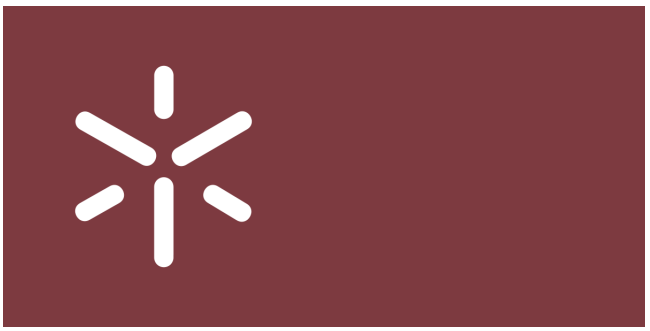

Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Projeto da U.C. Computação Gráfica

3ª Fase

Ano letivo 2023/2024

Realizado por:

A91672 - Luís Ferreira
A93258 - Bernardo Lima
A100543 - João Pastore
A100554 - David Teixeira

Conteúdo

Lista de Figuras

1 Introdução

A terceira fase do projeto da cadeira de Computação Gráfica envolveu a progressão do trabalho realizado nas fases anteriores, agregando novas funcionalidades às duas aplicações desenvolvidas, o *Generator* e o *engine*. As atualizações incluíram a capacidade do *generator* de produzir modelos baseados em patches de *Bezier*, bem como a capacidade do *Engine* de interpretar e aplicar novos tipos de translações e rotações. Além disso, o *Engine* também adotou o uso de *Vertex Buffer Objects* (VBOs) para proporcionar um desempenho visual aprimorado na renderização dos modelos gerados.

Os novos tipos de translações e rotações permitem a animação dessas transformações geométricas. As rotações podem ser estáticas ou dependentes do tempo da animação, enquanto as translações podem ser orientadas por curvas de *Catmull-Rom*.

Neste relatório, será apresentada uma descrição detalhada das decisões e abordagens adotadas para a implementação das funcionalidades propostas.

2 Parser

Grande parte das modificações requeridas na terceira fase do projeto (como referido anteriormente) incidiram sobre o *Engine* e *Generator*, com o intuito de incorporar novas funcionalidades e melhorias significativas na renderização e animação de modelos.

No *Generator*, foi implementada a criação de modelos com base em patches de Bézier. No *Engine*, as alterações incluiram a extensão dos elementos de translação e rotação para suportar curvas de *Catmull-Rom* e animações baseadas em tempo. Além disso, a renderização dos modelos foi melhorada com o uso de Vertex Buffer Objects (**VBOs**). Inicialmente, para alcançar este objetivo, foi essencial alterar estruturas de dados para armazenar todas as informações pertinentes do novo tipo de ficheiro *XML* e, por conseguinte, gerar as cenas conforme especificado.

2.1 Estruturas de Dados Alteradas

O *parser* foi atualizado para lidar com novos tipos de transformação, que incluem animações e transformações mais complexas. A `struct Transform` foi expandida para acomodar essas mudanças, e novos campos foram adicionados para capturar informações adicionais sobre as transformações. Eis o novo formato:

```
struct Transform {
    char type;
    float x;
    float y;
    float z;
    float angle;
    float time;
    bool align;
    std::vector<Position> points;
};
```

- `float time`: Representa o número de segundos para percorrer toda a curva.
- `bool align`: Indica se o objeto deve ser alinhado com a curva.
- `std::vector<Position> points`: Um conjunto de pontos fornecido para definir uma curva de *Catmull-Rom*.

Estas adições permitem uma representação mais flexível e detalhada das transformações, tornando o *parser* capaz de lidar com uma variedade mais ampla de cenas e animações.

2.2 *Parsing*

Em relação ao *parsing* de um ficheiro *XML*, foram necessárias alterações apenas no método: `void parseTransform()`, para que os novos campos da `struct Transform` fossem corretamente preenchidos.

3 Generator

Para gerar superfícies de Bézier, utilizou-se a função *generateBezierSurface*. Esta função aceita três argumentos principais: o caminho para um ficheiro de extensão ".patch"(*patchFilePath*), o nome de um ficheiro de saída (*outputFile*) e um parâmetro de tesselação (*tessellation*). A função produz um ficheiro com a extensão ".3d", que contém todas as informações necessárias para gerar uma superfície de Bézier. A função *generateBezierSurface* implementa várias estratégias chave para a criação de superfícies de Bézier:

- **Definição de Ponto:** Utiliza a estrutura *Ponto* para representar tanto os pontos de controlo quanto os pontos na superfície. Cada ponto é definido por coordenadas tridimensionais (x, y, z).
- **Função bezier:** Esta função calcula um ponto na superfície de Bézier utilizando os parâmetros u e v para interpolação bi-dimensional.

```
Ponto bezier(float u, float v, std::vector<Ponto>& controlPoints, std::vector<int>& indices) {
    std::vector<Ponto> tempPoints(4);
    for (int i = 0; i < 4; i++) {
        tempPoints[i] = formulae(u,
            controlPoints[indices[4 * i]],
            controlPoints[indices[4 * i + 1]],
            controlPoints[indices[4 * i + 2]],
            controlPoints[indices[4 * i + 3]]);
    }
    Ponto result = formulae(v, tempPoints[0], tempPoints[1], tempPoints[2], tempPoints[3]);
    for (int i = 0; i < 4; i++) {
        deletePonto(tempPoints[i]);
    }
    return result;
}
```

Figura 1: Função Bezier

Entradas: Recebe os parâmetros u e v, um vetor de pontos de controlo controlPoints e um vetor de índices.

Processamento: Primeiro, interpola linearmente entre grupos de quatro pontos de controlo para gerar quatro pontos temporários (*tempPoints*) usando a função $formulae$ e valores u. Em seguida, utiliza a função $formulae$ novamente, mas agora com o valor v para interpolar entre os pontos temporários, resultando no ponto final na superfície. Após calcular o ponto resultante, a função limpa a memória dos pontos temporários e devolve o ponto na superfície de Bézier.

- **Função formulae:** Calcula um ponto numa curva de Bézier cúbica para um dado parâmetro t usando a combinação linear dos pontos de controlo.

```
Ponto formulae(float t, Ponto point1, Ponto point2, Ponto point3, Ponto point4) {
    float aux = 1.0 - t;
    float pt1 = aux * aux * aux;
    float pt2 = 3 * aux * aux * t;
    float pt3 = 3 * aux * t * t;
    float pt4 = t * t * t;
    float x = pt1 * getX(point1) + pt2 * getX(point2) + pt3 * getX(point3) + pt4 * getX(point4);
    float y = pt1 * getY(point1) + pt2 * getY(point2) + pt3 * getY(point3) + pt4 * getY(point4);
    float z = pt1 * getZ(point1) + pt2 * getZ(point2) + pt3 * getZ(point3) + pt4 * getZ(point4);
    return newPonto(x, y, z);
}
```

Figura 2: Função Formulae

Entradas: O parâmetro t e quatro pontos de controlo ($point1$, $point2$, $point3$, $point4$).

Processamento: Utiliza a fórmula de Bernstein para calcular os coeficientes das bases polinomiais de Bézier. Aplica esses coeficientes para calcular as coordenadas x, y, e z do ponto na curva de Bézier. Devolve o ponto calculado na curva.

- **Função generateBezierSurface:** Esta função gera a malha de uma superfície de Bézier lendo a configuração dos patches de um ficheiro e escrevendo os pontos resultantes num outro ficheiro.

```
void generateBezierSurface(const std::string& patchFilePath, const std::string& outputFileName, int tessellation) {
    namespace fs = std::filesystem;

    // Construindo o caminho completo para o arquivo de saída na pasta 'output'
    fs::path outputFilePath = fs::current_path() / "../output" / outputFileName;

    std::ifstream patchFile(patchFilePath);
    if (!patchFile.is_open()) {
        std::cerr << "Erro ao abrir arquivo de entrada!" << std::endl;
        return;
    }

    std::ofstream outputFile(outputFilePath);
    if (!outputFile.is_open()) {
        std::cerr << "Erro ao abrir arquivo de saída no caminho: " << outputFilePath << std::endl;
        return;
    }

    std::string line;
    getline(patchFile, line);
    int numPatches = std::stoi(line);

    std::vector<std::vector<int>> patches(numPatches);
    for (int i = 0; i < numPatches; ++i) {
        getline(patchFile, line);
        std::istringstream iss(line);
        std::vector<int> indices(16);
        for (int j = 0; j < 16; ++j) {
            iss >> indices[j];
            if (iss.peek() == ',')
                iss.ignore();
        }
        patches[i] = indices;
    }

    getline(patchFile, line);
    int numControlPoints = std::stoi(line);
    std::vector<Ponto> controlPoints(numControlPoints);
    for (int i = 0; i < numControlPoints; ++i) {
        float x, y, z;
        getline(patchFile, line);
        std::replace(line.begin(), line.end(), ',', ' ');
        std::istringstream iss(line);
        iss >> x >> y >> z;
        controlPoints[i] = newPonto(x, y, z);
    }

    std::stringstream ss;
    float step = 1.0f / tessellation;
    int totalVertices = 0;

    for (auto& patch : patches) {
        for (float u = 0; u < 1.0f; u += step) {
            for (float v = 0; v < 1.0f; v += step) {
                Ponto p1 = bezier(u, v, controlPoints, patch);
                Ponto p2 = bezier(u + step, v, controlPoints, patch);
                Ponto p3 = bezier(u, v + step, controlPoints, patch);
                Ponto p4 = bezier(u + step, v + step, controlPoints, patch);

                ss << getX(p1) << "," << getY(p1) << "," << getZ(p1) << "\n";
                ss << getX(p2) << "," << getY(p2) << "," << getZ(p2) << "\n";
                ss << getX(p3) << "," << getY(p3) << "," << getZ(p3) << "\n";
                ss << getX(p2) << "," << getY(p2) << "," << getZ(p2) << "\n";
                ss << getX(p4) << "," << getY(p4) << "," << getZ(p4) << "\n";
                ss << getX(p3) << "," << getY(p3) << "," << getZ(p3) << "\n";

                deletePonto(p1);
                deletePonto(p2);
                deletePonto(p3);
                deletePonto(p4);

                totalVertices += 6;
            }
        }
    }

    outputFile << totalVertices << "\n" << ss.str();
    patchFile.close();
    outputFile.close();
    std::cout << "Arquivo '" << outputFileName << "' criado com sucesso em: " << outputFilePath << std::endl;
}
```

Figura 3: Função GenerateBezierSurface

Entradas: Caminho para o ficheiro de patches ($patchFilePath$), nome do ficheiro de saída ($outputFileName$), e o número de tesselações ($tessellation$).

Processamento: Lê o ficheiro de patches para obter os índices dos patches e os pontos de controlo. Para cada patch, calcula pontos na superfície para valores incrementados de u e v baseados na tesselação especificada. Utiliza a função *bezier* para calcular os vértices da malha. Os vértices são organizados em triângulos e armazenados num *stringstream* para serem escritos posteriormente no ficheiro de saída. Por fim, é criado um ficheiro que contêm os vértices da malha da superfície de Bézier.

Este processo permite a criação de superfícies de Bézier altamente detalhadas e precisas, adequadas para aplicações em design gráfico, animações e engenharia, onde superfícies suaves e precisamente controladas são essenciais.

4 Engine

4.1 VBO's

Nesta etapa do projeto de Computação Gráfica, foi-nos solicitada a implementação de VBOs (**Vertex Buffer Objects**) com o intuito de melhorar o desempenho na renderização de primitivas, permitindo a exibição de um maior número de objetos e de maior complexidade, através da utilização de buffers de vértices armazenados na memória da GPU. Os VBOs representam uma abordagem eficiente para armazenar e manipular dados de vértices, reduzindo a carga de comunicação entre a unidade central de processamento (CPU) e a unidade de processamento gráfico (GPU) pois existe um menor overhead no CPU devido a existirem menos chamadas ao mesmo. Com esta implementação, beneficiamos de renderização em lotes de geometrias, redução do consumo de memória do sistema e suporte para mudanças dinâmicas. Este conjunto de vantagens permite a criação de cenas mais elaboradas e interativas, ao mesmo tempo que otimiza o desempenho geral do projeto.

4.1.1 Inicialização

No início do programa, um array de IDs de buffer (`bufferId`) é criado para armazenar os IDs dos VBOs. Além disso, um array `info` é utilizado para armazenar o tamanho de cada buffer.

```
GLuint *buffers = NULL;
int info[100];
unsigned int figCount = 0;
GLuint bufferId[100];
```

4.1.2 Criação dos VBO's

A função `importFiguras` é responsável por criar VBOs para cada figura. Itera através da lista de figuras, gerando IDs de buffer utilizando `glGenBuffers`. De seguida, obtém os pontos das figuras armazenadas e introduz num vetor. Seguidamente, a função utiliza `glBindBuffer` para vincular cada VBO atual à *target* (figura armazenada) adequada, preenche cada buffer com os dados de vértices usando `glBufferData`, e armazena o tamanho de cada buffer no array `info`. Esta abordagem foi adotada para evitar uma reestruturação completa do código, embora uma possível melhoria seria a de criar os VBOs diretamente, em vez de preencher a lista de figuras antes e só depois criá-los.

```
void importFiguras(List figs)
{
        figCount = getListLength(figs);

        for (unsigned long i = 0; i < getListLength(figs); i++)
        {
                glGenBuffers(i + 1, &bufferId[i]);
                Figura fig = (Figura)getListElemAt(figs, i);
                List figPontos = getPontos(fig);
                vector<float> vVertices;

                for (unsigned long j = 0; j < getListLength(figPontos); j++)
                {
```

```
                    Ponto point = (Ponto)getListElemAt(figPontos, j);
                    vVertices.push_back(getX(point));
                    vVertices.push_back(getY(point));
                    vVertices.push_back(getZ(point));
                }
                info[i] = (vVertices.size() / 3);

                glBindBuffer(GL_ARRAY_BUFFER, bufferId[i]);
                glBufferData(GL_ARRAY_BUFFER, vVertices.size() *
  sizeof(float), vVertices.data(), GL_STATIC_DRAW);
            }
}
```

4.1.3 Desenho de Figuras

A função `drawFigures` vincula o VBO apropriado utilizando `glBindBuffer`, configura os pointers de vértices utilizando `glVertexPointer`, e então desenha a figura utilizando `glDrawArrays`. O `startpost` e `endpos` são usados de forma a que só seja desenhado as figuras pertencentes ao grupo atual de modo a que todas tenham as mesmas transformações.

```
void drawFigures(int startpos, int endpos) {
    for (unsigned long i = startpos; i < endpos; i++) {
        glBindBuffer(GL_ARRAY_BUFFER, bufferId[i]);
        glVertexPointer(3, GL_FLOAT, 0, 0);
        glDrawArrays(GL_TRIANGLES, 0, info[i]);
    }
}
```

4.2 Transformações

Uma das adições significativas nesta fase foi a implementação de transformações ao longo de um período de tempo como translações e rotações.

4.2.1 Rotações

Para as rotações, simplesmente adicionamos o cálculo necessário para saber qual o valor do angulo de rotação a cada *frame* renderizado usando a seguinte verificação no código e o cálculo:

```
if (r_time > 0.0f)
{
    r_angle = ((NOW - init_time) * 360.0f) / r_time;
}
glRotatef(r_angle, x, y, z);
```

Este código indica que se o campo `r_time` tiver um valor diferente de zero será necessário fazer uma rotação ao longo do tempo.

A expressão `(NOW - init_time)` calcula o tempo decorrido desde o início da transformação, onde `NOW` representa o tempo atual e `init_time` representa o tempo de início da transformação.

De seguida, o tempo decorrido é dividido pelo tempo total de rotação (`r_time`) especificado para a transformação. Esta divisão fornece a fração do tempo total que passou desde o início da transformação.

Multiplicando esta fração por 360 graus, obtemos o ângulo de rotação correspondente ao tempo decorrido, garantindo uma rotação completa ao longo do tempo especificado.

Finalmente, a função `glRotatef` é usada para aplicar a rotação ao objeto renderizado, onde o ângulo de rotação calculado é usado como parâmetro, juntamente com os eixos de rotação especificados por `x`, `y` e `z`.

Esta abordagem permite que objetos rodem suavemente ao longo do tempo, adicionando dinamismo e fluidez às animações e cenas renderizadas.

4.2.2 Translações

Uma das adições significativas nesta fase foi a implementação do suporte para Catmull-Rom splines. Esta adição permite uma interpolação suave da trajetórias dos objetos ao longo do tempo.

Primeiramente, pegamos nos valores da rota que queremos criar e introduzimos as coordenadas `x`, `y` e `z` num vetor para cada uma delas. Depois, calculamos novamente um `t` de forma parecida ao da rotação de modo a determinar a posição ao longo da curva. Esse parâmetro `t` varia de 0 a 1 e representa a posição relativa entre dois pontos consecutivos da curva.

Após este processo e também inicializarmos duas listas, enviamos tudo para a função `GlobalCatRomPoint` aonde `pos[]` é preenchido com as posições da figura naquele momento e `deriv[]` com as derivadas da curva nesse ponto, o que é útil para operações de alinhamento.

A função `GlobalCatRomPoint` opera da seguinte forma: uma vez fornecidos pelo menos 4 pontos e o tempo calculado, utiliza duas matrizes específicas para calcular a coordenada atual do ponto e a sua derivada. Esses cálculos são facilitados através de outras funções que auxiliam no processo de manipulação de matrizes.

$$p(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$p'(t) = [3t^2 \quad 2t \quad 1 \quad 0] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Finalmente, caso o campo de alinhamento esteja ativado (definido como *true*), os eixos do objeto são calculados usando a derivada e o vetor normal do objeto. Após multiplicar e normalizar os vetores, uma matriz é criada com esses vetores e multiplicada à matriz do objeto usando a função `glMultMatrixf`. Isto garante que o objeto assuma a orientação correta.

5 Sistema Solar

Para finalizar, foi-nos pedido que, além de verificar os resultados do nosso trabalho através dos ficheiros de teste fornecidos pelo professor, também criássemos o nosso próprio ficheiro .xml com uma representação do sistema solar.

Para tal efeito, importamos várias vezes a figura esfera, uma vez o "teapot", e aplicamos as transformações necessárias para animar o sistema de uma forma minimamente realista.

Todos os planetas foram definidos com uma rotação ao longo do tempo seguida de uma translação para simular a órbita em torno do sol, seguida de outra rotação para simular a própria rotação do planeta sobre o seu eixo.

No caso da lua, além dessas transformações serem aplicadas, dado que está presente no mesmo grupo que o planeta Terra, é aplicada outra translação e rotação para simular o movimento em torno da Terra e a sua própria rotação sobre o seu eixo.

Por fim, utilizamos as curvas de Catmull-Rom para dar uma trajetória ao cometa (que neste caso é a figura teapot).

Exemplo do ficheiro XML de planeta:

```
<!-- Earth -->
<group>
    <transform>
        <rotate time="30" x="0" y="1" z="0" />
        <translate x="55" y="0" z="0"/>
        <rotate time="5" x="0" y="1" z="0" />
    </transform>
    <models>
        <model file="../output/sphere.3d" />
    </models>

<!-- Moon -->
    <group>
        <transform>
            <translate x="5" y="0" z="0"/>
            <rotate time="5" x="0" y="1" z="0" />
            <scale x="0.273" y="0.273" z="0.273"/>
        </transform>
        <models>
            <model file="../output/sphere.3d" />
        </models>
    </group>
</group>
```

Exemplo do ficheiro XML do cometa:

```
<!— Comet —>
<group>
    <transform>
        <translate time="50" align="true">
            <point x="0" y="-10" z="-80" />
            <point x="-30" y="0" z="-50" />
            <point x="-40" y="0" z="-10" />
            <point x="0" y="10" z="100" />
            <point x="60" y="0" z="0" />
        </translate>
        <rotate time="10" x="0" y="1" z="0" />
        <scale x="0.3" y="0.3" z="0.3" />
    </transform>
    <models>
        <model file="../output/bezier_10.3d" />
    </models>
</group>
```

5.1 Resultados

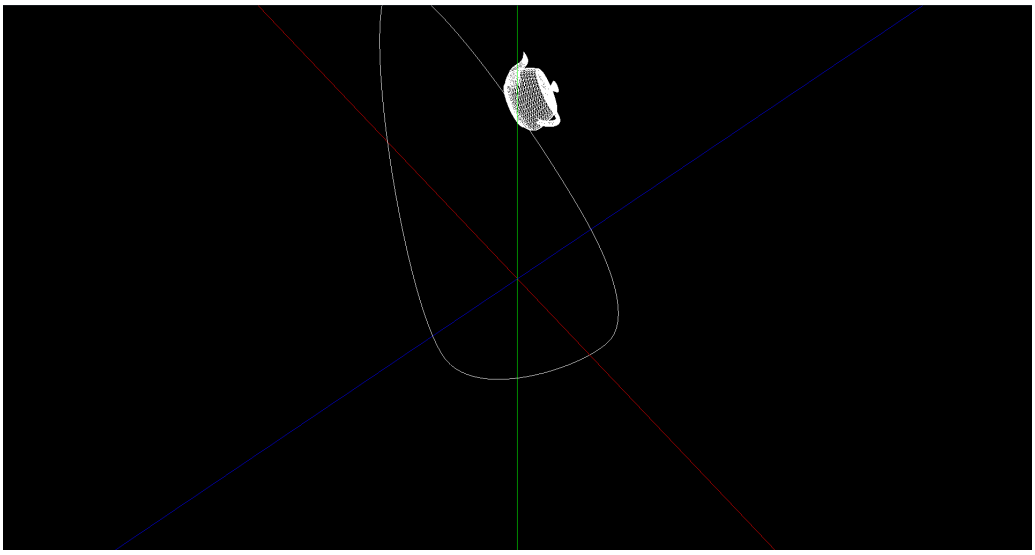

Figura 4: test_3_1.xml

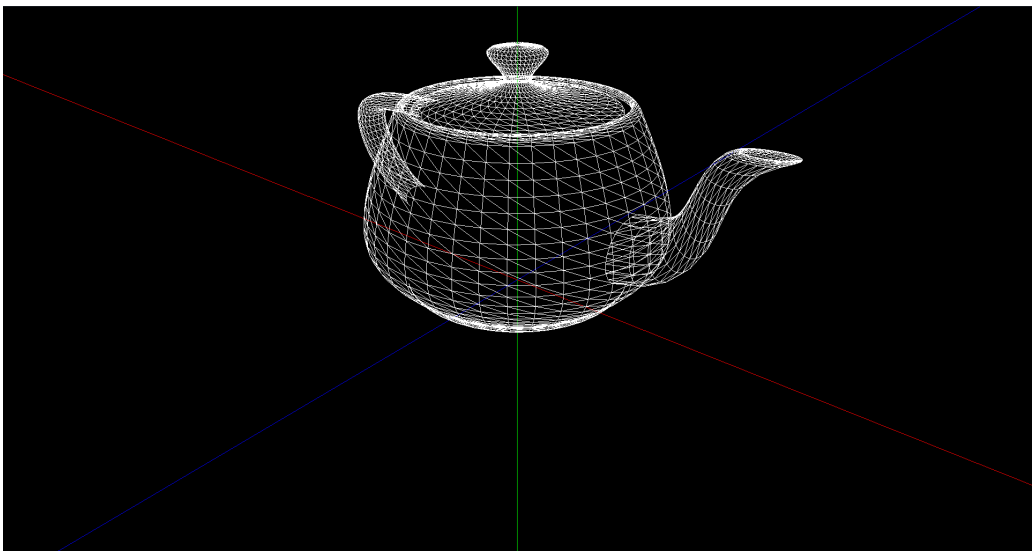

Figura 5: test_3_2.xml

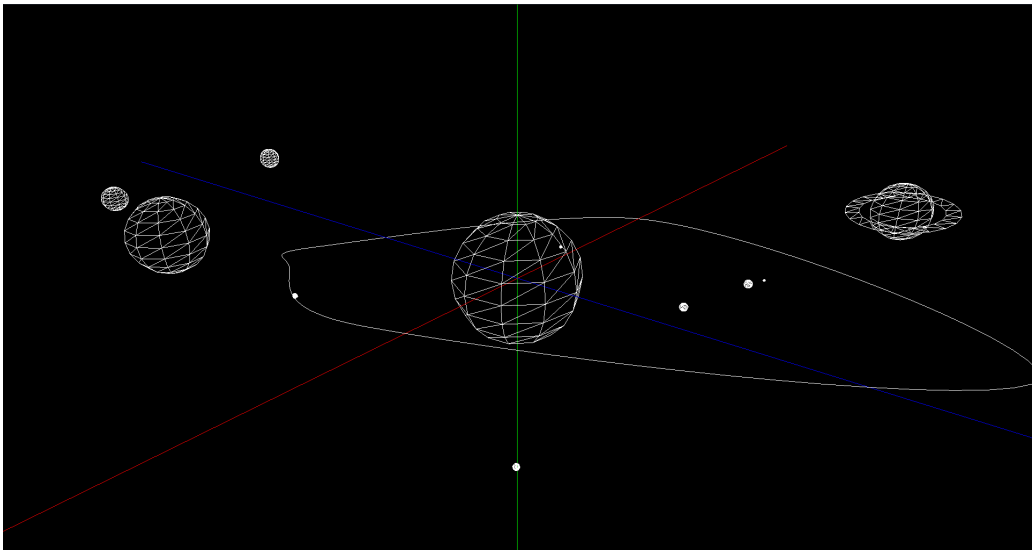

Figura 6: sistemasolar.xml

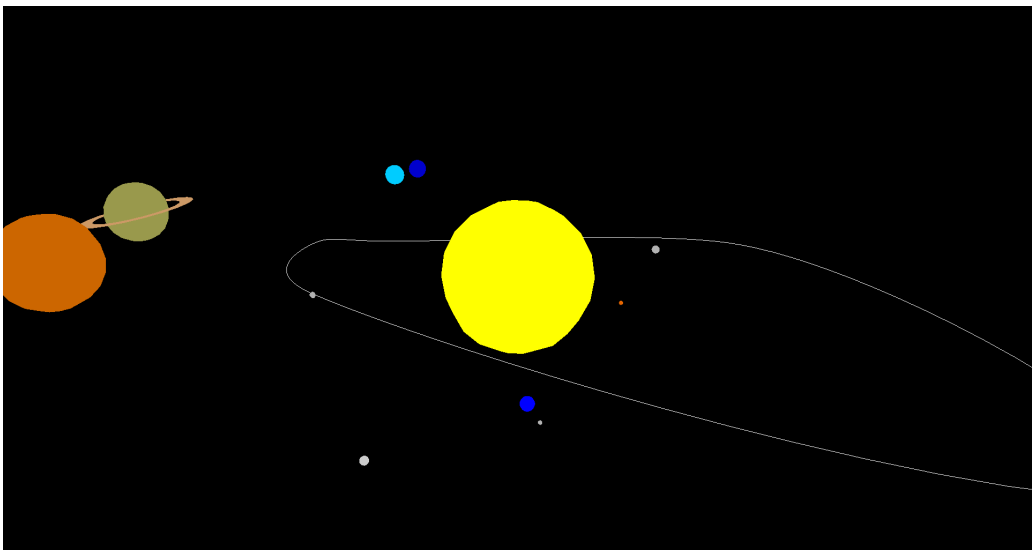

Figura 7: sistemasolar.xml (com cor)

6 Conclusão

Nesta fase do trabalho, melhorámos significativamente o *Engine* e o *Generator*, introduzindo novas funcionalidades e otimizações para a renderização e animação de modelos 3D. Implementámos a geração de modelos com base em patches de Bézier no *Generator* e ampliámos o suporte para curvas de Catmull-Rom e animações baseadas no tempo no *Engine*. Além disso, introduzimos o uso de Vertex Buffer Objects (VBOs) para melhorar o desempenho de renderização. Estas melhorias possibilitaram uma representação mais flexível e detalhada das transformações, conferindo dinamismo e fluidez às animações e cenas renderizadas. Como prova de conceito, criámos com sucesso um sistema solar em 3D utilizando ficheiros XML, demonstrando as capacidades desenvolvidas. Esta fase foi fundamental para a futura implementação de texturas e iluminação.