Computação Gráfica

Trabalho Prático - Fase 4

Universidade do Minho, Departamento de Informática
José Correia, Diogo Abreu, e Rodrigo Monteiro
{100610, 100646, 100706}@alunos.uminho.pt

Grupo 69

1. Introdução

Nesta última fase, foram adicionadas as funcionalidades de iluminação e texturas, e por isso foi necessário o cálculo das normais e das coordenadas de textura para cada tipo de figura.

2. Modificações nas estruturas de dados

2.1. Light

Foi adicionado um array de `Light` à classe `Config`, e esta passou a fazer *parsing* dos elementos XML com informação relativamente a iluminação.

```
vector<std::unique_ptr<Light>> lights;
```

Estruturamos esses dados da seguinte maneira, sendo que `PointLight` possui uma posição, `DirectionLight` uma direção, e `SpotLight` uma posição, direção e um valor de *cutoff*.

```
class Light {
public:
    enum class Type { Point, Directional, Spotlight };
    virtual ~Light() {}
    virtual Type get_type() const = 0;
    virtual std::string to_string() const = 0;
};

class PointLight : public Light { ... };
class DirectionalLight : public Light { ... };
class SpotLight : public Light { ... };
```

2.2. Figure

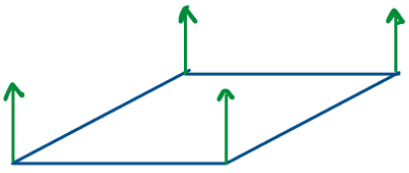
A classe `Figure` passou a ter informação acerca das normais e coordenadas de textura de cada ponto, e acerca da sua cor: `diffuse`, `ambient`, `specular`, `emissive` (*arrays* de 4 valores), e `shininess` (apenas um valor). (Os métodos `from_file` e `to_file` também foram modificados para incluir estes novos dados).

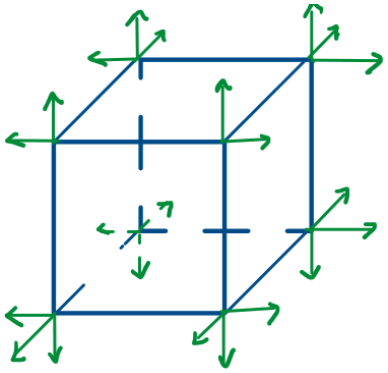
```
class Figure {
public:
    Figure(
        const vector<Point>& points = {},
        const vector<Point>& normals = {},
        const vector<Point>& texture_coords = {}
    );
    ~Figure();

    vector<Point> points;
    vector<Point> normals;
    vector<Point> texture_coords;
    std::string texture_file;

    std::string to_string() const;
    vector<float> to_vector();
    vector<float> get_normals_vector();
    vector<float> get_texture_coords_vector();
    // ...

private:
    vector<float> diffuse;
    vector<float> ambient;
    vector<float> specular;
    vector<float> emissive;
    float shininess;
};
```

3. Cálculo das normais

3.1. Plano e Caixa

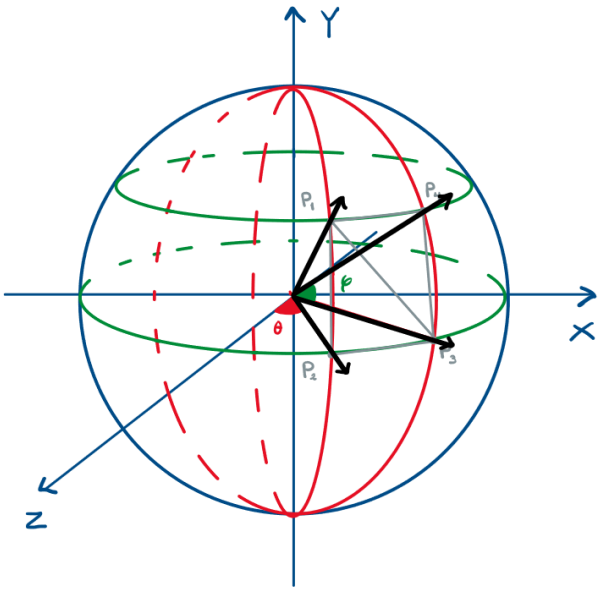
Cada normal dos planos da caixa aponta para o exterior do objeto.





3.2. **Esfera**

Cada normal da esfera é um vetor normalizado desde o interior da esfera até a um ponto da superfície.



3.3. **Cone**

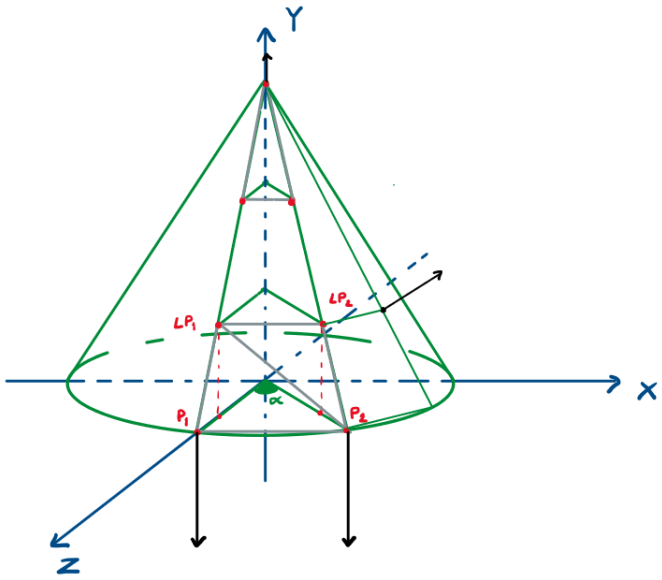
O cálculo das normais dos pontos do cone é composto por duas etapas: o cálculo das normais da base e o cálculo das normais dos vértices das faces laterais.

Para as normais da base, todos os seus pontos têm o vetor normal virado para baixo.

Para as normais das faces laterais, os vértices pertencentes a uma mesma aresta do cone têm sempre a mesma normal, independentemente da altura. Assim, o cálculo do vetor normal dos vértices de uma aresta, é obtido da seguinte forma:

Sendo o vetor B, o vetor que parte do centro da base até ao vértice do topo, e o vetor A o vetor que parte do centro da base até ao início da aresta, pode-se obter um vetor C através do produto vetorial A $\times$ B. A partir do vetor C, é possível calcular o vetor perpendicular à aresta, «vetor da aresta» $\times$ C = N.

Por fim, o vértice do topo tem uma normal igual a (0, 1, 0).



3.4. **Superfícies de Bezier**

Para calcular as normais de superfíces de Bezier, recorremos à utilização das seguintes fórmulas:

$$p(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

$$\frac{\partial p(u,v)}{\partial u} = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

$$\frac{\partial p(u,v)}{\partial v} = U M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

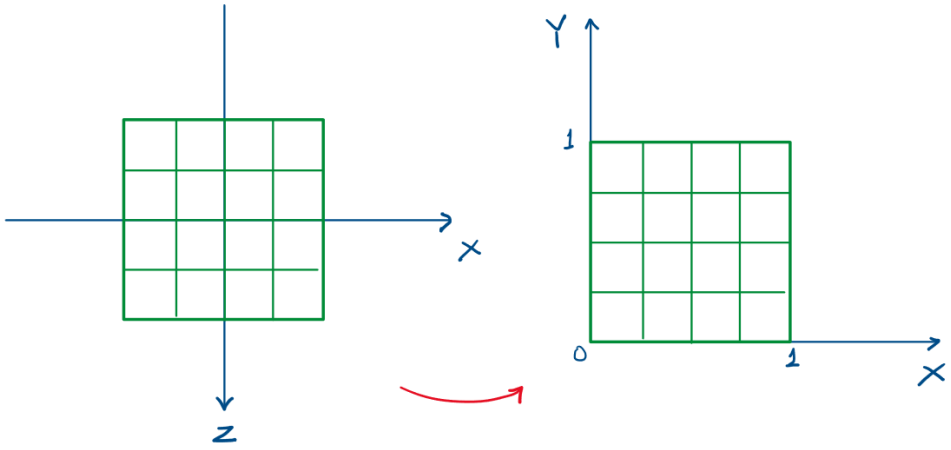
Dados os valores de u e v, o procedimento para o cálculo da normal é o seguinte:

$$\vec{u} = \frac{\partial p(u,v)}{\partial u}$$
$$\vec{v} = \frac{\partial p(u,v)}{\partial v}$$
$$\vec{n} = \vec{v} \times \vec{u}$$

4. Cálculo das coordenadas de textura

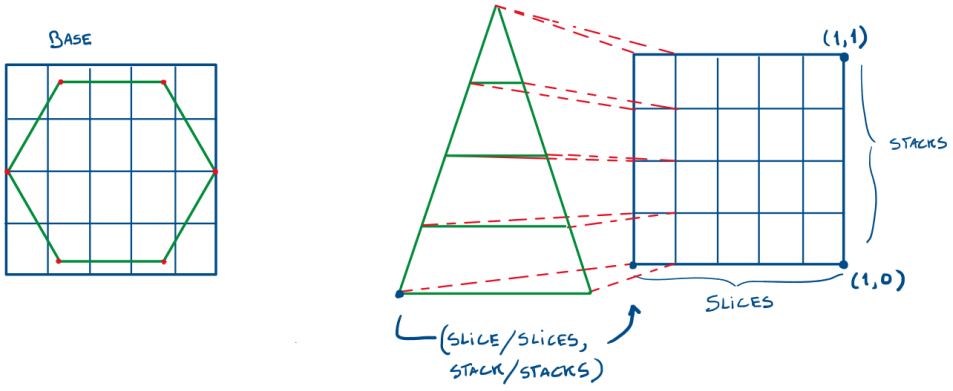
4.1. Plano e Caixa

Cada ponto do plano é mapeado para a textura, ficando com o X e Y entre 0 e 1.



4.2. Cone

Cada face do cone é mapeada para uma dada partição da textura de acordo com a *slice*. Cada face é dividida por *stacks*, formando uma célula que é mapeada para a respetiva célula da partição.



4.3. Esfera

Para o mapeamento das texturas da esfera, dividimos a imagem de textura pelo número de *stacks* e *slices* da esfera. Cada quadrado formado pela divisão da imagem de textura, será mapeado para a respectiva divisão da esfera.

4.4. Superfícies de Bezier

No cálculo das coordenadas de textura das superfícies de Bezier aplicamos a textura a cada *patch*.Portanto, podemos utilizar os parâmetros u e v no mapeamento das coordenadas de textura. Assim, as coordenadas de textura são dadas por $(1 - v, 1 - u)$.

5. Engine

5.1. Iluminação

Adição de código à função `loadBuffersData`:

```
glBindBuffer(GL_ARRAY_BUFFER, buffersN[*index]);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) *
figure_normals_vec.size(), figure_normals_vec.data(), GL_STATIC_DRAW);
buffersNSizes.push_back(figure_normals_vec.size() / 3);
```

Adição de código à função `drawGroups`:

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, f->get_diffuse().data());
glMaterialfv(GL_FRONT, GL_AMBIENT, f->get_ambient().data());
glMaterialfv(GL_FRONT, GL_SPECULAR, f->get_specular().data());
glMaterialfv(GL_FRONT, GL_EMISSION, f->get_emissive().data());
glMaterialf(GL_FRONT, GL_SHININESS, f->get_shininess());

glBindBuffer(GL_ARRAY_BUFFER, buffersN[*index]);
glNormalPointer(GL_FLOAT, 0, 0);
```

Nova função `execute_lights` (chamada na função `renderScene`):

```
void execute_lights() {
  for (unsigned int i = 0; i < c->lights.size(); i++) {
    const auto& light_ptr = c->lights.at(i);
    int CLight = gl_light(i);
    switch (light_ptr->get_type()) {
    case Light::Type::Point: {
      PointLight* point_light = dynamic_cast<PointLight*>(
        light_ptr.get()
      );
      if (point_light) {
        glLightfv(CLight, GL_POSITION, point_light->get_pos().data());
      }
      break;
    }
    case Light::Type::Directional: { ... }
    case Light::Type::Spotlight: { ... }
  }
}
```

5.2. **Texturas**

Na inicialização da `Engine`, começamos por carregar para os VBO's os dados relativos aos modelos do ficheiro XML, incluindo as coordenadas de textura e os caminhos para as imagens com as texturas.

Depois, na função `renderScene`, voltamos a percorrer os modelos pela mesma ordem de modo a garantir a aplicação das imagens de textura de forma ordenada aos modelos.

Adição de código à função `loadBuffersData`:

```
if (texture_file != nullptr && *texture_file != '\0') {
  glBindBuffer(GL_ARRAY_BUFFER, buffersTC[*index]);
  glBufferData(GL_ARRAY_BUFFER, sizeof(float)
    * figure_texture_coords_vec.size,
    figure_texture_coords_vec.data(), GL_STATIC_DRAW);
  loadTexture(texture_file, index);
}
```

Adição de código à função `drawGroups`:

```
const char* texture_file = f->texture_file.c_str();
if (texture_file != nullptr && *texture_file != '\0') {
  glBindTexture(GL_TEXTURE_2D, textures[*index]);
  glBindBuffer(GL_ARRAY_BUFFER, buffersTC[*index]);
  glTexCoordPointer(2, GL_FLOAT, 0, 0);
}
```

6. **Resultados dos testes**

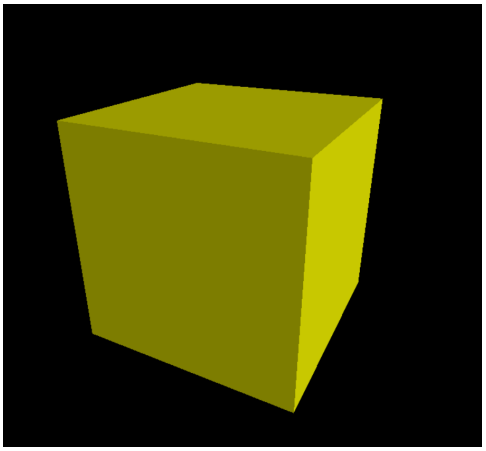


Figure 8: test_4_1.xml

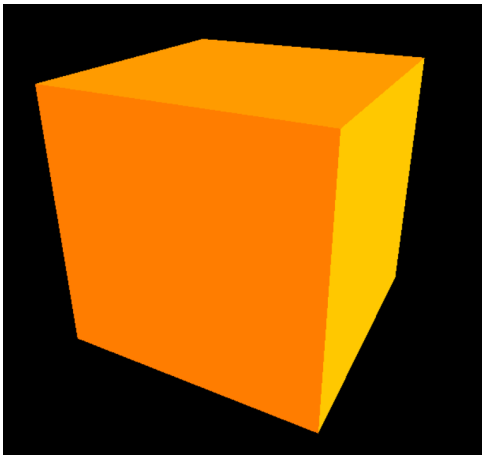


Figure 9: test_4_2.xml

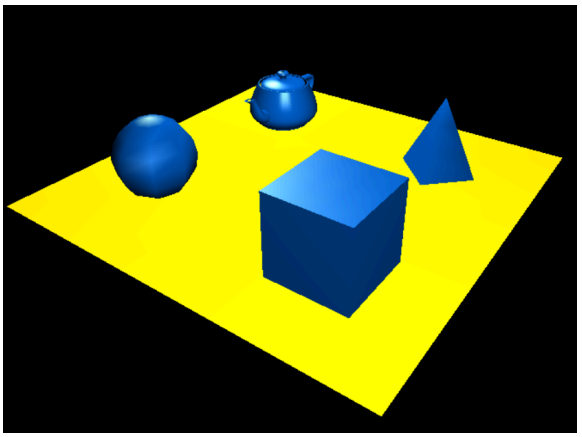


Figure 10: test_4_3.xml

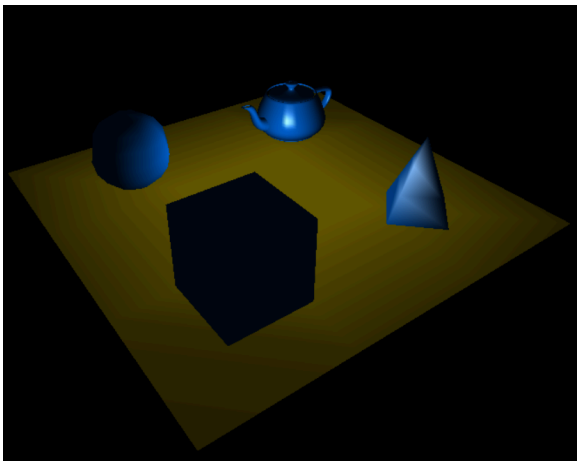


Figure 11: test_4_4.xml

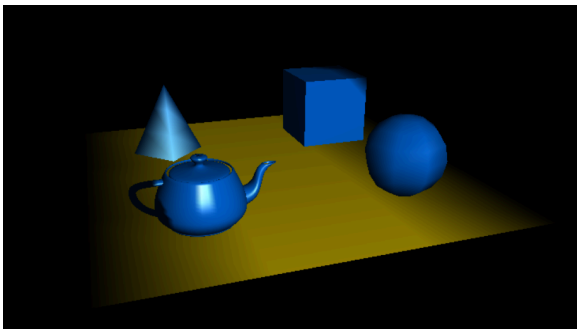


Figure 12: test_4_5.xml

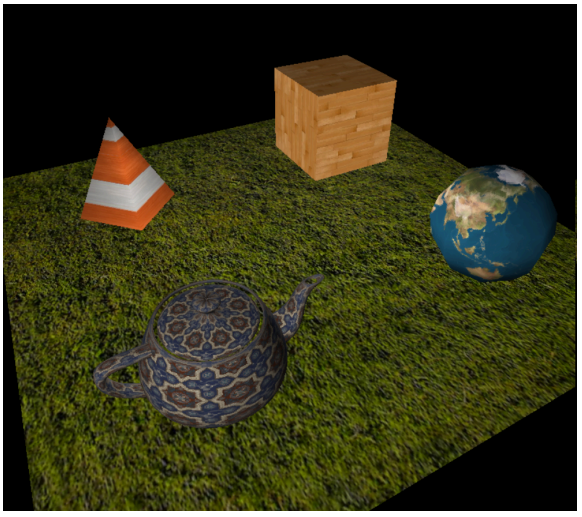


Figure 13: test_4_6.xml

7. Sistema solar

Utilizamos um *script* em Python para obter um ficheiro XML com um cenário do sistema solar. Para isso, o *script* recebe dados de dois ficheiros – um com dados acerca dos planetas e outro com dados acerca das luas – `planets_and_sun.json` e `satellites.json`. Desses dados, utilizamos as distâncias (entre os planetas e o sol), e os diâmetros, período de rotação e período de órbita.

`planets_and_sun.json`

```
{
    "id": 3,
    "name": "Earth",
    "mass": 5.97,
    "diameter": 12756.0,
    "density": 5514.0,
    "gravity": 9.8,
    "escapeVelocity": 11.2,
    "rotationPeriod": 23.9,
    "lengthOfDay": 24.0,
    "distanceFromSun": 149.6,
    "perihelion": 147.1,
    "aphelion": 152.1,
    "orbitalPeriod": 365.2,
    "orbitalVelocity": 29.8,
    "orbitalInclination": 0.0,
    "orbitalEccentricity": 0.017,
    "obliquityToOrbit": 23.4,
    "meanTemperature": 15.0,
    "surfacePressure": 1.0,
    "numberOfMoons": 1,
    "hasRingSystem": false,
    "hasGlobalMagneticField": false,
    "texture": "earth.jpg"
}
```

`satellites.json`

```
{
  "id": 1,
  "planetId": 3,
  "name": "Moon",
  "gm": 4902.801,
  "radius": 1737.5,
  "density": 3.344,
  "magnitude": -12.74,
  "albedo": 0.12
},
```

Exemplo de um `<group>` de um planeta gerado pelo *script* `solar-system-json-to-xml.py`:

```
<group>
    <transform>
      <rotate time="4" x="0" y="1" z="0"/>
      <translate x="413" y="0" z="0"/>
    </transform>
    <group>
      <!--Earth-->
      <transform>
        <rotate time="24" x="0" y="1" z="0"/>
        <scale x="6" y="6" z="6"/>
      </transform>
```

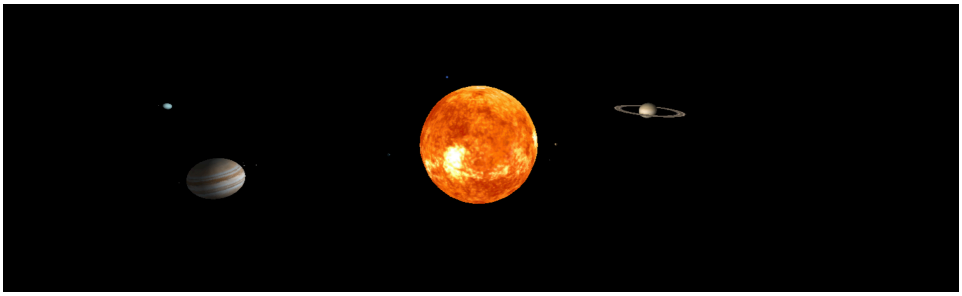
```
    <models>
      <model file="sphere_1_21_21.3d">
        <texture file="earth.jpg"/>
      </model>
    </models>
  </group>
  <group>
    <!--Moon-->
    <transform>
      <rotate time="9" x="0.1" y="1" z="0.0"/>
      <translate x="12" y="0" z="0"/>
      <scale x="1.738" y="1.738" z="1.738"/>
    </transform>
    <models>
      <model file="sphere_1_5_5.3d">
        <texture file="moon.jpg"/>
      </model>
    </models>
  </group>
</group>
```

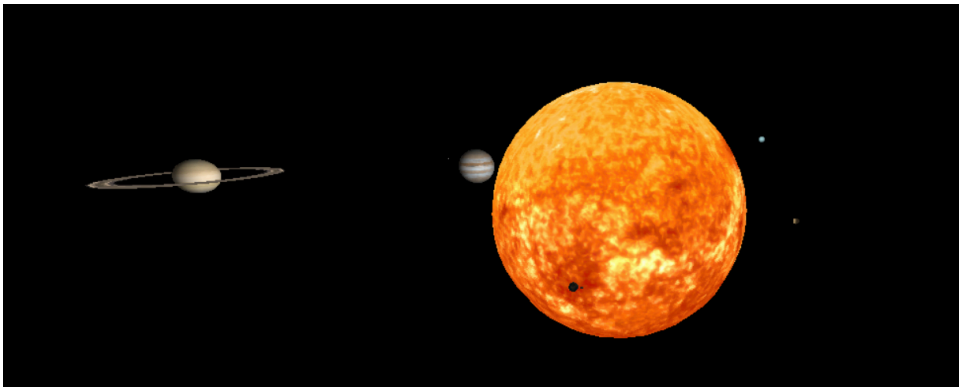
7.1. **Configurações**

Devido à proporção entre o raio dos planetas e a distância entre eles e o sol (sendo que as distâncias estão numa ordem de grandeza muito maior), neste contexto, as dimensões reais não são visualmente satisfatórias, e por isso declaramos algumas variáveis de configuração.

```
SPHERE_MODEL = 'sphere_1_21_21.3d'
LOWER_RES_SPHERE_MODEL = 'sphere_1_5_5.3d'
RING_MODEL = 'ring_3_4_20.3d'
BEZIER_COMET = 'bezier_10.3d'

PLANET_SCALE = 0.001
SUN_SCALE = 0.0004
MOON_SCALE = 0.001
DISTANCE_SCALE = 0.9
COMET_ORBIT_SCALE = 600
MAX_SATELLITES_PER_PLANET = 4
SUN_DEFAULT_DIAMETER = 1391400
ORBITAL_PERIOD_SCALE = 0.01
ROTATION_PERIOD_SCALE = 1
```

7.2. **Output**





8. **Conclusões**

Durante a conclusão da última fase deste projeto, concentramo-nos na implementação das funcionalidades de iluminação e texturas. Para isso, realizamos o cálculo das normais e coordenadas de textura para cada tipo de figura, uma tarefa essencial para a renderização dos modelos.

Ao longo do desenvolvimento, conseguimos implementar com sucesso todas as funcionalidades solicitadas para esta fase, além de algumas extras, o que nos deixou satisfeitos com os resultados alcançados.

Por fim, consideramos que conseguimos consolidar os conhecimentos adquiridos nas aulas de Computação Gráfica, realizando um trabalho adequado.

References

1. Solar-System: Planets JSON, `https://github.com/Lazzaro83/Solar-System/blob/master/planets.json`