Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Relatório do projeto

Fase 3 - Curvas, Superfícies cúbicas e VBO's

Computação Gráfica

Grupo de trabalho 37
André Santos, A61778
Diogo Machado, A75399
Lisandra Silva, A73559
Rui Leite, A75551

Braga, 1 de Maio de 2017

Conteúdo

Resumo

O presente relatório documenta a 3.ª fase do trabalho prático da Unidade Curricular de Computação Gráfica que consistiu na evolução do gerador e do motor desenvolvidos na fases anteriores. O gerador passou a criar um novo tipo de modelo tendo por base superfícies de Bezier. Relativamente a fase anterior, o motor apresenta novas funcionalidades como o uso de VBOs para o desenho dos modelos, em oposição ao modo imediato anteriormente usado, e ainda transformações dependentes do tempo, nomeadamente translações e rotações. Os pontos das translações são obtidos recorrendo a curvas cúbicas de Catmull-Rom. Foram ainda adicionados atributos aos elementos XML por forma a ser possível criar *scene* mais interessantes e dinâmicas. Foram feitas alterações ao sistema solar anteriormente elaborado com o objetivo de demonstrar as alterações feitas.

Estrutura do relatório

O relatório está organizado em 6 capítulos que pretendem ilustrar o processo de resolução do enunciado proposto.

No capítulo 1 apresentam-se os elementos XML usados para a construção de cenas (*scenes*).

No capítulo 2 apresentam-se as estruturas de dados usadas para guardar em memória os dados, bem como suportar e manipular toda a informação necessária para o processo de desenho.

No capítulo 3 faz-se uma introdução às noções de curvas e de superfícies de Bézier, uma apresentação da forma como deve ser invocado o programa Gerador por forma a ler um ficheiro com uma estrutura bem definida e produzir outro com todos os pontos nos triângulos respetivos.

O capítulo 4 detalha as alterações feitas ao motor de modo a suportar as novas funcionalidades, nomeadamente as translações e rotações animadas, bem como o desenho usando os Vertex Buffer Object's.

No capítulo 5 é feita uma explicação das funcionalidades com as quais o utilizador pode interagir com o motor recorrendo ao teclado e ao rato.

Por fim, no capitulo 6, é explicado como foi construido a cena de exemplo correspondente ao sistema solar, desde as figuras usadas bem como as decisões adotadas na contrução do ficheiro XML.

1. Leitura XML e ficheiros de pontos

1.1 Elementos XML

Nesta secção apresentam-se os elementos XML que podem ser usados para a construção de cenas. Estes elementos serão lidos pelo Motor para o desenho da cena. Para cada um dos elementos XML possíveis, além do seu nome, apresentam-se os atributos que estes podem ter e que valores podem tomar. Um elemento XML é muitas vezes também designado de nodo XML, pelo que estes termos serão usados indistintamente ao longo do relatório.

1.1.1 Nodo *<scene>*

O elemento `scene` é o nodo pai de todo o documento e não tem qualquer atributo. Todos os ficheiros descritivos de uma cena devem começar por abrir a *tag* deste elemento e terminar fechando a *tag*:

```
<scene>
    ...
    Grupos XML
    ...
</scene>
```

Todos os restantes elementos do XML são por isso filhos do elemento `scene`.

1.1.2 Nodo *<group>*

Nodo filho de `<scene>` que descreve um grupo. Um grupo contém a descrição de ficheiros com pontos a ser desenhados e um conjunto de transformações que deverão ser aplicadas a esses pontos. Poderá também ter outros nodos `group` como filhos. Este nodo apenas possui um atributo:

- ***name*** - Nome atribuído ao grupo pelo utilizador. Este nome não tem qualquer influência no comportamento do Motor, no entanto facilita a leitura do XML por parte de humanos. O valor do atributo poderá por isso ser qualquer *string*.

```
<scene>
    <group name="Sol">
    ...
    Transformacoes, models, grupos...
    ...
    </group>
</scene>
```

1.1.3 Nodo *<translate>*

Nodo filho de `<group>` que descreve uma translação. A translação nesta fase 3 pode ser de 2 tipos: não animada ou animada, sendo usados diferentes atributos para descrever cada um destes tipos de translação. A translação não animada corresponde a uma translação de um objeto de acordo com um vector constante ao longo do tempo. Na translação animada, o vector pelo qual é feita a translação varia ao longo do tempo de uma forma específica, conferindo o efeito de animação.

Translação não animada

Para uma translação não animada, são relevantes os seguintes atributos:

- **X** - Componente x do vetor da translação simples (sem animação). O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso não seja indicado nenhum valor para este atributo, assume-se que a componente x do vector de transação tem valor 0. Não é um atributo obrigatório.
- **Y** - Componente y do vetor da translação simples (sem animação). Caso não seja indicado nenhum valor para este atributo, assume-se que a componente y do vector de transação tem valor 0. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.
- **Z** - Componente z do vetor da translação simples (sem animação). Caso não seja indicado nenhum valor para este atributo, assume-se que a componente z do vector de transação tem valor 0. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.

Embora o Motor não o force, na especificação de uma translação não animada, pelo menos um dos atributos acima deve ser indicado.

Quando a translação é não animada, este elemento é um "elemento vazio" (*"empty element"* na terminologia do XML), pelo que a sua informação está apenas na *tag* e nos atributos. Dentro do grupo, este elemento poderá aparecer depois de outras transformações, mas nunca depois de um outro grupo.

```
<scene>
    <group name="Sol">
        <translate X="2.4"/>
        ...
        Transformacoes, models, grupos...
        ...
    </group>
</scene>
```

O exemplo apresentado acima corresponde por isso a fazer uma translação de $x = 2.4$, $y = 0$, $z = 0$ aos pontos de um grupo designado por "Sol".

Translação animada

Para uma translação animada, são relevantes os seguintes atributos:

- **time** - Tempo (em segundos) para efetuar a translação. O uso deste atributo corresponde a indicar que a translação será animada e implica a definição de pelo menos 4 nodos `point` como filhos que indicam os pontos de controlo da curva de catmull-rom usados para a translação no tempo especificado. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso se pretenda uma translação animada, este atributo é obrigatório.

- **drawCatmullCurve** - Este atributo apenas deve ser usado em conjunto com o atributo `time` nas translações animadas. O valor deste atributo deverá ser uma *string* "true" ou "false" caso se pretenda que a curva de Catmull-Rom de translação do objeto seja desenhada ou não, respetivamente. Caso seja indicado o atributo `time` e este atributo não seja especificado, este assume por defeito o valor de "false", ou seja, por defeito, a curva da translação não é desenhada.

- **nrCatmullPointsToDraw** - Indica o número de pontos da curva da Catmull-Rom que se pretende ver desenhados. Este atributo apenas afeta os pontos desenhados e não os pontos que o objeto efetivamente usa para percorrer a curva. Este atributo apenas deve ser usado caso se tenha definido o atributo `drawCatmullCurve` com o valor "true", sendo ignorado caso contrário. Caso se tenha definido o atributo `drawCatmullCurve` com o valor "true" mas não se especifique nenhum valor para o número de pontos da curva a desenhar, por defeito os pontos da curva desenhados serão os pontos de controlo indicados pelos nodos filhos `point`. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *int*. Não é um atributo obrigatório.

- **direction** - Indica se a animação de translação do objeto deve ser realizada no sentido dos ponteiros do relógio ou no sentido contrário. Deve ser usado apenas em conjunto com o atributo `time` para translações animadas. O valor deste atributo deverá ser uma *string* "cw"(de ***c****lock****w****ise*) ou "ccw" (de ***c****ounter****c****lock****w****ise*) caso se pretenda que a translação seja realizada no sentido dos ponteiros do relógio ou no sentido contrário, respetivamente. Caso a translação seja animada (existência do atributo `time`) e este atributo não seja indicado, assume-se a direcção contrária aos ponteiros do relógio por defeito. Não é um atributo obrigatório.

De seguida apresenta-se um exemplo de uma translação animada com duração de 3 segundos que passa em 4 pontos de controlo e em que são desenhados 100 pontos da curva de Catmull-Rom correspondente.

```
<group name="Phobos (Mars Moon)">
    <translate time="3.0" drawCatmullCurve="true"
       nrCatmullPointsToDraw="100">
        <point X="0.0" Y="0" Z="0.3"/>
        <point X="0.3" Y="0" Z="0"/>
        <point X="0" Y="0" Z="-0.3"/>
```

```
        <point X="-0.21" Y="0" Z="0.21"/>
    </translate>
    ...
    Transformacoes, models, grupos...
    ...
</group>
```

1.1.4 Nodo *<point>*

Nodo filho de `<translate>` usado para indicar um ponto de controlo da curva de Catmull-Rom de uma translação animada. Este elemento possui os seguintes atributos:

- **X** - Componente em x do ponto. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. É um atributo obrigatório.
- **Y** - Componente em y do ponto. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. É um atributo obrigatório.
- **Z** - Componente em z do ponto. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. É um atributo obrigatório.

Este elemento é um "elemento vazio" (*"empty element"* na terminologia do XML), pelo que a sua informação está apenas na *tag* e nos atributos.

1.1.5 Nodo *<rotate>*

Nodo filho de `<group>` que descreve uma rotação. Tal como acontece na translação, também a rotação pode ou não ser animada. A animação da rotação corresponde a uma rotação de 360º segundo um determinado eixo durante um período de tempo especificado. Tanto na rotação animada como na rotação não animada este elemento é um "elemento vazio" (*"empty element"* na terminologia do XML), pelo que a sua informação está apenas na *tag* e nos atributos. Dentro do grupo, este elemento poderá aparecer depois de outras transformações, mas nunca depois de um outro grupo. Em ambos os casos, é necessário indicar o eixo sobre o qual se efetuará a rotação, o que é dado pelos seguintes atributos:

- **X** - Componente x do vetor sobre o qual será feita a rotação. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso não seja indicado um valor para este atributo é assumido o valor 0. Não é um atributo obrigatório.
- **Y** - Componente y do vetor sobre o qual será feita a rotação. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso não seja indicado um valor para este atributo é assumido o valor 0. Não é um atributo obrigatório.

- **Z** - Componente z do vetor sobre o qual será feita a rotação. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso não seja indicado um valor para este atributo é assumido o valor 0. Não é um atributo obrigatório.

Rotação não animada

Na rotação não animada, além dos atributos anteriores, deve ser ainda indicado mais um:

- ***angle*** - Descreve o ângulo de rotação (em graus). O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório, embora deva ser indicado para a rotação ter sentido.

O exemplo apresentado abaixo corresponde a fazer uma rotação em torno do eixo Oz de 30^o aos pontos de um grupo designado por "Sol".

```
<scene>
    <group name="Sol">
        <rotate angle="30" X="0" Y="0" Z="1"/>
        ...
        Transformacoes, models, grupos...
        ...
    </group>
</scene>
```

Rotação animada

- **time** - Tempo (em segundos) para efetuar a rotação. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Caso se pretenda uma rotação animada, este atributo é obrigatório.

- **direction** - Indica se a animação de rotação do objeto deve ser realizada no sentido dos ponteiros do relógio ou no sentido contrário. Deve ser usado apenas em conjunto com o atributo `time` para rotações animadas. O valor deste atributo deverá ser uma *string* "cw"(de ***c**lock**w**ise*) ou "ccw" (de ***c**ounter**c**lock**w**ise*) caso se pretenda que a rotação seja realizada no sentido dos ponteiros do relógio ou no sentido contrário, respetivamente. Caso a rotação seja animada (existência do atributo `time`) e este atributo não seja indicado, assume-se a direção contrária aos ponteiros do relógio por defeito. Não é um atributo obrigatório.

O exemplo abaixo indica uma rotação do planeta terra de 360 graus segundo o vetor $(0, 1, 0)$ durante 5 segundos, na direcção contrária aos ponteiros do relógio.

```
<group name="Planet Earth">
    <rotate time="5" X="0" Y="1" Z="0" direction="ccw"/>
    ...
    Transformacoes, models, grupos...
    ...
</group>
```

1.1.6 Nodo *<scale>*

Nodo filho de `<group>` que descreve uma escala. Este elemento pode ter os seguintes atributos:

- ***uniform*** - Descreve o valor de uma escala uniforme (aplicada de igual forma em todos os eixos). O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório, no entanto, se este atributo for usado, deverá ser o único, todos os outros atributos serão ignorados.
- **X** - Indica a escala do eixo x. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.
- **Y** - Indica a escala do eixo y. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.
- **Z** - Indica a escala do eixo z. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.

Se for usado o atributo `uniform` este deve ser o único atributo a ser usado. O Motor não força a que isto aconteça, no entanto caso seja usado o atributo `uniform` em conjunto com outros, a escala será feita sempre segundo o valor de `uniform` - todos os outros serão ignorados. Caso não seja usado o atributo `uniform`, pelo menos um dos restantes referente às escalas em cada eixo (x, y e z) deve ser usado. Caso um atributo de uma escala de um eixo não seja indicado, é assumido por defeito o valor 1 para a escala desse eixo.

Este elemento é um "elemento vazio" (*"empty element"* na terminologia do XML), pelo que a sua informação está apenas na *tag* e nos atributos. Dentro do grupo, este elemento poderá aparecer depois de outras transformações, mas nunca depois de um outro grupo.

```
<scene>
    <group name="Sol">
        <scale uniform="0.11"/>
        ...
        Grupos e transformacoes
        ...
    </group>
</scene>
```

O exemplo apresentado acima corresponde a fazer uma escala uniforme de 0.11 ao desenho dos pontos de um grupo designado por "Sol". A mesma escala também poderia ter sido indicada da seguinte forma:

```
<scale X="0.11" Y="0.11" Y="0.11"/>
```

1.1.7 Nodo *<models>*

Elemento filho de `<group>` que irá conter a descrição dos modelos a ser desenhados. Não possui qualquer atributo. Deverá aparecer depois de todas as transformações do grupo e os elementos seguintes poderão apenas ser outros grupos.

```
<scene>
    <group name="Sol">
        Transformacoes

        <models>
            ...
            Modelos
            ...
        </models>
        Outros grupos
    </group>
</scene>
```

1.1.8 Nodo *<model>*

Nodo filho de `<models>` que descreve um modelo a ser desenhado. Este elemento pode ter os seguintes atributos:

- ***file*** - Descreve o nome de um ficheiro onde estarão os pontos a ser desenhados. É um atributo obrigatório.
- ***red*** - Valor para a cor de vermelho no sistema de cores RGB. Serve para indicar a cor dos pontos a ser desenhados. O valor deste atributo deverá ser uma *string* correspondente a um número entre 0.0 e 1.0 que possa ser convertido para *float*. Não é um atributo obrigatório.
- ***green*** - Valor para a cor verde no sistema de cores RGB. Serve para indicar a cor dos pontos a ser desenhados. O valor deste atributo deverá ser uma *string* correspondente a um número entre 0.0 e 1.0 que possa ser convertido para *float*. Não é um atributo obrigatório.
- ***blue*** - Valor para a cor azul no sistema de cores RGB. Serve para indicar a cor dos pontos a ser desenhados. O valor deste atributo deverá ser uma *string* correspondente a um número entre 0.0 e 1.0 que possa ser convertido para *float*. Não é um atributo obrigatório.
- ***mode*** - Descreve modo de desenho dos pontos pretendido. Este atributo pode tomar 3 valores: “FILL”, “LINE” e “POINT”. Isto fará que os pontos sejam desenhados pelo OpenGl usando o modo `GL_FILL`, `GL_LINE` e `GL_POINT`, respetivamente. Não é um atributo obrigatório.

Apenas o atributo `file` é obrigatório. Caso não seja indicado um atributo de cor, esse atributo toma por defeito o valor 1. Ou seja, se nenhum atributo de cor for indicado, usa-se por defeito a cor branca (“red=1, green=1, blue=1”). Caso o atributo `mode` não seja especificado, usa-se por defeito o modo “LINE” (`GL_LINE`).

Este elemento é um “elemento vazio” (*“empty element”* na terminologia do XML), pelo que a sua informação está apenas na tag e nos atributos.

```
<scene>
    <group name="Earth">
        Transformacoes
        <models>
            <model red="0.0" green="0.0"
            blue="0.5882352941176471"
            mode="FILL" file="esfera_1.3d"/>
        </models>
        ...
    </group>
</scene>
```

2. Estruturas de dados

2.1 Coordenadas3D

A classe `Coordenadas3D` foi usada para representar coordenadas num espaço a 3 dimensões. Esta classe consegue por isso representar tanto pontos como vetores. É constituída por 3 variáveis de instância correspondentes às coordenadas x, y e z.

```
struct Coordenadas3D {
    float x, y, z;
};
```

Esta classe implementa ainda algumas operações tais como soma, subtração, produto externo e produto por um escalar.

```
struct Coordenadas3D {

    Coordenadas3D operator+(const Coordenadas3D& p2);
    Coordenadas3D operator-(const Coordenadas3D& p2);
    Coordenadas3D crossproduct(const Coordenadas3D& p2);
    Coordenadas3D operator*(float k);

};
```

2.2 Transformações

Conforme visto na secção 1.1, cada grupo pode ter 3 tipos de transformação: translações, rotações e escalas. Para representar este facto foi criada uma classe transformação que consiste numa *union* dos diferentes tipos de transformação:

```
class Transformacao {
public:
        TipoTransformacao tipo;
        union UTr {
                Translacao t;
                Rotacao r;
                Escala e;
        } Tr;
};
```

Além da *union*, a classe possui ainda uma variável `tipo` que indica o tipo da transformação guardado na *union*. Esta variável permite facilmente saber o tipo da trans-

formação. O tipo `TipoTransformacao` corresponde a uma enumeração dos tipos de transformações referidos anteriormente.

```
enum TipoTransformacao { ROTACAO, TRANSLACAO, ESCALA };
```

Para cada uma destas transformações foi criada uma classe cujas variáveis de instância são os parâmetros de cada transformação conforme se apresenta de seguida.

2.2.1 Translação

A classe `Translacao` tem as seguintes variáveis de instância:

```
class Translacao {
public:
    float tx, ty, tz, time;
    std::vector <Coordenadas3D> ctrlPoints;
    bool ccw;
};
```

As variáveis de instância `tx`, `ty` e `tz` guardam o vector de uma translação não animada conforme visto na secção 1.1.3. As restantes variáveis são usadas no caso de uma translação animada: a variável `time` guarda o tempo da translação; a variável `ctrlPoints` guarda o conjunto de pontos de controlo da curva de Catmull-Rom da animação e a variável `ccw` se a direcção da translação animada é no sentido contrário aos ponteiros do relógio.

2.2.2 Rotação

A classe `Rotacao` tem as seguintes variáveis de instância:

```
class Rotacao {
public:
    float rang, rx, ry, rz;
    float time;
    bool ccw;
};
```

As variáveis `rx`, `ry` e `rz` guardam o eixo sobre o qual se dá a rotação, conforme visto em 1.1.5. Para as rotações não animadas a variável `rang` guarda o ângulo da rotação. Para as rotações animadas, a variável `time` guarda o tempo de rotação e a variável `ccw` guarda se a direção da rotação é no sentido contrário aos ponteiros do relógio.

2.2.3 Escala

A classe `Escala` tem as seguintes variáveis de instância:

```
class Escala {
public:
    float sx, sy, sz;
};
```

As variáveis `sx`, `sy` e `sz` guardam o factor de escala a aplicar a cada um dos eixos x, y e z respetivamente.

2.3 Definições Desenho

Cada objeto pode ser desenhado de diferentes formas. Nomeadamente, um objeto pode ter uma cor, pode ser desenhado apenas com pontos, linhas ou totalmente preenchido e os pontos que constituem o objeto a ser desenhado podem ser vistos de diferentes formas em *OpenGL*, como conjuntos de triângulos com `GL_TRIANGLES`, como conjunto de linhas fechadas com `GL_LINE_LOOP`, entre outros. Em termos de código, estas definições foram captadas pela classe `DefsDesenho`

```
class DefsDesenho {
public:
    float red, green, blue;
    GLenum modoPoligonos;
    GLenum modoDesenho;
};
```

As variáveis `red`, `green` e `blue` correspondem aos `float`'s necessários para representação de uma cor no sistema *RGB*. A variável `modoPoligonos` indica se o objeto deve ser desenhando com pontos, com linhas ou com as faces preenchidas, que corresponde às definições `GL_POINT`, `GL_LINE` e `GL_FILL` do *OpenGL*. A variável `modoDesenho` indica de que forma devem ser interpretados os pontos do desenho pelo *OpenGL*. Existem diversas possibilidades para esta definição, no entanto no contexto deste trabalho prático destaca-se a interpretação dos pontos como triângulos (com `GL_TRIANGLES`) ou como linhas fechadas (com `GL_LINE_LOOP`). A primeira será a opção mais usada para o desenho de figuras, enquanto que a segunda opção será mais útil para o desenho das curvas de Catmull-Rom.

Os tipos destas variáveis foram escolhidos de forma a permitir que as variáveis possam ser passadas diretamente às funções do OpenGl responsáveis por aplicar estas definições. Assim, a função `glColor3f()` recebe 3 *float*'s para a definição das cores e as funções `glPolygonMode()` e `glBegin()` para a definição do modo de polígonos e modo de desenho, respetivamente, recebem ambas um `GLEnum`.

2.4 Desenho

Cada objeto a ser desenhado encontra-se representado pela classe `Desenho`.

```
struct Desenho {
    DefsDesenho defsDesenho;
    std::vector<Coordenadas3D> pontos;
    int nBuff;
};
```

A variável `pontos` contem o conjunto de pontos que formam o objeto e a variável `defsDesenho` contém as definições de desenho que indicam de que forma os pontos vão ser desenhados, conforme visto na secção anterior. Nesta fase do trabalho o motor deve ser capaz de desenhar os objetos com VBO's. Nesse sentido foi ainda incluído no desenho a variável `nBuff` que guarda o número do buffer de dados do OpenGL onde se encontram os pontos.

Visto que os desenhos do motor serão feitos com VBO's, os dados dos pontos encontram-se armazenados na memória da placa gráfica, pelo que teoricamente não

seria necessário guardar os pontos do desenho nesta estrutura que será guardada em memória RAM. Isto sugere que a variável `pontos` poderia ser excluída de forma a se poupar memória. No entanto, para efeitos de comparação da performance entre o desenho em modo imediato e o desenho com VBO's, os pontos são guardados tanto nos buffers de dados do OpenGL como em memória RAM, através da variável `pontos`.

2.5 Grupo

Foi criada uma classe `Grupo` que pretende representar cada um dos grupos XML apresentados na secção 1.1.2. Esta classe tem as seguintes variáveis de instância:

```
class Grupo {
public:
    std::string nome;
    std::vector<std::string> ficheiros;
    std::vector<Transformacao> transformacoes;
    std::vector<Desenho> desenhos;
    std::vector<Desenho> catmullDes;
    int nrCRPtsToDraw;
};
```

- **nome** - *String* que contém o nome do grupo. Este nome corresponde ao valor do atributo `name` do elemento `<group>` do XML. Como referido na secção 1.1.2, este atributo serve apenas para tornar o XML mais legível para humanos. Isto implica que na prática guardar este elemento não é estritamente necessário para o funcionamento do programa, no entanto foi decidido armazenar o nome nesta variável para que quando for impresso um objeto do tipo `Grupo`, também possa ser apresentado o seu nome.

- **ficheiros** - *String*'s correspondentes aos valores dos atributos `file` de todos os elementos `<model>` que fazem parte do grupo. Tal como o nome do grupo, o armazenamento do nome dos ficheiros também tem como objetivo tornar mais interessante a informação mostrada ao utilizador quando um objeto do tipo `Grupo` é impresso.

- **transformacoes** - contém as transformações do grupo definidas pelos elementos `<rotate>`, `<translate>` e `<scale>`. As transformações do grupo são armazenadas neste `vector` pela mesma ordem com que são declaradas no XML.

- **desenhos** - Vector que contém todos os desenhos do grupo presentes no elemento `<models>` do XML. Os elementos deste vector correspondem aos desenhos indicados pelos elementos `<model>`, filhos de `<models>`.

- **catmullDes** - Vector que guarda os desenhos correspondentes às curvas de Catmull-Rom. Este vector apenas é usado no caso de translações animadas em que é indicado no XML que se pretende o desenho da curva. Estes desenhos

poderiam ter sido incluídos no vector `desenhos`, no entanto por motivos de organização de código optou-se por separar em dois vectores diferentes estes dois tipos de desenho.

- **nrCRPtsToDraw** - Número de pontos da curva de Catmull-Rom que se pretende que sejam desenhados. Este valor apenas é usado no caso de translações animadas em que é indicado no XML que se pretende o desenho da curva. Por simplificação, optou-se por assumir que caso haja várias curvas de Catmull-Rom a ser desenhadas dentro de um grupo, todas são desenhadas com o mesmo número de pontos.

3. Gerador

Nesta fase do trabalho prático foi necessário acrescentar uma funcionalidade ao programa Gerador: a de "gerar" modelos com base em superfícies de Bézier. Antes de explicitar todo o processo de desenvolvimento desta nova funcionalidade, é pertinente fazer uma introdução aos conceitos de Curva de Bézier e de Superfície de Bézier.

3.1 Curva de Bézier

Uma curva de Bézier é uma curva polinomial obtida por interpolação linear entre alguns pontos representativos, ditos de pontos de controlo. Estes devem ser, em quantidade, mais um do que o grau da curva, i.e., $n + 1$ pontos. São as curvas de Bézier que dão origem às Superfícies de Bézier, apresentadas de seguida.

Na figura 3.1 é possível ver a forma como é obtida uma curva de Bézier cúbica. Os pontos P_1, P_2, P_3 e P_4 são os chamados pontos de controlo, que definem a curva. O índice t é uma valor de parametrização para percorrer a curva e obter cada um dos pontos B_t, com $t \in [0, 1]$ (ou, em percentagem, $t \in [0, 100]\%$).

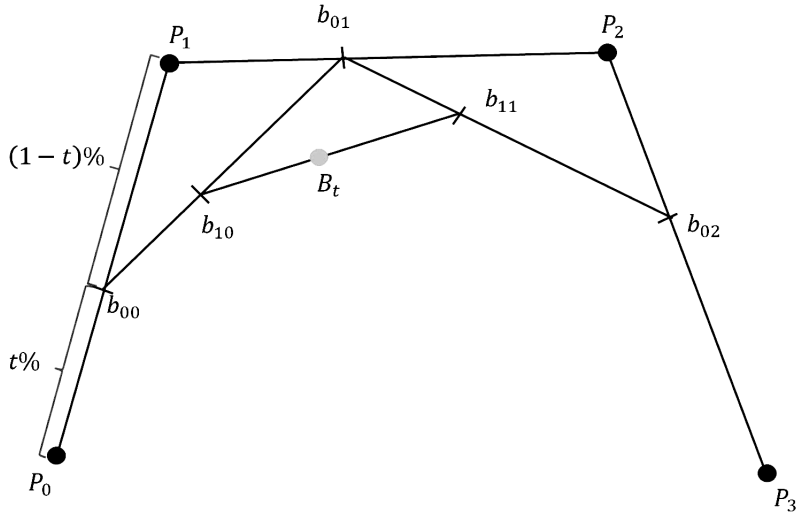

Figura 3.1: Curva de Bezier (de António Ramires, UMinho)

Cada um dos pontos $B_t = B(t)$ pode ser obtido por uma expressão analítica em função do parâmetro t:

$$B(t) = t^3P_3 + (-3t^3 + 3t^2)P_2 + (3t^3 - 6t^2 + 3t)P_1 + (-t^3 + 3t^2 - 3t + 1)P_0$$

3.2 Superfície de Bézier

Uma superfície de Bézier (*Bézier Patch*) é também obtida através de um conjunto de pontos de controlo. Se se tratar de uma superfície cúbica, tem-se que um *patch* é obtido pelo conjunto de 4 curvas de Bézier. Dado que cada curva tem 4 pontos de controlo, uma superfície tem $4 \times 4 = 16$ pontos de controlo. Como se pode ver pela figura 3.2, cada linha de 4 pontos de controlo pode ser visto como uma curva de Bézier em separado.

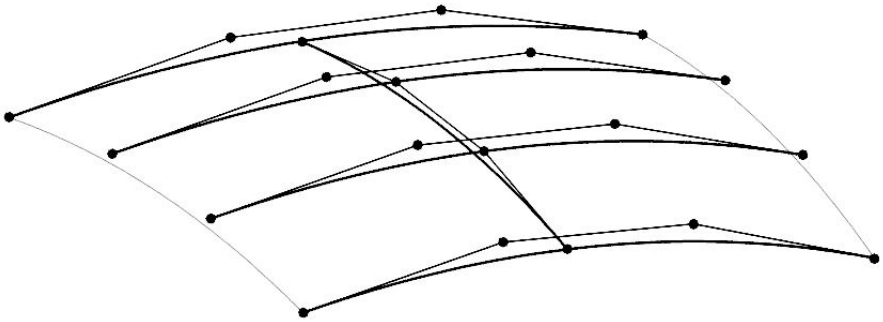

Figura 3.2: *Bézier Patch* (de math.stackexchange.com)

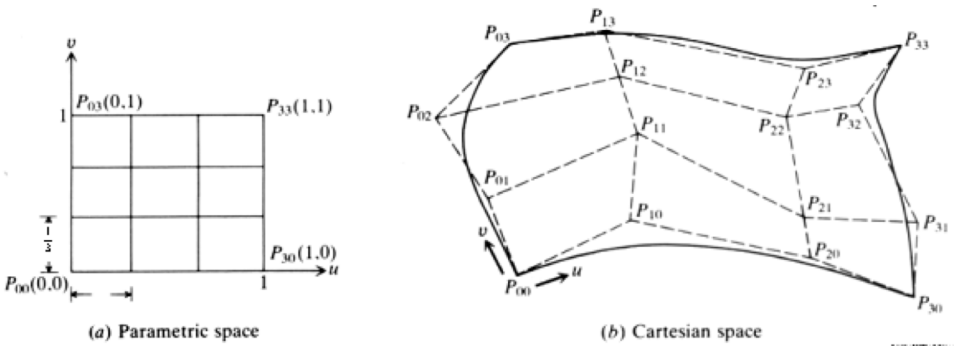

Figura 3.3: Pontos de um *Patch* (de designer.mech.yzu.edu.tw)

Cada um dos pontos $B(u, v)$ da superfície pode ser definido em função dos valor paramétricos u e v e dos pontos de controle P_{uv} (conforme a figura 3.3) pela seguinte expressão:

$$B(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix} \tag{3.1}$$

com M a matriz:

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

3.3 Implementação

Feita uma introdução às curvas e superfícies de Bézier, pode-se agora explicitar todas as decisões tomadas na implementação do programa Gerador. Este tem que ler um ficheiro com um ou mais *patches* e com todos os pontos de controlo de cada um e gerar um ficheiro .3d com todos os triângulos (com as coordenadas x, y e z de cada um dos três pontos de cada triângulo), por forma a ser possível de ler pelo programa Motor.

O ficheiro de entrada deve ter o seguinte formato:

```
Número de patches
Índice no ficheiro de cada um dos 16 pontos de cada patch
Número total de pontos
 Todos os pontos (coordenadas x, y e z)
```

FICHEIRO DE EXEMPLO

```
1
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
16
1.4, 0, 2.4
1.4, -0.784, 2.4
0.784, -1.4, 2.4
0, -1.4, 2.4
1.3375, 0, 2.53125
1.3375, -0.749, 2.53125
0.749, -1.3375, 2.53125
0, -1.3375, 2.53125
1.4375, 0, 2.53125
1.4375, -0.805, 2.53125
0.805, -1.4375, 2.53125
0, -1.4375, 2.53125
1.5, 0, 2.4
1.5, -0.84, 2.4
0.84, -1.5, 2.4
0, -1.5, 2.4
```

O comando que deve ser executado para se obter o ficheiro final com o Gerador deve fornecer ao programa dois níveis de *tessellation*, que corresponderão ao nível de divisões em u e em v, em cada *patch*, o nome do ficheiro de entrada e o nome do ficheiro de saída:

```
gerador bezier tessU tessV ficheiroIn ficheiroOut
```

3.3.1 Classe `SuperficieBezier`

Para a implementação desta funcionalidade do Gerador foi criada uma classe `SuperficieBezier`. Nesta ficam guardados dois `vector`: um de `Coordenadas3D` com todos os pontos de controlo do ficheiro e outro de `vector<int>` com os índices de cada um dos *patches*.

```
class SuperficieBezier {
public:
 std::vector<Coordenadas3D> pontosControlo;
 std::vector<std::vector<int>> patches;

}
```

Esta classe, para além de disponibilizar métodos de inserção nas suas variáveis de instância, disponibiliza uma função `getTriangulos` que será apresentada mais à frente, nesta secção.

3.3.2 Leitura do ficheiro de entrada

Para proceder à leitura do ficheiro de entrada for implementada a função `leBezier` que recebe como argumento o nome do ficheiro e retorna uma instância da classe `SuperficieBezier`.

Dada a estrutura do ficheiro de entrada apresentada em cima, a função pode ser resumida ao seguinte pseudo-código:

```
SuperficieBezier leBezier(char *fich_in) {
    SuperficieBezier superficie;

    getline(fich_in, linha);
    nPatches = stoi(linha);

    Para todos os patches:
        getline(fich_in, linha);
        Partir linha pela vírgula (,)
        Guardar todos os indíces em vector<int> patch
        Guardar patch em superficie

    getline(fich_in, linha);
    nPtsControlo = stoi(linha);

    Para todos os pontos de controlo:
        std::getline(fich_inp, linha);
        Partir linha pela vírgula (,)
        Guardar coordenadas x, y e z num Coordenadas3D
        Guardar ponto de controlo em superficie
    return superficie;
}
```

3.3.3 Método de obtenção dos triângulos

Depois de lido o ficheiro de entrada e de carregados todos os dados necessários para memória, pode-se proceder à implementação de um método cujo objetivo é o de criar os pontos de todos os triângulos e de os escrever num ficheiro capaz de ser lido pelo programa Motor. É nesta tarefa que se torna importante o método `getTriangulos` da classe `SuperficieBezier`, já enuciada acima.

O método `getTriangulos` retorna um `vector<Coordenadas3D>` com todos os pontos dos triângulos a desenhar. Este método é invocado pelo Gerador com os níveis de *tessellation* passados como parâmetro (doravante designados como divisões em u, `divsU`, e divisões em v, `divsV`). Por sua vez, o Gerador imprime todos os pontos no ficheiro de saída.

A forma como são obtidas as coordenadas dos pontos dos triângulos segue a introdução feita na secção 3.2 deste capítulo e pode ser resumida no seguinte pseudo-código:

```
std::vector<Coordenadas3D> getTriangulos(int divsU,int divsV) {
    std::vector<Coordenadas3D> res;
    float M[4][4] = { { -1.0f,  3.0f, -3.0f,  1.0f },
                      {  3.0f, -6.0f,  3.0f,  0.0f },
                      { -3.0f,  3.0f,  0.0f,  0.0f },
                      {  1.0f,  0.0f,  0.0f,  0.0f } };
    float Mt[4][4] = { { -1.0f,  3.0f, -3.0f,  1.0f },
                       {  3.0f, -6.0f,  3.0f,  0.0f },
                       { -3.0f,  3.0f,  0.0f,  0.0f },
                       {  1.0f,  0.0f,  0.0f,  0.0f } };

    Para todos patches da superficieBezier:
        // Construir matriz 4x4 com todos os pontos
        Coordenadas3D P[4][4];
        for (i = 0; i < 4; i++)
            for (j = 0; j < 4; j++)
                // em patch estão os indices dos pontos
                P[i][j] = pontosControlo[patch[i*4+j]];
        Fazer Mr = M x P
        Fazer R = Mr x Mt
        // R é uma matriz de Coordenadas3D

        float deltaU = (float) 1.0f / divsU;
        float deltaV = (float) 1.0f / divsV;
        for (i = 0; i < divsU; i++) {
            float u1 = i * deltaU;
            float u2 = (i + 1) * deltaU;
            for (j = 0; j < divsV; j++) {
                float v1 = j * deltaV;
                float v2 = (j + 1) * deltaV;
                Coordenadas3D a = calculaB(R, u1, v1);
                Coordenadas3D b = calculaB(R, u2, v1);
                Coordenadas3D c = calculaB(R, u1, v2);
                Coordenadas3D d = calculaB(R, u2, v2);

                Guardar em res pela ordem:
                    a - c - b - b - c - d
            }
        }
}
```

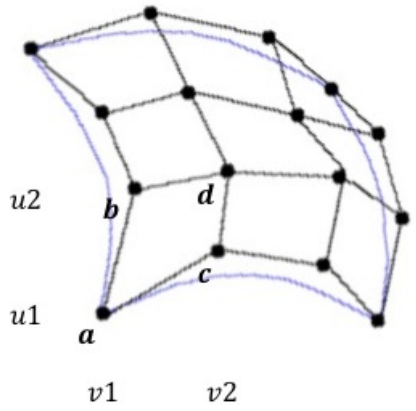

Figura 3.4: Dispozição dos pontos a, b, c e d e valores de $u1$, $u2$, $v1$ e $v2$ (adaptado de cs.helsinki.fi)

Neste método são usadas várias funções de multiplicação de matrizes que, para simplificação, decidiu-se omitir do pseudo-código. No entanto, uma delas será mais facilmente compreendida se for analisada em separado. Refere-se à função `calculaB` que recebe como parâmetros uma matriz 4×4 de `Coordenadas3D`, um valor de u e um valor de v.

Olhando para a expressão 3.1, na secção 3.2 deste capítulo, e também para o método `getTriangulos` apresentado em pseudo-código, pode-se notar que, até ao momento em que são instanciados os objetos `a`, `b`, `c` e `d`, a matriz R contém o resultado de:

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

faltando calcular:

$$\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} R \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Dados os valores de u e de v, que são calculados em ciclo no método `getTriangulos` como `u1`, `u2`, `v1` e `v2` e passados como argumento à função `calculaB`, conforme a localização de cada ponto (Figura 3.4), tem-se:

```
Coordenadas3D calculaB(Coordenadas3D R[4][4],float u,float v) {
    Coordenadas3D rf;
    Coordenadas3D Ru[4];
    Calcular vu[4] = { u^3, u^2, u, 1 };
    Calcular vv[4] = { v^3, v^2, v, 1 };
    Fazer Ru = vu x R
    Fazer rf = Ru x vv
    return rf;
}
```

Ficam assim calculados todos os pontos dos triângulos e imprimidos no ficheiro de saída.

3.4 Teste ao resultado do Gerador

De seguida pode ser visto o resultado que se obteve ao passar o ficheiro com os triangulos que o Gerador produziu ao receber como argumento o ficheiro com *patches* do Bule de chá disponibilizados pelo docente da UC.

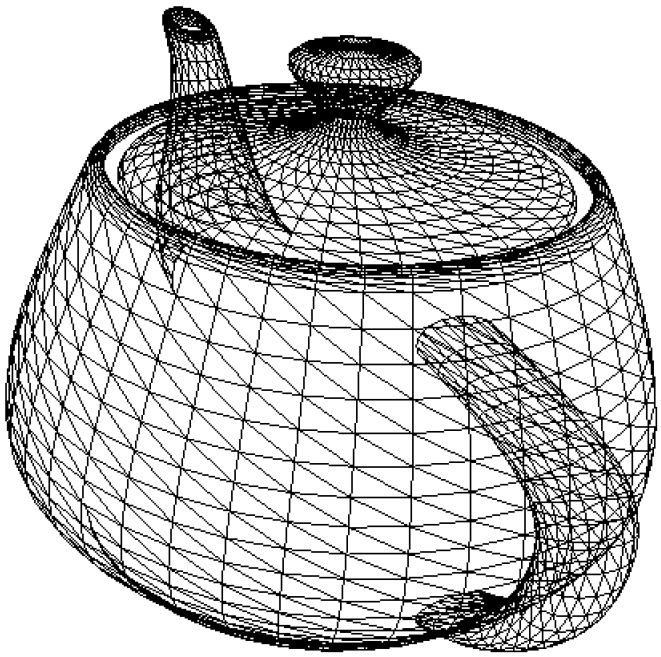

Figura 3.5: Bule de chá, desenhado a partir de superfícies de Bézier

4. Motor

Nesta 3º fase do trabalho prático, o motor foi alterado por forma a suportar funcionalidades tais como transformações em função do tempo e modelos desenhados com recurso a *Vertex Buffer Object*. Ao longo deste capítulo será explicada a implementação adotada e apresentadas as classes desenvolvidas que suportam estas novas funcionalidades.

4.1 Tranformações

O motor foi melhorado com o objetivo de suportar transformações em função do tempo em oposição às estáticas definidas em fases anteriores. As transformações entendidas foram a translação e rotação.

4.1.1 Translação

As novas translações que o motor é capaz de executar são feitas num dado tempo e ao longo de um conjunto de pontos providenciados por um curva cúbica de Catmull-Rom. De seguida apresenta-se um exemplo em XML de uma translação deste tipo.

```
<translate time=10 >
    <point X=1 Y=0 Z=1 />
    <point X=0.707 Y=0.707 Z=1 />
    <point X=0 Y=1 Z=1 />
    <point X=-1 Y=0 Z=1 />
</translate>
```

Curva Catmull-Rom

Dado um conjunto de pontos, uma curva Catmull-Rom é uma curva cúbica suave que passa por todos eles. Devido à definição da curva Catmull-Rom é sempre necessário um ponto inicial antes do segmento inicial da curva e um número mínimo de 4 pontos.

Este tipo de curva recorre à interpolação e é formada a partir de uma sequência de curvas de Hermite, cujas tangentes são calculadas automaticamente a partir dos pontos de controlo . Deste modo, é traçada uma curva de Hermite a cada par de pontos, calculada a derivada com o ponto anterior e com o ponto seguinte ao ponto atual, obtendo-se uma curva de Catmull-Rom.

Dados 4 pontos P_0, P_1, P_2 e P_3 e um valor de t entre 0 e 1, é calculada a curva entre os pontos P_1 e P_2 variando o valor de t com um dado incremento.

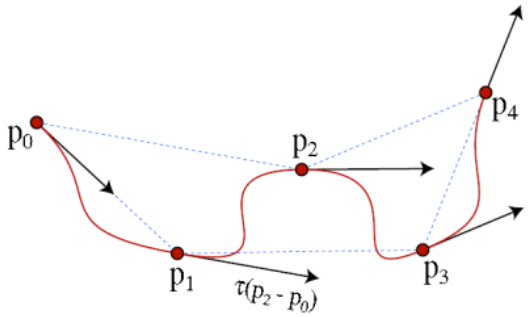

Figura 4.1: Pontos da Curva de Catmull-Rom

Dado um conjunto de pontos de controlo $\{P_0, P_1, .., P_n\}$, queremos que a curva passe por todos os pontos. Assim, a curva de Catmull-Rom é construída com segmentos de curvas de Hermite para cada par (P_i, P_{i+1}) de pontos de controlo.

Deste modo, a curva de de Catmull-Rom é construída tendo em conta a seguinte formulação matemática:

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

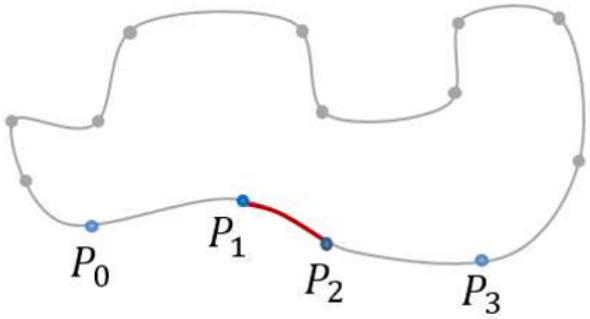

Figura 4.2: Curva de Catmull-Rom

Implementação

Feita uma introdução a curva de Catmull-Rom passa-se a explicar a forma como foi implementado em código o processo de criação e desenho da curva. Foi criado um conjunto de funções que permitem a criação desta tipo de curva e que encontra em `CatmullRom.h`.

A função `getGlobalCatmullRomPoint` recebe vetor dos pontos de controlo da curva `(p)`, o *global time* `(gt)` e calcula o ponto da curva e a respetiva derivada. O

calculo do ponto e da derivada é feita com recurso à função `getCatmullRomPoint`, de acordo com o seguinte algoritmo:

```
int point_count = p.size();
float t = gt * point_count;
int index = floor(t);
t = t - index;
int indices[4];

indices[0] = (index + point_count - 1) % point_count;
indices[1] = (indices[0] + 1) % point_count;
indices[2] = (indices[1] + 1) % point_count;
indices[3] = (indices[2] + 1) % point_count;
```

A função `getCatmullRomPoint` calcula o ponto recorrendo a formula apresentada anteriormente e a derivada de forma semelhante. Considerando M a matriz dessa fórmula e T o vetor $\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$, o pseudo-código desta função pode ser representado da seguinte forma:

```
void getCatmullRomPoint(float t, Coordenadas3D p0,
                        Coordenadas3D p1, Coordenadas3D p2,
                        Coordenadas3D p3, float *res,
                        float *deriv) {

    float Ax[4], Ay[4], Az[4];
    float Px[4], Py[4], Pz[4];

    Multiplica a matriz M por cada vetor P(x,y,z) e
    o resultado é guardado num vetor A (x,y,z), respetivamente;

    Multiplica o vetor T por cada vetor A (x,y,z) e
    constrói-se o Ponto a devolver (res);

    Deriva-se o vetor T obtendo-se o vetor T';

    Multiplica o vetor T' por cada vetor A (x,y,z) e
    constrói-se a derivada (deriv);

}
```

Deste modo, na função `XMLtoGrupo` foram feitas alterações de forma a suportar este tipo de translação. Assim, no caso de tratar de uma translação animada a função executa o seguinte pseudo-código:

```
Grupo XMLtoGrupo(xml_node node) {
    Para cada nodo:
        Se node_name for "translate" {
            Para cada atributo {
                Se name for "time" {
                    Constrói vetor com pontos de controlo
                    Verifica direção e desenho da curva
                    Se desenho curva for true {
                        Recolhe n.º pontos controlo a desenhar
                         se fornecido;
                        Calcula variação do t
                        Para o nº pontos controlo {
                            Invoca getGlobalCatmullRomPoint;
                            Guarda ponto resultado
                            no vector do desenho;
                        }
                        Guarda a translação construida
                        no vector catmullDes do Grupo;
                    }
                } Se não:
                    Constrói translação estática;
            }
        }
}
```

A função de desenho `desenhaGrupo` também teve de ser alterada de forma a poder suportar este tipo de translação.

Como se trata de uma translação animada num dado tempo foi necessário garantir que ocorria nesse tempo. Isso foi possível através do seguinte raciocínio:

- Guardar o tempo inicial;
- Calcular o tempo que passou desde o inicio do desenho;
- Calcular t de acordo com a seguinte fórmula:

$$t = \frac{(tempo\ passado - tempo\ inicial)\ \%\ tempo\ translacao}{tempo\ translacao}$$

O pseudo-código que permite o desenho de translações animadas é o seguinte:

```
void desenhaGrupo(tree<Grupo>::iterator it_grupo) {
    Para cada transformação {
        Caso Translação {
            if (time ==-1) {
                Translação estática;
            } else {
                Para cada Desenho do vector catmullDes:
                    Desenha pontos de acordo com definições;
                Calcula o t;
                Verifica direção;
                Invoca a função getGlobalCatmullRomPoint;
                Constrói matriz de rotação;
                Efetua translação;
            }
        }
    }
}
```

4.1.2 Rotação

Para além da translação dinâmica, nesta fase do trabalho, também iremos ter rotações dinâmicas. Este tipo de rotação deve ter o seguinte formato no ficheiro XML:

```
<rotate time=10 axisX=0 axisY=1 axisZ=0 />
```

Como se pode observar, comparativamente à rotação estática, o ângulo foi substituído pelo atributo *time*, isto é, passou a ser fornecido o número de segundos para executar uma rotação completa de 360^o graus em torno do eixo especificado.

Implementação

Por forma a suportar este tipo de rotação foram feitas adaptações as funções de leitura e desenho que passam a ser explicadas de seguida.

À função `XMLtoGrupo` apenas foi adicionada uma verificação no caso de a transformação se tratar de uma rotação. Essa verificação passa por ver se possui o atributo *time* e em caso afirmativo verifica ainda a direção de rotação.

Como se trata e uma rotação animada foi necessário garantir, à semelhança da translação animada, que era efetuada no tempo fornecido. O raciocínio foi o mesmo apenas foi adicionado o calculo do ângulo de rotação tendo em conta o `t` : $ang = 360.0f * t$. Deste modo, na função `desenhaGrupo` é efetuada o calculo do ângulo conforme descrito anteriormente, verificada a direção de rotação e efetuada a rotação tendo em conta estes dois parâmetros.

4.2 Vertex Buffer Objects (VBO's)

Um dos objetivos para esta fase do projeto como forma de melhorar o desempenho do motor era o de implementar o desenho dos objetos através de Vertex Buffer Objects (VBO's), substituindo o desenho em modo imediato das fases anteriores. Os VBO's consistem em buffers de memória da placa gráfica que podem conter informação diversa, nomeadamente informação relativa a coordenadas de pontos. Visto que se trata de memória na placa gráfica, é possível aceder à informação mais rapidamente do que no desenho em modo imediato em que a informação dos pontos se encontrava em objetos instâncias da classe `Coordenadas3D`, tratando-se por isso de objetos em memória RAM.

Em primeiro lugar, foi necessário ativar a escrita e uso destes buffers por parte da aplicação através da função `glEnableClientState`:

```
glEnableClientState(GL_VERTEX_ARRAY);
```

Depois da ativação foi necessário preencher a informação das coordenadas dos pontos nos buffers. A leitura dos pontos faz-se na função `XMLtoGrupo` que é responsável por passar a informação contida num elemento `<group>` XML para uma instância da classe `Grupo`. Conforme visto nas secções 2.5 e 2.4 um grupo consiste num conjunto de desenhos em que cada desenho por sua vez contém um conjunto de pontos. Para cada ficheiro de pontos no XML é criado um `Desenho` que contém os pontos do ficheiro assim como as definições de como esses pontos serão desenhados de acordo com o seguinte código:

```
Grupo XMLtoGrupo(xml_node node){
    Grupo res;

    Processa transformações....

    Para cada <model> em <models>:
        Desenho desenho;
        Processa definições de desenho...
        Para cada ponto no ficheiro do model:
            //Adiciona ponto ao desenho
            desenho.pontos.push_back(Coordenadas3D{ x,y,z });
        Adiciona desenho ao grupo resultado

    return res;
}
```

Este pseudo-código corresponde à forma de ler ficheiros e de colocar a informação em instâncias de objetos em memória RAM das fases anteriores. Note-se que no final de cada iteração do ciclo que percorre os modelos, o vector `desenho.pontos` contém todos os pontos de um modelo. Para tirar partido dos VBO's é necessário colocar agora a informação dos pontos deste vector nos buffers do OpenGL. Para isso, depois de ter os pontos no vector `desenho.pontos` pede-se ao OpenGl para que seja gerado um novo buffer:

```
GLuint nBuffer[1];
glGenBuffers(1, nBuffer);
```

Isto faz com que um buffer seja gerado e o identificador para esse buffer seja colocado em `nBuffer[0]`. Este identificador é necessário para indicar em cada momento de que buffer se quer ler/escrever. A seguir à criação do buffer, antes de escrever nele precisamos de o ativar, o que é feito com a seguinte instrução:

```
glBindBuffer(GL_ARRAY_BUFFER, nBuffer[0]);
```

O buffer está agora pronto a ser escrito. Para escrever no buffer usa-se a função `glBufferData` passando o endereço do vector de pontos e o seu tamanho:

```
glBufferData(GL_ARRAY_BUFFER,
                sizeof(float) * desenho.pontos.size() * 3,
                &desenho.pontos[0],
                GL_STATIC_DRAW);
```

A informação dos pontos no vector `desenho.pontos` fica desta forma copiada para um buffer em memória da placa gráfica. Isto implica que a informação no vector `desenho.pontos` deixa de ser necessária para o desenho dos pontos, podendo eventualmente ser eliminada. Neste trabalho optou-se por não eliminar esta informação uma vez que se pretende dar a opção de desenhar também em modo imediato, conforme se irá ver no capítulo seguinte.

O último passo corresponde a guardar no desenho o identificador do buffer que o `OpenGl` atribuiu. Isto serve para que na função de desenho do motor, para cada `Desenho` de cada `Grupo` seja sempre possível saber que buffer se deve activar para desenhar os pontos correspondentes a esse desenho:

```
desenho.nBuff = nBuffer[0];
```

O pseudo-código completo fica por isso:

```
Grupo XMLtoGrupo(xml_node node){
    Grupo res;

    Processa transformações....

    Para cada <model> em <models>:
        Desenho desenho;
        Processa definições de desenho...
        Para cada ponto no ficheiro do model:
            //Adiciona ponto ao desenho
            desenho.pontos.push_back(Coordenadas3D{ x,y,z });

        GLuint nBuffer[1];
        glGenBuffers(1, nBuffer);
        glBindBuffer(GL_ARRAY_BUFFER, nBuffer[0]);
        glBufferData(GL_ARRAY_BUFFER,
                        sizeof(float) * desenho.pontos.size() * 3,
                        &desenho.pontos[0],
                        GL_STATIC_DRAW);
        desenho.nBuff = nBuffer[0];
```

```
        //Adiciona desenho ao grupo resultado
        res.desenhos.push_back(desenho);

    return res;
}
```

Até aqui vimos como se pode activar e escrever informação nos buffers do `OpenGl`. Para tirar partido destes buffers é necessário agora ler a informação dos pontos que eles contêm e desenhar esses pontos. Recorde-se que a informação do desenho da cena está numa árvore de grupos, sendo a função `desenhaGrupo` responsável pelo desenho de cada grupo.

```
void desenhaGrupo(Grupo grupo) {
    Processa transformações do grupo...
    Para cada desenho do grupo:
        Define PolygonMode e cor de desenho...
        Se modo imediato:
            Desenha pontos em modo imediato
        else:
            //Desenho com VBO's
            glBindBuffer(GL_ARRAY_BUFFER, desenho->nBuff);
            glVertexPointer(3, GL_FLOAT, 0, 0);
            glDrawArrays(desenho->defsDesenho.modoDesenho, 0,
            desenho->pontos.size());
}
```

Com a chamada à função `glBindBuffer` ativa-se o buffer que contém os pontos que se pretende desenhar:

```
glBindBuffer(GL_ARRAY_BUFFER, desenho->nBuff);
```

Com a chamada à função `glVertexPointer` indica-se o formato da informação de cada vértice, neste caso, diz-se que cada vértice corresponde a 3 float's consecutivos do buffer:

```
glVertexPointer(3, GL_FLOAT, 0, 0);
```

Por fim, pede-se que os pontos do buffer sejam desenhados, de acordo com o modo de desenho indicado nas definições do desenho:

```
glDrawArrays(desenho->defsDesenho.modoDesenho, 0,
                desenho->pontos.size());
```

5. Interação com o utilizador

5.1 Teclado

As opções de interação do utilizador com o motor com recurso ao teclado não foram alteradas relativamente à fase 2.

5.2 Rato

Relativamente à fase 2, apenas se adicionou um novo menu ao botão direito do rato que permite ao utilizador escolher entre o desenho em modo imediato ou com recurso a VBO's:

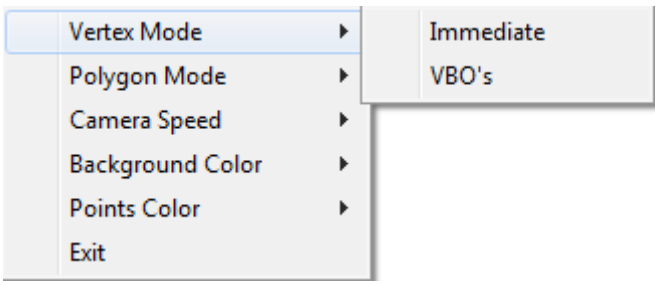

Figura 5.1: Menu do botão direito do rato com nova opção `VertexMode` que permite escolher entre o desenho em modo imediato ou com VBO's

Internamente no motor, o modo de desenho atualmente selecionado é dado por uma variável global:

```
bool modoImediato;
```

Esta variável indica que o modo desenho deve ser feito de modo imediato quando toma o valor `true` e com VBO's quando toma o valor `false`. Como o objetivo desta fase do trabalho é evidenciar o uso de VBO's, quando o programa inicia esta variável toma o valor `false`.

Quando o utilizador escolhe uma das opções do menu, é chamada a função `vertexMode_menu_func` que é responsável por atualizar o valor da variável em conformidade:

```
void vertexMode_menu_func(int opt) {
        switch (opt) {
        case 1:  modoImediato = true; break;
        case 2:  modoImediato = false; break;
        }
```

```
        glutPostRedisplay();
}
```

Para indicar que as opções do sub-menu `Vertex Mode` estão associadas à função `vertexMode_menu_func` e para a criação do sub-menu `Vertex Mode` no menu do botão direito foi acrescentado o seguinte código à função `criaMenus()`:

```
void criaMenus() {

    Criacao de outros sub-menus...

    //Associa opcoes de Vertex Mode a funcao
       vertexMode_menu_func
    int vertexMode_menu = glutCreateMenu(vertexMode_menu_func);
    glutAddMenuEntry("Immediate", 1);
    glutAddMenuEntry("VBO's", 2);

    //Associa Vertex Mode ao menu do botao direito
    int main_menu = glutCreateMenu(main_menu_func);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    glutAddSubMenu("Vertex Mode", vertexMode_menu);

    Associacao de outros sub-menus...
}
```

6. Construção do Sistema Solar

Para a construção da representação do Sistema Solar foi, tal como nas fases anteriores, escrito um ficheiro em formato XML, com a formatação e a estrutura apresentadas na secção 1 deste relatório. No entanto, existem algumas diferenças relativamente às fases anteriores, as quais serão explicadas ao longo deste capítulo.

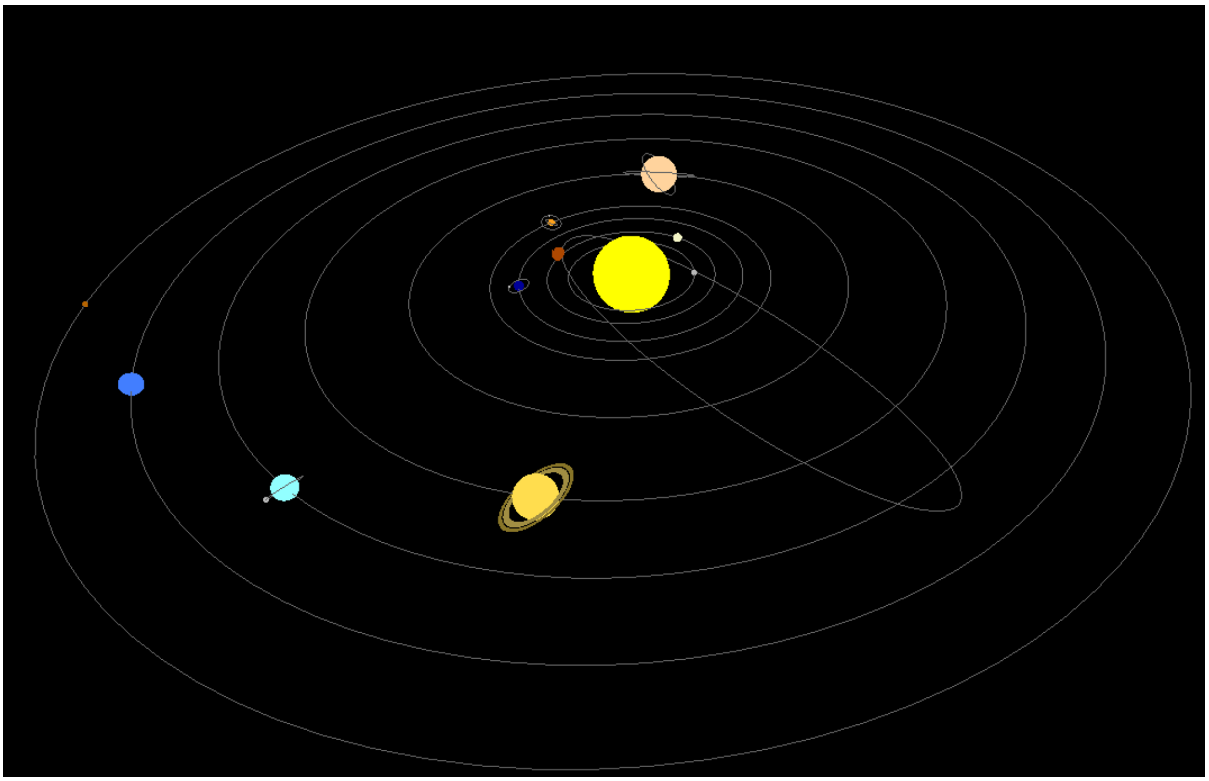

Figura 6.1: Representação geral do Sistema Solar

6.1 Figuras usadas na representação

6.1.1 Esferas - diminuição do número de pontos

A figura principal usada foi uma esfera, no entanto, contrariamente às fases anteriores usaram-se diferentes esferas para desenhar o sol, os planetas e as luas.
Para desenhar o sol, e por se tratar da maior estrutura do Sistema Solar, usou-se uma esfera com raio 1 e com 50 fatias e 50 camadas. Para desenhar os planetas usou-se uma esfera com raio 1, 25 fatias e 25 camadas, já que estes possuem um tamanho significativamente menor relativamente ao Sol. Por outro lado, para desenhar as luas,

uma vez que estas constituem as estruturas mais pequenas que se desenham com esferas, usou-se uma esfera com raio 1, 10 fatias e 10 camadas.
Esta decisão prendeu-se com o facto de otimizar o processo desenho, atribuindo mais pontos às figuras maiores, para que estas fiquem mais uniformes e menos pontos a estruturas mais pequenas, que não precisam do mesmo número de pontos para se obter uma estrutura uniforme.

6.2 Órbitas como curvas de Catmull-Rom

Na fase anterior as órbitas dos planetas foram desenhadas recorrendo à figura anel. Esta constitui outra das diferenças relativamente à fase anterior, pois agora as óbitas dos planetas são desenhadas recorrendo a curvas de Catmull-Rom.
Como já foi dito na secção 4.1.1, uma curva Catmull-Rom é uma curva cúbica suave que passa por um dado conjunto de pontos. Com base neste pressuposto, conforme o tamanho da órbita a desenhar, decidiu-se que o número de pontos fornecidos para desenhar a curva vai variar de modo a obter-se sempre uma curva o mais perfeita possível. Assim, para os planetas interiores, com órbitas mais pequenas, foram fornecidos 15 pontos de controlo, e para as suas luas foram fornecidos 8 pontos de controlo. Para os planetas exteriores, com órbitas maiores, foram fornecidos 20 pontos de controlo para desenhar as suas órbitas e 12 pontos de controlo para as órbitas das suas luas.

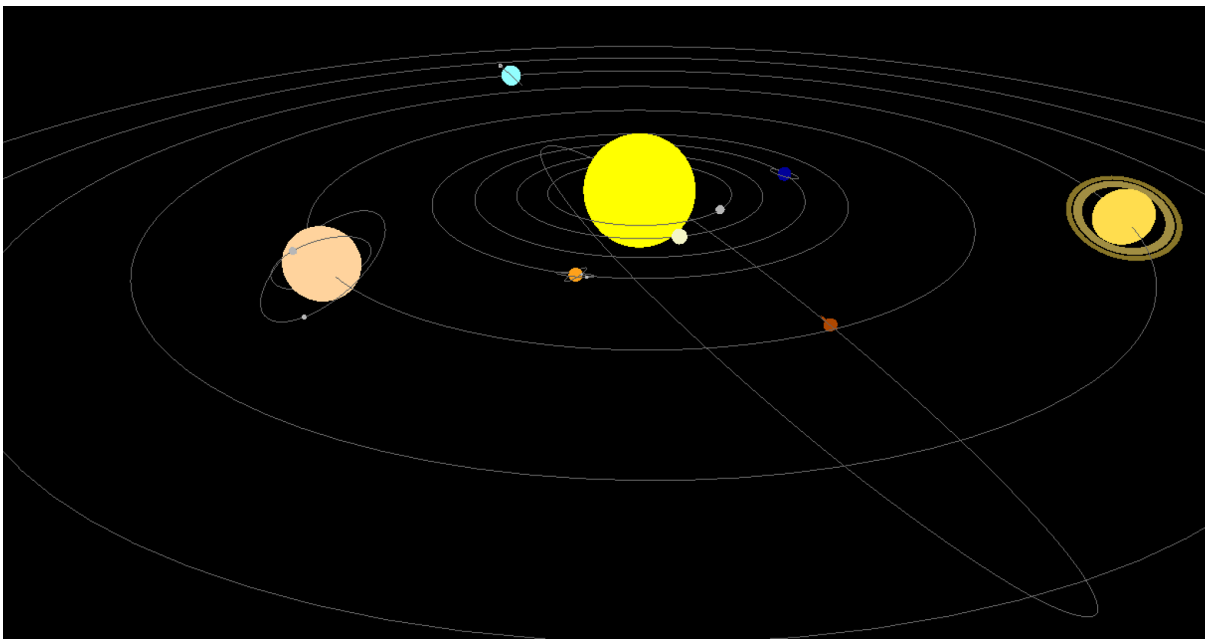

Figura 6.2: Órbitas desenhadas usando curvas de Catmull-Rom

6.2.1 Obtenção dos pontos de controlo

Para obter os pontos de controlo que vão ser usados para desenhar as órbitas com curvas de Catmull-Rom usou-se um algoritmo que tem em conta a distância do planeta ao sol, ou da lua ao seu planeta pai, e aumentando um $deltaAng = 2 * M\_PI/nPontos$ obtém-se através das fórmulas das coordenadas esféricas todos os pontos pretendidos.

6.3 Sistema Planeta-Luas

Na fase anterior considerou-se a noção de descendência entre os astros: o Sol era o "pai" de todos os astros do Sistema Solar e as luas eram "filhas" dos planetas respetivos. Nesta fase, essa abordagem foi modificada, o nodo "pai" agora é o Sistema Solar, o qual tem como filhos o Sol, os planetas sem luas, os Sistemas plantea-luas e o cometa. O sistema Planeta-Luas representa o conjunto do planeta com as suas respetivas luas, onde o planeta e as luas são nodos "irmãos" e ambos filhos de um nodo `Planet-System` (`Planet` corresponde ao nome do planeta respetivo). Neste sistema existe a translação animada que vai ser herdada por ambos, planeta e luas, fazendo com que os mesmos se movam segundo a mesma órbita de translação. A decisão de implementar estes nodos como "irmãos" prendeu-se com o facto de não se pretender que as luas herdem a rotação animada do seu respetivo planeta, nem a escala aplicada ao seu planeta, já que a mesma ia influenciar o desenho dos pontos para sua própria órbita de translação.

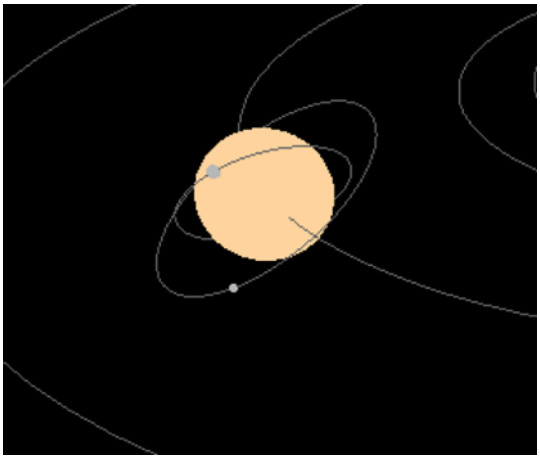

Figura 6.3: `Jupiter_system` - Representação do planeta Júpiter com as suas luas

6.4 Cometa

O cometa também é "filho" do Sistema Solar, e de modo a criar a representação do cometa o mais próximo possível da realidade, inicialmente faz-se uma translação e uma rotação no eixo dos $0x0$. De seguida, procede-se à translação animada do cometa, fornecendo 10 pontos de controlo para o desenho da curva de Catmull-Rom, sendo a obtenção destes pontos similar à das restantes órbitas, que já foi explicada em cima.

```
<group name="Cometa">
  <translate X="0" Y="-5" Z="5"/>
  <rotate angle="40" X="1" Y="0" Z="0"/>
  <translate time="20.0" drawCatmullCurve="true"
             nrCatmullPointsToDraw="100">
     <point X="0.0" Y="0" Z="9.0"/>
     <point X="2.9389262614623" Y="0" Z="7.28115294937452"/>
     <point X="4.7552825814757" Y="0" Z="2.78115294937452"/>
```

```
      <point X="4.7552825814757" Y="0" Z="-2.78115294937452"/>
      <point X="2.93892626146236" Y="0" Z="-7.2811529493745"/>
      <point X="6.1232339957367e-16" Y="0" Z="-9.0"/>
      <point X="-2.9389262614623" Y="0" Z="-7.2811529493745"/>
      <point X="-4.7552825814757" Y="0" Z="-2.7811529493745"/>
      <point X="-4.7552825814757" Y="0" Z="2.78115294937452"/>
      <point X="-2.9389262614623" Y="0Z="7.281152949374526"/>
  </translate>
  <scale uniform="0.1"/>
  <models>
      <model red="0.67843137254901" green="0.28235294117647"
            blue="0.0509803921568627" file="bezier.3d"/>
  </models>
</group>
```

Como se pode verificar, o ficheiro usado para desenhar a figura cometa foi o `bezier.3d`, que corresponde ao ficheiro de pontos obtido a partir da construção das superfícies de benzier, a qual teve como ficheiro de input o ficheiro disponibilizado pelo enunciado, e por isso o cometa é simbolicamente representado por um “teapot”, como se pode observar na figura 6.4.

Figura 6.4: Teapot representativo do cometa no Sistema Solar

Conclusão

Os objetivos propostos para esta fase do trabalho foram cumpridos. O gerador foi alterado de forma a poder gerar triângulos de acordo com *patches* de Bezier definidos num ficheiro. Tal implicou a implementação de um novo algoritmo de geração de pontos assim como a inclusão de um novo comando `bezier`. No que diz respeito ao motor, foi acrescentada a capacidade de desenhar objetos com recurso a VBO's e a capacidade de apresentar objetos com translações e rotações animadas definidas num intervalo de tempo. A utilização de VBO's permitiu que os pontos pudessem ser colocados na memória da placa gráfica em detrimento da memória RAM, o que melhorou significativamente a performance do motor. Destacam-se ainda as translações animadas, cuja implementação envolveu a utilização de curvas Catmull-Rom e respetivos algoritmos. O uso de translações e rotações animadas permite que o motor seja agora capaz de representar cenas mais dinâmicas.

Como funcionalidades adicionais aos requisitos inicialmente propostos para esta fase, destacam-se os atributos XML extra que foram implementados e que permitem um maior controlo da cena. O atributo `direction` permite controlar a direção das translações e rotações enquanto que os atributos `drawCatmullCurve` e `nrCatmullPointsToDraw` permitem a visualização das curvas de Catmull-Rom das translações com diferentes níveis de detalhe. A possibilidade de controlar a direcção das rotações foi útil para representar os diferentes movimentos de rotação dos planetas na cena do sistema solar e o desenho das curvas de catmull-rom proporcionou uma forma alternativa de representar as órbitas dos planetas com recurso a linhas fechadas (recorde-se que na fase 2 estas eram desenhadas com recursos a anéis do gerador).

Como melhorias ao trabalho, destaca-se uma possível reorganização do código do motor. O código do motor tem sido essencialmente mantido apenas num ficheiro desde a fase 1, no entanto tem vindo a crescer significativamente. Nesse sentido, para tornar o código mais navegável e mais fácil de ler e entender, sugere-se um *refactoring* ao código do motor em que o código seja dividido por mais ficheiros e por mais funções, visto que algumas funções estão a tornar-se excessivamente grandes. Relativamente ao gerador, poderia ser acrescentada a funcionalidade de gerar as figuras através de pontos e índices de pontos. Em conjunto com os VBO's tal poderia melhorar ainda mais a performance do motor, além de uma potencial redução da memória necessária.