Universidade do Minho
Escola de Engenharia

Computação Gráfica

TRABALHO PRÁTICO

3ª Fase

Mestrado Integrado em Engenharia Informática

Fábio Quintas Gonçalves, a78793

Francisco José Moreira Oliveira, a78416

Raul Vilas Boas, a79617

Vitor Emanuel Carvalho Peixoto, a79175

Ano letivo 2017/2018

Braga, Abril de 2018

ÍNDICE

1. Introdução 1
2. Generator 2
   2.1 Gerar pontos usando patches de Bezier 2
   2.2 Alterar modo de gerar pontos para VBOs 3
3. Engine 4
   3.1 Desenhar figuras com VBOs 4
   3.2 Estruturas de dados 4
   3.3 Curvas de Catmull-Rom 5
      3.3.1 Órbitas 5
      3.3.2 Translação 6
      3.3.3 Rotação 7
4. Ficheiro XML 8
5. Câmara de visualização 10
6. Resultado final 11
7. Conclusões e trabalho futuro 12
8. Bibliografia 13

1. INTRODUÇÃO

Após ter sido desenvolvido nas etapas anteriores um sistema solar estático, onde se desenvolveu um *Generator* capaz de gerar pontos de figuras, um *Engine* capaz de ler os pontos e desenhá-los e capaz também de ler transformações efetuadas às figuras, seguindo uma ordem hierárquica, nesta 3ª fase, a dificuldade aumentou alguns níveis, pedindo-nos os seguintes pontos:

- O *Generator* deve ser capaz de gerar pontos dos triângulos de qualquer objeto, fornecendo apenas um ficheiro com pontos de controlo e índices. Os pontos serão gerados usando um tipo de *splines* matemáticas, conhecidas por superfícies de Bezier.
- O *Engine* deverá ser capaz também de desenhar objetos usando *buffers*. Para tal, é necessário também alterar no *Generator* a ordem pela qual os pontos dos objetos são gerados.
- O sistema solar deverá ser dinâmico. Para tal, devemos expandir as funcionalidades da Rotação e da Translação. As movimentações serão definidas usando curvas de Catmull-Rom e um conjunto de pontos de controlo definidos.

2. GENERATOR

2.1 Gerar pontos usando patches de Bezier

Primeiro há que analisar a constituição de um ficheiro ".patch":

- A primeira linha desse ficheiro indica o número 'n' de *patches*;
- As seguintes 'n' linhas indicam os 16 índices dos patches;
- A seguir a essas 'n' linhas, temos o número 'm' de pontos de controlo;
- E de seguida 'm' linhas com as coordenadas dos pontos de controlo.

A função ***patchToArrays*** que recebe o ficheiro ".patch" é a responsável por ler esse ficheiro e armazenar os **índices de cada *patch*** e os **pontos de controlo** nos seus *arrays* respetivos.
Essa função é chamada pela ***bezier***, responsável por gerar os pontos. Esta função simplesmente itera pelo número de *patches* e pela *tesselation* bidimensionalmente e calcula os pontos. Os pontos são calculados pela função ***pontoBezier***, recorrendo ao Polinómio de Bernstein:

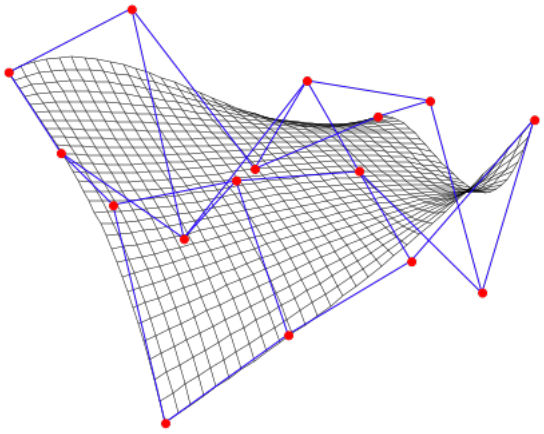

Figura 1 Superfície tridimensional interpolada, usando os pontos de controlo (a vermelho), recorrendo ao polinómio de Bernstein.

Inicialmente criamos os polinómios para ambas as dimensões 'u' e 'v' e iteramos dentro destes usando dois ciclos *for* aninhados. Dentro dos ciclos, calculamos os índices do *patch* e ponto de controlo e criamos um ponto que irá ser calculado através da multiplicação entre cada uma das coordenadas dos pontos de controlo pelos polinomiais de Bernstein de 'u' e 'v'. Esse ponto gerado é por fim retornado para a função ***bezier*** que, após ter calculado os pontos, escreve-os num ficheiro ".3d" e ainda liberta a memória alocada pelos *arrays* dos índices dos *patches* e dos pontos de controlo.

Na ***main***, o argumento a ser usado para chamar a função é "bezier" e pede também como argumentos o ficheiro ".patch", o nome que irá ter o futuro ficheiro ".3d" e o nível de *tesselation* que irá definir a precisão do objeto.

2.2 Alterar modo de gerar pontos para VBOs

De modo ao *Engine* conseguir gerar corretamente os objetos pretendidos, é necessário alterar as funções do *Generator* que geram os pontos dos triângulos, para a ordem que os VBOs pedem os pontos.

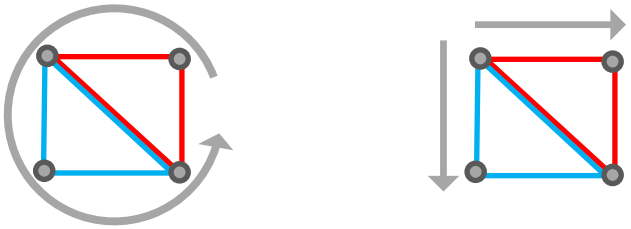

Figura 2 Diferença na ordem de leitura das coordenadas, relativamente ao modo imediato (esquerda) e com VBOs (direita).

Tomemos como exemplo os pontos do plano. Neste caso, iremos ter duas variáveis: largura e comprimento, entre as quais iremos iterar em dois ciclos for aninhados, como se pode ver abaixo:

Excerto do *plane* para VBOs.

```
1   for (float i = -(largura/2); i <= (largura/2); i++) {
2       for (float j = -(comprimento/2); j <= (comprimento/2); j++) {
3           pontos.push_back(Ponto(j, 0.0, i));
4           pontos.push_back(Ponto(j, 0.0, i+1));
5       }
6   }
```

Esta função gera assim as coordenadas dos pontos que geram um plano, recorrendo a VBOs.

Os objetos que foram adaptados para serem desenhados com VBOs foram: plano; esfera; anel e circunferência. De facto, os objetos necessários para desenhar o nosso sistema solar estão nesta lista, o que significa que todo o nosso sistema solar é desenhado recorrendo a VBOs, excetuando o cometa, que irá ser a *teapot* gerada com os *patches* de Bezier e usa triângulos.

3. ENGINE

3.1 Desenhar figuras com VBOs

Para preparar o *Engine* para ser capaz de produzir objetos através de VBOs usamos o código produzido na aula prática onde abordamos esta temática. Antes de começar a explicar o processo, consideramos que é de extrema importância explicar como é que o Engine reconhece se o ficheiro ".3d" tem de ser desenhado recorrendo a VBOs ou a GL_TRIANGLES. No *Generator*, definimos que os objetos a serem gerados com recurso a VBOs têm na primeira linha o número de pontos, que mais tarde será necessário. Assim, ao ler o ficheiro no *Engine*, a função ***getFigure*** foi construída para detetar se a primeira linha contém apenas um número (então a figura é para desenhar com VBOs) ou se o ficheiro se inicia de imediato com as coordenadas dos pontos (então é para desenhar com triângulos).

A função que irá desenhar recorrendo a *buffers* será a ***drawFiguresVBOs***. Essa função recebe a lista de figuras a desenhar e o número de pontos.

Dentro dessa função, o processo para desenhar uma figura é o seguinte:

Processo de preenchimento do *buffer* e execução do mesmo.

```
1   vector<Coordinate>::iterator it_coords;
2   int it = 0;
3   for (it_coords = it_fig->begin(); it_coords != it_fig->end(); it_coords++) {
4       array[it++] = it_coords->x;
5       array[it++] = it_coords->y;
6       array[it++] = it_coords->z;
7   }
8   glBufferData(GL_ARRAY_BUFFER, it*sizeof(float), array, GL_STATIC_DRAW);
9   glVertexPointer(3,GL_FLOAT,0,0);
10  glDrawArrays(GL_TRIANGLE_STRIP, 0, nPoints/3);
```

Após inicializarmos e criarmos o *array* que será o nosso *buffer*, iteramos pelas coordenadas e preenchemos o *array*. Por fim, desenha-se a figura, usando esse *array* preenchido.

3.2 Estruturas de dados

Dada a necessidade de um sistema solar dinâmico, recorrendo a pontos de controlo e da implementação de VBOs, é preciso alterar algumas das estruturas de dados existentes:

- A classe ***Rotate*** é uma classe nova. Dada a necessidade de armazenar o *time* (nº de segundos que leva uma rotação completa sobre si próprio) e distinguir este tipo de todos os outros.

Classe Rotate.

```
1   class Rotate {
2   public:
3       bool empty = true;
4       float x, y, z, angle=-1, time=-1;
5   };
```

- A classe ***Translate*** perde os pontos de translação X, Y e Z para armazenar todos os pontos da rota de translação no array de 2 dimensões *catmulls*. Inicialmente os pontos de controlo são lidos e armazenados no vetor *catmullPoints*, mas uma vez que o *render* de Catmull-Rom necessita de receber os pontos através de um array bidimensional, efetuamos a conversão numa função chamada ***vectorToArray***. É armazenado também o *time*, que representa o nº de segundos que demora a efetuar uma translação completa sobre o Sol.

Classe Translate.

```
1   class Translate {
2   public:
3       bool empty = true;
4       float time;
5       vector<Coordinate> catmullPoints;
6       float catmulls[50][3];
7   };
```

Todas as outras estruturas de dados permaneceram inalteradas.

3.3 **Curvas de Catmull-Rom**

3.3.1 Órbitas

As órbitas dos planetas e luas podem ser desenhadas recorrendo à interpolação de pontos de controlo. Isto permite-nos desde logo remover algumas funções e variáveis que tínhamos unicamente para desenhar órbitas. Para desenhar qualquer objeto recorrendo à leitura dos pontos de controlo pelas curvas de Catmull-Rom, é necessário chamar a função ***renderCatmullRomCurve*** das nossas aulas práticas.

No nosso caso, uma vez que apenas nos interessa desenhar as órbitas, recebemos o *Translate* de uma figura como argumento dessa função.

Esta função itera entre valores de 0 a 1, sendo que quantas mais iterações fizer, mais detalhado irá ser o desenho gerado, e para cada iteração, chama a função ***getGlobalCatmullRomPoint***. A essa função é passado como argumentos o índice da iteração dois arrays vazios (pos[3] e deriv[3]) e a translação da figura. O array *pos*, no fim da execução da função, irá estar povoado com os pontos a desenhar, que irão ser desenhados, através da função do GLUT: *glVertex3f*. A função ***getGlobalCatmullRomPoint*** já nos foi dada por inteiro nas aulas práticas. Sendo apenas necessário completar a função ***getCatmullRomPoint*** que é invocada por esta função. Nessa função são efetuados os cálculos necessários para preencher os *arrays pos* e *deriv*.

$$P(t) = [t^3 \quad t^2 \quad t \quad 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$P'(t) = [3t^2 \quad 2t \quad 1 \quad 0] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Figura 3 Processo de cálculo do *array pos* (P(t)) do *array deriv* (P'(t)).

3.3.2 Translação

A translação assenta nas mesmas funções usadas para desenhar a rota. Apenas neste caso, temos de calcular os pontos da translação e iterar sobre cada um deles ao longo do tempo.

A variável gt é a responsável por guardar a posição, tendo em conta o tempo decorrido desde o início da execução do programa (GLUT_ELAPSED_TIME) e a variável *time* de cada translação, que indica em quanto tempo uma figura deve efetuar uma translação completa.

Função *renderCatmullTranslate.*

```
1  float pos[3], deriv[3];
2  clocks = glutGet(GLUT_ELAPSED_TIME);
3  gt = fmod(clocks, (float) (trans.time * 1000) ) / (trans.time * 1000);
4  getGlobalCatmullRomPoint(gt, pos, deriv, trans);
5  glTranslatef(pos[0], pos[1], pos[2]);
```

3.3.3 Rotação

A rotação é igual à translação, mudando apenas numa condição: temos de distinguir as duas rotações possíveis num objeto. Um objeto pode ser sujeito a uma rotação no sentido da inclinação do seu eixo e uma rotação no sentido de movimentação sobre o seu próprio eixo. A classe da rotação junta as duas variáveis *angle* e *time*, no entanto, quando uma assume um valor a outra apresenta valor igual a -1. Assim, conseguimos fazer uma rotação de cada vez. A rotação no sentido de movimentação sobre o seu próprio eixo é efetuada através do tempo decorrido após a execução do programa, tal como a translação.

Função *renderRotate.*

```
1    if(rot.angle != -1)
2        glRotatef(rot.angle, rot.x, rot.y, rot.z);
3    else if(rot.time != -1){
4        clocks = glutGet(GLUT_ELAPSED_TIME);
5        float angle = 360 * (fmod(clocks, (float) (rot.time * 1000) ) / (rot.time * 1000));
6        glRotatef(angle, rot.x, rot.y, rot.z);
7    }
```

4. FICHEIRO XML

O ficheiro XML sofreu diversas mudanças, para ser capaz de respeitar os requisitos para esta etapa.

Para começar, adicionamos um cometa, tal como pedido. Optamos por adicionar o cometa Halley, visto ser provavelmente o cometa mais famoso do sistema solar. Este cometa utiliza como modelo a *teapot* gerada através dos *patches* de Bezier.

Cometa Halley

```
<group a = "Cometa">
    <translate time ="500">
        <point X="-3.0" Y="1.0" Z="0.0" />
        <point X="-1.34" Y="0.567" Z= "-2.60" />
        ...
        <point X="-1.34" Y="0.567" Z= "2.60" />
    </translate>
    <color R="0.7" G="0.7" B="0.7" />
    <scale X="0.02" Y="0.02" Z="0.02" />
    <models>
        <model file="teapot.3d" />
    </models>
</group>
```

De facto, no excerto acima, consegue-se observar que a *tag Translate* perdeu as suas coordenadas X, Y e Z, mas ganhou um conjunto de pontos de controlo que definem a sua órbita e translação através das curvas de Catmull-Rom e ainda uma variável *time* que indica o nº de segundos que demora a fazer uma translação à volta do Sol. Todos os pontos de controlo foram gerados manualmente, dividindo círculos ou ovais em diversas fatias e anotando os vértices ao longo dos círculos.

Para além do *Translate* adicionamos uma nova *tag Rotate*, dentro de um grupo novo:

Tags *Rotate*

```
<group a = "Terra">
    <translate time="36.5">
        <point X="4.50" Y="0.0" Z="0.00"  />
        <point X="4.32" Y="0.0" Z="-1.15" />
        ...
        <point X="3.88" Y="0.0" Z="2.23"  />
        <point X="4.32" Y="0.0" Z="1.15"  />
    </translate>
    <rotate angle="23.5" axisX="0" axisY="0" axisZ="1" />   <!— Inclinação do eixo –>
    <group>
        <rotate time="5" axisX="0" axisY="1" axisZ="0" />   <!—Rotação sobre o seu eixo –>
        <scale X="0.58" Y="0.58" Z="0.58" />
        <color R="0" G="0.45" B="1" />
        <models>
            <model file="planeta.3d" />
        </models>
    </group>
</group>
```

De facto, a primeira *tag* refere-se à inclinação e a segunda refere-se à rotação sobre o seu eixo, sendo que *time* define o nº de segundos que demora uma rotação. Estas têm de estar separadas nos grupos, de modo a não haver junção, pois iria juntar as duas rotações e produzir um movimento não pretendido. Quando encontrar uma *tag Rotate* com o *time* definido, regista todos os valores lá descritos e regista o *angle* como igual a -1 e vice-versa para o outro caso.

Também no XML foi removido tudo o que fazia referência ao desenho das órbitas, que já não vai ser mais necessário, uma vez que as órbitas são geradas através dos pontos de controlo das translações.

5. CÂMARA DE VISUALIZAÇÃO

A câmara não sofreu alterações relativamente à última etapa, sendo apenas adicionados abaixo a descrição das teclas ‘q’ e ‘e’.

As teclas registadas para os movimentos e alterações foram as seguintes:

‘w’ e ‘W’ para movimentar a câmara para a frente.
‘s’ e ‘S’ para movimentar a câmara para trás.
‘a’ e ‘A’ para movimentar a câmara para a esquerda.
‘d’ e ‘D’ para movimentar a câmara para a direita.
‘q’ e ‘Q’ para movimentar a câmara para baixo.
‘e’ e ‘E’ para movimentar a câmara para cima.

‘8’ para, mantendo o mesmo centro de visualização, mover a câmara para cima.
‘2’ para, mantendo o mesmo centro de visualização, mover a câmara para baixo.
‘4’ para, mantendo o mesmo centro de visualização, mover a câmara para a esquerda.
‘6’ para, mantendo o mesmo centro de visualização, mover a câmara para a direita.

‘LEFT_BUTTON’ e ‘KEY_UP’ para aproximar a câmara do centro.
‘RIGHT_BUTTON’ e ‘KEY_DOWN’ para afastar a câmara do centro.

‘i’ e ‘I’ para mudar o modo de visualização do modelo para *GL_FILL*.
‘o’ e ‘O’ para mudar o modo de visualização do modelo para *GL_LINE.*
‘p’ e ‘P’ para mudar o modo de visualização do modelo para *GL_POINT.*

Foram atribuídas pares de teclas de movimentação da câmara para acomodar a rapidez com que queremos percorrer o modelo, sendo que a segunda tecla atribuída nestes pares resulta numa maior alteração nas variáveis correspondentes.

6. RESULTADO FINAL

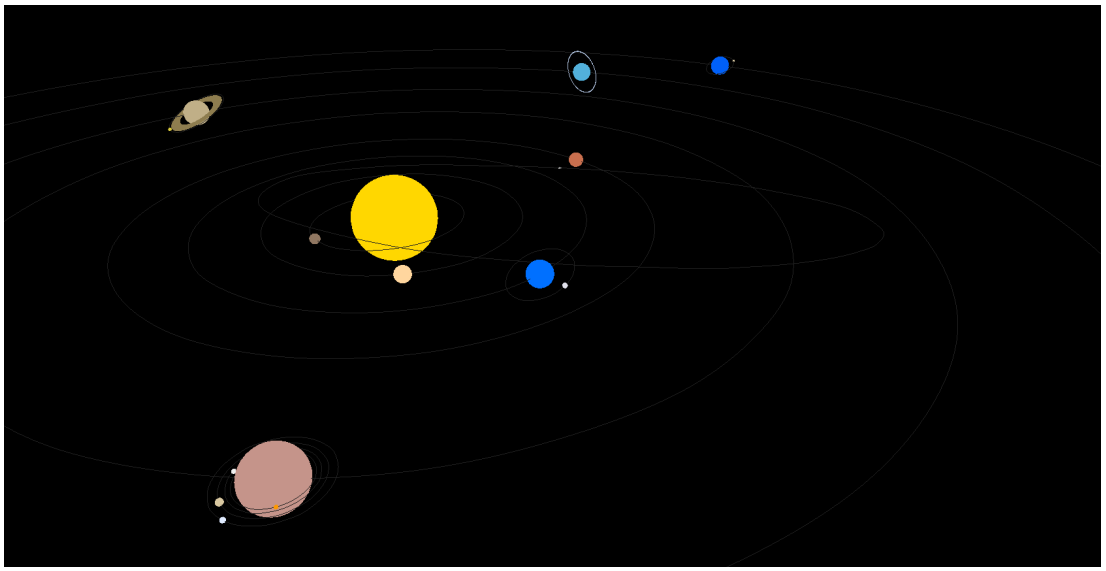

Figura 4 O Sistema Solar dinâmico.

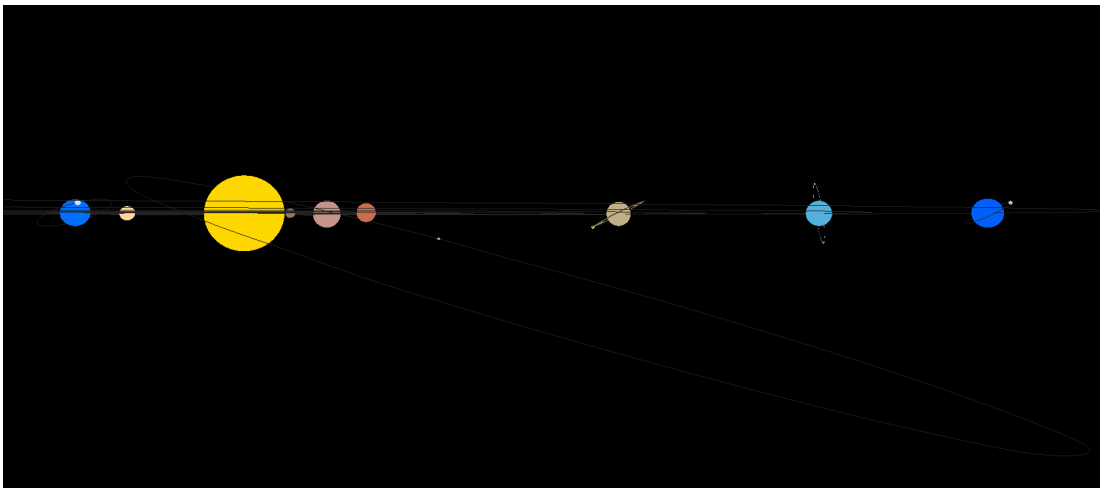

Figura 5 Órbitas dos planetas e órbita elíptica e não complanar do cometa Halley.

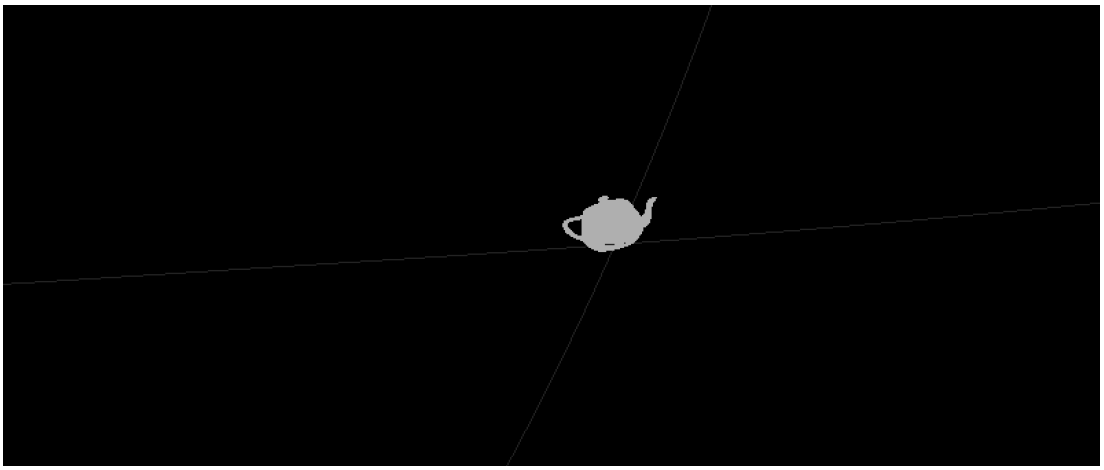

Figura 6 Cometa Halley, representado pela *teapot* gerada através de *patches* de Bezier.

7. CONCLUSÕES E TRABALHO FUTURO

Esta terceira fase permitiu absorver conhecimento relativamente aos processos de representação de figuras em computação gráfica. Este conhecimento foi adquirido através da matemática envolvente por detrás da renderização de imagem e o modo como o *OpenGL* processa a informação.

Permitiu-nos descobrir como gerar objetos, ou efetuar transformações sobre eles através de pontos de controlo e outros dados relativos. Neste caso, através de *patches* de Bezier e de curvas Catmull-Rom.

Para além disso, permitiu implementar um novo modo de gerar objetos, através de *buffers* e adquirir conhecimento com estes.

Indiretamente, a experiência com a linguagem C++ foi desenvolvida, permitindo abrir novos horizontes a uma nova linguagem, mas muito semelhante ao que temos vindo a usar ao longo do nosso percurso académico.

Futuramente, pretendemos implementar luminosidade, tendo como ponto de luz o Sol e também gerar os terrenos e texturas dos planetas. Alterações que queremos e devemos efetuar, capazes de tornar o nosso Sistema Solar cada vez mais real.

8. BIBLIOGRAFIA

Chris Lawson Bentley, *http://web.cs.wpi.edu/~matt/courses/cs563/talks/surface/bez_surf.html*, Worcester Polytechnic Institute, consultado a 20/04/2018.

Página Comunitária, *https://en.wikipedia.org/wiki/B%C3%A9zier_surface*, Wikipedia, a Enciclopédia Livre, consultado a 20/04/2018.