# Parsing ficheiro xml

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

Primeiramente 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 desse 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” dos mesmos, 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 (visto 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 um pouco mais de campos do que as anteriores. 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 o 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 chamamos a função **parseMaterial** para nos extrair as componentes de cor da Figura correspondente.

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

De seguida chamamos 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 mesmo 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 numero 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 numero 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 numero 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