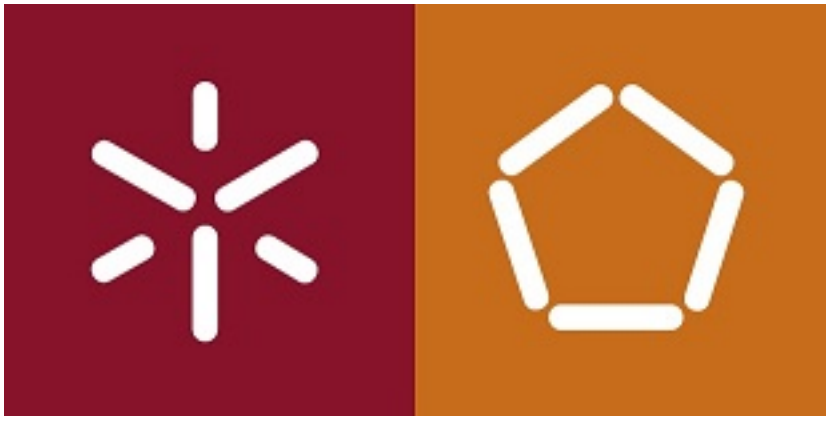

Universidade do Minho
Engenharia Informática

Trabalho Prático
Computação Gráfica
2º Semestre 2021/2022

Group 11

[A93313] Diogo Miguel - [A93180] Diogo Rebelo - [A84668] Francisco Peixoto - [A76603] Joel Araújo

Relatório CG - Fase 4

Junho, 2022

Resumo

Este relatório é alusivo à resolução da quarta fase do projeto da cadeira de computação gráfica, a última, cujo objetivo é evoluir o gerador de primitivas para incluir as coordenadas relativas às normais e às texturas de cada figura e, por sua vez, o *engine*, de forma a incluir essas diferenças nas representações gráficas de cada cena construída. Além das coordenadas, também trabalhamos sobre as luzes dos materiais e possíveis fontes de luz, de modo a enaltecer a aprazibilidade do resultado e a incluir todos os recursos básicos que um projeto nesta área deve apresentar.

Índex

Índex de Figuras

1 Cálculo das Normais

De forma a possibilitar a reflexão da luz nos objetos, as suas normais devem estar bem definidas e acompanhar o processo de desenho das mesmas. Para isso, em cada primitiva desenvolvida no nosso gerador, tivemos de acrescentar também as suas normais. Neste ponto, estas são constituídas por triângulos, cada um com três pontos e três normais.

```
1 triângulo = 3 vértices + 3 normais
```

```
-0.5000,-0.5000,0.5000, -0.4000,-0.5000,0.4000, -0.4000,-0.5000,0.5000,
↪ -0.5774,-0.5774,0.5774, -0.5298,-0.6623,0.5298, -0.4924,-0.6155,0.6155,
```

Figure 1: linha com pontos de vértice e normais

- **Plano:** No caso do plano, como estamos a falar de uma superfície horizontal, ou seja, que reside no plano x0z, os vetores normais presentes nos vértices serão sempre (0,1,0). Nos outros casos x0y e y0z, os vetores normais correspondentes serão (0,0,1) e (1,0,0), respetivamente.
- **Box:** Para a box, os valores serão idênticos aos do plano, ou seja, constantes para cada face, onde:
  - Face frontal (Plano x0y): vetor (0,0,1);
  - Face traseira (Plano x0y): vetor (0,0,-1);
  - Face esquerda (Plano y0z): vetor (1,0,0);
  - Face direita (Plano y0z): vetor (-1,0,0);
  - Face de cima (Plano x0z): vetor (0,1,0);
  - Face de baixo (Plano x0z): vetor (0,-1,0);
- **Cone:** No caso da base do cone, visto que a base está contida no plano x0z, o vetor normal dos pontos da base é sempre (0,-1,0). No entanto, nos pontos presentes nas faces, o vetor normal é calculado apartir da normalização do próprio ponto.
- **Sphere:** Para a sphere, devido a esta ser construída com origem no ponto (0,0,0), as normais da mesma são os próprios pontos normalizados, facilitanto bastante o seu cálculo.
- **Patch:** O cálculo das normais para os pontos do *patch* é feito utilizando as derivadas dos vetores auxiliares *'u"* e *'v'*, utilizados aquando da construção das superfícies de *Bezier*. Deste modo, como já temos dois vetores normalizados, utilizamos o produto cruzado, *cross product*, entre ambos e conseguimos obter a normal nesse ponto.

```
Point_3D Point_3D::crossProduct(Point_3D p) {

    return Point_3D(    this->y * p.getZ() - this->z * p.getY(),
                        this->z * p.getX() - this->x * p.getZ(),
                        this->x * p.getY() - this->y * p.getZ()
                    );
}
```

Figure 2: Cross product

2 Cores

Para conseguirmos lidar com as cores associadas aos objetos do mundo, tivemos de atentar nas componentes que lidam com a reflexão da luz, de maneira a definir que cor é que as suas superfícies refletiam mediante os diferentes tipos de reflexão.

Assim, atentamos quatro componentes básicas:

- **Difusa:** simula o impacto que uma fonte de luz tem sobre um objeto, sendo esta a componente visualmente mais significativa do modelo de iluminação. Quanto mais uma parte do objeto está virada para essa fonte, mas brilhante esta se torna.

```
// diffuse
float l2[] = { diffR, diffG, diffB, 1.0 };
glMaterialfv(GL_FRONT, GL_DIFFUSE, l2);
```

Figure 3: Estrutura da difusa

- **Especular:** simula o ponto de luz brilhante, normalmente branco, que aparece sobre objetos brilhantes. Esta faz-se acompanhar de uma variável complementar, *shininess*, que indica o tamanho e precisão dessa mancha, variando o seu valor de 0 até 128 (quando maior o valor, mais pequena e precisa será essa mancha).

```
// specular
float l3[] = { specR, specG, specB, 1.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, l3);
glMaterialf(GL_FRONT, GL_SHININESS, shininess);
```

Figure 4: Estrutura especular

- **Ambiente:** simula a luz indireta que um objeto recebe de outros objetos ao seu redor, servindo como uma constante de iluminação para garantir que o objeto terá sempre alguma cor.

```
// ambient
float l1[] = { ambiR * ambientStrength, ambiG *
↪  ambientStrength, ambiB * ambientStrength, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, l1);
```

Figure 5: Estrutura ambiente

- **Emissiva:** como é o caso do sol, há objetos no mundo que emitem luz para os outros ao seu redor, sendo essa a função desta componente emissiva, simular essa mesma emissão.

```
// emissive
float l0[] = { emissR, emissG, emissB, 1.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, l0);
```

Figure 6: Estrutura emissiva

Assim, para cada primitiva que esteja a ser computada, esta terá uma entidade intrínseca chamada ***Color***, cujo objetivo é armazenar os valores associados a estas componentes, fazendo uso de valores *default* quando não são especificados no ficheiro XML.

Para as executarmos, prévio ao respetivo desenho, indicamos com o *openGL* os valores respetivos.

```
void Color::Apply() {

float ambientStrength = 0.1f;

// diffuse
float l2[] = { diffR, diffG, diffB, 1.0 };
glMaterialfv(GL_FRONT, GL_DIFFUSE, l2);

// emissive
float l0[] = { emissR, emissG, emissB, 1.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, l0);

// specular
float l3[] = { specR, specG, specB, 1.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, l3);
glMaterialf(GL_FRONT, GL_SHININESS, shininess);

// ambient
float l1[] = { ambiR * ambientStrength, ambiG * ambientStrength,
↪  ambiB * ambientStrength, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, l1);
}
```

Figure 7: Definição da entidade *Color*

3 Coordenadas de texturas

As coordenadas de textura constam de pontos em duas dimensões (2D) em que, quer no eixo do x quer no eixo do y, o valor mínimo é 0 e o valor máximo é 1. Esses pontos tem como objetivo indicar que parte da textura se aplica num dado ponto da figura.

Usando como exemplo a figura mais simples, o plano, podemos ver abaixo um excerto de código implementado de modo a calcular os pontos de textura de um plano.

```
s = (float)i / divisions;
t = (float)j / divisions;
texs.push_back(Point_2D(s, t)); // tex coords
```

Figure 8: Calculo de textura do plano

Neste caso, se pensarmos num plano com n divisões, em que i está contido em [0,n] e representa uma das divisões horizontais do plano e j contido também em [0,n] referente ás divisões verticais, podemos descobrir os pontos de textura dividindo esses valores pelo número total de divisões que temos, visto que i varia entre 0 e o número de divisões, garantimos assim que esses valores se situam entre 0 e 1.

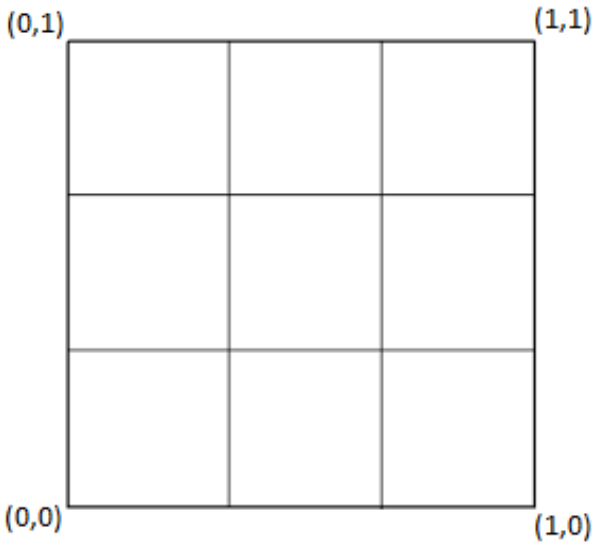

Figure 9: Plano com número de divisões 3

No caso da figura, o ponto de textura inferior direito é (1,0) visto que nesse caso o i (número de divisões verticais) é igual a 3, e o valor de j (número de divisões verticais) é 0. Assim, o valor das coordenadas de textura serão (3/3,0/3) = (1,0).

Como a box aplica vários planos, o seu método de cálculo de texturas é idêntico e no caso da esfera a única alteração é, visto que desenhamos a esfera de cima para baixo, as coordenadas no eixo dos y das texturas referente à mesma necessita de ser invertida, ou seja, subtrair a 1 o resultado obtido.

4 Texturas

Para aplicar as texturas, precisamos primeiro de proceder à leitura das coordenadas associadas à primitiva em questão e depois de uma imagem que sirva como textura, efetivamente.

Numa primeira abordagem, após a leitura dos pontos, temos de ativar um estado que indica à máquina de estados do *openGL* que estaremos a trabalhar com escrita e desenho de texturas,

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY).
```

De seguida, damos uso à biblioteca fornecida do *devil* para carregar a imagem, na qual especificamos a orientação do eixo de coordenadas utilizado, *'s'* e *'t'*,

```
ilOriginFunc(IL_ORIGIN_LOWER_LEFT),
```

associámo-la a um identificador, guardado na primitiva, com a função ***glBindTexture*** e assumimos um tratamento por *mipmapping* onde escolhemos a opção de tratamento linear para ambas a escolha de pixeis da textura a utilizar, assim como qual textura. No fim deste tratamento inicial, é necessário dar *unbind* da textura, visto que já paramos de trabalhar na mesma.

O próximo passo a tomar é associar o *vbo* dos pontos da textura à nossa primitiva, tal como fazemos com os pontos e as normais, e, no processo de desenho, voltar a dar um ***glBindTexture*** ao *id* previamente guardado, associar o *vbo* respetivo e a função de desenho fica com tudo o que precisa para a respetiva computação do desenho, lembrando que se deve dar sempre *unbind* às propriedades que não serão mais utilizadas.

5 Formato ficheiro .3d

Uma vez que chegamos a um ponto onde consideramos que não seriam realizadas mais mudanças significativas no formato dos nossos ficheiros *.3d*, definimos que em cada linha estará representada a informação alusiva a cada triângulo, ou seja,

```
1 triângulo = 3 pontos + 3 normais + 3 coordenadas de textura.
```

```
    -0.0000,1.0000,-0.0000, 0.0000,0.9921,0.1253, 0.0312,0.9921,0.1214,
↪   -0.0000,1.0000,-0.0000, 0.0000,0.9921,0.1253, 0.0312,0.9921,0.1214,
↪   0.0400,0.0400, 0.0000,0.0000, 0.0400,0.0000

    -0.0000,1.0000,-0.0000, 0.0312,0.9921,0.1214, 0.0604,0.9921,0.1098,
↪   -0.0000,1.0000,-0.0000, 0.0312,0.9921,0.1214, 0.0604,0.9921,0.1098,
↪   0.0800,0.0400, 0.0400,0.0000, 0.0800,0.0000

...
```

Figure 10: linhas com vértices, normais e coordenadas de textura

Cada vértice/ponto é apresentado através das coordenadas XYZ (os três primeiros pontos são os nove primeiros valores de cada linha no ficheiro *.3d*), de seguida temos os pontos normalizados que, por sua vez, também são três. No fim da linha, encontramos os últimos valores referentes às coordenadas de texturas, diferindo no formato dos pontos prévios por apenas se considerarem dois eixos, ao invés de três.

6 Fonte de Luz

De forma a corretamente simular os cenários pretendidos, o grupo introduziu o conceito de luz no *Engine*, as alterações feitas consistiram na criação de um módulo referente à Luz.

Foram usadas as bibliotecas do *openGL* de forma a obtermos corretamente os efeitos de luz pretendidos e é importante realçar a teoria por trás da iluminação das primitivas, dado que não só é importante introduzir no contexto as fontes de luz, mas também a forma como a luz interage com os diferentes materiais.

A classe Luz é constituída por três estruturas principais, que armazenam os dados sobre as possíveis fontes de luz a utilizar, e ainda um contador que permite até oito fontes de luz em simultâneo.

```
class Lights{

    private:
        int counter;
        vector<PointLight> pointLights;
        vector<DirectionalLight> directionalLights;
        vector<SpotLight> spotLights;
    ...
}
```

Figure 11: Estrutura da Classe Lights

```
void Lights::Apply() {
    int l1  = 0;

    bool never = true;
    for (PointLight pl : pointLights){
        l1 = this->getCounter();
        if(l1!=-1){
            never = false;
            pl.Apply(l1);
            setCounter(counter+1);
        }
    }
    ...
}
```

Figure 12: Método que aplica as Luzes

```
void PointLight::Apply(int c) {

    glEnable(c);
    float pos[] = { posX, posY, posZ, 1.0f };
    glLightfv(c, GL_POSITION, pos);
}
```

Figure 13: Método que aplica PointLights

Como podemos ver, no método ***Lights::Apply()*** estamos a utilizar *glEnable()* aplicado ao contador atual da luz, que utiliza GL_LIGHT0 caso só seja utilizada uma luz, GL_LIGHT1 caso sejam utilizadas duas fontes de luz, etc.

Tal como mencionado anteriormente, a aplicação da luz só é possível graças ao VBO que contém as coordenadas normais de cada primitiva, esses pontos juntamente com o tipo de material de cada primitiva geram então a forma como uma fonte de luz interage com os objetos.

7 Conclusão

Dado por concluída esta quarta e final fase do projeto, consideramos ter obtido uma boa prestação tanto na criação e desenvolvimento das primitivas, como a respetiva apresentação gráfica e manuseio do mundo com a nossa câmara. Este foi um trabalho que, por não termos a maior proficiência na área, nos obrigou a um grande compromisso, no que diz respeito ao estudo das matemáticas que sustentam cada parte da arquitetura, assim como da biblioteca do *openGL*.

No futuro, como temos uma maior perceção de cada parte que constitui um organismo deste tamanho, temos noção que poderíamos organizar melhor cada uma das partes da construção da primitiva, o que cada uma contém nas suas variáveis, assim como o tratamento da câmara e a reflexão ao uso de teclas com significados específicos. De maneira geral, consideramos que conseguimos evoluir do ponto de partida em que estamos, até usar algoritmos de computação mais complexos, mas consideramos que o produto final que entregamos contém tudo o que era necessário e expectável.

8 Anexos

Figure 14: Exemplo 1 - Sistema Solar obtido

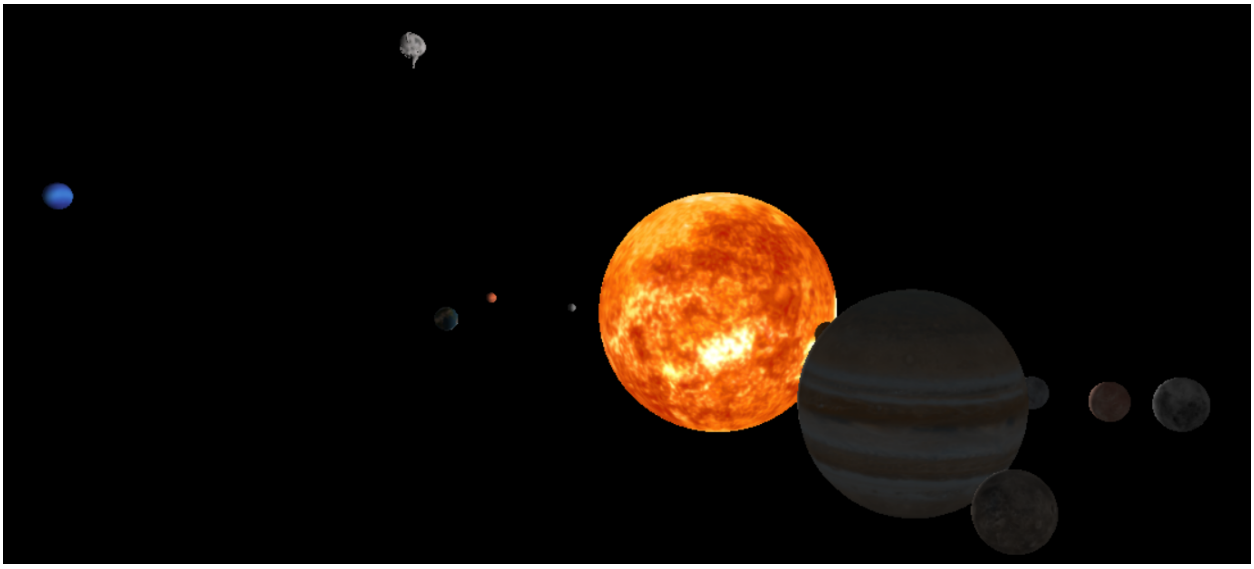

Figure 15: Exemplo 2 - Sistema Solar obtido

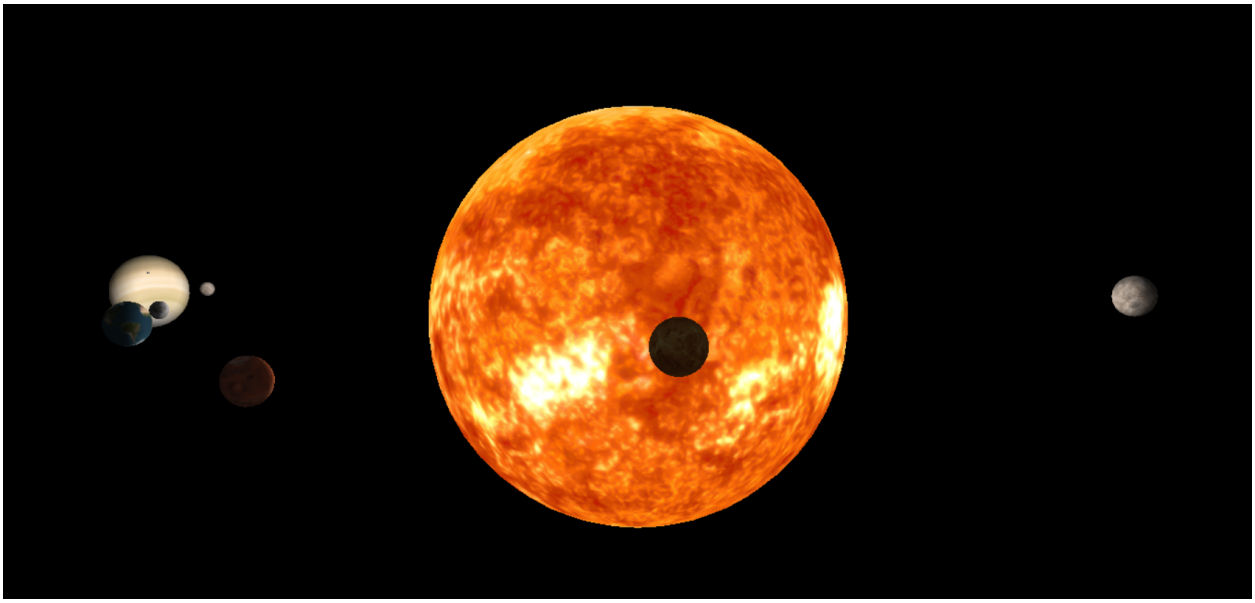

Figure 16: Exemplo 3 - Sistema Solar obtido

Figure 17: Exemplo 4 - Luz sem texturas

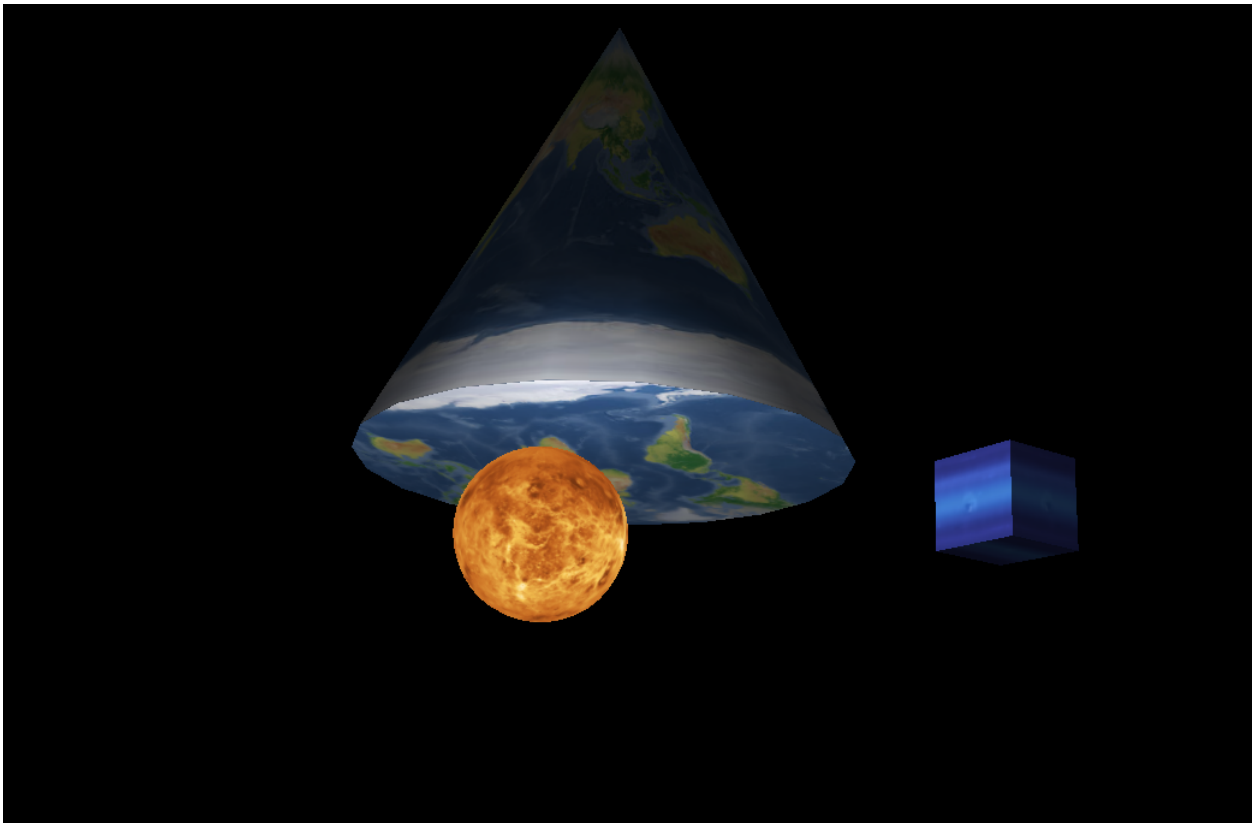

Figure 18: Exemplo 5 - Luz aplicada a vários objetos

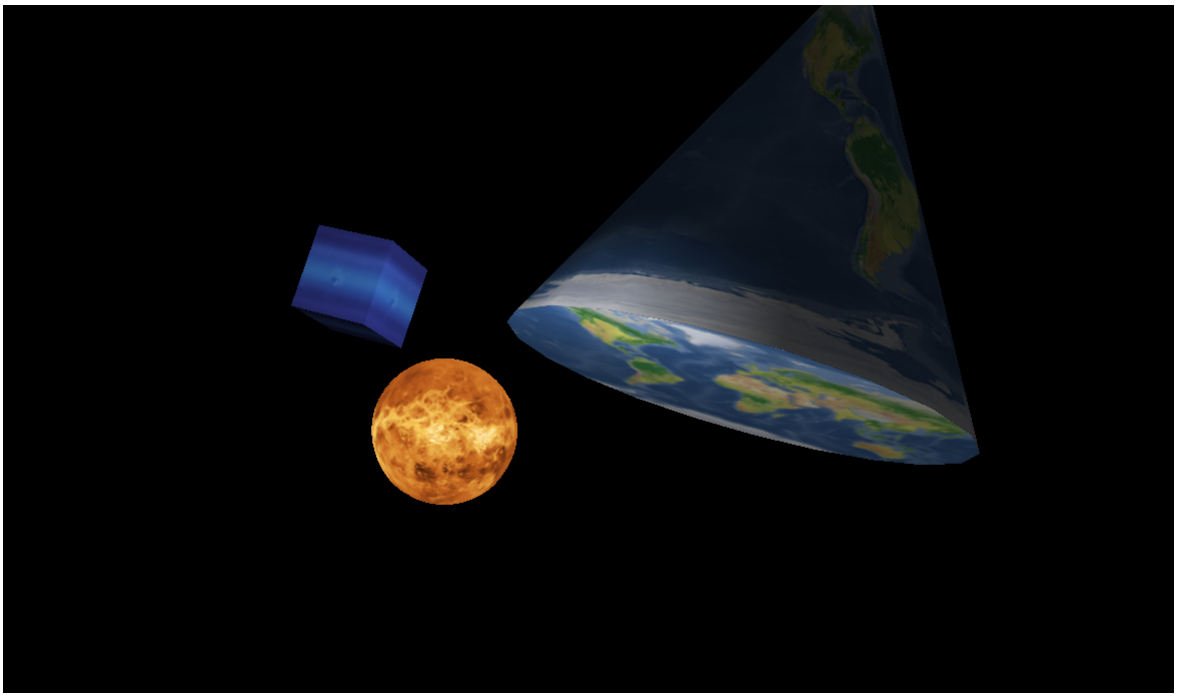

Figure 19: Exemplo 6 - Luz aplicada a vários objetos num ângulo diferente

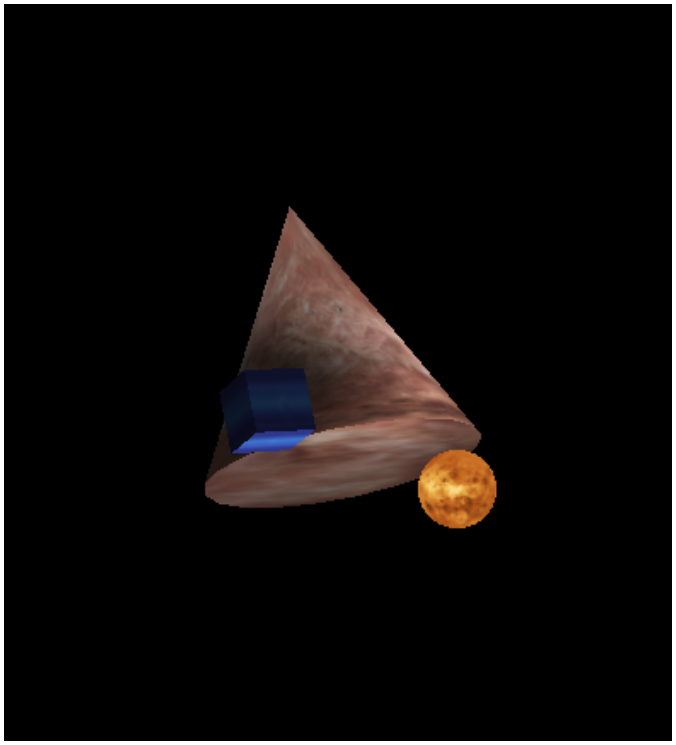

Figure 20: Exemplo 6 - Várias fontes de luz aplicadas a vários objetos