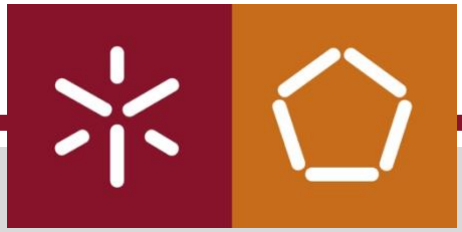
UNIVERSIDADE DO MINHO



Mestrado Integrado em Engenharia Informática

PROJETO DE COMPUTAÇÃO GRÁFICA FASE 4 : NORMAIS E COORDENADAS DE TEXTURAS.

maio, 2020




GRUPO 10
Carolina Cunha, A80142 | Hugo Faria, A81283 | João Diogo Mota, A80791 | Rodolfo Silva, A81716

ÍNDICE

ÍNDICE DE FIGURAS

BREVE DESCRIÇÃO DO ENUNCIADO PROPOSTO

Prevê-se, com este trabalho, o desenvolvimento de um mecanismo 3D baseado em mini gráficos de cenas e fornecer exemplos de uso que mostrem o seu potencial. Este trabalho será dividido em quatro fases distintas, de modo a facilitar a elaboração do mesmo. A conceção e desenvolvimento do trabalho seguirá uma abordagem suportada pela ferramenta *OpenGL* e a linguagem C++.

A última fase do projeto consistirá na introdução de fontes de luz, de normais nas figuras e de texturas aplicadas às mesmas.

Assim sendo, o *generator* será responsável pela implementação das coordenadas de texturas e das normais. Por outro lado, a adição de fontes luminosas ao cenário, bem como a ativação das funcionalidades de texturas, estará a encargo do *engine*, que irá ler e aplicar as normais e coordenadas de texturas dos ficheiros modelo.

PRIMITIVAS GRÁFICAS

Nesta fase, foi adicionada uma nova primitiva gráfica, permitindo aos elementos do grupo auto desafiarem-se para a construção de conjuntos de primitivas. Desta forma, foi construída a primitiva cilindro, permitindo a sua posterior utilização no boneco-de-neve.

CILINDRO

Uma vez que esta primitiva fora construída ao longo das aulas teórico-práticas desta unidade curricular, a sua implementação foi bastante simples. A construção do cilindro teve início com a construção das suas bases. Desse modo, é, primeiramente, necessário ter em consideração o número de fatias definido, de forma a dividir a figura em fatias de igual dimensão, obtendo assim o ângulo para cada.

```
float sliceStep = (float)(2 * M_PI) / slices;
```

Após efetuada a divisão do número de fatias, e recorrendo ao raio da base do cilindro, foram calculados os vértices dos triângulos que irão representar a sua base.

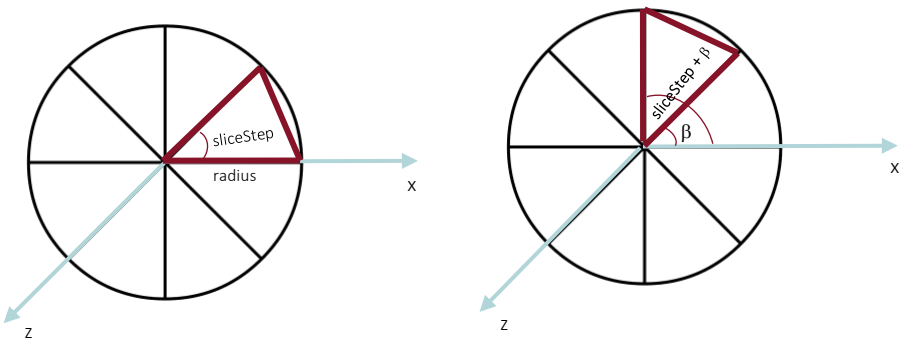


Figura 1: Representação do cálculo dos triângulos

O desenho dos triângulos é efetuado através de um ciclo que, para cada número de fatias, vai calcular os vértices do triângulo seguinte. Este cálculo é conseguido devido à existência do ângulo β, dado pelo valor `angle * i`, em que i representa o número de fatias até então, e β o valor do ângulo atual. Desta forma, o valor do ângulo do vértice seguinte é dado por `β = angle * (i + 1)`.

Uma vez que estamos perante pontos baseados num raio e ângulo, é necessária a utilização de coordenadas polares.

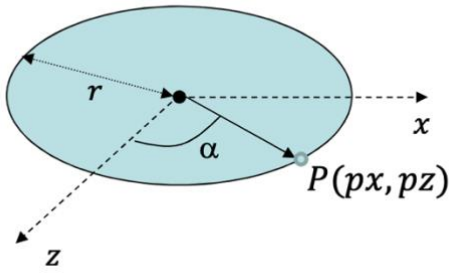
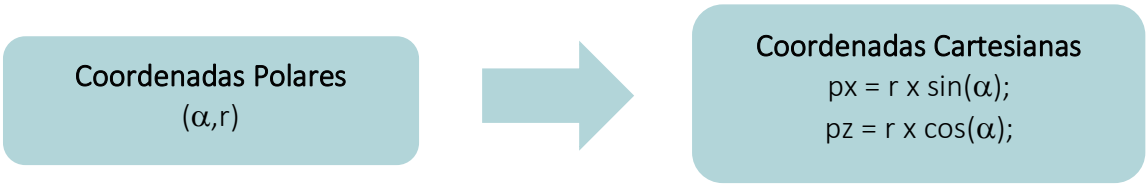
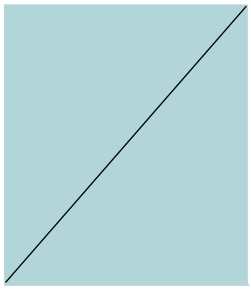


Figura 2: Representação da circunferência base para cálculos



No que diz respeito à face lateral do cilindro, para a sua construção foram utilizados dois triângulos, de modo a construir cada porção correspondente a uma fatia de cada divisão.



Desta forma, através de um ciclo capaz de iterar por cada fatia existente, foram calculados novos ângulos. Seguindo o raciocínio utilizado para a construção das bases do cilindro, foi possível a construção do mesmo.

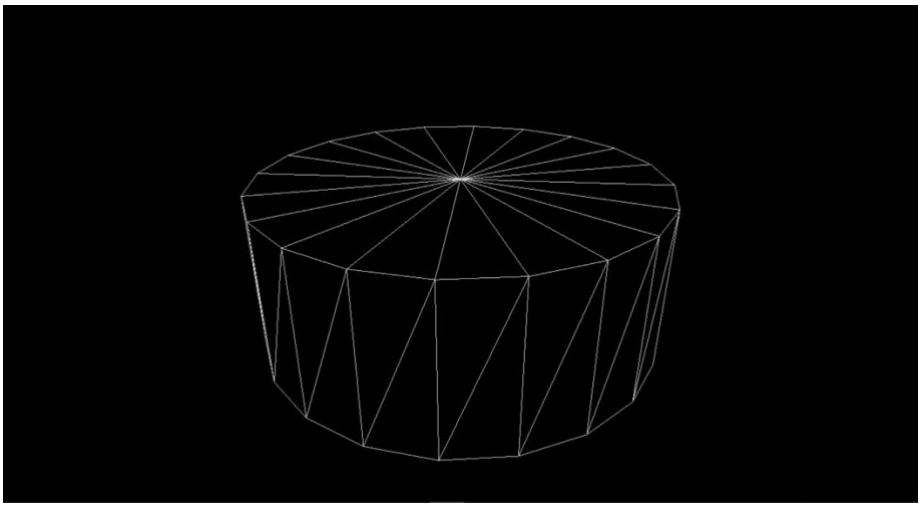


Figura 3: Representação do Cilindro em OpenGL

GENERATOR

NORMAIS

Para o cálculo das normais de primitivas gráficas curvas, foram consideradas as seguintes equações de cálculo da normal de um triângulo:

$$\overrightarrow{v_1} = p_2 - p_1 \qquad \overrightarrow{v_2} = p_3 - p_1 \qquad (1)$$

$$\vec{n} = \frac{\overrightarrow{v_1} \times \overrightarrow{v_2}}{|\overrightarrow{v_1} \times \overrightarrow{v_2}|} \qquad (2)$$

O produto externo dos vetores da Eq.1 retorna um vetor perpendicular a ambos, ou seja, perpendicular ao triângulo. Este vetor deve ser normalizado, como anteriormente mencionado, de forma a permitir o cálculo eficiente dos cossenos da equação de iluminação [1].

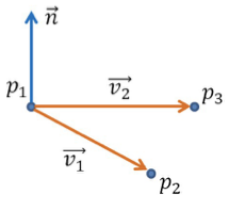


Figura 4: Cálculo do Vetor Normal [1]

O cálculo destas normais pressupôs o uso do modelo de ***Flat Shading***. Neste modelo, a equação de iluminação é executada apenas uma vez por triângulo e todos os seus pixéis têm a mesma cor. Só é necessária uma normal por triângulo. Por este motivo, triângulos com orientações diferentes podem ser claramente distinguidos na imagem renderizada final [1].

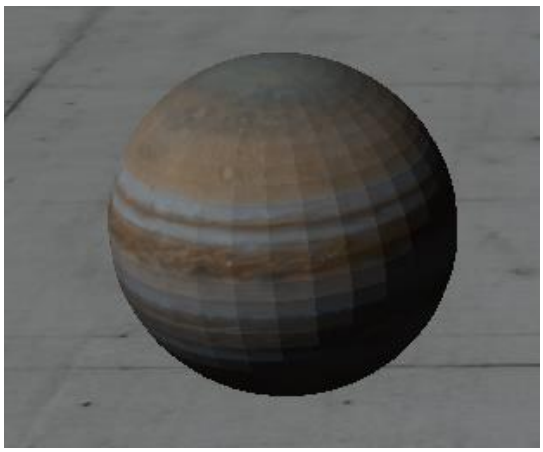


Figura 5: Esfera com Flat Shading

Como se verifica na Figura 5, é nitidamente visível a distinção dos triângulos com orientações diferentes. Uma vez que este não foi o resultado pretendido, foi alterado o modo de cálculo das normais.

Numa tentativa de aperfeiçoar o aspeto final das primitivas, optou-se pelo *Gouraud Shading*. Assim sendo, ao invés de calcular uma normal por triângulo, é calculada uma normal por vértice, o que implica que cada vértice de um triângulo pode ter uma normal diferente [1].

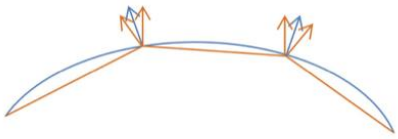


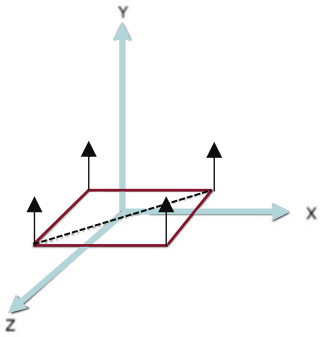
Figura 6: Aproximação às normais [1]

Desta forma, o resultado final é o apresentado na figura que se segue.

Figura 7: Esfera com Gouraud Shading

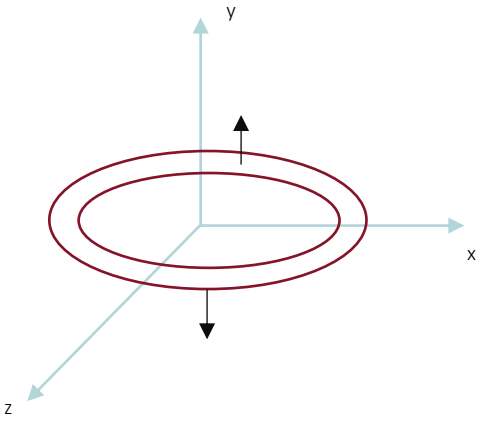
PLANO

Sabendo que o plano se encontra no eixo *xOz* , as normais dos vértices dos dois triângulos que o constituem serão representadas por um vetor $\vec{n} = (0,1,0)$.



ARCO

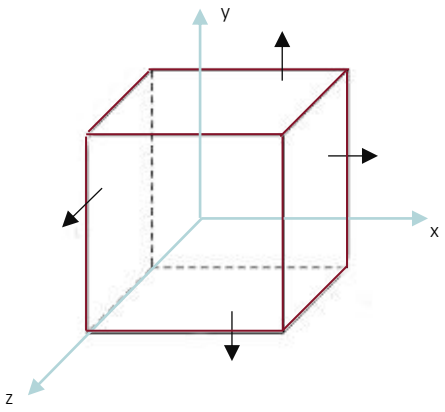
Uma vez que o arco se encontra no plano *xOz*, as suas normais são representadas pelo vetor $\vec{n} = (0,1,0)$. Dado que se pretendeu criar o arco de modo a que este seja visível em ambas as faces, as normais da face inferior do arco serão representadas pelo vetor $\vec{n} = (0,-1,0)$.



CAIXA

No que diz respeito à caixa, as suas normais irão variar consoante a face em questão. Deste modo, têm-se os seguintes vetores normais para cada vértice de cada face:

Superior	$\vec{n} = (0,1,0)$
Inferior	$\vec{n} = (0,-1,0)$
Frontal	$\vec{n} = (0,0,1)$
Traseira	$\vec{n} = (0,0,-1)$
Esquerda	$\vec{n} = (-1,0,0)$
Direita	$\vec{n} = (1,0,0)$



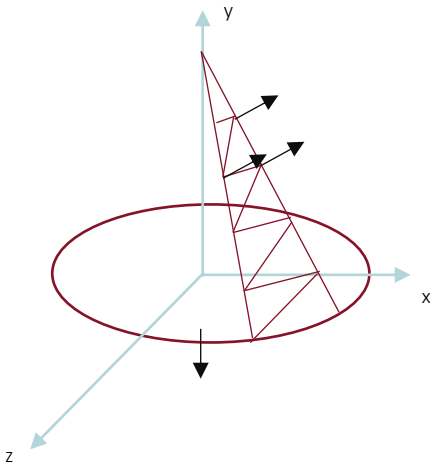
CONE

A obtenção das normais do cone requer a noção da existência de dois tipos distintos de normais. As normais da base são dadas pelo vetor $\vec{n} = (0,-1,0)$. Por sua vez, para a obtenção das normais da face lateral do cone, é necessário calcular o valor das normais para cada vértice dos seus triângulos.

Assim sendo, é crucial ter conhecimento do número de divisões e de fatias atribuídos à primitiva. Deste modo, obtêm-se as variáveis

```
float angle = ((2 * M_PI) / slices);
 float stackDiv = height / stacks;
```

Após calculadas estas duas variáveis, o cálculo das normais do cone foi idêntico ao cálculo dos triângulos do mesmo, aquando da sua implementação na primeira fase do projeto. No entanto, valores associados ao raio foram removidos, permitindo assim utilizar valores normalizados.



ESFERA

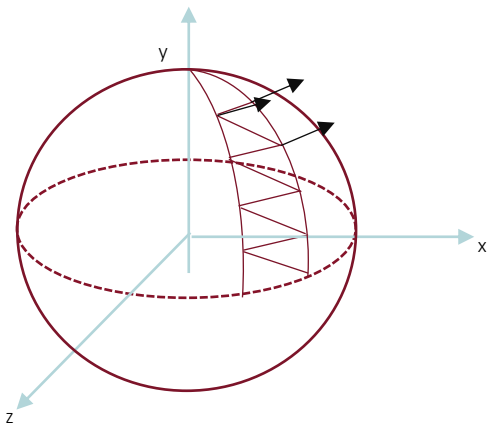
O cálculo dos vetores normais dos vértices da esfera é idêntico ao cálculo dos vetores normais da face lateral do cone.

Assim sendo, é necessário o cálculo das variáveis

```
float sliceStep = (float)(2* M_PI) / slices;
  float stackStep = (float)M_PI / stacks;

float stackAngle = M_PI / 2 – ((double)I * stackStep;
      float sliceAngle = j / sliceStep;
```

Após obtidas as variáveis, o processo de cálculo das normais da esfera é idêntico ao da face lateral do cone.



CINTO

O cálculo das normais do cinto é análogo ao cálculo das normais da esfera, pelo que deve apenas ser tido em consideração a necessidade de obter as seguintes variáveis:

```
float sliceAngle = j * ((2* M_PI) / slices);
float sliceAngle1 = (j+1) * ((2* M_PI) / slices);

float stackAngle = i * ((2* M_PI) / stacks);
float stackAngle = (i+1) * ((2* M_PI) / stacks);
```

CILINDRO

A primitiva do cilindro terá três tipos de normais. As normais da base do cilindro são dadas pelo vetor $\vec{n} = (0, -1, 0)$. Por sua vez, as normais do topo do cilindro serão dadas pelo vetor $\vec{n} = (0,1,0)$.

Relativamente à face lateral do cilindro, uma vez que os vetores normais são horizontais, a coordenada y será 0. Por sua vez, e recorrendo às coordenadas polares, as coordenadas do vetor normal do cilindro serão do tipo $\vec{n} = (rsin(\alpha), 0, rcos(\alpha))$.

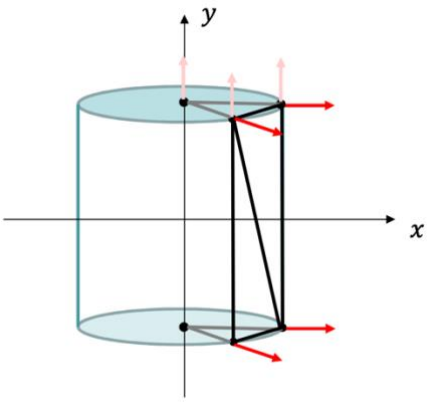
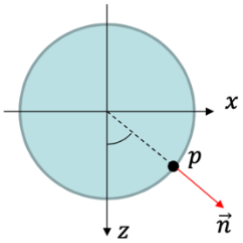


Figura 8: Normais de um cilindro



Coordenadas Polares

$x = \sin(\alpha)$

$z = \cos(\alpha)$

TEAPOT

Para obter os valores das normais para os pontos do *teapot*, foi adicionado um novo argumento à função *getBezierPoint*, permitindo armazenar o valor da normal calculada para um ponto.

Desta forma, após calcular os vetores $\overrightarrow{derivU}$ e $\overrightarrow{derivV}$, resultado das derivadas parciais dos pontos em ordem a u e v, respetivamente, vai ser realizado o produto externo sobre estes, seguindo-se a sua normalização. Desta forma, obtém-se o valor da normal para esse ponto.

Aplicando o mesmo processo aos restantes três pontos, torna-se possível a obtenção das normais para o *teapot*.

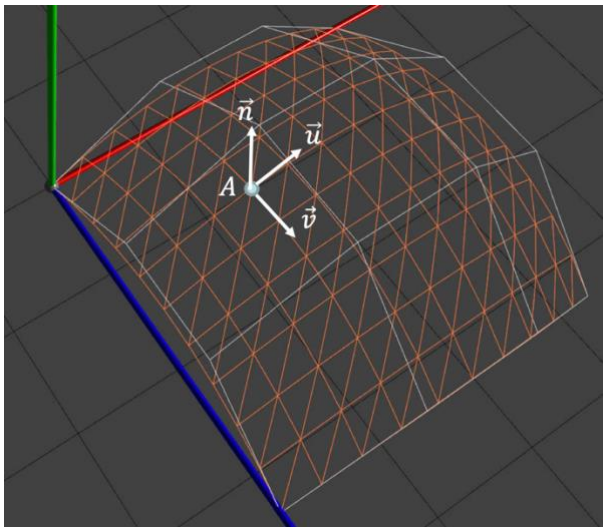


Figura 9: Cálculo das normais para o Patch

Cálculo da Normal

$$\vec{u} = \frac{dp}{du}$$

$$\vec{v} = \frac{dp}{dv}$$

$$\vec{n} = \vec{v} \times \vec{u}$$

COORDENADAS DE TEXTURAS

Texturas são uma componente fundamental da computação gráfica e consistem em imagens que que atribuem cor (e várias outras propriedades) aos modelos 3D. Esta atribuição é adquirida a partir de um processo denominado de *UV mapping*, que consiste em estabelecer o mapeamento entre localizações numa uma imagem e vértices de um modelo. Regra geral, o espaço U-V de texturas utiliza um valor adimensional, que varia entre 0 e 1 onde as coordenadas (0, 0) e (1, 1) correspondem a cantos opostos na imagem. [1]

PLANO

Dado que o plano pode ser observado como uma imagem *2D*, a associação das texturas a esta primitiva resume-se numa correspondência direta dos pontos da imagem ao espaço U-V. Assim sendo, temos como coordenadas de texturas $(1,1), (0,0), (1,0)$ para o triângulo esquerdo do plano e $(1,1), (0,1), (0,0)$ para o triângulo direito.

Figura 10: Aplicação da textura ao Plano

CAIXA

A associação da textura à caixa foi realizada face a face. De modo a ser possível esta associação, foi necessário ter em consideração o número de divisões da caixa. Desta forma, foi calculada a variável **tex**, que permite dividir a imagem da textura, assumindo um quadrado de uma unidade de lado, pelo número de divisões da caixa.

De seguida, para cada linha e coluna da caixa, são calculados os pontos correspondentes às coordenadas de textura em cada face.

Figura 11: Aplicação da textura à Caixa

CILINDRO

De modo a permitir associar ao cone a respetiva textura, procedeu-se, numa primeira fase, à análise da planificação da imagem, de forma a permitir a correta associação.

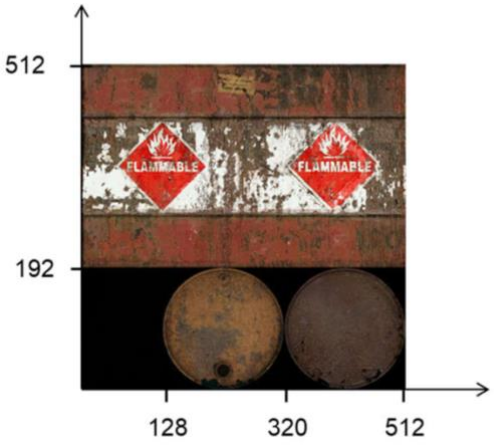


Figura 12: Espaço Imagem

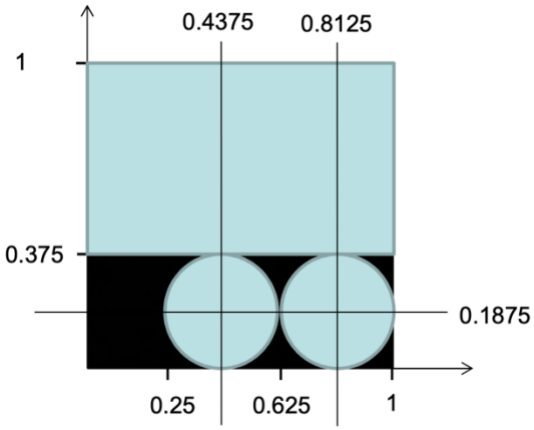


Figura 13: Espaço Textura

Assim sendo, verifica-se que a base do cilindro terá o seu centro no ponto $(0.8125,0.1875)$, e o seu topo no ponto $(0.4375,0.1875)$, sendo o seu mapeamento dependente do número de fatias do cilindro. De igual modo, será realizada a associação da textura à face lateral do cilindro, sendo que o mapeamento se encontra restrito aos valores entre os pontos $(0,0.375)$ e $(1,1)$.

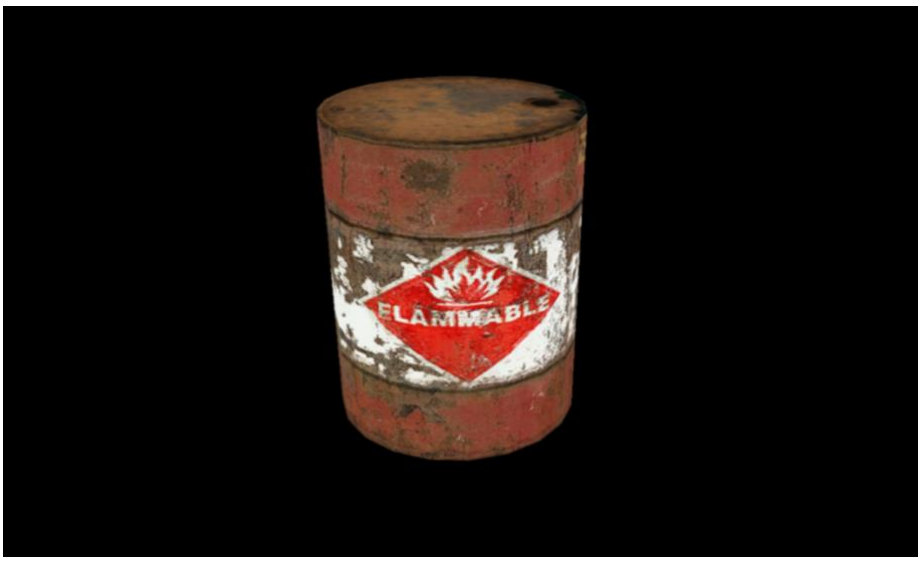


Figura 14: Aplicação da textura ao Cilindro

CONE

Para a atribuição da textura ao cone, foi utilizada a planificação do cilindro, uma vez que esta primitiva é semelhante à anteriormente apresentada. No entanto, esta primitiva tem a adição das divisões da figura.

Desta forma, a construção da base do cone foi idêntica à construção da base do cilindro. Por outro lado, o mapeamento da face lateral foi realizado de forma imediata, uma vez que a cada divisão do cone está relacionada uma zona da figura, sendo que uma fatia terá um conjunto de divisões associadas.

Figura 15: Aplicação da textura ao Cone

ESFERA

Para a textura de uma esfera, é crucial ter conhecimento do número de divisões e de fatias utilizadas para representar a figura.

De seguida, são calculadas as variáveis

```
xSlice = 1.0f / slices;
yStack = 1.0f / stacks;
```

Estas variáveis permitem obter o tamanho de cada fatia/divisão no espaço textura. Para cada divisão, é calculado um valor *stackStep*, dependente da divisão em questão, e do valor de *yStack*. Do mesmo modo, é calculado um valor *sliceStep*, dependente da fatia em estudo, bem como do valor de *xSlice*.

De seguida, são obtidas as coordenadas de texturas, recorrendo aos valores calculados.

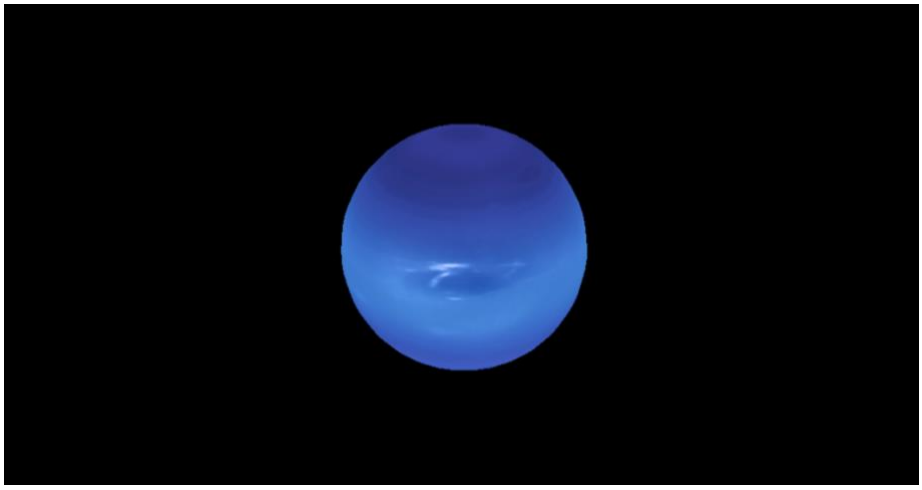


Figura 16: Aplicação da textura à Esfera

ARCO

Para as texturas do arco, a imagem foi estendida para cada fatia do mesmo. Assim, para cada fatia da figura, tem-se os seguintes tipos de coordenadas de texturas:

$(\mathbf{0.0}, \mathbf{1.0})$
$(\mathbf{1.0}, \mathbf{0.0})$
$(\mathbf{1.0}, \mathbf{1.0})$
$(\mathbf{0.0}, \mathbf{0.0})$

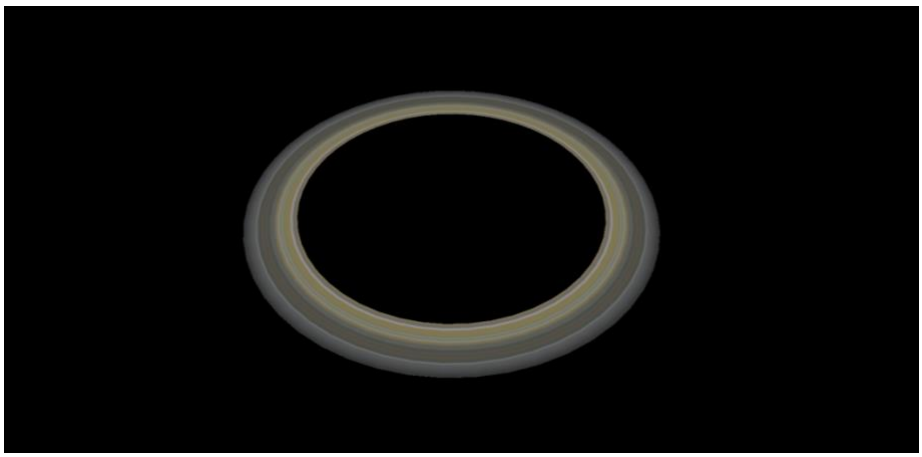


Figura 17: Aplicação da textura ao Arco

CINTO

Assim como para o mapeamento da face lateral do cone no espaço U-V das texturas, para o caso concreto do cinto, é também indispensável o cálculo das variáveis *xStack* e *ySlice*.

Deste modo, para cada divisão e fatia, são calculados os pontos correspondentes à posição dos vértices na imagem.

Figura 18: Aplicação da textura ao Cinto

Foram implementadas duas versões das texturas do cinto. Na primeira versão (representação esquerda), a imagem da textura é estendida pela primitiva. Em contrapartida, na segunda versão, (representação direita) das coordenadas de texturas, a imagem é estendida por cada divisão da primitiva, replicando a imagem o número de divisões da figura.

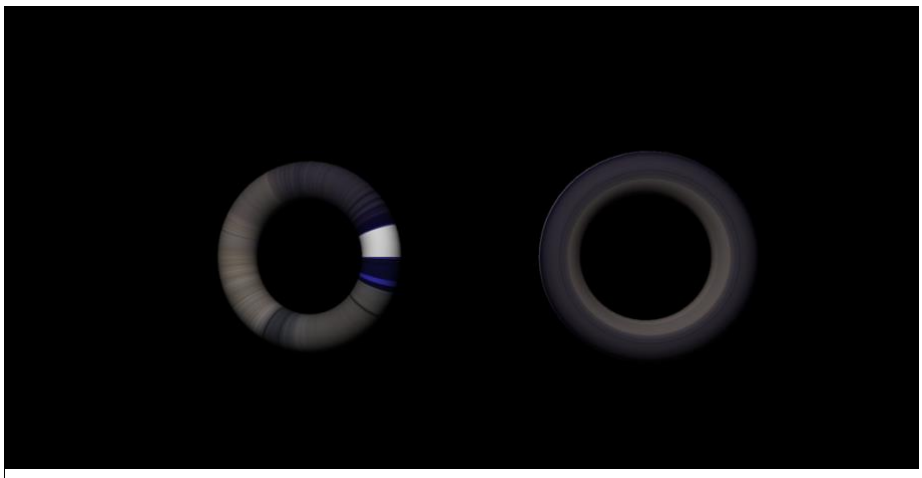


Figura 19: Aplicação das texturas ao Cinto versão 1 e 2

TEAPOT

Assim como para a obtenção das normais, o cálculo das coordenadas de texturas do *teapot* exigiu a inserção de um novo argumento na função *getBezierPoint*, permitindo armazenar os pontos da figura. Desta forma, a implementação das texturas foi realizada de forma direta, tendo em atenção a ordem com que foram desenhados os triângulos da mesma.

Figura 20: Aplicação da textura ao Teapot

ENGINE

ILUMINAÇÃO

Para a adição de iluminação, considere-se uma fonte de luz que emite luz numa determinada direção. Desta forma, é preciso ter em atenção três fatores:

- Intensidade da luz;
- Orientação do objeto em relação à fonte de luz;
- Distância à fonte de luz.

No caso do Sistema Solar, considera-se que os raios de luz chegam à Terra paralelos por a luz estar "infinitamente longe", pelo que a distância é irrelevante.

LUZ

Para os diferentes cenários produzidos, e de forma a atribuir-lhes uma aparência mais realista, foram utilizados diferentes tipos de luzes.

- Ponto de Luz – Consiste numa luz que é posicionada num ponto e emite feixes de luz em todas as direções à sua volta.

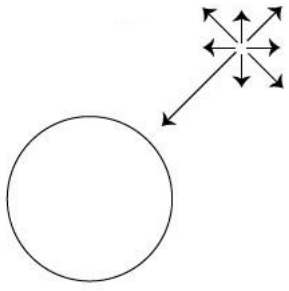


Figura 21: Representação de Ponto de Luz

- Luz Direcional – Consiste numa luz que está infinitamente longe e tem uma só direção, atingindo os objetos todos com a mesma direção.

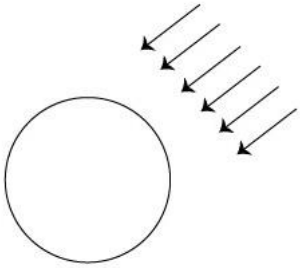


Figura 22: Representação de Luz Direcional

- Foco de Luz – Consiste numa luz posicionada num ponto p e com uma direção $\vec{d}$. A estas componentes, é adicionado um ângulo α de *Cutoff*, responsável por determinar o alcance da luz na direção $\vec{d}$.

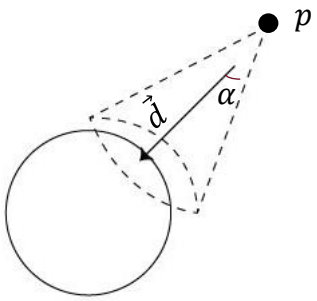


Figura 23: Representação do Foco de Luz

- Difusa – Representa a cor do objeto, ou seja, a cor que este possui quando atingido por uma luz branca.

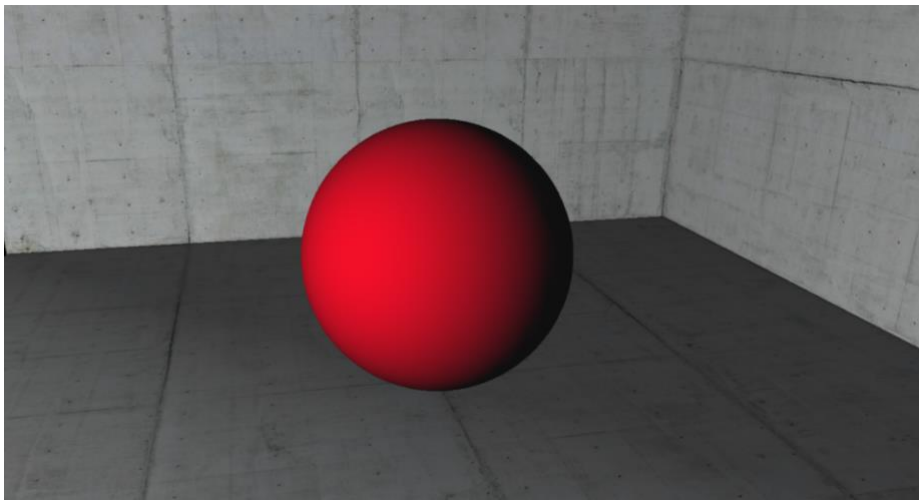


Figura 24: Aplicação da luz difusa

- Especular – Cor da reflexão da luz numa superfície brilhante.

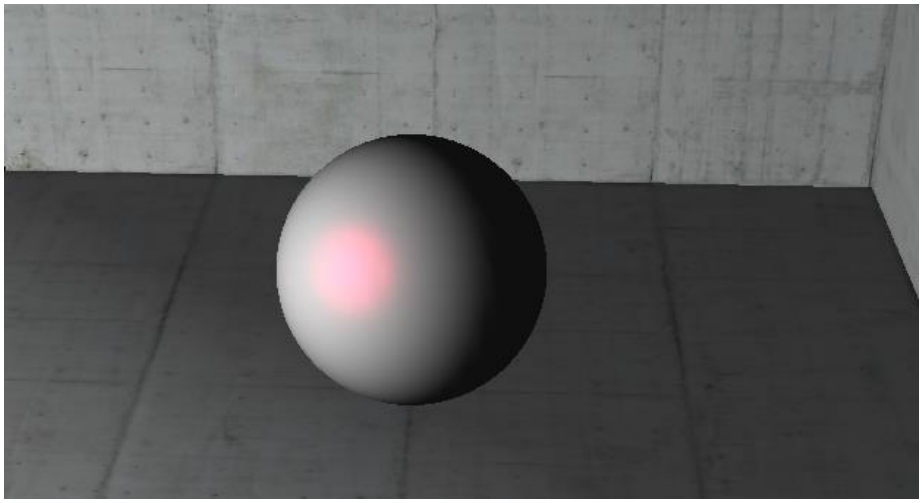


Figura 25: Aplicação da luz especular

- Emissiva – Cor da luz emitida por um objeto.

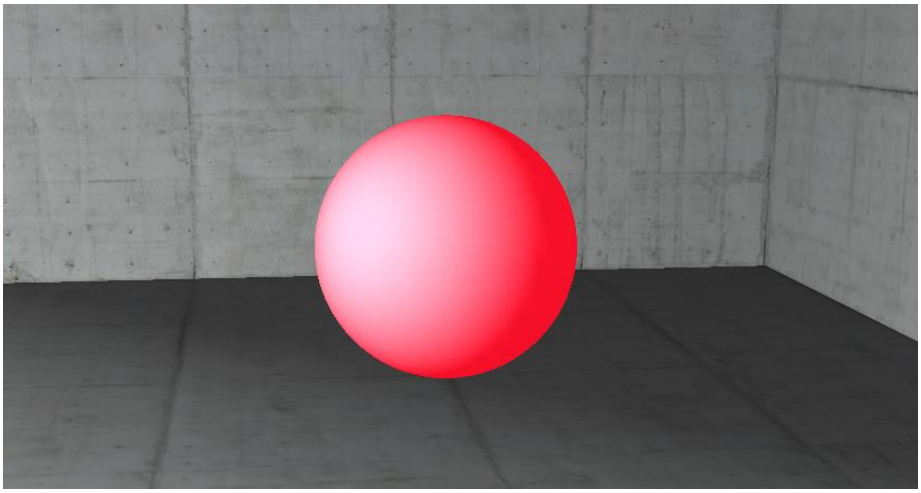


Figura 26: Aplicação da luz emissiva

- Ambiente – Corresponde à cor do objeto quando este se encontra sem luz.

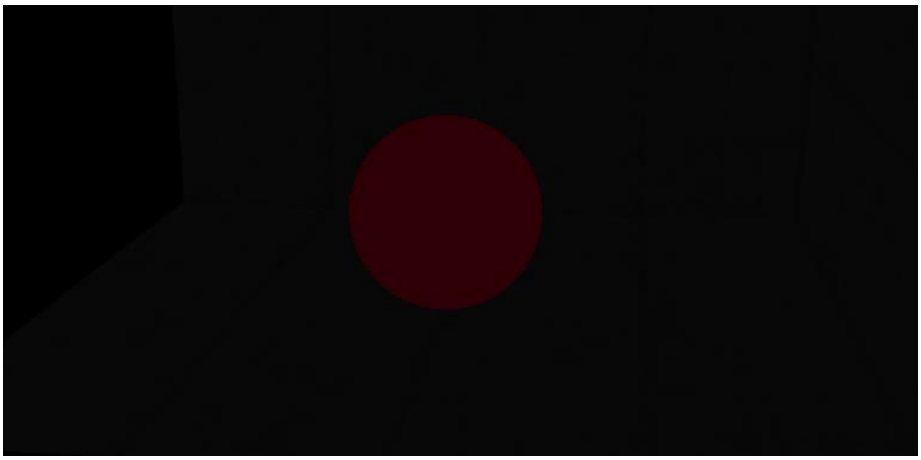


Figura 27: Aplicação da luz ambiente

A junção destas componentes possibilita a perceção complexa e real da luz e da sua reflexão sobre os objetos.

Na imagem que se segue, é possível visualizar a junção destas componentes.

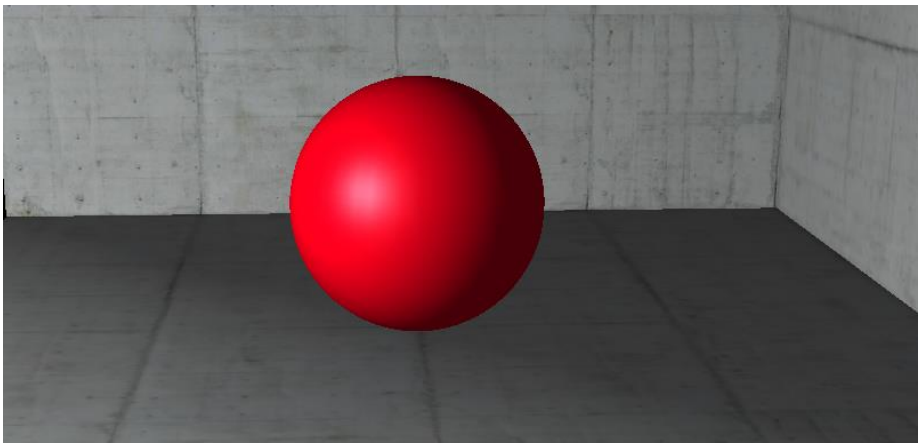


Figura 28: Aplicação das cores

Para concretizar a introdução da iluminação no projeto, o grupo viu-se forçado a alterar o *engine* da fase anterior. Assim sendo, foram criadas duas novas estruturas, *light* e *obj_Light*, que permitem guardar toda a informação necessária para a iluminação do cenário e as diferentes cores dos diversos objetos.

```
class light {
private:

        std::string tipo;

        float pointX;
        float pointY;
        float pointZ;

        float dirX;
        float dirY;
        float dirZ;

        float phi;
        float theta;
```

Esta estrutura permite armazenar os dados relativos ao tipo de luz(es) que serão aplicadas ao cenário. Deste modo, esta estrutura contém o **tipo da luz** a ser usada, podendo esta ser *ponto de luz*, *direcional* ou *foco de luz*; a sua **posição**, no caso de se tratar de um ponto de luz ou de um foco de luz; a sua **direção**, quando se pretende utilizar

uma luz direcional ou um foco de luz; um valor de **phi**, que representa o ângulo do *cutoff* da luz direcional e um valor de **theta**, que caracteriza o ângulo expoente da luz direcional.

```
class obj_Light {
private:

    std::string texture;

    int dValue;
    float diffR;
    float diffG;
    float diffB;

    int sValue;
    float specR;
    float specG;
    float specB;

    int eValue;
    float emR;
    float emG;
    float emB;

    int aValue;
    float ambR;
    float ambG;
    float ambB;
```

Por sua vez, a estrutura *obj_Light* permite guarda a informação dos diferentes componentes da cor correspondentes a um objeto, bem como a sua textura, caso exista.

Posto isto, a estrutura é constituída pelo **nome** da textura atribuída à figura (no caso de esta não possuir qualquer textura, o nome atribuído será "none"); um **valor** para **cada tipo de componente**, indicando se esse componente se encontra ativo ou não; um valor **diffR/G/B**, representando as componentes *Red, Green* e *Blue* da cor difusa; um valor **specR/G/B**, retratando as componentes *Red, Green* e *Blue* da cor especular; um valor **emR/G/B**, representando as componentes *Red, Green* e *Blue* da cor especular e um valor **ambR/G/B**, apresentando as componentes *Red, Green* e *Blue* da cor ambiente.

De seguida, foi adicionada uma nova função ao *parser*.

```
std::vector<light> getXMLight(std::string file_name);
```

Esta função vai dar *parse* ao campo *lights* do ficheiro *XML*, guardando no vetor, que será posteriormente retornado, todas as luzes definidas.

Na função *dadosXML*, foram adicionados parâmetros de forma a ler todas as variáveis relacionadas com as texturas e as luzes dentro dos modelos do ficheiro *XML*.

```
class xmlData {

    std::vector<std::string> modelos;
    std::vector<std::vector<triangulo>> pontos;
    std::vector<std::vector<triangulo>> coords;
    std::vector<std::vector<triangulo>> normais;
    std::vector<float> pontosDeControlo;
    std::vector<obj_Light> materiais;

    float translateX;
    float translateY;
    float translateZ;
    float timeTranslate;

    int rotateX;
    int rotateY;
    int rotateZ;
    float rotateAngulo;
    float timeRotate;

    float scaleX;
    float scaleY;
    float scaleZ;

    int ordemTranslate;
    int ordemRotate;
    int ordemScale;

    std::vector<xmlData> filhos;
```

À estrutura *xmlData* foram adicionados três vetores, *materiais*, *normais* e *coords*, permitindo guardar as informações alusivas às luzes e texturas das figuras na estrutura previamente criada, *obj_Light*, guardar as normais e as coordenadas de textura das figuras, respetivamente.

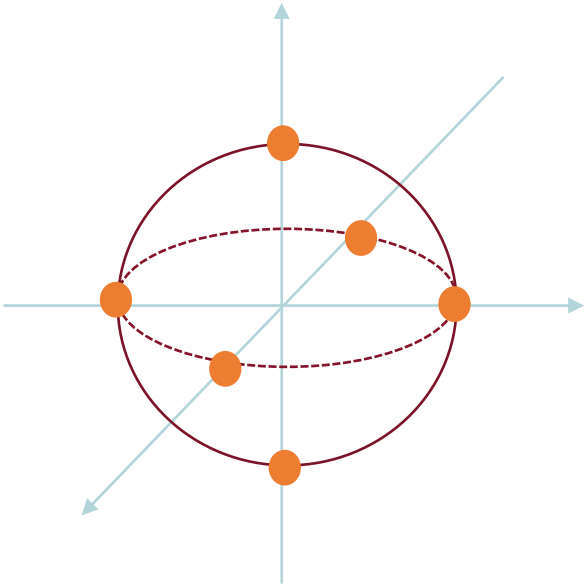
No que diz respeito ao *loader*, foram adicionados métodos de leitura de ficheiros *.coords* e *.normals*.

Por fim, na classe responsável pelo desenho dos cenários, foram acrescentadas duas novas funções, *initTexture*, que inicializa a textura e guarda a sua informação num *array*, sendo que esta é invocada uma vez por figura; e a *applyLight*, que aplica a luz, que é lida a partir do ficheiro *XML,* no cenário.

O restante processo passou pela atualização de outras funções previamente existentes, de forma a suportar a luz dos materiais, as coordenadas de texturas e as normais de cada figura, sendo que as texturas e as normais serão armazenadas em VBOs.

SISTEMA SOLAR

Para a implementação da iluminação no Sistema Solar, foram adicionados seis pontos de luz na superfície do Sol, como é possível verificar na figura que se segue.



BONECO DE NEVE

A construção do boneco de neve consistiu na junção da maioria das primitivas desenvolvidas ao longo da elaboração do projeto.

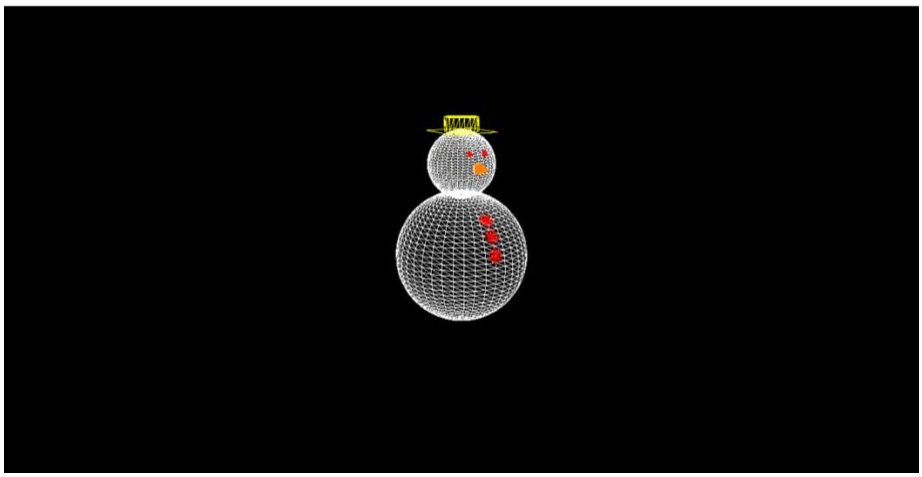


Figura 29: Representação do Boneco de Neve - LINE

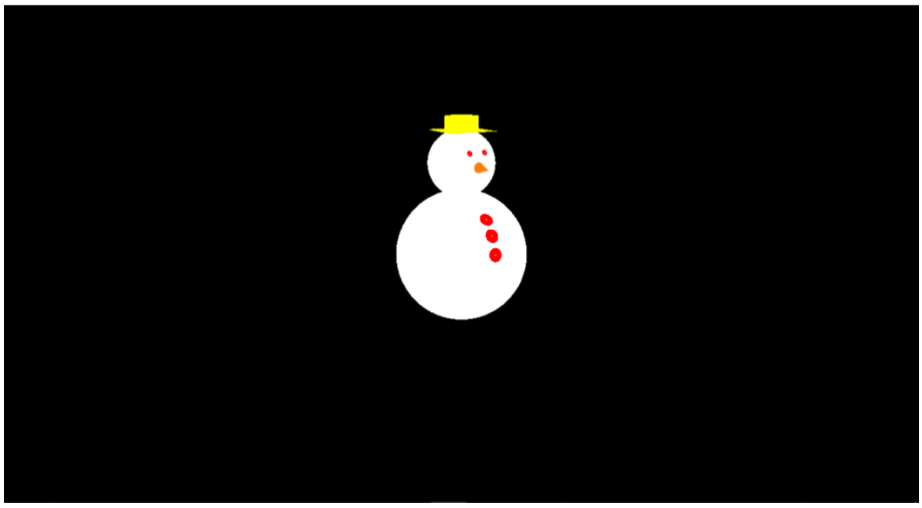


Figura 30: Representação do Boneco de Neve - FILL

No que diz respeito à iluminação introduzida no boneco de neve, foi aplicada a componente difusa da cor, permitindo obter o resultado que se segue.

Foram ainda utilizados dois tipos distintos de luz. Foi adicionada uma luz direcional, no sentido negativo do eixo yy. Adicionalmente, está presente um ponto de luz, na posição $(0,50,0)$, permitindo a iluminação das paredes da caixa onde se encontra o boneco de neve.

Figura 31: Boneco de Neve 1

Figura 32: Boneco de Neve 2

MOVIMENTO DA CÂMARA

Similarmente à fase anterior, de modo a facilitar a verificação da concretização das transformações geométricas, foram alterados os parâmetros relativos à posição da câmara. Nesta última fase, foi implementado um movimento de câmara em terceira pessoa. Foram também substituídas as ações das setas do teclado, pelo rato.

Para este efeito, e uma vez que é necessária a utilização de coordenadas esféricas, foi utilizada a função sphericalcartesian.

```
void sphericalcartesian() {
    camX = radius * sin(alfa) * cos(beta);
    camZ = radius * cos(alfa) * cos(beta);
    camY = radius * sin(beta);
}
```

Assim, tem-se

```
gluLookAt(camX + refX , camY + refY , camZ + refZ,
          refX , refY , refZ,
          0.0f , 1.0f , 0.0f);
```

Desta forma, é possível, através das teclas e do movimento do rato, alterar a posição da câmara, modificando os valores de alfa e de beta.

```
void processMouseButtons(int button, int state, int xx, int yy)
{
    if (state == GLUT_DOWN) {
        startX = xx;
        startY = yy;
        if (button == GLUT_LEFT_BUTTON)
            tracking = 1;
        else if (button == GLUT_RIGHT_BUTTON)
            tracking = 2;
        else
            tracking = 0;
    }
    else if (state == GLUT_UP) {
        if (tracking == 1) {
            alfa += (xx - startX);
            beta += (yy - startY);
        }
        else if (tracking == 2) {

            radius -= yy - startY;
            if (radius < 3)
                radius = 3.0;
        }
        tracking = 0;
    }
}
```

```
void processMouseMotion(int xx, int yy)
{
    int deltaX, deltaY;
    int alphaAux, betaAux;
    int rAux;

    if (!tracking)
        return;

    deltaX = xx - startX;
    deltaY = yy - startY;

    if (tracking == 1) {

        alphaAux = alfa + deltaX;
        betaAux = beta + deltaY;

        if (betaAux > 85.0)
            betaAux = 85.0;
        else if (betaAux < -85.0)
            betaAux = -85.0;

        rAux = radius;
    }
    else if (tracking == 2) {

        alphaAux = alfa;
        betaAux = beta;
        rAux = radius - deltaY;
        if (rAux < 3)
            rAux = 3;
    }

    camX = rAux * sin(alphaAux * 3.14 / 180.0) * cos(betaAux * 3.14 / 180.0);
    camZ = rAux * cos(alphaAux * 3.14 / 180.0) * cos(betaAux * 3.14 / 180.0);
    camY = rAux * sin(betaAux * 3.14 / 180.0);
}
```

Por outro lado, através das teclas regulares, é possível alterar o ponto para onde a câmara está a olhar, modificando os valores de refX, refY e refZ.

```
case 'w': {
    refZ -= 5 * cos(alfa);
    refX -= 5 * sin(alfa);
} break;
case 's': {
    refZ += 5 * cos(alfa);
    refX += 5 * sin(alfa);
} break;

case 'd': {
    refX += 5 * cos(alfa);
    refZ += 5 * (-sin(alfa));
} break;
case 'a': {
    refX -= 5 * cos(alfa);
    refZ -= 5 * (-sin(alfa));
} break;
case 'q': {
    refY += 3;
} break;
case 'e': {
    refY -= 3;
} break;
case '-': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
} break;
case '.': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
} break;
case ',': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
} break;
```

CONCLUSÕES

A realização da quarta fase do projeto do Sistema Solar permitiu que o grupo solidificasse conhecimento sobre a iluminação e animação de cenários, bem como o contínuo ganho de prática de aplicação de transformações e uso de VBOs.

Ao longo da realização das tarefas solicitadas para esta fase, o grupo deparou-se com algumas dificuldades no que diz respeito ao mapeamento das texturas às diversas primitivas gráficas, uma vez que é crucial ter em atenção a ordem dos pontos de criação de cada primitiva.

A implementação do Sistema Solar dinâmico, bem como o êxito na introdução de iluminação e texturas no cenário possibilitou o que o grupo considera ser o sucesso da última fase deste projeto.

BIBLIOGRAFIA

[1] Ramires, A., n.d. Terrain - Notas Para A Componente Prática De Computação Gráfica.

ANEXOS

Apresenta-se em seguida a representação gráfica do Sistema Solar dinâmico, com a introdução da iluminação e texturas nos respetivos corpos celestes.

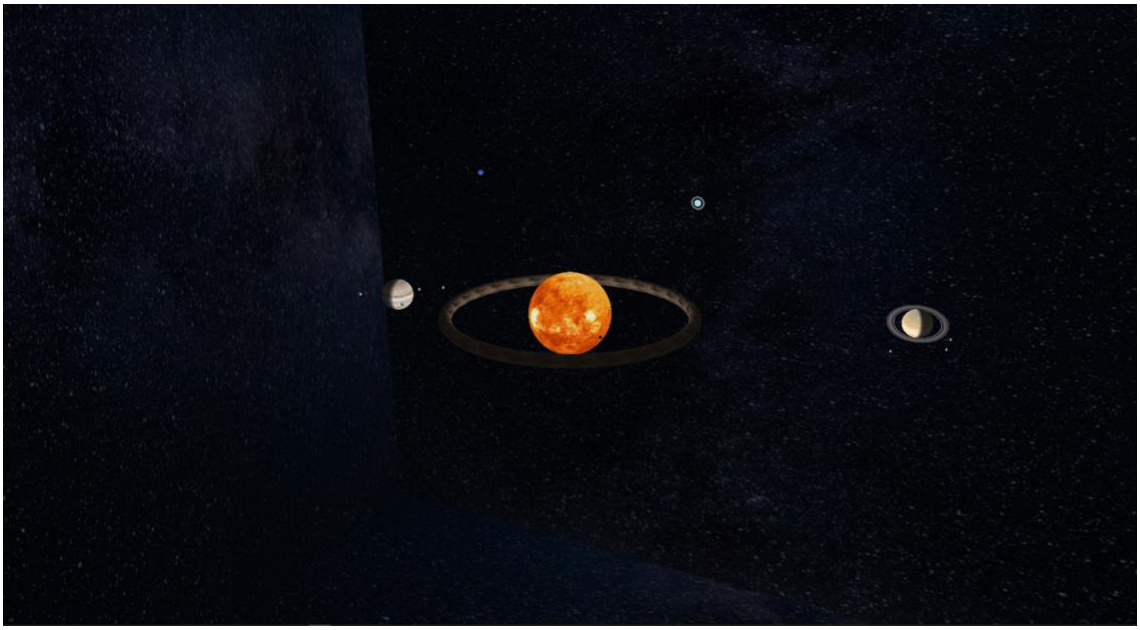


Figura 33: Sistema Solar 1

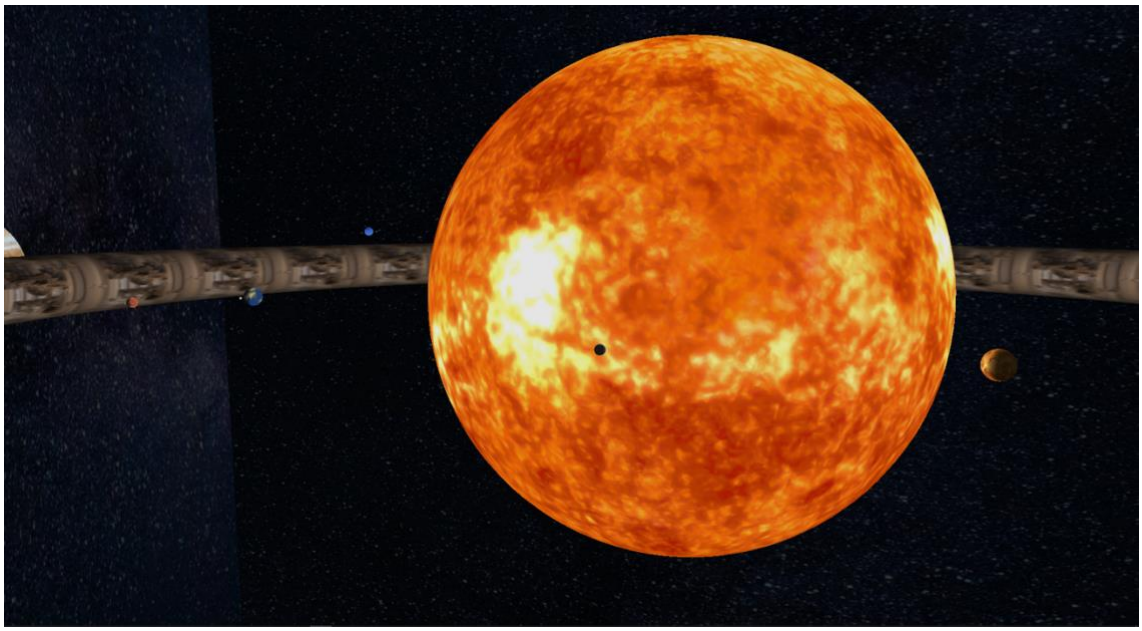


Figura 34: Sistema Solar 2

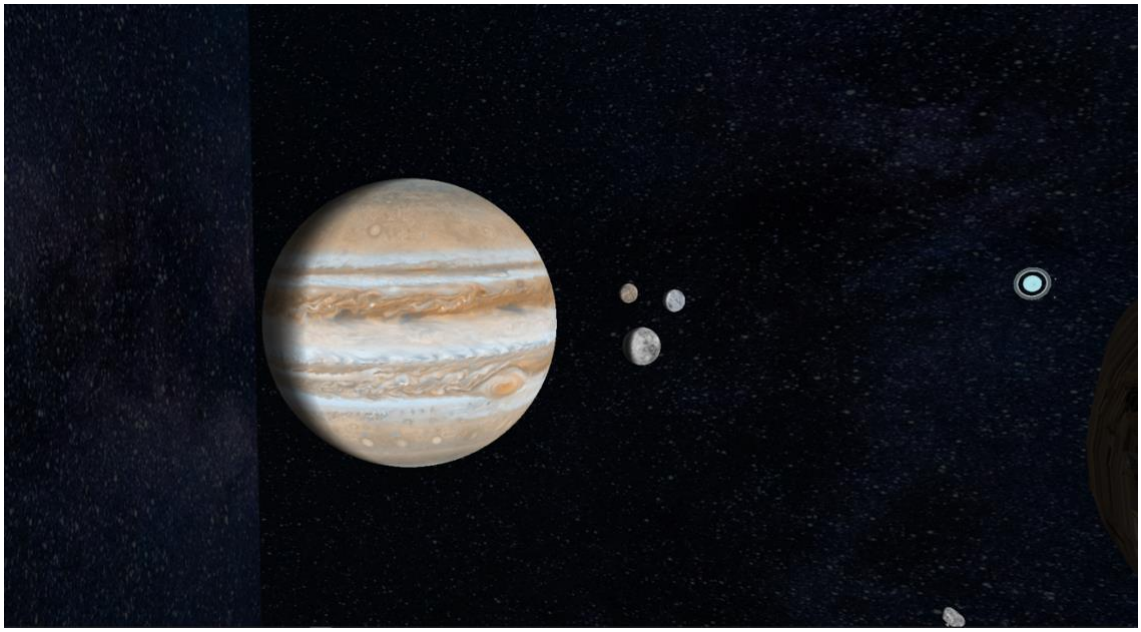


Figura 35: Sistema Solar 3

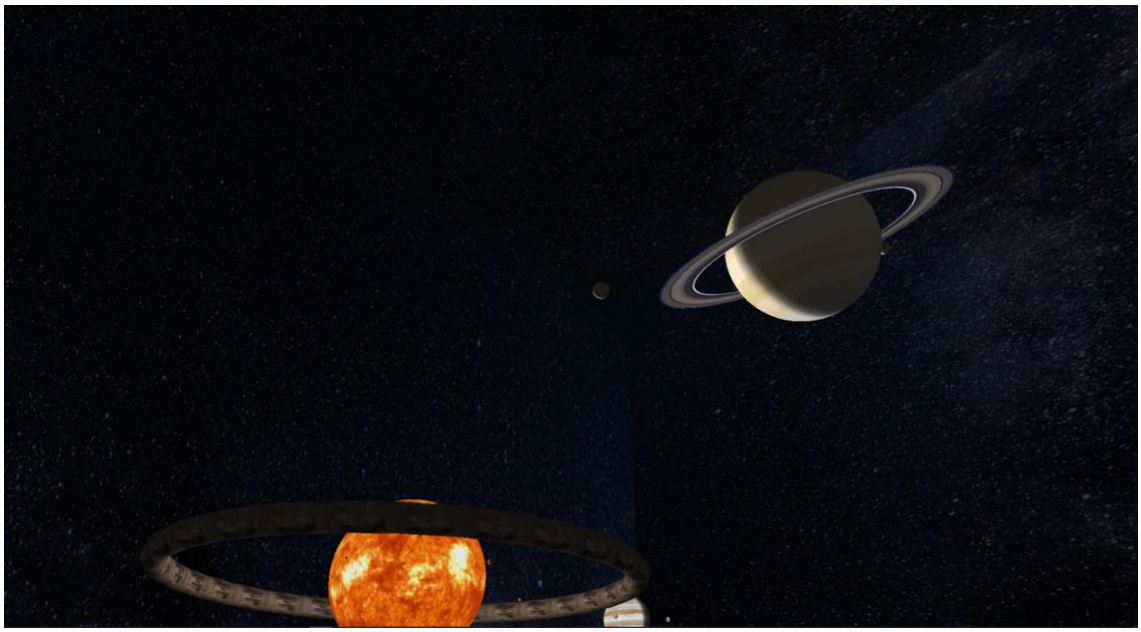


Figura 36: Sistema Solar 4

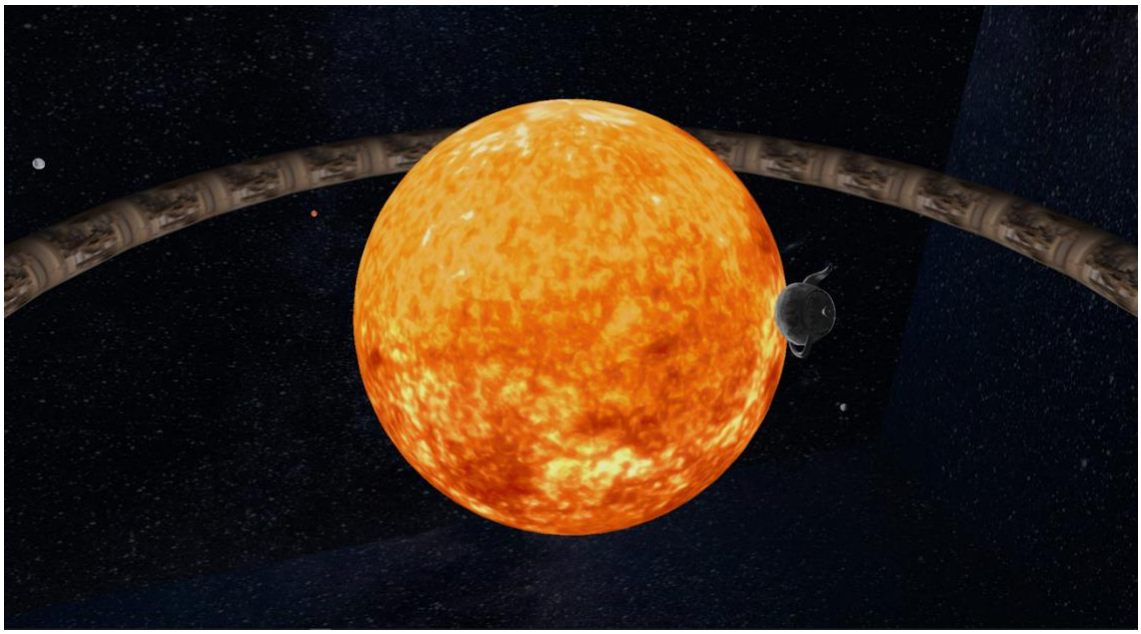


Figura 37: Sistema Solar 5