Universidade do Minho

Licenciatura em Engenharia Informática

GitHub do Projeto

Computação Gráfica - Fase 2

Grupo 43

Eduardo Pereira (A94881) Gonçalo Freitas (A96136)
Gonçalo Vale (A96923) José Pereira (A89596)

Ano Letivo 2023/2024




Conteúdo

1 Introdução

Este relatório documentou a fase 3 do trabalho prático da disciplina de Computação Gráfica. Nesta fase tínhamos diversos objetivos, tais como: desenhar os modelos com **VBOs**; alterar o *generator* para ser capaz de criar ficheiros através de *Bezier Patches* e de um ficheiro de origem; expandir os elementos *translate* e *rotate* no *engine* de forma a que estes ocorram continuamente.

É importante mencionar que o foco desta fase concentrou-se no *Engine*, apesar do *Generator* ter sido também alterado.

2 *Generator*

Nesta etapa, foi adicionado ao *generator* as funções necessárias para a criação de um objeto 3D, através do uso de *Bezier Patches* e de um ficheiro de origem com os pontos de controlo e nível de *tesselation* pertinentes ao sólido.

As malhas de *Bezier Patches* são superiores às malhas triangulares como representação de superfícies lisas. Eles exigem menos pontos (e, portanto, menos memória) para representar superfícies curvas, são mais fáceis de manipular e possuem propriedades de continuidade muito melhores.

Para o cálculo dos *Bezier Patches* vimo-nos obrigados a criar funções auxiliares: **multMatVec** que calcula o resultado da multiplicação entre uma matriz e um vetor, **multMatrix** que calcula o resultado da multiplicação entre duas matrizes, **bez** que calcula a superfície Bezier dados 2 parâmetros e uma matriz com os pontos de controlo .

Para fazer parsing do ficheiro patch de origem:

```
getline(b_file, line);
nPatches = stoi(line);

while(nPatches>0){
    getline(b_file, line);
    nPoints = stoi(line);
    std::vector<int> patch;
    char* token = strtok((char*)(line.c_str()), ", ");
    while(token != NULL){
        patch.push_back(atof(token));
        token = strtok(NULL, ", ");
    }
    patches.push_back(patch);
    nPatches--;
}
```

Através de **getline** obtemos o numero de *patches* que será a primeiro linha do ficheiro e, a partir deste, obtemos todos os *patches*, delimitados por virgulas, que serão usados para os cálculos que serão escutados mais à frente.

```
getline(b_file, line);
nPoints = stoi(line);
```

```
while(nPoints>0){
    getline(b_file, line);
    std::vector<float> controlPoint;
    char* token = strtok((char*)line.c_str(), ", ");
    while(token != nullptr){
        controlPoint.push_back(atof(token));
        token = strtok(nullptr, ", ");
    }
    tuple<float, float, float> t(controlPoint[0], controlPoint[1], controlPoint[2
    controlPoints.push_back(t);
    nPoints--;
}
```

A seguir aos *patches* é obtido o numero de pontos de controlo, assim como todos eles, os quais serão guardados em tuplos que representam o x, y, z.

Agora com os *patches* e pontos de controlo podemos utilizar o nível de *tesselation* e funções auxiliares para calcular os pontos da figura para o ficheiro em formato 3D. Para cada patch constrói 3 matrizes(a,b,c) para as coordenadas x,y,z dos pontos de controlo; depois disto, utilizando o nível de de tesselation e a função auxiliar **bez** computa as coordenadas x,y,z dos pontos na superfície de Bezier.

```
float a[4][4], b[4][4], c[4][4];

for(std::vector<int> indexes : patches){

    for(int i = 0; i < 4; i++){
        for(int j = 0; j < 4; j++){
            a[i][j] = get<0>(controlPoints[indexes[i*4+j]]);
            b[i][j] = get<1>(controlPoints[indexes[i*4+j]]);
            c[i][j] = get<2>(controlPoints[indexes[i*4+j]]);
        }
    }

    float temp[4][4];

    multMatrix(main_matrix,a,temp);
    multMatrix(temp,main_matrix,a);
    multMatrix(main_matrix, b, temp);
    multMatrix(temp, main_matrix, b);
    multMatrix(main_matrix, c, temp);
    multMatrix(temp, main_matrix, c);

    float tess = 1/tesselation;
    for(float i = 0.0;i<1.0;i += tess){
        for(float j = 0.0;j<1.0;j += tess){
            float x = bez(i,j,a);
            float y = bez(i,j,b);
            float z = bez(i,j,c);

            bezier_points << x << " " << y << " " << z << "\n";
```

```
            x = bez(i+tess,j,a);
            y = bez(i+tess,j,b);
            z = bez(i+tess,j,c);

            bezier_points << x << " " << y << " " << z << "\n";

            x = bez(i,j+tess,a);
            y = bez(i,j+tess,b);
            z = bez(i,j+tess,c);

           bezier_points << x << " " << y << " " << z << "\n";
        }
    }
}
```

Para criar uma ficgura com *Bezier Patches* no *generator.py*, deverá realizar o comando **./cg generator patch ../[local do ficheiro de origem] [nível de tesselation] [nome do ficheiro 3D]**

3 *Engine*

3.1 VBOs

Como pedido nesta fase, alteramos a forma de como o *engine* desenha os modelos, sendo agora com VBOs (Vertex Buffer Objects) ao invés de modo imediato. Para isso, foram necessárias as seguintes alterações:

- deixar de utilizar as funções de transformações oferecidas pelo *OpenGL* (glRotatef, glTranslatef, glScalef), e foi criado o módulo "tMatrix", onde são construídas as matrizes das mesmas transformações, e aplicadas a cada ponto pela ordem correta;
- criação do vetor *vertices*, onde são colocados todos os vértices já transformados, de forma a que o desenho possa ser feito com VBOs.

De forma a que possam haver animações, concebemos a seguinte função:

```
    float t_seg = (float)(glutGet(GLUT_ELAPSED_TIME) - startTime) / 1000.0f;

    vertices.clear();
    verticeCount = 0;

    fillVertices(xml.models, t_seg);

    // Bind the existing buffer
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(float) * verticeCount * 3, vertice
}
```

Esta função mantém a cada *frame* o tempo decorrido desde o inicio, de forma a que seja possível as translações através da curva de Catmull-Rom, e as rotações temporizadas à volta de um eixo. Após atualizar os vértices, com todas as transformações, os *buffers* são atualizados.

3.2 Módulo *tMatrix*

Neste módulo, onde são efetuados todos os cálculos referentes a matrizes e matrizes de transformações, é possível encontrar as seguintes funções (apenas serão listadas as mais relevantes):

- *tMat_identity*, onde é criada uma matriz identidade 4 por 4.
- *tMat_translate*, onde é calculada uma matriz de translação, de acordo com a fórmula dada nas aulas teóricas.
- *tMat_scale*, onde é calculada uma matriz de escala, de acordo com a fórmula dada nas aulas teóricas.
- *tMat_rotate*, onde é calculada uma matriz de rotação em torno de um eixo arbitrário, de acordo com a fórmula dada nas aulas teóricas.
- *calcTransformsMatrix*, que calcula a matriz de transformações completa, pela ordem correta que são dadas nos ficheiros XML.

3.3 Atualizações nas estruturas de dados

De forma a que os novos tipos de translações e rotações fossem suportados pelo nosso *engine*, foi necessário uma atualização nas estruturas de dados.

Esta atualização passou por adicionar um campo *float time* (relevante para ambos os novos tipos de transformações), e o campo *bool align* (relevante para o novo tipo de translação).

Como é possível ver, as alterações foram poucas, o que mostra que as nossas estruturas de dados encontravam-se prontas para possíveis expansões, permitindo uma simples e fácil atualização das mesmas.

3.4 Transformações

3.4.1 Translação temporizada, ao longo de uma curva Catmull-Rom

```
<translate time="10" align="True" >
    <point x="1" y="0" z="1" />
    <point x="0.707" y="0.707" z="1" />
    <point x="0" y="1" z="1" />
    ...
    <point x="-1" y="0" z="1" />
</translate>
```

Será recebido um conjunto de pontos (no mínimo 4) que serão usados como pontos de controlo para definir a curva cúbica Catmull-Rom, assim como o número de segundos que o respetivo modelo deverá levar para percorrer toda a curva uma vez. O objetivo é realizar as animações baseadas nesta curva. O elemento *"align"* especifica se a direção do objeto estará alinhada com a curva.

De forma a que uma figura viaje sobre uma curva orientada de acordo à sua derivativa, utilizamos as seguintes funções:

- void catmull_rom_getPoint(float gt, vector<float>cp, float* pos, float* deriv)

Para obter a posição e derivativa na curva para o valor de tempo atual(momento), através desta função. Nesta função temos: **gt** que representa o instante, num intervalo [0,1], **cp** que representa os pontos de controlo da curva, **pos** onde será colocado o valor da próxima posição na curva, **deriv** que é um vetor tangente à curva, que também será preenchido com o cálculo da mesma.

- void getCatmullRomPoint(float t, float* p0, float* p1, float* p2, float* p3, float* pos, float* deriv)

Dado o instante e os pontos de controlo, com recurso á matriz de Catmull-Rom, calcula posição de um objeto e a sua derivada, utilizando as seguintes fórmulas (retiradas dos slides da aula prática 8):

$$pos(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p0 \\ p1 \\ p2 \\ p3 \end{bmatrix}$$

$$deriv(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p0 \\ p1 \\ p2 \\ p3 \end{bmatrix}$$

void catmull_rom_renderCurve(vector<float>ctrl_pts, const tMat& transfMatrix)

Esta função tem o intuito de desenhar a curva de Catmull-Rom. Esta apenas será desenhada por ativação da parte do utilizador, pressionando a tecla "c"para tar *toggle* a estas funcionalidade.

Nesta função, apenas são dados os pontos de controlo da curva e a matriz de transformação, de modo a que o desenho da curva, na presença de animações que interfiram com a mesma, seja atualizada conforme o instante da animação. Sem estas, a curva apenas seria desenhada na sua posição inicial.

- Aligning

De forma a que o objeto fique alinhado com a curva (ou seja, posicionado na direção da tangente), os seguintes cálculos tiveram de ser realizados:

```
float x[3] = {deriv[0], deriv[1], deriv[2]};
normalize(x);
float z[3];
cross(x, prev_y, z);
normalize(z);
float y[3];
cross(z, x, y);
normalize(y);

// Create the rotation matrix from the aligned axes
tMat m = tMat_identity();
```

```
m.m[0][0] = x[0]; m.m[0][1] = y[0]; m.m[0][2] = z[0]; m.m[0][3] = 0;
m.m[1][0] = x[1]; m.m[1][1] = y[1]; m.m[1][2] = z[1]; m.m[1][3] = 0;
m.m[2][0] = x[2]; m.m[2][1] = y[2]; m.m[2][2] = z[2]; m.m[2][3] = 0;
m.m[3][0] = 0; m.m[3][1] = 0; m.m[3][2] = 0; m.m[3][3] = 1;

// Apply the rotation matrix to the model
transformMatrix = tMat_multiply(transformMatrix, m);

memcpy(prev_y, y, 3 * sizeof(float));
```

O **x** vai corresponder à tangente da curva no ponto em que se encontra. o que irá fazer com que a frente do objeto fique alinhada com essa tangente.

O **z** vai corresponder a um vetor perpendicular a **x** e ao vetor **prev_y**, cujo valor na primeira iteração será (0,1,0). Nas seguintes iterações, terá valor igual ao **y** anterior.

O **y** será um vetor perpendicular a **x** e **z**.

3.4.2 Rotação temporizada à volta de um eixo arbitrário

```
<rotate time="10" x="0" y="1" z="0" />
```

Nesta transformação, o elemento *"time"* indica o tempo em segundos para realizar uma rotação completa, segundo o eixo especificado nos elementos *x*, *y* e *z*.

Como agora apenas temos o tempo para dar uma volta completa a um eixo, foi necessário calcular um ângulo que fosse "incrementado"ao longo do tempo, de forma igual a cada iteração, de modo a que o conjunto das várias rotações ao longo dos segundos apontados no campo *"time"* façam 360º. Para isso, chegamos à seguinte solução:

```
int number_of_times = (int)(current_time / t.time);
float tmp = (float)(current_time - number_of_times * t.time) / t.time;
tmp = tmp * 360;
```

Neste excerto de código, *"current_time"* é o tempo ocorrido desde o início da animação (recorrendo ao *glutGet(GLUT_ELAPSED_TIME)*), sendo que, de forma a que a transição entre o fim de uma volta e o início da segunda seja suave, é calculado o *"number_of_times"*, de forma a que *tmp* seja sempre um valor no intervalo [0,1], e multiplicado por 360 de forma a que nos dê o ângulo da próxima posição referente ao eixo. Apesar da função *glutGet(GLUT_ELAPSED_TIME)* devolver o tempo em milissegundos, estamos a utilizar o tempo em segundos, pois, para o grupo, utilizar segundos é mais conveniente para definir a duração e a velocidade da animação, proporcionando um controlo mais preciso sobre o movimento.

4 Sistema solar e resultados

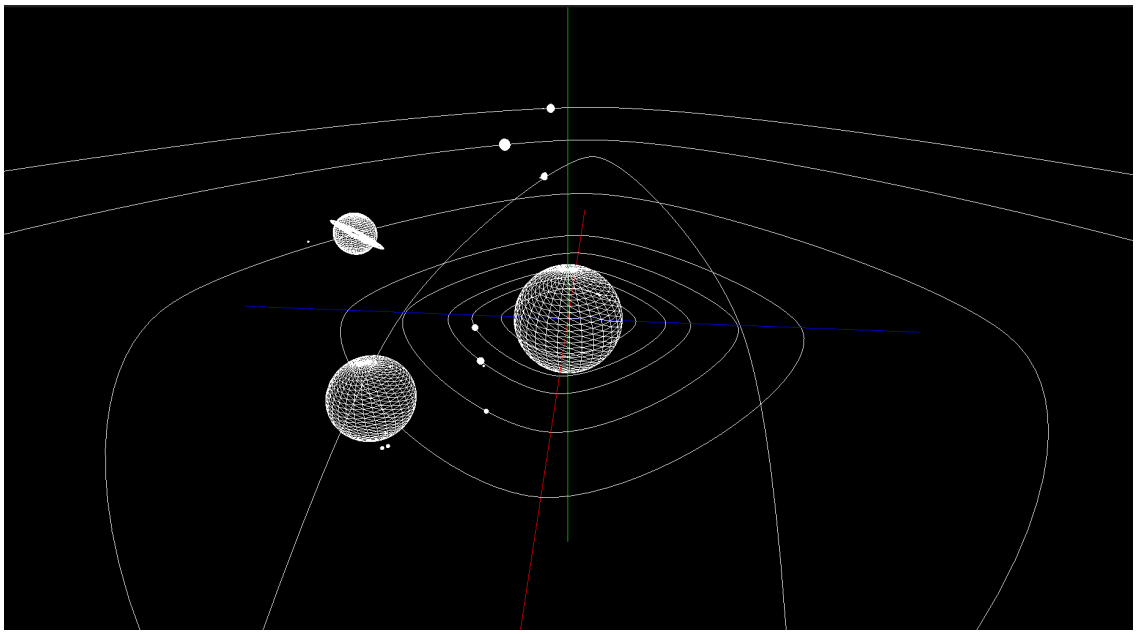

Figura 1: Representação so Sistema Solar através do programa criado

Como é possível ver na figura acima, o nosso programa é capaz de criar um esquema do sistema solar no qual são utilizadas e por isso demonstradas as transformações de modelos requeridas nesta fase do projeto.

Todos os exemplos providenciados pelo professor foram também testados e comprovados que funcionam corretamente.

4.0.1 Cálculos das translações

Por convenção, estabelecemos que a translação da Terra na nossa simulação tem duração de 5 segundos. Com isso, podemos calcular de forma precisa a duração das translações de todos os corpos celestes presentes na simulação:

Planeta	Período de translação (dias)	Tempo (s)
Mercúrio	87,97	1,204210699
Vênus	224,7	3,075891146
Terra	365,26	5
Marte	686,6888	9,4
Júpiter	4331,9836	59,3
Saturno	10760,5596	147,3
Urano	30685,4926	420,05
Netuno	60191,1954	823,95
Plutão	90799,9834	1242,95

Tabela 1: Períodos de translação dos planetas e seus equivalentes em segundos.

Satélite	Período de translação (dias)	Tempo (s)
Lua	27,32	0,373980179
Ío	1,769138	0,024217516
Europa	3,551	0,04860921
Ganímede	7,154553	0,097937811
Calisto	16,6890184	0,228453956
Titã	15,945	0,218269178
Tritão	-5,87685	-0,080447489

Tabela 2: Períodos de translação de alguns satélites e seus equivalentes em segundos.

5 Conclusão

Aproximando-nos agora do fim do projeto, começamos a ver uma definitiva consolidação dos conhecimentos adquiridos até agora, assim como uma maior facilidade na sua implementação no projeto.

Durante a implementação da função responsável por tratar dos *Bezier Patches*, deparámo-nos com alguns problemas uma vez que os cálculos necessários são bastante complexos de perceber e implementar. Não só isto, mas também, no **Engine**, encontramos algumas dificuldades na implementação das curvas Catmull-Rom, bem como o *aligning* to objeto com a curva.

Um possível problema que o programa poderá vir a encontrar é a sua eficiência, visto serem calculados todos os vértices em todos os *frames*. Como estão a ser efetuados cálculos complexos, estes podem representar um *bottleneck* para a *performance* do programa.