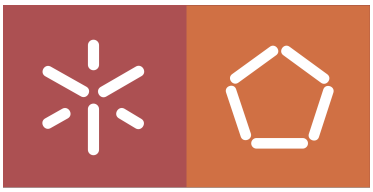

Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase 3 - Curvas, Superfícies Cubicas e VBO's

Relatório Trabalho Prático

Grupo 40

LEI - 3º Ano - 2º Semestre

A90969 Gonçalo Gonçalves
A98695 Lucas Oliveira
A89292 Mike Pinto
A96208 Rafael Gomes

Braga,
26 de abril de 2024

Conteúdo

1 Introdução **3**

2 Generator **4**
2.1 Classes Definidas 4
2.1.1 Classe *patch* 4
2.1.2 Classe MatrixOperations 5
2.2 Ficheiros *.patch* 7
2.2.1 Estrutura 7
2.2.2 Leitura e interpretação 7
2.3 Construção do Modelo 9

3 Engine **13**
3.1 Rotações Dinâmicas 13
3.2 Translações dirigidas por curvas de Catmull-Rom 13
3.3 Matriz de Rotação do Objeto 14
3.4 Desenho da trajetória 14
3.5 VBOs 15
3.6 Câmara 16
3.6.1 *First-Person-Camera* (FPC) 16
3.7 Comandos 20

4 Demo Scene **21**

5 Conclusão **24**

Lista de Figuras

2.1 *Teapot's* com 2 e 10 de "tecelagem"respetivamente. *Teapot's* gerados pela aplicação **Generator** e apresentados pela aplicação **Engine**. 12

4.1 Pontos calculados para a curva da orbita do planeta Terra. . . . 22
4.2 *Demo scene* dinâmica do sistema solar. 23

Capítulo 1

Introdução

Este relatório é elaborado no âmbito da terceira fase do trabalho prático da Unidade Curricular de Computação Gráfica do 2.º semestre do 3.º ano do curso de Engenharia Informática da Universidade do Minho.

Na presente fase é proposto a geração dos pontos de um modelo, através da aplicação ***Generator***, baseado em *Bezier patches*. Relativamente à aplicação ***Engine*** é proposto adicionar a possibilidade de criação de animações com base em curvas de ***Catmull-Rom*** alterando as transformações existentes de `Translação` e `Rotação`. Além disso, os pontos dos modelos a desenhar devem recorrer ao uso de `VBO's`, em oposição ao modo "imediato" utilizado nas fases anteriores.

Para a *demo scene* é solicitado uma versão dinâmica do sistema solar, que inclua um cometa, construido utilizando *patches* de *Bezier*, cuja trajetória seja definida com uma curva de ***Catmull-Rom***.

Este documento irá abordar os métodos utilizados, as funcionalidades implementadas e os resultados obtidos, oferecendo uma visão abrangente do trabalho realizado nesta fase do projeto.

Capítulo 2

Generator

Nesta etapa, é proposta a criação de um novo modelo baseado em ***Bezier patches***. Para criar esses modelos, é necessário fornecer à aplicação ***Generator*** o nome de um arquivo *.patch*, localizado na pasta `/patches` na raiz do projeto, com um valor de "tesselação" e o nome do arquivo *.3d* onde o resultado será armazenado.

Para implementar essa funcionalidade, foi necessário definir duas novas classes. Uma delas, no pacote `Utils`, é a classe `MatrixOperations`, que contém funções para operações em matrizes. Além disso, seguindo a abordagem das fases anteriores, foi definida uma nova classe, para representar a nova primitiva gráfica, no pacote da aplicação ***Generator***: a classe `Patches`.

2.1 Classes Definidas

2.1.1 Classe *patch*

Esta classe abriga todas as operações e métodos necessários para a geração de pontos baseados em *Bezier patches*. Está classe é definida conforme a listagem 2.1

```
class Patches : public Primitive {
private:
    int tesselation; // "tecelação" do modelo
    std::string patchFile; // Nome para o ficheiro .patch
    int numberPatches; // Número de patches
    int numberControlPoints; // Número de pontos de controlo
    std::vector<std::vector<int>> patchesIndex; // Indice pontos
    controlo
    std::vector<Point> controlPoints; // Pontos de controlo

public:
    Patches(int t, std::string pFile);
    void readPatchesFile();
    std::vector<Point> genPoints();
    Point calcPoint(float u, float v, std::vector<int>
    patchIndex);
};
```

Listagem 2.1: Definição da classe *patches*.

2.1.2 Classe MatrixOperations

Esta classe do pacote *Utils*, abrange todos os métodos necessários para manipular matrizes. Para sua representação, optou-se por usar um vetor de vetores de `floats`. Essa escolha foi feita devido à capacidade dos vetores de proporcionar uma estrutura flexível para lidar com matrizes de diferentes dimensões. Nesse contexto, cada vetor de *floats* (`vector<float>`) representa uma linha da matriz, onde cada índice corresponde ao valor numérico de uma coluna nessa linha. Enquanto isso, o vetor de vetores de *floats* (`vector<vector<float>>`) armazena todas as linhas da matriz.

Esta classe possui então uma função para multiplicação de matrizes, `Multiply-Matrices`, onde recebe duas matrizes, `A` e `B`, como se pode observar na listagem 3.14

```
std::vector<std::vector<float>>
MatrixOperations::MultiplyMatrices(std::vector<std::vector<float>>
matrixA, std::vector<std::vector<float>> &matrixB) {

    int linesA = matrixA.size(); // Linhas da Matriz A
    int collumsA = matrixA[0].size(); // Colunas da Matriz A

    int linesB = matrixB.size(); // Linhas da Matriz B
    int collumsB = matrixB[0].size(); // Colunas da Matriz B

    if (collumsA == linesB) // Verificar se as matrizes são
    compativeis para a multiplicação
    {
        std::vector<std::vector<float>> result(linesA,
        std::vector<float>(collumsB,0)); // Criar um vertor de
        vetores para a Matriz resultante da multiplicação

        for (int i = 0; i < linesA; ++i) {
            for (int j = 0; j < collumsB; ++j) {
                for (int k = 0; k < collumsA; ++k) {
                    result[i][j] += matrixA[i][k] *
                    matrixB[k][j];
                }
            }
        }
        return result;
    }
    else // Caso as Matrizes não sejam compativeis
    {
        std::cout << "Error: The dimensions of the matrices are
        not compatible for multiplication. CollumsA = " <<
        collumsA << "; linesB = " << linesB << std::endl;
        return std::vector<std::vector<float>>();
    }
}

.
```

Listagem 2.2: Função `MultiplyMatrices`

2.2 Ficheiros *.patch*

2.2.1 Estrutura

Os ficheiros *.patch* utilizados pela aplicação ***Generator*** seguem a seguinte estrutura:

1. A primeira linha do ficheiro, indica o **número de *patches* do modelo**.
2. Em seguida, são listados os índices dos pontos de controlo de cada *patch*. Cada linha representa um *patch*, e os índices são separados por vírgulas.
3. Após os índices dos *patches*, é listado o **número de pontos de controlo**.
4. As linhas restantes representam os **pontos de controlo**, contendo as suas coordenadas `x`, `y` e `z` separadas por vírgulas.

2.2.2 Leitura e interpretação

Antes de se iniciar o cálculo dos pontos de superfície, é necessário ler e interpretar os ficheiros *.patch*. Para esta finalidade foi definida a função `void Patches::readPatchesFile()` responsável por ler estes ficheiros. Esta função abre, inicialmente, o ficheiro especificado. Caso ocorra algum erro durante a abertura do ficheiro, é escrito para o *stdout* uma mensagem de erro e o programa é encerrado.

```
void Patches::readPatchesFile()
{
std::ifstream file("../Patches/" + patchFile);

// Caso ocorra algum erro ao ler o ficheiro .patch
if(!file.is_open())
{
    std::cout << "Error opening patches file: \"" <<
    patchFile << "\"" << std::endl;
    exit(1);
}
```

Listagem 2.3: Abertura de um ficheiro *.patch*.

Em seguida, a função lê o número de *patches* a serem utilizados no modelo, armazenando esse valor na variável de instância da classe, `numberPatches`.

```
std::string line;

// Ler o numero de patches
std::getline(file, line);
numberPatches = std::stoi(line);
```

Listagem 2.4: Leitura do número de *patches* de um modelo.

Para cada *patch*, a função lê uma linha do ficheiro, dividindo os valores usando o caráter delimitador ",", armazenando esses valores num vetor de inteiros. Cada vetor resultante é então adicionado ao vetor de instância `patchesIndex`, que contém os índices dos *patches*.

```
// Ler as linhas com os index dos patches
for(int i = 0; i < numberPatches; ++i)
{
    std::getline(file, line);
    std::vector<int> patchLine;

    // Separar a string nos valores separados por ","
    char* strNumbers = std::strtok(&line[0], ",");
    while(strNumbers != nullptr)
    {
        int number = std::stoi(strNumbers);
        patchLine.push_back(number);
        strNumbers = std::strtok(nullptr, ",");
    }
    patchesIndex.push_back(patchLine);
}
```

Listagem 2.5: Leitura dos índices dos pontos de cada *patch*.

Após ler todos os índices de todos os *patches* é lido o número de pontos de controlo, guardando esse valor na variável de instância da classe, `numberControlPoints`.

```
// Lê o número de pontos de controlo
std::getline(file, line);
numberControlPoints = std::stoi(line);
```

Listagem 2.6: Leitura do número de pontos de controlo.

Por cada ponto de controlo é então lida uma linha do ficheiro, dividindo os valores de cada linha pelo caráter "," e instanciando um objeto da classe `Point` com as coordenadas lidas. Por fim, cada ponto lido é adicionado ao vetor de instância da classe, `controlPoints`.

```
// Ler os pontos de Controlo
for(int i = 0; i < numberControlPoints; ++i)
{
    std::getline(file, line);

    float x = std::stof(std::strtok(&line[0], ","));
    float y = std::stof(std::strtok(nullptr, ","));
    float z = std::stof(std::strtok(nullptr, ","));
    Point point = Point(x, y, z);
    controlPoints.push_back(point);
}
```

Listagem 2.7: Leitura dos pontos de controlo.

2.3 Construção do Modelo

Após obter os *patches* e pontos de controlo a utilizar no modelo, para o cálculo dos pontos de superfície de *Bézier* basta aplicar a fórmula 2.1 para obter as coordenadas do Ponto a representar

$$p(u,v) = U \times M \times P \times M^T \times V^T \tag{2.1}$$

onde U e V representam matrizes das coordenadas u e v respetivamente. Estas matrizes são calculadas conforma as equações 2.2 e 2.3.

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \tag{2.2}$$

$$V = \begin{bmatrix} v^3 & v^2 & v & 1 \end{bmatrix} \tag{2.3}$$

A Matriz M representa a matriz 2.4

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{2.4}$$

Já a matriz P contém os pontos de controlo do *patch* a calcular.
Por fim é calculado as matrizes transpostas das matrizes M e V e aplicada a formula descrita em cima 2.1.

Foi então implementada a função `Point Patches::calcPoint(float u, float v, std::vector<int> patchIndex)` para calcular as coordenadas cartesianas de um ponto.

```
std::vector<std::vector<float>> bezierMatrix = { //
Dimensões: 4x4
        {-1, 3, -3, 1},
        {3, -6, 3, 0},
        {-3, 3, 0, 0},
        {1, 0, 0, 0}
};
```

Listagem 2.8: Representação da matriz M.

```
// Matrizes u e v para a interpolação
std::vector<std::vector<float>> uMatrix = {{u*u*u, u*u, u,
1}}; // Dimensões: 1x4
std::vector<std::vector<float>> vMatrix = {{v*v*v}, {v*v},
{v}, {1}}; // Dimensões 4x1
```

Listagem 2.9: Representação da matriz U e V.

```
// Matrizes com as coordenadas x, y e z dos pontos de
controlo do patch
std::vector<std::vector<float>> xMatrix = { // Dimensões: 4x4
        {controlPoints[patchIndex[0]].getX(),
        controlPoints[patchIndex[1]].getX(),
        controlPoints[patchIndex[2]].getX(),
        controlPoints[patchIndex[3]].getX()},
        {controlPoints[patchIndex[4]].getX(),
        controlPoints[patchIndex[5]].getX(),
        controlPoints[patchIndex[6]].getX(),
        controlPoints[patchIndex[7]].getX()},
        {controlPoints[patchIndex[8]].getX(),
        controlPoints[patchIndex[9]].getX(),
        controlPoints[patchIndex[10]].getX(),
        controlPoints[patchIndex[11]].getX()},
        {controlPoints[patchIndex[12]].getX(),
        controlPoints[patchIndex[13]].getX(),
        controlPoints[patchIndex[14]].getX(),
        controlPoints[patchIndex[15]].getX()}
};
```

Listagem 2.10: Representação da matriz das coordenadas x dos pontos de controlo. O cálculo das matrizes das coordendas y e z é semelhante.

```
// Calculo de matrizes intermedias: uMatrix x bezierMatrix ;
(bezierMatrix)Transposta x vMatrix
std::vector<std::vector<float>> uBezierMatrix =
MatrixOperations::MultiplyMatrices(uMatrix, bezierMatrix);
std::vector<std::vector<float>> bezierVMatrix =
MatrixOperations::MultiplyMatrices(bezierMatrix, vMatrix);

// Calculo das coordenadas x,y e z do ponto resultante da
interpolação
std::vector<std::vector<float>> uBezierPointXMatrix =
MatrixOperations::MultiplyMatrices(uBezierMatrix,xMatrix);
float x =
MatrixOperations::MultiplyMatrices(uBezierPointXMatrix,
bezierVMatrix)[0][0];

std::vector<std::vector<float>> uBezierPointYMatrix =
MatrixOperations::MultiplyMatrices(uBezierMatrix,yMatrix);
float y =
MatrixOperations::MultiplyMatrices(uBezierPointYMatrix,
bezierVMatrix)[0][0];

std::vector<std::vector<float>> uBezierPointZMatrix =
MatrixOperations::MultiplyMatrices(uBezierMatrix,zMatrix);
float z =
MatrixOperations::MultiplyMatrices(uBezierPointZMatrix,
bezierVMatrix)[0][0];

return {x,y,z};
```

Listagem 2.11: Calculo das coordenadas de um ponto de superfície.

Por fim esta função é utilizada pela função `genPoints()`, cuja finalidade passa por ler o ficheiro *.patch* e calcular as coordenadas de todos os pontos, guardando o seu valor num vetor de pontos, respeitando sempre a regra da mão direita.

Esta função inicia os valores de $u = v = 0.0f$ e $step = \frac{1}{tecellation}$ e calcula posteriormente os pontos de cada *patch*, recorrendo à função descrita anteriormente, conforme o valor de *tecellation* fornecido.
O valor *tecellation*, altera a "definição"do resultado, como se pode observar pela figura 2.1.

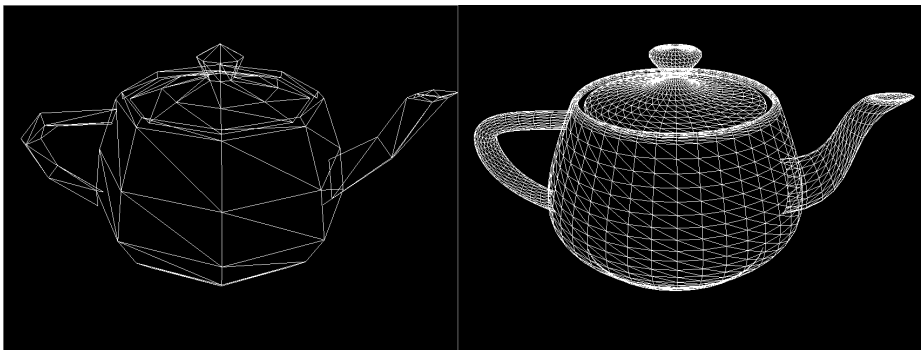

Figura 2.1: *Teapot's* com 2 e 10 de "tecelagem"respetivamente. *Teapot's* gerados pela aplicação **Generator** e apresentados pela aplicação **Engine**.

Capítulo 3

Engine

3.1 Rotações Dinâmicas

Para podermos utilizar rotações dinâmicas, isto é, dependentes do tempo e, portanto, animadas, definiu-se uma nova *classe-filho*, `Transformation`, `RotationDynamic`, que será identificada quando no *XML* possui o campo *time*.
Assim, o *XML* poderá indicar o tempo que um objeto demorará a realizar uma rotação completa de 360 graus, deduzindo o valor do ângulo de rotação em cada instante a partir da fórmula 3.1.

$$\alpha\,(t) = \frac{(t - t_0) * 360}{t_r} \tag{3.1}$$

onde $(t - t_0)$ corresponde ao intervalo de tempo decorrido e t_r ao tempo total definido no ficheiro *XML*.
Considerando que o ponto de partida é 0, então o tempo decorrido é obtido a partir da função `glutGet(GLUT_ELAPSED_TIME)`.

3.2 Translações dirigidas por curvas de Catmull-Rom

Foi necessária criar uma classe, *TranslationDynamic* para distinguir entre os dois tipos de translação. Alterou-se a função de *parser* do *XML* para distinguir os dois tipos de translações, no caso de encontrar o campo *time* então irá fazer também *parse* do campo *align* e de todos os pontos que são filhos do elemento XML *translate*.
Estes pontos correspondem aos pontos de controlo da curva de *Catmull-Rom* que define a trajetória do objeto, o *time* ao tempo que o objeto demora a percorrer toda a curva e o campo *align*, um *boolean*, que define se o objeto irá ficar alinhado com a curva.
Para criar as animações com as curvas de *Catmull-Rom* foi calculada a posição do objeto $P(t)$ com base na matriz de *Catmull-Rom*, M, em quatro pontos pertencentes à curva, tendo em conta um instante t.

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \tag{3.2}$$

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.3}$$

$$P = \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{3.4}$$

Após definir estas matrizes basta aplicar as seguintes fórmulas para obter a posição do objeto:

$$A = M * P \tag{3.5}$$

$$P(t) = T * A \tag{3.6}$$

3.3 Matriz de Rotação do Objeto

Para alinhar o objeto com a curva é necessário calcular a derivada. Para isto, basta derivar T:

$$T' = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \tag{3.7}$$

Considerando Y_0 = (0,1,0), para obter a matriz de rotação é necessário calcular:

$$X = ||P'(t)|| \tag{3.8}$$

$$Y_i = \frac{Z * X}{||Z * X||} \tag{3.9}$$

$$Z = \frac{X * Y_{i-1}}{||X * Y_{i-1}||} \tag{3.10}$$

Uma vez calculado o `X`, `Y` e o `Z` calcula-se a nova matriz de rotação multiplicada pela anterior a partir da função `glMultMatrixf`.

3.4 Desenho da trajetória

Para desenhar a trajetória é necessário calcular as várias posições dos pontos que pertencem à curva, tendo por base os pontos de controlo fornecidos, recorrendo aos métodos explicados no cálculo de pontos para as translações.
Assim, calculamos os pontos num intervalo $t \epsilon [0, 1[$ com um intervalo de 1/tecelagem, em que a tecelagem é calculada a partir da expressão $2 * duration/1000$.
Uma vez calculados estes vértices, eles serão ligados por uma linha, recorrendo à primitiva `GL_LINE_LOOP`.

3.5 VBOs

Com o intuito de armazenar os pontos do modelo de forma mais eficiente, foi recorrido ao uso de `VBOs`. Sendo assim, a sua implementação iniciou-se na classe *Model*, em que através da função `glGenBuffers(1, vbo)` é gerado um identificador único para o `VBO` e armazenado numa variável designada por `"vbo"`. Esta variável é então associada como um *buffer* através da função `glBindBuffer(GL_ARRAY_BUFFER, *vbo)`. Por fim, `glBufferData(GL_ARRAY_BUFFER, points.size() * sizeof(Point), &points[0], GL_STATIC_DRAW)` é usado para carregar os pontos do modelo no *buffer* de `VBOs`. Os pontos armazenados no vetor *points* são copiados para o *buffer*, que é assim usado para a renderização dos triângulos.

```
Model::Model(std::vector<Point> pts, float r1, float g1, float
↪  b1)
{
    points = std::move(pts);
    r = r1;
    g = g1;
    b = b1;

    glEnableClientState(GL_VERTEX_ARRAY);
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glBufferData(GL_ARRAY_BUFFER, points.size() * sizeof(Point),
    ↪  &points[0], GL_STATIC_DRAW);
}
```

Listagem 3.12: Função responsável pela criação dos *VBOs*.

Para desenhar os modelos, é necessário vincular o `VBO` que contém os dados dos pontos, utilizando a função `glBindBuffer(GL_ARRAY_BUFFER, *vbo)`. De seguida, através da função `glVertexPointer(3, GL_FLOAT, sizeof(Point), 0)`, é especificado como estão organizados os dados de um ponto e qual o seu formato. Por fim, `glDrawArrays(GL_TRIANGLES, 0, numTriangles * 3)`, é usada para desenhar os triângulos, indicando o índice do primeiro vértice a ser desenhado e o número total de vértices a serem desenhados.

```
void Model::draw(){
    int numTriangles = points.size() / 3;
    glColor3f(r,g,b);

    // Bind the VBO
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glVertexPointer(3, GL_FLOAT, sizeof(Point), 0);

    // Draw using VBO
    glDrawArrays(GL_TRIANGLES, 0, numTriangles * 3);
}
```

Listagem 3.13: Função responsável por desenhar os *VBOs*.

Para libertar a memória alocada para os *VBOs*, foi definida uma função *Model::clear()*. Esta função é utilizada após o termino do programa.

```
void Model::clear(){
    glDeleteBuffers(1, vbo);
}
```

Listagem 3.14: Função responsável por libertar a memoria alocada para os *VBOs*.

3.6 Câmara

Ainda nesta fase, foram implementadas alterações na movimentação da câmara. Nas fases anteriores, a câmara utilizada só podia mover-se em torno de um ponto fixo (normalmente, o ponto de coordenadas (0,0,0)). Como resultado, foi introduzido um novo modo para a câmara, *First-Person-Camera* (FPC). Além disso, foi adicionada a capacidade de "rodar"a câmara com o input do rato.

3.6.1 *First-Person-Camera* (FPC)

Diferentemente do modo de movimentação da câmara das fases anteriores, onde a câmara estava fixamente a olhar para um ponto fixo, rodando apenas à sua volta, o modo FPC, permite ao utilizador mover a câmara para a frente, para trás, para a esquerda, para a direita e para cima e para baixo.

Como resultado, surgiu a necessidade de alterar dinamicamente para qual ponto a câmara está direcionada (*look-up value*). Para isso, são utilizadas as seguintes fórmulas:

$$lookup_x = position_x + \cos\beta \times \sin\alpha \tag{3.11}$$

$$lookup_y = position_y + \sin\beta \tag{3.12}$$

$$lookup_z = position_z + \cos\beta \times \cos\alpha \tag{3.13}$$

onde $position_x$, $position_y$ e $position_z$ representam as coordenadas `x`, `y` e `z` onde se localiza a câmara.

Para alternar entre os dois modos de movimentação da câmara o utilizador necessita de pressionar a tecla **c**. O utilizador pode também pressionar a tecla **r** para movimentar a câmara para os valores iniciais.

Movimentação para a frente e trás

Quando movimentamos a câmara para a frente, essencialmente estamos a deslocar a câmara para o ponto no qual está a olhar. Da mesma forma, ao movimentar a câmara para trás, revertemos este deslocamento.
Para calcular estas alterações é necessário definir uma `matriz direção`, que irá representar a direção para o qual a camera está a olhar. Esta matriz é calculada subtraindo/adicionando as coordenadas da posição da câmara com as coordenadas para o qual a câmera está apontada, tal como demonstrado em 3.14

$$D = \begin{bmatrix} look_x \pm pos_x & look_y \pm pos_y & look_z \pm pos_z \end{bmatrix} \tag{3.14}$$

Após a normalização da matriz 3.14, é possível então calcular qual a posição da câmara após o movimento e para qual o ponto que estará apontada.
Para o cálculo da posição basta multiplicar a matriz por uma constante `k`, positiva caso o movimento seja em frente e negativa caso o movimento seja para trás, e somar o valor das componentes das coordenadas da matriz D às coordenadas da câmara, equações 3.15, 3.16 e 3.17.

$$newPos_x = pos_x + k \times D_{normalizado}[x] \tag{3.15}$$

$$newPos_y = pos_y + k \times D_{normalizado}[y] \tag{3.16}$$

$$newPos_z = pos_z + k \times D_{normalizado}[z] \tag{3.17}$$

Já para o cálculo das coordenadas para qual a câmara está a apontar basta somar à posição da câmara o valor das componentes das coordenadas da matriz D, equações 3.18, 3.19 e 3.20.

$$look_x = newPos_x + D_{normalizado}[x] \tag{3.18}$$

$$look_y = newPos_y + D_{normalizado}[y] \tag{3.19}$$

$$look_z = newPos_z + D_{normalizado}[z] \tag{3.20}$$

As seguintes funções descrevem o cálculo das equações anteriores:

```
void moveCameraForward()
{
    vector<float> direction =
    ↪  MatrixOperations::normalizeMatrix({cam.lookX - cam.posX,
    ↪  cam.lookY - cam.posY, cam.lookZ - cam.posZ});
    float k = 1.0f;
    cam.posX += k * direction[0];
    cam.posY += k * direction[1];
    cam.posZ += k * direction[2];

    cam.lookX = cam.posX + direction[0];
    cam.lookY = cam.posY + direction[1];
    cam.lookZ = cam.posZ + direction[2];
}

void moveCameraBackward()
{
    vector<float> direction =
    ↪  MatrixOperations::normalizeMatrix({cam.lookX - cam.posX,
    ↪  cam.lookY - cam.posY, cam.lookZ - cam.posZ});
    float k = -1.0f;
    cam.posX += k * direction[0];
    cam.posY += k * direction[1];
    cam.posZ += k * direction[2];

    cam.lookX = cam.posX + direction[0];
    cam.lookY = cam.posY + direction[1];
    cam.lookZ = cam.posZ + direction[2];
}
```

Listagem 3.15: Funções para mover a câmara para a frente e para trás

Movimentação para a esquerda e direita

No deslocamento da câmara para a esquerda e para a direita, essencialmente deslocamos a posição da câmara numa direção perpendicular ao da matriz direção. A matriz direção é calculada da mesma forma à matriz 3.14. É feito o produto da matriz direção com o da matriz *up* obtendo uma matriz perpendicular a estas. Esta matriz resultante representa a direção para a qual queremos mover a câmara lateralmente, equação 3.21

$$D_{lateral} = normalizar(cross(D, up)) \tag{3.21}$$

Este vetor é então normalizado.
Para o cálculo então da posição da câmara é feito um processo semelhante com as equações da secção anterior, equações 3.15, 3.16 e 3.17.
Estas equações são descritas pelas seguintes funções:

```
void moveCameraRight()
{
    vector<float> res = MatrixOperati-
    ↪  ons::normalizeMatrix(MatrixOperations::crossMatrix({cam.lookX
    ↪  - cam.posX, cam.lookY - cam.posY, cam.lookZ -
    ↪  cam.posZ},{cam.upX, cam.upY, cam.upZ}));
    vector<float> direction =
    ↪  MatrixOperations::normalizeMatrix({cam.lookX - cam.posX,
    ↪  cam.lookY - cam.posY, cam.lookZ - cam.posZ});
    float k = 1.0f;

    cam.posX += k * res[0];
    cam.posY += k * res[1];
    cam.posZ += k * res[2];

    cam.lookX = cam.posX + direction[0];
    cam.lookY = cam.posY + direction[1];
    cam.lookZ = cam.posZ + direction[2];
}


void moveCameraLeft()
{
    vector<float> res = MatrixOperati-
    ↪  ons::normalizeMatrix(MatrixOperations::crossMatrix({cam.lookX
    ↪  - cam.posX, cam.lookY - cam.posY, cam.lookZ -
    ↪  cam.posZ},{cam.upX, cam.upY, cam.upZ}));
    vector<float> direction =
    ↪  MatrixOperations::normalizeMatrix({cam.lookX - cam.posX,
    ↪  cam.lookY - cam.posY, cam.lookZ - cam.posZ});
    float k = -1.0f;

    cam.posX += k * res[0];
    cam.posY += k * res[1];
    cam.posZ += k * res[2];

    cam.lookX = cam.posX + direction[0];
    cam.lookY = cam.posY + direction[1];
    cam.lookZ = cam.posZ + direction[2];
}
```

Listagem 3.16: Funções para mover a câmara para a esquerda e para a direita

3.7 Comandos

Foi implmentado um menu para apresentar todos os comandos aceites pela aplicação. Este menu contempla a descrição dos comandos da câmara, alteração entre modos de *rendering* e *face culling* assim como alterações de visualização e *debug*.

```
----------------[Commands]----------------

##### Camera and Movement Commands #####
'c' -> Toggle between 'First-Person-Camera'(FPC) and
↪  'Spherical-Camera'(SC)
'awsd' -> Move the camera around.
'+' or 'q' -> In FPC increments the height. In SC moves the
↪  camera closer.
'-' or 'e' -> In FPC decrements the height. In SC moves the
↪  camera further.
'r' -> Resets the camera position.
'Mouse Left Press + Drag' -> Move the camera around

##### Rendering Modes #####
'f' -> Sets the render mode to FILL.
'l' -> Sets the render mode to LINES.
'p' -> Sets the render mode to POINTS.

##### Face culling #####
'b' -> Sets the face culling to BACK.
'n' -> Sets the face culling to FRONT.
'm' -> Sets the face culling to FRONT and BACK.

##### Toggles #####
'f1' -> Toggle the axis drawing.
'f2' -> Toggle animation.
'f3' -> Toggle drawing of Catmull-Rom curves.
'f11' -> Show camera information (DEBUG).
'f12' -> Show commands information.
```

Listagem 3.17: Menu com os comandos suportados pela aplicação

Capítulo 4

Demo Scene

Nesta fase, utilizou-se a *demo scene* do sistema solar desenvolvido na fase anterior. Nesta fase adicionou-se um cometa e definiram-se as trajetórias para os planetas, luas e cometa. Para o desenho das trajetórias utilizou-se curvas de *Catmull-rom* para modelar essas trajetórias conforme desejado.

Para o cálculo dos pontos de controlo utilizado nas curvas de *Catmull-Rom*, estes foram definidos com base na distância do corpo celeste relativamente ao sol (ou planeta no caso das luas). Desta forma foi possível definir orbitas circulares ao redor do sol, contidas no plano `XoZ`.

Como se pode observar na 4.1, foram definidos 24 pontos para os planetas ou, no caso das Luas, 8 pontos. Para o cálculo coordenadas **x** e **z** foi utilizado as seguintes equações:

$$X = raio * cos(angulo) \tag{4.1}$$

$$Z = raio * sen(angulo) \tag{4.2}$$

Onde

raio : Distância em relação do corpo celeste ao sol/planeta
angulo : Ângulo relativamente ao eixo do x.

```
<group> <!-- Terra -->
    <transform>
        <translate time="40" align="true">
            <point x="40.5" y="0" z="0" />
            <point x="39.12" y="0" z="-10.48" />
            <point x="35.07" y="0" z="-20.25" />
            <point x="28.64" y="0" z="-28.64" />
            <point x="20.25" y="0" z="-35.07" />
            <point x="10.48" y="0" z="-39.12" />
            <point x="-0.0" y="0" z="-40.5" />
            <point x="-10.48" y="0" z="-39.12" />
            <point x="-20.25" y="0" z="-35.07" />
            <point x="-28.64" y="0" z="-28.64" />
            <point x="-35.07" y="0" z="-20.25" />
            <point x="-39.12" y="0" z="-10.48" />
            <point x="-40.5" y="0" z="0.0" />
            <point x="-39.12" y="0" z="10.48" />
            <point x="-35.07" y="0" z="20.25" />
            <point x="-28.64" y="0" z="28.64" />
            <point x="-20.25" y="0" z="35.07" />
            <point x="-10.48" y="0" z="39.12" />
            <point x="0.0" y="0" z="40.5" />
            <point x="10.48" y="0" z="39.12" />
            <point x="20.25" y="0" z="35.07" />
            <point x="28.64" y="0" z="28.64" />
            <point x="35.07" y="0" z="20.25" />
            <point x="39.12" y="0" z="10.48" />
        </translate>
        <!-- <translate x="40.5" y="0" z="0" /> -->
        <scale x="0.72" y="0.72" z="0.72"/>
        <rotate time="43" x="1" y="0" z="1" angle="23.45"/>
    </transform>
    <models>
        <model file="sphere_demo.3d" r="0" g="0.29" b="0.54" />
    </models>
```

Figura 4.1: Pontos calculados para a curva da orbita do planeta Terra.

Para o cálculo da órbita do cometa relativamente ao sol é realizado um processo semelhante, porém, em vez de a órbita ser representada por uma circunferência, é representada por uma elipse, utilizando o mesmo número de pontos que as as luas, sendo sua órbita contida no plano `XoY`, obtendo assim ao resultado da figura 4.2.

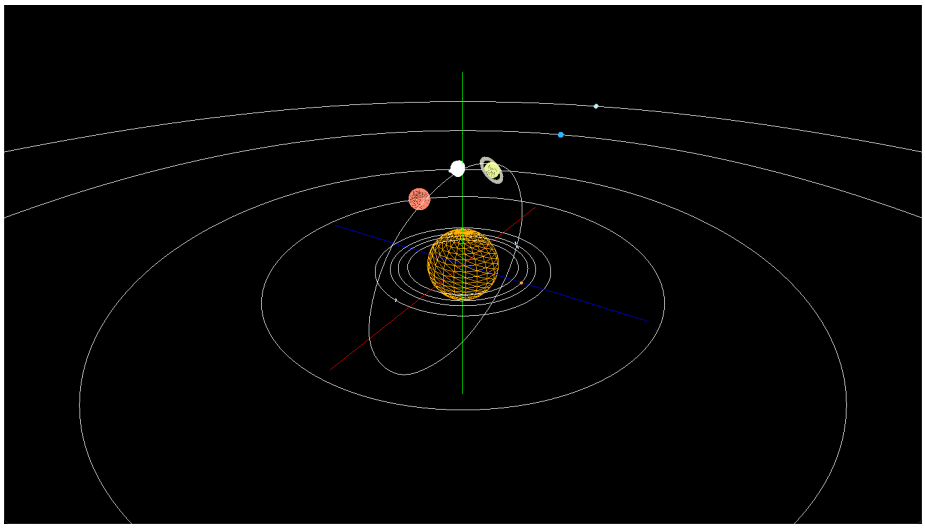

Figura 4.2: *Demo scene* dinâmica do sistema solar.

Capítulo 5

Conclusão

Nesta fase do trabalho-prático, foram solicitadas alterações nas aplicações ***Generator*** e ***Engine***. As alterações da aplicação ***Generator*** passaram por gerar os pontos de uma nova primitiva gráfica baseada em *Bezier Patches*, lidos via um ficheiro fornecido como argumento.

Já na aplicação ***Engine*** foi solicitado a alteração das transformações de **rotação** e **translação**, dando a possibilidade de definir movimento definido por curvas de *Catmull-Rom*. Para tal foi alterada a forma de como os ficheiros *xml* eram lidos pela aplicação, onde no caso, estas transformações podiam possuir opcionalmente uma *tag* de `time`. Além dos movimentos, foram implementados o desenho dos pontos recorrendo a **VBO's**.
Foi ainda adicionado à aplicação ***Engine*** um novo tipo de movimento da câmara e um menu com os comandos aceites pela aplicação.
Como *demo scene* foi ainda desenvolvido um sistema solar dinâmico, onde os corpos celestes possuíssem movimento e a adição de um cometa construido através de *Bezier Patches* e cuja órbita fosse definida através de uma curva de *Catmull-Rom*.

Em suma, acreditamos ter cumprido os objetivos propostos, expandindo tanto as funcionalidades da aplicação ***Generator*** como da aplicação ***Engine***. Além disso, ainda foram adicionados objetivos extra, como, por exemplo, a câmara em primeira pessoa. No entanto, estamos cientes que o nosso trabalho necessita de alguns ajustes, como, por exemplo, melhorar a gestão dos `vbo's` no desenho das curvas de *Catmull-Rom* e alterar o número de pontos utilizados para o desenho das curvas das órbitas dos corpos celestes.