Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Relatório do projeto

Fase 4 - Normais e Coordenadas de Textura

Computação Gráfica

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

Braga, 21 de Maio de 2017

Conteúdo

Resumo

O presente relatório visa documentar a quarta fase do trabalho prático da Unidade Curricular de Computação Gráfica qu e consistiu na aplicação de texturas e uso de iluminação.

O gerador sofreu alterações de forma a permitir criar ficheiros com pontos, normais e coordenadas de textura para cada figura pedida. Relativamente ao motor apresentam-se melhorias passando a permitir a leitura de texturas e aplicação das mesmas, a inclusão de modelos coloridos com possibilidade de várias componentes (difusa, ambiente, especular e emissiva) e ainda a utilização de luzes de diversos tipos.

Relativamente aos elementos XML, foram adicionados novos atributos por forma a permitir as novas funcionalidades e de 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 4 capítulos que pretendem ilustrar o processo de resolução do enunciado proposto.

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

No capítulo 2 são introduzidas as alterações feitas ao gerador de forma a permitir gerar ficheiros com pontos, normais e coordenadas de textura.

O capítulo 3 detalha as alterações feitas ao motor de modo a suportar as novas funcionalidades, nomeadamente aplicação de luzes e texturas.

Por fim, no capitulo 4, é explicado como foi construido a *scene* de exemplo correspondente ao sistema solar assim como a *scene* de teste criada com o objetivo de demonstrar as figuras que não são utilizadas no sistema solar.

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 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`.

1.2 Nodo *<lights>*

Nodo filho de `<scene>` que contém a indicação das luzes que fazem parte da cena. Este elemento deve conter como filhos elementos `<light>` que especificam os atributos de cada luz pertencente à cena, conforme será descrito de seguida. Este elemento, caso esteja presente (i.e, caso se pretenda luzes na cena) deverá ser o primeiro elemento da cena e deverá estar fora de qualquer grupo.

```
<scene>
    <lights>
        Define luzes atraves de elementos <light>
    </lights>
    ...
    Grupos XML
    ...
</scene>
```

1.3 Nodo *<light>*

Nodo filho de `<lights>` que especifica uma luz que será aplicada a toda a cena. Este elemento pode conter os seguintes atributos:

- ***type*** - Tipo de luz pretendida. O valor deste atributo deve ser a string "DIRECTIONAL" caso se pretenda uma luz direcional, "SPOTLIGHT" caso se pretenda uma luz do tipo *spotlight* ou "POINT" caso se pretenda apenas um ponto de luz. Este atributo é obrigatório.

- ***posX*** - Componente no eixo dos x da posição da luz. O valor deste atributo deverá ser uma string que possa ser convertida para *float*. No caso da luz ser direcional, este atributo representa a componente no eixo dos x da direção da luz. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0 (zero).

- ***posY*** - Componente no eixo dos y da posição da luz. O valor deste atributo deverá ser uma string que possa ser convertida para *float*. No caso da luz ser direcional, este atributo representa a componente no eixo dos y da direção da luz. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0 (zero).

- ***posZ*** - Componente no eixo dos z da posição da luz. O valor deste atributo deverá ser uma string que possa ser convertida para *float*. No caso da luz ser direcional, este atributo representa a componente no eixo dos z da direção da luz. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0 (zero).

- ***ambR*** - Intensidade da componente vermelha (*Red*) da componente ambiente da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***ambG*** - Intensidade da componente verde (*Green*) da componente ambiente da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***ambB*** - Intensidade da componente azul (*Blue*) da componente ambiente da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***ambA*** - Opacidade *alpha* da componente ambiente da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***diffR*** - Intensidade da componente vermelha (*Red*) da componente difusa da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***diffG*** - Intensidade da componente verde (*Green*) da componente difusa da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***diffB*** - Intensidade da componente azul (*Blue*) da componente difusa da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***diffA*** - Opacidade *alpha* da componente difusa da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***specR*** - Intensidade da componente vermelha (*Red*) da componente especular da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***specG*** - Intensidade da componente verde (*Green*) da componente especular da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***specB*** - Intensidade da componente azul (*Blue*) da componente especular da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***specA*** - Opacidade *alpha* da componente especular da luz. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***constAtt*** - Factor constante de atenuação da luz. Deverá ser uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***linearAtt*** - Factor linear de atenuação da luz. Deverá ser uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***quadAtt*** - Factor quadrático de atenuação da luz. Deverá ser uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***spotX*** - Apenas deve ser usado caso a luz seja do tipo *spotlight*. Indica a componente no eixo x da direcção para a qual o foco de luz aponta. Deve corresponder a uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***spotY*** - Apenas deve ser usado caso a luz seja do tipo *spotlight*. Indica a componente no eixo y da direcção para a qual o foco de luz aponta. Deve corresponder a uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***spotZ*** - Apenas deve ser usado caso a luz seja do tipo *spotlight*. Indica a componente no eixo z da direcção para a qual o foco de luz aponta. Deve corresponder a uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor -1.

- ***spotCutoff*** - Apenas deve ser usado caso a luz seja do tipo *spotlight*. Indica o ângulo de abertura (em graus) do foco de luz. Deve corresponder a uma string que possa ser convertida para *float*. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 180.

1.4 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>
```

1.5 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.

1.5.1 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".

1.5.2 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>
```

1.6 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.

1.7 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.

1.7.1 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>
```

1.7.2 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>
```

1.8 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"/>
```

1.9 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>
```

1.10 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.
- ***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.
- ***texture*** - Ficheiro que contém a imagem a ser aplicada como textura ao objecto. Não é um atributo obrigatório.
- ***ambR*** - Intensidade da componente vermelha (*Red*) da componente ambiente do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.2.
- ***ambG*** - Intensidade da componente verde (*Green*) da componente ambiente do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.2.
- ***ambB*** - Intensidade da componente azul (*Blue*) da componente ambiente do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será

convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.2.

- ***ambA*** - Opacidade *alpha* da componente difusa do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***diffR*** - Intensidade da componente vermelha (*Red*) da componente difusa do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.8.

- ***diffG*** - Intensidade da componente verde (*Green*) da componente difusa do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.8.

- ***diffB*** - Intensidade da componente azul (*Blue*) da componente difusa do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.8.

- ***diffA*** - Opacidade *alpha* da componente difusa do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***ambDiffR*** - O mesmo que usar os atributos `diffR` e `ambR` com o mesmo valor.

- ***ambDiffG*** - O mesmo que usar os atributos `diffG` e `ambG` com o mesmo valor.

- ***ambDiffB*** - O mesmo que usar os atributos `diffB` e `ambB` com o mesmo valor.

- ***ambDiffA*** - O mesmo que usar os atributos `diffA` e `ambA` com o mesmo valor.

- ***specR*** - Intensidade da componente vermelha (*Red*) da componente especular do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***specG*** - Intensidade da componente verde (*Green*) da componente especular do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***specB*** - Intensidade da componente azul (*Blue*) da componente especular do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***specA*** - Opacidade *alpha* da componente especular do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***emR*** - Intensidade da componente vermelha (*Red*) da componente emissiva do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***emG*** - Intensidade da componente verde (*Green*) da componente emissiva do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***emB*** - Intensidade da componente azul (*Blue*) da componente emissiva do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 0.

- ***emA*** - Opacidade *alpha* da componente emissiva do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre -1 e 1. Caso seja passado um número inteiro, este será convertido para um *float* entre -1 e 1. Não é um atributo obrigatório. Caso não seja indicado, assume por defeito o valor 1.

- ***shininess*** - Componente de brilho do material. Deverá ser uma string que possa ser convertida para *float* que represente um número entre 0 e 128. Caso não seja indicado, assume por defeito o valor 0.

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. Gerador

Nesta nova fase do projeto pretende-se que o GERADOR interprete os pedidos do utilizador e produza um ficheiro .3d, que contém os pontos correspondentes à figura solicitada, bem como as normais e as coordenadas de textura para cada ponto. O formato do ficheiro de saída terá a seguinte configuração:

```
p1x p1y p1z      // ponto P(x,y,z)
n1x n1y n1z      // vetor normal n(x,y,z) ao ponto P
t1s t1t          // coordenada de textura t(s,t) do ponto P
...
pix piy piz
nix niy niz
tis tit
...
```

Para a implementação da funcionalidade de gerar coordenadas de textura por parte do GERADOR, foi necessário criar uma classe que as representasse. Uma vez que têm a particularidade de serem definidas por apenas duas coordenadas s e t, não foi possível (nem seria correto) usar a classe `Coordenadas3D`.

Como tal, tem-se definida a classe `CoordsTextura` da seguinte forma:

```
class CoordsTextura {
public:
    float s, t;
};
```

De seguida apresentamos como se procedeu ao cálculo das normais e das coordenadas de textura para cada uma das figuras (a apresentação do cálculo das coordenadas dos pontos já foi apresentada na Fase 1).

2.1 Plano

Caso o utilizador solicite a geração de um plano em Z deverá introduzir um comando com a seguinte sintaxe:

```
gerador plane comprimento largura divsx divsz ficheiro.3d
```

A função responsável por gerar o plano possui a seguinte assinatura:

```
Figura& geraPlanoY(Coordenadas3D o, float comp, float larg,
        int divsx, int divsz, ORIENTACAO_FIG orientacao)
```

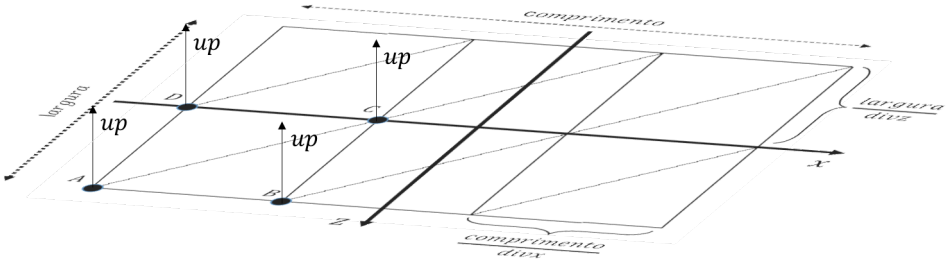
O resultado deste comando é um plano em XZ centrado na origem o (que por defeito é o ponto $C(0,0,0)$), com o comprimento, a largura, e o número de divisões no eixo X e em Z especificados.

Para além da função `geraPlanoY`, também foram implementadas as funções `geraPlanoX` e `geraPlanoZ`, similares à `geraPlanoY` mas os planos resultantes são em YZ e XY respetivamente.

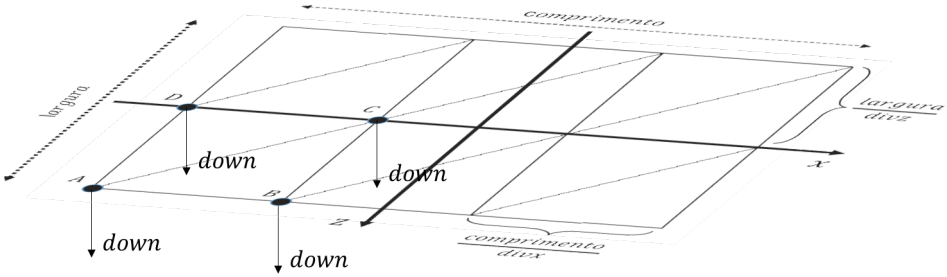
A orientação passada como parâmetro (por defeito é `CIMA`) vai ter influência quer na ordem dos pontos quer no sentido da normal. Este parâmetro foi acrescentado relativamente às fases anteriores para ser possível desenhar uma caixa invertida, que neste projeto vai ser usada para "delimitar" o Sistema Solar.

2.1.1 Cálculo das normais

No caso do plano em y, se o parâmetro orientação for `CIMA`, então a normal a todos os pontos do plano é o vetor $\vec{up} = (0,1,0)$. Se a orientação for `BAIXO`, então a normal a todos os pontos gerados é $\vec{down} = (0,-1,0)$.



(a) Orientação = `CIMA`



(b) Orientação = `BAIXO`

Figura 2.1: Normais do plano

Na função `geraPlanoX`, a orientação `CIMA` significa que a normal em todos os pontos será $\vec{dir} = (1,0,0)$ e a orientação `BAIXO` significa que a normal em todos os pontos será $\vec{esq} = (-1,0,0)$.

Na função `geraPlanoZ`, a orientação `CIMA` significa que a normal em todos os pontos será $\vec{frt} = (0,0,1)$ e a orientação `BAIXO` significa que a normal em todos os pontos será $\vec{trs} = (0,0,-1)$.

2.1.2 Cálculo das coordenadas de textura

Tendo em conta o numero de divisões em x, o numero de divisões em z e os pontos A,B,C,D apresentados na Figura 2.1 o cálculo das coordenadas de textura é realizado da seguinte maneira:

```
float deltaTexS = (float)1.0f / divsx;
float deltaTexT = (float)1.0f / divsz;

for (int j = 0; j < divsz; ++j) {
    for (int i = 0; i < divsx; ++i) {
        ctA = (i * deltaTexS, j * deltaTexT);
        ctB = ((i+1) * deltaTexS, j * deltaTexT);
        ctC = ((i+1) * deltaTexS, (j+1) * deltaTexT);
        ctD = (i * deltaTexS, (j+1) * deltaTexT);
```

O processo para o cálculo das coordenadas de textura nas funções `geraPlanoX` e `geraPlanoZ` é similar a este, variando apenas as divisões nos eixos respetivos.

2.2 Caixa

Nesta fase do projeto permite-se a criação de dois tipos de caixa: a caixa "normal" (descrita na Fase 1) e a caixa invertida, cuja diferença em relação à primeira é a orientação das faces que passa a ser para o "interior". Para gerar uma caixa o utilizador deverá introduzir um comando com as seguintes sintaxes, respetivamente:

```
gerador box comprimento largura altura divsx divsz divsy
ficheiro.3d

gerador invertedbox comprimento largura altura divsx divsz
divsy ficheiro.3d
```

O resultado deste comando é a criação de uma caixa centrada no ponto (0,0,0) com o comprimento, largura e altura indicados.

Para gerar a "box" e a "invertedbox" recorre-se à função `geraCaixa`, sendo que a orientação passada como parâmetro no primeiro caso `CIMA` para que as faces sejam visíveis do exterior da caixa. Para gerar a "invertedbox" a orientação passada como argumento é `BAIXO` e as faces passam a ser visíveis do interior da caixa.

Uma vez que uma caixa corresponde a 6 planos, para desenhar a caixa calculou-se o centro de cada uma das suas faces (já apresentado na Fase 1) e utilizaram-se as primitivas da secção 2.1 que geram planos XY, YZ e XZ dado um centro e uma orientação.

2.3 Circulo

Considere-se os pontos A e B de uma circunferência que tem centro O:

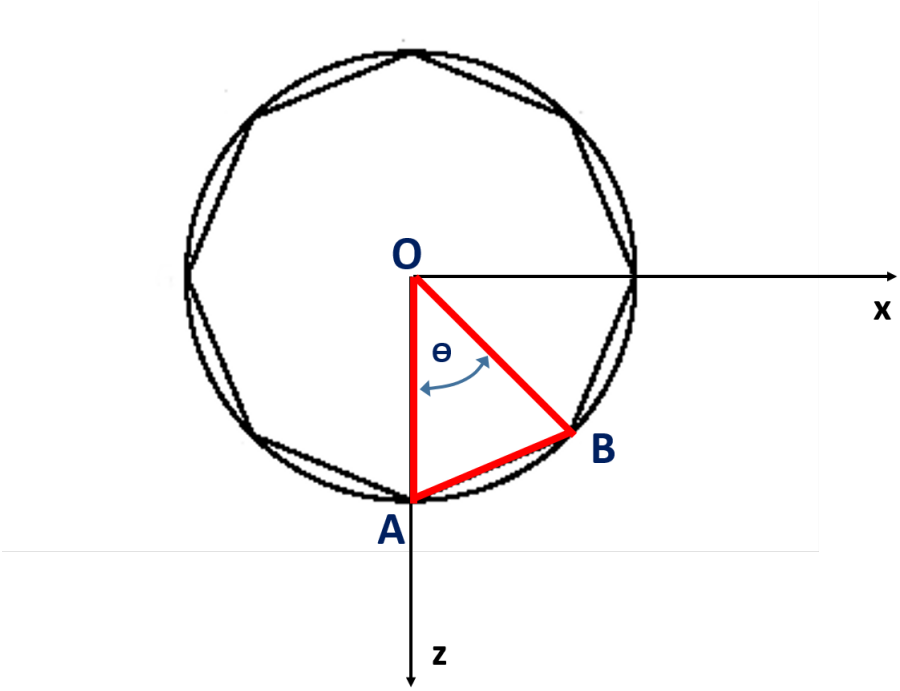


Figura 2.2: Pontos A, B e O do círculo

Recorde-se que nas fases anteriores estes pontos eram calculados da seguinte forma pela função `geraCirculo` (o ponto o é passado como argumento à função assim como o raio):

```
float deltaAz = (float)2 * M_PI / fatias;
for (int i = 0; i < fatias; ++i) {
    A = CoordsPolares(o, raio, deltaAz*i).toCartesianas();
    B = CoordsPolares(o, raio, deltaAz*(i + 1)).toCartesianas();

    if (orientacao == CIMA || orientacao == AMBOS) {
        pontos.push_back(A);
        pontos.push_back(B);
        pontos.push_back(o);
    }

    if (orientacao == BAIXO || orientacao == AMBOS) {
        pontos.push_back(A);
        pontos.push_back(o);
        pontos.push_back(B);
    }
}
```

Tal como acontece no plano, à função `geraCirculo` também pode ser passada uma orientação CIMA caso se pretenda que o círculo esteja virado no sentido positivo do eixo dos y ou BAIXO caso o círculo esteja virado para o sentido negativo do eixo dos y.

2.3.1 Cálculo das normais

Todos os pontos do círculo terão a mesma normal, dependendo se o círculo está virado para baixo ou para cima. Se estiver virado para cima a normal dos pontos do círculo será o vector $\overrightarrow{up} = (0, 1, 0)$, caso contrário será o vector $\overrightarrow{down} = (0, -1, 0)$

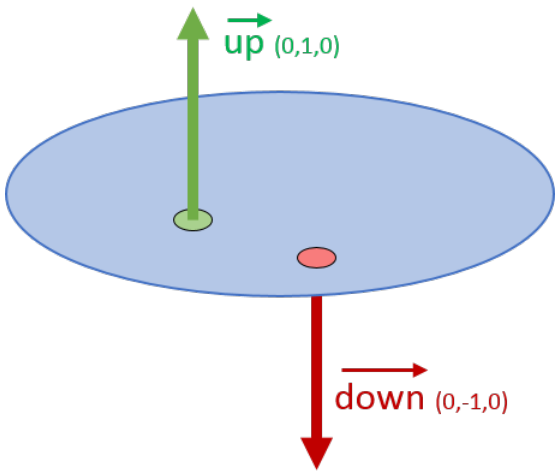


Figura 2.3: Vetores $\overrightarrow{up}$ e $\overrightarrow{down}$ constituem as normais do circulo caso este se encontre virado para cima ou para baixo, respetivamente

Juntando as normais ao código anterior ficamos com:

```
float deltaAz = (float)2 * M_PI / fatias;
Coordenadas3D up = Coordenadas3D{ 0,1,0 };
Coordenadas3D down = Coordenadas3D{ 0,-1,0 };

for (int i = 0; i < fatias; ++i) {
    A = CoordsPolares(o, raio, deltaAz*i).toCartesianas();
    B = CoordsPolares(o, raio, deltaAz*(i + 1)).toCartesianas();

    if (orientacao == CIMA || orientacao == AMBOS) {
        pontos.push_back(A);
        normais.push_back(up);

        pontos.push_back(B);
        normais.push_back(up);

        pontos.push_back(o);
        normais.push_back(up);
    }

    if (orientacao == BAIXO || orientacao == AMBOS) {
        pontos.push_back(A);
        normais.push_back(down);
```

```
            pontos.push_back(o);
            normais.push_back(down);

            pontos.push_back(B);
            normais.push_back(down);
        }
}
```

2.3.2 Cálculo das coordenadas de textura

Para o cálculo das coordenadas de textura foi usado como referência o template da figura 2.4.

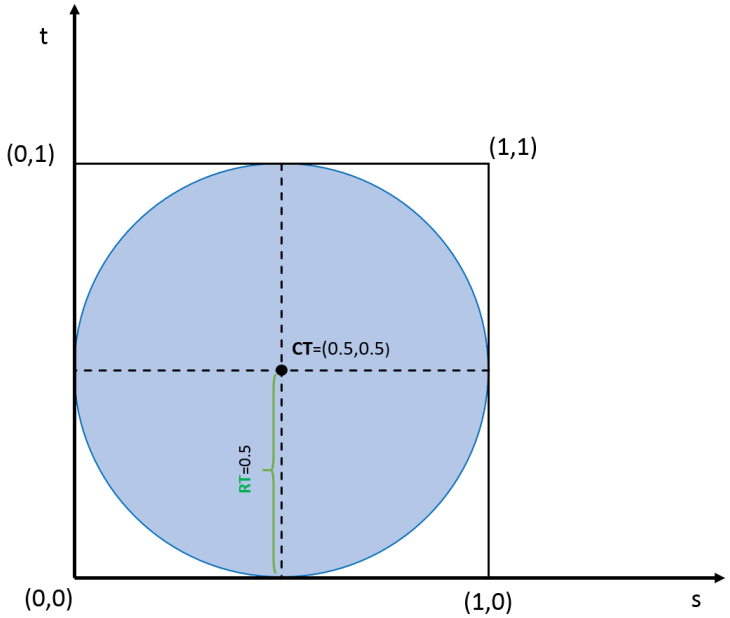


Figura 2.4: Template para coordenadas de textura do círculo

O centro do círculo do template fica nas coordenadas $CT = (0.5, 0.5)$ e tem de raio $RT = 0.5$. Com isto, é possível calcular as coordenadas de textura através de coordenadas polares. As equações para calcular s e t de um ponto de uma circunferência raio r e centro $C(c_s, c_t)$ com ângulo θ em relação ao eixo t são:

$$s = c_s \times r * \cos(\theta) \tag{2.1}$$

$$t = c_t \times r * \cos(\theta) \tag{2.2}$$

Aplicando estas equações aos pontos A, B e O referidos anteriormente, temos:

```
float deltaAz = (float) (2.0f * M_PI) / fatias;
Coordenadas3D A, B;
CoordsTextura ctA, ctB, ctO;
Coordenadas3D up = Coordenadas3D{ 0,1,0 };
Coordenadas3D down = Coordenadas3D{ 0,-1,0 };
```

```
for (int i = 0; i < fatias; ++i) {
    A = CoordsPolares(o, raio, deltaAz*i).toCartesianas();

    float s1 = 0.5f + 0.5*cos((deltaAz*i));
    float t1 = 0.5f + 0.5*sin((deltaAz*i));

    ctA = CoordsTextura{ s1,t1 };

    B = CoordsPolares(o, raio, deltaAz*(i + 1)).toCartesianas();

    float s2 = 0.5f + 0.5f*cos((deltaAz*(i + 1)));
    float t2 = 0.5f + 0.5f*sin((deltaAz*(i + 1)));
    ctB = CoordsTextura{s2,t2};

    ctO = CoordsTextura{ 0.5f,0.5f };

    //Coloca pontos, normais e coordenadas de textura
    //nas estruturas ...
}
```

2.4 Cilindro

Considere-se os pontos A, B, C e D da superfície lateral do cilindro:

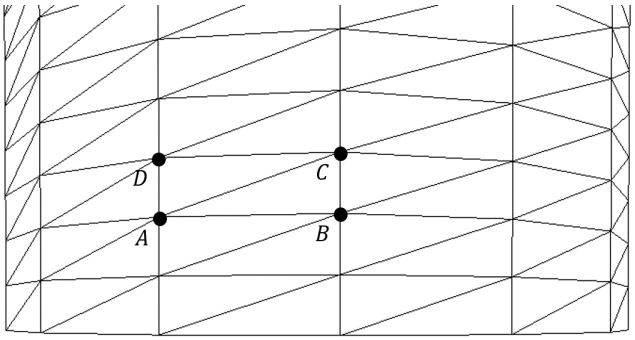


Figura 2.5: Pontos A, B, C e D da superfície lateral do cilindro

Recorde-se das fases anteriores que estes pontos foram gerados com recurso a coordenadas polares de acordo com o seguinte pseudo-codigo:

```
for (int j = 0; j < camadas; ++j) {
    Coordenadas3D cB = { o.x, o.y + deltaAlt*(j) , o.z };
    Coordenadas3D cA = { o.x, o.y + deltaAlt*(j+1) , o.z };

    for (int i = 0; i < fatias; i++) {
      CoordsPolares a = CoordsPolares(cB, raio, deltaAz*i);
      CoordsPolares b = CoordsPolares(cB, raio, deltaAz*(i+1));
```

```
        CoordsPolares c = CoordsPolares(cA, raio, deltaAz*(i+1));
        CoordsPolares d = CoordsPolares(cA, raio, deltaAz*i);

        //Coloca pontos na estrutura para serem desenhados...
    }
}
```

2.4.1 Cálculo das normais

Bases do cilindro

As bases do cilindro correspondem a círculos. A base de cima é um circulo virado para cima e a base de baixo é um círculo virado para baixo, conforme apresentado na secção 2.3. As normais das bases são por isso o vector $\overrightarrow{up} = (0, 1, 0)$ e $\overrightarrow{down} = (0, -1, 0)$ respetivamente.

Superficie lateral do cilindro

Para um ponto P na superficie lateral do cilindro e para um ponto C no eixo de revolução do cilindro à mesma altura que o ponto P, a normal do ponto P é dada por $\overrightarrow{N} = P - C$.

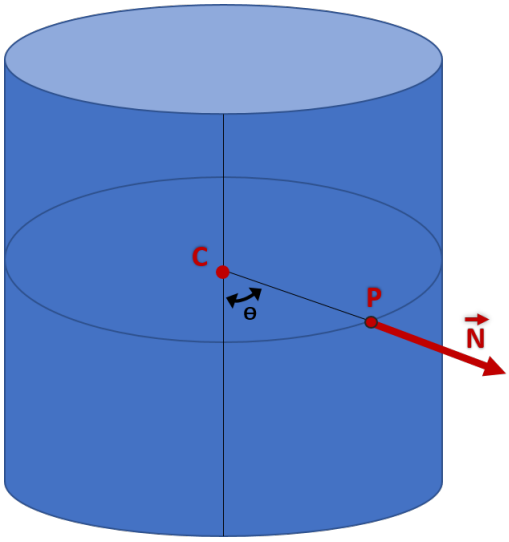


Figura 2.6: Normal da superfície lateral do cilindro

O cilindro é desenhado camada a camada e dentro de cada camada fatia a fatia. Em cada camada os pontos D e C da figura 2.5 ficam num nível mais elevado que os pontos A e B. Assim, em cada camada existem na verdade dois pontos C a considerar. Ao ponto que fica no eixo de revolução do cilindro que se encontra à mesma altura dos pontos A e B designaremos `cB` e ao ponto que fica no eixo de revolução do cilindro que se encontra à mesma altura dos pontos C e D desinaremos por `cA`.

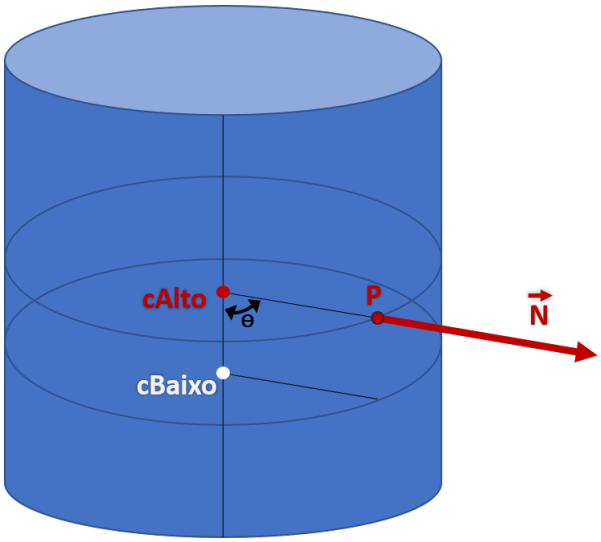


Figura 2.7: Normal da superfície lateral do cilindro

Temos assim o seguinte pseudo-código para o calculo das normais da superfície lateral do cilindro:

```
for (int j = 0; j < camadas; ++j) {
    Coordenadas3D cB = { o.x, o.y + deltaAlt*(j) , o.z };
    Coordenadas3D cA = { o.x, o.y + deltaAlt*(j+1) , o.z };
    for (int i = 0; i < fatias; i++) {
      CoordsPolares a = CoordsPolares(cB, raio, deltaAz*i);
      CoordsPolares b = CoordsPolares(cB, raio, deltaAz*(i+1));
      CoordsPolares c = CoordsPolares(cA, raio, deltaAz*(i+1));
      CoordsPolares d = CoordsPolares(cA, raio, deltaAz*i);

      Coordenadas3D nb = b.toCartesianas() - cB;
      Coordenadas3D nc = c.toCartesianas() - cA;
      Coordenadas3D na = a.toCartesianas() - cB;
      Coordenadas3D nd = d.toCartesianas() - cA;

      //Coloca pontos e normais nas estruturas
    }
}
```

2.4.2 Cálculo das coordenadas de textura

Para o cálculo das coordenadas de textura foi usado como referência o template da figura 2.8.

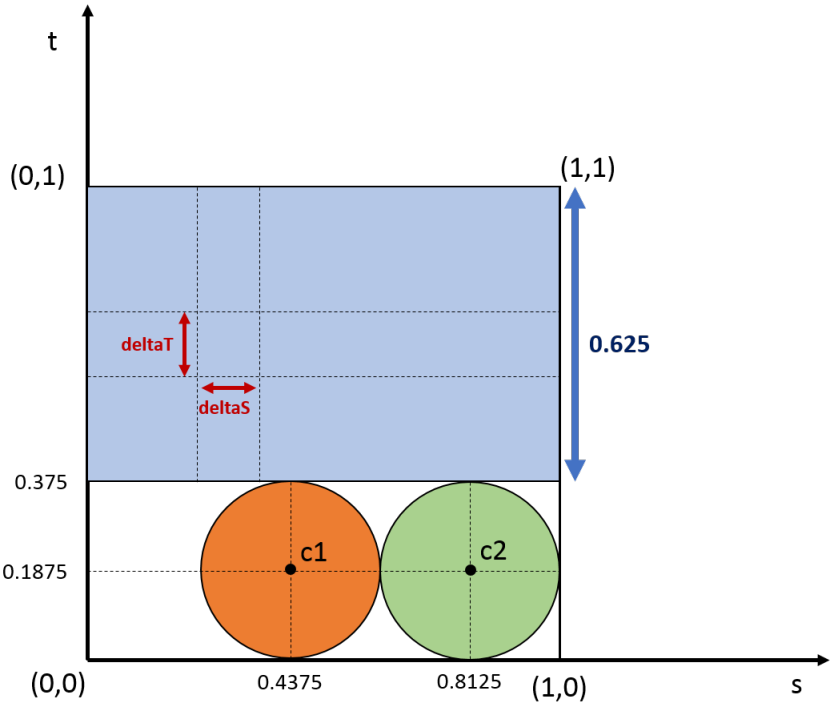


Figura 2.8: Template para coordenadas de textura do cilindro

Superfície lateral

A textura a aplicar à superfície lateral do cilindro corresponde à área azul da figura 2.8. Para associar cada ponto da superfície lateral do cilindro a um ponto na área azul dividiu-se a altura do rectângulo azul (0.625) pelo número de camadas e o comprimento (1) pelo número de fatias, dando origem a `deltaT` e `deltaS` respetivamente.

$$deltaT = 0.625/camadas$$

$$deltaS = 1/fatias$$

Estes valores permitem assim saber qual a variação de s e t no espaço das texturas quando se muda para uma fatia "ao lado" ou uma camada "acima" no cilindro. Assim, um ponto que esteja na fatia i da camada j, tem como coordenadas de textura:

$$s = deltaS * i$$

$$t = 0.375 + deltaT * j$$

Note-se que para t é necessário somar 0.375 ao resultado da multiplicação de `deltaT` pela camada j. Isto deve-se ao facto de a área azul da superfície lateral começar em $t = 0.375$, como se pode ver na figura 2.8. Tem-se assim o seguinte código relativamente às normais da superfície lateral do cilindro.

```
float deltaTexS = (float) 1.0f/fatias;
float deltaTexT = (float) (1.0f - 0.375f)/camadas;

for (int j = 0; j < camadas; ++j) {
```

```
    for (int i = 0; i < fatias; i++) {

    textCoords.push_back(CoordsTextura{deltaTexS * i, deltaTexT
        * j + 0.375f});

    textCoords.push_back(CoordsTextura{deltaTexS * (i+1),
       deltaTexT * j + 0.375f});

    textCoords.push_back(CoordsTextura{deltaTexS * (i+1),
       deltaTexT * (j+1) + 0.375f});

    textCoords.push_back(CoordsTextura{deltaTexS * i, deltaTexT
        * j + 0.375f});

    textCoords.push_back(CoordsTextura{deltaTexS * (i+1),
       deltaTexT * (j+1) + 0.375f});

    textCoords.push_back(CoordsTextura{deltaTexS * i, deltaTexT
        * (j+1) + 0.375f});

    //Poe pontos e normais na estrutura para serem desenhados
    }
}
```

Base inferior e superior

A base superior do cilindro é representada pelo círculo laranja da figura 2.8enquanto que a base inferior é representada pelo círculo verde. As coordenadas de textura em ambos os casos podem ser obtidas através de coordenadas polares de forma semelhante à que foi explicada na secção 2.3.2. As equações para calcular s e t de um ponto de uma circunferência raio r e centro $C(c_s, c_t)$ com ângulo θ em relação ao eixo t são:

$$s = c_s \times r * \cos(\theta) \tag{2.3}$$

$$t = c_t \times r * \cos(\theta) \tag{2.4}$$

Os centros dos círculos da base superior e inferior são dados por $C_1 = (0.4375, 0.1875)$ e $C_2 = (0.8125, 0.1875)$. Ambos os círculos têm raio $r = 0.1875$. Tem-se assim o seguinte código relativamente às coordenadas de textura da base superior e inferior do cilindro:

```
float deltaAz = (float) (2.0f * M_PI) / fatias;
float deltaAlt = (float) altura/camadas;
Coordenadas3D A, B;
CoordsTextura ctA, ctB;

CoordsTextura ctOrigemCima = CoordsTextura{0.4375f, 0.1875f};
CoordsTextura ctOrigemBaixo = CoordsTextura{0.8125f, 0.1875f};
```

```
float raioTex = 0.1875;

Coordenadas3D oCima = Coordenadas3D{ o.x, o.y + altura, o.z };

// Circulo superior
for (int i = 0; i < fatias; ++i) {
    A = CoordsPolares(oCima, raio, deltaAz*i).toCartesianas();
    float s1 = ctOrigemCima.s + raioTex*cos((deltaAz*i));
    float t1 = ctOrigemCima.t + raioTex*sin((deltaAz*i));

    ctA = CoordsTextura{ s1,t1 };

    B = CoordsPolares(oCima, raio, deltaAz*(i+1)).toCartesianas();

    float s2 = ctOrigemCima.s + raioTex*cos((deltaAz*(i+1)));
    float t2 = ctOrigemCima.t + raioTex*sin((deltaAz*(i+1)));

    ctB = CoordsTextura{s2,t2};

    //Coloca pontos,  normais e coordenadas de textura
    //nas estruturas para serem desenhados
}

// Círculo inferior
for (int i = 0; i < fatias; ++i) {
    A = CoordsPolares(o, raio, deltaAz*i).toCartesianas();
    float s1 = ctOrigemBaixo.s + raioTex*cos((deltaAz*i));
    float t1 = ctOrigemBaixo.t + raioTex*sin((deltaAz*i));

    ctA = CoordsTextura{ s1,t1 };

    B = CoordsPolares(o, raio, deltaAz*(i+1)).toCartesianas();

    float s2 = ctOrigemBaixo.s + raioTex*cos((deltaAz*(i+1)));
    float t2 = ctOrigemBaixo.t + raioTex*sin((deltaAz*(i+1)));
    ctB = CoordsTextura{s2,t2};

    //Coloca pontos,  normais e coordenadas de textura
    //nas estruturas para serem desenhados
}
```

2.5 Cone

Considere-se os pontos A, B, C e D da superfície lateral do cone:

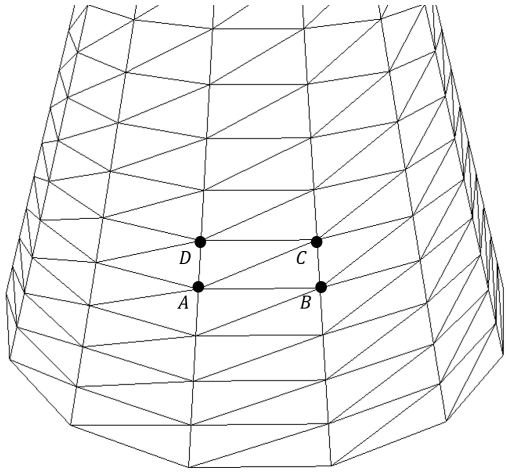


Figura 2.9: Pontos A, B, C e D da superfície lateral do cone

Recorde-se das fases anteriores que estes pontos foram gerados com recurso a coordenadas polares de acordo com o seguinte pseudo-codigo:

```
for (int j = 0; j < camadas; ++j) {
    Coordenadas3D cBaixo = {o.x,o.y+deltaAltura*(j) , o.z };
    Coordenadas3D cAlto = { o.x, o.y + deltaAltura*(j+1) , o.z
       };
    for (int i = 0; i < fatias; ++i) {
        A = CoordsPolares(Coordenadas3D{ o.x , o.y +
           deltaAltura*j,o.z},
        raio - (deltaRaio*j),
        deltaAz*i).toCartesianas();
        B = CoordsPolares(Coordenadas3D{ o.x , o.y +
           deltaAltura*j, o.z },
        raio - (deltaRaio*j),
        deltaAz*(i + 1)).toCartesianas();
        C = CoordsPolares(Coordenadas3D{ o.x , o.y +
           deltaAltura*(j + 1), o.z },
        raio - (deltaRaio*(j + 1)),
        deltaAz*(i + 1)).toCartesianas();
        D = CoordsPolares(Coordenadas3D{ o.x , o.y +
           deltaAltura*(j + 1), o.z },
        raio - (deltaRaio*(j + 1)),
        deltaAz*i).toCartesianas();

        //Coloca pontos,  normais e coordenadas de textura
        //nas estruturas para serem desenhados
    }
}
```

2.5.1 Cálculo das normais

Base do cone

A base do cone corresponde a um círculo virado para baixo, pelo que todos os pontos na base do cone terão como normal o vector $\overrightarrow{down} = (0, -1, 0)$ conforme visto na secção 2.3.

Superfície lateral do cone

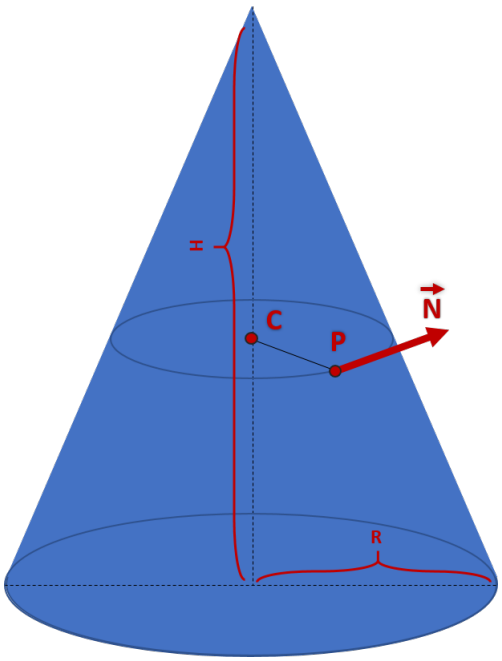


Figura 2.10: Normais cone

Podemos pensar na normal do cone como sendo uma normal semelhante à normal do cilindro mas ligeiramente inclinada para cima. De facto, a normal do cone tem as mesmas componentes em x e em z da normal do cilindro, que como visto na secção 2.4 é dada por $\overrightarrow{NC} = P - C$. A "inclinação" da normal do cone é dada pela componente y da normal, e pode ser calculada pelo quociente entre o raio e a altura. Assim temos que a normal $\overrightarrow{N}$ da superfície lateral do cone é dada por:

$$\overrightarrow{N} = (NC_x, raio/altura, NC_Z) \tag{2.5}$$

2.5.2 Cálculo das coordenadas de textura

Para o cálculo das coordenadas de textura foi usado como referência o template da figura 2.11.

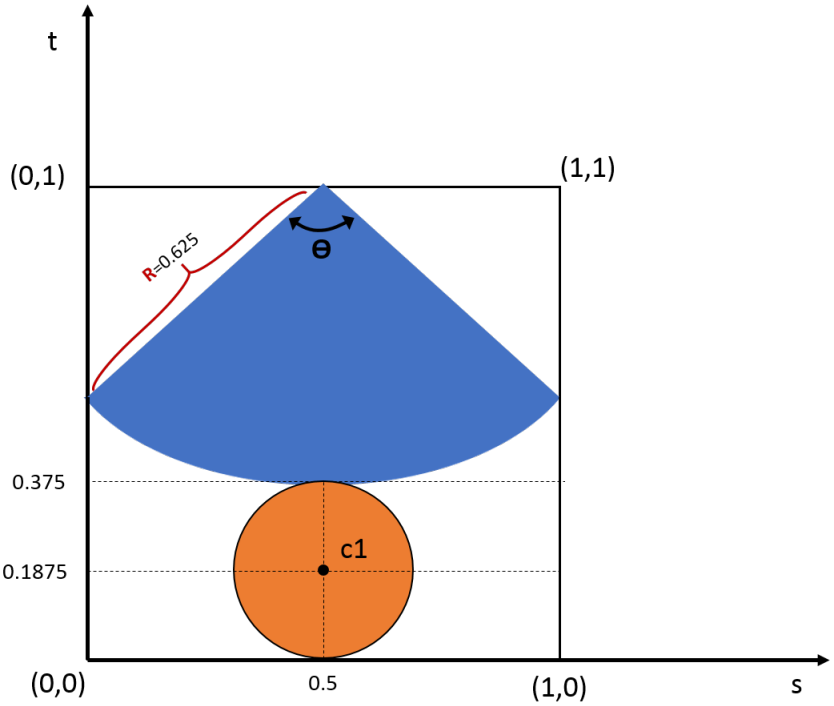


Figura 2.11: Template para coordenadas de textura do cone

Base do cone

As coordenadas de textura da base do cone correspondem a área laranja da figura 2.11 e são obtidas através de coordenadas polares de forma semelhante à explicada nas secções 2.3.2 e 2.4.2.

Superfície lateral

Para calcular as coordenadas de textura da superfície lateral do cone, numa primeira fase calculou-se o valor do ângulo θ, que pode ser obtido a partir do raio e do comprimento do arco circular. Seja l o comprimento do arco circular e R o raio, então o ângulo θ do arco circular é dado por:

$$\theta = \frac{l}{R} \tag{2.6}$$

Sabe-se que o comprimento l do arco da circunferência é equivalente ao perímetro da circunferência da base do cone, e o R é dado por $1 - 0.375 = 0.625$. Então:

$$\theta = \frac{2 \times \pi \times 0.1875}{0.625} \approx 108° \tag{2.7}$$

Depois de obtido o valor de θ, e tendo em consideração os pontos A,B,C e D representados na figura 2.9, do ponto A para o ponto B a variação do ângulo é dada por 2.8 e a variação do ponto A para o ponto D é dado por 2.9.

$$delta\theta = \frac{\theta}{fatias} \tag{2.8}$$

$$deltaR = \frac{R}{camadas} \tag{2.9}$$

A partir daqui para obter as coordenadas dos pontos basta considerar um circulo trigonométrico que tem centro em $C(0.5, 1)$, representado na figura 2.12 e obter as coordenadas usando as fórmulas do circulo trigonométrico.

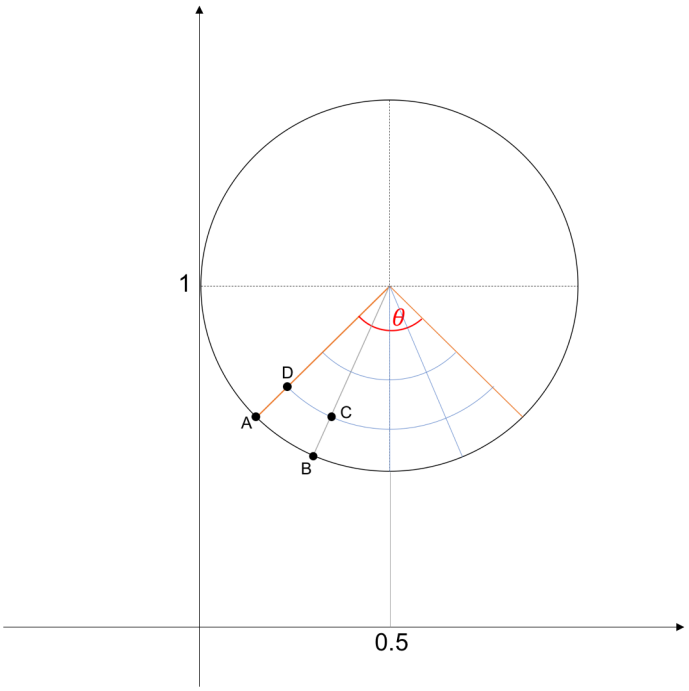


Figura 2.12: Superficie lateral do cone no circulo trigonométrico com centro em (0.5,1)

O ângulo inicial é dado por:

$$alfaInicial = \frac{3\pi}{2} - \frac{\theta}{2} \tag{2.10}$$

Ou seja, as coordenadas do ponto A serão dadas por:

$$R \times cos(alfaInicial), R \times sen(alfaInicial)$$

Com toda informação referida acima, podemos concluir que o pseudo-código para as coordenadas de textura de todos os pontos A, B, C e D pode ser dado por:

```
for (int j = 0; j < camadas; ++j) {
    for (int i = 0; i < fatias; ++i) {
        A = ((R - deltaR * j) * cos(alfaInicial + delta\theta *
            i) + 0.5, (R - deltaR * j) *sen(alfaInicial + delta
           \theta * i) + 1);
        B = ((R - deltaR * j) * cos(alfaInicial + delta\theta *
            (i+1)) + 0.5, (R - deltaR * j) *sen(alfaInicial +
           delta\theta * (i+1)) + 1);
        C = ((R - deltaR * (j+1)) * cos(alfaInicial + delta\
           theta * (i+1)) + 0.5, (R - deltaR *(j+1)) * sen(
           alfaInicial + delta\theta * (i+1)) + 1);
        D = ((R - deltaR * (j+1)) * cos(alfaInicial + delta\
           theta * i) + 0.5, (R - deltaR * (j+1))* sen(
           alfaInicial + delta\theta * i) + 1);
```

2.6 Esfera

Dados os pontos A, B, C e D na superfície da esfera e o a origem (Figura 2.13), tem-se que:

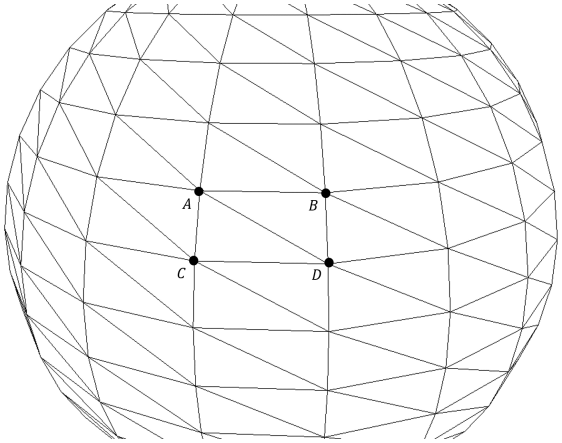


Figura 2.13: Pontos A, B, C e D na superfície da esfera

```
deltaAz = 2 * PI / fatias;
deltaPolar = PI / camadas;

for (int j = 0; j < camadas; ++j) {
    for (int i = 0; i < fatias; ++i) {
        A = (raio, deltaAz*i, deltaPolar*j) + o;
        B = (raio, deltaAz*(i+1), deltaPolar*j) + o;
        C = (raio, deltaAz*i, deltaPolar*(j+1)) + o;
        D = (raio, deltaAz*(i+1), deltaPolar*(j+1)) + o;
    }
}
```

2.6.1 Cálculo das normais

O cálculo das normais da esfera foi simples de calcular, dado que a função que gerava os pontos dos triângulos recebe como argumento a origem da figura, a normal em cada um desses pontos pode ser calcula da seguinte forma, tendo em conta a Figura 2.14.

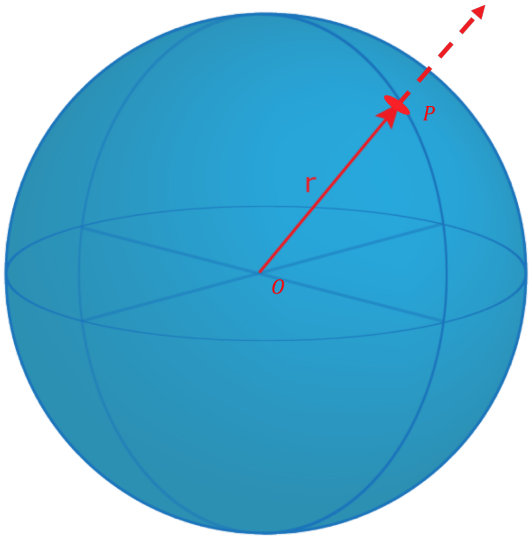


Figura 2.14: Exemplo de cálculo da normal num ponto da esfera

Dado o ponto $O(x_0, y_0, z_0)$ na origem do referencial da figura, a normal num qualquer ponto $P(x_1, y_1, z_1)$ da superfície da esfera pode ser calcula como sendo o vetor $\overrightarrow{OP}$, ou seja:

$$normal_P = \overrightarrow{OP} = P(x_1, y_1, z_1) - O(x_0, y_0, z_0) = (x_1 - x_0, y_1 - y_0, z_1 - z_0)$$

Portanto, a normal nos pontos A, B, C e D, respetivamente na, nb, nc e nd, pode ser calculada como:

```
na = A - o;
nb = B - o;
nc = C - o;
nd = D - o;
```

Estes pontos são guardados num `Vector` de `Coordenadas3D`, tal como os pontos dos triângulos.

2.6.2 Cálculo das coordenadas de textura

Em relação ao cálculo das coordenadas de textura, tendo em conta que a esfera está a ser construída "fatia a fatia", tem-se que estas podem calculadas através de:

```
ctA = (deltaAz*i / 2*PI, 1 - deltaPolar*j / PI);
ctB = (deltaAz*(i+1) / 2*PI, 1 - deltaPolar*j / PI});
ctC = (deltaAz*i / 2*PI, 1 - deltaPolar*(j+1) / PI});
ctD = (deltaAz*(i+1) / 2*PI, 1 - deltaPolar*(j+1) / PI});
```

Estas coordenadas são guardados num `Vector` de `CoordsTextura`. De notar que elas terão que estar no intervalo $[-1, 1]$ e, portanto, terão que ser divididas por $^2/_\pi$ no caso da coordenada s da textura (camadas) e por π no caso da coordenada t da textura (fatias).

2.7 Anel

Considere-se os pontos A, B, C e D na superfície do anel e o o centro (Figura 2.15):

```
deltaAz = 2*PI / fatias;
for (int i = 0; i < fatias; ++i) {
    A = (o, raioInterno, deltaAz*i);
    B = (o, raioExterno, deltaAz*i);
    C = (o, raioInterno, deltaAz*(i + 1));
    D = (o, raioExterno, deltaAz*(i + 1));
```

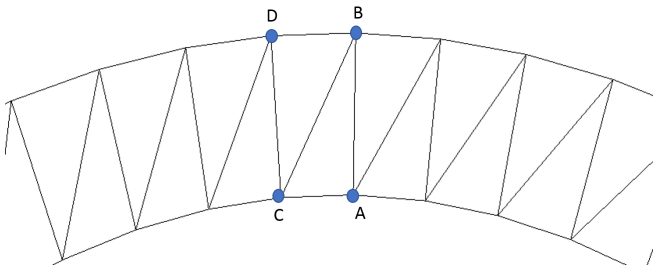


Figura 2.15: Pontos A, B, C e D na superfície do anel

2.7.1 Cálculo das normais

A normal em qualquer ponto do anel é o vetor $\overrightarrow{up}$ ou $\overrightarrow{down}$, conforme a orientação da figura. Se a figura estiver "PARA_CIMA" o vetor normal será o $\overrightarrow{up}$, se estiver "PARA_BAIXO" o vetor normal será o $\overrightarrow{down}$ (Figura 2.16).

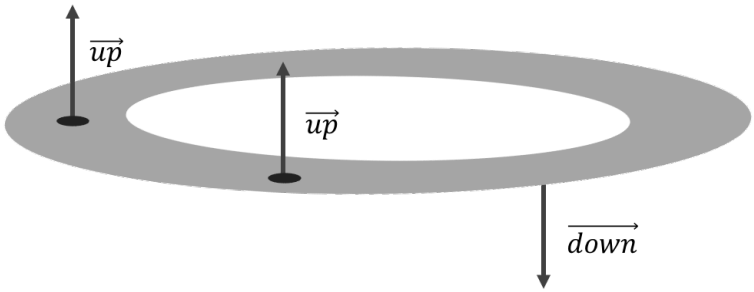


Figura 2.16: Exemplo de cálculo da normal num ponto do anel

2.7.2 Cálculo das coordenadas de textura

As coordenadas de textura do anel são calculadas da seguinte forma:

```
ctA = deltaAz*i / 2*PI, 1;
ctB = deltaAz*i / 2*PI, 0;
ctC = deltaAz*(i+1) / 2*PI, 1;
ctD = deltaAz*(i+1) / 2*PI, 0;
```

Sendo necessário as coordenadas de textura estar no intervalo $[-1, 1]$, é necessário dividir por $^2/_\pi$.

2.8 Superfície de Bézier

À semelhança das restantes figuras já apresentadas foi necessário adicionar o calculo das normais e das coordenadas de textura aos modelos gerados com base em *patches* de Bezier.

As funções necessárias para gerar o ficheiro com os pontos, as normais e as coordenadas de textura são : `lebezier` e `getPontos`. A primeira função já tinha sido apresentada na fase anterior e manteve-se inalterada. A função `getPontos` veio substituir a função `getTriangulos` dado que esta apenas devolvia um vector de pontos e agora passa a devolver uma instância da classe `ComponentesPonto`.

A classe `ComponentesPonto` é constituída pelas seguintes variáveis de instância:

```
class ComponentesPonto {
public:
    std::vector<Coordenadas3D> pontos;
    std::vector<Coordenadas3D> normais;
    std::vector<CoordsTextura> coordsTextura;
};
```

Decidiu-se criar esta classe apenas por uma questão de organização uma vez que permite na mesma função retornar os pontos, as normais e as coordenadas de textura através de uma instância desta classe.

2.8.1 Cálculo das normais

O calculo das normais foi feito recorrendo às tangentes. Para cada ponto à superfície (*B(u,v)*) foi calculada a derivada em ordem a *u* e em ordem a *v* e feito o produto externo destes dois vetores. O vetor resultante normalizado trata-se da normal nesse ponto. Este raciocínio pode ser refletido nas seguintes formulas matemáticas:

$$\frac{\partial B(u,v)}{\partial u} = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

$$\frac{\partial B(u,v)}{\partial v} = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

Este calculo da normal é realizado pela função `calculaNormal`, definida em `SuperficieBezier.h`. Esta função recebe como argumentos uma matriz que contem o resultado de MPM^T (cálculo efetuado na função `getPontos`) e os valores de *tesselation* `u` e `v`.

A função pode ser representada pelo seguinte pseudo código:

```
Coordenadas3D calculaNormal(Coordenadas3D r[4][4], float u,
   float v){
    Coordenadas3D du,dv,dr;
    Calcula derivada em ordem a u e guarda o resultado em du;
    Calcula derivada em ordem a v e guarda o resultado em dv;
    Calcula produto externo du por dv e guarda resultado em dr;
    return dr.normalize();
}
```

2.8.2 Cálculo das coordenadas de textura

As coordenadas de textura para cada ponto da superfície são dadas pela valor de *tesselation* em ordem a *u* e *v* nesse ponto. Deste modo, na função `getPontos` apenas é necessário colocar no vector de *CoordsTextura*.

O seguinte excerto representa um possível pseudo código da função `getPontos`:

```
ComponentesPonto getPontos(int divsU, int divsV) {
    ComponentesPonto cp;
    Para cada patch {
        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;

                Calcula Pontos;
                Calcula Normais;
                Coloca as seguintes CoordsTextura
                 no vector cp.coordsTextura:

                // Triangulo 1
                CoordsTextura{u1, v1};
                CoordsTextura{u1, v2};
                CoordsTextura{u2, v1};

                //Triangulo 2
                CoordsTextura{u2, v1};
                CoordsTextura{u1, v2};
                CoordsTextura{u2, v2};
            }
        }
    }
    return cp;
}
```

3. Motor

3.1 Luz

Para a implementação da funcionalidade de definir vários tipos luz por parte do MOTOR e representa-los na cena, decidiu-se criar uma classe que representa uma "Luz" e que a defina com todos os parâmetros necessários.

Começando pela luz, tem-se que a classe `Light` é definida da seguinte forma:

```
class Light {
public:
    string type;
    GLfloat amb[4] = {0.2f, 0.2f, 0.2f, 1.0f};
    GLfloat diff[4] = {1.0, 1.0, 1.0, 1.0};
    GLfloat pos[4] = {0.0, 0.0, 0.0, 1.0};
    GLfloat spotDir[3] = {0.0, 0.0, -1.0};
    GLfloat spec[4] = {0.0, 0.0, 0.0, 0.0};
    GLfloat cutoff = 180.0;
    GLfloat attenuation[3] = {1.0, 0.0, 0.0};
    GLfloat exponent = 0.0;
```

Os *arrays* `amb`, `diff` e `pos` definem a *directional light*, *point light* e *spotlight*. No caso desta última, é necessário também definir `spotDir`, `cutoff` e `attenuation` (esta relacionada com o `exponent`). Todos estes valores são usados na definição das luzes e em instruções explicitadas de seguida.

Na função que faz a leitura do ficheiro .XML são guardadas num `Vector` de `Light` todas as luzes encontradas. Para tal foi criada a função `XMLtoLights`, cujo pseudo-código pode ser analisado de seguida.

```
void XMLtoLights(xml_node luzes) {
    Para todas as luzes no ficheiro:
        Light l;
        Para todos os atributos da luz:
            Guardar tipo de luz
            Se tipo for "DIRECTIONAL":
                Fazer l.pos[3] = 0.0;
            Guardar se existir:
                componentes da posX
                componentes de diff
                componentes de amb
                componentes de spotdir
```

```
                cutoff
                exponent
                attenuation
        }
        Guardar l
    }
}
```

Depois de guardadas todas as luzes, é necessário ativá-las através do `glEnable(GL_LIGHTING)` e de `glEnable(GL_LIGHT0 + i)`, com $i \in [0,$luzes.size$]$.

Na função `renderScene` é usado o método `apply` da classe `Lights` a cada uma das luzes. Esta função pode ser definida como:

```
void apply(int light) {
    glLightfv(GL_LIGHT0 + light, GL_POSITION, pos);
    glLightfv(GL_LIGHT0 + light, GL_DIFFUSE, diff);
    glLightfv(GL_LIGHT0 + light, GL_AMBIENT, amb);
    glLightfv(GL_LIGHT0 + light, GL_SPECULAR, spec);
    glLightf(GL_LIGHT0 + light, GL_CONSTANT_ATTENUATION,
             attenuation[0]);
    glLightf(GL_LIGHT0 + light, GL_LINEAR_ATTENUATION,
             attenuation[1]);
    glLightf(GL_LIGHT0 + light, GL_QUADRATIC_ATTENUATION,
             attenuation[2]);
    if (type == "SPOTLIGHT") {
        glLightfv(GL_LIGHT0 + light, GL_SPOT_DIRECTION,
                  spotDir);
        glLightf(GL_LIGHT0 + light, GL_SPOT_CUTOFF,
                 spotCutoff);
        glLightf(GL_LIGHT0 + light, GL_SPOT_EXPONENT,
                 spotExponent);
    }
}
```

É também chamada a função `desenhaGrupo` que, no momento em que desenha as linhas das curvas de Catmull-Rom, desativa as luzes, com `glDisable(GL_LIGHTING)` e volta a ativá-las com `glEnable(GL_LIGHTING)` quando termina o desenha dessas linhas. Decidiu-se fazer esta definição para que as linhas das curvas apareçam na cena de forma homogénea e sem influência da luz.

Para que as figuras na cena sejam iluminadas pela luz, foi necessário ativar as normais nos pontos. Na função `main` do MOTOR foi ativado o *array* das normais através de: `glEnableClientState(GL_NORMAL_ARRAY)`. No momento em que são lidos os pontos, são também lidas as normais e guardadas num `Vector`, na função `XMLtoGrupo`.

Nesta é também gerado o *Buffer* e feito o *bind* do mesmo, através de:

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

O próximo passo é atribuir a semântica: na função `desenhaGrupo` é feito `glBindBuffer(GL_ARRAY_BUFFER, it->nBuffNormal)` e `glNormalPointer(GL_FLOAT, 0, 0)`, antes do `glDrawArrays()`.

3.2 Definição dos materiais

Para a implementação das funcionalidades do MOTOR foi necessário definir os materiais no desenho dos objetos, através do `glMaterial`.

Foram incluídos na classe `DefsDesenho` todas as componentes necessárias à definição dos materiais, nomeadamente:

```
float diffuse[4] = { 0.8f, 0.8f, 0.8f, 1.0f };
float ambient[4] = { 0.2f, 0.2f, 0.2f, 1.0f };
float specular[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
float emission[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
float shininess = 0.0f;
```

O primeiro passo a tomar é na função de leitura do XML que terá que guardar as propriedades dos materiais. Essas propriedades são definidas no nodo `models` e, para cada model, a sua obtenção segue o seguinte pseudo-codigo:

```
Guardar, se existir:
    Componente diff
    Componente amb
    Componente spec
    Componente emiss
    Componente amb e diff
    Shininess
```

Todas estas componentes são guardadas nas definições de desenho. As que não forem mencionada, são usadas as definições por defeito.

Depois de guardadas as informações do XML, passa-se às alterações feitas na função `desenhaGrupo`:

```
Para todos os desenhos do grupo, aplica-se o glMaterial:
    glMaterialfv(GL_FRONT, GL_DIFFUSE, def.diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT, def.ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, def.specular);
    glMaterialfv(GL_FRONT, GL_EMISSION, def.emission);
    glMaterialf(GL_FRONT, GL_SHININESS,def.shininess);
```

3.3 Texturas

Foram feitas alterações ao motor por forma a poder incluir texturas nos modelos lidos. Uma vez que o desenho é feito com recurso a VBO's foi necessário criar um array com as coordenadas de texturas, um *buffer* e copiar as coordenadas de textura desse array para o buffer. Deste modo, na `main` é efetuada a ativação do array (`glEnableClientState(GL_TEXTURE_COORD_ARRAY)`) e das funcionalidades das texturas (`glEnable(GL_TEXTURE_2D)`).

Na função `XMLtoGrupo`, função invocada pela função `leXML` e responsável por converter um nodo XML num `Grupo`, é realizado o carregamento da textura (caso seja passada como argumento no ficheiro XML), lidas as coordenadas de textura e feito o *bind* do buffer responsável pelas coordenadas de textura:

```
glGenBuffers(1, buffer_coords_textura);
glBindBuffer(GL_ARRAY_BUFFER, buffer_coords_textura[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) * desenho.
   pontos.size() * 2, &desenho.coordsText[0],
   GL_STATIC_DRAW);
desenho.nBuffTex = buffer_coords_textura[0];
```

Foi criada a função `loadTexture` que recebe como argumento o nome do ficheiro a carregar e devolve o identificador da textura. Esta função é chamada na leitura, caso surja o atributo "*texture*", e colocado o identificador na definição de desenho do respetivo modelo.

Lido o ficheiro XML e preenchidas as estruturas de dados com a informação necessária apenas falta desenhar as figuras. A função `desenhaGrupo`, invocada na `renderScene`, é responsável pela aplicação das transformações e pelo desenho da figura. Nesta função, para o desenho do modelo em si, são percorridas as definições de desenho, aplicadas essas mesmas definições, efetuado o *bind* da textura e dos buffers e finalmente desenhada a informação que está nos arrays. As instruções relativas à aplicação das texturas nesta função são:

```
glBindTexture(GL_TEXTURE_2D, it->idTex);

glBindBuffer(GL_ARRAY_BUFFER, it->nBuffTex);
glTexCoordPointer(2, GL_FLOAT, 0, 0);

glDrawArrays(it->defsDesenho.modoDesenho,0,it->pontos.size)

glBindTexture(GL_TEXTURE_2D, 0);
```

4. 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. Ao longo deste capítulo serão apresentadas as diferenças do ficheiro `sistema_solar.xml` relativamente às fases anteriores.

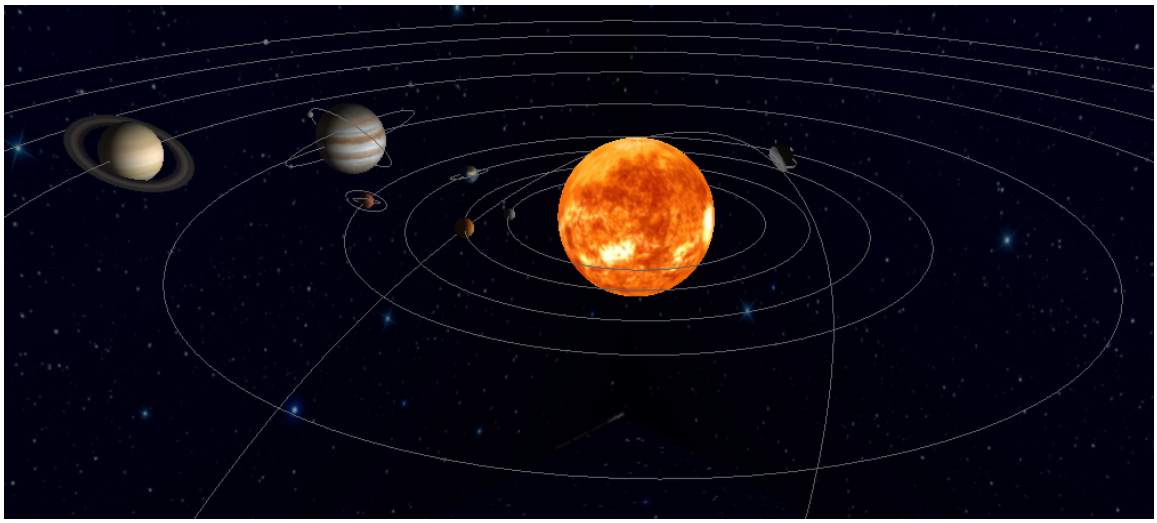


Figura 4.1: Representação geral do Sistema Solar

4.1 Iluminação

Para representar as luzes no ficheiro `.xml` foi criado o nodo do tipo `<lights>`, que é 'irmão' do nodo `<group name="Solar System»`. Decidiu-se que a luz presente no Sistema Solar é do tipo "Ponto luminoso" cuja posição será no centro do Sol, por isso o nodo 'filho' de `<lights>` carrega essa informação e foi definido da seguinte maneira:

```
<lights>
    <light type="POINT" posX="0" posY="0" posZ="0"/>
</lights>
```

4.2 Material

Uma vez que se pretende que conforme a estrutura a desenhar a mesma tenha diferentes propriedades de material, nesta secção serão explicitados quais as definições do material que foram acrescentadas ao ficheiro .XML para os diferentes tipos de estruturas que foram desenhadas.

4.2.1 Sol

Para ficar mais parecido com a realidade, definiu-se que o Sol seria um material emissor e por isso foram definidas as componentes emissivas do material da seguinte maneira:

```
<model mode="FILL" emR="1.0" emG="1.0" emB="1.0"
    file="esfera_50.3d" texture="texturas/2k_sun.jpg"/>
```

A componente $alpha$ não foi definida pois o seu valor é assumido por defeito.

Outra maneira de se ter alcançado este resultado seria ter criado uma esfera "especial" em que os vetores normais seriam definidos para o interior da esfera, no entanto, se aproveitarmos as capacidades das definições dos materiais conseguimos este resultado muito mais facilmente.

4.2.2 Planetas e Luas

A componente do material escolhida para este grupo de estruturas foi a difusa e, por isso, foram definidas as componentes difusas para todos os planetas e luas. No entanto, por uma questão estética, de modo a que estes ficassem um pouco mais iluminados e mais percetíveis, definiram-se também as componentes ambiente, conforme é demonstrado no seguinte exemplo:

```
<model mode="FILL" diffR="0.9" diffG="0.9" diffB="0.9"
    ambR="0.5" ambG="0.5" ambB="0.5"
    file="esfera_25.3d" texture="texturas/2k_saturn.jpg"/>
```

O resultado final desta definição pode ser visto no exemplo da Figura 4.2.

4.2.3 Cometa

Por último, relativamente ao cometa, optou-se para que a sua componente do material fosse especular, de modo a que o mesmo ficasse mais "brilhante", e definiram-se também as componentes para a luz ambiente de modo a que o cometa ficasse um pouco mais visível:

```
<model mode="FILL" specR="0.1" specG="0.1" specB="0.1"
    ambR="0.5" ambG="0.5" ambB="0.5"
    file="teapot.3d" texture="texturas/asteroid.jpg"/>
```

4.3 Texturas

Para cada uma das figuras desenhadas foi fornecida a respetiva textura, conforme o exemplo seguinte:

```
<models>
    <model mode="FILL" file="esfera_25.3d"
        texture="texturas/2k_saturn.jpg"/>
</models>
```

As texturas para cada uma das figuras foram obtidas nos sites sugeridos no enunciado. Na figura 4.2 apresentamos um exemplo do planeta Saturno com a respetiva textura aplicada.

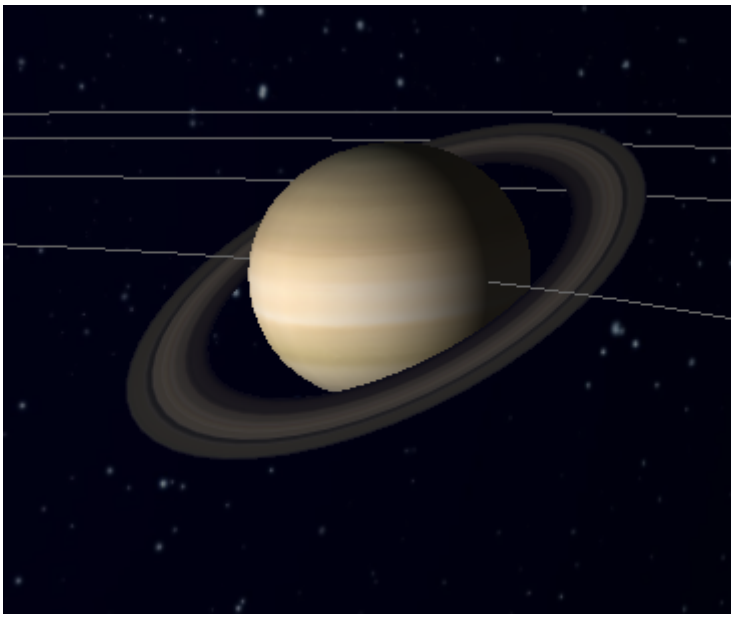


Figura 4.2: Planeta Saturno com a respetiva textura

4.4 Mapa de Estrelas

De modo a criar um "fundo" para o Sistema Solar decidiu-se que o mesmo seria desenhado dentro de uma caixa invertida (as faces são visíveis do lado de dentro da caixa). Assim, foi solicitado no gerador uma caixa com a $largura = 60$, $comprimento = 60$ e $altura = 60$, de modo a que o Sistema Solar ficasse completamente envolvido pela caixa e ainda houvesse algum espaço para a navegação da câmara.

Assim, no ficheiro .XML foi acrescentado o nodo `<group name="Stars">`, filho do nodo `"Sistema Solar"`, que vai ser responsável pelo desenho da referida caixa.

```
<group name="Stars">
    <models>
        <model mode="FILL" file="caixaInvertida.3d"
                texture="texturas/2k_stars.jpg"/>
    </models>
</group>
```

Neste modelo não foi necessário acrescentar as componentes do material, porque a figura usada é uma caixa invertida em que as normais têm o sentido para o interior da caixa. Assim, os valores do material vão ser os declarados por defeito, já mencionados na secção 3.2 e o resultado obtido é o pretendido.

4.5 Ficheiro de Teste

Uma vez que para a construção do Sistema Solar não são usadas todas as figuras que o gerador é capaz de produzir, decidiu-se criar um ficheiro `teste_figuras.XML` de teste em que todas as figuras são desenhadas de modo a comprovar que o gerador está correto. O resultado obtido pelo Motor é o seguinte:

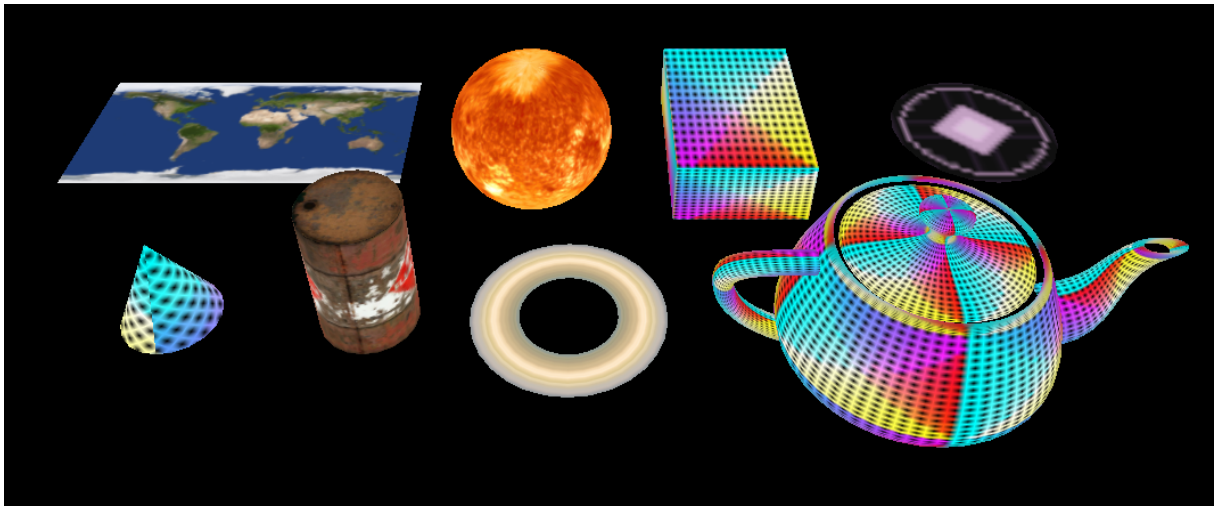


Figura 4.3: *Scene* de teste

Conclusão

Os objetivos propostos para esta fase do trabalho foram cumpridos. O gerador foi alterado de forma a poder gerar normais e coordenadas de textura para cada um dos pontos dos triângulos. No que diz respeito ao motor foi acrescentada a capacidade de leitura de novos atributos no ficheiro XML, nomeadamente no que diz respeito à luz, aos materiais e às coordenadas de textura. Para a implementação da luz, o motor passou a ser capaz de definir normais em todos os pontos das figuras e aplicar diversos tipos de luz. Relativamente às texturas, o motor passou a poder ler o ficheiros de texturas e a aplicar as mesmas. O desenho é realizado por defeito recorrendo a VBO's ou como funcionalidade adicional, recorrendo aos menus, efetuado em modo imediato.

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 melhorado permitindo gerar as normais e coordenadas de textura para as figuras que criamos como valorização nas fases anteriores.