Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Relatório do projeto

Fase 2 - Transformações Geométricas

Computação Gráfica

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

Braga, 29 de Abril de 2017

Conteúdo

Resumo

O presente relatório documenta a 2.ª fase do trabalho prático da Unidade Curricular de Computação Gráfica que consistiu na expansão das capacidades do motor gráfico desenvolvidas na 1ª fase do trabalho. Nesta fase, o XML de especificação da cena foi melhorado de forma a permitir translações, rotações e escalas de objetos que podem estar definidos numa hierarquia. Além destas transformações é ainda possível indicar definições de desenho no XML como a cor dos pontos e o modo de desenho dos polígonos através do XML. O XML nesta fase permite por isso que se construam cenas mais complexas. Tal levou à necessidade de implementar uma câmara em primeira pessoa para permitir ao utilizador visualizar toda a cena (eventualmente complexa) de uma forma fácil. Como demonstração das potencialidades do motor, foi construída uma cena de exemplo que representa o sistema solar.

Introdução

O relatório está organizado em 7 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 em suportar toda a informação necessária para o processo de desenho.

O capítulo 3 descreve o processo de leitura, nomeadamente como é feita a leitura do XML e como a informação é armazenada em estruturas.

O capítulo 4 prossegue com a exposição do ciclo de *Rendering*, ou seja, como são usadas as estruturas de dados na construção das "cenas".

No capítulo 5 é explicitado todo o processo de implementação da câmara. Trata-se de uma câmara em primeira pessoa (*First Person Camera*) desenvolvida com o objetivo de o utilizador se mover ao longo do espaço por forma a poder visualizar melhor o que pretende.

No capítulo 6 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 7, é explicado como foi construido a cena de exemplo correspondente ao sistema solar, desde as figuras usadas até às medidas tomadas assim como outras considerações.

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. 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">
    ...
    Pontos e transformacoes
    ...
    </group>
</scene>
```

1.1.3 Nodo *<translate>*

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

- **X** - Componente x do vetor da translação. O valor deste atributo deverá ser uma *string* correspondente a um número que possa ser convertido para *float*. Não é um atributo obrigatório.
- **Y** - Componente y do vetor da translação. 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. 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, pelo menos um dos atributos acima deve ser indicado, não sendo obrigatório ter os 3 atributos. Quando um dos atributos não é indicado, é assumido o valor 0 para a translação no respetivo 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">
        <translate X="2.4"/>
        ...
        Grupos e transformacoes
        ...
    </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".

1.1.4 Nodo *<rotate>*

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

- ***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.
- **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*. 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*. 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*. Não é um atributo obrigatório.

Embora o motor não o force, para efetuar uma rotação deve ser sempre indicado o ângulo. Os restantes atributos que definem o vetor sobre o qual será feita a rotação não são obrigatórios, embora pelo menos um deva ser indicado para a rotação ter sentido. Quando um dos atributos relativo às componentes do vetor não é indicado, é assumido o valor 0 para esse atributo.

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">
        <rotate angle="30" axisX="0" axisY="0" axisZ="1"/>
        ...
        Grupos e transformacoes
        ...
    </group>
</scene>
```

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

1.1.5 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 e 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"/>
```

1.1.6 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>
```

1.1.7 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 {
    float x, y, z;
    Coordenadas3D operator+(const Coordenadas3D& p2);
    Coordenadas3D operator-(const Coordenadas3D& p2);
    Coordenadas3D crossproduct(const Coordenadas3D& p2);
    Coordenadas3D times(float k);
};
```

Esta classe corresponde à classe `Ponto3D` da 1ª fase do projeto. A mudança de nome da classe nesta fase justifica-se pelo facto de haver interesse em representar tanto pontos como vetores usando coordenadas x, y e z e poder fazer operações tanto com pontos como vetores sem distinção (e.g somar pontos com vetores) pelo que o nome `Ponto3D` deixou de ser completamente descritivo da funcionalidade da classe.

2.2 Transformações

Conforme visto na secção 1.1, cada grupo pode ter 3 tipos de transformação: translações, rotações e escalas. Para cada uma destas transformações foi criada uma classe cujas variáveis de instância são os parâmetros de cada transformação.

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

class Translacao {
public:
    float tx, ty, tz; };
```

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

Para representar uma transformação em termos gerais foi criada a classe `Transformacao` que consiste numa `union` das 3 classes apresentadas acima.

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

        UTr() {memset(this, 0, sizeof(UTr));}
    } 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 transformação sem ter que se consultar a union, o que facilita a leitura. O tipo `TipoTransformacao` corresponde a uma enumeração dos tipos de transformações referidos anteriormente.

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

2.3 Definições Desenho

Como foi visto na secção 1.1, aos pontos dos ficheiros de cada elemento "*model*" podem ser aplicadas algumas definições de desenho, nomeadamente cores e modo de preenchimento de polígonos. Em termos de código, estas definições foram captadas pela classe `DefsDesenho` que representa as definições de desenho que são definidas em cada elemento `<model>`.

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

Esta classe contém os `float`'s necessários à representação de uma cor no sistema *RGB* assim como o modo de desenho representado por uma variável do tipo `GLenum`. O tipo destas variáveis foram escolhidos de forma a permitir que as variáveis possam ser passadas diretamente às funções do OpenGl responsáveis por aplicar estas definições, nomeadamente a função `glColor3f()` para a definição das cores e a função `glPolygonMode()` para a definição do modo de desenho.

2.4 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;
    vector<pair<DefsDesenho,vector<Coordenadas3D> >> pontos;
};
```

- **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.

- **pontos** - Este `vector` contém, sob a forma de pares, os pontos do grupo a ser desenhados assim como as definições que lhes vão ser aplicados. Cada par `pair<DefsDesenho,vector<Coordenadas3D>` representa um elemento `<model>`. O primeiro elemento do par (`DefsDesenho`) representa as definições do modelo, dado pelos atributos `red`, `green`, `blue` e `mode`. O segundo elemento (`vector<Coordenadas3D>`) contém todos os pontos presentes no ficheiro indicado pelo atributo `file` do elemento `<model>`. Estes pares por sua vez foram agrupados também num `vector`, uma vez que é permitido que um determinado grupo tenha vários elementos `<model>` e para cada um é necessário armazenar os seus pontos e definições de desenho. A ordem pela qual os pares são guardados no `vector` é a mesma pela qual os elementos `<model>` aparecem no XML.

2.5 Árvore de grupos

O Motor tem como função ler o ficheiro XML e guardar os dados lidos em memória para que possam ser lidos de forma eficiente pela função `renderScene()`. Além de ter os dados em memória, é conveniente ter os dados guardados com os tipos de dados corretos, prontos a ser usados pela função `renderScene()`. Como será detalhado na secção seguinte, a biblioteca `pugixml` guarda em memória dos dados do XML em forma de árvore. A informação do XML conforme lida pela biblioteca `pugixml` é no entanto armazenada sobre a forma de *strings*, o que não é ideal, visto que muita da informação presente no XML representa números. Estes números são usados como *float*'s pelas funções do OpenGl. Seria extremamente ineficiente efetuar a conversão de *strings* para *float* sempre que um determinado número fosse necessário por uma das funções do OpenGl. Para evitar esta ineficiência criou-se uma estrutura que é uma cópia da árvore dos nodos do ficheiro XML, mas em que a informação da nova árvore está com os tipos certos e prontos a ser usados pelo programa sem necessidade de conversão de tipos. Nas secções anteriores foi mostrado de que forma a classe `Grupo` (e as classes de que esta depende) traduzem a informação presente no XML com os tipos corretos. Uma vez que se pretende reproduzir a árvore do XML, esta nova estrutura criada será também uma árvore, que em vez de nodos XML irá conter grupos. A árvore que armazena os grupos do programa principal tem por isso a seguinte declaração:

```
tree<Grupo> arvoreG;
```

O tipo `tree` é disponibilizado por uma biblioteca C++ que implementa árvores *n*-árias e pode ser consultada em `http://tree.phi-sci.com/`. Esta árvore é preenchida uma vez quando o programa inicia (durante a leitura do XML) e usada pela função `renderScene()` para o desenho da cena.

3. Leitura XML

Quando o motor é iniciado, este deve ler um ficheiro XML com especificação de uma cena. Desta leitura resulta a construção de uma árvore de grupos que depois é percorrida pela função `renderScene()` para o desenho da cena. Neste capítulo explica-se como é feita a leitura do ficheiro XML e de que forma a árvore de grupos é construída. Este processo pode ser dividido em duas partes: em primeiro lugar é necessário converter cada um dos grupos XML para um objeto do tipo `Grupo` que lhe seja equivalente; em segundo lugar, cada um desses grupos deve ser colocado numa árvore que tenha exatamente a mesma estrutura da árvore XML.

3.1 Conversão grupos XML para objetos `Grupo`

Para converter cada um dos grupos XML num objeto do tipo `Grupo` foi definida a função `XMLtoGrupo` que recebe um nodo XML correspondente a um grupo e devolve um objeto do tipo `Grupo` contendo a mesma informação do nodo XML, mas com os tipos certos, prontos a ser usados pelas funções de desenho. A função pode ser descrita pelo seguinte pseudo-código:

```
Grupo XMLtoGrupo(xml_node){
    Grupo resultado
    Coloca informacao do nome do nodo no grupo
    Para todos os elementos xml dentro do grupo:
        SE elemento for transformacao:
            Armazena transformacao em resultado
        SE elemento for rotacao:
            Armazena rotacao em resultado
        SE elemento for escala:
            Armazena escala em resultado
        SE elemento for modelos:
            Para cada modelo em modelos:
                Armazena atributos de cor e modo de desenho no
                   grupo
                Guarda informacao sobre nome de ficheiro no
                   grupo
                Para cada ponto no ficheiro do modelo:
                    Guarda ponto no grupo c/ definicoes de
                       desenho
    return resultado;
}
```

Para cada elemento XML contido no elemento/nodo recebido como argumento, a função verifica qual o nome do elemento para ver se se trata de translação, uma rotação, uma escala ou uma definição de modelos.

Caso se trate de uma transformação (translação, rotação ou escala) é criado um objeto do tipo `Transformacao` contendo os atributos da transformação convertidos de *string* para *float*. A transformação é depois adicionada ao `vector` de transformações do grupo resultado.

Caso se trate de um elemento `models`, são reconhecidos os atributos `mode`, `red`, `green` ou `blue` para especificar a cor de desenho e `file` (campo obrigatório) para especificar o ficheiro .3d onde estão os pontos. As definições de desenho são guardadas num objeto do tipo `DefsDesenho` e o conteúdo do ficheiro de pontos é carregado para um `vector<Coordenadas3D>`. As definições de desenho e os pontos a que se aplicam são associadas por um par, e guardados no `vector` de pontos do `Grupo` resultado.

Quando não houver mais elementos no grupo XML, o `Grupo` resultado é retornado pela função.

3.2 Construção da árvore de grupos

Tendo a função `XMLtoGrupo` que permite transformar um grupo XML no seu objeto `Grupo` equivalente, a leitura do XML corresponde por isso a percorrer cada um dos grupos XML, transforma-los em objetos `Grupo` e adiciona-los a uma árvore com a mesma estrutura de nodos da árvore do XML. Para percorrer a árvore do XML foi efectuada uma travessia *depth-first* iterativa recorrendo a uma stack auxiliar. Uma travessia *depth-first* pode ser representada pelo seguinte pseudo-codigo:

```
DFS-iterativo(raiz) {
    S = stack()
    S.push(raiz)

    Enquanto stack nao vazia:
        v = S.pop()
        Para todos os filhos w de v:
            S.push(w)
}
```

O código acima permite efetuar uma travessia *depth-first* numa árvore. No entanto, o que pretendemos é percorrer a árvore do XML e ir construindo a árvore de grupos. A árvore de grupos terá por isso que ser construída ao mesmo tempo que os nodos já criados são percorridos. Além disso, a árvore de grupos terá também que ser "percorrida" da mesma forma e ao mesmo tempo que a árvore do XML para garantir que as árvores têm a mesma estrutura de nodos. O código da função `leXML()` corresponde por isso à travessia de duas árvores em simultâneo (de grupos e do xml). Durante a travessia das duas árvores são criados objetos do tipo `Grupo` que são adicionados à árvore dos grupos de acordo com o seguinte pseudo-codigo:

```
DFS-iterativo(raiz_xml) {

    tree<Grupo> arvore_grupos
    Stack_xml = stack()
    Stack_grupos = stack()

    raiz_grupo = XMLtoGrupo(raiz_xml)
    Define raiz_grupo como cabeca da arvore_grupos
    Stack_xml.push(raiz_xml)
    Stack_grupos.push(raiz_grupo)

    Enquanto stack nao vazia:
        nodo_xml = Stack_xml.pop()
        nodo_grupo = Stack_grupos.pop()

        Para todos os filhos xml_child de nodo_xml:
            grupo_covertido = XMLtoGrupo(xml_child)
            Adiciona grupo_convertido como filho de nodo_grupo
            Stack_xml.push(raiz_xml)
            Stack_grupos.push(grupo_convertido)

}
```

4. Descrição do ciclo de *Rendering*

Após ser feita a leitura do XML e armazenada a informação nas estruturas de dados, por forma a desenhar o sistema solar, é feita, na função `renderScene()`, a travessia da árvore onde estes elementos estão guardados.

Uma vez que os grupos estão guardados na variável global `arvoreG`, o processo de desenho desses grupos é iniciado colocando um iterador na raiz da árvore.

```
tree<Grupo>::iterator head = arvoreG.begin();
```

De seguida, processam-se todos os elementos da árvore, recorrendo à função `desenhaGrupo()` iniciando na raiz da árvore:

```
desenhaGrupo(head);
```

Esta função é responsável por percorrer a árvore e desenhar todos os grupos armazenados. Para percorrer a árvore é feita uma travessia em profundidade (*depth first search*), de modo a que todas as transformações referenciadas por um grupo pai, sejam propagadas para os seus filhos. Para controlar o sistema de coordenadas recorremos às funções `pushMatrix()` e `popMatrix()`. Assim, antes de efetuar as transformações de cada grupo, faz-se um `pushMatrix()` para salvaguardar o sistema de coordenadas atual numa *stack* de matrizes. Quando um grupo não tem mais filhos faz se um `popMatrix()` que permite restaurar o sistema de coordenadas anterior. Estas funções são usadas em conjunto com o objetivo de controlar as transformações aplicadas.

O processo usado para o desenho de grupos segue, para cada um dos grupos, os seguintes passos:

- Em primeiro lugar, é guardada a matriz do sistema de coordenadas atual (`pushMatrix()`).
- De seguida, é percorrido o `vector` de instâncias de `Transformacao` e para cada uma, de acordo com o seu tipo (seja `ROTACAO`, `TRANSLACAO` ou `ESCALA`), é aplicada a respetiva transformação.
- Após terem sido aplicadas as transformações, é percorrido o `vector pontos` (apresentado na secção 2.4) e desenhados todos os pontos segundo as definições de desenho.
- Com o objetivo de serem desenhados todos os grupos, são percorridos todos os filhos do nodo atual e para cada filho é invocada recursivamente a função `desenhaGrupo`.
- Por fim, é invocada a função `popMatrix()` por forma a restaurar o sistema de coordenadas para o ponto antes das transformações do grupo terem sido aplicadas.

O pseudo-código da função `desenhaGrupo` é o seguinte:

```
void desenhaGrupo(tree<Grupo>::iterator it_grupo) {
    glPushMatrix()
    Aplica as Transformacoes
    Desenha pontos de acordo com as definicoes de desenho
    Para todos os filhos
        Invoca a funcao desenhaGrupo recursivamento
    glPopMatrix()
}
```

5. Câmara

Um dos objetivos desta fase do projeto é instalar uma câmara em primeira pessoa, de modo a que o utilizador se possa mover ao longo do espaço, podendo assim visualizar melhor o que pretende ver ou da perspetiva que desejar.

Para implementar a câmara necessitamos de um ponto, que corresponde à posição da câmara, e de um vetor $\vec{D}$, que corresponde à direção para a qual a câmara está a olhar. Quando se pretender deslocar a câmara para a frente, ela desloca-se no sentido deste vetor, se se pretender deslocar a câmara para trás ela desloca-se no sentido contrário ao deste vetor. Caso se queira mudar o ponto para a câmara está a olhar, então as coordenadas deste vetor vão ser alteradas. Apesar da norma do vetor $\vec{D}$ não ser relevante para definir a direcção para onde a câmara está a olhar, é util que este seja normalizado (norma=1), para que seja mais fácil trabalhar com este vetor algebricamente. Por exemplo, aumentar a aumentar a velocidade de deslocamento da câmara corresponde a multiplicar $\vec{D}$ por um escalar k. Como a norma de $\vec{D}$ é 1, sabemos que multiplicar por $2k$ ao invés de k corresponde a duplicar a velocidade da câmara. Esta simplicidade decorre de em cada momento se saber que $\vec{D}$ tem norma 1.

Tendo por base estes pressupostos, definiu-se que a posição da câmara seria um ponto$P$ centro de uma esfera de raio 1. Considerando um ponto $Q(x, y, z)$ na superfície dessa esfera como sendo o ponto para onde a câmara está a olhar, sabe-se que $\vec{D} = Q - P$. A esquematização deste raciocínio é apresentada na Figura 5.1.

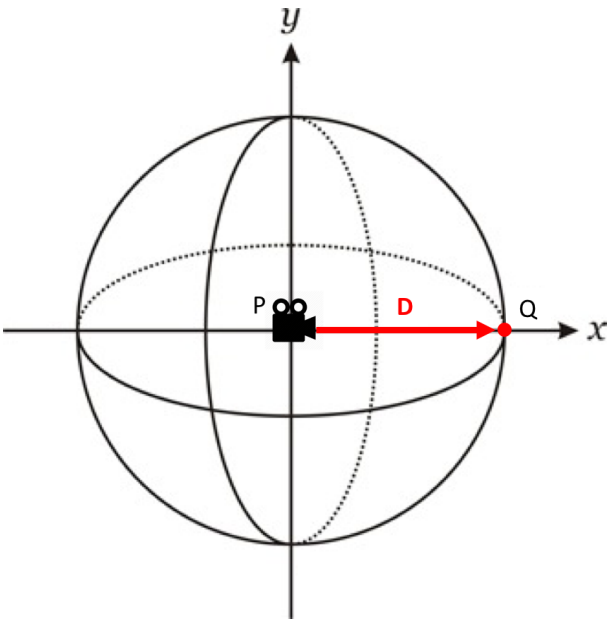

Figura 5.1: Esquematização da posição da câmara (P) e do vetor direção ($\vec{D}$)

Uma vez que a câmara pode então ser representada por um ponto central de uma esfera e um ponto na superfície da esfera, foi utilizada a classe `CoordsEsfericas` como base para a representação da câmara com auxílio de variáveis `Coordenadas3D` para representar o ponto P e $\vec{D}$.

Deste modo, decidiu-se proceder à criação da classe câmara, que é definida segundo os seguintes variáveis de classe:

```
class Camara {
public:
    Coordenadas3D p;
    CoordsEsfericas q;
    Coordenadas3D v_d;
}
```

Depois de definida a classe `Camara`, declarou-se como sendo uma variável global ao motor:

```
Camara camara;
```

Na função `renderScene()` os valores do ponto $P(px, py, pz)$, correspondentes à posição da câmara, $\vec{D}(dx, dy, dz)$, correspondentes à direção da câmara são passados como argumento à função `gluLookAt()` da seguinte forma:

```
gluLookAt(px, py, pz,
          px+dx,py+dy,pz+dz,
          0.0f,1.0f,0.0f);
```

O utilizador pode mudar a posição e orientação da câmara recorrendo ao teclado e ao rato. Para tal a classe `Camara` disponibiliza os métodos que permitem manipular as suas variáveis de forma a que seja possível:

- Mover a câmara para a frente ou para trás: as coordenadas da posição onde se encontra a câmara são alteradas, o centro da esfera no qual a câmara se encontra passa a ser essa posição e o ponto para onde a câmara está a olhar é também é atualizado (visto que o centro da esfera muda).

```
frente(float velocidade) {
    posicao = posicao + velocidade *vetor_direcao;
    atualiza o centro da esfera com a nova posicao
    actualiza ponto para onde a camara esta a olhar de
       acordo com novo nova posicao da camara
}

tras(float velocidade) {
    posicao = posicao - velocidade * vetor_direcao;
    atualiza o centro da esfera com a nova posicao
    actualiza ponto para onde a camara esta a olhar de
       acordo com nova posicao da camara
}
```

- Mover a câmara para a direita ou para a esquerda: declara-se o vetor $\vec{Up}(0,1,0)$, faz-se o produto externo (disponibilizado pela classe `Coordenadas3D`) entre o vetor $\vec{Up}$ e o vetor $\vec{D}$ (aplicando a regra da mão direita conforme se pretenda ir para a esquerda ou para a direita) e soma-se as coordenadas do vetor resultado à posição da câmara para se obter a nova posição, que passa a ser o novo centro da esfera. Tal como anteriormente, ao ser alterado o ponto onde a câmara está (centro da esfera) o ponto para onde a câmara está a olhar também precisa de ser atualizado.

```
direita(float velocidade) {
    direita = produtoExterno(vetor_direcao,Up);
    posicao = posicao + velocidade * direita;
    atualiza o centro da esfera para a nova posicao
    actualiza ponto para onde a camara esta a olhar de
       acordo com nova posicao da camara
}

esquerda(float velocidade) {
    esquerda =  produtoExterno(Up,vetor_direcao);
    posicao = posicao + velocidade * esquerda;
    atualiza o centro da esfera para a nova posicao
    actualiza ponto para onde a camara esta a olhar de
       acordo com nova posicao da camara
}
```

- Mover a câmara para cima ou para baixo: apenas se atualizam as coordenadas da posição da câmara (centro da esfera) onde a câmara se encontra.

```
cima(float velocidade) {
    posicao = posicao + velocidade * Up;
    atualiza o centro da esfera para a nova posicao
}

baixo(float velocidade) {
    posicao = posicao - velocidade * Up;
    atualiza o centro da esfera para a nova posicao
}
```

- Rodar a câmara: para rodar a câmara em qualquer sentido tiramos partido da classe `CoordsEsfericas`. Assim, dado um ângulo segundo o qual se queira rodar a câmara, bastará passar esse ângulo como argumento aos métodos já disponibilizados pela classe `CoordsEsfericas` para obter as novas coordenadas do ponto Q que está na superfície da esfera. Com as novas coordenadas do ponto Q podemos obter as novas coordenadas do vetor D, segundo a mesma fórmula usada em cima: $\vec{D} = Q - P$. Assim, rodar a câmara apenas vai ter influência nas coordenadas do vetor ($\vec{D}$) da câmara.

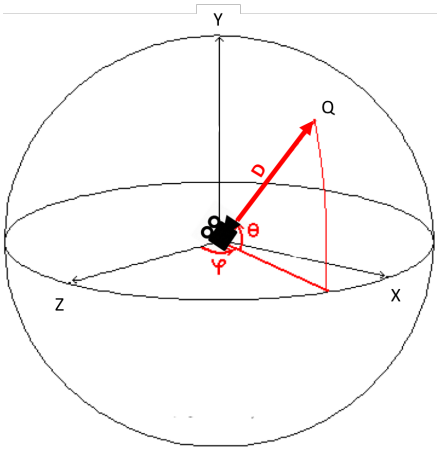

Figura 5.2: Alterar os ângulos θ e ϕ alteram a direcção de $\vec{D}$ e por isso alteram a posição do ponto Q para onde a câmara está a olhar. Mudar θ permite olhar para cima/baixo e alterar ϕ permite olhar para a esquerda e para a direita

```
lookBaixo(float ang){
    q.paraBaixo(ang);
    v_d = q.cCartesianas - p;
}

lookCima(float ang) {
    q.paraCima(ang);
    v_d = q.cCartesianas - p;
}

lookDireita(float ang) {
    q.paraEsquerda(ang);
    v_d = q.cCartesianas - p;
}

lookEsquerda(float ang) {
    q.paraDireita(ang);
    v_d = q.cCartesianas - p;
}
```

6. Interação com o utilizador

6.1 Teclado

Foram associadas ações às teclas do teclado, dando ao utilizador oportunidade de interagir com o programa com as funcionalidades descritas abaixo:

- **Tecla 'W'** - Move a câmara para a frente.
- **Tecla 'S'** - Move a câmara para trás.
- **Tecla 'A'** - Move a câmara para a esquerda.
- **Tecla 'D'** - Move a câmara para a direita.
- **Tecla 'Q'** - Move a câmara para cima.
- **Tecla 'E'** - Move a câmara para baixo.
- **Tecla 'J'** - Reduz velocidade da câmara.
- **Tecla 'K'** - Repõe a velocidade "normal" da câmara (valor original).
- **Tecla 'L'** - Aumenta velocidade da câmara.

Para implementar estas operações foi criada uma função `teclas_normais_func` que consiste num `switch` que faz uma ação diferente consoante a tecla pressionada.

```
void teclas_normais_func(unsigned char key, int x, int y) {
    switch (tolower(key)) {
        case 'w':
            camara.frente(cameraSpeed * M_PI / 360.0);
            break;
        case 's':
            camara.tras(cameraSpeed * M_PI / 360.0);
            break;
        case 'a':
            camara.esquerda(cameraSpeed * M_PI / 360.0);
            break;
        case 'd':
            camara.direita(cameraSpeed * M_PI / 360.0);
            break;
        case 'e':
            camara.baixo(cameraSpeed * M_PI / 360.0);
```

```
            break;
        case 'q':
            camara.cima(cameraSpeed * M_PI / 360.0);
            break;
        case 'j':
            cameraSpeed-=3;
            break;
        case 'k':
            cameraSpeed=10;
            break;
        case 'l':
            cameraSpeed+=3;
            break;
    }
    glutPostRedisplay();
}
```

As opções de movimentação da câmara consistem em operações disponibilizadas pela classe `Camara`, já apresentadas na secção 5.

Existem ainda teclas consideradas "especiais" pelo GLUT às quais também foram adicionadas ações que permitem rodar a câmara:

- **Tecla 'UP'** - Roda a câmara para cima.
- **Tecla 'DOWN'** - Roda a câmara para baixo.
- **Tecla 'LEFT'** - Roda a câmara para a esquerda.
- **Tecla 'RIGHT'** - Roda a câmara para a direita.

Visto que o GLUT trata as teclas 'UP', 'DOWN', 'LEFT' e 'RIGHT' como teclas especiais, teve que ser criada uma função `teclas_especiais_func` para processar as ações nestas teclas:

```
void teclas_especiais_func(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_LEFT:
            camara.lookEsquerda(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_UP:
            camara.lookCima(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_RIGHT:
            camara.lookDireita(cameraSpeed*M_PI/360.0);
            break;
        case GLUT_KEY_DOWN:
            camara.lookBaixo(cameraSpeed*M_PI/360.0);
            break;
        }
}
```

6.2 Rato

Além de terem sido associadas ações ao teclado, foram também associadas ações aos botões direito e esquerdo do rato. O botão direito do rato permite ao utilizador aceder a um menu de contexto com diferentes opções de visualização. O botão esquerdo do rato permite movimentar a câmara.

6.2.1 Menus - Botão direito

Para proporcionar alguma flexibilidade na visualização das figuras e também de forma a aumentar a interação do utilizador com o programa, foi incluído o menu representado na Figura 6.1. Este menu, que pode ser acedido em qualquer posição da janela, contém sub-menus e permite as seguintes funcionalidades:

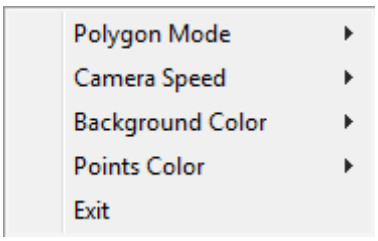

Figura 6.1: Menu de opções acessível com o clique do botão do lado direito do rato

O código de criação dos menus foi já apresentado na fase anterior, permanecendo na função `criaMenus()` do `Motor`. Nesta fase, dado que as cores e o modo de desenho dos triângulos podem ser especificadas no ficheiro XML, decidiu-se incluir uma opção nos sub-menus *Points Color* e *Polygon Mode* para voltar às definições originais.

A implementação desta funcionalidade fez-se com o uso de duas variáveis booleanas (uma para a cor e outra para o modo) para desativar/ativar o desenho dos triângulos a partir das definições de desenho carregadas para a árvore de suporte vinda do ficheiro XML ou, contrariamente, a partir das definições registadas pelos menus de opções.

Estas *flags* são testadas na função `desenhaGrupo()` e, conforme o seu valor, é usada a função `glPolygonMode()` e `glColor3f()` para ajustar as definições pretendidas pelo utilizador.

6.2.2 Movimento da câmara - Botão esquerdo

Como visto anteriormente, a orientação da câmara pode ser controlada com as setas direcionais do teclado. Além destas opções, também é possível fazer esta alteração com o botão esquerdo do rato.

Para rodar a câmara, o utilizador deve carregar no botão esquerdo e arrastar o rato na direção que pretende, mantendo o botão pressionado.

Para implementar esta funcionalidade em primeiro lugar foi necessário criar 2 variáveis globais com a posição x e y do rato relativamente à janela:

```
int mouse_x, mouse_y;
```

Quando o utilizador carrega no botão esquerdo do rato, a função `mouse_events_func` é chamada e guarda a posição (x, y) do rato nas variáveis globais.

```
void mouse_events_func(int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            mouse_x = x;
            mouse_y = y;
        }
        break;
    }
}
```

Quando o utilizador arrasta o rato, é chamada a função `mouse_motion_func`. Esta função regista em `deltaX` e `deltaY` qual o deslocamento do rato. Este deslocamento do rato é depois usado como parâmetro das funções que efetuam a rotação da câmara:

```
void mouse_motion_func(int x, int y) {
    if (deltaX > 0) {
        camara.lookDireita(deltaX*cf*cameraSpeed*M_PI/360.0);
    } else {
        camara.lookEsquerda(abs(deltaX)*cf*cameraSpeed*M_PI
           /360.0);
    if(deltaY >0) {
        camara.lookBaixo(deltaY*cf*cameraSpeed*M_PI/360.0);
    } else {
        camara.lookCima(abs(deltaY)*cf*cameraSpeed*M_PI/360.0);
    }
}
```

Por fim, registou-se estas funções no GLUT:

```
glutMouseFunc(mouse_events_func);
glutMotionFunc(mouse_motion_func);
```

7. Construção do Sistema Solar

Para a construção de uma representação do Sistema Solar foi escrito um ficheiro em formato XML com a formatação e a estrutura apresentadas na secção 1 deste relatório. Neste capítulo são apresentadas as decisões tomadas bem como as figuras usadas para a representação.

7.1 Figuras usadas na representação

Foi usada como figura principal uma esfera (obtida com o `Gerador`) com raio `1`, usada no desenho do Sol, dos planetas e das Luas.

Em relação aos anéis do planeta Saturno, decidiu-se criar uma figura "anel" cuja implementação pode ser encontrada de seguida. Esta figura foi também usada para desenhar as órbitas de cada um dos planetas em volta do Sol, através de anéis muito "finos".

7.1.1 Anel

Para gerar um anel o utilizador deverá recorrer ao `Gerador`, introduzindo um comando com a seguinte sintaxe:

```
gerador anel raioI raioE fatias ficheiro.3d
```

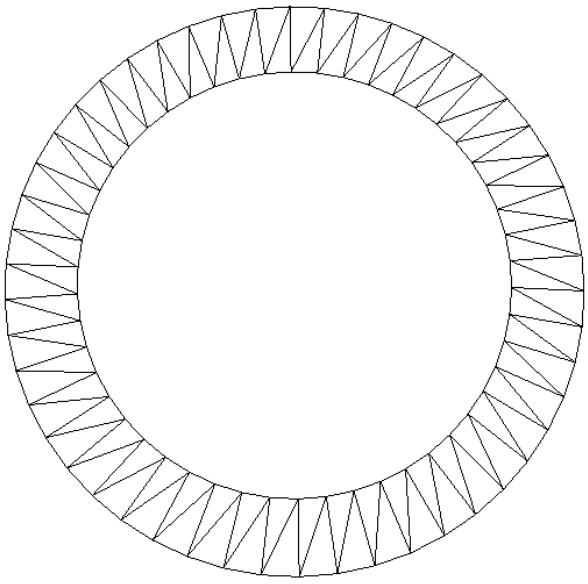

Figura 7.1: Esquema do desenho de um Anel com "fatias"

Para desenhar esta figura recorreu-se à classe `CoordsPolares` descrita no relatório anterior, obtendo-se um anel em XZ centrado na origem $C(0,0,0)$. Um anel é constituído por fatias, sendo cada "fatia" um retângulo constituído por 4 pontos tendo em conta o raio interno, o raio externo e o ângulo polar:

```
deltaAz = (float)2 * M_PI / fatias;
a = CoordsPolares(o, raioI,deltaAz*i).toCartesianas();
b = CoordsPolares(o, raioE, deltaAz*i).toCartesianas();
c = CoordsPolares(o, raioI, deltaAz*(i+1)).toCartesianas();
d = CoordsPolares(o, raioE, deltaAz*(i+1)).toCartesianas();
```

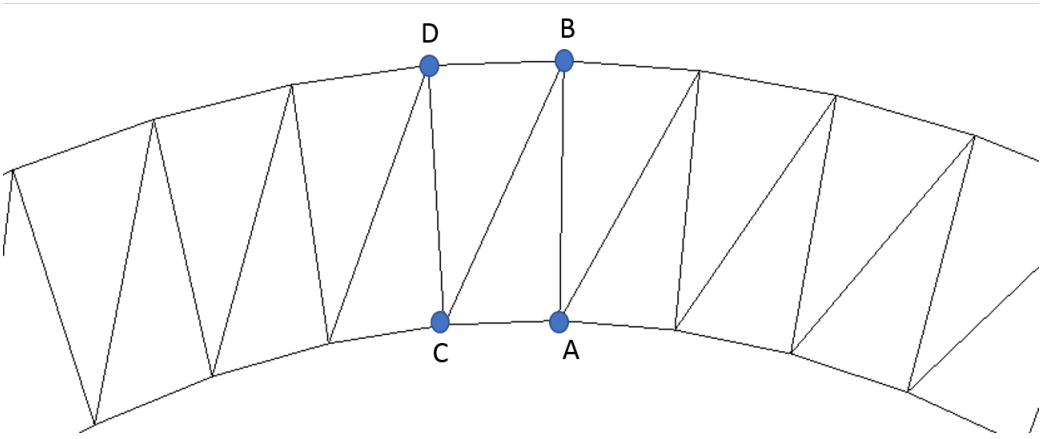

Figura 7.2: Representação do "rectângulo" base desenhado em cada iteração do ciclo interior, com indicação dos pontos A, B, C e D

Uma vez que é desejável que o anel seja visível a partir de qualquer direção, optou-se por se desenhar com ambas as orientações.

A função responsável pelo desenho de anéis é a função `geraAnel`, cujo pseudo-código é o seguinte:

```
Figura& geraAnel(Ponto3D C, float raioI, float raioE, int f) {
    Para cada fatia {
        Calcula CoordsPolares A, B,C D
        Coloca pontos no vetor pela ordem:
            A-B-C-C-B-D (Orientacao positiva do eixo dos Y)
            A-C-B-C-D-B (Orientacao negativa do eixo dos Y)
    }
    return *this
}
```

7.2 Medidas sistema solar

Inicialmente foi considerado o uso das medidas reais do Sistema Solar, aplicando uma escala de $1\ cm$ para $1\ UA$, porém chegou-se à conclusão de que tais medidas, apesar de já muito reduzidas, seriam insuportáveis de visualizar.

Posto isto, decidiu-se adotar uma representação definida pelo grupo de trabalho e que pode ser consultada na tabela 7.2, onde consta informação sobre a escala

aplicada à esfera inicial e a distância do astro ao seu "pai" (que pode ser o Sol, no caso dos planetas, ou um planeta, no caso das luas), usada na translação do objeto.

Astro	Escala	Astro pai	Distância ao pai
Sun	1.1	-	-
Mercury	0.085	*Sun*	$x = 1.8$
Venus	0.135	*Sun*	$x = 2.4$
Earth	0.14	*Sun*	$x = 3.2$
Moon	0.14	*Earth*	$x = y = z = 1$
Mars	0.11	*Sun*	$x = 4$
Phobos	0.14	*Mars*	$x = y = z = 0.9$
Deimos	0.1	*Mars*	$x = 0.9\ y = 0.8\ z = -1.1$
Jupiter	0.6032	*Sun*	$x = 6.20$
Ganymed	0.1	*Jupiter*	$x = y = z = 0.8$
Callisto	0.11	*Jupiter*	$x = y = z = -0.8$
Saturn	0.5032	*Sun*	$x = 8.9$
Uranus	0.2912	*Sun*	$x = 11$
Titania	0.11	*Uranus*	$x = z = 0.7\ y = -0.7$
Neptune	0.2832	*Sun*	$x = 13$
Pluto	0.075	*Sun*	$x = 15$

Tabela 7.1: Medidas de escalas e translações consideradas

Nota: As escalas são todas uniformes, i.e., aplicadas em x, y e z.

Note-se que no ficheiro em formato XML foi igualmente considerada a noção de descendência entre os astros: o Sol é o "pai" de todos os astros do Sistema Solar. As luas são "filhas" dos planetas respetivos. Com esta noção, as escalas aplicadas a um astro, aplicam-se a todos os seus astros filhos, daqui que a análise dos dados da tabela deve ter em conta este facto.

Em relação ao planeta Saturno, foram criados dois anéis, representados por um grupo cada. Para o desenho destes foram criados dois ficheiros de pontos da figura "disco", com um raio interno e externo diferentes. O mesmo foi feito para as órbitas de cada planeta, que correspondem a discos com uma espessura mínima e com um raio interno aproximadamente igual ao valor da distância de cada planeta ao Sol.

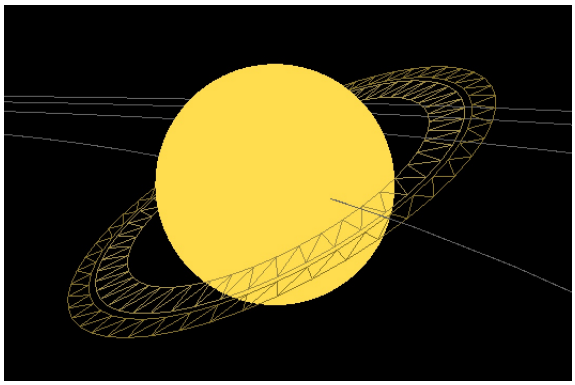

Figura 7.3: Saturno e os seus anéis

Para tornar a cena mais apelativa, foram definidas as cores que cada astro deve ter, conforme se pode ver nas figuras abaixo. Por último, os anéis de Saturno foram desenhados pelo modo `GL_LINE`, enquanto que os restantes planetas foram desenhados pelo modo `GL_FILL`.

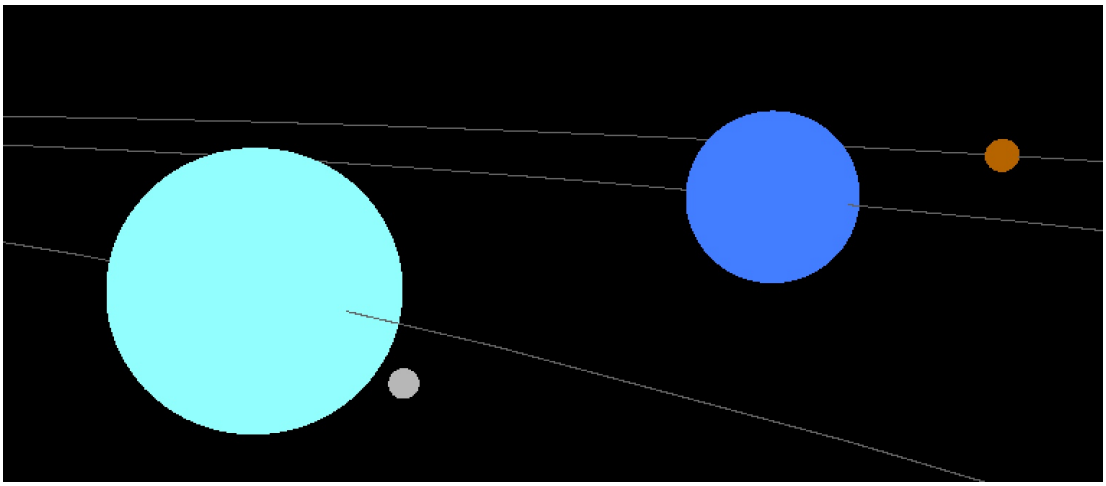

Figura 7.4: Planetas Úrano, Neptuno e Plutão

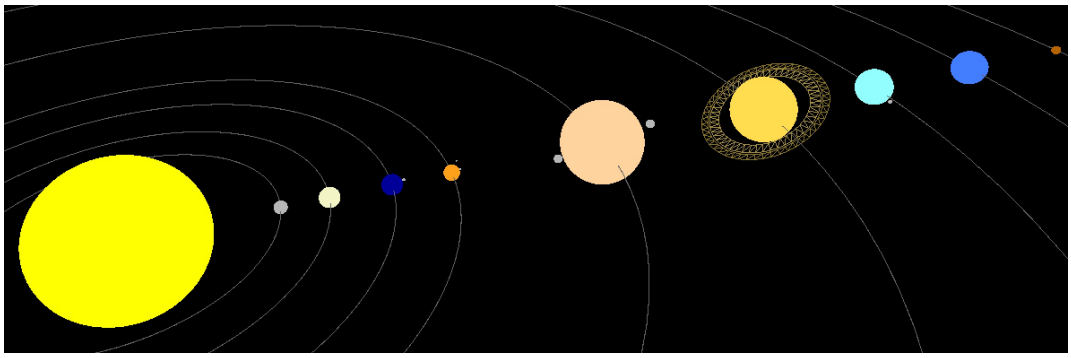

Figura 7.5: Visualização geral do Sistema Solar

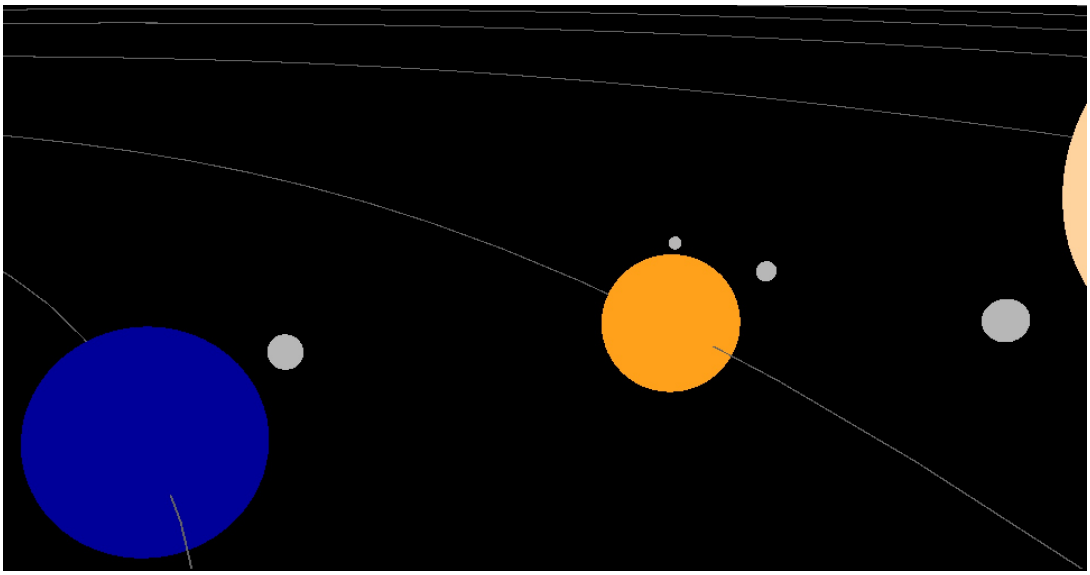

Figura 7.6: Algumas das luas desenhadas

Conclusão

Os objetivos propostos nesta fase do trabalho foram cumpridos. As funcionalidades do motor foram aumentadas por forma a permitir ler ficheiros XML com modelos definidos em hierarquia contendo transformações.

Como valorização do trabalho foi implementada uma câmara em primeira pessoa, foram adicionadas algumas extensões ao XML e foram adicionados alguns menus ao motor. A câmara em primeira pessoa permite ao utilizador explorar cenas complexas de forma mais fácil. Este aspeto é relevante, visto que o motor desenvolvido nesta fase tem suporte para que as cenas definidas no XML possam ser complexas e portanto uma forma fácil de visualizar a cena é necessária. As extensões ao XML realizadas tiveram em vista dar mais controlo ao utilizador sobre o desenho e facilitar a construção do XML. Nesse sentido foi adicionado o atributo `uniform` ao elemento `<scale>` do XML para poupar ao utilizador o trabalho de indicar manualmente os atributos `X`,`Y` e `Z` no caso de uma escala uniforme; os atributos `red`, `green` e `blue` do elemento `<model>` do XML permitem desenhar objetos com cores diferentes e o atributo `mode` permite mudar o modo de desenho. Isto permite a construção de cenas complexas e visualmente apelativas. Os menus adicionados ao motor constituem também um extra em relação aos objetivos mínimos iniciais, no entanto são úteis na medida em que permitem ao utilizador personalizar a experiência de visualização.

Como possíveis melhorias ao trabalho, foi considerado dar a possibilidade ao utilizador de alternar entre a câmara em primeira pessoa e a câmara colocada sobre uma esfera do primeiro trabalho.