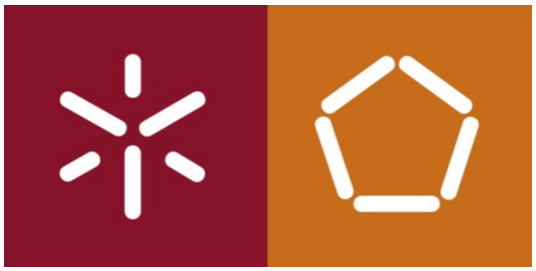


Universidade do Minho
Escola de Engenharia

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Relatório do Trabalho Prático 2

A78890 Alexandre Costa
A75248 Ana Sofia Gomes Marques
A65277 Flávio Manuel Machado Martins
A79799 Gonçalo Costeira

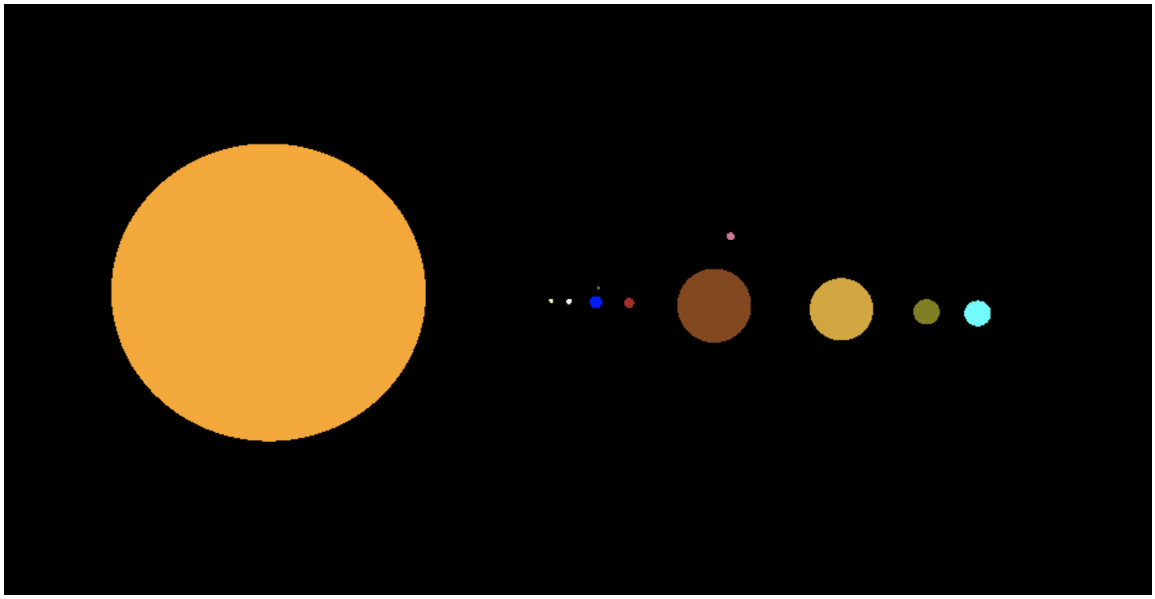
Grupo 50

29 Março 2020

Conteúdo

1 Introdução

Este relatório visa apresentar as decisões tomadas na realização da segunda fase do trabalho prático da Unidade Curricular de Computação Gráfica. Procuramos assim justificar todas as ponderações feitas na formulação do problema, assim como a abordagem tomada para gerar, a partir de um ficheiro de input em xml com elementos pré-definidos, uma maquete de um sistema solar de forma simples, fácil de interagir e fazendo um bom uso das funções de glPushMatrix e glPopMatrix da matriz de visualização do modelo (modelview Matrix) do glut.
Numa primeira fase será descrito o processo de leitura e tratamento dos dados contidos no ficheiro xml e posteriormente é explicado o modo como desenvolvemos as estruturas de dados que suportam o programa e o modo como cada elemento é renderizado na função renderscene.



2 Aplicação Sistema Solar

2.1 SistemaSolar.cpp

Tal como na primeira fase, continuamos a fazer leituras de um ficheiro xml (input.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>), como era pedido no enunciado do projeto. 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.

```
    tinyxml2::XMLDocument doc;

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

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

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

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

Foi decido implementar uma classe, em C++, inspirada numa estrutura de dados 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 reduz drasticamente a memória necessária em cálculos sistemáticos sobre o que está a tentar representar.

2.2 Classe Scene Graph

A classe Scene Graph encontra-se definida no ficheiro "draw.h"e serve de suporte à implementação do ficheiro de leitura. 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. 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.

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

```

É importante salientar que:

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.

```
<scene>
    <group>
        <group>
            <!-- Sol -->
            <translate x="-5" y="0" z="0" />
        <cor R="1.0" G="0.647" B="0" />
            <scale x="25" y="25" z="25" />
        <rotate x="0" y="0" z="0" ang="0" />
            <models>
                <model file="sphere.3d"/>
            </models>
        </group>
        <group>
```

2.3 Funcionamento

Fixada a estrutura e comportamento do classe. Basta, agora, expor o algoritmo de travessia que é efetuada quando a aplicação lê um ficheiro de configuração.

```
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]);
    glColor3f(cor[0], cor[1], cor[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();
}
```

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

Class principal para a codificação de um SceneGraph básico para os arrays que codificam certas transformações. A posicao 0 simboliza a coordenada x, a posicao 1 codifca a coordenada y, etc. No caso das rotações a posicao 0 codifica o ângulo e de seguida são dados os eixos de rotação.A função de desenho considera escalas primeiro, rotaões de seguida e finalmente translações.

2.4 Eng.cpp

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

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:

```
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:

```
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;
}
```

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

```
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.

A função doModels chama a função guardaPontos que, usando a estrutura Pontos, regista as coordenadas de cada ponto presente no ficheiro.
O 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, retornando os arrays de argumentos e os pontos das figuras, fazendo uma atualização da estrutura que no sistemaSolar.cpp é passada à renderScene.

3 Interação com a aplicação

De modo a tornar mais simples a compilação do nosso programa decidimos, em semelhança com o que foi feito na primeira fase, criar uma makefile que executa os vários comandos necessários, tendo esta uma opção para sistema macbook e outra para linux, uma vez que os menbros do grupo usam sistemas operativos diferentes.

Para possibilitar uma melhor visualização do sistema, a nossa aplicação permite nesta fase as seguintes interações:

1. Alteração da camara
    - Tecla Up - Deslocar para cima
    - Tecla Down - Deslocar para Baixo
    - Tecla Right - Deslocar para esquerda
    - Tecla Left - Deslocar para direita
    - Tecla + - Aumentar o zoom da camara
    - Tecla - - Diminuir o zoom da camara
2. Movimentação
    - Tecla W - Mover no sentido positivo do eixo Z
    - Tecla S - Mover no sentido negativo do eixo Z
    - Tecla A - Mover no sentido positivo do eixo X
    - Tecla D - Mover no sentido negativo do eixo X
3. Opções de visualização do modelo
    - Tecla l - Só linhas
    - Tecla f - Preenchido
    - Tecla p - Só vértices

4 Conclusão

Durante a realização deste trabalho fomos encontrando algumas dificuldades, mais concretamente na forma com interpretar o conteúdo do ficheiro xml de entrada e no modo como, utilizando as funções glPushMatrix e glPopMatrix da biblioteca glut poderíamos reutilizar as transformações já realizadas até um determinado ponto do modelo desenhado.
No nosso entender, todas as eventuais dificuldades foram totalmente ultrapassadas e compreendidas em grupo, simplificando assim a criação das estruturas de dados utilizadas e o modo como lidamos com a questão da hierarquia entre os grupos do ficheiro xml.
Apresentou-nos também mais uma oportunidade de melhorar as nossas capacidades de programação em C++, no uso de LaTex e de ficheiros XML.
Deste modo cumprimos todas as exigências pretendidas para esta segunda fase do trabalho, construindo uma maquete do sistema solar completa e enriquecida ainda com elementos como luas e cores dos planetas.
No final desta implementação, consideramos que fizemos uma boa reutilização das estruturas e classes já utilizadas na fase I e que esta fase foi bastante mais cativante, pelo facto de conseguir-mos interagir e moldar o sistema pretendido.

5 Referências

Lighthouse3d. (2017). GLUT Tutorial. online Available at:
http://www.lighthouse3d.com/tutorials/glut-tutorial/

Opengl-tutorial.org. (2017). Tutorial 3 : Matrices. online Available at:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/