Universidade do Minho
Escola de Engenharia

Computação Gráfica

Trabalho Prático - Fase 2

4 de Abril de 2021

Grupo 23

Carlos Filipe Coelho Ferreira, A89542
Joel Salgueiro Martins, A89575
José Carlos Leite Magalhães, A85852
Paulo Ricardo Antunes Pereira, A86475




Conteúdo

1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi proposto desenvolver um motor 3D, baseado num cenário gráfico, cujo propósito passa por explorar todas as suas potencialidades e capacidades. Esta exploração é feita através de diversos exemplos visuais e interativos, sendo utilizadas, para esse efeito, todas as ferramentas abordadas nas aulas práticas da disciplina.

O projeto encontra-se dividido em quatro fases, sendo que este relatório, estando a primeira já concluída, versará sobre a segunda etapa. Esta fase consiste em criar cenários gráficos hierárquicos usando, para isso, transformações geométricas tais como rotação, translação e escala.

As primitivas gráficas já implementadas na fase anterior serão alteradas e manipuladas com o objetivo de, a partir de um ficheiro XML, construir um modelo estático do Sistema Solar, composto pelo sol, planetas e respetivas luas.

2 Transformações geométricas

Em primeiro lugar decidiu-se criar uma estrutura que representasse cada modelo no sistema. Esta foi pensada de modo a que seja facilmente manipulada, sendo declarada da seguinte forma:

```
class Modelo {
        vector<Ponto> pontos;
        string nome;
    //(...)
};
```

Posteriormente, e de forma a satisfazer as três transformações (translação, rotação e escala), optou-se por implementar uma classe ***Instrucao***, cujas variáveis tomarão diferentes valores consoante o tipo de transformação, como veremos de seguida.

```
class Instrucao {
        string nome;
        float x;
        float y;
        float z;
        float angle;
        Modelo* model;
    //(...)
};
```

2.1 Translação

Uma translação desloca um ponto, ou um conjunto de pontos, numa determinada direção e comprimento. É uma movimentação em linha reta para um determinado ponto num dado referencial, partindo das suas coordenadas originais, em que não há alteração da figura, tanto nas dimensões como na forma.

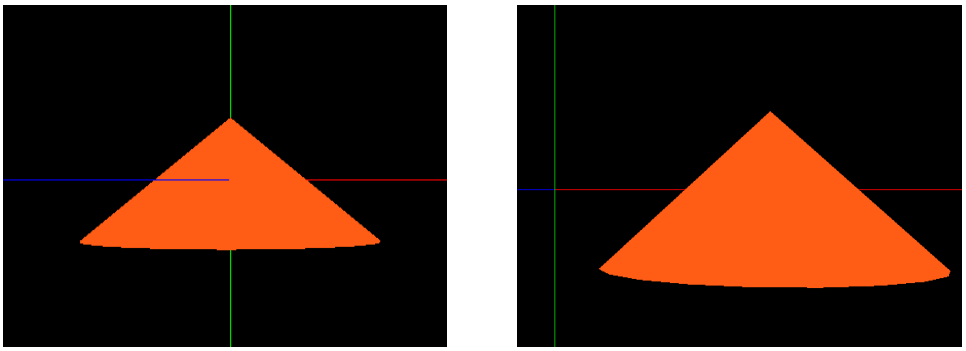
A implementação desta transformação tem como base a estrutura definida anteriormente, pelo que os seus campos são atualizados da seguinte forma:

```
class Instrucao {
//(...)
public:
    void translate(float x, float y, float z) {
                this->nome = "Translate";
                this->x = x;
                this->y = y;
                this->z = z;
        }
//(...)
};
```

Para aplicar a transformação em questão é necessário recorrer à biblioteca *OpenGL*. A primitiva utilizada é invocada quando o nome da instrução é *Translate*, como é possível verificar no seguinte excerto de pseudo-código:

```
class Instrucao {
    string nome;
//(...)
public:
    void apply() {
         //(...)
            else if (nome == "Translate")
                glTranslatef(this->x, this->y, this->z);
         //(...)
        }
};
```

É possível observar um caso prático do processo descrito anteriormente:



(a) Objeto original (b) Objeto após translação

Figura 1: **translate x="20"y="0"z="0"**

2.2 Rotação

A rotação de uma figura implica ter bem definido o ângulo de rotação e as coordenadas do vetor, pois este último funciona como eixo do movimento, preservando sempre as dimensões da figura.

A implementação desta transformação é feita de forma semelhante ao descrito em 2.1, diferindo na variável *angle*, responsável por armazenar o ângulo de rotação:

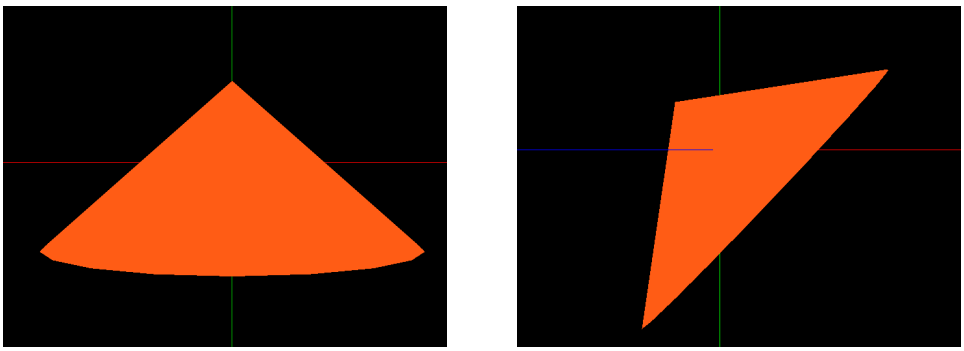
```
class Instrucao {
//(...)
public:
    //(...)
    void rotate(float x, float y, float z,float angle) {
                this->nome = "Rotate";
```

```
                this->x = x;
                this->y = y;
                this->z = z;
                this->angle = angle;
        }
//(...)
};
```

O processo de aplicação é análogo ao da translação, pelo que difere, naturalmente, na condição onde a primitiva correspondente é invocada, dentro da já apresentada função *apply*:

```
class Instrucao {
    string nome;
//(...)
public:
    void apply() {
        //(...)
        else if (nome == "Rotate")
          glRotatef(this->angle, this->x, this->y, this->z);
     //(...)
        }
};
```

É possível observar um caso prático do processo descrito anteriormente:



(a) Objeto original (b) Objeto após rotação

Figura 2: **rotate angle=-45"x="0"y="0"z="1"**

2.3 Escala

Escalar uma figura consiste no processo de encolher ou expandir a mesma, pelo que altera a sua forma. Há, para esse efeito, uma multiplicação das coordenadas da figura pelo fator de escala correspondente.

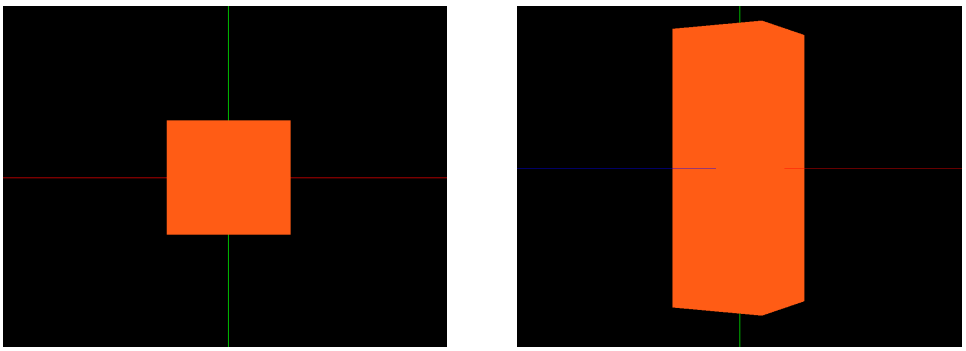
A implementação desta transformação é feita de forma simples, pelo que as componentes **x**, **y** e **z**, apresentadas de seguida, correspondem aos fatores de escala fornecidos pelo utilizador:

```
class Instrucao {
//(...)
public:
    //(...)
    void scale(float x, float y, float z) {
                this->nome = "Scale";
                this->x = x;
                this->y = y;
                this->z = z;
        }
        }
//(...)
};
```

De forma semelhante a 2.1 e 2.2, e tendo em conta que a primitiva correspondente à transformação é *glScalef*, esta é aplicada na condição em que o nome da instrução a tratar é *Scale*:

```
class Instrucao {
    string nome;
//(...)
public:
    void apply() {
            if (nome == "Scale")
                  glScalef(this->x, this->y, this->z);
     //(...)
        }
};
```

É possível observar um caso prático do processo descrito anteriormente:



(a) Objeto original (b) Objeto após escala

Figura 3: **scale x="1"y="5"z="1"**

3 Motor

3.1 Interpretação de ficheiros XML

Uma das principais diferenças entre a primeira fase e a que nos encontramos são os ficheiros XML, sendo que nesta etapa o processo desde a sua escrita até à sua interpretação tornou-se bastante mais complexo.

A título de exemplo, vejamos a estrutura de um ficheiro XML válido:

```
<scene>
  <group>
    <!--Sol-->
    <scale x="5" y="5" z="5"/>
    <colour r="255" g="92" b="21" />
    <models>
      <model file="planet.3d"/>
    </models>
  </group>
  <!--Mercurio-->
  <group>
    <translate x="0" y="0" z="7.74"/>
    <scale x="0.009" y="0.009" z="0.009"/>
    <colour r="101" g="75" b="22" />
    <models>
      <model file="planet.3d" />
    </models>
  </group>
</scene>
```

É considerado válido pois respeita as condições exigidas no enunciado, como por exemplo a regra de que cada **<group>**, no seu conjunto de transformações, apenas pode ter, no máximo, uma transformação de cada tipo (rotação, translação e escala).

Realça-se também que cada **<group>** pode ter vários sub-grupos, que, aos olhos do interpretador, se comportarão também como **<group>**.

Tendo em conta toda esta nova informação, foi necessário reformular todo o processo de leitura e *parsing* referente a este tipo de ficheiros. Para isso foi criada uma nova classe, ***leitorXML***, definida pelo seguinte:

```
class leitorXML {
        string ficheiro;
        Transformacao* transf;
    //(...)
};
```

A segunda componente da estrutura introduz uma nova classe, criada também nesta fase, com o objetivo tratar a informação presente nos XML, contendo na sua definição dois vetores de classes já vistas neste relatório:

```
class Transformacao {
        vector<Instrucao*> instrucoes;
        vector<Modelo*> models;
    //(...)
};
```

O processo de leitura e interpretação é feito utilizando várias funções, auxiliadas pela biblioteca *tinyxml*, já usada na fase 1. Entre elas, a função *parseGroup(...)*, responsável por verificar que tipo de elemento está a ser lido, bem como proceder ao processamento adequado. Estes elementos, por sua vez, podem ser:

- **translate**, **rotate** ou **scale**. O método de adição é análogo às três transformações, pelo que se verá de seguida o pseudo-código correspondente à translação:

```
class leitorXML {
    Transformacao* transf;
    //(...)
public:
 void parseGrupo(XMLElement* group) {
  transf->addPush(); //Falaremos no proximo topico
  for(XMLElement* elem = group->FirstChildElement(); elem;
    elem = elem->NextSiblingElement()){

   string type = elem->Value();
   if (type.compare("translate") == 0) {
          transf->addTranslate(elem->FloatAttribute("x"),
```

```
            elem->FloatAttribute("y"),
            elem->FloatAttribute("z"));
                  }
    //(...)
    }
  transf->addPop(); //Falaremos no proximo topico
};
```

Após recolhidos os dados, então estes são adicionados na estrutura:

```
class Transformacao {
   vector<Instrucao*> instrucoes;
//(...)
public:
   void addTranslate(float x, float y, float z) {
                  Instrucao* i =new Instrucao();
                  i->translate(x, y, z);
                  instrucoes.push_back(i);
         }
//(...)
};
```

- No caso do elemento ser **colour**, então, em primeiro lugar, é feita a divisão de todos os componentes por 255, devido à obrigatoriedade do *OpenGl* usar o sistema RGB. Posteriormente o elemento é adicionado à estrutura *instrucoes*, seguindo o mesmo método do tópico anterior;

- **models**: é feita a procura por subelementos do tipo **model**, de modo a saber quais as figuras a adicionar na estrutura ***Transformacao***;

- **model**: é utilizada a função *addModel(...)* para adicionar este elemento às estruturas presentes na classe ***Transformacao***, tendo duas hipóteses:

  - se o modelo em questão já se encontrar no objeto ***Model***, então é apenas adicionado ao objeto ***Instrucao***, e consequentemente ao vetor *instrucoes*;

  - caso contrário, além de adicionado ao objeto ***Instrucao***, é também inserido no objeto ***Model***. Os vetores *instrucoes* e *models*, da classe ***Transformacao***, são devidamente atualizados.

- **group:** é aplicada a recursividade. O ciclo *for* descrito na função faz com que todos os subgrupos do "pai"sejam visitados.

3.2 Tratamento dos dados

Os ficheiros contendo todas as coordenadas das várias figuras são construídos pelo Gerador, como já executado na fase 1. O conteúdo destes tem que ser interpretado e introduzido no sistema sendo que, para isso, definiu-se a estrutura *Ponto*:

```
struct Ponto { float x; float y; float z; };
```

De modo a facilitar a manipulação das várias figuras, decidiu-se então criar uma classe ***Modelo***. Como uma figura é um conjunto de pontos, optou-se por definir da seguinte forma:

```
class Modelo {
        vector<Ponto> pontos;
        string nome;
        //(...)
};
```

Para desesenhar as diversas figuras no ecrã tiramos partido da primitiva *glVertex3f*. A função que a invoca é a *desenhaPontos()*, em que a cada iteração são desenhados dois triângulos:

```
class Modelo {
    //(...)
public:
 void desenhaPontos() {
                Ponto p1, p2, p3;
                int size = pontos.size();
                glBegin(GL_TRIANGLES);
                for (int i = 0; i < size; i += 6) {
                        p1 = pontos.at(i);
                        p2 = pontos.at(i + 1);
                        p3 = pontos.at(i + 2);
                        glVertex3f(p1.x, p1.y, p1.z);
                        glVertex3f(p2.x, p2.y, p2.z);
                        glVertex3f(p3.x, p3.y, p3.z);
                        p1 = pontos.at(i + 3);
                        p2 = pontos.at(i + 4);
                        p3 = pontos.at(i + 5);
                        glVertex3f(p1.x, p1.y, p1.z);
                        glVertex3f(p2.x, p2.y, p2.z);
                        glVertex3f(p3.x, p3.y, p3.z);
                } glEnd();
        }
    //(...)
};
```

3.3 Hierarquia das transformações

À medida que se ia desenvolvendo a solução, cedo se reconheceu que a hierarquia das transformações teria de ser devidamente implementada. É necessário garantir não só que as transformações presentes num determinado grupo são aplicadas em todos os seus subgrupos, mas também que o contrário não aconteça.

A solução passou por tirarmos proveito das funcionalidades da biblioteca *OpenGl*, mais concretamente do sistema de *stack* de matrizes. Em primeiro lugar, no processo de interpretação do ficheiro XML, antes de cada iteração é feito o *addPush()* e no fim da mesma é feito o *addPop()*.

```
class leitorXML {
    Transformacao* transf;
    //(...)
public:
 void parseGrupo(XMLElement* group) {
  transf->addPush(); //Antes de cada iteracao e dado push
  for(XMLElement* elem = group->FirstChildElement(); elem; elem
      = elem->NextSiblingElement())
    //(...)
    }
  transf->addPop(); //No fim de cada iteracao e dado pop
};
```

Com este sistema, consegue-se que as transformações feitas num **group** "pai"sejam igualmente aplicadas no subgrupo "filho", mas as feitas neste último não afetem o **group** superior ("pai") ou outros **group** equivalentes ("irmãos"). O estilo das funções de adição ao objeto **Transformacao** é semelhante ao já apresentado, como mostrado de seguida:

```
class Transformacao {
   vector<Instrucao*> instrucoes;
//(...)
public:
   void addPop() {
                Instrucao* i = new Instrucao();
                i->popMatrix();
                instrucoes.push_back(i);
        }
   void addPush() {
                Instrucao* i = new Instrucao();
                i->pushMatrix();
                instrucoes.push_back(i);
        }
//(...)
};
```

Por último é necessário aplicar as primitivas *glPushMatrix()* e *glPopMatrix()* para as operações de *push* e *pop*, respetivamente. É feito de forma semelhante ao apresentado na secção 2:

```
class Instrucao {
    string nome;
//(...)
public:
    void apply() {
    //(...)
             else if (nome == "PopMatrix")  glPopMatrix();
             else if (nome == "PushMatrix") glPushMatrix();
         }
};
```

4 Gerador

O gerador desta fase sofreu uma pequena adição ao já implementado na primeira etapa, pois, tendo em conta o objetivo da construção do Sistema Solar, foi necessário criar uma nova figura de forma a poder representar os anéis de Saturno.

4.1 Anéis de Saturno

A estratégia usada baseia-se sobretudo no algoritmo da construção de uma *torus*, através de múltiplos triângulos. Vejamos de seguida os esboços produzidos pelo grupo, bem como as deduções matemáticas das fórmulas utilizadas no programa.

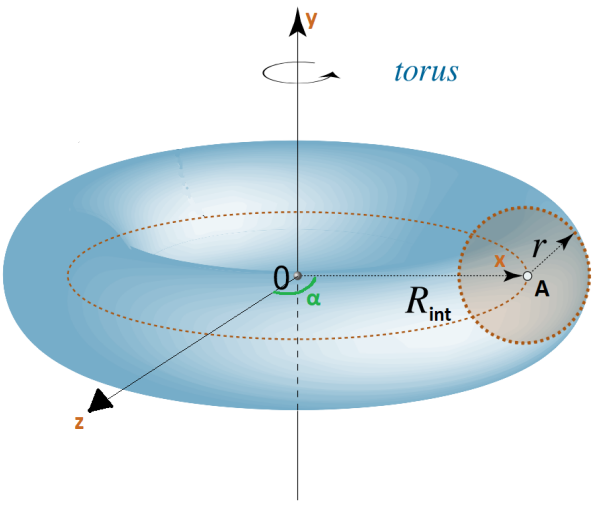


Figura 4: Esboço trigonométrico relativo à *torus* - 1

O ponto **A**, representado na figura, terá as coordenadas genéricas:

$$(x_A, y_A, z_A) = \begin{cases} x_A = r_{int} * sen(\alpha) \\ y_A = 0 \\ z_A = r_{int} * cos(\alpha) \end{cases}$$

Analisando o círculo a tracejado, onde se encontra o ponto **A**, com mais atenção, temos o seguinte:

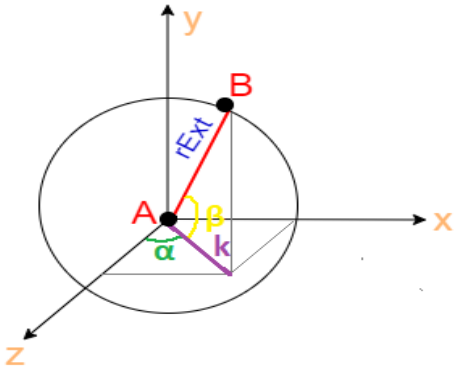


Figura 5: Esboço trigonométrico relativo à *torus* - 2

As coordenadas do ponto B, apresentado na figura, podem ser calculadas através do seguinte:

$$(x_B, y_B, z_B) = \begin{cases} x_B = k * sen(\alpha) \\ y_B = r_{Ext} * sen(\beta) \\ z_B = k * cos(\beta) \end{cases}$$

Como $k = r_{Ext} * cos(\beta)$ então teremos que as coordenadas do ponto B são dadas pelo seguinte:

$$(x_B, y_B, z_B) = \begin{cases} x_B = r_{Ext} * cos(\beta) * sen(\alpha) \\ y_B = r_{Ext} * sen(\beta) \\ z_B = r_{Ext} * cos(\beta) * cos(\alpha) \end{cases}$$

Ora, como neste caso a origem dos eixos é o ponto A, então temos:

$$\begin{cases} x = r_{Int} * sen(\alpha) + r_{Ext} * cos(\beta) * sen(\alpha) \\ y = 0 + r_{Ext} * sen(\beta) \\ z = r_{Int} * cos(\alpha) + r_{Ext} * cos(\beta) * cos(\alpha) \end{cases}$$

Como $A_x = r_{Int} * sen(\alpha)$, $A_y = 0$ e $A_z = r_{Int} * cos(\alpha)$, conclui-se que:

$$\begin{cases} x = (r_{Int} + r_{Ext} * cos(\beta)) * sen(\alpha) \\ y = r_{Ext} * sen(\beta) \\ z = (r_{Int} + r_{Ext} * cos(\beta)) * cos(\alpha) \end{cases}$$

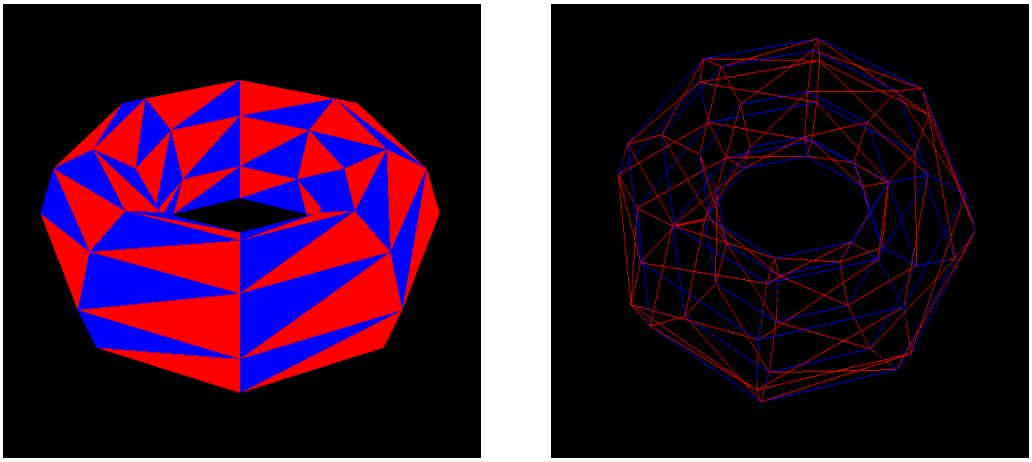
De seguida podemos ver a implementação das fórmulas mostradas anteriormente no código do nosso programa. Realça-se que são efetuados dois ciclos: o exterior, responsável por percorrer os anéis, e o interior, responsável pelas faces. Em cada iteração destes, o objetivo passa por obter o trapézio pretendido, bem como os dois triângulos que o compõe.

Note-se que, tendo em conta o esboço produzido, o nome dos ângulos foi adaptado no código com as seguintes correspondências: $\alpha = alfaExterno$ e $\beta = alfaInterno$.

```
void desenhaTorus(float rInt, float Exte, int faces, int aneis,
    char *ficheiro) {
     //(...)
  float aflaInt = (2 * M_PI) / faces;
  float alfExte = (2 * M_PI) / aneis;
  float alfaIntAnt, alfaExtAnt, alfaIntAtual=alfaExtAtual = 0;

  for (int i = 0; i < aneis; i++) {
    alfaExtAnt = alfaExtAtual;
    alfaExtAtual += alfExte;

    for (int j = 0; j < faces; j++) {
     alfaIntAnt = alfaIntAtual;
     alfaIntAtual += aflaInt;

     x1 = (rInt + Exte * cos(alfaIntAnt)) * sin(alfaExtAnt);
     y1 =  Exte * sin(alfaIntAnt);
     z1 = (rInt + Exte * cos(alfaIntAnt)) * cos(alfaExtAnt);

     x2 = (rInt + Exte * cos(alfaIntAtual)) * sin(alfaExtAnt);
     y2 =  Exte * sin(alfaIntAtual);
     z2 = (rInt + Exte * cos(alfaIntAtual)) * cos(alfaExtAnt);

     x3 = (rInt + Exte * cos(alfaIntAnt)) * sin(alfaExtAtual);
     y3 =  Exte * sin(alfaIntAnt);
     z3 = (rInt + Exte * cos(alfaIntAnt)) * cos(alfaExtAtual);

     x4 = (rInt + Exte * cos(alfIntAtual)) * sin(alfaExtAtual);
     y4 =  Exte * sin(alfaIntAtual);
     z4 = (rInt + Exte * cos(alfIntAtual)) * cos(alfaExtAtual);
    //(...)
}
```

Por fim, e de forma a demonstrar todo o processo, apresenta-se um exemplo de um *torus*, em duas diferentes opções de visualização, gerado com o comando:

```
./generator anel <raioInt> <raioExt> <faces> <aneis> <ficheiro>
```



(a) *GL Fill* (b) *GL Line*

Figura 6: *Torus*

5 Execução

5.1 Interação com o utilizador

De modo a tornar a nossa aplicação mais flexível, optou-se por adicionar alguma interação ao utilizador:

- Selecionando o botão direito do rato, é apresentado um menu onde é possível ver as mesmas figuras preenchidas (*GL_FILL*), que é a vista padrão, compostas por pontos (*GL_POINT*) e, de modo a, por exemplo, facilitar a visualização das figuras geométricas que a compõe, também é possível ver a mesma através de linhas (*GL_LINE*);
- A câmara utilizada é manipulável, pelo que através das teclas **W**, **A**, **S**, **D** é possível alterar a posição da câmara, e através das habituais **UP**, **DOWN**, **LEFT** e **RIGHT** é permitido ao utilizador alterar a direção da mesma.

5.2 Resultado final

As várias figuras apresentadas nesta subsecção foram desenhadas a partir de dois ficheiros diferentes, ambos gerados pelo grupo. Acrescenta-se que, para uma melhor legibilidade, optou-se por não aplicar a escala real, pois estaríamos a lidar com diferenças significativas de dimensões entre, por exemplo, o sol e qualquer outro planeta.

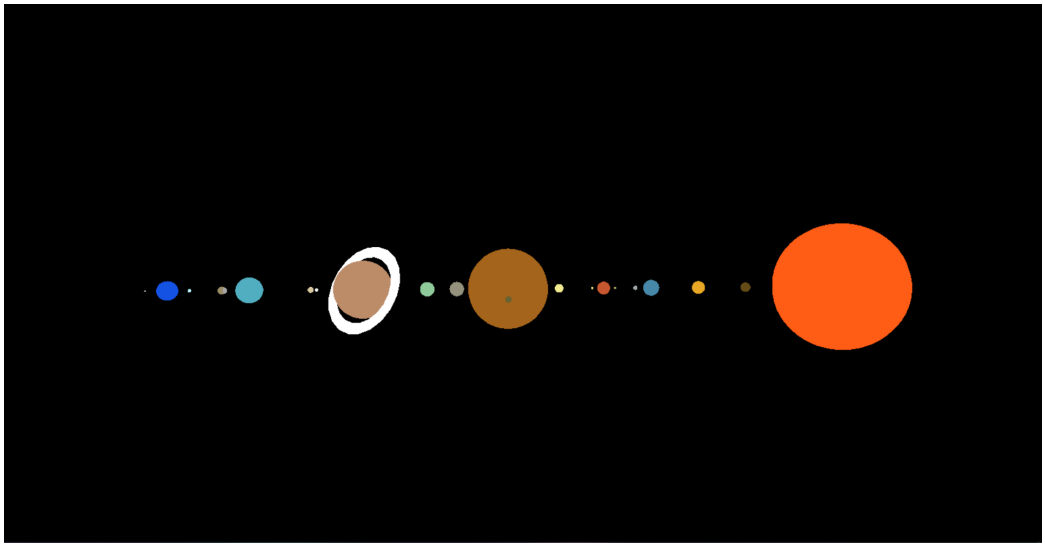


Figura 7: **sistema_linear.xml** - *GL Fill*

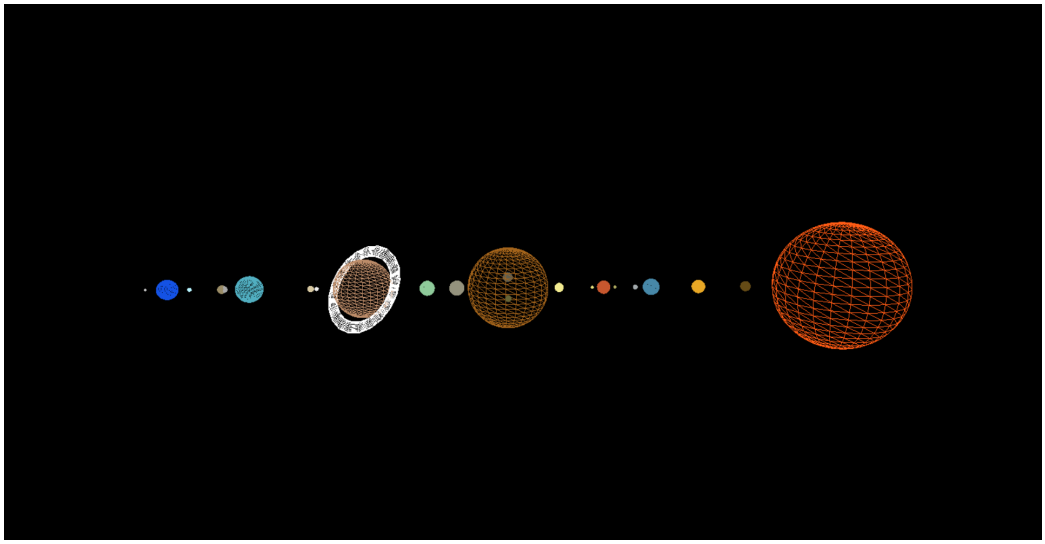


Figura 8: **sistema_linear.xml** - *GL Line*

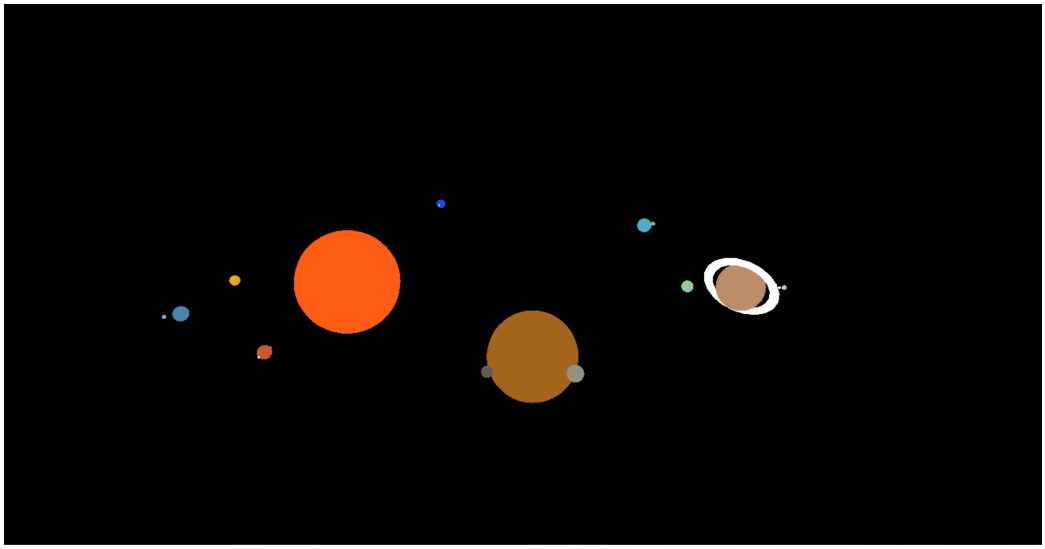


Figura 9: **sistema_disperso.xml** - *GL Fill*

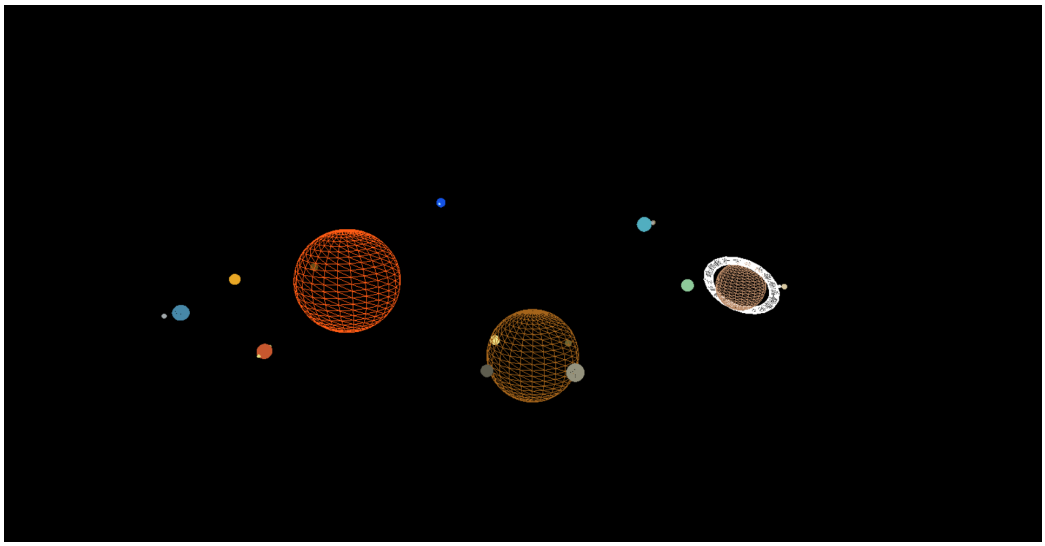


Figura 10: **sistema_disperso.xml** - *GL Line*

6 Conclusão

Quanto à parte pedagógica, concluímos que este projeto foi muito enriquecedor para o nosso coletivo, pois este trabalho permitiu que melhorássemos as nossas competências a nível prático relacionadas com a unidade curricular de Computação Gráfica.

O grupo considera bastante produtiva a exploração das ferramentas da disciplina, como o *OpenGL* e as suas primitivas, bem como a "manipulação" e cimentação de conceitos como as várias transformações geométricas existentes (translação, rotação e escala).

Um dos desafios foi não só o aprofundamento constante da linguagem C++, à medida que íamos avançando no projeto, mas também lidar com a interpretação e tratamento de ficheiros XML, sendo que estes se tornaram mais complexos relativamente aos utilizados na primeira fase.

Consideramos, no entanto, que todos os objetivos delineados para esta fase foram cumpridos, traduzindo-se no aprofundamento de mais uma componente imprescindível de Computação Gráfica.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim, uma janela aberta e curiosa para as próximas fases.