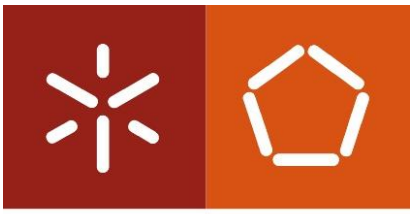


Universidade do Minho

Departamento de Informática

Relatório CG Fase 1
Grupo 31

Gustavo Lourenço a89561 João Machado a89510
Martim Almeida a89501

Ano Letivo 2020/21

Conteúdo

Capítulo 1

Introdução

Este relatório tem o objectivo de apresentar a segunda fase do Projecto da UC Computação Gráfica.

Neste relatório iremos apresentar os requisitos necessários presentes nesta fase:

- O *Generator* de ficheiros com a informação de um determinado Modelo que foi modificado de acordo com os requisitos pedidos desta fase;
- O *Engine* que inicialmente lê um ficheiro escrito em XML que contém os modelos e posteriormente faz *display* dos mesmos, e que para esta fase também foi modificada.
- Uma *Scene* que gera uma representação aproximada do nosso Sistema Solar que levou às modificações acima mencionadas e a adição de nova primitiva, uma *Torus*.

Capítulo 2

Generator

As Primitivas que são possíveis de gerar da fase anterior são:

- Plano
- Caixa
- Esfera
- Cone

Nesta fase implementamos uma nova primitiva, o Torus.

2.1 Torus

Para desenhar uma *Torus* é necessário passar como argumentos o raio inferior e superior da *Torus*, o número de *Slices* e o número de *Stacks*. Para desenhar uma *Torus*, para obter os pontos centrais das *Slices* da *Torus* começamos por calcular a média do raio inferior e superior da *Torus* e o raio que define a espessura da *Torus*. Após termos estes dois raios iteramos sobre as *Slices* e as *Stacks* e calculamos os pontos para representar duas circunferências que constituem uma secção da *Torus* após ser feita uma translação do ponto de origem para os respectivos pontos centrais da *Torus*

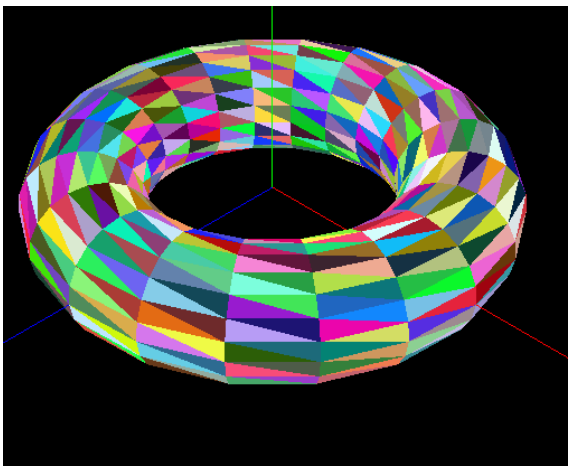


Figura 2.1: Torus desenhada pela Engine com Raio inferior = 1, Raio superior = 2, Slices = 20 e Stacks = 20

```
std::vector<Point> Torus::draw() {

    float step = 2 * M_PI / slices;
    float steph = 2 * M_PI / stacks;

    std::vector<Point> points;

    for (int i = 0; i < slices; i++) {
        //translate to the ring
        Vector center = Vector(radius*sin(step*i),0,radius*cos(step*i));
        Vector nextCenter= Vector(radius*sin(step*(i+1)),0,radius*cos(step*(i+1)));


        for (int j = 0; j < stacks; j++) {
            Point p1 = Point(SphericalPoint(step * i, steph * j, ringRadius));
            p1.add_vector(center);
            Point p2 = Point(SphericalPoint(step * i, steph * (j+1), ringRadius));
            p2.add_vector(center);
            Point p3 = Point(SphericalPoint(step * (i+1), steph * j, ringRadius));
            p3.add_vector(nextCenter);
            Point p4 = Point(SphericalPoint(step * (i+1), steph * (j+1), ringRadius
                ));
            p4.add_vector(nextCenter);

            points.push_back(p1);
            points.push_back(p3);
            points.push_back(p2);

            points.push_back(p2);
            points.push_back(p3);
            points.push_back(p4);
        }
    }
    return points;
}
```

Capítulo 3

Engine

O *Engine* foi a parte do projeto que mais mudanças sofreu. As principais mudanças feitas no *Engine* nesta fase foram:

- Foi adicionada uma câmara não estática ao *Engine*;
- Outra mudança foi a adicionação de novas funcionalidades ao *parsing* do programa.

3.1 Câmara

A câmara é baseada em coordenadas esféricas, por isso, a câmara começa apontada para um ponto com coordenadas, (x,y,z), que inicialmente é o ponto de origem, a câmara possui um raio, que inicialmente é 200 unidades, que indica a distância que existe entre a câmara e o ponto para qual esta está apontada também possui dois ângulos em radiano, α e β, que são os ângulos entre a câmara e os eixos, com os valores iniciais de 0.75 e 0.5 respectivamente, através destas variáveis conseguimos passar a coordenada da câmara para o sistema cartesiano.

3.1.1 Movimento da câmara

Para mover a câmara são utilizados as seguintes *keybinds*:

- As *keybinds* "+"e -"são utilizadas para diminuir ou aumentar a distância da câmara ao ponto observado, respectivamente.
- As *keybinds* "W"e "S"são utilizadas para aumentar ou diminuir o ângulo do β, respectivamente.
- As *keybinds* "A"e "D"são utilizadas para diminuir ou aumentar o ângulo α, respectivamente.
- A *keybind* "R"é utilizada para que os ângulos α e β, a distância da câmara ao ponto observado e a posição do ponto observado voltem a tomar os seus valores iniciais.
- As setas do teclado são utilizadas para mover o ponto que a câmara está a observar através dos eixos X e Z.

3.2 XML Parsing

Nesta fase, o ficheiro *XML* teve a sua complexidade aumentada, devido à implementação de conjuntos de Transformações Geométricas, Grupos e Modelos.

Com isso, o *Parser* realizado na fase anterior teve que ser modificado e teve que ser adicionado novas implementações de forma a satisfazer as várias opções presentes nesta fase. Assim, para esta fase, decidimos dividir o *Parser* em três tarefas:

- A leitura do ficheiro *XML*;
- O *parsing* de um ou vários Grupos, Transformações Geométricas e cores;
- O *parsing* do conjunto dos Modelos.

Para isso, dividimos o *Parser* original em três funções:

- *readFile*;
- *groupParser*;
- *modelsParser*.

3.2.1 *readFile*

A função *readFile* na fase anterior é a função responsável pelo *parse* total do ficheiro *XML*. Contudo, nesta fase, esta função serve para a leitura inicial do ficheiro *XML*, chamando de seguida a função *groupParser*.

```
void Models::readFile(char * fileName){

    tinyxml2::XMLDocument xmlDOC;
    xmlDOC.LoadFile(fileName);
    if (xmlDOC.ErrorID()){
        printf("%s\n", xmlDOC.ErrorStr());
        exit(0);
    }

    tinyxml2::XMLNode* scene = xmlDOC.FirstChildElement("scene");
    if (scene == NULL){
        printf("Scene not found.\n");
        exit(0);
    }

    *this = groupParser(scene, color);
}
```

3.2.2 *groupParser*

A função *groupParser* é o *Parser* principal, é nesta função que é verificada qual nodo do ficheiro *XML* nos encontramos, realizando posteriormente a recolha de toda a informação proveniente desse nodo. No caso de estarmos num nodo *models*, será chamada a função *modelsParser*. No caso da existência de vários grupos ou de um ou mais subgrupos dentro de um grupo, a função *groupParser* é chamada recursivamente.

```
Models Models::groupParser(tinyxml2::XMLNode* group, Color gColor){
    std::vector<Models> nGroups = std::vector<Models>();
    std::vector<Model> nModels = std::vector<Model>();
    Translate nTranslation = Translate();
    Rotate nRotation = Rotate();
    Scale nScale = Scale();
    Color nColor = gColor;

    tinyxml2::XMLNode* type = group->FirstChild();
```

```
10      while(type){
11          if(!strcmp(type->Value(), "models")){
12              std::vector<Model> auxModels = modelsParser(type);
13              for(Model m : auxModels)
14                  nModels.push_back(m);
15          }
16          else if(!strcmp(type->Value(), "model"))
17              nModels.push_back(Model(type->ToElement()->Attribute("file")));
18          else if(!strcmp(type->Value(), "translate")){
19              float x,y,z;
20              if(type->ToElement()->Attribute("X"))
21                  x = std::stof(type->ToElement()->Attribute("X"));
22              else
23                  x = 0;
24              if(type->ToElement()->Attribute("Y"))
25                  y = std::stof(type->ToElement()->Attribute("Y"));
26              else
27                  y = 0;
28              if(type->ToElement()->Attribute("Z"))
29                  z = std::stof(type->ToElement()->Attribute("Z"));
30              else
31                  z = 0;
32              nTranslation = Translate(x,y,z);
33          }
34          else if(!strcmp(type->Value(), "rotate")){
35              float angle,x,y,z;
36              if(type->ToElement()->Attribute("angle"))
37                  angle = std::stof(type->ToElement()->Attribute("angle"));
38              else
39                  angle = 0;
40              if(type->ToElement()->Attribute("axisX"))
41                  x = std::stof(type->ToElement()->Attribute("axisX"));
42              else
43                  x = 0;
44              if(type->ToElement()->Attribute("axisY"))
45                  y = std::stof(type->ToElement()->Attribute("axisY"));
46              else
47                  y = 0;
48              if(type->ToElement()->Attribute("axisZ"))
49                  z = std::stof(type->ToElement()->Attribute("axisZ"));
50              else
51                  z = 0;
52              nRotation = Rotate(angle,x,y,z);
53          }
54          else if(!strcmp(type->Value(), "scale")){
55              float x,y,z;
56              if(type->ToElement()->Attribute("X"))
57                  x = std::stof(type->ToElement()->Attribute("X"));
58              else
59                  x = 0;
60              if(type->ToElement()->Attribute("Y"))
61                  y = std::stof(type->ToElement()->Attribute("Y"));
62              else
63                  y = 0;
64              if(type->ToElement()->Attribute("Z"))
65                  z = std::stof(type->ToElement()->Attribute("Z"));
66              else
67                  z = 0;
68              nScale = Scale(x,y,z);
69          }
70          else if(!strcmp(type->Value(), "color")){
71              float r,g,b;
72              if(type->ToElement()->Attribute("R"))
73                  r = std::stof(type->ToElement()->Attribute("R"));
74              else
75                  r = 0;
76              if(type->ToElement()->Attribute("G"))
```

```
77                  g = std::stof(type->ToElement()->Attribute("G"));
78              else
79                  g = 0;
80              if(type->ToElement()->Attribute("B"))
81                  b = std::stof(type->ToElement()->Attribute("B"));
82              else
83                  b = 0;
84              nColor = Color(r,g,b);
85          }
86          else if(!strcmp(type->Value(), "group")){
87              nGroups.push_back(groupParser(type, nColor));
88          }
89          type = type->NextSibling();
90      }
91      return Models(nGroups, nModels, nTranslation, nRotation, nScale, nColor);
92  }
```

3.2.3 *modelsParser*

A função *modelsParser* é responsável pelo *parsing* do Conjunto de Modelos, sendo o seu resultado posteriormente retornado à função *groupParser*

```
1  std::vector<Model> Models::modelsParser(tinyxml2::XMLNode* models){
2      std::vector<Model> nModels = std::vector<Model>();
3      tinyxml2::XMLNode* type = models->FirstChild();
4      while(type){
5          if(!strcmp(type->Value(), "model"))
6              nModels.push_back(Model(type->ToElement()->Attribute("file")));
7          type = type->NextSibling();
8      }
9      return nModels;
10 }
```

3.2.4 Desenho do resultado do *Parsing*

Após a execução do *Parser* terminar, a função *drawModels* é responsável pela chamada das várias funções de desenho do *Engine* presente no trabalho.

```
1  void Models::drawModels(){
2      translation.transform();
3      rotation.transform();
4      scale.transform();
5      color.transform();
6      for(Model m: models)
7          m.drawModel();
8      for(Models ms: groups){
9          glPushMatrix();
10         ms.drawModels();
11         glPopMatrix();
12     }
13 }
```

Capítulo 4

Sistema Solar

Para criar o Ficheiro do XML do sistema solar era necessário simular as translações dos planetas em torno do Sol e representar um grande número de luas, devido as estes factores sentimos a necessidade de usar um *script* para facilitar a criação do Ficheiro de XML, para tal foi feito um *script* em *Python*.

Para a realização deste modelo nós utilizamos as transformações geométricas implementadas nesta fase, da seguinte maneira:

- *Rotate*: foi utilizado para rodar o anel de Saturno;
- *Translate*: foi utilizado para simular a translação dos planetas em torno do Sol, e das luas em torno dos seus respectivos planetas;
- *Scale*: foi utilizado para gerar corpos celestes de tamanhos diferentes, e gerar uma *Torus* mais consistem com um disco para simular o anel de Saturno;
- *Color*: foi utilizado para gerar corpos celestes de cores diferentes;
- *Group*: foi utilizado para agrupar os planetas com as suas luas e no caso de Saturno o seu anel.

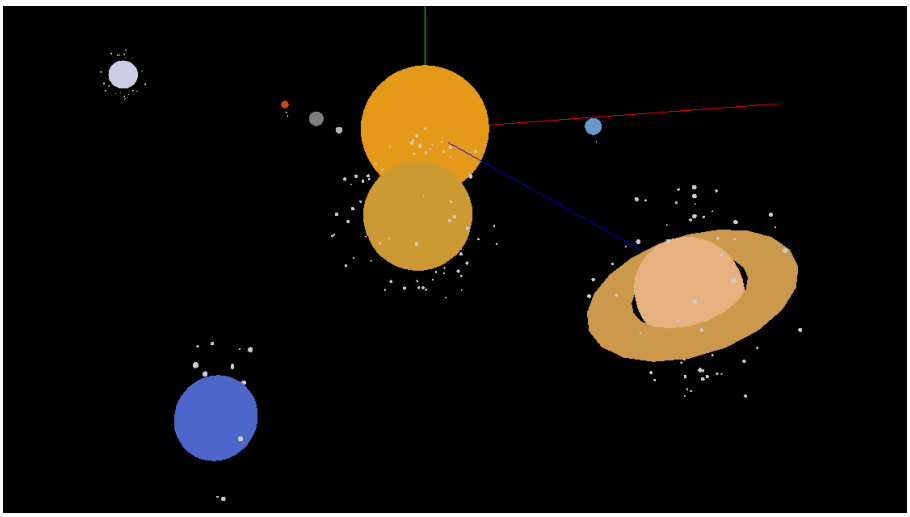
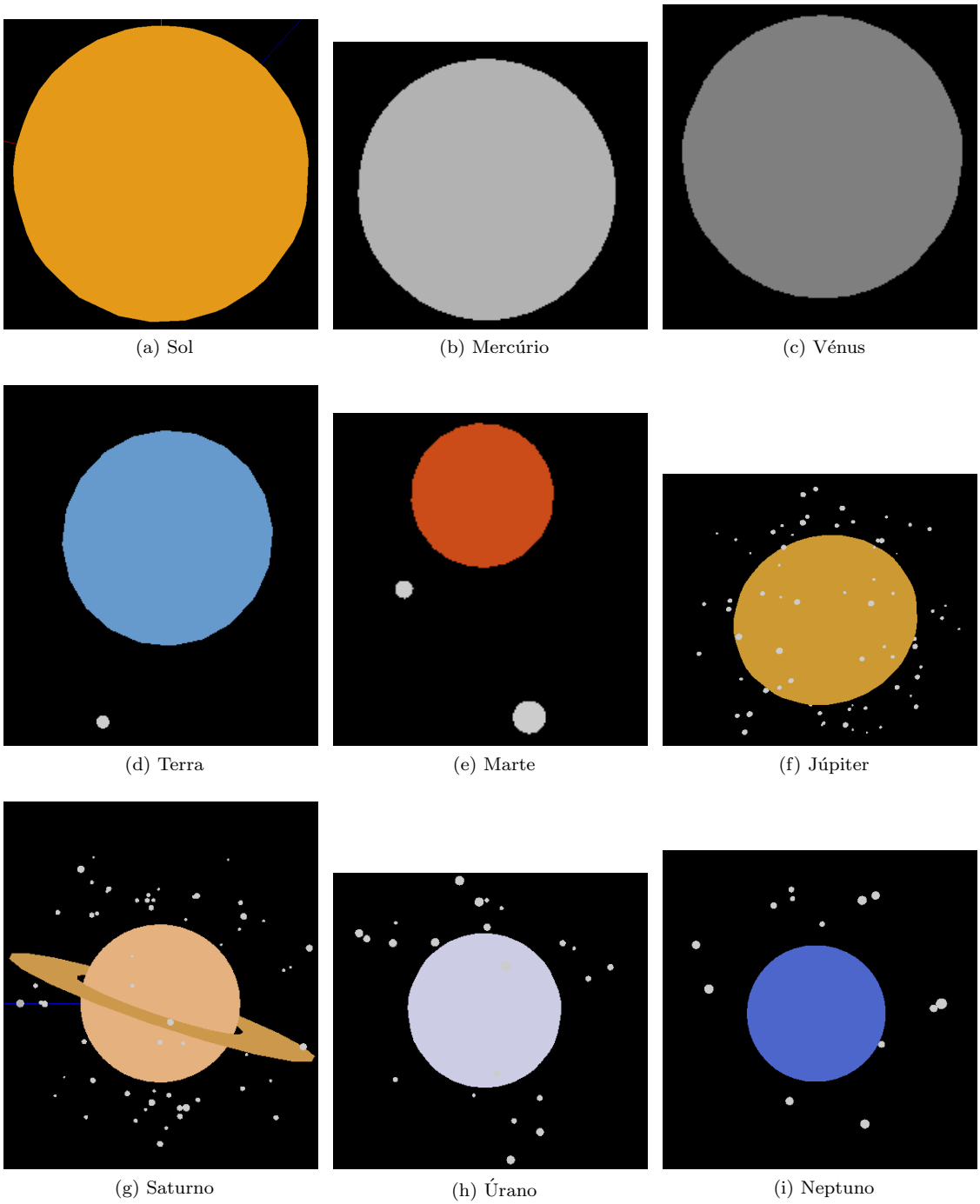


Figura 4.1: Sistema Solar



(a) Sol (b) Mercúrio (c) Vénus

(d) Terra (e) Marte (f) Júpiter

(g) Saturno (h) Úrano (i) Neptuno

Figura 4.2: Tabela dos Corpos Celestes

Capítulo 5

Conclusão

Com esta Fase aprendemos como aplicar diversos conteúdos leccionados nas aulas como a uso e implementação de transformações geométricas. Também aprofundamos ainda mais os conhecimentos da linguagem C++ assim como *CMakeLists*. Também completamos alguns dos objectivos por nós propostos na Fase anterior, como a implementação de uma câmara.

Futuramente, gostaríamos de implementar uma optimização para que o projecto funcione em todos os Sistemas Operativos assim como o uso de diversas directorias de forma a melhorar organizar o nosso projecto, também gostaríamos a forma como damos cor aos objectos, criando objectos com cores e padrões variados.