

Universidade do Minho
Escola de Engenharia

Computação Gráfica

Trabalho Prático – Fase IV

Grupo 3

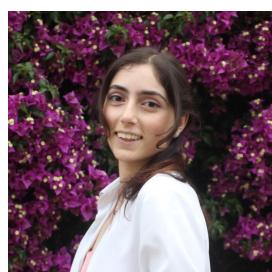
18 de maio de 2025



Ana Oliveira



Humberto Gomes



Mariana Cristino



Sara Lopes

A104437

A104348

A90817

A104179

Resumo

Nesta fase do trabalho prático, continuou-se o desenvolvimento dos programas `engine` e `generator`. Em particular, na `engine`, foram implementadas a leitura de texturas e a iluminação da cena. É lógico que estas funcionalidades exigiram alterações à estrutura de armazenamento de modelos em VBOs, ao formato XML da cena, e também a criação de novos *shaders*, para implementação dos modelos de iluminação e de *shading* de Phong. Adicionalmente, também foi implementado *object picking* e geração automática de normais, para modelos que não têm essa informação. Do lado do `generator`, foi necessário implementar a geração de normais e coordenadas de textura para as figuras, bem como atualizar a geração do Sistema Solar, para adicionar informação de texturas e iluminação. Em suma, apesar de se considerar que o trabalho desenvolvido foi além do que era pedido pelo enunciado, ainda haveria muitas possibilidades de funcionalidades a implementar em hipotéticas fases futuras (*instanced rendering, normal maps, sombras, tesselação, etc.*).

1 *Generator*

1.1 Formato .3d

O formato `.3d` exportado pelo `generator` e utilizado pela `engine` é o Wavefront OBJ [1]. Apenas uma pequena fração das funcionalidades deste formato são suportadas e, nesta fase, foram necessárias adições aos mecanismos de escrita e leitura de ficheiros Wavefront OBJ para suportar coordenadas de textura e normais. Este formato é textual, onde cada linha pode ser um comentário, uma posição, uma coordenada de textura, uma normal, ou uma face triangular, como mostra o exemplo abaixo, na ordem apresentada:

```
# Comment
v 0.5 0.5 1
vt 0.3 0.3
vn 0 1 0
f 1/2/3 4/5/6 7/8/9
```

Quando uma linha começa com `v`, `vt` ou `vn`, devem seguir-se as coordenadas de uma posição, de uma textura, ou de um vetor normal, respetivamente. Quando uma linha começa com `f`, deve seguir-se uma face triangular, ou seja três pontos. O `generator` e a `engine` suportam dois tipos de ponto:

- Apenas um número, um índice de uma posição, ou seja, um elemento do tipo `v` (começando a contar em 1);

- Da forma $v/t/n$, onde estão presentes três índices, um para uma posição, um para uma coordenada de textura, e outro para um vetor normal.

Como o *parser* de ficheiros Wavefront OBJ foi reimplementado na fase anterior para se basear em expressões regulares, foi trivial adicionar o suporte para coordenadas de textura e vetores normais.

1.2 Plano Horizontal

A geração de normais do plano horizontal é trivial, todos os pontos partilham o mesmo vetor normal, $(0, 1, 0)$. Quanto às coordenadas de textura, s aumenta entre 0 e 1 à medida que z aumenta, e t aumenta entre 0 e 1 à medida que x aumenta. Na expressão abaixo, N representa o número de divisões do plano, i a variável de iteração que aumenta ao longo do eixo x , e j a variável de iteração que aumenta ao longo do eixo z :

$$s_j = \frac{j}{N} \quad t_i = \frac{i}{N} \quad i, j \in \{1, 2, \dots, N\}$$

1.3 Cubo

Para a geração das normais do cubo, já durante a geração das posições do vértice do cubo foi necessário gerar as seis normais, uma para cada face (ver relatório da primeira fase). A normal de uma face é utilizada em todos os vértices dessa face.

Em relação à geração de coordenadas de textura, foram implementados dois modos de geração. No modo simples, a totalidade da textura é utilizada nas seis faces do cubo: tal como no plano, em cada face, s e t variam entre 0 e 1. Já no modo *multi-textured*, a cada face é aplicada uma parte diferente da textura:

$$s \in \left[s_0, s_0 + \frac{1}{4}\right] \quad t \in \left[t_0, t_0 + \frac{1}{3}\right],$$

tal que s_0 e t_0 assumem diferentes valores para diferentes faces, de modo que a textura é aplicada às faces do cubo como descreve a imagem abaixo:

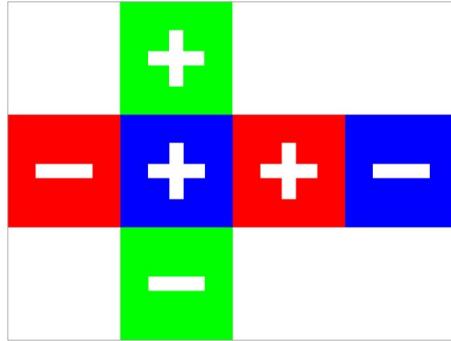


Figura 1: Identificação das faces de um cubo numa textura para o seu modo *multi-textured*.

1.4 Esfera

As normais da esfera são calculadas de forma muito simples: para cada vértice, a normal é o vetor que vai do centro da esfera até esse vértice, normalizado. Como a esfera está centrada na origem, isso equivale a normalizar o próprio vetor posição do vértice.

Em todos os pontos da esfera que não os polos, para determinar as coordenadas de textura (s, t) , o valor de s varia entre 0 e 1 ao longo da longitude, e o valor de t varia entre 0 e 1 ao longo da latitude:

$$s_j = \frac{j}{N_{\text{slices}}} \quad t_i = \frac{i}{N_{\text{stacks}}} \quad i \in \{1, 2, \dots, N_{\text{stacks}}\} \quad j \in \{1, 2, \dots, N_{\text{slices}}\}$$

Nos polos da esfera, como são criados triângulos, e não quadrados, o método apresentado acima não é aplicável. Logo, o vértice do polo é replicado várias vezes, e criam-se triângulos cujo polo tem como coordenada s o valor de s do ponto médio dos vértices da base, como mostra a figura abaixo:

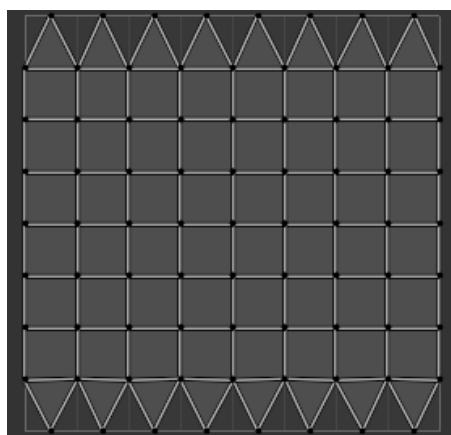


Figura 2: Triângulos da esfera no espaço de textura.

Matematicamente, t assume o valor 1 no polo norte e 0 no polo sul, enquanto que s é dado pela seguinte expressão:

$$s_j = \frac{j + \frac{1}{2}}{N_{\text{slices}}}$$

Com este método, parte da textura é descartada, mas torna-se possível aplicar uma imagem 2D na superfície da esfera de forma contínua, repartindo a textura entre os vários triângulos sem quebras visuais visíveis.

1.5 Cone

No cone, as normais são definidas de forma diferente para diferentes vértices. Em primeiro lugar, na base do cone as normais são verticais e apontam para baixo: $(0, -1, 0)$, enquanto que na sua superfície lateral, as normais são vetores perpendiculares a cada aresta, sendo dadas pela seguinte expressão:

$$\vec{n} = \begin{bmatrix} \cos \theta \\ \frac{r}{h} \\ \sin \theta \end{bmatrix},$$

onde θ é o ângulo azimutal da aresta, r o raio da base e h a altura do cone. É lógico que o vetor acima deve ser normalizado antes de ser adicionado ao modelo. Analisando o vetor acima, facilmente se conclui que os termos $\cos \theta$ e $\sin \theta$ do vetor fazem com que a sua projeção horizontal aponte para fora da aresta a que se refere. Quanto à componente y deste vetor, considerando o plano vertical que contém o eixo do cone e a aresta, $\frac{r}{h}$ é o declive de uma reta perpendicular à aresta, de declive $-\frac{h}{r}$.

O processo explicado acima também é utilizado no vértice do cone, que é replicado várias vezes, uma vez para cada aresta.

As coordenadas de textura são também atribuídas da forma diferente consoante a parte do cone a ser gerada. As coordenadas da base do cone foram uma circunferência de centro $(\frac{1}{2}, \frac{1}{2})$ tangente à fronteira da textura (raio $\frac{1}{2}$):

$$s_j = \frac{1 + \cos \theta_j}{2} \quad t_j = \frac{1 + \sin \theta_j}{2} \quad j \in \{1, 2, \dots, N_{\text{slices}}\},$$

onde θ é o ângulo azimutal. Na superfície lateral, o valor de s varia ao longo da circunferência, e t varia com a altura:

$$s_j = \frac{j}{N_{\text{slices}}} \quad t_i = \frac{i}{N_{\text{stacks}}} \quad i \in \{1, 2, \dots, N_{\text{stacks}}\} \quad j \in \{1, 2, \dots, N_{\text{slices}}\}$$

Tal como acontece na esfera, parte da textura será descartada junto aos polos.

1.6 Cilindro

No cilindro, as normais são definidas conforme a parte do modelo a ser gerada. Nas bases inferior e superior do cilindro, as normais são $(0, -1, 0)$ e $(0, 1, 0)$, respectivamente. Na superfície lateral do cilindro, as normais são vetores radiais no plano XZ , ou seja, $(x, 0, z)$ após uma normalização.

Para o cilindro, foram implementados dois modos de gerar coordenadas de textura. No modo simples, as coordenadas das bases são calculadas tal como no cone, formando uma circunferência de centro $(0.5, 0.5)$. Na superfície lateral, o valor de s varia ao longo da circunferência, e t varia com a altura, também como no cone. Já no modo *multi-textured*, aplicam-se as mesmas regras, mas os intervalos dos valores s e t são ajustados para percorrer apenas partes de uma imagem maior que contém várias secções, duas bases e a superfície lateral:

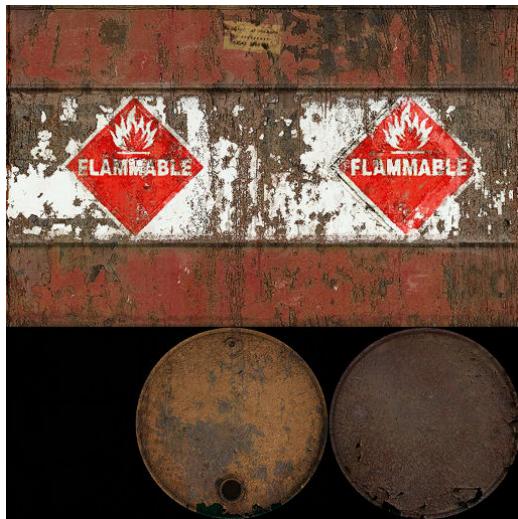


Figura 3: Textura para um cilindro *multi-textured*.

1.7 *Torus*

Previamente, para a geração dos pontos do *torus*, já era necessário conhecer o vetor normal a cada ponto: este corresponde ao vetor do raio da secção radial nessa *slice*, normalizado:

$$\hat{n} = \begin{bmatrix} \cos \theta \cos \phi \\ \sin \phi \\ \sin \theta \cos \phi \end{bmatrix},$$

onde θ representa o ângulo em torno do eixo central, e ϕ o ângulo na circunferência que é a secção transversal do tubo.

A geração de coordenadas de textura é muito simples: a coordenada s aumenta à medida que se roda um ponto à volta do eixo central do *torus*, e t varia entre 0 e 1 em cada secção transversal do tubo:

$$s_i = \frac{i}{N_{\text{slices}}} \quad t_j = \frac{j}{N_{\text{sides}}} \quad i \in \{1, 2, \dots, N_{\text{slices}}\} \quad j \in \{1, 2, \dots, N_{\text{stacks}}\}$$

1.8 Geração de Modelos com Base em *Patches* de Bézier

Ao gerar uma superfície de Bézier, é necessário calcular as normais e as coordenadas de textura de cada ponto da malha. Para calcular a normal de um ponto, é necessário conhecer em primeiro lugar as derivadas parciais da curva em relação a u e a v nesse ponto:

$$\frac{\partial p(u, v)}{\partial u} = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} M P_c M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}, \quad u, v \in [0, 1]$$

$$\frac{\partial p(u, v)}{\partial v} = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M P_c M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}, \quad u, v \in [0, 1]$$

Para uma descrição mais detalhada das expressões acima, é recomendada a consulta do relatório da fase 3. As expressões acima são aplicadas uma vez para cada coordenada, x , y , e z , dando origem a dois vetores, que se denominarão du e dv . Estes vetores são tangentes à superfície no ponto, pelo que o vetor normal pode ser dado por:

$$\vec{n} = dv \times du$$

Logicamente, este vetor deve ser normalizado antes de ser adicionado ao modelo. Durante o cálculo das derivadas parciais, valores como $du = \vec{0}$ ou $dv = \vec{0}$ podem surgir na fronteira da superfície e dar originar vetores normais inválidos. Para evitar estes casos, na fronteira da superfície são utilizados valores de u e v ligeiramente acima de zero para garantir a estabilidade do cálculo das normais.

As coordenadas de textura s e t são diretamente baseadas nos parâmetros u e v de cada *patch*, respetivamente, que variam entre 0 e 1.

1.9 Outras Figuras

Para as restantes figuras, devido à sua complexidade e ao pouco tempo disponível para a conclusão desta fase do trabalho prático, não foram adicionadas nem coordenadas de textura nem normais. No entanto, a `engine` ainda é capaz de importar estes modelos e de os desenhar com iluminação, graças a um algoritmo de geração automática de normais implementado e descrito posteriormente neste relatório.

1.10 Sistema Solar

Nesta fase, a cena do Sistema Solar evoluiu significativamente com a introdução de iluminação, texturas e materiais. Em primeiro lugar, a cada modelo, foi adicionada uma textura. Depois, para simular o Sol, uma *point light* foi colocada na origem da cena. Para o modelo do Sol não ser representado pela sua cor ambiente, foi-lhe adicionado um material com uma componente emissiva branca.

2 Engine

2.1 Geração Automática de Normais

A `engine` é capaz de carregar modelos que não tenham informação sobre coordenadas de texturas ou normais. Caso não haja informação sobre as coordenadas de textura de um modelo, é possível desenhá-lo a uma cor sólida, mas é necessário que se tenha informação sobre as suas normais para o iluminar corretamente. Como esta nem sempre está presente, foi implementado um algoritmo para gerar normais de modelos automaticamente.

Este algoritmo considera o modelo inteiro como um único *smoothing group*, e calcula a normal de cada vértice como a média das normais dos triângulos a que este pertence, pesada pela área dos triângulos. Logo, é necessário, em primeiro lugar, uma forma de calcular a normal de um triângulo. Para um triângulo $[ABC]$, esta pode ser calculada do seguinte modo:

$$\hat{n}_{[ABC]} = \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{\|\overrightarrow{AB} \times \overrightarrow{AC}\|}$$

Depois, a área de cada triângulo, A , pode ser calculada pela fórmula de Heron:

$$S = \frac{\|\overrightarrow{AB}\| + \|\overrightarrow{AC}\| + \|\overrightarrow{BC}\|}{2}$$

$$A = \sqrt{S(S - \|\overrightarrow{AB}\|)(S - \|\overrightarrow{AC}\|)(S - \|\overrightarrow{BC}\|)}$$

Logo, sendo F o conjunto de faces triangulares nas quais um ponto está presente, o vetor normal desse ponto é dado por:

$$\hat{n} = \frac{\sum_{f \in F} A_f \hat{n}_f}{\left\| \sum_{f \in F} A_f \hat{n}_f \right\|} = \frac{\sum_{f \in F} A_f \hat{n}_f}{\left\| \sum_{f \in F} A_f \hat{n}_f \right\|}$$

Em termos de implementação deste algoritmo, um dicionário é utilizado para armazenar associações entre posições de pontos e normais pesadas. Iteram-se por todas as faces do modelo e, para cada face, calcula-se a sua normal e a sua área. Para cada ponto nessa face, adiciona-se

ao valor armazenado de normal pesada $A \hat{n}$. Após iterar por todas as faces, itera-se por todas as posições no dicionário, e define-se a normal de cada ponto o como resultado da normalização do vetor normal pesado a si associado.

2.2 VBOs

Com a adição de coordenadas de textura e normais, foi necessário criar mais VBOs para as armazenar. Agora, cada modelo passa a ter um VAO associado a três VBOs, como mostra a figura abaixo:

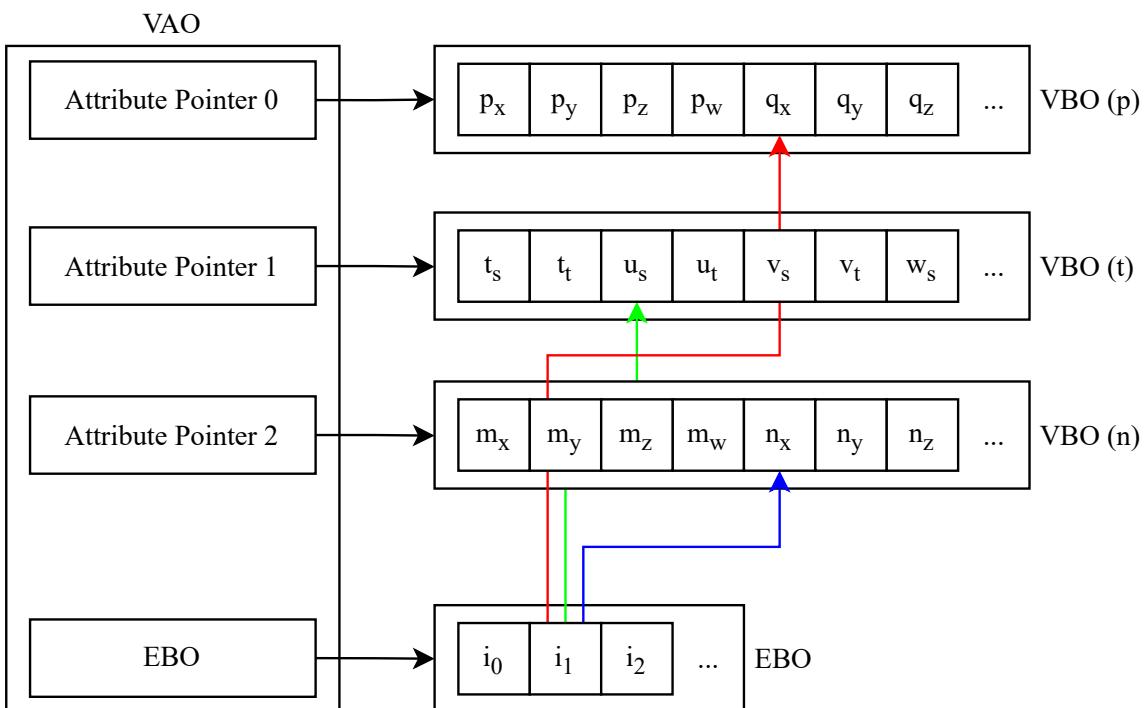


Figura 4: Organização do VAO, dos VBOs, e do EBO de um modelo.

Pode observar-se que, ao contrário do que acontece nos ficheiros .3d, não pode haver vértices formados por posições, coordenadas de textura, e normais com diferentes índices. Para isso, seria necessário mais do que um *buffer* de índices, e isso não é suportado pelo OpenGL. Logo, após carregar um modelo, é necessário converter o esquema de indexação de um ficheiro Wavefront OBJ para o de um *index buffer*. Para o fazer, um algoritmo simples pode ser utilizado:

- Cria-se um *index buffer* e três outros *buffers*, para armazenamento das posições, das coordenadas de textura, e das normais, inicialmente vazios;

- Armazena-se um dicionário que associa tuplos posição-c.textura-normal a índices (elementos do *index buffer*);
- Iteram-se por todos os vértices no modelo. Para cada vértice:
 - Constrói-se o tuplo posição-c.textura-normal associado ao vértice, que se procura no dicionário;
 - Caso seja encontrado, adiciona-se o índice encontrado ao *index buffer*.
 - Caso contrário, coloca-se o tuplo no dicionário; adicionam-se a posição, a coordenada de textura e a normal aos seus respetivos *buffers*; e adiciona-se o índice de um destes elementos ao *index buffer*.

2.3 Adições ao *Schema XML*

Para suportar as funcionalidades obrigatórias desta fase do trabalho prático, o *schema XML* foi alargado para suportar materiais com múltiplas componentes de cor, texturas, e fontes de luz diversas.

2.3.1 Materiais

A cada modelo, pode ser associado um conjunto de propriedades de material, que influenciam a forma como este interage com a luz. Estas propriedades são definidas no elemento `<color>`, onde é possível definir as componentes `diffuse` (componente difusa), `ambient` (componente ambiente), `specular` (componente especular) e `emissive` (componente emissiva), bem como o valor de `shininess`. Caso o elemento `<color>` não esteja presente, um conjunto pré-definido de valores são utilizados por omissão.

```

<model file="sphere.3d">
  <color>
    <diffuse R="200" G="200" B="200" />
    <ambient R="50" G="50" B="50" />
    <specular R="0" G="0" B="0" />
    <emissive R="0" G="0" B="0" />
    <shininess value="0" />
  </color>
</model>

```

2.3.2 Texturas

O elemento `<texture>` é utilizado para definir o caminho para uma imagem a ser aplicada como textura de um modelo. A ausência deste elemento implica que o modelo será renderizado apenas com base nas cores definidas no seu material.

```
<model file="cylinder.3d">
    <texture file="metal.jpg" />
</model>
```

No carregamento da cena, o caminho da textura é resolvido com base na diretoria do ficheiro XML.

2.3.3 Iluminação

Nesta fase, foi adicionado suporte para três diferentes tipos de fonte de luz: *point lights*, luzes direcionais e *spotlights*. Todas as fontes de luz devem ser declaradas dentro do elemento `<lights>`, filho do elemento `world`. Cada elemento `<light>` possui um atributo `type`, e argumentos adicionais consoante o tipo de luz especificado:

- `point`: `posX`, `posY`, `posZ` (posição);
- `directional`: `dirX`, `dirY`, `dirZ` (direção);
- `spotlight`: posição e direção (como as `point` e `directional lights`), bem como o ângulo de corte (`cutoff`).

```
<lights>
    <light type="point"          posX="0" posY="10" posZ="0" />
    <light type="directional"   dirX="1" dirY="1" dirZ="1"/>
    <light type="spotlight"     posX="0" posY="10" posZ="0"
          dirX="1" dirY="1" dirZ="1"
          cutoff="45" />
</lights>
```

2.4 Texturas e Iluminação

O principal objetivo desta fase do trabalho prático é a adição de texturas e iluminação ao projeto. Com a informação de coordenadas de textura e normais nos modelos, e informação

sobre luzes, materiais, e caminhos para imagens nos ficheiros de cena, é agora possível que a `engine` desenhe os objetos de uma cena iluminados e com texturas.

Para carregar as texturas para a memória, a biblioteca `stb_image` [2] é utilizada. Como o referencial de coordenadas de textura em OpenGL tem a sua origem no canto inferior esquerdo, mas o referencial regularmente utilizado em imagens tem a sua origem no canto superior esquerdo, a função `stbi_set_flip_vertically_on_load` é utilizada para inverter verticalmente todas as imagens lidas. Depois de carregar uma imagem, é possível criar uma textura, vinculá-la, definir os seus parâmetros de *wrapping* e *filtering*, enviar os dados da textura para a GPU, gerar *mipmaps*, e libertar a memória ocupada pela imagem inicialmente carregada. Tal como é feito para o carregamento de modelos, caso uma cena referecie a mesma imagem mais do que uma vez, apenas uma textura será criada, assim poupando memória da GPU.

Para desenhar os objetos da cena com texturas e iluminação, foi necessário criar novos *shaders* para o efeito. Como alguns objetos na cena não são iluminados (eixos, linhas de animação, esferas encapsuladoras, *etc.*), é necessário trocar de programa sempre que se deseja desenhar um destes objetos. Para minimizar o número de trocas de programa, uma operação com um custo elevado para o desempenho, todos os objetos não iluminados são desenhados em primeiro lugar, e só depois se renderizam os restantes.

Ao contrário da *fixed-function pipeline* do OpenGL, que implementa *shading* de Gouraud, os *shaders* desenvolvidos utilizam *shading* de Phong, onde as equações da luz são computadas para cada fragmento, com base em normais interpoladas pelo *shader* de vértices. Assim, as manchas especulares são representadas muito mais claramente do que seriam caso a *fixed-function pipeline* tivesse sido utilizada. Ademais, devido ao uso de *shaders*, é possível ter mais do que oito luzes numa cena, o número exigido pelo enunciado. Agora, o limite máximo de luzes é ditado pelo número máximo de variáveis uniformes num programa, que depende do *hardware* utilizado.

Antes de apresentar os *shaders* desenvolvidos, é necessário apresentar alguma notação para as matrizes utilizadas:

- P - Matriz de projeção, que converte coordenadas do espaço da câmara para *clip-space*;
- V - Matriz de vista, que converte coordenadas do espaço do mundo para o espaço da câmara;
- M - Matriz do modelo, que converte coordenadas do espaço local (do modelo) para o espaço do mundo;

Comece-se por perceber o funcionamento do *shader* de vértices. Este, como o outro *shader* previamente desenvolvido, também multiplica as coordenadas dos pontos do modelo a desenhar

pela matriz PVM (passada ao *shader* numa variável uniforme), colocando-as em *clip-space* antes de serem passadas ao *shader* de fragmentos. No entanto, agora também é necessário tratamento das coordenadas de texturas e das normais. As coordenadas de textura são passadas ao *shader* de fragmentos sem qualquer transformação (apenas interpolação). As normais, para continuarem perpendiculares às superfícies a que se referem, mesmo após a aplicação de escalas não uniformes, são multiplicadas pela matriz $(M^T)^{-1}$ [3], também passada ao *shader* numa variável uniforme. Isto converte-as para o espaço do mundo, onde os cálculos das equações da luz serão feitos¹. Em último lugar, é calculada e passada ao *shader* de fragmentos (após uma interpolação) a posição do vértice no espaço do mundo, ou seja, a posição do vértice do modelo multiplicada pela matriz M , passada ao *shader* de vértices numa variável uniforme.

O *shader* de fragmentos, antes de poder calcular a cor de qualquer ponto, deve, em primeiro lugar, renormalizar o vetor normal que recebe interpolado do *shader* de fragmentos, visto que o seu comprimento pode diminuir durante a interpolação. Desta operação resulta o vetor que se denominará \hat{n} . Ademais, também é necessário calcular a direção do fragmento para a câmara, \hat{e} , dada pela diferença entre a posição da câmara e a posição do fragmento, seguida de uma normalização.

Para cada luz, é necessário calcular o vetor normalizado que aponta para a luz, \hat{l} . Para as luzes direcionais, este valor é constante e passado ao *shader* por uma variável uniforme. Para as *point lights* e *spotlights*, este vetor é a diferença entre a posição da luz e a posição do fragmento, seguido de uma normalização.

De um modo geral, o contributo da i -ésima luz para a componente difusa da cor de um fragmento, D_i é dado pela seguinte expressão [3], onde K_d representa a componente difusa do material aplicado ao objeto:

$$D_i = K_d \times \max \left(0, \cos \left(\angle(\hat{n}, \hat{l}) \right) \right)$$

Como os vetores \hat{n} e \hat{l} se encontram normalizados, e tendo em conta que $\vec{u} \cdot \vec{v} = \|\vec{u}\| \|\vec{v}\| \cos(\angle(\vec{u}, \vec{v}))$ é possível simplificar a expressão acima do seguinte modo:

$$D_i = K_d \times \max \left(0, \hat{n} \cdot \hat{l} \right)$$

O contributo da i -ésima luz para a componente especular da cor do fragmento, S_i , é calculado

¹Apesar de, para o cálculo das equações da luz ser feito no espaço do mundo, ser necessária mais uma variável uniforme (a posição da câmara), achámos este método um pouco mais intuitivo e fácil de depurar do que fazer os mesmos cálculos no espaço da câmara.

de acordo com a seguinte expressão, onde K_s e s representam a componente especular e o fator *shininess* do material aplicado ao objeto, respectivamente:

$$S_i = K_s \times (\max(0, \hat{e} \cdot \hat{r}))^s$$

Na expressão acima, \hat{r} representa o raio da luz refletido, calculado pela função `reflect` do GLSL.

Em relação à diferença entre os vários tipos de luz, já se explicou que é necessário calcular a direção da luz para as *point lights* e *spotlights*. Em relação a estas luzes, não se implementou qualquer forma de atenuação, visto que o formato XML da cena não permite a sua definição e, por omissão, em OpenGL as luzes não sofrem qualquer atenuação [4]. Também pelo mesmo motivo, nas expressões apresentadas acima, a intensidade da luz não é considerada, visto que esta é a mesma para em todas as luzes, 1. No entanto, é necessário algum cuidado com as *spotlights*. Para o contributo de uma destas luzes não ser nulo, é necessário que o fragmento esteja dentro do cone de luz, ou seja, que o ângulo entre o raio de luz e o simétrico do vetor de direção da *spotlight* seja inferior ao *cutoff* θ [5]:

$$\begin{aligned} & \angle(\hat{l}, -\hat{d}) \leq \theta \\ \Leftrightarrow & \cos(\angle(\hat{l}, -\hat{d})) \leq \cos \theta \\ \Leftrightarrow & \hat{l} \cdot (-\hat{d}) \leq \cos \theta \end{aligned}$$

Na prática, o cosseno do ângulo de *cutoff* de uma *spotlight* é passado ao *shader* numa variável uniforme, e o *shader* faz a comparação acima para verificar se um fragmento é ou não iluminado por essa *spotlight*.

Estando calculadas as contribuições de todas as luzes, a cor de um fragmento pode ser determinada do seguinte modo, onde K_a e K_e representam a componentes ambiente e emissiva do material aplicado ao objeto, respectivamente [3] [6]:

$$K = K_a + K_e + \sum_i D_i + \sum_i S_i$$

Quando um objeto é desenhado com uma textura, a cor amostrada da textura é utilizada, nas expressões acima, como a componente difusa do material, e a componente ambiente é uma

fração da cor da textura (dependente do material).

Abaixo, segue-se um diagrama a explicar como estes *shaders* desenvolvidos se encaixam na *pipeline* de renderização:

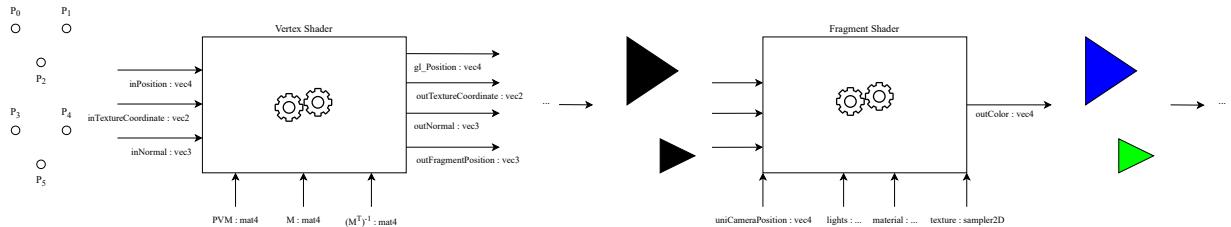


Figura 5: Esquema dos *shaders* desenvolvidos na *pipeline* de renderização.

2.5 Object Picking

Para implementar *object picking*, sempre que o utilizador clica na janela da `engine`, é criado um *framebuffer* com dois *attachments*, um para cor e outro para profundidade. Depois, a cena é desenhada para este *framebuffer*. A cada entidade na cena, é atribuído um identificador sequencial, que é utilizado para determinar a cor com que será desenhado no *framebuffer*: os 24 bits menos significativos do identificador são mapeados para as componentes R, G, e B da cor da entidade, oito *bits* de cada vez. Note-se que apenas são desenhadas as entidades da cena, e todas a uma cor sólida.

Após a cena ser desenhada para o *framebuffer*, lê-se, com a função `glReadPixels`, o *pixel* do *framebuffer* na posição do rato. Depois, transforma-se esta cor novamente num identificador, sendo assim possível identificar uma entidade com base num clique do utilizador.

No entanto, um identificador não é algo muito útil para se apresentar ao utilizador. Logo, o formato XML da cena foi modificado para permitir a atribuição de nomes a entidades, como mostra o exemplo abaixo:

```
<model name="Earth" file="sphere.3d">
  ...
</model>
```

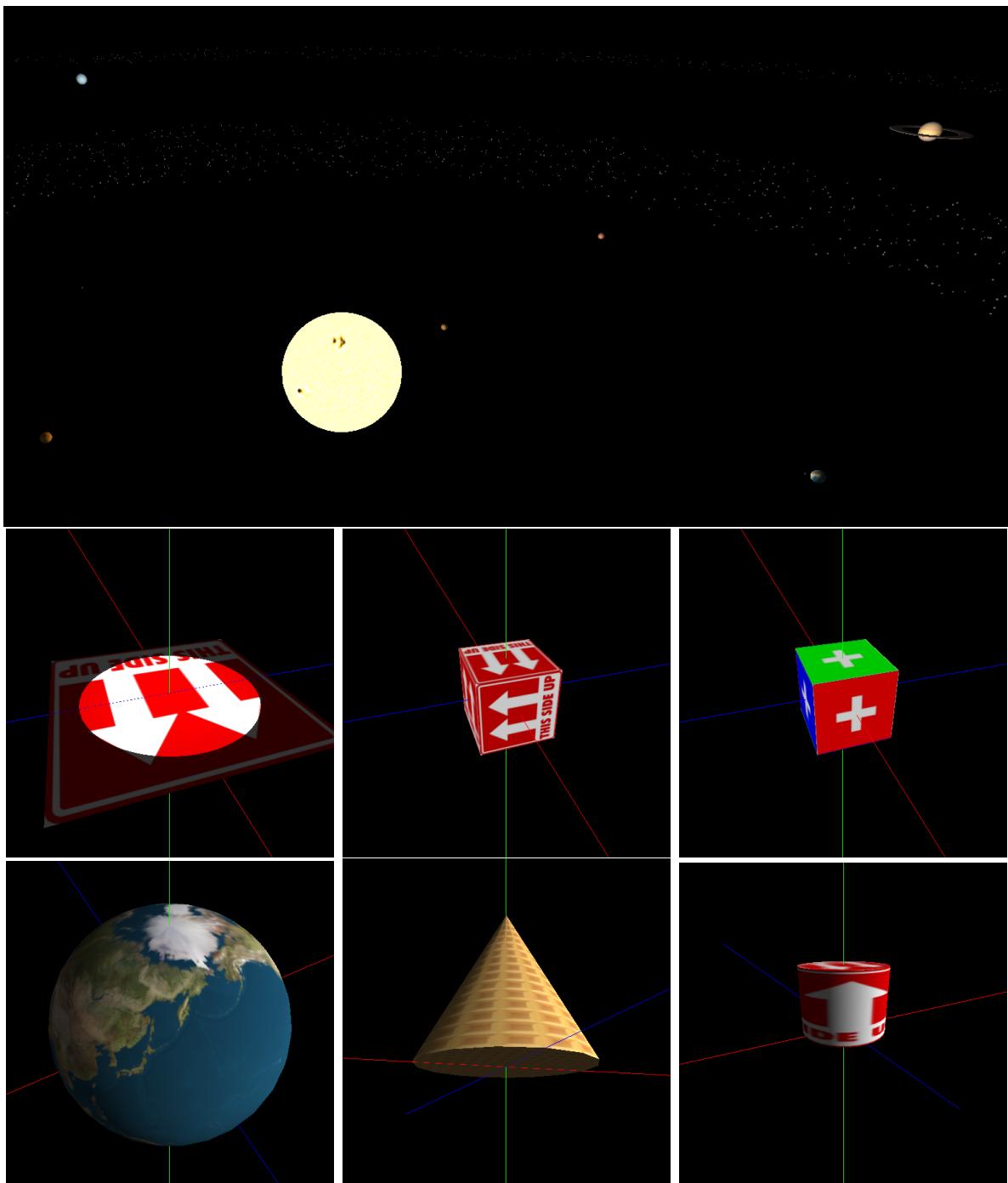
Quando a cena é desenhada para *object picking*, cria-se um dicionário que associa identificadores de entidades aos seus nomes. Logo, após se ter o identificador de um objeto, é possível aceder a este dicionário para consultar o seu nome e apresentá-lo na UI da `engine`.

3 Resultados Obtidos

Nesta secção, procuram-se apresentar algumas capturas de ecrã da engine das novas cenas criadas e das cenas fornecidas pela docência da UC.

3.1 Novas Cenas e Cenas Modificadas

Nesta fase, foram adicionadas iluminação e texturas a algumas cenas previamente criadas, e novas cenas também foram desenvolvidas:



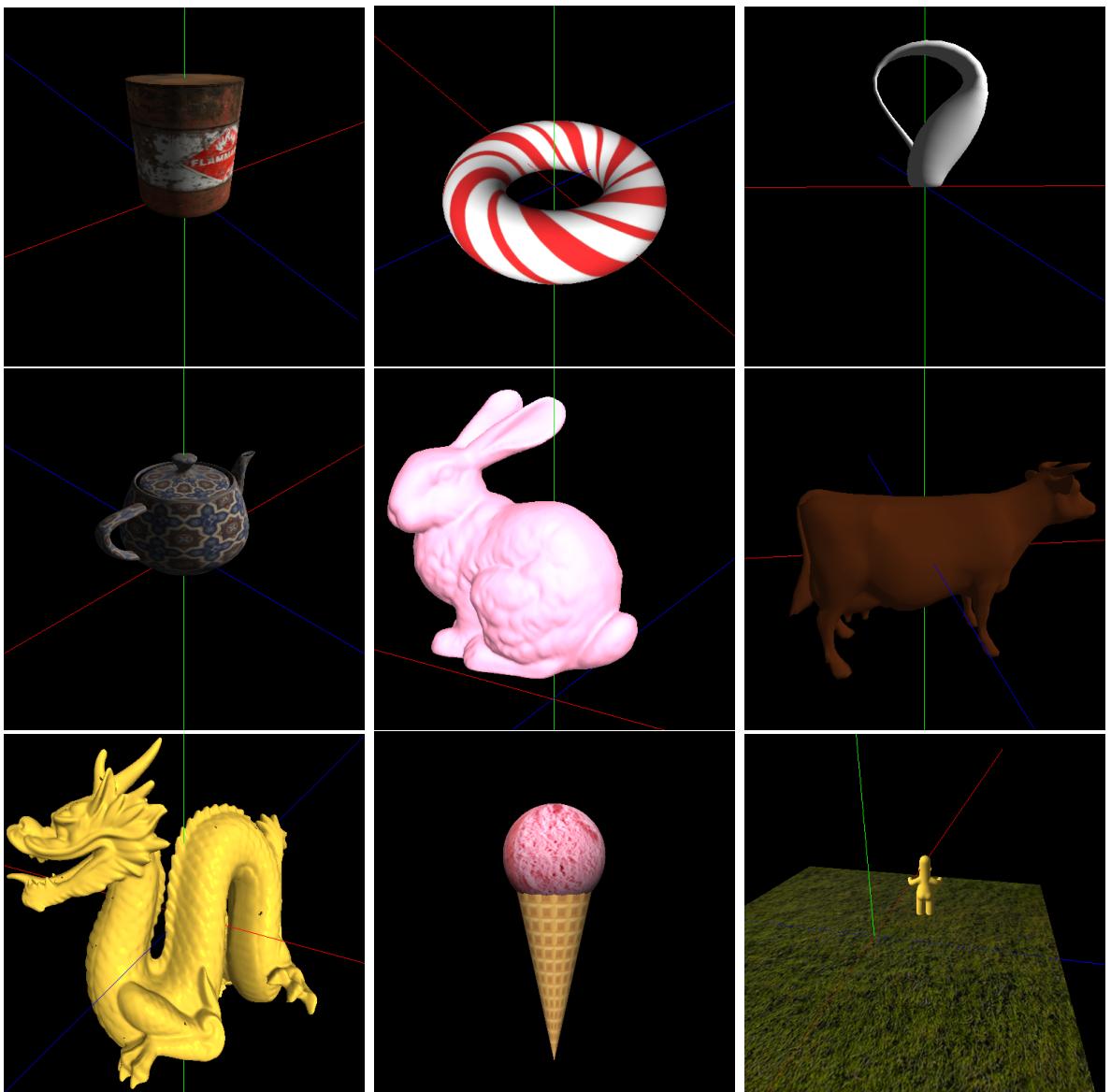


Figura 6: Renderização das cenas da autoria do grupo de trabalho.

3.2 Cenas Fornecidas pela Docência da UC

A docência da UC forneceu, juntamente com o enunciado do trabalho, algumas cenas a serem testadas no trabalho. A `engine` renderizou-as não exatamente como as capturas de ecrã fornecidas, visto que estas foram renderizadas com *shading* de Gouraud, mas a `engine` desenvolvida implementa *shading* de Phong:

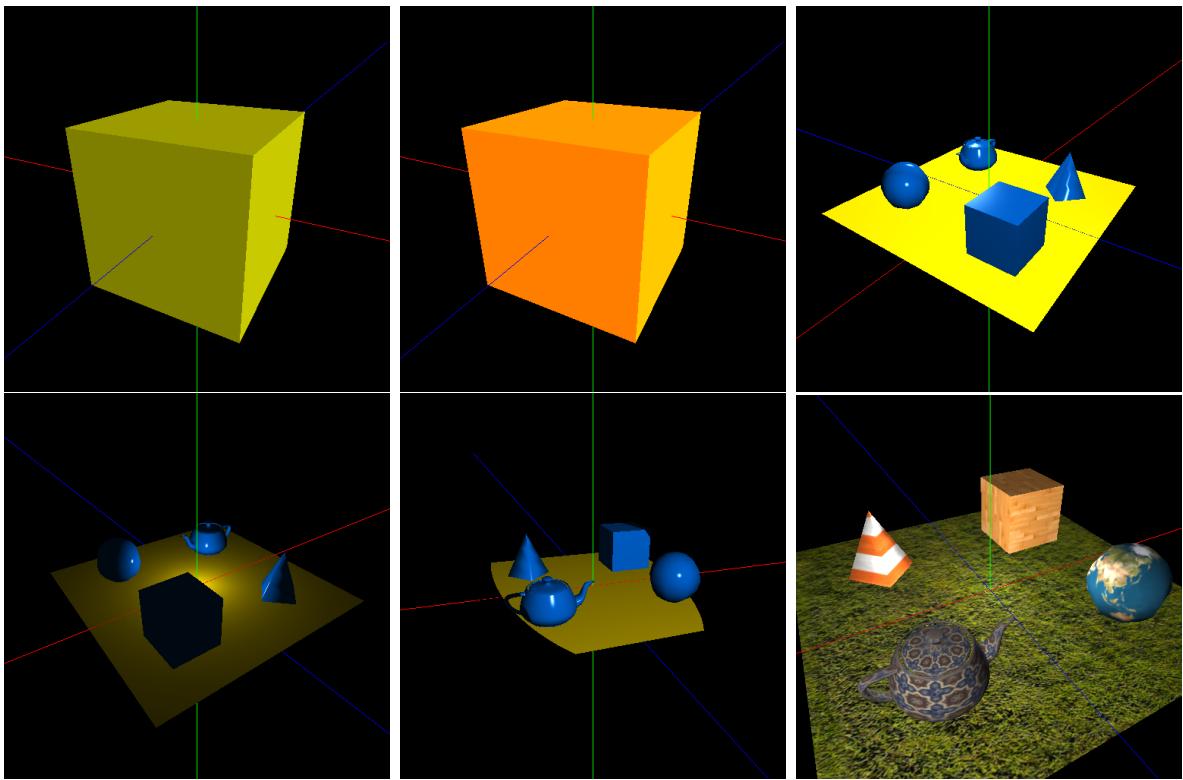


Figura 7: Renderização das cenas de teste fornecidas pela docência da UC.

A maior diferença entre os modelos de *shading* pode ser observada na cena com a *spotlight*, onde o corte entre a regiões iluminada e não iluminada é abrupto, em vez de suave. Note-se também que as manchas especulares no cone não têm a aparência correta, visto que o número de *slices* é muito baixo, resultando numa má aproximação de um cone. Caso se desejasse gerar uma pirâmide, as normais deveriam ser perpendiculares às faces e não às arestas.

4 Conclusão

Em suma, considera-se que a quarta fase do trabalho prático foi concluída com sucesso. Apesar desta fase ter sido a mais exigente, requerendo alterações a diversas partes da `engine` e do `generator`, o nosso grupo foi capaz de utilizar todo o conhecimento que tem vindo a adquirir ao longo do último semestre para implementar todas as funcionalidades pedidas, e ainda algumas adicionais! Foi também uma grande ajuda a reestruturação arquitetural do código feita na 3.^a fase, que tornou mais simples a adição de novas funcionalidades.

As maiores dificuldades nesta fase sentiram-se no `generator`, no que toca à adição de coordenadas de texturas e normais, tendo sido difícil garantir que todas as figuras tinhambem um aspecto correto, e descobrir a origem dos erros que se iam encontrando: coordenadas de texturas erradas *vs.* distorção natural inevitável, ou normais erradas *vs.* implementação incorreta da

iluminação.

Em relação às funcionalidades previstas na 3.^a fase, *object picking* foi implementado, mas não houve tempo para implementar *instanced rendering*. No entanto, para as cenas desenvolvidas, a falta desta funcionalidade não se provou um problema, visto que o elemento do grupo com a placa gráfica menos capaz (Intel HD Graphics 630), conseguia correr à taxa de atualização do seu ecrã (60Hz) a cena mais complexa, o Sistema Solar, que pode exigir milhares de *draw calls*.

Conclui-se este trabalho com grande satisfação em relação ao resultado final, que se considera cumprir os requisitos colocados pelo enunciado, bem como implementar muitas outras funcionalidades. No entanto, um possível ponto que poderia ser melhorado seria o subsistema de câmaras, que tem em falta aceleração e desaceleração suaves quando perante *input* do utilizador. Apesar desta ser a última fase do trabalho prático, há muitas funcionalidades que poderiam ser implementadas caso houvesse tempo para tal em hipotéticas futuras fases, desde aspectos simples como uma *skybox* e LODs, como outros mais complexos, apenas possíveis por se ter arquiteturado o projeto para usar *shaders*, como sombras, reflexões, *normals maps*, tesselação, *physically based rendering*, etc.

5 Bibliografia

- [1] “Wavefront OBJ File Format Summary.” FileFormat.Info. Accessed: May 14, 2025. [Online.] Available: <https://www.fileformat.info/format/wavefrontobj/egff.htm>
- [2] “stb.” GitHub. Accessed: May 13, 2025. [Online.] Available: <https://github.com/nothings/stb>
- [3] “Basic Lighting.” Learn OpenGL. Accessed: May 13, 2025. [Online.] Available: <https://learnopengl.com/Lighting/Basic-Lighting>
- [4] “glLight”. Khronos Registry. Accessed: May 13, 2025. [Online.] Available: <https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/glLight.xml>
- [5] “Light casters.” Learn OpenGL. Accessed: May 13, 2025. [Online.] Available: <https://learnopengl.com/Lighting/Light-casters>
- [6] “Multiple Lights.” Learn OpenGL. Accessed: May 13, 2025. [Online.] Available: <https://learnopengl.com/Lighting/Multiple-lights>