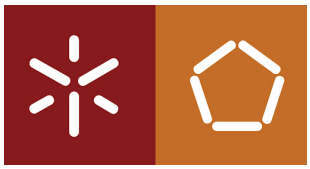
Universidade do Minho

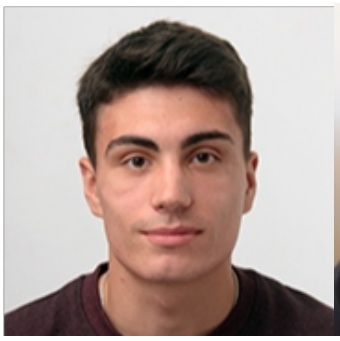
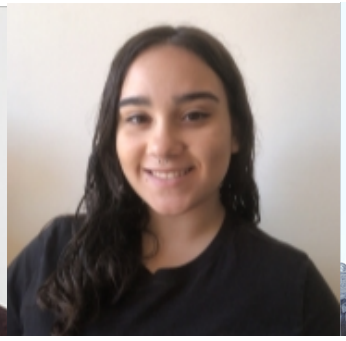
Departamento de Informática



licenciatura em engenharia informática

Computação gráfica - 4ª fase

Grupo 38

Henrique Pereira | Mariana Morais | Ana Alves | Simão Antunes

Henrique Morais Pereira A100831
Mariana Filipa Morais Gonçalves A100662
Ana João da Rocha Alves A95128
Simão Pedro Ferreira Antunes A100597

Maio de 2024

Conteúdo

1 Introdução

A unidade curricular de Computação Gráfica propõe, recorrendo à biblioteca GLUT, a utilização do OpenGL para a construção de modelos 3D com recurso à linguagem de programação C++. Pretende-se, assim, que o relatório sirva de suporte ao trabalho realizado para esta fase, mais propriamente, dando uma explicação e elucidando o conjunto de decisões tomadas ao longo da construção de todo o código fonte e descrevendo a estratégia utilizada para a concretização dos principais objetivos propostos.

Posto isto, temos de forma muito breve o assunto do presente relatório que se refere à última de um projeto dividido em 4 fases, tal como indicado pela equipa docente.

2 Descrição do Problema

2.1 Fase 4

Nesta fase, o generator deve gerar coordenadas de textura e normais para cada vértice. Na engine, devemos ativar as funcionalidades de iluminação e textura, bem como ler e aplicar as normais e coordenadas de textura dos arquivos de modelo. Além disso, o XML deve permitir a definição dos componentes de cor difusa, especular, emissiva e ambiente, bem como o brilho. Também deve ser possível definir as fontes de luz no arquivo XML.

2.1.1 Generator

O *Generator* agora deve:

- Gerar coordenadas de textura e normais para cada vértice.
- Escrever os triângulos resultantes, juntamente com suas coordenadas de textura e normais, num arquivo.

2.1.2 Engine

A *Engine* agora deve:

- Ativar as funcionalidades de iluminação e textura.
- Ler e aplicar as normais e coordenadas de textura dos arquivos de modelo.
- Permitir a definição dos componentes de cor difusa, especular, emissiva e ambiente, bem como o brilho, no arquivo XML.
- Permitir a definição das fontes de luz no arquivo XML.

3 Resolução do Problema

3.1 Generator

De forma a ser possível colocar iluminação e aplicar texturas nas primitivas é necessário modificar o gerador desenvolvido anteriormente. Para isto, deve-se calcular as normais e as coordenadas de textura de cada ponto para cada primitiva, e escrevê-las no respetivo ficheiro.

3.1.1 Normas

- **Esfera:** Para cada ponto da esfera vamos definir o seu vetor normal, perpendicular à superfície. Este tem a mesma direção do ponto. Assim, tendo em conta a fórmula para o cálculo das coordenadas do ponto, vamos calcular a normal, sendo que a única diferença e o facto de não multiplicarmos o valor do raio (vetor tem dimensão 1).

$$\begin{aligned} x &= r \cdot \sin(\beta) \cdot \cos(\alpha) \\ y &= r \cdot \cos(\beta) \\ z &= r \cdot \sin(\beta) \cdot \sin(\alpha) \end{aligned}$$

$$\begin{aligned} nx &= \sin(\beta) \cdot \cos(\alpha) \\ ny &= \cos(\beta) \\ nz &= \sin(\beta) \cdot \sin(\alpha) \end{aligned}$$

- **Plano:** Um ponto P num plano é definido por duas coordenadas: x e z. Os pontos no plano são calculados da seguinte maneira:

$$\begin{aligned} x &= -\frac{\text{length}}{2} + i \cdot \text{spacing} \\ y &= 0 \\ z &= \frac{\text{length}}{2} + j \cdot \text{spacing} \end{aligned}$$

  onde spacing $= \frac{\text{length}}{\text{divisions}}$, i é o índice da divisão atual no eixo x e j é o índice da divisão atual no eixo z.

  As normais são calculadas como o produto vetorial de dois vetores definidos pelos pontos adjacentes no plano. O produto vetorial de dois vetores é um vetor que é perpendicular a ambos. Como todos os pontos no plano têm $y = 0$, todas as normais apontam na direção y

- **Caixa:** As coordenadas dos pontos na caixa são calculadas da seguinte maneira:

$$\begin{aligned} x &= -\frac{\text{length}}{2} + i \cdot \text{spacing} \\ y &= \pm\frac{\text{length}}{2} \quad \text{(dependendo do lado da caixa)} \\ z &= -\frac{\text{length}}{2} + j \cdot \text{spacing} \end{aligned}$$

  onde spacing $= \frac{\text{length}}{\text{divisions}}$, i é o índice da divisão atual no eixo x e j é o índice da divisão atual no eixo z.

As normais para um plano são constantes, pois a superfície é plana. As normais para cada lado da caixa são calculadas da seguinte forma:

$$\begin{aligned} nx &= 0 \\ ny &= \pm 1 \quad \text{(para o topo e a base da caixa)} \\ nz &= 0 \end{aligned}$$

$$\begin{aligned} nx &= \pm 1 \quad \text{(para os lados esquerdo e direito da caixa)} \\ ny &= 0 \\ nz &= 0 \end{aligned}$$

$$\begin{aligned} nx &= 0 \\ ny &= 0 \\ nz &= \pm 1 \quad \text{(para a frente e a parte de trás da caixa)} \end{aligned}$$

- **Cone:** Um ponto P na superfície de um cone é definido por três parâmetros: o raio r, a altura h, e o ângulo α. A relação entre as coordenadas cilíndricas e as coordenadas cartesianas é dada pelas seguintes equações:

$$\begin{aligned} x &= r \cdot \sin(\alpha) \\ y &= h \\ z &= r \cdot \cos(\alpha) \end{aligned}$$

Onde r é o raio do cone na altura atual, h é a altura atual do cone, e α é o ângulo no plano XY (varia de 0 a 2π).

As normais para um cone são calculadas usando as seguintes fórmulas:

$$\begin{aligned} nx &= r \cdot \sin(\alpha) \\ ny &= \sin(\text{atan}(r/h)) \\ nz &= r \cdot \cos(\alpha) \end{aligned}$$

Onde r é o raio do cone na altura atual, h é a altura do cone, e α é o ângulo no plano XY.

- **Torus:** Um ponto P na superfície de um torus é definido por três parâmetros: o raio interno r_{int}, o raio externo r_{ext}, e os ângulos α e β. A relação entre as coordenadas cilíndricas e as coordenadas cartesianas é dada pelas seguintes equações:

$$\begin{aligned} x &= (r_{\text{ext}} + r_{\text{int}} \cdot \cos(\beta)) \cdot \cos(\alpha) \\ y &= r_{\text{int}} \cdot \sin(\beta) \\ z &= (r_{\text{ext}} + r_{\text{int}} \cdot \cos(\beta)) \cdot \sin(\alpha) \end{aligned}$$

Onde r_{ext} é o raio externo do torus, r_{int} é o raio interno do torus, α é o ângulo no plano XY (varia de 0 a 2π), e β é o ângulo no plano do círculo interno do torus (varia de 0 a 2π).

As normais para um torus são calculadas usando as seguintes fórmulas:

$$\begin{aligned} nx &= -\cos(\beta) \cdot \cos(\alpha) \\ ny &= -\sin(\beta) \\ nz &= -\cos(\beta) \cdot \sin(\alpha) \end{aligned}$$

- **Bezier Patch:**

  Um ponto P em um Bezier Patch é definido por dois parâmetros: u e v. A relação entre as coordenadas paramétricas e as coordenadas cartesianas é dada pela seguinte equação:

$$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i(u) \cdot B_j(v) \cdot P_{ij}$$

  onde $B_i(u)$ e $B_j(v)$ são os polinómios de Bernstein de grau 3, e P_{ij} são os pontos de controle do patch de Bézier.

  As normais para um Bezier Patchr são calculadas como o produto vetorial das derivadas parciais da superfície de Bezier em relação a u e v. As derivadas parciais são calculadas da seguinte maneira:

$$\frac{\partial P}{\partial u} = \sum_{i=0}^{3} \sum_{j=0}^{3} \frac{dB_i(u)}{du} \cdot B_j(v) \cdot P_{ij}$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i(u) \cdot \frac{dB_j(v)}{dv} \cdot P_{ij}$$

  onde $\frac{dB_i(u)}{du}$ e $\frac{dB_j(v)}{dv}$ são as derivadas das funções de base de Bernstein.
  A normal N em um ponto $P(u,v)$ é então calculada como:

$$N(u,v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

  onde $\times$ denota o produto vetorial.

3.1.2 Coordenadas de textura

- **Esfera:** Para uma esfera, as coordenadas de textura são normalmente calculadas com base nos ângulos azimutal α e polar β. As coordenadas de textura são calculadas da seguinte maneira:

$$\begin{aligned} tx &= \frac{i}{\text{slices}} \\ tz &= \frac{j}{\text{stacks}} \end{aligned}$$

  Onde: - i é o índice da fatia atual, que varia de 0 a slices. - j é o índice da pilha atual, que varia de 0 a stacks.

  Portanto, tx e tz são as coordenadas de textura u e v, respectivamente. Eles variam de 0 a 1 à medida que i e j variam de 0 a slices e stacks, respectivamente.

- **Plano:** Para um plano, as coordenadas de textura são normalmente calculadas com base na posição do ponto no plano. As coordenadas de textura são calculadas da seguinte maneira:

$$tx = \frac{i}{\text{divisions}}$$
$$tz = \frac{j}{\text{divisions}}$$

Onde i é o índice da divisão atual no eixo x e j é o índice da divisão atual no eixo z. Portanto, tx e tz são as coordenadas de textura u e v, respectivamente. Eles variam de 0 a 1 à medida que i e j variam de 0 a divisions.

- **Caixa:** As coordenadas de textura são calculadas da seguinte maneira:

$$tx = \frac{i}{\text{divisions}}$$
$$tz = \frac{j}{\text{divisions}}$$

Onde i é o índice da divisão atual no eixo x e j é o índice da divisão atual no eixo z. Portanto, tx e tz são as coordenadas de textura u e v, respectivamente. Eles variam de 0 a 1 à medida que i e j variam de 0 a divisions.

- **Cone:** As coordenadas de textura são calculadas da seguinte maneira:

$$tx = \frac{j}{\text{slices}}$$
$$tz = \frac{i}{\text{stacks}}$$

Onde j é o índice da fatia atual no eixo XY e i é o índice da pilha atual no eixo Z. Portanto, tx e tz são as coordenadas de textura u e v, respectivamente. Eles variam de 0 a 1 à medida que j e i variam de 0 a slices e stacks, respectivamente.

- **Torus:**

Para um torus, as coordenadas de textura são calculadas com base na posição do ponto na superfície do torus. As coordenadas de textura são calculadas da seguinte maneira:

$$tx = \frac{i}{\text{slices}}$$
$$tz = \frac{j}{\text{stacks}}$$

Onde i é o índice da fatia atual no eixo XY e j é o índice da pilha atual no eixo do círculo interno do torus. Portanto, tx e tz são as coordenadas de textura u e v, respectivamente. Eles variam de 0 a 1 à medida que i e j variam de 0 a slices e stacks, respectivamente.

- **Bezier Patch:**

  As coordenadas de textura são normalmente representadas como um par de valores (u, v), onde u e v variam de 0 a 1.

  Para um Bezier Patch, as coordenadas de textura são calculadas com base na posição do ponto no patch. As coordenadas de textura são calculadas da seguinte maneira:

$$tx = u$$

$$tz = v$$

3.2 Engine

3.2.1 Parsing dos ficheiros XML

Para a realização do parsing dos novos ficheiros XML foi necessário criar estruturas para guadar a informação relevante dos mesmos. Para tal foram criadas três estruturas novas e outras duas sofreram alterações significativas: Light, Color e Model.

```
struct Light {
    string type;
    float position[4]  = {0.0f, 0.0f, 0.0f, 1.0f};
    float direction[4] = {0.0f, 0.0f, 0.0f, 0.0f};
    float cutoff = 0.0f;
};
```

Figura 1: Struct Light

Primeiramente começamos por definir uma estrutura Light capaz de acomodar os 3 tipos de luz, com uma *string type* para fácil identificação.

```
struct Color {
    float diffuse[4]  = {0.8f, 0.8f, 0.8f, 1.0f};
    float ambient[4]  = {0.2f, 0.2f, 0.2f, 1.0f};
    float specular[4] = {0.0f, 0.0f, 0.0f, 1.0f};
    float emissive[4] = {0.0f, 0.0f, 0.0f, 1.0f};
    float shininess = 0.0f;
};
```

Figura 2: Struct Color

Para representar uma cor criamos uma estrutura Color, esta já preenchida para o caso do ficheiro XML não especificar uma cor.

```
// Structure to store a model
struct Model {
    string name;
    int vertexCount;
    int indexBegin;
    int textureID;
    Color color;
};
```

Figura 3: Struct Model

Por uma questão de organização decidimos criar uma estrutura Model onde guardamos o nome do mesmo, o número de pontos necessário para o desenhar, o índice do VBO, o id de textura e a cor.

```
// Structure to store a xml group
struct Group {
    vector<Transformation> transformations; // Geo
    vector<Model> models;                   // Mod
    vector<Group> subGroups;                // Sub
};

struct World {
    Window *win = new Window;
    Camera *cam = new Camera;
    vector<Light> lights;
    vector<Group> groups;
};
```

Figura 4: Struct Group e World

Como criamos uma estrutura Model que guarda o índice do VBO e o número de pontos necessários para o desenhar, foi necessário modifcar a estrutura Group para acomodar um vetor de modelos em vez do vetor de pares. Já na estrutura World foi preciso a adição de um vetor de luzes uma vez que estas não dependem de um grupo.

Posto isto passamos para o parsing dos ficheiros XML. Para tal tivemos de estender as funcionalidades já implementadas para poder aplicar as novas funcionalidades. Assim, definimos as variáveis globais *vertexPoints* para guardar os vértices lidos, *normalPoints* para guardar as normais lidas, *texturePoints* para guardar as coordenadas de textura e *textureIDs* e criamos as funções *loadTexture*, *parseLights*, *parseColor*.

3.2.2 Cor

```
Color parseColor(XMLElement * colorElement) {
    Color color;
    (...)
    if (diffuseElement) {
        float r, g, b;
        diffuseElement->QueryFloatAttribute("R", &r);
        diffuseElement->QueryFloatAttribute("G", &g);
        diffuseElement->QueryFloatAttribute("B", &b);
        color.diffuse[0] = r / 255.0f;
        color.diffuse[1] = g / 255.0f;
        color.diffuse[2] = b / 255.0f;
    }
    if (ambientElement) {
        float r, g, b;
        ambientElement->QueryFloatAttribute("R", &r);
        ambientElement->QueryFloatAttribute("G", &g);
        ambientElement->QueryFloatAttribute("B", &b);
        color.ambient[0] = r / 255.0f;
        color.ambient[1] = g / 255.0f;
        color.ambient[2] = b / 255.0f;
    }
    if (specularElement) {
        float r, g, b;
        specularElement->QueryFloatAttribute("R", &r);
        specularElement->QueryFloatAttribute("G", &g);
        specularElement->QueryFloatAttribute("B", &b);
```

```
        color.specular[0] = r / 255.0f;
        color.specular[1] = g / 255.0f;
        color.specular[2] = b / 255.0f;
    }
    if (emissiveElement) {
        float r, g, b;
        emissiveElement->QueryFloatAttribute("R", &r);
        emissiveElement->QueryFloatAttribute("G", &g);
        emissiveElement->QueryFloatAttribute("B", &b);
        color.emissive[0] = r / 255.0f;
        color.emissive[1] = g / 255.0f;
        color.emissive[2] = b / 255.0f;
    }
    if (shininessElement) {
        shininessElement->QueryFloatAttribute("value", &color.shininess);
    }
    return color;
}



void drawGroups(vector<Group> groups) {
    for(const Group& g : groups){
        (...)
        for (Model m : g.models) {
            glMaterialfv(GL_FRONT, GL_DIFFUSE, m.color.diffuse);
            glMaterialfv(GL_FRONT, GL_AMBIENT, m.color.ambient);
            glMaterialfv(GL_FRONT, GL_SPECULAR, m.color.specular);
            glMaterialfv(GL_FRONT, GL_EMISSION, m.color.emissive);
            glMaterialf(GL_FRONT, GL_SHININESS, m.color.shininess);
            (...)
        }
        drawGroups(g.subGroups);
        glPopMatrix();
    }
}
```

A cor de um modelo é lida através do ficheiro XML através da parseColor e posteriormente aplicada na *drawGroups*.

3.2.3 Iluminação

```
vector<Light> parseLights(XMLElement * lightsElement) {
    vector<Light> lights;
    XMLElement * lightElement = lightsElement -> FirstChildElement("light");
    while (lightElement) {
        Light l;
        const char * type = lightElement -> Attribute("type");
        l.type = type;
        if (!strcmp(type, "point"))  {
            lightElement->QueryFloatAttribute("posx", &l.position[0]);
            lightElement->QueryFloatAttribute("posy", &l.position[1]);
            lightElement->QueryFloatAttribute("posz", &l.position[2]);
        }
        if (!strcmp(type, "directional")) {
            lightElement->QueryFloatAttribute("dirx", &l.direction[0]);
```

```
            lightElement->QueryFloatAttribute("diry", &l.direction[1]);
            lightElement->QueryFloatAttribute("dirz", &l.direction[2]);
        }
        if (!strcmp(type, "spot")) {
            lightElement->QueryFloatAttribute("posx", &l.position[0]);
            lightElement->QueryFloatAttribute("posy", &l.position[1]);
            lightElement->QueryFloatAttribute("posz", &l.position[2]);
            lightElement->QueryFloatAttribute("dirx", &l.direction[0]);
            lightElement->QueryFloatAttribute("diry", &l.direction[1]);
            lightElement->QueryFloatAttribute("dirz", &l.direction[2]);
            lightElement->QueryFloatAttribute("cutoff", &l.cutoff);
        }
        lights.push_back(l);
        lightElement = lightElement -> NextSiblingElement("light");
    }
    return lights;
}



void renderScene(void) {
    int i = 0;
    (...)
    glLoadIdentity();
    if (helpMenu) {
        (...)
    } else {
        (...)
        for (Light l : world.lights) {
            float amb[4]  = {0.2f, 0.2f, 0.2f, 1.0f};
            float diff[4] = {1.0f, 1.0f, 1.0f, 1.0f};
            float spec[4] = {1.0f, 1.0f, 1.0f, 1.0f};
            glLightfv(GL_LIGHT0 + i, GL_AMBIENT, amb);
            glLightfv(GL_LIGHT0 + i, GL_DIFFUSE, diff);
            glLightfv(GL_LIGHT0 + i, GL_SPECULAR, spec);
            if (l.type == "point") {
                glLightfv(GL_LIGHT0 + i, GL_POSITION, l.position);
            }
            if (l.type == "directional") {
                glLightfv(GL_LIGHT0 + i, GL_POSITION, l.direction);
            }
            if (l.type == "spot") {
                glLightfv(GL_LIGHT0 + i, GL_POSITION, l.position);
                glLightfv(GL_LIGHT0 + i, GL_SPOT_DIRECTION, l.direction);
                glLightf(GL_LIGHT0 + i, GL_SPOT_CUTOFF, l.cutoff);
            }
            i++;
        }
        (...)
    }
    (...)
}
```

As luzes de um ficheiro XML são lidas através da parseLight e armazandas num vetor para depois serem aplicadas na *renderScene* antes de desenhar-mos os modelos.

3.2.4 Textura

```
int loadTexture(string s) {
    unsigned int t,tw,th;
    unsigned char *texData;
    unsigned int texID;
    // Iniciar o DevIL
    ilInit();
    // Colocar a origem da textura no canto inferior esquerdo
    ilEnable(IL_ORIGIN_SET);
    ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
    // Carregar a textura para memória
    ilGenImages(1,&t);
    ilBindImage(t);
    ilLoadImage((ILstring)s.c_str());
    tw = ilGetInteger(IL_IMAGE_WIDTH);
    th = ilGetInteger(IL_IMAGE_HEIGHT);
    // Assegurar que a textura se encontra em RGBA (Red, Green, Blue, Alpha) com um byte (0 - 255) por
    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
    texData = ilGetData();
    // Gerar a textura para a placa gráfica
    glGenTextures(1,&texID);
    glBindTexture(GL_TEXTURE_2D,texID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    // Upload dos dados de imagem
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData);
    glGenerateMipmap(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, 0);
    return texID;
}
```

A função *loadTexture* é responsável por carregar uma textura a partit de uma imagem, convertê-la em um formato adequado para OpenGL e, em seguida, carregá-la para a GPU.

3.2.5 Resultados Obtidos

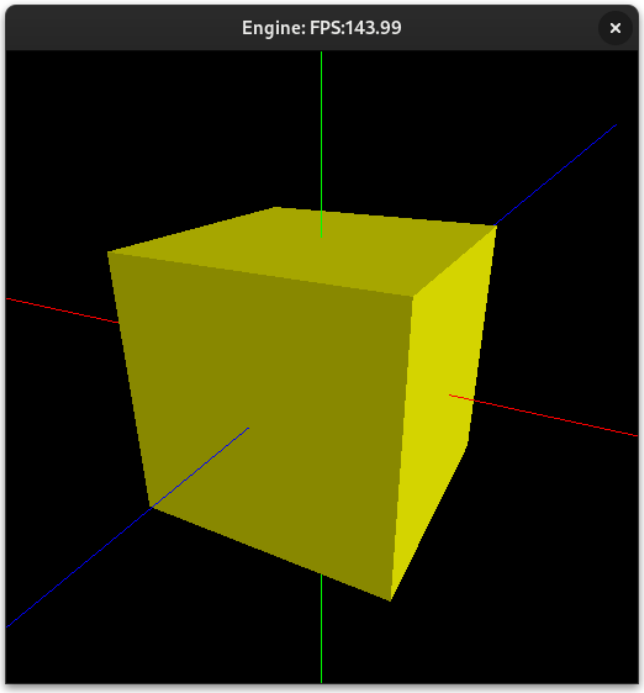


Figura 5: Ficheiro test_4_1.xml

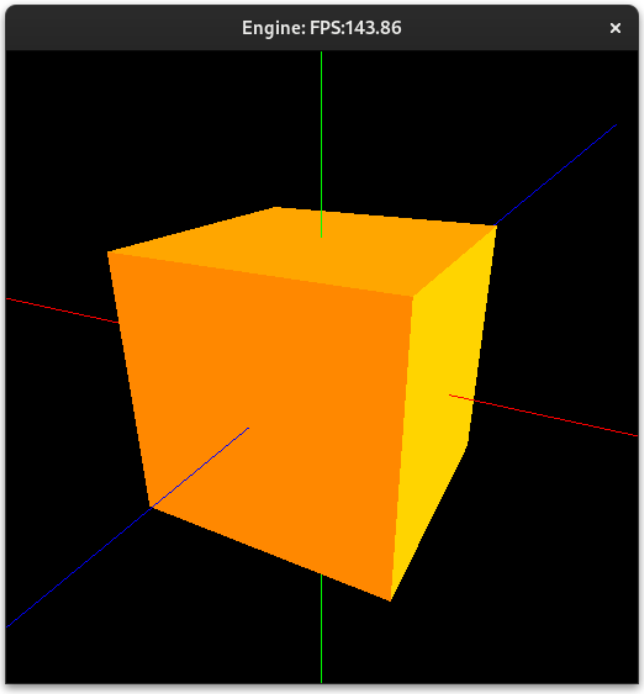


Figura 6: Ficheiro test_4_2.xml

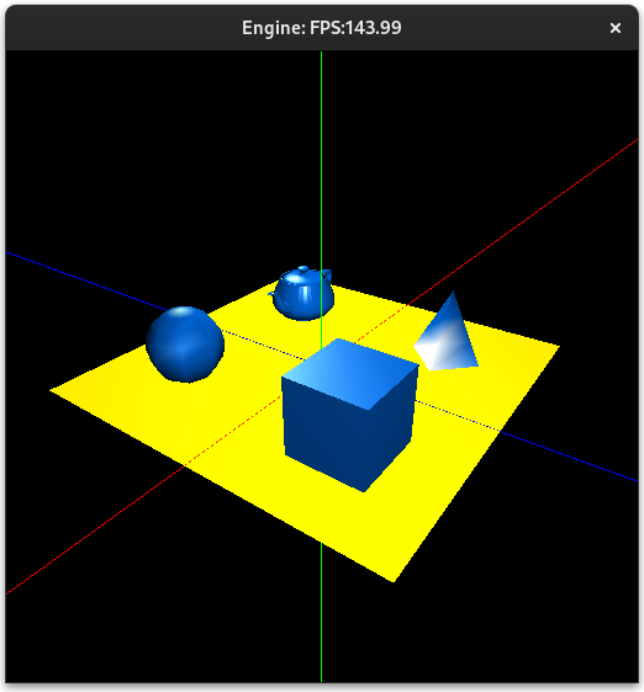


Figura 7: Ficheiro test_4_3.xml

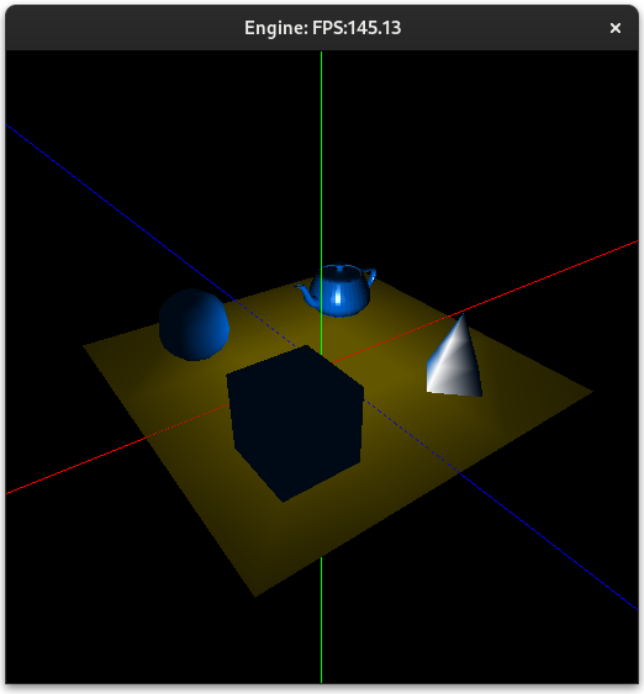


Figura 8: Ficheiro test_4_4.xml

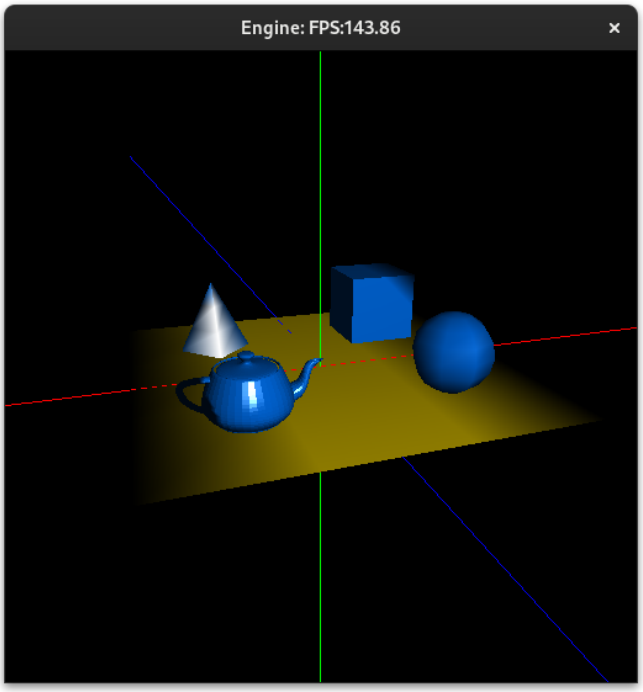


Figura 9: Ficheiro test_4_5.xml

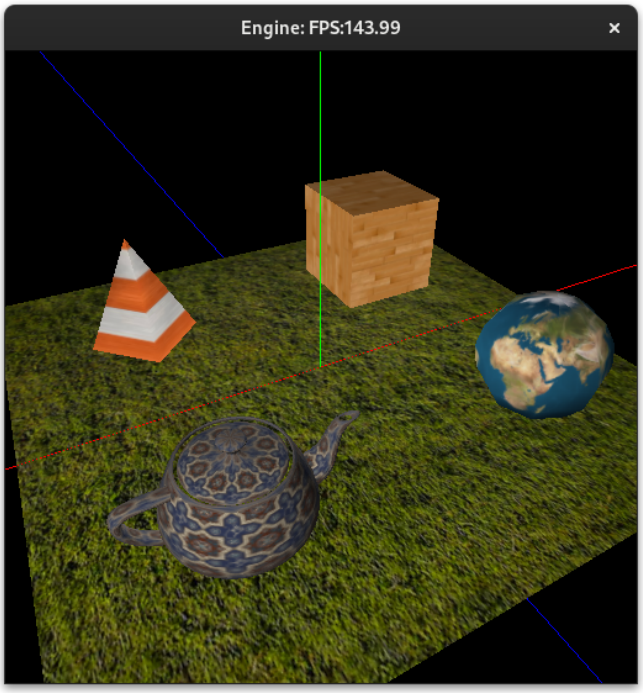


Figura 10: Ficheiro test_4_6.xml

3.3 Modelo do Sistema Solar

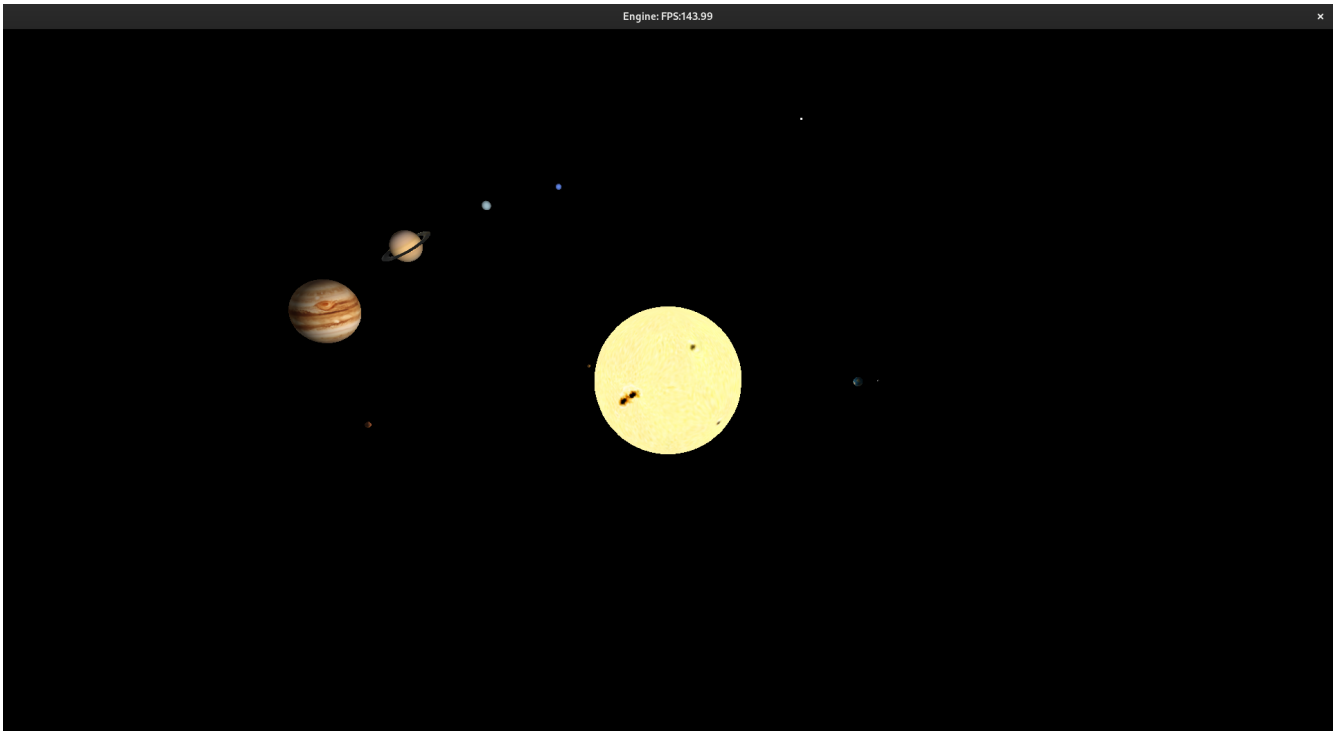


Figura 11: Sistema Solar Completo

Relativamente ao modelo apresentado na fase anterior o que desenvolvemos para esta não sofreu alterações significativas, sendo praticamente identico. Apenas adicionamos um ponto de luz central para representar a luz do Sol (light type=”point”posx=”0”posy=”0”posz=”0”) e as texturas dos vários planetas.

4 Conclusão

Em termos gerais, acreditamos que esta fase do projeto foi bem-sucedida, pois conseguimos atender a quase todos requisitos iniciais.

A realização desta quarta fase mostrou tudo aquilo que foi o empenho do grupo no sentido do desenvolvimento do projeto e para além dissoso o grupo expandiu os conhcimentos da matéria aprendida ao longo das aulas para esta terceira fase, apesar das dúvidas e desafios que foram surgindo relativamente, neste caso, à aplicação das coordenadas normais e de textura e à implementação da iluminação, cor e brilho.

Este projeto começou com a criação de algumas primitivas, mas rapidamente percebemos a escala e complexidade que ele teria quando nos foi pedido para aplicar transformações geométricas na segunda fase. Com a adição de curvas, patches de Bezier e VBOs na terceira fase, começamos a entender a conexão entre as diferentes tarefas, o que nos permitiu ter uma ideia de como seria o resultado final. Agora, com o uso de normais e coordenadas de textura para a aplicação de iluminação e texturas, conseguimos finalmente alcançar o objetivo do projeto. Algo que lamentamos é a não implementação de funcionalidades extras, como a cintura de asteroides, a combinação da câmera first person com o modo explorer, entre outras coisas.

Em resumo, após a conclusão deste projeto, concluímos que os tópicos ensinados foram devidamente assimilados e aprofundados através da resolução dos diferentes problemas propostos. Com isso em mente, acreditamos que o esforço investido no projeto permitirá uma melhor compreensão da parte teórica da disciplina de Computação Gráfica.