Fase 3
Catmull-Rom Curves, *Bézier Patches* e Funcionalidades Adicionais

Grupo 42
Ângelo Sousa[A80813], Diogo Ribeiro[A84442], Rui Mendes[A83712], and Rui Reis[A84930]

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a80813,a84442,a83712,a84930}@alunos.uminho.pt

Resumo Extensão do motor gráfico às transformações com fator temporal, curvas de *Catmull-Rom*, *patches* de *Bézier*, funcionalidades extra em XML e figuras adicionais do *generator*. Bem como a aplicação desses mecanismos à demonstração tecnológica.

1 Introdução

Ao contrário de fases anteriores, pretende-se agora adicionar um comportamento mais fluido e realista. Nomeadamente, através da imposição do fator temporal em todo o tipo de transformações geométricas. Desta forma, é possível distribuir os efeitos de uma transformação geométrica ao longo do tempo.

Para além disso, é imperioso que qualquer sistema de modelagem seja capaz de construir curvas e superfícies, que subsequentemente definem trajectórias e objetos, respectivamente. Pois, só desta forma somos capazes de definir curvas com um numero finito de pontos, através da interpolação entre estes.

Consequentemente, começamos com o debate da teoria, e respectiva implementação, subjacente a todas as componentes requeridas nesta fase. Passando de seguida, para as componentes extra que decidimos embutir no sistema.

Por fim, apresentamos a concretização destes comportamentos num caso prático, nomeadamente como extensão da demonstração tecnológica construída anteriormente.

2 Transformações com Tempo

De forma a conseguir incorporar a dimensão temporal nas transformações, foi primeiro necessário compreender que é possível redefinir a definição de passagem do tempo quando possuímos um tempo limite máximo, que é na prática a situação que temos.

Assuma-se que uma dada transformação é suposto demorar 100 segundos, então a passagem do tempo, segundo esta transformação, pode ser relativizada.

Por isso, em vez de dizermos que passaram 10 segundos, dizemos que passou 10% do tempo total pretendido.

O que isto nos permite fazer, é definir o tempo como um factor de proporcionalidade t, sendo que $t \in [0, 1]$, pois no máximo passou 100% do tempo da transformação, neste caso pode-se parar de aplicar a transformação, ou, simplesmente, reiniciar de novo a transformação com $t \leftarrow 0$, permitindo assim um *loop* de transformações, interessante e ideal para a rotação e translação de planetas.

Finalmente, pode-se resumir este comportamento a uma simples formula, correspondente à mecânica utilizada no *engine*.

$$t_i \leftarrow t_{i-1} + t \qquad , \forall t_{i-1} \leq 1 \tag{1}$$

Ou seja, ao fator da iteração anterior é somado o fator de proporcionalidade que se verifica nesta iteração. Sendo que t corresponde à proporção do tempo que passou desde a última iteração, segundo o tempo total associado à transformação.

3 *Catmull-Rom Curves*

Relativamente a curvas de *Catmull-Rom*, definimos a classe `CRCSpline` que visa encapsular estes métodos. Sendo que são mantidas todas as características referentes a este tipo de curva. Nomeadamente, o seguinte comportamento:

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

$$P^*(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{3}$$

$$\nabla P^*(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} M \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{4}$$

Aplicando-se as respectivas propriedades de translação e rotação para deslocar e orientar um dado referencial, de acordo com a curva desejada. Este ajustamento, referente à orientação do referencial segundo a tangente da curva e a sua respectiva deslocação ao longo da mesma, pode ser obtido através do seguinte algoritmo.

Algorithm 1: Algoritmo para reajustamento do referencial na curva.

Input: $t \in [0,1]$, Y_{i-1}
Result: Posição e orientação reajustadas
$P \leftarrow P^*(t)$;
$X_i \leftarrow \|\nabla P^*(t)\|$;
$Z_i \leftarrow \|X_i \times Y_{i-1}\|$;
$Y_i \leftarrow \|Z_i \times X_i\|$;
$Y_{i-1} \leftarrow Y_i$;
$M \leftarrow \{X_i, Y_i, Z_i\}$;
glTranslatef(P_x,P_y,P_z);
glMultMatrixf(M);

4 *Bézier Patches*

Relativamente às *Patches* de *Bézier*, consideramos apenas a existência e definição de *patches* bicúbicas, ou seja, *patches* que são totalmente definidas através de 16 pontos de controlo. Este é o tipo de geometria mais utilizada, sendo por isso a única coberta aqui. Excluindo assim *patches* com uma geometria biquadrática.

Uma *patch* de *Bézier* pode ser vista como uma como uma interpolação com parâmetro $v \in [0,1]$ entre 4 curvas de *Bézier* distintas com parâmetro $u \in [0,1]$. Desta forma, surge naturalmente a seguinte definição de *patch*.

$$B(u,v) = \sum_{j=0}^{3} \sum_{i=0}^{3} B_i(u) P_{ij} B_j(v) \tag{5}$$

Assumindo U e V como sendo produtos do functor F, aplicado a u e v, respectivamente. Verificamos que:

$$F(x) = \begin{bmatrix} x^3 \\ x^2 \\ x \\ 1 \end{bmatrix} \tag{6}$$

$$U = F(u)^T \tag{7}$$

$$V = F(v) \tag{8}$$

Com estas definições em mente, é possível derivar uma definição mais explícita desta superfície.

$$B(u,v) = (UM)P(M^T V) \tag{9}$$

O que apresenta uma definição computacionalmente atrativa. Porém, como afinamento final decidimos que esta equação é transversal, em formato, também às derivadas parciais da superfície, pelo que concluímos que uma transformação T

que tenha em consideração estas variações de argumento é imperiosa em termos de eficiência. Por isso, através da seguinte derivação de $T(\gamma, \delta)$, conseguimos obter um comportamento que pode ser diretamente programado na linguagem escolhida. Assumindo $\alpha = \gamma M$ e $\beta = M^T \delta$, obtemos:

$$\begin{aligned}
T(\gamma, \delta) &= (\gamma M) P (M^T \delta) \\
&= \alpha P \beta \\
&= \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \\
&= \begin{bmatrix} \alpha_0 P_{00} & \alpha_1 P_{01} & \alpha_2 P_{02} & \alpha_3 P_{03} \\ \alpha_0 P_{10} & \alpha_1 P_{11} & \alpha_2 P_{12} & \alpha_3 P_{13} \\ \alpha_0 P_{20} & \alpha_1 P_{21} & \alpha_2 P_{22} & \alpha_3 P_{23} \\ \alpha_0 P_{30} & \alpha_1 P_{31} & \alpha_2 P_{32} & \alpha_3 P_{33} \end{bmatrix}^T \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=0}^{3} \alpha_i P_{0i} & \sum_{i=0}^{3} \alpha_i P_{1i} & \sum_{i=0}^{3} \alpha_i P_{2i} & \sum_{i=0}^{3} \alpha_i P_{3i} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} \\
&= \sum_{j=0}^{3} \left[\beta_j \sum_{i=0}^{3} \alpha_i P_{ji} \right]
\end{aligned}$$

Através desta transformação, obtemos a capacidade de sucintamente descrever a superfície e respectivas derivadas parciais com uma maior facilidade.

$$B(u, v) = T(U, V) \tag{10}$$

$$\frac{\partial B(u, v)}{\partial u} = T(\frac{\partial U}{\partial u}, V) \tag{11}$$

$$\frac{\partial B(u, v)}{\partial v} = T(U, \frac{\partial V}{\partial v}) \tag{12}$$

Pelo que é visível a utilidade desta transformação T.

Finalmente, podemos definir as normais à superfície como o produto externo entre as derivadas parciais.

$$N = \frac{\partial B(u, v)}{\partial u} \times \frac{\partial B(u, v)}{\partial v} \tag{13}$$

Cujo respectivo versor é:

$$\hat{N} = \frac{N}{\|N\|} \tag{14}$$

4.1 Implementação no *Generator*

Em termos práticos, todo este comportamento é contido na classe `BezierSuf` do *generator*. Especificamente, define-se a transformação $T(\gamma, \delta)$ através do metódo `multMatrixes`. O que permite as seguintes definições sucintas:

```
Vec3 BezierSuf::getBezierSufPoint(float u, float v,
    int idPatch) {
    return multMatrixes(getCubic(u), getCubic(v),
                                idPatch);
}
```

Permitindo também uma definição sucinta do método que calcula o vetor normal à superficie.

```
Vec3 BezierSuf::getBezierSufNorm(float u, float v,
    int idPatch) {
    Vec3 bu = getBezierU(u, v, idPatch);
    Vec3 bv = getBezierV(u, v, idPatch);

    return (bu.crossprod(bv)).normalize();
}
```

4.2 Criação de Objetos com *Patches* de *Bézier*

Para gerar objectos 3D utilizando *patches* é necessária a criação prévia de um ficheiro com a formatação indicada pelo docente da UC, no qual são indicadas todas as *patches* necessárias, bem como os respectivos pontos de controlo. De seguida, o `generator` deve ser invocado com os seguintes parâmetros.

```
generator patch {patch name} {tesselation} {output file}
```

Por exemplo, para gerar um *teapot* com *tesselation level* de 10 invocariamos o `generator` da seguinte forma, assumindo que a especificação do *teapot* está no ficheiro `teapot.patch`.

```
generator patch teapot.patch 10 teapot.3d
```

5 Funcionalidades adicionais em XML

Nesta fase, para que o nosso *parser* fosse capaz de processar as novas funcionalidades, foram feitas algumas alterações ao modelo anterior.

As novas funcionalidades consistem em definir a posição inicial da câmara e definir cores aos diferentes modelos.

De modo a que estas funcionalidades fossem implementadas, foram adicionadas novas funções na *engine*. Para a posição inicial da câmera são lidas as coordenadas da mesma, definidas no ficheiro XML, e depois através da função

initialCamera é inicializada a câmera com as mesmas. Para definir as cores nos modelos, foi acrescentado à classe *DrawEvent* três inteiros que representam cada um, o valor RGB respetivo. Para a cor ser devidamente impressa, foi acrescentado à função *popDraw*, na classe *Group*, uma *glColor3f* com os respetivos valores RGB do *DrawEvent*. Para que os valores RGB sejam atribuídos ao respetivo modelo que vai ser desenhado, devem ser passados como atributo do mesmo, no ficheiro XML.

6 Figuras Adicionais do *Generator*

6.1 Torus

De modo a complementar o nosso módulo `generator` decidimos adicionar a possibilidade de gerar um Torus, sendo que fornecemos os seguintes parâmetros para gerar a figura: innerRadius (distância da origem à circunferência interior), outerRadius (distância da origem à circunferência exterior), height (espessura do torus), stacks (o numero de lados de cada anel), e slices (o numero de anéis do torus).

Sendo `r = height/2`, `R = centro de cada anel`, cada vértice é dado por:

```
x = (R + r * cos(phi)) * cos(theta);
y = r * sin(phi);
z = (R + r * cos(phi)) * sin(theta);
```

Sendo que theta é o ângulo que o vértice faz com a origem, e phi é o angulo que o vértice faz com o centro do slice.

Sendo que estamos a considerar a utilização de VBO's, também geramos a ordem pela qual os vertices são desenhados. O processo repete-se para cada slice sendo que apenas no último slice diverge. Sendo que `verts = nº vértices`, o código elaborado é o seguinte:

```
    int j = 0;
    for (j = 0; j < verts; j+=(stacks+1)) {
        if (j < verts-(stacks+1)) {
            for (int i = 0; i < stacks; i++)
                Torus::addIndex(j + i, j + i + 1,
                j + i + (stacks + 2));

            for (int i = 0; i < stacks; i++)
                Torus::addIndex(j + i + stacks + 2,
                j + i + (stacks + 1), j + i);
        } else {
            for (int i = 0; i < stacks; i++)
                Torus::addIndex(j+i,j+i+1,i+1);

            for (int i = 0; i < stacks; i++)
                Torus::addIndex(i+1,i,j+i);
```

```
        }
    }
```

Seguindo a lógica do excerto de código acima apresentado, obtemos a ordem pela qual devemos representar cada vértice.

7 *Path Tracing*

Em OpenGL não há diferenciação entre a matriz *model*, matriz referente ao posicionamento do referencial no sistema, e a matriz *view*, matriz referente ao posicionamento da câmara. Desta forma, OpenGL verifica só uma matriz *modelview*, correspondente a uma agregação das duas matrizes definidas acima.

Porém, este tipo comportamento não é ideal se o nosso desejo for rastrear a trajectória seguida por um dado grupos de objectos, por isto precisamos de ter acesso direto à matriz *model*, o que obviamente não possuímos. Para ultrapassar este problema o que fazemos é guardar a matriz *view*, sendo que esta é definida primeiro, e, sucessivamente, aplicamos todas as transformações associadas a um dado grupo sobre uma matriz identidade, obtendo assim a matriz *model*.

Partindo da matriz *model*, conseguimos extrair as suas respectivas *real world coordinates* e adicionar um ponto, referente ao ponto atravessado naquele instante, por um dado grupo.

Tendo em conta que as transformações geométricas existem por grupo, então adicionamos um *path* por grupo. Ou seja, mesmo que um dado grupo possua vários objectos a ser desenhados, dentro dele, apenas haverá uma única trajectória.

Finalmente, para não ignorar a matriz *view*, o que fazemos, quando pretendemos desenhar objetos, no ecrã é uma pré-multiplicação da matriz *model* pela *view*, obtendo a *modelview*, que permitirá o correto desenho e funcionamento de todas as componentes envolvidas. Como se pode verificar de seguida.

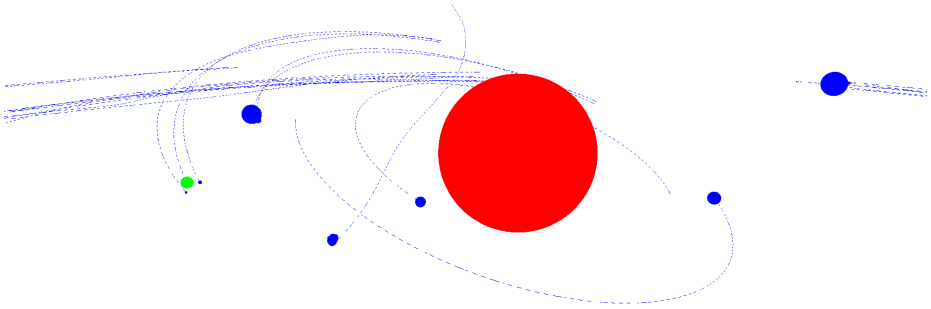

Figura 1. Demonstração tecnológica, com *path tracing*.

Ademais, atualmente a capacidade de *tracing* está sempre ativa. Porém, na fase seguinte pretendemos adicionar a capacidade de ativar e desativar esta componente.

8 Seleção de foco através do rato

Como componente adicional, fizemos a seleção de objetos presentes no sistema através do rato aos quais queremos que a câmara foque e siga o movimento dos mesmos.

Para realizar esta tarefa utilizamos uma ferramenta presente no OpenGL, o *Stencil Buffer*, e com este *buffer*, que vai sendo preenchido ao mesmo tempo em que estamos a desenhar os objetos, guardamos o indice do objeto que está a ser desenhado.

Através da captura do pixel escolhido com o rato é possível obter a informação sobre qual o índice do objeto desenhado e presente naquela posição do ecrã e com esse mesmo índice podemos obter várias informações sobre o objeto uma vez que o índice que é obtido através do *Stencil Buffer* é o índice-1 dos grupos onde estão armazenados os modelos.

Depois de sabermos qual foi o objeto selecionado basta calcular as coordenadas do centro do objeto e colocar a câmara sempre paralela ao centro e a olhar para o mesmo. Para calcular o centro do objeto utilizamos as transformações que sabemos que este realiza e aplicamos essas transformações a uma matriz identidade que é posteriormente multiplicada por o ponto *(0,0,0,1)* para sabermos onde ao certo está o centro do objeto.

Foi necessário mudar um pouco a estrutura de cada grupo para que fosse possível saber quais as transformações que cada tem de realizar tendo em conta

que há a possibilidade de o grupo que estamos a considerar ser um sub-grupo de outro que tenha também as suas próprias transformações (como os satélites naturais) e nesse caso foi necessário que cada um soubesse quais os índices dos seus antecessores para que as coordenadas estivessem realmente certas.

Para tornar mais interessante e dinâmico este modo de visualização permitimos a deslocação vertical nesta situação.

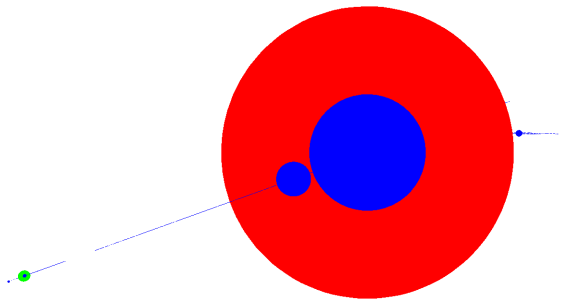

Figura 2. Exemplo de foco da câmara na Terra

9 Demonstração Tecnológica

De forma a demonstrar as capacidades tecnológicas do nosso engine, foram feitas extensões à prévia demonstração tecnológica. Nomeadamente, apostou-se em tornar o sistema solar no mais dinâmico possível, mantendo sempre os requisitos indicados.

Desta forma, para cada planeta, e para cada uma das respectivas luas, foi adicionada a capacidade de rotação dos planetas sobre o seu próprio eixo, operação esta que decorre num tempo característico de cada corpo.

Ademais, foram adicionados mecanismos de translação dos satélites naturais em torno dos seus planetas. Para além disso, foi também inserida a translação dos planetas em torno do sol, o que permite a criação de um sistema semelhante à realidade.

Relativamente ao cometa, decidimos inserir como cometa um *teapot* com algumas alterações, inserindo pontos adequados à formação da respectiva trajectória deste cometa.

Permitindo assim obter como resultado final a seguinte demonstração:

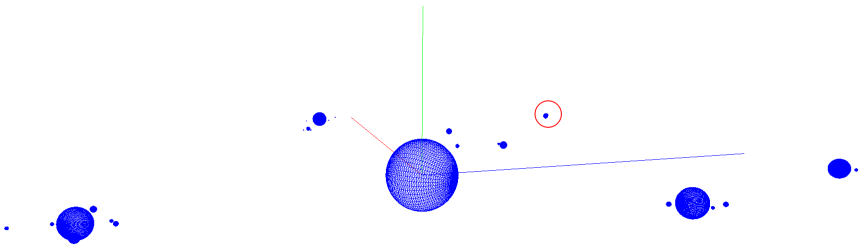

Figura 3. Demonstração com cometa a vermelho, sem *path tracing*.

10 Conclusões e Trabalho Futuro

Através da conciliação de diferentes conceitos e respectiva adaptação ao contexto deste sistema, conseguimos estender um sistema previamente estático a um sistema dinâmico com capacidade interpolação entre pontos. Permitindo também *inputs* mais abrangentes, tendo em conta as funcionalidades e figuras adicionais do XML e `generator`. Para além disso, conseguimos com sucesso dotar o nosso sistema da capacidade de focar num objecto através do rato, permitindo assim obter um ponto de vista em primeira pessoa, por parte do objeto seleccionado.

Em suma, conseguimos com sucesso implementar os requisitos pretendidos para a 3ª fase, incluindo como extra diversas capacidades que achamos interessantes. Tudo isto com relativa facilidade, tendo em conta toda a modularidade imposta por versões anteriores.