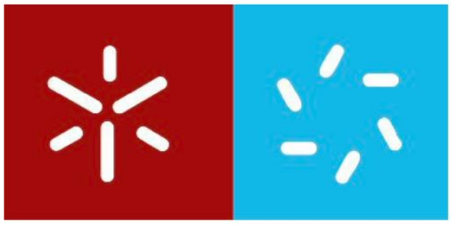

26 DE ABRIL DE 2024

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 terceira fase do trabalho prático de Computação Gráfica, é solicitado que nosso arquivo "Generator.cpp" seja capaz de produzir pontos para a construção de objetos a partir de um “patch” fornecido, com o objetivo de calcular os pontos da superfície de “Bézier” correspondente. No "Engine.cpp" desta fase, o objetivo é lidar com as curvas “Catmull-Rom”.

Com isso, mantivemos todo o código anterior com alguns ajustes e acrescentamos funções para alcançar os objetivos propostos. A estrutura foi mantida com um arquivo para o gerador ("Generator.cpp") e outro para o motor ("Engine.cpp").

O "Generator.cpp" continua responsável por produzir os vértices necessários para a construção de um modelo específico e armazená-los num arquivo ".3d", além de agora possuir a capacidade de criar as superfícies de Bézier conforme mencionado.

O "Engine" interpreta um arquivo XML selecionado como entrada. A partir deste arquivo, são extraídos diversos parâmetros, como o posicionamento da câmera e outros detalhes relevantes, juntamente com um arquivo ".3d" gerado pelo "Generator". Além disso, nesta fase, temos a capacidade de manipular curvas “Catmull-Rom” e para além disso contamos com o uso “VBO’s” para aumentar a eficiência do código.

2. CMakeLists.txt

Inicialmente, atualizamos nosso arquivo “CMakeLists.txt” relativo ao “engine.cpp” de forma a incluir a biblioteca “glew.h”, necessária para o uso de “VBO’s”.

```
if(WIN32)

    message(STATUS "Toolkits_DIR set to: " ${TOOLKITS_FOLDER})
    set(TOOLKITS_FOLDER "" CACHE PATH "Path to Toolkits folder")

    if(NOT EXISTS "${TOOLKITS_FOLDER}/glut/GL/glut.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glut/glut32.lib")
        message(FATAL_ERROR "GLUT not found")
    endif(NOT EXISTS "${TOOLKITS_FOLDER}/glut/GL/glut.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glut/glut32.lib")

    if(NOT EXISTS "${TOOLKITS_FOLDER}/glew/GL/glew.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glew/glew32.lib")
        message(FATAL_ERROR "GLEW not found")
    endif(NOT EXISTS "${TOOLKITS_FOLDER}/glew/GL/glew.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glew/glew32.lib")

    include_directories(${TOOLKITS_FOLDER}/glut ${TOOLKITS_FOLDER}/glew)
    target_link_libraries(${PROJECT_NAME} ${OPENGL_LIBRARIES}
                                                                ${TOOLKITS_FOLDER}/glut/glut32.lib
                                                                ${TOOLKITS_FOLDER}/glew/glew32.lib )

    if(EXISTS "${TOOLKITS_FOLDER}/glut/glut32.dll" AND EXISTS "${TOOLKITS_FOLDER}/glew/glew32.dll")
        file(COPY ${TOOLKITS_FOLDER}/glut/glut32.dll DESTINATION ${CMAKE_BINARY_DIR})
        file(COPY ${TOOLKITS_FOLDER}/glew/glew32.dll DESTINATION ${CMAKE_BINARY_DIR})
    endif(EXISTS "${TOOLKITS_FOLDER}/glut/glut32.dll" AND EXISTS "${TOOLKITS_FOLDER}/glew/glew32.dll")

    set_property(DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR} PROPERTY VS_STARTUP_PROJECT ${PROJECT_NAME})
```

3. Generator

O código permaneceu basicamente o mesmo, mas foi complementado para atender aos desafios propostos nesta fase.
Foi introduzida uma estrutura chamada "Vertices", responsável por armazenar um vetor de "floats". Essa estrutura foi criada com o propósito de armazenar os pontos de controlo necessários para o cálculo dos pontos de uma dada curva. Além disso, definimos uma função chamada "bezieGenerator" para calcular os pontos da superfície de "Bezie" a partir de um “patch” fornecido.

3.1. Curvas de Bezier

Com base nessas considerações, definimos a função "bezieGenerator(std::string patch, int subdivs, std::string file_name)". Esta função recebe um “patch”, o número de subdivisões a serem calculadas e o nome do arquivo onde os pontos gerados serão armazenados.

Primeiramente, o arquivo relativo ao "patch" recebido é aberto. Tendo em consideração a estrutura dos arquivo ".patch" fornecida pelo professor, é lida a primeira linha, que representa o número de “patches” do arquivo. Em seguida, são lidas todas as linhas seguintes correspondentes aos “patches”, que contêm 16 valores cada. Estes valores são armazenados na matriz "patch_indices".
Após os “patches”, a próxima linha representa o número de pontos de controlo. As linhas seguintes contêm os pontos de controlo, que são armazenados no vetor "pontos de controle".

Com toda a informação do “patch” armazenada em variáveis, é iniciado o cálculo dos pontos da superfície de “Bézier”. Este cálculo é complexo, envolvendo a variável "t" para representar a posição do ponto ao longo do segmento de reta entre dois pontos de controle,

assim como o "step" para representar o incremento de "t" entre cada passo, dependendo do parâmetro "subdivs" fornecido.
Durante o cálculo, são criadas duas matrizes tridimensionais, "ctrl" para armazenar os pontos de controle ao longo da curva e "ret" para armazenar os pontos da superfície de "Bézier". Um “loop” é então executado para cada "patch". Dentro desse "loop", primeiramente são armazenados 4 pontos consecutivos do "patch" e calculados os valores da curva gerada por esses 4 pontos, com a devida atualização do valor de "t". O cálculo dos valores ao longo da curva é feito através do polinômio de “Bézier” estudado em aulas teóricas (nota-se que esse cálculo deve ser realizado para as 3 variáveis, "x", "y" e "z").
Em cada iteração, são calculados 6 pontos que formam dois triângulos. Todos os pontos calculados são armazenados na “string” (“str”), que posteriormente é escrita no arquivo final especificado como parâmetro da função.

```
PS C:\Users\eduar\OneDrive\Ambiente de Trabalho\Projeto CG\phase3\generator\build\Debug> .\class2.exe patch teapot.patch
 10 bezier_10.3d
A Rodar!
PS C:\Users\eduar\OneDrive\Ambiente de Trabalho\Projeto CG\phase3\generator\build\Debug>
```

Figura 1: Execução do “generator” com o comando “patch”

```
1.4 0 2.4
1.3778 -0.254497 2.4
1.38491 0 2.43889
1.3778 -0.254497 2.4
1.36295 -0.251754 2.43889
1.38491 0 2.43889
1.38491 0 2.43889
1.36295 -0.251754 2.43889
1.38021 0 2.46806
1.36295 -0.251754 2.43889
1.35833 -0.2509 2.46806
1.38021 0 2.46806
1.38021 0 2.46806
1.35833 -0.2509 2.46806
1.38426 0 2.4875
1.35833 -0.2509 2.46806
1.36231 -0.251635 2.4875
1.38426 0 2.4875
1.38426 0 2.4875
1.36231 -0.251635 2.4875
1.3954 0 2.49722
1.36231 -0.251635 2.4875
1.37328 -0.253661 2.49722
1.3954 0 2.49722
1.3954 0 2.49722
1.37328 -0.253661 2.49722
```

Figura 2: Excerto de ficheiro gerado

4. Engine

4.1 Código

O código global do arquivo permaneceu praticamente inalterado, contudo, foram feitas algumas adições para atender às solicitações da terceira fase. Primeiramente, foi definida uma nova estrutura "Vertices", que armazena um vetor de pontos, semelhante ao "generator.cpp", para armazenar os pontos de controle de uma curva.
Em seguida, modificamos nossa estrutura "transformacao" para incluir parâmetros adicionais: tempo, tempo atual (necessários para calcular o alinhamento do objeto ao longo das curvas e rotações de objetos com base no tempo), o parâmetro "orientado" (denominado "align" no arquivo XML), que especifica se o objeto está orientado ao longo de uma curva ou não. Além disso, adicionamos um vetor de pontos de controle da curva e um parâmetro "t", que representa a percentagem ao longo da linha para o cálculo das curvas.

```
// Semelhante ao Generator.cpp
struct Vertices {   // Estrutura que armazena um vetor de floats, para armazenar pontos de controlo
    vector <float> cord;
};

// Struct que armazena as transformações
struct transformacao {
    string nome;
    float x;
    float y;
    float z;
    float angulo;               // para rotates
    float tempo = 0;
    float tempo_atual = 0;
    string orientado;           // Se o objecto deve ser orientado na curva (true ou false), align
    vector <Vertices> coord;    // pontos de control
    float t = 0;                // t de percentagem da reta
};
```

Figura 3: Struct Transformacao e Vertices

As funções de manipulação de arquivo permaneceram praticamente idênticas às da fase anterior, com exceção da função "parser_group". Nesta função, ao lidar com os comandos "translate" e "rotate", é verificado se o atributo "time" está definido. Se estiver, são criadas novas transformações: "timed_translate" e "timed_rotate", respetivamente. Para o "timed_translate", é feita uma verificação adicional no campo "align", conforme mencionado anteriormente.

```
if(nome == "translate") {
    // Caso seja translate mas sem Time, temos uma translação normal
    if (tipo->Attribute("time") == nullptr) {
        trans.nome = nome;
        trans.angulo = 0;
        trans.x = atof(tipo->Attribute("x"));
        trans.y = atof(tipo->Attribute("y"));
        trans.z = atof(tipo->Attribute("z"));
    }
    // Caso tenha Time, temos uma "timed_translate"
    else {
        trans.nome = "timed_translate";
        trans.angulo = 0;
        trans.tempo = atof(tipo -> Attribute("time")) * 1000;
        trans.orientado = tipo -> Attribute("align");   // Verifica se é uma translação orientada ou não, true or false

        trans.coord = {}; // Vetor vazio
        for (XMLElement* pc = tipo -> FirstChildElement("point"); pc != NULL; pc = pc -> NextSiblingElement()) {
            Vertices p;
            p.cord.push_back(atof(pc -> Attribute("x")));
            p.cord.push_back(atof(pc -> Attribute("y")));
            p.cord.push_back(atof(pc -> Attribute("z")));
            trans.coord.push_back(p); // Coloca os pontos de controlo no vetor "coord" da transformação
        }
    }

}
```

Figura 4: Translação

```
else if(nome == "rotate"){
    // Caso se trate de uma rotação sem parâmetro "time" temos uma rotação normal
    if (tipo->Attribute("time") == nullptr) {
        trans.nome = nome;
        trans.angulo = atof(tipo -> Attribute("angle"));
        trans.x = atof(tipo -> Attribute("x"));
        trans.y = atof(tipo -> Attribute("y"));
        trans.z = atof(tipo -> Attribute("z"));
    }
    // Caso haja parâmetro time, temos uma "timed_rotation"
    else {
        trans.nome = "timed_rotation";
        trans.tempo = atof(tipo -> Attribute("time")) * 1000;
        trans.x = atof(tipo -> Attribute("x"));
        trans.y = atof(tipo -> Attribute("y"));
        trans.z = atof(tipo -> Attribute("z"));
    }
}
```

Figura 5: Rotação

Tratada a manipulação do arquivo de entrada, temos então o código para o cálculo das curvas de "Catmull-Rom". Estas funções foram estudadas em aula prática e sendo as seguintes:

- "buildRotMatrix", que cria uma matriz de rotação dado vetores "x", "y" e "z";
- "cross", que calcula o produto vetorial entre dois vetores;
- "normalize", que normaliza um vetor;
- "mulMatrixVetor", que realiza a multiplicação de uma matriz por um vetor;
- "getCatmullRomPoint", que, dado um parâmetro t entre 0 e 1, calcula um ponto na curva Catmull-Rom, além de calcular a derivada desse ponto;
- "getGlobalCatmullRomPoint", que retorna um ponto global na curva com base no parâmetro t global e para isso invoca a função anterior;
- "renderCatmullRomCurve", que desenha a curva gerada usando "GL_LINE_LOOP".

Exemplo:

```
// retorna um ponto global na curva Catmull-Rom com base no t global
void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv, vector <Vertices> coord) {
    const int POINT_COUNT = coord.size();

    float t = gt * POINT_COUNT;  // Este é o t global real
    int index = floor(t);        // Determina em qual segmento estamos
    t = t - index;               // Determina onde dentro do segmento estamos

    // Índices armazenam os pontos de controle
    int indices[4];
    indices[0] = (index + POINT_COUNT - 1) % POINT_COUNT;
    indices[1] = (indices[0] + 1) % POINT_COUNT;
    indices[2] = (indices[1] + 1) % POINT_COUNT;
    indices[3] = (indices[2] + 1) % POINT_COUNT;

    // Obtém o ponto Catmull-Rom para o parâmetro local t
    getCatmullRomPoint(t, coord[indices[0]].cord, coord[indices[1]].cord, coord[indices[2]].cord, coord[indices[3]].cord, pos, deriv);
}

// Renderiza a curva Catmull-Rom
void renderCatmullRomCurve(vector <Vertices> coord) {
    // Desenha a curva usando segmentos de linha com GL_LINE_LOOP
    float pos[3], deriv[3];
    float LINE_SEGMENTS = 100;

    glBegin(GL_LINE_LOOP);
    for(int i = 0; i < LINE_SEGMENTS; i++){
        // Obtém um ponto na curva Catmull-Rom para um determinado t global
        getGlobalCatmullRomPoint(1 / LINE_SEGMENTS * i, pos, deriv, coord);

        // Desenha o ponto na curva
        glVertex3f(pos[0], pos[1], pos[2]);
    }
    glEnd();
}
```

Figura 6: Excerto das funções estudadas na aula prática

Definidas as funções necessárias para o tratamento das curvas de "Catmull-Rom", passamos para a função "renderScene", que sofreu bastantes modificações. Além das transformações usuais, agora também são verificadas as transformações "timed_translate" e "timed_rotate".

No caso de "timed_translate", é calculada a curva de "Catmull-Rom" e aplicadas as transformações necessárias. Se a transformação tiver "orientado" igual a "true", é feito ainda o cálculo da matriz de rotação do objeto com base nas derivadas do ponto, e a matriz gerada é multiplicada pela matriz de transformação global. Após isso, ocorre a atualização dos parâmetros t, tempo e tempo_atual da transformação.

```
else if (objetos[i].transformacoes[j].nome == "timed_translate") {
    float pos[3], deriv[3];                                     // Posicao e Derivada na curva
    renderCatmullRomCurve(objetos[i].transformacoes[j].coord);      // desenha a curva
    getGlobalCatmullRomPoint(objetos[i].transformacoes[j].t, pos, deriv, objetos[i].transformacoes[j].coord);
    glTranslatef(pos[0], pos[1], pos[2]);                           // aplica a translação

    // se for orientada, vai calcular a rotação do objeto com base nas derivadas da posicao da curva
    if (objetos[i].transformacoes[j].orientado == "true"){
        // Vetor x é definido como a derivada da curva
        float x[3] = {deriv[0], deriv[1], deriv[2]};
        // Vetores y e z serão calculados a partir do vetor x
        float y[3];
        float z[3];
        float mat[16];              // Matriz de rotação para orientação
        float prev_y[3] = {0,1,0};  // Vetor de orientação y anterior

        // normaliza x e calcula z apartir de x.prev_y
        normalize(x);
        cross(x, prev_y, z);
        // normaliza z e calcula y apartir de z.x
        normalize(z);
        cross(z, x, y);

        // normaliza y
        normalize(y);

        // constroi a matriz de rotação
        buildRotMatrix(x, y, z, mat);
        // multiplica a matriz de gerada pela matriz atual de transformação
        glMultMatrixf(mat);
    }

    // Atualiza o tempo atual para uso na próxima iteração
    float new_time = glutGet(GLUT_ELAPSED_TIME);
    // Calcula a diferença de tempo desde a última atualização
    float diff = new_time - objetos[i].transformacoes[j].tempo_atual;
    // Atualiza o parâmetro t da curva Catmull-Rom com base na diferença de tempo
    objetos[i].transformacoes[j].t += diff / objetos[i].transformacoes[j].tempo;
    // Atualiza o tempo atual
    objetos[i].transformacoes[j].tempo_atual = new_time;
}
```

Figura 7: "RenderScene" para o caso "timed_translate"

Caso seja uma "timed_rotation", o ângulo de rotação é definido com base no tempo desde o início da execução do programa.

```
else if (objetos[i].transformacoes[j].nome == "timed_rotation") {
    // calcula o angulo de rotação com base no tempo decorrido desde o inicio da execução do programa
    glRotatef(glutGet(GLUT_ELAPSED_TIME) * 360 / objetos[i].transformacoes[j].tempo, objetos[i].transformacoes[j].x,
}
```

Figura 8: "RenderScene" para o caso "timed_rotation"

Em seguida, temos o desenho dos objetos, agora realizado através de "VBO's", conforme solicitado no enunciado desta fase.
Por fim, na função "renderScene", temos o cálculo dos "frames", os quais são exibidos como título da janela, como já visto em aulas práticas anteriores.

```
    // apos as transformacoes de cada objeto, o seu desenho
    // desenho dos triangulos usando VBOs
    glBindBuffer(GL_ARRAY_BUFFER, VBOBuffer[i]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, VBOBuffCount[i]);

    glPopMatrix();
}

// FPS
int timer = glutGet(GLUT_ELAPSED_TIME);                 //
int fps;
char str[64];                                           //

frame++;
if(timer - timebase > 1000){
    // calcula o valor dos fps
    fps = frame * 1000.0 / (timer - timebase);
    timebase = timer;
    frame = 0;
    sprintf(str, "FPS: %d", fps);
    // define o título da janela com o valor do FPS
    glutSetWindowTitle(str);
}
```

Figura 9 – Desenho com "VBO's" e cálculo dos "frames"

Finalizando o código, a função "main" permanece semelhante, exceto pela substituição dos testes pelos testes da fase 3 e a inicialização dos buffers para o uso de "VBO's" no desenho das figuras.

```
glewInit();

glGenBuffers(objetos.size(), VBOBuffer);

for (int i = 0; i < objetos.size(); i++) {
    VBOBuffCount[i] = objetos[i].pontos.size() / 3;
    glBindBuffer(GL_ARRAY_BUFFER, VBOBuffer[i]);
    glBufferData(GL_ARRAY_BUFFER, objetos[i].pontos.size() * sizeof(float), objetos[i].pontos.data(), GL_STATIC_DRAW);
}
```

Figura 10 – Inicialização dos "VBO's" na função "main"

Será importante referir que definimos os buffers dos "VBO's" da seguinte forma:

```
// VBOs
const int objs = 20;
GLuint VBOBuffer[objs], VBOBuffCount[objs];
```

Figura 11 - Definição dos "VBO's"

Sendo que cada vetor do buffer corresponde a um objeto, consideramos assim uma limitação do código sendo que existe um número máximo de imagens objetos que podemos definir.

4.2 Testes/Resultados finais

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_3_1.xml":

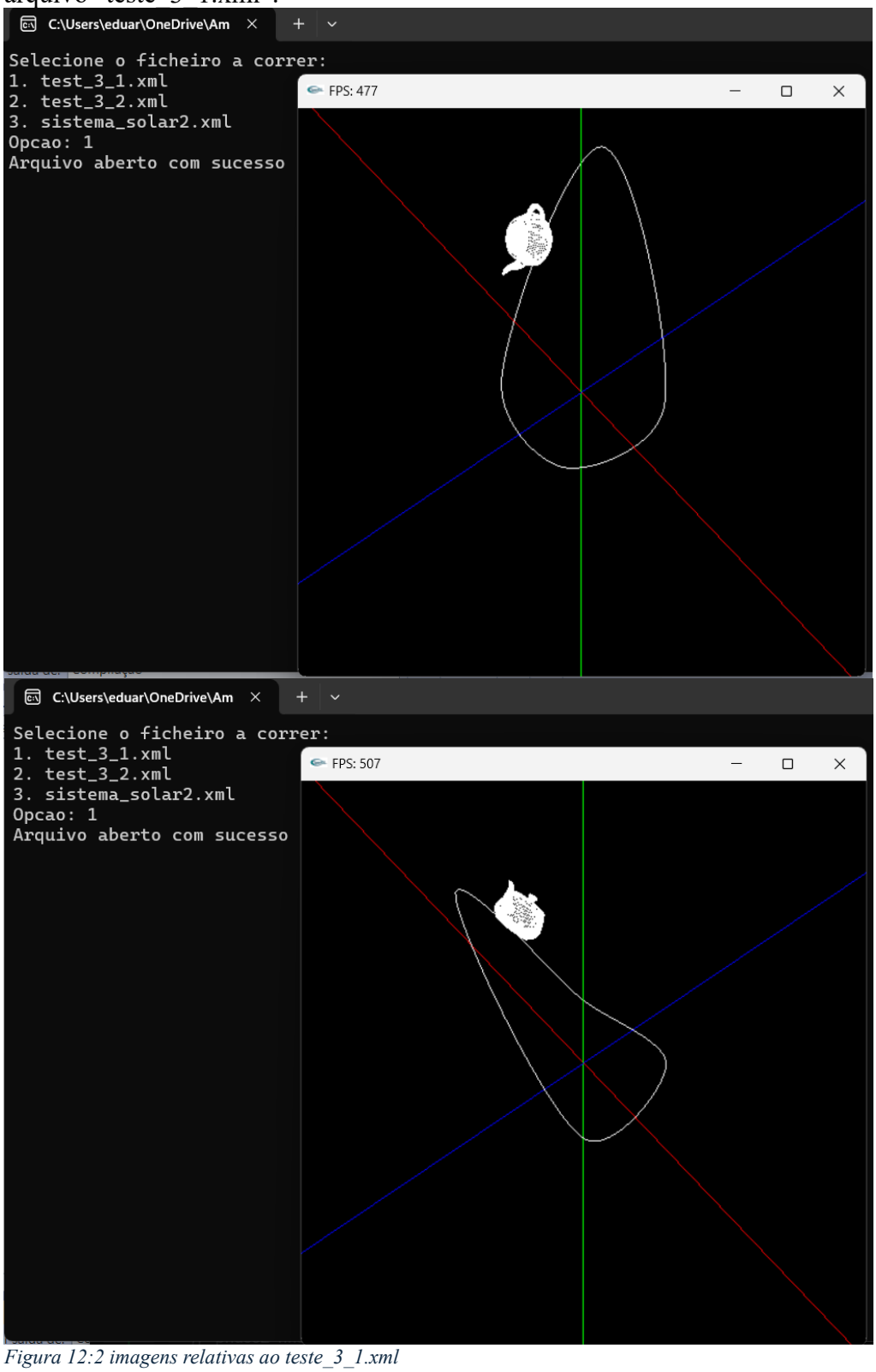

Figura 12:2 imagens relativas ao teste_3_1.xml

Resultado obtido da execução do engine.cpp, onde a opção selecionada corresponde ao arquivo "teste_3_2.xml":

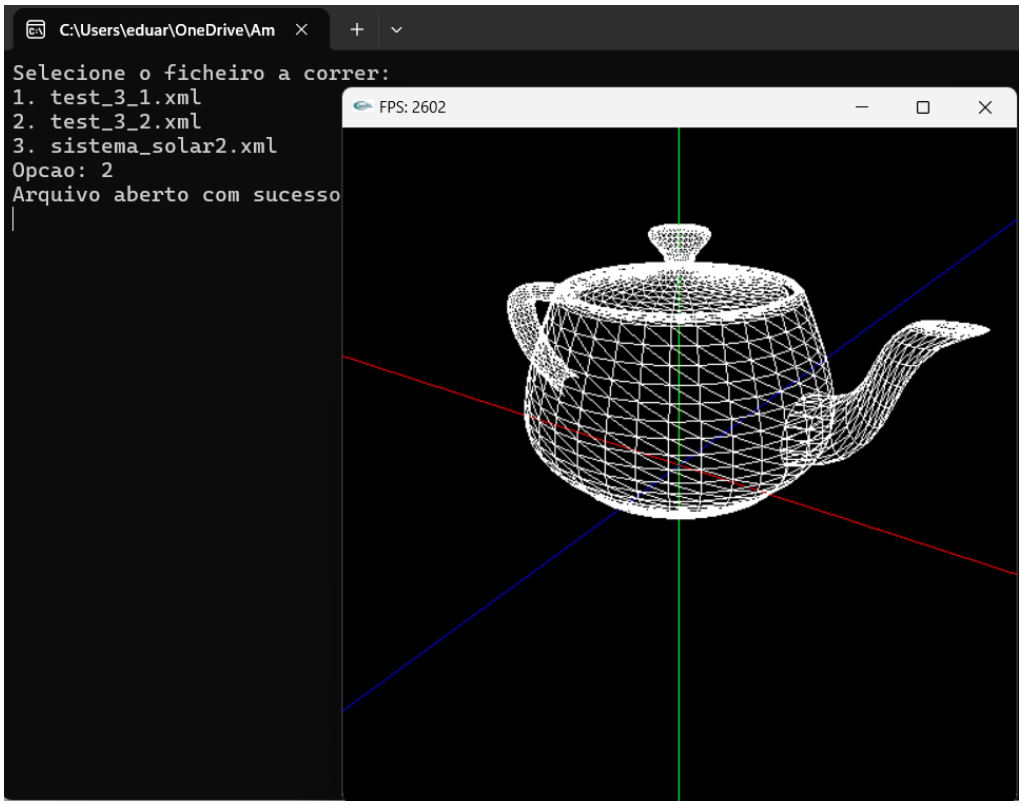

Figura 13: teste_3_2.xml

5. Sistema Solar

Para a definição do sistema solar, aproveitamos o sistema já feito na fase anterior e estabelecemos a rotação para o sol e a translação dos restantes corpos celestes utilizando curvas de "Catmull-Rom". Os pontos de controlo utilizados para definir as curvas foram mais fruto da intuição e de testes empíricos do que propriamente cálculos precisos. Reconhecemos essa abordagem como uma limitação, porém, dadas as restrições de tempo e os recursos limitados do grupo, consideramos que obtivemos um resultado satisfatório. É importante ressaltar que o enunciado solicitava a definição do cometa através de um patch de Bezier. Diante da incerteza sobre como proceder, optamos por utilizar o arquivo do "teapot" utilizado em outros testes, aplicando escalas para achatar o objeto ao longo dos eixos "x" e "z".

```
<world>
    <window width="800" height="600"/>
    <camera>
        <position x="50" y="100" z="200" />
        <lookAt x="50" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="100" far="500" />
    </camera>

    <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" />
            </models>
        </group>
        <group>
            <!-- Mercurio -->
            <transform>
                <translate time = "4" align = "true">
                    <point x = "35" y = "0" z = "0" />
                    <point x = "25" y = "0" z = "-25" />
                    <point x = "0" y = "0" z = "-35" />
```

```
    <group>
        <!-- Cometa -->
        <transform>
            <translate time = "15" align = "true" >
                <point x = "85" y = "55" z = "0" />
                <point x = "60" y = "40" z = "-60" />
                <point x = "0" y = "0" z = "-85" />
                <point x = "-60" y = "-40" z = "-60" />
                <point x = "-85" y = "-55" z = "0" />
                <point x = "-60" y = "-40" z = "60" />
                <point x = "0" y = "0" z = "85" />
                <point x = "60" y = "40" z = "60" />
            </translate>
            <scale x="3" y="1" z="2" />
        </transform>
        <models>
            <model file="bezier_10.3d" />
            <!-- model file="bezier_10.3d sphere_1_8_8.3d -->
        </models>
    </group>
</group>
```

Figura 14: Representação do sol, Mercúrio e cometa do sistema solar.

Resultado:

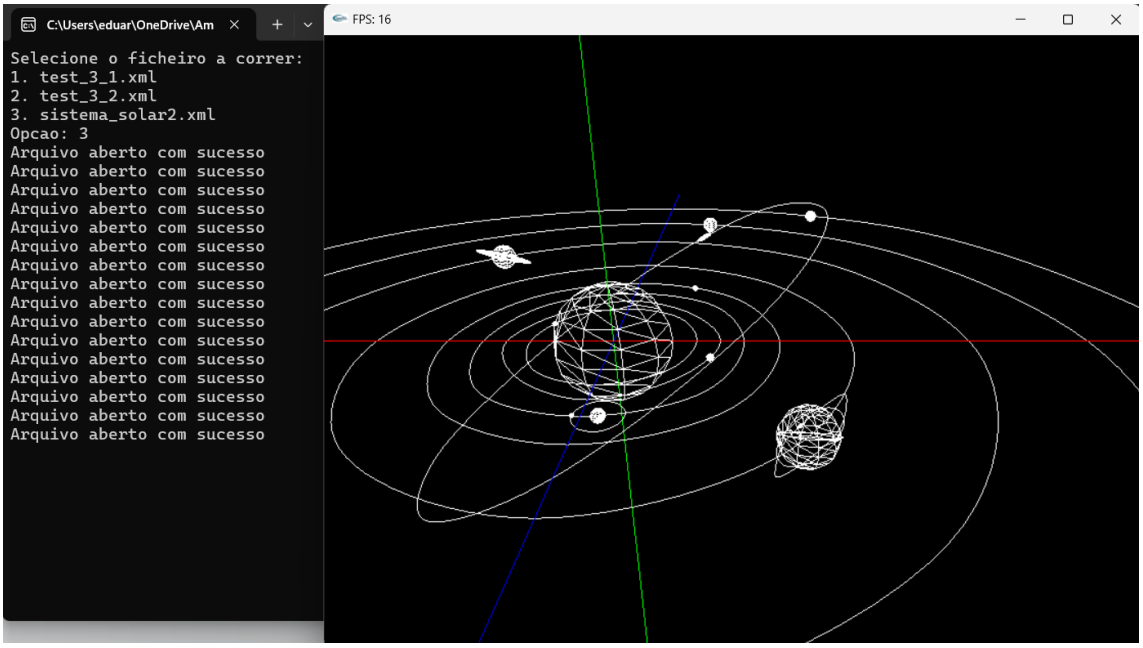

Figura 15: Resultado da execução do arquivo "sistema_solar.xml"

6. Conclusão

A terceira fase deste trabalho prático de Computação Gráfica representou um avanço significativo em relação às fases anteriores. O código foi adaptado para desenhar os objetos usando "VBO's", introduzindo a produção de objetos através de "patches" e a implementação de curvas de "Catmull-Rom", entre outras alterações não tão significativas. Além disso, foi atualizado o arquivo "sistema_solar.xml" incorporando todas essas novas especificidades adicionadas ao código.

Acreditamos ter alcançado todos os objetivos propostos, embora talvez não de forma totalmente precisa, conforme discutido ao longo do relatório, como as limitações relacionadas ao número de objetos na cena e a incerteza quanto ao cometa do sistema solar. Contudo, consideramos que nos aproximamos bastante desses objetivos. É importante ressaltar que o facto de o grupo ser composto apenas por 2 membros está, cada vez mais, a tornar-se um obstáculo cada vez maior , devido ao aumento da carga de trabalho e, principalmente, à complexidade do código.