Universidade do Minho

Mestrado Integrado em Engenharia Informática

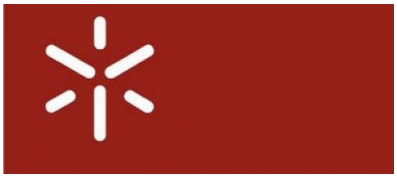
Computação Gráfica

Transformações Geométricas

Grupo:
Etienne Costa A76089
Joana Cruz A76270
Rafael Alves A72629
Maurício Salgado A71407

Docente:
António Ramires

25 de Março de 2019



Conteúdo

1 Introdução

O relatório apresentado diz respeito à segunda fase do projeto proposto no âmbito da unidade curricular de Computação Gráfica. O trabalho consiste na criação de um cenário através do processamento de ficheiros XML e aplicação de várias transformações geométricas em OpenGL tal como translações, rotações e escalas. Numa primeira fase é explicado o modo como desenvolvemos as estruturas de dados que suportam o programa, posteriormente será descrito o processo de leitura e tratamento dos dados contidos no ficheiro XML e e por último a utilização de VBO's e o modo como cada elemento é renderizado na função renderscene. No final do relatório encontram-se alguns exemplos de execução.

2 Engine

O programa Engine está dividido em duas fases: leitura, extração de dados de ficheiros XML para armazenamento em estruturas apropriadas 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 - os modelos que contém e as outras operações em OpenGL a efetuar (translações, rotações e escalas), assim como a sua cor. Na segunda fase, o programa encarrega-se de desenhar a cena descrita no ficheiro XML através de um ciclo de rendering. Com os novos ficheiros de configuração XML de entrada pedidos nesta fase, foi necessário reestruturar o engine e o parser, assim como desenvolver classes para armazenamento da informação pretendida. Como já foi referido no relatório anterior, o programa possui ainda funcionalidades relativas ao movimento da cena:

- deslocação de um modelo com as teclas - *a, d* para translações no eixo dos xx; *w, s* para translações no eixo dos yy; *4, 5* para translações no eixo dos zz
- rotação de um modelo com as setas do teclado através da camera
- rotação no eixo vertical de um modelo com teclas *q, e* com recurso à função glRotatef
- definição de opções Fill, Line ou Point em OpenGL relativas ao desenho dos modelos com as teclas *f,l,p*
- zoom in e zoom out com teclas *x, z*

3 Estruturas de dados

Devido à necessidade de guardar diferentes tipos de informações, lidas a partir do ficheiro XML e posteriormente usadas para desenhar o cenário sistema solar, definimos as seguintes estruturas de dados.

3.1 Vertex

Esta classe representa um ponto num referencial a três dimensões, com as coordenadas x, y e z, o que se torna bastante útil para a representação dos vértices utilizados posteriormente para o desenho dos triângulos que elaboram as figuras primitivas.

```
class Vertex{
public:
        float x;
        float y;
        float z;
        Vertex();
        Vertex(float xx, float yy, float zz);
        ~Vertex();
};
```

3.2 Model

Esta classe armazena todos os pontos para a criação das figuras primitivas (plane, box, cone, sphere) que estão presentes e descritos no ficheiro.

```
class Model{
public:
        string fileName;
        vector<Vertex> vertexes;
        Model();
        Model(string path);
        ~Model();
};
```

3.3 Group

Esta é a classe que guarda os dados retirados de um grupo do ficheiro XML. Nesta estrutura é possível armazenar todas as informações que estão associadas a um determinado grupo, as respetivas transformações geométricas se existirem, os modelos associados a esse grupo, assim como os grupos filhos contidos. Decidimos

também guardar a cor no grupo. Como as seguintes transformações geométricas - translações e escalas - acabam por ser um ponto com três coordenadas do tipo float, decidimos utilizar a classe acima definida Vertex. O mesmo se passa com as rotações, apenas com a particularidade que também existe um ângulo de rotação, daí esse ser armazenado também.

```
class Group{
public:
        Vertex rotation;
        float rotationAngle;
        Vertex translation;
        Vertex scale;
        Vertex color;
        vector<Model> models;
        vector<Group> subGroups;
        Group(void);
        Group(Vertex rotation, float rotAngle,
        Vertex translation, Vertex scale, Vertex color,
        vector<Model> models, vector<Group> subGroups);
        ~Group();
};
```

4 Processamento de um ficheiro XML

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

- Leitura e Processamento do cenário – Consiste na abertura do ficheiro que contém o cenário que 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), a cor do grupo ou desenho de uma primitiva.

- Armazenamento nas estruturas de dados – De modo a se conseguir redesenhar um modelo quantas vezes for necessário, em estruturas acima referidas. Na nossa função main do programa apenas lemos o ficheiro uma vez e armazenamos nas estruturas apropriadas.

4.1 Parser

Esta é a classe que efetua o processo acima descrito. Agora fazemos uma pequena descrição sobre os métodos utilizados.

- ParseXMLFile - sendo a cena sempre um conjunto de grupos, esta função verifica se o documento XML apresenta um formato correto e após isso vai começar a efetuar o parsing de cada grupo

- ParseGroup - percorre um nodo do tipo grupo do XML e extrai a informação correspondente. Se o grupo em questão contiver outros grupos dentro de si, essa informação também será processada. Esta função faz o controlo de possíveis ficheiros de configuração XML errados, caso exista mais que uma transformação do mesmo tipo num grupo, apenas processa a primeira. Caso a informação a ler seja uma transformação geométrica a função parseAttributes é invocada

- ParseModel - processa a informação correspondente ao ficheiro da figura primitiva

- ParseAttributes - processa a informação sempre que é encontrada uma transformação (traslate, rotate ou scale) criando um vértice e adicionando-o ao grupo que está a ser gerado pelo parseGroup.

5 Utilização de VBO's

Vertex Buffer Objects podem ser vistos como uma estrutura abstrata de dados que permite melhorar consideravelmente a performance geral do OpenGl. A vantagem de usar um VBO é a seguinte:

- Sem a utilização de VBOs, utilizando o modo de desenho imediato do OpenGL, os vértices são definidos em memória RAM e copiados um a um, para a placa gráfica, conforme a ordem com que devem ser processados. Este processo é repetido sempre que os queremos desenhar.

- Com a utilização VBOs, todos os vértices são definidos e copiados para um buffer, que é passado numa fase inicial, e uma única vez, para a memória da placa gráfica, antes de qualquer pedido para os processar e desenhar. Deste modo, o processamento dos triângulos a desenhar torna-se muito mais eficiente.

Apesar de apenas ser pedido na próxima fase, decidimos já implementar o desenho das primitivas recorrendo ao uso de VBO's, dado a estas questões de performance. Para isso no nosso motor, após a leitura e processamento do ficheiro, começamos por gerar quantos buffers quanto o número de grupos que temos com recurso à função **glGenBuffers**. De seguida, para preencher cada um dos VBOs com as coordenadas dos vértices, relativos a um determinado grupo criámos um método **fillBuffers**. Neste método, é feito um ciclo para iterar cada um dos grupos, e preencher cada um dos índices do array de VBOs com as coordenadas dos vértices dos respetivos modelos do grupo. Tendo preenchido o array com as coordenadas, é ativada a posição do array de VBOs sobre o qual se vão inserir os dados, com recurso à função **glBindBuffer**. O referido índice é preenchido com os elementos do array de coordenadas determinado, usando para isso a função **glbufferData**. Como no preenchimento dos VBOs deve também ser tida em consideração a ordem pela qual os grupos são processados, caso o nó recebido como argumento tenha filhos, a função é chamada recursivamente para o respetivo Grupo filho. Só depois desta verificação, é que caso existam irmãos no nodo, a função é iterada recursivamente para o Grupo irmão.

6 Ciclo de rendering

Primeiramente decidimos explicar os conceitos das funções **glPushMatrix()** e **glPopMatrix()**, que se tornaram bastante importantes para o correto desenho do cenário. Para conseguir preservar um estado da matriz de visualização de um determinado modelo, é utilizada a função **glPushMatrix()** da biblioteca glut, que irá guardar a matriz atual numa stack. Quando for pretendido recuperar a última matriz guardada na referida stack, basta para isso utilizar a a função **glPopMatrix()**.
Tendo em conta os conceitos mencionados será descrito o processo utilizado para a execução do ciclo de rendering. Decidimos implementar uma função drawScene que será invocada no renderScene e irá percorrer a estrutura de dados final - uma lista de objectos do tipo Group.

1. Antes de realizar qualquer alteração à matriz de visualização do modelo é efetuado um **glPushMatrix()** de modo a guardar o estado da matriz atual

2. Após isso, são realizadas as transformações correspondentes, por esta ordem uma rotação, uma translação e uma escala que vão ser aplicadas à matriz atual, utilizando as funções do glut **glRotatef**, **glTranslatef** e **glScalef** respetivamente. Define-se também a cor a aplicar aquele grupo

3. Realizadas as transformações, ativa-se o buffer correspondente e vão ser desenhados todos os vértices dos modelos do grupo, com recurso às funções **glVertexPointer** e **glDrawArrays**

4. Caso o nodo atual tenha descendência, a função drawScene é chamada recursivamente para o filho, sem que o **glPopMatriz()** seja efetuado, para que não se percam as transformações já realizadas

5. Caso o grupo atual não tenha nenhum grupo filho, é finalmente efetuado o **glPopMatrix()**, uma vez que vamos querer voltar ao estado da matriz de visualização do modelo anterior, sem as alterações que foram efetuadas pela atual iteração da função drawScene. Este processo é efetuado porque as transformações dos grupos seguintes já não serão sobrepostas às transformações atuais

7 Exemplos de Execução

7.1 Figuras primitivas

Exemplo de um cenário básico contendo algumas transformações geométricas e diferentes primitivas gráficas.

```
<scene>
    <group>
        <color G=0.4 B=0.8/>
        <translate X=-10/>
        <scale X=1 Y=1 Z=1/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
    <group>
        <color R=0.37 G=0.37 B=0.37/>
        <rotate angle=-90 axisZ=1/>
        <translate Y=-2/>
        <scale X=2 Y=2 Z=2/>
        <models>
            <model file="cone.3d"/>
        </models>
    </group>
    <group>
        <color R=0.8 G=0.4/>
        <translate X=10/>
        <scale X=0.9 Y=0.9 Z=0.9/>
        <models>
            <model file="box.3d"/>
        </models>
    </group>
</scene>
```

Figura 1: Ficheiro XML para um exemplo com algumas primitivas e transformações

Figura 2: Cenário com as primitivas gráficas

7.2 Figuras primitivas com diferentes escalas

```
<scene>
    <group>
        <color R=0.37 G=0.37 B=0.37/>
        <scale X=2 Y=2 Z=2/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
    <group>
        <color G=0.4 B=0.8/>
        <translate X=10/>
        <scale X=2 Y=5 Z=2/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
</scene>
```

Figura 3: Ficheiro XML para demonstrar a mesma primitiva com diferentes escalas

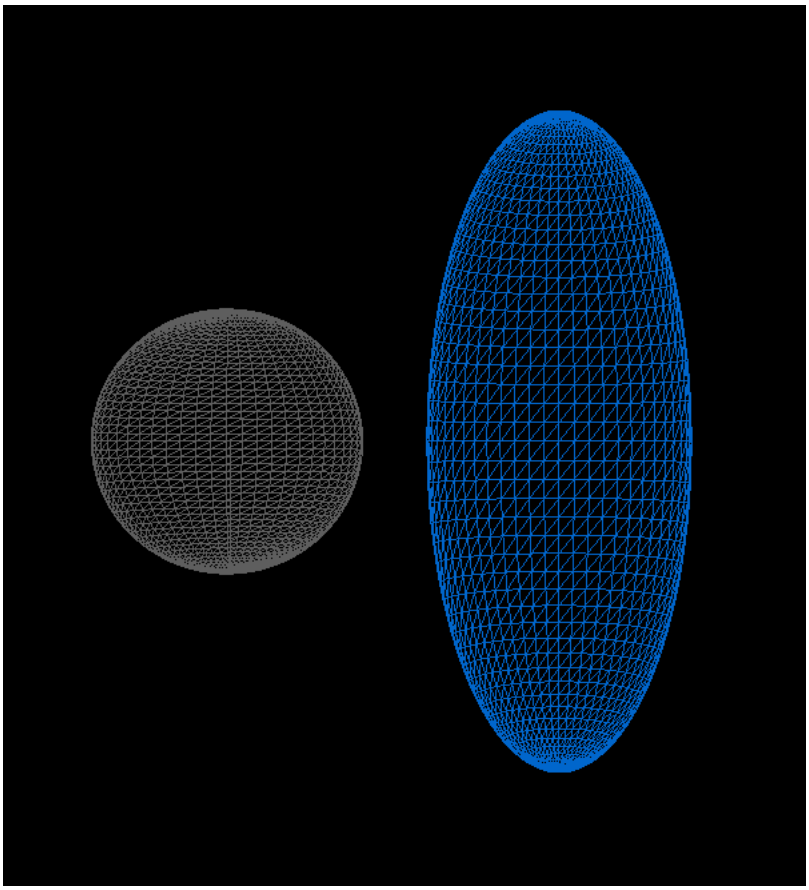


Figura 4: Cenário com as primitivas gráficas com diferentes escalas

7.3 Sistema solar estático

Para o nosso cenário do sistema solar estático desenvolvemos dois ficheiros de configuração XML. Em ambos os ficheiros, para cada planeta do Sistema Solar criamos um grupo irmão, mas para as luas de um planeta usamos como base as tranformações já efetuadas no planeta e aplicamos as novas transformações necessárias. A primitiva gráfica usada para o desenho do Sol, de cada planeta, das luas e do anel de Saturno é sempre a esfera, com diferentes escalas aplicadas. Neste primeiro cenário, são apenas aplicadas translações e escalas, e apenas uma rotação para o anel de Saturno. Neste os planetas ficam todos alinhados no eixo dos xx com excepção das luas.

```xml
<scene>
    <group>
        <!-- SOL -->
        <color R=1 G=0.2 B=0.2/>
        <scale X=2 Y=2 Z=2/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
    <group>
        <!-- MERCÚRIO -->
        <color R=0.37 G=0.37 B=0.37/>
        <translate X=5/>
        <scale X=0.1 Y=0.1 Z=0.1/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
    <group>
        <!-- VENÚS -->
        <color R=0.8 G=0.4/>
        <translate X=7/>
        <scale X=0.35 Y=0.35 Z=0.35/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
    <group>
        <!-- TERRA -->
        <color B=1/>
        <translate X=9/>
        <scale X=0.3 Y=0.3 Z=0.3/>
        <models>
            <model file="sphere.3d"/>
        </models>
        <group>
        <!-- LUA -->
        <color R=0.37 G=0.37 B=0.37/>
        <translate X=2.7/>
        <scale X=0.15 Y=0.15 Z=0.15/>
        <models>
            <model file="sphere.3d"/>
        </models>
        </group>
    </group>
```


Figura 5: Excerto do exemplo de um ficheiro XML de um Sistema Solar estático

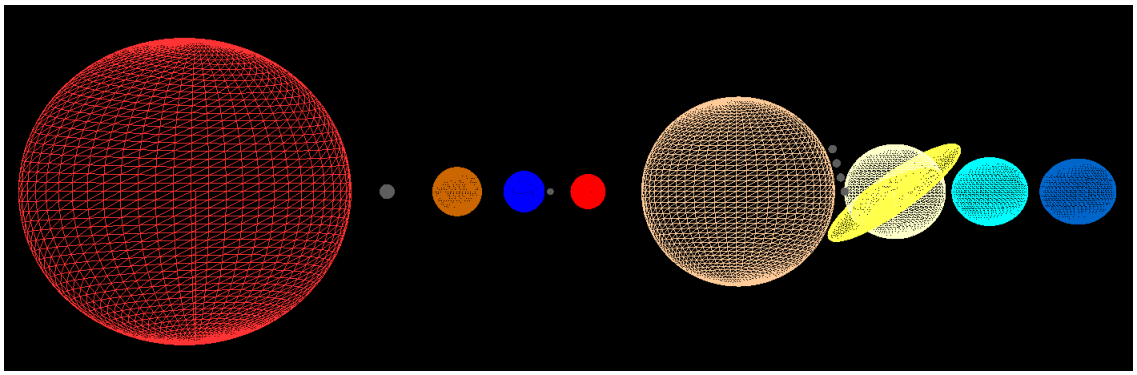


Figura 6: Cenário de um Sistema Solar estático

Já neste ficheiro decidimos aplicar rotações no eixo dos yy, para conseguirmos modificar a direção dos eixos dos xx e zz, aplicando as devidas translações de seguida.

```
<group>
    <!-- Vénus -->
    <color R=0.8 G=0.4/>
    <rotate angle=45 axisY=1/>
    <translate Z=10/>
    <scale X=0.3 Y=0.3 Z=0.3/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
<group>
    <!-- Terra -->
    <color B=1/>
    <rotate angle=90 axisY=1/>
    <translate Z=15/>
    <scale X=0.3 Y=0.3 Z=0.3/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
<group>
    <!-- Marte -->
    <color R=1/>
    <rotate angle=135 axisY=1/>
    <translate Z=20/>
    <scale X=0.3 Y=0.3 Z=0.3/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
<group>
    <!-- Júpiter -->
    <color R=1 G=0.8 B=0.6/>
    <rotate angle=180 axisY=1/>
    <translate Z=25/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
```

Figura 7: Excerto do exemplo de um ficheiro XML de um Sistema Solar estático

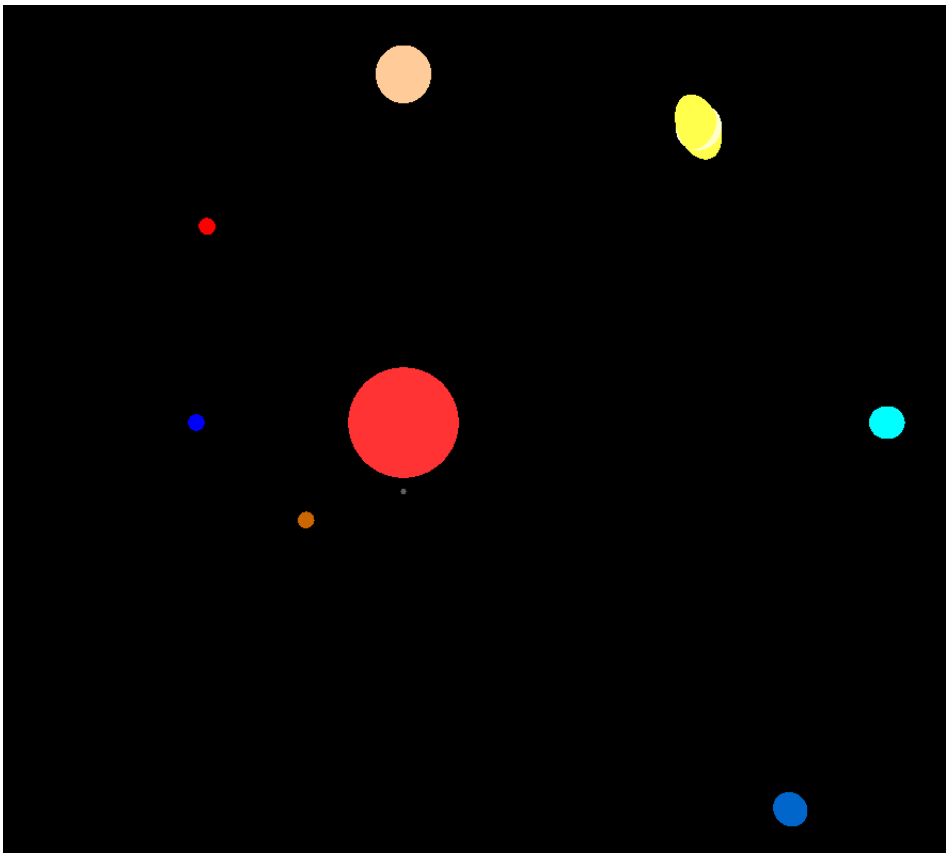


Figura 8: Cenário de um Sistema Solar estático

8 Conclusão

Nesta segunda fase inicialmente encontramos algumas dificuldades na interpretação do ficheiro XML, assim como a melhor abordagem para a sequência de transformações - se primeiro aplicar uma rotação e só após uma translação ou o contrário - que após uma análise detalhada foram ultrapassadas. Desenvolvemos o nosso conhecimento de OpenGL, pois com a introdução das transformações, evoluímos bastante sobre como manipular as matrizes de forma a que o desenho das primitivas gráficas fosse nas posições pretendidas. Quanto à aplicação Engine, várias alterações foram feitas, devido ao aumento da dimensão do projeto, decidimos estruturar em classes e segmentar o código. Cumprimos todos os requisitos propostos, apenas ficou por otimizar o nosso processamento de leitura de ficheiros de configuração XML caso encontre um ficheiro XML inválido com múltiplas transformações do mesmo tipo. Além do trabalho futuro já proposto na próxima fase, iremos desenvolver uma figura primitiva Torus para o desenho do anel de Saturno.