Computação Gráfica (3º ano de LCC)

Trabalho Prático (Fase 2) — Grupo 3

Relatório de Desenvolvimento

André Lucena Ribas Ferreira (A94956) Carlos Eduardo da Silva Machado (A96936)
Gonçalo Manuel Maia de Sousa (A97485)

14 de abril de 2023

Resumo

Este relatório aborda a solução proposta para o enunciado da 2ª fase do Trabalho Prático da Unidade Curricular "Computação Gráfica".

Conteúdo

Capítulo 1

Introdução

O presente relatório tem como objetivo apresentar a solução concebida pelo Grupo 3 para a 2ª fase do Trabalho Prático da Unidade Curricular "Computação Gráfica".

Esta fase consiste em apenas modificar o *engine* de forma a possuir, agora, a possibilidade de definir hierarquias de grupos e transformações. Além disso, também construímos uma demo estática do sistema solar.

1.1 Estrutura do Relatório

Para além deste, o relatório compreende diferentes Capítulos. Em 2 apresenta-se extensão à implementação da aplicação *generator*. Em 3 apresenta-se as extensões à implementação da aplicação *engine*. Em 4 expõe-se imagens tiradas aos modelos gerados a partir dos *xmls* dos *test files*. Em 5 apresenta-se a conclusão do relatório.

Capítulo 2

Generator

Decidimos implementar como figura extra o Cilindro, já tendo implementado o *Torus* no guião passado.

Os parâmetros necessários para definir um cilindro são: o raio (*radius*), a altura (*height*), as fatias (*slices*) e pilhas (*stacks*).

A implementação do cilindro é análoga às outras primitivas que já definimos, nomeadamente da esfera, por se basear em definir os triângulos verticalmente para cada *slice* e posteriormente rodar sobre o seu eixo.

Para tal, definimos um ângulo alfa através de $2*\pi/slices$, através desse alfa vamos poder criar o triângulo correspondente a cada fatia do cilindro

Iniciamos com a construção da base do cilindro, com um triângulo para cada slice criando três pontos que juntos geram um triângulo. Um deles será sempre a origem, para que a base do cone fique no plano xOz, enquanto os outros dois serão pontos no perímetro da circunferência, no sentido dos ponteiros do relógio para a orientação da base ser no sentido negativo do eixo Oy. A construção da base de cima do cilindro que segue o mesmo principio da base baixo, diferindo apenas na altura. A função é semelhante à outras criadas, utilizamos um **std:vector** que será útil para guardar os pontos em tuplos. Consideram-se circunferências ao longo da altura do cilindro, com saltos ditados pelos valores *division_height_step*. Para cada uma das *slices*, definem-se dois triângulos orientados para fora do sólido geométrico. Para tal, utilizam-se coordenadas análogas à rotação ao redor do eixo *Oy*, por se tratar de uma circunferência.

```
1  vector<tuple<float,float, float>>* generate_cylinder(float radius, float height, int slices,
       int stacks){
2      vector<tuple<float,float, float>>* point_array = new vector<tuple<float,float,float>>;
3
4      float division_height_step = height/stacks;
5      float alfa = 2*M_PI/slices;
6
7      for (int i = 0; i < slices; i++) {
8          //bottom part
9          point_array->push_back(make_tuple(0.0f,-height/2, 0.0f));
10         point_array->push_back(make_tuple(radius*sin(alfa * (i+1)),-height/2, radius*cos(
       alfa * (i+1))));
11         point_array->push_back(make_tuple(radius*sin(alfa * i),-height/2, radius*cos(alfa *
       i)));
12
13         //top part
14         point_array->push_back(make_tuple(radius*sin(alfa * i),height/2, radius*cos(alfa * i
       )));
15         point_array->push_back(make_tuple(radius*sin(alfa * (i+1)),height/2, radius*cos(alfa
        * (i+1))));
16         point_array->push_back(make_tuple(0.0f,height/2, 0.0f));
17
18         //middle part
```

```
19          for(int j=0; j<stacks; j++){
20              double bot_height = -height/2 + j*division_height_step;
21              double top_height = bot_height + division_height_step;
22
23              point_array->push_back(make_tuple(radius*sin(alfa * i),bot_height, radius*cos(
    alfa * i)));
24              point_array->push_back(make_tuple(radius*sin(alfa * (i+1)),bot_height, radius*
    cos(alfa * (i+1))));
25              point_array->push_back(make_tuple(radius*sin(alfa * i),top_height, radius*cos(
    alfa * i)));
26
27              point_array->push_back(make_tuple(radius*sin(alfa * i),top_height, radius*cos(
    alfa * i)));
28              point_array->push_back(make_tuple(radius*sin(alfa * (i+1)),bot_height, radius*
    cos(alfa * (i+1))));
29              point_array->push_back(make_tuple(radius*sin(alfa * (i+1)),top_height, radius*
    cos(alfa * (i+1))));
30          }
31    }
32
33     return point_array;
34 }
```

Depois, no main, basta extrair do **std::vector** o número de pontos com o método **size()** e o array de tuplos com o método **data()** e escrevemos em ficheiro com a função *points_write* criada no guião anterior.

```
1 if(!strcmp(argv[1], "cylinder")){
2         vector<tuple<float,float, float>>* cylinder = generate_cylinder(atof(argv[2]), atof(
    argv[3]), atoi(argv[4]), atoi(argv[5]));
3         points_write(argv[6], cylinder->size(), cylinder->data());
4         free(cylinder);
5     }
```

Capítulo 3

Engine

Neste capítulo, vamos abordar as mudanças que fizemos ao código relativo ao *engine* de modo a suprir as novas necessidades enunciadas na fase 2 e alguns extras.

Deste modo, vamos dividir este capítulo em:

- Nova Definição de Grupos e Transformações Geométricas
- Câmara

3.1 Nova Definição de Grupos e Transformações Geométricas

Nesta fase temos de ter a possibilidade de criar subgrupos para cada grupo do ficheiro **xml**, à exceção do primeiro grupo, criando uma hierarquia. Então alteramos a classe *Group* para possuir um **std::vector** de apontadores para *Group*, onde se guardarão os seus filhos, já que estes vão herdar as transformações de trás. No ficheiro **xml**, cada grupo pode possuír um campo *transform* com transformações, sendo que só podemos ter um transformação de cada tipo (*rotate*, *translate* e *scale*) e a ordem das transformações é importante. Tendo isso em conta, na classe é adicionado um **std::vector** *transformations* que guarda um *char* com a inicial de cada transformação na ordem que foi dada. Além disso, guardamos as coordenadas x, y e z, no caso da rotação também o ângulo, nos *arrays* correspondentes.

```
1  class Group{
2  public:
3      std::vector<Model*> models;
4
5      std::vector<char> transformations;
6      float translate[3];
7      float rotate[4];
8      float scale[3];
9
10     std::vector<Group*> subGroups;
11 };
```

No ficheiro *parser.cpp*, a função *parse_group* foi alterada para possuir recursividade e transformações. Além disso, passamos tudo relativo aos modelos para uma função *parse_group_models* e criamos a função *parse_group_transform* para lidar com o *parsing* **xml** das transformações.

```
1  void parse_group(xml_node<> *group_node, Group* group){
2      xml_node<>* temp;
3
4      // Transformacoes
5      if((temp = group_node->first_node("transform")))
6          parse_group_transform(temp, group);
```

```
7
8       // Modelos
9       if((temp = group_node->first_node("models")))
10          parse_group_models(temp, group);
11
12      // Grupos
13      for(temp = group_node->first_node("group"); temp; temp = temp->next_sibling("group")){
14          Group *groupChild = new Group;
15          group->subGroups.push_back(groupChild);
16          parse_group(temp, groupChild);
17      }
18  }
```

```
1   void parse_group_models(xml_node<> *node_Models, Group* group){
2       for(xml_node<> *node_models = node_Models->first_node();node_models; node_models =
        node_models->next_sibling()){
3           // Criar fstream e abrir
4           fstream filestream;
5           filestream.open(node_models->first_attribute()->value(), ios::in|ios::binary);
6           // Ler inteiro para o n
7           int n;
8           filestream.read((char*)&n,sizeof(int));
9
10          // Ler array de tuplos
11          tuple<float,float,float>* tuples = new tuple<float,float,float>[n];
12          filestream.read((char*)tuples, sizeof(tuple<float,float,float>) * n);
13
14          // fechar o ficheiro
15          filestream.close();
16
17          // Criar o model, guardar os tuplos e o inteiro no model, guardar o model no group
18          Model* model = new Model;
19          model->figure = tuples;
20          model->size = n;
21          group->models.push_back(model);
22      }
23  }
```

Na função *parse_group_transform*, fazemos o que foi descrito anteriormente: percorremos os nodos *translate* dentro do nodo *transform* através de um ciclo *for* pois não temos certeza do número exato de transformações que o o grupo poderá ter, apenas que podemos ter no mínimo 0 transformações e no máximo 3 (uma de cada tipo).

Em cada transformação, caso não encontremos algum dos atributos, tomamos como *default* o valor 0. A lógica é idêntica à das outras funções de *parsing*, comparamos o nome do nodo com as nossas possibilidades, se a string é a mesma, verificamos a existência de cada um dos atributos e valores.

```
1   void parse_group_transform(xml_node<> *node_transform, Group* group){
2
3       for(xml_node<> *node_temp = node_transform->first_node(); node_temp; node_temp =
        node_temp->next_sibling()){
4
5           if(!strcmp(node_temp->name(), "translate")){
6               group->transformations.push_back('t');
7
8               xml_attribute<> *attr;
9               if((attr = node_temp->first_attribute("x")))
10                  group->translate[0] = atof(attr->value());
11              else
12                  group->translate[0] = 0;
```

```
13
14              if((attr = node_temp->first_attribute("y")))
15                  group->translate[1] = atof(attr->value());
16              else
17                  group->translate[1] = 0;
18
19              if((attr = node_temp->first_attribute("z")))
20                  group->translate[2] = atof(attr->value());
21              else
22                  group->translate[2] = 0;
23          } else if(!strcmp(node_temp->name(), "rotate")){
24              group->transformations.push_back('r');
25
26              xml_attribute<> *attr;
27              if((attr = node_temp->first_attribute("angle")))
28                  group->rotate[0] = atof(attr->value());
29              else
30                  group->rotate[0] = 0;
31
32              if((attr = node_temp->first_attribute("x")))
33                  group->rotate[1] = atof(attr->value());
34              else
35                  group->rotate[1] = 0;
36
37              if((attr = node_temp->first_attribute("y")))
38                  group->rotate[2] = atof(attr->value());
39              else
40                  group->rotate[2] = 0;
41
42              if((attr = node_temp->first_attribute("z")))
43                  group->rotate[3] = atof(attr->value());
44              else
45                  group->rotate[3] = 0;
46          } else if(!strcmp(node_temp->name(), "scale")){
47              group->transformations.push_back('s');
48
49              xml_attribute<> *attr;
50              if((attr = node_temp->first_attribute("x")))
51                  group->scale[0] = atof(attr->value());
52              else
53                  group->scale[0] = 0;
54
55              if((attr = node_temp->first_attribute("y")))
56                  group->scale[1] = atof(attr->value());
57              else
58                  group->scale[1] = 0;
59
60              if((attr = node_temp->first_attribute("z")))
61                  group->scale[2] = atof(attr->value());
62              else
63                  group->scale[2] = 0;
64          }
65      }
66  }
```

No *engine.cpp*, passamos a parte de desenho do grupo para uma função própria, também recursiva, que vai em percorrer cada modelo e desenhá-lo de acordo com as transformações indicadas, repetindo-o para todos os seus filhos, recursivamente, de modo a que funciona como uma travessia em profundidade. Em relação às transformações, percorremos o **std::vector** com as inicias de cada uma delas, e através de um *switch*, decidimos a função do *glut* respetiva,

inserindo os elementos do array da transformação.

```
void drawGroup(Group* group){
  glPushMatrix();

  for(char transformation: group->transformations){
    switch(transformation){

      case 't': {
        glTranslatef(group->translate[0], group->translate[1], group->translate[2]);
        break;
      }

      case 'r': {
        glRotatef(group->rotate[0], group->rotate[1], group->rotate[2], group->rotate[3]);
        break;
      }

      case 's':{
        glScalef(group->scale[0], group->scale[1], group->scale[2]);
        break;
      }

    }
  }

  glBegin(GL_TRIANGLES);
  for(Model* groupModel: group->models){
    for(int i=0; i<groupModel->size; i++){
      glVertex3f(get<0>(groupModel->figure[i]), get<1>(groupModel->figure[i]), get<2>(
    groupModel->figure[i]));
    }
  }
  glEnd();

  for(Group* groupChild: group->subGroups)
    drawGroup(groupChild);


  glPopMatrix();
}
```

3.2 Câmara

Como funcionalidade adicional, de modo a melhor visualizar a *demo* criada, adicionou-se componentes FPS de movimentação e de rotação à câmara da cena.

Para tal, tornou-se necessário definir 3 vetores:

- Vetor *up*, dado como parâmetro do ficheiro *xml*, que representa o vetor vertical da câmara;
- Vetor *d*, que representa a direção da vista da câmara, calculada subtraindo *lookAt* de *position*, ambos dados como parâmetros do ficheiro *xml*;
- Vetor *r*, que representa a direção de deslocamento para a direita da câmara. É calculada pelo produto externo $d \times up$.

e também diversas variáveis, que ditam a diferença de rotação e quanto já foi alterado em cada um dos eixos de rotação/translação:

```
float  camera_move_delta = 1, look_rotate_delta_up = M_PI / 32, look_rotate_delta_right =
    M_PI / 32;
int camera_side = 0, camera_up = 0, camera_front = 0, look_rotate_up = 0, look_rotate_right
    = 0;
float saved[3];
```

Desse modo, são possíveis os seguintes comandos:

- **'w'**: Deslocar no sentido do vetor d;
- **'a'**: Deslocar no sentido contrário ao vetor r;
- **'s'**: Deslocar no sentido contrário ao do vetor d;
- **'d'**: Deslocar no sentido do vetor r;
- **'u'**: Rodar o vetor d no sentido positivo ao redor do vetor r;
- **'j'**: Rodar o vetor d no sentido negativo ao redor do vetor r;
- **'h'**: Rodar o vetor d no sentido positivo ao redor do vetor up;
- **'k'**: Rodar o vetor d no sentido negativo ao redor do vetor up;
- **spacebar**: Deslocar no sentido positivo do eixo y;
- **shift** + **spacebar**: Deslocar no sentido negativo do eixo y;
- **'r'**: Repor os valores iniciais da *demo*.

3.2.1 Movimento da Câmara

O movimento da câmara tem em consideração o minimizar dos erros obtidos pelo somar da posição ao longo de vários movimentos. O movimento numa dada direção é adquirido por multiplicação de um escalar até ocorrer uma rotação do eixo da direção, o vetor d. Nesse momento, guarda-se a posição atual de modo a poder repor o escalar e considerar uma nova direção.

Este comportamento encontra-se implementado no vetor *saved*, que guarda a última posição onde o vetor direção foi alterado, e a função *save_position*, que calcula a última posição como ocorreria na função *render_scene*.

```
void save_position() {
  float d[3] = { camera_global->lookAt[0] - camera_global->position[0],
    camera_global->lookAt[1] - camera_global->position[1],
    camera_global->lookAt[2] - camera_global->position[2] };

  normalize_vector(d);

  normalize_vector(camera_global->up);

  rotate_over_vector(d, camera_global->up, look_rotate_right * look_rotate_delta_right);

  float r[3] = { d[1] * camera_global->up[2] - camera_global->up[1] * d[2],
    d[2] * camera_global->up[0] - camera_global->up[2] * d[0],
    d[0] * camera_global->up[1] - camera_global->up[0] * d[1] };

  normalize_vector(r);

```

```
18    rotate_over_vector(d, r, look_rotate_up * look_rotate_delta_up);
19
20    saved[0] += d[0] * camera_move_delta * camera_front + r[0] * camera_move_delta *
        camera_side + camera_up * camera_move_delta * camera_global->up[0];
21    saved[1] += camera_side * camera_move_delta * r[1] + camera_front * camera_move_delta * d
        [1] + camera_up * camera_move_delta * camera_global->up[1];
22    saved[2] += camera_side * camera_move_delta * r[2] + camera_front * camera_move_delta * d
        [2] + camera_up * camera_move_delta * camera_global->up[2];
23    camera_front = 0;
24    camera_up = 0;
25    camera_side = 0;
26  }
```

3.2.2 Funções Auxiliares

Todos os vetores em cima mencionados são calculados e normalizados, utilizando a função *normalize_vector*.

```
1  float normalize_vector(float p[3]) {
2    float norm = sqrt(pow(p[0], 2) + pow(p[1], 2) + pow(p[2], 2));
3    p[0] = p[0] / norm;
4    p[1] = p[1] / norm;
5    p[2] = p[2] / norm;
6    return norm;
7  }
```

As rotações sobre os diferentes vetores têm em consideração os cálculos necessários para que tal ocorra, nomeadamente a de efetuar duas rotações para colocar o vetor de rotação pretendido no plano yOz e no eixo z, efetuando a rotação ao redor desse eixo e, por fim, recolocar o vetor através de rotações contrárias. Este comportamento encontra-se implementado na função *rotate_over_vector*, considerando a matriz de rotação com os parâmetros apenas a depender do ponto inicial, do vetor e do ângulo de rotação.

```
1  void rotate_over_vector(float p[3], float v[3], float angle) {
2    float q[3];
3    float omc = 1 - cos(angle), s = sin(angle), c = cos(angle);
4
5    q[0] = p[0] * (pow(v[0], 2) * omc + c) + p[1] * (v[0] * v[1] * omc - v[2] * s) + p[2] * (v
       [0] * v[2] * omc + v[1] * s);
6    q[1] = p[0] * (v[1] * v[0] * omc + v[2] * s) + p[1] * (pow(v[1], 2) * omc + c) + p[2] * (v
       [1] * v[2] * omc - v[0] * s);
7    q[2] = p[0] * (v[2] * v[0] * omc - v[1] * s) + p[1] * (v[2] * v[1] * omc + v[0] * s) + p
       [2] * (pow(v[2], 2) * omc + c);
8
9    p[0] = q[0];
10   p[1] = q[1];
11   p[2] = q[2];
12 }
```

3.2.3 Alterações à Função *render_scene*

Utilizando estas duas funções, e tendo em conta o comportamento pretendido, alterou-se a função que coloca a cena para se calcular tanto a nova posição como o novo ponto de vista da câmara. Posteriormente, este código será colocado numa única função que devolverá os parâmetros a se passar à função *gluLookAt*.

Em primeiro lugar, calcula-se o vetor *d*. A variável *norm* armazena a norma deste vetor, após normalizado, para se ter calculada a distância entre o ponto original e o ponto de vista original, distância esta que não se deve alterar com as rotações e com as movimentações da câmara.

Normaliza-se o vetor *d* para, no cálculo do deslocamento, se poder definir com clareza qual o *delta* do movimento, não estando dependente da distância que o ponto de vista estava da posição inicial.

```
float d[3] = { camera_global->lookAt[0] - camera_global->position[0],
    camera_global->lookAt[1] - camera_global->position[1],
    camera_global->lookAt[2] - camera_global->position[2] };

float norm = normalize_vector(d);
```

Após se normalizar o vetor *up*, calcula-se a rotação "horizontal"da direção, rodando o vetor *d* ao redor do *up*:

```
normalize_vector(camera_global->up);

rotate_over_vector(d, camera_global->up, look_rotate_right * look_rotate_delta_right);

```

Apenas neste momento é que se calcula o vetor *r*, já que este depende desta nova direção que *d* tomou. Posteriormente, efetua-se a sua rotação.

```
float r[3] = { d[1] * camera_global->up[2] - camera_global->up[1] * d[2],
    d[2] * camera_global->up[0] - camera_global->up[2] * d[0],
    d[0] * camera_global->up[1] - camera_global->up[0] * d[1] };

normalize_vector(r);

rotate_over_vector(d, r, look_rotate_up * look_rotate_delta_up);
```

O deslocamento da câmara é calculado da seguinte forma, tendo em conta os *delta* de translação e as direções frente/direita:

```
float desl[3] = { d[0] * camera_move_delta * camera_front + r[0] * camera_move_delta *
    camera_side ,
    camera_side * camera_move_delta * r[1] + camera_front * camera_move_delta * d[1] +
    camera_up * camera_move_delta,
    camera_side * camera_move_delta * r[2] + camera_front * camera_move_delta * d[2]
  };
```

Por fim, a posição da câmara é este deslocamento somado à última posição guardada na variável *saved*, e o ponto de vista o mesmo somando a distância de vista multiplicada pela direção da vista, o vetor *d*.

```
gluLookAt(  desl[0] + saved[0], desl[1] + saved[1], desl[2] + saved[2],
        saved[0] + desl[0] + d[0] * norm,
        saved[1] + desl[1] + d[1] * norm,
        saved[2] + desl[2] + d[2] * norm,
        camera_global->up[0], camera_global->up[1], camera_global->up[2]);
```

Capítulo 4

Resultados

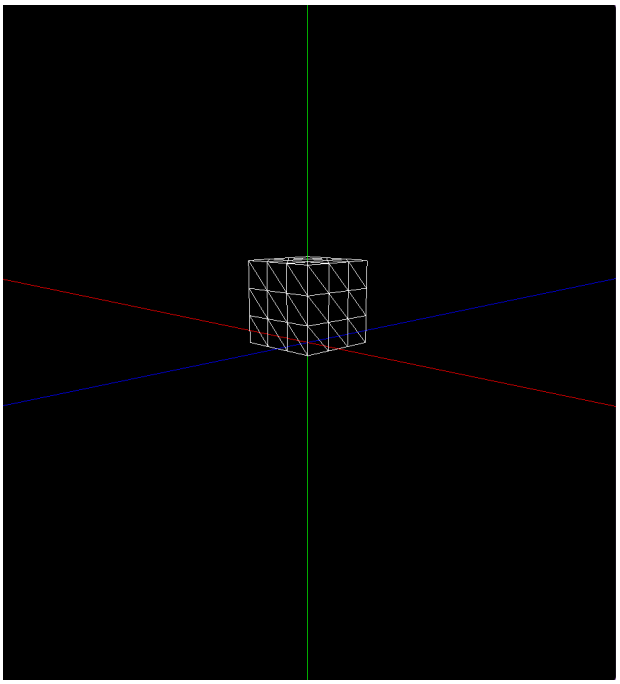
Neste capítulo apresentamos os resultados obtidos da execução de ambas as aplicações utilizando os ficheiros de teste fornecidos.

4.1 Demo

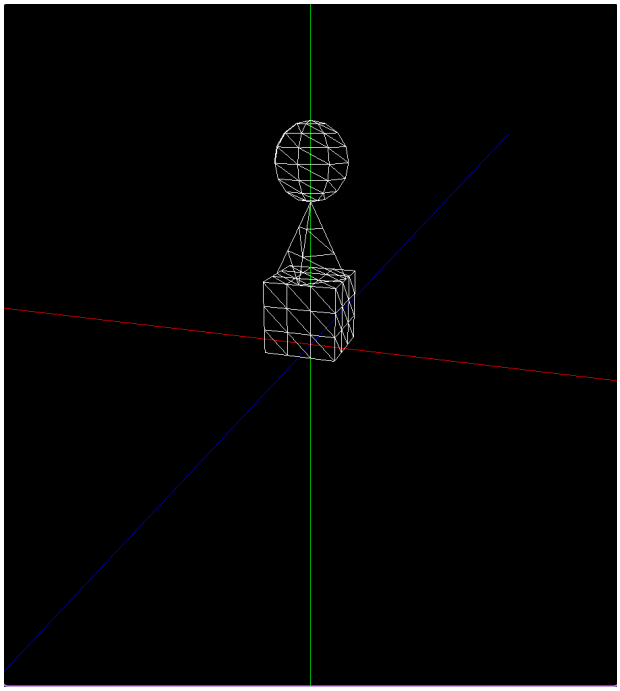
A Demo consistiu em gerar cada um dos planetas do Sistema Solar, incluindo também as suas luas, provisionais ou não. Para tal, foram usados *scripts* para se poder gerar valores aleatórios de rotação ao redor dos planetas para as suas luas, tal como calcular distâncias e escalas reais a partir da definição de escala do sol, isto é, a sua escala ditava quanto é que eram reduzidos os kilometros para o sistema de coordenadas usada.

Para o desenho de Saturno, utilizou-se um *Torus* com o valor de escala de y próximo de 0, para aparecer praticamente plano.

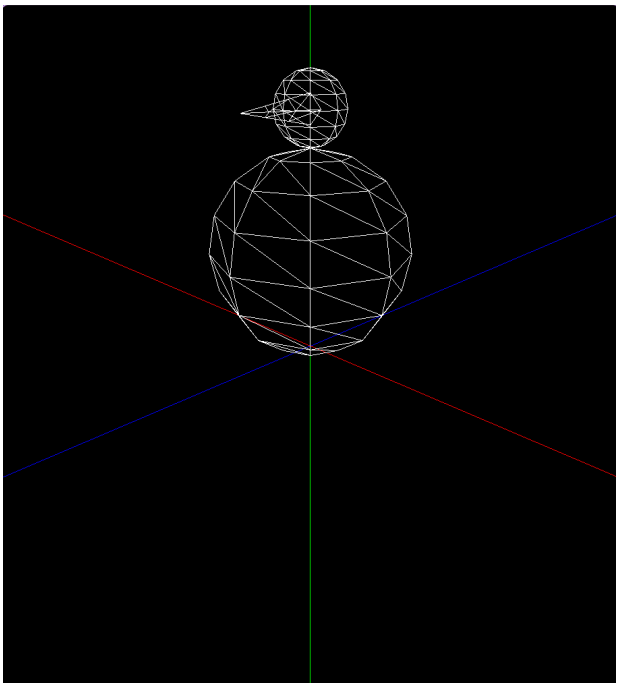
Como a escala dos corpos celestes é a real, apesar de não o ser perfeitamente a sua distância, nota-se a ocorrência de um fenómeno onde as luas são desenhadas em algumas posições e noutras não. Em primeira hipótese pensamos tratar-se de **Z-fighting** mas tal não correspondia com a ideia de estarem a disputar entre si por um dado pixel, já que todas pareciam apenas ocupar um. Não encontramos nem solução nem explicação para este acontecimento no desenvolver deste relatório.



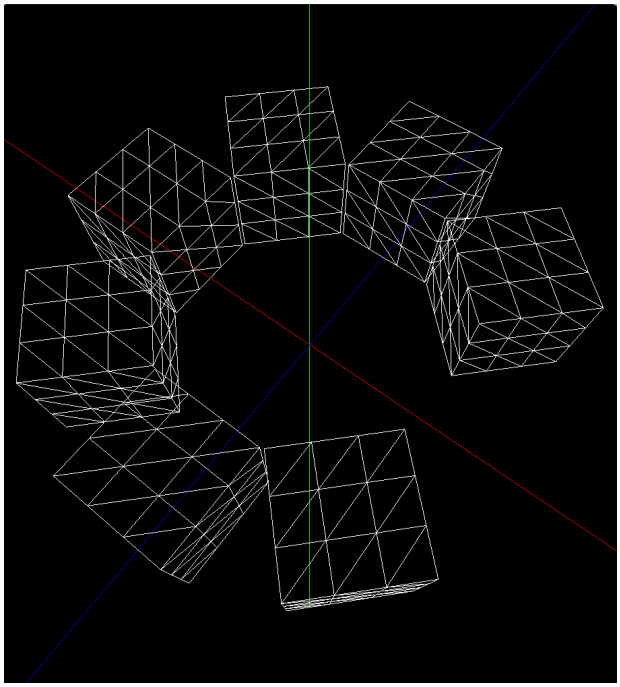
(a) Teste 1



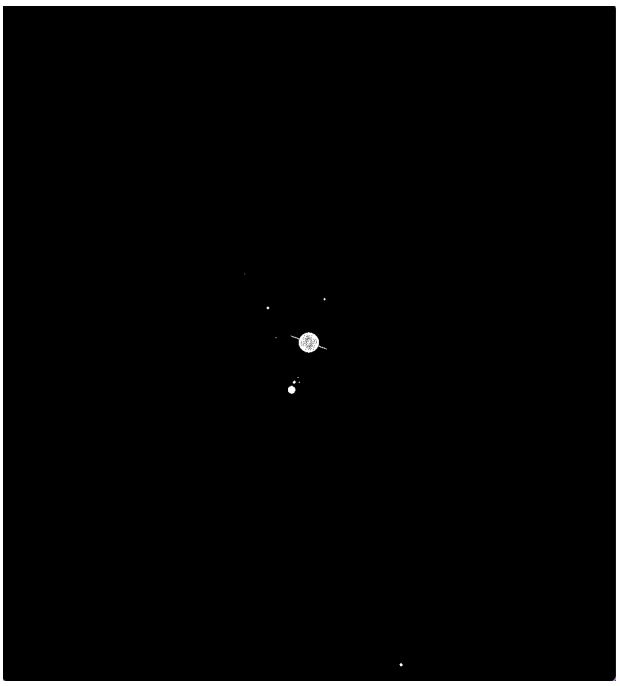
(b) Teste 2



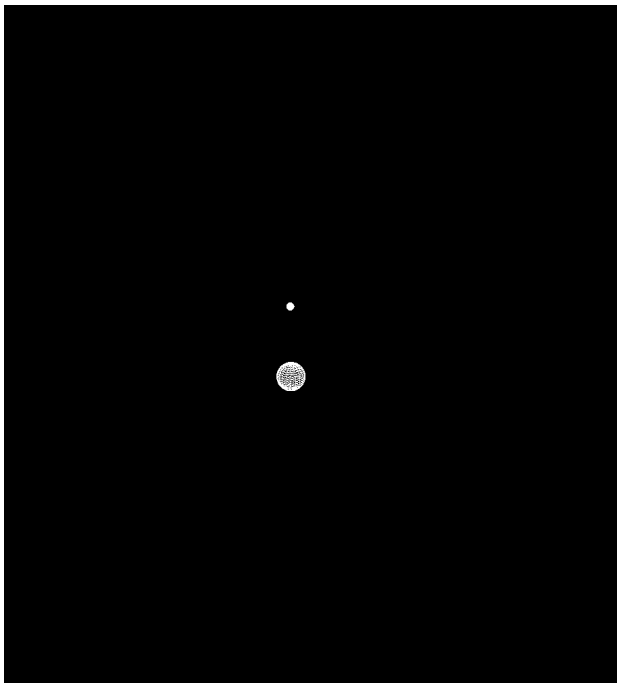
(c) Teste 3



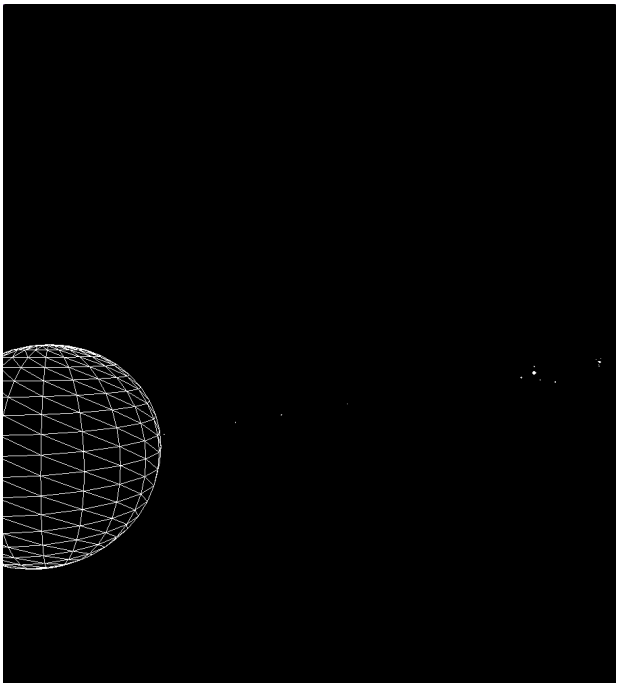
(d) Teste 4



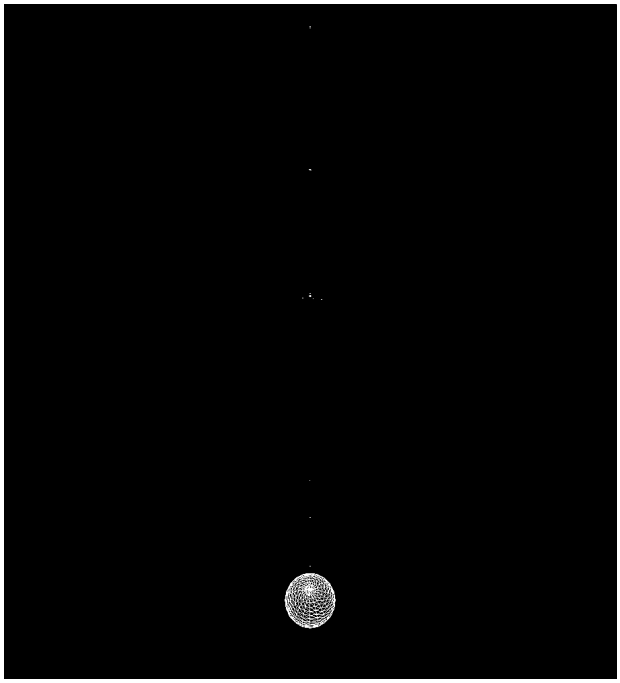
(a) Saturno



(b) Terra



(c) Sol



(d) Sistema Solar visto de cima

Capítulo 5

Conclusão

Em suma, ao longo deste relatório foi implementada lógica de grupos aninhados para codificar transformações geométricas em **xml**. Além disso, implementaram-se funcionalidades de uma câmara FPS, com movimento e rotação. Pretende-se implementar mais funcionalidades para a câmara, no futuro, nomeadamente do modo explorador em órbita de cada um dos planetas.