Computação Gráfica (3º ano de MIEI)

Terceira Fase

Relatório de Desenvolvimento

André Gonçalves (a80368)

João Queirós (a82422)

Luís Alves (a80165)

Rafaela Rodrigues (a80516)

20 de Abril de 2019

Resumo

Este relatório inicia-se com uma breve contextualização, seguindo-se a descrição dos problemas propostos e o que deve ser desenvolvido para os solucionar. De seguida são descritas as alterações efetuadas às duas aplicações previamente desenvolvidas: o gerador e o motor gráfico. Para o gerador é indicado o método de obtenção de vértices e índices de cada primitiva, e para o motor gráfico são indicadas as alterações realizadas para suportar translações dinâmicas, VBOs e rotações dinâmicas. Por fim, é apresentado a nova cena do Sistema Solar, terminando este relatório com uma conclusão sobre o trabalho desenvolvido, apontando os seus pontos fortes e fracos, bem como dificuldades sentidas.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

O presente relatório foi elaborado no âmbito da Terceira Fase do Trabalho Prático da Unidade Curricular de Computação Gráfica, que se insere no 2º semestre do 3º ano do primeiro ciclo de estudos do Mestrado Integrado em Engenharia Informática.

1.2 Objetivos e Trabalho Proposto

Pretende-se com este relatório formalizar toda a análise e modelação envolvida na construção da solução ao problema proposto, incluindo estruturas de dados e algoritmos utilizados.
Nesta terceira fase foi proposta a criação de modelos baseados em *patches* de Bezier, bem como a extensão das transformações geométricas translação e rotação. Para além disso, também foi alterado o modo de desenho dos modelos, passando de modo imediato para VBOs indexados.

1.3 Resumo do trabalho a desenvolver

Uma vez que é necessário o desenvolvimento de um novo tipo de modelo, será necessário alterar o programa *generator* para, dado um ficheiro *patch* poder gerar uma superfície de Bezier. Para além disso, também será necessário alterar as restantes primitivas para gerarem os vértices e os índices necessários para serem renderizados com recurso a VBOs.
Quanto ao *engine*, será alterado o *parsing* dos ficheiros .3d de forma a ter em conta os vértices/índices de cada modelo, será alterado o *parsing* das rotações e translações para poderem incluir curvas de Catmull-Rom e rotações sobre o próprio eixo, e será alterado o modo de desenho para VBOs indexados.

Capítulo 2

Arquitetura da Solução

Esta fase pode ser vista como tendo duas componentes de desenvolvimento distintas: a geração de modelos com recurso a VBOs indexados e a alteração do motor gráfico para suportar cenas dinâmicas. As alterações feitas são reportadas de seguida.

2.1 Gerador

Tendo em conta que agora se pretende renderizar os modelos com recuso a VBOs indexados, era necessário que a estrutura do ficheiro .3d gerado pela aplicação fosse alterada. Por isso, primeiramente surgem os vértices, com cada componente (X, Y e Z) separada por um espaço e cada vértice separado por uma quebra de linha. De seguida surgem os índices, sendo que cada índice é separado por uma quebra de linha.

2.1.1 Primitivas da 1ª Fase

Para cada primitiva da 1ª fase, foram alterados os métodos de geração de vértices de forma a reduzir as suas repetições. Por isso, para cada uma das primitivas, ao invés de se realizarem os cálculos tal como explanado na 1ª Fase para a definição dos vértices, foram efetuadas algumas alterações, que se resumem de seguida.

Plano

Para obter os vértices do plano(centrado na origem), divide-se o lado por 2, obtendo-se assim os valores (em módulo) das coordenadas de x e z de cada vértice, sendo que y é constante e igual a 0. O sinal de x e z é então ajustado adequadamente para obter os quatro vértices necessários.

```
x = size / 2;
z = size / 2;

guardaVertice(x,0,z);
guardaVertice(-x,0,-z);
guardaVertice(-x,0,z);
guardVertice(x,0,-z);
```

Quanto aos índices, a sua criação é dada pelo seguinte algoritmo, que desenha os dois triângulos segundo a regra da mão direita de forma a que o plano seja visível de Y positivo

```
guardaIndice(0);
guardaIndice(1);
guardaIndice(2);
guardaIndice(1);
guardaIndice(0);
guardaIndice(3);
```

Caixa

Para a criação de uma caixa, cujo centro é a origem do referencial, foram utilizados 4 parâmetros: o número de *divisions*, o comprimento, a altura e a largura da caixa. As *divisions* representam o número de partes em que está divida cada face da caixa. Tendo isto em consideração, cada face será constituida pelo seguinte número de quadriláteros:

$$divisions^2$$

Cada quadrilátero tem, para cada lado, as seguintes dimensões:

$$nx = x/div; \tag{2.1}$$

$$ny = y/div; \tag{2.2}$$

$$nz = z/div; \tag{2.3}$$

Para obter os vértices da caixa a estratégia utilizada foi a de calcular para cada face todos os seus vértices com base no número de *divisions* e dos vértices mais "positivos"para os mais "negativos".
Exemplo das faces do plano XZ:

```
nx = x / divisions;
nz = x / divisions;
x0 = x / 2;
y0 = y / 2;
z0 = z / 2;
// Face de Cima
for(i = 0; i <= divisions; i++)
    for(j = 0; j <= divisions; j++)
        guardaVertice(x0 - i * nx,y0,z0 - j * nz);
// Face de baixo
for(i = 0; i <= divisions; i++)
    for(j = 0; j <= divisions; j++)
        guardaVertice(x0 - i * nx, -y0, z0 - j * nz);
```

De forma análoga é feita a geração dos vértices para as restantes faces (do plano XY e YZ). É importante notar que há então $(divisions + 1)^2$ vértices em cada face.
Posteriormente procedemos à elaboração dos índices, sendo que, como para cada plano, é gerada primeiro uma face que será visível para uma orientação, e depois para a orientação contrária, bastou uma só função para gerar todos os índices de todas as faces. É a que se apresenta de seguida.

```
currentIndex = 0;
for(numPlanos = 0; numPlanos < 3; numPlanos++)
```

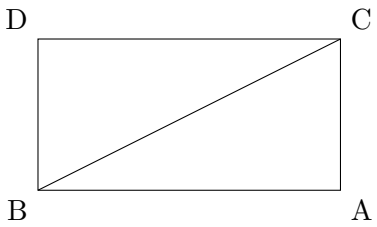

Figura 2.1: Quadrilátero na Caixa

```
    for (i = 0; i < divisions; i++)
        for (j = 0; j < divisions; j++)
             indexA = i * (divisions + 1) + j + currentIndex;
             indexB = (i + 1) * (divisions + 1) + j + currentIndex;
             indexC = indexA + 1;
             indexD = indexB + 1;
             guardaIndice(indexA);
             guardaIndice(indexC);
             guardaIndice(indexB);

             guardaIndice(indexB);
             guardaIndice(indexC);
             guardaIndice(indexD);

    currentIndex += (divisions + 1) * (divisions + 1);

// Face com orientaçao contraria
    for (i = 0; i < divisions; i++)
        for (j = 0; j < divisions; j++)
        indexA = i * (divisions + 1) + j + currentIndex;
         indexB = (i + 1) * (divisions + 1) + j + currentIndex;
         indexC = indexA + 1;
         indexD = indexB + 1;

         guardaIndice(indexD);
         guardaIndice(indexC);
         guardaIndice(indexB);

         guardaIndice(indexC);
         guardaIndice(indexA);
         guardaIndice(indexB);

    currentIndex += (divisions + 1) * (divisions + 1);
```

É possível congregar esta geração de índices num ciclo com 3 iterações devido a uma propriedade obtida com a geração dos vértices: uma sequência de vértices de uma face é seguida de uma sequência de vértices da face oposta, pelo que a disposição dos triângulos é a oposta, sendo por isso utilizadas sequências diferentes de vértices de um dado quadrilátero.

Cone

No caso do cone, a estratégia usada para obter os vértices baseia-se no número de slices e stacks.
Primeiramente, é calculado o vértice central da base do cone. De seguida, para cada slice, são calculados tantos vértices quanto o número de stacks, sendo o algoritmo o seguinte:

```
guardaVertice(0,-altura/2,0,file);

for(i=0; i<= slices;i++){
    for (j = 0; j < stacks +1; j++) {
        div = j / stacks;
        radius = (1.0 - (div)) * botRad;

        float x = radius * cos(i * angle);
        float y = j * division - baseY;
        float z = radius * sin(i * angle);

        guardarVertice(x,y,z,file);
    }
}
```

Relativamente aos índices, o algoritmo criado permite-nos ligar os vértices de uma slice com os da slice seguinte. Para ligar a última slice à primeira, foram guardados novamente os índices da primeira slice.
Para os triângulos que constituem a base do cone, os seus índices são calculados para cada slice e antecedem os índices da superfície lateral dessa mesma face.
O exemplo seguinte demonstra a ordem pelos quais os vértices serão guardados:
O algoritmo usado foi:

```
for(i=0;i<k;i++){
//para ligar última slice à primeira
if (i == slices)
    l = 0;
else
    l = i;

//índices da base
b1=0;
b2=k*l+1;
b3=k*(l+1)+1;
guardaIndice(b1);
guardaIndice(b2);
guardaIndice(b3);

for(j=1;j<stacks+1;j++){

    iA= k*l+j;
```

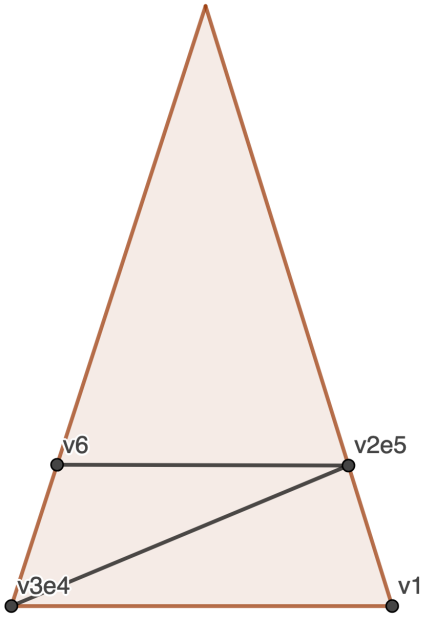

Figura 2.2: Exemplificação do algoritmo para índices do cone

```
            iB= k*l+j+1;
            iC= k*(l+1)+j;
            iD= k*(l+1)+j+1;

            guardaIndice(iA);
            guardaIndice(iB);
            guardaIndice(iC);
            guardaIndice(iC);
            guardaIndice(iB);
            guardaIndice(iD);
        }
    }
```

Esfera

Para obter os vértices de uma esfera, é utilizado o seguinte algoritmo:

```
    for(stack = 0; stack <= stacks; stack++)
        beta = stack * (pi / stacks)
        for(slice = 0; slice <= slices; slice++)
            alfa = slice * (2 * pi / slices)
            x,y,z = calculaCoordenadasEsfericas(raio,beta,alfa)
            guardaVertice(x,y,z)
```

Para cada *stack* e para cada *slice* é repetido o primeiro vértice no fim de cada iteração de forma a simplificar o cálculo dos índices, que é o que se apresenta de seguida:

```
for(stack = 0; stack < stacks; stack++)
    for(slice = 0; slice < slices; slice++)
        indexA = stack * (slices + 1) + slice;
        indexB = (stack + 1) * (slices + 1) + slice;
        indexC = indexA + 1;
        indexD = indexB + 1;

        guardaIndice(A);
        guardaIndice(B);
        guardaIndice(C);

        guardaIndice(B);
        guardaIndice(D);
        guardaIndice(C);
```

Os índices A, B, C e D são os que se apresentam na figura 2.3.

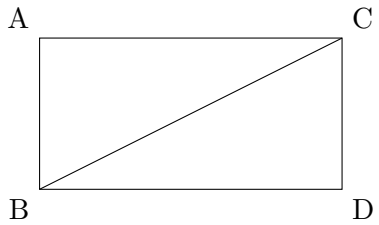

Figura 2.3: Quadrilátero na Esfera

2.1.2 Patches de Bezier

De forma a ser possível gerar um modelo a partir de Patches de Bezier, é necessário processá-lo, sendo que são armazenados os pontos de controlo num `vector` de `Vertex`, e os índices num `vector` de inteiros.

Tendo esses dois vetores e um nível de tesselação, é possível gerar um conjunto de vértices e índices que correspondem a uma superfície de Bezier.

O nível de tesselação representa o número de intervalos entre 0 e 1, sobre os quais vão variar u e v. Assim, tal como nas primitivas da primeira fase, são gerados inicialmente os vértices que compõem a superfície através do seguinte algoritmo.

```
for(i = 0; i < numero de indices; i += 16)
    for(a = 0; a < 4; a++)
        for(b = 0; b < 4; b++)
            indice = indices[i + a * 4 + b];
            verticesX = pontos de controlo[indice] -> x;
            verticesY = pontos de controlo[indice] -> y;
            verticesZ = pontos de controlo[indice] -> z;

    for(u = 0; u <= nível de tesselação; u++)
        for(v = 0; v <= nível de tesselação; v++)
            getBezier(u / nível de tesselação, v / nível de tesselação,
```

```
            verticesX, verticesY, verticesZ, coordenadas);
        guardaVertice(coordenadasX, coordenadasY, coordenadasZ);
```

Em relação ao algoritmo, são avançados 16 índices em cada iteração do ciclo exteror, uma vez que para cada patch, são necessários 16 vértices. De seguida, são copiados os componentes X, Y e Z de cada um dos 16 vértices do patch para 3 matrizes (uma para cada um dos componentes). Posteriormente, (nível de tesselação + 1) × (nível de tesselação + 1) vezes, são calculados esse número vértices que devem fazer parte da superfície de Bezier. Tal como nas primitivas anteriores, os primeiros vértices são repetidos no fim para evitar cálculos extra na escrita dos índices.

Tendo então os vértices, é necessário calcular os índices para posteriormente ser possível interpretar o modelo como uma superfície definida por triângulos. É seguido o seguinte algoritmo para esse cálculo:

```
indices por patch = (nível de tesselação + 1) ^ 2;
for(i = 0; i < número de índices; i += 16)
    patch = i / 16;
    for(u = 0; u < nível de tesselação; u++)
        for(v = 0; v < nível de tesselação; v++)
            indexA = patch * indices por patch + (nível de tesselação + 1) * u + v;
            indexB = indexA + 1;
            indexC = patch * indices por patch +
                (nível de tesselação + 1) * (u + 1) + v;
            indexD = indexC + 1;

            guardaIndice(A);
            guardaIndice(C);
            guardaIndice(B);

            guardaIndice(C);
            guardaIndice(D);
            guardaIndice(B);
```

Os índices A, B, C e D são os que se apresentam na figura 2.4.

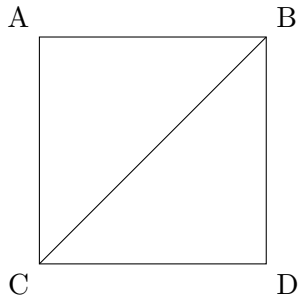

Figura 2.4: Quadrilátero na superfície de Bezier

Quanto ao cálculo de cada um dos pontos de Bezier, este segue a seguinte equação:

$$B(u,v) = U \times M \times P(x,y,z) \times M^T \times V \tag{2.4}$$

M é a matriz de Bezier, P a matriz composta pelos dezasseis pontos de cada patch e M^T é a transposta da matriz de Bezier (que, por ser simétrica, é igual à matriz de Bezier).

Já U e V, são os vetores [u^3 , u^2 , u , 1] e [v^3 , v^2 , v , 1], para cada iteração de u e de v.

Posto isto, é efetuada a multiplicação de matrizes por vetores, sendo que a multiplicação de P com o resultado da multiplicação de M×V é efetuada 3 vezes, uma para cada componente do vértice (X, Y e Z). O mesmo acontece na multiplicação seguinte, M×PMV.

Por fim, o valor da coordenada do ponto é obtido multiplicando o vetor U com o vetor resultante das multiplicações anteriores, sendo efetuadas também aqui 3 multiplicações.

Todo este processo é o que se ilustra no seguinte algoritmo:

```
dados u, v, P(x), P(y) e P(z):
    V = { v^3, v^2, v, 1};
    MV = multiplicaMatrizPorVetor(M,V);

    PMV[x] = multiplicaMatrizPorVetor(P(x),MV);
    PMV[y] = multiplicaMatrizPorVetor(P(y),MV);
    PMV[z] = multiplicaMatrizPorVetor(P(z),MV);

    MPMV[x] = multipicaMatrizPorVetor(M,PMV[x]);
    MPMV[y] = multipicaMatrizPorVetor(M,PMV[y]);
    MPMV[z] = multipicaMatrizPorVetor(M,PMV[z]);

    U = { u^3, u^2, u, 1};
    x = multiplicaVetorPorVetor(U,MPMV[x]);
    y = multiplicaVetorPorVetor(U,MPMV[y]);
    z = multiplicaVetorPorVetor(U,MPMV[z]);
```

Estão então calculadas as coordenadas X, Y e Z de (u,v).

2.2 Motor gráfico

Em relação ao motor gráfico, foi necessário alterar o seu modo de renderização para utilizar VBOs indexados e a interpretação de translações e rotações. As alterações realizadas descrevem-se de seguida.

2.2.1 VBOs

De forma a ser possível acomodar VBOs, foi alterada a estrutura `Model`, uma vez que esta apenas continha uma cor, um nome e um vetor de `Vertex`. Assim, passou a não ter um vetor de `Vertex` mas sim dois vetores, um de *floats*, sendo que cada sequência de 3 floats representa uma componente x, y e z de um dado vértice; um vetor de *unsigned int*, que contém os índices associados a este modelo; e dois buffers, utilizados pelo GLUT: um para os índices e outro para os vértices.

Para além disso, foram adicionados dois métodos à classe *Modelo*. O primeiro, `initializeVBO`, copia o conteúdo dos vetores de vértices e índices para os buffers criados, através das funções `glBufferData` e `glGenBuffers`. O segundo, `drawVBO`, utiliza a função `glDrawElements` para poder desenhar o conteúdo dos buffers anteriormente alocados.

2.2.2 Translação

Para uma translação poder ser feita ao longo de diversos pontos que compõem uma curva de Catmull-Rom, também foram necessárias algumas modificações à estrutura Translação.

Assim, foi adicionada uma versão extendida da anteriormente desenvolvida, sendo que agora são armazenados pontos (caso seja uma translação dinâmica) num vetor de `Vertex`, são armazenadas as coordenadas do vetor Y anterior e o tempo que deverá demorar uma translação.
Para calcular a posição de um determinado grupo num dado tempo, foi desenvolvida uma função, `get GlobalCatmullRomPoint`, que dado um instante de tempo, devolve a posição e a derivada do objeto nesse instante de tempo nessa posição.
De forma a obter a proporção do instante de tempo atual, com o instante de tempo na curva, recorre-se à equação que a seguir se apresenta, sendo *time* o tempo registado no ficheiro .xml.

$$t = glutGet(GLUT\_ELAPSED\_TIME)/time; \tag{2.5}$$

Por fim, é utilizada a mesma função desenvolvida para as aulas práticas, em que a posição é dada por:

$$pos = T \times M \times P \tag{2.6}$$

Sendo T o vetor [t^3 , t^2 , t , 1], M a matriz de Catmull-Rom, e P as coordenadas dos pontos de controlo.
A derivada é obtida através da equação que se segue:

$$deriv = dT \times M \times P \tag{2.7}$$

Sendo dT o vetor [$3 \times t^2$, $2 \times t$, 1 , 0], M a matriz de Catmull-Rom e P as coordenadas dos pontos de controlo.

2.2.3 Rotação

Relativamente à extensão da rotação, a única alteração feita foi a adição de uma variável de tempo, que representa o valor $\times 10^3$ que um modelo demora a efetuar uma rotação sobre o eixo definido de 360º. Para a qualquer instante se obter qual o grau de rotação desejado, recorre-se à seguinte equação:

$$ang = glutGet(GLUT\_ELAPSED\_TIME) \times 360/time \tag{2.8}$$

Sendo *time* o valor definido para ser efetuada a rotação.

2.3 Sistema Solar

No que diz respeito ao sistema solar, as alterações visíveis relativamente à fase anterior são significativas.
Cada planeta do nosso sistema (e respetiva lua), percorre uma órbita e possui rotação sobre si mesmo. Para além disso, foi adicionado o cometa Teapot, que também possui uma órbita à volta do sol.
Para além disso, todos os planetas, luas, anéis e cometa foram criados a partir das suas primitivas usando VBOs.

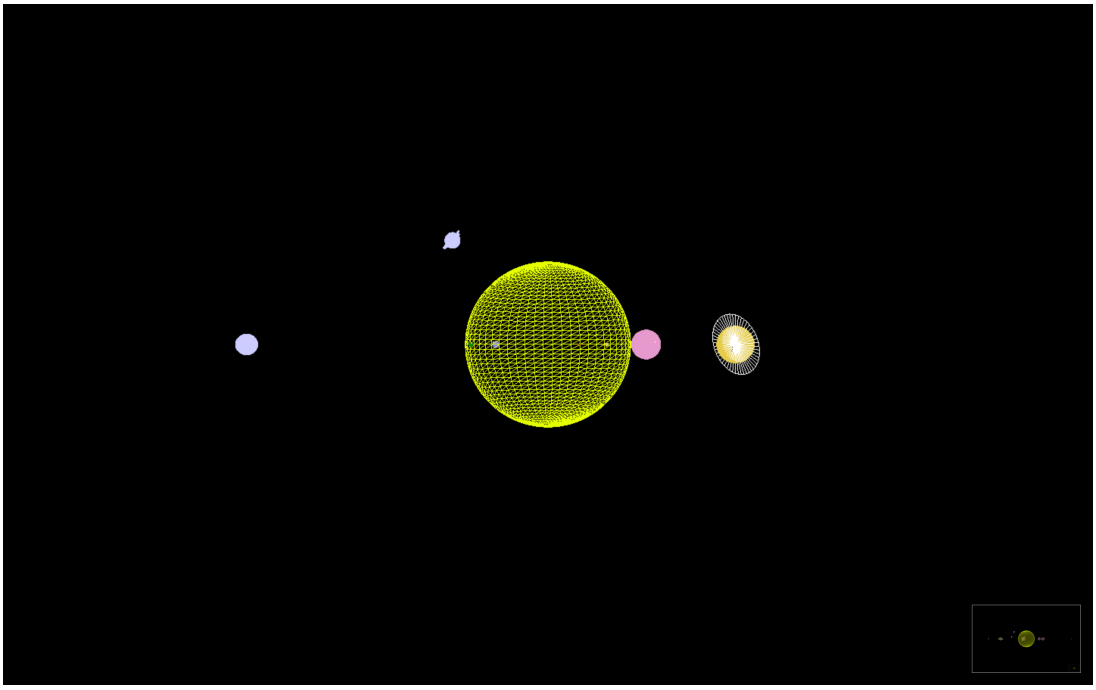

Figura 2.5: Sistema Solar

Capítulo 3

Conclusão

Com este trabalho foi possível aplicar os conhecimentos teóricos relativamente a curvas de Bezier e Catmull-Rom, sendo que se revelou mais moroso o trabalho efetuado para alterar o gerador do que o motor gráfico em si. Isto deveu-se à alteração da geração de vértices para VBOs indexados, e à maior complexidade na geração de superfícies de Bezier. Quanto ao motor gráfico, as alterações não foram muito profundas, tendo sido alterado o método de execução de transformações geométricas para as translações e rotações, passando cada uma delas a conter um método que trata da atualização das matrizes resultantes dessas transformações.