Computação Gráfica

3º ano de LCC/MIEI

2º Fase de Entrega

Relatório de Desenvolvimento

Geometric Transforms

João Gomes

a70400

João Dias

A72095

Joel Morais

A70841

Luís Ventuzelos

a73002

31 de Março de 2017

Resumo

Este documento apresenta o desenvolvimento do projeto de *Computação Gráfica* numa colaboração entre o Mestrado Integrado de Engenharia Informática *(MIEI)* e a Licenciatura de Ciências de Computaçao *(LCC)* correspondente ao ano lectivo *2016/2017*, analisando as estratégias utilizadas e as principais funções implementadas em cada uma das tarefas propostas.

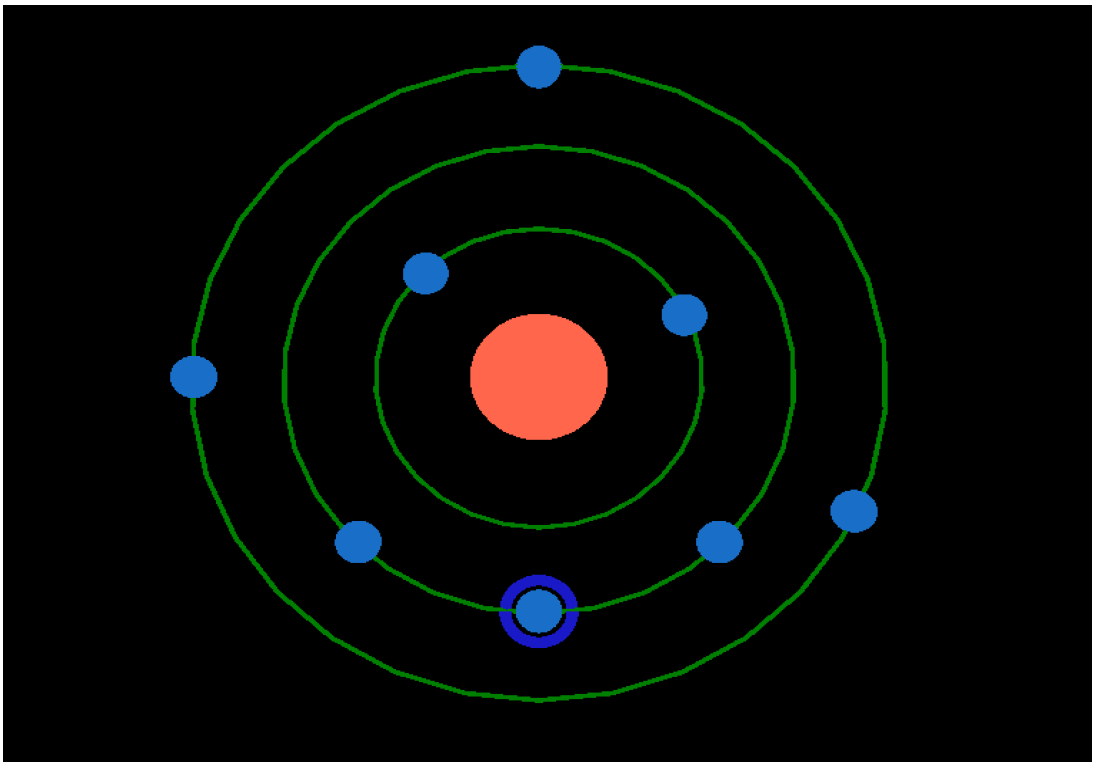

Figura 1: Sistema Solar

Capítulo 1

Introdução

No âmbito da unidade curricular de *Computação Gráfica* foi-nos proposto o desenvolvimento de um projeto dividido em quatro fases. Este relatório refere-se à segunda fase, na qual o objetivo é a criação de um Sistema Solar através de sucessivas transformações geométricas. Essencialmente, nesta fase do projecto, procedemos à alteração do nosso motor de maneira a ele poder ler transformações geométricas de um ficheiro XML e proceder ao seu respetivo desenho.

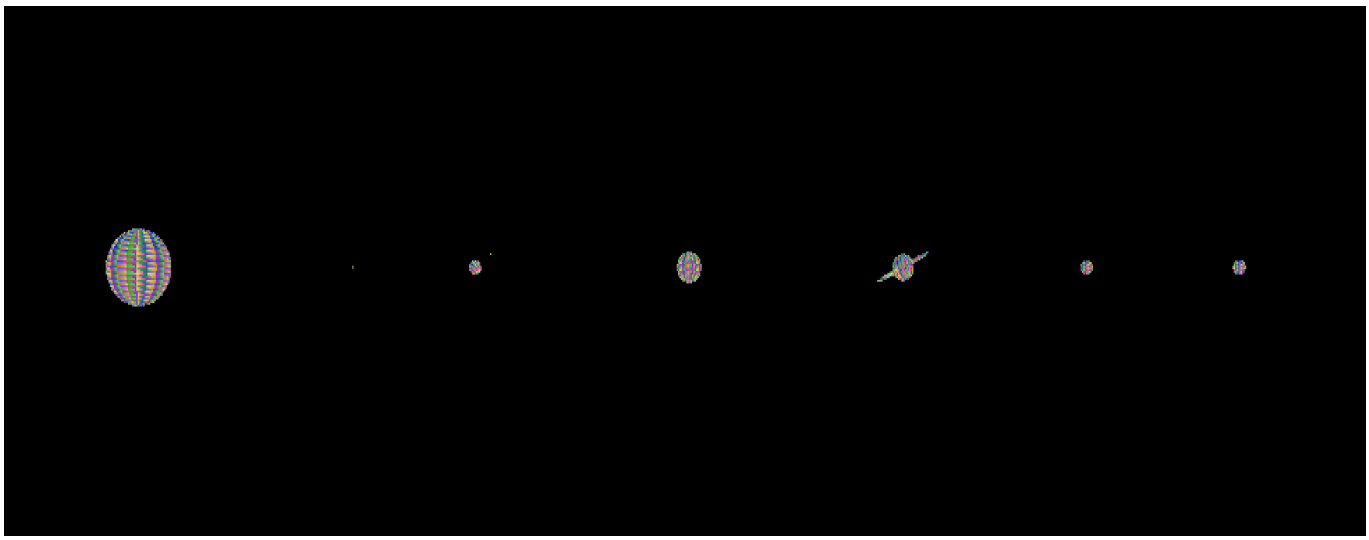

Figura 1.1: Sistema Solar

Capítulo 2

Motor

Neste capítulo vamos explicar o funcionamento do *Motor* do nosso programa, desde as estruturas utilizadas até ao desenho do modelo.

2.1 Estruturas de dados

Nesta secção vamos explicar como funcionam as estruturas de dados escolhidas, e a razão pela qual as escolhemos.

2.1.1 Classe Model

A classe **Model** terá a função de guardar os pontos para desenhar cada modelo, isto é, os pontos que o nosso *Gerador* escreveu no ficheiro **.3d** que vamos utilizar para o desenho de um certo modelo. Assim, a classe **Model** terá os seguintes campos:

- nome;
- pontos;

O campo **nome** será o nome do ficheiro **.3d** que vamos utilizar, e este poderá servir para sabermos se vamos desenhar um planeta ou um anel de um planeta, com o objetivo de utilizarmos cores diferentes para cada tipo de figura.
O campo **pontos**, é um vector de *float*, que terá os pontos guardados no ficheiro **.3d** em questão.

```
class Model{
public:
    string nome;
    vector <float> pontos;
};
```

2.1.2 Classe Group

A classe **Group** tem os seguintes campos:

- transf;
- mod;
- grup;

O map **transf** vai conter, pela ordem que é lido no ficheiro **XML**, a transformação geométrica aplicada (que será a chave) e os valores de **x**,**y** e **z** de cada transformação geométrica e o ângulo(**ang**) no caso das rotações, estes ultimo

valores (guardados num vetor de *float*), formam o valor associado a cada chave do *map <int,vector<float>>*.
O vector **mod** vai conter cada um dos modelos que vamos desenhar, do grupo em questão.
O vector **grup** serve para o caso de um determnado grupo ter subgrupos, e nesse caso, guardamos os dados desses subgrupos neste vector de grupos.

```
class Group{
public:
    map<int,vector<float> > transf;
    vector <Model*> mod;
    vector <Group*> grup;
};
```

2.1.3 Classe Scene

A classe **Scene** vai ser a primeira classe a ser inicializada, isto é, dentro do campo de um objeto do tipo **Scene** vamos guardar todos os dados presentes no ficheiro **XML** lido. Esta classe tem apenas o campo, que vai conter todos os dados relativos a cada **Group** do ficheiro XML.

```
class Scene{
public:
    vector<Group*> grupos;
};
```

Assim, de um modo geral, a relação entre cada uma das estruturas será a seguinte:

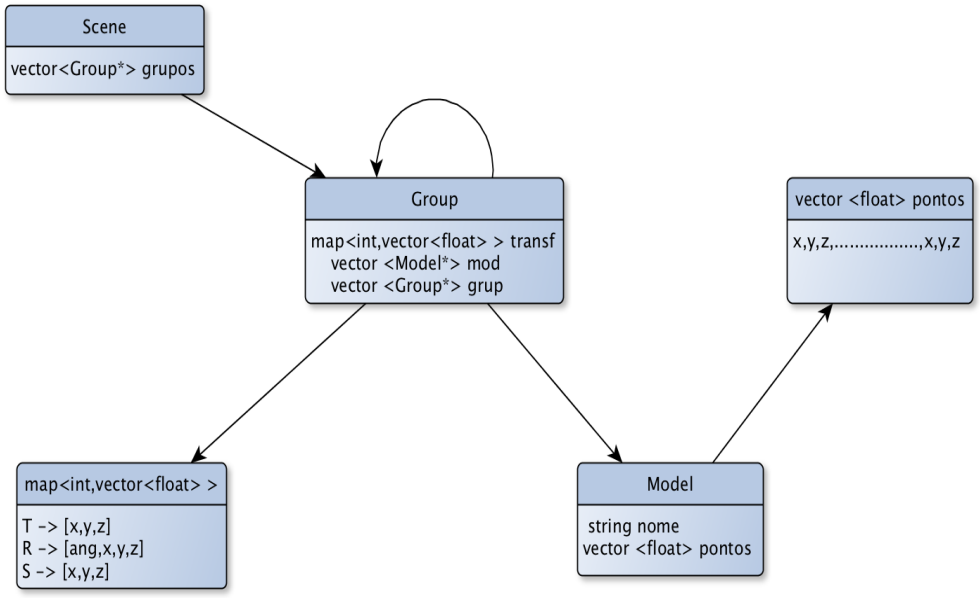

Figura 2.1: Classes

2.2 Leitura do Ficheiro.xml

Usando a biblioteca de parsing para XML, o *TinyXML-2*, procedemos ao carregamento e subsequente à leitura do nosso ficheiro**XML**.
Primeiramente carregamos o ficheiro da seguinte maneira:

```
ficheiro = "Sistema.xml";
XMLDocument doc;
int loadOkay = doc.LoadFile(ficheiro.c_str());
```

e guardamos em scene (que é de tipo XMLElement) como esta exemplificado em baixo:

```
 XMLElement *scene = doc.FirstChildElement("Scene");
```

De seguida inicializamos o objeto c com o construtor vazio da classe **Scene** chamamos a função beginParsing.

```
c = new Scene();
c = beginParsing(scene);
```

2.2.1 Função beginParsing()

A função **beginParsing** recebe como argumento **XMLElement* cena** que vai armazenar as características de cada Grupo, no vetor **grupos**, campo da classe **Group**.
Esta função devolve um objeto do tipo **Scene**, que mais tarde servirá como argumento da função **drawScene**.

2.2.2 Função readGroup()

Esta função, para cada grupo que temos no ficheiro **XML**, vai criar uma classe **Group** correspondente a esse grupo, com os seus modelos e subgrupos.
No exemplo de baixo podemos ver como tratamos de uma translação no nosso programa. O mesmo será feito para as outras transformações.

```
}else if(tag == "Translate"){
    atri = filho->Attribute("x");
    if(atri != NULL) aux.push_back(atof(atri));
    atri = filho->Attribute("y");
    if(atri != NULL) aux.push_back(atof(atri));
    atri = filho->Attribute("z");
    if(atri != NULL) aux.push_back(atof(atri));
    g->transf.insert(pair<int,vector<float> >(T,aux));
    aux.clear();
}
```

Por sua vez, se dentro de um grupo temos outro grupo(ou seja um subgrupo), chamamos recursivamente este método para criarmos um novo objeto **Group** por cada um dos subgrupos encontrados.

```
if(tag == "Group"){
    g->grup.push_bac(readGroup(filho);
}
```

Para terminar vamos armazenar os ficheiros .3d, que no caso do ficheiro **XML** são os nossos **Models**, através da função **read_File3D()** que será explicada de seguida.

```
else if(tag == "Models")
       { /** UMA TAG MODELS SO PODE TER UM MODEL **/
           XMLElement* neto = filho->FirstChildElement();
           ficheiro = neto->Attribute("file"); // vai buscar os atributos de file
           if (ficheiro != NULL) {
               g->mod.push_back(read_File3D(ficheiro));
           }
       }
```

2.2.3 Função read_File3D()

Esta função simplesmente lê os pontos guardados no ficheiro .3d em causa e guarda no vetor **pontos** (da classe **Model**) do modelo que estamos a ler.

Em suma os passos do nosso algoritmo de parsing estão representados pelo seguinte autómato:

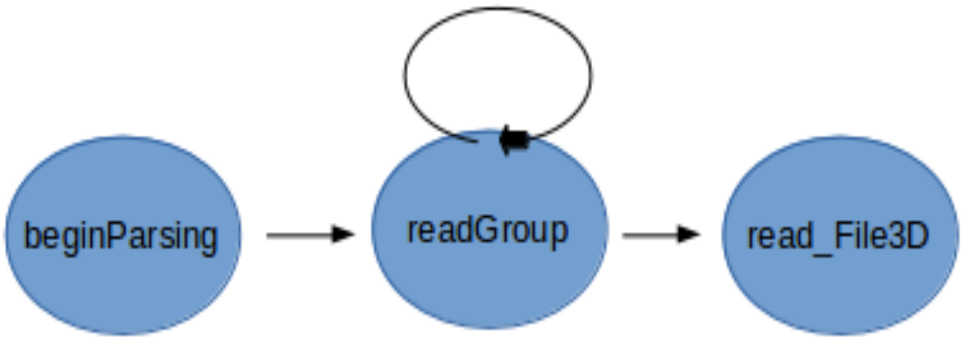

Figura 2.2: Parsing

2.3 Desenho das figuras

2.3.1 Função drawScene()

Depois de armazenarmos os nossos grupos no objeto da classe **Scene** procedemos ao **renderScene**, e aqui chamamos a função **drawScene** que simplesmente desenha cada um dos grupos que lemos no XML, do primeiro grupo para o ultimo.

```
for (i=0; i < c->grupos.size();i++){
       drawFigures(c->grupos[i]);
```

2.3.2 Função drawFigures()

Através das funções **glPushMatrix()** e **glPopMarix()** controlamos as transformações que se aplicam a determinados objetos, regendo-nos pela leis de hierarquia e de hereditariadade, ou seja, um grupo de objetos só é afetado pelas transformações de grupos que estão hierarquicamente acima herdando assim essas transformações (um pai transmite as suas transformações ao filho porém um filho ou irmão não as passa ao seu pai/irmão). Assim, após a execução de destes dois comandos, voltamos ao estado inicial, isto é, todas as transformações geométricas que aplicamos deixam de ter efeito para as próximas figuras a desenhar

Nesta fase foram executados 3 passos essenciais:

1º - Fazemos as transformações que lemos no XML (percorrendo o vetor ordem, com o iterator **o**)

```
 for (it=p->transf.begin(); it != p->transf.end(); ++it){
       aux = it->second;
       if(it->first == T){
           glTranslatef(aux[0],aux[1],aux[2]);

       }else if(it->first == S){
           glScalef(aux[0],aux[1],aux[2]);

       }else if(it->first == R){
           glRotatef(aux[0],aux[1],aux[2],aux[3]);
```

2º - Desenhamos as figuras desenhando triângulos, com a função **glVertex3f**, lendo os pontos do vetor **pontosAux** que é uma copia do vetor **pontos** do modelo que estamos a tratar. Para lermos cada ponto, lemos 3 a 3 elementos do vetor mencionado. Nesta função variámos também as cores de cada triângulo que desenhamos. De seguida, podemos observar o explicado acima:

```
       glBegin(GL_TRIANGLES);
       float red=0.2,green=0.5,blue=0.1; // para ir mudando as cores
       for(i = 0; i < pontosAux.size(); i+=3){ /** desenha triangulos, lendo 3 a 3 do vetor de pontos **/
           glColor3f((GLfloat) fmod(red, 1.0), (GLfloat) fmod(green, 1.0), (GLfloat) fmod(blue, 1.0));
           glVertex3f(pontosAux[i], pontosAux[i+1], pontosAux[i+2]);
           red+=0.1;green+=0.2;blue+=0.2;
       }
       glEnd();
   }
```

3º - Caso haja subgrupos, chamamos recursivamente a função **drawFigures()**

```
vector<Group*>::iterator k;
  for(k = p->grup.begin();k != p->grup.end();++k){
      drawFigures(*k);
  }
```

Em suma os passos do nosso algoritmo de rendering estão representados pelo seguinte autómato:

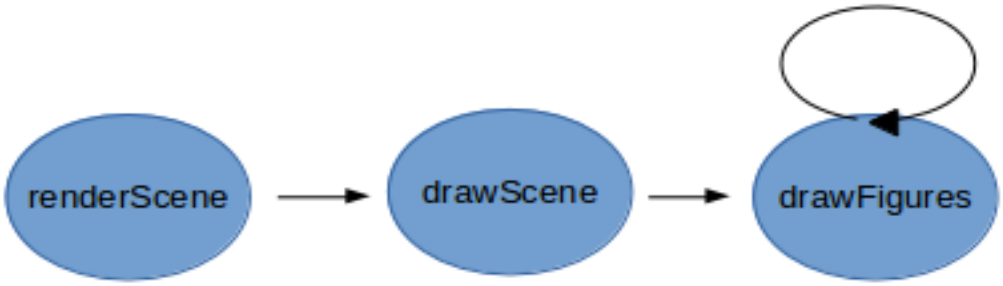

Figura 2.3: Draw

Capítulo 3

Sistema Solar

Neste capítulo vamos explicar como geramos cada um dos planetas, explicando as transformações geométricas que foram aplicadas, e a sua respetiva justificação.
No Sistema Solar temos 10 corpos celestes:

- Sol;
- Mercúrio;
- Vénus;
- Terra;
- Lua;
- Marte;
- Júpiter;
- Saturno:
- Urano;
- Neptuno;

O resultado final, após tods as transformações geométricas que iremos explicar, o nosso sistema solar será o seguinte:

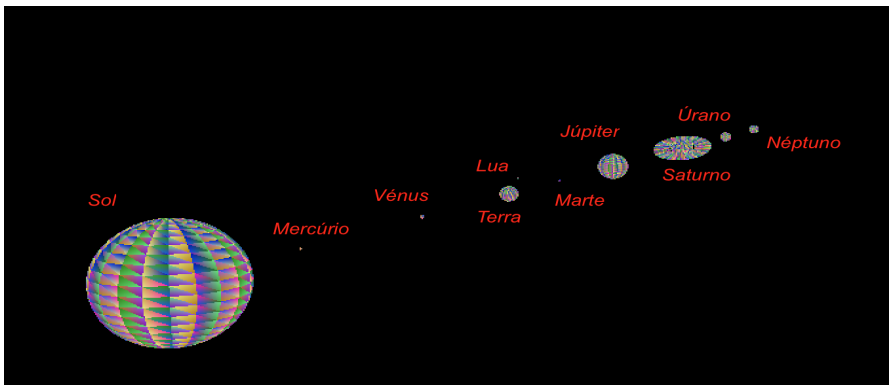

Figura 3.1: Sistema Solar

3.1 Planetas

Todos os planetas e a lua do planeta Terra são uma esfera de raio 1, que após algumas transformações geométricas, fazemos com que umas tenham dimensões diferentes das outras e que estejam a distâncias diferentes em relação ao Sol. Para a criação de cada um dos planetas utilizamos o nosso gerador para desenhar uma esfera centrada na origem. No caso do planeta Saturno desenhamos também os seus anéis e, para isto, decidimos adicionar uma nova figura ao nosso gerador, à qual chamamos **anel**, que iremos explicar mais à frente. O planeta Terra também é um planeta que se distingue dos demais, por ser o planeta ao qual chamamos casa extraímos daí um certo sentimento nostálgico e decidimos muni-lo de um satélite natural, vulgarmente denominado de Lua, sendo essa lua tal como os outros astros uma esfera de raio 1. Nas seguintes secções vamo-nos focar apenas nos planetas Terra e Saturno, visto que são diferentes dos restantes.

3.1.1 Saturno

Para desenharmos Saturno, basta termos uma esfera centrda na origem e o seu anel, também centrado na origem. Para gerarmos o anel de Saturno baseamo-nos na estratégia de desenho de um cilíndro com slices e stacks, isto é, para cada slice desenhamos as **n** stacks, sem nos podermos esquecer do túnel no interior do anel. Assim, são necessários os seguintes valores para a construção do anel:

- raio interior(**rI**)
- raio exterior(**rO**)
- altura
- stacks
- slices

O raio total (ao qual chamaremos **raio**) é a soma de **rI** com **rO**. Com a altura do anel e o número de stacks conseguimos calcular a altura de cada uma das stacks, dividindo a altura total pelo número de stacks, isto é, **h** = altura/stacks. Para se saber a altura da stack que estamos a desenhar basta: **hC** = iteração*h; para sabermos a altura da próxima camada: **hP** += h.
A grande diferença entre o anel e um cilindro é que cada fatia da base não é formada apenas por 1 triângulo, mas por dois, pois é como se cortassemos a ponta de um triângulo, tal como podemos observar na figura:

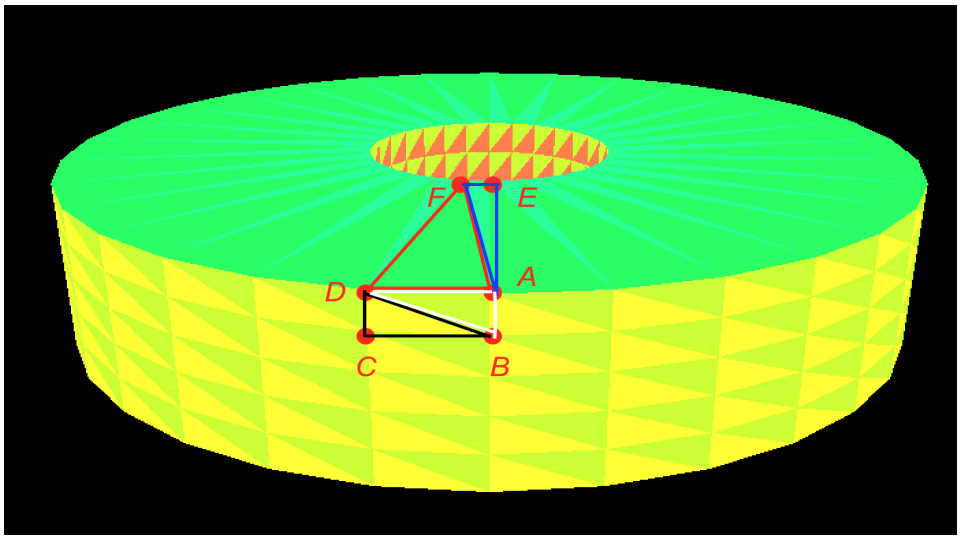

Figura 3.2: Anel

Analisando a figura acima e considerando **ang** como o ângulo da iteração atual,**angP** o ângulo da próxima iteração, utilizando as equações cartesianas da circunferência, conseguimos chegar as coordenadas de cada um dos pontos ilustrados:

- **A** = (raio*sin(angP),hP,raio*cos(angP))
- **B** = (raio*sin(angP),hC,raio*cos(angP))
- **C** = (raio*sin(ang),hC,raio*cos(ang))
- **D** = (raio*sin(ang),hP,raio*cos(ang))
- **E** = (rI*sin(angP),hP,rI*cos(angP))
- **F** = (rI*sin(ang),hP,rI*cos(ang))

Para desenharmos estes pontos temos que considerar que 1 slice corresponde a 1 iteração do ciclo exterior, e uma stack corresponde a 1 iteração do ciclo interior, seguindo a regra da mão direita. Para desenharmos o tunel que se observa na figura, em cada iteração do ciclo interior além de desenharmos as stacks exteriores, temos tambem que desenhar as stacks interiores, isto é, mais quatro pontos e, assim, temos os seguintes 4 pontos, os pontos **F** e **E** e outros dois que serão duais aos pontos **B** e **C** apresentados acima, estas stacks terão de ser desenhadas com a regra da mão esquerda, senão não as vamos ver:

- **K** = (rI*sin(angP),hC,rI*cos(angP))
- **P** = (rI*sin(ang),hC,rI*cos(ang))

Assim, numa versão de pseudocodigo a função que desenha o anel será a seguinte:

```
void drawRing(rI,rO,height,slices,stacks){
   h  = height/stacks,heightCurrent,heightProx;
   ang,angP,radius = rI+rO;

   for(i=0;i<slices;i++){
        heightCurrent=0,heightProx=h;
        ang = i*2*M_PI/slices;
        angP = (i+1)*2*M_PI/slices;

        desenhaFatiaTopo();
        desenhaFatiaBase();

        for(int j=0; j < stacks;j++){
         // face de fora (regra da mao direita)
         triangulo(B,D,C);
         triangulo(A,D,B);

         /// face de dentro (regra da mao esquerda)
         triangulo(F,E,K); // K -> B(dual)
         triangulo(P,F,K); // P -> C(dual)

         heightCurrent = heightProx; // altura da fatia i+1
            heightProx+=h; // altura da fatia i+2
        }
   }
}
```

Desenhando o Saturno e o seu anel, obtemos esta figura:

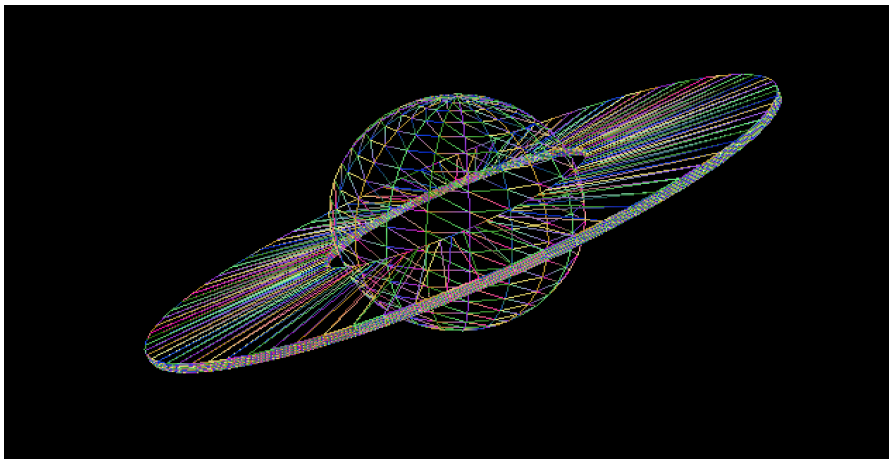

Figura 3.3: Saturno

3.1.2 Terra

Para o desenho do planeta Terra e da Lua iremos necessitar de duas esferas.

Para a criação da Terra iremos fazer uma translação a partir do Sol, no caso da Lua, aplicamos umas translação no sistema da Terra e depois procedemos ao desenho da Lua, tal como descrito na seguinte figura:

```
<!-- Terra + Lua -->
<Group>
    <Translate x="11" y="0" z="0"/>
    <Rotate ang="35" x="0" y="0" z="1"/>
    <Group>
        <Scale x="0.2" y="0.2" z="0.2"/>
        <Models>
            <Model file ="esfera.3d" />
        </Models>
    </Group>
    <Group>
        <Translate x="0.6" y="0" z="0"/>
        <Scale x="0.02" y="0.02" z="0.02" />
        <Models>
            <Model file ="esfera.3d" />
        </Models>
    </Group>
</Group>
```

Figura 3.4: Terra e Lua - xml

Assim, localmente, este XML irá desenhar os dois corpos da seguinte maneira:

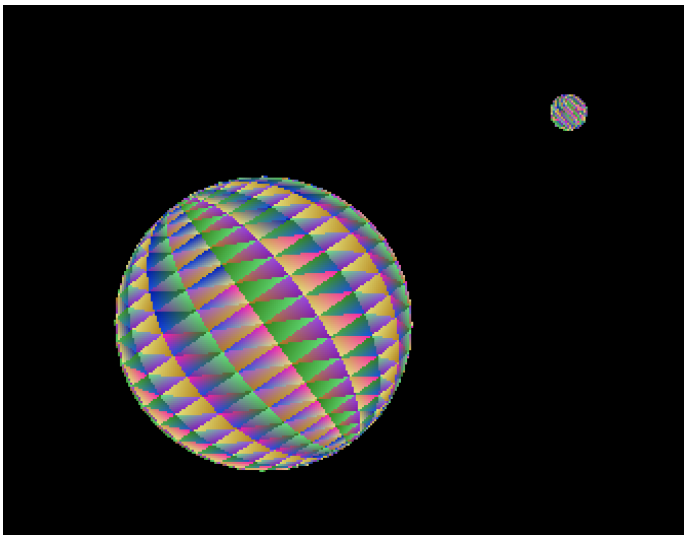

Figura 3.5: Terra e Lua - glut

Capítulo 4

Ficheiro XML

Como referimos anteriormente, existe no ficheiro de formato **XML** a informação relativa às transformações que vamos aplicar a cada grupo e às respetivas figuras. Através das nossas funções transferimos esses dados para uma estrutura intermédia, sobre os quais o nosso algoritmo de desenho é aplicado, consequentemente produzindo o sistema solar. Porém o nosso ficheiro **XML** tem que seguir algumas regras de estruturação para que o nosso algoritmo de parsing seja capaz de retirar toda a informação necessária de forma correta. Assim, vamos dar agora uma maior atenção a essas regras, para que não haja problemas em futuros testes da aplicação. Como sabemos a ordem em que as transformações geométricas são feitas é relevante, portanto podemos garantir que as transformações dos **Group** filhos não passam para os seus irmãos isolando cada transformação dentro de uma tag **Group**.Além disto, dentro de um **Group** não podemos ter transformações repetidas. Temos também algumas regras mais especificas que têm obrigatoriamente que ser cumpridas, para garantirmos o bom funcionamento da aplicação:

1ª regra:
O ficheiro **XML** deve ser inicializado e terminado com a tag **Scene**;

2ª regra:
Cada grupo é inicializado e terminado com a tag **Group**;

3ª regra:
Uma translação corresponde à tag **Translate**;

4ª regra:
Uma rotação corresponde à tag **Rotate**;

5ª regra:
Uma escala corresponde à tag **Scale**;

6ª regra:
Dentro de uma tag **Models** apenas pode existir um modelo;

7ª regra:
Um modelo corresponde à tag **Model**;

8ª regra:
Uma transformação não pode ter apenas o valor do eixo que vai ser alterado, tem que conter o valor de todos os eixos, sendo seguintes valores, os apropriados para os eixos que não serão afetados pela transformação:

- **Translate** - 0;
- **Scale** - 1;
- **Rotate** - 0 (para os eixos sobre os quais não vamos fazer a rotação);

9ª regra:
Para cada translação a designação dos eixos deve ser a seguinte:

- Eixo X - **x**;
- Eixo Y - **y**;
- Eixo Z - **z**;

No caso de uma rotação, devemos também incluir o respectivo angulo, que deverá ter a seguinte designação: **ang**;
Podemos agora ver um pequeno excerto do ficheiro **XML** que escrevemos para gerar o nosso sistema solar, com todas as regras acima aplicadas:

```
<!-- Terra + Lua -->
<Group>
    <Translate x="11" y="0" z="0"/>
    <Rotate ang="35" x="0" y="0" z="1"/>
    <Group>
        <Scale x="0.2" y="0.2" z="0.2"/>
        <Models>
            <Model file ="esfera.3d" />
        </Models>
    </Group>
    <Group>
        <Translate x="0.6" y="0" z="0"/>
        <Scale x="0.02" y="0.02" z="0.02" />
        <Models>
            <Model file ="esfera.3d" />
        </Models>
    </Group>
</Group>
```

Figura 4.1: Exemplo XML

Capítulo 5

Conclusão

A realização deste projeto tem como objetivo a criação de um sistema solar implementado , no nosso caso, com 10 corpos celestes (o Sol,8 planetas e 1 lua). Consideramos que esta fase se dividiu essencialmente em quatro etapas chave:

- Criação do ficheiro XML de acordo com as regras propostas no enunciado;
- Criação de uma estrutura de dados que sirva de suporte para a leitura do ficheiro XML;
- Funções de leitura do ficheiro XML;
- Funções de desenho das figuras armazenados na estrutura de dados;

Consideramos esta fase mais trabalhosa que a primeira pois deparamo-nos com algumas dificuldades que nos impossibilitaram um percurso mais direto, mas ,mesmo assim, não menos conciso que o realizado.

A primeira, e maior dificuldade encontrada, foi a passagem do ficheiro XML para a nossa estrutura de dados, pois para além de haver a necessidade óbvia dessa transferência de informação para uma estrutura intermédia, ficamos agora com a questão sobre a própria identidade dessa estrutura. Conseguir que o grupo chegasse a um consenso sobre a esqueleto desta estrutura para a resolução deste problema foi o ponto mais importante e demoroso da primeira fase, pois existe a noção que as nossa escolhas inicias serão repercutidas ao longo do projeto.

A segunda dificuldade encontrada foi na criação dos anéis de Saturno, dificuldade pois não sabíamos se a criação de um novo objeto geométrico seria uma transgressão e ao faze-lo entravamos numa área cinzenta relativamente ao objetivo deste mesmo projeto, porem, após dialogo com o docente essas duvidas forem esclarecidas e consequentemente o problema foi ultrapassado.

Depois de ultrapassadas estas questões as restantes etapas do nosso projeto foram realizadas com relativa facilidade.

Todavia, apesar do sucesso óbvio deste projeto e da resposta a todos os objetivos aos quais nos propomos, certas arestas deste trabalho poderiam ter sido ainda mais limadas, havendo assim espaço para um melhoramento. Um desses exemplos seria a melhoria dos nossos algoritmos de parsing, do ficheiro XML, e respetivo desenho das figuras pois, apesar concluirmos esse objetivo de forma satisfatória, acreditamos que com mais tempo teríamos uma estrutura de dados ainda mais simples e eficiente.

Resumidamente, consideramos a realização desta fase foi satisfatória visto que cumprimos com todos os requisitos pedidos.

Bibliografia

[1] http://www.grinninglizard.com/tinyxmldocs/annotated.html

[2] http://www.grinninglizard.com/tinyxml2/

[3] http://www.cplusplus.com

[4] http://joshworth.com/dev/pixelspace/pixelspace_solarsystem.html

[5] Documentação fornecida pelo professor