Computação Gráfica

Trabalho Prático (Parte 2)

Luís Miguel Ramos (A83930)
Válter Carvalho (A84464)

29 de Março de 2020

Grupo nº 26
Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

Concluída a primeira fase deste projeto, demos inicio à segunda fase.

Esta tem como objetivo fazer alterações nas primitivas gráficas através de tranformações geométricas, tais como, rotação, translação e escala.

O objetivo final desta fase foi construir um modelo do Sistema Solar dado através dum ficheiro XML.

2 Transformações geométricas

2.1 Rotação

Como pretendemos uma movimentação circular de dado objeto, ou seja, uma rotação, é necessário estarem definidos o ângulo de rotação e as coordenadas do vetor que funciona como eixo deste movimento e, para além disso, as dimensões da figura não sofrem alterações.

A nossa implementação é, então a criação de uma classe que contém estes campos, isto é:

```
class Rotate {
public:
    float ang, x, y, z;
(...)
}
```

No entanto, para efetivamente aplicar esta transformação, temos de utilizar a primitiva do OpenGL, que é dada pelo seguinte método:

```
class Rotate {
public:
(...)
    void apply() {
        glRotatef(this->ang, this->x, this->y, this->z);
    }
}
```

É possível observar uma rotação nas figuras abaixo, sendo que a da direita representa a rotação da imagem da esquerda.

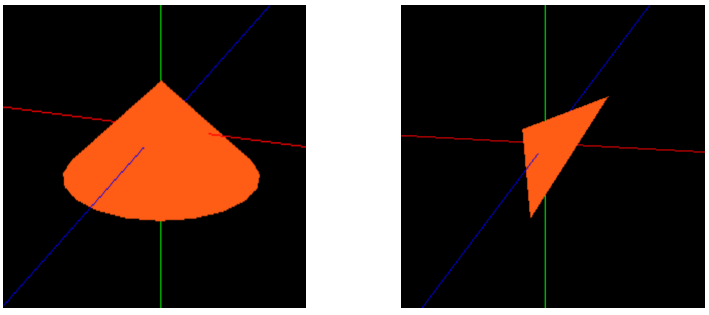


Figura 1: rotate angle=”60”x=”0”y=”0”z=”1”

2.2 Translação

Para as translações, é necessário saber o vetor que funciona como direção do movimento para calcular a posição final do objeto. São essencialmente movimentações em linha reta para um determinado ponto num dado referencial apartir das suas coordenadas originais, não alterando a figura na suas dimensões nem forma.

Deste modo, a classe que nós implementamos está definida da seguinte forma:

```
class Translate {
public:
  float x, y, z;
(...)
}
```

Tal como em 2.1, é necessário dar referência à primitiva do OpenGL que permite aplicar esta transformação, isto é, foi criado o seguinte método:

```
class Translate {
public:
(...)
    void apply() {
        glTranslatef(this->x, this->y, this->z);
    }
}
```

É possível observar uma translação nas figuras abaixo, sendo que a da direita representa a translação da imagem da esquerda.

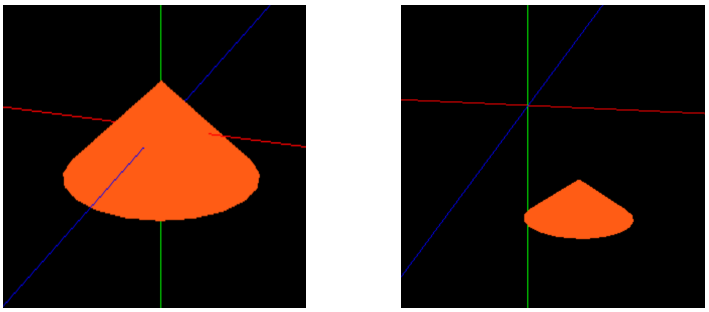


Figura 2: translate x=”5”y=-5”z=”7”

2.3 Escala

Escalar um objeto é essencialmente encolher ou expandir o próprio, ou seja, altera a sua forma. Para se obter este efeito, faz-se uma multiplicação das coordenadas do objeto por um fator de escala dado como argumento.

Tal como anteriormente, foi definido uma nova classe para este efeito com a seguinte assinatura:

```
class Scale {
public:
  float x, y, z; // nao as coordenadas mas sim o fator de
      escala nas 3 componentes
(...)
}
```

De igual modo a 2.1 e 2.2, o OpenGL já tem definida esta transformação, que é dada pelo seguinte método dentro da classe:

```
class Scale {
public:
(...)
    void apply() {
        glScalef(this->x, this->y, this->z);
    }
}
```

É possível observar uma escala nas figuras abaixo, sendo que a da direita representa a escalaa da imagem da esquerda.

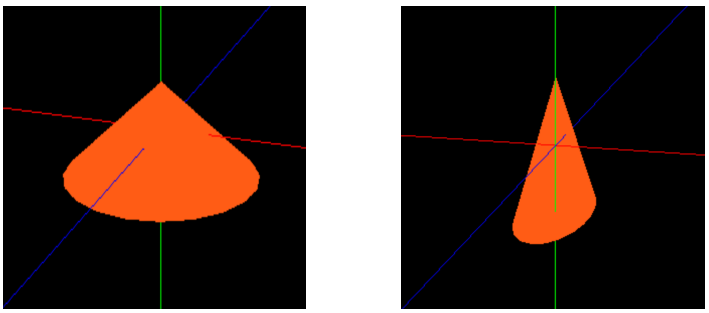


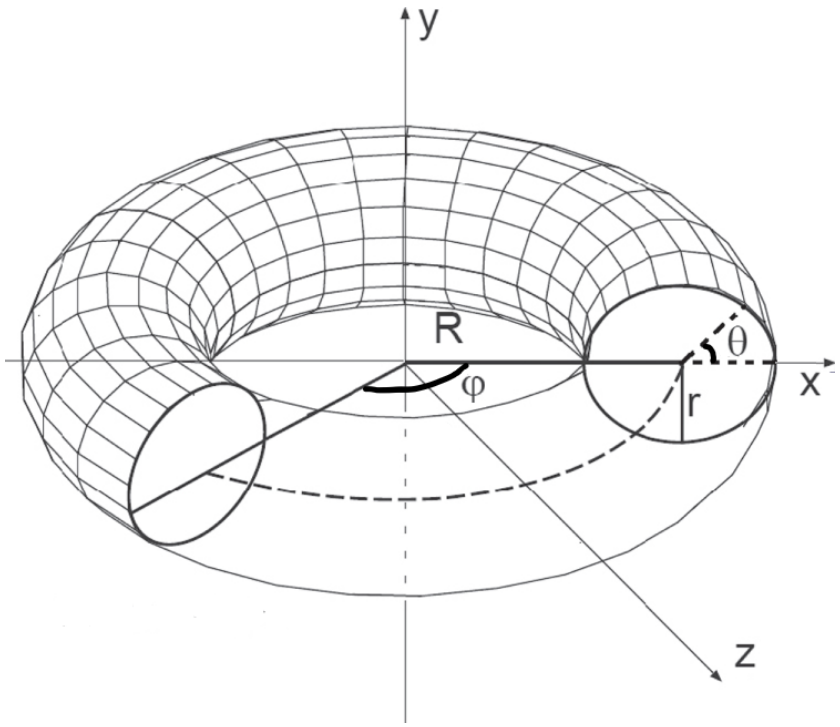
Figura 3: scale x=”0.8”y=”4”z=-2”

3 Gerador

3.1 Anéis de Saturno

Era necessário, como requisito obrigatório, entregar nesta fase um modelo estático do sistema solar e, para isso, foi necessário uma forma de desenhar os anéis de Saturno através do gerador.

Então, decidimos construir a primitiva de desenho *drawRing* que utiliza como algoritmo a construção de uma *torus* de maneira semelhante à primeira fase, isto é, através de múltiplos triângulos. Seguindo a informação disponível sobre esta forma geométrica na Wikipedia, ficamos a saber que cada um dos pontos pertencentes à superfície era definido da seguinte forma:



$$x(\theta, \varphi) = (R + r\cos\theta)\cos\varphi$$
$$y(\theta, \varphi) = (R + r\cos\theta)\sin\varphi$$
$$z(\theta, \varphi) = r\sin\theta$$

Para desenhar esta figura com triângulos, é necessário saber o nº de círculos em que o anel está dividido e o número de divisões desses círculos, para assim obter em cada iteração um dos vários quadrados que estão na superfície, consequentemente, os dois triângulos que o formam.

O algoritmo está descrito de seguida:

```
stepTheta = (2 * M_PI) / sides; // angulo de separacao de
    fatias nos circulos
stepPhi = (2 * M_PI) / rings; // angulo de separacao entre
    circulos
for (int i = 0; i < rings; i++) {
        theta1 = stepTheta * i;
        theta2 = stepTheta * (i + 1.0f);

        for (int j = 0; j < sides; j++) {
            phi1 = stepPhi * j;
            phi2 = stepPhi * (j + 1.0f);

            x1 = (rc + ro * cosf(phi1)) * cosf(theta1);
            y1 = (rc + ro * cosf(phi1)) * sinf(theta1);
            z1 = ro * sin(phi1);

            x2 = (rc + ro * cosf(phi2)) * cosf(theta1);
            y2 = (rc + ro * cosf(phi2)) * sinf(theta1);
            z2 = ro * sin(phi2);

            x3 = (rc + ro * cosf(phi1)) * cosf(theta2);
            y3 = (rc + ro * cosf(phi1)) * sinf(theta2);
            z3 = ro * sin(phi1);

            x4 = (rc + ro * cosf(phi2)) * cosf(theta2);
            y4 = (rc + ro * cosf(phi2)) * sinf(theta2);
            z4 = ro * sin(phi2);
}
```

Tendo o algoritmo pronto foi, portanto, adicionado mais um comando possível para o gerador:

1. generator ring <R><r><sides><rings><file>

4 Motor

4.1 Alteração do parse XML

Para garantirmos que funciona corretamente, temos de fornecer ao Motor um ficheiro XML válido. Um ficheiro XML válido tem o seguinte comportamento:

<scene>
  <group>
    <translate x="CX"y="CY"z="CZ"**/>**
    <rotate angle="DEGREES"x="CX"y="CY"z="CZ"**/>**
    <scale x="CX"y="CY"z="CZ"**/>**
    <colour r="RED_{RGB}"g = "$GREEN_{RGB}$"b = "$BLUE_{RGB}$"**/>**
    <models/>
      <model file=file1**/>**
      (...)
      <model file=fileN**/>**
    </models/>
    <group>
    (...)
    </group>
  (...)
  </group>
  <group>
(...)
  </group>
(...)
</scene>

Ou seja, definimos que uma **<scene>** tem vários **<group>**. Cada **<group>** tem:

1. um conjunto de transformações (apenas 1 de cada tipo, isto é, no máximo 1 rotação, 1 translação e 1 escala) e são **opcionais**

2. os ficheiros (**<models>**) que contêm as figuras a que as transformações do grupo respetivo serão aplicadas

3. a cor que terão as figuras (para distinguir os planetas nesta fase era pertinente este campo)

4. os sub-grupos, que são também outros **<group>** (cada grupo pode ter mais que 1 sub-grupo).

Para tratar estes dados foi preciso, então, refazer o nosso processo de parsing. Criamos uma nova classe, *XMLReader*, que utilizando novamente o *tinyxml*, verifica que tipo de elemento estamos a processar e processa de forma diferente todos os elementos.

Caso seja:

- **scene**: é assumido por defeito que o ficheiro começa sempre com este elemento;
- **translate, rotate ou scale**: é adicionada esta transformação ao objeto *Transformations* do grupo atual;
- **colour**: é colocada esta cor objeto *Transformations* atual (todos os componentes são divididos por 255 porque estão definidos como inteiros no XML, como o OpenGL utiliza RGB com valores de 0 a 1 era necessário esta mudança);
- **models**: é feito a procura pelos sub-elementos **model** para descobrir quais são as figuras a adicionar ao objeto *Transformations*;
- **model**: é adicionado ao objeto *Transformations* um objeto do tipo *Model* (visto em mais detalhe em 4.2), que representa a figura, pelo ficheiro que está nomeado dentro do elemento **file**;
- **group**: é criado um novo objeto de *Transformations* (a ser visto em 4.3) e é preenchido com os seus atributos e elementos (outros grupos ou figuras). É adicionado ao objeto *Transformations* do grupo pai se for um sub-grupo.

4.2 Representação de ficheiros

Os ficheiros gerados pelo Gerador e referenciados no ficheiro XML têm de ser carregados no programa uma única vez e, como já visto na fase 1, decidimos implementar a estrutura de dados *Point* que guardará informação de todos os vértices da figura:

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

Portanto, como as figuras nesta fase não são mais que um conjunto destes pontos, escrevemos a seguinte definição de classe para as mesmas:

```
class Model {
    vector<Point> points;
(...)
}
```

Para efetivamente desenhar a figura no ecrã, implementamos um método ao objeto *Model* que retira 3 *Point* a cada iteração e desenha 1 triângulo com a função *glVertex3f* e com a cor dada por argumento, até iterar o vetor por completo:

```
class Model {
(...)
public:
    void drawModel(float red, float green, float blue) {
        int size = this->points.size();
        Point p1, p2, p3;
        glColor3f(red, green, blue);
        glBegin(GL_TRIANGLES);
        for (int i = 0; i < size; i += 3) { //
            p1 = this->points.at(i);
            p2 = this->points.at(i + 1);
            p3 = this->points.at(i + 2);
            glVertex3f(p1.x, p1.y, p1.z);
            glVertex3f(p2.x, p2.y, p2.z);
            glVertex3f(p3.x, p3.y, p3.z);
        }
        glEnd();
  }
}
```

4.3 Implementação de Várias Transformações

Colocando este raciocínio em C++, o que fizemos foi traduzir diretamente os pontos enumerados na secção 4.1, que definem a estrutura. Decidimos, então, definir um objeto do tipo *Transformations* que é, na prática, um **<group>** do ficheiro XML.

```
class Transformations {
    Translate* translate; // translacao opcional
    Rotate* rotate; // rotacao opcional
    Scale* scale; // escala opcional
    vector<Model*> models; // ficheiros presentes no grupo
    vector<Transformations*> subgroups; // sub-grupos deste
       grupo
    float red, green, blue; // cor dos modelos do grupo em
       RGB
(...)
}
```

Para aplicarmos as transformações na ordem correta, isto é, as transformações de um grupo são aplicadas em todos os seus sub-grupos (o contrário não acontece), tiramos proveito do sistema de *stack* de matrizes fornecido pelo OpenGL, através das primitivas *glPushMatrix()* e *glPopMatrix()* para as *GL_MODELVIEW*, uma vez que este aninhamento de transformações entre os vários **<group>** e os seus filhos equivale exatamente ao comportamento de uma *stack*.

Assim, a matriz do grupo pai é colocada diretamente na stack mal é encontrado um sub-grupo, para se reverterem as transformações associadas a estes.

Portanto, para desenhar no ecrã um **<group>**, temos o seguinte método:

```
class Transformations{
(...)
public:
(...)
    void drawAll() {
        glPushMatrix();
        if (this->rotate) this->rotate->apply();
        if (this->translate) this->translate->apply();
        if (this->scale) this->scale->apply();

        for (Model* m : this->models) {
            m->drawModel(red,green,blue);
        }
        for (Transformations* t : this->subgroups) {
            t->drawAll();
        }
        glPopMatrix();
    }
(...)
}
```

Tendo em conta o que foi visto anteriormente, agora que temos um ficheiro XML mais complexo, já não guardamos um *vector<Point>* para definir as nossas figuras dentro do nosso *main.cpp*.

Passamos então a ter:

```
vector<Transformations>* transformations;
```

Que representam os vários **group** dentro de uma **scene**. O método *draw()*, que desenha as várias figuras, é agora iterativo, onde faz referência ao método *drawAll()* disponibilizado na classe *Transformations*:

```
void draw(){
    for (Transformations t : *transformations) {
        t.drawAll();
    }
}
```

5 Executar o projeto

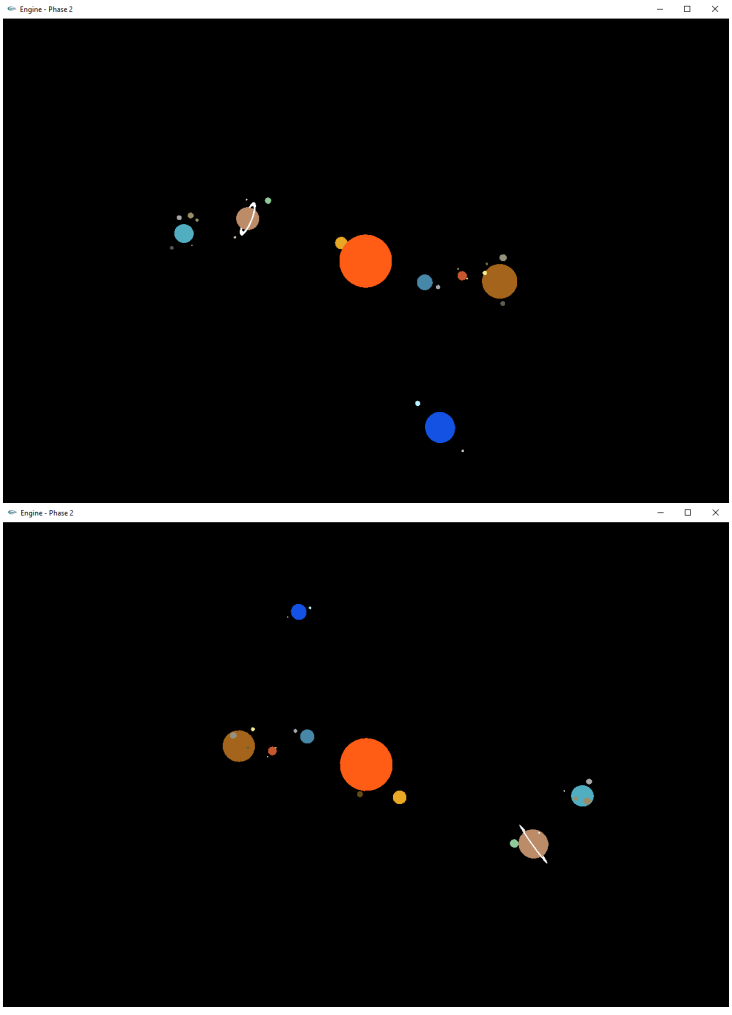
Para correr tanto o Gerador como o Motor, à semelhança da 1ª parte do projeto, **todos** os ficheiros (XML ou as figuras gerados pelo Gerador) têm de estar dentro da pasta **files** do projeto gerada pelo CMake. A título de exemplo, se a pasta onde o projeto foi gerado se chamar **build**, então terá de existir uma pasta **/build/files/**. A pasta será criada por defeito caso exista pelo menos a criação de uma figura pelo Gerador, caso contrário terá de ser adicionada manualmente.

6 Resultado

6.1 Sistema Solar

Para esta fase decidimos não usar um sistema solar em escala real (**system_scale.xml**) porque os planetas ficam demasiado pequenos para serem visíveis e o sol demasiado grande. Utilizamos, portanto, uma escala mais liberal, que está no ficheiro **system.xml**.

Obtivemos os seguintes resultados:



7 Extras

7.1 Câmara controlada com o rato

Para ser mais fácil movimentar a câmara (usando uma implementação de câmara de explorador), foi implementado este efeito de movimento num periférico que permite melhor este controlo, que é o rato.

Foram acrescentadas duas funções que registam quando o rato é **deslocado** ou quando um dos botões é **clicado**.

```
1  float cam_radius = 200.0f; // raio da esfera que a camara
       esta posicionada
2  float cam_alpha = 0, cam_beta = 0; // deslocamento para a
       esquerda/direita (alpha) ou para cima/baixo (beta)
3  float cx, cy, cz; // posicao da camara numa esfera (
       perspetiva exploradora)
4  float lastX, lastY; // ultima posicao na janela
5  bool start = true; // so para calcular a primeira iteracao do
        lastX e lastY
6  bool zoom = false; // se estamos perante um zoom-in ou zoom-
       out
7  (...)
8  void mouseMove(int x, int y) {
9      if (start) { // se for a primeira vez que executa grava
           as primeiras coordenadas que o rato clicou
           relativamente a janela
10         lastX = x;
11         lastY = y;
12         start = false;
13     }
14     // e preciso saber o deslocamento nos eixos X e Y do
           ponteiro na janela a partir da ultima iteracao
15     float x_off = x - lastX; // offset relativo ao ultimo X
           clicado (angulo)
16     float y_off = y - lastY; // offset relativo ao ultimo Y
           clicado (angulo)
17     if (lastX >= 0) { // so executa se um dos botoes for
           clicado
18         if (zoom) { // se for o botao direito a ser clicado e
               realizado um zoom-in ou zoom-out
19             if (cam_radius + y_off > 0) { // para nao
                   inverter as figuras
20                 cam_radius += y_off; // atualizar o raio em
                       que a camara esta posicionada
21             }
```

```
22          }
23          else { // se for o botao esquerdo atualizamos a
                posicao na esfera (camara)
24              cam_alpha -= x_off * 0.001f; // alpha representa
                    o angulo no eixo xOz (X da janela), e
                    atualizado consoante o quanto o rato foi
                    movido para a esquerda ou direita (offset)
25              float camOffset = y_off * 0.001f; // e calculado
                    o offset no eixo dos Y da janela consoante o
                    quanto o rato foi movido para cima ou para
                    baixo
26              if (cam_beta + camOffset < -1.5f ) { // para nao
                    passar dos -1.5 radianos (parte das regras da
                    camara exploradora)
27                  cam_beta = -1.5f;
28              }
29              else if (cam_beta + camOffset > 1.5f) { // mesma
                    situacao que a regra anterior
30                  cam_beta = 1.5f;
31              }
32              else { // caso nao ultrapasse o limite, e
                    retornado o novo angulo no eixo xOy (Y da
                    janela)
33                  cam_beta += camOffset;
34              }
35          }
36          lastX = x; // ultimas coordenadas que o rato clicou
                atualizadas
37          lastY = y;
38      }
39      spherical2Cartesian();
40      glutPostRedisplay();
41  }
```

```
1
2   void mouseButton(int button, int state, int x, int y) {
3     // quando um does botoes do rato ja nao esta a ser clicado
4     if (state == GLUT_UP) {
5       lastX = -1; // como X,Y sao sempre positivos, ao dizer
            que lastX e negativo e equivalente a dizer que nao foi
             clicado (util para a funcao de cima, nao mexer na
            camara se o botao esquerdo nao tiver sido clicado)
6     }
7     else {
8       lastX = x; // caso esteja a ser premido, gravar
            coordenadas
```

```
9       lastY = y;
10    }
11    if (button == GLUT_RIGHT_BUTTON) { // se for o botao
          direito, damos a indicacao a funcao de cima que estamos
          perante um zoom-in ou zoom-out
12      state == GLUT_UP ? zoom = false : zoom = true;
13    }
14  }
```

É evidente que estas funções nada fariam se não fossem disponibilizadas as primitivas do OpenGL que permite associar este tipo de eventos (como a interação com periféricos) com o comportamento do programa.

```
1  int main(int argc, char **argv){
2      (...)
3      glutMouseFunc(mouseButton);
4      glutMotionFunc(mouseMove);
5      (...)
6  }
```

7.2 Boneco de Neve

Uma demonstração extra que decidimos fazer para esta fase foi a criação de um homem de neve.

A cena é dividida nos vários componentes do boneco de neve, como os seus braços, nariz, olhos, botões, boca, base, barriga, cabeça. Os braços e o nariz são cones, os botões, os olhos e a boca são esferas, assim como a base, cabeça e barriga. Todas estas figuras são colocadas no seu respetivo lugar usando as transformações geométricas necessárias, como é possível analisar com mais detalhe no ficheiro anexado (**snowman.xml**).

Seguem-se os resultados obtidos:

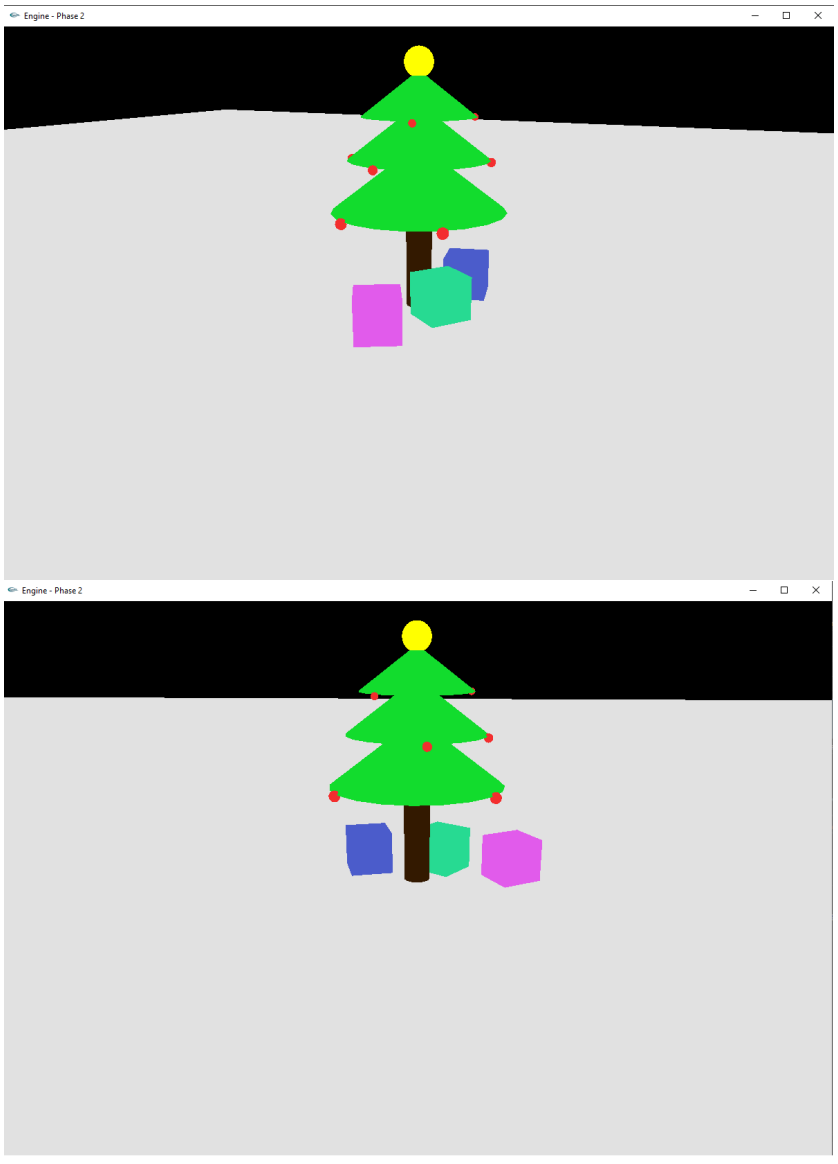
7.3 Árvore de Natal

Como outro projeto extra para esta fase, criamos um modelo de uma árvore de natal com presentes, enfeites e um solo.

A cena é dividida em vários grupos: chão, tronco, presentes, folheado e enfeites. O chão é definido por um plano, o tronco por um cilindro, os

vários presentes são cubos (caixas com 0 divisões no Gerador), o folheado são empilhamentos de cones e os enfeites são esferas, colocadas nas respetivas posições através de transformações, como é possível ver no ficheiro XML anexado (**christmas_tree.xml**).

Em seguida estão as imagens que mostram como ficou a cena final.



8 Conclusão

Em suma, nesta segunda fase do trabalho, o grupo considera que já aparenta ter um conhecimento da matéria mais aprodundado, tendo em conta que tudo o que foi pedido no enunciado foi conseguido, e que ainda deu tempo para adicionar uns extras.

Foi dada a oportunidade de perceber o quão útil é o uso de transformações geométricas e a importância da ordem destas mesmas.

Por fim, terminada esta segunda fase, o objetivo será começar já a trabalhar na fase seguinte, de modo a tornar o modelo mais realista e mais apelativo, e continuando assim aprofundar o conhecimento sobre Computação Gráfica.