4ª Fase – Textura, Luz e View Frustum

Computação Gráfica – Universidade do Minho

2017/2018

Carlos Gonçalves – A77278

Jorge Oliveira – A78660

José Ferreira – A78452

Ricardo Peixoto – A78587

Índice

1. Introdução

O presente relatório tem como objetivo explicar a conceção da 4º fase do projeto que consistia na construção gradual de um sistema solar. Como seria de esperar, nesta fase será utilizado tudo o que foi desenvolvido nas fases anteriores e serão acrescentadas novas funcionalidades que obrigarão a algumas alterações no trabalho já realizado. O principal objetivo seria então acrescentar ao nosso sistema solar, em particular aos elementos que o constituem, iluminação e texturas. Para isso foram necessárias varias alterações em vários componentes, uma vez que o gerador precisava de gerar estas novas coordenadas, o parser precisava de ter a capacidade de as reconhecer, o criador de XML precisava de as referir (ficheiro de texturas utilizado e tipo de iluminação) e o motor precisava de as aplicar. Se tudo isto fosse implementado com sucesso o próximo objetivo seria a criação de uma camara em terceira pessoa e a integração de um mecanismo de *view frustum culling*, que iria aumentar significativamente a performance do nosso sistema solar. Além disso, apesar de não ser proposto, colocamos como meta criar um mecanismo anti colisões que daria um grau de realismo mais elevado.

2. Câmera

2.1. Progresso anterior

De maneira a conseguir uma aplicação o mais modular possível, decidimos, na segunda fase, que a manipulação da câmera seria feita numa classe à parte do "motor". Assim sendo criamos uma classe "Camera" responsável por todos os movimentos da câmera. Para que esta classe conseguisse realizar estas manipulações precisava de uma variável para guardar a posição espacial em que se encontrava ("posicao"), a distância à origem ("raio"), um ângulo para movimentação horizontal ("alpha") e outro ângulo para movimentação vertical ("beta"), um valor para definir o gradiente de variação do angulo ("angulo") e variáveis para controlarem o movimento do rato ("mouseX" e "mouseY") e se algum dos seus botões estava a ser pressionado ("mouseTracking"). Sendo esta câmera sempre focada no ponto (0,0,0) não foi necessário guardar o ponto no qual esta estava focada.

Para a terceira fase do projeto decidimos implementar uma câmera que não possuía o foco fixo num determinado ponto, ou seja, que permitia rodar a câmera em qualquer direção sem esta se deslocar. Para tal precisamos de definir uma nova variável ("foco") para guardar o ponto de foco da câmera. Dado que a esta se podia mover livremente em qualquer direção decidimos também criar uma variável ("velocidade") para representar o gradiente de variação da posição da câmera. Visto que a distância à origem já não seria importante removemos a variável "raio". As variáveis “alpha” e “beta” passaram a ser usadas para definir o foco dada a sua posição.

Esta mudança para uma câmera de estilo FPS levou-nos também a adaptar os métodos anteriormente definidos para a câmera de foco fixo. Como era possível mover tanto a câmera como apenas rodá-la (mover o foco da mesma) tivemos de abandonar o método "atualizaPosicao" que possuíamos anteriormente e desenvolver dois novos métodos, o "moverCamera" e o "rodarCamera". O método "moverCamera" era responsável por alterar a posição da câmera tendo em conta o ponto de foco da mesma. Já o método "rodarCamera" era responsável por alterar o foco da câmera tendo em conta os ângulos "alpha" e "beta".

```
void Camera::moverCamera(int a, int b, int c) {

    float v = velocidade;
    float dx = foco->getX() - posicao->getX();
    float dy = foco->getY() - posicao->getY();
    float dz = foco->getZ() - posicao->getZ();

    if (b == 0) {
        posicao->addPonto(a*v*dz, 0, c*v*dx);
        foco->addPonto(a*v*dz, 0, c*v*dx);
    }
    else{
        posicao->addPonto(a*v*dx, b*v*dy, c*v*dz);
        foco->addPonto(a*v*dx, b*v*dy, c*v*dz);
    }
}
```

```
void Camera::rodarCamera(float a, float b){

    float focoX = posicao->getX()
        + 10 * sin(-a * 3.14 / 180.0) * cos(-b * 3.14 / 180.0);
    float focoY = posicao->getY()
         + 10 * sin(-b * 3.14 / 180.0);
    float focoZ = posicao->getZ()
        + 10 * cos(-a * 3.14 / 180.0) * cos(-b * 3.14 / 180.0);
    foco->setX(focoX);
    foco->setY(focoY);
    foco->setZ(focoZ);
}
```

2.2. Quarta Fase

Na quarta fase, decidimos implementar uma câmera em terceira pessoa cujo foco seria um objeto (o foguetão) e cuja câmera rodava em torno do mesmo, sendo possível aproximá-la ou afastá-la. Para tal, em adição às variáveis descritas acima adicionamos uma variável “raio” para guardar a distância entre a câmera e o foco. A função das variáveis “alpha” e “beta” é agora ajudar a definir a posição da câmera dado o seu foco.

Decidimos também manter a câmera em primeira pessoa permitindo a troca entre estes dois tipos de câmera a qualquer momento. Contudo haviam grandes diferenças entre estas câmeras. Apesar do modo de efetuar rotação ser igual nas duas câmeras, isto é, alterar as variáveis “alpha” e “beta”, na câmera em primeira pessoa esta variáveis são usadas para definir para onde se olha enquanto que na câmera em terceira pessoa são usadas para definir de onde se olha. Na câmera em primeira pessoa a variável “foco” é sempre calculada em função das variáveis “posicao”, “alpha” e “beta”. Por outro lado, na câmera em terceira a variável “posicao” é sempre calculada em função das variáveis “foco”, “alpha” e “beta” (ex: o foguetão move-se e a câmera segue-o). Por estas razões decidimos dividir a classe câmera em duas classes distintas, “Camera1stP” e “Camera3rdP”.

Vetores

Tanto na câmera em primeira pessoa desenvolvida na fase anterior como na câmera em terceira pessoa desenvolvida nesta fase, decidimos utilizar vetores para armazenar a direção para a qual a câmera estava virada. Sempre que eram efetuadas alterações na posição ou no foco da câmera, este vetor era atualizado usando o método “alteraVetoresCamera” que se pode ver na figura abaixo.

```
void Camera1stP::alteraVetoresCamera(){

    float dx = foco->getX() - posicao->getX();
    float dy = foco->getY() - posicao->getY();
    float dz = foco->getZ() - posicao->getZ();
    Vec3 d(dx, dy, dz);

    Vec3 upAux(0,1,0);
    camVecZ = d
    camVecZ.normalize();
    camVecX = upAux*camVecZ
    camVecX.normalize();
    camVecY = camVecZ*camVecX
    camVecY.normalize();
}
```

```
void Camera3rdP::alteraVetoresCamera(){

    float dx = foco->getX() - posicao->getX();
    float dy = foco->getY() - posicao->getY();
    float dz = foco->getZ() - posicao->getZ();
    Vec3 d(dx, dy, dz);

    Vec3 upAux(0,1,0);
    camVecZ = d
    camVecZ.normalize();
    camVecX = upAux*camVecZ
    camVecX.normalize();
    camVecY = camVecZ*camVecX
    camVecY.normalize();
}
```

View Frustum

Usamos também um mecanismo em ambas as câmeras que permite desenhar apenas os grupos de objetos que estão dentro do campo de visão da câmera (view frustum) que é abordado com mais detalhe na secção View Frustum desde relatório.

```
void Camera1stP::setFrustumDef(){
    Vec3 p(posicao->getX(),posicao->getY(),posicao->getZ());
    Vec3 l(foco->getX(),foco->getY(),foco->getZ());
    Vec3 u(0,1,0);
    frustum->setCamDef(p,l,u);
}

void Camera1stP::setFrustumInternals(float a, float b, float c, float d){
    frustum->setCamInternals(a,b,c,d);
}
```

Colisões

Por último implementamos um mecanismo de deteção de colisões em cada uma das câmeras que impede que a câmera se aproxime mais do que uma certa distância dos objetos para esta não colidir com os mesmos. Este mecanismo também será abordado mais detalhadamente na secção View Frustum na subsecção Colisões. Após detetar que a câmera trespassou o limite de proximidade permitido para o dado objeto é chamado o método “voltaAtras” que anula esse movimento da câmera fazendo o movimento inverso. Para tal, sempre que é efetuado algum movimento da câmera registamos o movimento inverso nas variáveis “mX”, “mY” e “mZ”. Os métodos “alteraPosicaoCamera” e “alteraFocoCamera” serão abordados nas subsecções Camera1stP e Camera3rdP desta secção, respetivamente.

```
void Camera1stP::voltaAtras(){

    alteraPosicaoCamera(mX,mY,mZ);
}
```

```
void Camera3rdP::voltaAtras(){

    alteraFocoCamera(mX,mY,mZ);
}
```

Câmera em primeira pessoa - Camera1stP

Para a câmera em primeira pessoa adotamos os dois métodos definidos anteriormente, "moverCamera" e "rodarCamera", e apenas fazemos o movimento da câmera através dos vetores que indicam a direção do movimento (no método “alteraPosicaoCamera”). Os argumentos recebidos no segundo método, x, y e z, são as coordenadas que indicam em que direção a câmera se deve mover (para a frente, para trás, para o lado esquerdo, para baixo, etc).

```
void Camera1stP::alteraFocoCamera(){

    float focoX = posicao->getX()
        + sin(-alpha * 3.14 / 180.0) * cos(-beta * 3.14 / 180.0);
    float focoY = posicao->getY()
        + sin(-beta * 3.14 / 180.0);
    float focoZ = posicao->getZ()
        + cos(-alpha * 3.14 / 180.0) * cos(-beta * 3.14 / 180.0);
    foco->set(focoX, focoY, focoZ);

        alteraVetoresCamera();
}
```

```
void Camera1stP::alteraPosicaoCamera(float x, float y, float z) {

    Vec3 move(posicao->getX() , posicao->getY() , posicao->getZ());
    move = move + (camVecX*x + camVecY*y+ camVecZ*z)*velocidade;

    posicao->set(move.x, move.y, move.z);

    alteraFocoCamera();
}
```

Câmera em terceira pessoa - Camera3rdP

A câmera em terceira pessoa é muito similar à câmera em primeira pessoa. A diferença está que em vez de serem a posição e os ângulos a definir o foco, são o foco e os ângulos a definirem a posição como se pode observar nas seguintes figuras.

```
void Camera3rdP::alteraPosicaoCamera(){

    float posicaoX = foco->getX()
        + raio * sin(-alpha * 3.14 / 180.0) * cos(-beta * 3.14 / 180.0);
    float posicaoY = foco->getY()
        + raio * sin(+beta * 3.14 / 180.0);
    float posicaoZ = foco->getZ()
        + raio * cos(-alpha * 3.14 / 180.0) * cos(-beta * 3.14 / 180.0);
    posicao->set(posicaoX, posicaoY, posicaoZ);

    alteraVetoresCamera();
}
```

```
void Camera3rdP::alteraFocoCamera(float x, float y, float z) {

    Vec3 move(foco->getX() , foco->getY() , foco->getZ());
    move = move + (camVecX*x + camVecY*y+ camVecZ*z)*velocidade;

    foco->set(move.x, move.y, move.z);

    alteraPosicaoCamera();
}
```

Teclado

Para processar as teclas pressionadas no teclado decidimos manter os métodos "normalKeys" e "specialKeys", com algumas alterações. O primeiro método é responsável pela alteração da posição da câmera ou do foco da câmera nas câmeras Camera1stP e Camera3rd, respetivamente (teclas 'w', 'a', 's', 'd', 'c', 'b') e pela alteração do gradiente de variação da posição, "velocidade" (teclas '+', '-'). O segundo método é responsável pela alteração dos ângulos e, consequentemente, do foco da câmera (Camera1stP) ou da posição da câmera (Camera3rdP) (teclas '↑', '←', '→', '↓') e pela alteração do gradiente de variação do ângulo, "angulo" (teclas 'Pg Up', 'Pg Dn'). Apresentamos alguns exemplos de seguida.

Camera1stP :: normalKeys

```
case '-':
    velocidade *= 0.67;
    if(velocidade < 0.005)
        velocidade = 0.01;
    break;

case 'w':
    res = true;

    alteraPosicaoCamera(0,0,+1);
    mX = 0;
    mY = 0;
    mZ = -1;
    break;
```

Camera3rdP :: normalKeys

```
case 'a':
    res = true;
    alteraFocoCamera(+1,0,0);
    mX = -1;
    mY = 0;
    mZ = 0;
    break;

case 'c':
    res = true;
    alteraFocoCamera(0,+1,0);
    mX = 0;
    mY = -1;
    mZ = 0;
    break;
```

Camera1stP :: specialKeys

```
case GLUT_KEY_LEFT:
    alpha -= angulo;
    alteraFocoCamera();
    break;

case GLUT_KEY_UP:
    beta -= angulo;
    alteraFocoCamera();
    break;
```

Camera3rdP :: specialKeys

```
case GLUT_KEY_DOWN:
    beta += angulo;
    alteraPosicaoCamera();
    break;

case GLUT_KEY_PAGE_DOWN:
    angulo *= 0.67;
    break;
```

Rato

Para monitorizar o rato recorremos aos métodos "mouseButtons" e "mouseMotion". O botão direito do rato é usado para rodar a câmera em torno dela própria mantendo a sua posição fixa enquanto que o botão esquerdo do rato é usado mover a posição câmera na direção do ponto de foco (para trás e para a frente) mantendo o ponto de foco fixo.

MouseButtons

O método "mouseButtons" é chamado quando se deteta que algum dos botões do rato é pressionado ou . Como cada botão do rato vai realizar uma ação diferente, registamos qual o botão clicado através da variável "mouseTracking". Para calcular o deslocamento do rato no ecrã também é necessário registar as posições horizontal e vertical do rato no ecrã no momento em que se clica num dos botões. Esses valores são registados nas variáveis "mouseX" e "mouseY". Não foi necessário fazer quaisquer alterações a este método da fase anterior para esta.

```
void Camera1stP::mouseButtons(int button, int state, int xx, int yy) {

    if (state == GLUT_DOWN)  {
        mouseX = xx;
        mouseY = yy;
        if (button == GLUT_LEFT_BUTTON)
            mouseTracking = 1;
        else if (button == GLUT_RIGHT_BUTTON)
            mouseTracking = 2;
        else
            mouseTracking = 0;
    }
    else if (state == GLUT_UP)
        mouseTracking = 0;
}
```

MouseMotion

Por sua vez, o método "mouseMotion" calcula constantemente os deslocamentos horizontal e vertical efetuados pelo rato no ecrã e atua tendo em conta o botão do rato que está a ser pressionado. Caso seja o botão esquerdo (mouseTracking = 1), as variáveis "alpha" e "beta" são alteradas tendo em conta os deslocamentos horizontal e vertical, respetivamente, e é calculado o novo foco ou a nova posição da câmera nas câmeras Camera1stP e Camera3rdP, respetivamente. Caso seja o botão direito (mouseTracking = 2), é calculada a nova posição da câmera na Camera1stP ou a distância entre a posição e o foco (raio) na Camera3rdP, através do deslocamento vertical do rato. Caso sejam efetuados movimentos, o método regista os movimentos inversos nas variáveis mX, mY e mZ, como referido anteriormente, para evitar colisões da câmera com outros objetos. Podemos observar como o rato controla de forma diferente cada tipo de câmera nas seguintes figuras, onde “deltaX” e “deltaY” são as variações das coordenadas do rato no ecrã.

```
if (mouseTracking == 1) {

    alpha += deltaX*angulo;
    beta += deltaY*angulo;

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

    alteraFocoCamera();
}

else if (mouseTracking == 2) {
    mX = 0; mY = 0; mZ = -deltaY;
    alteraPosicaoCamera(0,0,deltaY);
}
```

```
if (mouseTracking == 1) {

    alpha += deltaX*angulo;
    beta += deltaY*angulo;

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

    alteraPosicaoCamera();
}

else if (mouseTracking == 2) {

    raio+=deltaY;
    if (raio < 2) raio = 2;
    mX = 0; mY = 0; mZ = -raio;
    alteraPosicaoCamera();
}
```

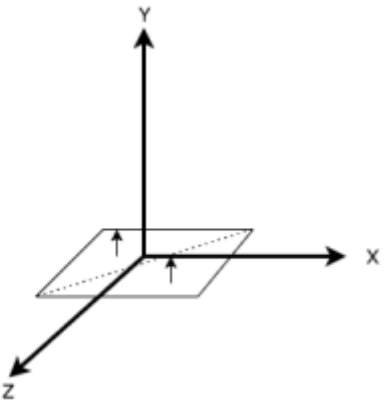
3. Aplicação

3.1. Normais e Texturas

Nesta fase e por forma a conseguirmos uma ilustração mais real do Sistema Solar decidimos aplicar normais, importantes para a iluminação, e texturas.

Plano

As normais do plano são bastante simples, visto que apenas têm de apontar para “cima” consoante o plano no qual são desenhados, ou seja, são todas iguais e como são desenhados no plano xOz o eixo dos y corresponde ao eixo das “alturas”, logo as normais correspondem ao vetor (0,1,0).

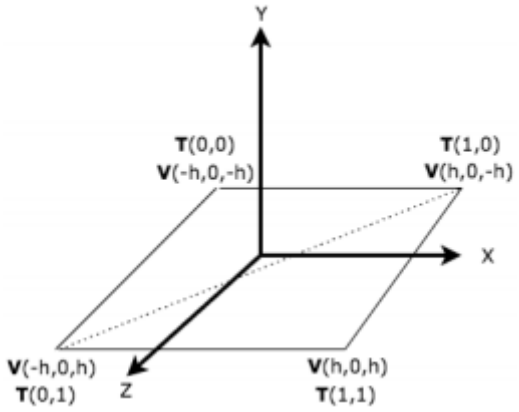


Relativamente aos pontos de textura do plano, estes também são bastante simples, visto que apenas temos de fazer corresponder cada vértice do plano ao vértice do espaço das texturas.

A correspondência é a seguinte:

- (-l/2,0,-l/2) -> (-1,-1)
- (-l/2,0,l/2) -> (-1,1)
- (l/2,0,-l/2) -> (1,-1)

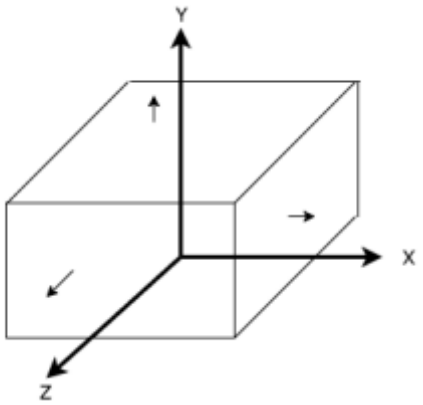
- (l/2,0,l/2) -> (1,1)



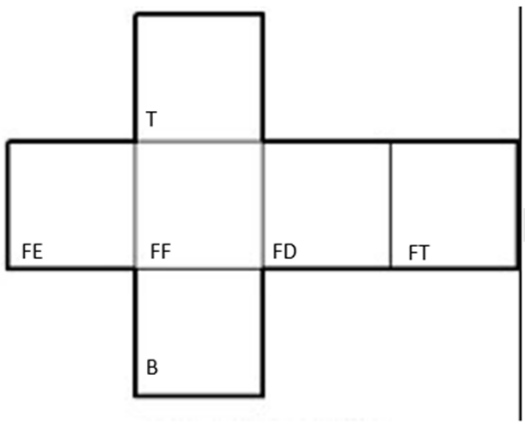
Caixa

Relativamente às normais da caixa, estas seguem o mesmo raciocínio que o plano, no entanto nesta figura temos de considerar que cada face corresponde a um plano diferente, pelo que as normais serão as mesmas na mesma face, mas diferentes para as diferentes faces. Aqui a ideia é que a normal aponte no sentido de fora da caixa, pelo que serão as seguintes:

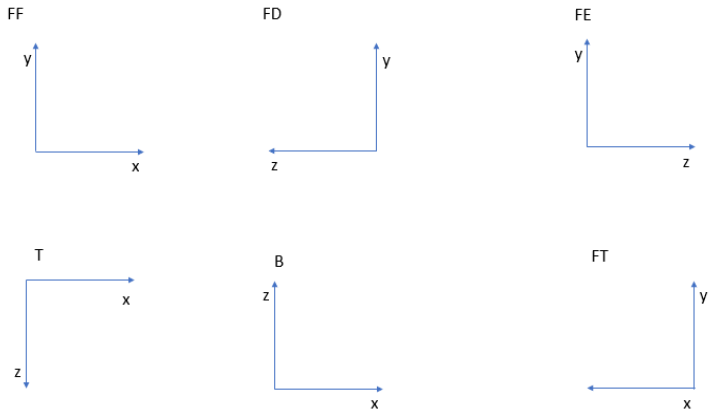
- Face Frontal - vetor (0,0,1)
- Face Traseira - vetor (0,0,-1)
- Face Direita - vetor (1,0,0)
- Face Esquerda - vetor (-1,0,0)
- Topo - vetor (0,0,1)
- Base - vetor (0,0,-1)



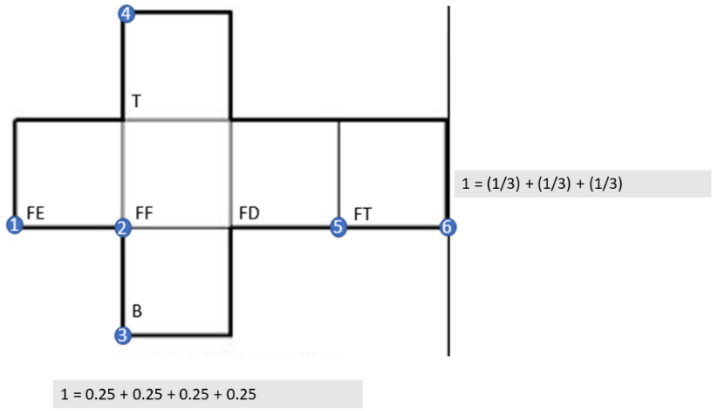
Para calcular os pontos de textura desta figura tivemos de analisar uma imagem 2D (correspondente ao espaço de texturas) com a construção da mesma:



Com a análise da planificação do cubo chegamos aos seguintes resultados para o desenho de cada face e os eixos respetivos:



Como podemos ver nas imagens acima descritas, o eixo x das texturas corresponde ao valor apresentado e o eixo y também, isto apenas teve de ser realizado olhando para os eixos das faces e somando os respetivos eixos.



Para iniciar o processo de conversão para coordenadas de textura tivemos de analisar os pontos iniciais para cada face, apresentamos esses pontos em cima e as suas coordenadas de seguida:

1. (0 , 1/3)

2. (0.25 , 1/3)

3. (0.25, 0)

4. (0.25, 1)

5. (0.75, 1/3)

6. (1 , 1/3)

Após esta análise tivemos de pensar como seriam feitas as iterações, ou seja, qual seria o valor da iteração para cada face, chagamos aos seguintes resultados:

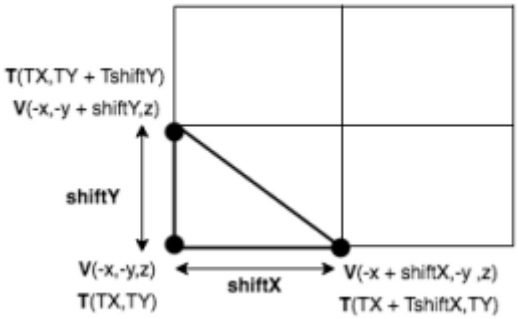
• deltaXText = 0.25/divisions -> para todas as faces envolventes ao X, visto que este apenas varia no sentido do eixo dos x da coordenadas de textura

• deltaYText = (1/3)/ divisions -> para todas as faces envolventes ao Y, visto que este apenas varia no sentido do eixo dos y da coordenadas de textura

• deltaZText1 = 0.25/ divisions -> para todas as faces envolventes ao Z que variam no eixo dos x das coordenadas de textura

• deltaZText2 = (1/3)/divisions -> para todas as faces envolventes ao Z que variam no eixo dos y das coordenadas de textura

De seguida apresentamos um exemplo para a face da frente, as outras são análogas (variando apenas no sentido positivo ou negativo como apresentado anteriormente):



Esfera.

Para calcular a normal da esfera apenas temos de verificar que esta tem de apontar no sentido oposto ao sentido do centro da mesma, logo o vetor correspondente à normal da mesma será:

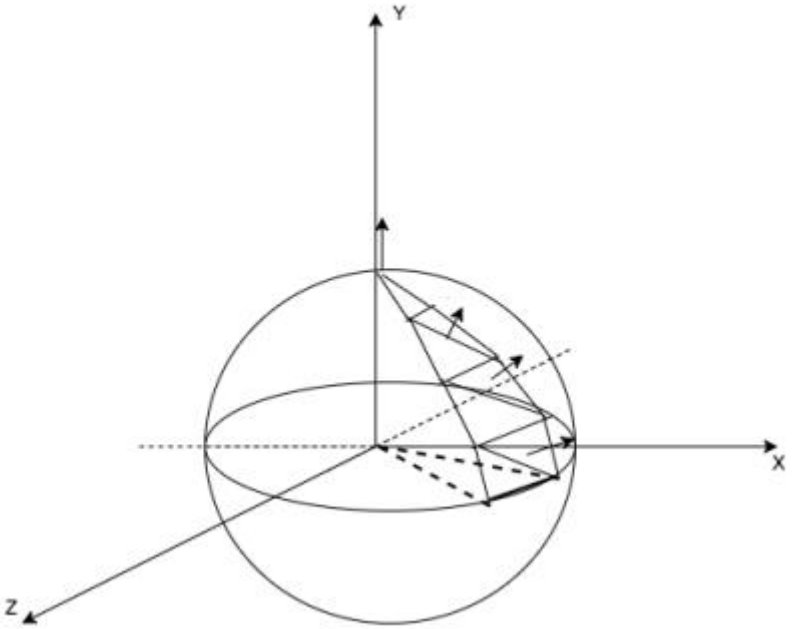
(sin(beta * j) * sin(alfa*i), cos(beta * j) , sin(beta * j) * cos(alfa*i))

Na iteração (i,j) e com:

• alfa = 2*pi/fatias

• beta = pi / camadas.

Ou seja, é muito parecido às coordenadas do ponto, apenas terá de ser normalizado (divido pelo raio).



Torus.

Para obter os vetores normais do torus foi preciso observar o processo de construção do mesmo uma vez que nos dá as orientações dos vetores normais de cada vértice no momento do desenho deste. Para obter os vetores das normais utilizamos então a seguinte formula:

Normal(cos(A)*cos(L), sin(A)*cos(L), sin(L))

onde,

A – Desvio do anel

L – Desvio de cada lado que forma um anel

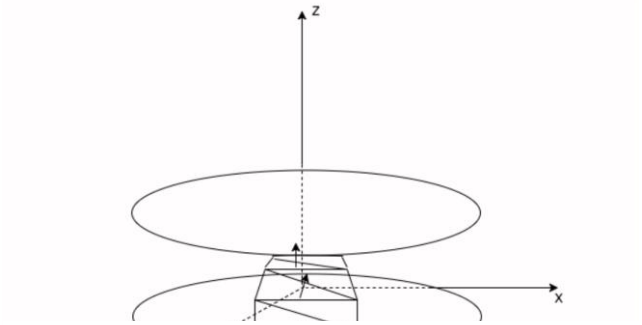


Figura 1 Representação normais torus

Figura 2 Representação normais torus

Para as coordenadas de texturas utilizamos um mapeamento simples. Dividimos a imagem da nossa textura em varias tiras correspondentes ao numero de anéis do torus. Posteriormente dividimos cada tira pelo numero de camadas do torus e atribuímos cada retângulo resultante à camada correspondente.

A formula utilizada foi a seguinte:

Textura(i/slice , j/sides)

onde,

i – Corresponde ao número do anel

j – Corresponde ao número do lado

slices – Número total de anéis

sides – Número total de lados

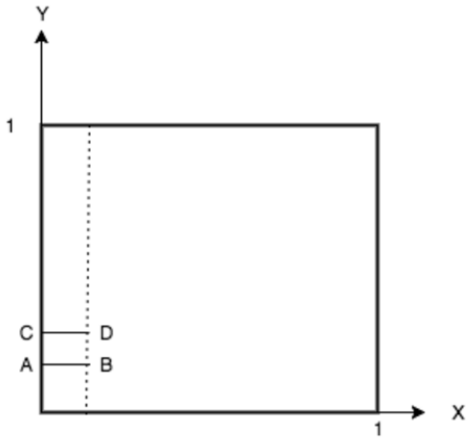


Figura 3 Explicação das texturas do torus

Cone.

Assim como no cilindro dividimos o cone em 2 partes para o calculo das normais.

Base: vetor(0,0,-1) (assim como a base do cilindro)

Corpo: vetor (sin(alfa),L/camada, cos(alfa))

onde,

alfa - Amplitude atual do vértice

L- Comprimento do corpo do cone

Camada - Camada a ser desenhada

Para mapearmos as texturas do cone utilizamos o mesmo método que foi utilizado e explicado anteriormente, na justificação do torus.

Teapot.

Para calcular as normais do teapot em cada ponto foi necessário primeiramente calcular as tangentes de B(u,v) em ordem a u e a v, onde u e v indicam um ponto na superfície de Bézier. Depois foi necessário fazer a normalização dos pontos resultantes e o *cross product* dos pontos já normalizados.

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

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

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

Deste modo desenvolvemos o método “calcularBezierNormal”. Neste método fazemos as multiplicações da matriz derivada de U por cada uma das colunas da matriz MPM^{-T} (os pontos de controlo da superfície sendo que cada superfície tem 16 pontos de controlo) resultando num vetor de Pontos com 4 posições que posteriormente multiplicámos pela matriz V originando um Ponto P1. De seguida multiplicamos a matriz U por cada uma das colunas da matriz MPM^{-T} resultando também num vetor de Pontos com 4 posições e multiplicamos pela matriz de V derivada em ordem a v originando um segundo Ponto P2. Após isso, normalizamos os dois pontos resultantes e efetuamos o *cross product* entre eles. Estas operações são realizadas para todos os pontos da superfície de Bézier usando as variáveis u e v no método “teapot” da classe “gerador”, que variam entre “0” e “divs-1” sendo “divs” o número de divisões nas quais foi dividida a superfície do teapot. Cada superfície do teapot irá executar todo o processo descrito anteriormente.

Para calcular as coordenadas dos Pontos para as texturas apenas tivemos de dividir as variáveis i, i+1, j, j+1 por “divs”.

Tanto para os pontos das normais como para os pontos das texturas foi necessário ter o cuidado de os inserir por ordem correta nos respetivos vetores conforme os pontos das superfícies haviam sido inseridos.

```
// Para cada superfície
for(int ind=0; ind < (int) indices.size(); ind++) {

    // Gerar pontos de controlo para o patch "ind"
    {...}

    for (int i = 0; i < divs; ++i) {

        float u1 = (float) i / (divs);
        float u2 = (float) (i + 1) / (divs);

        for (int j = 0; j < divs; ++j) {

            float v1 = (float) j / (divs);
            float v2 = (float) (j+1) / (divs);

            // calcular as normais da superfície
            Ponto* n1 = calcularBezierNormal(u1, v1, pontos_controlo);
            {...}

            // calcular as texturas da superfície
            Ponto* t1 = new Ponto(u1, v1, 0);
            {...}

            // inserir as normais no vetor
            (*normais).push_back(n1);
            (*normais).push_back(n3);
            (*normais).push_back(n4);
            (*normais).push_back(n1);
            (*normais).push_back(n4);
            (*normais).push_back(n2);

            // inserir as texturas no vetor
            (*texturas).push_back(t1);
            {...}
        }
    }
}
```

3.2. Estruturas

Nesta última fase decidimos acrescentar algumas estruturas ao nosso código, bem como atualizar algumas das restantes por forma a corresponder corretamente aos objetivos definidos. De seguida vamos apresentar as alterações às estruturas existentes e o porquê de termos realizado essas alterações, no final apresentamos o diagrama de classes do nosso projeto.

Luz

Por forma a introduzir iluminação no nosso projeto do sistema solar optamos por criar uma classe que corresponde a uma fonte de luz: a classe Luz. No entanto, e dado que existem

diferentes tipos de fontes de luz e com caraterísticas diferentes decidimos que esta classe Luz seria “abstrata” e que seria realizada em cada uma das suas subclasses: LuzPontual, LuzDirecional e LuzFoco.

No que diz respeito à classe Luz decidimos colocar algo que fosse comum a todas as luzes, ou seja a posição (tanto seja um ponto como um vetor no caso da luz direcional), as diferentes componentes ambiente, especular e difusa da luz que vai incidir nos objetos. Decidimos também ter um número entre 0 e 7, que corresponde às 8 luzes permitidas pelo OpenGL.

No que diz respeito às componentes que correspondem “à cor da luz” decidimos inicializar as mesmas com os valores por defeito para as luzes 1 a 7, sendo que caso seja a luz 0 estes valores são atualizados. Para além disso apenas consideramos estes valores para estas componentes, visto não deixarmos que o utilizador passe o valor dos mesmos.

```
class Luz {


private:
 float posicao[4];
 float ambiente[4] = {0,0,0, 1};;
 float difusa[4] = {0,0,0, 1};;
 float especular[4] = {0,0,0, 1};;
 int numero;


public:
 Luz();


 Luz(int, float[4]);


 GLfloat* getPosicao();
 GLfloat* getAmbiente();
 GLfloat* getDifusa();
 GLfloat* getEspecular();


 int getNumero();


 virtual void desenhaLuz() = 0;


};
```

```
Luz::Luz(int nr, float pos[4]) {

    if(nr == 0){
        for(int i = 0; i < 3; i++){
            difusa[i] = 1;
            especular[i] = 1;
        }
    }

    numero = (nr > 7 ? 7 : nr);

    for(int i = 0 ; i < 4; i++){
        posicao[i] = pos[i];
    }
}
```

Luz Pontual

No que diz respeito a uma luz pontual, como já foi referido anteriormente, o único parâmetro necessário a ser passado para além dos utilizados na classe Luz é o valor da atenuação. Aqui devemos referir que apenas permitimos atenuações lineares, visto que as constantes deixamos com o seu valor por defeito 1 e as quadráticas optamos também por não alterar.

```
private:

 float atenuacao;

public:
 LuzPontual();

 LuzPontual(float[4],int,float);

 void desenhaLuz();

};
```

A função desenhaLuz() apenas tem de desenhar cada componente da luz que definimos para esta classe.

```
void LuzPontual::desenhaLuz(){

 int luzA = this->getNumero();

 if(luzA>7){
       luzA = 7;
 }

 // posicao
 glLightfv(GL_LIGHT0 +luzA, GL_POSITION, this->getPosicao());

 // cores
 glLightfv(GL_LIGHT0 +luzA, GL_AMBIENT, this->getAmbiente());
 glLightfv(GL_LIGHT0 +luzA, GL_DIFFUSE, this->getDifusa());
 glLightfv(GL_LIGHT0 +luzA, GL_DIFFUSE, this->getEspecular());

 //atenuacao
 glLightf(GL_LIGHT0 +luzA,GL_LINEAR_ATTENUATION,atenuacao);

}
```

Luz Direcional

A classe **LuzDirecional** ainda é mais simples do que a anterior visto que não tem qualquer atributo para além dos identificados na sua superclasse.

```
private:

public:
 LuzDirecional();

 LuzDirecional(float[4],int);

 void desenhaLuz();
```

```
};
```

Aqui só devemos referir que ao contrário dos outros casos, o valor das posições tem um valor 0 na última posição indicando que é um vetor.

O processo de desenho da luz é semelhante, sendo que apenas não tem a instrução da atenuação.

Luz Foco

A classe **LuzFoco** corresponde a uma luz do tipo SPOT pelo que necessita de mais atributos dos presentes na sua superclasse. Como referimos anteriormente, necessita de um ângulo correspondente ao ângulo de propagação da mesma, de uma direção da propagação (um vetor), de uma atenuação (mais uma vez apenas definimos a linear) e de um expoente correspondente à intensidade de distribuição da luz.

```
class LuzFoco : public Luz {


private:


 float angulo;
 float direcao[3];
 float atenuacao;
 float expoente;


public:
 LuzFoco();


 LuzFoco(float[4], int,float,float[3],float,float);


 void desenhaLuz();


};
```

Na função que “desenha” a luz apenas tem de acrescentar as componentes relativas à luz do tipo SPOT como mostramos de seguida:

```
 glLightfv(GL_LIGHT0 +luzA, GL_SPOT_DIRECTION, direcao);
 glLightf(GL_LIGHT0 +luzA, GL_SPOT_CUTOFF, angulo);
```

```
glLightf(GL_LIGHT0 +luzA,GL_SPOT_EXPONENT, expoente);
```

Figura

Nesta fase tivemos de alterar um pouco a classe **Figura**, visto que esta passaria a conter também vetores correspondentes às normais e às texturas. Para além disto guardamos o número de pontos de normais e texturas. Guardamos também um inteiro correspondente ao id de textura, um objeto do tipo **Material** correspondente à cor da figura e um objeto do tipo **ViewFrustumeColisao** com os limites da figura e que auxilia na decisão de desenhar ou não a mesma.

```
class Figura {


    GLuint pontos[1];

    int numeroPontos;

    GLuint normais[1];

    int numeroNormais;

    GLuint texturas[1];

    int numeroTexturas;

    int idTextura;

    Material* cor;

    ViewFrustumeColisao * teste;


private:


    void
adicionaPontos(vector<Ponto*>,vector<Ponto*>,vector<Ponto*>);


public:

    Figura();


Figura(vector<Ponto*>,vector<Ponto*>,vector<Ponto*>,int,Material*,View
FrustumeColisao*);

    GLuint getPontos();

    bool desenha(Plane[6],glm::mat4);

    string toString();

};
```

O processo de inicialização dos vetores de normais e texturas é semelhante ao processo de inicialização dos pontos, referido na fase anterior.

Alteramos também a função que desenha uma figura, passando esta a receber 2 parâmetros: os planos correspondentes ao *Frustum* da câmara e a matriz com as transformações geométricas realizadas até ao momento, ou seja, a matriz com as transformações do espaço global. Antes de desenhar alguma coisa testa se pode desenhar, com o auxílio de uma função do teste, verificando se este objeto se encontra dentro do *Frustum* da câmara. Caso não se encontre retorna falso e não desenha nada. Caso esteja dentro, verifica se a componente de cor não é nula e caso não seja "desenha" essa componente, e associa o id de textura do objeto e desenha os respetivos *VBOs* (dos pontos, normais e texturas).

```
bool Figura::desenha(Plane pl[6], glm::mat4 matriz){
    if(!teste->estaDentro(pl,matriz)){
        return false;
    }

    if(cor!= nullptr){
        cor->desenha();
    }
    glBindTexture(GL_TEXTURE_2D, idTextura);
    glBindBuffer(GL_ARRAY_BUFFER, pontos[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, normais[0]);
  glNormalPointer(GL_FLOAT,0,0);

  glBindBuffer(GL_ARRAY_BUFFER,texturas[0]);
  glTexCoordPointer(2,GL_FLOAT,0,0);
        glDrawArrays(GL_TRIANGLES, 0, numeroPontos);
  return true;

  }
```

Material

Esta classe corresponde às diferentes componentes de cor dos objetos: difusa, especular, ambiente e emissiva. Esta classe contém um vetor para cada uma das diferentes

componentes da cor com 4 componentes, sendo as primeiras 3 os valores R, G e B respetivamente, a última é sempre 1. Para além disto contém também um valor correspondente ao “shininess” que corresponde ao brilho da componente especular.

```
float difusa[4];
float ambiente[4];
float especular[4];
float emissiva[4];
float shininess;
public:
	Material();
	Material(Cor*, Cor*, Cor*, Cor*, float);
	void desenha();
```

View Frustum e Colisão

Por forma a aumentar a performance e a aproximar o modelo da realidade decidimos ter uma classe que verifica se uma determinada figura pode ser desenhada e se a câmara se pode mover consoante os objetos do sistema.

Esta classe tem um valor booleano a indicar se a figura é ocupada por um volume correspondente a uma esfera ou a uma caixa. Tem também um valor do raio, que caso seja correspondente a uma esfera será igual à distância do centro (ponto (0,0,0)) a um dos pontos de controlo. Tem também os pontos de controlo correspondentes aos vértices de uma caixa. No que toca à colisão consideramos todos os objetos como uma esfera.

```
class ViewFrustumeColisao{

 bool esfera;
 vector<Ponto*> pontosControlo;
 float raio;
 float coordenadas[6];

 int planoRejeitado = 0;

 private:
	void atualizaCoordenadas(float[8][4]);
	float distancia(float pl[4], float p[4]);
```

```
        void multiplicaMatriz(float p[4], float tg[16]);
        bool estaDentroEsfera(Plane[6], float centro[4]);
        bool estaDentroCaixa(Plane[6]);
    public:
        ViewFrustumeColisao();
        ViewFrustumeColisao(bool,vector<Ponto*>,float);

        bool estaDentro(Plane[6],glm::mat4);

        bool possoMover(float,float,float,float,glm::mat4);
    };
```

Frustum

Por forma a poder aplicar o mecanismo de View Frustum necessitamos de ter o Frustum da câmara. Para isso utilizamos o código disponibilizado no site *lighthouse3d* sendo que iremos apresentar o processo de cálculo do mesmo.

```
class FrustumG
{
private:

    enum {
        TOP = 0,
        BOTTOM,
        LEFT,
        RIGHT,
        NEARP,
        FARP
    };


public:
    Plane pl[6];
    Vec3 ntl,ntr,nbl,nbr,ftl,ftr,fbl,fbr;
    float nearD, farD, ratio, angle,tang;
```

```
    float nw,nh,fw,fh;


    FrustumG();
    ~FrustumG();


    void setCamInternals(float angle, float ratio, float nearD, float
farD);
    void setCamDef(Vec3 &p, Vec3 &l, Vec3 &u);
    int pointInFrustum(Vec3 &p);
    int sphereInFrustum(Vec3 &p, float raio);
    //int boxInFrustum(AABox &b);


    void drawPoints();
    void drawLines();
    void drawPlanes();
    void drawNormals();
};
```

Plano do Frustum

Por forma a auxiliar na definição do Frustum da câmara também utilizamos a classe Plane disponibilizado no mesmo site. Esta classe contém as normais a um plano, bem como o valor D que com o conjunto das normais corresponde à equação do mesmo. Para além disso calcula a distância de um ponto ao plano.

```
class Plane
{


public:


    Vec3 normal,point;
    float d;



    Plane( Vec3 &v1,  Vec3 &v2,  Vec3 &v3);
    Plane(void);
    ~Plane();
```

```
void set3Points( Vec3 &v1,  Vec3 &v2,  Vec3 &v3);
void setNormalAndPoint(Vec3 &normal, Vec3 &point);
void setCoefficients(float a, float b, float c, float d);
float distance(Vec3 &p);

void print();

};
```

3.3. Parsing

Nesta última fase introduzimos algumas novas estruturas, como referido anteriormente, por forma a tornar o modelo uma melhor aproximação ao mundo real. Era de esperar que isso afetasse a forma como é efetuado o parsing do ficheiro xml, e realmente foi o que aconteceu. No entanto, as alterações necessárias não foram tantas quanto isso, sendo que apenas foi necessário acrescentar algumas funções para que o parser se comportasse como o esperado. De seguida vamos apresentar as alterações que realizamos bem como os algoritmos de parsing.

Grupo

Inicialmente alteramos a forma como realizamos parsing de um grupo, visto que agora passamos a conter luzes (representativas da iluminação) dentro do mesmo, logo tínhamos de ter uma função que realizasse a extração da sua informação. Decidimos que as luzes seriam o primeiro elemento do grupo, ou seja, apareceriam antes das transformações do mesmo. Logo a única alteração necessária neste contexto foi criar uma função que faça o parsing das respetivas e na função que faz o mesmo ao grupo chamamos essa função logo no início.

```
Grupo* parseGrupo(XMLElement* elemento){
 Grupo* res = new Grupo(numeroGrupo++);
      if(strcmp(elemento->Name(),"lights") == 0){
        parseLuzes(elemento, res);
        elemento = elemento->NextSiblingElement();
      }
 parseOperacoes(&elemento,res);
 if(strcmp(elemento->Name(),"models") == 0){
       parseModelos(elemento, res);
       elemento = elemento->NextSiblingElement();
```

```
}
 while((elemento) && (strcmp(elemento->Name(),"group") == 0)){
       XMLElement *elementoFilho = elemento->FirstChildElement();
       if(elementoFilho){
             Grupo* filho = parseGrupo(elementoFilho);
             res->adicionaGrupo(filho);
       }
       elemento = elemento->NextSiblingElement();
 }
 return res;
 }
```

Luzes

Por forma a introduzir o novo conceito das luzes decidimos criar uma função que faz o parsing das mesmas. Para além disto e visto que o OpenGL apenas permite inserir 8 luzes decidimos ter o número da luz como uma variável global, sendo que quando aparece uma nova luz incrementamos esse valor, para que a próxima seja uma luz diferente da atual. No entanto, se aparecerem mais do que 8 luzes as últimas corresponderão todas à mesma.

A função que faz o parsing das luzes auxilia-se em 3 funções, que correspondem ao parsing de uma luz de um tipo específico (como referido anteriormente). Esta apenas verifica qual o nome do tipo da luz, de entre dos seguintes: POINT, SPOT, DIRECTIONAL. Quando verifica que é de um dado tipo chama a função que faz o parsing de luzes correspondente a esse tipo.

```
void parseLuzes(XMLElement *elemento, Grupo* g){

    XMLElement * pElement = elemento->FirstChildElement("light");

    while (pElement != nullptr){

       if(pElement->Attribute("type")){

           if(strcmp(pElement->Attribute("type"),"POINT")==0){
                parseLuzPontual(pElement,g);
           }
```

```
                if(strcmp(pElement->Attribute("type"),"SPOT")==0){
                    parseLuzFoco(pElement,g);
                }

                if(strcmp(pElement-
>Attribute("type"),"DIRECTIONAL")==0){
                    parseLuzDirecional(pElement,g);
                }

            }
             pElement = pElement->NextSiblingElement("light");
        }
```

Como referido anteriormente as luzes pontuais são caraterizadas por um ponto (posição) e uma atenuação. Inicialmente começamos com os valores por “defeito”, ou seja, o ponto é a origem e a atenuação 0. Para verificar qual o ponto (se este será diferente do valor por “defeito”) chamamos a função **parseLuz** que verifica qual é o ponto de uma luz (esta função foi criada por ser comum a todos os tipos de luzes). Depois de verificar o ponto, verifica se o elemento correspondente no ficheiro XML tem um atributo com o nome “atenuation” correspondente à atenuação e caso contenha extrai esse valor e atualiza o valor da atenuação.

No final cria um objeto do tipo LuzPontual e adiciona-o ao grupo.

```
    void parseLuzPontual(XMLElement* elemento,Grupo* grupo){
        float pos[4] = {0,0,0,1};
        float atenuacao = 0;
        if(elemento){
            float tempo=0;
            parseLuz(pos,elemento);
            if(elemento->Attribute("atenuation")){
                const      char*      tempoAux      =      elemento-
>Attribute("atenuation");
                tempo = atof(tempoAux);
                atenuacao = tempo;
            }
            LuzPontual* l = new LuzPontual(pos,nrLuz++,atenuacao);
            grupo->adicionaLuz(l);
```

```
    }
}
```

Para realizar o parsing de uma luz do tipo **Direcional** decidimos ter uma função que realiza o mesmo. A luz direcional é bastante simples visto que apenas necessita de uma direção (já que nós assumimos as cores sempre como as default) pelo que iniciamos essa direção como o vetor (0,0,1) e chamamos a função **parseLuz** para extrair informação sobre a direção (a direção e a posição são bastante parecidas diferindo apenas no último valor, no entanto a função **parseLuz** apenas atualiza as componentes X, Y e Z pelo que pode ser utilizada por ambas).

No final criamos um objeto do tipo LuzDirecional e adicionamos ao grupo.

```
void parseLuzDirecional(XMLElement* elemento,Grupo* grupo){
    float pos[4] = {0,0,1,0};
    if(elemento){
        parseLuz(pos,elemento);
        LuzDirecional* l = new LuzDirecional(pos,nrLuz++);
        grupo->adicionaLuz(l);
    }
}
```

Como referido anteriormente para a luz do tipo **SPOT** são necessários mais campos do que o tipo de luz mencionado anteriormente. Para além da posição e atenuação, nos quais os mecanismos são semelhantes à luz pontual, são também necessários o ângulo da direção da luz, a direção da mesma e o expoente (correspondente à intensidade da distribuição da luz). Para os diferentes campos inicializamos cada um com os valores por “defeito” do OpenGL e quando encontramos um atributo no elemento xml que corresponda aos campos, atualizamos o valor do mesmo.

- Cut_off -> corresponde ao ângulo
- Exponente -> corresponde ao expoente
- dirX -> ao valor X da direção
- dirY -> ao valor Y da direção
- dirZ -> ao valor Z da direação

No final criamos um objeto do tipo LuzFoco e adicionamos ao grupo.

```
void parseLuzFoco(XMLElement* elemento,Grupo* grupo){
    float pos[4] = {0,0,0,1};
    float atenuacao = 0;
```

```
        float direcao[3] = {0,0,-1};
        float angulo = 180;
        float expoente = 0;
        if(elemento){
            float tempo=0;
            float x=0,y=0,z=0;
            parseLuz(pos,elemento);
            if(elemento->Attribute("atenuation")){
                const      char*      tempoAux      =      elemento-
>Attribute("atenuation");
                tempo = atof(tempoAux);
                atenuacao = tempo;
            }
            if(elemento->Attribute("cut_off")){
                const char* tempoAux = elemento->Attribute("cut_off");
                tempo = atof(tempoAux);
                if(tempo >= 0 && tempo <= 90){
                    angulo = tempo;
                }

            }
         if(elemento->Attribute("exponent")){
                const char* tempoAux = elemento->Attribute("exponent");
                tempo = atof(tempoAux);
                if(tempo > 0 && tempo <= 128) {
                    expoente = tempo;
                }
            }
            if(elemento->Attribute("dirX")){
                const char* xAux = elemento->Attribute("dirX");
                x = atof(xAux);
                direcao[0] = x;
            }
            if(elemento->Attribute("dirY")){
                const char* yAux = elemento->Attribute("dirY");
                y = atof(yAux);
                direcao[1] = y;
```

```
            }
            if(elemento->Attribute("dirZ")){
                const char* zAux = elemento->Attribute("dirZ");
                z = atof(zAux);
                direcao[2] = z;
            }
            LuzFoco*                l                =                new
LuzFoco(pos,nrLuz++,angulo,direcao,atenuacao,expoente);
            grupo->adicionaLuz(l);
    }
    }
    }
```

A função que realiza o parsing de uma luz recebe a posição como argumento e caso encontre, no elemento XML, o atributo:

- posX -> atualiza o campo X da posição (o 1º elemento)
- posY -> atualiza o campo Y da posição (o 2º elemento)
- posZ -> atualiza o campo Z da posição (o 3º elemento)

```
    void parseLuz(float* pos, XMLElement* elemento){
        float x=0,y=0,z=0;

        if(elemento->Attribute("posX")){
            const char* xAux = elemento->Attribute("posX");
            x = atof(xAux);
            pos[0] = x;
        }
        if(elemento->Attribute("posY")){
            const char* yAux = elemento->Attribute("posY");
            y = atof(yAux);
            pos[1] = y;
        }
        if(elemento->Attribute("posZ")){
            const char* zAux = elemento->Attribute("posZ");
            z = atof(zAux);
            pos[2] = z;
```

```
    }

}
```

Modelos

Para além de termos acrescentado as componentes de iluminação, acrescentamos também normais, texturas e componentes de cor aos diferentes modelos. Acrescentamos também, como referido, uma classe que corresponde aos limites de cada modelo, para ser utilizada como auxílio nos mecanismos de colisões e view frustum. Para isso tivemos de alterar a forma como a informação dos modelos é extraída.

Primeiro verificamos se o elemento tem um atributo do tipo “file” correspondente ao nome do ficheiro .3d. Caso contenha, então podemos processar ao parsing da Figura correspondente.

De seguida verificamos se contém o atributo “texture” que indica qual o nome do ficheiro correspondente à textura da Figura.

De seguida invocámos a função **parseMaterial** para nos extrair as componentes de cor da Figura correspondente.

Posteriormente chamamos a função **parseViewFrustum**, que nos extrai as informações correspondentes aos limites da Figura.

Depois invocámos a função **extraiFicheiro**, passando como argumentos o nome do ficheiro, o nome do ficheiro de textura, o objeto **Material** correspondente à cor e o objeto **ViewFrustumeColisao** correspondente aos limites.

De seguida adicionamos ao grupo a Figura retornada pela função **extraiFicheiro**.

```
void parseModelos(XMLElement *elemento, Grupo* g){
 XMLElement * pElement = elemento->FirstChildElement("model");
 while (pElement != nullptr){
       const char* cenas = pElement->Attribute("file");
       if (cenas != nullptr){
            string fich(cenas, strlen(cenas));
            string textura;
            Material* material = nullptr;
            const char* xAux = pElement->Attribute("texture");
            if(xAux!= nullptr){
                string texAux(xAux,strlen(xAux));
                textura = texAux;
```

```
            }
            material = parseMaterial(pElement);
            ViewFrustumeColisao * vf = parseViewFrustum(pElement);
            Figura* f = extraiFicheiro(fich,textura,material,vf);
            g->adicionaFigura(f);
    }
```

Material

Relativamente ao parsing das componentes de cor de uma Figura decidimos iniciar todas as componentes com o valor nulo e o valor do shininess com o valor -1.

Sempre que encontrávamos um atributo no elemento que correspondesse a uma componente de cor então inicializávamos a componente correspondente com os valores por defeito e fazíamos o parsing dessa componente. Por exemplo, quando encontramos um atributo do tipo specR ou specG ou specB então podemos inicializar a componente especular com os valores (0,0,0) e depois vamos ver quais dos 3 atributos acima existem e atualizamos cada um dos valores (R, G ou B) na componente consoante o atributo. Todos os casos são análogos.

De seguida apesentamos os valores iniciais para cada componente:

• Emissiva -> R=0; G=0; B=0

• Especular -> R=0; G=0; B=0

• Ambiente -> R=0.2; G=0.2; B=0.2

• Difusa-> R=0.8; G=0.8; B=0.8

Caso encontrássemos um atributo do tipo “shine” então atualizamos o valor do campo shininess.

No final caso todas as componentes fossem nulas e o valor de shininess fosse igual a -1, então significava que não teriam sido realizadas alterações, ou seja a Figura não teria componente de cor, pelo que retornávamos um valor nulo.

Caso isto não se observasse, então para cada componente que fosse nula iniciávamos essa componente com o seu valor por “defeito” e caso o shininess fosse inferior a 0 ou superior a 128 então tornávamos esse com o valor 0. De seguida criávamos um objeto do tipo Material com estes campos e retornávamos esse objeto.

```
Material* parseMaterial(XMLElement* elemento){
    Cor* emissiva = nullptr;
    Cor* especular nullptr;
    Cor* difusa=  nullptr;
```

```
        Cor* ambiente = nullptr;
        float shininess = -1;

        if(elemento->Attribute("emiR")          ||          elemento-
>Attribute("emiG")|| elemento->Attribute("emiB"){
            emissiva = new Cor(0,0,0);
            if(elemento->Attribute("emiR")){
                    const char* xAux = elemento->Attribute("emiR");
                    x = atof(xAux);
                    emissiva.setR(x);
                }
                if(elemento->Attribute("emiG")){
                    const char* yAux = elemento->Attribute("emiG");
                    y = atof(yAux);
                    emissiva.setG(y);;
                }
                if(elemento->Attribute("emiB")){
                    const char* zAux = elemento->Attribute("emiB");
                    z = atof(zAux);
                    emissiva.setB(z);
                }
        }
        if(elemento->Attribute("specR")          ||          elemento-
>Attribute("specG")|| elemento->Attribute("specB"){
            especular = new Cor(0,0,0);
            if(elemento->Attribute("specR")){
                const char* xAux = elemento->Attribute("specR");
                x = atof(xAux);
                especular.setR(x);
            }
            if(elemento->Attribute("specG")){
                const char* yAux = elemento->Attribute("specG");
                y = atof(yAux);
                especular.setG(y);
            }
            if(elemento->Attribute("specB")){
                const char* zAux = elemento->Attribute("specB");
```

```
                z = atof(zAux);
                especular.setB(z);
            }
        }

        if(elemento->Attribute("diffR")          ||          elemento-
>Attribute("diffG")|| elemento->Attribute("diffB"){
            difusa = new Cor(0.8,0.8,0.8);
            if(elemento->Attribute("diffR")){
                const char* xAux = elemento->Attribute("diffR");
                x = atof(xAux);
                difusa.setR(x);
            }
            if(elemento->Attribute("diffG")){
                const char* yAux = elemento->Attribute("diffG");
                y = atof(yAux);
                difusa.setG(y);
            }
            if(elemento->Attribute("diffB")){
                const char* zAux = elemento->Attribute("diffB");
                z = atof(zAux);
                difusa.setB(z);
            }
        }

        if(elemento->Attribute("ambR")           ||          elemento-
>Attribute("ambG")|| elemento->Attribute("ambB"){
            ambiente = new Cor(0,0,0);
            if(elemento->Attribute("ambR")){
                const char* xAux = elemento->Attribute("ambR");
                x = atof(xAux);
                ambiente.setR(x);
            }
            if(elemento->Attribute("ambG")){
                const char* yAux = elemento->Attribute("ambG");
                y = atof(yAux);
                ambiente.setG(y);
```

```
            }
            if(elemento->Attribute("ambB")){
                const char* zAux = elemento->Attribute("ambB");
                z = atof(zAux);
                ambiente.setB(z);
            }
        }

        if(elemento->Attribute("shine")){
            const char* zAux = elemento->Attribute("shine");
            z = atof(zAux);
            shininess = z;
        }

        if( (emissiva == nullptr ) && (difusa == nullptr ) && (especular
== nullptr ) && (ambiente == nullptr ) && (shininess == -1) ){

            return nullptr;
        }

        else{
            if( emissiva == nullptr) {
                emissiva = new Cor(0,0,0);
            }

            if( ambiente == nullptr) {
                ambiente = new Cor(0.2,0.2,0.2);
            }

            if( difusa == nullptr) {
                difusa = new Cor(0.8,0.8,0.8);
            }

            if( especular == nullptr) {
                especular = new Cor(0,0,0);
            }
```

```
            if( shininess < 0 || shininess > 128 ){
                shininess = 0;
            }

        }

        Material * material = new Material(difusa,ambiente,especular,emissiva,shininess);
    }
```

ViewFrustum

Para realizar o parsing das componentes que limitam o espaço de uma Figura, e como referido anteriormente apenas consideramos dois tipos de volumes: esferas e caixas. Por defeito consideramos que é uma esfera e que o raio é 0, sendo que não tem pontos de controlo.

Se encontrarmos um atributo do tipo “limit” no elemento XML, verificamos se esse atributo é “E” ou seja, uma esfera. Caso seja, então verificamos a existência de um atributo do tipo “raio” e atualizamos o valor do raio com o valor desse atributo.

Caso não seja uma esfera, então iteramos os elementos XML por atributos do tipo “P” (correspondentes aos pontos de controlo) e enquanto existirem pontos, até um máximo de 8 adicionamos ao vetor dos pontos de controlo. Para além disso atualizamos o valor booleano que indica se é ou não uma esfera para falso. De referir apenas que os nossos modelos vão ter sempre 8 pontos de controlo, e caso isso não se verifique em algum caso então deixaria de ser uma caixa.

No final criamos um objeto do tipo **ViewFrustumeColisao** com os parâmetros descritos acima (valor booleano, pontos de controlo e o raio) e retornamos esse objeto.

```
ViewFrustumeColisao* parseViewFrustum(XMLElement* elemento){
    bool esfera = true;
    float raio = 0;
    vector<Ponto*> pontosC;

    if(elemento->Attribute("limit")){
        const char* xAux = elemento->Attribute("limit");
        if(xAux[0] == 'E'){
            esfera = true;
            if(elemento->Attribute("raio")){
                const char* xAux = elemento->Attribute("raio");
                float x = atof(xAux);
```

```
                raio = x;
            }
        }

        else{
            esfera=false;
            elemento = elemento->NextSiblingElement("P");
            for(int i = 0; i < 8 && elemento; i++){
                float x=0,y=0,z=0;
                if(elemento->Attribute("posX")){
                    const char* xAux = elemento->Attribute("posX");
                    x = atof(xAux);

                }
                if(elemento->Attribute("posY")){
                    const char* yAux = elemento->Attribute("posY");
                    y = atof(yAux);

                }
                if(elemento->Attribute("posZ")){
                    const char* zAux = elemento->Attribute("posZ");
                    z = atof(zAux);

                }

                Ponto* p = new Ponto(x,y,z);
                pontosC.push_back(p);

                elemento = elemento->NextSiblingElement("P");

            }
        }

    }

    ViewFrustumeColisao* res = new
ViewFrustumeColisao(esfera,pontosC,raio);
```

```
    return res;
}
```

Extrair a informação do ficheiro e criar a Figura

Para extrair a informação do ficheiro alteramos um pouco o algoritmo visto que agora o ficheiro .3d contém não só os pontos, como as normais e os pontos de textura correspondentes. No entanto a estrutura do ficheiro permanece muito parecida, ou seja, aparece sempre o número de pontos antes de se iniciar a lista dos mesmos e depois aparecem as 3 coordenadas correspondentes, diferindo apenas nos pontos de textura no qual apenas existem 2 coordenadas. O algoritmo é o seguinte:

1. Abre o ficheiro (passado como parâmetro)

2. Ler o número de pontos

3. Enquanto não se atingir o número de pontos:

4. Retirar os valores correspondentes às 3 coordenadas

5. Criar o ponto correspondente e adicionar ao vetor dos pontos

6. Ler o número de normais

7. Enquanto não se atingir o número de normais:

8. Retirar os valores correspondentes às 3 coordenadas

9. Criar o ponto correspondente e adicionar ao vetor das normais

10. Ler o número de pontos de textura

11. Enquanto não se atingir o número de pontos textura:

12. Retirar os valores correspondentes às 2 coordenadas

13. Criar o ponto correspondente e adicionar ao vetor dos pontos de textura

14. Fecha o ficheiro e extrai o valor das texturas para um inteiro a partir do ficheiro passado como parâmetro

15. Criar um objeto do tipo **Figura** com os diferentes vetores de pontos, o inteiro correspondente às texturas e os argumentos passados como parâmetros (o objeto do tipo **Material** correspondente à cor e o objeto do tipo **ViewFrustumeColisao** correspondente aos limites)

16. Retornar o objeto criado

3.4. Ficheiro XML

Em relação à terceira fase foram pequenas as alterações efetuadas ao ficheiro XML gerado. As alterações efetuadas devem-se à luz na cena, às texturas dos diferentes elementos e também da inclusão do “view frustum” e das colisões.

Para a luz apenas tivemos de colocar um novo campo, no sol, que tem como objetivo dizer qual o tipo da fonte de luz, no nosso caso “point ligth”. E em cada objeto foi colocado as caraterísticas emissivas de cada um.

As maiores modificações foram efetuadas por causa do “view frustum”. Para que tal fosse possível de implementar precisávamos de definir *Boxes* (caixas) e esferas para delimitar o objeto. Para os planetas, como seria óbvio, a figura adotada foi uma esfera, enquanto que para o foguetão, teapot e o anel, devido às suas caraterísticas, decidimos usar a caixa.

Para a definição de qual o tipo de objeto a ser usado no frustum, no campo do *model* colocámos o par “limit=X”, sendo que X poderia ser E (que indica uma esfera) e B (que indica uma caixa). Caso seja uma esfera de seguida apresentámos qual o raio da mesma, que tem o valor do raio do planeta correspondente. Caso seja uma caixa foi preciso definir os pontos de controle de cada elemento, que foram calculados, vendo o máximo e o mínimo de cada uma das coordenadas de cada elemento.

De salientar ainda que para contornar o problema da terceira fase relativo ao movimento dos asteroides (onde apenas se viam as trajetórias e não os asteróides) decidimos colocar uma *flag*, assim no *engine* quando fosse apresentada a *flag*, este não desenhava a trajetória.

3.5. View Frustum

Construção

Para calcular o View Frustum da câmara optamos por utilizar a formula geométrica. Para esta fórmula precisamos de diferentes parâmetros da câmara: o valor do near, do far, o rácio do espaço (largura <-> altura) e o valor do ângulo de visão da mesma.

Estes diferentes campos são importantes visto que o frustum da câmara é parecido ao nosso, ou seja, é definido por uma pirâmide consoante o ângulo de visão (a abertura da pirâmide é dependente do ângulo). No entanto não é bem uma pirâmide visto que o plano mais distante é o que corresponde ao valor do far e o plano mais aproximado correspondente ao valor do near, sendo uma pirâmide truncada começando em “near” e não em 0. Por isso precisamos destes valores para definir os 8 pontos que vão definir os 6 planos do frustum.

De seguida apresentamos os valores das alturas e larguras para os planos correspondentes ao near e far da câmara (serão planos do frustum).

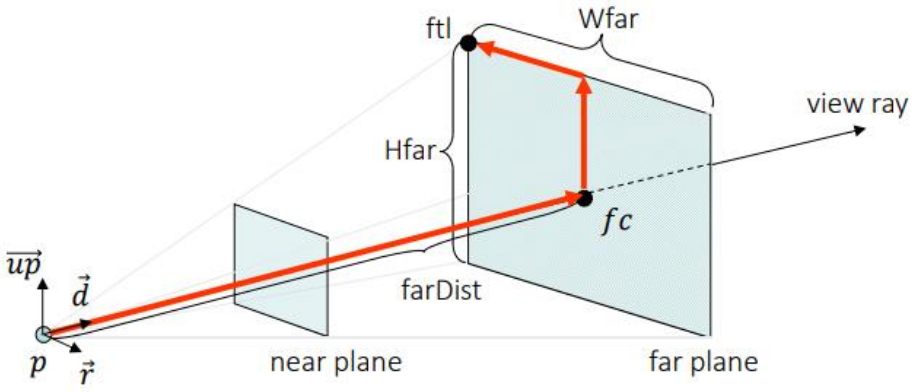
fov corresponde ao ângulo de visão, nearDist e farDist aos valores do near e do far e ratio ao rácio do “ecrã”.

$$Hnear = 2 \times \tan(\frac{fov}{2}) \times nearDist$$
$$Wnear = Hnear \times ratio$$
$$Hfar = 2 \times \tan(\frac{fov}{2}) \times farDist$$
$$Wfar = Hfar \times ratio$$

Para calcuar o Frustum é muito fácil perceber que este é dependente da câmara, logo dependerá dos eixos da câmara, pelo que antes de calcular o mesmo tivemos de calcular os eixos da câmara. Utilizamos os seguintes cálculos:

- Z = (Pos – L) <- normalizado
- X = up * Z <- normalizado, com up = (0,1,0)
- Y = Z * X <- normalizado
- D = -Z

De referir que como são vetores as operações correspondem a produtos vetoriais.



Depois de definidas os eixos da câmara apenas temos de definir os pontos de controlo pois estes são suficientes para definir os 6 planos.

Analisando a imagem podemos verificar que:

- fc = $far * \vec{d}$
- ftl = fc + $\frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x} = far * \vec{d} + \frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x}$

Na imagem up corresponde ao y e r ao x.

Os restantes pontos são: (f->far/n->near) (t->top/b->bottom) (l->left(r->right)

- ftl = $far * \vec{d} + \frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x}$
- ftr = $far * \vec{d} + \frac{Hfar}{2} * \vec{y} + \frac{Wfar}{2} * \vec{x}$
- fbl = $far * \vec{d} - \frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x}$
- fbr = $far * \vec{d} - \frac{Hfar}{2} * \vec{y} + \frac{Wfar}{2} * \vec{x}$

- ntl = $near * \vec{d} + \frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x}$
- ntr = $near * \vec{d} + \frac{Hfar}{2} * \vec{y} + \frac{Wfar}{2} * \vec{x}$
- nbl = $near * \vec{d} - \frac{Hfar}{2} * \vec{y} - \frac{Wfar}{2} * \vec{x}$
- nbr = $near * \vec{d} - \frac{Hfar}{2} * \vec{y} + \frac{Wfar}{2} * \vec{x}$

Para calcular cada plano do Frustum precisamos do vetor normal ao mesmo e de um ponto, visto que estes definem o plano. Para esta ser normalizada, basta que o vetor esteja normalizado.

A equação do plano é:

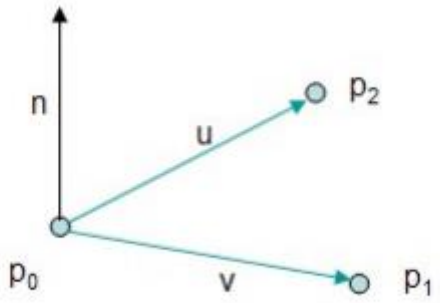
- Ax + By + Cz + D = 0. O vetor normal ao mesmo é (A,B,C).

Para calcular o valor de D apenas necessitamos de um ponto que pertença ao plano e substituir as suas coordenadas dando o resultado:

- p0 -> (pX,pY,pZ)
- -D = A * pX + B * pY + C * pZ = $\vec{n}$. p0 (produto escalar entre 2 vetores pois o ponto pode ser considerado como um vetor)

Para calcular a normal ao plano, são necessários 3 pontos do mesmo, sendo que esta corresponde ao produto vetorial entre os vetores:

- $\vec{v}$ = p1 – p0
- $\vec{u}$ =p2 - p0
- $\vec{n}$ = $\vec{v}$ * $\vec{u}$



Logo para calcular cada plano do frustum apenas precisamos de 3 pontos que pertençam a esse plano. Dos 8 pontos calculados conseguimos criar conjuntos de 3 pontos que vão definir os 6 planos. Cada conjunto de 3 pontos que define o plano são:

- Topo -> (ntr,ntl,ftl)
- Baixo -> (nbl,nbr,fbr)

- Esquerdo -> (ntl,nbl,fbl)
- Direito -> (nbr,ntr,fbr)
- Próximo -> (ntl,ntr,nbr)
- Afastado -> (ftr,ftl,fbl)

De referir que cada um dos planos definidos anteriormente têm a sua normal a apontar para “dentro” do frustum pelo que a distância de um ponto a cada plano será superior a 0 se o ponto estiver “dentro” do frustum na perspetiva desse plano.

A distância de um ponto ao plano, visto que este está normalizado, é apenas o produto escalar entre o ponto e a normal do plano somando a componente d do plano.

- P = (x0,y0,z0)
- dist(p) = $\vec{n}$ • (x0,y0,z0) + d

Teste

Visto que decidimos apenas ter 2 tipos de volumes para cada figura, esferas e caixas, os testes para perceber se estes pertencem ao frustum é mais simplificado. De seguida vamos apresentar o algoritmo que representa o teste para cada uma das primitivas.

Esfera

Para testar se uma esfera está dentro do frustum temos de ter em conta que não devemos (nem é possível) testar todos os pontos da mesma. No entanto, há uma forma bastante simples para testar a esfera, calcula-se a distância do centro da esfera relativamente a todos os planos (como referido anteriormente) e caso esta seja (para todos os planos):

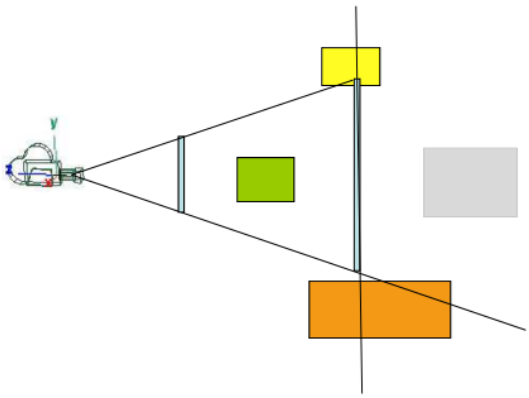
- positiva então a esfera está dentro do frustum, visto que o seu centro também o está
- negativa, mas em valor absoluto inferior ao raio, então significa que apesar do seu centro não se encontra no frustum existem partes da esfera que estão dentro, pelo que a esfera está dentro do frustum
- outro caso significa que está fora

Ou seja, o algoritmo é o seguinte:

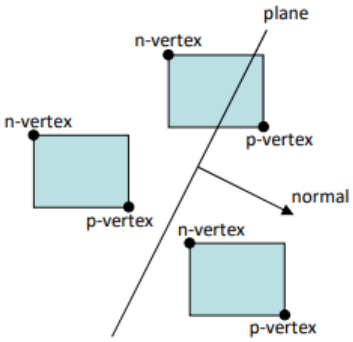
1. enquanto não percorrer todos os planos
    a. calcula distância do centro da esfera ao plano i
    b. caso distância seja inferior a -raio, então retorna falso (distancia < -raio)
2. retorna verdadeiro

Caixa

O teste de uma caixa é um pouco mais complexo do que de uma esfera, visto que estas são compostas por 8 vértices e não existe um raio a envolver as mesmas nem um centro. Para além disto ainda têm outro problema que é: o facto de existirem caixas que estão nas extremidades e têm alguns vértices dentro do frustum e outros fora. Primeiro é necessário perceber como eliminar esta dificuldade na qual alguns vértices estão dentro e outros estão fora, mas para isso apenas temos de pensar que só descartamos as caixas onde todos os vértices estão fora do mesmo plano. Este mecanismo faz com que por vezes sejam desenhadas algumas caixas a mais, no entanto esse número é tão pequeno que compensa a complexidade computacional em comparação com qualquer outro algoritmo possível.



Por forma a eliminar ainda algum esforço computacional no teste, podemos perceber que não é necessário testar os 8 pontos para cada plano, o que exigia (8*6=)48 testes por cada caixa. É possível verificar que os pontos mais próximos de cada plano são aqueles que se encontram no sentido da sua normal, ou seja:



No exemplo acima, os testes vão sempre recair no “p-vertex” visto que este é o vértice correspondente à direção da normal devido a: normal aponta no sentido positivo do x e negativo do y pelo que se escolhe o ponto com maior x e menor y da caixa, ou seja o “p-vertex”.

Logo é fácil perceber que para testar a caixa em cada plano, apenas temos de analisar o valor da normal do mesmo (que está identificado nos valore A, B e C) e verificar o seguinte:

- caso a componente x seja positiva escolhe a coordenada com x máximo, caso contrário o x mínimo
- caso a componente y seja positiva escolhe a coordenada com y máximo, caso contrário o y mínimo
- caso a componente z seja positiva escolhe a coordenada com z máximo, caso contrário o z mínimo

No final desta análise temos o ponto que está mais "próximo" do plano no sentido da normal (onde a distância é superior, visto que o que queremos é uma distância superior a 0) e apenas testamos esse ponto.

O algoritmo é o seguinte:

1. enquanto não percorrer todos os planos
    a. calcula o valor das normais do plano i e obtém vértice a testar
    b. calcula distância do vértice obtido ao plano i
    c. caso distância seja inferior a 0 então retorna falso (distancia < 0)
2. retorna verdadeiro

Otimizações

Segundo a "Translation-Rotation Coherency (Assarsson and Möller)":

- se um objeto é rejeitado pelo plano esquerdo e rodamos a câmara para a direita então o objeto continuará fora do frustum
- se um objeto é rejeitado por o plano mais próximo (near) e movemos a câmara para a frente este continuará rejeitado

Com isto chegaram a "Temporal Coherency (Assarsson and Möller)":

- guardar o plano que rejeitou o objeto
- testar esse plano primeiro

Por forma a otimizar o nosso teste, optamos por implementar a estratégia acima descrita, isto é: guardamos o plano que rejeitou o objeto (um inteiro) e realizamos o teste primeiro a esse plano e depois continuamos pelos outros de uma forma circular, por exemplo se fosse o 4 que rejeitou o teste seria: 4,5,0,1,2,3. No entanto é necessário definir qual o valor inicial, mas como no início ainda nada o rejeitou basta iniciar esse valor com o 0.

Implementação

Para verificar que uma figura se situava dentro do frustum ou não, decidimos guardar uma stack de matrizes, que correspondem às matrizes anteriores à atual para que quando se realize glPopMatrix() a matriz atual se atualize como a matriz sem as transformações geométricas desde o glPushMatrix(). Para além desta stack temos também a matriz atual que corresponde à matriz com o espaço global (as transformações realizadas até ao momento). Quando realizamos uma operação do tipo glPushMatrix() criamos uma nova matriz igual à atual e adicionamos essa matriz à stack. Quando realizamos o inverso (popMatrix) retiramos a matriz do topo da stack e atualizamos a matriz atual como essa matriz. A matriz inicial tem de ser a matriz identidade, pois é essa da qual partimos.

Por forma a utilizar estas matrizes, e visto que já existem bibliotecas que implementam as transformações geométricas auxiliamo-nos na biblioteca glm que realiza as operações que desejamos (rotações, translações e escalas) tendo apenas atenção que esta nos dá a matriz transposta à que queremos, sendo que quando queremos calcular a matriz atual para verificar onde se encontra um ponto (por forma a verificar se se encontra no frustum ou se a câmara não colidiu com este) temos de obter a transposta desta, bastando para isso apenas inverter os índices (i,j) para (j,i).

Quando testamos se uma Figura pode ser ou não desenhada passamos a matriz atual, pelo que com os limites definidos pelo Frustum fazemos as seguintes operaçãoes:

- caso seja uma esfera então atualizamos o ponto onde se encontra o centro a partir do ponto (0,0,0) que “é o centro por defeito” e utilizamos a função de teste de uma esfera
- caso seja uma caixa atualizamos cada um dos vetores de pontos de controlo correspondentes aos vértices da caixa. De seguida atualizamos um vetor com os mínimos e máximos de cada coordenada (x,y e z) pois serão úteis para a função que calcular a distância de uma caixa a um plano e depois chamamos essa função.
- A atualização de cada ponto é feita multiplicando a matriz do espaço global pela coordenada do ponto!

Inicialmente pensamos em ter um frustum com partições, ou seja, um frustum não só para as figuras, mas também para os grupos. Este frustum seria sempre representado por uma Caixa (por facilidade), e começaria com os 8 pontos de controlo iguais e correspondentes ao ponto (0,0,0). Depois sempre que adicionamos uma figura realizamos as operações do grupo (para que a matriz fique com os valores correspondentes ao espaço do grupo) à matriz do grupo, a partir da matriz identidade. Depois atualizamos casa ponto analisando as coordenadas máximas e mínimas da figura da seguinte forma:

- Calculamos as coordenadas com a análise dos valores (se algum é superior para as máximas e se algum é inferior para as mínimas)
- Atualizamos os 8 pontos a partir das novas coordenadas, com os respetivos:
    - (xmin,ymin,zmin)
    - (xmin,ymin,zmax)
    - (xmin,ymax,zmin)
    - (xmin,ymax,zmax)
    - (xmax,ymin,zmin)
    - (xmax,ymin,zmax)
    - (xmax,ymax,zmin)
    - (xmax,ymax,zmax)

Depois seria potencialmente mais eficiente calcular o teste para cada grupo, visto que se um grupo não fosse visível então as figuras que estavam “dentro” do mesmo também não seriam. No entanto não optamos por esta opção por 2 razões:

- A maior parte dos nossos grupos tem poucos figuras e os que têm mais teriam uma caixa de volume muito elevado
- Exige um pré-processamento muito elevado

Colisões

Para além de termos implementado o View Frustum decidimos também implementar colisões, sendo que para este caso foi mais simples. O mecanismo de colisões que utilizamos foi também para cada figura e neste caso definimos o seu volume como uma esfera em todo o caso. Para testar se existe uma colisão apenas temos de verificar se a nova posição da câmara não se encontra a uma distância do centro da esfera que define o volume da Figura inferior ou igual ao valor do raio da esfera.

No entanto, este algoritmo tem um pequeno problema, que é: se o passo da câmara for demasiado grande, esta pode “atravessar” a esfera e isso não deveria acontecer.

Para evitar isto poderíamos aplicar um no qual passamos a posição anterior da câmara, a vetor que corresponde à direção por onde esta se moveu e o valor do passo. Depois apenas tínhamos de calcular a interseção da esfera com o segmento de reta correspondente ao movimento da câmara, resolvendo o seguinte sistema:

$$(x - x0)^2 + (y - y0)^2 + (z - z0)^2 = r^2$$

$$x = x1 + k * a1$$

$$y = y1 + k * a2$$

$$z = z1 + k * a3$$

$$0 \leq k \ \leq n$$

Na qual:

- O ponto (x0,y0,z0) é o centro da esfera
- r é o raio da esfera
- O vetor (a1,a2,a3) é o vetor na qual a câmara se moveu
- O ponto (x1,y1,z1) é a posição da câmara antes de se mover
- n é o valor do passo da câmara

Da mesma forma que para o frustum inicialmente pensamos em aplicar um algoritmo no qual detetávamos colisões com os grupos também, pelo que em cada grupo se o teste desse falso, ou seja a câmara não se pudesse mover teríamos de verificar nos elementos dentro desse grupo o teste. No entanto caso o teste desse verdadeiro já não precisávamos de verificar os elementos do grupo pois estes estávamos contidos no espaço do grupo e não seriam “atingidos”. Pelas mesmas razões não aplicamos este método.

No entanto neste caso temos de verificar que existem duas vertentes que podem ser problemáticas para o nosso sistema.

A primeira tem a ver com o facto de nós realizarmos o teste antes de desenharmos os objetos e como algumas transformações dependem do tempo passado, o objeto vai ter uma posição diferente na altura do teste e na altura do desenho. Isto podia ser resolvido de uma forma bastante simples, guardando o valor do tempo numa variável na altura em que realizamos o teste da câmara e depois o objeto é desenhado consoante esse tempo, eliminado o problema.

A outra vem do facto dos objetos se “moverem” consoante o tempo e nós apenas realizarmos o teste ao mover a câmara. No entanto, a câmara pode não se mover e passar a colidir com um objeto que se moveu, pelo que a câmara devia alterar a sua posição (quase como se fosse empurrada). Resolver este problema já é um pouco mais complicado, no entanto pensamos que pode ser resolvido da seguinte forma:

- em vez de testar apenas quando se move a câmara, testar sempre
- quando um objeto colide com a câmara, então este terá de ficar na posição máxima entre o objeto e câmara segundo o eixo na qual ela se moveu (esta posição máxima corresponde à posição na qual a câmara não se pode mover mais pois encontrará o objeto)

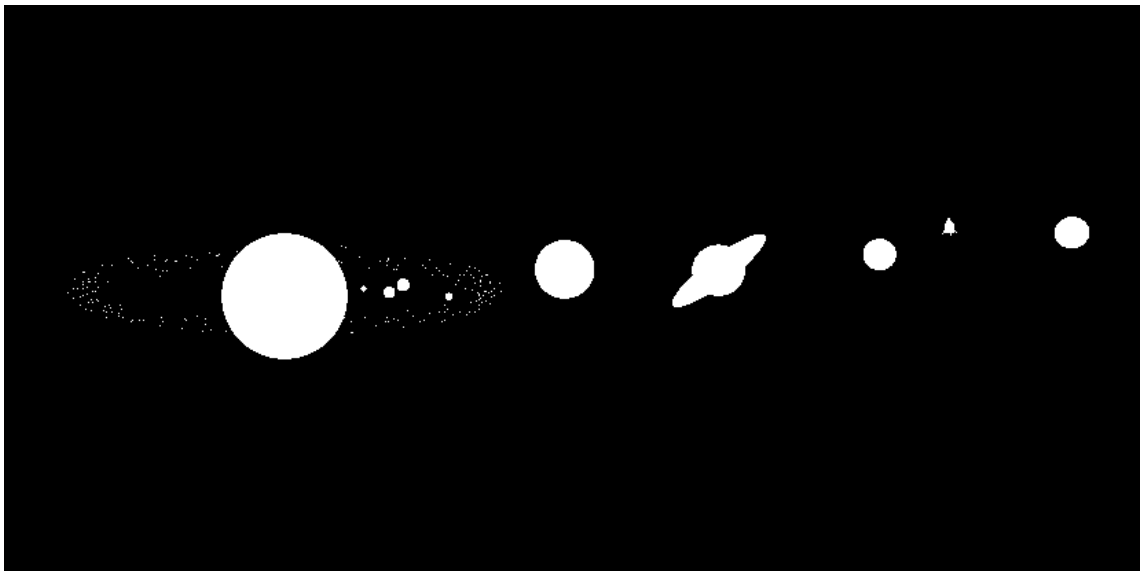
4. Conclusão

Nesta fase do projeto a parte que mais suscitou dúvidas e criou mais dificuldades foi a implementação do mecanismo de *view frustum culling* uma vez que era um conceito novo, do qual não tínhamos conhecimento e que não foi abordado nas aulas práticas desta unidade curricular. No entanto com o empenho de todos fomos superando gradualmente as dificuldades encontradas. Tal como referido na introdução também conseguimos com sucesso implementar um mecanismo de anti colisões no nosso sistema solar tornando-o mais real. Com a conclusão desta fase do projeto concluímos um ciclo de trabalho que durou grande parte do semestre e que culminou num resultado muito positivo. Achamos que atingimos com sucesso todos os objetivos traçados inicialmente e sentimos que graças a este trabalho ficamos com bases sólidas na área de computação gráfica que serão muito úteis no futuro.

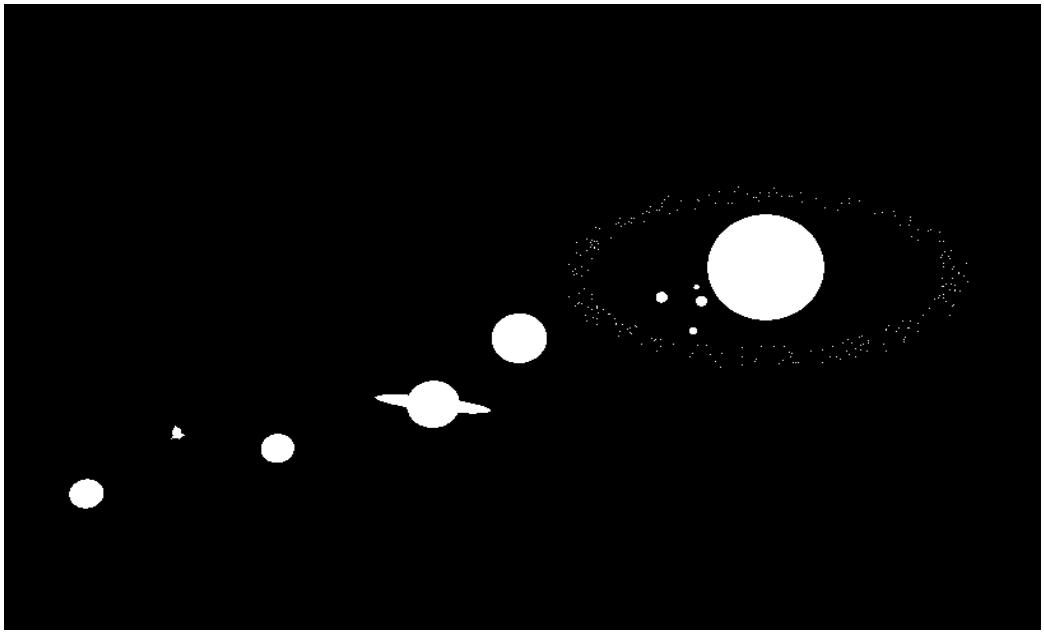
Anexo I

Segue em anexo algumas fotos do nosso sistema solar:

Vista lateral sistema solar:



Vista traseira sistema solar:



Foguetão: