UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Computação Gráfica

TRABALHO PRÁTICO - FASE IV

Adriana Henriques Esteves Teixeira Meireles A82582
Ana Marta Santos Ribeiro A82474
Carla Isabel Novais da Cruz A80564
Jéssica Andreia Fernandes Lemos A82061

MIEI - 3º Ano - 2º Semestre

Braga, 15 de maio de 2019

Índice

1 Introdução

Nesta quarta fase do projeto, prentende-se que ao sistema solar desenvolvido anteriormente sejam adicionadas texturas, iluminação e materiais, de modo a obter uma implementação mais realista do mesmo. De forma a cumprir os objetivos, foi necessário efetuar alterações tanto na aplicação *engine* como na *generator*. Deste modo, no presente relatório iremos abordar as modificações efetuadas em cada uma das aplicações, bem como explicar todas as decisões tomadas ao longo do processo.

Por fim, com o intuito de demonstrar o modo como funciona o projeto e o resultado obtido, serão apresentados alguns exemplos.

2 Organização do Código

Tendo em conta a implementação realizada nas fases anteriores, optamos por seguir a mesma abordagem, continuando a ter duas aplicações, a *Engine* e *Generator*.

2.1 Generator

Esta aplicação irá permitir a geração dos vértices das primitivas, bem como as normais e as coordenadas das texturas de cada uma.

2.1.1 Point

Nesta última fase decidimos manter a estrutura Point que permite armazenas as coordenadas de um ponto.

2.1.2 Figures

Para a realização desta fase do projeto foi necessário obter as normais e as coordenadas das texturas das primitivas elaboradas em fases anteriores.

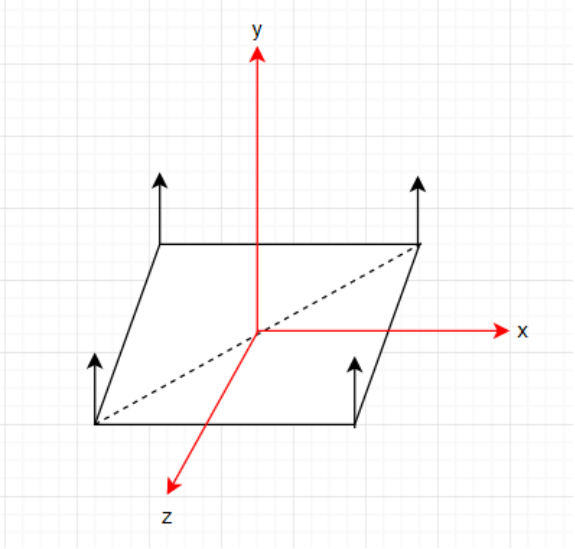
⇒ **Plane**

Normal

Dado que o plano se encontra no eixo xOz, verificamos que o vetor normal, em cada vértice que constitui o plano, é exatamente igual a

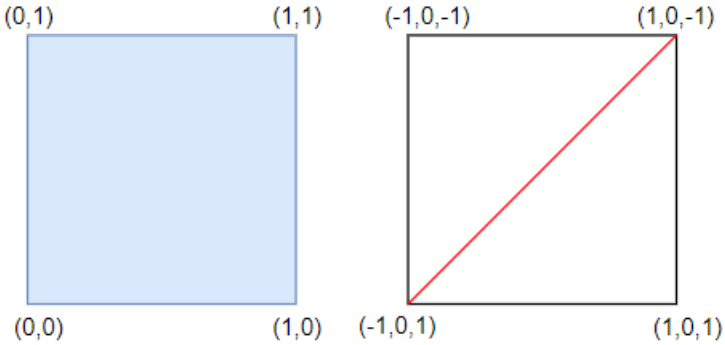
$$\overrightarrow{N} = \{0, -1, 0\}$$

Na figura em baixo, mostramos a representação de cada um dos vetores normais do plano.



Texturas

Para a obtenção das coordenadas das texturas, cada uma vai estar incluída na área ocupada pelos triângulos que formam o plano. De seguida, pode observar-se as coordenadas da textura que serão utilizadas em cada um dos vértices do plano.



Em baixo é apresentada a normal e as coordenadas da textura associadas a cada vértice do plano.

Vértice	Normal	Textura
(1,0,1)	(0,1,0)	(1,0)
(1,0,-1)	(0,1,0)	(1,1)
(-1,0,1)	(0,1,0)	(0,0)
(-1,0,1)	(0,1,0)	(0,0)
(1,0,-1)	(0,1,0)	(1,1)
(-1,0,-1)	(0,1,0)	(0,1)

⇒ **Box**

Normais

De forma a obter o conjunto das normais associadas a um vértice, foi necessário definir para cada uma das 6 faces que constituem a box, a normal correspondente.

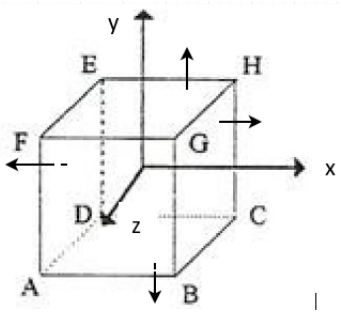


Figura 1. Sistema Solar

Assim, de seguida apresentamos os vetores normais asscociado a cada uma:

- **Topo** – (0,1,0)
- **Base** – (0,-1,0)
- **Face Frontal** – (0,0,1)
- **Face Traseira** – (0,0,-1)
- **Face Direita** – (1,0,0)
- **Face Esquerda** - (-1,0,0)

Texturas

De modo a obter as texturas da box, representamos a sua planificação numa figura 2D em que identificamos cada face, tal como se pode observar de seguida:

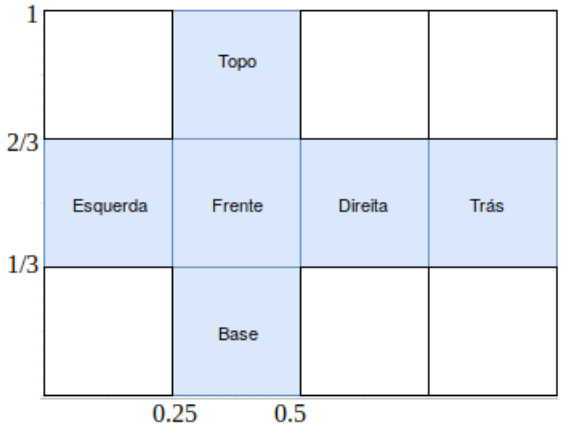


Figura 2. Textura da box

Tendo em conta que a planificação se encontra limitada num quadrado de largura 1, foi possível definir os pontos que permitem obter as coordenadas de cada face. Assim sendo, tomaremos como exemplo a face frontal, para demonstrar o processo.

Como é possível observar na Figura 2 as coordenadas para esta face podem tomar os seguintes valores: x[0.25,0.5] e y[1/3,2/3]. De seguida, é necessário gerar os pontos da textura para cada um dos triângulos que constituem esta face. Para tal, iremos explicar tomando o exemplo apresentado na primeira fase, em que a box tem largura, comprimento e altura 2 e o número de divisões pretendidas é 2. Assim, tal como demonstrado no relatório da primeira fase, iremos obter a seguinte divisão da face:

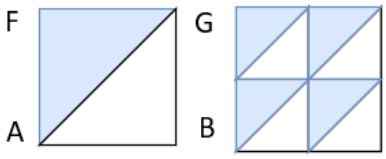


Figura 3. Divisao da face da box

Para obter as coordenadas de cada um dos pontos, começamos por obter o tamanho da base e da altura dos triângulos mais pequenos. Para tal, dividimos o tamanho do triângulo grande pelo número de divisões pretendidas. Assim, foi possível replicá-lo na horizontal e na vertical para obter todos os triângulos pretendidos.

É importante referir que foi necessário diferenciar o modo como são obtidas as coordenadas dos triângulos gerados a partir do triângulo azul do triângulo branco, dado que o valor das coordenadas da textura dos vértices são diferentes. Assim, é possível verificar as coordenadas dos mesmos:

- **Coordenadas dos triângulo branco**

```
for (int i = 0; i < divisions; i++) {
    for (int j = 0; j < divisions; j++) {
        (*textureList).push_back(0.25f + i * 0.25f /divisions);
        (*textureList).push_back(1.0f/3.0f + j * 1.0f/3.0f /
           divisions);
        (*textureList).push_back(0.25f + (i + 1) * 0.25f /
           divisions);
        (*textureList).push_back(1.0f/3.0f + j * 1.0f/3.0f /
           divisions);
           (*textureList).push_back(0.25f + (i + 1) * 0.25f /
           divisions);
        (*textureList).push_back(1.0f/3.0f + (j + 1) * 1.0f /
           3.0f / divisions);
```

- **Coordenadas dos triângulo azul**

```
for (int i = 0; i < divisions; i++) {
    for (int j = 0; j < divisions; j++) {
        (*textureList).push_back(0.5f - i * 0.25f / divisions);
        (*textureList).push_back(2.0f/3.0f - j * 1.0f/3.0f /
           divisions);
        (*textureList).push_back(0.5f - (i+1) * 0.25f /
           divisions);
           (*textureList).push_back(2.0f/3.0f - j * 1.0f/3.0f /
           divisions);
        (*textureList).push_back(0.5f - (i+1) * 0.25f /
           divisions);
        (*textureList).push_back(2.0f/3.0f - (j+1) * 1.0f/3.0f /
           divisions);
```

⇒ **Esfera**

Normais

A obtenção das normais da esfera é realizada através da consideração da origem e de qualquer ponto que esteja na esfera. Dado que o desenho da esfera já partia deste pressuposto, assim, estas coordenadas podem corresponder às normais da figura. Desta forma obtemos o vetor $\vec{N} = P - O = P$.

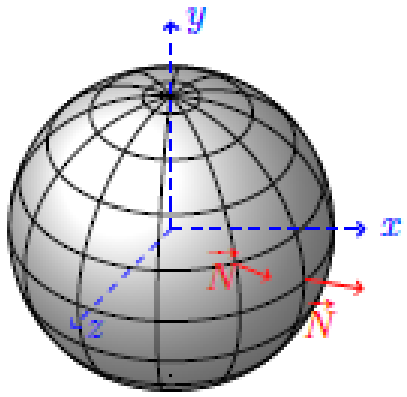


Figura 4. Normais da esfera

Depois é necessário normalizar o vetor que é realizada através de uma fórmula já existente, $\vec{N} = \frac{P}{\|P\|}$.

Texturas

Após a criação das normais, é necessário definir as coordenadas relativas à textura. Os vértices da esfera são gerados por camadas e o mesmo acontecerá nesta fase, considerando que cada camada da esfera corresponde à camada da textura. Desta forma, a cada três pontos gerados, associa-se a estes uma textura. Com o desenrolar do ciclo for, é possível obter os vértices, as normais e as respetivas texturas:

```
for (int i = 0; i < layers; i++){
    beta = i * (M_PI / layers) - M_PI_2;
    nextBeta = (i + 1) * (M_PI / layers) - M_PI_2;
    for (int j = 0; j < slices; j++){
        alpha = j * 2 * M_PI / slices;
        nextAlpha = (j + 1) * 2 * M_PI / slices;
        //..
        (*texture).push_back( (float) j/slices );
        (*texture).push_back( (float)(i+1)/layers );
        (*texture).push_back( (float) j/slices );
        (*texture).push_back( (float) i/layers );
        (*texture).push_back( (float)(j+1)/slices );
        (*texture).push_back( (float)(i+1)/layers );
        (*texture).push_back( (float)(j+1)/slices );
        (*texture).push_back( (float)(i+1)/layers );
```

```
16          (*texture).push_back( (float) j/slices );
17          (*texture).push_back( (float) i/layers );
18          (*texture).push_back( (float)(j+1)/slices );
19          (*texture).push_back( (float) i/layers );
20          //..
21      }
22  }
```

⇒ **Cone**

Normais

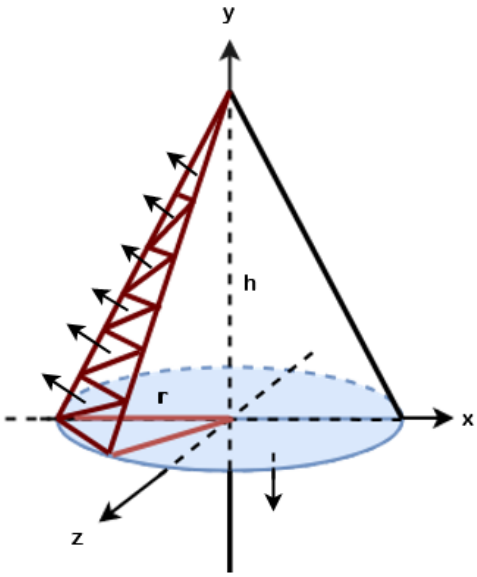


Figura 5. Normais do cone

Para o cálculo dos vetores normais do cone, começamos por identificar o vetor normal à base deste. Tendo em conta que este encontra-se contido no plano xOz, o vetor normal será:

$$\overrightarrow{N} = \{0, -1, 0\}$$

```
1       p3.x = 0;
2       p3.y = -1;
3       p3.z = 0;
4       //save point
```

De seguida, para obter os vetores normais do restante cone calculamos o mesmo para um dos pontos de uma circunferência paralela à base.

Como podemos verificar pela Figura 6, tendo em conta o ângulo α, conseguimos obter o vetor normal:

$$\overrightarrow{N} = \{cos(\alpha), 0, sin(\alpha)\}$$

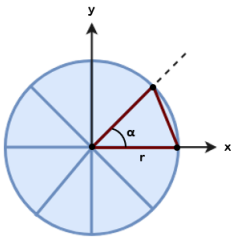


Figura 6. Circunferência paralela ao plano x0z

Para obter a inclinação do cone, neste caso θ, tivemos em consideração o raio e a altura. Observando a Figura 11 conseguimos chegar à seguinte expressão:

$$\tan\theta = \frac{h}{r} \Rightarrow \theta = \text{atan}\,(\frac{r}{h})$$

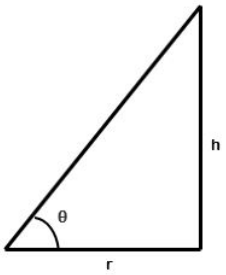


Figura 7. Triângulo

Assim, qualquer ponto da lateral do cone terá como normal:

$$\overrightarrow{N} = \{cos(\alpha) * cos(\theta), sin(\theta), sin(\alpha) * cos(\theta)\}$$

Para a criação deste recorremos à função *drawNormalPoints* que irá permitir gerar os pontos. Esta terá em conta a inclinação do cone e os valores de *teta* bem como de *tetaNext*.

```
Point drawNormalPoints(float angle, float teta) {
    Point p;
    p.x = cos(angle) * sin(teta);
    p.y = sin(angle);
    p.z = cos(angle) * cos(teta);

    return p;
}
```

Texturas

Para a aplicação das texturas começamos por definir o formato que estas terão.

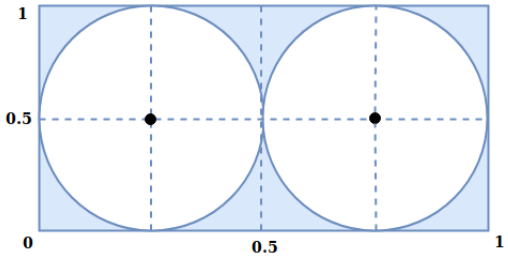


Figura 8. Formato da textura

Tendo em conta a figura apresentada decidimos que a circunferência da esquerda representará a base do cone e a da direita a sua lateral. Para formar o vetor que conterá os valores correspondentes à textura, recorremos aos valores de *teta* e *tetaNext*.

```
    (*texture).push_back(0.25f); //x1
    (*texture).push_back(0.5f); //y1
    (*texture).push_back(0.25f + cos(tetaNext) / 4.0f); //x2
    (*texture).push_back(0.5f + sin(tetaNext) / 2.0f); //y2
    (*texture).push_back(0.25f + cos(teta) / 4.0f); //x3
    (*texture).push_back(0.5f + sin(teta) / 2.0f); //y3
```

Enquanto que para a textura lateral do cone teremos em consideração os quadrados que são formados.

```
    float res = (float) (layers - i) / layers;
    float resNext = (float) (layers - (i+1)) / layers;
```

```
3
4       texture.push_back(0.75f + 0.25f * cos(teta) * res); //x1
5       texture.push_back(0.5f +  0.5f * sin(teta) * res); //y1
6
7       texture.push_back(0.75f + 0.25f * cos(tetaNext) * res); //x2
8       texture.push_back( 0.5f +  0.5f * sin(tetaNext) * res); //y2
9
10      texture.push_back(0.75f + 0.25f * cos(tetaNext) * resNext); //x3
11      texture.push_back( 0.5f +  0.5f * sin(tetaNext) * resNext); //y3
12
13      texture.push_back(0.75f + 0.25f * cos(teta) * res); //x4
14      texture.push_back( 0.5f +  0.5f * sin(teta) * res); //y4
15
16      texture.push_back(0.75f + 0.25f * cos(tetaNext) * resNext); //x5
17      texture.push_back( 0.5f +  0.5f * sin(tetaNext) * resNext); //y5
18
19      texture.push_back(0.75f + 0.25f * cos(teta) * resNext); //x6
20      texture.push_back( 0.5f +  0.5f * sin(teta) * resNext); //y6
```

⇒ **Cilindro**

Normais

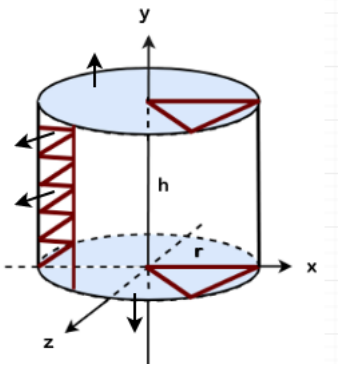


Figura 9. Normais do Cilindro

Para o cálculo dos vetores normais, podemos dividir o cilindro em três partes, o topo, a base e a parte lateral. Assim, de seguida serão apresentadas as normais de cada vértice:

- **Topo** – (0,0,1)

- **Base** – (0,0,-1)
- **Parte lateral** – (sin(teta),0,cos(teta)) - em que teta é a amplitude em que se encontra o vértice

Texturas

De forma a obter as coordenadas da textura do cilindro, começamos por representar a sua planificação como pode ser observado de seguida:

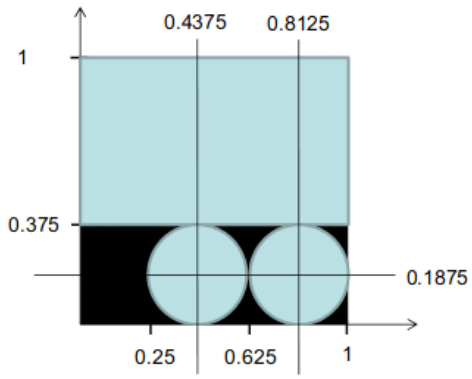


Figura 10. Planificação do Cilindro

Nesta consideramos que o topo do cilindro é a circunferência da esquerda e a base a da direita. Tendo isto em consideração de seguida será explicado para cada parte como se obtém as coordendas.

Tanto para a base do cilindro como para o topo é necessário posicionar no centro da circunferência e obter os seus pontos.

```
1  //base
2  for (int i = 0; i < slices; i++) {
3          (*texture).push_back(0.8125f);
4          (*texture).push_back(0.1875f);
5          (*texture).push_back(0.8125f + 0.1875f * sin(teta + alpha));
6          (*texture).push_back(0.1875f + 0.1875f * cos(teta + alpha));
7          (*texture).push_back(0.8125f + 0.1875f * sin(teta));
8          (*texture).push_back(0.1875f + 0.1875f * cos(teta));
9  }
10 //topo
11 for (int i = 0; i < slices; i++) {
12         (*texture).push_back(0.4375f);
13         (*texture).push_back(0.1875f);
```

```
14          (*texture).push_back(0.4375f + 0.1875f * sin(teta));
15          (*texture).push_back(0.1875f + 0.1875f * cos(teta));
16          (*texture).push_back(0.4375f + 0.1875f * sin(tetaNext));
17          (*texture).push_back(0.1875f + 0.1875f * cos(tetaNext));
18  }
```

Para a parte lateral do cilindro é preciso para cada slice iterar sobre o números de layers, como podemos verificar na Figura 9. Deste modo a parte lateral pode ser obtida da seguinte forma:

```
1  for (int i = 0; i < layers; i++) {
2           for (int j = 0; j < slices; j++) {
3               (*texture).push_back((1.0f/slices) * (j));
4               (*texture).push_back(i*0.625f/layers + 0.375f);
5
6               (*texture).push_back((1.0f/slices) * (j + 1));
7               (*texture).push_back(i*0.625f/layers + 0.375f);
8
9               (*texture).push_back((1.0f/slices) * (j + 1));
10              (*texture).push_back((i+1)*0.625f/layers + 0.375f);
11
12              (*texture).push_back((1.0f/slices) * (j));
13              (*texture).push_back(i*0.625f/layers + 0.375f);
14
15              (*texture).push_back((1.0f/slices) * (j + 1));
16              (*texture).push_back((i+1)*0.625f/layers + 0.375f);
17
18              (*texture).push_back((1.0f/slices) * j);
19              (*texture).push_back((i+1)*0.625f/layers + 0.375f);
20          }
21 }
```

⇒ **Torus**

Normais

Para a obter as normais do torus, seguimos o mesmo raciocínio aplicado na esfera, em que o próprio ponto corresponde à sua normalização. Assim, as expressões ficam iguais à da esfera.

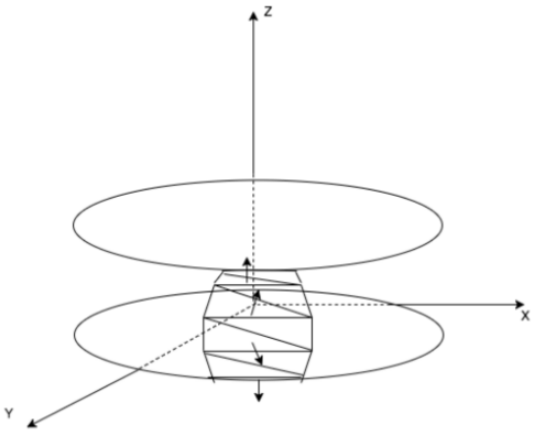


Figura 11. Normais do Torus

Texturas

Relativamente à textura desta figura, cada anel do torus corresponde a uma textura, fazendo com que chegando ao fim de todas as iterações tenha sido preenchido cada um destes aneis, pelo que o torus está preenchido na totalidade:

```
for (int i = 0; i < layers; i++){
    float beta = i * (2*M_PI)/layers;
    float nextBeta = (i+1) * (2*M_PI)/layers;
    for (int j = 0; j < slices; j++){
        float alpha = j  * (2*M_PI)/slices;
        float nextAlpha = (j+1) * (2*M_PI)/slices;
        //...
        (*texture).push_back( (float) j/slices );
        (*texture).push_back( (float) i/layers );
        (*texture).push_back( (float)(j+1)/slices );
        (*texture).push_back( (float) i/layers );
        (*texture).push_back( (float) j/slices );
        (*texture).push_back( (float)(i+1)/layers );
        (*texture).push_back( (float) j/slices );
        (*texture).push_back( (float)(i+1)/layers );
        (*texture).push_back( (float)(j+1)/slices );
        (*texture).push_back( (float) i/layers );
        (*texture).push_back( (float)(j+1)/slices );
        (*texture).push_back( (float)(i+1)/layers );
        //...
    }
}
```

2.1.3 Patch

De modo a calcular as normais, recorremos às matrizes apresentadas para a obtenção dos pontos na fase anterior:

⇒ Matrizes a e b

$$a = \begin{bmatrix} a^3 & a^2 & a & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} b^3 & b^2 & b & 1 \end{bmatrix}$$

⇒ Matrizes com Pontos de Controlo

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_30 & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

⇒ Matrizes de Bézier

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

Tendo em conta os valores de a e b, que se encontram entre 0 e 1, é possível obter um ponto da superfície de Bézier, como podemos verificar de seguida.

$$P(a,b) = a \times M \times P \times M \times b;$$

Normais

Para calcular a normal do ponto *P(a,b)*, foi necessário efetuar o produto cruzado das suas tangentes normalizadas, que neste caso correspondem aos vetores a e b. Assim,

implementamos a função *getTangent*, que permite obter as tangentes, aplicando de seguida a função *cross* que devolve o produto cruzado das mesmas para que pudesse ser normalizado, tal como pode ser observado de seguida:

```
  // declaracao de variaveis e obtencao dos pontos de controlo
for(int i = 0; i < tessellation; i++){
        for (int j = 0; j < tessellation; j++) {
            u = i*t;
            v = j*t;
            uu = (i+1)*t;
            vv = (j+1)*t;
            p0 = getPoint(u, v, coordenadasX, coordenadasY,
                coordenadasZ);
            tangenteU =
                getTangent(u,v,coordenadasX,coordenadasY,coordenadasZ,0);
            tangenteV =
                getTangent(u,v,coordenadasX,coordenadasY,coordenadasZ,1);
            cross(tangenteU,tangenteV,res);
            normalize(res);
            n0 = new Point(res[0],res[1],res[2]);
            //calculo dos restantes pontos, normais e texturas
        }
}
```

Texturas

Foi ainda necessário definir as texturas pela mesma ordem que os ponto foram gerados. Assim, de seguida é apresentado o código que permite obter as texturas:

```
 // declaracao de variaveis e obtencao dos pontos de controlo
for(int i = 0; i < tessellation; i++){
        for (int j = 0; j < tessellation; j++) {
            u = i*t;
            v = j*t;
            uu = (i+1)*t;
            vv = (j+1)*t;
            //Calculo dos pontos e das normais
            textureList->push_back(1-u); textureList->push_back(1-v);
            textureList->push_back(1-uu); textureList->push_back(1-v);
            textureList->push_back(1-u); textureList->push_back(1-vv);
            textureList->push_back(1-u); textureList->push_back(1-vv);
            textureList->push_back(1-uu); textureList->push_back(1-v);
```

```
14              textureList->push_back(1-uu);
                    textureList->push_back(1-vv);
15          }
16  }
```

2.2 Engine

Nesta fase tornou-se necessário disponibilizar funcionalidades de iluminação bem como a aplicação de texturas.

2.2.1 Parser

⇒ **Ficheiros de Input**

O formato dos ficheiros input sofreu alterações pelo que foi necessário fazer um correto parsing dos ficheiros XML para representar um determinado caso. De seguida, é apresentado um exemplo de escrita em formato XML para cada uma das novas funcionalidades:

- **Iluminação:** Deverão ser indicadas cada uma das luzes.

```
1       <lights>
2             <light type="POINT" posX="0" posY="0" posZ="0"
                  diffR="1" diffG="1" diffB="1" />
3       </lights>
```

- **Texturas:** Deverá ser indicada a localização da textura bem como o respetivo ficheiro.

```
1       <models>
2         <model file="sphere.3d"
              texture="../../texture/mercury.jpg"/>
3       </models>
```

- **Materiais:** As primitivas podem estar associadas a materiais.

```
1       <models>
2        <model file="sphere.3d" texture="../../texture/earth.jpg"
            diffR="0.9" diffG="0.9" diffB="0.9" />
```

```
3       <model file="sphere.3d" texture="../../texture/stars.jpg"
            emiR="0.4" emiG="0.4" emiB="0.4" />
4      </models>
```

Assim, decidimos implementar também a *parseLights* e *parseMaterial* que são responsáveis por realizar o parsing da informação da iluminação e dos materiais, respetivamente. Para além disso alteramos a *parseModels* de modo a permitir a aplicação de texturas no projeto.

2.2.2 Point

Tal como indicado na secção do *Generator*, nesta aplicação também mantivemos a estrutura Point já desenvolvida numa fase inicial.

2.2.3 Transformation

Esta classe, tal como na fase anterior, é responsável por guardar todas as informações relativas a uma dada transformação. Para além disso, nesta são implementadas as curvas de Catmull-Rom.

2.2.4 Light

De modo a possibilitar a iluminação dos objetos caso esta lhes incida, desenvolvemos esta classe, que conterá a informação de uma determinada luz.

⇒ **Cor**

Tendo em conta que é possível associar diferentes cores a uma luz, optamos por utilizar as seguintes:

- **GL_AMBIENT:** intensidade de uma luz ambiente que uma fonte permite num dado cenário.
- **GL_DIFFUSE:** uma fonte pode projetar uma luz direcional. Quando esta atingue outro objeto espalha-se de forma uniforme pela superfície.
- **GL_SPECULAR:** tem impacto na cor do destaque especular de um dado objeto.

⇒ **Posição**

Para além das diferentes cores, incluímos dois tipos de luzes, uma pontual e uma direcional.

- **POINT:** As coordenadas indicadas representam a posição exata da luz, emitindo a luz para todas as direções a partir deste ponto. Para uma coordenada x, y e z, teremos *position* = { x,y,z,1 }
- **DIRECTIONAL:** Neste caso as coordenadas caracterizam a direção da luz. Para uma coordenada x, y e z, teremos *position* = { x,y,z,0 }

De modo a aplicar a cada luz as propriedades indicadas elaboramos a *draw*:

```
1  void Light::apply(GLenum number)
2  {
3      glLightfv(GL_LIGHT0+number, GL_POSITION, info);
4
5      for (const int atrb : attributes)
6      {
7          switch(atrb)
8          {
9              case DIFFUSE:
10                 glLightfv(GL_LIGHT0+number, GL_DIFFUSE, info+4);
11                 break;
12
13             case AMBIENT:
14                 glLightfv(GL_LIGHT0+number, GL_AMBIENT, info+8);
15                 break;
16
17             case SPECULAR:
18                 glLightfv(GL_LIGHT0+number, GL_SPECULAR, info+12);
19                 break;
20         }
21     }
22 }
```

2.2.5 Material

Nesta classe são introduzidos parâmetros necessários à representação de cores produzidas através de vetores com a informação respetiva. Assim temos os vetores *diffuse*,

ambient, *specular* e *emission*. Estes são representados através da primitiva *Transformation*. Desta forma, o conteúdo de cada será:

- **GL_DIFFUSE:** Aplicado o material e a luz branca, define-se a cor primária do modelo.
- **GL_EMISSION:** Emite a luz do próprio material após este ser aplicado.
- **GL_SPECULAR:** A superficie parece brilhante após a aplicação do material.
- **GL_AMBIENT:** Aplicado o material, o modelo reflete a cor em questão. Em todas as superfícies, a luz de incidência é igual.

De modo a aplicarmos os materiais implementamos a função *draw*.

```
void Material::draw() {
    if(diffuse[3] != -1)
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
    if(ambient[3] != -1)
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, emission);
}
```

2.2.6 Camera

Nesta fase, decidimos implementar uma nova câmera capaz de se mover livremente pelo cenário, com recurso ao teclado. É possível encontrar as seguintes variáveis:

- **positionX, positionY, positionZ:** Coordenadas x, y e z da posição da câmara.
- **lookX, lookY, lookZ:** Coordenadas x, y e z que definem a direção da câmara.
- **alpha:** Ângulo que define a direção da câmera no plano xOz;
- **teta:** Ângulo que define a direção da câmera no plano xOy;
- **mouseLeftIsPressed:** Permite verificar se o utilizador pressionou a tecla do rato.
- **mousePositionX, mousePositionY:** Coordenadas x e y do rato quando se pressiona a janela.

A direção da câmera é dada em função dos ângulos alpha e beta. A movimentação do rato permite a variação destes. Cada uma das direções da câmara, pode ser obtida através de um ponto de uma superfície esférica. De seguida, encontra-se a obtenção das coordenadas da direção da câmara:

```
1    lookX = float(cos(teta)*cos(alpha)) ;
2    lookY = float(sin(teta));
3    lookZ = float(cos(teta)*sin(alpha));
```

A posição da câmera pode ser alterada através da utilização das teclas F7, F8, F9 e F10. Através do scroll wheel do rato, é possível aumentar ou diminuir a altura da câmera. Por default, a câmera encontra-se na seguinte posição:

```
1    positionX = -100.0f;
2    positionY = 60.0f;
3    positionZ = 0.0f;
```

A câmera move-se segundo uma velocidade que é, por default, 1.7. Em baixo encontra-se a explicação para cada uma das teclas para mover a câmara:

- **F7:** Move a câmera para a frente. Para tal, basta somar o vetor direção ao vetor posição;

```
1        positionX += lookX * 1.7f;
2        positionY += lookY * 1.7f;
3        positionZ += lookZ * 1.7f;
```

- **F8:** Move a câmera para trás. Para tal, basta subtrair o vetor direção ao vetor posição;

```
1        positionX -= lookX * 1.7f;
2        positionY -= lookY * 1.7f;
3        positionZ -= lookZ * 1.7f;
```

- **F9:** Move a câmera para o lado esquerdo. Para tal, é necessário encontrar o vetor perpendicular aos vetores direção e ao vetor orientação da câmera para que o produto dos dois seja um vetor perpendicular. Este vetor permite a movimentação da câmera ao longo do mesmo. Por último, devemos subtrair esse vetor ao vetor posição;

```
        float up[3], dir[3];
        up[0] = up[2] = 0;
        up[1] = 1;
        dir[0] = lookX ;
        dir[1] = lookY ;
        dir[2] = lookZ;
        float res[3];

        cross(dir,up,res);

        positionX -=  res[0] * 1.7f;
        positionY -= res[1] * 1.7f;
        positionZ -= res[2] * 1.7f;
```

- **F10:** Semelhante à anterior no entanto move a câmera para o lado direito.

```
        float up[3], dir[3];
        up[0] = up[2] = 0;
        up[1] = 1;
        dir[0] = lookX ;
        dir[1] = lookY ;
        dir[2] = lookZ;
        float res[3];

        cross(dir,up,res);

        positionX +=  res[0] * 1.7f;
        positionY += res[1] * 1.7f;
        positionZ += res[2] * 1.7f;
```

⇒ **gluLookAt**

Recorremos à função gluLookAt para fornecer os valores relativos à posição, direção e orientação da câmera ao Glut.

```
    gluLookAt (
            camera->getXPosition(), camera->getYPosition(),
               camera->getZPosition(),
            camera->getOrX(), camera->getOrY(), camera->getOrZ(),
0.0f, 1.0f, 0.0f);
```

Os vetores PositionX, PositionY e PositionZ definem a posição da câmara. Os valores referentes à orientação da câmera, são definidos através da soma da posição da câmera com o vetor direção, deste modo garantimos que a câmera continua a olhar para o mesmo objeto. A orientação da câmera não é alterada.

```
float Camera::getXPosition() { return positionX; }
float Camera::getYPosition() { return positionY; }
float Camera::getZPosition() { return positionZ; }
float Camera::getOrX() { return positionX + lookX; }
float Camera::getOrY() { return positionY + lookY; }
float Camera::getOrZ() { return positionZ + lookZ; }
```

2.2.7 Shape

Para melhorar o desempenho do programa na fase anterior implementamos VBO's para as primitivas. Devido à alteração dos ficheiros .3d, que agora para além das primitivas contêm as normais e as coordenadas das texturas, foi necessário modificar a *prepareBuffer*:

```
    // construir array de float com vertices
    // construir array de float com normais
    glGenBuffers(3,buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(float) * numVertex[0] * 3,
                 vertexs,
                 GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(float) * numVertex[1] * 3,
                 normals,
                 GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, buffer[2]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(float) * numVertex[2],
                 &(texture[0]),
                 GL_STATIC_DRAW);
```

Desta forma, a *draw* também teve de sofrer alterações:

```
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);

    if(numVertex[1] > 0) {
        glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
        glNormalPointer(GL_FLOAT, 0, 0);
    }

    if(numVertex[2] > 0) {
        glBindBuffer(GL_ARRAY_BUFFER, buffer[2]);
        glTexCoordPointer(2, GL_FLOAT, 0, 0);
        glBindTexture(GL_TEXTURE_2D, text);
    }

    glDrawArrays(GL_TRIANGLES, 0, (numVertex[0]) * 3);
    glBindTexture(GL_TEXTURE_2D, 0);
```

Durante o processo de leitura do ficheiro de configuração é possível carregar uma textura, pelo que recorremos à *loadTexture* caso o modelo tenha alguma textura associada. Esta permite carregar em memória os dados.

2.2.8 Group

Tal como nas fases anteriores esta classe é responsável por guardar toda a informação de um grupo, nomeadamente as transformações geométricas, as primitivas e ainda os grupos filhos.

2.2.9 Scene

De modo a facilitar o desenho do cenário optámos por criar esta classe que contém as luzes bem como o grupo principal.

```
    vector<Light*> lights;
    Group *mainGroup;
```

Nesta definimos a *applyLights* que irá auxiliar na aplicação das luzes:

```
void Scene::applyLights()
{
```

```
3       GLenum number = 0;
4       for(Light *l : lights)
5           l->apply(number++);
6   }
```

Desta forma para aplicarmos as luzes, na *engine*, basta recorrer à *applyLights* após a aplicação das propriedades da câmera. O desenho do resto do cenário é realizado de forma semelhante à fase anterior.

3 Demonstração

De forma a correr e demostrar todas as aplicações referidas, iremos apresentar todos os passos para executar cada uma destas.

- **Generator**

```
$ cd generator
$ mkdir build && cd build
$ cmake ..
$ make
$ ./generator torus 0.5 3 20 20 torus.3d
$ ./generator sphere 3 20 20 sphere.3d
$ ./generator -patch teapot.patch 10 teapot.3d
```

Figura 12. Generator

- **Engine**

```
$ cd engine
$ mkdir build && cd build
$ cmake ..
$ make
$ ./engine SolarSystem.xml
```

Figura 13. Engine

3.1 Usabilidade

De forma a auxiliar o utilizador, foi criado um menu de ajuda que permite ao utilizador saber quais os comandos a utilizar de modo a interagir com o sistema. Assim, este sabe de que forma poderá manipular o Sistema Solar e visualizar o que pretende.

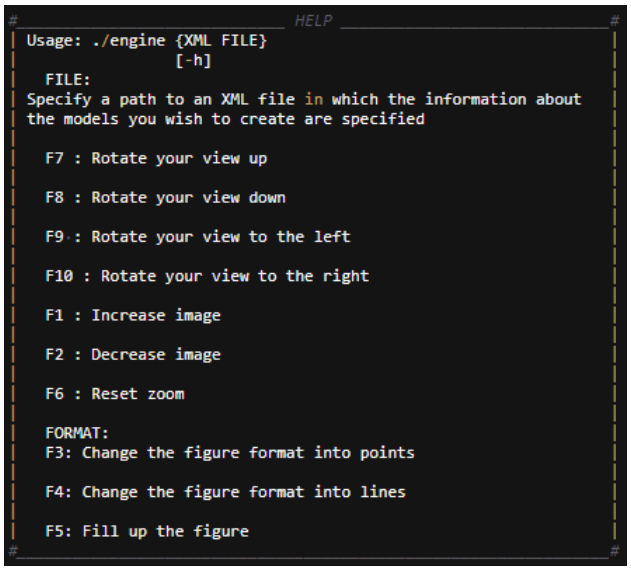


Figura 14. Menu de Ajuda

3.1.1 Menu do Sistema Solar

De modo a tornar o nosso sistema mais interativo, é possível o utilizador parar ou acionar os movimentos do sistema, bem como, sair do mesmo. Para aceder a este menu, basta pressionar o lado direito do rato. Os sub-menus do menu *System movements* foram criados de forma *responsive*. Neste menu será possível:

- Parar ou Retomar movimentos do sistema
- Fechar janela

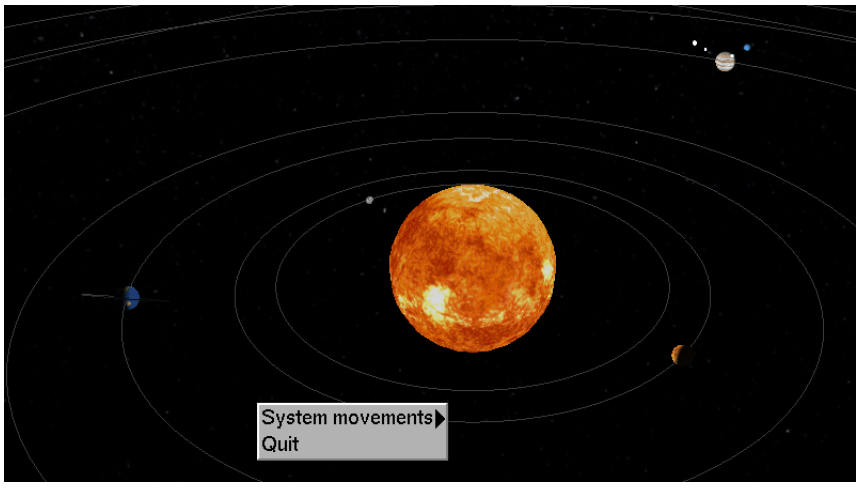


Figura 15. Menu do Sistema Solar

3.2 Sistema Solar

Em baixo é apresentado o exemplo do Sistema Solar que serviu de base para a criação do nosso. De seguida é ilustrado o Sistema Solar desenvolvido até agora. Este contém todos os planetas, os seus satélites naturais e ainda um cometa, que contêm órbitas definidas por curvas de Catmull Rom.

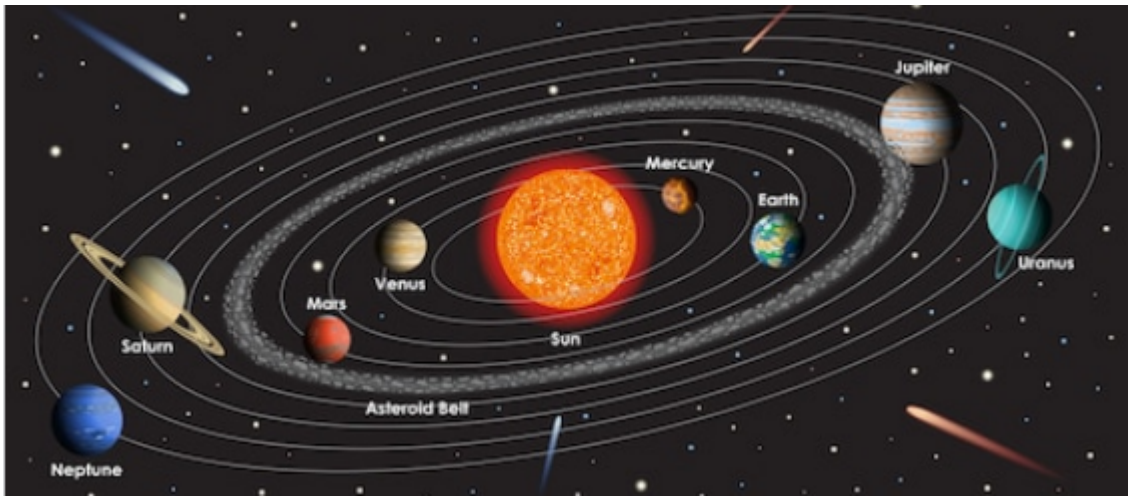


Figura 16. Sistema Solar

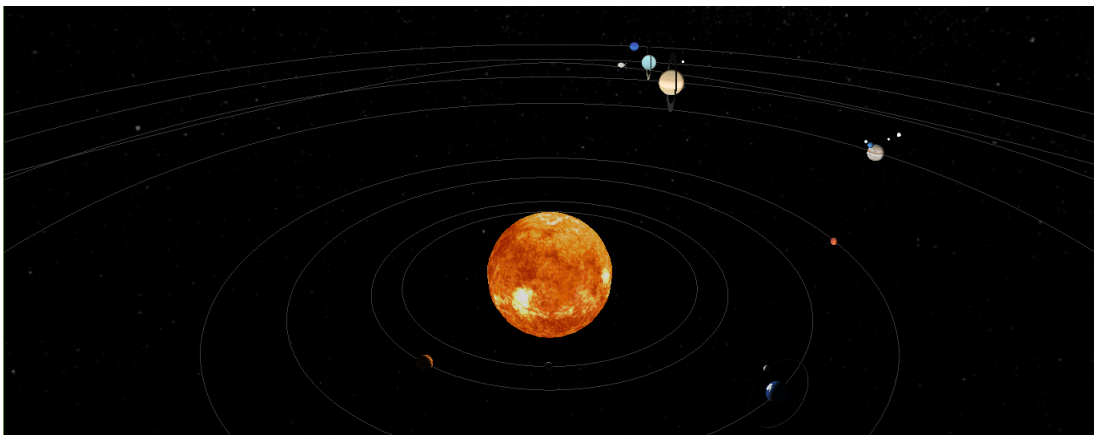


Figura 17. Sistema Solar

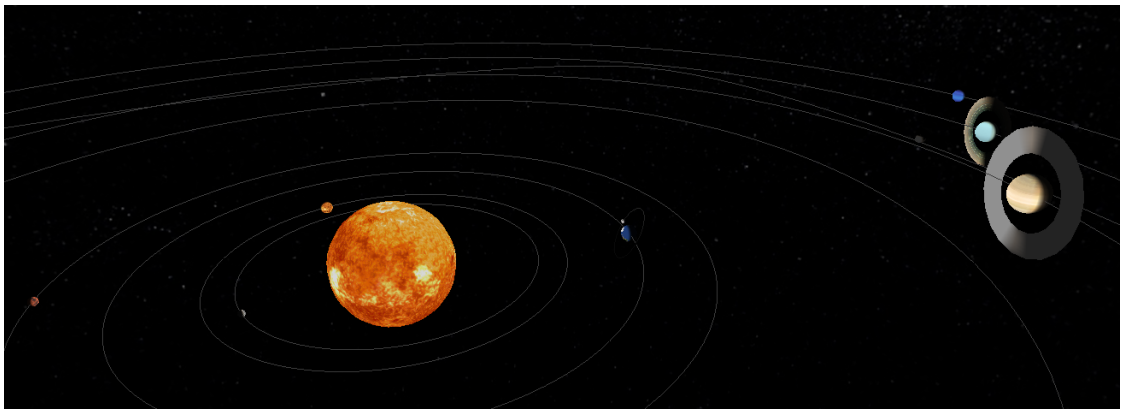


Figura 18. Sistema Solar

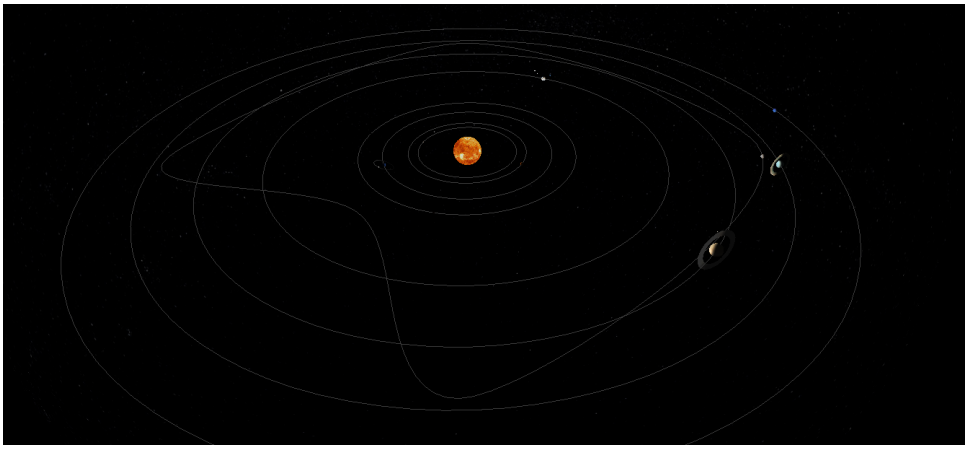


Figura 19. Sistema Solar

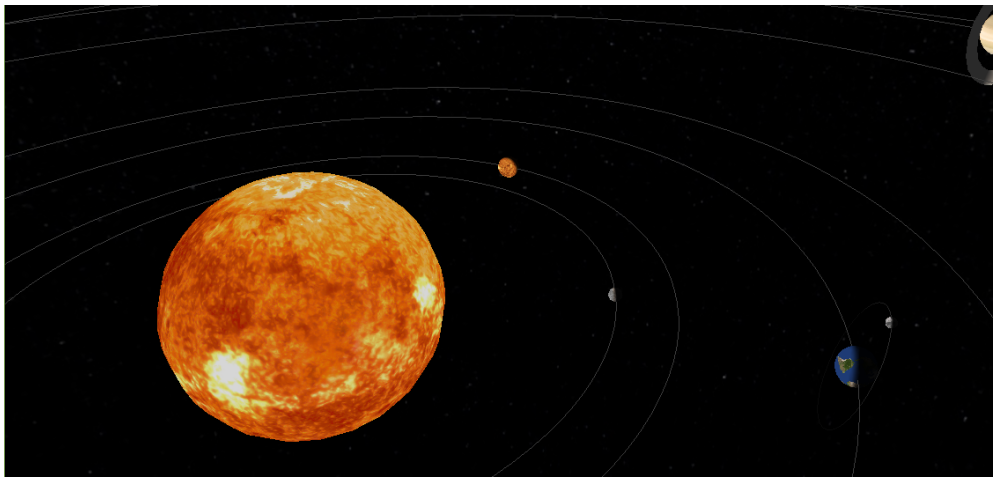


Figura 20. Sistema Solar

3.3 Primitivas

Figura 21. Torus

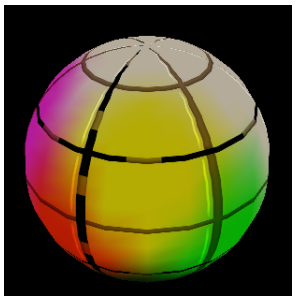


Figura 22. Esfera

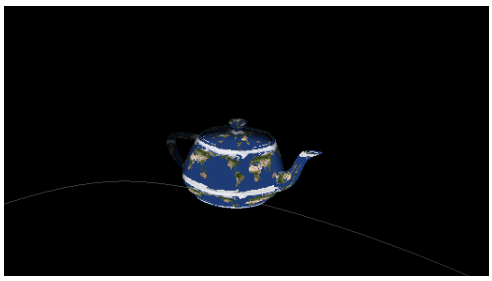


Figura 23. Teapot

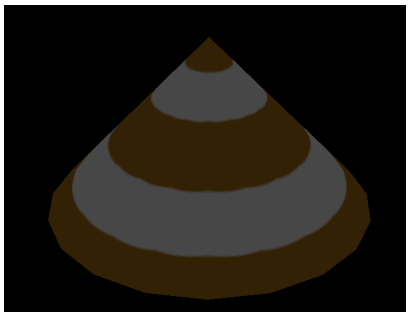


Figura 24. Cone

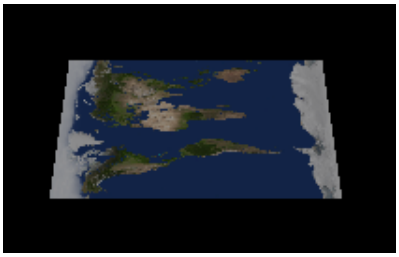


Figura 25. Plano

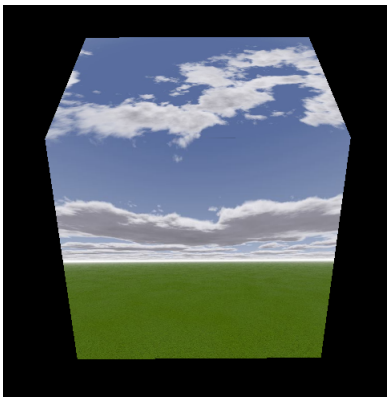


Figura 26. Box

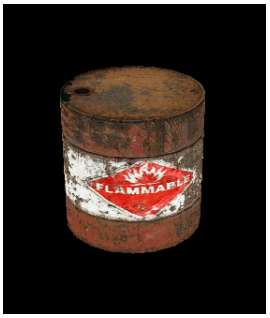


Figura 27. Cilindro

4 Conclusão

De uma perspetiva geral, consideramos que a realização desta fase foi relativamente bem sucedida, visto que pensamos cumprir com todos os requisitos estabelecidos inicialmente bem como alguns extras.

Apesar de este projeto ter sido iniciado com a criação de algumas primitivas, rapidamente nos apercebemos da proporção e complexidade que o projeto iria possuir quando na segunda fase nos foi pedido para aplicar transformações geométricas. Com a aplicação das curvas, patches de Bezier e VBOs na terceira fase, começamo-nos a aperceber da ligação entre as diferentes tarefas, permitindo-nos idealizar minimamente como seria o resultado final. Agora com utilização de normais e coordenadas de textura para a aplicação de iluminação de texturas conseguimos finalmente concretizar o objetivo do projeto. Ao longo do trabalho também fomos capazes de implementar funcionalidades extras. Este é o caso do menu de ajuda, de uma câmara capaz de se mover livremente, entre outras.

Em suma, após terminar a realização deste projeto, concluímos que as matérias lecionadas foram devidamente consumadas e aprofundadas com a concretização dos diferentes problemas que nos foram sendo propostos.