Computação Gráfica - 2.ª Fase

Dinis Lourenço Gomes
a87993@alunos.uminho.pt

Rui Miguel Gonçalves Dias
a89480@alunos.uminho.pt

Universidade do Minho - Departamento de Informática

1 Introdução

Nesta fase do trabalho tenciona-se implementar a possibilidade de desenhar *scenes* com transformações geométricas. Pelo que será necessário alterar a aplicação engine tanto na secção do parsing como na secção do desenho.

A *scene* passa agora a ser composta por *groups* que contêm cada um um conjunto de transformações geométricas (*translation*, *rotation* e *scale*), que apenas podem estar dentro dos grupos e afetam em simultâneo o grupo atual e os respetivos subgrupos, e um conjunto de *models*. Com a adição que cada grupo poderá ainda conter subgrupos ou grupos filho.

2 Implementação

Os tópicos seguintes vão descrever o raciocínio utilizado durante cada secção do trabalho.

2.1 Parsing do XML

Para o parsing dos ficheiros de XML é utilizado a biblioteca *tinyxml2*. Uma vez que é necessário agrupar as transformações geométricas e os modelos dentro de grupos foi utilizada uma estrutura ***Group***, referida no capitulo seguinte, para conter esta informação.

Como extra foi adicionada a cada modelo uma componente ***Color*** para se poder definir a respetiva cor a partir do *XML*.

Uma vez que já havia sido feito uma função de parse para os modelos, o raciocínio desta fase foi baseado em fazer uma função de parsing para cada transformação geométrica e, caso existisse, os valores default seriam atualizados e posteriormente associados ao grupo. Uma vez que o próprio grupo pode ter subgrupos, é necessário chamar a função ***parseGroup*** recursivamente. Havendo a necessidade de armazenar a ordem pela qual as transformações ocorrem, foi implementado um sistema de organização dentro de cada grupo.

O codigo seguinte demonstra o comportamento do parser sobre um grupo :

```
1  void parseGroup(XMLElement* fnode, vector<Group>& groups){
2      // Dentro de cada grupo
3      while(fnode){
4          // Default Construtors para as TGs
5          Translate t = Translate();
6          Rotate r = Rotate();
7          Scale s = Scale();
8
9          // Vetor de ordem das TGs
10         vector<char> order;
11
12         // Uma das TGs
13         XMLElement* fchild = fnode->FirstChildElement();
14
15         // Nome de nodo filho
16         char const* nodename;
17
18         // Flag que verifica se encontrou um model ou um grupo
19         int flag = 1;
20
21         // Ciclo que percorre os child elements ate chegar aos models
22         while(fchild && flag != 0){
23             // Passagem do nome do node para string para comparação
```

```
            nodename = fchild->Name();
            string stemp = nodename;

            // verifica se não e um model ou um grupo
            if(stemp != "models" && stemp != "grupo"){
                // Se for um translate
                if(stemp == "translate"){
                    // Executa o parsing do translate
                    t = parseTrans(fchild);

                    // Coloca Translate no vector de ordem
                    order.push_back('T');
                }

                // Se for um rotate
                else if(stemp == "rotate"){
                    // Executa o parsing do rotate
                    r = parseRot(fchild);

                    // Coloca Rotate no vector de ordem
                    order.push_back('R');
                }

                // Se for um scale
                else if(stemp == "scale"){
                    // Executa o parsing da scale
                    s = parseScale(fchild);

                    // Coloca Scale no vector de ordem
                    order.push_back('S');
                }
            }

            else{
                // Foi encontrado um models | group
                flag = 0;
            }

            // Seguir para o proximo sibling
            fchild = fchild->NextSiblingElement();
        }

        // Secção dos models
        vector<Model> models = parseModels(fnode);

        // Vector de subgroups
        vector<Group> subGroups;

        // Temp é o grupo dentro do grupo se existir
        XMLElement* temp = fnode->FirstChildElement("group");

        // Verifica se existe um grupo dentro do grupo
        if (temp){
            // Chama a parsing do grupo para os filhos do main group recursivamente
            parseGroup(temp, subGroups);
        }

      // Criação do grupo temporário e respetiva adição ao vetor final
        groups.push_back (Group(t, r, s, models, subGroups, order));

        // Avança para o grupo seguinte
        fnode = fnode->NextSiblingElement();
    }
}
```

2.2 Estruturas Utilizadas

1. **Model**

   Composto pelo nome do ficheiro e as coordenadas dos pontos necessários para a construção deste modelo. Decidiu-se como extra adicionar uma classe cor ao modelo.

   Estas coordenadas encontram-se guardadas como um vetor de tuplos como se pode observar no seguinte código:

```
#include <iostream>
#include <vector>
#include <tuple>
#include "color.cpp"
using namespace std;

class Model{
    public:
        string filename; //nome do file associado.
        Color color; //cor associada ao modelo
        vector< tuple<float, float, float> > coordenadas;

    // Default Constructor
    Model(string f, Color c, vector< tuple<float, float, float> > coord){
        filename = f;
        color = c;
        coordenadas = coord;
    }
};
```

2. **Color**

   Composto pelas componentes R, G, B necessárias para definir uma cor através da função ***glColor3f()***

```
class Color {
  public:
    float r, g, b; //RGB

    // Default Color -> Dark Orange
    Color(){
        r = 188.0f;
        g = 102.0f;
        b = 17.0f;
    }
};
```

3. **Translate**

   Composto pelas componentes de translação necessárias para a execução da função ***glTranslatef***, como se pode observar no seguinte código:

```
class Translate {
  public:
    float x, y, z; //Coordenada x, y, z

    // Default constructor
    Translate(){
        x = 0;
        y = 0;
        z = 0;
    }
};
```

4. **Rotate**

   Composto pelas componentes de rotação necessárias para a execução da função ***glRotatef***, como se pode observar no seguinte código:

```
class Rotate {
    public:
        float angle,            // Angle of rotation
        axisX, axisY, axisZ;    // Axis

    // Default Constructor
        Rotate(){
```

```
            angle = 0;
            axisX = 0;
            axisY = 0;
            axisZ = 0;
        }
};
```

5. **Scale**

   Composto pelas componentes de escala necessárias para a execução da função ***glScalef***, como se pode observar no seguinte código:

```
class Scale{
    public:
        float x, y, z;

    // Default Constructor
    Scale(){
        x = 1;
        y = 1;
        z = 1;
    }
};
```

6. **Group**

   Um grupo é então composto pelas estruturas referidas acima, associadas as respetivas transformações geométricas, um conjunto de grupos, neste caso subgrupos ou grupos filho e um vetor que contem as iniciais de cada transformação pela ordem que vão ser executadas.

```
class Group{
    public:
        // Componente de Translação do grupo
        Translate trans;

        // Componente de Rotação do grupo
        Rotate rot;

        // Componente de Escala do grupo
        Scale scale;

        // Vector de modelos dentro do grupo
        vector<Model> models;

        // SubGrupos dentro do grupo
        vector<Group> subGroups;

        // Order Vector - pode conter T | R | S
        vector<char> order;

    // Construtor
    Group(Translate t, Rotate r, Scale s, vector<Model> ms, vector<Group> sgs,
    vector<char> odr){
        trans = t;
        rot = r;
        scale = s;
        models = ms;
        subGroups = sgs;
        order = odr;
    }
};
```

2.3 Desenho das Transformações Geométricas

Em cada grupo é feito um ***glPushMatrix()*** ao entrar e um ***glPopMatrix*** ao sair do grupo, para evitar *resets* manuais do eixo.

Para desenhar as transformações geométricas em cada grupo utilizou-se a função ***drawGroup***, que acede ao vetor *order* para tomar conhecimento das transformações a ser executadas, na sua respetiva ordem. Ou seja, itera-se sobre o vetor e executa-se a devida transformação. Uma vez que um grupo pode conter subgrupos é necessário chamar a função recursivamente.

```
void drawGroup(Group group) {
    glPushMatrix();

    // Geometric Transformations
    for (vector<char>::const_iterator i = group.order.begin(); i != group.order.end();
    i++) {
        switch (*i) {
        // Translate
        case 'T':
            glTranslatef(group.trans.x, group.trans.y, group.trans.z);
            break;

        // Rotate
        case 'R':
            glRotated(group.rot.angle, group.rot.axisX, group.rot.axisY, group.rot.
    axisZ);
            break;

        // Scale
        case 'S':
            glScaled(group.scale.x, group.scale.y, group.scale.z);
            break;
        }
    }

    // desenhar os modelos
    drawFigures(group.models);

    // desenhar sub-grupos se existirem
    for (vector<Group>::const_iterator i = group.subGroups.begin(); i != group.
    subGroups.end(); i++) {
        drawGroup(*i);
    }
    glPopMatrix();
}
```

2.4 Sistema Solar

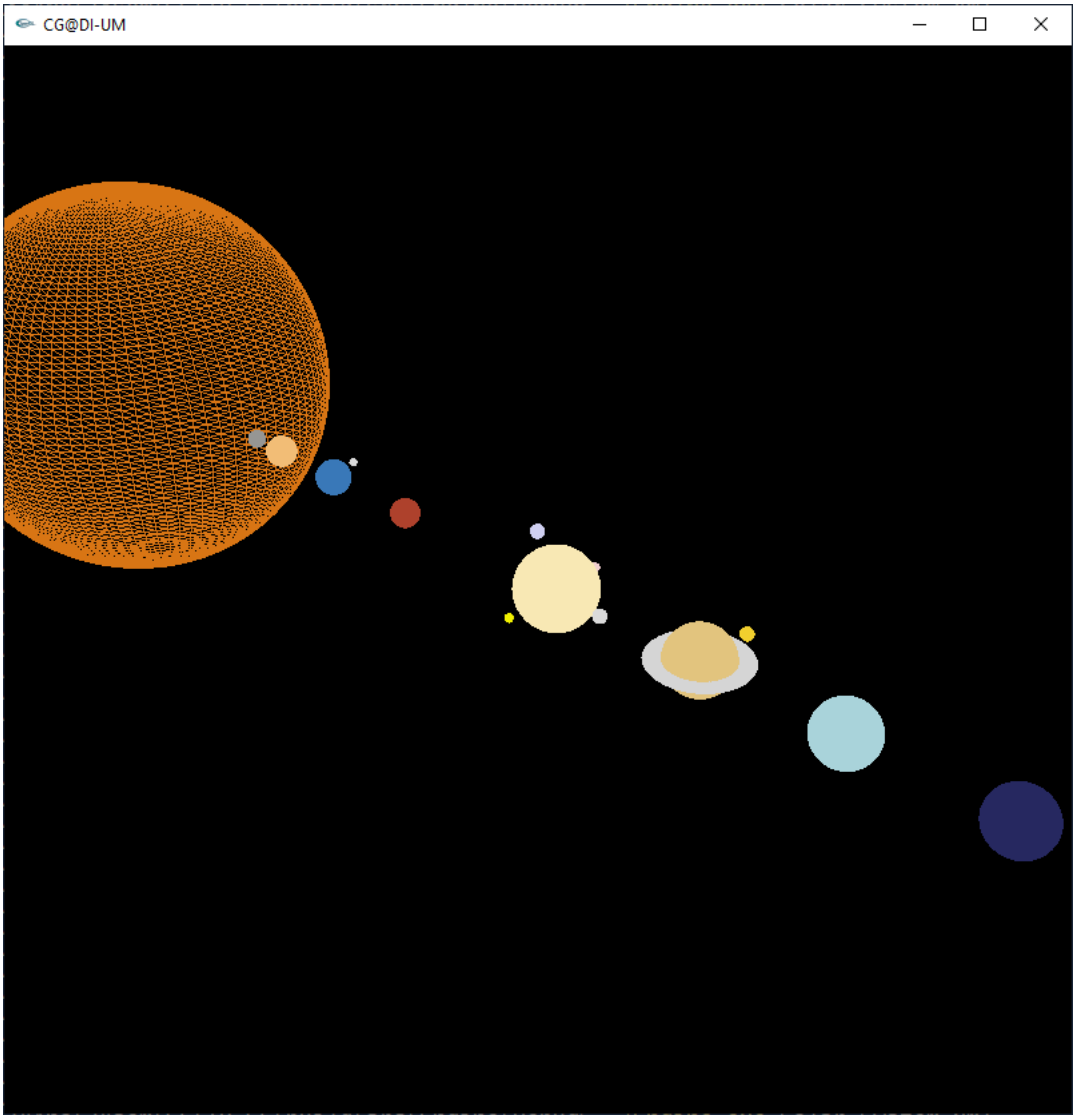

Figure 1: Solar System by G16

2.5 Conclusão

Esta fase do trabalho prático apesar de mais focada no parsing do ficheiro XML e no armazenamento dos dados obtidos através deste, é crucial para trabalho futuro uma vez que a fase 3 terá esta como base.