Computação Gráfica

Mestrado Integrado em Engenharia Informática
3ºano-2ºsemestre

José Pinto A81317 Luís Correia A81141
Pedro Barbosa A82068

Trabalho Prático - Parte 2

Resumo

Foi-nos proposto na UC de Computação Gráfica o desenvolvimento de um mini mecanismo 3D baseado num cenário gráfico utilizando as ferramentas utilizadas na cadeira, *C++* e *OpenGL*.

Este relatório é referente à segunda fase onde nos foi proposta a implementação de transformações geométricas.

Março 2019

Conteúdo

1 Introdução **1**

2 Engine **1**
2.1 Classes 1
2.1.1 GeometricTransforms 1
2.1.2 Group 2
2.1.3 Model 2
2.1.4 Models 3
2.2 Processamento da configuração 3
2.3 Renderização 4

3 Generator **5**

4 Sistema Solar **6**

5 Conclusão **7**

Lista de Figuras

1 Classe *GeometricTransforms* 1
2 Classe *Group* 2
3 Classe *Model* 2
4 Classe *Models* 3
5 Segmento de código da função *parserXML*. 4
6 Desenha subgrupos 5
7 Cálculo dos vértices do anel 6
8 Modelo estático do Sistema Solar 7

1 Introdução

Nesta segunda fase do trabalho prático foi-nos proposta a introdução de transformações geométricas. Para isso, nos ficheiros XML de configuração é possível conter transformações geométricas (*translate*, *rotate* e *scale*). Cada conjunto de transformações geométricas diz respeito a um ou mais modelos 3D, estando isto tudo englobado pela *tag* **<group>**. É também possível ter um *group* dentro de um *group*.

Por último, foi-nos pedido uma *demo scene* com a configuração relativa ao sistema solar.

2 Engine

Nesta segunda fase do trabalho prático foram feitas alterações e adições em relação ao que foi feito na primeira fase. Os subcapítulos seguintes vão-se focar nessas modificações realizadas sobre o *engine*.

2.1 Classes

Foi necessária a criação de classes que pudessem auxiliar e guardar a informação extraída dos XML através do *parser*. Para isso, criaram-se as seguintes classes.

2.1.1 GeometricTransforms

A classe *GeometricTransforms* (figura 1) foi criada para poder guardar informação sobre as transformação geométricas.

Esta classe possui um inteiro denominado *type* que indica o tipo de transformação. Se este for 0 é um **translate**, 1 é um **rotate** e se for 2 significa que é um **scale**. A classe possui também 3 *floats* (x, y, z) relativos aos parâmetros necessários para cada transformação e também um *float angle* que é apenas usado no caso da transformação geométrica ser um **rotate**.

```
class GeometricTransforms {
public:
    int type;
    float x;
    float y;
    float z;
    float angle;
};
```

Figura 1: Classe *GeometricTransforms*

2.1.2 Group

A classe *Group*(figura 2) possui informação sobre um grupo. Esta vai armazenar a informação retirada do ficheiro XML que está contida entre a tag **<group>** e **</group>**.

Esta contém um *vector* com todas as transformações geométricas, outro com os nomes dos ficheiros onde se encontram os modelos e eventualmente um último *vector* que possui os grupos aninhados caso existam.

```
class Group{
public:
    vector<GeometricTransforms> transforms;
    vector<string> models;
    vector<Group> subGroups;
};
```

Figura 2: Classe *Group*

2.1.3 Model

A classe *Model* é utilizada para armazenar a informação de um modelo. Ou seja, o nome do ficheiro e o conjunto de vértices lá presentes.

```
class Model{
    public:
        //nome do ficheiro a partir do qual o modelo foi carregado
        string name;
        vector<Vertice> vertices;
};
```

Figura 3: Classe *Model*

2.1.4 Models

```
class Models{
    public:
        vector<Model> vec;

        bool contains(string name);

        void addModel(string name);

        //preenche o vetor vertices de cada modelo com os dados presentes nos respetivos ficheiros
        void loadModels();


        vector<Vertice>& getModel(string name);
};
```

Figura 4: Classe *Models*

A classe *Models* é simplesmente uma agregação de todos os objectos da classe *Model*. Uma alternativa ao uso desta classe seria guardar directamente em cada grupo os modelos respectivos. No entanto, devido ao número reduzido de primitivas gráficas, é provável que o mesmo modelo esteja presente várias vezes na mesma configuração. Ao colocar a informação num único sítio e ao identificá-la unicamente através do nome do ficheiro, é possível que no resto do programa apenas seja preciso guardar o nome do ficheiro e, quando necessário, pedir a um objecto da classe *Models* uma referência para a localização dos vértices. Desta forma evita-se desperdício de memória e de tempo de execução.

As funções desta classe são funções típicas de uma lista, excepto a *loadModels*, cuja funcionalidade será explicada na secção seguinte.

2.2 Processamento da configuração

Por questões de organização, o código relativo a esta secção encontra-se nos ficheiros *loadConfig.h* e *loadConfig.cpp*.

Para esta fase, a alteração do *parser* era vital. Na fase anterior apenas existia um *parser* que ia buscar o nome dos ficheiros 3d para mais tarde extrair a informação nestes contida sobre os vértices.

Foi preciso então modificar o *parser* para que este pudesse também processar as transformações geométricas e os modelos contidos dentro de grupos. Também foi tido em conta que podiam existir grupos dentro de grupos.

Para a leitura do XML recorreu-se ao *parser tinyXML2*.

O processo é dividido em duas etapas. O processamento da configuração no ficheiro XML e o processamento dos modelos nos ficheiros 3d. Na primeira etapa a informação da configuração é colocada nas classes apresentadas anteriormente. Cada grupo não aninhado é colocado num vector de objectos da

classe *Group* (a informação dos grupos aninhados está contida nas instâncias dos outros grupos). É importante mencionar as transformações geométricas de cada grupo são guardadas pela ordem que se encontradas. Cada novo nome de ficheiro modelo encontrado é adicionado a um objecto *allModels* da classe *Modelos*.

Após concluído o processamento da configuração, todos os nomes dos ficheiros modelo já estão presentes no objecto *allModels*. Assim sendo, basta invocar a função *loadModels*. Para cada nome de ficheiro presente no objecto, a função processa o ficheiro correspondente de forma idêntica à da fase anterior.

Todo este processo é encapsulado através da função *loadConfig*. Basta chamar uma vez esta função com os parâmetros adequados (nome do ficheiro configuração, referências para um objecto *Groups* e outro objecto *Models*) antes de iniciar a fase de desenho.

Na figura 5 temos um segmento de código da função **parserXML** que ilustra como é feita a extração do ficheiro XML. O *parser* começa o processamento através da raiz e sempre que encontra a *tag* **group** invoca uma função auxiliar que trata desse *group*(**parseGroup**), colocando posteriormente esse grupo processado num *vector* global que possui todos estes.

```
if( string(child->Name()).compare("group") == 0) {
    Group g = parseGroup(child);
    res.push_back(g);
}
```

Figura 5: Segmento de código da função *parserXML*.

A função **parseGroup** está encarregue de extrair a informação contida dentro das *tag group*, devolvendo um objecto **Group**. Esta percorre toda a informação contida na *tag* **<group>** extraindo toda a informação dentro contida.

2.3 Renderização

O código desta secção encontra-se no ficheiro *main.cpp*.

Logo do início do programa é necessário invocar a função *loadConfig*. São passadas a esta função as referências de duas variáveis globais (*groups* e *allModels*) de forma a que o resultado da função possa ser utilizado dentro do ciclo principal do *Glut*.

Uma vez feitas todas estas alterações, um passo importante é modificar a função *renderScene*.

A função agora vai percorrer o *vector* global que possui todos os grupos. Para cada grupo é invocada a função **drawGroup** que tem como objectivo desenhar um **group**

A *drawGroup* começa por fazer **push** à matriz(*glPushMatrix()*). De seguida são realizadas todas as transformações geométricas relativas ao grupo pela ordem que se encontravam na configuração e depois são desenhados os modelos.

Para desenhar cada modelo basta utilizar o nomes guardados no grupos para obter os vectores de vértices necessários. Caso o grupo tenha outros grupos aninhados, é chamada recursivamente a função *drawGroup* para cada grupo (figura 6).Cada grupo é terminado com um **pop**(*glPopMatrix()*) para que as transformações geométricas afectem apenas esse grupo e os seus subgrupos.

```
for(int i = 0; i < group.subGroups.size(); i++ ){
    drawGroup(group.subGroups[i]);
}
```

Figura 6: Desenha subgrupos

3 Generator

Como para esta fase foi-nos pedido uma *demo scene* do sistema solar, foi necessária a adição de uma função no *generator* que conseguisse gerar os vértices para um *anel*, necessário ao planeta Saturno.

A construção do anel está dividido em duas fases. Numa primeira fase, constroem-se os triângulos de maneira a formar um anel com apenas a parte de fora completa. De seguida vamos desenhar os triângulos da parte de dentro do anel. O algoritmo usado para calcular os vértices do triângulo vai ser bastante semelhante ao algoritmo usado para construir um círculo sendo apenas necessárias algumas modificações. A mais evidente é a utilização de dois raios, um deles limita o interior e o outro o exterior do anel. O número de triângulos usados para a construção do anel depende do número de *slices* que o utilizador indica, sendo que o total é igual a *slices**2*2 (isto deve-se ao facto de serem feitos tantos triângulos para o exterior como para o interior do anel e ao facto do anel estar desenhado de forma a ser visto pelos dois lados. Para ser visto de baixo não é necessário calcular de novo os vértices, basta introduzi-los no modelo pela ordem inversa dos anteriores). Na imagem seguinte é possível verificar as contas efectuadas para calcular os vértices dos triângulos usados para a criação do anel. Os ponto b1, b2 e b3 constroem os triângulos da parte exterior do anel, e os restantes pontos, b4, b5 e b6 constroem o interior do anel.

```
bx1 = radius1 * sin((i+1)*alpha);
by1 = 0;
bz1 = radius1 * cos((i+1)*alpha);

bx2 = radius2 * sin((i+1)*alpha);
by2 = 0;
bz2 = radius2 * cos((i+1)*alpha);

bx3 = radius2 * sin(i*alpha);
by3 = 0;
bz3 = radius2 * cos(i*alpha);
bx4 = radius1 * sin((i)*alpha);
by4 = 0;
bz4 = radius1 * cos((i)*alpha);

bx5 = radius1 * sin((i+1)*alpha);
by5 = 0;
bz5 = radius1 * cos((i+1)*alpha);

bx6 = radius2 * sin(i*alpha);
by6 = 0;
bz6 = radius2 * cos(i*alpha);
```

Figura 7: Cálculo dos vértices do anel

4 Sistema Solar

Para criar um modelo estático do sistema solar foi criado um ficheiro XML com os modelos usados para representar os planetas e respectivas luas e o Sol. Os únicos modelos usados para representar todos os objectos do modelo são esferas com a excepção dos anéis de Saturno que usa o modelo de anel. Para além disso, o modelo do Sol é composto por uma esfera com 20 *stacks* e 20 *slices* enquanto que os planetas e luas apenas têm 10. Tendo em conta que alguns planeta possuem um elevado número de Luas decidimos apenas representar algumas delas. Assim sendo, o nosso sistema solar é composto por 8 planetas, 21 luas(1 na Terra, 2 em Marte, 7 em Júpiter, 6 em Saturno, 3 em Úrano e 2 em Neptuno) e o Sol.

Cada planeta tem o seu próprio grupo onde são efectuadas as transformações geométricas necessárias para o representar. Para facilitar o cálculo da posição dos planetas começamos por fazer uma rotação seguida de uma translação para definir a posição do planeta. Uma vez definida a posição do planeta, é feito o *scale* do mesmo para definir o seu tamanho relativo aos restantes planetas. Para além do planeta, um grupo pode conter um ou mais subgrupos para representar as luas. As transformações geométricas efectuadas no grupo do planeta também afectam os subgrupos das luas.

Para facilitar a visualização dos planetas e respectivas luas o modelo não foi feito à escala uma vez que as distâncias entre os planetas e o tamanho do Sol em relação a estes ia tornar a visualização dos mesmos quase impossível, sendo necessário alterar a posição da câmera para os conseguir ver.

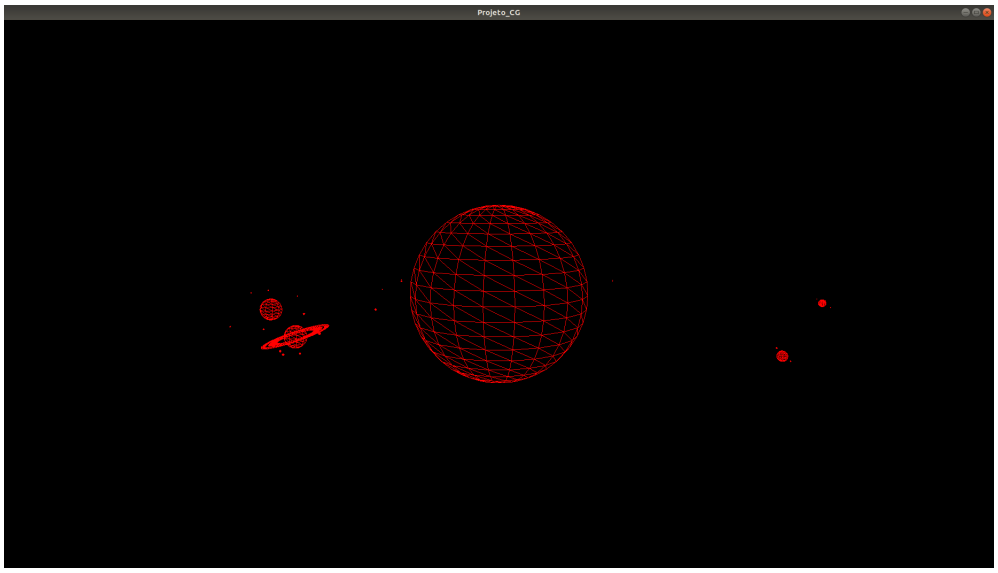

Figura 8: Modelo estático do Sistema Solar

5 Conclusão

Esta fase contribuiu para solidificar alguns conhecimentos adquiridos nas aulas teóricas. Um deles é a importância da ordem das transformações geométricas, do impacto que um simples ato de trocar a ordem destas transformações causa. Esta fase também permitiu familiarizar-mo-nos com as funções do *glut* que permitem pôr na *stack* a matriz relativa à câmara (*glPushMatrix* e *glPopMatrix*), permitindo fazer transformações geométricas que afectam esta matriz e posteriormente poder voltar à matriz inicial.

Por último, foi possível usar os modelos construídos na fase anterior(principalmente a esfera) para a construção de um ficheiro configuração XML que representasse o Sistema Solar.