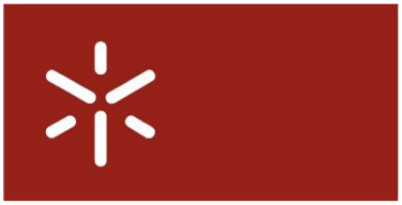

Computação Gráfica

Mestrado Integrado em Engenharia Informática

Fase 2

Transformações Geométricas

GRUPO 19 - PL2

A85227 João Pedro Rodrigues Azevedo

A85729 Paulo Jorge da Silva Araújo

A83719 Pedro Filipe Costa Machado

A89983 Paulo Jorge Moreira Lima

Braga

Março 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Este relatório é relativo ao trabalho prático da UC de Computação Gráfica que tem como objetivo final desenvolver uma *engine* para resolução de modelos **3D**, mostrando o seu potencial através da apresentação final de um sistema solar, na fase 4, com texturas e luzes.

1.2 Objetivos para a fase 2

A segunda fase teve como principais objetivos a criação de cenários hierárquicos, no qual uma *scene* seria definida a partir de uma árvore onde cada nodo contém um conjunto de transformações geométricas (translação, rotação e escala) e, opcionalmente, um conjunto de modelos. Cada nodo (**group**) pode ser constituído por vários nodos filho.

Sendo assim, para esta fase pretende-se acrescentar funcionalidades ao leitor de *xml* de modo a que consiga processar novos elementos e correr no *engine* um ficheiro de configuração capaz de representar um modelo do Sistema Solar.

Capítulo 2

Atualização do *Engine*

2.1 Plano de trabalho

Em primeiro lugar, foi necessário estabelecer um plano de trabalho para esta fase e, deste modo, procedemos à realização das seguintes tarefas:

1. Estabelecer **novas estruturas de dados** para os novos elementos (*groups*, *models* e transformações);
2. **Atualizar a função de carregamento do ficheiro de configuração** de modo a suportar esta nova estrutura;
3. Criação de um ficheiro de configuração ***xml* capaz de apresentar um Sistema Solar**.

2.2 Estruturas de dados

Para estabelecer quais seriam as melhores estruturas para guardar as transformações e modelos a desenhar foi preciso observar a nova estrutura do ficheiro *xml*. Observemos o exemplo seguinte:

```
<scene>
    <group>
        <translate X=5 Y=0 Z=2 />
        <rotate angle=45 axisX=0 axisY=1 axisZ=0 />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>
</scene>
```

A partir deste exemplo, e de muitos outros apresentados no enunciado do trabalho, percebemos que a *scene* principal tem como nodos filho um ou mais *group*. Por outro lado, cada *group* constitui um conjunto de transformações [1] aplicáveis a todos os elementos dentro do *group*, um conjunto de modelos que já foram analisados na fase anterior e também outros *group*.

Deste modo, a nossa estrutura de dados não seria muito diferente. Estabelecemos como estrutura principal um *Group* que conteria como variáveis de instância um vetor de transformações, um vetor modelos e um vetor de outros *Group*.

[1] *Translate, Scale* e *Rotate*.

Na prática, criámos as seguintes classes C++ (que se encontram armazenadas num *header* chamado **model-info.h**):

- **Group**: Classe principal que armazena todos os **modelos**, **transformações** e outros **Groups** lidos do ficheiro.

```
class Group {

    public:

        //Sequence of transformations
        vector<Transformation>* transformations;

        //List of models: inside <models> tag
        vector<MODEL_INFO>* models;

        //List of models: inside <group> tag
        vector<Group>* groups;

};
```

- **Transformation**: Classe que representa, de forma geral, uma transformação que será constituída por 3 variáveis principais **x, y e z** no caso de uma Translação e uma Escala, já no caso de uma Rotação temos uma variável adicional **angle**.

```
class Transformation {

    public:
        //Constructor for Rotate
        Transformation(const string &description, float x,
                                                  float y,
                                                  float z,
                                                  float angle);
        //Constructor for Translate and Scale
        Transformation(const string &description, float x,
                                                  float y,
                                                  float z);

    public:
            //transformation name
            string description;
            float x, y, z;
            float angle;
    };

typedef class Transformation Rotation, Translation, Scale;
```

Assim, como se pode ver no código apresentado acima, temos dois construtores diferentes para cada caso e temos também definido um tipo de dados comum **Transformation** para os três tipos de transformação com o objetivo único de facilitar a leitura e escrita do código.

Na classe **Group** armazenamos estas transformações num vetor de modo a, posteriormente, aplicá-las no OpenGL de forma ordenada.

- **MODEL_INFO**: Com já foi referido na fase anterior, esta classe armazena o nome do modelo e o respetivo vetor de vértices carregado para memória, sendo um vértice representado pela classe **POINT_3D**.

```
class MODEL_INFO {

    public:
        string name;
        vector<POINT_3D> vertices;

};
```

- **MODEL_INFO**: Com já foi referido na fase anterior, esta classe armazena o nome do modelo e o respetivo vetor de vértices carregado para memória, sendo um vértice representado pela classe **POINT_3D**.

2.3 Atualização da função de leitura do ficheiro *xml*

Tendo as classes criadas procedemos à atualização do procedimento de leitura das *tags* presentes no ficheiro de configuração.

Como explicitado na primeira fase, para fazer o *parsing* do ficheiro *xml* utilizamos as funções disponibilizadas pelo *software tinyxml2*. Todo o Pseudocódigo a seguir mostrado visa suportar o código desenvolvido no módulo **load-xml.cpp/.h** presente na pasta *engine/Models*.

Vamos então explicar o processo de leitura adotado:

```
//Funcao principal que carrega o xml para um vetor de Groups
vector<Group>* load_xml_config(string xml_config_filename);
```

1. Encontrar a *root tag* `<scene>` (caso não exista reportar o ficheiro como inválido):

   ROOT = FILE.PrimeiroFilho("scene");

2. Encontrar a primeira *tag* `<group>` e percorrer `<scene>` apenas de `<group>` em `<group>`:

(ver na próxima página)

```
tagGroup = ROOT -> PrimeiroFilho("group")

enquanto tagGroup valida

    novo = new Group()
    processaGroup(tagGroup, novo)
    adicionaGrupo(novo)
    tagGroup = tagGroup -> ProximoFilho("group")

fim enquanto
```

3. Processar a *tag* `<group>` (procedimento **processaGroup()**):

```
    filhoTag = tagGroup -> PrimeiroFilho()

    enquanto filhoTag valida

        se Nome(filhoTag) == translate
            processTranslationTag()

        senao se Nome(filhoTag) == rotation
            processRotationTag()

        senao se Nome(filhoTag) == scale
            processScaleTag()

        senao se Nome(filhoTag) == models
            processModelsTag()

        senao se Nome(filhoTag) == group
            novoG = new Group()
            processaGroup(filhoTag, novoG) //processo recursivo
            adicionaGrupo(novoG)

        filhoTag = filhoTag -> ProximoFilho()

    fim enquanto
```

4. Processar a *tag* `<models>` (procedimento **processModelsTag()**):

```
    tag = modelsTag -> PrimeiroFilho("model")

    //Percorrer todos os <model> em <models>
    enquanto tag valida

        model = new MODEL_INFO()
        model.nome = tag -> atributo("file")
        model.vertices = new vector<POINT_3D>()
        //Carregar vertices
        load_model(model)

    fim enquanto
```

Nota: O processo de carregar vértices, mais propriamente o procedimento **load_model_vertices()** é igual ao desenvolvido na primeira fase pelo que não será explicado outra vez.

5. Processar as tags `<translate>`, `<rotate>` e `<scale>`:

   - `<translate>`: Criar um novo objeto **Transformation** com o tipo **Translation** e inserir os valores de **x, y e z** da *tag*:

     ```
     Transformation t = new Translation("Translation", x, y, z);

     t->x = tag -> consultarAtributo("X");
     t->y = tag -> consultarAtributo("Y");
     t->z = tag -> consultarAtributo("Z");

     group -> adicionaTransformacao(t)
     ```

   - `<scale>`: Criar um novo objeto **Transformation** com o tipo **Scale** e inserir os valores de **x, y e z** da *tag* (processo semelhante ao de cima).
   - `<rotate>`: Criar um novo objeto **Transformation** com o tipo **Rotation** e inserir os valores de **x, y, z e angle** da *tag*:

     ```
     Transformation t = new Rotation("Rotation", x, y, z, angle);
     //(...igual...)
     t->angle = tag -> consultarAtributo("angle");
     group -> adicionaTransformacao(t)
     ```

Após esta breve explicação aproveitamos para referir que todo o código referido acima se trata de pseudocódigo, sendo que alguns procedimentos escritos podem não existir com a assinatura referida acima que serviu apenas para ilustrar o processo.

De notar também que durante a leitura do ficheiro qualquer erro detetado pode parar o processo de leitura do ficheiro xml sendo aprensentado um aviso no *stdout*, como por exemplo, a *tag* raiz não ser a `<scene>`.

2.4 Atualização da função de geração de gráficos com OpenGL

Após atualizar as estruturas e a função que lendo do ficheiro *xml* as atualiza com as transformações, modelos e outros agrupamentos dos referidos anteriormente, partimos para a penúltima tarefa: atualizar a função de renderização do nosso cenário gráfico usando OpenGL.

Deste modo, a leitura da estrutura deve ser ordenada, isto quer dizer que, as transformações devem ser aplicadas pela ordem que foram lidas de modo a evitar misturar as matrizes associadas a cada passo da geração gráfica.

Todo o código C++ referente a esta secção encontra-se no módulo **load-graphics.cpp** e **draw-elements.cpp**. Segue-se então o processo de leitura da estrutura adotado:

1. Ler todos os **group** do vetor de groups principal e carregar os grupos, um a um, utilizando a função:

   ```
   drawGroupElements(group);
   ```

2. Desenhar um **group** (drawGroupElements()):

   ```
   glPushMatrix()

   transformations = group -> trasnformations
   models = group -> models
   groups = group -> groups
   ```

```
//Aplicar transformacoes
para cada T em trasnformations
    se Nome(T) == "Translation"
        glTranslate3f(T -> x, T -> y, T -> z)
    se Nome(T) == "Scale"
        glScale3f(T -> x, T -> y, T -> z)
    se Nome(T) == "Rotation"
        glRotate3f(T -> angle, T -> x, T -> y, T -> z)
fim para

//Desenhar modelos
para cada M em models
    //Desenhar os triangulos -> (fase 1)
    drawModelVertices(M)
fim para

//Desenhar modelos
para cada G em groups
    //chamada recursiva
    drawGroupElements(G)
fim para

glPopMatrix()
```

No código aprensentado acima, o mais importante foi ter adicionado as funções **glPushMatrix()** e **glPopMatrix()** de modo a que as transformações aplicadas a cada **group** sejam apenas para o mesmo e não interfiram com o resto dos elementos fora desse grupo.

Por outro lado, a ordem das trasnformações também foi importante ter sido seguida, para cumprir o objetivo estabelecido pelas regras do ficheiro *xml* desta fase.

De notar também que os vértices do modelo e os respetivos triângulos foram desenhados da mesma forma que a descrita na 1ª fase do trabalho pelo que não achamos necessário descrever o processo de leitura do modelo de novo.

2.5 Outras alterações ao programa *Engine*

Foi adicionado também a possibilidade de inserir um ficheiro *xml* manualmente no programa: **./engine ficheiro.xml** ou usar o *default* presente na pasta **examples/XML-Examples** pelo que todos os ficheiros a utilizar devem ser colocados nessa pasta.

2.6 Ficheiro de configuração XML para o Sistema Solar

Para o desenvolvimento do sistema solar consideramos uma composição de vários agrupamentos de esferas na configuração do ficheiro **xml**.

Inicialmente, a esfera que representa um planeta foi gerada com raio 1 através do programa **generator** desenvolvido na Fase 1.

Devemos dizer que a configuração final não segue estritamente a escala real tratando-se, na verdade, de uma aproximação à mesma de modo a que seja possível visualizar todos os planetas no ecrã (utilizando para isso **pgup** e **pgdn** para alterar o *zoom*).

Relativamente aos satélites naturais dos planetas optamos por apresentar nesta fase apenas um (no caso de o ter).

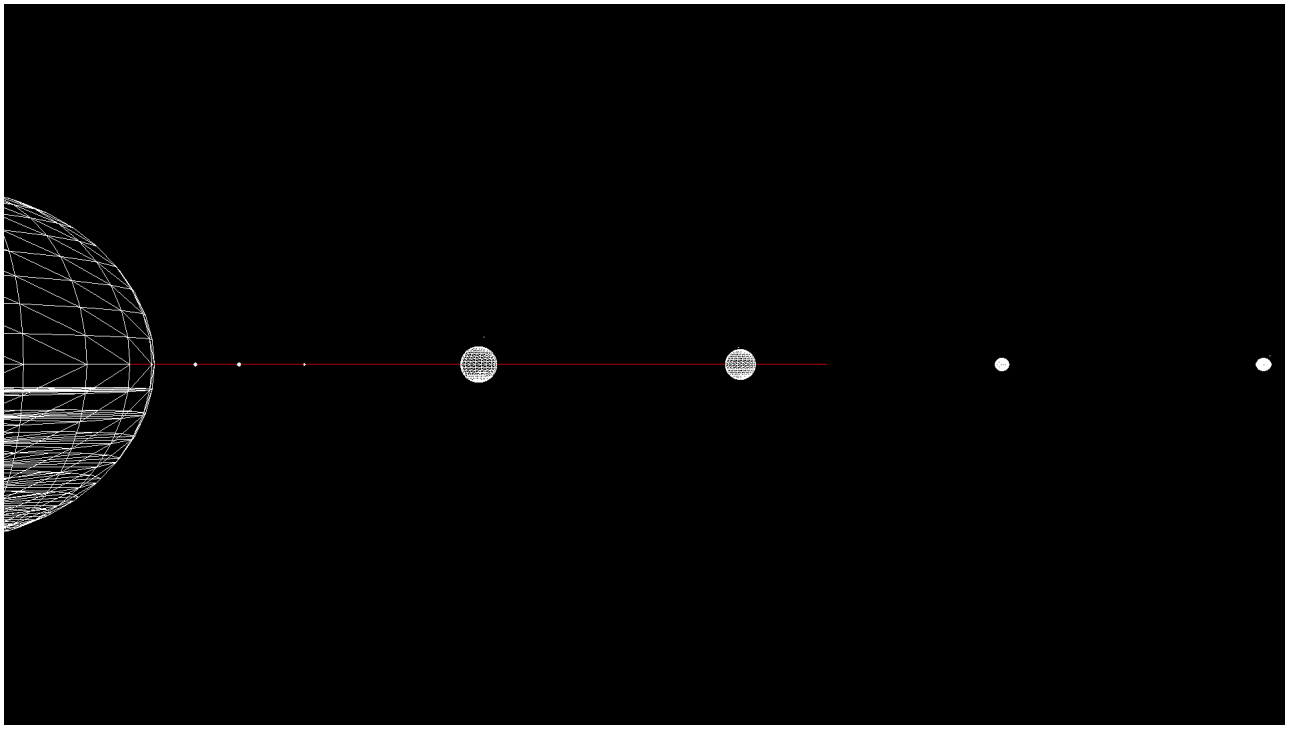

Figura 2.1: Sistema solar

Capítulo 3

Conclusão

A segunda fase do trabalho possibilitou um maior processamento por parte do ficheiro de configuração *xml*.

Assim, foi possível aplicarmos transformações (*rotate*, *scale*, *translate*) aos modelos definidos na fase anterior, pelo que, permitiu construir *scenes* mais interessantes, tais como o Sistema Solar, como era pedido para esta fase.

Em suma, esta fase relevou-se fulcral para o desenvolvimento do projeto com fim à criação de um Sistema Solar, pois permitiu a criação deste sistema a nível de transformações geométricas, possibilitando um nível visual deste. Embora que ainda esteja numa fase inicial, este sistema serve como bom ponto de partida para as próximas duas fases.

Conclui-se então que os objetivos definidos para a segunda fase foram cumpridos.