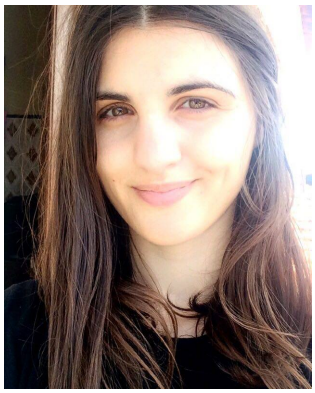
Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Computação Gráfica

Fase 2
Geometric Transforms

Grupo 18



Célia Figueiredo a67637

Luís Pedro Fonseca a60993

Nelson Carvalho Vieira a54764

Braga, 2 de Abril de 2017

Conteúdo

Resumo

O presente relatório descreve o trabalho efetuado para a realização da segunda fase do projeto, onde foi pedido a implementação e representação de um Sistema Solar com o uso da ferramenta *OpenGL*. Nesta fase efetuaram-se alterações dos ficheiros para que estes incluíssem as transformações geométricas e estas pudessem ser aplicadas ao modelo 3D, como por exemplo, ser possível guardar a translação de um objeto.

1. Introdução

No âmbito da Unidade Curricular de Computação Gráfica pertencente ao plano de estudos do 3º ano do Mestrado Integrado em Engenharia Informática foi proposto o desenvolvimento de um sistema solar.

Concluída a primeira fase tem-se agora de introduzir transformações geométricas, *translate*, *rotate* e *scale*, ao projeto. Assim, nesta fase, o *engine* deverá ser capaz de realizar translações, rotações e escalas das várias figuras a desenhar. As transformações geométricas a realizar deverão ser especificadas no ficheiro XML, seguindo um formato definido, assim como as figuras a desenhar.

A demonstração para a segunda fase é um modelo estático do sistema solar com o Sol, Planetas e Luas definidos em hierarquia.

2. Transformações Geométricas

Neste capíulo vamos mostrar quais as transformaçõess que se poderão encontrar e aplicar a cada um dos obejtos que fazem parte das primitivas ou no sistema solar. Sendo que destacamos as três transformações usadas neste projeto:

- Translação - A translação serve para movimentarmos um objeto de um ponto para outro, isto é , podemos ter um objeto num ponto e ao aplicarmos uma translação este muda para uma posição diferente.

  Em OpenGL usamos a função predefinida *glTranslatef()* que recebe três parâmetros cada um correspondente ao deslocamento dada uma direção paralela a um dos eixos ortogonais.

- Rotação - Quando estamos a falar do rotação significa que transformamos um objeto alterando a sua posição rodando em torno de determinado eixo.

  Em OpenGL usamos a função predefinida *glRotatef()* que recebe quatro parâmetros um correspondente ao ângulo e os outros três relativos ao eixo de rotação.

- Escala - significa que estamos a alterar a relação das dimensões do objeto em causa, isto é podemos ter um objeto e torná-lo mais ou menos achatado ou simplesmente aumentar ou reduzir o seu tamanho.

  Em OpenGL usamos a função predefinida *glScalef()* que recebe três parâmetros cada um correspondente à relação da nova dimensão com a dimensão original relativas aos eixos ortogonais.

2.1 Descrição do processo de Leitura

Pretende-se que a leitura do ficheiro de configuração XML da cena assim como de qualquer ficheiro de pontos 3D seja executada apenas uma vez. Deste modo não só os ficheiros como também a estrutura de armazenamento dos dados interna ao software e o modo de leitura foram pensados com esse propósito.

A função *add_read_Model* foi criada para a leitura do modelo contido num ficheiro.

```
void add_read_Model(char* filename, struct sModel** mod){
fopen(filename, LEITURA);
if( existe ficheiro){
lê a primeira linha - lê número de vertices;
Aloca espaço necessário;
for(ler todos os pontos do ficheiro){
guarda o ponto em memória;
```

```
}
}
}
```

A função *xml_read_engine* serve para arrancar a leitura do XML de configuração.

```
void xml_read_engine(const char* filename,Scene s){
    TiXmlDocument d;
    TiXmlElement *root = NULL;
    TiXmlNode *scene = NULL;

    if(consegue abrir o ficheiro) {
        root = d.RootElement();
        scene = procura grupo "scene" no xml;
        if(encontrou "scene") {
            inicializa o grupo na variável global;

            lê recursivamente o ficheiro para a estrutura;
        }
    }
}
```

A função *xml_group_read* tem o objetivo de retirar dos nodos do ficheiro XML os dados referidos nas "tags"através de atributos de forma a armazená-los ordenada e hierarquicamente na estrutura. As tags contidas nesse ficheiro serão as seguintes: *group*, *translate*, *rotate*, *scale*, *color*, *model*. Parte desta função é também exectutada recursivamente.

```
   void xml_group_read(struct sGroup* g,TiXmlNode* node) {
      for(todos os subnodos do "node" de entrada){
         if(tag == "group"){
            xml_group_read(subgrupo, subnodo);
            grupo em tratamento muda para o seguinte;
         }
         if(tag == "translate"){
            retira os valores dos atributos - 3 floats;
            cria um novo comando;
            define o comando como uma translação;
            armazena os valores lidos ;
          }
         if(tag == "rotate"){
            retira os valores dos atributos - 4 floats;
            cria um novo comando;
            define o comando como uma rotação;
            armazena os valores lidos ;
          }

          if(tag == "scale") {
            retira os valores dos atributos - 3 floats;
            cria um novo comando;
```

```
            define o comando como uma escala;
            armazena os valores lidos ;
          }

          if(tag == "color"){
            retira os valores dos atributos - 3 floats;
            cria um novo comando;
            define o comando como uma coloração;
            armazena os valores lidos ;
          }
          if(tag == "model"){
            retira o valor do atributo - 'filename';
            cria um novo comando;
            define o comando como uma modelação;
            armazena o filename lido ;
          }
    }
```

Ou seja, cada nodo *group* é composto por um conjunto de ficheiros *model*, até 3 transformações diferentes e opcionais *translate*, *scale*, *rotate* e uma de aspeto visual *color* e vários nodos filho *group*, ou seja assenta sob uma árvore de *group*. A vantagem desta alternativa é realizar transformações relativas a outro objeto ao invés de absolutas à origem.

Por exemplo, para desenhar as luas de Urano, basta deslocar com base na distância a Urano, ao invés de ser com base na distância ao sol, o que facilitará depois nas translações. Seguindo o exemplo anterior, como Urano realiza uma translação à volta do sol, no ficheiro de configuração Urano fará parte de um elemento que é filho do Sol, e o mesmo para a lua em relação a Urano. As informações necessárias relativamente aos elementos são em todos os casos atributos como *file* para representar o nome de um ficheiro num elemento *model* ou como os eixos e ângulo de uma rotação, segue de seguida um exemplo:

```
<group>
     <translate x='108.099' y='0' z='0'/> URANO
          <group>
               <scale x='5.1108' y='5.1108' z='5.1108'/>
               <model file='sphere.3d'/>
          </group>
          <group>
               <translate x='7' y='-1' z='-1'/> L_UMBRIEL
               <group>
                    <scale x='0.10374' y='0.10374' z='0.10374'/>
                    <model file='sphere.3d'/>
               </group>
          </group>
          <group>
               <translate x='6.5' y='-1' z='-1'/> L_TITANIA
               <group>
                     <scale x='0.10374' y='0.10374' z='0.10374'/>
                     <model file='sphere.3d'/>
               </group>
```

```
            </group>
            <group>
                <translate x='6.7' y='-1' z='-1'/> L_OBERON
                <group>
                    <scale x='0.10374' y='0.10374' z='0.10374'/>
                    <model file='sphere.3d'/>
                </group>
            </group>
</group>
```

Para a leitura do ficheiro foi utilizada a biblioteca *tinyxml* que disponibiliza uma API pré-definida para auxiliar no parser de ficheiros deste tipo.

2.2 Descrição das estruturas de dados para armazenar os Grupos

A estrutura implementada coleciona toda a informação necessária sobre os modelos a carregar e as suas transformações; como tal é composto por 4 elementos principais :

- *scene* é uma estrutura global e vai ser preenchida com a informação contida no fiheiro XML
- *group* é uma estrutura que guarda os comandos e/ou os subgrupos a eles pertencentes
- *command* é uma estrutura que armazena as transformações geométricas e/ou visuais
- *model* é uma estrutura que contem a listagem dos pontos de determinado modelo

De acordo com a estrutura definida pelo ficheiro de configuração, foi necessário criar uma estrutura para carregar toda essa informação. Assim utiliza-se a estrutura *sScene* que é composta pelo conjunto de grupos e modelos, como se pode observar na figura 2.1:

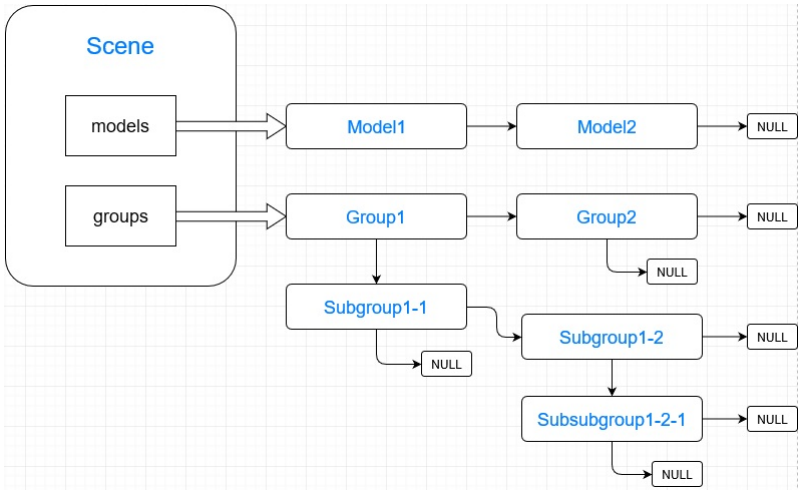


Figura 2.1: Esquema representativo da estrutura de dados utilizada

O modelo da estrutura de dados *sCommand* é implementado com tipo de comando, a informação e um apontador para o seguinte, como pode ser observado na figura 2.2:

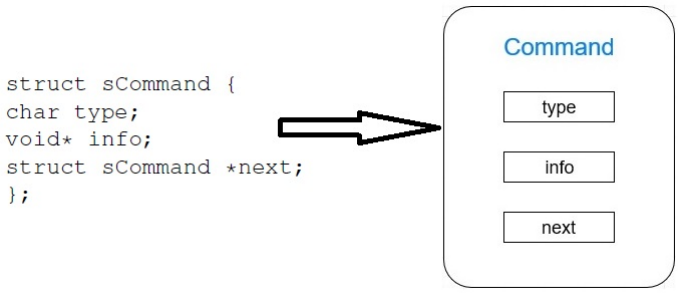


Figura 2.2: Estrutura para Comando

O modelo da estrutura de dados *sGroup* é implementado com uma lista de comandos referente a esse grupo e subgrupos, apontador para o subgrupo e um apontador para o seguinte, como pode ser observado na figura 2.3:

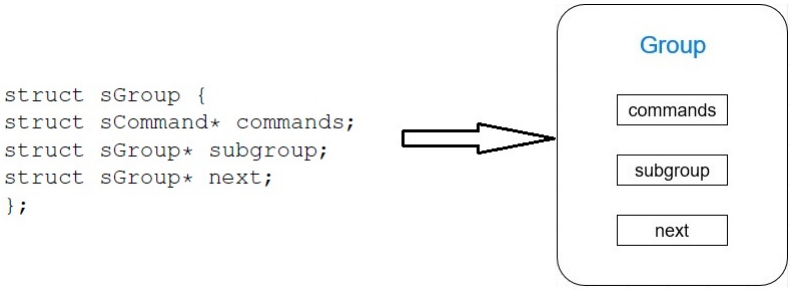


Figura 2.3: Estrutura para Grupo

O modelo da estrutura de dados *sModel* é implementado com um nome, número de pontos gerados, um conjunto de pontos e um apontador para o modelo seguinte, como pode ser observado na figura 2.4:

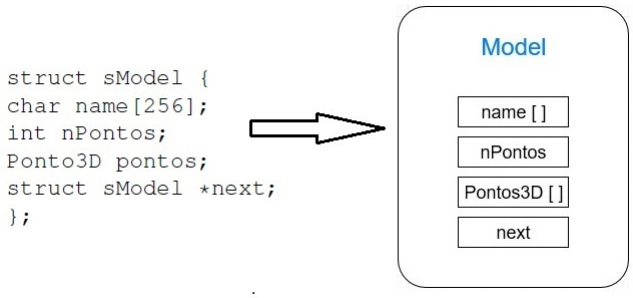


Figura 2.4: Estrutura para o Model

2.3 Descrição do ciclo de rendering

Para a elaboração do rendering foi necessária a criação de funções que visam a conversão de símbolos gráficos num arquivo visual.

A função *drawGroup* é utilizada para arrancar o desenho dos grupos e respetivos subgrupos, mais uma vez de modo recursivo.

```
void drawGroup(struct sGroup* g, Scene s) {
     if(existe grupo) {
          glPushMatrix(); // guarda a matriz
          for(todos os comandos 'c' ){
                drawCommand(c, s);
           }
          drawGroup(subgrupo, s);
          drawGroup(grupo seguinte, s);
          glPopMatrix();// volta para a matriz guardada
    }
}
```

A função *drawcommand* é utilizada para imprimir a lista de comandos referentes às transformações geométricas e impressão dos modelos para o ecrã.

```
void drawCommand(struct sCommand* c, Scene s){
  if(tipo do comando == 'TRANSLAÇÃO' ){
     glTranslatef(dados do comando = 3 floats);
  }
  if(tipo do comando ==  'ROTAÇÃO'){
    glRotatef(Dados do comando = 4 floats);
  }
  if(tipo do comando == 'ESCALA'){
    glScalef(Dados do comando  = 3 floats);
  }
  if(tipo do comando == 'COR'){
    glColor3f(Dados do comando  = 3 floats);
  }
  if(tipo do comando == 'MODELAR'){
    drawModel(Dados do comando = nome do ficheiro, cena global );
  }
}
```

A função *drawModel* é utilizada para desenhar os pontos correspondentes ao modelo em questão utiliza as funções *add_read_Model* já explicadas anteriormente na secção do processo de leitura.

```
void drawModel(char* model, Scene s) {
   while(não for o modelo desejado){
      avança entre os modelos;
   }
   if(não existir modelo){
```

```
      add_read_Model(model,&(s->models));// lê o modelo do ficheiro
      drawModel(model,s); //volta a tentar desenhar o modelo
   } else {
      glBegin(GL_TRIANGLES);
      for(todos os pontos ){
         glVertex3f(ponto);
      }
      glEnd();
   }
}
```

3. Demo

A Demo, para a segunda fase do projeto, corresponde ao modelo estático do Sistema Solar, e contem o Sol, os Planetas e Luas. Para a elaboração da Demo começou-se por criar o Sol que se encontra no centro do Sistema Solar, centrado nas coordenadas (0,0,0). Em seguida foram desenhados os planetas e respetivas luas pela sua ordem.

Apesar de as distâncias entre os Planetas não estarem à escala, no que toca aos planetas tentou-se ser o mais fiel possivel, mas devido ao facto de o Sol ser imensamente maior que grande parte dos Planetas tivemos que fazer algumas alteraçõs às suas escalas, de modo a ser possível representar todos os Planetas e algumas das luas do Sistema Solar, não tendo planetas imensamente pequenos o que dificultaria a sua visualização. Esta decisão foi também tomada devido ao facto de que se seguíssemos as escalas reais à risca a Lua da Terra não seria desenhada por ser demasiado pequena.

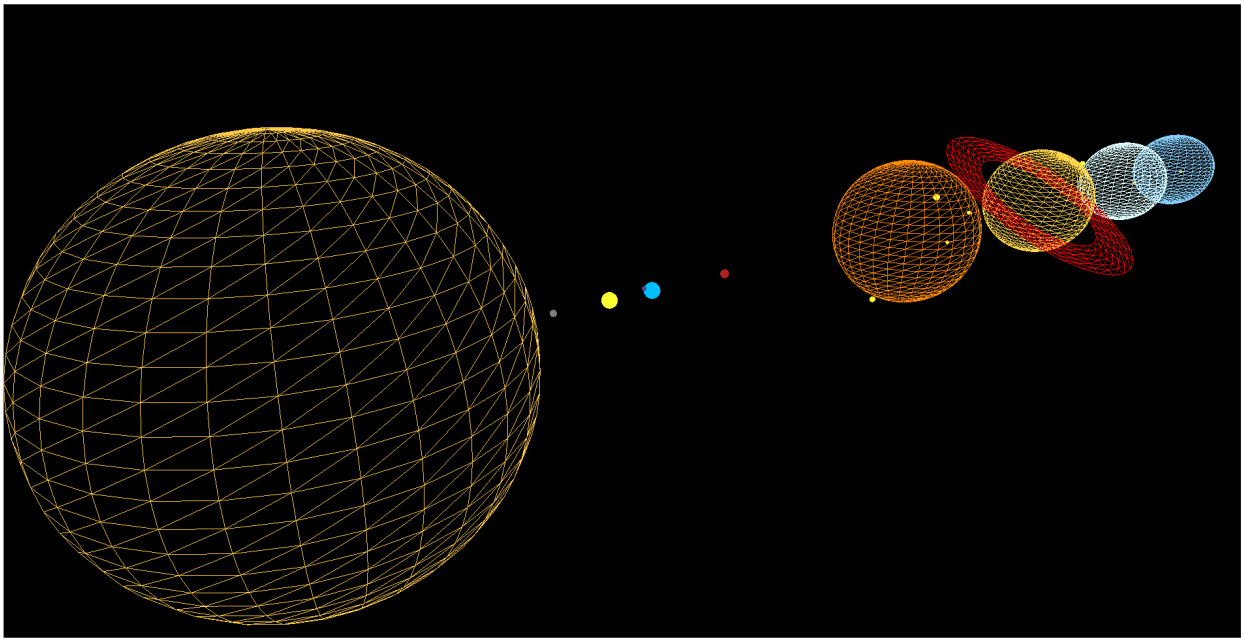


Figura 3.1: Sistema Solar com os planetas desenhados e Sol no centro

Para desenhar Mercúrio e Vénus não existe grande diferença entre eles e o Sol, fazendo uma escala e uma translação colocamos estes planetas no sítio que desejamos com o tamanho mais indicado. Para os restantes planetas já temos que usar uma relação Pai-Filho, devido ao facto, de os restantes planetas a serem desenhados conterem uma ou mais Luas, ou, no caso de Saturno, um anel. Essa relação é feita no ficheiro XML colocando as luas ou aneis como filhos do Planeta a que pertencem.

4. Conclusão

Nesta fase procederam-se a vários ajustes a nível da estrutura das classes, mas também otimizações de código. Assumir isto como uma prioridade significou aliviar a implementação de novas funcionalidades futuras que podem ser mais exigentes a nível de processamento. Culmatou-se desta forma alguns dos problemas de desorganização do código apresentados anteriormente.

Foram realizadas algumas alterações a nivel da câmara, dando um maior controle ao utilizador, permitindo, para além do que já podia realizar na primeira fase, fazer zoom in e zoom out e melhorou-se a rotação da câmara.

Por outro lado, tentamos estar atentos a possíveis problemas de otimização, tal como a leitura repetida de ficheiros iguais. Por exemplo, para dois planetas definidos pelo mesmo ficheiro, a estrutura carregada poderia ter essa informação repetida, e o ficheiro poderia ser lido duas vezes, o que não está a acontecer.

Em suma, achamos que, com esta fase 2, definiu-se as estruturas base a utilizar e simplificou-se a introdução de novas funcionalidades tendo-se melhorado as funcionalidades já existentes.