# 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 visto 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 presentas 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\*,ViewFrustumeColisao\*);

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 à 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();

};