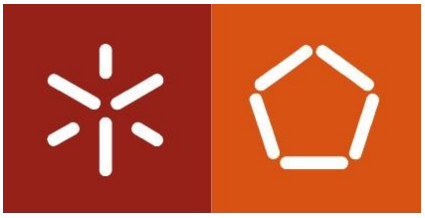

Universidade do Minho

Departamento de Informática

Computação Gráfica
Grupo 45
Trabalho Prático
Fase 2

6 de maio de 2021

Ariana Lousada (A87998)

Rui Armada (A90468)

Carolina Vila Chã (A89495)

Sofia Santos (A89615)

Conteúdo

Capítulo 1

Introdução

O principal objetivo desta segunda fase do trabalho prático consiste atualizar o *engine* concebido na fase anterior de modo a ser possível aplicar transformações, nomeadamente translações, rotações e mudança de escala, aos modelos que constituem as nossas cenas.

Para tal, é necessário também alterar a estrutura dos nossos ficheiros XML, de modo a serem compatíveis com estas novas operações. Implementamos também um sistema hierárquico nos mesmos, onde cada nodo ("grupo") pode conter um conjunto de tranformações e modelos, para além de possíveis sub-nodos com as suas próprias transformações e modelos. Para além das suas próprias transformações, os sub-grupos herdam as transformações do seu grupo mãe.

Para mostrar o correto funcionamento do nosso *engine* após esta fase, criámos um modelo estático do Sistema Solar, que tira proveito destas novas funcionalidades.

Capítulo 2

Desenvolvimento

2.1 Transformações

De modo a transformar as figuras corretamente, pensámos em duas possíveis implementações.

A primeira consistiria em guardar, para cada grupo, o conjunto de transformações a aplicar, numa estrutura como uma lista com vários valores, mais especificamente o valor a transladar em cada eixo, o ângulo de rotação e os eixos da mesma e os valores de escala para cada eixo. O problema que encontrámos com este método é que não permitia reconhecer as ordens das transformações, por isso teríamos que usar outra estrutura, possivelmente bastante complexa, para armazenar esta ordem. Com este problema à vista, esta primeira implementação revelou-se difícil de implementar e com um possível problema de memória, caso tivéssemos cenas com bastantes níveis de sub-grupos e transformações, visto que cada um precisaria de ter estas estruturas.

Decidimos assim usar a nossa segunda implementação, que consiste em usar uma matriz para cada grupo ou sub-grupo. Esta matriz contém as transformações a aplicar aos modelos de cada grupo. Por outras palavras, para cada transformação que um grupo contém, a matriz dessa transformação é multiplicada pela matriz anterior, que no caso de sub-grupos é a matriz do grupo mãe, e no caso do grupo "principal"é a matriz identidade. Assim, somos capazes de preservar a ordem das transformações, apenas temos de multiplicar as matrizes das mesmas pela ordem correta. .

- Matriz de translação

```
double trans[16] = {1,0,0,0,
                    0,1,0,0,
                    0,0,1,0,
                    x,y,z,1};
```

- Matriz de rotação

```
double rotate[16] =
{x^2*(1-c)+c   ,x*y*(1-c)+z*s, x*z*(1-c)-y*s, 0,
```

```
x*y*(1-c)-z*s,y^2*(1-c)+c   , y*z*(1-c)+x*s, 0,
x*z*(1-c)+y*s,y*z*(1-c)-x*s, z^2*(1-c)+c   , 0,
0             , 0           , 0             , 1};
```

Nesta matriz, $s = sin(\alpha)$ e $c = cos(\alpha)$, sendo α o ângulo da rotação, e x, y e z são as coordenadas do vetor de rotação.

- Matriz de escala

```
double trans[16] = {x,0,0,0,
                    0,y,0,0,
                    0,0,z,0,
                    0,0,0,1};
```

Para além destas matrizes, desenvolvemos ainda uma simples função para multiplicar matrizes. Assim, apenas temos de armazenar uma matriz em memória por grupo.

A forma como fazemos esse armazenamento é através de um struct.

```
struct group {
    std::vector<std::vector<Point>> solids;
    GLdouble matrix[16];
    int colors[3];
};
```

Para além de uma lista com os modelos semelhante à da fase anterior e da matriz das transformações, temos ainda um pequeno array que usamos para implementar cores no nosso *engine*, que armazena três valores inteiros entre 0 e 255 em formato RGB.

O código responsável por desenhar os grupos encontra-se na função `renderScene()` e é o seguinte:

```
for (unsigned long group = 0; group < scene.size(); group++) {
    std::vector<std::vector<Point>> solids = scene[group].solids;
    for (unsigned long i = 0; i < solids.size(); i++) {
        glPushMatrix();
        std::vector<Point> solid = solids[i];
        glMultMatrixd(scene[group].matrix);
        glColor3ub(scene[group].colors[0],scene[group].colors[1],scene[
    group].colors[2]);
        glBegin(GL_TRIANGLES);
            for (unsigned long j = 0; j < solid.size(); j++)
                glVertex3d(solid[j].x(), solid[j].y(), solid[j].z());
        glEnd();
        glPopMatrix();
    }
}
```

Usamos a função `glMultMatrixd()` do GLUT para multiplicar a nossa matriz pela matriz *modelview* selecionada.

2.2 Parsing XML

De modo a fazer *parse* dos ficheiros XML criados nesta segunda fase do nossa projeto, desenvolvemos a função `parseGroup()`:

```
void parseGroup(XMLElement* group, GLdouble * matrix, int * colors) {
    bool hasTrans = false, hasRotate = false, hasScale = false, hasModels
     = false;
    [...]
    XMLElement* elem = group->FirstChildElement();
    while(elem) {
        if(!strcmp(elem->Name(), "models")) {
            if(!hasModels) hasModels = true;
            else {
                puts("Error - too many models in the same group.");
                exit(1);
            }
            [...]
        }
        else if(!strcmp(elem->Name(), "translate")) {
            if(!hasTrans) hasTrans = true;
            else {
                puts("Error - too many transformations applied to the
    same group.");
                exit(1);
            }
            [...]
        }
        else if(!strcmp(elem->Name(), "rotate")) {
            if(!hasRotate) hasRotate = true;
            else {
                puts("Error - too many rotations applied to the same
    group.");
                exit(1);
            }
            [...]
        }
        else if(!strcmp(elem->Name(), "scale")) {
            if(!hasScale) hasScale = true;
            else {
                puts("Error - too many scales applied to the same group."
    );
                exit(1);
            }
            [...]
        }
        else if(!strcmp(elem->Name(), "color")) {
            [...]
        }
        else if(!strcmp(elem->Name(), "group")) {
            parseGroup(elem, matrix, colors);
        }
        elem = elem->NextSiblingElement();
```

```
45      }
46
47      if(hasModels) {
48          struct group x;
49          x.solids = solids;
50          memcpy(&x.colors, colors, 3 * sizeof(int));
51          memcpy(&x.matrix, matrix, 16 * sizeof(double));
52
53          scene.push_back(x);
54      }
55  }
```

Por uma questão de organização, apenas incluímos no relatório as partes da função relativas ao *parsing* dos ficheiros XML. A função completa pode ser consultada no nosso código.

Uma vez que neste caso, ao contrário da primeira fase, temos vários grupos e sub-grupos diferentes num mesmo ficheiro, cada um com várias possíveis transformações e modelos, não podemos simplesmente reutilizar o *parser* desenvolvido na fase anterior. Deste modo, com esta nova função conseguimos fazer *parsing* de uma forma recursiva e cobrindo todas as novas características presentes nos ficheiros XML.

2.3 Extra: Câmara FPS

Para além do que foi solicitado, desenvolvemos ainda uma câmara semelhante à usada por jogos na primeira pessoa, como FPSs, para sermos capazes de visualizar o nosso modelo do sistema solar de forma mais prática.

É possível trocar entre o modo explorador e o modo FPS usando a tecla 'v'. Dentro do modo FPS podemos usar as teclas WASD para mover a câmara, e premindo o botão esquerdo do rato, podemos ainda usar o movimento do cursor para mudar a direção da câmara.

Como não é uma funcionalidade requerida nesta fase, não iremos incluir o seu código neste relatório, mas é possível consultá-lo no nosso ficheiro `engine.cpp`. Iremos apenas referir que usámos a seguinte equação:

$$P' = P + k \times D$$

onde P é a posição antiga da câmara, P' a nova posição da câmara, D o vetor de deslocamento e k a sensibilidade do deslocamento para implementar o deslocamento da câmara. Para implementar a rotação da câmara usamos dois ângulos, calculados a partir da posição do rato, que modificam o vetor D.

2.4 Torus

De modo a conseguirmos desenhar o anel de Saturno, acrescentámos uma figura ao *generator* desenvolvido na primeira fase, o *Torus*. A função `drawTorus()` é a responsável pela geração dos vértices desta mesma figura.

Depois de aplicar esta figura no sistema solar, com um valor de escala vertical igual a 0 e uma ligeira rotação no eixo z, obtivemos o seguinte resultado:

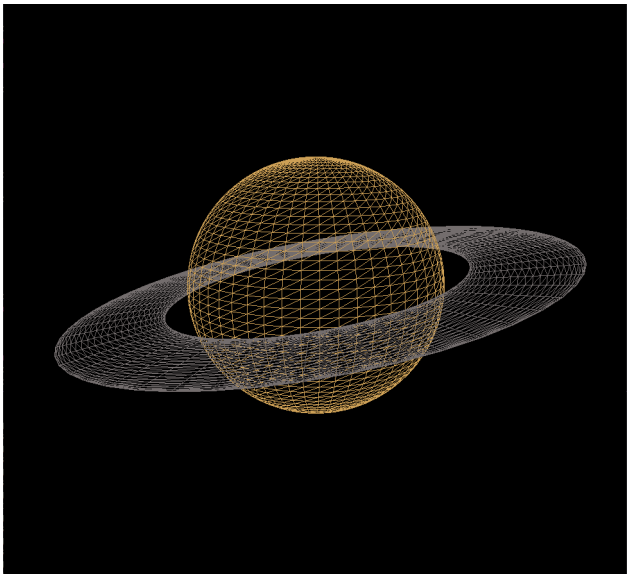

Figura 2.1: Planeta Saturno no modelo inicial do Sistema Solar construído.

2.5 Modelo inicial - Sistema Solar

N.B.: As distâncias entre os planetas são bastante mais reduzidas no nosso modelo do que na realidade, de modo a facillitar a visualização do sistema solar completo. Tentámos ainda preservar os tamanhos de cada corpo celeste, mas tivemos de reduzir o tamanho do Sol e dos planetas gasosos e aumentar o tamanho das luas de Marte pela mesma razão.

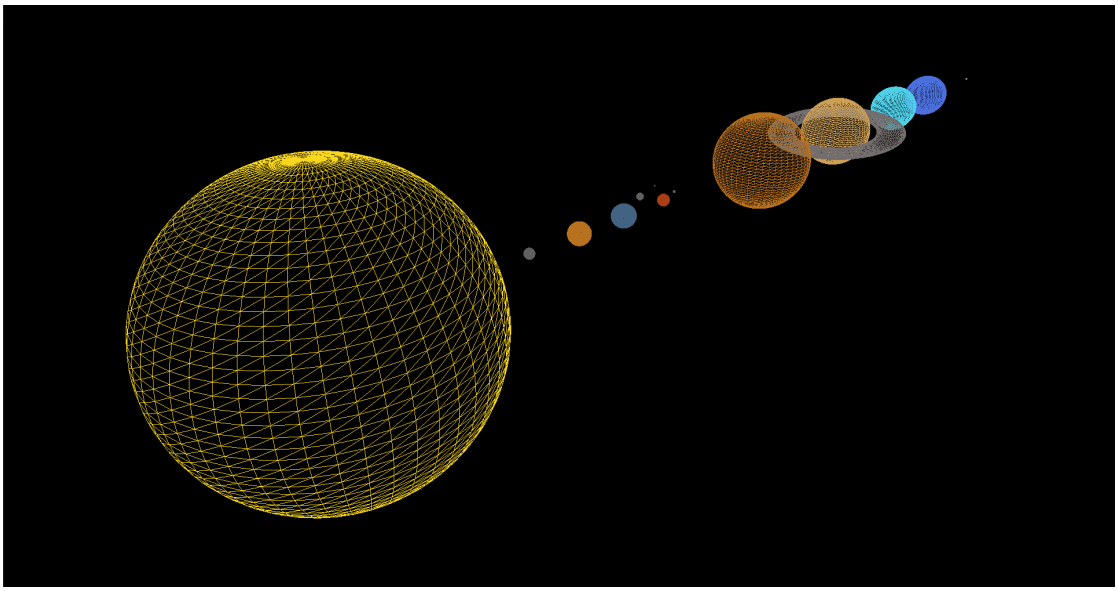

Figura 2.2: Sistema Solar completo

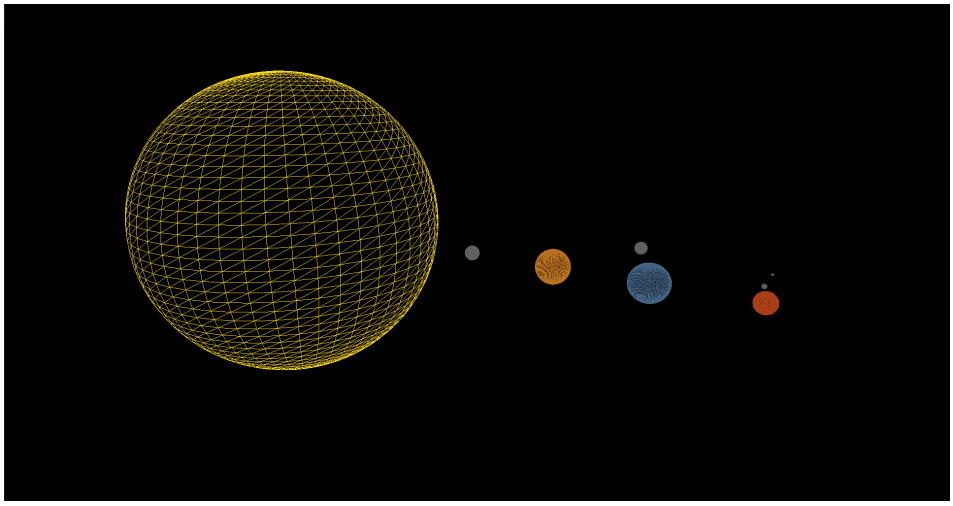

Figura 2.3: Sol e planetas telúricos, com as suas respetivas luas.

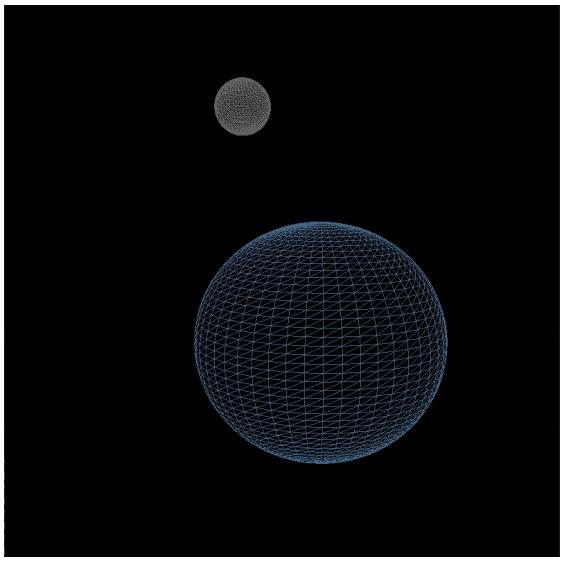

Figura 2.4: Terra e Lua.

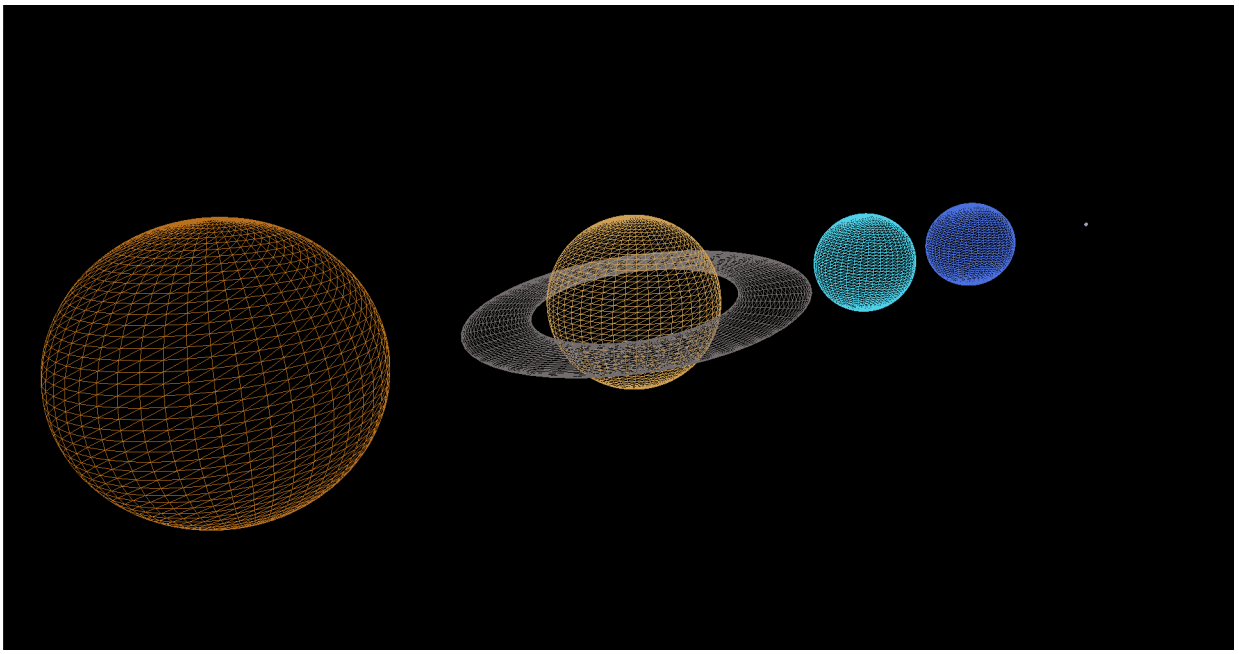

Figura 2.5: Planetas gasosos e Plutão.

Capítulo 3

Conclusão

A realização desta fase do projeto permitiu-nos consolidar os conhecimentos adquiridos durante as aulas acerca de transformações de modelos 3D, matrizes e câmaras. Com isto, fomos capazes de contruir um modelo inicial do sistema solar, que vai ser necessário nas próximas fases deste trabalho prático.