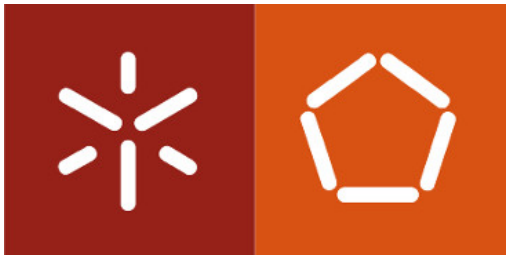

Universidade do Minho

Mestrado integrado em Engenharia Informática

Trabalho Prático - 4ª Fase

Grupo 35

André Rafael Olim Martins, A84347

Filipe Emanuel Santos Fernandes, A83996

José Diogo Xavier Monteiro, A83638

Susana Vitória Sá Silva Marques, A84167

Computação Gráfica

3º Ano, 2º Semestre

Departmento de Informática

30 de maio de 2021

Índice

1 Introdução

Nesta quarta e última fase do trabalho prático foi necessário realizar diversas modificações aos diversos componentes do nosso programa bem como implementar novas funcionalidades, assim de seguida são descritas todas as alterações de maior relevo.

No *Generator*, serão feitas algumas alterações de maneira a que este seja capaz, de através de uma imagem, ter a capacidade de calcular as normais e a textura para os vários vértices das diferentes primitivas geográficas.

No *Engine*, são incluídas novas funcionalidades e algumas modificações. Como o ficheiro XML tem também agora informações em relação à iluminação do cenário, o *Parser* responsável por ler esses ficheiros é alterado e também é alterada a forma de processamento da informação para construir o cenário. Para além disso, sofre mais algumas alterações nos VBOs e com a preparação das texturas durante o processamento do ficheiro de configuração.

2 Engine

2.1 Classe: Transformações()

Sendo esta classe que guarda toda a informação relativa ao conjunto de transformações aplicadas a um grupo (filhos incluídos) apenas faria sentido serem aqui implementados todos os processos de associação de texturas aos modelos. Logo, foram criados dois VBOs para guardar informação quanto às normais dos pontos e quanto às coordenadas de textura e o VBO dos que gurdavam os pontos em fases anteriores também passou a ser implementado nesta classe. A informação sobre a textura, para além das coordenadas, inclui o seu ID e dimensões da imagem a ser texturizada. Ao mesmo tempo, foi alterada a maneira de interpretação dos mesmos, a inicialização dos mesmos e o carregamento das texturas durante o processo de leitura do ficheiro de configuração. Isto foi conseguido através da criação dos métodos: desenha(), use_VBO() e newText(), unicamente se o conjunto tiver uma textura associada.

2.2 Leitura

Dado que os ficheiros correspondentes aos modelos sofreram alterações foi também necessário alterar a componente de leitura do *engine* de forma a identificar a nova informação descrita nos mesmos, mais uma vez. Para além das alterações aos ficheiros .3d os ficheiros *xml* que descrevem uma cena podem também agora conter a descrição de luzes.

2.3 Luz

O primeiro passo para utilizar luzes no programa passa por descrever as mesmas, desde qual o seu tipo até aos seus parâmetros, deste modo é necessário que seja especificado a quando a leitura da *scene* quais as diversas componentes dessa luz, e posteriormente que essas mesmas definições sejam guardadas dentro do programa, para que possam depois ser utilizadas a quando o desenho da imagem.

Para as propriedades de iluminação se manterem é necessário o cálculo correto das normais dos diversos modelos. O grupo optou por, em vez de calcular as

normais para cada triângulo, calcular as mesmas mas para cada ponto constituinte da figura desejada, de maneira a conseguir uma aproximação mais credível da realidade.

Ao mesmo tempo, para conseguir esta aproximação realista, é necessário abordar diferentes parâmetros para a cor. Esta é constituída por 4 componentes:

- Diffuse colour: Corresponde à cor que um objeto apresenta quando exposto a luz branca, ou seja, define a cor do próprio objeto e não a reflexão da luz no mesmo.
- Ambient colour: Cor que um objeto apresenta quando não está exposto a nenhum feixe de luz.
- Emissive Colour: Cor que um objeto emite após ser exposto a luz
- Specular light: Brilho que aparece nos objetos quando iluminados

Ao construir um sistema de iluminação é importante ter em conta a origem da luz, pois é esta que vai proporcionar os ângulos de iluminação dos diferentes constituintes do nosso sistema solar. Ao mesmo tempo, também, é preciso ter em conta a direção da mesma. Assim, existem duas opções para a implementação deste sistema, sendo estas:

- Luz proveniente de um ponto que emite feixes de luz em todas as direções
- Luz proveniente de um ponto que emite um único feixe de luz direcionado

Desta maneira, a distinção entre estas duas situações é dada por um array **pos** constituído pelas coordenadas. No nosso caso, o pretendido é que a luz seja proveniente do Sol. Para isto, a posição será (0,0,0) e, como queremos que este provenha unicamente da estrela, a sua direção será para todos os lados, por isso, 1 (isPoint). Logo, a representação do array é **pos[4]=0,0,0,1**. Em relação aos outros componentes:

```
GLfloat amb[3]= {0.0,0.0,0.0};
GLfloat diff[4]= {1.0f,1.0f,1.0f,1.0f};
GLfloat matt[4]= {5,5,5};
```

Estes valores, dado que não se alteram ao longo da execução podem simplesmente ser definidos de imediato. Para que posteriormente a luz funcionasse correctamente é em primeiro lugar necessário activar a utilização de Luz no *OpenGL*, e posteriormente assegurar que tudo aquilo que é desenhado possui as corretas normais associadas, de forma a que esta seja apresentada corretamente. Isto pode ser alcançado definindo um VBO associado a cada modelo, semelhante àquele que contem os vértices mas desta vez contendo todas as normais correspondentes a cada vértice.

2.4 Textura

De forma semelhante às normais mencionadas no ponto anterior, para podermos definir as coordenadas de textura basta adicionarmos mais um VBO a cada modelo que contenha as mesmas, com um pequeno detalhe de que é apenas são necessários dois valores por cada vértice do modelo.

2.5 Processo de Renderização

O processo usado nesta fase foi maioritariamente semelhante ao da fase anterior, sendo acrescentadas funcionalidades em relação às texturas. Assim, antes de serem desenhadas as transformações em relação a um grupo é feita a verificação se este possui textura ou não, desenhando apenas se tiver, desta maneira é-nos permitido ter satélites com textura e satélites apenas com cor.

2.6 Câmara FPS

Para uma melhor interação com o modelo apresentado foi desenvolvida uma câmara FPS. Foi então criada a classe `Camera` que contém os três seguintes pontos:

1. Posição - que irá indicar a posição atual da câmara
2. Direção(`front`) - representa a direção da câmara
3. Up - que servirá para indicar o ângulo da câmara

Esta classe irá ainda conter todos os métodos que irão utilizar e manipular estes pontos.

2.6.1 Movimento

Por se tratar de uma câmara *first person shooter*, foi desenvolvido primeiramente o seu movimento (deslocamento) na cena. A câmara terá três eixos:

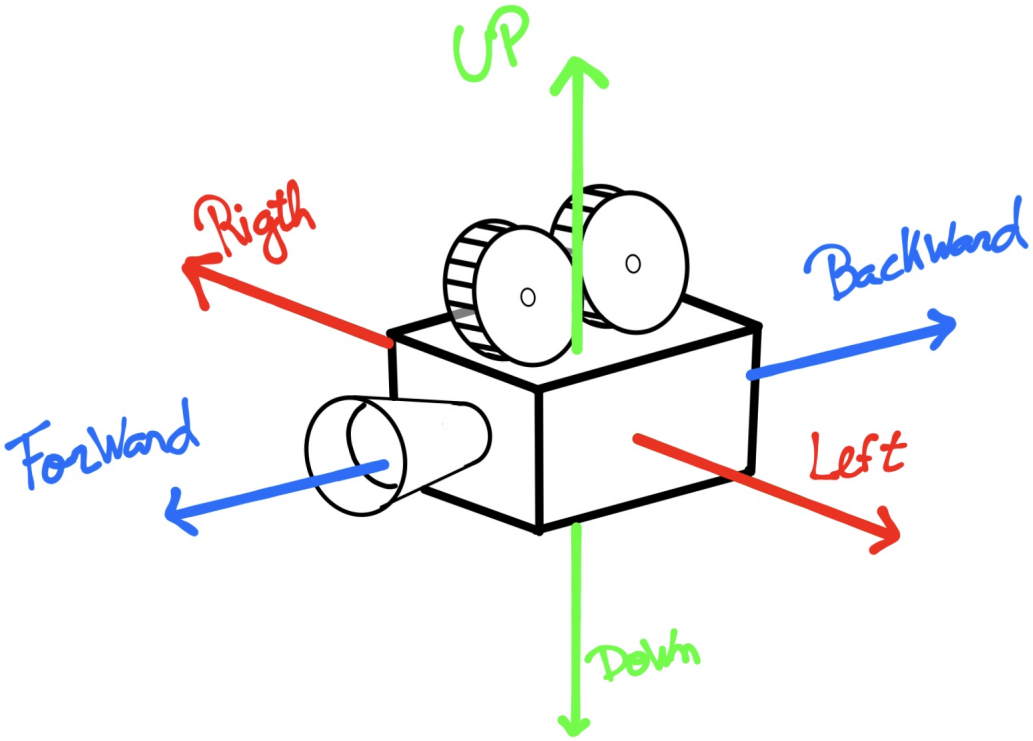

Figura 1: Eixos para o movimento da câmara.

Estes eixos demonstram como será realizado o deslocamento da câmara na cena. Como podemos ver na imagem, a câmara poderá deslocar-se para a frente e para trás (usando as teclas W e S, respetivamente), poderá realizar movimento lateral, através das teclas A (esquerda) e D (direita), e por fim, poderá deslocar-se

para cima ou para baixo, utilizando as teclas especiais, seta para baixo e seta para cima.

Com estas definições em mente, passamos os seguintes pontos à `gluLookAt`:

```
gluLookAt(posicao, posicao + front, up);
```

Como podemos ver o *look at point* é a soma entre a posição e a direção(front) em que estamos a olhar.

2.6.2 Look around

Para o "olhar"foram usadas *Euler Angles* tendo sido usados o *yaw* para permitir à câmara olhar para os lados e o *pitch* para dizer o quando a câmara pode olhar para cima e para baixo.

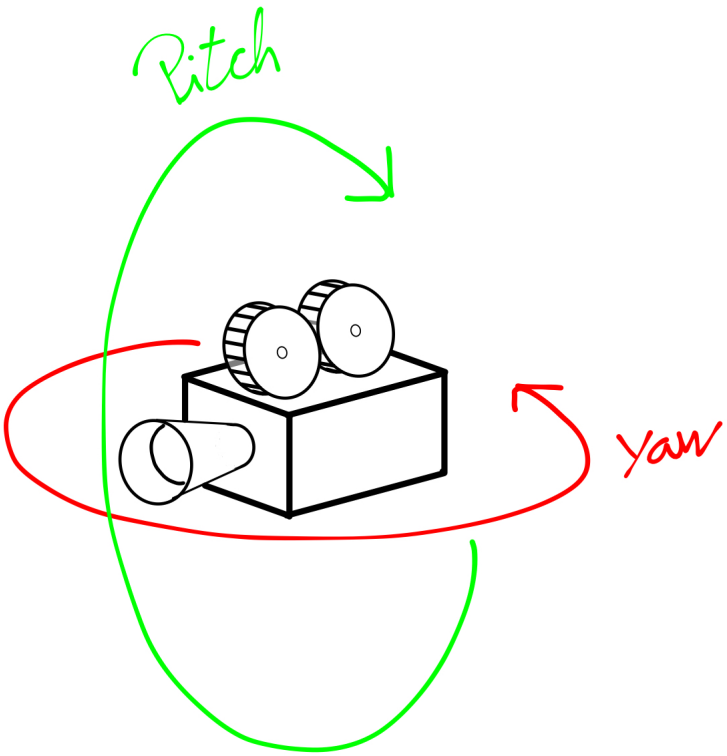

Figura 2: Euler Angles

Para o cálculo do *yaw* e do *pitch* foram usadas as coordenadas do rato antes e

depois do movimento, onde posteriormente será atualizado o ponto direção(*front*). Para a captação dos movimentos do rato foi usado o:

```
glutPassiveMotionFunc(mouseMotion);
```

3 Generator

Nesta última fase do nosso projeto, foram nos pedidas novas atualizações ao *Generator*. Neste caso, seria a adição do cálculo das normais dos objetos, de modo a que se torne possível a aplicação da iluminação e também o cálculo das coordenadas onde serão aplicadas as texturas.

Desse modo, a "arquitetura"dos nossos modelos 3D acabou por sofrer alterações. Previamente, os únicos pontos imprimidos seriam as coordenadas para cada ponto que definia os triângulos "base"de cada estrutura.

```
Plano:
    Coordenada1.x Coordenada1.y Coordenada1.z
    Coordenada2.x Coordenada2.y Coordenada2.z
    Coordenada3.x Coordenada3.y Coordenada3.z
...
```

Nesta fase, vimo-nos na necessidade de adicionar duas linhas para cada ponto de cada plano, uma que codifica o vetor normal ao ponto e outra que trate das texturas.

```
Plano:
    Coordenada1.x Coordenada1.y Coordenada1.z
    NormalP1.x NormalP1.y NormalP1.z
    TextP1.a TextP1.b
    Coordenada2.x Coordenada2.y Coordenada2.z
    NormalP2.x NormalP2.y NormalP2.z
    TextP2.a TextP2.b
    Coordenada3.x Coordenada3.y Coordenada3.z
    NormalP3.x NormalP3.y NormalP3.z
    TextP3.a TextP3.b
...
```

3.1 Normais

Começando pelo cálculo das normais, o método evidentemente sofrerá alterações mediante as estruturas sobre as quais pretendemos realizar os cálculos. Estes valores, tal como mencionamos previamente, serão utilizados pelo Engine de modo a que seja possível realizar a aplicação da iluminação nos nossos modelos.

3.1.1 Ring

Optamos por começar por esta figura devido à sua própria simplicidade. Considerando que se assenta sobre o eixo xz, é evidente que a normal à superfície será constante e igual para todos os pontos, ocorrendo apenas variação para o eixo y. Claro que depois terá de ser alterada mediante a orientação do plano, a face "superior"ficará com um valor positivo e a face inferior com o oposto.

O seguinte exemplo demonstra a diferença no cálculo entre um dos pontos da superfície superior e inferior:

```
fprintf(f, "%f %f %f \n", sin(i)*radius_int, 0.0, cos(i)*radius_int);
fprintf(f, "%f %f %f \n", sin(i) * radius_int, 1.0, cos(i) * radius_int);

fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
fprintf(f, "%f %f %f \n", sin(i) * radius_out, -1.0, cos(i) * radius_out);
```

3.1.2 Sphere

O cálculo das normais para a esfera acaba por ser, tal como no caso do anel, bastante simples de resolver. Esta figura possui o seu centro na origem e qualquer um dos seus pontos está situado a uma distância igual ao raio da origem. Por esta mesma razão, conseguimos perceber que o vetor que faz a ligação de qualquer ponto à origem da esfera é perpendicular à tangente nesse ponto, ou seja, a definição da normal. Assim, conseguimos calcular de forma imediata o valor da normal para cada um dos pontos aplicando a normalização.

A implementação deste cálculo pode ser evidenciada no seguinte excerto:

```
fprintf(f,"%f %f %f\n",esfericaX(angulo,beta,radius),
```

```
                        proxima_rodela,
                        esfericaZ(angulo,beta,radius));
Vec3f v1 = Vec3f(esfericaX(angulo, beta, radius),
                 proxima_rodela,
                 esfericaZ(angulo, beta, radius));
v1.normalize();
fprintf(f,"%f %f %f\n",v1.x,v1.y,v1.z);
```

3.1.3 Bezier

Por fim, temos o cálculo das normais para as superfícies de Bezier. Neste caso, sabendo o que são normais e a sua definição, podemos entender que o que pretendemos é calcular a normal de todos os pontos da superfície de Bezier, tendo em conta que o resultado de todos esses cálculos resultara numa outra superfície completamente perpendicular à inicial.

Tendo chegado a essa conclusão, o que vamos precisar de fazer será calcular duas tangentes sobre um ponto da superfície, uma sobre o eixo u e outra sobre o eixo v. Assim, realizando o produto das tangentes que calcularmos iremos obter a normal para esse mesmo ponto.

Em termos matemáticos, a tangente de uma curva obtida através de uma função poderá ser calculada através da derivada dessa função.

Sabendo isto, passamos à derivação da função que computa os pontos numa patch de Bezier quando fornecidos um ponto **u** e um **v**. Sendo que esta função recebe dois parâmetros distintos, a derivação terá de ser feita com respeito a ambos os parâmetros. Assim, o resultado da derivação serão duas funções distintas, uma que calcula a tangente **u** e outra a tangente **v**. O resultado da derivação, evidentemente, terá semelhanças para ambas as tangentes, assim, uma terceira função surge:

```
Vec3f derBezier(const Vec3f *P, const float &t) {
    return -3 * (1 - t) * (1 - t) * P[0] +
           (3 * (1 - t) * (1 - t) - 6 * t * (1 - t)) * P[1] +
           (6 * t * (1 - t) - 3 * t * t) * P[2] +
```

```
            3 * t * t * P[3];
}
```

De seguida, temos a implementação das funções que realizam o cálculo das tangentes tal como calculamos:

```
Vec3f dUBezier(const Vec3f *controlPoints, const float &u, const float &v) {
    Vec3f P[4];
    Vec3f vCurve[4];
    for (int i = 0; i < 4; i++) {
        P[0] = controlPoints[i];
        P[1] = controlPoints[4 + i];
        P[2] = controlPoints[8 + i];
        P[3] = controlPoints[12 + i];
        vCurve[i] = calcBezierCurve(P, v);
    }
    return derBezier(vCurve, u);
}

Vec3f dVBezier(const Vec3f *controlPoints, const float &u, const float &v) {
    Vec3f uCurve[4];
    for (int i = 0; i < 4; i++) {
        uCurve[i] = calcBezierCurve(controlPoints + 4 * i, u);
    }
    return derBezier(uCurve, v);
}
```

Por fim, tendo todas as funções necessárias ao cálculo das normais, basta aplicarmos iterativamente para cada patch que formos calcular. Tal como referimos no inicio desta secção, primeiro calculamos as tangentes e depois normalizamos o produto das duas.

```
Vec3f dU = dUBezier(controlPoints,
                    m / (float)tessellation,
                    j / (float)tessellation);
```

```
Vec3f dV = dVBezier(controlPoints,
                    m / (float)tessellation,
                    j / (float)tessellation);
Normals[z] = dU.crossProduct(dV).normalize();
```

3.2 Coordenadas das texturas

Estas coordenadas foram feitas tendo em conta o eixo (u,v) das texturas, mapeando assim os vários pontos da figura para este eixo. Deste modo quando é feita a renderização, a textura é aplicada à figura.

Para o *teapot*, este mapeamento foi feito *patch* a *patch* devido à enorme complexidade da figura, logo não seria viável o mesmo método que foi usado na esfera e no anel de Saturno. Sendo as novas coordenadas directamente proporcionais ao nivel de tessellation do teapot.

Nestes últimos 2, o processo foi mais directo porém o raciocínio é o mesmo, tendo apenas em conta as normais, o número de *slices* e *stacks* das figuras em vez do nivel de tessellation.

4 Demonstração

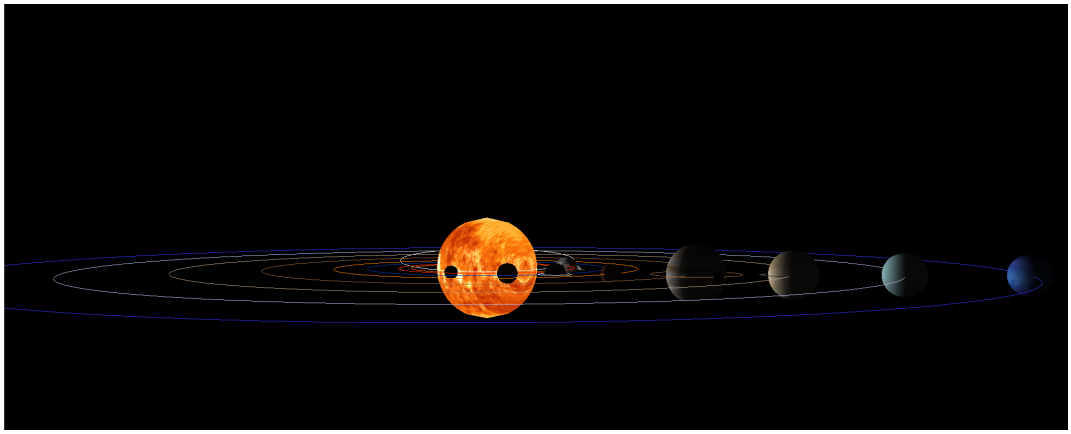

Figura 3: Print lateral

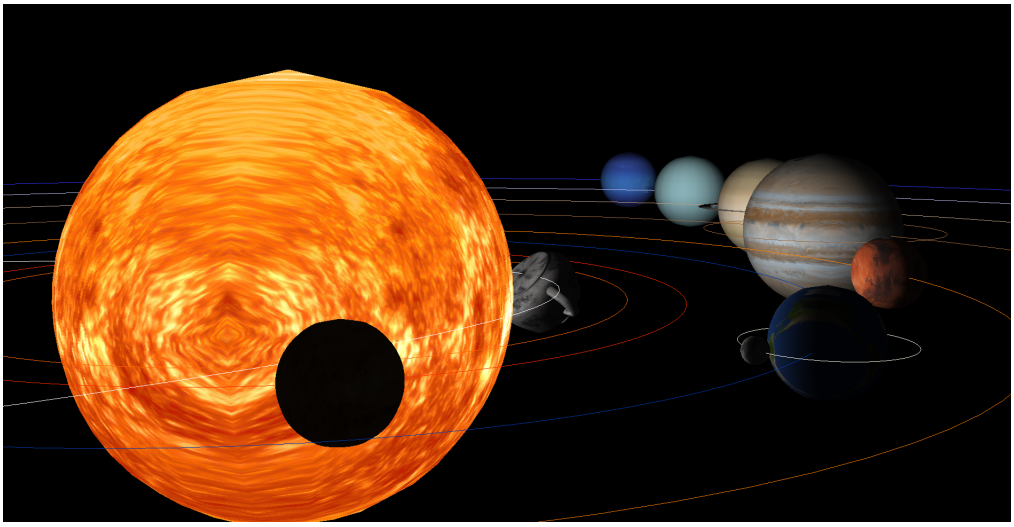

Figura 4: Print perspetiva

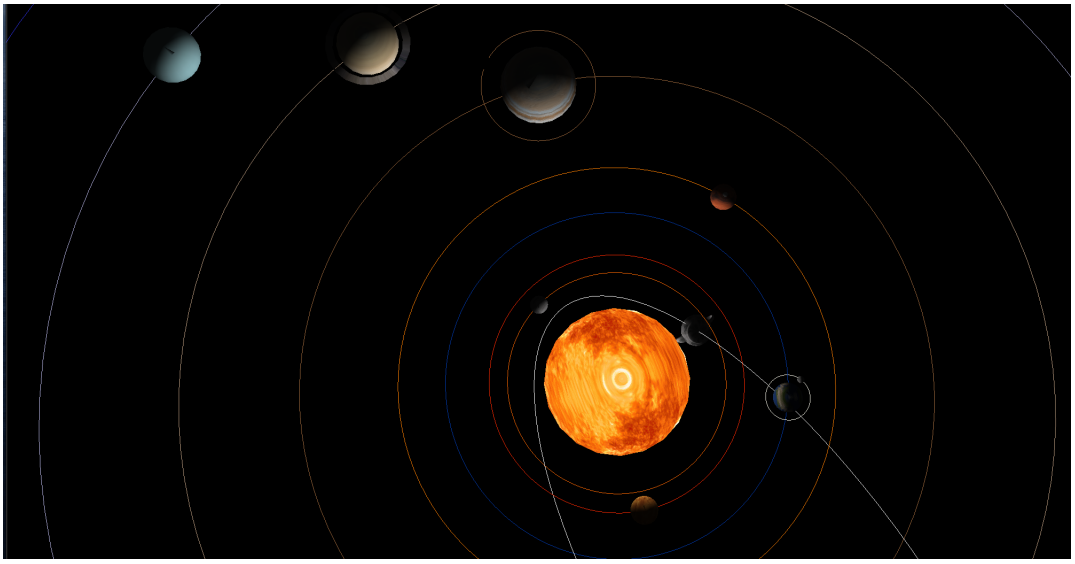

Figura 5: Print topo

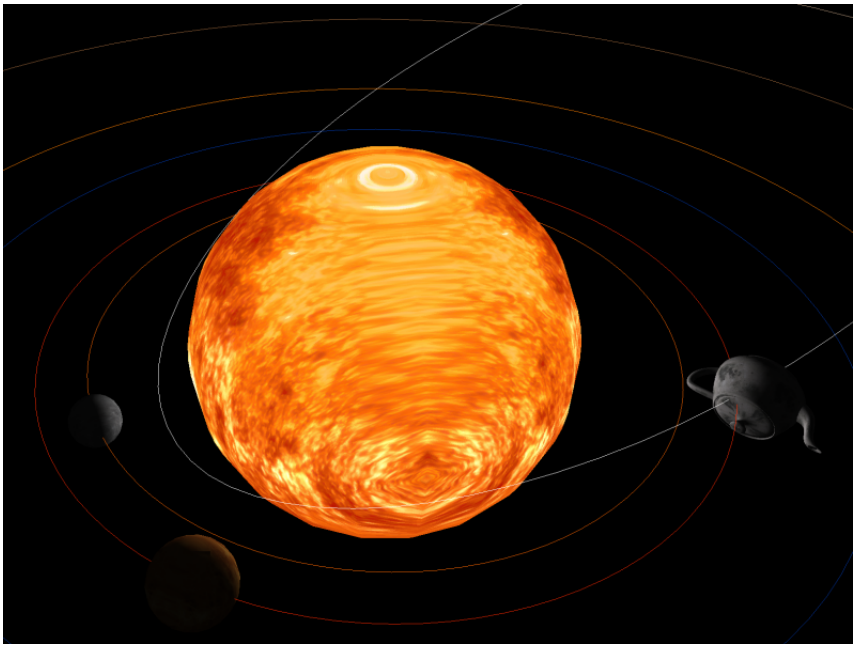

Figura 6: Sol e cometa

5 Conclusão

Nesta quarta e final fase do projecto, foi-nos pedido para implementar a iluminação e aplicação de texturas nas diversas figuras usadas na nossa demonstração do sistema solar. Para isto efectuamos alterações no *Generator* para que este fosse capaz de calcular as normais e as coordenadas das texturas.

O *Engine* tambem sofre modificações de modo a capacitar o mesmo da leitura e execução dos novos dados gerados no sistema e desta forma apresentar graficamente uma demonstração do sistema solar com texturas no vários planetas, sol, lua e no teapot que serve de cometa.

No ponto de vista do grupo, e após uma análise sucinta dos resultados obtidos, achamos que fomos capazes de pôr em uso os conhecimentos lecionados de forma a criar a demonstração e a implementar transformações de forma satisfatória.