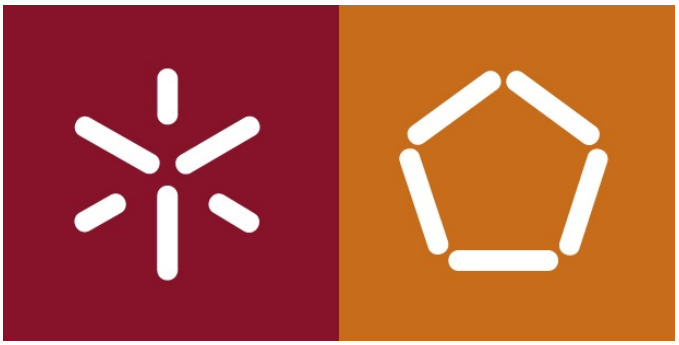
Universidade do Minho

Mestrado Integrado em Engenharia Informática



Computação Gráfica

Relatório do Trabalho Prático 2ªFase

Geometric Transforms

Grupo 36

Carlos Gomes
A77185

Nuno Silva
A78156

25 de Março de 2019

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Foi nos proposto, através da unidade curricular Computação Gráfica, a criação de um mecanismo 3D baseado num cenário gráfico, mediante do qual teríamos de utilizar várias ferramentas, entre as quais C++ e OpenGl.

Este trabalho foi dividido em quatro partes, sendo esta segunda fase tem como objetivo a criação de cenários hierárquicos usando transformações geométricas, tendo como finalidade a criação de um sistema solar estático.

1.2 Resumo

Tendo em conta se tratar da segunda parte do projeto prático, é natural que se mantenha algumas funcionalidades criadas na primeira parte, e algumas delas terem sido alteradas, para assim se cumprir com os requisitos necessários.

Com isto, verficamos que a principal mudança que surge nesta fase está inteiramente relacionada com a forma como o *engine*, previamente criado na fase anterior, lê e processa a informação contida nos ficheiros **XML** que irá receber.

A estrutura dos ficheiros, anteriormente referidos, sofreu uma grande mudança, agora em vez de estes conterem unicamente o nome dos ficheiros com as primitivas que se pretende exibir, estes contêm a formação de diversos grupos hierárquicos com esses mesmos ficheiros. Estes grupos têm associado a si diversas transformações geométricas, tais como *translate*, *rotate* e *scale* que serão responsáveis pelo modo como cada uma das primitivas, previamente criadas na fase anterior, são exibidas.

Assim, será necessário, não só alterar a forma como o nosso *engine* lê estes ficheiros, como também a forma como este processa essa mesma informação. Deste modo, será

necessária a criação de novas classes que terão como objetivo armazenar e relacionar esta mesma informação.

Assim, este conjunto de modificações tem como finalidade conseguirmos gerar e exibir primitivas gráficas que, no seu conjunto representem um modelo estático do Sistema Solar.

Capítulo 2

Arquitetura do Código

Tendo em mente a continuação do trabalho desenvolvido na fase anterior, e de várias opções de desenvolvimento, foi decidido manter as duas aplicações (**generator** e **engine**). Sendo esta última alvo de algumas modificações mais acentuadas tendo em vista o cumprimento dos requisitos necessários. Ficou tambem decidido a alteração da estrutura do trabalho, de forma a torna-lo mais organizado.

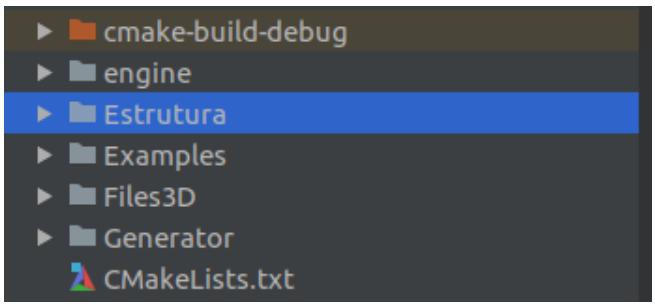


Figura 2.1: Organização do trabalho

2.1 Aplicações

Esta secção é destinada a elucidação sobre as aplicações necessárias para exibir e gerar os diversos modelos requiridos. Uma vez que houve alteração da estrutura dos ficheiros de configuração escritos em **XML**, foi obviamente notória a necessidade de alterar a forma como o *engine* processa esses mesmos ficheiros.

2.1.1 Gerador

generator.cpp : Aplicação onde estão definidos os métodos que posteriormente irão gerar os respetivos vértices das diferentes formas geométricas.

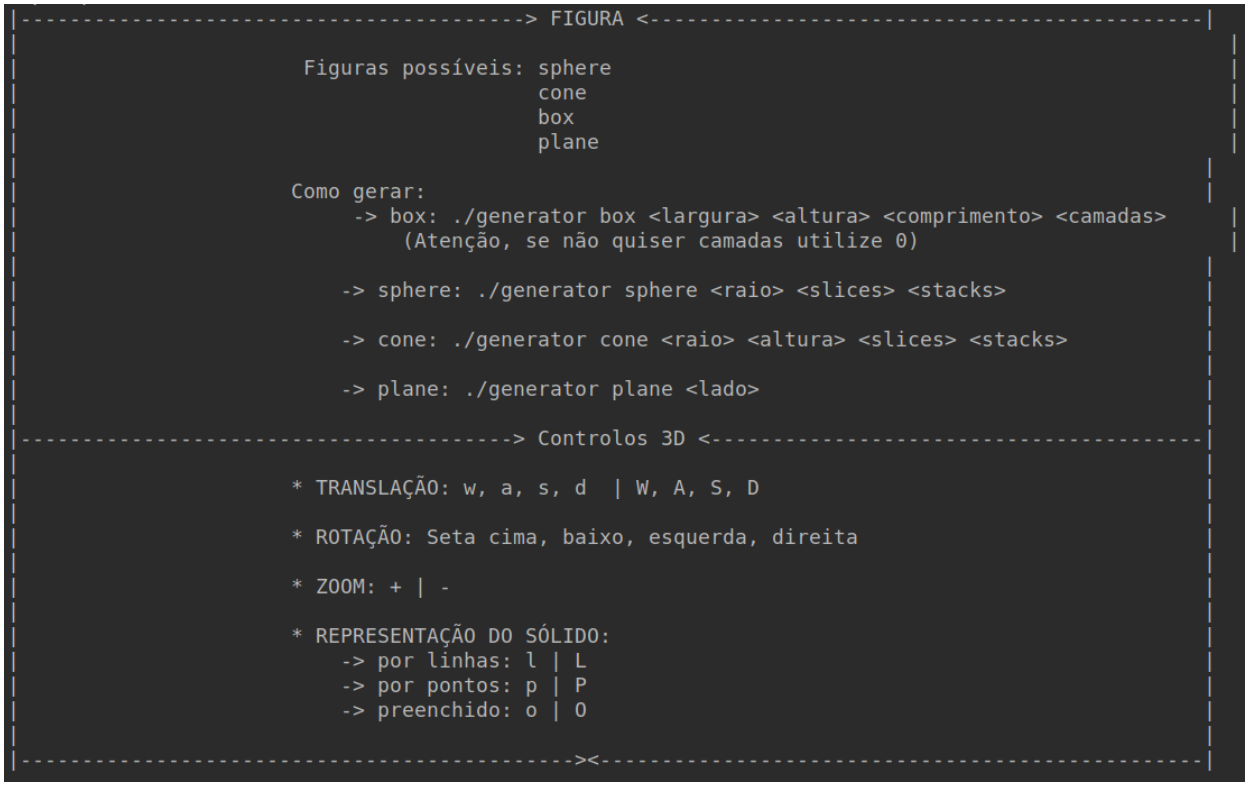


Figura 2.2: Menu de ajuda do Generator

2.1.2 Motor

engine.cpp : Aplicação que possui as funcionalidades principais. Permite a apresentação dos modelos graficamente, através de uma janela.

Assim sendo, como passarão a existir grupos de primitivas com informações associadas, é clara a necessidade de armazenar a informação de maneira diferente, sendo esta renderizada pelo **GLUT** também de uma forma distinta da anterior.

2.2 Classes

Para esta fase, o grupo decidiu criar 6 classes. Surgindo assim uma classe para cada transformação geométrica (**Cor**, **Escala**, **Translacao**, **Rotacao**) que armazenam a informação das mesmas, tendo depois a classe **Transformacao** que irá conter essas classes. Todas elas contidas na directoria *"Estrutura"*.

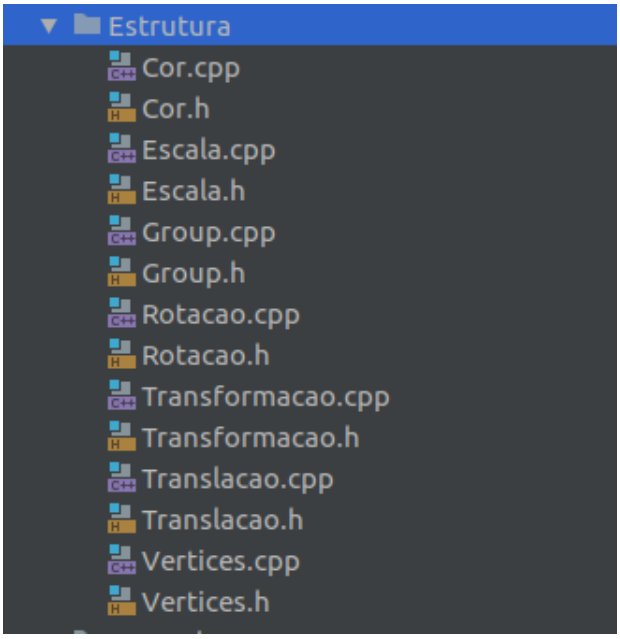


Figura 2.3: Classes contidas em "Estrutura"

```
class Transformacao {

    Translacao* transl;
    Rotacao* rotacao;
    Escala* escala;
    Cor* cor;
```

Figura 2.4: Parte da classe Transformacao

Por último a classe **Group**, que irá armazenar um conjunto de formas, associando-as às respetivas transformações geométricas.

```
class Group {
    Transformacao* tranformacoes;
    std::vector<Vertices*> vertics;
    std::vector<Group*> next;
```

Figura 2.5: Parte da classe Group

2.2.1 Translação

translate.h - Ficheiro que irá indicar os tipo de valores recebidos numa translação.

```
#ifndef GENERATOR_TRANSLATE_H
#define GENERATOR_TRANSLATE_H


struct translations {
    float x;
    float y;
    float z;
};


#endif //GENERATOR_TRANSLATE_H
```

Figura 2.6: Ficheiro translate.h

Classe aplicada sobre a translação, que poderá ser encontrada em **Transformacoes.h**

```
class Translacao : public Transformacoes{

    public:
        Translacao();
        Translacao(float x, float y, float z);
        void aplicaEfeito(float x, float y, float z);
        void aplicaEfeito2 ();


};
```

Figura 2.7: Parte do ficheiro Transformacoes.h

Tendo ainda uma parte mais focada na translação em si no ficheiro transformacao.cpp

```
// TRANSLACAO

Translacao::Translacao() {

}

Translacao::Translacao(float x, float y, float z): Transformacoes(x,y,z) {

}

void Translacao::aplicaEfeito(float x, float y, float z) {
    glTranslatef(x,y,z);
}


void Translacao::aplicaEfeito2() {
    glTranslatef(getX(),getY(),getZ());
    std::cout << getX() << " " << getY() << " " << getZ() << " " << std::endl;
}
```

Figura 2.8: Parte correspondente a translacao, no ficheiro Transformacoes

2.2.2 Rotação

rotation.h - Ficheiro que irá indicar os tipo de valores recebidos numa rotação.

```
#ifndef GENERATOR_ROTATION_H
#define GENERATOR_ROTATION_H

struct rotations {
    float angle;
    float x;
    float y;
    float z;
    bool boolean;
};

#endif //GENERATOR_ROTATION_H
```

Figura 2.9: Ficheiro rotation.h

Classe aplicada sobre a rotação, que poderá ser encontrada em **Transformacoes.h**

```
class Rotacao : public Transformacoes{
    float angle;

    public:
        Rotacao();
        Rotacao(float x, float y, float z, float angulo);
        void insereRotacao(float x, float y, float z, float angulo);
        void aplicaEfeito(float x, float y, float z, float angulo);
        void aplicaEfeito2 ();
};
```

Figura 2.10: Parte do ficheiro Transformacoes.h

Tendo ainda uma parte mais focada na rotação em si no ficheiro transformacao.cpp

```
// ROTACAO

Rotacao::Rotacao() {
}
Rotacao::Rotacao(float x, float y, float z, float angulo) : Transformacoes(x,y,z) {
    angle = angulo;
}

void Rotacao::insereRotacao(float x, float y, float z, float angulo)  {
    x = x;
    y = y;
    z = z;
    angulo = angulo;
}


void Rotacao::aplicaEfeito(float x, float y, float z, float angulo) {
    glRotatef(angulo,x,y,z);
}


void Rotacao::aplicaEfeito2() {
    glRotatef(angle,getX(),getY(),getZ());


}
```

Figura 2.11: Parte correspondente a rotacao, no ficheiro Transformacoes

2.2.3 Cor

Cor.h - Classe que armazena informação relevante para a alteração de cor durante a renderização das figuras pretendidas. Uma vez que a cor é processada segundo o **modelo RGB** são necessárias três variáveis de instancia, uma para cada cor *(Red, Green e Blue)*.

```
#ifndef GENERATOR_COR_H
#define GENERATOR_COR_H


class Cor {
    float rr;
    float gg;
    float bb;

    public:
        Cor();
        Cor(float r1, float g1, float b1);
        void insereR1(float r1);
        void insereG1(float g1);
        void insereB1(float b1);
        float getR1();
        float getG1();
        float getB1();
};


#endif //GENERATOR_COR_H
```

Figura 2.12: Ficheiro Cor.h

Classe aplicada sobre a cor, que poderá ser encontrada em **Cor.cpp**

```
#include "Cor.h"

Cor::Cor() {

}

Cor::Cor(float r1, float g1, float b1) {
    rr = r1;
    gg = g1;
    bb = b1;
}


void Cor::insereR1(float r1) {
    rr = r1;
}
void Cor::insereG1(float g1) {
    gg = g1 ;
}
void Cor::insereB1(float b1) {
    bb = b1;
}



float Cor::getR1() {
    return rr;
}
float Cor::getG1() {
    return gg;
}
float Cor::getB1() {
    return bb;
}
```

Figura 2.13: Ficheiro Cor.cpp

2.2.4 Escala

scale.h - Ficheiro que irá indicar os tipo de valores recebidos numa escala.

```
#ifndef GENERATOR_SCALE_H
#define GENERATOR_SCALE_H


struct scales {
    float x;
    float y;
    float z;
};

#endif //GENERATOR_SCALE_H
```

Figura 2.14: Ficheiro rotation.h

Classe aplicada sobre a escala, que poderá ser encontrada em **Transformacoes.h**

```
class Escala : public Transformacoes {
    public:
        Escala();
        Escala(float x, float y, float z);
        void aplicaEfeito (float x, float y, float z);
        void aplicaEfeito2 ();
};
```

Figura 2.15: Parte do ficheiro Transformacoes.h

Tendo ainda uma parte mais focada na escala em si no ficheiro **transformacao.cpp**

```
// ESCALA
Escala::Escala() {

}

Escala::Escala(float x, float y, float z) : Transformacoes(x,y,z) {

}

void Escala::aplicaEfeito (float x, float y, float z) {
    glScalef(x,y,z);
}

void Escala::aplicaEfeito2() {
    glScalef(getX(),getY(),getZ());
}
```

Figura 2.16: Parte correspondente a escala, no ficheiro Transformacoes

2.2.5 Transformação

Transformacoes.cpp - Ficheiro que irá conter todas as aplicações de diversas transformações geométricas.

Figura 2.17: Partes das transformações geométricas que serão aplicadas as figuras

2.2.6 Group

Consiste na divisão de um ficheiro **XML** em grupos, para posteriormente a cada grupo ser aplicado transformações geométricas tais como *scale, translate* e *rotation.*

```
#include "Transformacao.h"
#include "Vertices.h"
#include <iostream>
#include <vector>

#ifndef GENERATOR_GROUP_H
#define GENERATOR_GROUP_H


class Group {
    Transformacao* tranformacoes;
    std::vector<Vertices*> vertics;
    std::vector<Group*> next;


    public:
        Group();
        Group(Transformacao* t, std::vector<Vertices*> vert, std::vector<Group*> n);
        void insereTransformacoes(Transformacao* t);
        void insereVerts(std::vector<Vertices*> vert);
        void insereNext(std::vector<Group*> n);
        Transformacao* getTransformacoes();
        std::vector<Vertices*> getVertices();
        std::vector<Group*> getGroups();
};


#endif //GENERATOR_GROUP_H
```

Figura 2.18: Ficheiro group.h

```
Group::Group() { ... }

Group::Group(Transformacao* t, std::vector<Vertices*> vert, std::vector<Group*> n) {
    tranformacoes=t;
    vertics=vert;
    next=n;
}

void Group::insereTransformacoes(Transformacao* t) {
    tranformacoes = t;
}

void Group::insereVerts(std::vector<Vertices*> vert) {

    vertics = vert;
}

void Group::insereNext(std::vector<Group*> n) {
    next = n;
}

Transformacao* Group::getTransformacoes() {
    return tranformacoes;
}

std::vector<Vertices*> Group::getVertices() {
    return vertics;
}

std::vector<Group*> Group::getGroups() {
    return next;
}
```

Figura 2.19: Ficheiro group.h

2.3 Parsing

De modo a explorar o conteúdo dos ficheiros XML, foi utilizado um *parser* como o tinyxml2. Foi escolhido este *parser* pois é de facil utilização com APIs semelhantes, menos alocação de memória e possui um rápida leitura de ficheiros.

Poderá ser encontrado no ficheiro **tinyxml2.cpp**, na diretoria *engine* .

```
// Função que recebe um string que é o caminho e vai ler o ficheiro e a medida que vai lendo o ficheiro vai meter na estrutura as coordenadas
std::vector<Vertices*> lerFicheiro(std::string caminho) {
    std::vector<Vertices*> verts;
    std::ifstream ficheiro(caminho);
    std::string linha;


    if(ficheiro.fail()) {
        std::cout << "Bip bip! não consegui encontrar o ficheiro 3D!"<< std::endl;
    }
    else {
        while(getline(ficheiro,linha)) {
            size_t pos;
            Vertices* vertice = new Vertices();


            vertice->insereX(std::stof(linha,&pos));

            linha.erase(0,pos+1);
            vertice->insereY(std::stof(linha,&pos));

            linha.erase(0,pos+1);
            vertice->insereZ(std::stof(linha,&pos));

            verts.push_back(vertice);
        }
    }

    return verts;
}
```

Figura 2.20: Extrato do ficheiro Parser.cpp

Capítulo 3

Primitiva Geométrica - Torus

Um *Torus* é um sólido geométrico que apresenta o formato aproximado de uma camara de pneu. Em geometria, pode ser definido como o lugar geométrico tridimensional formado pela rotação de uma superfície circular plana de raio interior, em torno de uma circunferência de raio exterior. Como tal, os parâmetros para gerar um *Torus* são **radius** (raio interior), **distance** (raio exterior), **stacks** (número de divisões radiais) e **slices** (número de lados por cada secção radial).

3.1 Algoritmo

Para a construção do *Torus* é preciso considerar que a sua constituição se baseia nos raios interior e exterior. Para tal, é preciso definir que eixos vão ficar responsáveis para definir as circunferências que vamos percorrer para poder desenhar o *Torus*. Com isto, os eixos Y e X definem uma circunferência com o raio exterior **distance**, e os eixos X, Y e Z definem uma circunferência com o raio interior **radius**.

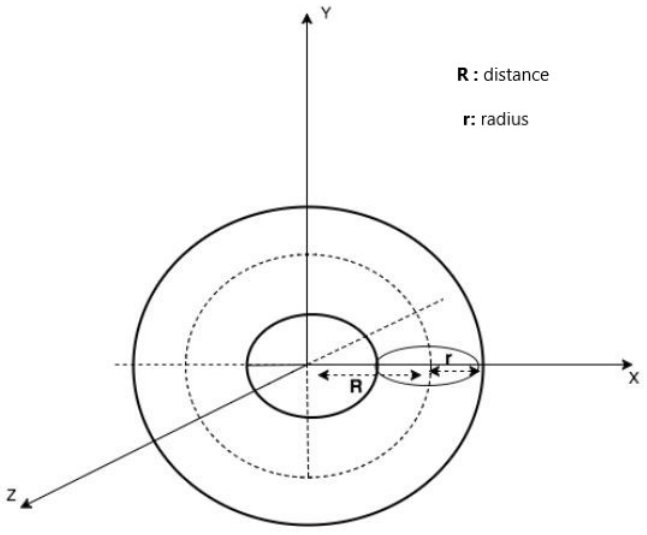


Figura 3.1: Vista de cima da geração de um Torus

Com base neste pensamento, chegamos ao seguinte algoritmo.

```
for (int i=0;i< slices; i++) {
    for(int j=0; j<stacks; j++) {


        coordenadas1.x = cos(theta)*(distance + radius * cos(phi));
        coordenadas1.y= sin(theta)*(distance + radius * cos(phi));
        coordenadas1.z= radius*sin(phi);
        result.push_back(coordenadas1);



        coordenadas2.x = cos(theta + desl1)*(distance + radius * cos(phi));
        coordenadas2.y = sin(theta + desl1)*(distance + radius * cos(phi));
        coordenadas2.z = radius*sin(phi);
        result.push_back(coordenadas2);


        coordenadas3.x = cos(theta + desl1)*(distance + radius * cos(phi + desl2));
        coordenadas3.y = sin(theta+desl1)*(distance + radius * cos(phi+desl2));
        coordenadas3.z = radius*sin(phi+desl2);
        result.push_back(coordenadas3);


        coordenadas4.x = cos(theta + desl1)*(distance + radius * cos(phi + desl2));
        coordenadas4.y = sin(theta+desl1)*(distance + radius * cos(phi+desl2));
        coordenadas4.z = radius*sin(phi+desl2);
        result.push_back(coordenadas4);

        coordenadas5.x =  cos(theta)*(distance + radius * cos(phi + desl2));
        coordenadas5.y =  sin(theta)*(distance + radius * cos(phi + desl2));
        coordenadas5.z =  radius*sin(phi + desl2);
        result.push_back(coordenadas5);

        coordenadas6.x =  cos(theta)*(distance + radius * cos(phi));
        coordenadas6.y=   sin(theta)*(distance + radius * cos(phi));
        coordenadas6.z =  radius*sin(phi);
        result.push_back(coordenadas6);
        phi = desl2 * (j + 1);
    }
    theta = desl1 * (i + 1);
}
```

Figura 3.2: Algoritmo de geração de um Torus

Desta forma, com auxílio das funções cos e sin, podemos obter facilmente os pontos que formem as circunferências.

Desta forma, passamos a desenhar entre estes dois limitadores um anel, como representado na figura acima. Para tal, do mesmo modo que percorremos a circunferência externa, percorremos a circunferência interna, isto é, adicionamos theta em cada interação, para dar a volta à circunferência interna

Com isto, as iterações baseiam-se em cada anel (conjunto de 2 limitadores), iterar (adicionar theta), e aplicar o mesmo processo de desenho. Terminando a circunferência interna, itera-se a amplitude da circunferência externa (adicionar phi), e repetir o mesmo processo até completar a circunferência externa toda.

Capítulo 4

Análise de Resultados - Sistema Solar

O resultado final não correspondeu ao esperado, apesar de cumprir com os requisitos mínimos. Contudo o grupo tentou aproximar o mais possível este projeto com um sistema solar real. Utilizando cores, escalas próximas das reais e criação de satélites naturais (mais conhecido por *"luas"*). Ainda assim a aproximação com um sistema solar poderia ser maior, atraves de orbitas e da aplicação de uma inclinação aos aneis de Saturno, isto tudo seria possível com a utilização de um "Torus"ao qual encontramos diversas dificuldades na sua concepção.

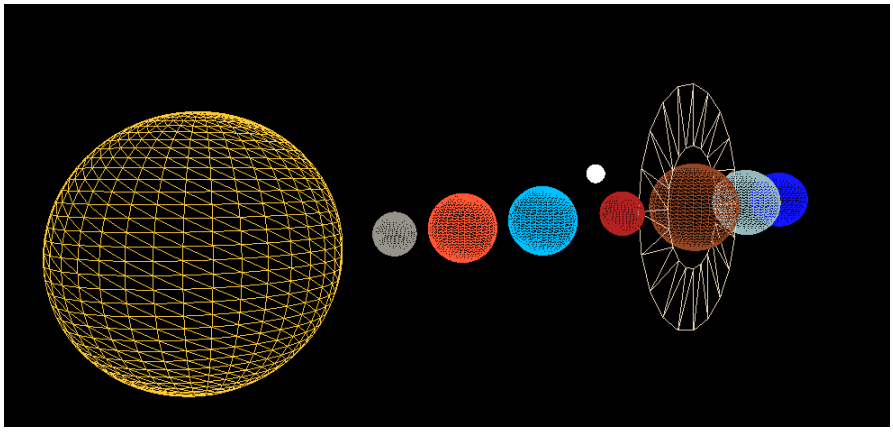


Figura 4.1: Sistema solar demonstrado em linhas

Nesta representação verificamos, os planetas gerados atraves de pontos

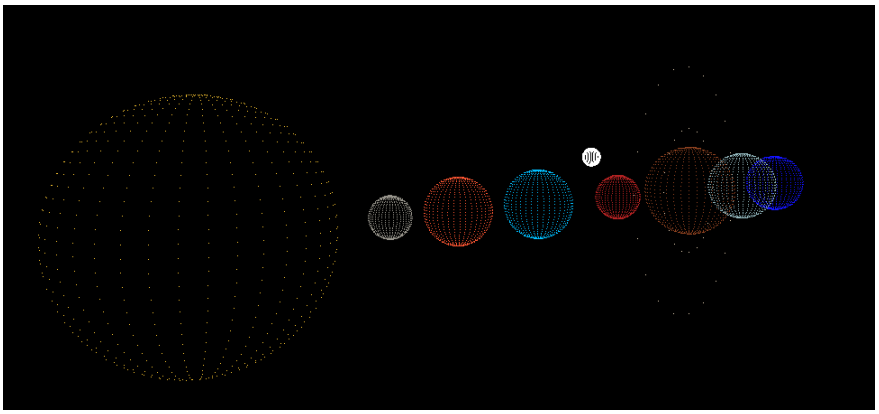


Figura 4.2: Sistema solar demonstrado em pontos

Por fim visualiza-mos o sistema solar colorido, com as cores que achamos apropriadas para cada planeta

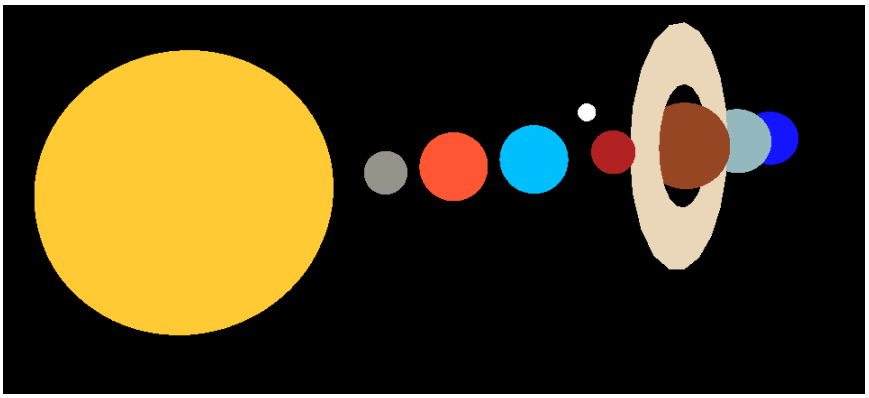


Figura 4.3: Sistema solar demonstrado totalmente pintado

Capítulo 5

Conclusão/Trabalho futuro

O desenvolvimento desta segunda fase do trabalho, foi um pouco mais trabalhosa e demorada em relação à primeira fase. Isto deve-se ao facto da alteração da estrutura aplicada aos ficheiros **XML**, consequentemente alterações no *parse* destas informações.

Ainda assim, é de realçar os conhecimentos adquiridos na 1ª fase que nos ajudaram na resolução de diversos problemas que surgirão ao longo do tempo.

Pensamos que o resultado final desta fase corresponde às expectativas, na medida em que o Sistema Solar pretendido para esta fase se enquadra perfeitamente naquilo que realmente era esperado.

No entanto, esperamos que nas restantes fases do projeto consigamos aperfeiçoar cada vez mais o modelo em questão, de forma a torna-lo mais realista e agradável possível.