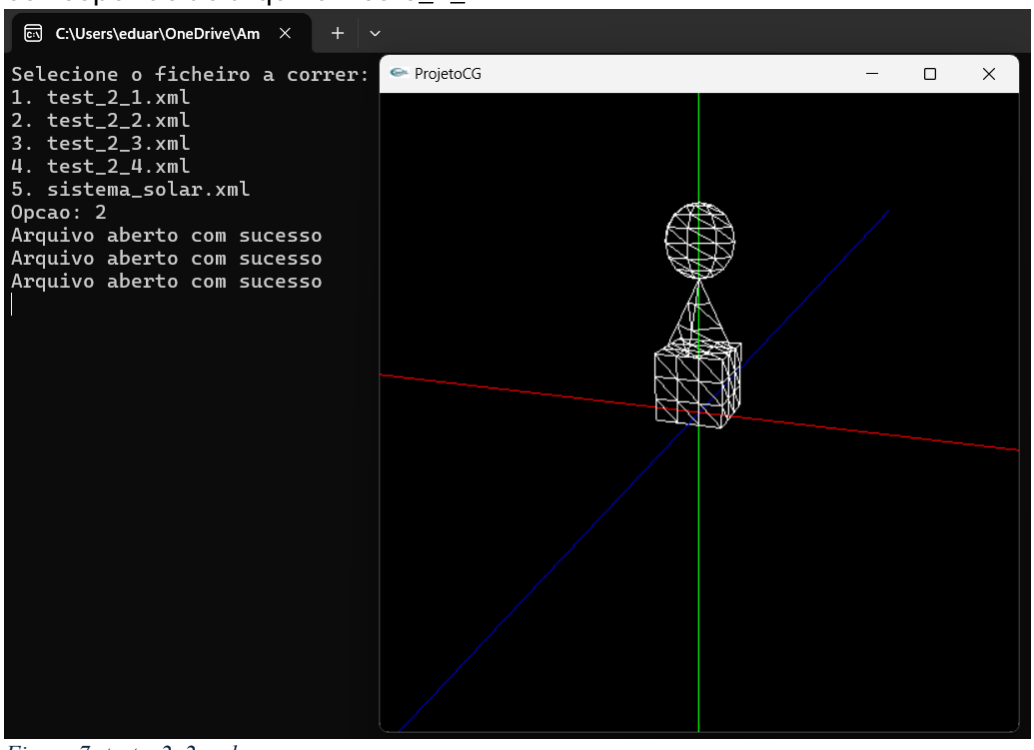

Figura 7: teste_2_2.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_2_3.xml":

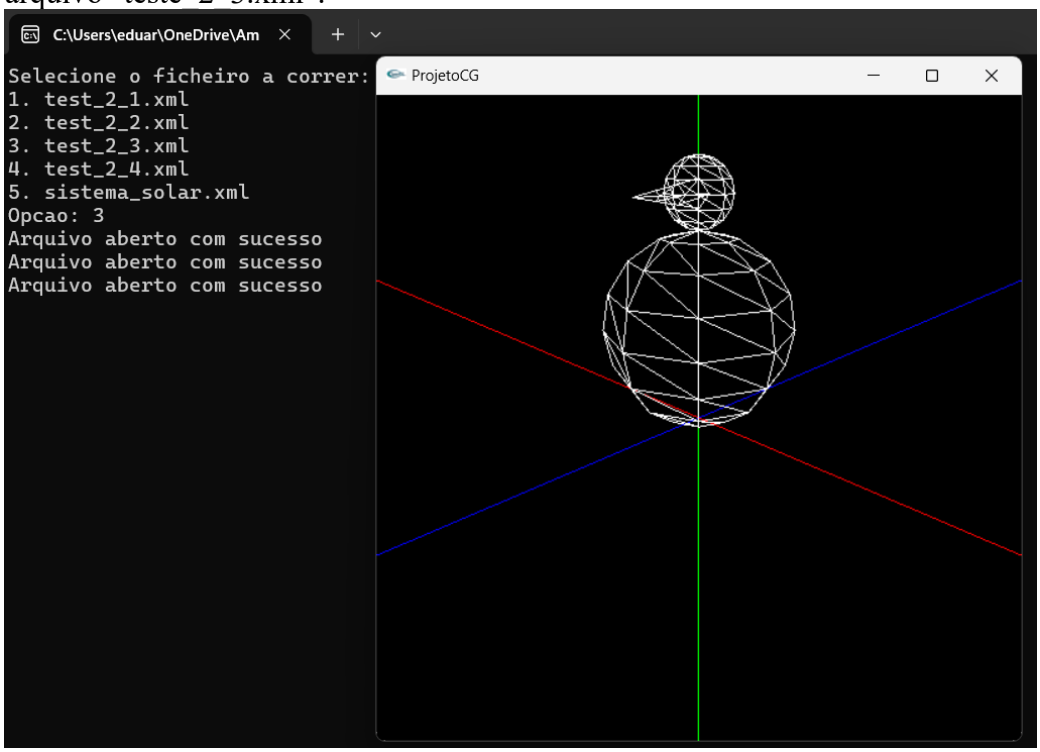

Figura 8: teste_2_3.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_2_4.xml"

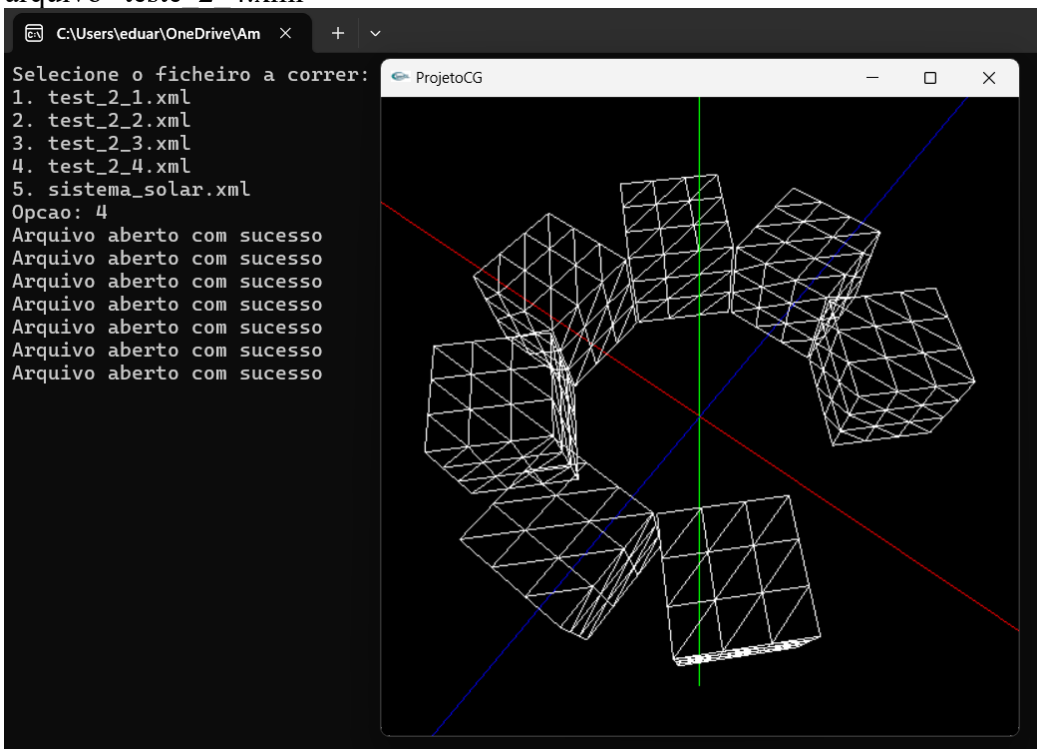

Figura 9: teste_2_4.xml

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

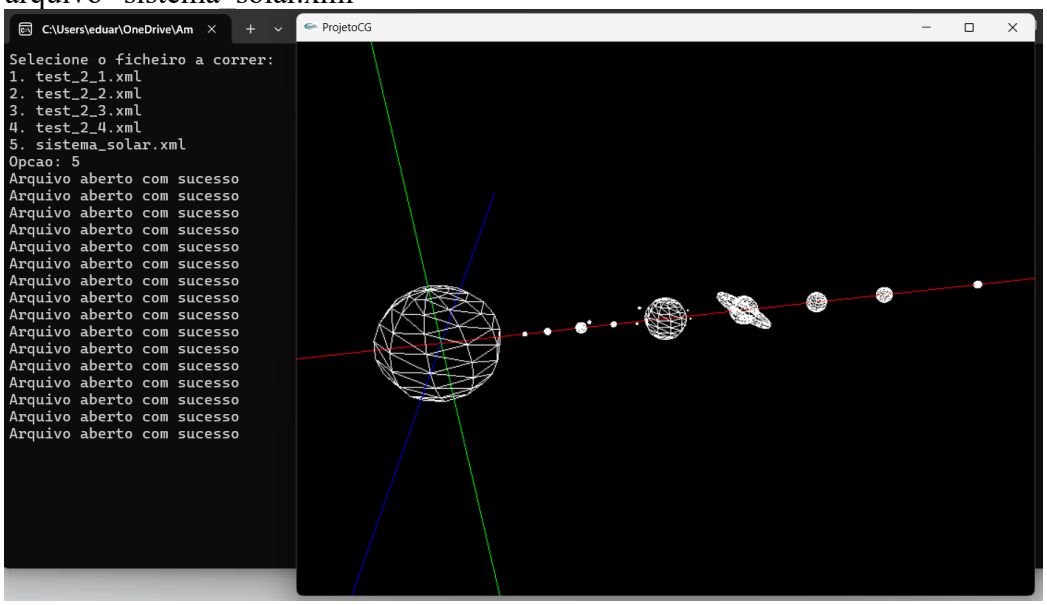

Figura 10: sistema_solar.xml

6. Conclusão

Neste trabalho, implementámos novas funcionalidades ao nosso projeto anterior. O projeto já permitia a geração e o desenho de figuras, mas agora adicionámos a capacidade de aplicar transformações a essas figuras. Além disso, criámos um ficheiro "XML" capaz de desenhar o sistema solar, utilizando as figuras já geradas pelo nosso "generator.cpp".

Acreditamos ter cumprido todos os objetivos propostos nesta fase. No entanto, reconhecemos que o último teste da primeira fase deixou de ser gerado corretamente. Isso ocorre porque são passados dois ficheiros de entrada no mesmo grupo (no arquivo "xml"), e com a nossa implementação apenas o último ficheiro está a ser desenhado.

Este trabalho permitiu-nos aprofundar os conhecimentos adquiridos nas aulas e aplicá-los de forma prática.