



**Universidade do Minho**  
Escola de Engenharia  
Licenciatura em Engenharia Informática

## **Unidade Curricular de Computação Gráfica**

Ano Letivo de 2022/2023

## **Parte 4 - Normals and Texture Coordinates**

Filipa Gomes (A96556)   Miguel Gomes (A93294)   Rita Lino (A93196)

4 de junho de 2023

**CG**

# Resumo

Neste relatório explicitaremos as nossas decisões na realização da quarta fase do trabalho da Unidade Curricular de Computação Gráfica. Visando, deste modo, explicar como foi implementado as coordenadas das normais e das texturas no *generator* e explicar como foi o processo de implementação das luzes e materiais em cada objeto.

**Área de Aplicação:** Computação Gráfica.

**Palavras-Chave:** OpenGL, C++, XML, Transformações geométricas, Primitivas gráficas, Normais, Texturas, Materiais, Luz.

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Coordenadas de Normais e Texturas</b>	<b>2</b>
2.1	Coordenadas de Normais . . . . .	2
2.2	Coordenadas das Texturas . . . . .	2
2.3	Formatação dos pontos do objeto . . . . .	3
2.4	Preparação dos dados . . . . .	4
2.5	Resultados obtidos . . . . .	5
<b>3</b>	<b>Luzes e materiais</b>	<b>7</b>
3.1	Materiais . . . . .	7
3.2	Luzes . . . . .	8
3.3	Resultados obtidos . . . . .	9
<b>4</b>	<b>Testes e Resultados Finais</b>	<b>11</b>
4.1	Testes . . . . .	11
4.2	Sistema Solar . . . . .	15
<b>5</b>	<b>Conclusões e trabalhos futuros</b>	<b>17</b>
5.1	Otimizações . . . . .	17
5.2	Extras . . . . .	17

# **Lista de Figuras**

2.1	Ficheiro .3d de uma esfera com raio 1, 8 <i>stacks</i> e 8 <i>slices</i> . . . . .	4
2.2	Resultado obtido do teste 4.6. (sem luzes implementadas). . . . .	6
3.1	Resultados obtidos do teste 4.6 com a implementação de luzes e materiais. . . . .	10
4.1	Resultado obtido do teste 4.1 . . . . .	11
4.2	Resultado obtido do teste 4.2 . . . . .	12
4.3	Resultado obtido do teste 4.3 . . . . .	12
4.4	Resultado obtido do teste 4.4 . . . . .	13
4.5	Resultado obtido do teste 4.5 . . . . .	14
4.6	Sistema Solar (com asteroides) implementado com texturas, luzes, normais e materiais. . .	15
4.7	Sistema Solar (sem asteroides) implementado com texturas, luzes, normais e materiais. . .	16
5.1	Plano implementado com textura com repetição <i>mipmap</i> . . . . .	18
5.2	Plano implementado com textura sem repetição . . . . .	18

# 1 Introdução

O objetivo desta tarefa prática é desenvolver um motor 3D baseado em grafos de cena e fornecer exemplos de uso que demonstrem o seu potencial. A tarefa está dividida em quatro fases, sendo esta a quarta. Para cada fase, serão fornecidos um conjunto de ficheiros XML de configuração para fins de teste e avaliação. Cada configuração é acompanhada pela respetiva saída visual.

Neste relatório fazemos uma descrição das funcionalidades pedidas para a última fase do projeto. Assim sendo, esta fase envolve a implementação de coordenadas normais e de textura, tal como luzes e materiais.

Em resumo, esta última fase baseia-se numa aplicação de todos os conceitos aprendidos na UC de Computação Gráfica. Sendo a continuação das fases anteriores, bastou fazer alterações de modo que se fizesse uso de texturas, luzes e materiais, conforme o indicado nos ficheiros de teste em XML.

# 2 Coordenadas de Normais e Texturas

Para a implementação de tais coordenadas foi necessário alterar o código do *generator* em todos os objetos. Foram criados vetores normais, de textura e finais (para a implementação dos mesmos no *engine*).

## 2.1 Coordenadas de Normais

Antes de ser feita a implementação atual, foram feitos cálculos de normais para cada triângulo, ou seja, conjunto de três pontos do objeto. Isto levou a que fossem feitos bastantes cálculos desnecessários e que levavam a uma incongruência por parte dos objetos com superfícies curvas (ex: curvas de bezier, cilindro, etc.), visto que o mesmo ponto poderá ter mais que uma normal.

Decidiu-se então, implementar um vetor normal por cada ponto do objeto. Com esta implementação deixariam de haver incongruências e problemas de criar várias normais para um ponto só.

```
1 std :: vector<Point3> draw_sphere( double radius , int slices , int stacks ) {
2     std :: vector<Point> normal ;
3     std :: vector<Point> points ;
4
5     // código incompleto por motivos de demonstração
6
7     auto n1 = Point(p1.x / length , p1.y / length , p1.z / length );
8     auto n2 = Point(p2.x / length , p2.y / length , p2.z / length );
9     auto n3 = Point(p3.x / length , p3.y / length , p3.z / length );
10    auto n4 = Point(p4.x / length , p4.y / length , p4.z / length );
11
12    n1 . normalize () ;
13    n2 . normalize () ;
14    n3 . normalize () ;
15    n4 . normalize () ;
16
17    points . push_back (p1) ;
18    normal . push_back (n1) ;
19
20    points . push_back (p2) ;
21    normal . push_back (n2) ;
22
23 // ( ... )
24 }
```

## 2.2 Coordenadas das Texturas

Foram implementadas duas opções de textura, uma à base de *mipmaps*, que implementa a textura inteira por cada par de triângulos, que forma um quadrado, do objeto, e outra em que as coordenadas da textura

são calculadas dependente do formato da imagem correspondente à textura, recorrendo às *slices/stacks* do mesmo objeto. No caso da implementação dos pares de coordenadas de textura das curvas *debezier* esta mostrou ser uma dificuldade, uma vez que a textura final não fica alinhada consigo mesma na segunda forma de os calcular.

```

1 std :: vector<Point3> draw_sphere(double radius , int slices , int stacks){
2     std :: vector<Point> points;
3     std :: vector<Point> texture;
4
5 //código incompleto por motivos de demonstração
6
7     auto t1 = Point((float)current_slice / slices , (float)current_stack / stacks
8 , 0);
9     auto t2 = Point((float)next_slice / slices , (float)current_stack / stacks ,
10 0);
11    auto t3 = Point((float)next_slice / slices , (float)next_stack / stacks , 0);
12    auto t4 = Point((float)current_slice / slices , (float)next_stack / stacks ,
13 0);
14
15
16    points.push_back(p1);
17    texture.push_back(t1);
18
19    points.push_back(p2);
20    texture.push_back(t2);
21
22 // (...)
23 }
```

## 2.3 Formatação dos pontos do objeto

De modo que fosse possível implementar as normais e texturas, decidiu-se criar um formato de um vetor de pontos que assumisse cada coordenada de um ponto, cada coordenada das normais e as coordenadas da textura, tudo no mesmo ponto. Em suma, cada triângulo, conjunto de três pontos teria o seguinte formato:  $x,y,z:nx,ny,nz:tx,ty;x,y,z:nx,ny,nz:tx,ty;x,y,z:nx,ny,nz:tx,ty$ ; onde os pontos  $x, y, z$  são as coordenadas do vértice,  $nx, ny, nz$  são o vetor normal desse vértice e  $tx, ty$  as coordenadas de textura relativas a esse vértice.

```

1 std :: vector<Point3> draw_sphere(double radius , int slices , int stacks){
2     std :: vector<Point3> return_points;
3     std :: vector<Point> points;
4     std :: vector<Point> normal;
5     std :: vector<Point> texture;
6
7 //código incompleto por motivos de demonstração
8 //x,y,z:nx,ny,nz:tx,ty;x,y,z:nx,ny,nz:tx,ty;x,y,z:nx,ny,nz:tx,ty; for each
9 triangle
10    for (int i = 0; i < points.size(); i++){
11        return_points.push_back(Point3(points[i] , normal[i] , texture[i]));
12    }
13
14    return return_points;
15 }
```

Com isto, foi necessário alterar os ficheiros gerados pelo *generator*, de modo que na primeira linha esteja o número de pontos do objeto, e por cada linha esteja as coordenadas das normais, pontos e textura de um

triângulo.

```

sphere_1_0.3d
File: sphere_1_0.3d
384
0,-1.6,12323e-17;0,-1.6,12323e-17;0,0;4,32978e-17;-1,4,32978e-17;4,32978e-17,-1,4,32978e-17;0,125,0;0,270598,,0,92388,0,270598,,0,92388,0,270598;0,125,0,125;
0,-1.6,12323e-17;0,-1.6,12323e-17;0,0;0,270598,,0,92388,0,270598;0,125,0,125;0,-0.92388,0,382683;0,0,125;0,270598,-0,92388,0,270598;0,125,0,125;0,0,0,92388,0,382683;0,-0,92388,0,382683;0,0,125;0,5,-0.707107,0,5;0,5,-0.707107,0,707107;0,0,0,707107;0,0,25;
0,-0.92388,0,382683;0,-0,92388,0,382683;0,0,125;0,5,-0.707107,0,5;0,5,-0.707107,0,707107;0,0,0,707107;0,0,25;

```

Figura 2.1: Ficheiro .3d de uma esfera com raio 1, 8 *stacks* e 8 *slices*.

## 2.4 Preparação dos dados

De modo que fosse possível implementar as normais e as texturas foram criados VBOs e cópias dos mesmos.

```

1 auto Model::prepare_data() -> void {
2     // points, normals, textures
3     vector<float> p,n,t;
4
5     for (int vertex = 0; vertex < this->points.size(); vertex++){
6         p.push_back(this->points[vertex].x);
7         p.push_back(this->points[vertex].y);
8         p.push_back(this->points[vertex].z);
9
10        n.push_back(this->normal_vectors[vertex].x);
11        n.push_back(this->normal_vectors[vertex].y);
12        n.push_back(this->normal_vectors[vertex].z);
13
14        t.push_back(this->texture_points[vertex].x);
15        t.push_back(this->texture_points[vertex].y);
16    }
17
18    this->vertices_count = p.size() / 3;
19    this->normal_count = n.size() / 3;
20    this->texture_count = t.size() / 2;
21
22    // VERTICES
23
24    // (...)
25
26    //VBO
27    glGenBuffers(1, &(this->vertices));
28
29    //Copy
30    glBindBuffer(GL_ARRAY_BUFFER, this->vertices);
31    glBufferData(
32        GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
33        sizeof(float) * p.size(), // tamanho do vector em bytes
34        0, // os dados do array associado ao vector
35        GL_STATIC_DRAW // indicativo da utilização (estático e para desenho)
36    );
37    glBufferSubData(
38        GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
39        0, // offset
40        sizeof(float) * p.size(), // tamanho do vector em bytes
41        p.data() // os dados do array associado ao vector
42    );
43
44    // (...)
45

```

```

46 //UNBIND
47 gBindBuffer(GL_ARRAY_BUFFER, 0);
48 gBindVertexArray(0);
49
50 // NORMALS
51
52 //VBO
53 gGenBuffers(1, &(this->normal));
54 gBindBuffer(GL_ARRAY_BUFFER, this->normal);
55
56 //Copy
57 gBufferData(
58     GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
59     sizeof(float) * n.size(), // tamanho do vector em bytes
60     n.data(), // os dados do array associado ao vector
61     GL_STATIC_DRAW // indicativo da utilização (estático e para desenho)
62 );
63
64 // TEXTURES
65
66 //VBO
67 gGenBuffers(1, &(this->textures));
68 gBindBuffer(GL_ARRAY_BUFFER, this->textures);
69
70 //Copy
71 gBufferData(
72     GL_ARRAY_BUFFER, // tipo do buffer, só é relevante na altura do desenho
73     sizeof(float) * t.size(), // tamanho do vector em bytes
74     t.data(), // os dados do array associado ao vector
75     GL_STATIC_DRAW // indicativo da utilização (estático e para desenho)
76 );
77
78 }

```

## 2.5 Resultados obtidos

No final da implementação das normais e texturas foram obtidos resultados quase ótimos, como se pode ver na seguinte figura 2.2.

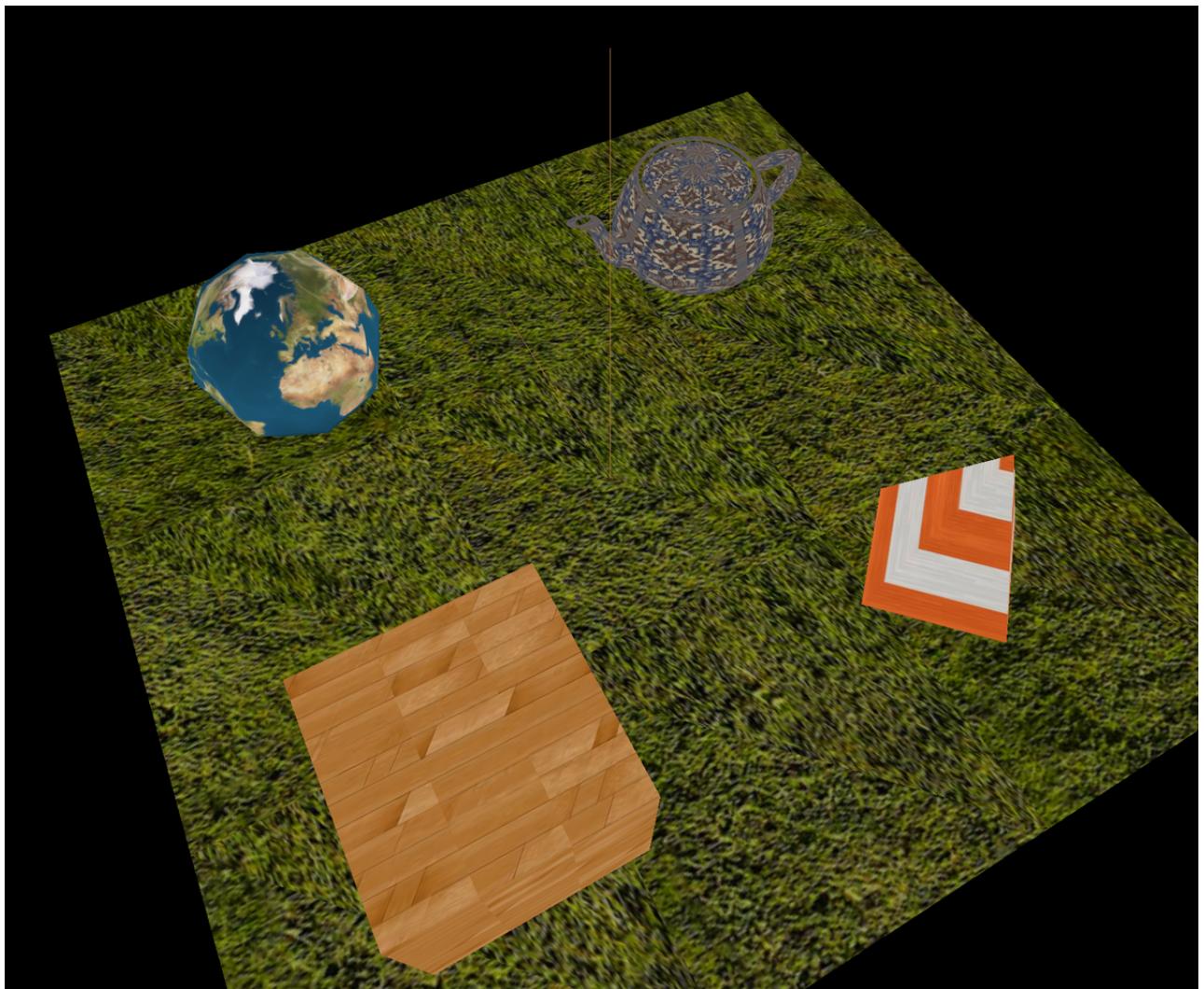


Figura 2.2: Resultado obtido do teste 4.6. (sem luzes implementadas).

# 3 Luzes e materiais

Tomou-se a decisão de aplicar os materiais primeiro por ser menos complexo que as luzes, à primeira vista.

## 3.1 Materiais

Começou-se por implementar uma classe **Color** que contem os atributos de uma certa cor, especificada nos ficheiros XML de teste, em formato RGB.

```
1 class Color {
2     public:
3         float r;
4         float g;
5         float b;
6
7     Color(float r, float g, float b);
8     Color(string hex);
9     Color();
10
11    auto apply() -> void;
12    auto sshow() -> string;
13    auto show() -> void;
14
15
16};
```

Definiu-se logo a seguir uma classe **Reflection** que contem todos os atributos de um material, especificado também nos ficheiros XML de teste.

```
1 class Reflection{
2     public:
3         bool has_material;
4         Color ambient;
5         Color diffuse;
6         Color specular;
7         Color emissive;
8         float shininess;
9
10        Reflection(Color ambient, Color diffuse, Color specular, Color emissive,
11                     float shininess);
12        Reflection();
13
14        auto show() -> void;
15        auto apply() -> void;
16};
```

Para a aplicação dos materiais e dos seus atributos criou-se a função ***apply*** da classe ***Reflection***, que implementa o material conforme o seu tipo e valores RGB, através da função ***glMaterialfv***.

```

1 auto Reflection::apply() -> void{
2
3     float ambient[4] = {this->ambient.r, this->ambient.g, this->ambient.b, 1.0};
4     float diffuse[4] = {this->diffuse.r, this->diffuse.g, this->diffuse.b, 1.0};
5     float specular[4] = {this->specular.r, this->specular.g, this->specular.b,
6                           1.0};
7     float emissive[4] = {this->emissive.r, this->emissive.g, this->emissive.b,
8                           1.0};
9
10    glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
11    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
12    glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
13    glMaterialfv(GL_FRONT, GL_EMISSION, emissive);
14    glMaterialf(GL_FRONT, GL_SHININESS, this->shininess);
15
16 }
```

## 3.2 Luzes

Como de costume, primeiramente foi necessário criar uma classe ***Light*** que contem toda a informação da luz, desde o id, tipo, posição e direção. Também para essa classe foram implementadas funções de ligar e desligar a luz.

```

1 class Light{
2     public:
3         unsigned int _id;
4         char type;
5         Point position;
6         Point direction;
7         float cutoff;
8
9         Light(Point position, Point direction, float cutoff);
10        Light(Point position);
11        Light(int o, Point direction);
12
13        auto show() -> void ;
14        auto render() -> void ;
15        auto on() -> void ;
16        auto off() -> void ;
17 }
```

Para eventualmente ser possível renderizar as luzes especificadas nos ficheiros XML de teste, implementaram-se os tipos de luzes da seguinte maneira: caso leia no ficheiro XML uma direção (um ponto) e uma *flag* do tipo *int* será do tipo **d**, caso leia uma posição (um ponto) apenas, será do tipo **p**, caso receba ambas, direção (ponto) e posição (ponto), mas um *float* que corresponde ao *cutoff* então será do tipo **s**.

```

1 auto Light::render() -> void{
2     if(this->type == 's'){
3         GLfloat light_position[] = {this->position.x, this->position.y, this->
4                                       position.z, 1.0};
4         GLfloat light_direction[] = {this->direction.x, this->direction.y, this->
5                                       direction.z};
5         glLightfv(GL_LIGHT0 + this->_id, GL_POSITION, light_position);
```

```

6         glLightfv(GL_LIGHT0 + this->_id, GL_SPOT_DIRECTION, light_direction);
7         glLightf(GL_LIGHT0 + this->_id, GL_SPOT_CUTOFF, this->cutoff);
8     }
9     else if(this->type == 'p'){
10        GLfloat light_position [] = {this->position.x, this->position.y, this->
11        position.z, 1.0};
12        glLightfv(GL_LIGHT0 + this->_id, GL_POSITION, light_position);
13    }
14    else if(this->type == 'd'){
15        GLfloat light_direction [] = {this->direction.x, this->direction.y, this-
16        >direction.z};
17        glLightfv(GL_LIGHT0 + this->_id, GL_POSITION, light_direction);
18    }
19}
20
21auto Light::on() -> void{
22    glEnable(GL_LIGHT0 + this->_id);
23    float dark[4] = {0.2, 0.2, 0.2, 1.0};
24    float white[4] = {1.0, 1.0, 1.0, 1.0};
25    float black[4] = {0.0f, 0.0f, 0.0f, 0.0f};
26    // light colors
27    glLightfv(GL_LIGHT0 + this->_id, GL_AMBIENT, dark);
28    glLightfv(GL_LIGHT0 + this->_id, GL_DIFFUSE, white);
29    glLightfv(GL_LIGHT0 + this->_id, GL_SPECULAR, white);
30}
31
32auto Light::off() -> void{
33    glDisable(GL_LIGHT0 + this->_id);
34}

```

### 3.3 Resultados obtidos

Após muitas tentativas e problemas com mudança de cor da luz ser *random*, acabou-se por conseguir o resultado objetivo, como se pode observar na figura 3.1.

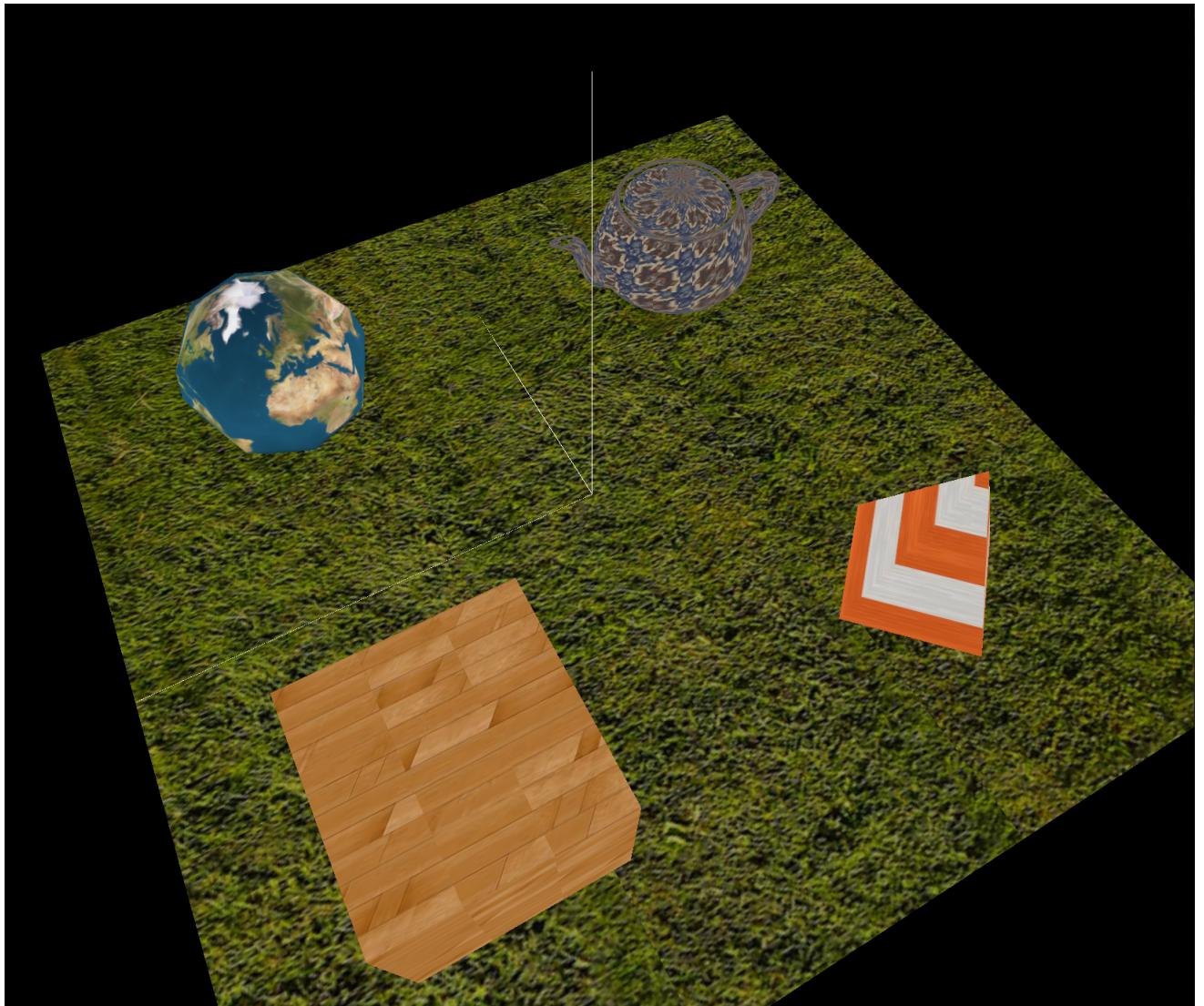


Figura 3.1: Resultados obtidos do teste 4.6 com a implementação de luzes e materiais.

# 4 Testes e Resultados Finais

## 4.1 Testes

Após a finalização do projeto, foi possível implementar todos os testes com sucesso. Neste capítulo serão apresentados os resultados de cada teste após a implementação das luzes, materiais, normais e texturas.

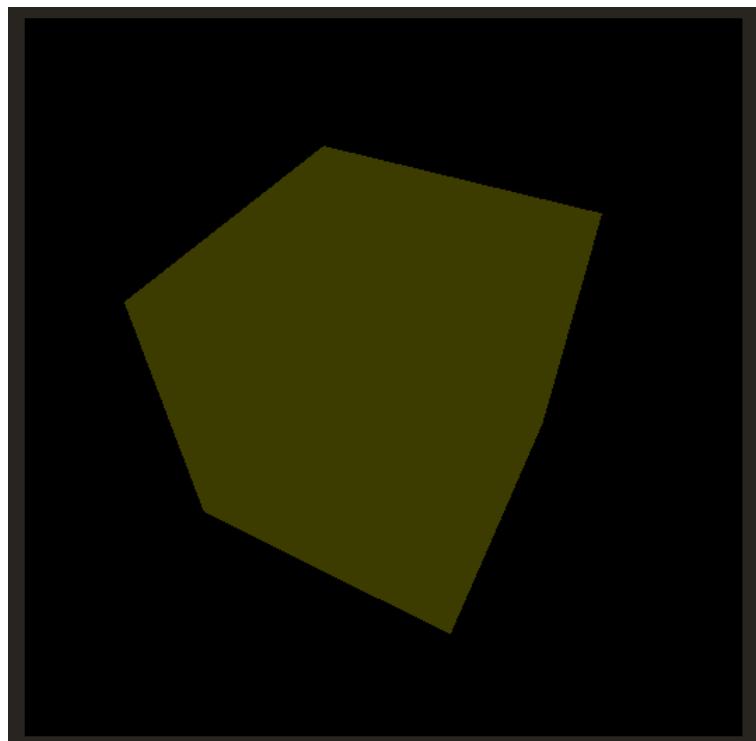


Figura 4.1: Resultado obtido do teste 4.1



Figura 4.2: Resultado obtido do teste 4.2

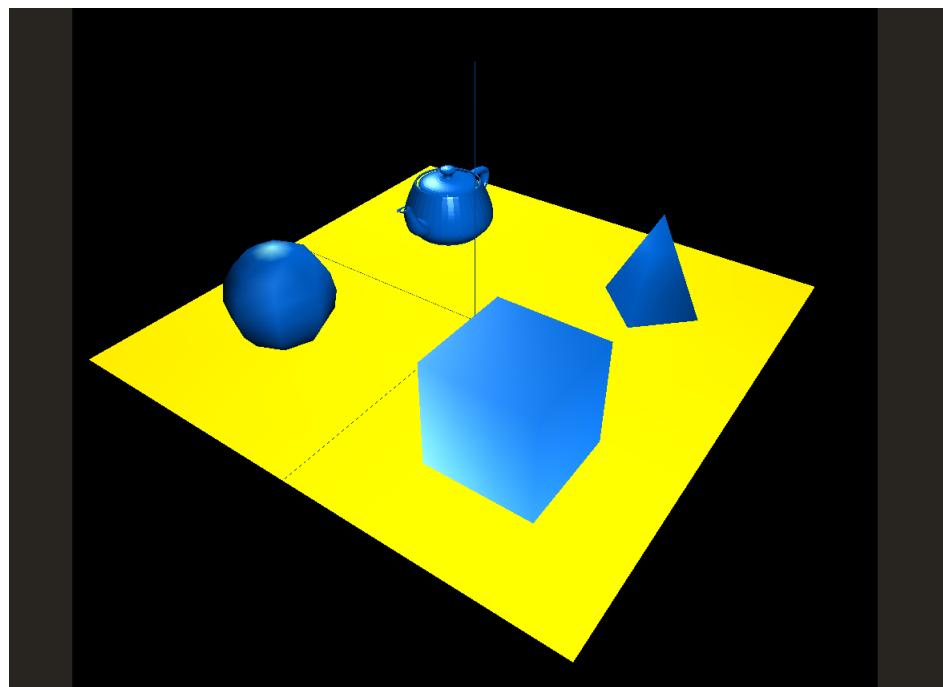


Figura 4.3: Resultado obtido do teste 4.3

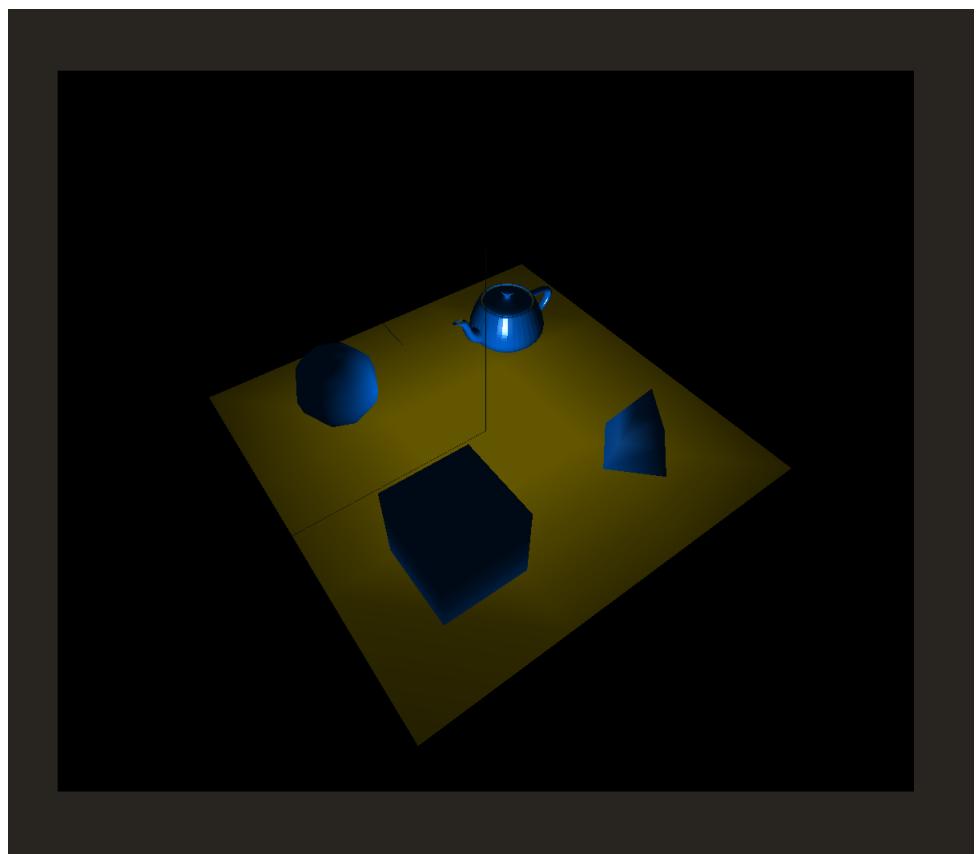


Figura 4.4: Resultado obtido do teste 4.4

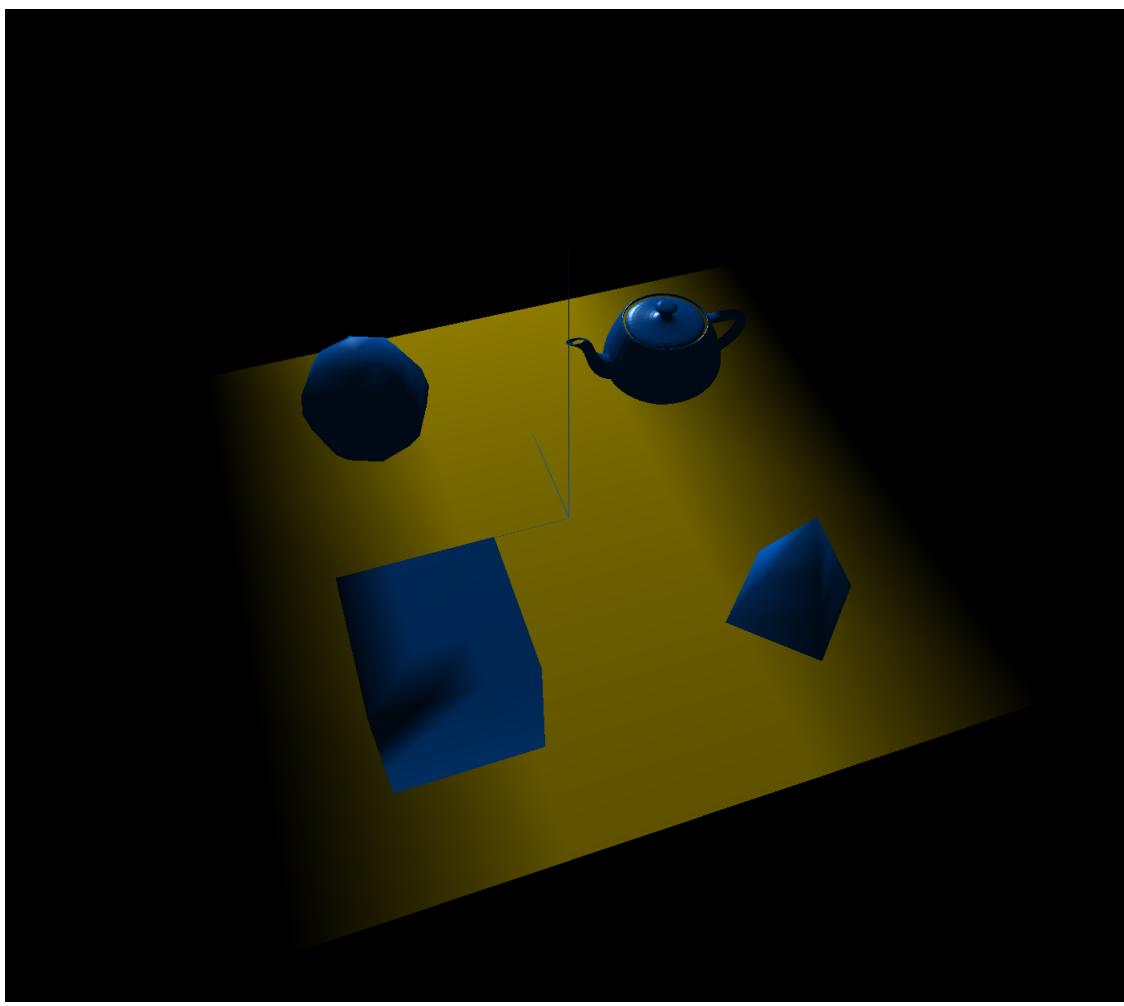


Figura 4.5: Resultado obtido do teste 4.5

## 4.2 Sistema Solar

No Sistema Solar foram aplicadas texturas de acordo com imagens reais dos planetas e astros, o que leva a um modelo muito mais realista (e bonito). Devido ao elevadíssimo número de asteroides (500), a *performance* do mesmo é bastante lenta, pois consome muita memória.

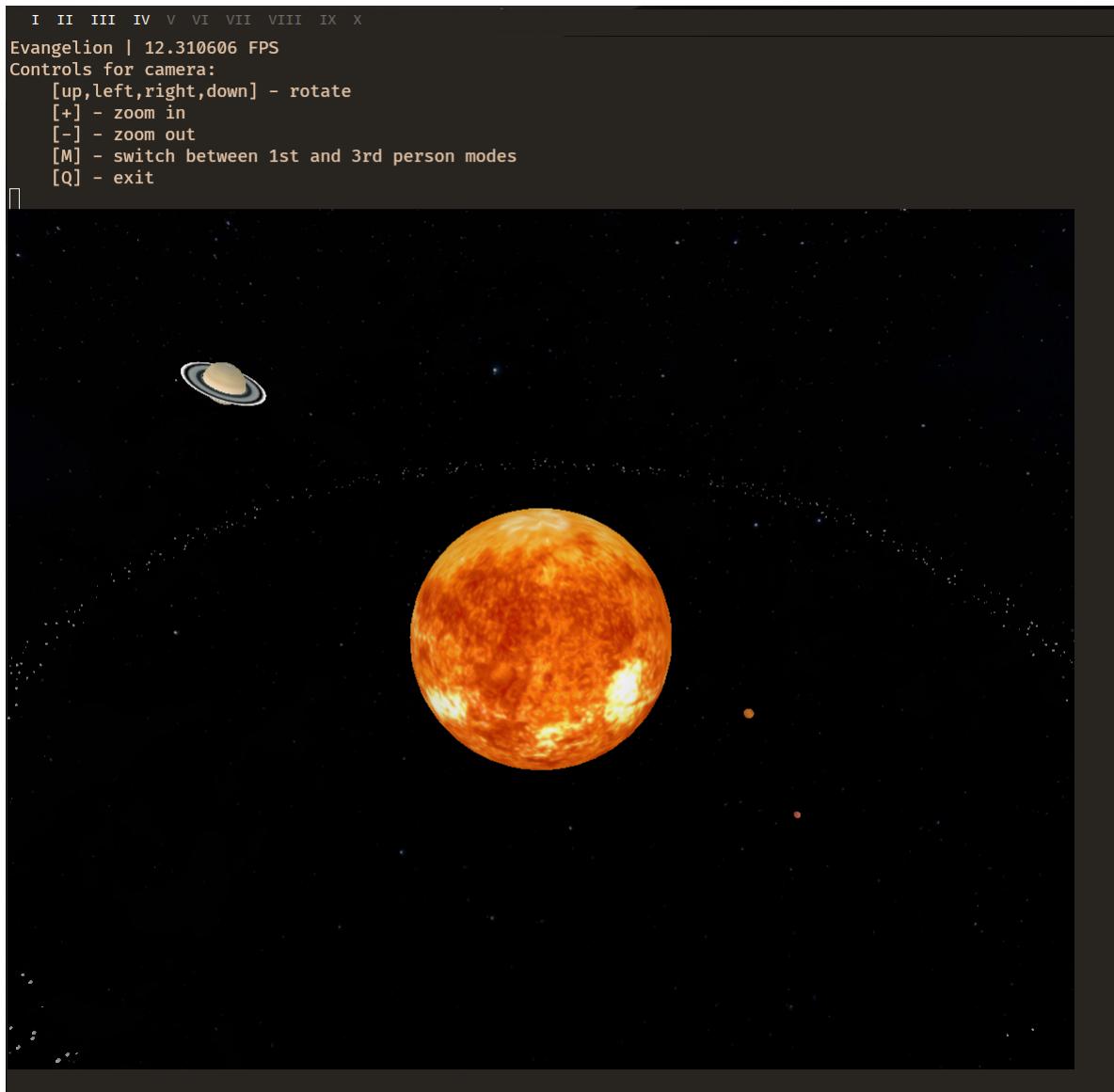


Figura 4.6: Sistema Solar (com asteroides) implementado com texturas, luzes, normais e materiais.



Figura 4.7: Sistema Solar (sem asteroides) implementado com texturas, luzes, normais e materiais.

# 5 Conclusões e trabalhos futuros

A realização desta fase do projeto permitiu-nos consolidar os conhecimentos adquiridos durante as aulas que decorreram ao longo do segundo semestre, o que nos permitiu expandir e implementar novas utilidades no projeto já criado nas fases anteriores. O Sistema Solar agora contém texturas adequadas de fotos reais do mesmo, os objetos têm texturas adequadas ao seu formato, as luzes e os materiais dão vida e realismo a todos os objetos o que é fantástico.

Considera-se que foi efetuado um ótimo trabalho, o grupo entusiasmou-se bastante com a realização do mesmo, desde as texturas à iluminação, todo o processo foi gratificante, apesar de trabalhoso.

## 5.1 Otimizações

Como possível otimização do trabalho realizado, seria a alteração dos três VBOs utilizados para um VBO apenas. Possível utilização de VBOs com índices de modo a evitar memória duplicada. Paralelização de modo a melhorar a *performance* de carregamento dos modelos.

## 5.2 Extras

Existe a possibilidade de em vez de utilizar uma textura para o modelo inteiro, de usar a textura "*on repeat*" por cada par de triângulos do objeto.

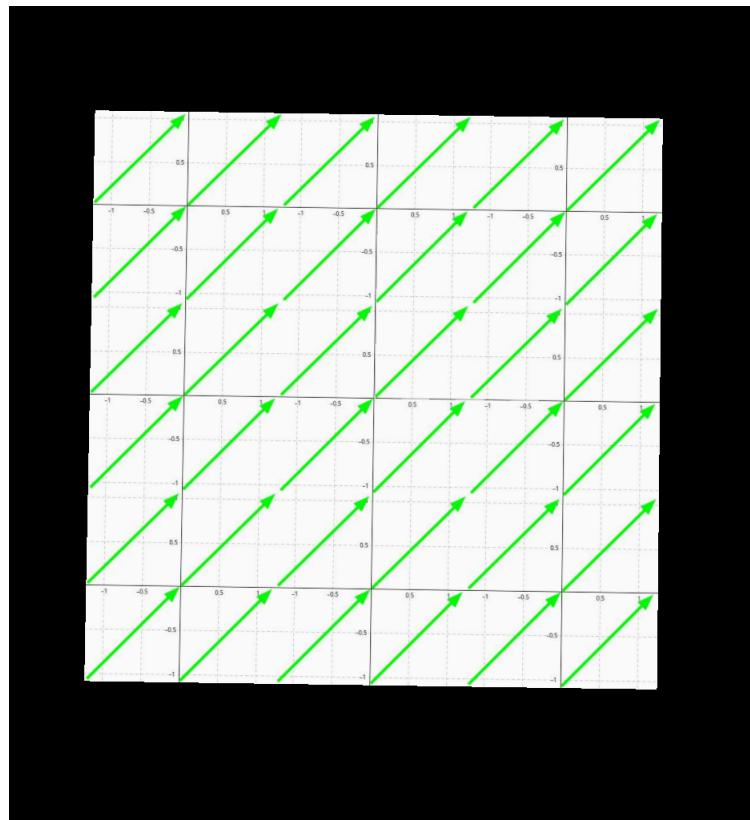


Figura 5.1: Plano implementado com textura com repetição *mipmap*

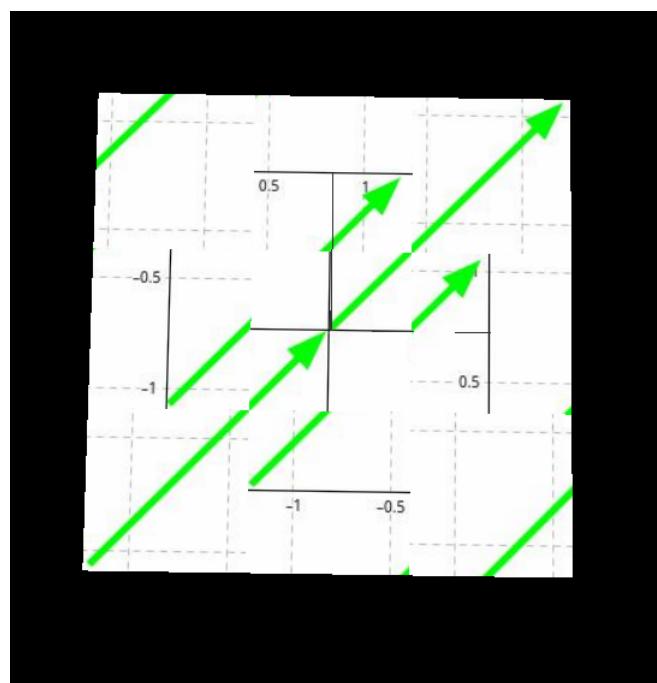


Figura 5.2: Plano implementado com textura sem repetição