Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

4ª Fase - Luzes e Texturas

Diogo Matos
A100741

Júlio Pinto
A100742

Mário Rodrigues
A100109

Miguel Gramoso
A100835

May 27, 2024

CG

Índice

1. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Computação Gráfica (CG) no ano letivo 2023/2024.

A quarta fase do projeto consistiu em adicionar novas funcionalidades tanto ao ***engine*** como ao ***generator***. No *generator*, foi necessário alterar este para conseguir gerar os vetores normais e as coordenadas de textura para os vértices. Enquanto que no *engine* foi necessário permitir que este receba vetores normais e coordenadas de textura, tal como permitir a adição de luzes, materiais e texturas.

2. *Generator*

2.1. Ficheiros `.3d`

Para a garantir as alterações necessárias para o *generator*, mencionadas anteriormente, foi necessário alterar a estrutura dos ficheiros `.3d` que era utilizada até ao momento. Os seguintes excertos de ficheiros `.3d` mostram a nova estrutura em contraste com a estrutura utilizada anteriormente, respetivamente.

```
# 324
p -1.5 -1.5 1.5 0 0 1 -0.5 -0.5
p -0.5 -1.5 1.5 0 0 1 -0.166667 -0.5
p -1.5 -0.5 1.5 0 0 1 -0.5 -0.166667
p -1.5 -0.5 1.5 0 0 1 -0.5 -0.166667
p -0.5 -1.5 1.5 0 0 1 -0.166667 -0.5
p -0.5 -0.5 1.5 0 0 1 -0.166667 -0.166667
```

Listing 1: Excerto exemplar de um ficheiro `.3d`, de acordo com a **nova estrutura** definida

```
-1 -1 1
-0.333333 -1 1
-1 -0.333333 1
-1 -0.333333 1
-0.333333 -1 1
-0.333333 -0.333333 1
-1 -1 -1
```

Listing 2: Excerto exemplar de um ficheiro `.3d` na **estrutura anterior**

Foi acrescentada uma linha inicial no formato '`# <nr>`', responsável por ditar o número total de linhas que o ficheiro contem, dando a possibilidade de otimização da leitura dos ficheiros. Para além disso, para cada uma das restantes linhas, foi adicionada, para além da informação de posição já existente previamente, informação relativa ao valor da normal e da coordenada de textura, no seguinte formato '`p <posX posY posZ> <normX normY normZ> <textX textY>`', onde `pos`, `norm` e `text` referem-se às coordenadas de posição, normal e textura, respetivamente.

Esta nova estrutura garante, assim, as informações necessárias à aplicação de iluminação e texturas pelo *engine*.

2.2. Plano

2.2.1. Normais

No Plano, sabemos que todos os pontos terão a mesma normal, para além disso sabemos que este se encontra no plano xz. Sendo assim, deduz-se que a normal deste será: `(0,1,0)`, ou seja o vetor normalizado do eixo y.

2.2.2. Coordenadas de Textura

A textura é aplicada repetidamente ao longo do plano para cada subdivisão, pelo que as coordenadas de textura são calculadas de acordo com o número total de divisões do plano e de acordo com a subdivisão em que a iteração se encontra num dado momento. Sendo que o número de uma dada subdivisão do plano nunca será maior que o número total de divisões, temos coordenadas com valores entre 0 e 1, como esperado.

De seguida apresenta-se o código que demonstra este processo de cálculo:

```
for (int i = 0; i < divisions; i++) {
  for (int j = 0; j < divisions; j++) {

    // (...)
    float u1 = static_cast<float>(i) / static_cast<float>(divisions);
    float u2 = static_cast<float>(i + 1) / static_cast<float>(divisions);
    float v1 = static_cast<float>(j) / static_cast<float>(divisions);
    float v2 = static_cast<float>(j + 1) / static_cast<float>(divisions);

    textures.push_back(Point2D(u1, v1));
    textures.push_back(Point2D(u1, v2));
    textures.push_back(Point2D(u2, v1));

    textures.push_back(Point2D(u2, v1));
    textures.push_back(Point2D(u1, v2));
    textures.push_back(Point2D(u2, v2));
  }
}
```

2.3. Caixa

2.3.1. Normais

As normais dos pontos da caixa dependem apenas do lado em que cada um está inserido, sendo o valor da normal constante ao longo de cada lado. Assim, temos os seguintes valores possíveis:

- Topo: (0, 1, 0)
- Base: (0, −1, 0)
- Laterais:
  - (-1, 0, 0)
  - (1, 0, 0)
  - (0, 0, −1)
  - (0, 0, 1)

2.3.2. Coordenadas de Textura

A textura da caixa é aplicada de igual forma a cada uma das suas faces, sendo cada quadrado de divisão mapeado à porção correspondente da textura. Este mapeamento é obtido através da normalização das coordenadas do vértice na caixa para que fiquem no intervalo de $[0, 1]$, o que se calcula somando às coordenadas iniciais metade do comprimento do lado da caixa, deslocando o intervalo para $[0, \text{comprimento lado}]$, e dividindo depois o valor resultante pelo comprimento do lado da caixa, deslocando o intervalo para $[0, 1]$ como pretendido.

O código seguinte demonstra este cálculo:

```
textures.push_back(Point2D((v1 + halfSize) / length, (u1 + halfSize) / length));
textures.push_back(Point2D((v2 + halfSize) / length, (u1 + halfSize) / length));
textures.push_back(Point2D((v1 + halfSize) / length, (u2 + halfSize) / length));
textures.push_back(Point2D((v1 + halfSize) / length, (u2 + halfSize) / length));
textures.push_back(Point2D((v2 + halfSize) / length, (u1 + halfSize) / length));
textures.push_back(Point2D((v2 + halfSize) / length, (u2 + halfSize) / length));
```

Onde `halfSize` representa metade do comprimento do lado da caixa, `length` o comprimento do lado da caixa e `u1` e `v1` as coordenadas do ponto na caixa.

2.4. Esfera

2.4.1. Normais

São contrário do plano e da caixa, já será necessário alguns cálculos para obter as normais da esfera para cada ponto. Sabemos que o vetor normal da esfera para um ponto pode ser calculado através do seu centro e do ponto em questão. Visto que o centro da esfera será sempre o ponto `(0,0,0)` sabemos que o vetor será equivalente às coordenadas do ponto, sendo assim, basta então normalizar as coordenadas do ponto da esfera para obter o valor da normal.

2.4.2. Coordenadas de Textura

A maneira como calculamos as coordenadas de textura da esfera é relativamente similar à maneira de como calculamos as coordenadas para um plano. Porém ao invés de iterarmos em torno de divisões, iremos iterar então com base nos ângulos *theta* e *phi*.

De seguida apresenta-se o código que demonstra o processo de iteração:

```
for (int i = 0; i < slices; ++i) {
    float theta1 = static_cast<float>(i) * static_cast<float>(M_PI) /
                   static_cast<float>(slices);
    float theta2 = static_cast<float>(i + 1) * static_cast<float>(M_PI) /
                   static_cast<float>(slices);

    for (int j = 0; j < stacks; ++j) {
      float phi1 = static_cast<float>(j) * 2.0f * static_cast<float>(M_PI) /
                   static_cast<float>(stacks);
      float phi2 = static_cast<float>(j + 1) * 2.0f * static_cast<float>(M_PI) /
                   static_cast<float>(stacks);

      // Texture coordinates
      float u1 = phi1 / (2.0f * static_cast<float>(M_PI));
      float u2 = phi2 / (2.0f * static_cast<float>(M_PI));
      float v1 = theta1 / static_cast<float>(M_PI);
      float v2 = theta2 / static_cast<float>(M_PI);

      Point2D t1(u1, v1);
      Point2D t2(u2, v1);
      Point2D t3(u1, v2);
      Point2D t4(u2, v2);

      textures.push_back(t1);
      textures.push_back(t4);
      textures.push_back(t2);
      textures.push_back(t1);
      textures.push_back(t3);
      textures.push_back(t4);
    }
}
```

2.5. Cone

2.5.1. Normais

Para começar, a base do cone é um círculo no plano XY. Para todos os pontos na base, a normal é a mesma, apontando diretamente para baixo no eixo Y, ou seja, a normal é (0, −1, 0). As normais das laterais do cone

são mais complexas, pois precisam ser calculadas para cada ponto na superfície lateral. Sabemos que estas serão perpendiculares à superfície inclinada.

Para calcularmos iremos também iterar sobre cada *slice* e *stack*, ou seja utilizando um ângulo e uma altura. Para calcular a normal de um ponto numa posição dada por um ângulo `a` e altura `y`: `(sin(a), raio_base / altura, cos(a))`. Como último passo, normalizamos o vetor.

2.5.2. Coordenadas de Textura

A textura do cone é representada por uma imagem quadrada que deve ser mapeada corretamente à base e à lateral do cone.

Como a base é um círculo e a textura é um quadrado, as suas coordenadas são transformadas de modo a caberem no mesmo da seguinte forma: $x = 0.5 + 0.5 * \sin(a)$ e $y = 0.5 + 0.5 * \cos(a)$, tendo em conta que $(0.5, 0.5)$ é a coordenada central da textura.

Já para a lateral do cone, é necessário mapear coordenadas cilíndricas de modo a serem representadas em 2D, o que se obtém da seguinte forma:

```
for (int i = 0; i < stacks; i++) {
    for (float a = 0; a < (2 * M_PI); a += sliceAngle) {
        // ...
        textures.push_back(Point2D(a / (2 * M_PI), yCima / height));
        textures.push_back(Point2D(a / (2 * M_PI), yBaixo / height));
        textures.push_back(Point2D((a + sliceAngle) / (2 * M_PI), yBaixo / height));

        textures.push_back(Point2D(a / (2 * M_PI), yCima / height));
        textures.push_back(Point2D((a + sliceAngle) / (2 * M_PI), yBaixo / height));
        textures.push_back(Point2D((a + sliceAngle) / (2 * M_PI), yCima / height));
    }
}
```

2.6. Cilindro

2.6.1. Normais

Relativamente ao cilindro, sabemos já desde de início as normais do topo e da base, `(0,1,0)` e `(0,-1,0)`, respetivamente. Em relação às normais das laterais do cilindro, podemos assumir que estas correspondem a um vetor que une a projeção do centro da base do cilindro ao ponto, na mesma *stack*, dependendo apenas do ângulo em que se encontra. Desta maneira, as normais da face lateral assumem o seguinte formato: $(\sin(\alpha), 0, \cos(\alpha))$.

2.6.2. Coordenadas de Textura

Devido ao formato único da textura do cilindro, que contem as texturas da base ,do topo e das laterais em um único ficheiro, temos de saber o centro de cada textura para as aplicar corretamente.

$\text{Centro textura da base} = \text{Point}(0.8125,0.1875)$

$\text{Centro textura do Topo} = \text{Point}(0.4375,0.1875)$

$\text{Raio da texturas circulares} = \text{Point}(0,0.375)$

$\text{Altura textura lateral} = 0.625$

```
// ...

// topo
// ponto central
n = {0.0f, 1.0f, 0.0f};
v = {0.0f, height / 2.0f, 0.0f};
t = {0.4375f, 0.1875f};

points.push_back(v);
normals.push_back(n);
textures.push_back(t);

...

//lateral
n = {sin(i + 1 * delta), 0.0f, cos(i + 1 * delta)};
v = {radius * sin((i + 1) * delta), height / 2.0f, radius * cos((i + 1) * delta)};
t = {(i + 1) / static_cast<float>(slices), 1.0f};

points.push_back(v);
normals.push_back(n);
textures.push_back(t);

n = {sin(i * delta), 0.0f, cos(i * delta)};
v = {radius * sin(i * delta), height / 2.0f, radius * cos(i * delta)};
t = {i / static_cast<float>(slices), 1.0f};

points.push_back(v);
normals.push_back(n);
textures.push_back(t);

n = {sin(i * delta), 0.0f, cos(i * delta)};
v = {radius * sin(i * delta), -height / 2.0f, radius * cos(i * delta)};
t = {i / static_cast<float>(slices), 0.375f};

points.push_back(v);
normals.push_back(n);
textures.push_back(t);

//...
```

2.7. Torus

2.7.1. Normais

O método de cálculo das normais do *torus* é similar ao cálculo das normais do cilindro, mas orientam-se pelo centro do anel do *torus*. Para cada ponto é calculado o vetor de distância ao centro do anel, resultando na normal desse ponto.

2.7.2. Coordenadas de Textura

A textura aplicada ao *torus* foi feita de maneira a que esta cubra o *torus* por completo à volta do seu anel. Para calcularmos as suas coordenadas de textura apenas necessitamos de dividir a *slice* atual pelo número de *slices* e a *stack* atual pelo número de *stacks*, como demonstra o código que se segue:

```
// (...)

// Normals
Point n1 = Point(x1 - majorRadius * std::cos(theta1), y1,
                 z1 - majorRadius * std::sin(theta1));
Point n2 = Point(x2 - majorRadius * std::cos(theta1), y2,
                 z2 - majorRadius * std::sin(theta1));
Point n3 = Point(x3 - majorRadius * std::cos(theta2), y3,
                 z3 - majorRadius * std::sin(theta2));
Point n4 = Point(x4 - majorRadius * std::cos(theta2), y4,
                 z4 - majorRadius * std::sin(theta2));

// Assign normals to each vertex of the triangles
normals.push_back(n1);
normals.push_back(n2);
normals.push_back(n4);

normals.push_back(n1);
normals.push_back(n4);
normals.push_back(n3);

// Texture coordinates
float u1 = static_cast<float>(j) / static_cast<float>(sides);
float u2 = static_cast<float>(j + 1) / static_cast<float>(sides);
float v1 = static_cast<float>(i) / static_cast<float>(rings);
float v2 = static_cast<float>(i + 1) / static_cast<float>(rings);

textures.push_back(Point2D(u1, v1));
textures.push_back(Point2D(u2, v1));
textures.push_back(Point2D(u2, v2));

textures.push_back(Point2D(u1, v1));
textures.push_back(Point2D(u2, v2));
textures.push_back(Point2D(u1, v2));
```

2.8. Bézier

2.8.1. Normais

Para calcular as normais nas superfícies de *Bézier*, utilizamos as derivadas parciais de cada ponto ao longo da curva. Ao utilizar as derivadas parciais obtemos dois vetores (v, u), tangentes à superfície no ponto a calcular. Após o cálculo podemos facilmente obter um vetor paralelo à superfície, calculando o *cross product* de u e v.

2.8.2. Coordenadas de Textura

Utilizando os cálculos já efetuados para a normal, conseguimos calcular as coordenadas para as texturas dividindo u e v pelo nível de tesselação definido.

O código que se segue demonstra esse processo de cálculo:

```
// ...

        float u1 = static_cast<float>(u) / tessellation;
        float v1 = static_cast<float>(v) / tessellation;
        float u2 = static_cast<float>(u + 1) / tessellation;
        float v2 = static_cast<float>(v + 1) / tessellation;

        ...
        // Calculate normals using derivatives for smooth shading
        Point du1 = bezierPatchDU(patchControlPoints, u1, v1);
        Point dv1 = bezierPatchDV(patchControlPoints, u1, v1);
        Point normal1 = dv1.cross(du1);  // Switched the order
        float length1 =
            std::sqrt(normal1.x * normal1.x + normal1.y * normal1.y +
                      normal1.z * normal1.z);
        if (length1 != 0) {
          normal1.x /= length1;
          normal1.y /= length1;
          normal1.z /= length1;
        }

        Point du2 = bezierPatchDU(patchControlPoints, u2, v1);
        Point dv2 = bezierPatchDV(patchControlPoints, u2, v1);
        Point normal2 = dv2.cross(du2);  // Switched the order

        ...
        // Calculate texture coordinates
        Point2D texCoord1(u1, v1);
        Point2D texCoord2(u2, v1);
        Point2D texCoord3(u1, v2);
        Point2D texCoord4(u2, v2);

        // Add texture coordinates to the list
        textures.push_back(texCoord1);
        textures.push_back(texCoord3);
        textures.push_back(texCoord2);

        textures.push_back(texCoord2);
        textures.push_back(texCoord3);
        textures.push_back(texCoord4);

        //...
```

3. *Engine*

3.1. `Materials`

De maneira a conseguirmos lidar com os `Materials` foi criada a seguinte estrutura de dados:

```
struct Material{
  glm::vec4 ambient;
  glm::vec4 diffuse;
  glm::vec4 specular;
  glm::vec4 emission;
  float shininess;
};
```

Listing 10: Estrutura de dados para `Materials`

Nesta estrutura é guardado a componente ambiente, a componente difusa, a componente especular, o componente emissivo e por último o seu brilho. Tendo estes valores, podemos utilizá-los quando for necessário desenhar o modelo. Para tal, implementamos a função `setupMaterial()`, que, tal como o nome indica, irá preparar esta *struct* para ser utilizada pelo OpenGL.

3.2. `Lights`

Para as luzes, tal como para os materiais, criamos uma *struct* própria.

```
struct Light {
  LightType type;
  glm::vec4 position;
  glm::vec4 direction;
  float cutoff;
};
```

Listing 11: Estrutura de dados `Light`

Como podemos ver pela *struct* acima, é guardado o tipo da luz (`LightTyp`), a sua posição (`position`), a sua direção (`direction`) e o seu *cutoff*. Dependendo do tipo, este valores serão inicializados ou não. Para criar cada uma das luzes utilizamos as seguintes funções:

```
Light createDirectionLight(glm::vec4 direction);

Light createPointLight(glm::vec4 position);

Light createSpotLight(glm::vec4 position, glm::vec4 direction, float cutoff);
```

Listing 12: Construtores para a *struct* `Light`

Porém, devido à maneira que implementamos as luzes, estas ainda não interagem com o `GLUT`. Sendo assim, criamos uma função, `setupLights()`, responsável pela associação das luzes a um `LIGHT_N`, tal como todo o resto da preparação da cena.

```
bool setupLights(std::vector<Light> lights) {
  if (lights.size() != 0) {
    glEnable(GL_RESCALE_NORMAL);
    float amb[4] = {1.0f, 1.0f, 1.0f, 1.0f};

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
    glEnable(GL_LIGHTING);
    for (int i = 0; i < lights.size(); i++) {
      float white[4] = {1.0, 1.0, 1.0, 1.0};

      glEnable(GL_LIGHT0 + i);
      glLightfv(GL_LIGHT0 + i, GL_DIFFUSE, white);
      glLightfv(GL_LIGHT0 + i, GL_SPECULAR, white);
    }
    return true;
  }

  return false;
}
```

Listing 13: Função `setupLights`

Por último é necessário renderizar as luzes na cena. Para isso utilizamos a função `drawLights()`:

```
void drawLights(std::vector<Light> lights) {
  for (int i = 0; i < lights.size() && lights.size() < 8; i++) {
    const Light& light = lights[i];

    switch (light.type) {
      case DIRECTIONAL: {
        float direction[4] = {light.direction.x, light.direction.y,
                              light.direction.z, 0.0f};
        glLightfv(GL_LIGHT0 + i, GL_POSITION, direction);
        break;
      }
      case POINT: {
        float position[4] = {light.position.x, light.position.y,
                             light.position.z, 1.0f};
        glLightfv(GL_LIGHT0 + i, GL_POSITION, position);
        break;
      }
      case SPOT: {
        float postion[4] = {light.position.x, light.position.y,
                            light.position.z, 1.0f};
        glLightfv(GL_LIGHT0 + i, GL_POSITION, postion);
        float direction[4] = {light.direction.x, light.direction.y,
                              light.direction.z, 0.0f};
        glLightfv(GL_LIGHT0 + i, GL_SPOT_DIRECTION, direction);
        glLightf(GL_LIGHT0 + i, GL_SPOT_CUTOFF, light.cutoff);
        break;
      }
    }
  }
}
```

Listing 14: Função de `drawLights`

Utilizando esta função conseguimos renderizar todas as luzes da cena de forma adequada.

3.3. Vertex

Para esta fase, decidimos criar uma *struct* chamada `Vertex`. Esta *struct*, que representa um vértice, é composta pela posição do vértice (`position`), pelo vetor normal do ponto e por uma coordenada de textura (`texture`) (a posição relativa do ponto na textura).

```
struct Vertex {
  Point position;
  Point normal;
  Point2D texture;
```

Listing 15: *Struct* `Vertex`

Foi necessário criar esta *struct* devido à criação dos VBOs, onde é necessário existir uma distinção entre os vértices com a mesma posição dos vetores normais ou coordenadas de textura diferente.

3.4. Model

Para a realização desta fase foi necessário alterar algumas classes anteriormente definidas. A classe `Model` foi alterada para adicionar algumas coisas:

```
class Model {
 public:
  std::string filename, texture_filepath;
  std::vector<Vertex> vbo;
  std::vector<unsigned int> ibo;
  int id;
  bool initialized = false;
  BoundingSphere bounding_sphere;
  Material material;

  Model();
  Model(std::string filename, std::vector<Vertex> points);

  void initModel();
  void drawModel();
  void setupModel();
  bool loadTexture();
  void drawNormals();

  std::vector<Vertex> getPoints();

 private:
  GLuint _vbo, _ibo, _normals, _textures, _texture_id;
  std::vector<Vertex> _points;
  Model(std::string filename, std::vector<Vertex> vbo,
        std::vector int> ibo, int id, std::vector<Vertex> points);
};
```

Listing 16: Classe `Model`

Como podemos ver, à classe `Model` foram adicionados alguns atributos e métodos. Para começar, adicionamos o atributo `material`, que representa a estrutura `Materials` para o modelo, a estrutura `bounding_sphere`, que tem como objetivo conter os dados necessários à utilização de *Frustum Culling*, e a *string* `texture_filepath`, de modo conseguirmos aceder ao ficheiro de textura. Para além destes atributos todos, ainda temos alguns atributos privados como os *unsigned int* `_texture_id`, `_textures` e `_normals` que serão responsáveis pelo *binding* dos valores respetivos na GPU.

Para esta fase foi também necessário atualizar as funções de `setupModel()` e `drawModel()`:

```
void Model::setupModel() {
  std::vector<float> points = positionsFloats(this->vbo);
  std::vector<float> normals = normalFloats(this->vbo);
  std::vector<float> textures = textureFloats(this->vbo);

  glGenBuffers(1, &this->_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, this->_vbo);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float) * points.size(), points.data(),
               GL_STATIC_DRAW);

  glGenBuffers(1, &this->_normals);
  glBindBuffer(GL_ARRAY_BUFFER, this->_normals);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float) * normals.size(), normals.data(),
               GL_STATIC_DRAW);

  glGenBuffers(1, &this->_textures);
  glBindBuffer(GL_ARRAY_BUFFER, this->_textures);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float) * textures.size(),
               textures.data(), GL_STATIC_DRAW);

  // Generate and bind index buffer
  glGenBuffers(1, &this->_ibo);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->_ibo);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int) * this->ibo.size(),
               this->ibo.data(), GL_STATIC_DRAW);
}
```

Listing 17: Função `setupModel`

```
void Model::drawModel() {
  initModel();

  glBindTexture(GL_TEXTURE_2D, this->_texture_id);

  glBindBuffer(GL_ARRAY_BUFFER, this->_vbo);
  glVertexPointer(3, GL_FLOAT, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, this->_normals);
  glNormalPointer(GL_FLOAT, 0, 0);

  glBindBuffer(GL_ARRAY_BUFFER, this->_textures);
  glTexCoordPointer(2, GL_FLOAT, 0, 0);

  glColor3f(1.0, 1.0, 1.0);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->_ibo);
  glDrawElements(GL_TRIANGLES, this->ibo.size(), GL_UNSIGNED_INT, 0);

  glBindTexture(GL_TEXTURE_2D, 0);
}
```

Listing 18: Função `drawModel`

Como podemos ver na função `drawModel()` e pelos métodos da classe, foi necessário criar as funções `initModel()` e `loadTexture()`. A função `initModel()` acaba por ser um *wrapper* para a função `setupModel()` e para a função `loadTexture()`.

Para a função `loadTexture()` decidimos utilizar a biblioteca `<stb_image.h>` em vez da biblioteca *DevIL*.

```
bool Model::loadTexture() {
  // Load image data
  int width, height, num_channels;
  unsigned char* image_data = stbi_load(this->texture_filepath.data(), &width,
                                        &height, &num_channels, STBI_rgb_alpha);

  if (!image_data) {
    std::cerr << "Failed to load texture: " << this->texture_filepath
              << std::endl;
    return false;
  }

  glGenTextures(1, &this->_texture_id);
  glBindTexture(GL_TEXTURE_2D, this->_texture_id);

  // Set texture parameters
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  // Para provar que a textura do teapot ser diferente se deve ao Mipmap troca se os
comandos abaixo pelos comentados
  // glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR_MIPMAP_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  // Upload data to GPU
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, image_data);
  glGenerateMipmap(GL_TEXTURE_2D);

  // Unbind the texture
  glBindTexture(GL_TEXTURE_2D, 0);

  // Free image data after uploading it
  stbi_image_free(image_data);

  return true;
}
```

Listing 19: Função `loadTexture`

Da biblioteca <stb_image> utilizamos a função `stbi_load()`. Esta função irá ler os *bytes* da imagem e irá guardar estes consoante o número de canais pedidos. Neste caso, esta irá guardar os *bytes* no *buffer* de maneira a ser compatível com RGBA. Deste modo, qualquer tipo de modo de cor funcionará.

Para além disso, nesta função definimos os parâmetros para *Mipmapping*. Decidimos utilizar então as *flags* do OpenGL, `GL_LINEAR_MIPMAP_LINEAR` para o `GL_TEXTURE_MIN_FILTER` e a `GL_LINEAR` para a `GL_TEXTURE_MAG_FILTER`. Devido a isto, obtemos alguns resultados diferentes dos demonstrados pelos testes dos docentes.

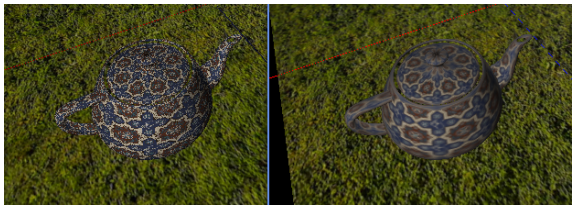

Figura 1: Resultados - Teste dos docentes à esquerda

Para provar que se deve ao *Mipmapping* basta alterar os parâmetros do `GL_TEXTURE_MIN_FILTER` para `GL_LINEAR`.

3.4.1. VBOs (*Vertical Buffer Objects*)

Como mencionado previamente, foi necessário alterar os VBOs. Agora, em vez da função `generateVBO()` receber um vetor de pontos, recebe um vetor de vértices. Assim, este irá comparar para além da posição, as normais e as coordenadas de textura.

3.4.2. IBOs (*Index Buffer Objects*)

Para os IBOS, foi necessário também alterar a função `generateIBO()` para que receba vetores de vértices. Porém, o seu comportamento manteve-se se igual.

3.5. Frustum Culling

Um dos nossos objetivos pessoais para esta fase foi a implementação de *Frustum Culling*. Para tal, foi necessário definir duas classes/estruturas, a `Frustum` e a `BoundingSphere`.

Para esta implementação guiamo-nos pelos *websites* *LearnOpenGL* e *Lighthouse3D*.

3.5.1. Frustum

Para definir o *Frustum*, foi necessário primeiramente definir uma estrutura `Plane`. Como sabemos que um plano pode ser definido ou por 3 pontos ou por um ponto e uma normal, optamos por definir este com um ponto e uma normal.

```
struct Plane {

    glm::vec3 point = { 0.f, 0.f, 0.f };
    glm::vec3 normal = { 0.f, 1.f, 0.f };

    Plane() = default;
    Plane(const Plane& other) = default;
    Plane(const glm::vec3& normal, glm::vec3 point);

    float distanceToPoint(const glm::vec3& point) const {
        return glm::dot(normal, point - this->point);
    }


};
```

Listing 20: Estrutura `Plane`

Tendo então esta estrutura de dados, conseguimos definir o nosso `Frustum`:

```
struct Frustum {
    Plane nearFace;
    Plane farFace;
    Plane rightFace;
    Plane leftFace;
    Plane topFace;
    Plane bottomFace;
    bool on = true;
  };
```

Listing 21: Estrutura do `Frustum`

Como podemos ver, o `Frustum` é constituído por 6 planos e um booleano. Cada um deste planos serve para delimitar o espaço da câmara. O booleano serve para sabermos se pretendemos usar o `Frustum` ou não para renderizar os modelos.

Para sabermos se um modelo se encontra dentro do `Frustum`, é necessário definir uma função `isInsideFrustum()`. Esta função irá receber os dados de, no caso, a `Bounding Sphere` e irá ver se o modelo está dentro ou fora do `Frustum`, através da equação do plano.

Utilizando a equação do plano, conseguimos obter a distância do plano ao ponto central da esfera. Caso este valor seja positivo, sabemos que este estará à frente do plano. Visto que as normais dos planos do `Frustum` estão voltadas para o espaço da câmara, conseguimos deduzir que, caso um modelo esteja a uma distância positiva de todos os planos, este estará dentro do espaço da câmara, logo será renderizado. Mais à frente explicamos como fazemos este cálculo para as `BoundingSpheres`.

3.5.2. Bounding Sphere

Para o `Frustum` funcionar é necessário definir uma maneira de sabermos o que está dentro e fora deste. Para isso, utilizamos a `Bounding Sphere`. Decidimos utilizar este método, em vez de por exemplo um AABB, devido ao contexto em que desenvolvemos o nosso *engine*.

Definimos então a seguinte estrutura:

```
struct BoundingSphere {

    glm::vec3 center;
    float radius;

    BoundingSphere() = default;
    BoundingSphere(const BoundingSphere& other) = default;
    BoundingSphere(const glm::vec3& center, float radius) : center(center), radius(radius)
{}
    BoundingSphere(std::vector<Vertex> points);

};
```

Listing 22: Estrutura `BoundingSphere`

Como podemos ver, a esfera é composta pelo seu centro e pelo raio. Para obtermos o centro utilizamos o `BoundingSphere(std::vector<Vertex> points)`, este construtor irá aproximar o centro da esfera fazendo a média entre o valor máximo e mínimo das coordenadas do modelo.

Como falado anteriormente, é necessário saber se a esfera está ou não dentro do `Frustum`. Para isso, definimos este pequeno excerto de código que poderá ser usado tanto no *Frustum* como na `BoundingSphere`:

```
    glm::vec3 center = glm::vec3(transformations * glm::vec4(this->center, 1.0f));
    float radius = this->radius * glm::length(glm::vec3(transformations[0])) / 2;

    return frustsum.nearFace.distanceToPoint(center) > -radius &&
           frustsum.farFace.distanceToPoint(center) > -radius &&
           frustsum.rightFace.distanceToPoint(center) > -radius &&
           frustsum.leftFace.distanceToPoint(center) > -radius &&
           frustsum.topFace.distanceToPoint(center) > -radius &&
           frustsum.bottomFace.distanceToPoint(center) > -radius;
```

Listing 23: Código para saber se a esfera está dentro do `Frustum`

Através destes cálculos, conseguimos então determinar se uma esfera está ou não no espaço da câmara.

3.6. Câmara FPS

Para esta fase, decidimos abandonar a nossa câmara orbital e implementar uma câmara FPS. Esta câmara utiliza um sistema de *input* que guarda o tempo que a tecla esteve pressionada, conseguindo assim obter um movimento *smooth* ao contrário de um movimento reativo.

3.7. ImGUI

Como mencionado em relatórios anteriores, para além do *Frustum Culling* tínhamos como objetivo implementar uma pequena interface utilizado a biblioteca *ImGUI*.

Para isso, definimos três funções, uma de *setup* (`setupMenu()`), uma de render (`renderMenu()`) e uma de *shutdown* (`shutDownMenu()`). A função de *setup* e *shutdown* são responsáveis pela inicialização e encerramento da interface, enquanto que a função de `renderMenu()` é responsável por toda a lógica da renderização da interface.

Abaixo encontra-se uma imagem do *engine* utilizando as interfaces:

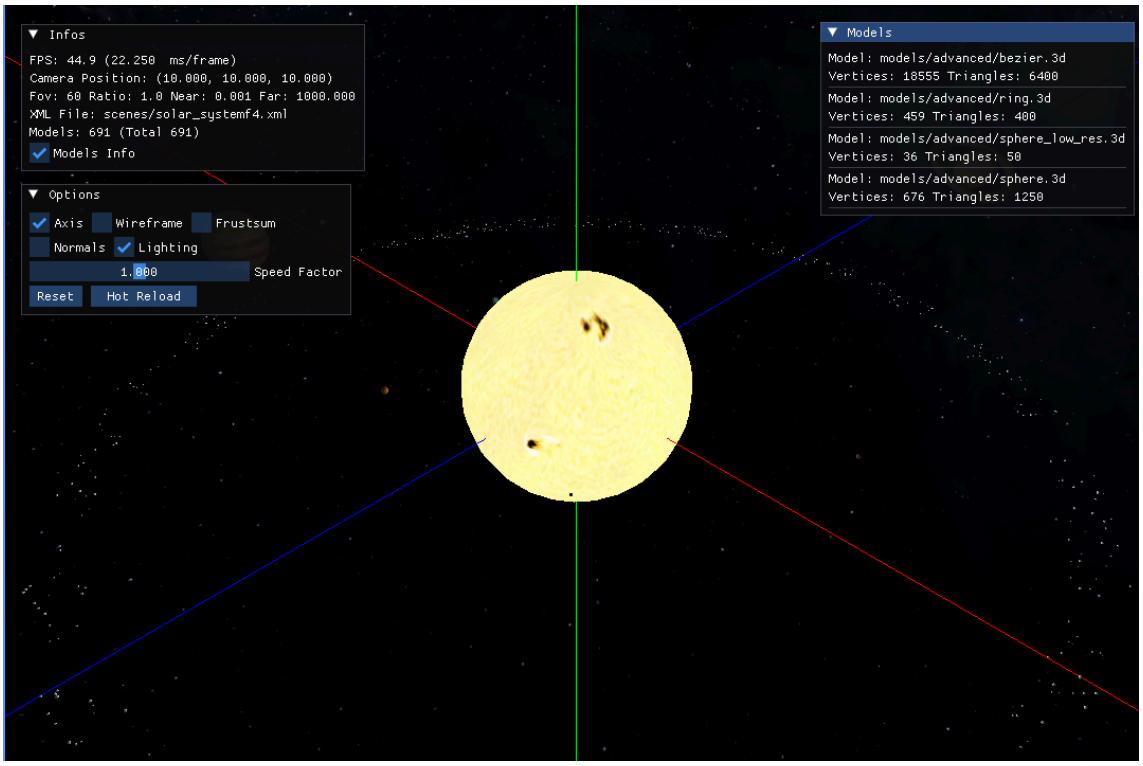

Figura 2: Interfaces

4. Sistema Solar

Como forma de melhor testar as novas funcionalidades incluídas nesta fase, melhoramos a *scene* relativa ao Sistema Solar adicionando texturas, materiais e luzes.

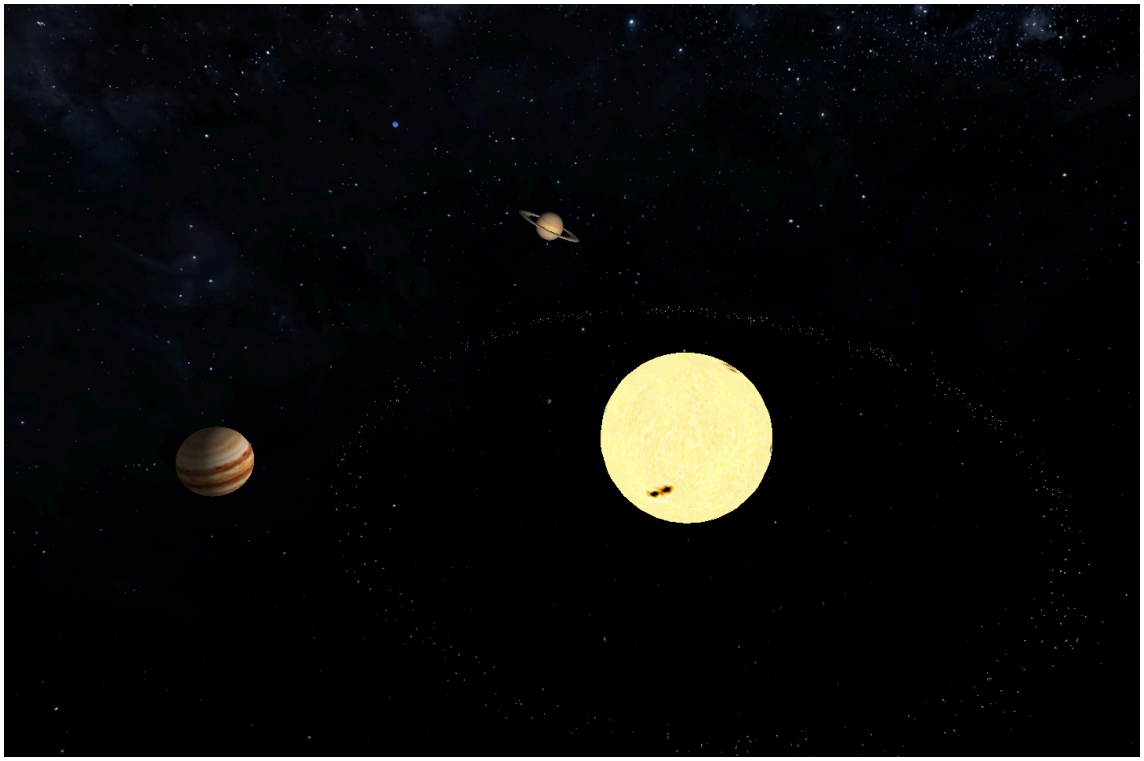

Figura 3: Sistema Solar

5. Testes

De modo a testar as diversas funcionalidades implementadas nesta fase, foram realizados um conjunto de testes que se apresentam de seguida.

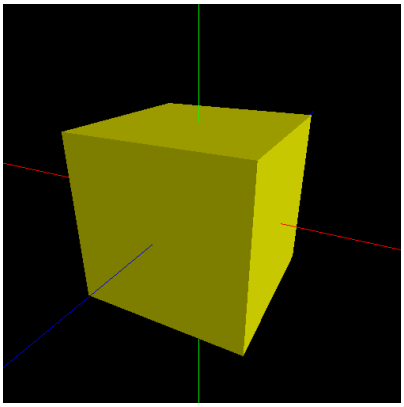

Figura 4: Teste nº1

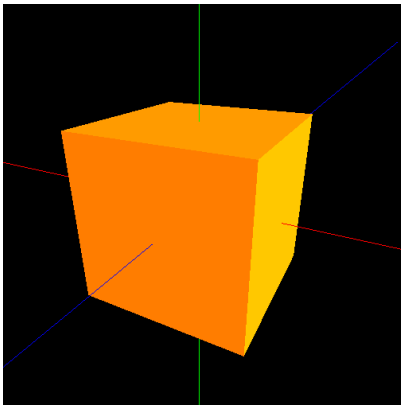

Figura 5: Teste nº2

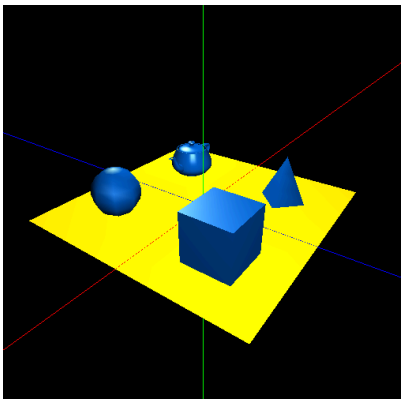

Figura 6: Teste nº3

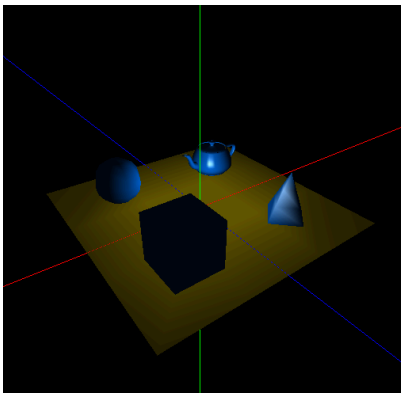

Figura 7: Teste nº4

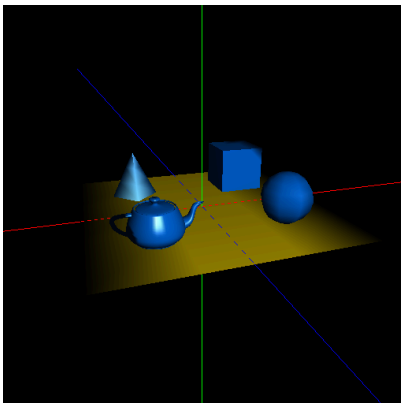

Figura 8: Teste nº5

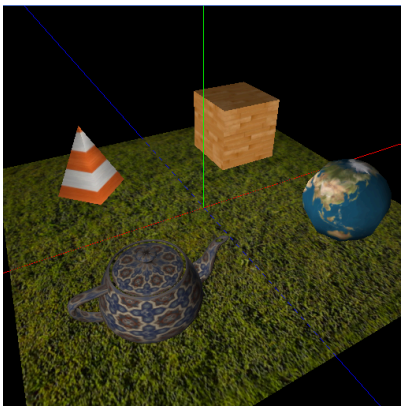

Figura 9: Teste nº6

6. Conclusões e Trabalho Futuro

Consideramos que os resultados que obtemos nesta fase foram os esperados, apesar de não conseguirmos realizar algumas otimizações no que toca a memória ou em alguns cálculos.

Conseguimos implementar diversos extras desde ImGUI a *Frustum Culling*, assim como extras mais pequenos como *Hot Reloading*. Acreditamos que todo o desenvolvimento do projeto foi uma experiência bastante enriquecedora e permitiu-nos desenvolver os nossos conhecimentos sobre Computação Gráfica num ambiente académico.