Computação Gráfica

Trabalho Prático

Fase 4 - Normais e Coordenadas de Textura

Luís Almeida	João Pedro Antunes	Fernando Lobo	Diogo Monteiro
A84180	A86813	A87988	A71452

Resumo

O presente documento descreve a resolução do enunciado do trabalho prático adotada pelo grupo. Nesta fase foi nos proposta a geração de normais e coordenadas de textura por parte do generator, assim como a possibilidade do engine ler esta informação gerada, ler texturas e aplicar as mesmas a modelos. É nos também pedida a implementação de luz na nossa cena. Assim, começamos por gerar as normais e coordenadas de textura de cada modelo no generator, alteramos o engine para lidar com esta nova informação, e por fim estendemos os nossos ficheiros XML para podermos especificar as luzes que desejamos ter na nossa cena. Como extra, resolvemos adicionar a possibilidade de fornecermos um *Height Map* ao generator, para podermos simular o relevo dos diferentes planetas do sistema solar.

Conteúdo

Capítulo 1

Introdução

Nesta fase do trabalho prático é necessária a geração das coordenadas de textura e das normais de cada modelo por parte do generator, assim como a especificação da luz num ficheiro XML que irá estar presente na cena. Assim, podemos dividir a resolução deste problema em vários sub-problemas:

1. Gerar Coordenadas de Textura
2. Gerar Normais
3. Alterar estrutura do XML para especificar luz

Capítulo 2

Análise e Especificação

2.1 Generator

Nesta secção vamos analisar os problemas que temos de resolver para acrescentarmos as funcionalidades que o generator deverá ter nesta fase.

2.1.1 Geração de Normais

Nesta fase, a adição de fontes de luz fez com que fosse necessário efetuar algumas modificações a nível do *generator*, quanto ao cálculo das normais dos objetos. Para tal é necessário ajustar as classes, já existentes, que calculavam os pontos, de maneira a calcularem também as normais dos objetos. Para algumas figuras mais simples, como a esfera e o plano as soluções são triviais, no entanto, figuras que contenham arestas e vértices precisam de um pouco mais de cuidado nestas situações, pois o mesmo vértice irá pertencer a faces diferentes com normais completamente distintas, obrigando a uma especial atenção no cálculo das mesmas.

2.1.2 Geração de Coordenadas de Textura

Para a implementação de Materiais e imagens como texturas, fazia-se necessária uma nova abordagem quanto a constituição dos modelos $3D$ realizados no gerador. Agora, além do acréscimo dos vetores normais, o gerador deve também ser capaz de gerar as coordenadas de textura para a aplicação de imagens nos modelos.

Cada modelo apresentará uma maneira especifica de se calcular as coordenadas. O grupo procurou aplicar coordenadas com menor distorção possível em cada figura. Reaproveitando, em parte a planificação apresentada no método de geração de *Mesh* apresentado para o gerador desde a fase 1 do projeto.

2.2 Engine

Nesta secção vamos analisar os problemas que temos de resolver para acrescentarmos as funcionalidades que o engine deverá ter nesta fase.

2.2.1 Luzes

Para armazenarmos a informação relativa a cada luz presente na cena, podemos criar uma classe. Esta armazenará os valores de cada componente da luz, i.e., difusa, especular e ambiente, assim como a intensidade e posição da mesma. Podemos depois pensar em subclasses desta classe mais geral para termos luzes mais específicas, como *directional light, point light* e *spotlight*. Assim, ao fazermos o parsing do ficheiro XML, quando encontramos uma luz criamos um novo objeto consoante o tipo de luz especificado e adicionamo-lo

ao array de luzes que estará presente em memória. Posto isto, basta iterarmos as luzes na *renderScene* e fazer o "place"de cada uma delas.

2.2.2 Materiais

Esta nova *feature* requerida para nossa engine envolve a aplicação de materiais aos modelos na *Engine*. Isto é, devemos especificar através de comandos passados pelo arquivo XML, valores relativos aos atributos básicos de renderização de um objeto tais como *diffuse, emissive* e *specular*, além, claro de permitir a aplicação de imagens como texturas no *diffuse channel*.

O problema passaria então por descrever uma maneira eficiente de se fazer o setup de cada material antes de sua devida aplicação nos modelos, especialmente os que envolvem o carregamento de imagens para memória.

Assim, o problema poderá ser apresentado com dois ramos para sua solução, a aplicação direta na Engine, e o devido parsing do XML para "capturar"os elementos necessários para a descrição do material.

Capítulo 3

Concepção/desenho da Resolução

3.1 Generator

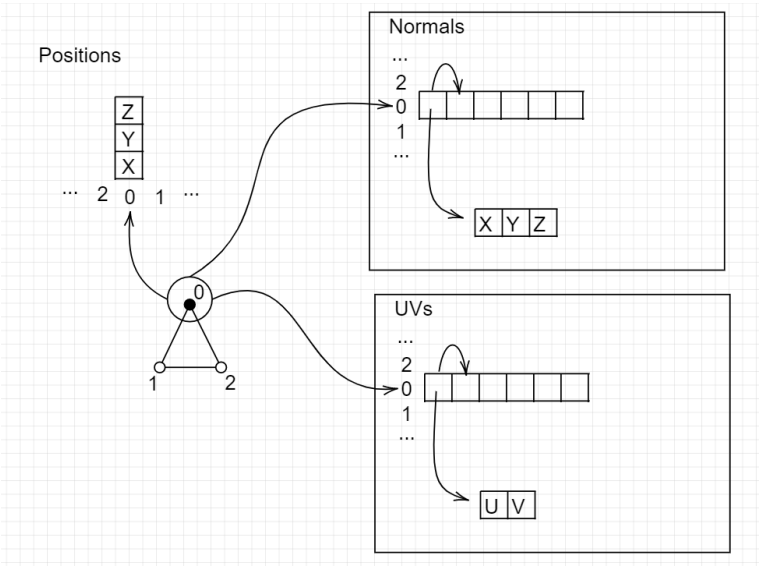
A fim de acomodar as novas *features* necessárias no *Generator*, algumas mudanças tiveram de ser feitas quanto a maneira com que o generator escreve vértices e agrupa triângulos.

Relembrando o algoritmo utilizado para a primeira fase do trabalho, a função de base "`trisPolyLine()`"recebia uma matriz de vértices, produzia os respetivos triângulos e os escrevia para o arquivo. Esta definição era suficiente até então, mas se tornou obsoleta a medida que o grupo atingiu esta fase no trabalho, onde posição poderia conter múltiplos vértices, isto é, vértices podem ser descritos pela mesma posição, mas com normais e coordenadas de Textura diferentes, caracterizando, portanto, vértices diferentes.

A alteração feita, portanto, foi alterar a escrita dos vértices para os arquivos. Foi adicionado um sistema com *Hash Maps* para caracterizar os vértices com cada uma de suas derivações. O *Hash Map* básico era o mesmo proposto até então, que relacionava o índice do vértice a posição espacial do mesmo. A seguir, outros 2 *Maps* foram adicionados, um com os vetores normais e outro com as coordenadas de textura, estes dois útlimos funcionam como queues para cada vértice.

A "`trisPolyLine()`"portanto, computa o triângulo e os índices pertencetes a ele pela ordem necessária. A seguir, entretanto, associa aos vértices selecionados a posição (fixa), uma normal e uma coordenada UV, provenientes de um dequeue dos *Maps* associados, assim, basta no gerador, dar-se enqueue pela ordem dos triângulos em que cada vértice estará.

Assim, após o processo descrito na primeira fase do *generator*, o algoritmo prosseguirá por descrever o vértice com suas novas características como descrito no diagrama seguinte:

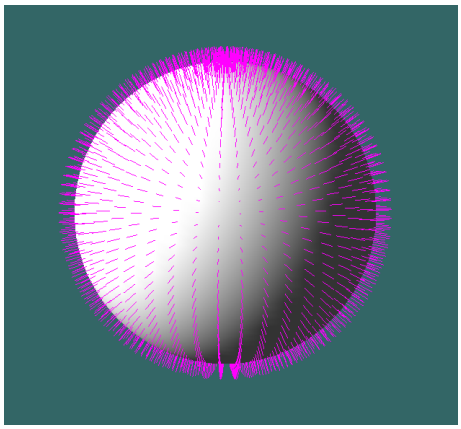


3.1.1 Geração de Normais

Sendo que algumas figuras contêm arestas é necessário ter cuidado, principalmente nesses casos, quanto ao cálculo das normais, visto que o mesmo vétice partilha várias faces diferentes.

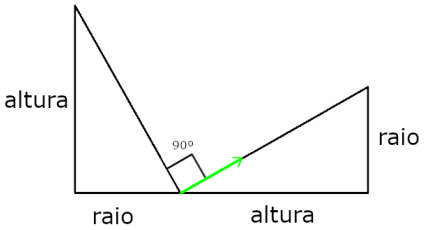
Esfera

No caso da esfera, as normais são relativamente simples de calcular, visto que os seus valores são obtidos diretamente através dos seus pontos, contudo tem de se ter em atenção que estes valores têm que ser normalizados.

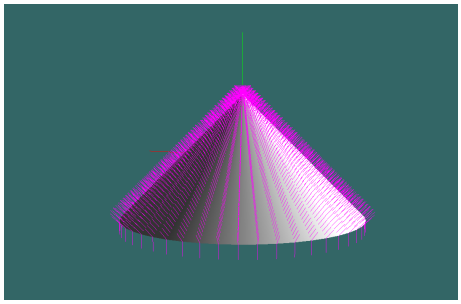


Cone

Relativamente ao cone, as normais dos vértices da base são triviais, isto é, associa-se o vetor definido pelas coordenadas $(0, -1, 0)$, a todos os pontos da base. Falta agora definir as normais da superfície lateral do cone. Para isso calculam-se os vetores perpendiculares à superfície da seguinte maneira:

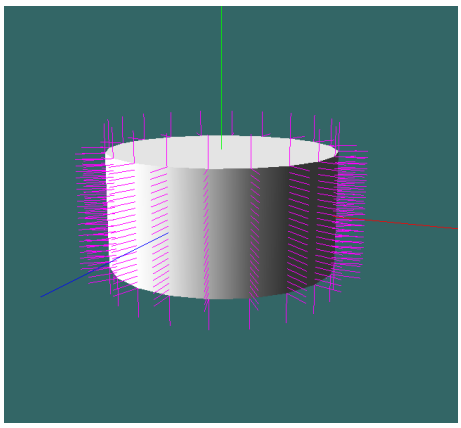


Numa dimensão $2d$ o valor de X corresponderia à altura e o de Y ao raio, sendo de seguida normalizados, tal como está representado na figura anterior. O mesmo aplica-se numa dimensão $3d$, ou seja, o valor de Y corresponde ao raio do cone e o valor de X e Z são calculados através do seno e do cosseno, respetivamente, do ângulo da slice em questão multiplicando esse valor pela altura do cone. Obtidos estes valores apenas normailiza-se as coordenadas.



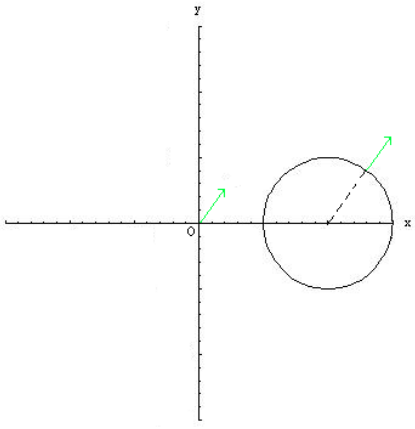
Cilíndro

O cálculo das normais da base e do topo do cilindro seguem a mesma lógica das do cone. Seriam definidas pelo velor $(0, 1, 0)$ caso se tratasse do topo e $(0, -1, 0)$ para a base. Para as laterais as coordenadas das normais são iguais às dos pontos, só que normalizadas, à exceção do valor de Y que neste caso seria zero. Da mesma maneira que se teve atenção nas normais das arestas do cone, o mesmo raciocínio aplica-se aqui.

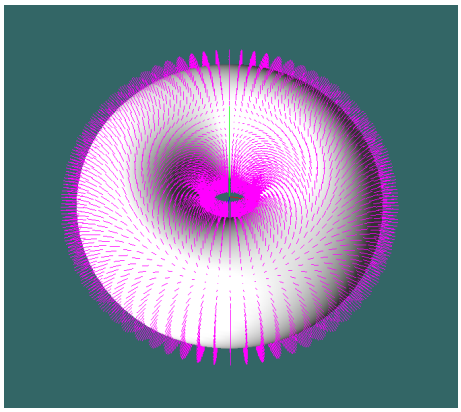


Tórus

Para calcular as normais do tórus pode-se seguir a mesma lógica que calcular os vetores perpendiculares à tangente da superfície de uma circunferência, para cada slice.

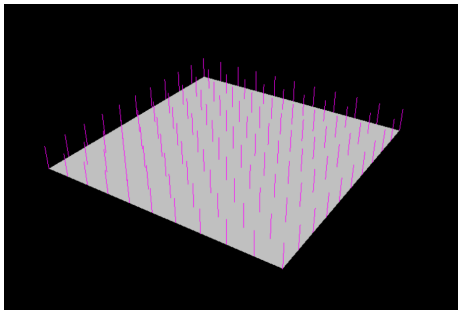


Como se pode ver, se a circunferência estivesse centrada na origem só se teria de normalizar as coordenadas do ponto contido na mesma.



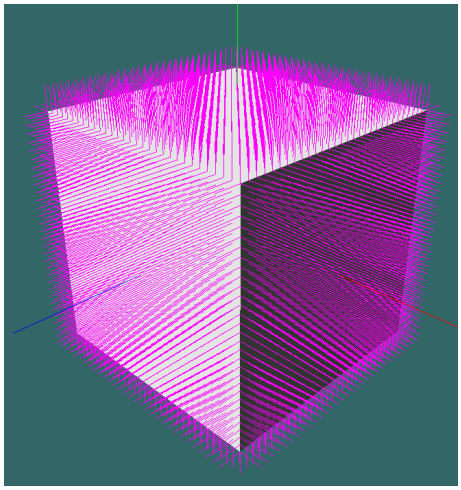
Plano

O plano é o mais trivial de todas as figuras pois as normais em cada ponto estão definidas pelas coordenadas $(0, 1, 0)$.



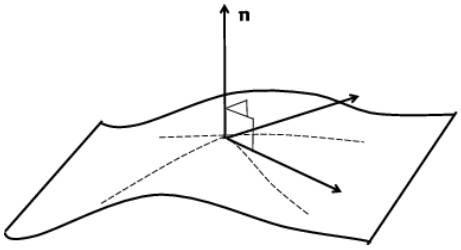
Box

As normais para os pontos da base e do topo são calculadas da mesma maneira que para o cilindro. As das faces laterais são definidas de acordo com a orientação das rotações do plano para a respetiva lateral, por exemplo, se se fizer a rotação de 90°do plano em relação ao eixo do X, as normais terão coordenadas $(0, 0, 1)$. Isto é efetuado analogamente para cada uma das faces.

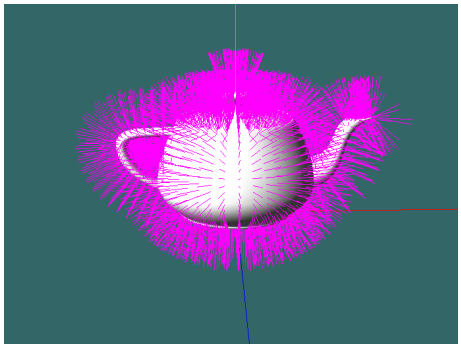


Patches

No caso da *teapot*, como esta tem superfícies curvas, para se obter as normais, recorre-se ao produto vetorial das tangentes no ponto, de maneira a obter-se a normal do plano formado por essas tangentes, tal como representado de seguida:



O resultado obtido é o seguinte:



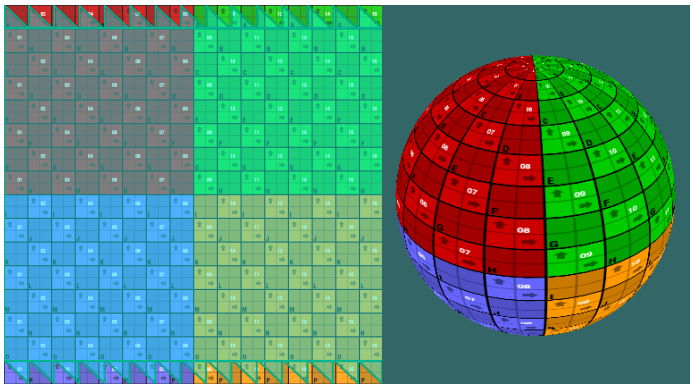
3.1.2 Geração de Coordenadas de Textura

Como já referido, teremos, assim como nas normais, atenção especial a cada *Shape*, garantindo a ordem pelas quais ocorre o enqueue das coordenadas para que o algoritmo tenha eficiência máxima.

Esfera

Para definir as coordenadas de UV da esfera, começaremos por imaginar um corte em uma das *slices*, seguindo do vértice do topo até o da base. A seguir, realizaremos pequenos cortes, formando triângulos, com o vértice do topo até os vértices da primeira stack, e do vértice da base com o da última stack.

Se considerássemos agora esta figura espalmada no plano UV, teríamos algo semelhante a zona verde apresentada a seguir.



A maior atenção neste caso é a repetição dos vértices na *slice* cortada e claro, o vértice do topo e da base, que aparecem múltiplas vezes. Os demais vértices só possuem 1 coordenada de textura.

O cálculo das posições é relativamente simples, seguindo o método de planificação já utilizado para a função de geração dos triângulos, podemos associar diretamente uma posição associando o U com a *slice* e V com a *stack* de cada ponto.

$$U = Slice_{vertex} * \frac{1}{Slices}$$

$$V = Stack_{vertex} * \frac{1}{Stacks}$$

Cone

Assim como feito na esfera, o Cone apresenta um desenvolvimento muito semelhante quanto aos cortes e a associação de U, V com *slices* e *stacks*.

$$U = Slice_{vertex} * \frac{1}{Slices}$$

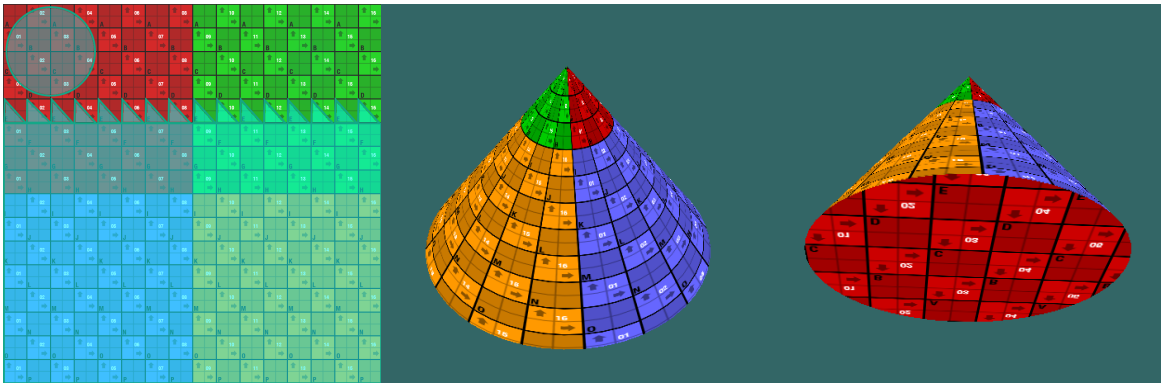
$$V = Stack_{vertex} * \frac{1}{Stacks}$$

A maior diferença em relação a esfera encontra-se na base, não seria possível utilizar a mesma técnica de triângulos da esfera sem obter uma grande distorção, a técnica utilizada, portanto, foi "recortar"a base, colocando-a em outro "pedaço"do plano da UV. desta vez a associação de U, V com *slices* e *stacks* não funcionaria.

Para obter o resultado entretanto, foi associada a cada vértice da base um ângulo, e um ponto central na UV, onde estaria o vértice central da base. Se cada ângulo somasse um total de 360^{o} facilmente era obtida uma circunferência em torno do vértice reduzindo a distorção, como pretendido.

$$U = U_{centro} + cos(Slice_{vertex} * \frac{360}{Slices})$$
$$V = V_{centro} + sin(Slice_{vertex} * \frac{360}{Slices})$$

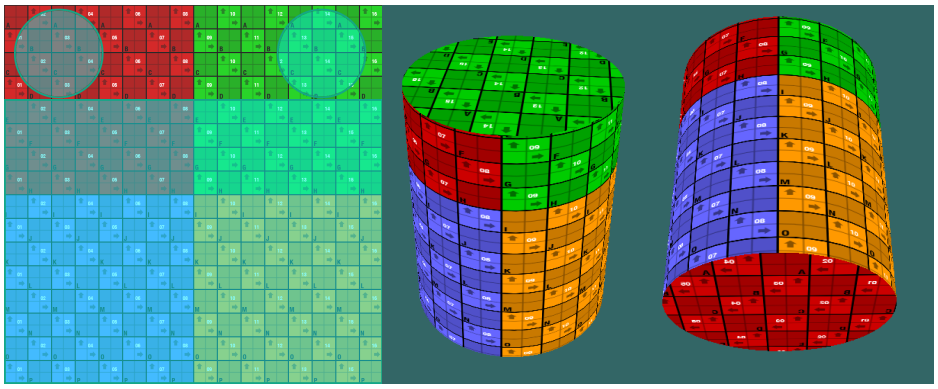
o formato final encontra-se a seguir:



Cilíndro

Para o cilíndro, trata-se da mesma técnica aplicada ao cone, porém repetindo o processo aplicado na base para a "tampa"do cilíndro.

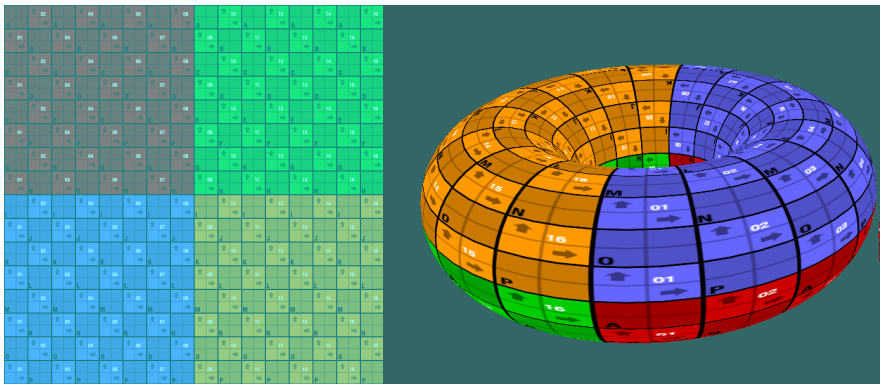
Assim, encaramos o corpo com um simples corte vertical, e dividimos um pedaço da UV para cada um destes vértices, simplesmente dividindo o tamanho do intervalo pelo número de *slices* e *stacks* e atribuindo a cada vértice correspondente. Para a base e o topo, simplesmente escolhem-se pontos na UV onde estarão o vértice central do topo e da base. Para a primeira e ultima *stack*, então, iteramos sobre as *slices* atribuindo um ângulo ao qual podemos então atribuir um valor de U e V a partir dos pontos centrais definidos.



Tórus

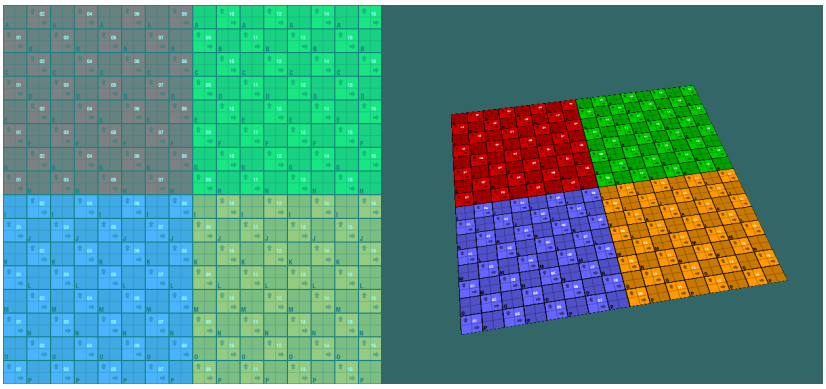
O *Shape* Tórus, um pouco mais difícil de visualizar planificado, pode ser visto com dois grandes cortes, um em torno de uma de suas *stacks*, e um em torno de uma de suas *slices*. Uma vez realizados os cortes, a planificação é imediata, sem necessidade de mais cortes.

A associação das coordenadas, muito semelhante a das demais figuras apresentadas, trata-se de uma relação direta ente a *slice* e *stack* de cada vértice, "normalizando"tais valores para estarem compreendidos entre 0 e 1.



Plano

O Plano trata-se da mais simples das figuras, divide-se o intervalo entre 0 e 1 pelo número de *slices* em X e em Z e associa cada vértice à correspondente *slice*, o resultado é o próprio plano UV sobre o plano.

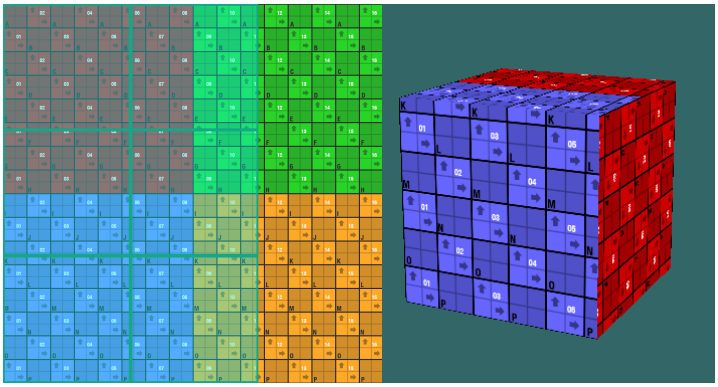


Box

A *Box* é constiuída no *generator* por 6 planos, 1 para cada uma de suas faces. Assim, a estratégia seria adicionar versatilidade ao plano a fim de facilitar o posicionamento das UVs na *Box*. A solução foi limitar o *Padding* da UV do plano em sua construção, assim, ao criar a *Box* definiriamos um U e V inicial e um intervalo máximo no qual o plano poderia ser alocado.

Pela simplicidade do cálculo da UV no plano, a tarefa foi relativamente fácil, e o intervalo a ser dividido pelos vértices do plano passou de $[0, 1]$ para $[min, max]$ tanto para U quanto para V com min, max parâmetros de construção do plano.

Assim, as coordenadas de textura da *Box* foram definidas a custa das coordenadas de 6 planos limitados a 6 intervalos na UV. Os planos foram alocados em somente $\frac{2}{3}$ da UV, mas com o motivo de reduzir a distorção da imagem aplicada ao plano.



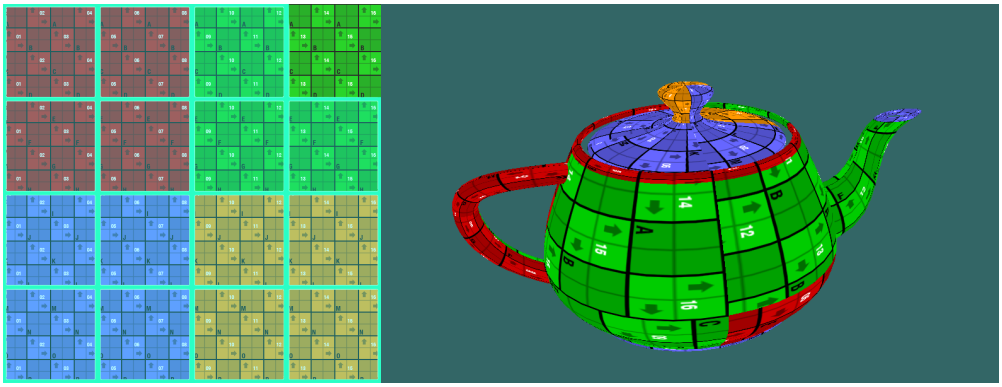
Patches

Para as Patches, as coordenadas de texturas são definidas semelhante às do plano. As patches podem ser vistas, de maneira geral, como planos distorcidos. A tesselation pode definir o equivalente a *slices* em X e em Z, semelhantemente aos planos.

A mesh da Patch, portanto, pode ser vista como equivalente a de uma *Box*, em que cada patch (semelhantemente aos planos) teria um *padding* na UV, sendo os vértices então distribuidos igualmente pelo intervalo no pad, com U e V a variar uniformemente consoante as variáveis u e v utilizadas na geração da curva da *patch*.

Para tanto a maior dificuldade neste ponto seria definir o *padding* de cada *patch* (U e V máximo e mínimo).

Para manter certa proporção, mesmo que distorcida. O *padding* é calculado pelo quadrado do *ceiling* da raíz quadrada do número de *patches*, sendo o tamanho máximo do intervalo definido por $\frac{1}{n}$ com $n = \lceil \sqrt{k} \rceil$ para k o número de patches. Isto é, por exemplo, para uma mesh de 10 patches, seriam calculados 16 *pads*, logo o intervalo dado entre U e V teria no máximo $\frac{1}{4}$ de espaçamento. A ordem pela qual as patches distribuem-se pelos *pads* na UV depende da ordem pela qual são escritas no arquivo, a seguir, apresenta-se a UV da *teapot*, que possui 15 *patches*,



3.2 Engine

3.2.1 Luzes

Assim, e conforme referido no capítulo anterior, começamos por definir a classe mais geral LightRef, que deverá depois suportar subclasses que implementam um tipo específico de luz:

```
class LightRef{


    public:
            static int total;
            int id;
            float intensityMultiplier;

            float vectsUtil[4];

            GLfloat ambt[4];
            GLfloat diff[4];
            GLfloat spec[4];

            LightRef(std::vector<float> vals, float isPoint){

                    this->id = total++;
                    for (int i = 0; i < 3; ++i){
                            vectsUtil[i] = vals[i];
                    }

                    vectsUtil[3] = isPoint;
            }

            void setup(){
                    for (int i = 0; i < 3; ++i){
                            diff[i] *= intensityMultiplier;
                            spec[i] *= intensityMultiplier;
                    }

                    glEnable(GL_LIGHT0+id);
                    glLightfv(GL_LIGHT0+id, GL_AMBIENT, ambt);
                    glLightfv(GL_LIGHT0+id, GL_DIFFUSE, diff);
                    glLightfv(GL_LIGHT0+id, GL_SPECULAR, spec);
            }

            virtual void place() = 0;
};
```

A variável *intensityMultiplier* diz respeito à intensidade da luz, *vectsUtil* corresponde à posição da luz e *ambt,diff,spec* armazenam os valores das componentes ambiente, difusa e especular da luz, respetivamente. Para implementarmos uma *PointLight*, basta "obrigar"o construtor da classe *LightRef* a receber um ponto como posição da câmera. Definimos assim a seguinte classe:

```
class PointLight : public LightRef{
    public:
            PointLight(std::vector<float> vals):LightRef(vals, 1.0f){
            }

            void place(){
                    glLightfv(GL_LIGHT0+id,GL_POSITION, vectsUtil);
            }
};
```

Ora para definirmos uma *DirectLight*, o raciocínio será o mesmo, sendo agora necessário que o construtor receba um vetor como posição da câmera:

```
class DirectLight : public LightRef{
    public:
            DirectLight(std::vector<float> vals):LightRef(vals, 0.0f){
            }


            void place(){
```

```
                glLightfv(GL_LIGHT0+id,GL_POSITION, vectsUtil);
        }
};
```

Finalmente implementamos a *spotlight*. Para este tipo de luz é necessário fornecer um ponto, um vetor de direção, um ângulo de *cutoff* que nos dá o raio da *spotlight* e um expoente que nos dá a "smoothness"da luz, ou seja, um fator que dita o quão "esborratada"a luz fica à medida que se distancia do ponto que ilumina. Esta informação ser-nos-à passada no xml.

```
class SpotLight : public LightRef{
    public:
        float directSpot[3];
        float exponent;
        float cutoff;

        SpotLight(std::vector<float> vals, std::vector<float> directs, float
            expArg, float cutArg):LightRef(vals, 1.0f){
                for (int i = 0; i < 3; ++i){
                        directSpot[i] = directs[i];
                }

                if (expArg < 0) expArg = 0;
                if (expArg > 128) expArg = 128;
                exponent = expArg;

                if (cutArg < 0) cutArg = 0;
                if (cutArg > 90) cutArg = 180;
                cutoff = cutArg;
        }

        void place(){
                glLightfv(GL_LIGHT0+id,GL_POSITION, vectsUtil);
                glLightfv(GL_LIGHT0+id,GL_SPOT_EXPONENT, &exponent);
                glLightfv(GL_LIGHT0+id,GL_SPOT_CUTOFF, &cutoff);
                glLightfv(GL_LIGHT0+id,GL_SPOT_DIRECTION, directSpot);

        }
};
```

3.2.2 Materiais

A aplicação dos materiais na *Engine*, assim como feito em outros problemas já apresentados passou por recorrer à orientação a objetos do C++. O objetivo desta nova classe, seria, portanto manter a informação necessária do *user input* passado no XML para a constituição do material, assim, o *setup* inicial do material incluindo o *load* de imagens para textura, seria feito exclusivamente uma vez, sendo ent'ao apenas aplicado no modelo a ser desenhado através de um método para dar *bind* às especificações de suas variáveis.

O objeto da classe *Material*, portanto, será constituindo por 3 pontos principais:

1. **variáveis de instância** : irão armazenar os valores de cada componente necessária para descrever o material e suas características (*diffuse, emissive* e *specular*);
2. **Construtor e *Setup* inicial** : Neste Ponto o Material pasará pela fase mais complicada, em que deverá dar load de texturas, se necessário, e carregar para memória os elementos necessários para caracterização;
3. **Método de *Setup***: Este será o método invocado na rotina de *Render Scene*, logo deverá ser o mais optimizado possível, este método deverá ser invocado toda vez antes da realização do *draw* do modelo ao qual tal material estará aplicado;

A seguir alguns *snippets* dos itens acima mencionados:

```
class Material{
public:

    std::vector<float> diff;
    std::vector<float> spec;
    std::vector<float> ambt;
    std::vector<float> emsv;

    unsigned int t = 5, tw, th;
    unsigned int texSet = 0;
    unsigned char *texData;
...

    Material(std::string fileName, std::vector<float> diff, std::vector<float> spec,
        std::vector<float> ambt, std::vector<float> emsv, int invertCull){

            unsigned int k;

            this->diff.assign(diff.begin(),diff.end());
            this->spec.assign(spec.begin(),spec.end());
            this->ambt.assign(ambt.begin(),ambt.end());
            this->emsv.assign(emsv.begin(),emsv.end());

            ilGenImages(1,&(k));
            this-> t = k;
            ilBindImage(t);
            ilLoadImage((ILstring)fileName.c_str());
            tw = ilGetInteger(IL_IMAGE_WIDTH);
            th = ilGetInteger(IL_IMAGE_HEIGHT);
            ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
            texData = ilGetData();
            glGenTextures(1,&texID);
            this->invertCull = invertCull;


            glBindTexture(GL_TEXTURE_2D,texID);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, texData);
    }

    void setup(){
            GLfloat colD[] = {diff[0], diff[1], diff[2], diff[3]};
            GLfloat colS[] = {spec[0], spec[1], spec[2], spec[3]};
            GLfloat colA[] = {ambt[0], ambt[1], ambt[2], ambt[3]};
            GLfloat colE[] = {emsv[0], emsv[1], emsv[2], emsv[3]};

            glBindTexture(GL_TEXTURE_2D,texID);

            if (diff[3]>= 0) glColor3f(diff[0], diff[1], diff[2]);
            if (ambt[3]>= 0) glMaterialfv(GL_FRONT, GL_AMBIENT, colA);
            if (spec[3]>= 0) glMaterialfv(GL_FRONT, GL_SPECULAR, colS);
            if (diff[3]>= 0) glMaterialfv(GL_FRONT, GL_DIFFUSE, colD);
            if (emsv[3]>= 0) glMaterialfv(GL_FRONT, GL_EMISSION, colE);

            if (invertCull){
                    glEnable(GL_CULL_FACE);
                    glCullFace(GL_FRONT);
                    glFrontFace(GL_CCW);
            }
```

```
        }
```

3.3 XML

No XML foram feitas modificações em métodos já existentes, como por exemplo no método de *parse* do modelo, de maneira a detetar, caso exista, a textura de uma determinada figura e, também, obter os atributos referentes ao tipo e quantidade de luz que um objeto emite. Foi ainda criado um método de *parsing* que identifica o tipo de luz, a sua posição, a intensidade e os atributos necessários para a produzir. Esta informação é toda armazenada num objeto da classe *Scene*, que serão depois usados para gerar a *scene* pretendida.

Segue-se um exemplo da *label lights* em *XML*:

```
<lights>
    <light type="POINT" I="30" X="1" Y="1" Z="1" dR="1" dG="1" dB="1" aR="0" aB="0" aG="0"
        aA="1"/>
</lights>
```

E um exemplo da *label model* com as respetivas modificações:

```
<models>
    <model file="sphere.3d" diffR="1.0" diffG="1.0" diffB="1.0" diffA="1.0"/>
    <model file="sphere.3d" emsvR="1.0" emsvG="1.0" emsvB="1.0" emsvA="1.0"/>
    <model file="sphere.3d" ambtR="1.0" ambtG="1.0" ambtB="1.0" ambtA="1.0"/>
    <model file="sphere.3d" specR="1.0" specG="1.0" specB="1.0" specA="1.0"/>
</models>
```

Também é possível definir no *XML* se se pretende inverter determinado modelo. Para tal basta adicionar um atributo no modelo denominado *invert*, cujo valor será "T", caso se pretenda realizar a inversão.

```
<models>
    <model file="sky.3d" invert="T"  texture="8k_stars_milky_way.jpg" emsvR="0.4" emsvG="
        0.4" emsvB="0.4" emsvA="1.0" />
</models>
```

Capítulo 4

Extras

4.1 Height Maps

Apresenta-se a seguir a funcionalidade extra desenvolvida para o *Generator* para aplicação de *Height Maps* em *Shape*s.

Ao contrário da última versão apresentada dos *Height Maps*, esta apresenta-se implementada diretamente no *Generator* e não na *Engine*, permitindo cálculo de normais para geração de iluminação consoante a superfície produzida e aumentando a eficiência na leitura de tais modelos.

Os *Height Maps* desenvolvidos podem ser aplicados em dois diferentes contextos:

4.1.1 Planos

Muito semelhante ao apresentado na fase anterior, aplicado entretanto ao cálculo de UVs e Normais. Neste contexto, o *Height Map* é ajustado para a quantidade de polígonos requerida nos argumentos do generator. Para tanto, o mesmo processo descrito para a fase anterior é realizado, contudo com mais controlo sobre a quantidade de vértices em cada *slice* em X e em Z.

O cálculo realizado trata-se de "normalizar"o valor da *slice* de X e de Z de um dado vértice, em relação ao total de vértices por *slice* em cada um dos eixos. Assim, um dado vértice estaria representado por uma dada linha e coluna em valores normalizados (entre 0 e 1), multiplicamos, portanto, os valores pela respectiva *height* e *width* da imagem do *Height Map* e arredondamos o valor, obtendo então, a linha e coluna do pixel representante daquele vértice, o qual então extraímos a informação de altura necessária. Ou seja,

$$pixel = \frac{SliceX_{vertex}}{SlicesX} * Height_{image} * Width_{image} + \frac{SliceZ_{vertex}}{SlicesZ} * Width_{image}$$

O valor de altura é então normalizado em ordem a 255, o valor máximo, para então ser multiplicado pela intensidade requisitada.

Assim, o vértice seria deslocado para cima (Y) com a intensidade descrita no pixel.

O cálculo da normal, demonstra-se um *Cross-product* entre os vetores entre os pontos vizinhos, muito semelhante ao cálculo realizado para as *patches*.

4.1.2 Esféricos

Os *Height Maps* esféricos demonstram-se muito semelhantes aos aplicados em planos, o processo de determinação da intensidade da altura é o mesmo. Realizando os mesmos cálculos, com a variação de se substituir os valores de "*SlicesX*"pelo valor de *stacks* da esfera.

O cálculo da normal é executado igualmente ao apresentado.

A maior diferença encontra-se no deslocamento dos pontos, dada a intensidade da altura, devemos para cada vértice, definir o que é de facto "cima"para o ponto. Para isso utilizaremos o cálculo de normais utilizado na esfera comum, desenvolvida em capítulos anteriores.

Os vetores resultantes coincidem com as normais geométricas da superfície esférica, basta então que multipliquemos tais vetores pela intensidade de altura pretendida para cada vértice e somarmos a estes mesmos vértices os vetores resultantes. Como resultado obteremos o deslocamento necessário relativo ao vetor apontando para "cima"de cada vértice.

4.2 Funcionalidades do Engine

Nesta secção apresentaremos as *hotkeys* e as suas funcionalidades que decidimos implementar no Engine. Designamos a cada número no teclado uma funcionalidade diferente:

- 1 - Funciona como um *toggle* de desenho dos modelos da cena em *wireframe*, ou seja, liga e desliga o desenho dos modelos em *wireframe*
- 2 - Funciona como um *toggle* que desenha (ou não) os eixos da cena
- 3 - Funciona como um *toggle* que desenha (ou não) os eixos da câmera
- 4 - Aumenta a velocidade com a qual um modelo percorre uma curva de Catmull-Rom, até um máximo. Na nossa cena, interage com a *teapot* que orbita o sistema solar
- 5 - Diminui a velocidade com a qual um modelo percorre uma curva de Catmull-Rom, até um mínimo
- 6 - Funciona como um *toggle* que permite parar/arrancar os modelos que percorrem um curva de Catmull-Rom
- 7 - Funciona como um *toggle* que desenha as curvas de Catmull-Rom presentes na cena. Como esta operação é pesada, desligam-se as luzes da cena para desenhar as curvas. Quando já não as quisermos ver, ligam-se as luzes da cena
- 8 - Liga as luzes da cena
- 9 - Funciona como um *toggle* que desenha as normais de cada modelo na cena. Como esta operação é pesada, desligam-se as luzes da cena para desenhar as normais. Quando já não quisermos ver as normais, ligam-se as luzes da cena
- 0 - Funciona como um *toggle* que substitui as texturas de cada modelo por UVCheckers e vice-versa

4.3 Funcionalidades do Generator

Nesta secção demonstramos como interagir com o generator:

- sphere <raio ><slices ><stacks >- cria uma esfera com um raio, número de slices e número de stacks especificados
- cone <raio ><slices ><stacks >- cria um cone com um raio, número de slices e número de stacks especificados
- plane <xDim ><zDim ><xSlices ><zSlices >- cria um plano no plano xz com as dimensões no eixo x xDim, eixo z zDim e número de slices para cada eixo
- box <xDim ><yDim ><zDim ><xSlices ><ySlices ><zSlices >- cria um cubo com dimensões <xDim ><yDim ><zDim >em cada eixo x,y,z respetivamente, <xSlices ><ySlices ><zSlices >como número de slices em cada eixo

- cylinder <raio ><altura ><slices ><stacks >- cria um cilindro com raio, altura, número de slices e número de stacks especificados
- torus <raioInterno ><raioExterno ><slices ><stacks >- cria um torus de dimensão dada por uma circunferência de raio <raioExterno >e uma espessura dada por uma circunferência de raio <raioInterno >
- patch <fileName ><tesselation >- gera uma superfície de bezier com uma tesselação <tesselation >especificada pelos patches de bezier no ficheiro <fileName >
- heightPlane <fileName ><xDim ><zDim ><intensidade ><xSlices ><zSlices >- gera um plano com relevo de dimensões <xDim ><zDim >e com slices em cada eixo <xSlices ><zSlices >. Especifica-se a intensidade do relevo e o Height Map a aplicar ao plano
- heightSphere <fileName ><raio ><intensidade ><slices ><stacks >- gera uma esfera de raio <raio >com slices <slices >e stacks <stacks >. A sua superfície tem relevo de intensidade <intensidade >e o mesmo é definido pelo Height Map <fileName >

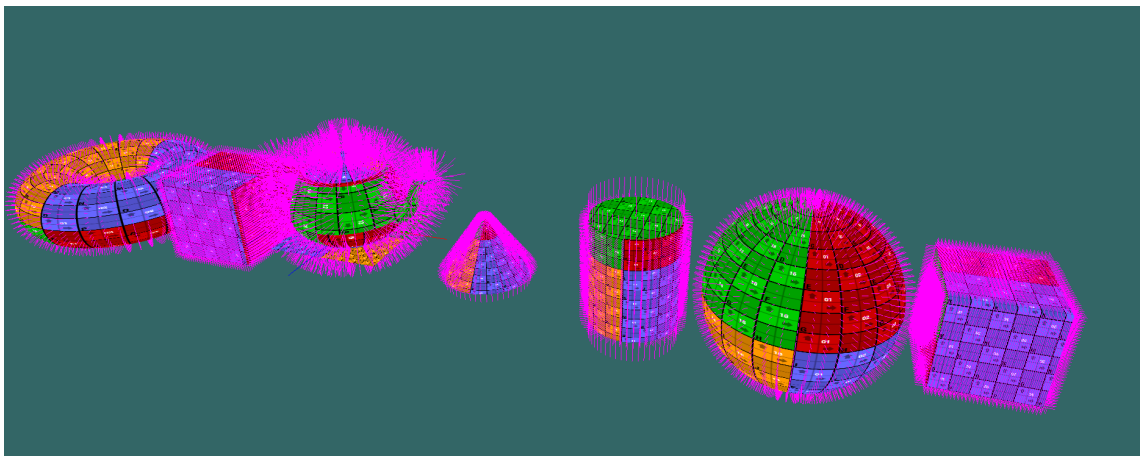
4.4 Skybox

A geração da *skybox* consiste apenas na alteração pela qual os triângulos do objeto são desenhados, ou seja, para conseguirmos ter o efeito desejado basta usar algum tipo de *label* que identifique se se pretende inverter ou não os triângulos e, caso se pretenda inverter, usa-se as seguintes instruções, do *OpenGL*, que se encontram no método *setup* da classe *Materials*:

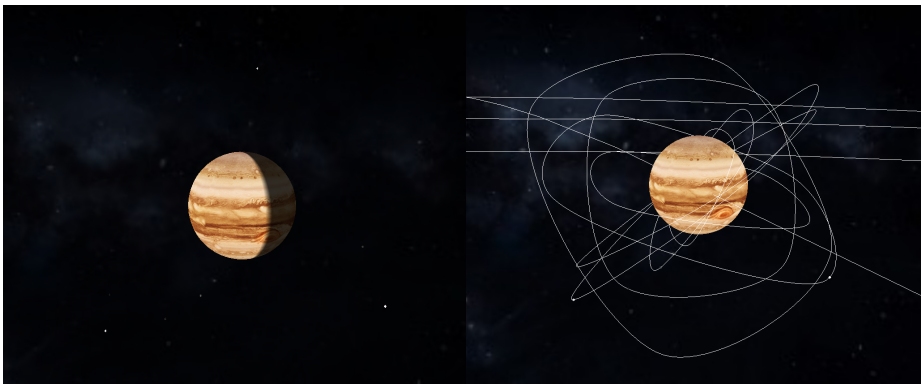
```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
glFrontFace(GL_CCW);
```

4.5 Testes realizados e Resultados

Normais e texturas das figuras:



Júpiter com as respetivas luas e as suas curvas associadas:



Height maps gerados na Terra e na Lua:

Aplicação extra de uma *height map* dum plano, com textura e luz:

Sistema Solar completo:

Capítulo 5

Conclusão

Foram cumpridos os objetivos desta fase final do trabalho prático, ficando a aplicação terminada. Recordemos que introduzimos a possibilidade de usar texturas e iluminação nos nossos modelos. Como extra, adicionamos também height maps para termos relevo nos nossos planetas. Devido à flexibilidade da solução orientada a objetos adotado pelo grupo, a aplicação é facilmente extensível para lidar com novos modelos ou qualquer outra funcionalidade que se deseje implementar.