Computação Gráfica

Trabalho Prático CG - 2017/18

Fase 2 - Transformações Geométricas

Relatório de Desenvolvimento

André Pereira (A79196)

Filipe Miranda (A78992)

José Alves (A78178)

8 de Abril de 2018

Conteúdo

Capítulo 1

Introdução

O trabalho prático **Transformações Geométricas** surge inserido na disciplina de Computação Gráfica e corresponde à segunda fase do trabalho prático desta unidade curricular. Versa sobre o uso do **TinyXML** e corresponde, essencialmente, em acrescentar ao parser a leitura de transformações geométricas e de hierarquias. Ao longo deste documento podemos encontrar a exposição do problema (a sua descrição informal, os dados do problema e o que foi pedido que concebessemos), o desenho da resolução do problema e alguns resultados/testes. Por fim, apresenta-se uma breve conclusão e o código do programa.

Estrutura do Relatório

Este relatório inicializa-se com uma breve introdução ao trabalho prático. No capítulo 2 é feita uma análise do problema proposto, onde se apresenta o enunciado do trabalho e a especificação dos seus requisitos. De seguida, são apresentadas as estratégias para a resolução do mesmo, onde no capítulo 4 são especificados os resultados obtidos. Finalizando, no capítulo 5 encontram-se as conclusões a que chegamos e algumas reflexões.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Esta fase do trabalho consiste na criação de cenários hierárquicos usando transformações geométricas. Um cenário é definido como uma árvore (neste caso, corresponde à árvore da estrutura de dados usada no código), onde cada nodo contém um conjunto de transformações geométricas (translações, rotações e escalas) e, opcionalmente, um conjunto de modelos. Cada nodo também pode conter nodos filhos.

2.2 Especificação dos Requisitos

As transformações geométricas apenas podem existir dentro de um grupo e são aplicadas a todos os modelos e subgrupos, onde a ordem é relevante (como já verificamos na primeira ficha de consolidação). O demo requisitado para esta fase correponde a um modelo estático do sistema solar, incluindo o sol, os planetas e as luas definidas numa hierarquia.

2.2.1 Exemplo

Exemplo de um ficheiro de configuração XML:

```
<scene>
    <group>
        <translate X=1 />
        <models>
            <model file="sphere.3d" />
        </models>
        <group>
            <translate Y=1 />
            <models>
                <model file="cone.3d" />
            </models>
        </group>
    </group>
</scene>
```

Capítulo 3

Concepção/Desenho da Resolução

3.1 Árvore de Classes

A resolução implementada para criar cenários hierárquicos consiste em converter um ficheiro de configuração XML numa árvore de classes. Um ficheiro de configuração é aceitável caso possua a seguinte estrutura:

- O primeiro elemento corresponde à tag *scene*;
- O segundo elemento corresponde à tag *group*, visto que as transformações geométricas apenas podem existir dentro de um grupo;
- Dentro do elemento *group*, encontram-se os subelementos que o constituem tais como os modelos, as transformações geométricas (translações, rotações e escalas) e possíveis grupos também;

Para cada um destes elementos é criada uma classe com o nome correspondente, onde se encontram as variáveis de instância e o método *apply()* que executa a devida transformação.

Elemento	Método Apply()	Variáveis de Instância
Model	glVertex3f(v1,v2,v3); *	string modelo;
Scale	glScalef(x,y,z);	float x, y, z;
Translate	glTranslatef(x,y,z);	float x, y, z;
Rotate	glRotatef(angle,x,y,z);	float x, y, z, angle;
Group	glPushMatrix();	————————

Figura 3.1: Variáveis de instância e método de cada elemento.

** Este método apply() deve-se ao facto de cada ficheiro modelo ser constituído por pontos que definem a figura. Por cada 3 pontos (v1,v2,v3) que encontra no ficheiro é desenhado um triângulo através do glVertex3f e o processo repete-se até ao fim do ficheiro.*

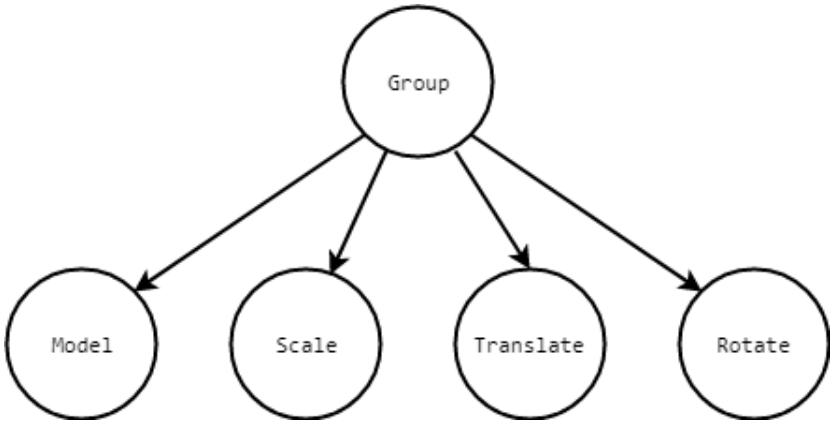

Figura 3.2: Generalização da árvore de classes.

3.2 Conversão de um Ficheiro XML para uma Árvore

Suponhamos que temos o ficheiro de configuração que se encontra em baixo. A árvore resultante será constituída por um nodo pai do tipo *group* e pelos nodos filhos. Neste exemplo os nodos filhos correspondem a uma figura box (ao encontrar uma tag do tipo *models* é gerado um nodo para cada modelo que aparecer), um grupo, uma translação e outro grupo. Por sua vez, os nodos filhos do tipo grupo também terão nodos filhos consoante os seus constituintes. Neste exemplo, em ambas as situações é feita uma translação e é desenhada uma esfera. Como se pode verificar cada um destes grupos herda as transformações geométricas do grupo pai. Como resultado, temos a árvore que se encontra na Figura 3.3.

```
<scene>
    <group>
        <models>
            <model file="box.3d" />
        </models>
        <group>
            <translate X=1 Y=0 Z=0 />
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <translate X=1 Y=1 Z=0 />
        <group>
            <translate X=-1 Y=0 Z=0 />
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
    </group>
</scene>
```

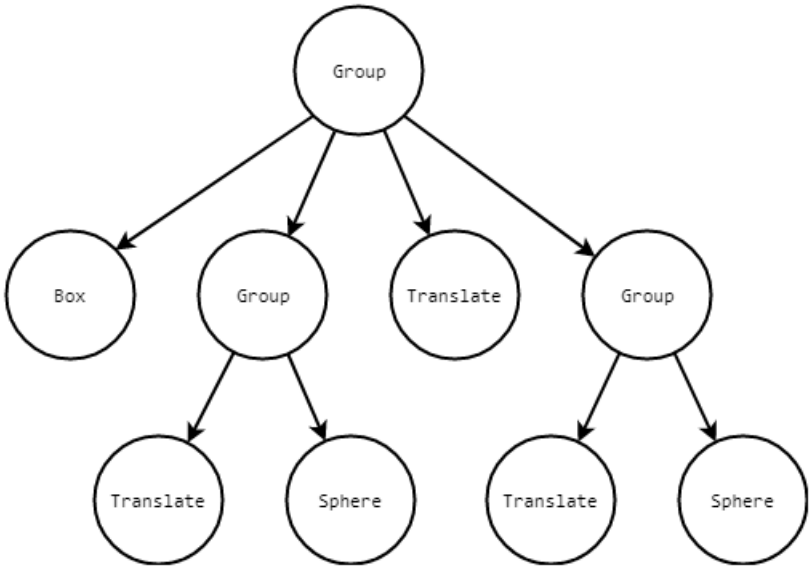

Figura 3.3: Árvore resultante do ficheiro XML.

Para criar a árvore, começa-se por abrir o ficheiro XML e verificar se a primeira tag corresponde a *scene*. É feito um strcmp com o valor da base e caso seja diferente de zero, avisa que o ficheiro é inválido e o programa é terminado. No caso de não ser, prossegue e averigua se o primeiro elemento filho corresponde a um *group* (obrigatoriamente tem que haver um), para que possa aceder aos seus elementos. No caso de não possuir elementos, termina o programa, dado que o ficheiro se encontra vazio. Sendo diferente de vazio, obtém a confirmação de que o elemento é do tipo *group* e começa a ler o ficheiro. Ao ler o ficheiro, percorre os elementos seguintes de maneira a encontrar mais tags que sejam reconhecíveis (modelos ou transformações geométricas). Se encontrar tags diferentes destas, o programa termina devido à configuração inválida do ficheiro. Ao percorrer os elementos filhos, no caso de encontrar um do tipo *group*, o nodo é criado, mas os filhos deste não são logo adicionados, dado que primeiro tem que se acabar de percorrer os outros filhos. Quando tal aconntece, é colocado o endereço do nodo numa stack auxiliar para depois adicionar os elementos desse grupo.

3.3 Travessia da Árvore

A árvore resultante consiste numa struct (mais precisamente, uma árvore n-ária), onde cada nodo é constituído por um vetor de filhos, a classe e o nome. Estando a árvore definida, a sua travessia é feita através do algoritmo da busca em profundidade (depth-first), facilitando a execução dos push e dos pop ao encontrar nodos do tipo *group*.

```
typedef struct node {
    Group* g;
    char* label;
    vector<struct node*> sons;
} *Arvore;
```

Voltando ao exemplo da Figura 3.3, ao percorrer a árvore, é desenhada uma box e é feito um push da matriz (dado que foi encontrado um nodo do tipo *group*), de maneira a ser feita uma translação e o desenho de uma esfera. Visto que foram percorridos todos os filho do grupo, é feito um pop da matriz para "ignorar" as transformações geométricas que foram lá implementadas. De seguida é feita uma translação e novamente um push da matriz visto que vai fazer a travessia de um grupo. Neste grupo, é feita uma translação e é desenhado um novo modelo (uma esfera). Aqui encontra-se um exemplo de herança das transformações geométricas do nodo pai, sendo aplicadas a todos os modelos e subgrupos. Acabando de percorrer os filhos do grupo, volta a fazer pop da matriz e de seguida é feito novamente visto que a travessia da árvore foi finalizada. Tendo a árvore criada, no renderscene apenas é preciso "processa" a árvore, e aplicar o método de cada classe presente em cada nodo da árvore. A árvore é percorrida em profundiade e, ao encontrar um nodo com a classe "translate" por exemplo, irá realizar um glTranslatef(x,y,z) no glut.

Capítulo 4

Codificação e Testes

4.1 Testes realizados e Resultados

O demo requisitado para esta fase correponde a um modelo estático do sistema solar, incluindo o sol, os planetas e as luas definidas numa hierarquia. Mostram-se a seguir alguns testes feitos e os respectivos resultados obtidos. Comecemos pelo ficheiro de configuração para o Sistema Solar:

```
<scene>
    <group>
        <models>
            Sol
            <model file="sphere.3d" />
        </models>
        <group>
            Mercurio
            <translate X=1.5236 Y=0 Z=0 />
            <scale X=0.02439 Y=0.024439 Z=0.02439/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Venus
            <translate X=2.0472 Y=0 Z=0 />
            <scale X=0.06051 Y=0.06051 Z=0.06051/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Terra
            <translate X=2.49 Y=0 Z=0 />
            <scale X=0.06372 Y=0.06372 Z=0.06372/>
            <models>
                <model file = "sphere.3d" />
            </models>
            <group>
                Lua
                <translate X = 0.06872 Y=0 Z= 0.06872/>
                <scale X=0.018 Y=0.018 Z=0.018/>
                <models>
                    <model file = "sphere.3d" />
                </models>
```

```
            </group>
        </group>
        <group>
            Marte
            <translate X=3.3188 Y=0 Z=0 />
            <scale X=0.03402 Y=0.03402 Z=0.03402/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Jupiter
            <translate X=4.7 Y=0 Z=0 />
            <scale X=0.68366 Y=0.68366 Z=0.68366/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Saturno
            <translate X=6.62 Y=0 Z=0 />
            <scale X=0.60268 Y=0.60268 Z=0.60268/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Urano
            <translate X=9.66 Y=0 Z=0 />
            <scale X=0.25559 Y=0.25559 Z=0.25559/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
        <group>
            Neptuno
            <translate X=12.85 Y=0 Z=0 />
            <scale X=0.24622 Y=0.24622 Z=0.24622/>
            <models>
                <model file = "sphere.3d" />
            </models>
        </group>
    </group>
</scene>
```

```
D:\BACKUP SSD\Windows\Desktop\UM\Cadeiras\3º ANO\2º Semestre\Computação Gráfica\Aula4\build\Release>class4.exe solar.xml
Vendor: Intel
Renderer: Intel(R) HD Graphics 630
Version: 4.4.0 - Build 22.20.16.4708

Use Arrows to move the camera up/down and left/right
F1 and F2 control the distance from the camera to the origin

Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
Ficheiro 'sphere.3d' importado com sucesso.
```

Figura 4.1: Execução do programa com o ficheiro XML correspondente ao Sistema Solar.

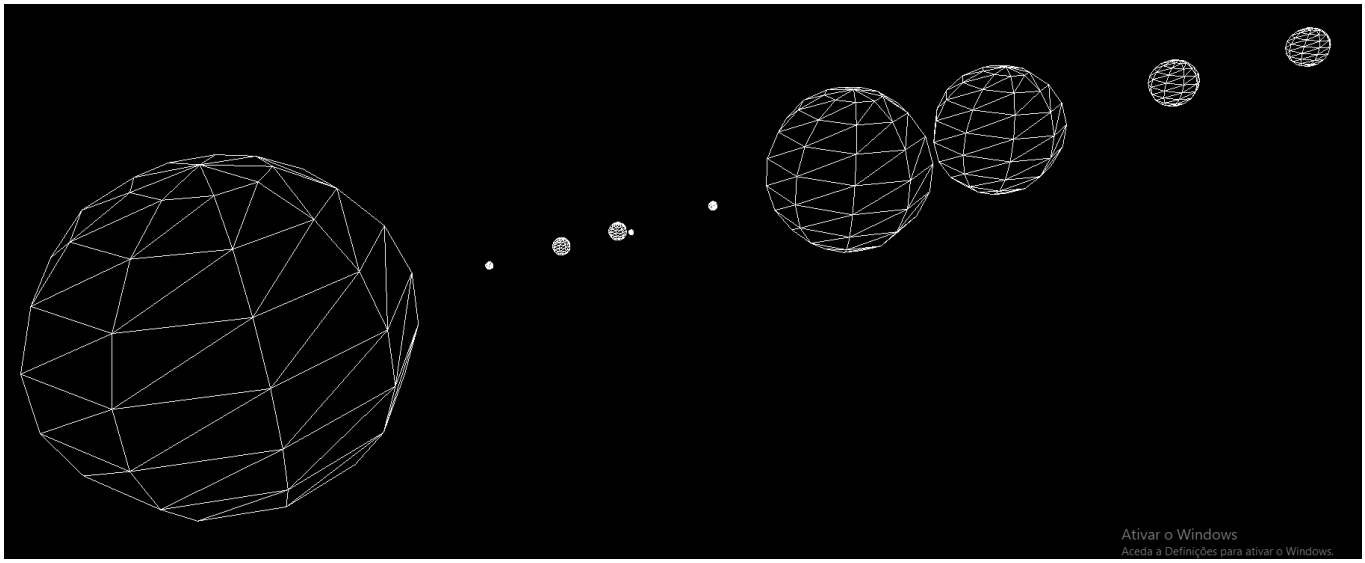

Figura 4.2: Cenário correspondente ao Sistema Solar, obtido através da execução do programa.

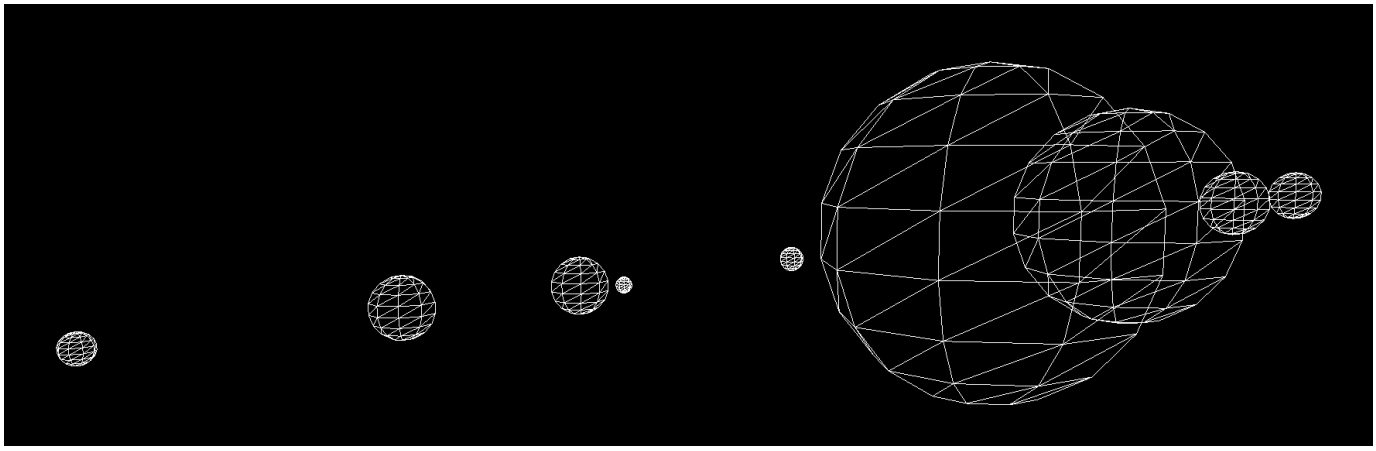

Figura 4.3: Observação do cenário a partir de Mercúrio.

Capítulo 5

Conclusão

O objetivo desta segunda fase do trabalho prático de Computação Gráfica consistiu no enriquecimento do parser do XML, permitindo a leitura de transformações geométricas e de hierarquias. De maneira a atingir este objetivo, inicialmente foi concebida uma árvore de classes, de maneira a entender como deveria ser efetuada a leitura dos ficheiros de configuração. Após a conversão dos ficheiros de configuração em árvores, são percorridas em profundidade e o método de cada classe presente em cada nodo é aplicado no glut. Achamos que conseguimos atingir o nosso objetivo e expressamos o nosso contentamento em finalizar esta fase do trabalho prático, devido aos problemas de implementação que foram surgindo ao longo da concepção da resolução.

Apêndice A

Código do Programa

Lista-se a seguir o código do programa que foi desenvolvido.

```
#include "tinyxml\tinystr.h"
#include "tinyxml\tinyxml.h"
#include <stdio.h>

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

#define _USE_MATH_DEFINES
#include <math.h>

#include <iostream>
#include <vector>
#include <fstream>
#include <cstring>
#include <sstream>

using namespace std;

float alfa = 0.0f, beta = 0.0f, radius = 5.0f;
float camX, camY, camZ;

class Group {

public:
    Group::Group() {
        id = 0;
    }
    Group::Group(int a) {
        id = a;
    }
    int getId() {
        return id;
    }

    virtual int apply() {
        glPushMatrix();
        return 0;
```

```
    }

private:
    int id;
};

class Scale : public Group {

public:
    Scale::Scale(float a, float b, float c) {
        x = a;
        y = b;
        z = c;
    }
    Scale::Scale() {
        x = 0;
        y = 0;
        z = 0;
    }

    int apply() {
        glScalef(x, y, z);
        return 2;
    }

private:
    float x, y, z;
};

class Translate : public Group {

public:
    Translate::Translate(float a, float b, float c) {
        x = a;
        y = b;
        z = c;
    }
    Translate::Translate() {
        x = 0;
        y = 0;
        z = 0;
    }

    int apply() {
        glTranslatef(x, y, z);
        return 3;
    }

private:
    float x, y, z;
};

class Rotate : public Group {

public:
    Rotate::Rotate(float l, float a, float b, float c) {
        angle = l;
        x = a;
        y = b;
```

```
        z = c;
    }
    Rotate::Rotate() {
        x = 0;
        y = 0;
        z = 0;
        angle = 0;
    }

    int apply() {
        glRotatef(angle, x, y, z);
        return 1;
    }

private:
    float x, y, z, angle;
};

class Model : public Group {

public:
    Model::Model(string s) {
        modelo = s;
    }

    string getModelo() {
        return modelo;
    }

    int apply() {

        ifstream file(modelo);
        string str;

        getline(file, str);

        glBegin(GL_TRIANGLES);

        while (getline(file, str)) {

            float v1, v2, v3;

            istringstream ss(str);

            ss >> v1;
            ss >> v2;
            ss >> v3;

            glVertex3f(v1, v2, v3);
        }

        glEnd();

        return 4;
    }

public:
    string modelo;
};
```

```
typedef struct node {

    Group* g;
    char* label;
    vector<struct node*> sons;

} *Arvore;

Arvore cg;
int idx = 0;

int xml_parser(char* fxml) {

    string fich_xml = (string)fxml;

    int num = 0;
    int cap = 0;

    //Objeto para ler XML
    TiXmlDocument doc;

    if (!doc.LoadFile(fich_xml.c_str())) {

        //nome do ficheiro de configuracao (path) nao enontrado a partir da diretoria corrente
        printf("Erro ao carregar o ficheiro de configuracao XML.\n");
        return 1;
    }

    TiXmlNode* base = doc.FirstChild();

    if (strcmp(base->Value(), "scene") != 0) {

        printf("XML nao comeca com o elemento \" <scene> \"\n");
        return 1;
    }

    TiXmlElement* elementos = base->FirstChildElement("group");
    TiXmlElement* modelos;

    if (elementos == NULL) {

        printf("base-> nenhum elemento encontradoo");
        return 0; // <group> nao encontrado
    }

    cg = new struct node;

    Arvore cgaux = new struct node;

    cgaux = cg;
    cgaux->g = new Group(idx++);
    cgaux->label = "group";

    // queue de apontadores de elementos para percorrer a arvore de hierarquias
    vector<TiXmlElement*> stackgroups;

    //queue que guarda os nós para adicionar os seus possiveis filhos posteriormente
    vector<struct node *>stack_nodes_group;
```

```
// começar a percorrer a hierarquia
elementos = elementos->FirstChildElement();

while (elementos != NULL) {

    if (strcmp(elementos->Value(), "group") == 0) {

        stackgroups.push_back(elementos->FirstChildElement());

        Arvore aux = new struct node;
        aux->g = new Group(idx++);
        aux->label = "group";
        aux->sons.clear();

        cgaux->sons.push_back(aux);

        stack_nodes_group.push_back(aux);

        cap++;
    }

    if (strcmp(elementos->Value(), "scale") == 0 || strcmp(elementos->Value(), "translate") == 0) {

        float x, y, z;
        x = y = z = 0.0;

        if (elementos->Attribute("X")) {
            x = atof(elementos->Attribute("X"));
        }

        if (elementos->Attribute("Y")) {
            y = atof(elementos->Attribute("Y"));
        }

        if (elementos->Attribute("Z")) {
            z = atof(elementos->Attribute("Z"));
        }

        if (strcmp(elementos->Value(), "scale") == 0) {

            Arvore aux = new struct node;
            aux->g = new Scale(x,y,z);
            aux->label = "scale";
            aux->sons.clear();

            cgaux->sons.push_back(aux);
        }
        else {

            Arvore aux = new struct node;
            aux->g = new Translate(x, y, z);
            aux->label = "translate";
            aux->sons.clear();

            cgaux->sons.push_back(aux);
        }
    }
```

```
        if (strcmp(elementos->Value(), "rotate") == 0) {

            float x, y, z, angle;
            x = y = z = angle = 0.0;

            if (elementos->Attribute("axisX")) {
                x = atof(elementos->Attribute("axisX"));
            }

            if (elementos->Attribute("axisY")) {
                y = atof(elementos->Attribute("axisY"));
            }

            if (elementos->Attribute("axisZ")) {
                z = atof(elementos->Attribute("axisZ"));
            }

            if (elementos->Attribute("angle")) {
                angle = atof(elementos->Attribute("angle"));
            }

            Arvore aux = new struct node;
            aux->g = new Rotate(angle, x, y, z);
            aux->label = "rotate";
            aux->sons.clear();

            cgaux->sons.push_back(aux);
        }

        if (strcmp(elementos->Value(), "models") == 0) {

            modelos = elementos;
            modelos = modelos->FirstChildElement("model");

            while (modelos != NULL) {

                const char *nome = modelos->Attribute("file");

                FILE *test;
                if ((fopen_s(&test, nome, "r")) == 0) {

                    cout << "Ficheiro \'" << nome << "\' importado com sucesso." << endl;

                }

                Arvore aux = new struct node;

aux->g = new Model((char*)nome);
aux->label = "model";
aux->sons.clear();

                cgaux->sons.push_back(aux);

                modelos = modelos->NextSiblingElement();
            }
        }

        elementos = elementos->NextSiblingElement();
```

```
        if (elementos == NULL) {

            if (cap!=0 ) {

                elementos = stackgroups[num];

                cgaux = stack_nodes_group[num];

                num++;
                cap--;
            }
        }
    }

    return 0;
}

void depth_first(struct node *senpai, struct node* xx) {

    if (senpai != NULL) {
        if (strcmp(senpai->label, "group") == 0 && xx==NULL) {

        glPopMatrix();

        }
    }

    if (xx != NULL) {

        (xx->g)->apply(); //executar a respetiva transformaçao/draw da classe


        int ssize = xx->sons.size();

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

            if (i<ssize)
                depth_first(xx, xx->sons[i]);
            else
                depth_first(xx, NULL);
        }
    }
}

void draw_models() {

    //perccorer a arvore de classes
    depth_first(NULL, cg);

    glPopMatrix();

}

void spherical2Cartesian() {

    camX = radius * cos(beta) * sin(alfa);
    camY = radius * sin(beta);
    camZ = radius * cos(beta) * cos(alfa);
}
```

```
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(camX, camY, camZ,
        0.0, 0.0, 0.0,
        0.0f, 1.0f, 0.0f);

    //glutWireSphere(1, 2, 3);
    //glutWireTorus(0.4, 1, 5, 10);

    draw_models();

    // End of frame
    glutSwapBuffers();

}

void processKeys(unsigned char c, int xx, int yy) {

    // put code to process regular keys in here

}

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

    switch (key) {
```

```
    case GLUT_KEY_RIGHT:
        alfa -= 0.1; break;

    case GLUT_KEY_LEFT:
        alfa += 0.1; break;

    case GLUT_KEY_UP:
        beta += 0.1f;
        if (beta > 1.5f)
            beta = 1.5f;
        break;

    case GLUT_KEY_DOWN:
        beta -= 0.1f;
        if (beta < -1.5f)
            beta = -1.5f;
        break;

    case GLUT_KEY_F2: radius -= 0.1f;
        if (radius < 0.1f)
            radius = 0.1f;
        break;

    case GLUT_KEY_F1: radius += 0.1f; break;
    }
    spherical2Cartesian();
    glutPostRedisplay();

}

void printInfo() {

    printf("Vendor: %s\n", glGetString(GL_VENDOR));
    printf("Renderer: %s\n", glGetString(GL_RENDERER));
    printf("Version: %s\n", glGetString(GL_VERSION));

    printf("\nUse Arrows to move the camera up/down and left/right\n");
    printf("F1 and F2 control the distance from the camera to the origin\n\n");

}

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

    // init GLUT and the window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(800, 800);
    glutCreateWindow("MOTOR 3D");



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



    // Callback registration for keyboard processing
```

```
    glutKeyboardFunc(processKeys);
    glutSpecialFunc(processSpecialKeys);


    //  OpenGL settings
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glPolygonMode(GL_FRONT, GL_LINE);

    spherical2Cartesian();

    printInfo();

    if (argc < 2) {
        printf("[Loading files] Ficheiro de configuracao nao encontrado!\n");
        return 0;
    }


    xml_parser(argv[1]);


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

    return 1;
}
```