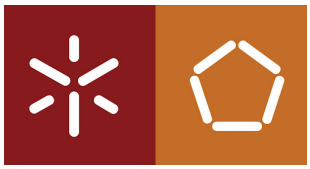
Universidade do Minho

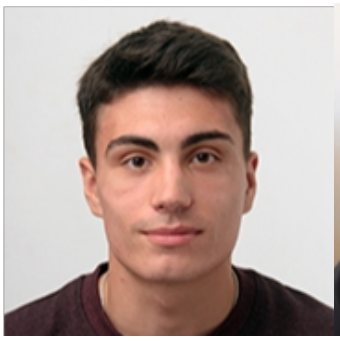
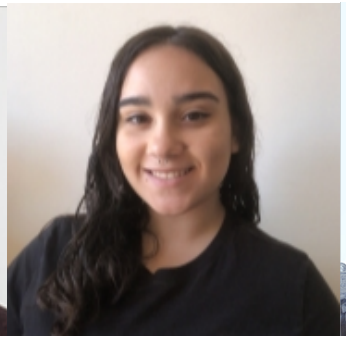
Departamento de Informática



licenciatura em engenharia informática

Computação gráfica - 3ª fase

Grupo 38

Henrique Pereira | Mariana Morais | Ana Alves | Simão Antunes

Henrique Morais Pereira A100831
Mariana Filipa Morais Gonçalves A100662
Ana João da Rocha Alves A95128
Simão Pedro Ferreira Antunes A100597

Abril de 2024

Conteúdo

1 Introdução

A unidade curricular de Computação Gráfica propõe, recorrendo à biblioteca GLUT, a utilização do OpenGL para a construção de modelos 3D com recurso à linguagem de programação C++. Pretende-se, assim, que o relatório sirva de suporte ao trabalho realizado para esta fase, mais propriamente, dando uma explicação e elucidando o conjunto de decisões tomadas ao longo da construção de todo o código fonte e descrevendo a estratégia utilizada para a concretização dos principais objetivos propostos.

Com a criação dos modelos propostos no enunciado pretendemos consolidar os conceitos abordados nas aulas, mais propriamente a aplicação de curvas e implementação de VBO's.

Posto isto, temos de forma muito breve o assunto do presente relatório que se refere à fase 3 de um projeto dividido em 4 fases, tal como indicado pela equipa docente.

2 Descrição do Problema

2.1 Fase 3

A terceira fase deste projeto tem como objetivo o desenvolvimento de novas funcionalidades aplicadas tanto ao Engine como ao Generator desenvolvidos anteriormente. Estes agora devem conter a implementação de curvas de Catmull-Rom e de Bezier respetivamente. Para além disso incluir-se o uso de VBO's.

2.1.1 Generator

O *Generator* tem como requisito os seguintes pontos:

- Ler um arquivo que contém os pontos de controlo de Bezier.
- Receber como parâmetros o nome desse arquivo e o nível de tesselation desejado.
- Utilizar os pontos de controlo de Bezier e o nível de tesselação para gerar uma superfície e dividir essa superfície em triângulos.
- Escrever os triângulos resultantes num ficheiro.

2.1.2 Engine

O *Engine* tem como requisito os seguintes pontos:

- Estender os elementos de translação e rotação da engine.
- Para a translação, utilizar pontos para definir uma curva de Catmull-Rom e o tempo para percorrê-la.
- Incluir um campo "align"para especificar se o objeto deve seguir a curva.
- Na rotação, permitir que o ângulo seja substituído pelo tempo para realizar uma rotação completa.
- Utilizar VBO's para desenhar os modelos, em vez do modo imediato.

2.1.3 Sistema Solar Dinâmico

- Criar uma cena de demonstração com um sistema solar dinâmico, incluindo um cometa com uma trajetória definida por uma curva de Catmull-Rom. O cometa deve ser construído usando patches de Bezier, como os pontos de controlo fornecidos para o teapot.

3 Resolução do Problema

3.1 Curvas de Bezier

Inicialmente, é extraído o número total de patches da primeira linha do arquivo e armazenamos esse valor, com o auxílio da função readFiles, na variável numberPatch.

De seguida, para cada linha seguinte que contém os índices dos patches, estes são convertidos em inteiros e os armazenados num vetor, que por sua vez contém os índices de todos os patches.

Após a leitura dos índices dos patches, as coordenadas dos pontos de controlo são lidas, convertendo cada conjunto de coordenadas (x, y, z) em valores de ponto flutuante. Essas coordenadas são então armazenadas como pontos 3D na struct Point e adicionadas ao vetor controlPoints, que armazena todos os pontos de controlo.

No final do processo de leitura, os vetores patches e controlPoints ficam preenchidos com as informações dos patches e dos pontos de controlo.

Para interpolar uma curva de Bezier, primeiro precisamos calcular os seus coeficientes. Isso é feito utilizando a fórmula de Bernstein, que envolve uma matriz de coeficientes binomiais. Essa matriz é pré-computada e armazenada para evitar cálculos repetitivos.

Após a leitura dos patches e pontos de controle, cada patch é transformado em uma matriz de controlo P, onde cada linha representa um ponto de controlo do patch. Em seguida, esta matriz P é multiplicada pela matriz de Bezier M:

$$matrix = M \times P$$

Essa operação é realizada para cada patch no vetor patches. A matriz M é fixa e definida pela expressão:

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Depois, a matriz resultante é multiplicada pela transposta de M:

$$matrix = matrix \times M^T$$

Com a nova matriz calculada, cada ponto da curva de Bezier é determinado. Isso é feito utilizando a parametrização da curva de Bezier:

$$p(u,v) = [u^3, u^2, u, 1] * matrix = \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Para cada combinação de valores de u e v dentro do intervalo [0, 1], os coeficientes u e v são aplicados à matriz matrix, e o resultado é um ponto na curva de Bezier. Esses pontos são calculados para uma malha de tesselation determinada.

Para cada ponto da curva de Bezier, os parâmetros "u"e "v"variam de acordo com o nível de tesselation recebido. Esse aumento é determinado pela seguinte equação:

$$step = \frac{1}{tesselation}$$

Isso significa que conforme a tesselation aumenta, o intervalo entre os pontos de Bezier diminui, resultando em uma representação mais detalhada da curva.

De seguida, e através da função multMatrixVector, o vetor "u"é multiplicado pela matriz matrix, resultando num novo vetor que virá a ser m multiplicado pelo vetor "v", resultando nas coordenadas (x, y, z) do ponto de Bezier.

Esses pontos de Bezier são então armazenados na matriz grid. Essa matriz é preenchida de acordo com a iteração sobre os valores de "u"e "v", formando uma grelha de pontos que representam a superfície aproximada pela curva de Bezier. Cada ponto na grade corresponde a um ponto de Bezier calculado a partir dos patches de controlo, como podemos ver em baixo:

$(0, 0)$	$(0.25, 0)$	$(0.5, 0)$	$(0.75, 0)$	$(1, 0)$
$(0, 0.25)$	$(0.25, 0.25)$	$(0.5, 0.25)$	$(0.75, 0.25)$	$(1, 0.25)$
$(0, 0.5)$	$(0.25, 0.5)$	$(0.5, 0.5)$	$(0.75, 0.5)$	$(1, 0.5)$
$(0, 0.75)$	$(0.25, 0.75)$	$(0.5, 0.75)$	$(0.75, 0.75)$	$(1, 0.75)$
$(0, 1)$	$(0.25, 1)$	$(0.5, 1)$	$(0.75, 1)$	$(1, 1)$

Tabela 1: Pontos de Bezier

Para se conseguir representar o teapot por triângulos é necessário recorrer à triangulação da grelha como está exemplificado na figura seguinte.

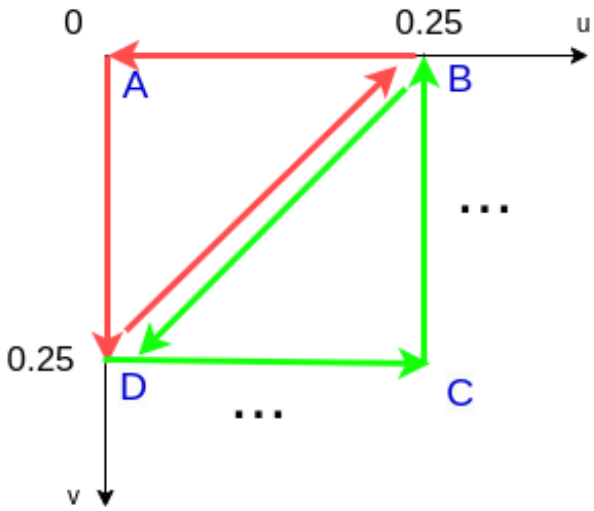


Figura 1: Triangulação da grelha

O valor de tesselation que é passado como parâmetro altera apenas a definição do modelo final.

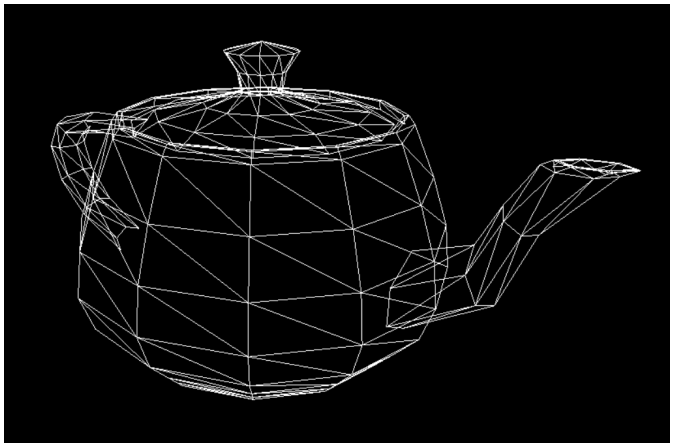


Figura 2: Valor de tesselation 3

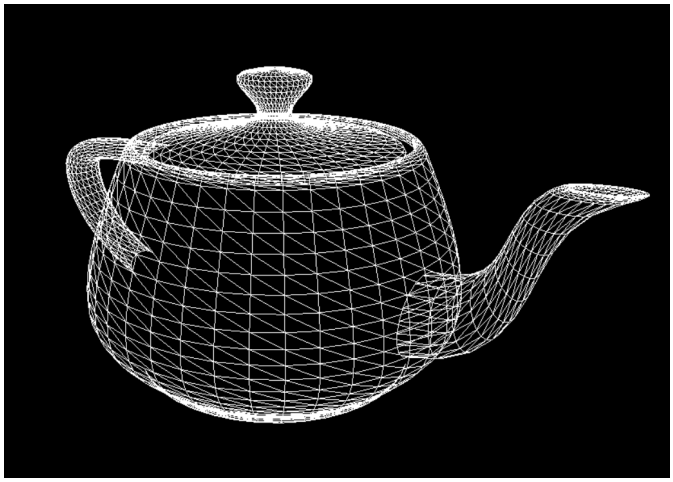


Figura 3: Valor de tesselation 10

3.2 Curvas de Catmull-Rom

3.2.1 Estruturas de dados

Primeiramente foi necessário estender a estrtutura Trasnformation para acomodar os novos valores presentes nos XML e modifcar a estruturas Group e criar uma nova estrutura World. Para isso criamos um módulo structures onde definimos todas as estruturas necessárias para esta fase.

```
struct Transformation {
    string name;
    float x;
    float y;
    float z;
    float angle;
    float time;
    bool align;
    vector<Point> points;
};
```

Figura 4: Struct Transformation

Assim, foram adicionados os campos *angle*, *time*, *align* e *points* à estrutura Transformation, necessários para podermos calcular e desenhar uma curva de Catmull-Rom.

A estrutura group também foi alvo de algumas alterações, nomeadamente a substituição da estrutura *Transformations* por um vetor de trasnformações e dos vetores *points* e *models* por um vetor de pares de inteiros para guadar o índice do VBO e o número de pontos necessários para o desenhar.

```
struct Group {
    vector<Transformation> transformations;
    vector<Group> subGroups;
    vector<pair<int,int>> vboIndexes;
};
```

Figura 5: Struct Group

Foi também criada uma nova estrutura o world para uma melhor organização e estruturação do código.

```
struct World {
    Window *win = new Window;
    Camera *cam = new Camera;
    vector<Group> groups;
};
```

Figura 6: Struct World

3.2.2 Leitura do ficheiro XML

Com as novas estruturas de dados prontas passamos para o parsing dos ficheiros XML. Para isso criamos um módulo parser.cpp onde definimos todas as funções relativas ao parsing de ficheiros. Assim, passamos a função loadPoints da Engine para este novo módulo, definimos uma função para a leitura da camara, da janela e modificamos as funções já existentes parseGroup e parseXMl. A lógica de ambas as funções mantém-se praticamente igual, exceto no que diz respeito à leitura dos modelos.

De maneira a evitar estar a carregar sempre o mesmo modelo caso ele esteja presente mais do que uma vez num dado ficheiro XML procedemos à implementação de um map onde usamos como key o nome do ficheiro.3d e guardamos lá um par de inteiros. Assim, quando a função parseGroup encontra um ficheiro.3d, ela primeiro verifica se o mesmo se encontra presente no map. Caso não esteja, chama a função loadPoints para ler os pontos e armazená-los num vetor. Esses mesmos pontos são posteriormente adicionados a outro vetor de floats passado como argumento à função parseGroups.

```
map<string, pair<int, int>> files;
int i = 0;

//
Group parseGroup(XMLElement* groupElement, vector<float> *vertices) {
    (...)
    XMLElement * modelsElement = groupElement -> FirstChildElement("models");
    if (modelsElement) {
        XMLElement * modelElement = modelsElement -> FirstChildElement("model");
        while (modelElement) {

            const char * filename = modelElement -> Attribute("file");
            string modelFile = "../../Models/";
            modelFile += filename;

            if (files.find(modelFile) != files.end()) {
                auto it = files.find(modelFile);
                group.vboIndexes.push_back(it->second);

            } else {
            //  Load points into vector
                vector<Point> points = loadPoints(modelFile);

            //  Insert in map to avoid loading the same model again
                files.insert({modelFile, pair<int,int>{i,points.size()}});

                group.vboIndexes.emplace_back(i, points.size());

                i += points.size();
                for (Point p : points){
                    vertices->push_back(p.x);
                    vertices->push_back(p.y);
                    vertices->push_back(p.z);
                }
            }

            modelElement = modelElement -> NextSiblingElement("model");
        }
    }
    (...)
    return group;
}
```

3.2.3 Desenho dos VBO's

Relativamente à utilização de VBOs, não alterou em grande escala a implementação que tínhamos até então, sendo necessárias apenas algumas modificações.

Assim, a nossa engine conta com duas variáveis globais, *verticesLoad* e *vertices*, um vetor de floats e um GLuint respetivamente. Estas são inicializadas através da função parseXML que vai guardar todos os pontos lidos num dado ficheiro XML no verticesLoad e através da função init, responsável por construir os buffers.

```
vector<float> verticesLoad;
GLuint vertices;

(...)

void init() {
    glewInit();
    glEnableClientState(GL_VERTEX_ARRAY);

//  Build the vertex arrays
    glGenBuffers(1, &vertices);
    glBindBuffer(GL_ARRAY_BUFFER, vertices);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * verticesLoad.size(), verticesLoad.data(), GL_STATIC_

//  OpenGL settings
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

(...)

int main(int argc, char** argv) {
    parseXML(argv[1], &world, &verticesLoad);

    (...)
}
```

Após ambas as variáveis serem inicializadas basta apenas chamar a função drawGroups que, dado um vetor com todos os grupos presentes no XML e um índice, desenha os mesmos usando os VBO's e para cada grupo são aplicadas as respetivas transformações presentes no vetor.

```
void drawGroups(vector<Group> groups, int *index) {
    for(const Group& g : groups){
        glColor3f(1.0, 1.0, 1.0);
        glPushMatrix();
        for (const auto& t : g.transformations) {
            if (t.name=="translate") {
                if (t.time!=0) animatedTranslate(t);
                else glTranslatef(t.x, t.y, t.z);
            }

            if (t.name=="scale") {
                glScalef(t.x, t.y, t.z);
            }

            if (t.name=="rotate") {
                if (t.time!=-1) animatedRotate(t);
                else glRotatef(t.angle,t.x, t.y, t.z);
            }

            if (t.name=="color") {
                glColor3f(t.x, t.y, t.z);
            }
        }
        for (pair<int,int> t : g.vboIndexes) {
            glBindBuffer(GL_ARRAY_BUFFER, vertices);
            glVertexPointer(3, GL_FLOAT, 0, 0);
            glDrawArrays(GL_TRIANGLES, t.first, t.second);
        }

        drawGroups(g.subGroups, index);

        glPopMatrix();
    }
}
```

3.2.4 Animações

Para criar animações com curvas de Catmull-Rom foi calculada a posição do objeto P(t) com base na matriz de Catmull-Rom, M, em 4 pontos pertencentes à curva P0, P1, P2 e P3 e na matriz T, tendo em conta um instante t.

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$

$$P = \begin{bmatrix} P_0x & P_1x & P_2x & P_3x \\ P_0y & P_1y & P_2y & P_3y \\ P_0z & P_1z & P_2z & P_3z \end{bmatrix}$$

Depois de conhecer estas matrizes basta aplicar as seguintes fórmulas para obter a posição do objeto.

$$A = P \cdot M$$

$$P(t) = A \cdot T$$

Estes cálculos são efetuados na função renderCatmullRomCurve.
Ainda relativamente à posição do objeto, foi necessário calcular a sua derivada de forma a poder obter a matriz de rotaçao que alinha o objeto com a curva. Para isto, conforme as fórmulas abaixo e com a matriz T', foi possível obter a derivada da posição, P'(t). Note-se que a matriz A diz respeito à calculada para a posição do objeto.

$$T' = \begin{bmatrix} 3t^2 & 2t & t & 0 \end{bmatrix}$$

$$P'(t) = A \cdot T'$$

Para obter a matriz de rotação, precisou-se de calcular primeiro o seguinte:

$$X = \|P'(t)\|$$

$$Z = \frac{X \cdot Y_{i-1}}{\|X \cdot Y_{i-1}\|}$$

$$Y_i = \frac{Z \cdot X}{\|Z \cdot X\|}$$

Note-se que ao efetuar estes cálculos considerou-se Y0 = (0, 1, 0). Após obter X, Y e Z bastou recorrer à função buildRotMatrix e, posteriormente a glMultMatrixf para multiplicar a matriz de rotação pela atual.

3.3 Resultados obtidos

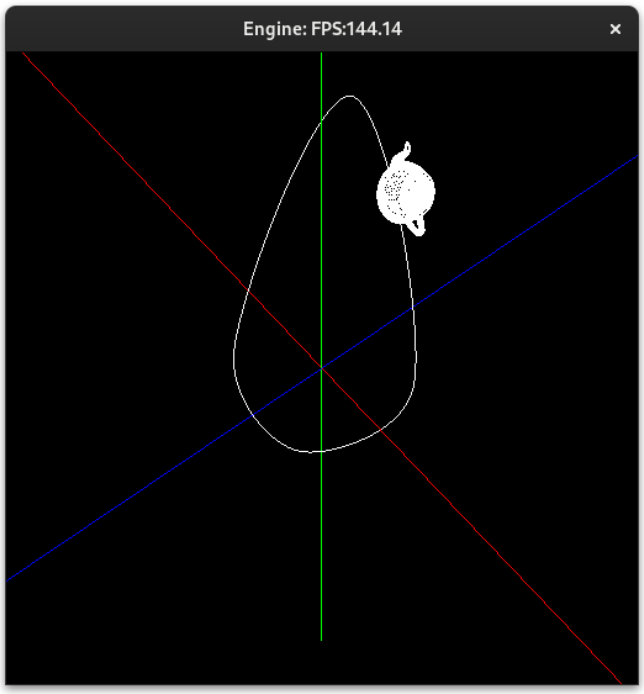


Figura 7: Ficheiro test_3_1.xml

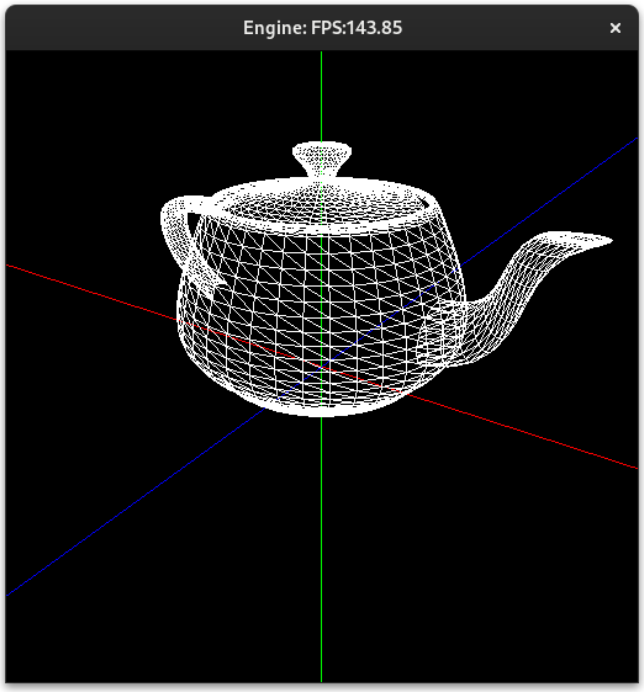


Figura 8: Ficheiro test_3_2.xml

4 Demo do Sistema Solar

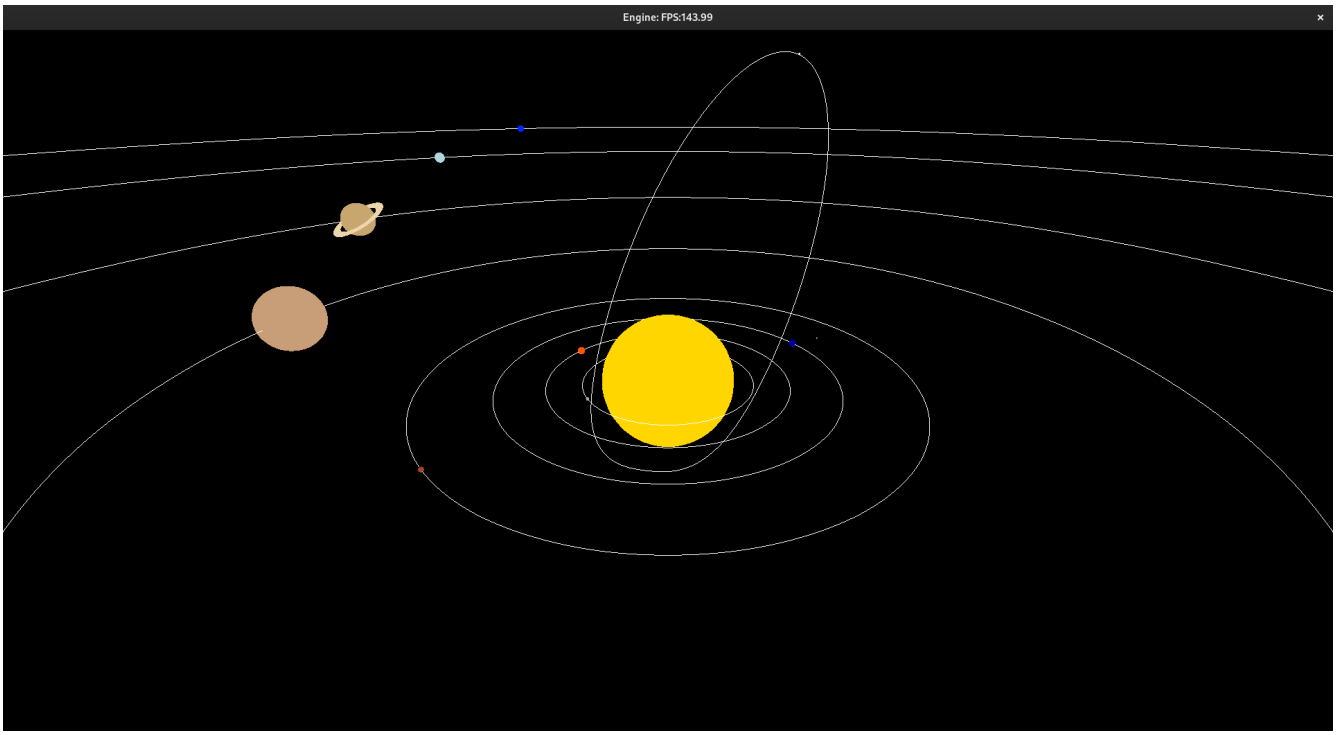


Figura 9: Demo

No modelo do sistema solar descrito no ficheiro XML, criamos representações visuais do Sol, dos 8 planetas (Mercúrio, Vénus, Terra, Marte, Júpiter, Saturno, Urano e Neptuno), a Lua e um cometa. Para tornar o modelo o mais realista possível, também incluimos os anéis de Saturno. O cometa está representado usando um modelo de teapot desenvolvido com a técnica que utiliza os patches de Bezier e terá uma trajetória aleatória.

Agora, como essas formas geométricas têm dimensões e posições fixas, precisamos de aplicar as transformações para alterar sua orientação, tamanho e localização em relação a um ponto de referência. Usamos escalas, rotações e translações para desenhar essas figuras de forma mais próxima da realidade. No entanto, devido às grandes diferenças de tamanho entre os diferentes elementos do sistema, não conseguimos usar uma escala completamente fiel à realidade. Fizemos pequenas aproximações para garantir uma visualização adequada de todas as figuras. Além disso, para os satélites, precisamos considerar a escala do planeta ao qual pertencem.

Começamos por definir as escalas para o Sol, para os planetas, para a Lua e para o cometa.

Nome	Escala
Sol	7
Mercúrio	0.4
Vénus	1
Terra	1.1
Lua	0.22
Marte	0.6
Júpiter	3
Saturno	2.5
Urano	1.8
Neptuno	1.65

Tabela 2: Tabela das escalas

A Lua, como é um satélite da terra, partilha a mesma escala que esta. Portanto, é necessário ajustar as coordenadas dos pontos da trajetória do satélite de acordo com a escala utilizada para representar o planeta.

Além dos pontos da curva orbital, as translações também têm uma componente temporal, expressa em segundos, que indica o tempo necessário para um planeta ou satélite completar sua órbita. Tanto o Sol quanto os planetas e a lua possuem períodos de rotação. Assim, foi necessário calcular o tempo de translação e rotação para cada planeta e satélite.

Para garantir a precisão dos tempos de rotação e translação de cada planeta ou satélite, foi desenvolvido um algoritmo. Inicialmente, organizamos os planetas e satélites numa tabela, ordenando-os de acordo com seus tempos de rotação e translação. Em seguida, calculamos a diferença de tempo entre cada planeta ou satélite em relação ao anterior na tabela. Observamos que o maior tempo correspondia a Netuno, ao qual atribuímos um tempo de 180 segundos (o máximo) para garantir a visibilidade de seu movimento.

Após isso, foi decidido que as diferenças multiplicativas (como $\times 2$, $\times 3$, etc.) seriam multiplicadas por 5.5, o que resultou em diferenças irrealistas. Essa discrepância foi então subtraída do tempo original do planeta ou lua, ajustando-o. A mesma diferença também foi subtraída do tempo do próximo planeta ou lua na tabela. Por exemplo, para Urano e Netuno, com uma diferença multiplicativa de 11 segundos ($2 \times 5.5 = 11$), o tempo ajustado para Urano seria de 169 segundos (180 - 11 = 169).

Em casos em que a diferença entre dois planetas ou satélites não é multiplicativa, mas sim uma soma de um valor específico, estabelecemos que, para a maior soma possível (+400), a diferença de tempo entre ambos seria de 1.6 segundos. Para outras situações, onde a soma está no intervalo [10, 400[, aplicamos uma regra de três simples para calcular a diferença. Por exemplo, se a soma for 30, a diferença resultante seria de 0.12 segundos ($30 \times 1.6 / 400$).

Para diferenças muito pequenas, como a diferença entre Terra e Marte, estabelecemos uma diferença de 0.01 segundos para somas entre]0,1[e de 0.02 segundos para somas entre [1,10[.

Nome	**Dados da rotação em Horas**	**Diferença para o anterior (R)**	**Dados da translação em Horas**	**Diferença para determinado astro (T)**
Júpiter	9.9	0	103944	*6 (Marte)
Saturno	10.23	+0,3	258216	*2,5 (Júpiter)
Neptuno	16	+6	1444560	*2 (Urano)
Urano	-17.9	+2	739176	*3 (Saturno)
Terra	23.9	+6	8767	*1,5 (Vénus)
Marte	24.6	+0,7	16488	*2 (Terra)
Sol	648	*2	—	—
Lua	672	+20	672	0
Mercúrio	1392	*2	2088	*1,5 (Lua)
Vénus	-5832	+400	5400	*2,5 (Mercúrio)

Tabela 3: Ordem crescente dos tempos de rotação e translação e relação com outros astros

Como Vénus e Urano rodam no sentido contrário ao movimento dos restantes planetas este tempo e representado como negativo. Quanto ao cometa a trajetória foi definida aleatoriamente.

Ficando os tempos finais definidos da seguinte forma:

Nome	**Rotação**	**Translação**
Sol	51.82	-
Mercúrio	62.9	71.15
Vénus	-86.5	84.9
Terra	6.99	94.75
Lua	51.9	51.9
Marte	7	105.75
Júpiter	6.92	138.75
Saturno	6.93	152.5
Urano	-6.97	169
Neptuno	6.95	180

Tabela 4: Tempos de rotação e translação em segundos

5 Conclusão

Fazendo uma retrospetiva da fase consideramos que foi um sucesso uma vez que cumprimos todos os requisitos estabelecidos.

A realização desta terceira fase mostrou tudo aquilo que foi o empenho do grupo no sentido do desenvolvimento do projeto, podendo observar os resultados do mesmo de uma forma mais prática. Para além disso o grupo expandiu os conhcimentos da matéria aprendida ao longo das aulas para esta terceira fase, apesar das dúvidas e desafios que foram surgindo relativamente, neste caso, à aplicação dos dois tipos de curvas.

Para além disto, alteramos a nossa estrutura de dados, o que possibilitou um carregamento mais eficiente de dados. Com isto e com a utilização de VBOs foi possível melhorar o desempenho da Engine. Relativamente à cosntrução do ficheiro XML para a apresentação do sistema solar, existiram algumas dificuldades na definição de rotações e translações entre os diferentes astros.

Tudo isto permite então estarmos confortáveis com os procedimentos e métodos necessários para realizar com sucesso a próxima fase, visto os conhecimentos adquiridos serem fundamentais para uma boa implementação do projeto na última fase.