Computação Gráfica

Trabalho Prático CG - 2017/18

Fase 3 - Curvas, Superfícies Cúbicas e VBOs

Relatório de Desenvolvimento

André Pereira (A79196)

Filipe Miranda (A78992)

José Alves (A78178)

27 de Abril de 2018

Conteúdo

Capítulo 1

Introdução

O trabalho prático **Curvas, Superfícies Cúbicas e VBOs** surge inserido na disciplina de Computação Gráfica e corresponde à terceira fase do trabalho prático desta unidade curricular. Versa sobre o uso do **TinyXML** e corresponde, essencialmente, em enriquecer o gerador de criar modelos com base nos patches de Bezier e extender os elementos translação e rotação do motor. 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 em enriquecer o gerador de maneira a que seja capaz de criar um novo tipo de modelo, com base nos patches de Bezier. O gerador recebe o nome do ficheiro onde são definidos os pontos de controlo de Bezier e o nível de tesselação requerido como parâmetros e gera um ficheiro com a lista dos triângulos para desenhar a superfície. Relativamente ao motor, são extendidos os elementos de translação e rotação.

Considerando uma translação, é fornecido um conjunto de pontos que definem uma curva cúbica de Catmull-Rom, bem como o número de segundos em que deve percorrer a curva na sua totalidade, com o objetivo de realizar animações com base nestas curvas. Os modelos podem ter uma transformação que depende do tempo ou estática como nas fases anteriores. Já numa rotação, o ângulo pode ser substituido pelo parâmetro do tempo, isto é, o número de segundos que demora para fazer um rotação total de 360 graus em torno do eixo especificado. Para medir o tempo é usada a função *glut_elapsed_time*.

2.2 Especificação dos Requisitos

Devido à definição da curva de Catmull-Rom, é sempre necessário um ponto inicial antes do segmento de curva inicial e outro ponto após o último segmento. Portanto, o número mínimo de pontos terá que ser 4. Nesta fase, também é necessário que os modelos sejam desenhados com VBOs, contrariamente a como foi feito nas fases anteriores. O demo requisitado para esta fase corresponde a um modelo dinâmico do sistema solar, incluindo um cometa com uma trajetória definida através das curvas de Catmull-Rom. O cometa deve ser construído usando os patches de Bezier, com os pontos de controlo do teapot.

2.2.1 Exemplo de Translações e Rotações

```
...
<translate time=10 >
    <point X=1 Y=0 Z=1 />
    <point X=0.707 Y=0.707 Z=1 />
    <point X=0 Y=1 Z=1 />
    ...
    <point X=-1 Y=0 Z=1 />
</translate>
...
<rotate time=10 axisX=0 axisY=1 axisZ=0 />
...
```

Capítulo 3

Concepção/Desenho da Resolução

3.1 Classe Catmull

Tendo em conta a concepção feita na etapa anterior relativamente à árvore das classes, foi criada a classe Catmull, que engloba dois construtores (translação e rotação), cuja distinção é feita através de uma flag para saber qual o método apply que deve ser aplicado. No caso de ser uma translação, o motor recebe um tempo (em segundos) e um vetor de floats, que correspondem aos pontos de controlo. Ao entrar neste construtor, a flag estará sinalizada a 1 e o método apply corresponde a invocar a função *getGlobalCatmullRomPoint* para calcular os pontos da curva que irá percorrer. Caso seja uma rotação, o motor recebe um tempo na mesma, bem como as coordenadas x, y e z para saber por qual eixo vai ser executada. A flag estará sinalizada a 0 e o método apply corresponde a calcular o angulo através da fórmula angaux = $\frac{360}{fps*time}$, incrementá-lo ao ângulo que foi calculado previamente (rangle), invocar a função *glRotatef(rangle, xa, ya, za)* e o processo repete-se por cada fps.

3.2 VBOs

Nesta fase, um dos requisitos correspondie a desenhar os modelos com VBOs, ao invês de como foi feito nas fases anteriores. Um vertex buffer object é um recurso do OpenGL que melhora significativamente o desempenho em relação à renderização no modo imediato. Enquanto que dantes o método apply da classe modelo correspondia a fazer *glVertex3f(v1, v2, v3)* a cada 3 pontos que encontrasse no ficheiro, agora é inicializado um VBO. Gera-se um buffer, junta-lhe o vetor dos pontos bem como os dados relativos a cada um deles e desenha-os, invocando *glVertexPointer(3, GL_FLOAT, 0, 0)* e *glDrawArrays(GL_TRIANGLES, 0, numvertices*3)*. No final é apagado o buffer, evitando que consuma memória excessivamente.

3.3 Parser XML

Relativamente ao parser do XML foram adicionadas as novas condições que processam a informação no caso de no ficheiro XML forem encontradas tags relativas às translações e às rotações, mas que contenham o parâmetro do tempo. No caso de encontrar uma translação com o parâmetro do tempo, o XML terá que ler todos os pontos de controlo (bem como guardá-los num vetor) e irá criar uma clase do tipo Catmull com os parâmetros do tempo e do vetor. No caso de encontrar uma rotação com o parâmetro do tempo, será criada uma classe do tipo Catmull com os parâmetros que se encontravam nessa tag.

3.4 Patches de Bezier

Um ficheiro de texto que contém a descrição de um conjunto de patches tem o seguinte formato:

- a primeira linha contém o número de patches;

- as linhas a seguir, uma para cada patch, contém os índices dos pontos de controlo (16 para cada patch);
- a próxima linha contém o número de pontos de controlo e depois os próprios pontos de controlo, um por linha.

O gerador recebe o nome do ficheiro onde são definidos os pontos de controlo de Bezier (o teapot.patch, por exemplo) e o nível de tesselação requerido como parâmetros e gera um ficheiro com a lista dos triângulos para desenhar a superfície. Tal como acontecia na esfera com as stacks e as slices, quanto maior for o nível de tesselação mais ”smooth” será a superfície.

Capítulo 4

Codificação e Testes

4.1 Testes realizados e Resultados

O demo requisitado para esta fase corresponde a um modelo dinâmico do sistema solar, incluindo um cometa com uma trajetória definida através das curvas de Catmull-Rom. O cometa deve ser construído usando os patches de Bezier, com os pontos de controlo do teapot. Mostram-se a seguir alguns testes feitos e os respectivos resultados obtidos, começando pelo ficheiro de configuração para o Sistema Solar já com as novas implementações:

```
<scene>
    <group>
        <models>
            Sol
            <model file="sphere.3d" />
        </models>
        <group>
            Mercurio
            <translate X=1.5236 Y=0 Z=0 />
            <rotate time=58  axisX=0 axisY=1 axisZ=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 />
            <rotate time=480 axisX=0 axisY=-1 axisZ=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 />
                <rotate time=2 axisX=0 axisY=1 axisZ=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 />
    <rotate time=2.2 axisX=0 axisY=1 axisZ=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 />
    <rotate time=0.9 axisX=0 axisY=1 axisZ=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 />
    <rotate time=0.85 axisX=0 axisY=1 axisZ=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 />
    <rotate time=1.3 axisX=0 axisY=1 axisZ=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 />
    <rotate time=1.2 axisX=0 axisY=1 axisZ=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.

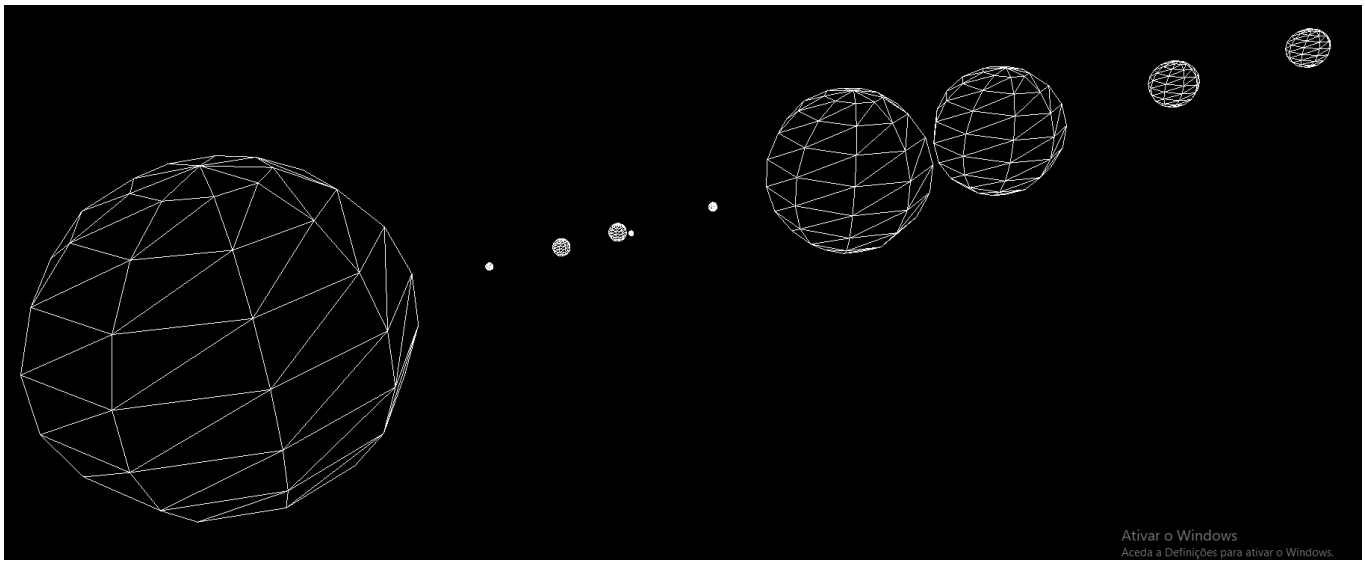

Figura 4.2: Cenário correspondente ao Sistema Solar, obtido através da execução do programa.

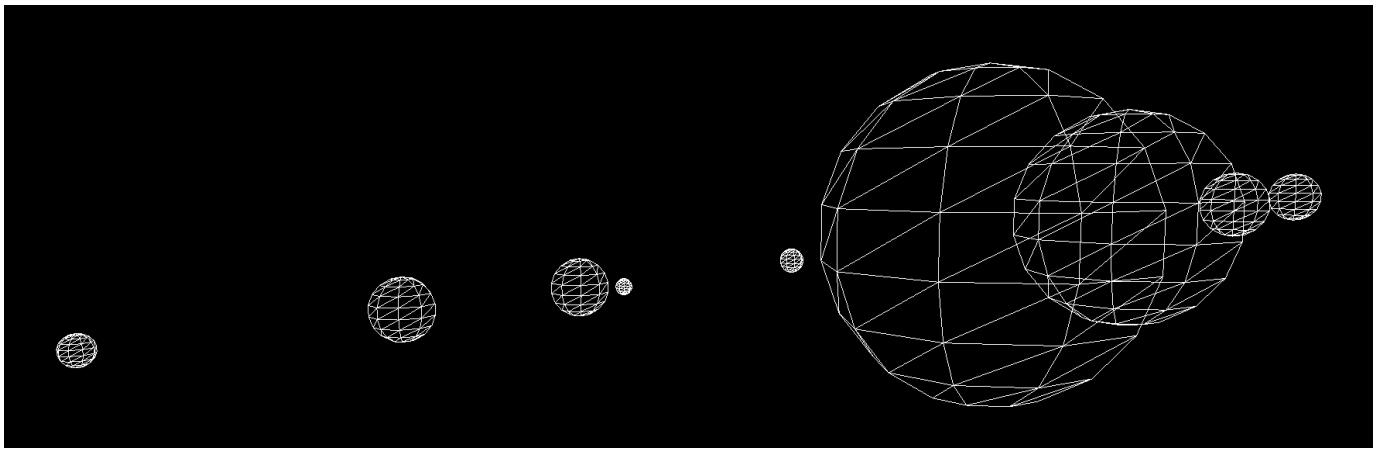

Figura 4.3: Observação do cenário a partir de Mercúrio.

Capítulo 5

Conclusão

O objetivo desta terceira fase do trabalho prático de Computação Gráfica consistiu em criar modelos com base nos patches de Bezier e extender os elementos que fazem parte de uma translação e de uma rotação. Foi adicionada uma nova classe à árvore das classes (Catmull), a classe dos modelos foi alterada de maneira a que agora sejam desenhados com VBOs e o parser foi enriquecido de maneira a que aceite os novos constituintes dos ficheiros de configuração. Achamos que podiamos ter apresentado um melhor resultado nesta fase do trabalho prático, mas mesmo assim exprimimos o nosso contentamento em a termos finalizado, 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 sphericalCartesian() {

    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;
}
```