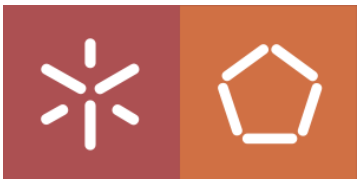


Universidade do Minho

Departamento de Informática

Computação Gráfica

Fase III – Curvas, Superfícies Cúbicas e VBOs

João Neves (a81366) Luís Manuel Pereira (a77667)
Rui Fernandes (a89138) Tiago Ribeiro (a76420)

Maio 2021

Resumo

O presente relatório descreve o trabalho prático realizado no âmbito da disciplina de *Computação Gráfica*, ao longo do segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Esta fase consiste na adição de novas funcionalidades e modificações ao trabalho já realizado, tais como o uso de curvas e superfícies de Bézier, curvas de Catmull-Rom e o uso de VBOs (*Vertex Buffer Objects*) para o desenho das primitivas gráficas. Adicionou-se também um cometa ao modelo do Sistema Solar, construído a partir de um *patch* de Bézier.

Neste documento descreve-se sucintamente a aplicação desenvolvida e discutimos as decisões tomadas durante a realização do trabalho prático.

Conteúdo

Lista de Figuras

1 Introdução

Os principais objetivos desta terceira fase do trabalho prático passam pelo desenvolvimento de modelos baseados em *patches* de Bézier, a extensão de transformações geométricas, nomeadamente translações e rotações, a partir de curvas de Catmull-Rom e, por fim, a renderização de modelos recorrendo a VBOs.

De maneira a incorporar todos os objetivos desta fase no projeto, foi necessária a alteração de vários elementos relativos ao trabalho desenvolvido durante a segunda fase, nomeadamente o *generator*, onde foram acrescentados métodos relativos à criação dos *patches* de Bézier, e o *engine*, onde foram elaborados métodos relativos às curvas de Catmull-Rom, bem como alterado o modo de renderização dos modelos, fazendo, agora, uso de VBOs.

Por fim, o modelo do Sistema Solar foi alterado para demonstrar estas capacidades, fazendo com que os planetas orbitassem em torno do Sol e com que as luas orbitassem em torno dos diferentes planetas, contando ainda com a adição de um cometa.

2 *Generator*

O *generator*, tal como na fases anteriores, é responsável por gerar ficheiros que contêm as coordenadas do conjunto de vértices das primitivas gráficas que se pretende gerar, conforme diversos parâmetros especificados. A única mudança que ocorreu nesta nova fase foi a inclusão de um processo de modo a poder gerar modelos baseados em *patches* de Bézier.

2.1 *Patches* de Bézier

O *generator* é agora capaz de ler *patch files* que definem superfícies de Bézier e transformá-los em ficheiros `.3d` que o `engine` está preparado para receber.

Importa salientar que formato dos ficheiros *patch* é simples, contendo duas secções – *patches* e pontos de controlo – e caracteriza-se da seguinte forma:

— A primeira linha contém o número de *patches*;

— As restantes linhas contêm os 16 índices de cada um dos pontos de controlo que constituem os *patches* da figura;

— Em seguida, segue-se uma linha que contém o número de pontos de controlo necessários para gerar a figura;

— Por fim, aparecem os pontos de controlo. Note-se que a ordem destes é importante, uma vez que cada ponto tem um índice associado que é utilizado.

Superfícies de Bézier

Por forma a definir uma superfíce de Bézier, utilizaram-se 16 pontos de controlo, representados numa grelha 4×4.

Desta forma, podemos considerar cada linha da grelha como uma curva Bézier, percorrendo horizontalmente a linha de modo a calcular os pontos correspondentes às curvas e, posteriormente, passando para a linha seguinte.

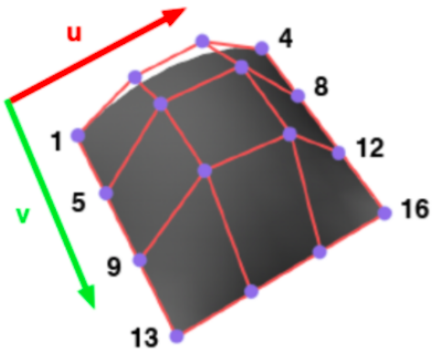


Figura 2.1: *Patch* de Bézier e respetivos pontos de controlo

Dependendo do nível de tesselação, serão calculados os vários pares (u, v) que posteriormente serão escritos para um ficheiro `3d`, de modo a gerar triângulos que representam

corretamente o modelo a desenhar. Assim, conclui-se que quanto maior o nível de tesselação, mais uniforme será o modelo gerado.

Deste modo, começou-se por definir as matrizes necessárias:

Matrizes a e b

$$a = \begin{bmatrix} a^3 & a^2 & a & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} b^3 & b^2 & b & 1 \end{bmatrix}$$

Matrizes com Pontos de Controlo

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

Matrizes de Bézier

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Tendo por base os valores de a e b, que se encontram contidos no intervalo $[0, 1]$, torna-se possível obter um ponto da superfície de Bézier, através da seguinte fórmula:

$$P(a, b) = a \times M \times P \times M \times b$$

Neste sentido, a função `getPoint`, que permite obter as coordenadas de um ponto, por aplicação desta expreesão. Para além disto, foi necessário elaborar a função `getPatchPoints` que, fazendo uso da função `getPoint`, é responsável por determinar as coordenadas dos vértices dos triângulos necessários à triangulação da figura.

Curvas de Bézier

O processamento de *patches* de Bézier é feito a partir do conceito de curvas de Bézier. Assim, a curva Bézier é definida em função de um parâmetro t, designado por Tesselação, que varia entre 0 e 1, ou seja para qualquer valor de t no intervalo $[0, 1]$, o valor da função nesse ponto representa um ponto da curva no espaço tridimensional. Para se calcular a curva completa, é necessário calcular os pontos da curva à medida que se incrementa o valor de t.

Para obter estas curvas, é necessário recorrer a quatro pontos de controlo – P_0, P_1, P_2 e P_3. Estes, utilizados numa função juntamente com um parâmetro t, tornam possível a criação de uma curva. De seguida, poderá observar-se a fórmula que define uma curva de Bézier, bem como um exemplo de uma curva.

$$P(t) = (1 - t^3)P_0 + 3t(1 - t^2)P_1 + 3t^2(1 - t)P_2 + t^3P_3$$

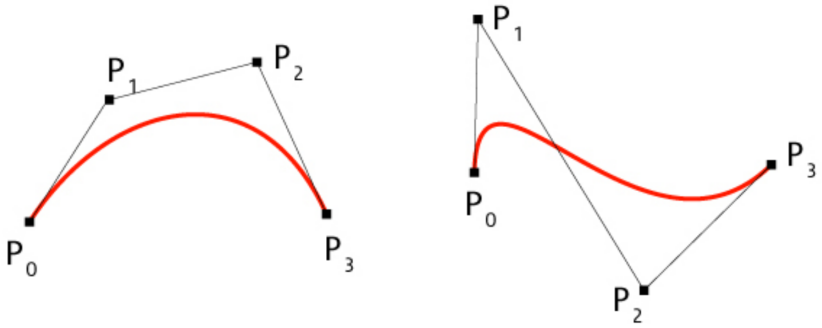


Figura 2.2: Curvas de Bézier e respetivos pontos de controlo

Tendo em vista a obtenção de uma curva mais pormenorizada, deverá recorrer-se ao uso de um maior número de pontos, por forma a definir mais corretamente a curva.

3 *Engine*

O objetivo do *engine* continua a ser o mesmo – permitir a apresentação de uma janela e exibição dos modelos requisitados. Além disso, permite, também, a interação com os mesmos a partir de certos comandos. Tal como na fase anterior do trabalho prático, existe um ficheiro XML que vai ser interpretado. No entanto, para esta fase do projeto, a arquitetura deste ficheiro evolui de maneira a cumprir os requisitos. Desta forma, foram feitas algumas alterações nos métodos de *parsing* e consequentemente alterações nas medidas de armazenamento e também de renderização.

3.1 Alterações no ficheiro XML

O ficheiro XML sofreu diversas mudanças, de forma ser capaz de respeitar os requisitos para esta fase do trabalho prático.

3.1.1 Translação

Para definir uma translação animada, cria-se um bloco *translate* com o parâmetro `time`, e dentro deste define se a lista de pontos que a translação ira seguir, tal como se pode ver de seguida..

```
<translate time="10">
   <point X="25.000000" Y="0.000000" Z="0.000000" />
   <point X="23.096989" Y="0.000000" Z="9.567086" />
   <point X="17.677670" Y="0.000000" Z="17.677670" />
</translate>
```

3.1.2 Rotação

Uma rotação animada troca apenas o ângulo por uma duração, que indica o tempo, em segundos, necessário para uma rotação de 360°, como se pode ver de seguida.

```
<rotate time="10" Y="1" />
```

3.2 VBOs

Os VBOs oferecem um aumento na *performance* em comparação com o modo de renderização imediata, devido ao facto de os dados serem armazenados na memória gráfica em vez de na memória do sistema, sendo diretamente renderizados pelo GPU.

Extraída a informação relativa a uma figura, neste caso as coordenadas dos seus vértices, os quais provêm de um ficheiro `3d`, esta é utilizada para inicializar e preencher o *buffer* relativo à figura em questão. Note-se que foi implementado um VBO por figura – *i.e.* por cada instância da classe `Shape`– pois estas são independentes umas das outras, na sua maioria.

Logo após a leitura do ficheiro com os pontos, estes ficam guardados num vetor. Este último é percorrido, inserindo os vértices num *array* que é utilizado no VBO da figura em questão.

3.2.1 Classe `Shape`

O OpenGL permite o uso de VBOs por forma a inserir toda a informação dos vértices diretamente na placa gráfica. Assim, começou-se por implementar os *arrays* que contêm os vértices das primitivas a desenhar, bem como uma variável que indica o número de vértices que o *buffer* contém.

```
class Shape {
private:
    unsigned int nVertices;
    GLuint buffer[1];
    // (...)
};
```

Aquando da instanciação de objetos desta classe, preenche-se o *buffer* com os vértices que lhe são passados como argumento, guardando, também, na memória da placa gráfica `nVertices * 3` vértices.

```
Shape::Shape(const vector<Vertex *> &vertices) {
    nVertices = vertices.size();
    float *vert = (float *)malloc(nVertices * 3 * sizeof(float));

    size_t i = 0;
    for (vector<Vertex *>::const_iterator v_it = vertices.begin();
         v_it != vertices.end(); ++v_it) {
        vert[i++] = (*v_it)->getX();
        vert[i++] = (*v_it)->getY();
        vert[i++] = (*v_it)->getZ();
    }

    glGenBuffers(1, buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * nVertices * 3, vert,
                 GL_STATIC_DRAW);

    free(vert);
}
```

Por fim, por forma a desenhar a informação guardada, foi desenvolvdo o seguinte método, que desenha todos os vértices guardados, tornando possível aumentar o desempenho, sendo visível um aumento do número de *frames* por segundo, em comparação aos obtidos quando os modelos são desenhados através de renderização imediata.

```
void Shape::drawShape() {
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, nVertices);
}
```

3.3 Curvas de Catmull-Rom

Um dos objetivos principais nesta fase consiste definir as órbitas dos planetas através da definição das curvas de Catmull-Rom.

3.3.1 Classe `Transform`

Relativamente à fase anterior do trabalho prático, foram adicionadas novas variáveis de instância à classe **`Transform`**, associadas a transformações novas que foram criadas. Para além disto, adicionaram-se funções e variáveis necessárias à implementação das curvas de Catmull-Rom.

```
class Transform {
private:
    std::string type;
    float angle, x, y, z, time;
    std::vector<Vertex *> controlPoints, pointsCurve;
    bool deriv;
    float vector[4] = {1, 0, 0};
    // (...)
};
```

De modo a representar as estas curvas, utilizou-se a *spline* de Catmull-Rom. Para a definição da curva serão necessários pelo menos quatro pontos.

Assim, considerando que o valor de t está contido no intervalo $[0, 1]$, as coordenadas de um ponto da curva e a sua derivada são dadas por:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$P'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Desta forma, será possível obter um modelo do Sistema Solar dinâmico. Como tal, implementou-se a função `getGlobalCatmullRomPoint` que permitirá a obtenção de coordenadas dos pontos e as suas derivadas. Esta função recorre à função auxiliar `getCatmullRomPoint`, que utiliza as matrizes e vetores referidos anteriormente, por forma a gerar os valores de retorno.

Com o intuito de crir as órbitas dos planetas, defiu-se a função `setCatmullPoints`, os pontos da curva a partir dos pontos lidos no ficheiro XML, recorrendo à função `getGlobalCatmull RomPoint`, uma vez que esta permite obter as coordenadas do próximo ponto da curva para um dado valor t. Deste modo, percorre-se um ciclo que começa em 0 e termina em 1, usando incrementos de 0.01, por forma a gerar 100 pontos da curva.

```
void Transform::setCatmullPoints() {
    float ponto[4];
    float der[4];

    for (float i = 0; i < 1; i += 0.01) {
        getGlobalCatmullRomPoint(i, ponto, der);
        Vertex *p = new Vertex(ponto[0], ponto[1], ponto[2]);
        pointsCurve.push_back(p);
    }
}
```

Tendo as coordenadas dos pontos da curva, a função `drawOrbits` é responsável pelo desenho da curva

```
void drawOrbits(Transform *t) {
    glColor3f(1.0f, 1.0f, 1.0f);
    glBegin(GL_LINE_LOOP);
    for (Vertex *p : t->getPointsCurve()) {
        glVertex3f(p->getX(), p->getY(), p->getZ());
    }
    glEnd();
}
```

3.4 Movimento dos Planetas

Recorrendo ao uso de transformações geométricas, tais como rotações e translações, tornou-se possível dotar os planetas de movimento relativamente a outra figura ou a eles próprios.

Assim, foram criada novas variáveis globais, por forma a capacitar o utilizador da capacidade de parar e retomar o movimento dos planetas, assim como alterar a velocidade do movimento.

Desta forma, são utilizados os diferentes tipos de rotações e translações descritos anteriormente. Esta movimentação tem por base o decorrer do tempo de vida da aplicação, sendo este o impulsionador do movimento de todas as figuras. Neste sentido, foram criadas as seguintes variáveis globais:

— `eTime` – Tempo decorrido desde que o movimento dos planetas está ativo;

— `cTime` – Tempo decorrido desde que o programa foi iniciado.

— `speed` – Variável utilizada para controlar a velocidade da animação:

— `stop` – Variável que controla a existência ou ausência de movimentação. Tomará o valor `false` caso o movimento esteja ativo, ou o valor `true`, caso o movimento esteja inativo.

3.4.1 Transformações Geométricas

Para aplicar as transformações geométricas, foi necessário definir a função `apply_trans`, que, após a sua leitura dos ficheiros XML, irá aplicar estas transformações.

Rotação

Por forma a implementar o movimento de rotação dos planetas, que lhes permite rodar sobre o próprio eixo, foi necessário adicionar a variável `time` à classe `Transform`, que indica o tempo que uma figura demora a realizar uma rotação completa sobre o seu próprio eixo. Por forma a utilizar esta variável eficazmente, foi necessário também alterar a função `parseRotate` de modo a acomodar estas alterações.

Novamente, é necessário recorrer à variável `eTime` de forma a determinar o tempo já decorrido durante a movimentação dos planetas. Recorrendo à multiplicação deste valor pelo valor do ângulo de rotação, obtêm-se diferentes ângulos à medida que o tempo passa. Assim, torna-se possível dotar a figura de um movimento de rotação sobre o seu próprio eixo. Tendo tudo isto em causa, definiu-se o seguinte excerto de código para o processamento das transformações de movimento de rotação.

```
if (!strcmp(type, "rotateTime")) {
    float aux = eTime * angle;
    glRotatef(aux, x, y, z);
}
```

Translação

Foi também necessário capacitar os planetas de um movimento de translação em torno do Sol, usando para isso, também, a variável `time` referida anteriormente. Neste caso, esta variável define o tempo necessário para que uma determinada figura ou grupo percorra a curva definida pelos seus pontos de controlo, contidos no nodo `translate`. Para aplicar este movimento, foi necessário alterar a função `parseTranslate`, de modo a acomodar estas alterações.

Tal como mencionado na secção 3.1.1, o tempo de translação é passado como parámetro no ficheiro XML. O objetivo consiste em que, com o decorrer do tempo, o planeta se desloque ao longo da sua órbita, aplicando o movimento de translação desejado. Para tal ser possível, recorremos à variável `eTime`, que guarda o tempo decorrido enquanto que o sistema se encontrava em movimento, e ainda à função `getGlobalCatmullRomPoint`.

Para além de tudo isto, a função `apply_trans` terá também uma condição que será utilizada para definir a trajetória do cometa. Este deverá não só mover-se segundo a sua trajetória, como também manter se na orientação da curva. Deste modo, no caso do cometa e de outras figuras que se pretenda que o movimento seja descrito desta forma, o ficheiro XML deverá conter a informação disposta da seguinte forma:

```
<translate time="250" derivative="1">
    <point X="250.000000" Y="0.000000" Z="0.000000" />
    <point X="0.000000" Y="0.000000" Z="250.000000" />
    <point X="-250.000000" Y="0.000000" Z="0.000000" />
    <point X="-100.000000" Y="0.000000" Z="-100.000000" />
    <point X="0.000000" Y="0.000000" Z="-250.000000" />
</translate>
```

Surge agora aparece um parâmetro novo – a derivada do ponto da curva. De forma a que o cometa esteja orientado, é necessário criar uma matriz que irá ser utilizada na função `glMultMatrixf`, com o objetivo de manter o cometa alinhado com a curva.

Desta forma, o seguinte excerto é responsável pelos movimentos de translação.

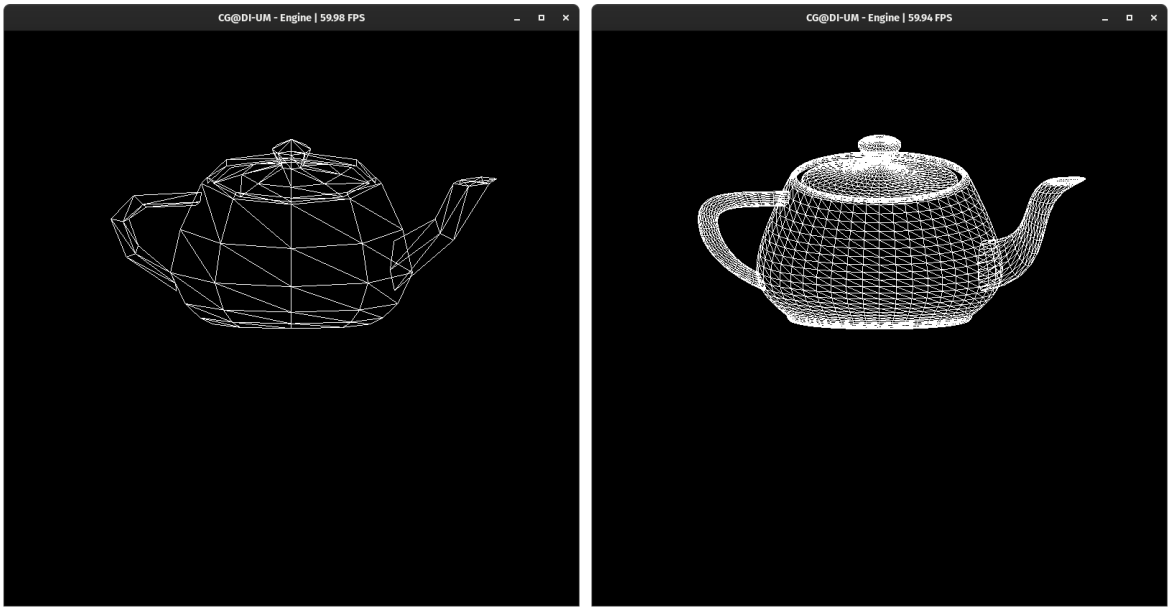
```
if (!strcmp(type, "translateTime")) {
    float p[4], d[4];
    float dTime = eTime * time;

    t->getGlobalCatmullRomPoint(dTime, p, d);

    drawOrbits(t);
    glTranslatef(p[0], p[1], p[2]);

    if (t->isDeriv()) {
        float res[4];
        t->normalize(d);
        t->cross_product(d, t->getVector(), res);
        t->normalize(res);
        t->cross_product(res, d, t->getVector());
        float matrix[16];
        t->normalize(t->getVector());
        t->rotMatrix(matrix, d, t->getVector(), res);
        glMultMatrixf(matrix);
    }
}
```

Note-se que na aplicação das transformações geométricas foi utilizada a função `glutGet(GLUT_ELAPSED_TIME)` para o cálculo do tempo passado para que desta forma a velocidade dos planetas não esteja dependente da *frame rate* do programa.

4 Análise de Resultados

4.1 *Teapot*

De seguida, apresentam-se imagens que representam a renderização um *teapot* com valores de tesselação de 2 e 10, respetivamente, criados a partir do ficheiro `teapot.patch` fornecido, e utilizando as novas funcionalidades do *generator*, que permitem criar pontos através de *patches* de Bézier.



(a) *Teapot* renderizado com um nível de tesselação igual a 2

(b) *Teapot* renderizado com um nível de tesselação igual a 10

Figura 4.1: Renderização de um *teapot* a partir de *patches* de Bézier

4.2 Modelo do Sistema Solar

No modelo do Sistema Solar, cada um dos astros tem a sua própria órbita e esta é animada, à excepção do Sol.

De facto, todos os planetas foram representados com o máximo de cuidado no sentido de respeitar diferenças de tamanho e distância, mas sem nunca perder de vista que se trata de uma animação e que, por isso, não consegue nem deve respeitar a 100% a escala real. Os movimentos de rotação e translação foram representados tentando também uma aproximação o mais possível à realidade. Houve ainda a preocupação de adicionar um cometa gerado através de um ficheiro de *input* de um *patch* de Bézeier para demonstrar também as capacidades do *generator*.

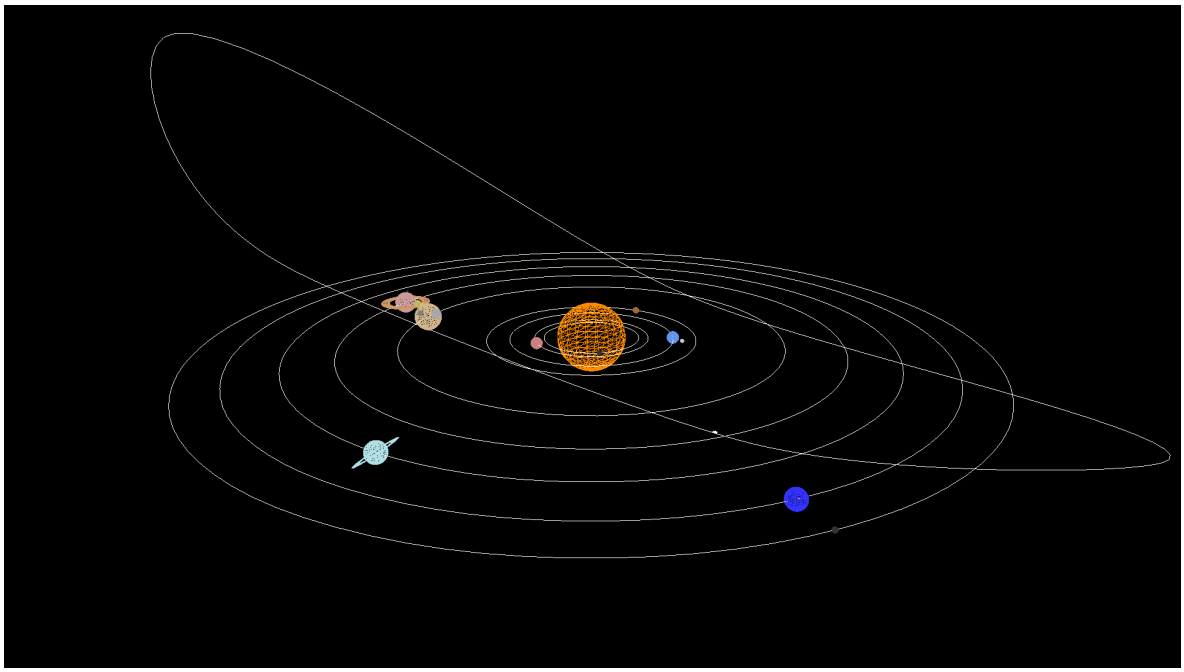


Figura 4.2: Modelo do Sistema Solar renderizado com linhas

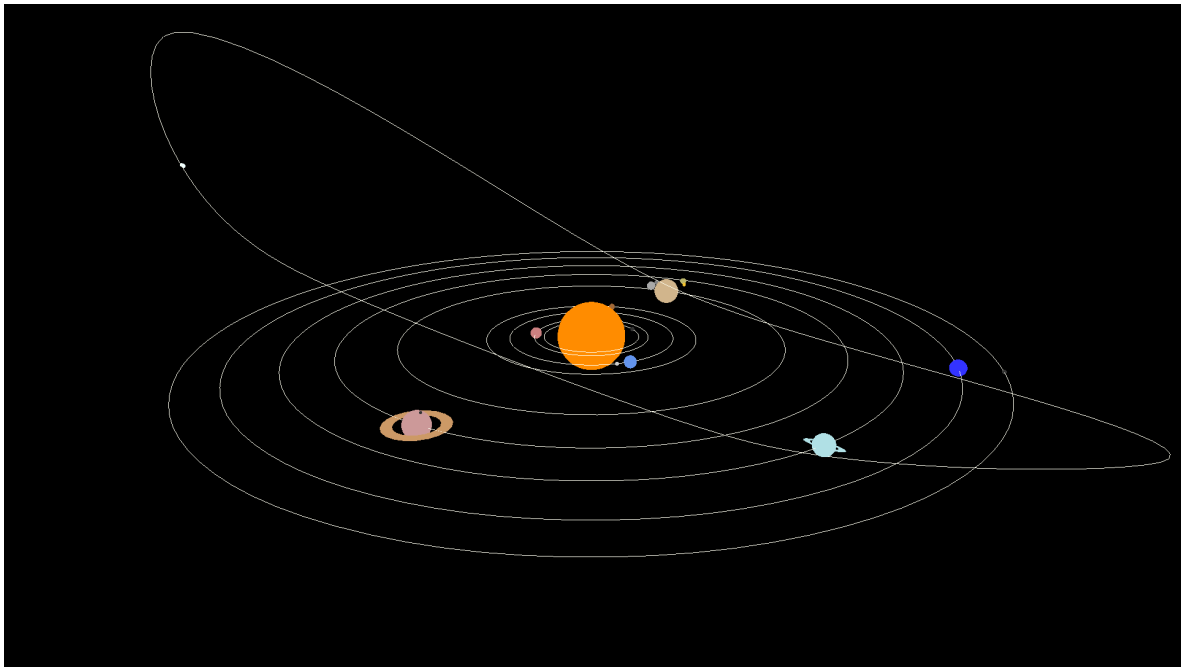


Figura 4.3: Modelo do Sistema Solar renderizado com as figuras preenchidas

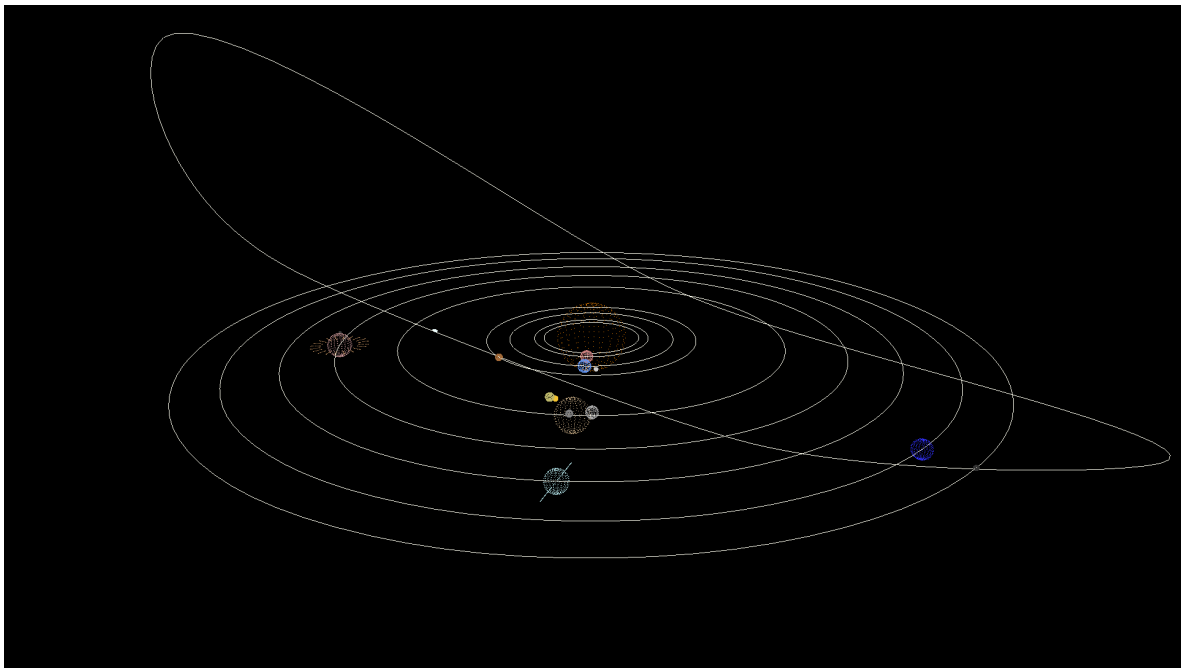


Figura 4.4: Modelo do Sistema Solar renderizado com pontos

4.3 *Performance*

Tal como referido anteriormente, o uso de VBOs permite aumentar de forma significativa o desempenho do programa, sendo visível um aumento do número de *frames* por segundo, em comparação aos verificados quando se renderiza a mesma *scene* mas, desta vez, em que os modelos eram desenhados de uma forma mais primitiva – através de renderização imediata – tal como se pode ver de seguida.

Importa salientar que esta medição de *performance* foi realizada tendo o *vertical sync* desligado de forma a garantir que os resultados obtidos não estejam limitados pela sincronização vertical.

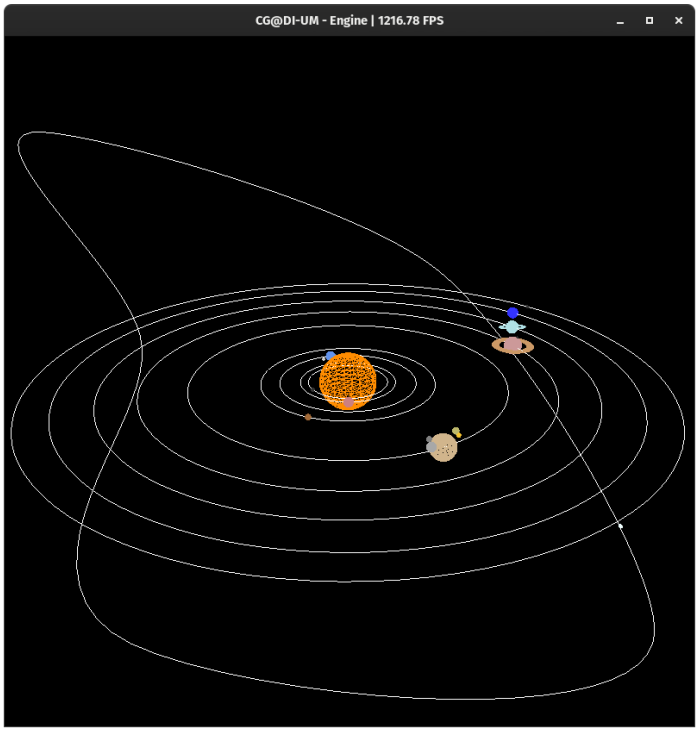


Figura 4.5: Modelo dinâmico do Sistema Solar renderizado com *Vertex Buffer Objects*

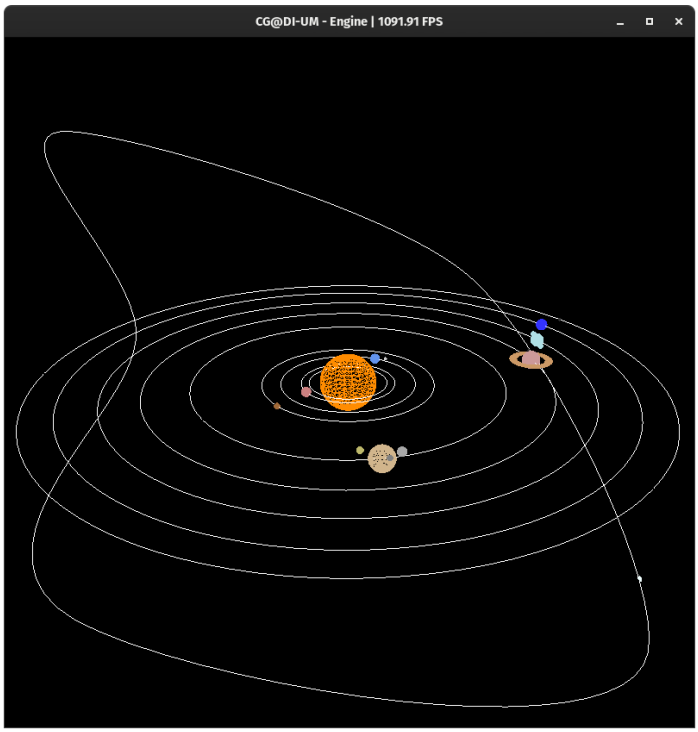


Figura 4.6: Modelo dinâmico do Sistema Solar renderizado através de renderização imediata

5 Conclusão

Esta terceira fase do trabalho prático permitiu assimilar conhecimento relativamente aos processos de representação de figuras em Computação Gráfica. Este conhecimento foi adquirido através da matemática envolvente por detrás da renderização de imagens 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 Bézier e de curvas Catmull-Rom.

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

Com a elaboração desta fase do projeto prático foi possível aplicar conhecimentos teóricos e práticos relativos tanto às curvas de Bézier como às de Catmull-Rom utilizadas para efetuar transformações sobre os objetos através de pontos de controlo e outros dados relativos, assim como o uso de VBOs para o desenho das primitivas gráficas.

Assim, considerámos que o resultado final desta fase corresponde às expectativas, na medida em que conseguimos desenvolver um modelo, agora dinâmico, do Sistema Solar, tal como era pedido no enunciado. De facto, todas as funcionalidades pedidas foram implementadas. No entanto, algumas optimizações não foram implementadas, como por exemplo, o uso de índices nos VBOs.

Desta forma, esperamos que para a próxima e última fase que se avizinha, consigamos concluir o projeto de forma a obter um resultado final ainda mais realista e visualmente agradável ao utilizador.

Referências

[1] Rendering Cubic Bezier Patches. https://web.cs.wpi.edu/~matt/courses/cs563/talks/surface/bez_surf.html. (Acedido em 23/04/2021).