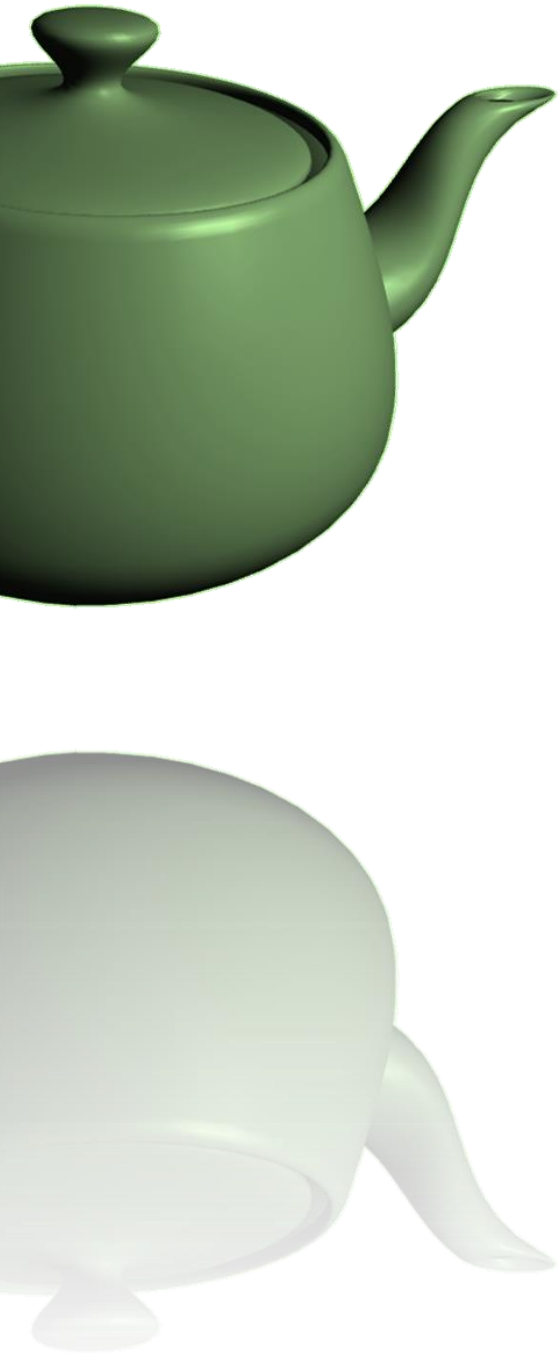
Universidade do Minho

Mestrado integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica



Geometric Transforms

Relatório

Trabalho elaborado por:
Carlos Pedrosa - a77320
David Sousa - a78938
Manuel Sousa - a78869

Índice

Índice de ilustrações

Introdução

No âmbito da unidade curricular de Computação Gráfica foi-nos proposto o desenvolvimento de um mini cenário baseado em gráficos 3D. Assim, o projeto encontra-se dividido em 4 fases. Neste relatório pretendemos demonstrar todos os passos que foram efetuados no sentido a cumprir a segunda tarefa proposta.

Nesse sentido, a segunda fase encontra-se dividida em duas partes. A primeira prende-se com a alteração do programa que permite fazer a leitura do XML. Assim, nesta parte o foco passa por processar corretamente os dados. A segunda, por sua vez, pressupõe a elaboração de um ficheiro em que seja possível a partir deste, gerar todo o sistema solar. Assim, o programa terá de desenhar o modelo contemplado.

Fase 2

Desenho do sistema solar

Um dos objetivos fundamentais desta fase era a implementação de um ficheiro XML que permitisse desenhar todo o sistema solar da melhor forma possível. Assim, nos subcapítulos seguintes faremos referência às principais alterações efetuadas para atingir o que era proposto.

1. Alteração do parser XML

Um dos pontos fulcrais deste trabalho é a leitura correta dos ficheiros XML. Assim, o desafio prendeu-se sobretudo na correta extração da informação contida nestes ficheiros. De facto, na fase anterior só era necessário guardar os nomes dos ficheiros para mais tarde ir buscar a informação neles presente. Nesta fase, para além disso é também necessário processar a informação contida nos campos ***translate***, ***rotate***, ***scale*** e ***file***, sendo que como complicação acrescida podem ainda existir grupos intrínsecos a outros grupos.

Para todo este processamento seria necessária a criação de duas estruturas de dados auxiliares (descritas em pormenor nos subcapítulos seguintes), uma que irá armazenar a informação auxiliar e outra que irá, efetivamente, guardar os dados que posteriormente serão processados.

Em primeiro lugar, vale notar de que não sabíamos quantos grupos embutidos poderiam existir, usamos assim a recursividade em C++, de modo a, ultrapassar este desafio. Consideramos também a hierarquia implícita dos ficheiros do tipo "XML". Esta hierarquia revelou ser bastante útil, uma vez que, quando saímos de um ***group*** pretendemos apagar a informação na estrutura de dados auxiliar no ***campo[i]***, em que o i é o nível do ***group*** em que nos encontramos.

O restante é trivial, o ***parser*** começa o processamento a partir da raiz sendo que quando encontra um grupo principal, isto é, a primeira **tag <group>** da hierarquia, chama uma função auxiliar para processar esse grupo com o argumento 0, pois trata-se do primeiro elemento encontrado. Sempre que o ***parser*** lê uma instrução *openGL* insere-a na estrutura auxiliar no ***campo[i]*** sendo que quando lê um ***model file=*** copia a informação da estrutura auxiliar para a principal para mais tarde ser tratada. Por fim, se existirem grupos intrínsecos a outros grupos, isto é, se encontramos uma ***tag <group>*** dentro de um outro ***group***, usamos recursividade. Deste modo, chamamos a mesma função auxiliar só que com o argumento ***hierarquia = i + 1***, ou seja, vamos descendo um nível na hierarquia para guardar as informações noutro campo, para não destruir as processadas no ***campo[i]*** da estrutura.

```
if (strcmp((char*)child2->Value(), "group") == 0) {
    parseGroup(child2, g, i + 1, files);

    for (int j = i + 1; j < 10; j++)
        g[j] = NULL;
}
```

Figura 1 - Demonstração da recursividade no parser

2. Estruturas de dados

Depois de explicadas as alterações efetuadas no parser é importante exemplificar as opções tomadas para guardar as informações que acima enunciamos. Assim, destas é essencial explicar o porquê da estrutura implementada.

O principal objetivo desta estrutura (para além de guardar as transformações aplicadas aos diferentes objetos), era o de preservar as transformações ao longo dos diferentes níveis da hierarquia do ficheiro XML. Visto que o ficheiro é composto por diferentes grupos (***<group>*),** e que estes grupos podem conter novos grupos dentro deles, é necessário que as transformações de um grupo principal sejam também aplicadas aos grupos subjacentes. De forma a simplificar a explicação da estrutura escolhida, iremos explicar cada estrutura auxiliar a esta individualmente.

- **Oper**

Tipo de dados que contém o nome da operação (***char* typeOper***), e diversos valores que dizem respeito a cada eixo do sistema de coordenadas (***double x, y, z, angle***). O intuito deste tipo de dados é o de guardar a informação, de uma forma estruturada, relativa a uma determinada operação

.

```
typedef struct oper {
    double x, y, z, angle;
    char* typeOper;
} Oper;
```

Figura 2 - Estrutura de dados, Oper

- **FileOper**

Este tipo de dados é o que atribui a cada ficheiro **.3d** (***char* fileName***) determinadas operações, que por sua vez são guardadas sobre a forma de um vetor. Este vetor é então composto por diversos apontadores para tipos de dados "Oper" (***vector<Oper*> operations***).

```
typedef struct foper {
    char* fileName;
    vector<Oper*> operations;
} FileOper;
```

Figura 3 - Estrutura de dados, FileOper

- **Group e LOper**

Estas estruturas de dados (consideradas por nós como estruturas auxiliares) surgiram após termos percebido que existia a possibilidade de grupos dentro de grupos. Posto isto, resolvemos criar uma estrutura de dados nomeada de "Group", baseada num array, em que cada posição deste contém um apontador para uma lista ligada que guarda a informação sobre as diferentes operações que vão sendo encontradas ao longo do ficheiro. Esta lista ligada foi nomeada de LOper. Cada posição do array foi interpretada como o nível da hierarquia do ficheiro XML. Este nível ia aumentando conforme as **tags *<group>*** surgiam. Quando se saísse de uma **tag *</group>*** principal, o nível voltava a 0. Sempre que este voltava a aumentar no ficheiro, era criada uma nova lista ligada. A posição seguinte do array tratava de guardar um apontador para essa mesma lista. É importante referir que à medida que diferentes ***<group>*** independentes iam sendo encontrados dentro de um ***<group>*** principal, o nível anterior ao atual não era eliminado de forma a que fosse copiado para a estrutura as operações que dizem respeito a um determinado objeto.

```
typedef struct loper {
    char* opName;
    double x, y, z, angle;
    struct loper* next;
} LOper;
```

```
typedef LOper* Group[10];
```

Figura 4 - Estrutura de dados LOper e respetivo array de estruturas

Em suma, o processo de desenho é feito através da leitura do vetor das operações - ***vector<FileOper*> files*** - e também do vetor que contém todas as coordenadas de todos os vértices de cada ficheiro ***.3d***. Este último vetor, o qual já foi usado na primeira fase do trabalho prático, contém diversos triplos que são construídos à medida que vai sendo feita a leitura de cada ficheiro ***.3d***. Estes triplos são do tipo ***tuple<double, double, double>***. Para facilitar a representação destes no código produzido, resolvemos criar um ***typedef***, e denominar o triplo por "coords". O vetor resultante irá, portanto, ser do tipo: ***vector<coords> triangles***. Posto isto, passaremos então a explicar como é que a informação é processada a partir destes vetores, e como é que o motor desenha os objetos pretendidos.

3. Motor

Com a introdução de transformações nos objetos, o algoritmo usado para desenhar os mesmos foi um pouco alterado, uma vez que estas transformações devem ser interpretadas em primeiro lugar. Como as estruturas usadas para guardar a informação foram vetores, resolvemos usar ***iterators*** de forma a aceder e a processar cada elemento individualmente. Todo o algoritmo é envolvido num primeiro ciclo ***while***, que só termina assim que não existirem mais coordenadas no ***vector<coords> triangles***. Isso significará que todos os triângulos referentes aos diversos objetos previamente calculados, foram (teoricamente) desenhados. Dentro deste ciclo principal, existem dois ciclos fulcrais, que dizem respeito às transformações dos objetos e ao desenho dos triângulos. Visto que as nossas estruturas são compostas por diversos vetores (3 neste caso), foram usados, portanto, 3 ***iterators*** distintos. De forma a facilitar a explicação, dividiremos esta em 2 partes respetivas aos dois ciclos interiores.

- Primeiro ciclo (Percorrer ***vector<Oper*> operations***):

A estratégia usada passa por ler cada um dos ***FileOper**** do ***vector<FileOper*>*** files, obter o ***vector<Oper*> operations*** que existe dentro de cada um destes ***FileOper*,*** percorrer esse mesmo vector e aplicar as diferentes transformações que vão sendo lidas. É de notar aqui que não é necessário identificar qual é o nome do objeto à qual estão a ser aplicadas as transformações, visto que as estas, bem como as coordenadas a elas referentes são guardadas de forma ordenada conforme o ficheiro XML. Isto faz com que o processamento da informação dos vetores seja facilitado. Para além disso, é importante realçar o uso da função ***glPushMatrix()*** que ao permitir guardar o estado do eixo, era usada sempre que necessitávamos de voltar ao inicio(por exemplo, quando um grupo era iniciado independente de outro).

```
itOper = fo->operations.begin();
glPushMatrix();
while (itOper != fo->operations.end()) { // Fazer transformações.
    Oper* op = *itOper;

    if (strcmp(op->typeOper, "scale") == 0)
        glScalef(op->x, op->y, op->z);

    if (strcmp(op->typeOper, "rotate") == 0)
        glRotatef(op->angle, op->x, op->y, op->z);

    if (strcmp(op->typeOper, "translate") == 0)
        glTranslatef(op->x, op->y, op->z);

    itOper++;
}
```

Figura 5 - Ciclo while relativo às transformações

- Segundo ciclo (Percorrer ***vector<coords> triangles***):

Apesar da forma de leitura não ter mudado da primeira para a segunda fase (ou seja, de lermos 3 coordenadas numa só iteração, e depois desenhar o respetivo triângulo que estas compõem), deparamo-nos com o facto da impossibilidade de saber quando é que terminava um objeto, e começava outro. Isto acontecia, pois, o ***vector<coords> triangles*** continha todas as coordenadas dos triângulos a serem desenhados, não existindo nenhuma "marca" indicadora do objeto à qual pertenciam. Desta forma, resolvemos criar essa "marca" introduzindo uma coordenada que servisse como ponto de paragem aquando a leitura do vetor. Esta coordenada era introduzida depois de serem extraídas todas as coordenadas de um determinado ficheiro .3d (ver ficheiro extractCoords.cpp). Devido à nossa forma de leitura (3 coordenadas numa iteração), resolvemos então introduzir 3 destas coordenadas "indicadoras", para assim não existirem problemas na leitura. Estas coordenadas (***tuple<double, double, double>***) são compostas pelo menor valor decimal que é possível representar.

FLT_MIN	1E-37 or smaller		
DBL_MIN	1E-37 or smaller	MINimum	Minimum representable positive floating-point number.
LDBL_MIN	1E-37 or smaller		

```
coords mark1 = make_tuple(DBL_MIN, DBL_MIN, DBL_MIN);
triangles.push_back(mark1);
coords mark2 = make_tuple(DBL_MIN, DBL_MIN, DBL_MIN);
triangles.push_back(mark2);
coords mark3 = make_tuple(DBL_MIN, DBL_MIN, DBL_MIN);
triangles.push_back(mark3);
```

Figura 6 - Representação de DBL_MIN em C++ e trecho código com as marcas referidas

A vida deste ciclo é então determinada pelo ***vector<coords> triangles***, e uma ***flag*** de valor **booleano**, que indica se a "marca" já foi extraída do vector. Se sim, o ciclo termina, e isso significará que um objeto foi desenhado na sua totalidade, podendo assim avançar para o processamento de um novo. Este valor **booleano** é controlado pela função ***isMark()***, que simplesmente faz a comparação da coordenada lida com a coordenada "marca".

```
bool isMark(coords aux_1, coords aux_2, coords aux_3)
{
    coords mark = make_tuple(DBL_MIN, DBL_MIN, DBL_MIN);

    return (aux_1 == mark && aux_2 == mark && aux_3 == mark);
}
```

Figura 7 - Função isMark()

Se não se verificar a existência da "marca", é então criado o bloco ***glBegin(GL_TRIANGLES) ... glEnd()***, que trata de desenhar o triângulo previamente lido. Neste ciclo, é notório o uso da função ***glPopMatrix().*** Cada alteração em OpenGl conduz a uma composição de matrizes. Assim, este facto conduz a um efeito cumulativo, indesejado neste caso. Para tal, esta função permite restaurar o estado anteriormente gravado.

```
while (itTri != triangles.end() && flag != false) {
    coords aux_1 = *itTri; itTri++;
    coords aux_2 = *itTri; itTri++;
    coords aux_3 = *itTri; itTri++;

    if (!isMark(aux_1, aux_2, aux_3)) {
        glBegin(GL_TRIANGLES);
            glVertex3d(get<0>(aux_1), get<1>(aux_1), get<2>(aux_1));
            glVertex3d(get<0>(aux_2), get<1>(aux_2), get<2>(aux_2));
            glVertex3d(get<0>(aux_3), get<1>(aux_3), get<2>(aux_3));
        glEnd();
    } else {
        flag = false;
    }
}
glPopMatrix();
```

Figura 8 - Segundo ciclo While presente na função drawTriangles()´

4. Pequeno exemplo do que até agora foi referenciado

Grupo intrínseco a outro

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

1ºIteração: Trata-se do primeiro nível da hierarquia, pelo que alteraremos o ***g[0]***. Encontrou um ***group***, é invocada a função ***parseGroup*** (função auxiliar já referenciada), que permite carregar todos os elementos para a estrutura (neste caso o translate de X=1 e o nome do ficheiro).

2ºIteração: Trata-se do segundo nível da hierarquia, pelo que uma tag ***group*** é apresentado intrinsecamente a outra. Os níveis anteriores não são eliminados, de forma a, que seja copiado para a estrutura toda a informação que diz respeito ao objeto (nível 0 + nível 1). Neste caso, encontramo-nos no ***g[1]***. É guardada informação referente ao translate Y=1 e ao nome do ficheiro, mas também como se trata de um grupo embutido,

também irão ser guardadas para este ficheiro as informações relativas as operações do nível superior.

Estas operações vão fazendo uso da recursividade pelo que as iterações mencionadas são aplicadas ao longo do ficheiro XML.

5. Ficheiro XML de configuração do sistema solar

```
<scene>
  <group>
        <translate Z="6" />
        <models>
            <model file="sphere.3d" /> <!-- Earth -->
        </models>
        <group>
            <translate Z="10" />
            <scale X="0.38" Y="0.38" Z="0.38" />
            <models>
                <model file="sphere.3d" /> <!-- Mercury -->
            </models>
        </group>
        <group>
            <translate Z="4.5" />
            <scale X="0.9" Y="0.9" Z="0.9" />
            <models>
                <model file="sphere.3d" /> <!-- Venus -->
            </models>
        </group>
```

```
        <model file="sphere.3d" /> <!-- Jupiter -->
    </models>
    <group>
        <translate Y="1.3" />
        <scale X="0.15" Y="0.15" Z="0.15" />
        <models>
            <model file="sphere.3d" />
        </models>
        <group>
            <translate X="8.67" Y="-8.67"/>
            <scale X="0.75" Y="0.75" Z="0.75" />
            <models>
                <model file="sphere.3d" />
            </models>
        </group>
    </group>
</group>
<group>
```

Figura 9 - Excerto da configuração XML para o Sistema Solar

Pelo excerto acima representado, é possível verificar que tentamos tirar partido do facto de existirem grupos intrínsecos a outros. Assim, uma vez que as transformações dos grupos anteriores serão também aplicadas, fizemos o cálculo dos restantes planetas a partir das medidas da Terra. De realçar ainda, o desenho de dois satélites naturais em Júpiter para enfatizar estes factos.

Terra: 6 371 km de raio
149 600 000 km de distancia do sol

Sol:109,1979 vezes maior à terra

Planeta	Distância ao Sol (km)	Tamanho em relação à Terra
Mercúrio	57 910 000	0,3829
Vénus	108 200 000	0,9499
Marte	227 900 000	0,5319
Júpiter	778 500 000	10,9733
Saturno	1 429 000 000	6,0001
Urano	2 871 000 000	3,9809
Neptuno	4 495 000 000	3,8647
Plutão	5 900 000 000	0,1868

Começamos por desenhar as figuras consoante as escalas apresentadas. No entanto, fomos ajustando as medidas pois, apesar de tudo, em certos casos, uma diminuição do tamanho tornava melhor a visualização de todo o sistema.

Assim, os três primeiros planetas possuem operações de translações positivas (de acordo com a forma como se encontra o referencial), pois encontram-se antes da Terra, os restantes possuem estas negativos. A partir da distância da Terra ao Sol e, consequentemente da distância dos restantes planetas à Terra, adaptamos as diferentes escalas. O valor **Z=6** da Terra será o valor base. Uma vez que a distância de mercúrio à Terra é de 91650000 o valor de translação será de 10 (mais os 6 devido à característica que os grupos intrínsecos apresentam). E assim, sucessivamente.

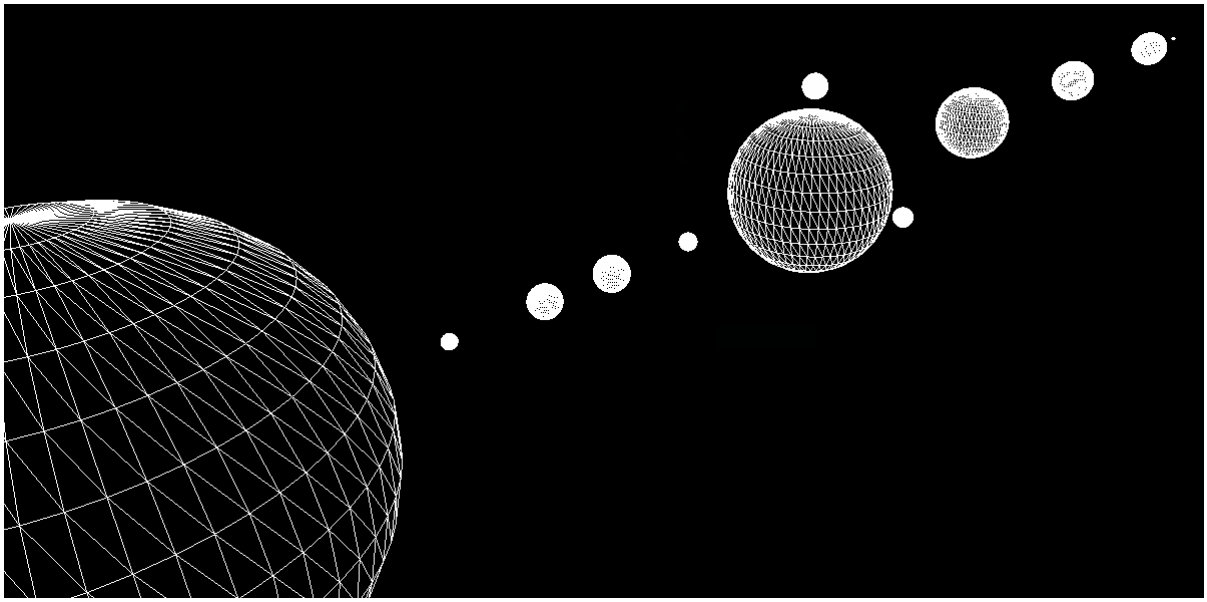


Figura 10 - Esquema do sistema solar obtido

Conclusão

De acordo com o objetivo definido na segunda fase consideramos que este foi alcançado. De facto, todos os propostos no enunciado foram conseguidos. Para além disso, foi ainda possível adicionar outros extras.

Assim, a presente fase do trabalho permitiu obter um conhecimento mais prático relativamente às diferentes transformações. Conseguimos entender a relevância da ordem nestas, bem como o seu impacto no desenho de diversas figuras.

Em suma, o presente trabalho tornou-nos mais objetivos e racionais sendo que foi uma mais valia para todos os elementos do grupo.

Bibliografia

- http://www.cplusplus.com/reference/cfloat/;