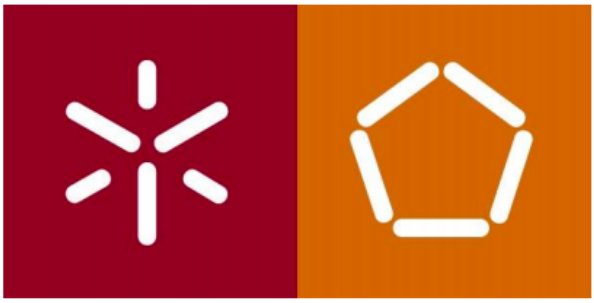

Universidade do Minho

Mestrado Integrado em Engenharia Informática

04/05/2020

# Computação Gráfica
# Fase 3

António Gonçalves (A85516)
João Fernandes (A84034)
Eduardo Conceição (A83870)
Rita Rosendo (A84475)

# Contents

# 1 Introdução

Este relatório irá descrever a terceira fase do projeto da cadeira de Computação Gráfica. Nesta fase do trabalho, foi-nos proposto que, a partir das fases anteriores, desenvolvessemos novas funcionalidades aplicadas ao Generator e Engine de modo a desenhar um sistema solar dinâmico, incluindo um cometa com uma trajetória definida usando uma curva Catmull-Rom.

# 2 Análise do Problema

Nesta fase do projeto, o Generator deve ser capaz de criar um novo tipo de modelo baseado em Patches de Bezier. Este modelo deverá receber as várias *patches* e pontos de controlo de um ficheiro *input* que deverá ser passado como argumento à *main*. Como para as outras figuras na primeira fase, as coordenadas da *patch* de Bezier deverão ser guardadas num ficheiro *output*, que também deverá ser passado à main. O último argumento que a *main* receberá será o nível de tesselagem.

Quanto à Engine, esta deverá ser alterada para que as coordenadas dos vértices dos vários modelos lidos sejam guardadas em *Vertex Buffer Objects* (VBOs), e deverá também acomodar a translação de figuras baseada em curvas de Catmull-Rom.

# 3 Resolução

## 3.1 *Generator*

No que toca ao Gerador, teve de ser adicionado à *main* suporte para mais um tipo de pedido. Este pedido trata-se de uma figura definida por *patches* de Bezier. Para obter esta figura, será necessário, primeiro, ler o ficheiro de *input*. Este ficheiro deverá ter o seguinte formato:

- Na primeira linha está o **número de *patches*** a ler;
- Nas seguintes linhas, estão as ***patches*** a ser lidas, cada uma contendo 16 inteiros separados por vírgulas;
- Na próxima linha, vem o **número de pontos de controlo** que estão no ficheiro. Este número está presente no ficheiro, mas será apenas descartado na leitura, uma vez que não apresenta nenhum propósito prático.
- Nas restantes linhas estão os **pontos de controlo**. Cada linha tem 3 valores de vírgula flutuante, representando os valores x, y e z das coordenadas dos pontos de controlo.

Assim, o ficheiro com o formato descrito será lido pela função *readBezierPatchFile*, à qual apenas é passado o nome do ficheiro de input. Nesta função, os valores dos índices dos pontos de controlo serão guardados num *vector* de *floats*, que, por sua vez, será guardado numa variável global 'patchIndexes', por sua vez um *vector* com todas as *patches*. As coordenadas dos pontos de controlo serão guardadas num *array* unidimensional, também um *vector*, denominado 'bezierPoints', que, por sua vez, também é uma variável global. Uma vez que todos estes valores sejam lidos, a função termina retornando o número de *patches* lido.

De seguida, será invocada a função *drawBezierPatch*, que é composta, principalmente, por um ciclo *for* onde lemos cada uma das *patches* guardadas e, em cada uma, começamos por preencher 3 matrizes 4x4, chamadas 'x_matriz', 'y_matriz' e 'z_matriz'. Nestas matrizes, iremos guardar os valores de todas as coordenadas que compoem uma *patch*.

De seguida, percorremos, para cada *patch*, dois ciclos *for*, um para as variávies 'u' e 'un' e o outro para 'v' e 'vn'. Nestes ciclos, começamos em 0 e incrementamos por 1 as variáveis até estas serem iguais ao valor da tecelagem, e, em cada iteração, tomam-se os valores:

```
u = (float) ul * step
un = (float) (ul + 1) * step

v = (float) vl * step
vn = (float) (vl + 1) * step
```

Em que 'step' corresponde a

```
step = 1.0f/(float)tesselation
```

Em cada um dos ciclos, vamos calcular os valores que os novos pontos vão assumir com base em 'u', 'v', 'un', 'vn' e as 3 matrizes que mencionamos anteriormente. Assim, teremos de calcular 4 pontos em cada iteração, de forma a formar 2 triângulos: A, B, C e D.

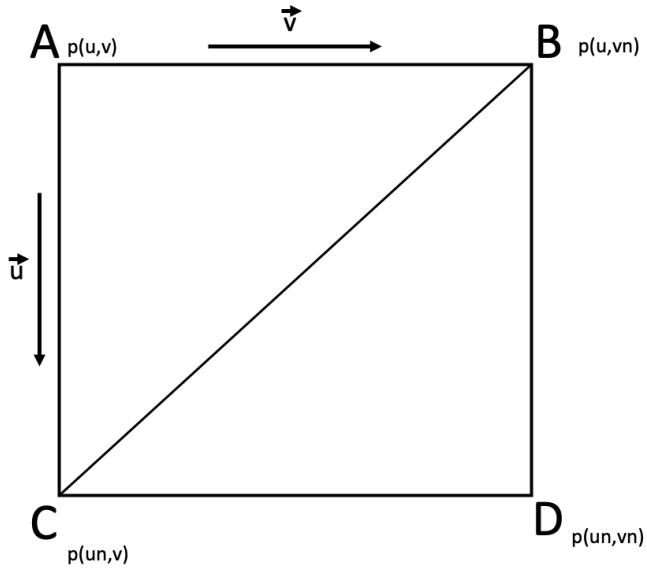


Figure 1: Pontos calculados em cada iteração, e os respetivos triângulos

Para calcular estes valores, teremos de usar a seguinte fórmula:

$x = [u^3u^2u1] * M * x\_matriz * M^T * [v^3v^2v1]^T$
$y = [u^3u^2u1] * M * y\_matriz * M^T * [v^3v^2v1]^T$
$z = [u^3u^2u1] * M * z\_matriz * M^T * [v^3v^2v1]^T$

Em que a matriz M corresponde a:

$$\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

E $M^T$ corresponde à sua transposta que, como podemos ver ao analizar a matriz M, é igual a M.

A partir destes cálculos, obtemos os pontos que escrevemos no ficheiro *output*, sendo que primeiro é escrito o triângulo ACB, e depois o triângulo BCD.

Tendo tudo isto em consideração, obtemos toda a lógica relacionada com a escrita de um objeto descrito por *patches* de Bezier e, por consequência, as mudanças principais do *Generator*.

## 3.2 *Engine*

Para esta fase do trabalho será necessário alterar bastantes funcionalidades da *engine*, em relação à anterior. Para melhor representar o Sistema Solar, iremos movimentar os planetas em torno do Sol, bem como simular um movimento de rotação dos mesmos.

Começamos por adicionar à estrutura *GeometricTransf* algumas informações, bem como à estrutura *GeoTPoint*. Para além disso criamos também uma outra estrutura, *Translacao*, que será usada para o movimento de Translação dos planetas.

```
struct GeoTPoint{
    float angle;
    float x;
    float y;
    float z;
    float t;
};
```

Figure 2: Nova estrutura *GeoTPoint*

```
struct GeometricTransf{
    GeoTPoint translate;
    GeoTPoint rotate;
    GeoTPoint scale;
    GeoTPoint color;
    GeoTPoint rotacao;
    Translacao trans;

};
```

Figure 3: Nova estrutura *GeometricTrans*

O principal objetivo destas mudanças será normalizar o máximo possível a leitura das novas informações do ficheiro XML, uma vez que a maioria das ações necessárias para esta fase podem ser guardadas num *GeoTPoint*, mesmo que esta não necessite de todas as suas variáveis.

A estrutura *Translacao* será explicada mais à frente, uma vez que esta será utilizada para guardar a informação necessária para que ocorra o movimento de translação de cada planeta e de cada satélite.

Para além disso, visto que a estrutura Point deixa de ser necessária para guardar a informação dos pontos para desenhas, isto uma vez que iremos utilizar VBOs para tal, alteramos tambem esta estrutura.

```
struct Point {
    float coords[3];
};
```

Figure 4: Nova estrutura *Point*

À principal função de *parsing* foram adicionadas todas as informações necessárias, para que, aquando da leitura do ficheiro, os dados referentes às cores, *rotations*, *translations*, *scales* e rotações sobre si mesmas estejam adequadas.

### 3.2.1 Movimento de Translação

Nesta fase do projeto, o movimento de Translação dos planetas e dos satélites será feito a partir da utilização de curvas de *Catmull-Rom*.

Assim, começamos por criar uma estrutura chamada *Translacao*, que irá guardar a informação referente ao movimento de translação dos corpos.

```
struct Translacao{
    float u[3];
    float t;
    vector<Point> points;
    int c;
};
```

Figure 5: Nova estrutura *Point*

Mal se encontre uma *tag translacao* no ficheiro XML, o *parser* irá começar a recolher a informação. Logo a seguir à *tag* aparece o tempo, que afeta a velocidade do movimento, seguindo-se dos vários pontos que formam a orbita do planeta do grupo. O vetor U será usado para mais tarde realizar as contas associadas à curva de *Catmull-Rom*. O *parsing* dos pontos irá continuar até não encontrar mais pontos. No nosso caso definimos 8 pontos para desenhar a órbita dos Planetas, estando estes sempre à mesma distância do Sol.

```
<translacao tempo="50">
 <ponto pX="15" pY="0" pZ="0"/>
 <ponto pX="10.6" pY="0" pZ="10.6"/>
 <ponto pX="0" pY="0" pZ="15"/>
 <ponto pX="-10.6" pY="0" pZ="10.6"/>
 <ponto pX="-15" pY="0" pZ="0"/>
 <ponto pX="-10.6" pY="0" pZ="-10.6"/>
 <ponto pX="0" pY="0" pZ="-15"/>
 <ponto pX="10.6" pY="0" pZ="-10.6"/>
</translacao>
```

Figure 6: Exemplo da informação referente a uma Translação

Para desenhar a órbita e mover o corpo ao longo desta utilizamos os cálculos e as funções do Guião 8, das aulas práticas, uma vez que a maior parte das funções seriam as mesmas, sendo a única diferença será no final da função *getGlobalCatmullRomPoint* em vez de irmos buscar as coordenadas a um *array*, estas encontram-se num vetor com *Points*.

Sendo assim, utilizamos a função *translacaoAni* para controlar o movimento e reposicionar o corpo ao longo da sua órbita. Contudo, ao longo da realização do tra-

balho foram aparecendo vários problemas em volta desta parte, sendo que não a conseguimos concluir. Conseguimos ver, a partir de vários *prints* ao longo do código, que a informação está a ser retirada do ficheiro XML, mas as estruturas onde guardamos estes dados eram eliminadas, problema que não conseguimos resolver atempadamente.

### 3.2.2 Movimento de Rotação

Para realizar o movimento de rotação começamos por desenhar o eixo de cada planeta, apenas para facilitar a observação do movimento.

Posto isto, como já foi dito anteriormente, adicionámos um novo *GeoTPoint* à estrutura *GeometricTransf* para representar a informação da rotação, sendo esta o *t*, que representa o tempo que demora a executar uma volta, e os valores de *x y* e *z*, que representam o eixo pelo qual se está a realizar a rotação.

```
<rotacao tempo="62.9" eX="0" eY="1" eZ="0" />
```

Figure 7: Exemplo de *rotacao* no ficheiro XML

Aquando da leitura do ficheiro, caso o *parser* encontre um *tag rotacao*, este irá comceçar a extrair a informação, que será guardada nas transformações do *Grupo*.

Para a animação criamos a função *rotacaoAni*, que a partir do *GeoTPoint* com a informação retirada do XML irá girar a figura. Este movimento é feito a partir do tempo passado a partir do *GLUT_ELAPSED_TIME*.

```
void rotacaoAni(GeoTPoint a){
    float tempoP = glutGet(GLUT_ELAPSED_TIME);
    float ang = 360/(a.t*100);//Define a rapidez da rotacao

    glRotatef( angle: tempoP*ang,a.x,a.y,a.z);
}
```

Figure 8: Função que executa o movimento de Rotação dos planetas

A rotação dos planetas faz com que os seus satélites aparentem ter um movimento de translação, mas uma vez que não conseguimos acabar essa parte, isto acontece porque ao rodar o Grupo onde o planeta está, também os seus sub-grupos irão rodar.

### 3.2.3 *Vertex Buffer Object*

Para alterar a *Engine*, foi necessário primeiro alterar a estrutura *Group*. Esta estrutura guardava um *vector* de estruturas *Point*, mas, de agora em diante, este *vector* passa a ser um *array* de GLuint de tamanho 1, e também será guardado o número de vértices que serão guardados neste *array*.

Começemos pelas mudanças na leitura de um ficheiro. Agora, quando é chamada a função *loadModels*, a esta será passada um objeto *Group*, que depois será devolvido. Primeiro, lemos o ficheiro e guardamos todos os valores x, y e z de cada ponto no ficheiro *input*, e guardamos num *vector* de *floats*. De seguida, geramos o *buffer* de

pontos com o atributo 'points' da estrutura *Group*, com a função *glGenBuffers*, fazemos *bind* do mesmco com *glBindBuffer*. A seguir, atribuimos a esse *buffer* a informação guardada no *vector* de *floats* com *glBufferData*. Guardamos também o número de vértices lidos na estrutura *Group*.

Depois, na função *draw*, que nos permite desenhar o conteúdo do *buffer*, utilizamos a função *glBindBuffer* para fazer *bind* do *buffer* do Grupo em que estamos, *glVertex-Pointer* para establecer o tamanho de cada ponto e o tipo de valores que vão ser lidos, e, por fim, a função *glDrawArrays* para apresentar o conteúdo do *buffer* que foi mais recentemente *bounded*.

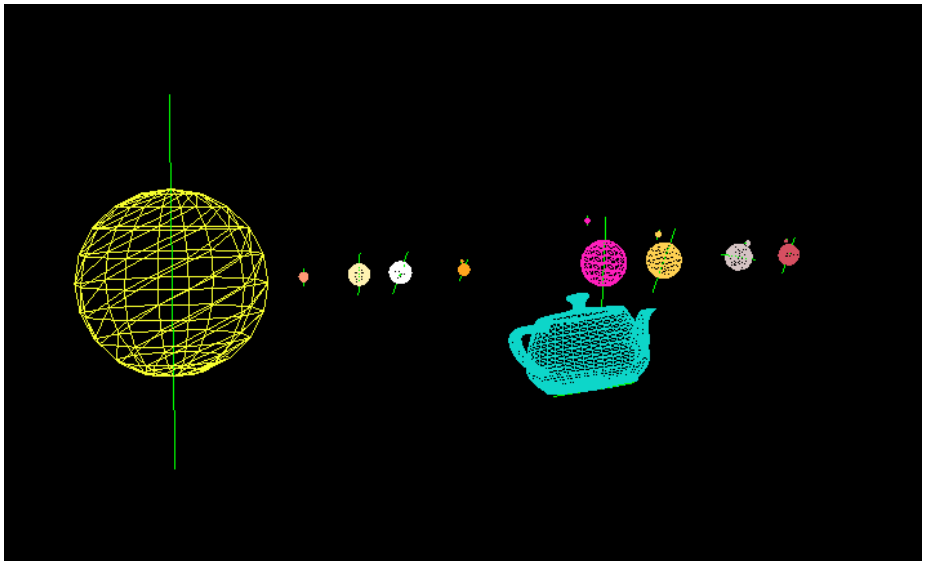

Figure 9: Representação final

# 4 Conclusão e Trabalho Futuro

Em primeiro lugar, conseguimos implementar corretamente a nova forma geométrica pedida para o *Generator*, bem como a utilização de VBOs para guardar as coordenadas dos pontos na *Engine*, como nos foi pedido. No entanto, uma vez que tivemos problemas na implementação dos movimentos de translação, acabamos por não fazer tudo aquilo que nos era pedido para esta entrega.

Apesar de termos conseguido gerir bem a informação presente no ficheiro XML e de acharmos a forma que implementamos organizada, não conseguimos completar as funcionalidades que nos foram pedidas. Assim sendo, teremos como trabalho acrescido na próxima fase completar as tarefas que não conseguimos completar nesta , ou seja, implementar a translação corretamente. Mais ainda, aproveitaremos a próxima fase para implementar uma câmara melhor, em primeira pessoa, e uma representação da cintura de asteroides entre Marte e Júpiter, para além dos objetivos pedidos.

Com isto, e em jeito de resumo, o trabalho que realizamos nesta fase não alcançou todos os objetivos que se supunham, mas conseguimos mesmo assim implementar a maior parte daquilo que nos foi pedido, sendo que teremos de compensar o que não fizemos nesta fase na próxima.