Relatório Computação Grafica - Fase 4

Marco Sousa[1,2[62608]], José Malheiro[1,2[93271]], and Miguel Fernandes[1,2[94269]]

[1] Universidade do Minho
[2] Licenciatura em Engenharia Informática, Braga, Portugal

Resumo Com a construção do esqueleto da cena principal, em conjunto com as animações dos elementos integrantes, o próximo passo seria a população do modelo com texturas e iluminação. No caso das texturas, para cada umas das figuras geométricas deve ser calculadas as respetivas coordenadas para mapear a imagem no objeto, utilizando os conceitos de *mipmapping*(*Nearest, on the lef & Linear, on the right*). Na tentativa de definir a iluminação, todas as normais de cada objeto devem ser calculadas.
Deste modo, o modelo do **sistema solar** será alterado para incluir estas alterações e produzir a maquete final.

Keywords: OpenGL · GLUT · Figuras Geométricas · 3D · C++ · Texturas · Iluminação · Normais · Coordenadas de Textura

1 Introdução

1.1 Contextualização

No seguimento da fase III do projeto da disciplina de Computação Gráfica da Licenciatura em Engenharia Informática da Universidade do Minho, foi proposta, numa última fase, a implementação de texturas e iluminação no projeto, através do cálculo das **coordenadas de textura** e **normais** a cada figura geométrica. Tudo irá culminar na atualização do modelo do sistema solar previamente construído com estes parâmetros.

1.2 Breve Descrição do Enunciado Proposto

O cerne da última fase (IV) do projeto implica alterações ao nível do *generator* e do *engine* previamente definidos. Deste modo, pretende-se:

generator Calcular as normais e coordenadas de textura para **cada vértice** das figuras geométricas.

engine Ativar as funcionalidades de iluminação e texturização, utilizando as normais e coordenadas de textura definidas.

No que toca ao *generator*, os novos ficheiros *.3D* devem ser alterados para incluir os vetores **normais** e as coordenadas de **textura** da respetiva figura geométrica. Deste modo, serão incluidas as *tags*:

normals Todos os vetores normais da figura geométrica.
texture Todas as coordenadas de textura da figura geométrica.

Relativamente à *engine*, devem ser criadas e implementadas as **cores** e texturas de cada figura geométrica permitida. Adicionalmente, será necessário definir a(s) luz(es) (tipo e parâmetros) para a cena principal. Deste modo, para cada modelo (*model*) do ficheiro *.XML* a ler, encontram-se as *tags*:

texture Inclui o *path* para a textura, no atributo *file*, a ser aplicada na figura geométrica.
color Para cada figura geométrica devem ser definidos os componentes:
diffuse Cor **difusa** do objeto - (Em *RGB*).
ambient Cor **ambiente** do objeto - (Em *RGB*).
specular Cor **especular** do objeto - (Em *RGB*).
emissive Cor **emissiva** do objeto - (Em *RGB*).
shininess Valor da refletividade da luz no objeto.

...sendo necessário utilizar as coordenadas de textura calculadas no generator para desenhar a imagem no objeto.

Para cada cena podem ser definidas a origem para uma ou mais luzes, do tipo:

point Especificada a posição da luz.
directional Epecificada a direção da luz.
spotlight Epecificadas a posição e direção das luzes, bem como o ângulo *cutoff*.

A partir destas alterações apresentar todo o sistema solar como um modelo texturizado e iluminado, obedecendo à estrutura previamente definida para a cena principal.

2 Trabalho Realizado

Funcionalidades Implementadas:

1. *Generator*
   (a) Gerar as normais para cada vértice das figuras geométricas. (2.1)
   (b) Gerar as coordenadas de textura para cada figura geométrica. (2.2)
   (c) Atualização do ficheiro *.3D*, com as *tags normals* e *texture*.(2.3)

ADICIONAL Criação de um círculo de asteróides. (2.3)

2. *Engine*
   (a) Implementação da iluminação:
      i. num *point*, através da sua posição.
      ii. *directional*, através da direção do foco.
      iii. de um *spotlight*, através da posição, direção e *cutoff*
   (b) Implementação de texturas, a partir do *path* da imagem a inserir.
   (c) Implementação das cores difusa, ambiente, especular, emissiva e o coeficiente de *shininess* do objeto.
3. Sistema Solar
   (a) Transição para um modelo iluminado e texturizado.
   (b) Adição do cinturão de asteroides.

2.1 Cálculo das Normais

Para permitir a iluminação de cada figura geométrica criou-se uma nova instância da classe `t_points normals(number_vectors)`, para armazenar todos os vetores normais.

Reutilizou-se a classe *t_points*, representando cada **vetor** com a classe `Point(x, y, z )`. ... apesar da nomenclatura não traduzir diretamente, vértices e vetores têm o mesmo *layout* para as suas coordenadas, evitando-se criar classes desnecessárias.

A variável *number_vectors* corresponde ao número total de **vetores** do objeto. ... uma vez que será necessário construir vetores para cada um vértice, corresponde ao número de vértices do objeto.

Plane Um plano, ou *plane*, é constituído por dois quadriláteros, idênticos, um orientado para o lado positivo do eixo Y e outro orientado para o lado negativo deste mesmo eixo.

Pode-se afirmar que:
Todos os vértices que se encontram na mesma superfície, têm a mesma normal.

Pelo que:

$$v1 \in plano_1 \tag{1}$$

$$v2 \in plano_1 \implies normal_{v1} = normal_{v2} \tag{2}$$

Neste sentido, só existem **2 vetores normais únicos**. Um para representar o plano, numa visão de cima, e outro para o representar numa visão de baixo.

Dado a ser construído sobre o plano XZ, estes dois vetores são estáticos:

$$vetorNormal_{planosuperior} = (0, 1, 0) \tag{3}$$

$$vetorNormal_{planoinferior} = (0, -1, 0) \tag{4}$$

Box Semelhante ao plano, a *Box* ou caixa, tem **6 superfícies planas**.

Deste modo, só existem **6 vetores normais** diferentes para cada vértice da figura geométrica.

A caixa encontra-se desenhada na origem, centrada, sendo que todas as suas faces são perpendiculares aos eixos XYZ(positivos e negativos). Consequentemente, a normal a cada plano vai ser paralela aos eixos.

Deste modo, apresenta:

$$vetorNormal_{planoxpositivo} = (1, 0, 0) \tag{5}$$

$$vetorNormal_{planoxnegativo} = (-1, 0, 0) \tag{6}$$

$$vetorNormal_{planoypositivo} = (0, 1, 0) \tag{7}$$

$$vetorNormal_{planoynegativo} = (0, -1, 0) \tag{8}$$

$$vetorNormal_{planozpositivo} = (0, 0, 1) \tag{9}$$

$$vetorNormal_{planoznegativo} = (0, 0, -1) \tag{10}$$

$$\tag{11}$$

Sphere O cálculo das normais na esfera é feito com base na seguinte proprieada geométrica: Se p é um vértice da esfera: $p = (raio * sin(\alpha) * cos(\beta), raio * sin(\beta), raio * cos(\alpha) * cos(\beta))$ e como a esfera está centrada no ponto (0, 0, 0), o vetor normal é obtido através da normalização do vetor n, que é igual a p - (0, 0, 0).

Cone O cálculo das normais do cone foram feitas com 2 abordagens diferentes, para a base, uma vez que esta é desenhado sobre o plano XZ, consequentemente a normal será a apontar para o Y negativo, isto é (0, -1, 0).

Para a lateral do cone, a normal foi obtida a partir do produto cartesiano de um vetor tangente à face lateral do cone com

Cylinder O cilindro é composto por 2 bases, sendo o a normal da base inferior o inverso da normal da base superior. Como este é construído sobre o XZ, a normal da base superior é (0, 1, 0) e da base inferior (0, -1, 0).

Para calcular as normais do parte lateral do cone, foi utilizada a seguinte propriedade geométrica:

Se p é um vértica da face lateral do cilindro: $p = (raio * sin(\alpha), y, raio * cos(\alpha))$ então: $n = (sin(\alpha), 0, cos(\alpha))$

Torus As normais do *torus* são calculadas de uma forma muito similar à das esferas, tendo os pontos:

$c = (raio * sin(\alpha) * cos(\beta), raio * sin(\beta), raio * cos(\alpha) * cos(\beta))$

$p = (tamanho * sin(\alpha) * cos(\beta), tamanho * sin(\beta), tamanho * cos(\alpha) * cos(\beta))$

em que raio é o a distância do centro(0, 0, 0) ao centro do *torus* e o tamanho é o raio interno do *torus*, então sabemos que $c + p$ e $c - p$ são vértice do *torus* com normais inversas. Desta forma, é possível concluir que a normal destes vértices são obtidas, respetivamente, pela normalização do vetor obtido a partir subtração do ponto p pelo ponto c e da subtração do ponto c pelo ponto p.

2.2 Cálculo das Coordenadas de textura

Foi necessário, para a implementação das texturas no modelo, calcular as coordenadas de textura para cada figura geométrica.

Neste caso, estas coordenadas são *2D*, pelo que não era viável utilizar novamente a classe *t_points*. Optou-se por utilizar um **vetor** de *floats*, `vector<float> p_textures`.

Segue-se para cada figura geométrica:

Plane Cada plano é construído com um comprimento, *l*, e número de divisões, *div*. Deste modo, para enquadrar nas coordenadas de textura, que oscilam entre 0 e 1, foi necessário criar um *step*. ... como tem o mesmo comprimento horizontal e verticalmente, apenas é criado um.

Este step é calculado como:

$$step = 1.0/num_{divisoes} \tag{12}$$

... pelo que temos a translação da distância de uma divisão no plano, para o eixo da textura a inserir.

Do mesmo modo que foram inseridos os vértices, seguindo a **regra da mão direita**, as coordenadas de textura foram inseridas no seu *vetor*. ... construíndo os triângulos do mesmo modo que estavam a ser inicialmente.

Efetua-se o mesmo processo para ambas as superfícies do plano, dado a serem idênticas.

Box No caso da caixa, dada à sua semelhança com o plano (é repetição deste em 6 orientações diferentes), optou-se pela mesma estratégia.

A caixa apresenta, do mesmo modo, dois parâmetros: comprimento (*length*) e divisões (*divisions*), pelo que traduz-se diretamente do **plano**.

Sphere Identificou-se 2 casos de mapeamento, extremidades e resto da esfera. Assim, a construção das extremidades foi o maior desafio, levando a algum estudo para perceber a melhor estratégia a adotar. Apesar de se ter experimentado efetuar *cubemapping*, cálculo através dos ângulos, conversão das coordenadas em *texpoints*, o melhor resultado obteve-se através da utilização de uma abordagem mais linear.

Assim, para o resto da esfera utilizou-se um *step* que correspondia a $1/stacks$ ou $1/slices$ e foi-se construíndo em simultâneo com o resto dos cálculos. Posteriormente, colocou-se condições de controlo para os pontos de extremidade, pois estes tinham um comportamento diferente. Definiu-se, assim, que o vértice superior teria a mesma coordenada s e mais um *step* na coordenada t.

O resultado obtido não está perfeito, pois consegue-se identificar algum **blur** quando se aproxima a câmera.

Cone Para coordenadas de textura da base do cone, o centro da base corresponde ao centro da textura(0.5, 0.5) e cada slice corresponde a um slice também na textura com o mesmo ângulo α mas com um raio de 0.5.

Para a parte lateral foram criados 2 *steps*:

- $xStep = 1.0/slices$
- $yStep = 1.0/stacks$

A construção do cone é feita com base num ciclo que itera o número de *stacks* vezes em que o yStep começa a 0 e incremete yStep a cada iteração. Em que cada iteração d este ciclo é percorrido outro ciclo com número de *slices* iterações, eu que o xStep começa a 0 e incrementa xStep a cada iteração. Com estas duas variavéis é facilmente obtido as coordenas de textura de cada ponto.

Cylinder A estratégia utilizada no cilindro foi a mesma que no cone, sendo as texturas das duas bases do cilindro obtidas da mesma forma que a base do cone e o mesmo para as coordenadass de texturas da face lateral.

Torus Não houve disponibilidade para o desenvolvimento das *texcoord* do tórus.

2.3 *Writer*

Para poder inserir as duas novas *tags* dentro dos ficheiros *.3D*, pequenas alterações ao *writer* tiveram de ser feitas.

Para começar, todas as funções anteriormente criadas para construir cada uma das figuras geométricas foram alteradas. Agora, retornam um tuplo com **3 valores**:

1. Todos os vértices do objeto.
2. Todos os vetores normais do objeto.
3. Todas as coordenadas de textura para o objeto.

. . . respetivamente.

Para clarificação, o tuplo apresenta-se como:

```
1    tuple<t_points, t_points, std::vector<float>>
```

Após essa alteração, só restou atualizar a função:

```
1    void write_xml(const char* filepath, GLenum type, t_points
↪    all_points, t_points all_normals, std::vector<float> texCoords)
```

. . . recebe atualmente cada uma das listas de vértices/vetores do tuplo e escreve-as pela ordem estipulada.

Assim, preservou-se a estrutura previamente definida para a função *int main(int argc, const char** argv)*, sendo feitos mínimos *refactors*.

Adicional Para popular o modelo com mais elementos decidiu-se adicionar uma nuvem de esferas de organização aleatória, um cinturão de asteróides.

A função que criava este modelo, conjunto de figuras:

```
1    std::tuple<t_points, t_points, std::vector<float>>
↪    create_asteroids(double distMin, double distMax, int maxSize, int
↪    slices, int stacks, double alphaMax, double betaMax, int
↪    numAsteroids)
```

. . . sendo cada asteróide uma esfera.

Seria necessário indicar a :

- **distMin**: A distância mínima a um asteroide;
- **distMax**: A distância máxima a um asteroide;
- **maxSize**: O raio máximo do asteroide;
- **slices**: O número de slices da esfera;
- **stacks**: O número de stacks da esfera;
- **alphaMax**: O valor máximo do ângulo alpha;
- **betaMax**: O valor máximo do ângulo beta;
- **numAsteroids**: O número de asteroides.

Para efetuar deste modo, foi necessário criar uma variação da figura geométrica **esfera**, a função:

```
1     std::tuple<t_points, t_points, std::vector<float>> create_sphere(int
↪   radius, int slices, int stacks, Point offset)
```

...sendo adicionado um *offset* aleatório compreendido entre a distância mínima e máxima, para posicionar as esferas, com base na variação de um ângulo α, horizontal, e um ângulo β, vertical.

Nesta última fase proposta pelo enunciado do trabalho prático foi necessário a criação de novas classes, sendo estas acrescentadas com base na estrutura realizada na primeira fase. As novas classes foram criadas com base nos novos elementos do ficheiro XML, nomeadamente, *texture*, *color* e *lights*, mantendo assim a responsabilidade de leitura do XML para a classe responsável por manipular estes elementos.

Apesar de não existir *normals* no ficheiro XML, foi criada a classe *normals*, responsável pela leitura das normais presentes no ficheiro .3d criado pelo *generator*.

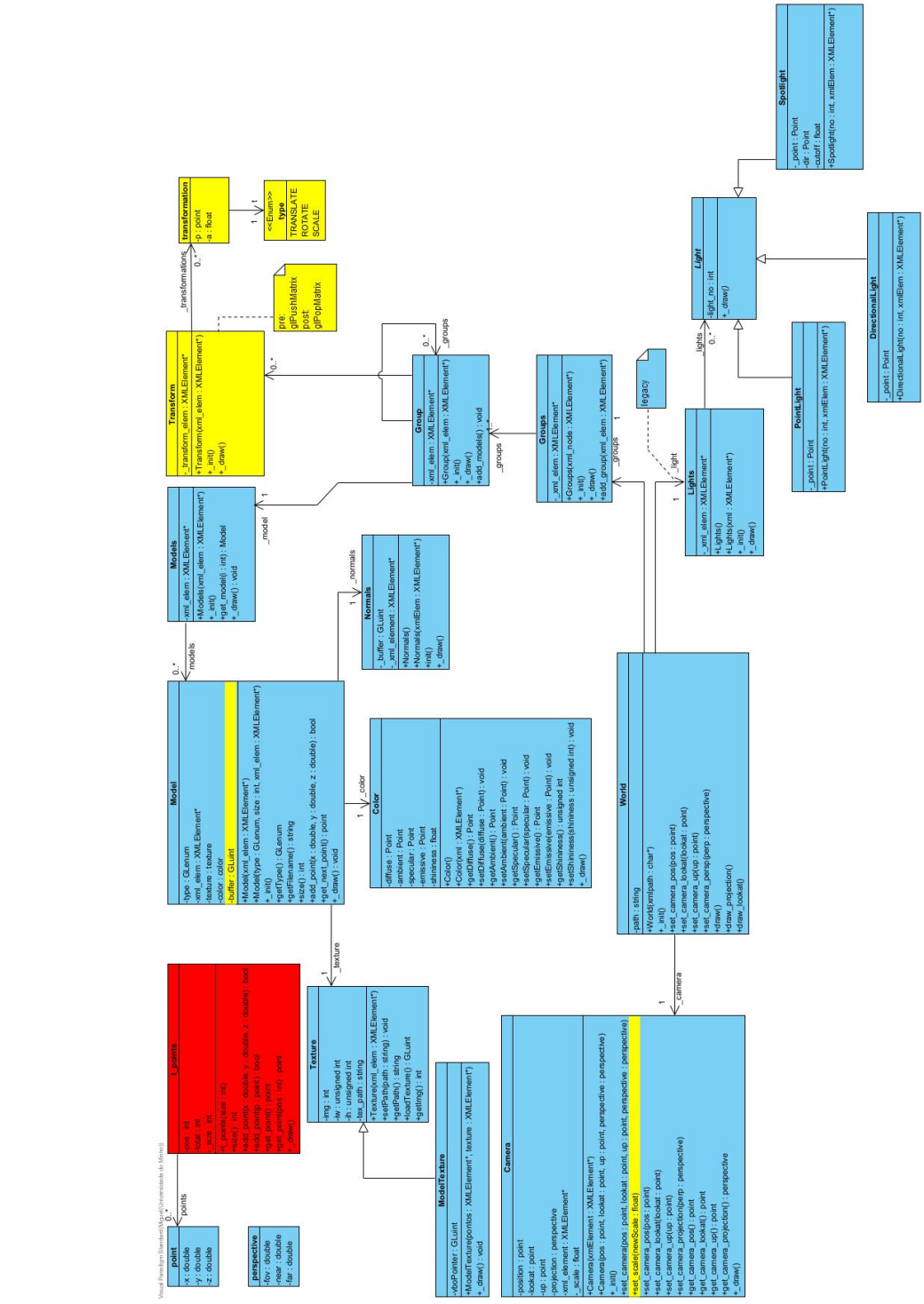

Figura 1: Diagrama de classes

2.4 Luzes

Para tratar das diferentes tipos diferentes de luzes foram criadas foram criadas três subclasses da classe abstrata *Light*:

- *DirectionalLight*
- *PointLight*
- *Spotlight*

Em que a classe abstrata tem como argumento o número da luz, e as subclasses os parâmetros necessários para a sua ativação.

A classe *Lights* que é classe agregadora de todas as luzes do modelo e a responsável por processar o elemento *lights* do ficheiro XML...

Todas as luzes são inicializadas com os valores *default* de luz ambiente, difusa e especular semelhantes à **LIGHT0**. Apenas difere na luz ambiente, pois utiliza-se um valor diferente de 0 para haver alguma luminosidade da parte de trás dos objetos, obtendo uma melhor simulação da realidade.

No seguimento da ativação das luzes, tornou-se necessário a ativação das normais com recurso a **VBO**s.

Assim, com recurso a uma classe criada para o efeito, designada **Normals**, foi possível instituir a utilização de normais em cada modelo. A estratégia utilizada consistiu em ler o ficheiro XML, gerar um *buffer* na gráfica, efetuar o seu binding e popular com pontos. Posteriormente, ao realizar o **rendering**, efetua-se a verificação se existe alguma textura associada ao modelo e, caso exista, é executado o método *_draw* da classe *Normals*.

2.5 Cores

A classe *Color* tem como argumentos as 5 componentes da luz, que são inicializadas por defeito com os valores passados pela equipa docente no enunciado do trabalho.

```
19 private:
20         float _diffuse[4] = {200.f / 255,200.f / 255.f,200.f / 255.f,
   ↪  1.f};
21         float _ambient[4] = {50.f / 255.f ,50.f / 255.f ,50.f / 255.f ,
   ↪  1.f};
22         float _specular[4] = {0.f,0.f,0.f, 1.f};
23         float _emissive[4] = {0.f,0.f,0.f, 1.f};
24         float _shininess = 0.f;
25
```

Com base no ficheiro XML recebido, os argumentos são atualizados com os valores recebidos.

Na função *_draw* é feito o *rendering* das cores com o auxilio das funções *glMaterialfv* e *glMateriali*.

2.6 Texturas

Para tratar das Texturas foram criadas 2 classes, uma superclasse *Texture* e a sua subclasse *ModelTexture*.

Na classe *Texture* é onde é feito o processamento da imagem com a textura a partir da função *loadTexture*. Esta utiliza como parâmetros de *wrap* o *repeat* tanto em S como em T. De maneira a obter uma textura com melhor qualidade e ainda melhorar o desempenho foi utilizado o *MipMapping*, fazendo proveito da primitiva *linear* para a escolha do *pixel* e da *nearest* para a escolha da textura.

A classe *ModelTexture* por sua vez é responsável por processar as coordenadas de texturas recebidas no ficheiro ".3d", colocando-as no *buffer vboPointer*.

2.7 Modelo do Sistema Solar

O sistema solar foi atualizado, inserindo luzes e também as cores e textura para cada modelo. Além disso também foi adicionado um cinturão de asteroides.

Como o sistema solar desenvolvido é apenas constituído por apenas uma estrela, que é o sol, existe apenas uma única fonte de luz que é ponto situado no centro do sol (0, 0, 0). A luz trata-se de um ponto de maneira a emitir luz em todas as direções.

As texturas escolhidas para cada elemento do sistema foram retiradas do *site Solar System Scope*.

As componentes da cor foram escolhidas de maneira a aproximar o modelo do sistema solar da realidade.

- A cor difusa para cada modelo foi escolhida com base na cor predominante da textura.
- A cor ambiente foi escolhida de forma a tentar representar a atmosfera envolvente.
- O único elemento que apresenta cor emissiva é o sol, uma vez que é o único que emite luz.
- Nenhum dos elementos do sistema solar verifica a reflexão especular logo a cor especular e a *shininess* são sempre nulas.

Tal como referido anteriormente, foi adicionado o cinturão de asteroides que se situa entre Marte e Júpiter. O cinturão possui um movimento de rotação sobre o sol.

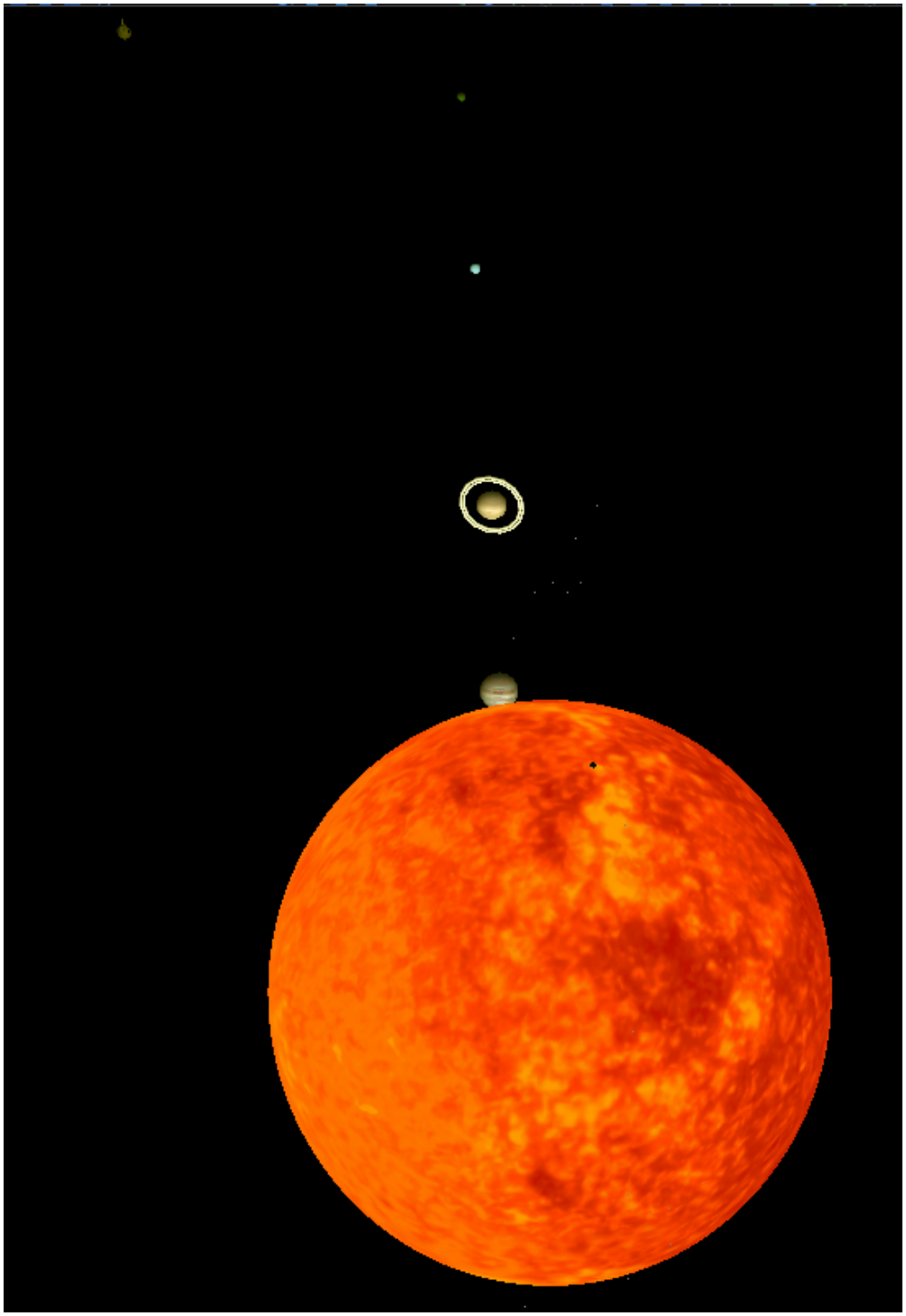

Figura 2: Sistema solar 1

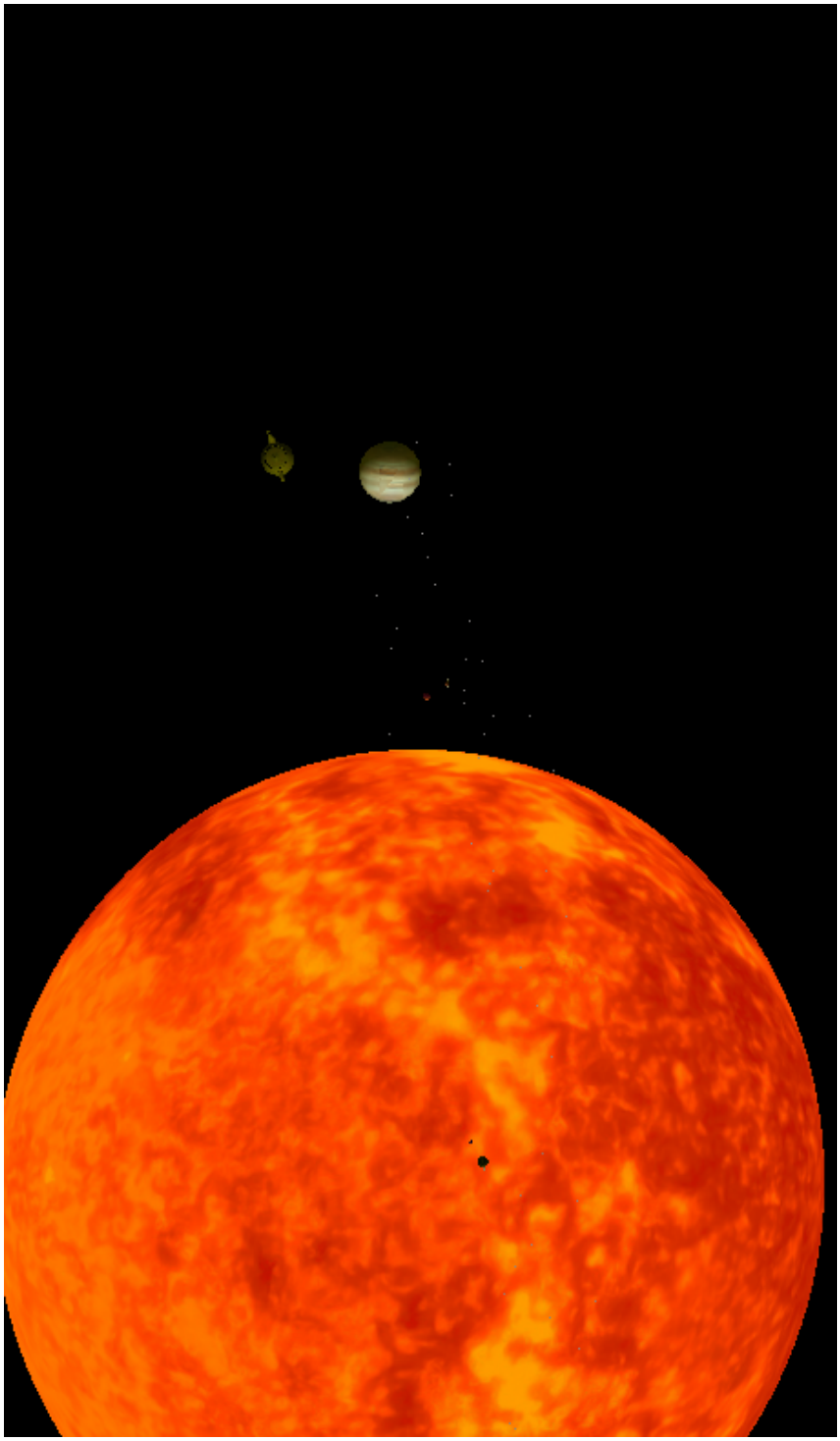

Figura 3: Sistema solar 2

2.8 Testes realizados

Para avaliar o desenvolvimento do projeto foram testados os ficheiros de testes disponiblizados pela equipa docente e comparou-se com as imagens com o resultado. As seguintes imagens compararam o modelo desenvolvido pelo grupo(lado esquerdo) com as imagens disponiblizadas.

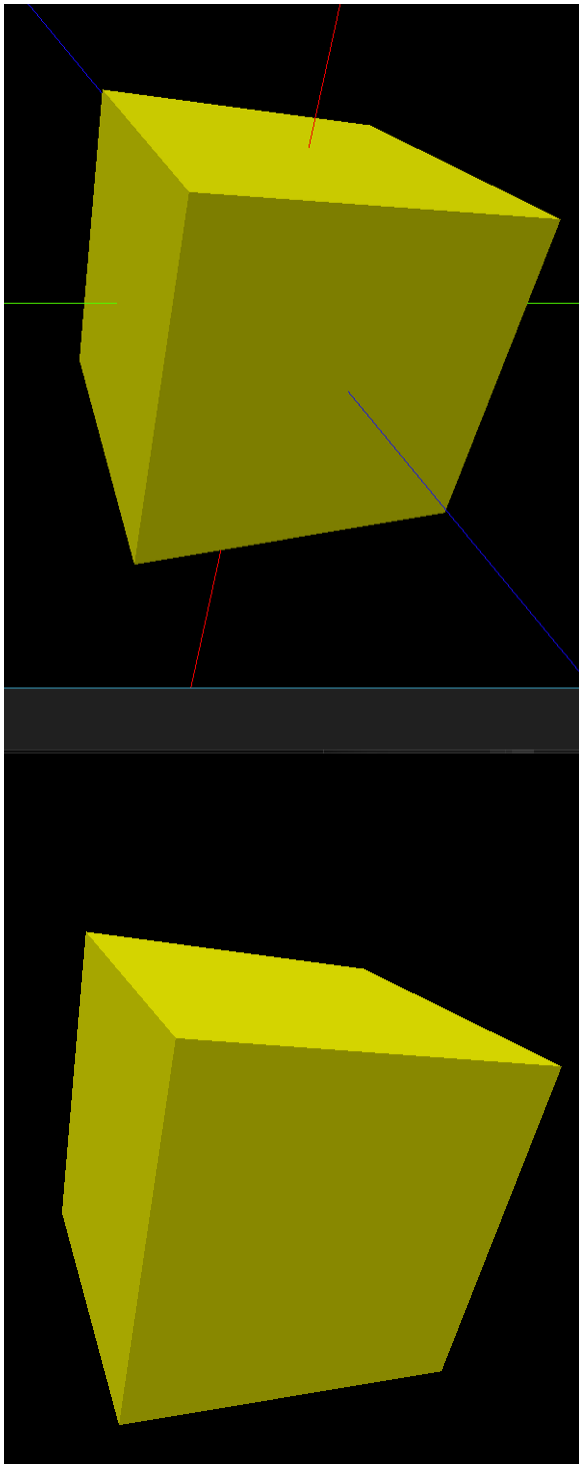

Figura 4: Resultado do *test_4_1*

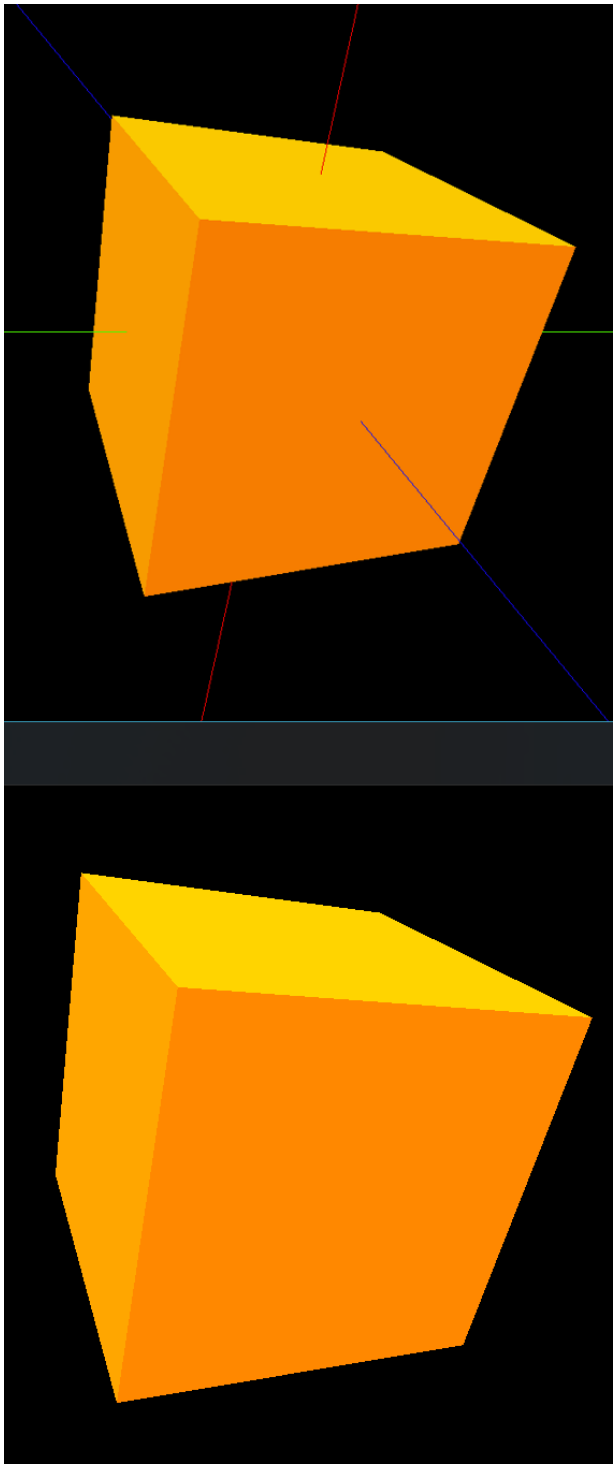

Figura 5: Resultado do *test_4_2*

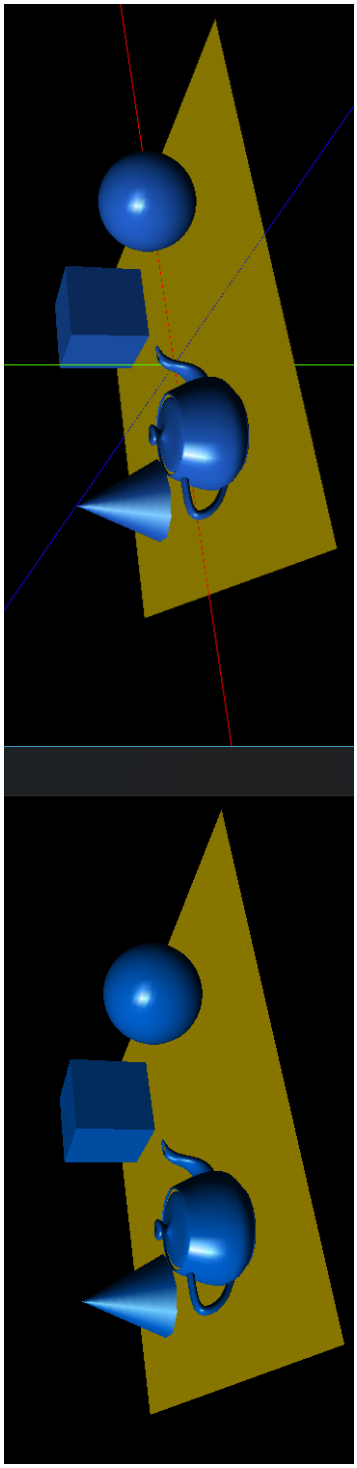

Figura 6: Resultado do *test_4_3*

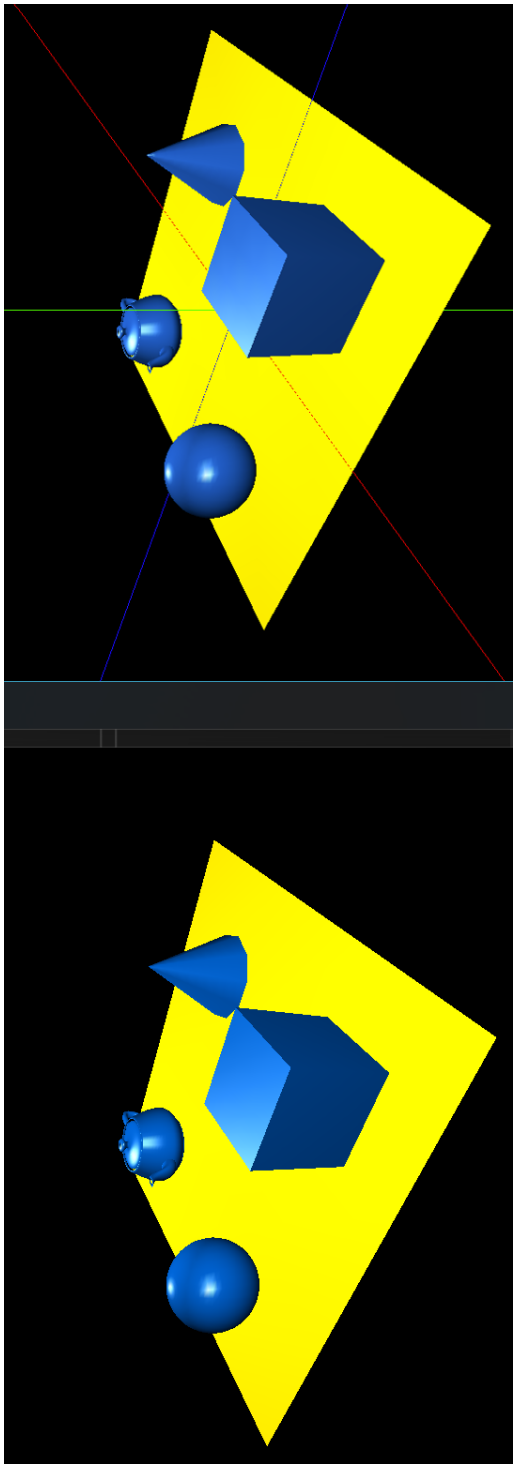

Figura 7: Resultado do *test_4_4*

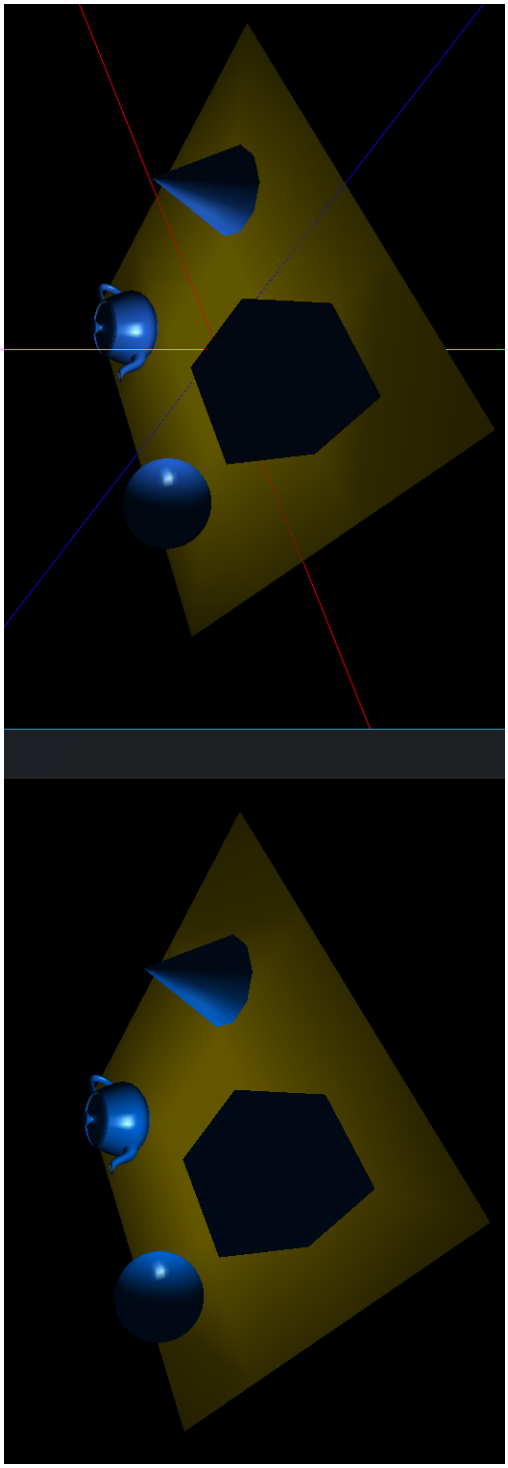

Figura 8: Resultado do *test_4_5*

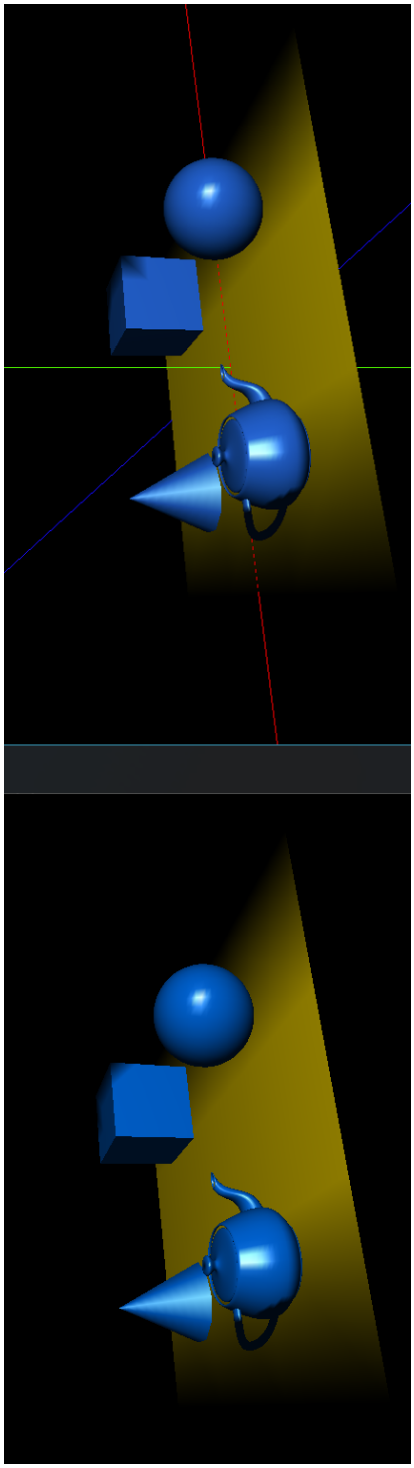

Figura 9: Resultado do *test_4_6*

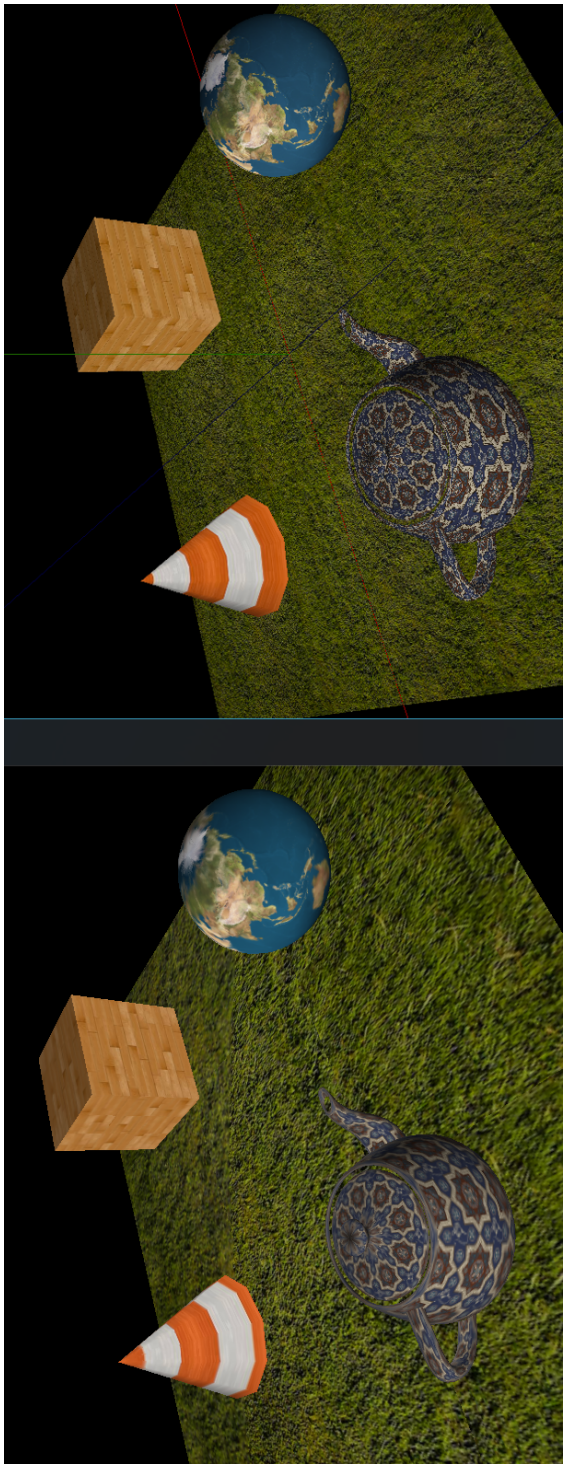

Figura 10: Resultado do *test_4_7*

3 Conclusões

Esforço que permaneceu até a última fase do projeto, o grupo atualizou ambos os programas, *generator* e *engine*, seguindo a arquitetura inicialmente estabelecida. Deste modo, verifica-se a coesão das decisões tomadas inicialmente, sendo apenas notável a evolução do projeto com a adição de novos elementos.

Não obstante as primitivas objetivo definidas em fases anteriores, nomeadamente relativas ao uso de memória e na organização, foi o desejo do grupo aplicar todo o seu conhecimento agregado no decorrer do ano letivo.

É de notar nesta fase:

- Criação dos vetores normais a cada vértice das figuras geométricas.
- Criação das coordenadas de texturas respetivas.
- Atualização do sistema solar, para um modelo texturado e com iluminação.

No seguimento do trabalho desenvolvido, o grupo pretendia acrescentar algumas funcionalidades, nomeadamente a utilização de índices, acréscimo de algumas *demos* mais elaboradas, melhoria da interação do utilizador com o sistema (apresentação de um menu, possibilidade de carregar em múltiplas teclas em simultâneo), entre outros. Não obstante, a equipa docente ter flexibilizado a entrega do trabalho, tais funcionalidades adicionais não foram cumpridos por falta de disponibilidade do grupo.