Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Fase 2 - Transformações Geométricas

Grupo 32

César Henriques - A64321

João Gomes - A74033

Rafael Magalhães - A75377

Tiago Fraga - A74092

31 de Março de 2017

Conteúdo

1 Introdução

Para esta fase, o motor da fase 1 deve ser melhorado de forma a ser capaz de criar cenas com transformações definidas por uma ordem hierárquica de grupos. A cena será configurada num ficheiro XML. O motor deve ler e interpretar o XML para guardar a informação necessária e, no fim, desenhar a cena. O objetivo final desta fase é criar uma representação estática do nosso Sistema Solar.

2 Motor

2.1 Classes

Nesta fase do trabalho prático, tivemos de recorrer a criação de classes e algumas estruturas, de modo a guardar os dados provenientes do ficheiro *XML*.

- **Classe *Group* :**

```
class Group {
       public:
             vector<Transformation*> transformations;
             vector<Primitive> models;
             vector<Group> childgroups;
       public:
             Group() {};
             ~Group() {};
             void drawGroup();
};
```

Esta classe é constituída por um *vector* de *Transformation* que têm como função armazenar todas as transformações, um *vector* de *Primitive* que guarda todos os modelos presentes no ficheiro *XML* e por último um *vector* de *Group* que armazena os restantes *Group* contidos no ficheiro. Contêm ainda três métodos sendo dois deles construtores da classe e *drawGroup* o método que vai desenhar, as figuras lidas, aplicando as respectivas transformações.

- **Classe *Transformation* :**

```
class Transformation {
// Varíaveis utilizadas nas transformações.
public:
      float angle, x, y, z;
public:
      virtual void transform() {};
      Transformation() {};
      virtual ~Transformation() {};
};

// classe que aplica os Translates.
class Translate : public Transformation {
public:
      Translate(float x, float y, float z);
      void transform();
};
Translate::Translate(float x, float y, float z){
    Translate::x = x;
```

```
    Translate::y = y;
    Translate::z = z;
}
void Translate::transform() {
    glTranslatef(x, y, z);
}

// Classe que aplica os Rotates.

class Rotate : public Transformation {
public:
      Rotate(float angle, float x, float y, float z);
  void transform();
};

Rotate::Rotate(float angle, float x, float y, float z) {
    Rotate::angle = angle;
    Rotate::x = x;
    Rotate::y = y;
    Rotate::z = z;
}

void Rotate::transform() {
    glRotatef(angle, x, y, z);
}

// Classe que aplica as escalas.
class Scale : public Transformation {
public:
    Scale(float x, float y, float z);
    void transform();
};
Scale::Scale(float x, float y, float z) {
    Scale::x = x;
    Scale::y = y;
    Scale::z = z;
}
void Scale::transform() {
    glScalef(x, y, z);
}
```

Nesta Classe estão definidas todas as transformações, embora nesta fase do trabalho prático não seja necessários *Rotates*, encontram-se ja implementados, diminuindo o trabalho das fases prosteriores.

2.2 Processo de Leitura

Nesta fase tivemos que elaborar algumas melhorias no processo de leitura do ficheiro *XML*, de modo a conseguir alcançar os objectivos propostos. No desenvolvimento deste processo tivemos que acrescentar alguns métodos, de forma a

guardar todos os dados nas respectivas classes, explicadas no ponto anterior.

- **Método *loadXML* :**
  Este método já implementado na primeira fase do trabalho prático serve para ler e guardar os dados do ficheiro *XML*, com a particularidade de invocar o método *loadGroup*.

```
int loadXML() {
    XMLDocument doc;

    XMLError eResult = doc.LoadFile("config.xml");
    XMLCheckResult(eResult);

    XMLNode * pRoot = doc.RootElement(); //Obter o nodo scene
    if (pRoot == nullptr) return XML_ERROR_FILE_READ_ERROR;
//Obter o primeiro elemento group
    XMLElement * groupList = pRoot->FirstChildElement("group");
    if (groupList == nullptr) return XML_ERROR_PARSING_ELEMENT;

    // Aqui encontra-se a chamada do método referido anteriormente.
    return loadGroup(scene, groupList);
```

- **Método *loadGroup***

  Este método vai guardar os dados dentro da tag *Group* nas respectivas classes.

```
// Recebe como parâmetros um argumento da Classe Group, um apontador para
// um XMLElement com
int loadGroup(Group &main_group, XMLElement * root_elem) {
    int return_value = 1;
    //retorna o primeiro elemento do grupo
    XMLElement* elem = root_elem->FirstChildElement();
    // Controlo de erros.
    if (elem == nullptr) return XML_ERROR_PARSING_ELEMENT;

    while(elem != nullptr){
        // Em caso de houver um childgroup , invoca novamente a função de modo
        //a guardar os dados que esse childgroup contêm.
        if (strcmp(elem->Name(), "group") == 0) {
            Group childGroup;
            loadGroup(childGroup, elem);
            main_group.childgroups.push_back(childGroup);
        }
        // guarda os translates com ajuda de um método auxiliar.
        else if (strcmp(elem->Name(), "translate") == 0)
             return_value *= storeTranslate(main_group, elem);
        // guarda todos os rotates.
        else if (strcmp(elem->Name(), "rotate") == 0)
             return_value *= storeRotate(main_group, elem);
```

```
        // guarda todas as escalas
        else if (strcmp(elem->Name(), "scale") == 0)
              return_value *= storeScale(main_group, elem);
         // guarda todos os modelos
        else if (strcmp(elem->Name(), "models") == 0)
             return_value *= storeModels(main_group, elem);

        if (return_value != 1) return return_value;
        elem = elem->NextSiblingElement();
    }
    return 1;
```

Este método representa todo o processo de leitura, sendo um dos que foi adicionado em relação a primeira fase.

- **Método *storeTranslate*** Este método faz o parse da a instrução *translate* dada no *XML*, transformando a string num float e guardando, criando assim um novo objecto da classe *Translate*, para mais tarde ser aplicada esta transformação.

```
int storeTranslate(Group &store_group, XMLElement * store_elem) {
    float x = 0, y = 0, z = 0;
    const XMLAttribute *attribute = store_elem->FirstAttribute();
    // Controlo de Erros
    if (!attribute) return XML_ERROR_PARSING_ATTRIBUTE;

    while (attribute != nullptr) {
     // Comparação de modo a obter os valores em caso de sucesso.
        if (strcmp((attribute->Name()), "X") == 0 || strcmp((attribute->Name()), "x")
            x = stof(attribute->Value());
        else if (strcmp((attribute->Name()), "Y") == 0 || strcmp((attribute->Name()),
            y = stof(attribute->Value());
        else if (strcmp((attribute->Name()), "Z") == 0 || strcmp((attribute->Name()),
            z = stof(attribute->Value());
        attribute = attribute->Next();
    }
    // Criação do novo objecto da Classe Translate.
    Translate *translate = new Translate(x,y,z);
    // Objecto guardado na estrutura.
    store_group.transformations.push_back(translate);
    return 1;
}
```

Quer os métodos ***storeScale***,***storeModels*** e ***storeRotate*** o seu funcionamento, e a sua implementação são idênticas a este método apresentado, a diferença em relação a este método encontra-se no *storeRotate*, visto ter também a variável de ângulo da rotação para armazenar.

- **Método *loadModels* :** Este método está encarregue de guardar todos os modelos definidos no ficheiro *XML*, sendo um método da primeira fase do trabalho prático.

```
int loadModels(Group &group, vector<string>& list) {
    string line;

    for (auto const& fileName : list) {
        // Estrutura onde é guardado o elemento a desenhar.
        Primitive primitive;
        ifstream reader(fileName);
        if (!reader) {
        return 0;
        }
        else {
            while (getline(reader, line))
               primitive.push_back(line);
        }
        cout << "Loaded Primitive: " << fileName << endl;
        // guardar as primitivas na estrutura dos modelos na Classe Group.
        group.models.push_back(primitive);
     }
     return 1;
}
```

2.3 Ciclo de Rendering

Depois de tudo guardado nas respectivas estruturas, precisamos de ler as estruturas começando pelas Transformações, depois de aplicadas desenhar o modelo e para isso foi definido um método da classe *Group*.

```
void Group::drawGroup() {
     // Guarda a Matriz inicial, antes de ser aplicado as transformações.
     glPushMatrix();
     // Aplica as transformações guardadas.
     for (auto &transformation : transformations) transformation->transform();
     // Desenha os modelos guardados.
     for (auto const &model : models) drawPrimitive(model);
     // Desenha os restantes Group guardados.
     for (auto &child : childgroups) child.drawGroup();
     // Retorna a Matriz inicial.
     glPopMatrix();
}
```

Este método depois de guardar a matriz inicial, aplica as transformações guardadas e só depois desenha o modelo, se existirem *childgroups* faz uma chamada recursiva para esses grupos. Assim, garantimos que as transformações aos *childs* são feitas por cima das transformações do grupo pai.

3 Demo sistema solar

O ficheiro *XML* está organizado da seguinte forma, temos um grupo para o Sol e dentro deste grupo temos os grupos dos planetas e quando os planetas possuem luas, elas encontram-se definidas dentro do grupo do planeta que as possui. Inicialmente definimos as escalas para desenhar os planetas conforme escalas reais, mas como os desenhos iam ficar grandes demais para representar no ecrã, optamos por definir escalas que nos permiti-se representar no ecrã e ao mesmo tempo corresponder a realidade, sendo assim ficamos com Sol como a maior esfera desenhada e Júpiter como o maior planeta desenhado, sendo os restantes representados da mesma forma. Os valores utilizados nos *translates* foram criados de modo a que fosse visível a que planeta pertencia cada lua representada.

```
<?xml version="1.0" encoding="UTF-8"?>
     <scene>
         <group>
         // Translate para que a representação da scene apareça totalmente no ecrã.
           <translate x="-50.5"/>
            <!--Sol-->
            <scale x="5.5" y="5.5" z="5.5"/>
            <models>
                   <model file="sphere.3d"/>
            </models>
            <!--Terra-->
            <group>
                <translate x="7.5"/>
                <scale x="0.25" y="0.25" z="0.25"/>
                <models>
                       <model file="sphere.3d"/>
                 </models>
                <!--Lua-->
                  <group>
                    <translate x="1.5" z="1.5"/>
                    <scale x="0.15" y="0.15" z="0.15"/>
                    <models>
                           <model file="sphere.3d"/>
                    </models>
                  </group>
           </group>
     </scene>
```

De modo a mostrar o que foi explicado a cima, colocamos parte do nosso ficheiro, mostrando a hierarquia criada em que o Sol engloba todos os planetas e em que os planetas que possuem luas, englobam as mesmas.

Figura 1: Sistema Solar

4 Conclusão

Podemos concluir que os objetivos para esta fase do trabalho prático foram alcançados. O nosso motor gráfico é agora capaz de ler ficheiros XML mais complexos que contenham translações, rotações e escalas ordenadas por grupos que serão aplicadas de forma hieraquica a um ou mais modelos.