Parsing do ficheiro XML

Para realizar o *parsing* do ficheiro XML primeiro analisamos as *tags* que este continha, sendo que delimitamos as seguintes:

* Uma principal chamada *scene*
* Uma secundária chamada *group* que está contida na principal, em cada uma destas podem estar também:
  + *translate/rotate/scale* que correspondem às transformações geométricas a um determinado grupo
  + *models* que são os ficheiros que o grupo contém (as figuras que irão ser desenhadas)
  + *group*, outros grupos

Posto isto, decidimos que a função principal teria de retornar um apontador para uma estrutura do tipo **Grupo** que seria a cena principal. Para conseguirmos ter acesso ao grupo principal, teríamos de aceder ao primeiro elemento filho do ficheiro com a tag *scene* e ao primeiro filho desse com a tag *group.* Depois chamávamos uma função que fazia o *parsing* de um grupo, ou seja:

XMLElement \* elemento = xmlDoc.FirstChildElement("scene")->FirstChildElement("group");

XMLElement \* elementoFilho = elemento->FirstChildElement();

Grupo\* grupo = parseGrupo(elementoFilho); Por OS IFS?

## Parsing de um grupo

Para realizar o *parsing* de um grupo reparamos que o ficheiro vem sempre ordenado da seguinte forma: transformações geométricas, modelos e os grupos restantes. Com isto, definimos uma ordem para realizar o *parsing* do grupo e criamos funções auxiliares, para realizar o *parsing* às operações geométricas e aos modelos, por forma a organizar melhor o código.

Ou seja, primeiro realizamos o *parsing* às operações geométricas a e aos modelos e só depois aos restantes grupos:

Nas operações temos de passar o apontador para o elemento para que este conteúdo seja alterado.

parseOperacoes(&elemento,res);

cout << elemento->Name() << endl;

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();

}

## Parsing de operações geométricas

Para realizar a leitura das operações geométricas tivemos de separar em três casos: rotações, escalas e translações. Para tornar mais fácil a compreensão, decidimos criar funções auxiliares para cada uma das operações. Ao realizar a leitura, comparamos o nome do elemento XML com *rotate/scale/translate* e caso fosse um dos mencionados chamamos a função auxiliar correspondente e avançamos para o próximo elemento, caso não fosse abandonávamos a função, pois já seria um modelo ou grupo.

if(strcmp((\*elemento)->Name(),"rotate") == 0){

parseRotacao(\*elemento,grupo);

}

else{

if(strcmp((\*elemento)->Name(),"scale") == 0){

parseEscala(\*elemento,grupo);

}

else{

if(strcmp((\*elemento)->Name(),"translate") == 0){

parseTranslacao(\*elemento,grupo);

}

else{

return;

}

}

}

(\*elemento)=(\*elemento)->NextSiblingElement();

if(\*elemento){

parseOperacoes(elemento,grupo);

}

Para realizar o *parsing* da rotação/escala/translação apenas teríamos de veiricar se continham os atributos (*X,Y,Z* no caso das últimas e *angle,axisX,axisY,axisZ* no caso da primeira) e caso contivessem então líamos o valor do número correspondente. Para inicializar as variáveis teríamos de inicializar ao valor neutro para a operação (tentar por de forma diferente esta frase, explicar de outra forma, ou pelo menos mudar um dos dois “inicializar”), ou seja, para as rotações e translações inicializamos a 0 e para as escalas a 1.

Depois adicionávamos cada operação ao grupo correspondente.

if(elemento){

float x=0,y=0,z=0,angulo=0;

if(elemento->Attribute("angle")){

const char\* anguloAux = elemento->Attribute("angle");

angulo = atof(anguloAux);

}

if(elemento->Attribute("X")){

const char\* xAux = elemento->Attribute("X");

x = atof(xAux);

}

if(elemento->Attribute("Y")){

const char\* yAux = elemento->Attribute("Y");

y = atof(yAux);

}

if(elemento->Attribute("Z")){

const char\* zAux = elemento->Attribute("Z");

z = atof(zAux);

}

Rotacao\* r = new Rotacao(x,y,z,angulo);

grupo->adicionaOperacao(r);

}

Os casos da translação e escala são análogos.

## Parsing dos modelos

Antes de realizar a leitura verificamos que cada *tag models* contém um conjunto de *tag model* que corresponde ao ficheiro. Para aceder a este necessitávamos de obter o primeiro filho do elemento XML com a *tag model* e depois continuar a iterar sobre estes elementos até que não houvesse mais nenhum.

Depois reparamos que cada modelo tem um atributo *file* que contém o nome do ficheiro, pelo que para obter o nome do mesmo acedemos a esse atributo. Após termos o nome do ficheiro (o **3D** criado anteriormente) chamamos a função que extrai a informação desse ficheiro e nos dá um apontador para uma estrutura de **Figura** e adicionamos essa mesma ao **grupo**.

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

while (pElement != nullptr){

const char\* cenas = pElement->Attribute("file");

if (cenas == nullptr) return;

string fich(cenas, strlen(cenas));

Figura\* f = extraiFicheiro(fich);

g->adicionaFigura(f);

pElement = pElement->NextSiblingElement("model");

}

### Extrair informação do ficheiro para a figura

Para extrair a informação do ficheiro **3D** previamente criado, apenas necessitamos de obter o inteiro que se encontra na primeira linha, correspondente ao número de vértices (e linhas) que estão nesse mesmo ficheiro. Depois iteramos cada linha, até chegarmos ao número obtido (que corresponde ao final do ficheiro) e para cada linha obtemos 3 valores de vírgula flutuante, correspondentes ao x,y e z de cada vértice, respetivamente. Após obtermos esses 3 pontos criávamos um novo objecto da classe **Ponto** e adicionávamos esse objeto à **Figura** correspondente. No final retornávamos o apontador para a **Figura**.

int count;

string line;

getline(inputFileStream, line);

count = stoi(line);

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

getline(inputFileStream, line);

stringstream ss(line);

vector<float> numbers;

for(int k = 0; k < 3; k++) {

string aux;

ss >> aux;

float j = stof(aux);

numbers.push\_back(j);

}

Ponto \*p = new Ponto(numbers.at(0),numbers.at(1),numbers.at(2));

figura->adicionaPonto(p);

}