Universidade do Minho – 2016/2017

Relatório de Computação Gráfica

Fase 2 – Transformações Geométricas

Grupo 10:
Carlos Pereira, a61887
João Barreira, a73831
Rafael Braga, a61799

Índice

1. Introdução

Esta segunda fase consiste no processamento de ficheiros *XML* e aplicação de várias operações em *OpenGL* (translações, rotações e escalas).

Neste relatório constará uma descrição pormenorizada da *Engine*, da fase de leitura e extração das informações contidas nos ficheiros *XML* e o desenho das cenas com o auxílio de eventuais primitivas gráficas.

No final deste relatório, apresentaremos alguns exemplos de execução a partir de vários modelos em ficheiros *XML*.

2. Descrição do programa *Engine*

O programa *Engine* está dividido em duas fases: leitura e extração de dados de ficheiros *XML* e o desenho das cenas com o auxílio de eventuais primitivas gráficas.

Na primeira fase, o programa lê os ficheiros *XML*, extraindo as suas informações, nomeadamente o que diz respeito aos grupos, aos modelos e às outras operações em *OpenGL* (translações, rotações e escalas).

Na segunda fase, o programa encarrega-se de desenhar a cena descrita no ficheiro *XML* através de um ciclo de *rendering* e com o recurso a um vetor relativo à totalidade das operações que constam no ficheiro lido.

Além disso, o programa possui ainda funcionalidades relativas ao movimento da cena, como por exemplo:

- deslocação de um modelo com as teclas *w*, *a*, *s* e *d*;
- rotação de um modelo com as setas do teclado;
- movimentação da câmara com o rato;
- definição de opções em *OpenGL* relativas ao *face culling* (*GL_BACK*, *GL_FRONT* e *FRONT_AND_BACK*) e ao desenho dos modelos (*GL_POINT*, *GL_LINE* e *GL_FILL*), através de um menu na janela de execução.

3. Processamento de um modelo em formato *XML*

O processamento de um modelo em formato *XML* pode ser visto como duas fases distintas:

- Leitura e *Parsing* do modelo – Consiste na abertura do ficheiro que contém o modelo em modo de leitura e extração da respetiva hierarquia em *XML.* Nesta fase são também retiradas as componentes que caraterizam uma transformação geométrica (translação, escala ou rotação) ou desenho de uma primitiva.
- Armazenamento nas estruturas de dados – De modo a se conseguir redesenhar um modelo quantas vezes for necessário, as instruções que caraterizam uma transformação geométrica são armazenadas em estruturas adequadas às mesmas.

Para uma melhor organização do código, optou-se por separar estas duas fases do programa *Engine* num conjunto de módulos apropriados às mesmas. As duas subsecções seguintes descrevem com detalhe a organização desses módulos.

3.1. Leitura e *Parsing*

A leitura e *parsing* de um ficheiro *XML* que contenha um modelo é efetuada pelo módulo *xmlParser*. Para o *parsing* propriamente dito, é utilizada a ferramenta *tinyXML*. Assim, o ficheiro é lido apenas uma só vez e a sua hierarquia (conjunto de nodos e respetivos atributos) são tratados por esta ferramenta. Com a hierarquia definida basta extraírem-se os diferentes componentes que caraterizam uma transformação geométrica.

O principal objetivo do módulo *xmlParser* consiste na criação de um vetor de operações em *OpenGL* (mais detalhes na secção **3.2**). Uma operação em *OpenGL* é criada a partir dos vários atributos que constituem um nodo. É importante referir que, caso a sintaxe de uma instrução *XML* seja inválida, é atribuído o valor verdadeiro a uma variável chamada *invalidDoc* e o *parsing* termina. Neste caso, uma mensagem de erro apropriada será armazenada numa *string,* com o nome *errorString,* e o vetor de operações será vazio.

Apresentam-se de seguida os pseudocódigos responsáveis pelo tratamento de cada nodo do ficheiro *XML* e respetiva extração dos seus atributos para a criação do vetor de operações em *OpenGL.*

3.1.1. Translações, escalas e rotações

Uma translação e uma escala apresentam, no máximo, três atributos distintos (componentes *x, y* e *z*). Assim, o processo do armazenamento da informação de uma translação ou de uma escala é o seguinte:

```
Para cada atributo a enquanto invalidDoc == falso fazer a = proximoAtributo() {
        Se a == componenteX {
                x = valorAtributo(a)
        }
        Senão se a == componenteY {
                y = valorAtributo(a)
        }
        Senão se a == componentZ {
                z = valorAtributo(a)
        }
        Senão {
                mensagemErro()
                invalidDoc = verdadeiro
        }

        adicionarVetorOperações(x, y, z)
}
```

Uma rotação apresenta o mesmo algoritmo descrito acima, apenas acrescentando uma componente que caracteriza o ângulo desta.

3.1.2. Modelos

As operações de desenho de triângulos são compostas por um conjunto de vértices e três valores reais que consistem nas tonalidades em vermelho, verde e azul desses vértices. Estes dados podem ser adquiridos através dos nodos do tipo *models* e *model.* Um nodo *models* pode conter inúmeros nodos do tipo *model,* mas não de outro tipo. O algoritmo para este tipo de nodos é, portanto, uma travessia pelos seus nodos filhos.

```
Para cada subnodo m enquanto invalidDoc == falso fazer m = proximoSubNodo() {
        Se m == model {
                parsingModelo(m)
        }
        Senão {
                mensagemErro()
                invalidDoc = verdadeiro
        }
}
```

O nodo *model* dispõe o nome do ficheiro onde estão guardados os vértices de uma primitiva, bem como as tonalidades dessa primitiva. Cada nodo do tipo *model* apenas pode ter a si associado um nome de um ficheiro.

```
numCores = 0
diffR = 0
diffG = 0
diffB = 0
numModels = numModels + 1

Para cada atributo a enquanto invalidDoc == falso fazer a = proximoAtributo() {
        Se a == file {
                lerVertices(valorAtributo(a))
        }
        Senão se a == vermelho {
                numCores = numCores + 1
                diffR = valorAtributo(a)
        }
        Senão se a == vermelho {
```

```
                numCores = numCores + 1
                diffG = valorAtributo(a)
        }
        Senão se a == azul {
                numCores = numCores + 1
                diffB = valorAtributo(a)
        }
        Senão {
                mensagemErro()
                invalidDoc = verdadeiro
        }
}
```

Caso não sejam fornecidas tonalidades, temos como a cor branca como cor do modelo por defeito:

```
Se numCores == 0 {
        diffR = diffG = diffB = 1
}

adicionarVetorOperações(vértices, diffR, diffG, diffB)
```

Relativamente ao nodo *model* sempre que a leitura do seu ficheiro não é feita com sucesso o *parsing* não termina. Em vez disso, a *string errorString* recebe uma mensagem de *warning* a indicar que não se conseguiu ler o ficheiro *<nome do ficheiro>.* Como não faz sentido o desenho de uma cena sem modelos, sempre que é lido um modelo uma variável *numModels* é incrementada. Por sua vez, sempre que não se consegue abrir o ficheiro relativo a um modelo uma variável de nome *failedModels* é incrementada. Assim, pode-se obter informação de quantos modelos foram carregados com sucesso ou não.

3.1.3. *Scenes* e *Groups*

Os nodos do tipo *scene* e *group* apresentam comportamentos muito semelhantes. Ambos podem possuir nodos que definem transformações geométricas, ou até mesmo nodos do tipo *group.* A única exceção consiste no facto de apenas haver um nodo *scene* por ficheiro *XML* e este ser a raiz. Por simplificação, sejam estes dois tipos de nodos um tipo

mais genérico denominado *container.* Sempre que se inicia a travessia num nodo deste tipo, deve ser efetuada uma *pushMatrix()* em *OpenGL.* Por sua vez, depois de se terem executado todas as operações relativas aos nodos filhos de um *container* deve ser efetuada uma operação de *popMatrix()* em *OpenGL.*

Um *container* tem a si associado dois tipos de erros especiais. As operações de translação, rotação ou escala devem ser definidas no início deste, já que estas operações afetam todos os seus filhos. Além disso, não deve possuir mais que uma *tag* do tipo *models, translate, scale* ou *rotate.* Para o tratamento destes erros usaram-se cinco *stacks* em *C++*. Quatro destas *stacks* guardam a informação acerca da existência, ou não, de uma operação geométrica (translação, rotação ou escala) e de uma *tag* models num *container.* A última *stack* regista um valor booleano que indica se uma operação geométrica está definida no início de um *container* ou não. Sempre que se efetua o *parsing,* são adicionadas aos topos das *stacks* o valor falso, ou seja, assume-se no início que um *container* não possui qualquer transformação geométrica ou modelos. Por sua vez, sempre que se termina o *parsing,* os topos das *stack* são retirados. Ou seja, quanto mais em baixo um *container* estiver na hierarquia do ficheiro *XML,* mais no topo das *stacks* este estará.

O algoritmo que descreve o processo do *parsing* de nodos, *scene* e *group,* e respetivas operações é o seguinte:

adicionarVetorOperações(pushMatrix())

pushContainerStacks()

Para cada tag *t* enquanto *invalidDoc == falso* fazer *m = proximaTag() {*
    parseTag(t)
}

popContainerStacks()

adicionarVetorOperações(popMatrix())

3.1.4. Seleção dos diferentes tipos de nodos

O seguinte pseudocódigo mostra apenas como são processados os diferentes tipos de nodos:

Se ***tag == models {***

Regista a informação de que um *container* já possui uma *tag* qualquer

Se ***topoSomethingInContainer() == falso {***

topoSomethingInContainer = verdadeiro

}

Se ***topoModelsInContainer() == verdadeiro {***

mensagemErro()

invalidDoc = verdadeiro

}

Senão ***{***

aumentarModelosNaStack()

parseModels(tag)

}

}

Senão se ***tag == group {***

Regista a informação de que um *container* já possui uma *tag* qualquer

Se ***topoSomethingInContainer() == falso {***

topoSomethingInContainer = verdadeiro

}

parseGroup(tag)

}

Senão se ***tag == translate {***

Se ***topoTranslatesInContainer() == verdadeiro*** **ou**

topoSomethingInContainer == verdadeiro {

mensagemErro()

invalidDoc = verdadeiro

}

Senão ***{***

topoTranslatesInContainer = verdadeiro

parseTranslate(tag)

```
        }
    }
    Senão se tag == rotate {
        Se topoRotatesInContainer() == verdadeiro ou
          topoSomethingInContainer == verdadeiro {
            mensagemErro()
            invalidDoc = verdadeiro
        }
        Senão {
            topoRotatesInContainer = verdadeiro
            parseRotate(tag)
        }
    }
    Senão se tag == scale {
        Se topoScalesInContainer() == verdadeiro ou
          topoSomethingInContainer == verdadeiro {
            mensagemErro()
            invalidDoc = verdadeiro
        }
        Senão {
            topoScalesInContainer = verdadeiro
            parseScale(tag)
        }
    }
```

3.2. Otimizações e estruturas de dados

Como foi mencionado anteriormente, um ficheiro *XML* é apenas lido uma vez, sendo a sua hierarquia preservada através da ferramenta *tinyXML.* Para além disso, através de uma variável do tipo booleano, a extração de todos os dados dessa hierarquia termina mal ocorra algum erro. Com o uso das *stacks* provenientes da linguagem *C++* foi possível tratar de erros complexos (erros de translações, rotações e escalas mal posicionadas numa *scene* ou num *group)* de uma maneira eficiente, já que a inserção e remoção de dados nestas estruturas executa em tempo constante. Uma otimização efetuada a nível da leitura dos ficheiros de vértices foi conseguida à custa de uma estrutura de dados, também

proveniente da linguagem *C++*, do tipo *map.* Esta estrutura armazena em cada posição uma chave (neste caso escolheu-se uma *string* com um nome de um ficheiro de vértices) que está associada a outro tipo de dados (um vetor de vértices). Assim, antes de se proceder à leitura de um ficheiro de vértices verifica-se se este já foi lido e se encontra no *map*. Caso isto seja verdade, devolve-se a lista de vértices que este ficheiro possui sem se efetuar de novo a sua leitura. Esta otimização é muito útil em cenas que são constituídas por um grande número de modelos idênticos, tais como a do sistema solar.

Embora estas otimizações sejam úteis a nível da obtenção dos dados para o desenho de uma cena, torna-se também necessário que não sejam calculadas sempre que se efetua uma operação de desenho. Para isso, recorreu-se à hierarquia de classes do *C++.* Pode-se ver um desenho de uma cena como uma sequência de passos e instruções em *OpenGL* (rotações, translações, desenho de triângulos, etc.). Ou seja, pode-se ver cada um desses diferentes tipos de instruções como uma operação em *OpenGL.* Uma operação representa então uma classe base genérica e cada tipo de operação em si uma classe derivada. Cada classe derivada é responsável por armazenar os dados necessários para ser executada (tipicamente um conjunto de valores reais, ou uma lista de vértices para o desenho de triângulos). Assim sendo, as classes derivadas apresentam as suas próprias definições de uma método abstrato (com o nome *execute()*) proveniente da classe base *GLOperation*. Com esta hierarquia definida pode-se recorrer ao polimorfismo em *C++* e agrupar cada uma destas operações numa sequência à custa de um vetor *glOperations* do tipo *GLOperation*.* O uso deste vetor torna o cálculo muito eficiente já que este é efetuado apenas uma só vez e a inserção de uma operação no vetor tem um custo constante. Na secção **4** (Ciclo de *Rendering*) ilustraremos o quanto esta abordagem simplifica em muito o redesenho de uma cena.

4. Ciclo de *Rendering*

Através do aproveitamento do polimorfismo em *C++*, o ciclo de *rendering* tornou-se bastante mais simples e eficiente devido ao facto de existir uma classe principal chamada *GLOperation.* Esta classe possui um método abstrato *execute* que é implementado por cada uma das suas subclasses.

Assim, o ciclo de *rendering* apresenta-se apenas como um ciclo *for* que percorre o vetor de operações e invoca o método abstrato *execute* para cada uma delas.

Apresenta-se de seguida, o pseudocódigo da função *drawScene* que é invocada na *renderScene*:

```
Para i = 0 até tamanhoVetor() {
    fazer vetor[i].execute();
}
```

5. Exemplos de execução

5.1. Modelo do Sistema Solar

Na **Figura 1** temos um excerto do código que permite gerar um modelo do Sistema Solar. O código para se gerar cada um dos planetas é praticamente igual, diferenciando na translação, na rotação e na escala. A translação varia porque têm posições diferentes. A rotação, apesar de serem esferas, está a variar para que não fiquem todos com o mesmo sentido. Altera-se o valor da escala porque os planetas têm tamanhos diferentes. O resultado do ficheiro *XML* está demonstrado na **Figura 2**.

```
<group>
    <rotate angle=50 axisY=1/>
    <translate z=-10/>
    <scale x=0.3 y=0.3 z=0.3/>
    <models>
        <model file="demos/sphere.3d" diffR=0.8 diffG=0.4/>
    </models>
</group>
```

Figura 1 – Excerto de código do modelo do Sistema Solar

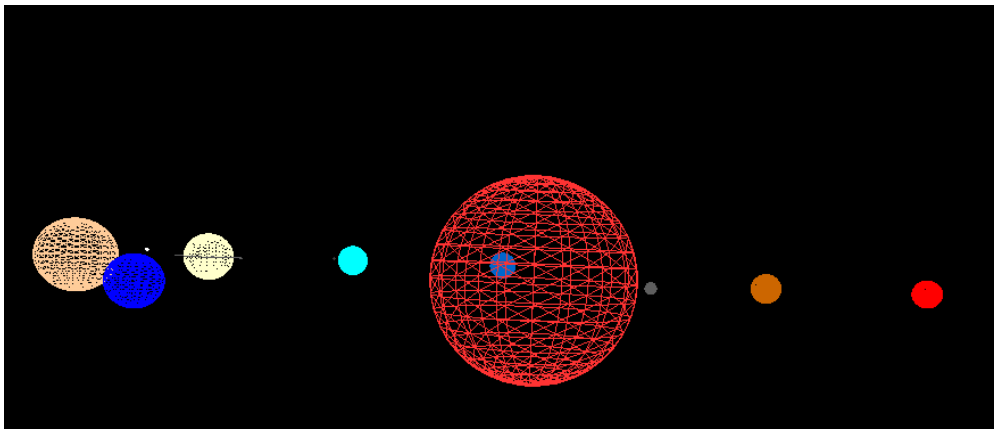

Figura 2 – Modelo do Sistema Solar

5.2. Primitivas com diferentes escalas

A **Figura 3** é o exemplo de um código *XML* que altera a escala de um ficheiro com os pontos de uma primitiva geométrica. Neste caso são as coordenadas dos pontos de uma esfera. Aqui, para o mesmo ficheiro *sphere.3d*, alterámos a escala duas vezes e fizemos uma translação de forma a comparar o resultado desta alteração na escala (**Figura 4**).

```
<scene>
  <group>
    <translate x=3/>
    <scale x=1 y=2 Z=2/>
    <models>
      <model file="demos/sphere.3d" diffr=1/>
    </models>
  </group>
  <group>
    <translate x = -3/>
    <scale X=1 Y=0.5 z=1.5/>
    <models>
      <model file="demos/sphere.3d" diffG=1/>
    </models>
  </group>
</scene>
```

Figura 3 – Exemplo de código que altera a escala de primitivas

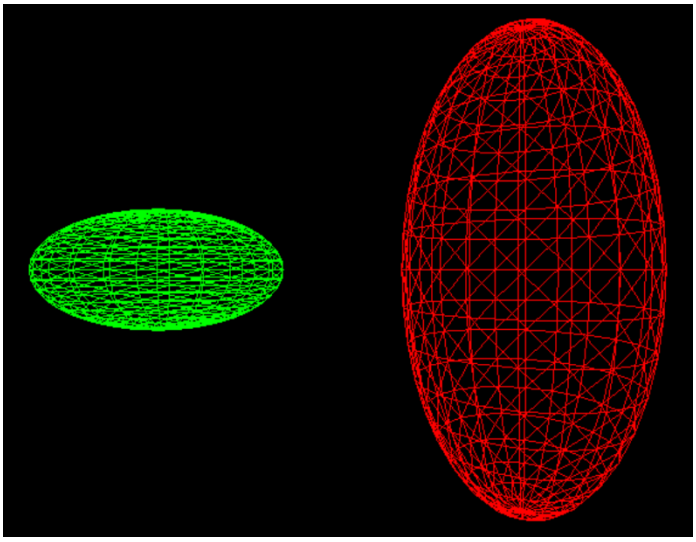

Figura 4 – Primitivas geradas depois de ser alterada a sua escala

5.3. Ficheiros *XML*

Da **Figura 5** à **Figura 8** temos exemplos de código *XML* que o nosso *engine* consegue detetar como errado. Por exemplo, na **Figura 5**, temos mais que uma *tag translate* dentro do mesmo *group*.

```
<scene>
  <group>
    <translate X=5 Y=0 Z=2 />
    <group>
      <translate Y=1 />
      <models>
        <model file="cone.3d" />
      </models>
    </group>
    <translate X=5 Y=0 Z=2 />
    <group>
      <translate X=1 />
      <models>
        <model file="cone.3d" />
      </models>
    </group>
  </group>
</scene>
```

Figura 5 – Exemplo de *group* errado

Na **Figura 6** está declarado mais do que uma *tag models* dentro do mesmo *group*.

```
<scene>
  <group>
    <rotate angle=4 axisx=1/>
    <scale x=3/>
    <models>
      <model file="demos/box.3d"/>
    </models>
  </group>
  <group>
    <translate z=10/>
    <models>
      <model file="demos/sphere.3d"/>
      <model file="demos/plane.3d"/>
    </models>
    <group>
      <rotate angle=40 axisy=1/>
      <models>
        <model file="demos/cone.3d"/>
      </models>
    </group>
    <models>
      <model file="demos/boxdiv.3d"/>
    </models>
  </group>
</scene>
```

Figura 6 – Exemplo de *models* errado

Na **Figura 7** há um *translate* num eixo *w* que não existe. O *engine,* ao deparar-se com erros destes, devolve a mensagem apropriada a informar.

```
<scene>
  <group>
    <translate x=1/>
    <models>
      <model file="demos/box.3d"/>
    </models>
    <group>
      <translate w=4/>
      <models>
        <model file="demos/cone.3d"/>
      </models>
    </group>
  </group>
</scene>
```

Figura 7 – Exemplo de um *translate* errado

Na **Figura 8** acrescentámos ao código *XML* utilizado na **Figura 3** um *model* que não foi gerado – *ring.3d*. O nosso *Engine*, ao chegar a essa linha, apresenta uma mensagem de erro a indicar que não conseguiu ler o ficheiro. Apesar disso, os modelos que efetivamente existem são desenhados (**Figura 9**).

```
<scene>
  <group>
    <translate x=3/>
    <scale x=1 y=2 Z=2/>
    <models>
      <model FILE="demos/sphere.3d" diffr=1/>
      <model file="demos/ring.3d"/>
    </models>
  </group>
  <group>
    <translate x = -3/>
    <scale X=1 Y=0.5 z=1.5/>
    <models>
      <model FilE="demos/sphere.3d" diffG=1/>
    </models>
  </group>
</scene>
```

Figura 8 – Exemplo de um ficheiro *ring.3d* inexistente

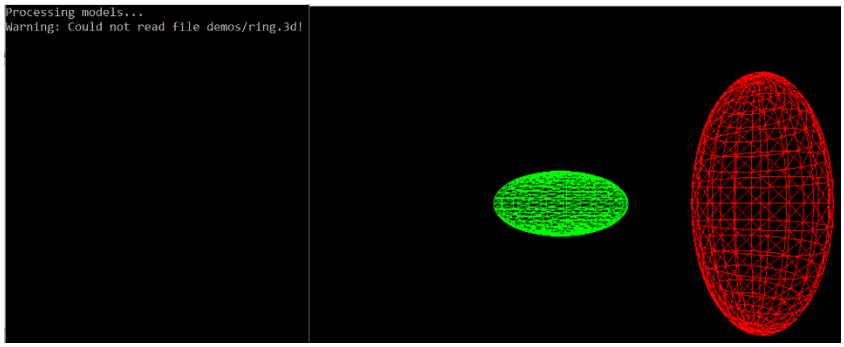

Figura 9 – Resultado da execução do ficheiro *XML* na **Figura 8**

6. Conclusão

Esta segunda fase do trabalho prático foi importante pois pudemos melhorar o trabalho realizado anteriormente com a introdução das transformações geométricas.

Para além disso, nesta fase já foram feitas várias otimizações a nível da leitura dos dados, tarefa na qual o polimorfismo do *C++* se revelou uma ferramenta bastante útil.

Nas fases seguintes, esperamos vir a realizar uma otimização mais aprofundada, nomeadamente a nível do desenho dos modelos com o recurso a *VBO's*, em vez de utilizar o desenho direto a partir do uso de expressão em *OpenGL* "*GL_TRIANGLES*" (como acontece atualmente no módulo *trianglesDrawing*).