

Computação Gráfica: Etapa#2

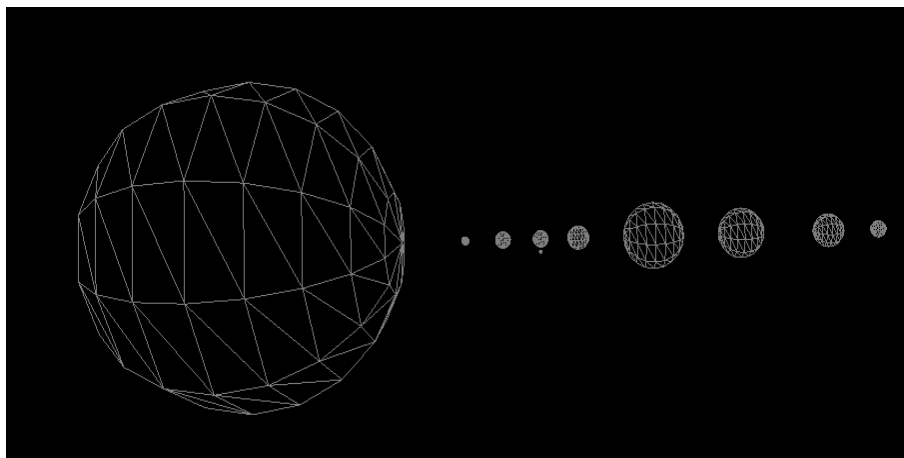
Bernardo Rodrigues
a79008@alunos.uminho.pt

César Silva
a77518@alunos.uminho.pt

Pedro Faria
a82725@alunos.uminho.pt

Rui Silva
a77219@alunos.uminho.pt

Universidade do Minho — 18 de Março de 2019



Resumo

A **Computação Gráfica**, uma das muitas áreas da informática, que assume um papel fulcral em interações homem-máquina e na visualização de dados. o seu espectro de aplicações varia desde geração de imagens até a simulação do mundo real.

O presente relatório refere-se às diferentes fases de entrega da componente prática da Unidade Curricular de **Computação Gráfica** enquadrada no curso de *Ciências da Computação* da *Universidade do Minho*.

Conteúdo

1	Introdução	4
2	Primeira Fase	5
2.1	Gerador	5
2.1.1	Caixa	5
2.1.2	Cone	5
2.1.3	Esfera	7
2.1.4	Plano	8
2.2	Motor	9
3	Segunda Fase	11
3.1	A classe Scene Graph	11
3.2	Motor	13
4	Conclusão	17
A	Código do Gerador	18
B	Código do Motor	23
C	Código do ficheiro SceneGraph.h	26
D	Código do SceneGraph.cpp	28
E	Código da Main	31

1 Introdução

Na primeira fase, foram desenvolvidas duas aplicações. Um **Gerador** de modelos, que aceite argumentos a partir do terminal, com a função de gerar os pontos de uma primitiva gráfica desejada pelo utilizador, imprimindo estes para um ficheiro. E a última, um **Motor** que interpreta os pontos gerados anteriormente de acordo com um ficheiro de configuração dado. Com base nos tópicos enunciados e ferramentas exploradas nas aulas implementamos o **Gerador** e o **Motor** na linguagem **C++**. Em particular esta última faz uso de biblioteca **TinyXML2** para que a leitura dos documentos que lhe são dados com input seja feita de forma simples e consistente. E por fim, utilizamos a API fornecida pelo **OpenGL** para dar vida aos nossos modelos.

Na segunda, adicionamos features ao parser de ficheiros de configuração de **scenes**, nomeadamente, o reconhecimento de **scenes** dispostas hierarquicamente usando transformações geométricas (translações, rotações e de escala).

2 Primeira Fase

2.1 Gerador

O nosso **Gerador** é um pequeno programa em C++, cuja implementação é disponibilizada em anexo, que depois de compilado, o correspondente executável escreve para um ficheiro (um por linha) os vértices da primitiva gráfica desejada como iremos ilustrar nas seguintes secções.

2.1.1 Caixa

Para geração desta primitiva o utilizador deverá invocar a aplicação a com o nome do executável seguido dos comprimentos dos lados de um paralelepípedo no eixos com X's, Y's e Z's respetivamente e por fim o nome do ficheiro destino. A sintaxe é demonstrada no exemplo abaixo:

Command Line

```
$ g++ -o gerador gen.cpp
$ ./gerador caixa 3 4 5 osmeusvertices
$ ls
$ gen.cpp          gerador          osmeusvertices.txt
```

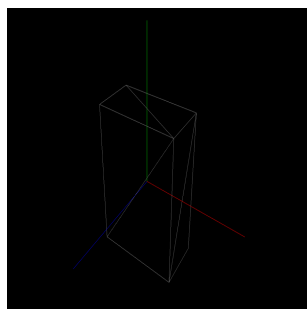


Figura 1: Caixa



Notice: Todas as faces são desenhadas com dois triângulos apontados para o exterior pela norma da mão direita mencionada nas aulas. Um zoom excessivo, ou a escolha de dimensões superiores à distância da câmara à origem(onde este é centrado) pode levar ao aparente desaparecimento do modelo.

2.1.2 Cone

O **Cone** recebe como argumentos o raio da base, a sua altura, o número de slices e stacks.

A construção deste começa por fixar um uma *slice* calculando os vértices da base correspondente, de seguida todas as *stacks* relativas, sendo o última stack - a do bico - um caso especial.

O algoritmo usa noções como semelhança de triângulos para cálculo dos sucessivos raios das circunferências formadas pelas *stacks*.



Info: Apresentamos o significado das variáveis usadas no programa que gera os pontos do **Cone**:

stk - diferença entre stacks consecutivas

slcd - diferença entre slices consecutivas

raiod - diferença entre o raio de duas stacks consecutivas

stk - stack atual

slc - slice atual

nslc - próxima slice

nstk - próxima stack

nr - próximo raio

r - raio atual

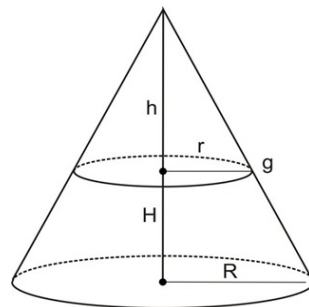
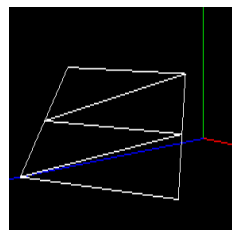
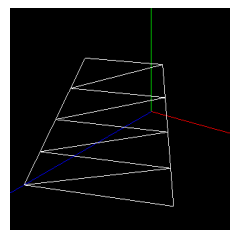


Figura 2: Semelhança de triângulos num cone

Apresentamos algumas imagens que ilustram a criação do cone.

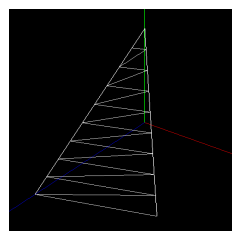


(a) 2 stacks

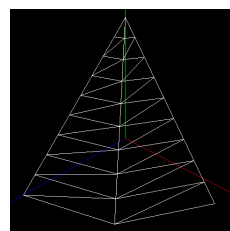


(b) 4 stacks

Figura 3: Progressão das stacks do cone



(a) 1 slice



(b) 2 slices

Figura 4: Progressão das slices do cone

Finalmente obtemos:

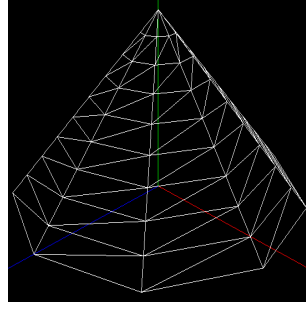


Figura 5: Cone

2.1.3 Esfera

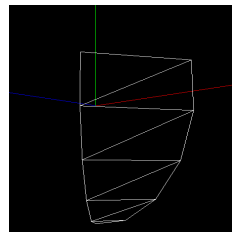
A **Esfera** recebe como argumentos o seu raio, o número de slices e stacks. A sua construção usa coordenadas esféricas usando o raio dado como argumentos e manipulando 2 ângulos *Alfa* e *Beta*. *Alfa* é dado por:

$$alfa = \frac{2 \times \Pi}{slices}$$

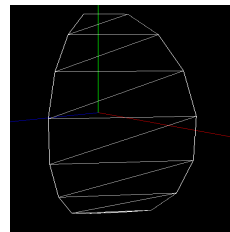
Este nos programas é usado juntamente com o raio para calcular os pontos de *slices* consecutivas. De seguida *beta* é dado por:

$$beta = \frac{\Pi \div 2}{stacks}$$

Este desempenha uma função igual ao anterior considerando *stacks*.

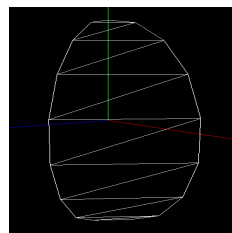


(a) 4 stacks

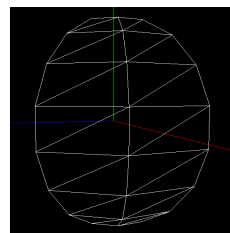


(b) 8 stacks

Figura 6: Progressão das stacks da esfera



(a) 1 slice



(b) 2 slices

Figura 7: Progressão das slices da esfera

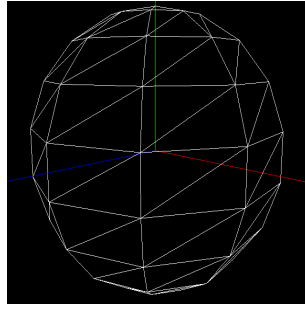


Figura 8: Esfera

2.1.4 Plano

O requisito estabelecido no guião do trabalho veio em muito simplificar a representação do plano XZ ao ponto de para a sua computação seja apenas necessário um único argumento que representa o tamanho da porção visível desejada.

Command Line

```
$ g++ -o gerador gen.cpp
$ ./gerador plano 5 pontos
$ ls
$ gen.cpp          gerador          pontos.txt
```

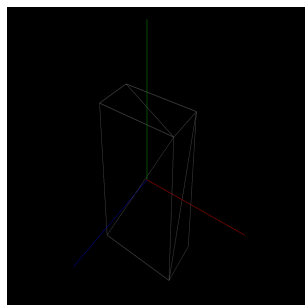


Figura 9: Caixa



Notice: Os planos são infinitos a referência ao tamanho é feita apenas para facilitar a observação do mesmo. São desenhados 4 triângulos, em vez de dois, para que esta primitiva seja visível de todas as perspectivas.

2.2 Motor

O motor é a parte do nosso trabalho que faz o linking de todas as outras partes que foram realizadas. O motor, lê um ficheiro xml (conf.xml). Este ficheiro contém uma estrutura bastante simples, dividida em scenes.

```
conf.xml

<scene>
    <model file="esfera.txt" />
    <model file="cone.txt" />
    <model file="caixa.txt" />
</scene>
```

Cada ficheiro que está referenciado nas tags **<model>** é um ficheiro de texto criado pelo nosso gerador e contém todos os pontos das figuras que queremos representar. Como o motor lê os ficheiros de pontos a partir dum ficheiro **XML** utilizamos um parser xml para **C++** chamado **tinysql-2**.

```
main.cpp

...
tinysql2::XMLDocument doc;

doc.LoadFile("./conf.xml");

tinysql2::XMLNode *scene = doc.FirstChild();

tinysql2::XMLElement* model;

while(scene) {
    for(model = scene->FirstChildElement();
        model != NULL;
        model = model->NextSiblingElement()) {
        const char * file;
        file = model->Attribute("file");
        guardaPontos(file);
    }
    scene = scene->NextSiblingElement();
}
...
```

Com este snippet de código, criamos um objeto do tipo **XMLDocument**. De seguida, com a função **LoadFile**, abrimos o ficheiro de configuração **conf.xml**, e começamos a manipular o seu conteúdo. Criamos um objeto do tipo **XMLNode** e associamos-lhe a primeira **tag** do ficheiro **conf.xml** que é a raiz da estrutura do nosso ficheiro. No ciclo **while**, percorremos todos o **ChildElements** de **scene**, que são as tags que guardam os nossos ficheiros das figuras. Cada ficheiro de figura tirado dos **models** é passado à função **guardaPontos**, que será explicada de seguida.

main.cpp

```
...
struct Pontos {
    float a;
    float b;
    float c;
};

std::vector<Pontos> pontos;

void guardaPontos(std::string ficheiro) {
    std::ifstream file;
    std::string s = "./";
    s.append(ficheiro.c_str());
    file.open(s.c_str());
    float a,b,c;
    while(file >> a >> b >> c) {
        Pontos aux;
        aux.a = a;
        aux.b = b;
        aux.c = c;
        pontos.push_back(aux);
    }
}
...
```

Criamos uma estrutura **Pontos** que tem como campos 3 *floats*, que servem para registar as coordenadas destes. De seguida usamos um vector que utiliza a estrutura enunciada para os guardar. À função **guardaPontos** são passados nomes de ficheiros. A função abre os ficheiros e lê pontos linha a linha, guardando cada coordenada *x*, *y* e *z* num **Ponto** e de seguida inserindo-o no vector.



Info: A instrução:

```
file >> a >> b >> c
```

associa a cada uma das variáveis *a*, *b* e *c*, as coordenadas da linha que está a ser lida em cada iteração do ciclo.

Por ultimo temos a função **printPontos**:

main.cpp

```
...
void printPontos(std::vector<Pontos> pontos) {
    for(int i = 0; i < pontos.size(); i++) {
        glVertex3f(pontos[i].a,
                   pontos[i].b,
                   pontos[i].c);
    }
}
...
```

Esta função é chamada na `renderScene` e gera todos os pontos percorrendo o vector.

3 Segunda Fase

Após ponderarmos o enunciado, o grupo, decidiu implementar uma classe, em C++, inspirada numa estrutura de dados largamente conhecida na área, denominada por **Scene Graph**.

A origem desta estrutura de dados remonta aos primórdios dos primeiros jogos de vídeo sobre simulação de voo mas actualmente é vulgarmente incluída qualquer aplicação que lide com **Graphic Rendering**.

Esta estrutura de dados foi revolucionária pelo facto de reduzir drasticamente a memória necessária em cálculos sistemáticos sobre o mundo que está a tentar representar. Os cálculos passaram a ser feitos **in place** na estrutura dispensando na totalidade a necessidade de memória auxiliar.

3.1 A classe Scene Graph

Apesar de disponibilizarmos a implementação em anexo iremos contemplar alguns pormenores neste capítulo.

Face a como os ficheiros **XML** são organizados via uma *árvore - DOM Tree* - implementamos uma classe que usa os princípios de essa mesma estrutura para guardar os dados de tudo o que é exposto na cena.

Por exemplo se quiséssemos desenhar um cavalo e o seu cavaleiro, não de maneira independente, mas como se o cavalo fosse uma extensão do seu cavaleiro. O **Scene Graph** correspondente teria no nodo **Cavalo** “pendurado” no nodo **Cavaleiro**.

Cada nodo, é um **Group** e nele guardamos as transformações geométricas que a ela lhe dizem respeito assim como um vector de pontos do **model** que queremos desenhar, e por fim, um array de outros **Scene Graphs** que representam o próximo nível de descendentes.

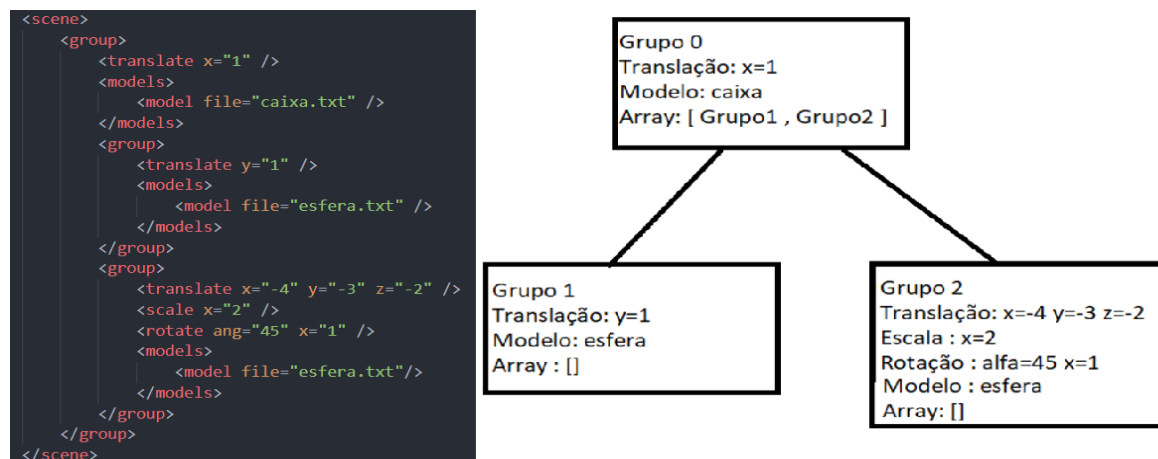
```
main.cpp

class SceneGraph {

    array<float, 3> scale;
    array<float, 3> trans;
    array<float, 4> rot;
    vector<vector<Ponto>> modelos;
    vector<SceneGraph> filhos;

}
```

Estamos então perante a definição de uma árvore de **Scene Graphs**. Para facilitar a visualização deste conceito dispomos o seguinte exemplo.





Info:

- Todos os nodos filhos herdam as transformações dos pais.
- Cada nodo pode ter um qualquer número de filhos, mas apenas um pai, i.e., não existe herança múltipla.
- Da travessia desde a raiz, até a uma folha, resultam todas as dependências que um certo objecto tem na **scene** em questão.

Fixada a estrutura e comportamento do classe. Basta, agora, expor o algoritmo de travessia que é efectuada quando a nossa modesta aplicação lê um ficheiro de configuração. O resto da **API** é disponibilizada em anexo.

main.cpp

```
void SceneGraph::draw() const {

    glPushMatrix();

    glScalef(scale[0], scale[1], scale[2]);
    glRotatef(rot[0], rot[1], rot[2], rot[3]);
    glTranslatef(trans[0], trans[1], trans[2]);

    glBegin(GL_TRIANGLES);

    for( vector<Pontos> const &pnts :
        this->modelos ) {
        for( Pontos const &p : pnts ) {
            glVertex3f(p.a, p.b, p.c);
        }
    }

    glEnd();

    for( SceneGraph const &tmp :
        this->filhos ) {
        tmp.draw();
    }

    glPopMatrix();

}
```



Info: Com base nos conteúdos abordados nas aulas teóricas decidimos adotar a convenção de ordenar as transformações geométricas **glRotatef()**, **glTranslatef()** e **glScalef()** pela sequência exposta no código acima.

3.2 Motor

De maneira semelhante à primeira fase, o motor continua a fazer leituras de um ficheiro xml (conf.xml). Por outro lado, este ficheiro apresenta uma complexidade superior ao anterior, contendo groups, models e ainda tags de translação (<translate>), rotação (<rotate>) e mudança de escala (<scale>).

```
conf.xml

<scene>
  <group>
    <translate x="1" />
    <models>
      <model file="caixa.txt" />
    </models>
  </group>
  <group>
    <translate y="1" />
    <models>
      <model file="esfera.txt" />
    </models>
  </group>
  <group>
    <translate x="-4" y="-3" z="-2" />
    <scale x="2" />
    <rotate ang="45" x="1" />
    <models>
      <model file="esfera.txt"/>
    </models>
  </group>
</scene>
```

Cada ficheiro que está referenciado nas tags <model> é um ficheiro de texto criado pelo nosso gerador e contém todos os pontos das figuras que queremos representar. Como o motor lê os ficheiros de pontos a partir dum ficheiro XML utilizamos um parser xml para C++ chamado **tinyxml-2**.

```
main.cpp

...
tinyxml2::XMLDocument doc;
//new file
doc.LoadFile("./conf.xml");
tinyxml2::XMLNode *scene = doc.FirstChild();
if (scene == nullptr) perror("Erro de Leitura.\n");

s_gg = doGroup(scene->FirstChildElement("group"));
...
```

Com este snippet de código, criamos um objeto do tipo XMLDocument. De seguida, com a função **Load-File**, abrimos o ficheiro de configuração **conf.xml**. Criamos um objeto do tipo **XMLNode** e associamos-lhe a primeira **tag** do ficheiro conf.xml que é a raiz da estrutura do nosso ficheiro. Através da função doGroup, começamos a manipular o seu conteúdo.

engine.cpp

```
SceneGraph doGroup(tinyxml2::XMLElement* group) {
    SceneGraph s_g;
    tinyxml2::XMLElement* novo =
        group->FirstChildElement();
    for(novo; novo != NULL;
        novo = novo->NextSiblingElement()) {
        //printf("%s\n", novo->Name());
        if(!strcmp(novo->Name(), "group")) {
            s_g.addFilho(doGroup(novo));
        } else if(!strcmp(novo->Name(), "models")) {
            s_g.setModelo(doModels(novo));
        } else if(!strcmp(novo->Name(),
            "translate")) {
            s_g.setTrans(doTranslate(novo));
        } else if(!strcmp(novo->Name(), "rotate")) {
            s_g.setRot(doRotate(novo));
        } else if(!strcmp(novo->Name(), "scale")) {
            s_g.setScale(doScale(novo));
        } else {
            perror("Formato XML Incorreto.\n");
        }
    }
    return s_g;
}
```

No ciclo *for*, percorremos todos o ChildElements da group, que são as tags que guardam os nossos ficheiros das figuras. Dentro de cada tag **group** pode haver outra tag **group**, ativando a recursão da função doGroup. Mas, há outras tags que podem aparecer, tais como **models** <translate>, <rotate> e <scale>. Caso a tag lida seja <translate>, a função doGroup evoca a função doTranslate:

engine.cpp

```
array<float,3> doTranslate(
    tinyxml2::XMLElement* translate) {

    array<float, 3> trans;

    const char * x;
    const char * y;
    const char * z;
    x = translate->Attribute("x");
    y = translate->Attribute("y");
    z = translate->Attribute("z");

    x == nullptr ? trans[0] = 0 : trans[0] = atof(x);
    y == nullptr ? trans[1] = 0 : trans[1] = atof(y);
    z == nullptr ? trans[2] = 0 : trans[2] = atof(z);

    return trans;
}
```

Caso a tag lida seja <rotate>, a função doGroup evoca a função doRotate:

engine.cpp

```
array<float,4> doRotate(tinyxml2::XMLElement* rotate) {  
  
    array<float,4> rot;  
  
    const char * x;  
    const char * y;  
    const char * z;  
    const char * ang;  
    x = rotate->Attribute("x");  
    y = rotate->Attribute("y");  
    z = rotate->Attribute("z");  
    ang = rotate->Attribute("angle");  
  
    ang == nullptr ? rot[0] = 0 : rot[0] = atoi(ang);  
    x == nullptr ? rot[1] = 0 : rot[1] = atof(x);  
    y == nullptr ? rot[2] = 0 : rot[2] = atof(y);  
    z == nullptr ? rot[3] = 0 : rot[3] = atof(z);  
  
    return rot;  
}
```

Caso a tag lida seja **<scale>**, a função doGroup evoca a função doScale:

engine.cpp

```
array<float,3> doScale(tinyxml2::XMLElement* scale) {  
  
    array<float,3> sca;  
  
    const char * x;  
    const char * y;  
    const char * z;  
  
    x = scale->Attribute("x");  
    y = scale->Attribute("y");  
    z = scale->Attribute("z");  
  
    x == nullptr ? sca[0] = 1 : sca[0] = atof(x);  
    y == nullptr ? sca[1] = 1 : sca[1] = atof(y);  
    z == nullptr ? sca[2] = 1 : sca[2] = atof(z);  
  
    return sca;  
}
```

Qualquer uma destas 3 funções, retorna um array com os argumentos que serão passados às funções Glut que executarão as transformações: a função doTranslate, retorna um array com 3 argumentos para a função **glTranslate**, a função doRotate, retorna um array com 4 argumentos para a função **glRotate** e a função doScale, retorna um array com 3 argumentos para a função **glScale**.

Por outro lado, ainda pode aparecer uma tag **<models>**, ou seja é chamada a função doModels que significa que, dentro dessas tags, vai haver uma tag **<model file= "nome do ficheiro.txt">**, ficheiro este que tem todos os pontos gerados pelo gerador.

engine.cpp

```
std::vector<std::vector<Pontos>>
doModels(tinyxml2::XMLElement* models) {
    std::vector<std::vector<Pontos>> pPontos;
    tinyxml2::XMLElement* novo =
        models->FirstChildElement();
    for(novo; novo != NULL;
        novo = novo->NextSiblingElement()) {
        const char * file;
        file = novo->Attribute("file");
        pPontos.push_back(guardaPontos(file));
    }
    return pPontos;
}
```

A função `doModels` chama a função **guardaPontos** que, usando a estrutura **Pontos**, regista as coordenadas de cada ponto presente no ficheiro. A estrutura tem campos 3 *floats*, que servem para guardar as coordenadas dos pontos. De seguida usamos um vector que utiliza a estrutura enunciada para os guardar. A função abre os ficheiros e lê pontos linha a linha, guardando cada coordenada *x*, *y* e *z* num **Ponto** e de seguida inserindo-o no vector.

engine.cpp

```
...
struct Pontos {
    float a;
    float b;
    float c;
};

std::vector<Pontos> pontos;

void guardaPontos(std::string ficheiro) {
    std::ifstream file;
    std::string s = "./";
    s.append(ficheiro.c_str());
    file.open(s.c_str());
    float a,b,c;
    while(file >> a >> b >> c) {
        Pontos aux;
        aux.a = a;
        aux.b = b;
        aux.c = c;
        pontos.push_back(aux);
    }
}
...
```

Este array de pontos que a `guardaPontos` retorna, preenche o array `pPontos` instanciado na `doModels`, que por sua vez é retornado por esta função. Na função `doGroup`, uma `SceneGraph` é passada como objeto aos Setters definidos na estrutura `SceneGraph`, que chamam as funções acima faladas que, retornando os arrays de argumentos e os pontos das figuras, fazem uma atualização da estrutura, que na **main.cpp** é passada à `renderScene`.

4 Conclusão

Também nos ajudou a possuir mais discernimento sobre a geometria e cálculos matemáticos por detrás de todo um esquema geométrico em 3 dimensões.

Durante a realização dos geradores conseguimos perceber que a propagação dos erros nos cálculos dos ângulos pode ter impacto na apresentação das figuras geométricas.

Apresentou-nos também mais uma oportunidade de aprender e melhorar as nossas capacidades de programação em C++, no uso de LaTeX e de ficheiros XML.

Todas as aptidões aqui aprendidas e/ou desenvolvidas, não só a nível escolar mas como a nível de cooperação e de trabalho de equipa. vão-nos permitir uma melhor realização de projetos futuros.

Esta secção poderá ser modificada ao longo das fases de entrega.

A Código do Gerador

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#define cp1(r, a)      (r * sin(a))
#define cp2(r, a)      (r * cos(a))

#define ce1(r, a, b)   (r * cos(b) * sin(a))
#define ce2(r, b)      (r * sin(b))
#define ce3(r, a, b)   (r * cos(b) * cos(a))

void printSphere(float radius, int slices, int stacks, FILE* f){
    // variaveis que vem o numero da stack atual
    float stkd, slcd;
    float slc, stk, nslc, nstk;

    int j;

    // delta dos angulos por stack
    stkd = M_PI / stacks;

    // delta dos angulos por slice
    slcd = 2 * M_PI / slices;

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

        j = 1;
        slc = i * slcd;
        nslc = (i+1) * slcd;
        stk = -M_PI_2 + j * stkd;

        fprintf(f,"%f%f%f\n",0.0, -radius, 0.0);
        fprintf(f,"%f%f%f\n",ce1(radius, nslc, stk), ce2(radius, stk), ce3(radius, radius, nslc, stk));
        fprintf(f,"%f%f%f\n",ce1(radius, slc, stk), ce2(radius, stk), ce3(radius, radius, slc, stk));

        for(; j < stacks-1; ++j) {

            slc = i * slcd;
            nslc = (i+1) * slcd;
            stk = -M_PI_2 + j * stkd;
            nstk = -M_PI_2 + (j+1) * stkd;

            fprintf(f,"%f%f%f\n",ce1(radius, nslc, nstk), ce2(radius, nstk), ce3(radius, radius, nslc, nstk));
            fprintf(f,"%f%f%f\n",ce1(radius, slc, stk), ce2(radius, stk), ce3(radius, radius, slc, stk));
            fprintf(f,"%f%f%f\n",ce1(radius, nslc, stk), ce2(radius, stk), ce3(radius, radius, nslc, stk));

            fprintf(f,"%f%f%f\n",ce1(radius, slc, nstk), ce2(radius, nstk), ce3(radius, radius, slc, nstk));
        }
    }
}
```

```

        fprintf(f,"%f%f%f\n",ce1(radius, slc, stk), ce2(radius, stk), ce3(radius, stk));
        fprintf(f,"%f%f%f\n",ce1(radius, nslc, nstk), ce2(radius, nstk), ce3(radius, nstk));

    }

    slc = i * slcd;
    nslc = (i+1) * slcd;
    stk = -M_PI / 2 + j * stkd;

    fprintf(f,"%f%f%f\n",0.0, radius, 0.0);
    fprintf(f,"%f%f%f\n",ce1(radius, slc, stk), ce2(radius, stk), ce3(radius, stk));
    fprintf(f,"%f%f%f\n",ce1(radius, nslc, stk), ce2(radius, stk), ce3(radius, stk));

}

}

void printBox ( float xx , float yy , float zz , FILE *f ){
    // front e back
    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,zz/2 );
    fprintf(f,"%f%f%f\n",xx/2,-yy/2,zz/2);
    fprintf(f,"%f%f%f\n",xx/2,yy/2,zz/2);

    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,zz/2);
    fprintf(f,"%f%f%f\n",xx/2,yy/2,zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,yy/2,zz/2);

    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",xx/2,yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",xx/2,-yy/2,-zz/2);

    fprintf(f,"%f%f%f\n",xx/2,yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,yy/2,-zz/2);
    // top e bot
    fprintf(f,"%f%f%f\n",xx/2,yy/2,zz/2);
    fprintf(f,"%f%f%f\n",xx/2,yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,yy/2,-zz/2);

    fprintf(f,"%f%f%f\n",xx/2,yy/2,zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",xx/2,-yy/2,zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",xx/2,-yy/2,-zz/2);

    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,-zz/2);
    fprintf(f,"%f%f%f\n",xx/2,-yy/2,zz/2);
    fprintf(f,"%f%f%f\n",-xx/2,-yy/2,zz/2);
    //sides

```

```

fprintf(f,"%f%f%f\n",xx/2,-yy/2,zz/2);
fprintf(f,"%f%f%f\n",xx/2,-yy/2,-zz/2);
fprintf(f,"%f%f%f\n",xx/2,yy/2,-zz/2);

fprintf(f,"%f%f%f\n",xx/2,yy/2,-zz/2);
fprintf(f,"%f%f%f\n",xx/2,yy/2,zz/2);
fprintf(f,"%f%f%f\n",xx/2,-yy/2,zz/2);

fprintf(f,"%f%f%f\n",-xx/2,-yy/2,zz/2);
fprintf(f,"%f%f%f\n",-xx/2,yy/2,-zz/2);
fprintf(f,"%f%f%f\n",-xx/2,-yy/2,-zz/2);

fprintf(f,"%f%f%f\n",-xx/2,-yy/2,zz/2);
fprintf(f,"%f%f%f\n",-xx/2,yy/2,zz/2);
fprintf(f,"%f%f%f\n",-xx/2,yy/2,-zz/2);
}

void printCone(float radius, float altura, int slices, int stacks, FILE *f) {

    float stkd, slcd, raiod;
    float stk, slc, nslc, nstk, nr, r;

    stkd = altura / stacks;
    slcd = 2 * M_PI / slices;
    raiod = radius / stacks;

    int j;

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

        // codigo responsavel por gerar uma slice da base

        slc = i * slcd;
        nslc = (i+1) * slcd;

        fprintf(f,"%f%f%f\n",0.0, 0.0, 0.0);
        fprintf(f,"%f%f%f\n",cp1(radius, nslc), 0.0, cp2(radius, nslc));
        fprintf(f,"%f%f%f\n",cp1(radius, slc), 0.0, cp2(radius, slc));

        // codigo responsavel por gerar as slices laterais
        for(j = stacks ; j > 1; j--) {

            slc = i * slcd;
            nslc = (i+1) * slcd;
            stk = (stacks - j) * stkd;
            nstk = (stacks - (j-1)) * stkd;
            r = j * raiod;
            nr = (j - 1) * raiod;

            fprintf(f,"%f%f%f\n",cp1(nr, nslc), nstk, cp2(nr, nslc));
            fprintf(f,"%f%f%f\n",cp1(r, slc), stk, cp2(r, slc));
            fprintf(f,"%f%f%f\n",cp1(r, nslc), stk, cp2(r, nslc));
        }
    }
}

```

```

        fprintf(f,"%f_uf_uf_uf_\n",cp1(nr, slc), nstk, cp2(nr, slc));
        fprintf(f,"%f_uf_uf_uf_\n",cp1(r, slc), stk, cp2(r, slc));
        fprintf(f,"%f_uf_uf_uf_\n",cp1(nr, nslc), nstk, cp2(nr, nslc));

    }

    // codigo responsavel por gerar a slice do topo
    slc = i * slcd;
    nslc = (i+1) * slcd;
    stk = (stacks - j) * stkd;
    r = j * raiod;

    fprintf(f,"%f_uf_uf_uf_\n",0.0, altura, 0.0);
    fprintf(f,"%f_uf_uf_uf_\n",cp1(r, slc), stk, cp2(r, slc));
    fprintf(f,"%f_uf_uf_uf_\n",cp1(r, nslc), stk, cp2(r, nslc));

}

}

void printPlano(float tam,FILE *f){

    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,-tam/2);

    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,-tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,tam/2);

    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,-tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,-tam/2);

    fprintf(f,"%f_uf_uf_uf_\n",tam/2,0.0,-tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,-tam/2);
    fprintf(f,"%f_uf_uf_uf_\n",-tam/2,0.0,tam/2);

}

int main( int i ,char *args[] ) {

    if ( !strcmp("cone", args[1]) ){
        FILE *f = fopen( args[6] ,"w");
        printCone( atof(args[2]), atof(args[3]), atof(args[4]), atof(args[5]), f);
        fclose(f);
    }
    else if ( !strcmp("caixa", args[1]) ){
        FILE *f = fopen( args[5] ,"w");
        printBox( atof(args[2]), atof(args[3]), atof(args[4]), f);
    }
}

```

```

        fclose(f);
    }
    else if ( !strcmp("esfera", args[1]) ){
        FILE *f = fopen( args[5], "w");
        printSphere( atof( args[2] ), atof( args[3] ), atof( args[4] ), f);
        fclose(f);
    }
    else if ( !strcmp("plano", args[1]) ){
        FILE *f = fopen( args[3], "w");
        printPlano( atof( args[2] ), f );
        fclose(f);
    }

    return 0;
}

```

B Código do Motor

```
#include "sg.h"
#include "tinyxml2.h"
#include <fstream>
#include <vector>

vector<Pontos> guardaPontos(std::string ficheiro) {

    std::vector<Pontos> pontos;

    std::ifstream file;
    //change this to your folder's path.
    std::string s = "./";
    s.append(ficheiro.c_str());
    file.open(s.c_str());
    float a,b,c;
    while(file >> a >> b >> c) {
        Pontos aux;
        aux.a = a;
        aux.b = b;
        aux.c = c;
        pontos.push_back(aux);
    }
    return pontos;
}

std::vector<std::vector<Pontos>> doModels(tinyxml2::XMLElement* models) {
    std::vector<std::vector<Pontos>> pPontos;
    tinyxml2::XMLElement* novo = models->FirstChildElement();
    for(novo; novo != NULL; novo = novo->NextSiblingElement()) {
        const char * file;
        file = novo->Attribute("file");
        pPontos.push_back(guardaPontos(file));
    }
    return pPontos;
}

array<float,3> doTranslate(tinyxml2::XMLElement* translate) {

    array<float, 3> trans;

    const char * x;
    const char * y;
    const char * z;
    x = translate->Attribute("x");
    y = translate->Attribute("y");
    z = translate->Attribute("z");

    x == nullptr ? trans[0] = 0 : trans[0] = atof(x);
    y == nullptr ? trans[1] = 0 : trans[1] = atof(y);
    z == nullptr ? trans[2] = 0 : trans[2] = atof(z);

    return trans;
}

array<float,4> doRotate(tinyxml2::XMLElement* rotate) {
```

```

    array<float,4> rot;

    const char * x;
    const char * y;
    const char * z;
    const char * ang;
    x = rotate->Attribute("x");
    y = rotate->Attribute("y");
    z = rotate->Attribute("z");
    ang = rotate->Attribute("angle");

    ang == nullptr ? rot[0] = 0 : rot[0] = atof(ang);
    x == nullptr ? rot[1] = 0 : rot[1] = atof(x);
    y == nullptr ? rot[2] = 0 : rot[2] = atof(y);
    z == nullptr ? rot[3] = 0 : rot[3] = atof(z);

    return rot;
}

array<float,3> doScale(tinyxml2::XMLElement* scale) {

    array<float,3> sca;

    const char * x;
    const char * y;
    const char * z;

    x = scale->Attribute("x");
    y = scale->Attribute("y");
    z = scale->Attribute("z");

    x == nullptr ? sca[0] = 1 : sca[0] = atof(x);
    y == nullptr ? sca[1] = 1 : sca[1] = atof(y);
    z == nullptr ? sca[2] = 1 : sca[2] = atof(z);

    return sca;
}

SceneGraph doGroup(tinyxml2::XMLElement* group) {
    SceneGraph s_g;
    tinyxml2::XMLElement* novo = group->FirstChildElement();
    for(novo; novo != NULL; novo = novo->NextSiblingElement()) {
        //printf("%s\n", novo->Name());
        if(!strcmp(novo->Name(), "group")) {
            s_g.addFilho(doGroup(novo));
        } else if(!strcmp(novo->Name(), "models")) {
            s_g.setModelo(doModels(novo));
        } else if(!strcmp(novo->Name(), "translate")) {
            s_g.setTrans(doTranslate(novo));
        } else if(!strcmp(novo->Name(), "rotate")) {
            s_g.setRot(doRotate(novo));
        } else if(!strcmp(novo->Name(), "scale")) {
            s_g.setScale(doScale(novo));
        } else {
            perror("Formato XML Incorreto.\n");
        }
    }
}

```



```
    }  
    return s_g;  
}
```

C Código do ficheiro SceneGraph.h

```
#include <vector>
#include <string>
#include <array>
#include <GL/glut.h>

#ifndef __ESTRUTURAS__
#define __ESTRUTURAS__

using namespace std;

/*
 * Estrutura usada para guardar as coordenadas de um ponto
 * num espaco 3D
 */
struct ponto {
    float a;
    float b;
    float c;
};

typedef struct ponto Pontos;

/*
 * Class principal para codificacao de um SceneGraph basico
 * para os arrays que codificam certas rotacoes, etc a posicao 0
 * simboliza a coordenada x, a posicao 1 codifica a coordenada y, etc.
 * No caso das rotacoes a posicao 0 codifica o angulo
 * e de seguida sao dados os eixos de rotacao.
 *
 * A funcao de desenho considera escalas primeiro, rotacoes de seguida e
 * finalmente translacoes
 */
class SceneGraph {

    // variaveis
    array<float, 3> scale;
    array<float, 3> trans;
    array<float, 4> rot;
    vector<vector<Pontos>> modelos;
    vector<SceneGraph> filhos;

public:
    // Construtores
    SceneGraph();

    // Setters
    void setScale( array<float, 3> );
    void setTrans( array<float, 3> );
    void setRot( array<float, 4> );
    void setModelo( vector<vector<Pontos>> );

    // Getters
    array<float, 3> getScale();
    array<float, 3> getTrans();
    array<float, 4> getRot();
};
```

```

        vector<vector<Pontos>> getModelos();
        vector<SceneGraph> getFilhos();

        // Funcoes adicionais
        void addFilho( SceneGraph );
        void addModelo( vector<Pontos> );

        // Funcao responsavel por desenhar a estrutura
        void draw() const;
};

#endif

```

D Código do SceneGraph.cpp

```
#include "sg.h"

using namespace std;

// SceneGraph
// Construtor

SceneGraph::SceneGraph() {

    this->scale.fill(1.0f);
    this->trans.fill(0.0f);
    this->rot.fill(0.0f);

}

// Setters

void SceneGraph::setScale( array<float, 3> escala) {

    this->scale = escala;

}

void SceneGraph::setTrans( array<float, 3> transl ) {

    this->trans = transl;

}

void SceneGraph::setRot( array<float, 4> rota) {

    this->rot = rota;

}

void SceneGraph::setModelo(vector<vector<Pontos>> pontos) {

    this->modelos = pontos;

}

// Getters

array<float, 3> SceneGraph::getScale() {

    return this->scale;

}

array<float, 3> SceneGraph::getTrans() {

    return this->trans;

}
```

```

array<float, 4> SceneGraph::getRot() {
    return this->rot;
}

vector<vector<Pontos>> SceneGraph::getModelos() {
    return this->modelos;
}

vector<SceneGraph> SceneGraph::getFilhos() {
    return this->filhos;
}

// Funcoes Adicionais

void SceneGraph::addFilho( SceneGraph c ) {
    this->filhos.push_back( c );
}

void SceneGraph::addModelo( vector<Pontos> c ) {
    this->modelos.push_back( c );
}

// Funcao responsavel por desenhar a estrutura

void SceneGraph::draw() const {
    glPushMatrix();

    glScalef(scale[0], scale[1], scale[2]);
    glRotatef(rot[0], rot[1], rot[2], rot[3]);
    glTranslatef(trans[0], trans[1], trans[2]);

    glBegin(GL_TRIANGLES);

    for( vector<Pontos> const &pnts : this->modelos ) {
        for( Pontos const &p : pnts ) {
            glVertex3f(p.a, p.b, p.c);
        }
    }

    glEnd();

    for( SceneGraph const &tmp : this->filhos ) {
        tmp.draw();
    }

    glPopMatrix();
}

```

}

E Código da Main

```
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#define _USE_MATH_DEFINES
#include <math.h>

#include <iostream>
#include "../Deps/tinyxml2.h"
#include <string>
#include <vector>
#include <fstream>
#include <list>
#include "../Deps/sg.h"
#include "../Deps/engine.h"

using namespace tinyxml2;

SceneGraph s_gg;

#define GROWF 0.01

double alfa = M_PI / 4;
double beta = M_PI / 4;
float raio = 350.0f;

void changeSize(int w, int h) {

    // Prevent a divide by zero, when window is too short
    // (you cant make a window with zero width).
    if(h == 0)
        h = 1;

    // compute window's aspect ratio
    float ratio = w * 1.0 / h;

    // Set the projection matrix as current
    glMatrixMode(GL_PROJECTION);
    // Load Identity Matrix
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set perspective
    gluPerspective(45.0f ,ratio, 1.0f ,1000.0f);

    // return to the model view matrix mode
    glMatrixMode(GL_MODELVIEW);
}
```

```

void renderScene(void) {

    // clear buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set the camera
    glLoadIdentity();
    gluLookAt(raio * cos(beta) * cos(alfa), raio * cos(beta) * sin(alfa), raio * sin(beta),
              0.0,75.0,0.0,
              0.0f,0.0f,1.0f);

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    glColor3f(0.5,0.5,0.5);
    s_gg.draw();

    // End of frame
    glutSwapBuffers();
}

void processSpecialKeys(int key, int xx, int yy) {

    // put code to process special keys in here
    switch(key) {
        case GLUT_KEY_LEFT:
            alfa += GROWF;
            break;
        case GLUT_KEY_RIGHT:
            alfa -= GROWF;
            break;
        case GLUT_KEY_UP:
            beta += GROWF;
            break;
        case GLUT_KEY_DOWN:
            beta -= GROWF;
            break;
        default:
            ;
    }

    glutPostRedisplay();

}

int main(int argc, char **argv) {

    tinyxml2::XMLDocument doc;
    //new file
    doc.LoadFile("./conf.xml");

```



```

        tinyxml2::XMLNode *scene = doc.FirstChild();
        if (scene == nullptr) perror("Erro de Leitura.\n");

        s_gg = doGroup(scene->FirstChildElement("group"));

// init GLUT and the window
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DEPTH|GLUT_DOUBLE|GLUT_RGBA);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(800,800);
        glutCreateWindow("CG@DI-UM");

// Required callback registry
        glutDisplayFunc(renderScene);
        glutReshapeFunc(changeSize);

// Callback registration for keyboard processing
        glutSpecialFunc(processSpecialKeys);

// OpenGL settings
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_CULL_FACE);

// enter GLUT's main cycle
        glutMainLoop();

        return 1;
}

```