Universidade do Minho
Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Ano Letivo de 2023/2024

Trabalho Prático

Parte 3- Curvas, superficies e VBO's

Grupo 33

Diogo Gabriel Lopes Miranda (a100839)

Fábio Daniel Rodrigues Leite (a100902)

João Ricardo Ribeiro Rodrigues (a100598)

Sandra Fabiana Pires Cerqueira (a100681)

Março, 2024

C G

Resumo

Este relatório documenta a terceira fase do projeto da unidade curricular de Computação Gráfica, sendo a continuação do trabalho desenvolvido nas duas fases anteriores. Iniciamos com uma contextualização do projeto, seguida do trabalho que nos foi proposto e o que teríamos de fazer para o realizar.

De seguida, são explicadas as alterações efetuadas a nível do Generator, incluindo a sua nova capacidade de criar modelos a partir de *patches* de *Bézier*, e as significativas melhorias e mudanças no Engine, que agora suporta novos tipos de translações e rotações dinâmicas e utiliza *Vertex Buffer Objects (VBOs)* para otimizar a renderização e a implementação de transformações geométricas animadas, possibilitadas por rotações que variam com o tempo e translações guiadas por curvas de Catmull-Rom.

Posteriormente são apresentadas imagens relativas aos resultados obtidos nos testes desta fase e o estado do nosso sistema solar.

Concluímos com uma avaliação crítica do trabalho realizado, destacando os pontos fortes, as limitações observadas e as dificuldades enfrentadas durante o desenvolvimento.

Palavras-chave: Computação Gráfica, GLUT, pontos, vetores, XML, figura geométrica, Generator, Engine, Open GL

Índice

Resumo 2
Lista de Figuras 4
1. Introdução 5
1.1 Contextualização 5
1.2 Descrição do trabalho proposto 5
1.3 Resumo do trabalho a desenvolver 5
2. Resolução do problema 6
2.1 Estrutura do Projeto 6
3.Alterações Fase 3 7
3.1 Alterações no Generator 7
3.1.1 Criação de modelos basados em Patches de Bezier 7
3.1.2 Criação dos pontos 3D utilizando Curvas de Bezier 8
3.2 Alterações no Engine 12
VBOs 12
Alteração nas transformações e curvas catmullRoom 15
Rotação baseada em tempo 15
Desenhar a Curva 17
Translações na curva 18
4. Resultados dos testes da fase 3 20
4.1 Test_3_1 20
4.2 Test_3_2 20
Discussão dos resultados obtidos 21
4. Sistema Solar 21
4. Conclusão 23

Lista de Figuras

Figura 1- Estrutura do projeto 6
Figura 2 - Estrutura para guardar informações do patch 7
Figura 3 - Obtenção do número de patches 7
Figura 4 - Leitura e armazenamento dos patches 8
Figura 5 - Obtenção no número de control points 8
Figura 6 - Leitura e armazenamento dos control points 8
Figura 7 - Função 'generatorFigura' 8
Figura 8 - População do vetor temporário 'patchControlPoints' 9
Figura 9 - Cálculo do step 9
Figura 10 - Célula da grelha, e grelha completa 9
Figura 11 - Cálculo das coordenadas (u,v) 10
Figura 12 - Fórmula matemática para calcular Bezier 10
Figura 13 - Matrix com os control points 10
Figura 14 - Fórmula do polinómio de Bernstein 10
Figura 15 - Função evaluateBezier 11
Figura 16 - Cálculo das coordenadas do ponto 11
Figura 17 - Calcula os pontos 3D e escreve-os no ficheiro de saida 11
Figura 18- Modelo.h 12
Figura 19-Função storeBuffer 13
Figura 20 - Função pushGrupo 13
Figura 21-Função pushBuffer 13
Figura 22 - Função drawTriangles 14
Figura 23-drawGrupo 14
Figura 24- Classe Transformacao 15
Figura 25 - Função para ler o ficheiro xml 15
Figura 26 - Fórmula matemática para calcular os pontos da curva 17
Figura 27 - Número de segmentos utilizados 17
Figura 28 - Função para calcular os pontos da nossa curva 18
Figura 29 - trecho função 'applyTransformation' 18
Figura 30 - Função para calcular o ponto seguinte da curva 19
Figura 31 - Função para movimentar o objeto ao longo da curva 19
Figura 32 - Função 'alinharCurva' 19
Figura 33- Teste dos professores 20
Figura 34- Resultado obtido na nossa execução 20
Figura 35- Teste dos professores 20
Figura 36-Resultado obtido na nossa execução 20

1. Introdução

1.1 Contextualização

Nesta terceira fase do trabalho prático, iremos continuar o trabalho já feito nas duas primeiras fases, aplicando o conhecimento adquirido ao longo das últimas semanas, nomeadamente a questão dos *patches* de *Bezier* e dos *VBO's*.

1.2 Descrição do trabalho proposto

Nesta etapa do projeto, foi-nos então proposto o desenvolvimento de modelos utilizando *patches* de *Bézier,* juntamente com a expansão das transformações geométricas de translação e rotação. Adicionalmente, foi- nos pedido que modificasse-mos o método de renderização dos modelos, para usarmos *Vertex Buffer Objects (VBOs)*.

Além destes requisitos, também foi-nos pedido que criassemos uma cena demonstrativa dinâmica do Sistema Solar, que inclui um cometa com trajetória definida por uma curva de Catmull-Rom. O cometa deve ser construído usando patches de Bézier, utilizando, por exemplo, os pontos de controle fornecidos para o modelo do bule.

1.3 Resumo do trabalho a desenvolver

Dado o requisito de desenvolver um novo tipo de modelo, será necessário adaptar o *Generator* para que, ao receber um ficheiro patch, ele consiga gerar uma Superfície de *Bezier*.

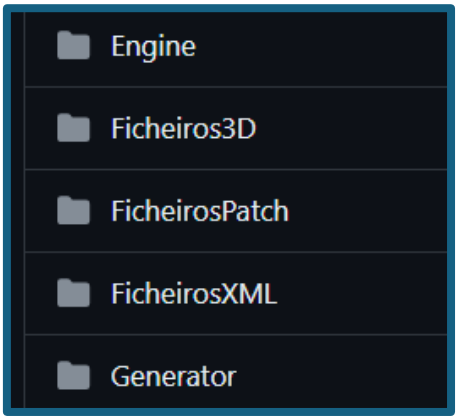
Relativamente ao *Engine*, será efetuada uma modificação no *parsing* das rotações e translações, permitindo-lhes incluir curvas de *Catmull-Rom* e rotações em torno do próprio eixo. Adicionalmente, o método de desenho terá de ser alterado para utilizar *VBOs* indexados.

Para além destas alterações, será ainda necessário criar um sistema solar dinâmico, no qual um cometa seguirá uma trajetória definida por uma curva de *Catmull-Rom*. Este cometa deverá ser construído utilizando os *patches de Bezier* mencionados anteriormente.

2. Resolução do problema

2.1 Estrutura do Projeto

Tendo em mente tudo aquilo que tínhamos de desenvolver, como descrito da secção 1 deste relatório, o grupo optou por estruturar o projeto da seguinte forma:



- *Figura 1- Estrutura do projeto*

- **Generator:** contém todo o código necessário ao cálculo das coordenadas para a representação das diferentes primitivas (plano, caixa, cone, esfera, anel e teapot), bem como a criação e escrita dos pontos nos ficheiros .3d para que possam ser usados no Engine;
- **Engine:** Contém todo o código necessário para que seja possível ver as primitivas;
- **Ficheiros3D:** Onde são guardados os ficheiros .3d gerados pelo Generator
- **FicheirosPatch:** Onde esta o ficheiro teapot.patch, usado para gerar a figura teapot a partir dos Bezier points.
- **FicheirosXML**: Onde estão armazenados os ficheiros XML com as configurações que o Engine tem de ler para dar *display,* bem como o resultado suposto dos mesmos para efeitos de testes.

3.Alterações Fase 3

Nesta secção iremos então explicar, como procedemos à implementação das novas funcionalidades.

3.1 Alterações no Generator

3.1.1 Criação de modelos basados em Patches de Bezier

Para o nosso Generator conseguir gerar os modelos baseados em *patches* de *Bezier*, precisamos primeiramente de ler o *.patch* e extrair a informação dele.

Para responder a esta necessidade começamos por definir as estruturas de dados que vão ser usadas para guardar as informações lidas do ficheiro *patch*.

Assim, definimos o *'patchIndices'* que é um vetor de vetores de números inteiros, responsável por armazenar os índices dos *control points* de cada *patch*. Cada *patch* é constituído por 16 índices referentes às posições específicas do vetor responsável por armazenar os *control points*. De modo a conseguirmos armazenar esses pontos, criamos a estrutura 'controlPoints ', que consiste em um vetor de 'Point3D', onde cada 'Point3D' é uma *struct* que contém as coordenadas de um ponto (x,y,z), usadas para representar a posição de um ponto no espaço.

```
struct Point3D {
    float x, y, z;
};

std::vector<std::vector<int>> patchIndices;
std::vector<Point3D> controlPoints;
```

Figura 2 - Estrutura para guardar informações do patch

Após criarmos as estruturas para guardar as informações do ficheiro *patch*, passamos para a leitura do mesmo. Para conseguirmos ler esse ficheiro criamos a função '*readPatchFile'*, que começa por abrir o ficheiro *patch* e guardar o número de *paches* que o mesmo contém.

```
void readPatchFile(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error opening input file!" << std::endl;
        return;
    }

    int numPatches, numControlPoints;
    std::string line;

    // Leitura do número de patches
    file >> numPatches;
    std::getline(file, line); // Ignorar o resto da linha após ler numPatches
```

Figura 3 - Obtenção do número de patches

Após guardar o número de patches, a função entra lê e armazena o conteúdos das linhas seguintes, nas quais estão presentes os patches a realizar. Cada um desses patches contém os índices dos pontos de controlo que vão ser utilizados para aquele patch específico. Esses índices são então extraídos e armazenados em '*patchIndices*'.

```
patchIndices.resize(numPatches);
for (int i = 0; i < numPatches; ++i) {
    std::getline(file, line);
    std::istringstream iss(line);
    int index;
    while (iss >> index) {
        patchIndices[i].push_back(index);
        if (iss.peek() == ',') iss.ignore();
    }
}
```

Figura 4 - Leitura e armazenamento dos patches

Após a leitura e após guardamos todos os *patches*, vamos fazer um processo semelhante para os *control points*. Começando por guardar quantos *control points* o nosso ficheiro contém.

```
// Leitura do número de pontos de controle
file >> numControlPoints;
std::getline(file, line);
```

Figura 5 - Obtenção no número de control points

Tendo guardado o número de *control points,* vamos ler o resto ficheiro linha a linha, de modo a guardar todos as coordenadas (x,y,z) dos nossos *control points*.

```
controlPoints.resize(numControlPoints);
for (int i = 0; i < numControlPoints; ++i) {
    if (!getline(file, line)) {
        std::cerr << "Failed to read line for control point " << i << std::endl;
        continue;
    }
    std::istringstream iss(line);
    char comma;
    if (!(iss >> controlPoints[i].x >> comma >> controlPoints[i].y >> comma >> controlPoints[i].z)) {
        std::cerr << "Failed to read control point " << i << std::endl;
    }
}
```

Figura 6 - Leitura e armazenamento dos control points

3.1.2 Criação dos pontos 3D utilizando Curvas de Bezier

Após a leitura do ficheiro *patch* e após guardarmos todas as suas informações, passamos para a geração dos pontos 3D, que vão ser utilizados pelo nosso programa *Engine* para gerar a figura pretendida. Para responder a essa necessidade criamos a função 'generatorFigura'.

```
void generatorFigura(const std::string& filenamePatch, int tesselationLevel, const std::string& outputFile)
```

Figura 7 - Função 'generatorFigura'

A nossa função vai percorrer individualmente cada *patch* presente no ficheiro de entrada. Para cada *patch*, os índices dos *control points* são utilizados para reunir os respetivos pontos de controlo necessários para a tesselação.

A tesselação consiste em dividir uma superfície grande em várias superfícies mais pequenas e mais simples, no nosso caso, são superfícies definidas por curvas de bezier, esta divisão acontece num plano (u,v), que varia de 0 a 1 em ambas as direções. Ou seja, quão maior for o nível da tesselação, maior será o nível de detalhe da figura resultante, uma vez que, ao dividirmos o nosso plano em uma grelha, quantos mais divisórias ela tiver, maior será o número de pontos gerados e consequentemente o detalhe obtido será maior.

Após reunirmos todos os pontos de controlo cujo índice corresponde ao presente no patch em questão, armazenamos esses pontos no vetor temporário 'patchControlPoints'.

```
for (const auto& patch : patchIndices) {
    std::vector<Point3D> patchControlPoints;
    for (int index : patch) {
        patchControlPoints.push_back(controlPoints[index]);
    }
```

Figura 8 - População do vetor temporário 'patchControlPoints'

Tendo os *control points* necessários devidamente armazenado, passamos para o cálculo das coordenadas dos pontos 3D que vão ser escritos no ficheiro de saída.

Para esse cálculo, primeiramente, temos de calcular o '*step*', o '*step*' corresponde ao comprimente das arestas dos vários quadrados que vão definir a nossa grelha. Para calcular este valor dividimos a tamanho total da aresta do nosso plano, pleno nível de tesselação, nesta fase o nivél de tesselação usado foi 10, tal como pedido nos testes dos professores, o que significa que o nosso plano vai estar dividido em uma grelha 10 por 10 e cada aresta dos quadrados da grelha vai ser igual a 0,1 (1/10=0,1).

```
float step = 1.0f / tesselationLevel;
```

Figura 9 - Cálculo do step

Tendo o '*step*' calculado, o passo seguinte será calcular os valores das coordenadas dos pontos 'u', 'v', 'uNext' e 'vNext', estes pontos correspondem às coordenadas dos vértices do quadrado para o qual estamos a calcular as coordenadas no plano de Bezier, como se pode ver pela figura 10.

Figura 10 - Célula da grelha, e grelha completa

Os pontos A, B, C e D da nossa figura exemplo tem as seguintes coordenadas:

- Ponto D: (u,v);

- Ponto C: (uNext,v)
- Ponto A: (u,vNext)
- Ponto B: (uNext,vNext)

Para calcular essas coordenadas, fazemos as seguintes operações:

```
for (int i = 0; i < tesselationLevel; ++i) {
    float u = i * step;
    float uNext = (i + 1) * step;
    for (int j = 0; j < tesselationLevel; ++j) {
        float v = j * step;
        float vNext = (j + 1) * step;
```

Figura 11 - Cálculo das coordenadas (u,v)

Tendo as coordenadas da célula podemos finalmente calcular as coordenadas da superfície de Bezier.

Para realizar esses cálculos definimos a função '*evaluateBezier*', para criar esta função seguimos a seguinte fórmula matemática para calcular Bezier:

$$\mathbf{S}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3}\mathbf{P}_{ij}B_{i,3}(u)B_{j,3}(v)$$

Figura 12 - Fórmula matemática para calcular Bezier

Tal como podemos ver na fórmula, ela utiliza uma matriz 4x4 'P', que é constituída pelos 16 *control points* relativos ao *patch* em que nos encontramos.

$$\begin{bmatrix} \mathbf{P}_{00} & \mathbf{P}_{01} & \mathbf{P}_{02} & \mathbf{P}_{03} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \mathbf{P}_{12} & \mathbf{P}_{13} \\ \mathbf{P}_{20} & \mathbf{P}_{21} & \mathbf{P}_{22} & \mathbf{P}_{23} \\ \mathbf{P}_{30} & \mathbf{P}_{31} & \mathbf{P}_{32} & \mathbf{P}_{33} \end{bmatrix}$$

Figura 13 - Matrix com os control points

Outro parâmetro utilizado é o polinómio de Bernstein 'B', que é definido pela seguinte fórmula matemática:

$$\begin{aligned} B_{0,3}(t) &= (1-t)^3 \\ B_{1,3}(t) &= 3t(1-t)^2 \\ B_{2,3}(t) &= 3t^2(1-t) \\ B_{3,3}(t) &= t^3 \end{aligned}$$

Figura 14 - Fórmula do polinómio de Bernstein

Para o calcular temos de substituir o parâmetro 't' pelas coordenadas 'u' e 'v' do vértice da célula em que nos encontramos.

Tendo em conta os cálculos matemáticos necessários, definimos a nossa função '*evaluateBezier*' da seguinte forma:

```
Point3D evaluateBezier(const std::vector<Point3D>& patchControlPoints, float u, float v) {
    // Coeficientes de Bernstein para um polinômio de grau 3
    float Bu[4] = {
        (1 - u) * (1 - u) * (1 - u),  // u^0
        3 * (1 - u) * (1 - u) * u,    // u^1
        3 * (1 - u) * u * u,          // u^2
        u * u * u                     // u^3
    };

    float Bv[4] = {
        (1 - v) * (1 - v) * (1 - v),  // v^0
        3 * (1 - v) * (1 - v) * v,    // v^1
        3 * (1 - v) * v * v,          // v^2
        v * v * v                     // v^3
    };

    Point3D result = {0, 0, 0};

    // Calcular o ponto na superfície de Bezier usando a matriz de Bernstein
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
            float coeff = Bu[i] * Bv[j]; // Coeficiente do polinômio de Bernstein para (u, v)
            result.x += coeff * patchControlPoints[i * 4 + j].x;
            result.y += coeff * patchControlPoints[i * 4 + j].y;
            result.z += coeff * patchControlPoints[i * 4 + j].z;
        }
    }

    return result;
}
```

Figura 15 - Função evaluateBezier

Tendo todos os valores apurados podemos aplicar a fórmula da figura 12, para calcular as coordenadas de um ponto.

```
// Calcular o ponto na superfície de Bezier usando a matriz de Bernstein
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        float coeff = Bu[i] * Bv[j]; // Coeficiente do polinômio de Bernstein para (u, v)
        result.x += coeff * patchControlPoints[i * 4 + j].x;
        result.y += coeff * patchControlPoints[i * 4 + j].y;
        result.z += coeff * patchControlPoints[i * 4 + j].z;
    }
}
```

Figura 16 - Cálculo das coordenadas do ponto

Voltando à nossa função '*generatorFigura*', ela chama a função '*evaluateBezier*' 4 vezes, uma para cada vértice da célula do plano que estamos a calcular, e de seguida, após determinar os pontos escreve-os no ficheiro 3D correspondente.

```
// Calcula os vertices da posição da grelha em que nos encontramos
Point3D p1 = evaluateBezier(patchControlPoints, u, v);
Point3D p2 = evaluateBezier(patchControlPoints, uNext, v);
Point3D p3 = evaluateBezier(patchControlPoints, u, vNext);
Point3D p4 = evaluateBezier(patchControlPoints, uNext, vNext);

// Write two triangles of this cell
writeVertex(outFile, p1.x, p1.y, p1.z, p3.x, p3.y, p3.z, p4.x, p4.y, p4.z);
writeVertex(outFile, p1.x, p1.y, p1.z, p4.x, p4.y, p4.z, p2.x, p2.y, p2.z);
```

Figura 17 - Calcula os pontos 3D e escreve-os no ficheiro de saida

Este processo vai repetir-se para todas as células do plano, quando o mesmo tiver completo passamos para o próximo patch. Quando todos os patches tiverem sido lidos e os pontos completamente gerados, a figura está pronta para ser desenhada.

3.2 Alterações no Engine

Ao nível do engine foram efetuadas mudanças significativas para suportar as curvas de Catmull-Rom, rotações e translações com o novo parâmetro “tempo”, permitindo animações dinâmicas. Além disso, para melhorar a eficiência foram implementados Vertex Buffer Objects (VBOs), agilizando o processo de renderização das figuras. De seguida, iremos explicar como foram efetuadas todas estas modificações.

VBOs

Para conseguirmos suportar os VBOs como pedido, tivemos de alterar a nossa estrutura Modelo, uma vez que inicialmente ela continha apenas um vetor de pontos e valores RGB para armazenar as cores. Para otimizar o desempenho e aproveitar os benefícios de renderização dos VBOs, adicionámos um identificador de buffer ,o “GLuint buffer”, à nossa classe. Este “GLuint buffer” irá servir então para armazenar o identificador do buffer gerado pelo OpenGL, onde os dados dos vértices do modelo são armazenados no GPU. Assim, cada modelo irá possuir o seu próprio buffer, o que permite que os dados do modelo sejam enviados uma única vez para a placa gráfica no início da execução e sejam acedidos rapidamente durante as renderizações, o que melhora o desempenho em cenas complexas.

```h
#ifndef MODELO_H
#define MODELO_H

#include <vector>
#include <GL/glut.h>

struct Point3D {
    float x, y, z;
    Point3D(float _x, float _y, float _z) : x(_x), y(_y), z(_z) {}
};

class Modelo {
public:
    std::vector<Point3D> pontos;
    float r, g, b;
    GLuint buffer;
};

#endif /* MODELO_H */
```

Figura 18- Modelo.h

Com a estrutura do modelo pronta, o próximo passo foi implementar a geração dos buffers para cada modelo. Para isso, foi criada a função *storeBuffer*, responsável por gerar um novo *buffer*, alocar e enviar os dados dos vértices para a GPU. Esta função é então chamada durante a inicialização do programa para cada modelo, garantindo que todos os modelos tenham os seus buffers criados e configurados corretamente.

```
GLuint storeBuffer(const std::vector<Point3D>& pontos) {
    GLuint buffer;
    glGenBuffers(1, &buffer); // Gera um novo buffer
    glBindBuffer(GL_ARRAY_BUFFER, buffer);

    // Aloca e envia os dados para a GPU
    glBufferData(GL_ARRAY_BUFFER, pontos.size() * sizeof(Point3D), &pontos[0], GL_STATIC_DRAW);

    // Desvincula o buffer ao terminar de configurá-lo
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    return buffer; // Retorna o ID do novo buffer criado
}
```

Figura 19-Função storeBuffer

Além disso, foi implementada a função *pushGrupo*, que percorre recursivamente a hierarquia de grupos, gerando os buffers para todos os modelos contidos em cada grupo. Isso garante que todos os modelos da cena tenham os seus buffers corretamente associados.

Durante a execução da *pushGrupo*, cada modelo é processado individualmente. Para preencher os VBOs com as coordenadas dos pontos armazenados no *std::vector<Point3D> globalPoints*, a função *storeBuffer* é chamada para cada modelo. Como mencionado anteriormente, esta função para vai então criar um novo buffer, alocar e enviar os dados dos vértices para a GPU.

```
void pushGrupo(Grupo& grupo) {
        for (Modelo& modelo : grupo.modelos) {
                modelo.buffer = storeBuffer(modelo.pontos);
        }
        for (Grupo& filho : grupo.filhos) {
                pushGrupo(filho);
        }
}
```

Figura 20 - Função pushGrupo

Posteriormente, a função *pushBuffer* percorre os grupos existentes e chama a função *pushGrupo* para cada um deles. Isto garante que todos os modelos contidos em cada grupo tenham os seus buffers associados corretamente.

```
void pushBuffer() {
    for (size_t i = 0; i < gruposLista.size(); ++i) {
        std::cout << "Buffers criado para o grupo " << i << std::endl;
        pushGrupo(gruposLista[i]);
    }
}
```

Figura 21-Função pushBuffer

Posto isto, para desenhar as figuras utilizando VBOs, tivemos de modificar a nossa função *drawTriangles*. Na nossa nova versão dela, os pontos utilizados para gerar os

triângulos da figura são lidos do buffer e não do vetor onde estavam armazenados os pontos na fase anterior.

Com as alterações realizadas, a função *drawTriangles* agora recebe um parâmetro adicional, o buffer, que representa o identificador associado ao modelo. Em vez de utilizar glBegin e glEnd para definir os triângulos, a função passou a usar *glDrawArrays*, o que permite desenhar diretamente a partir dos dados armazenados no buffer. Adicionalmente, o *glBindBuffer* é utilizado para vincular o buffer antes de chamar *glVertexPointer* e *glDrawArrays*, assegurando que estamos a usar corretamente os dados armazenados.

```
void drawTriangles(vector<Point3D> pontos, GLuint buffer, float r, float g, float b) {
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glColor3f(r, g, b);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glPolygonMode(GL_FRONT_AND_BACK, tipo);

    switch (tipo) {
    case GL_FILL:
    case GL_LINE:
    case GL_POINT:
        glDrawArrays(GL_TRIANGLES, 0, pontos.size() * 3);
        break;
    default:
        std::cerr << "Modo de desenho desconhecido." << endl;
        break;
    }

    glBindBuffer(GL_ARRAY_BUFFER, 0); // Desvincula o buffer para evitar estado indesejado
}
```

Figura 22 - Função drawTriangles

A função drawGrupo também foi atualizada para fornecer o buffer associado a cada modelo ao chamar a função drawTriangles.

```
void drawGrupo(Grupo gR) {


    std::vector<Grupo> filhos = gR.filhos;
    std::vector<Modelo> m = gR.modelos;
    std::vector<Transformacao> t = gR.transforms;

    glPushMatrix();

    for (int i = 0; i < t.size(); i++) {
        applyTransformations(t[i]);
    }


    for (int i = 0; i < m.size(); i++) {
        drawTriangles(m[i].pontos, (m[i].buffer ), m[i].r, m[i].g, m[i].b);
        // std::cout << m[i].buffer << std::endl;
    }

    for (int i = 0; i < filhos.size(); i++) {
        drawGrupo(filhos[i]);
    }

    glPopMatrix();

}
```

Figura 23-drawGrupo

Alteração nas transformações e curvas catmullRoom

Rotação baseada em tempo

Nesta nova fase as rotações tiveram de ser adaptadas para lidarem com mais um parâmetro, o time. O time representa o tempo que a figura demora a completar uma volta de 360º. Para tal, tivemos de adicionar o parâmetro “time” à nossa classe *Transformacao*.

```
class Transformacao {
public:
    int type;
    float angle;
    float x;
    float y;
    float z;
    float time = 0;
```

Figura 24- Classe Transformacao

Para o nosso programa poder receber e processar o novo parâmetro do ficheiro xml tivemos de adaptar a nossa função para ela consiga ler e guardar o time. Ao identificar um elemento de rotação no XML, verificamos se há um atributo "time" associado a ele. Se existir, o tempo é lido e atribuído à transformação correspondente.

```
if (strcmp(trf->Value(), "rotate") == 0) {
    float x = 0, y = 0, z = 0, angle = 0, time = 0;
    if (trf->Attribute("x") != nullptr) x = stof(trf->Attribute("x"));
    if (trf->Attribute("y") != nullptr) y = stof(trf->Attribute("y"));
    if (trf->Attribute("z") != nullptr) z = stof(trf->Attribute("z"));
    if (trf->Attribute("angle") != nullptr) angle = stof(trf->Attribute("angle"));
    Transformacao t = *new Transformacao(1, x, y, z, angle);

    if (trf->Attribute("time") != nullptr) { time = stof(trf->Attribute("time")); t.time = time; }

    transformacoesLista.push_back(t);
    grupo->transforms.push_back(t);
}
```

Figura 25 - Função para ler o ficheiro xml

Tendo o valor do time, calculamos o ângulo de rotação que tem de ser aplicado ao modelo a cada renderização na função *applyTransformations* que foi modificada para

tal.

A função *applyTransformations* é então chamada para cada objeto que precisa de ser rotacionado. Quando a rotação é dependente do tempo, em vez de ser aplicado um ângulo fixo, calculamos o ângulo atual com base no tempo percorrido desde o início da execução do programa.

Utilizamos a função *glutGet(GLUT_ELAPSED_TIME)* para obter o tempo decorrido em milissegundos, o que nos fornece a base para calcular a fração do movimento de rotação completo que deve ter ocorrido até o momento atual.

O resultado é um movimento de rotação contínuo que se repete a cada intervalo definido pelo time, criando uma animação cíclica que simula uma rotação continua.

```
case(1): // Rotate
    // Verificar se o ângulo de rotação está especificado.
    if (t.angle != 0) {
        // Aplicar uma rotação estática com o ângulo especificado ao redor do eixo (t.x, t.y, t.z).
        glRotatef(t.angle, t.x, t.y, t.z);
    }
    else {
        float aux, anguloRot;
        int tempoPrograma;
        // Verificar se o tempo para completar uma rotação de 360 graus está especificado.
        if (t.time != 0) {
            // Obter o tempo total decorrido desde o início do programa em milissegundos.
            tempoPrograma = glutGet(GLUT_ELAPSED_TIME);
            // Calcular o tempo decorrido na rotação atual modulando pelo tempo total de rotação em milissegundos.
            aux = tempoPrograma % (int)(t.time * 1000);
            // Converter o tempo decorrido em um ângulo de rotação proporcional.
            anguloRot = (aux * 360) / (t.time * 1000);
            // Aplicar uma rotação dinâmica com o ângulo calculado ao redor do eixo (t.x, t.y, t.z).
            glRotatef(anguloRot, t.x, t.y, t.z);
        }
    }
    break;
```

Figura 26- rotação

Translação com Curvas de Catmull-Rom

À semelhança da rotação, também a translação teve de ser adaptada, uma vez que, esta pode agora receber três novos parâmetros, sendo eles, o *time*, o *align* e os pontos que vão ser utilizados para a curva de *Carmull-Rom*.

O *time*, especifica o intervalo de tempo durante o qual o objeto deve percorrer o trajeto completo definido pela curva de *Catmull-Rom*. Este parâmetro é crucial para sincronizar a animação com o tempo de execução desejado, garantindo que a transição de um ponto a outro seja suave e contínua.

O *align* é a flag que determina se o objeto deve ou não alinhar-se com a tangente da curva em cada ponto do caminho. Quando ativado, este recurso garante que o objeto esteja sempre orientado de acordo com a direção do movimento, criando uma perceção mais natural e coerente do deslocamento.

Por fim, os pontos que compõem a curva de Catmull-Rom são necessários para definir o caminho específico que o objeto vai seguir. Através de um conjunto de pontos de controlo, a curva é gerada e o movimento do objeto é interpolado ao longo desta, resultando numa trajetória.

Posto isto, a classe Transformacao foi novamente expandida para guardar estes parâmetros, tal como se pode ver na seguinte figura :

```
#ifndef TRANSFORMACAO_H
#define TRANSFORMACAO_H

#include <vector>
#include "Modelo.h"

class Transformacao {
public:
    int type;
    float angle;
    float x;
    float y;
    float z;
    float time = 0;
    bool align = true;
    std::vector<Point3D> catmullRomPoints;


    Transformacao(); // Default constructor
    Transformacao(float type, float a, float b, float c, float angle = 0); // Parameterized constructor
};

#endif /* TRANSFORMACAO_H */
```

Figura 27- Classe Transformacao

Tendo tudo preparado para guardar os parâmetros, alteramos a nossa função *readGroup* para os ler e guardar caso existam, como se pode ver pela seguinte imagem:

```
if (strcmp(trf->Value(), "translate") == 0) {
    float x = 0, y = 0, z = 0;
    float time = 0;
    if (trf->Attribute("x") != nullptr) x = stof(trf->Attribute("x"));
    if (trf->Attribute("y") != nullptr) y = stof(trf->Attribute("y"));
    if (trf->Attribute("z") != nullptr) z = stof(trf->Attribute("z"));
    Transformacao t = *new Transformacao(0, x, y, z);

    if (trf->Attribute("time") != nullptr) { time = stof(trf->Attribute("time")); t.time = time; }

    if (trf->Attribute("align") != nullptr) {
        if (strcmp(trf->Attribute("align"), "false") == 0 || strcmp(trf->Attribute("align"), "False") == 0)
            t.align = false;
    }
```

Figura 28- readGroup

Desenhar a Curva

Para desenhar uma curva de catmull-rom, são necessários, no mínimo, 4 pontos de controlo. Esses pontos representam as posições por onde a nossa curva tem de passar.

De modo a desenhar a curva, temos de a dividir em vários segmentos mais pequenos, onde cada segmento é definido por 4 pontos de controlo, por exemplo nesta fase os testes que utilizamos só possuem 4 pontos de controlo então a nossa curva gerada por eles vai ser definida num único segmento.

Para conseguirmos então calcular os pontos que vão definir a nossa curva, utilizamos a seguinte fórmula matemática:

$$P(t) = \frac{1}{2}\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}\begin{bmatrix} -1 & 2 & -1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Figura 29 - Fórmula matemática para calcular os pontos da curva

Na fórmula, os parâmetros P0, P1, P2 e P3 são os pontos de controlo utilizados e substituímos o parâmetro t pelo seu valor correspondente. Este parâmetro, t, varia de 0 a 1 e é determinado pelo número de segmentos utilizado para gerar a nossa curva. Isto é, dividimos 1 pelo número de segmentos e o valor de t vai variando consoante o resultado.

Para garantir a precisão dos testes desta fase, dividimos o nosso segmento total em 20 pequenos segmentos.

```
int NUM_SEG = 20;
```

Figura 30 - Número de segmentos utilizados

Quando maior o número de segmentos, maior o nível de detalhe e mais suave é a curva gerada.

Tendo por base a operação matemática referida acima, na figura 26, começamos por desenvolver o ficheiro '*catmull-rom*', que vai ser responsável pelos cálculos para gerar a nossa curva.

É lá que está definida a nossa função responsável por calcular os pontos para a nossa curva, sendo ela a '*getGlobalCatmullRomPoint*'.

```
void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv, std::vector<Point3D> pontos) {

    int tamLoop = pontos.size(); // Points that make up the loop for catmull-rom interpolation
    float t = gt * tamLoop; // this is the real global t
    int index = floor(t);  // which segment
    t = t - index; // where within  the segment

    // indices store the points
    int indices[4];
    indices[0] = (index + tamLoop - 1) % tamLoop;
    indices[1] = (indices[0] + 1) % tamLoop;
    indices[2] = (indices[1] + 1) % tamLoop;
    indices[3] = (indices[2] + 1) % tamLoop;

    getCatmullRomPoint(t, pontos[indices[0]], pontos[indices[1]], pontos[indices[2]], pontos[indices[3]], pos, deriv);
}
```

Figura 31 - Função para calcular os pontos da nossa curva

Este método vai ser chamado pela função *'applyTransformation'*, no *engine*, para calcular de forma iterativa todos os pontos necessários para a curva.

```
case(0): // 0 - Translate
    if (t.time != 0 && t.catmullRomPoints.size() >= 4) { // se for uma curva de catmull rom
        float pos[3];
        float deriv[3];


        // ---------------------- Build and Draw curve

        int NUM_SEG = 20;
        float te = 0.0f, inc = 1.0f / NUM_SEG;

        getGlobalCatmullRomPoint(te, pos, deriv, t.catmullRomPoints);
        // draw curve using line segments with GL_LINE_LOOP
        glColor3f(1.0f, 1.0f, 1.0f);
        glBegin(GL_LINE_LOOP);
        for (int i = 0; i < NUM_SEG; i++, te += inc) {
            getGlobalCatmullRomPoint(te, pos, deriv, t.catmullRomPoints);
            glVertex3f(pos[0], pos[1], pos[2]);
        }
        glEnd();
```

Figura 32 - trecho função 'applyTransformation'

Desta forma, caso no xml o parâmetro '*translate*' possuir os dados time e pontos de controlo suficientes, ele vai calcular a respetiva curva de *catmull-rom*.

Translações na curva

A translação na curva de *catmull-rom*, vai ser utilizada para movimentar um determinado objeto ao longo dessa curva. Para tal, temos de calcular o ponto seguinte da nossa curva, através da função '*getGlobalCatmullRomPoint',* para o qual o nosso objeto se vai deslocar, determinando assim, a translação do objeto ao longo da trajetória.

Cálculo do ponto para onde o objeto se desloca a seguir

Ao contrário do cálculo efetuado para a curva, o parâmetro 't' é calculado tendo por base o '*time*' fornecido pelo ficheiro xml. Desta forma, conseguimos calcular o ponto para onde a nossa figura tem de se seguir.

```
float time = glutGet(GLUT_ELAPSED_TIME) % (int)(t.time * 1000) / (t.time * 1000);
getGlobalCatmullRomPoint(time, pos, deriv, t.catmullRomPoints);
```

Figura 33 - Função para calcular o ponto seguinte da curva

Tendo o ponto seguinte calculado, utilizamos a função '*glTranslatef*' para mover o objeto para essa nova posição, resultando na translação do objeto ao longo da curva.

```
glTranslatef(pos[0], pos[1], pos[2]);
alinhamentoCurva(deriv);
```

Figura 34 - Função para movimentar o objeto ao longo da curva

Utilizamos ainda a função 'alimentoCurva', para garantir que o objeto se mantenha com uma orientação adequada em relação à curva, à medida que o mesmo se movimenta por ela.

```
void alinhamentoCurva(float* deriv) {

    float Z[3];
    float Yi[3] = { 0,1,0 };
    float X[3] = { deriv[0],deriv[1],deriv[2] };
    float m[16];
    float Y[3];

    cross(X, Yi, Z); // y antigo, Yi
    cross(Z, X, Y); // y novo, Y

    normalize(X);
    normalize(Y);
    normalize(Z);

    buildRotMatrix(X, Y, Z, m);

    glMultMatrixf((float*)m);

    //glutSwapBuffers();
    //
    Y[0] = Yi[0];
    Y[1] = Yi[1];
    Y[2] = Yi[2];
}
```

Figura 35 - Função 'alinharCurva'

Desta forma, conseguimos movimentar o nosso objeto ao longo da curva de catmull-rom, de uma forma suave e natural.

4. Resultados dos testes da fase 3

Para testar se todas estas alterações foram efetuadas corretamente, recorremos ao uso dos testes disponibilizados pela equipa docente, para esta terceira fase, obtendo os seguintes resultados:

4.1 Test_3_1

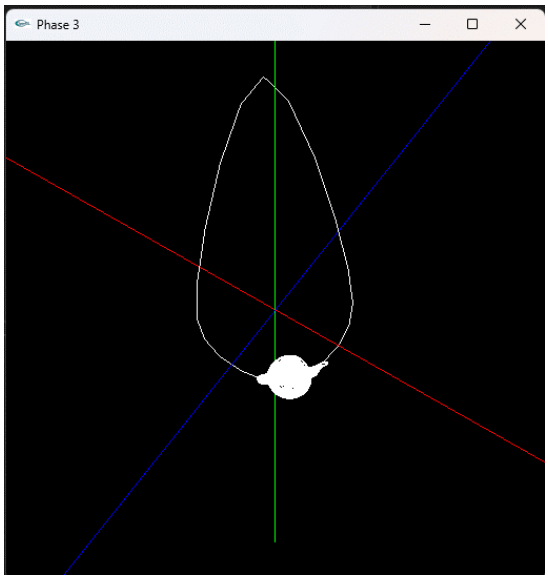


Figura 37- Resultado obtido na nossa execução

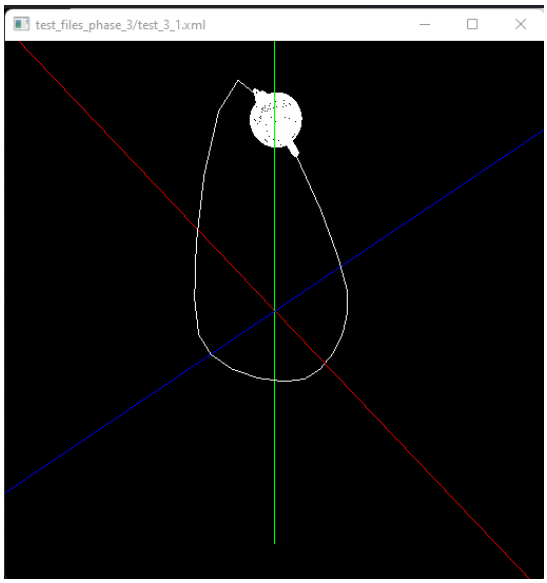


Figura 36- Teste dos professores

Note-se que como neste teste a chávena e a curva estão animadas, o grupo não conseguiu tirar um print no exato momento do teste dos professores, mas dá para perceber que estão iguais, mesmo pela demo *scence* enviada em anexo com o trabalho prático.

4.2 Test_3_2

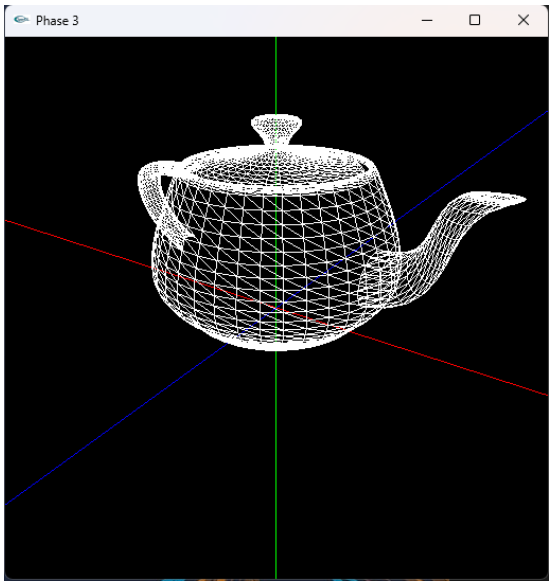


Figura 39-Resultado obtido na nossa execução

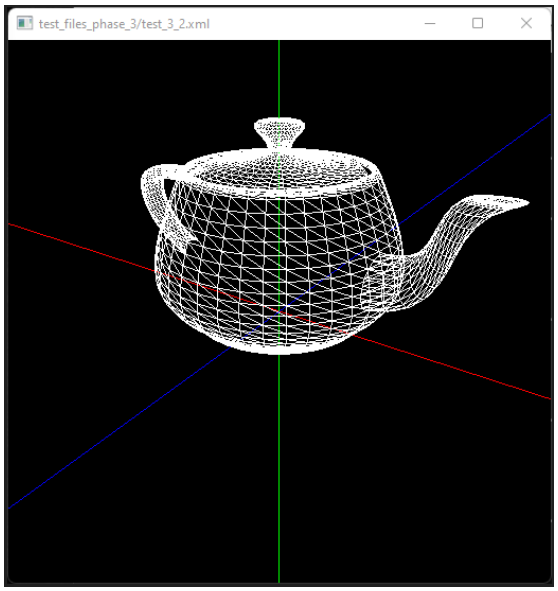


Figura 38- Teste dos professores

Discussão dos resultados obtidos

Como se pode observar, através da análise das figuras acima apresentadas, os resultados que obtivemos estão todos de acordo com os que foram fornecidos pelos professores. Portanto concluímos que as alterações, efetuadas no *Engine* para gerar as cenas, com a hierarquia e as transformações, estão de acordo com o pretendido e a funcionar corretamente.

4. Sistema Solar

Ao nível do Xml do sistema solar, como nos foi pedida uma demo animada tivemos de efetuar alterações no mesmo.

Agora, os planetas orbitam em torno do sol graças à aplicação de uma sequência de transformações. Primeiramente, utilizamos uma rotação combinada com um translate para determinar a órbita de cada planeta. Esta rotação inclui um parâmetro de tempo que especifica a duração necessária para completar uma órbita em torno do sol. Para traduzir esse tempo para a animação, adotamos uma escala onde consideramos que 365 dias correspondiam a 60 segundos.

Acrescentámos ainda que, para as distâncias e dimensões dos planetas, optamos por não seguir uma escala realista, apenas ajustamos os parâmetros visualmente para garantir uma melhor experiência de visualização. No entanto, mantivemos a ordem de grandeza dos planetas e a ordem de distância ao sol, garantindo assim a precisão relativa do sistema solar representado na nossa demonstração. Essa abordagem permitiu-nos equilibrar a fidelidade científica com a estética visual.

Além disso, adicionamos outra rotação para permitir que os planetas também girem em torno de seus próprios eixos, aumentando assim a fidelidade da simulação. No XML, incluímos ainda a representação do cometa, modelado como um teapot, utilizando pontos de Bézier para sua construção e uma curva de Catmull para definir sua trajetória.

5.Extras

Como um acréscimo aos recursos já existentes, como o foguete, o cometa e a estação espacial gerados anteriormente com as primitivas disponíveis em nosso generator, também implementamos uma curva de Catmull em cada para melhorar a trajetória destes elementos.

Sistema Solar imagens

As seguintes imagens dizem então respeito à nossa demo do sistema solar, onde são visíveis os nossos planetas e os nossos extras com as suas catmull rom curves.

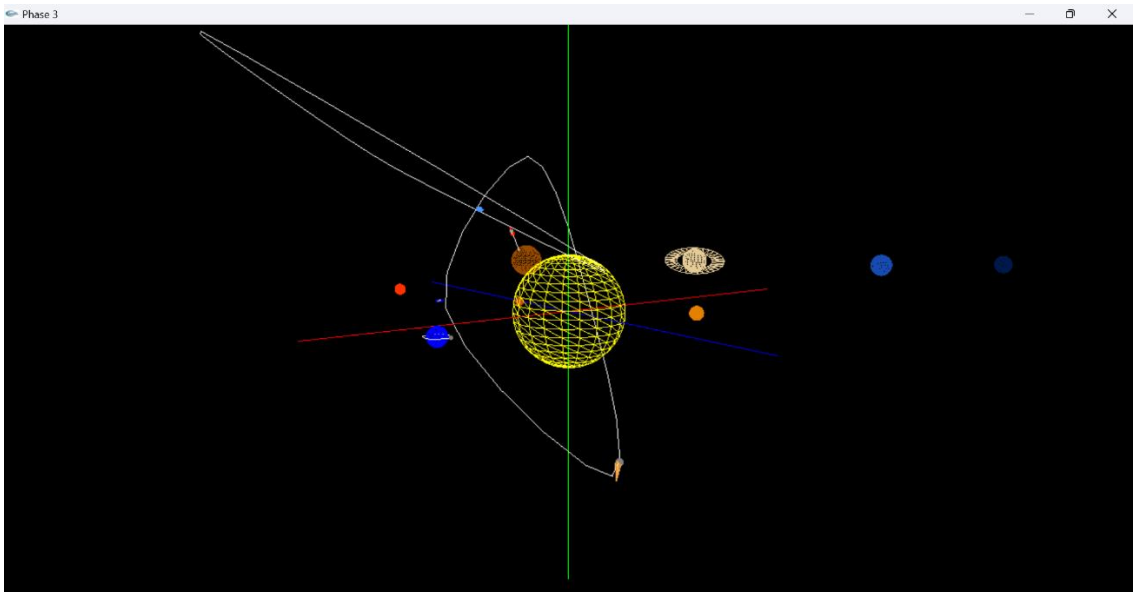


Figura 40- Sistema solar modo GLline

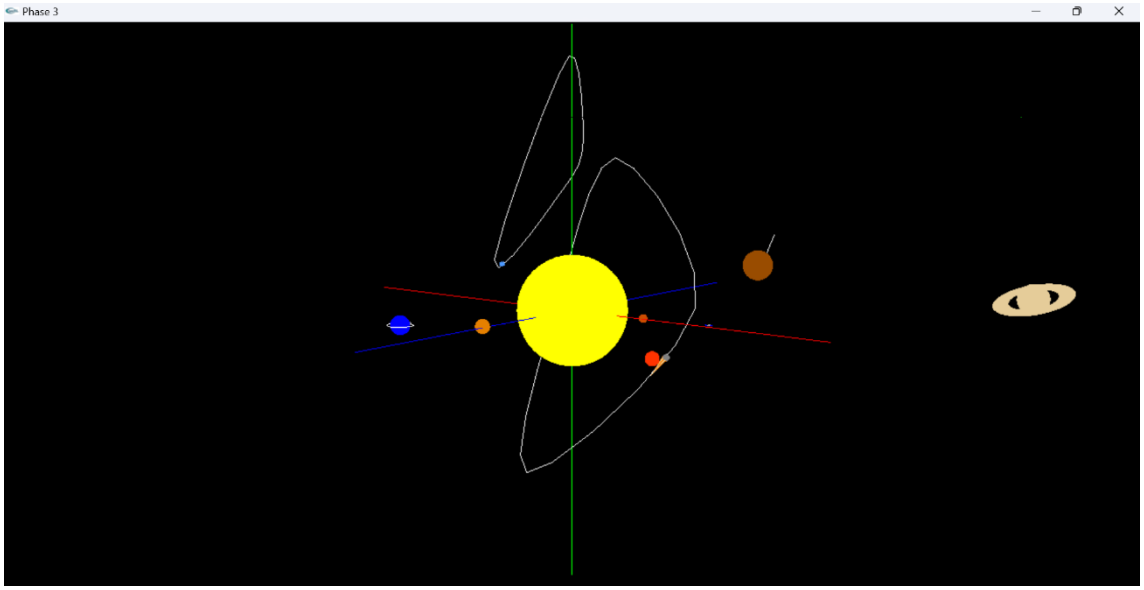


Figura 41-Sistema solar GLfill

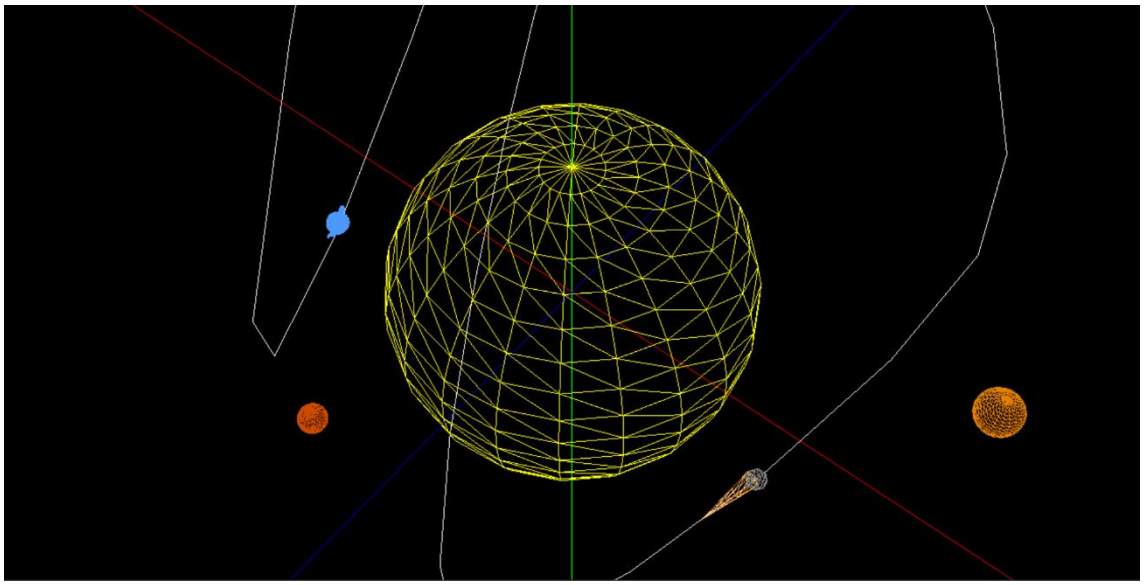


Figura 42- Cometa teapot

4. Conclusão

Durante a terceira fase do nosso projeto, conseguimos aprofundar o nosso conhecimento sobre as curvas de Bézier e Catmull-Rom.

As alterações no Generator possibilitaram a implementação de superfícies de Bézier no teapot.patch, melhorando significativamente a qualidade visual da figura através de superfícies mais suaves e realistas.

Adicionalmente, melhoramos o engine para suportar novas funcionalidades de rotação e translação, incluindo a definição do tempo para uma rotação completa e a adição de curvas Catmull-Rom para movimentação ao longo de uma trajetória específica. Apesar dos desafios enfrentados no manuseamento dos buffers, superámo-los e assegurámos que todos os modelos fossem corretamente desenhados com VBOs.

A cena de demonstração desenvolvida, um sistema solar dinâmico, ilustra o resultado destas implementações, incluindo um o cometa Teapot cuja trajetória foi definida por uma curva Catmull-Rom.