

**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

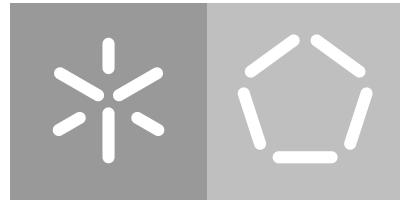
Grupo 42

Ângelo Sousa, a80813  
Diogo Ribeiro, a84442  
Rui Mendes, a83712  
Rui Reis, a84930

### **Relatório da 4<sup>a</sup> Fase**

### **Aplicação de Texturas e Luz e Funcionalidades Adicionais**

Maio 2020



**Universidade do Minho**

Escola de Engenharia

Departamento de Informática

Grupo 42

Ângelo Sousa, a80813  
Diogo Ribeiro, a84442  
Rui Mendes, a83712  
Rui Reis, a84930

### **Relatório da 4<sup>a</sup> Fase**

### **Aplicação de Texturas e Luz e Funcionalidades Adicionais**

Computação Gráfica  
Mestrado Integrado em Engenharia Informática

Maio 2020

---

## CONTÉUDO

---

1	INTRODUÇÃO	1
2	REFORMA DO CUBO	2
2.1	Necessidade da Reforma	2
2.2	Algoritmo Normal	2
3	EXTENSÃO DO GERADOR	3
3.1	Normais à Superfície	3
3.1.1	Plano	3
3.1.2	Cubo	4
3.1.3	Esfera	4
3.1.4	Cone	4
3.1.5	Toróide	5
3.1.6	Fita de Moebius	6
3.1.7	Patches de Bézier	8
3.2	Coordenadas de Textura	8
4	APLICAÇÃO DE TEXTURAS	9
5	IMPLEMENTAÇÃO DA LUZ	11
5.1	Materials	11
5.2	Iluminação	12
5.2.1	Composição de Luzes	13
6	IMPLEMENTAÇÃO DE CUBEMAPS	14
7	DEMONSTRAÇÕES TECNOLÓGICAS	15
7.1	Sistema Solar	15
7.2	Santuário de Atena	16
7.3	Minecraft	17
8	CONCLUSÕES E TRABALHO FUTURO	18

# 1

---

## INTRODUÇÃO

---

De forma a dar continuidade ao projeto que tem vindo a ser desenvolvido em fases anteriores, serve o presente documento para apresentar as alterações efetuadas nesta fase, enumerando todas as diferentes categorias e potencialidades deste sistema.

Com o objectivo de tornar este engine o mais fidedigno, e completo, possível, foram adicionados vários componentes, permitindo assim uma completude deste sistema.

Começamos com a introdução das alterações que tiveram de ser efetuadas ao generator para acautelar os novos comportamentos desejados, nomeadamente, na especificação de coordenadas de texturas e normais, de extra importância para o objetivo final desta fase.

Passando de seguida, à introdução de shaders na temática deste trabalho, que permitem a transição para OpenGL moderno, apesar de terem muitas consequências retro-ativas. Com os shaders já contemplados, somos agora capazes de implementar comportamentos de extremo interesse, com um universo de diferentes novos comportamentos.

Finalmente, explicitamos de que forma foi feita a implementação de texturas e luzes no ambiente de OpenGL moderno com shaders em GLSL.

De forma terminal a esta fase, finalizamos pela apresentação de dois novos extras ao sistema, nomeadamente, a implementação de cubemaps e um parser de Wavefront Objects, que permite trazer uma grande diversidade ao sistema.

# 2

---

## REFORMA DO CUBO

---

### 2.1 NECESSIDADE DA REFORMA

Tendo sido desenvolvido na primeira fase deste projecto, o algoritmo desenvolvido para gerar o cubo sofreu precocemente de um *overturning* característico. Inicialmente, conseguimos desenvolver um algoritmo muito complexo que era tal que conseguia definir um cubo, com um qualquer número de divisões, utilizando unicamente uma vez cada um dos vértices.

O que parecia, inicialmente, uma ótima ideia colidiu com a necessidade da definição de normais dos vértices do cubo. Com o antigo algoritmo, definir um normal para cada vértice tornava-se impossível para os vértices que compõem as arestas e vértices do cubo original. Isto porque, estes vértices possuem naturalmente mais do que uma normal, correspondente a cada uma das faces. Pelo que pode-se dizer que os vértices destas componentes possuem mais do que uma identidade.

Por esta razão, foi refeito o algoritmo, agora com a concentração em encontrar um mecanismo capaz de gerar cada uma das faces individualmente, e, só depois as unir a todas para formar o cubo original. Permitindo a repetição de vértices específicos, quando o objetivo é que estes possuam mais de uma normal associada.

### 2.2 ALGORITMO NORMAL

De forma a ultrapassar este problema desenvolvemos um algoritmo para gerar faces do cubo partindo unicamente da sua normal, e respectivas medidas das divisões em todos os eixos.

Um cubo pode ser visto como composto por uma banda em seu torno e duas tampas laterais, como indicado no relatório da fase 1.

# 3

---

## EXTENSÃO DO GERADOR

---

### 3.1 NORMAIS À SUPERFÍCIE

Para conseguirmos aplicar o nosso engine à capacidade de iluminação, para já, no modelo local, precisamos obrigatoriamente de definir normais para cada uma destas superfícies. Das mais diversas formas, estas normais são utilizadas em conjunto com o ponto de vista da câmara e o ponto onde se encontra a fonte de iluminação de forma a conseguir produzir resultados fidedignos.

#### 3.1.1 *Plano*

Considera-se que o plano assenta sobre o eixo  $xz$ , desta forma é notável que a normal à superfície deverá ser constante em cada um dos vértices da superfície. Assumindo este plano como voltado para cima, facilmente concluímos que a normal deverá corresponder unicamente a  $\vec{n} = \hat{j}$ .

Com influência no guião 6 desta mesma UC, e tirando partido dos mesmos recursos, achamos que seria ideal embutir o nosso sistema com a capacidade de gerar terrenos baseados em *height maps*, fornecendo a possibilidade de gerar terrenos a partir de uma imagem, o que é de enorme interesse no âmbito da geração automática de terrenos.

Porém, definir as normais à superfície torna-se mais desafiante, pois leva à necessidade de interpolação entre ponto adjacentes para obter a normal à superfície de acordo com o produto externo. Assumindo sempre que os pontos nas bordas ficam com a normal  $\hat{j}$ . Este procedimento foi alvo de debate no guião 10, sendo por isso suave em tecnicidades no presente documento.

Continua, de qualquer das formas, a ser de grande interesse verificar o resultado prático desta tecnologia, que permitiu, tirando proveito de mecanismo adiante, obter o seguinte resultado:

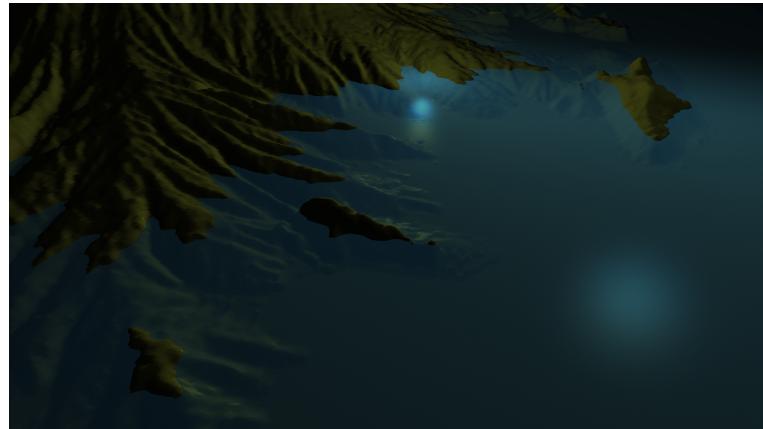


Figura 1: Terreno gerado utilizando um *height map*

### 3.1.2 *Cubo*

Partindo do algoritmo normal especificado acima, verifica-se que a normal, a cada uma das faces é imediata, isto porque a própria fase é construída partindo do vetor normal.

Com esta normal em consideração, para explicitar a normal associada a um dado vértice, é suficiente verificar a qual das faces estes pertence e efetuar a respectiva atribuição.

### 3.1.3 *Esfera*

Também um dos objetos primordiais cuja definição de normais se torna trivial partindo da forma como a sua geração está implementada. Sabe-se que a esfera possui o seu centro na origem, e que qualquer ponto dista do centro da esfera um valor exatamente igual ao raio.

Ademais, somos também capazes de verificar, pela razão acima, que o vetor que liga a origem a qualquer ponto da esfera é perpendicular à tangente naquele mesmo ponto, sendo por isso a definição perfeita de vetor normal.

Como tal, surge imediatamente, que esta normal deve responder unicamente ao valor de cada um dos pontos, devidamente normalizados.

### 3.1.4 *Cone*

O cone apresenta-se como uma das figuras sobre a qual possuímos menos controlo da normal, como tal, e devido à equação simples que surge do cone, decidimos que definir o cone matematicamente, e de seguida obter a normal pelo método do gradiente seria a alternativa mais vantajosa.

Primeiramente, confrontando o problema da base do cone, verificamos que está voltada para baixo, logo todos os pontos da superfície da base possuem uma normal igual a  $\vec{n} = -\hat{j}$ .

De seguida, focando na banda lateral do cone, devemos denotar que o raio da base e a altura do cone podem divergir entre si, sendo portanto necessário observar a razão entre estas duas variáveis, que nos fornece um método de controlo de variação das duas componentes.

$$c = \frac{R}{h} \quad (1)$$

Com a equação em cima em conta, temos agora a capacidade de definir o cone da seguinte forma, sempre assumindo que a base do cone assenta sobre o plano  $xz$ .

$$x^2 + z^2 = c^2 y^2 \quad (2)$$

Notoriamente, se forçarmos  $R = h$ , obtendo um cone com tanto da altura como de raio da base, verificaremos que a equação acima degenera na sua versão mais simplificada de  $x^2 + z^2 = y^2$ .

De forma a conseguirmos obter o gradiente, é nos suficiente resolver a equação em ordem a  $y$ , obtendo os seguintes resultados.

$$y = \sqrt{\frac{x^2 + z^2}{c^2}} \quad (3)$$

$$f(x, y, z) = y - \frac{\sqrt{x^2 + z^2}}{c} \quad (4)$$

Com isto em conta, a definição do gradiente em si é imediata.

$$\nabla f(x, y, z) = \left( \frac{-x}{c\sqrt{x^2 + z^2}}, 1, \frac{-z}{c\sqrt{x^2 + z^2}} \right) = \left( \frac{h \cdot \sin(\theta)}{R}, 1, \frac{h \cdot \cos(\theta)}{R} \right) \quad (5)$$

A partir deste mesmo gradiente, somos agora capazes de definir a normal em qualquer ponto da superfície lateral como sendo:

$$\vec{n} = \nabla f(x, y, z) \quad , \forall P = (x, y, z) : P \in \text{superfície lateral} \quad (6)$$

### 3.1.5 Toróide

Para extrair a normal do é apenas necessário tirar partido da sua equação paramétrica, na qual são utilizadas as variáveis  $u$  e  $v$ . Desta forma, e tirando partido das derivadas parciais destas equações, conseguimos, quase que imediatamente, obter a normal à superfície.

$$x(u, v) = (R + r \cdot \cos(u)) \cos(v) = R \cdot \cos(v) + r \cdot \cos(u) \cdot \cos(v) \quad (7)$$

$$y(u, v) = r \cdot \sin(u) \quad (8)$$

$$z(u, v) = (R + r \cdot \cos(u)) \sin(v) = R \cdot \sin(v) + r \cdot \cos(u) \cdot \sin(v) \quad (9)$$

$$\frac{\partial x(u, v)}{\partial u} = -r \cdot \sin(u) \cdot \cos(v) \quad (10)$$

$$\frac{\partial y(u, v)}{\partial u} = r \cdot \cos(u) \quad (11)$$

$$\frac{\partial z(u, v)}{\partial u} = -r \cdot \sin(u) \cdot \sin(v) \quad (12)$$

$$\frac{\partial x(u, v)}{\partial v} = -R \cdot \sin(v) - r \cdot \cos(u) \cdot \sin(v) \quad (13)$$

$$\frac{\partial y(u, v)}{\partial v} = 0 \quad (14)$$

$$\frac{\partial z(u, v)}{\partial v} = R \cdot \cos(v) + r \cdot \cos(u) \cdot \cos(v) \quad (15)$$

### 3.1.6 Fita de Moebius

Para extrair a normal da fita de Moebius é apenas necessário tirar partido da sua equação paramétrica, na qual são utilizadas as variáveis  $u$  e  $v$ . Desta forma, e tirando partido das derivadas parciais destas equações, conseguimos, quase que imediatamente, obter a normal à superfície.

$$x(u, v) = \left(1 + \frac{v}{2} \cdot \cos\left(\frac{u}{2}\right)\right) \cdot \cos(u) \quad (16)$$

$$y(u, v) = \frac{v}{2} \cdot \sin\left(\frac{u}{2}\right) \quad (17)$$

$$z(u, v) = \left(1 + \frac{v}{2} \cdot \cos\left(\frac{u}{2}\right)\right) \cdot \sin(u) \quad (18)$$

$$\frac{\partial x(u, v)}{\partial u} = -\left(1 + \frac{v}{2} \cdot \cos\left(\frac{u}{2}\right)\right) \cdot \sin(u) - \frac{v \cdot \sin\left(\frac{u}{2}\right) \cdot \cos(u)}{4} \quad (19)$$

$$\frac{\partial y(u, v)}{\partial u} = \frac{v \cdot \cos\left(\frac{u}{2}\right)}{4} \quad (20)$$

$$\frac{\partial z(u, v)}{\partial u} = \left(1 + \frac{v}{2} \cdot \cos\left(\frac{u}{2}\right)\right) \cdot \cos(u) - \frac{v \cdot \sin\left(\frac{u}{2}\right) \cdot \sin(u)}{4} \quad (21)$$

$$\frac{\partial x(u, v)}{\partial v} = \frac{\cos\left(\frac{u}{2}\right) \cdot \cos(u)}{2} \quad (22)$$

$$\frac{\partial y(u, v)}{\partial v} = \frac{\sin\left(\frac{u}{2}\right)}{2} \quad (23)$$

$$\frac{\partial z(u, v)}{\partial v} = \frac{\cos\left(\frac{u}{2}\right) \cdot \sin(u)}{2} \quad (24)$$

Desta forma, facilmente se obtém que a normal à superfície é o resultado imediato do produto externo entre as derivada parcial em ordem a  $u$  e a derivada parcial em ordem a  $v$ . Porém, devido à complexidade matemática desta figura, a satisfação das equações não foi suficiente, sendo que, por essa mesma razão, tivemos a necessidade de verificar imediatamente, na prática, se estas normais forneciam o resultados esperado.

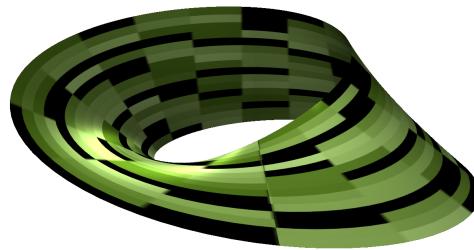


Figura 2: Fita de Moebius, 100 stacks e 100 slices.

Como se pode verificar, seguindo o padrão da luz, é imediato verificar que foi possível gerar com sucesso as normais desta figura.

### 3.1.7 Patches de Bézier

Tendo já sido debatido na fase prévia, não se pode deixar de salientar que as normais para este tipo de figura foram definidas exatamente da mesma forma. Partindo das derivadas parciais das variáveis paramétricas somos então capazes, por meio de produto externo, de obter a normal em si.

Tendo em conta que as derivadas parciais surgem, quase que, gratuitamente partindo da própria equação  $B(u, v)$ , então este método das derivadas parciais é, sem dúvida, o mais vantajoso.

## 3.2 COORDENADAS DE TEXTURA

As coordenadas de textura, ou por vezes chamadas de *texels*, necessitam de um cuidado especial, de forma a garantir um bom mapeamento da textura para a respectiva figura. Considera-se um bom mapeamento como sendo a capacidade de utilizar todos os pontos de uma textura adequadamente, produzindo resultados consistentes.

Para todas as figuras, as coordenadas de textura são definidas da mesma forma, com pouquíssimas variações entre diferentes figuras. A *texel* para cada uma das figuras é atribuída consoante a stack e slice em que um dado vértice se encontra, ou seja, as componentes das coordenadas são diretamente proporcionais ao número da stack e slice. Em objetos mais restritivos, como por exemplo, o cubo ou plano, simplesmente é feito um mapeamento direto tal que a textura ocupe cada uma das faces da figura.

# 4

---

## APLICAÇÃO DE TEXTURAS

---

De forma a contemplar texturas, neste novo contexto com *shaders* foi necessário recorrer a um **sampler2D**, que efetua a amostragem da textura a ser utilizada consoante a textura que foi associada ao **OpenGL**. Com este mecanismo, basta a seguinte linha de código em GLSL para conseguirmos captar a textura que está associada a cada fragmento.

```
uniform sampler2D ourTexture;
```

Esta textura, na prática passa a uma cor, consoante a *texel* do fragmento, sendo que para atribuir a textura ao respectivo fragmento basta efetuar a seguinte operação, na main:

```
vec4 texColor = texture(ourTexture, texCoord);
FragColor = texColor;
```

É efetuado assim o mapeamento direto da textura amostrada para o respectivo valor que se pretende utilizar. Futuramente, é do nosso interesse misturar esta cor de fragmento com a luz calculada, o que será analisado de seguida.

Com este nível de liberdade, oferecido pela programação em shaders, temos agora a capacidade embutir diversos comportamentos de interesse no sistema, por exemplo, a título de exemplo, podemos tentar mistura a textura de um fragmento com a sua normal à superfície, o que pode ser realizado com o seguinte excerto de código:

```
FragColor = vec4(normCoord, 1) * texColor;
```

O que permite obter resultados interessantes, como o que se segue, no qual podemos distintamente verificar a existência de uma textura cuja cor se encontrada misturada com o seu vetor normal à superfície.

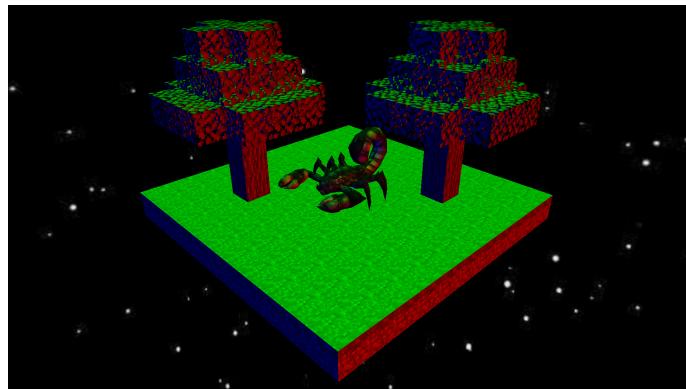


Figura 3: Mistura de normais com texturas.

De igual forma podemos também aplicar um filtro sobre a textura, tal que está possua uma cor mais avermelhada, através do seguinte excerto:

```
FragColor = vec4(1, 0, 0, 1) * texColor;
```

O que permite obter o seguinte resultado, no qual evidentemente denotamos um tom avermelhado, com estes exemplos é nós possível ganhar alguma noção sobre as possibilidades que a programação em shaders abre, só ao nível das texturas.

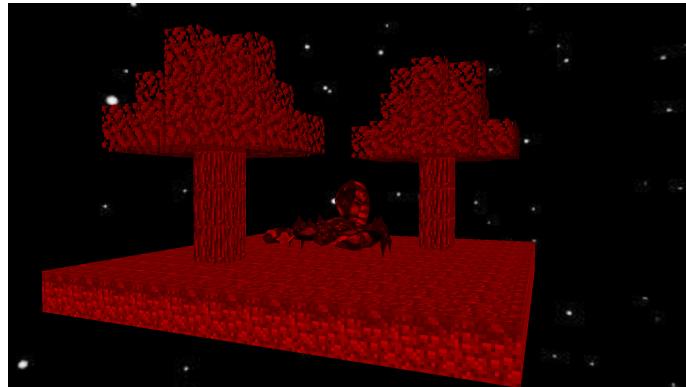


Figura 4: Mistura de normais com texturas.

# 5

---

## IMPLEMENTAÇÃO DA LUZ

---

A luz demonstrou-se, sem dúvida, com um dos assuntos mais sensíveis desta última fase. Inicialmente, o objetivo era implementar este mecanismo recorrendo unicamente a OpenGL, porém, rapidamente nos apercebemos que atualmente este funcionamento já não é utilizado. Por isso, e apesar do *overhead* associado a um primeiro contacto com shaders, decidimos que seria mais vantajoso do ponto de vista funcional, especialmente no âmbito da iluminação.

A vantagem mais básica que podemos apontar é que, tendo a iluminação feita no *fragment shader*, então automaticamente estamos a utilizar o modelo de shading de Phong, ao invés de Gouraud, que o OpenGL obriga, o que permite obter resultado mais fidedignos.

Porém, a título informativo, se o objetivo fosse aplicar o método de shading de Gouraud, então teríamos de tratar da iluminação no *vertex shader*, calculando assim a equação da luz apenas 1 vez por vértice.

### 5.1 MATERIALS

Como indicado em capítulos anteriores, é sempre tido em conta os diferentes materiais que podem compor um objeto. Com isto em mente, é preciso ter sempre em conta que as características do material estão diretamente associadas com a luz que deve ser produzida, propriedades como a componente difusa, especular, ambiente ou de brilho devem ser todas tidas em conta quando estamos a calcular a equação da luz.

Desta forma, e sabendo que cada tipo de material é desenhado separadamente, podemos transmitir o material que estamos a considerar para o OpenGL da seguinte forma:

```
blockIndex = glGetUniformLocation(id_shader, "Materials");
glUniformBlockBinding(id_shader, blockIndex, 1);
 glBindBufferBase(GL_UNIFORM_BUFFER, 1, (*material_bufs)[i]);
```

Este mecanismo tira partido de blocos uniformes, previamente carregados no sistema, e, por meio de *binding points*<sup>1</sup> é capaz de transmitir ao GLSL esta informação.

---

<sup>1</sup> <https://www.lighthouse3d.com/tutorials/glsl-tutorial/uniform-blocks/>

O shader em questão, que neste caso é o `base` shader, capta esta informação no *fragment shader*, através do seguinte mecanismo:

```
struct Material {
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};

layout (std140) uniform Materials {
    Material mat;
};
```

O *layout standard 140* foi usado para permitir uma facilidade com a comunicação para o shader, evitando assim o uso de *offsets* específicos, claro que isto possui algumas implicações, como discutido adiante.

Assim, somos capazes de ter em atenção as diferentes componentes de cada um dos materiais e utiliza-las no cálculo da iluminação.

## 5.2 ILUMINAÇÃO

Com o tratamento de materiais já definidos, somos agora capazes de começar a lidar com a luz *de per si*. Primeiramente, precisamos de ter a noção de que existem diferentes tipos de luzes, nomeadamente:

- **Luz Pontual** - Situada numa posição bem definida e que produz luz em todas as direcções.
- **Luz Direccional** - Luz infinitamente instantemente que ilumina o "universo" segundo uma determinada direção.
- **Luz de Foco** - Corresponde a uma *spotlight*, situada numa posição a iluminar numa determinada direção, que permite obter focos de luz específicos.

Num sistema, é também preciso ter atenção, que podes existir várias luz a atuar simultaneamente no sistema, e que no final estaremos perante uma combinação de todas estas. As luzes são definidas em C++ e de seguida comunicadas ao shader, que as armazena da seguinte forma:

```
layout (std140) uniform Lights {
```

```
Light lights[MAX_LIGHT_UNITS];
};
```

Exerto no qual se assume que `Light` encapsula todas as propriedades de cada luz, inclusive o inteiro `type` e `isOn`, que indica o tipo de luz e se essa luz se encontra ligada. Pela definição acima, podemos perceber que surge uma nova vantagem em usar shaders: o número de luzes que podemos utilizar fica totalmente ao nosso controlo, e temos assim capacidade de utilizar mais de 8 luzes, o valor predifinido no OpenGL.

As diferentes luzes são calculadas de diferentes formas, cada uma com o seu cálculo específico no qual é contabilizado o tipo de material e o seu brilho, bem como todas as componentes da luz em questão.

### 5.2.1 Composição de Luzes

As várias luzes podem ser compostas entre si acumulando as suas intensidades, e de seguida multiplicando este valor pela respectiva textura, isto permite obter a iluminação acumulada de todas as componentes naquele fragmento.

Como tal, basta percorrer o array de todas as luzes, e dependendo da tipicidade desta, aplicamos a adequada função de processamento da luz, se e só se esta luz estiver ligada. O pode ser implementado pelo seguinte mecanismo:

```
vec3 res = vec3(0.0);

for(int i = 0; i < MAX_LIGHT_UNITS; i++) {
    if(lights[i].isOn == 1) {
        switch(lights[i].type){
            case POINT_LIGHT:
                res += calcPointLight(lights[i],Normal,viewDir);
                break;
            case DIRECTIONAL_LIGHT:
                res += calcDirLight(lights[i],Normal,viewDir);
                break;
            case SPOT_LIGHT:
                res += calcSpotLight(lights[i],fragPos,Normal,viewDir);
                break;
        }
    }
}
FragColor = vec4(res,mat.ambient.w) * texColor;
```

Assim, somos capazes de misturar todas as luzes, a última linha, a componente `mat.ambient.w` permite fornecer a todos os fragmentos uma componente translúcida.

# 6

---

## IMPLEMENTAÇÃO DE CUBEMAPS

---

Baseado num artigo específico sobre *cubemaps*<sup>1</sup> fomos capazes de estender este mecanismo para o funcionamento do nosso *engine*. Para o funcionamento deste mecanismo a ideia, de forma geral, assenta em fazer uma amostragem das texturas utilizadas. Porém, em vez de ser utilizado um *sampler* bidimensional, é agora necessário recorrer a um amostrador tridimensional, em que a amostrar da textura, a utilizar naquele fragmento, é dependente da sua posição no espaço.

As texturas que compõem o cubemap são previamente carregadas para o OpenGL. Após ser carregado para o OpenGL, e após o desenho de todos os outros objetos, é, por fim, desenhada a *sky box*, recorrendo a um shader especializado.

Para definir a skybox, no shader, basta indicar que a coordenada do fragmento não se trata de um ponto, mas sim de um vetor, desta forma, conseguimos colocar a skybox infinitamente longe.

Para tal, basta no *vertex shader* definir a posição da seguinte forma:

```
gl_Position = projection * view * vec4(aPos, 0.0);
```

Onde *aPos* corresponde à posição do fragmento no espaço, finalmente o *fragment shader* fica encarregue por mapear a skybox, utilizando o sampler tridimensional, para a coordenada de textura em uso.

---

<sup>1</sup> <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

# 7

---

## DEMONSTRAÇÕES TECNOLÓGICAS

---

### 7.1 SISTEMA SOLAR

Como proposto pelos docentes foi desenvolvido uma demonstração tecnológica que visa emular o comportamento do sistema solar. Nomeadamente, nos seguinte aspectos:

1. Todos os planetas giram à volta do sol, e sobre o seu próprio eixo.
2. O sol é a única fonte de luz, que ilumina todos os outros planetas e luas.
3. As luas giram à volta dos seus planetas, com períodos diferentes.
4. Existe um cometa a ultrapassar o sistema solar.

O que podemos ver concretizado na seguinte imagem, com todas as necessárias componentes e transformações geométricas. Nunca esquecendo, que para simular as estrelas distantes, colocamos explicitamente uma *skybox* correspondente às estrelas.



Figura 5: Vista lateral do sistema solar



Figura 6: Vista do sistema solar, do ponto de vista do cometa

## 7.2 SANTUÁRIO DE ATENA

Baseado nos míticos contos de mitologia das diferentes divindades, pretendeu-se prestar homenagem. Desta forma, dedicamos à deusa grega *Athena* uma estátua gigante, situada no meio de uma montanha guardar por letais escorpiões que caminham ao longo da pista de entrada, garantindo a segurança do artifício. Esta montanha encontra-se num terreno não identificado, porém, podemos vislumbrar a zona em seu redor.



Figura 7: Vista aérea do santuário



Figura 8: Ponto de vista de um personagem na ponte do santuário

### 7.3 MINECRAFT

Evidentemente que não nos podíamos esquecer do icônico cenário do Minecraft, um mundo rodeado por blocos simples. Formando um ambiente de jogo interessante, utilizado as texturas *atlas* desenvolvidas, conseguimos emular esse comportamento. Obtendo o seguinte resultado:



Figura 9: Cenário baseado em Minecraft

Importante salientar a transparência dos materiais utilizados, por exemplo, conseguimos ver, por entre as folhas da árvore, as estrelas no fundo. Ademais, conseguimos também ver a transparência característica da água, sendo possível ver o subsolo deste cenário.

# 8

---

## CONCLUSÕES E TRABALHO FUTURO

---

O engine, parser, e todas as ferramentas que foram desenvolvidas no âmbito deste projeto, todas apresentaram uma enorme componente didáctica, e todas envolviam uma forte ligação entre elas. O generator deveria ser capaz de fornecer inputs adequados ao parser, que por si mesmo assentava no engine para complementar o seu funcionamento interno.

Com isto em mente, não podemos deixar de sentir que o trabalho em Computação Gráfica envolve uma grande dinâmica entre diferentes componentes e tecnologias, com uma constante adaptação ao que é mais útil ao utilizador. Desta forma temos que este sistema é uma peça tecnológica de moderada complexidade, com uma enorme variedade de componentes interligadas entre si, tirando partido de lirrarias externas, bem como mecanismos mais avançados de shaders utilizando GLSL.

Com um olhar no futuro, não podemos deixar de denotar que muito falta para podermos referenciar o nosso *toy engine* como uma ferramenta verdadeiramente útil na prática. Porém, salientamos alguns dos pontos nos quais gostaríamos de nos focar, e implementar, no futuro de forma a completar este mesmo sistema e obter, dessa forma, um resultado ainda mais gratificante.

1. ***View Frustum Culling***: Porém, devido à utilização do OpenGL modern, achamos que deveríamos implementar este comportamento diretamente no *tessellation shader*.
2. ***Shadow Mapping***: tendo em mente os resultados satisfatórios que obtivemos, não podemos deixar de pensar no que seria se conseguíssemos incluir alguns mecanismos de iluminação global no sistema, e de que forma estes poderiam ser usados à nossa vantagem.
3. ***Normal Mapping***: em texturas mais específicas, temos de considerar que as normais podem não seguir simplesmente as normais da figura sobre a qual assentam, e que a própria textura em si acarreta alguma consideração pelo relevo da textura, razão pelo qual achamos que seria interessantíssimo a incorporação deste recurso.
4. ***Loaders adicionais***: O universo de diferentes extensões de modelos de computação gráfica é enorme, por essa mesma razão achamos que seria de enorme interesse, não só

completar o *loader* de objetos do tipo *Wavefront Objects*, mas também suplementar o sistema de um maior universo de extensões.

5. **Texturas Comprimidas:** De forma a garantir uma menor ocupação de ficheiros, existe o potencial de ler texturas que estão comprimidas em memória, como, por exemplo, o formato DDS<sup>1</sup>. Achamos que a implementação deste mecanismo seria de um grande interesse do ponto de vista prático.
6. **Perlin Noise<sup>2</sup> na geração procedural de terreno infinito:** Tendo obtido os resultados indicados com *height maps*, não podemos deixar de expressar o quanto interessante seria sermos capazes de gerar um terreno automaticamente utilizando ruído, permitindo a criação de um mundo, na prática, infinito, ao estilo de muitos vídeo jogos no mercado atualmente.

Para não mencionar que o espectro de potenciais implementações é praticamente ilimitado, por essa razão, imensas ideias se encontram omitidas desta lista, apesar de possuírem tanto ou mais interesse do ponto de vista prática.

Finalmente, em meio de conclusão, achamos que o trabalho desenvolvido foi de um enorme interesse, temos sido capazes de vislumbrar todas as diversas componentes que perfazem um verdadeiro engine, permitindo também uma experiência em primeira pessoa de todas as tecnologias introduzidas nas aulas teóricas, como a iluminação global, curvas, e texturas. Consideramos que, no âmbito do que foi pedido, conseguimos desenvolver uma tecnologia capazes de satisfazer e ultrapassar os requisitos mínimos por uma grande margem, permitindo obter uma implementação moderna das tecnologias em questão.

---

1 [https://en.wikipedia.org/wiki/DirectDraw\\_Surface](https://en.wikipedia.org/wiki/DirectDraw_Surface)

2 [https://en.wikipedia.org/wiki/Perlin\\_noise](https://en.wikipedia.org/wiki/Perlin_noise)