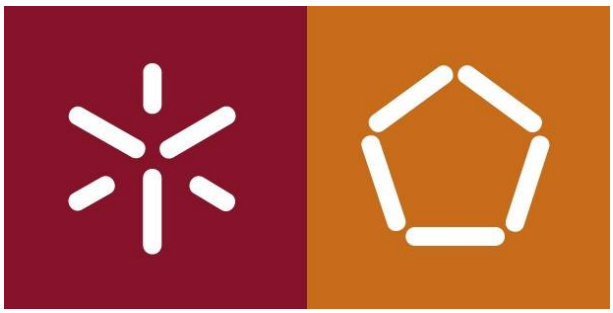

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Sistema Solar

Joan Rodriguez (a89980)
Mário Santos (a70697)
Pedro Costa (a85700)
Rui Azevedo (a80789)
29 de Março de 2020

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório diz respeito à segunda fase do trabalho prática da cadeira Computação Gráfica. Esta fase do trabalho tem como objectivo criar cenas hierárquicas usando transformações geométricas e, opera essencialmente sobre a aplicação *Engine* do sistema.

A aplicação *Engine* tem duas fases cruciais: a fase de leitura e armazenamento dos dados contidos nos ficheiros *XML* e a fase de desenho das cenas usando as primitivas gráficas, definidas na primeira fase do trabalho, e as respectivas transformações geométricas contidas nos ficheiros.

É pretendido, no fim desta fase do trabalho, criar um ficheiro *XML* correspondente a um sistema solar estático e simplista.

2 Arquitectura do *Engine*

A aplicação *Engine* está dividia em duas partes: a parte de leitura e armazenamento dos dados contidos no ficheiro *XML* e a parte de desenho das cenas com a informação extraída dos ficheiros.

2.1 Formato do ficheiro *XML*

O ficheiro *XML* é composto essencialmente por três elementos: *scene*, *model* e *group*. A *scene* corresponde a uma colecção de nodos que correspondem à representação espacial do cena gráfica, o *model* representa uma primitiva gráfica e o *group* faz com que um conjunto de objectos sejam tratados como um só onde no seu interior podem estar definidas transformações geométricas, outros *groups* aninhados bem como *model's*, maioritariamente como a folha desta árvore.

A baixo, pode-se ver uma estrutura possível de um ficheiro *XML* deste tipo.

```
<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>
```

2.2 Arquitectura de classes

As transformações geométricas necessárias para o desenvolvimento deste trabalho são: translações, rotações e escalas. Para isto, foram definidas as classes correspondentes para uma melhor representação da estrutura do sistema.

Foi criada uma classe abstracta *Transformation* necessária para se poder armazenar diferentes tipos de transformações geométricas apenas numa estrutura de dados. Todas as outras classes, *Rotation*, *Scale* e *Translation* herdam da classe abstracta, tirando assim partido da programação orientada aos objectos para um desenvolvimento mais modular do sistema. A classe abstracta *Transformation* contém um método virtual *execute* que faz com que as suas sub-classes sejam obrigadas a implementar. Este método, nas sub-classes, invoca a função do *OpenGl* correspondente à transformação respectiva.

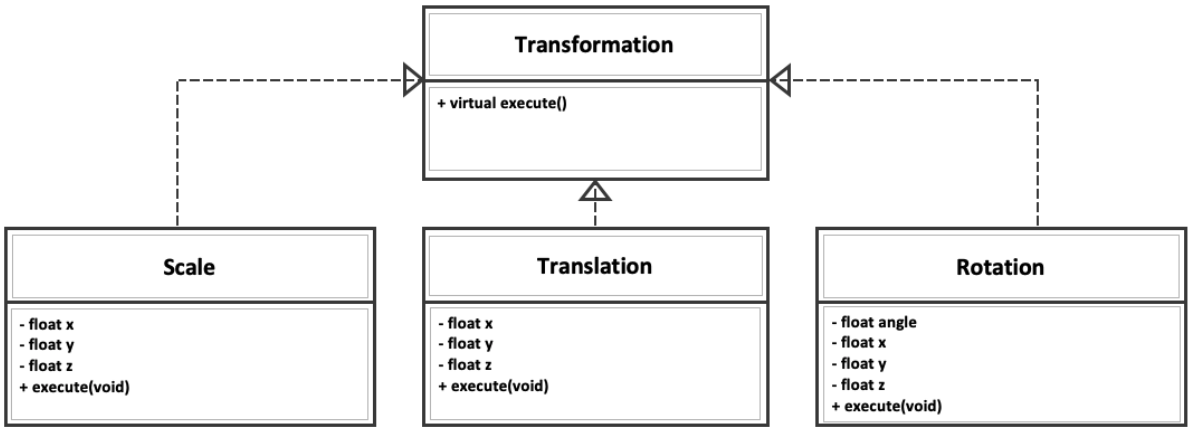

Figura 1: Diagrama de classes para as transformações geométricas

2.3 Funcionalidades

O *Engine* desenvolvido apresenta um conjunto de interações que o utilizador pode ter com o sistema, a enumerar:

- deslocação do modelo usando as teclas *a, d, w, s*
- menu com as seguintes funcionalidades:
  - Diferentes modos de desenho: desenho com linhas, pontos e com preenchimento usando, respectivamente, as macros do *OpenGl GL_LINE, GL_POINT* e *GL_FILL*
  - Escolha de cores para as primitivas desenhadas

3 *Parsing* do documento *XML*

Tal como aconteceu na primeira fase do trabalho, para um *parsing* eficiente do ficheiro *XML* foi usada a biblioteca *open source tinyxml2*. Como já foi referido anteriormente, o objectivo deste *parsing* é eencontrar *tags* correspondentes aos diferentes elementos que se espera encontrar no ficheiro *XML*, *i.e*, *scene*, *group* e *model*.

De seguida, é apresentado, em pseudocódigo, os algoritmos usados para o *parsing* do ficheiro *XML*.

Algorithm 1 Algoritmos de *parsing*

$vector < Engine\_Primitive > allPrimitiveLocal$, **Global**

```
1: function readXML
2:     root ← firstChildElem(scene)
3:     vector < Transformation > transformations
4:     if root then
5:         parseGroups(root, transformations)
6:
1: function parseGroups(root, transformations)
2:     group ← firstChildElem(group)
3:     vector < Transformation > localTransformations = transformations
4:     while group do
5:         parseGeometricTransformations(group, localTransformations)
6:         parseGroupElements(group, localTransformations)
7:         parseGroups(group, localTransformations)
8:         group ← nextSiblingElement()
9:         localTransformations.clear()
10:
1: function parseGeometricTransformations(root, transformations)
2:     elem ← group.firstChildElem()
3:     while elem do
4:         if elem = translate then
5:             transformations.add(parseTranslation(elem.firstAttribute))
6:         if elem = rotate then
7:             transformations.add(parseRotation(elem.firstAttribute))
8:         if elem = scale then
9:             transformations.add(parseScale(elem.firstAttribute))
10:        group ← nextSiblingElement()
11:
1: function parseGroupElems(group, transformations)
2:     Engine_Primitive p
3:     elems ← group.firstChildElem(models)
4:     elem ← elems.firstChildElem(model)
5:     while elem do
6:         s ← elem.attribute(file)
7:         p = readFile(s)
8:         p.setTransformations(transformations)
9:         allPrimitivesLocal.push_back(p)
10:        elem ← elem.nextSiblingElement()
11:
```

Algorithm 2 Algoritmos de *parsing*

```
1: function parse(Translation|Scale)(attribute) :: Transformation
2:     while attribute do
3:         if attribute.name() =′ X′ then
4:             x ← attribute.value()
5:         if attribute.name() =′ Y′ then
6:             y ← attribute.value()
7:         if attribute.name() =′ Z′ then
8:             z ← attribute.value()
9:         return (Translation(x, y, z)|Scale(x, y, z))
10:
1: function parseRotation(attribute) :: Transformation
2:     while attribute do
3:         if attribute.name() =′ angle′ then
4:             angle ← attribute.value()
5:             if attribute.name() =′ axisX′ then
6:                 x ← attribute.value()
7:             if attribute.name() =′ axisY′ then
8:                 y ← attribute.value()
9:             if attribute.name() =′ axisZ′ then
10:                z ← attribute.value()
11:            return Rotation(angle, axisX, axisY, axisZ)
12:
13:
```

Os algoritmos representados na página anterior, dizem respeito à procura das diferentes *tags*. De maneira a guardar toda a informação num sítio só, foi criada uma classe *Engine_Primitive* composta por um vector de vértices e por outro vector de transformações que irá, posteriormente, ser usado no *engine.cpp*. A função que dá início ao processo de *parsing* é a função *readXML*, estando responsável por identificar a *tag scene*. Uma vez encontrada a *tag*, é chamada a função *parseGroups* que procura, recursivamente, *tags group*.A função *parseGeometricTransformation* é responsável por encontrar as transformações geométricas e invocar a respectiva função de *parsing* para devolver uma transformação. As funções *parseRotation*, *parseTranslation* e *parseScale* são responsáveis por criar um objecto do tipo *Transformation* para posteriormente ser adicionado ao conjunto de transformações. Por fim, a função *parseGroupElems* é responsável por ler o modelo contido no ficheiro *XML* e, após esta leitura, adicionar tanto o modelo como o conjunto de transformações à variável global *allPrimitiveLocal*.

4 Desenho do modelo

A função responsável por desenhar as primitivas bem como as transformações geométricas é a função *drawVerticesOp* que posteriormente irá ser usada na *renderScene*. A função já aplica o conceito *Vertex Buffer Objects VBO's*, para que o processo de desenho seja mais eficiente.

De seguida, é apresentado, de uma forma genérica, a função referida.

Algorithm 3 Algoritmo de Desenho

```
1: vector<Engine_Primitive> allPrimitives, Global
2: function drawVerticesOp
3:     for i ← 0 to allPrimitives.length() do
4:         vertices ← allPrimitives.at(i).getVertices()
5:         for j ← 0 to vertices.length() do
6:             vertices.at(j).drawVertices()
7:         transformations ← allPrimitives.getTransformations()
8:         for k ← 0 to transformations.length() do
9:             transformations.at(k).execute()
   =0
```

5 Output

Após o desenvolvimento do sistema, pode-se ver o resultado final que representa o sistema solar. O sistema tem a representação dos planetas, das respectivas luas e o sol no centro do sistema.

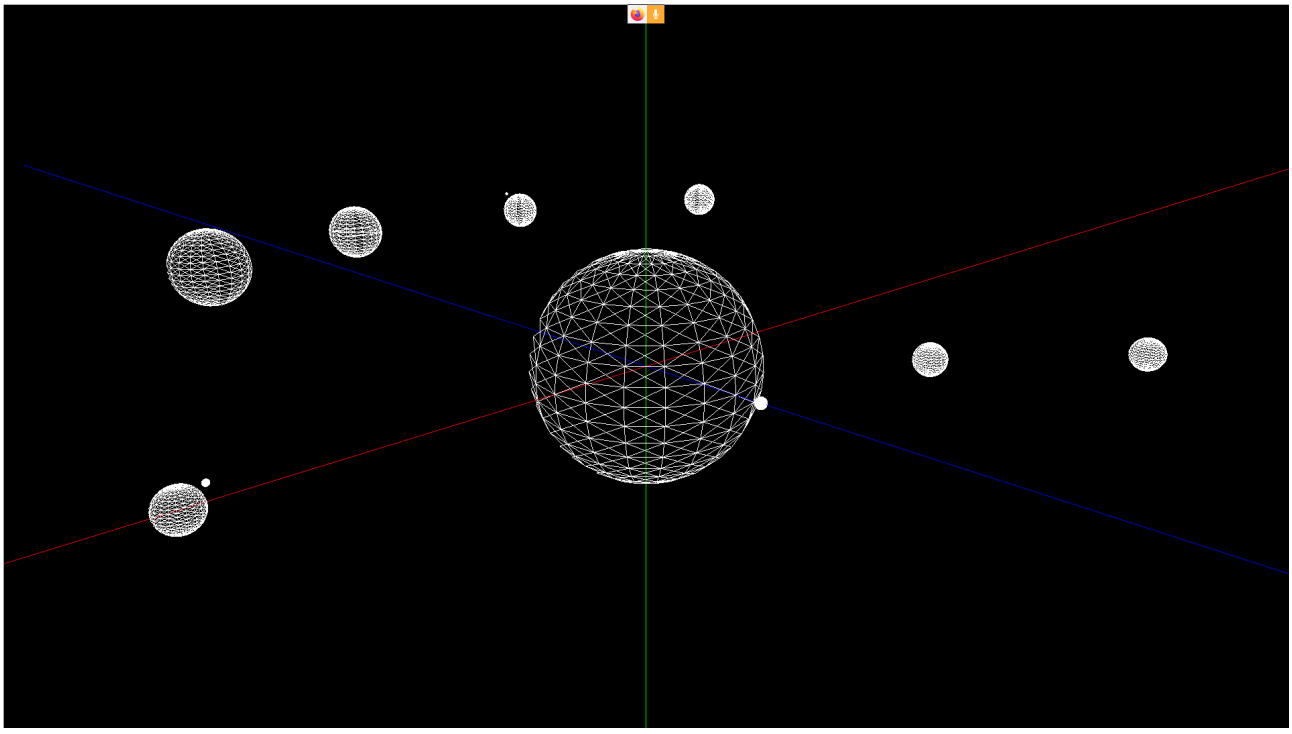

Figura 2: Sistema solar

6 Conclusão

Em comparação com a primeira fase do trabalho, é de notar que esta foi menos complexa e exigente, uma vez que que tirou-se partido das primitivas criadas anteriormente. Esta fase tornou o sistema mais interessante pois agora é capaz de ler um ficheiro que contém um conjunto de primitivas e transformações geométricas associadas e desenhar o cenário correspondente.