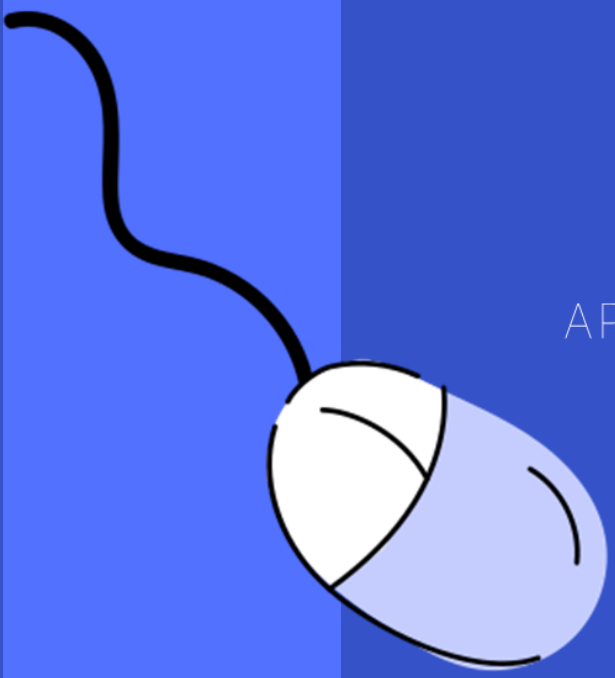
RELATÓRIO

WebGL

Trabalho 2



GRUPO 00

APOLO GUERRA LEIRIA Nº52001

BRUNA DUARTE Nº51658

TIAGO CASTRO Nº53494

PROF. ANA PAULA CLÁUDIO

PROF. NUNO GARCIA

COMPUTAÇÃO GRÁFICA

2020/2021

Ciências ULisboa

Índice

Figuras

Introdução

Este trabalho teve como objetivo a utilização de uma biblioteca *WebGL,* de modo a efetuar alterações numa cena tridimensional, adicionando objetos, animações específicas e variações sua na luminosidade. No desenvolvimento deste projeto tivemos a oportunidade de implementar na cena através objetos criados a partir de *WebGL*, assim como objetos criados e exportados do Blender.

Web Graphics Library, ou *WebGL*, é uma interface de programação de aplicações (API) em JavaScript, que oferece suporte para a renderização de gráficos a duas ou três dimensões. Pode ser implementado em aplicações web sem ser necessário correr a plug-ins do browser.

Tendo como base os ficheiros HTML e Java Script (JS) indicados e fornecidos pelos docentes, fomos alterando e acrescentando elementos a este código para atingir os resultados pretendidos. De forma sucinta podemos dizer que procuramos desenvolver as capacidades necessárias para a obtenção de imagens realistas. Este realismo é atingido com a combinação de luzes na cena, o *shading* que se evidencia nos objetos, a interação da reflexão e difusão da luz entre os objetos, assim como no movimento dos objetos e posição do observador. Com a combinação destes elementos consegue-se produzir imagens realistas. Assim, e seguindo a estrutura do enunciado do projeto, pudemos implementar objetos na cena e visualizá-los de diferentes formas (questão 1), aplicar diferentes efeitos de *shading* e verificar a influência das diferentes componentes da luz (questão 2), explorar os movimentos nos objetos e na posição do observador (questão 3), assim como implementar e combinar objetos elaborados pelo *WebGL* e *Blender* (questão 4).

Após resolução de cada um dos exercícios, estes foram compilados em pastas diferentes por cada uma das 4 questões, estando identificados pelo número da questão e pelo número da alínea. Cada resolução de exercício contempla o ficheiro HTML e respetivo ficheiro JS.

Na descrição dos procedimentos executados para a resolução de cada questão, a metodologia utilizada passou por explicar a sequência de passos, com a identificação das linhas de código alteradas e efeitos pretendidos. No final de cada uma das questões também são exemplificados resultados com algumas imagens.

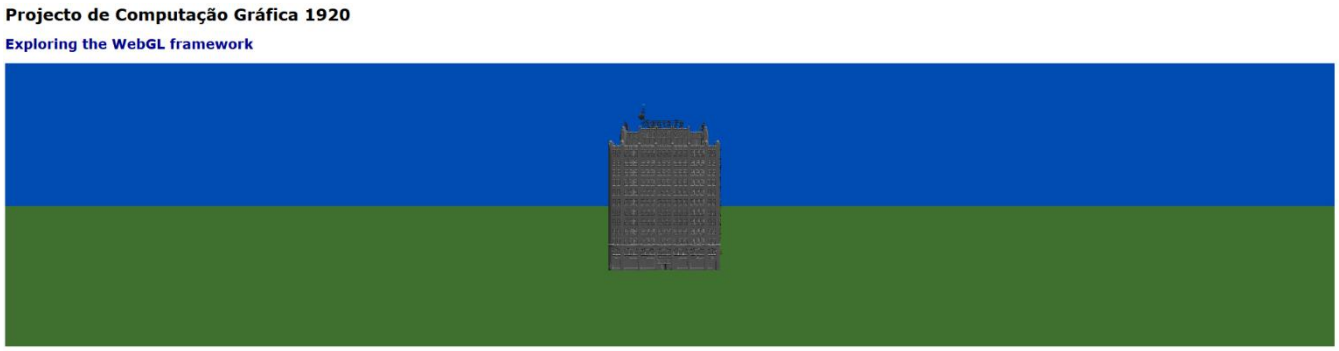


Figura 1 - Cena 3D original

Questão 1: Modificações na cena original

1.1 Adicionar 3 novos objetos à cena (pirâmide amarela, cubo verde, esfera azul)

Para a implementação destes objetos, o grupo desenvolveu o código com base no *Sample 5*, que posteriormente foi alterado para ficar semelhante ao *Sample 7*, nomeadamente no que diz respeito ao *shading*. Colocamos o cubo verde à esquerda, pirâmide amarela ao centro e a esfera azul à direita, conforme se observa na figura seguinte.

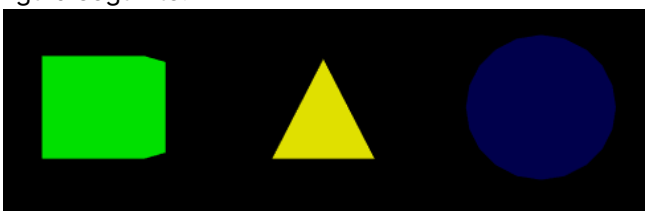


Figura 2 – Cubo, pirâmide e esfera.

Na implementação do código para representar vários objetos simultaneamente, a opção do grupo foi adaptar a função *drawScene*, e incluir uma nova função *draw*, que de forma iterativa recebia os *buffers* de cada um dos objetos e restantes parâmetros necessários à sua representação (cf. números 3 e 4 da figura 3).

Analisando a figura 3 do topo para a base, veremos que foram criadas funções que inicializavam o *buffer* de cada um dos objetos, sendo estes incluídos numa lista que seria enviada para a função *draw*. Esta função draw para além de receber os vários buffers, também implementa uma lista de matrizes de composição de transformações geométricas para cada um deles (m1, m2, m3). Depois faz uma iteração com base na dimensão da lista de *buffers* e ao longo desta, implementando cada um dos objetos solicitados.

O código para a implementação de cada um dos buffers está nas linhas 175, 334 e 503 do ficheiro "Ex_1.1.js". As funções que iniciam o buffer da pirâmide e do cubo são semelhantes, mudando apenas o número de vértices, faces e cores implementadas. Já para a implementação da esfera, o código implementado teve como base a informação presente no link indicado pelo docente. A implementação das cores no cubo, pirâmide e esfera está respetivamente nas linhas de código 445 a 467, 277 a 298, e 615 a 631.

Figura 3 – Implementação dos buffers dos objetos e função draw.

1.2 Modificar a posição da fonte de luz (iluminar a parte do cubo de frente para a câmara)

Para alterar a fonte de luz de forma a esta ficar de frente para o cubo, foi necessário alterar os valores determinados no *Vertex Shader* para o vetor da direção da fonte de luz (figura 4). Para tal consideramos os valores da matriz de translação que é aplicada ao cubo no momento da sua materialização, neste caso uma translação de -4.5 no eixo X.

```
26  const vsSource = `
27    attribute vec4 aVertexPosition;
28    attribute vec3 aVertexNormal;
29    attribute vec4 aVertexColor;
30
31    uniform mat4 uNormalMatrix;
32    uniform mat4 uModelViewMatrix;
33    uniform mat4 uProjectionMatrix;
34
35    varying lowp vec4 vColor;
36    varying highp vec3 vLighting;
37
38    void main(void) {
39      gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
40      vColor = aVertexColor;
41
42      highp vec3 ambientLight = 0.3 * vec3(1, 1, 1);        //INTENSIDADE*COR DA LUZ AMBIENTE
43      highp vec3 directionalLightColor = vec3(1, 1, 1);      //COR DA FONTE DE LUZ
44      highp vec3 directionalVector = normalize(vec3(-4.5, 0.0, 5));     //DIREÇÃO DA FONTE DE LUZ
45
46      highp vec4 transformedNormal = uNormalMatrix * vec4(aVertexNormal, 1.0);
47
48      highp float directional = max(dot(transformedNormal.xyz, directionalVector), 0.0);    //COS(TETA)
49      vLighting = ambientLight + (directionalLightColor * directional);                      //LUZ AMBIENTE + DIFUSA
50    }
51  `;
```

```
133
134       let ml = mat4.create();
135       mat4.translate(ml, ml,[-4.5, 0.0, -16.0]);
```

Figura 4 – Alteração dos parâmetros da direção da fonte de luz em função da posição do cubo.

Nas figuras seguintes observamos respetivamente a comparação do efeito da iluminação com esta alteração, face à iluminação inicial com vetor de direção igual a (5, 5, 5). Facilmente se depreende que na figura 5 temos a face frontal do cubo totalmente iluminada, e que a sua face lateral direita se encontra com alguma sombra. Já na figura 6 vemos que com a direção da fonte de luz diferente, tanto a face frontal como a lateral do cubo já se encontram totalmente iluminadas.

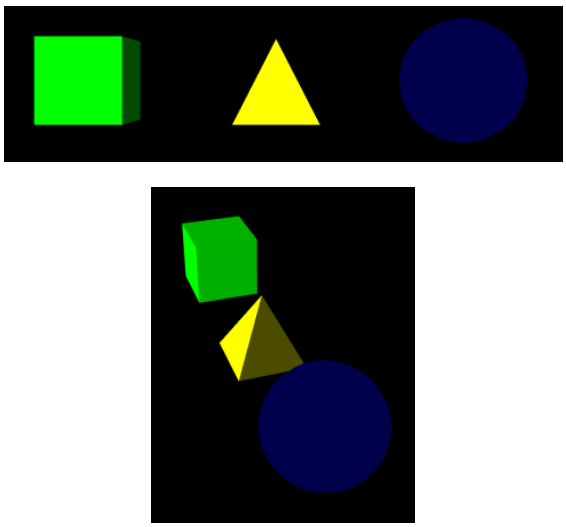


Figura 5 – Efeito de iluminação centrado no cubo.

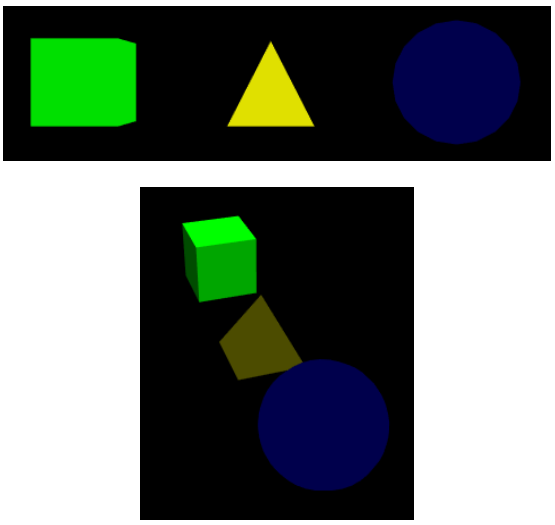


Figura 6 – Efeito de iluminação com orientação (5, 5, 5).

1.3 Modificar a posição da câmara visão lateral e visão de topo da cena

Para a modificação da câmara foi necessário alterar a *modelViewMatriz*. Estas alterações encontram-se nas linhas de código número 751 e 755 respetivamente. Para a alteração da câmara de forma a obter uma vista lateral da cena utilizamos a função *lookAt* centrada na coordenada (0, 0, 0). Para uma melhor visualização optamos por ter uma vista lateral superior e que nos permitisse ver todos os objetos, daí a posição do observador escolhida ter sido (15, 10, 5). Se o objetivo fosse ter uma vista lateral e

perpendicular à disposição dos objetos teríamos de selecionar a posição do observador com a coordenada de Y e Z igual a zero, como por exemplo (15, 0, 0). Para além desta função, foi necessário introduzir uma translação com o mesmo valor que o determinado no início da função *draw* para cada objeto (base_location). Nas figuras 8 e 9 seguinte podemos observar a implementação deste código e a sua sequência, assim como os resultados.

```
722  function drawScene(gl, programInfo, buffers, deltaTime, matriz, n_points, base_location) {
723
724
725      // Create a perspective matrix, a special matrix that is
726      // used to simulate the distortion of perspective in a camera.
727      // Our field of view is 45 degrees, with a width/height
728      // ratio that matches the display size of the canvas
729      // and we only want to see objects between 0.1 units
730      // and 100 units away from the camera.
731
732      const fieldOfView = 45 * Math.PI / 180;   // in radians
733      const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
734      const zNear = 0.1;
735      const zFar = 100.0;
736      const projectionMatrix = mat4.create();
737
738      // note: glmatrix.js always has the first argument
739      // as the destination to receive the result.
740      mat4.perspective(projectionMatrix,
741                       fieldOfView,
742                       aspect,
743                       zNear,
744                       zFar);
745
746      // Set the drawing position to the "identity" point, which is
747      // the center of the scene.
748      let modelViewMatrix = matriz;
749
750      // QUESTÃO 1.3 --> Visão lateral da cena
751      mat4.lookAt(modelViewMatrix, [15, 10, 5], [0, 0, 0], [0.0, 1.0, 0.0]);
752      mat4.translate(modelViewMatrix,modelViewMatrix,base_location);
753
754      // QUESTÃO 1.4 --> Visão de topo da cena
755      // mat4.rotate(modelViewMatrix, modelViewMatrix, 0.6, [1, 0, 0]);
756      // mat4.translate(modelViewMatrix, modelViewMatrix,[0.0, -3.0, 0.0]);
```

Figura 7 - Código para a vista lateral.

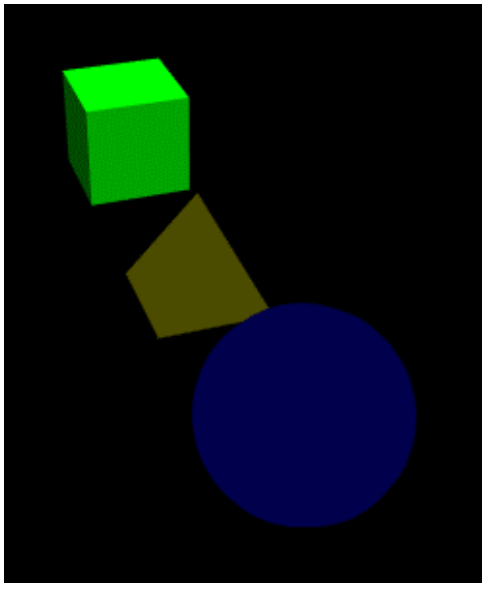


Figura 8 - Vista lateral.

Para a implementação da vista de topo, o grupo optou por fazer uma translação para cima (eixo Y em 3 unidades) e posteriormente aplicar uma rotação em torno do eixo X para "apontar" a câmara para baixo (figura 10). Caso fosse necessário ter uma vista de topo verticalmente centrada na parte superior aos objetos, além de alterar o ângulo de rotação em torno do eixo do X de forma a aumentar a inclinação essa direção se tornar paralela ao eixo Y, também teríamos de compensar a translação inicial que os nossos objetos sofreram na função *draw*. Na figura 11 está representada a imagem de topo obtida.

```
746     // Set the drawing position to the "identity" point, which is
747     // the center of the scene.
748     let modelViewMatrix = matriz;
749
750     // QUESTÃO 1.3 --> Visão lateral da cena
751     // mat4.lookAt(modelViewMatrix, [15, 10, 5], [0, 0, 0], [0.0, 1.0, 0.0]);
752     // mat4.translate(modelViewMatrix,modelViewMatrix,base_location);
753
754     // QUESTÃO 1.4 --> Visão de topo da cena
755     mat4.rotate(modelViewMatrix, modelViewMatrix, 0.6, [1, 0, 0]);
756     mat4.translate(modelViewMatrix, modelViewMatrix,[0.0, -3.0, 0.0]);
```

Figura 9 - Código para a vista de topo.

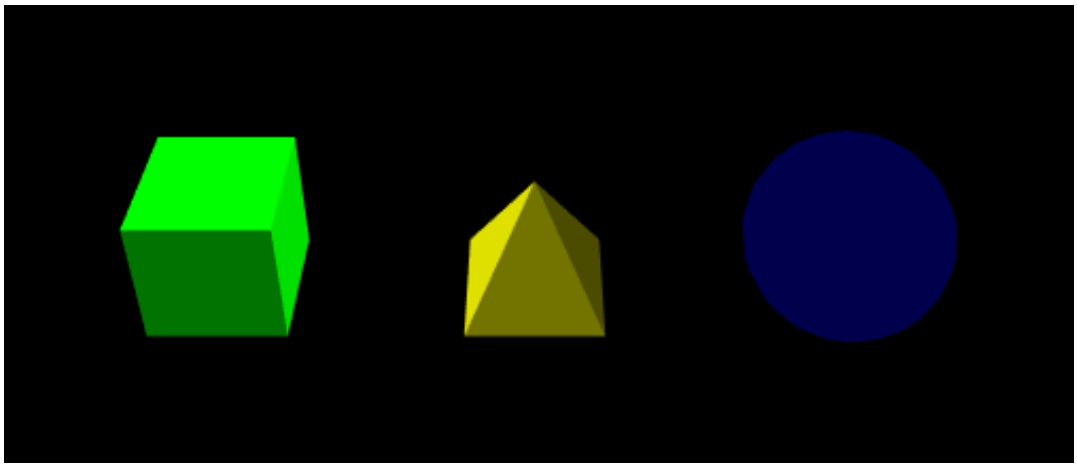


Figura 10 - Vista de topo.

Questão 2: Iluminação

Em Computação Gráfica, a iluminação é definida através de um modelo de computação da iluminação da cena. Neste modelo, são consideradas as propriedades das fontes de luz (direção, intensidade, posição, cor e forma), as propriedades dos materiais (reflexão, absorção, e transmissão da luz incidida sobre os materiais), a interação entre diferentes fontes de luz, e a interação entre os materiais iluminados.

Existem dois tipos de iluminação: local, e global. Num modelo de iluminação local, apenas são consideradas as interações individuais entre a fonte de luz e o objeto, e a posição do observador. Num modelo de iluminação global, para além das interações mencionadas anteriormente, são também consideradas as interações de luz entre os objetos.

A um nível mais abstrato, a equação de um modelo de iluminação local pode ser vista como a soma de três parcelas (componente ambiente, difusa e especular):

$$I = I_{ambiente} + f_{at}\,(I_{difusa} + I_{especular}) \quad \textbf{(1)}$$

Onde $I_{ambiente}$ é relativo à iluminação ambiente, I_{difusa} depende do ângulo θ e não da posição do observador, e $I_{especular}$, que depende da posição do observador (ângulos θ e α)

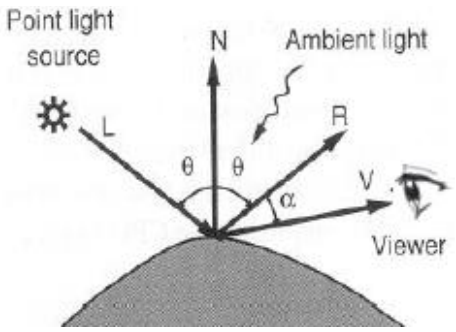


Figura 11 - Modelo de iluminação local (Esquema)

Para este trabalho, apenas foram implementados modelos de iluminação local: Gouraud Shading e Phong Shading.

2.1 Implementação do método de Phong Shading (componentes luz ambiente e difusa) e comparação com método de Gouraud

Na criação de um cenário, é necessário ter em atenção os pontos de luz presentes, sendo que estes definem as cores que observaremos. A luz, ao incidir nos objetos, pode sofrer reflexão – especular ou difusa –, transmissão ou absorção. O modo como percecionamos as cores nos objetos dependem da intensidade da luz que incide em cada ponto, da posição e orientação de um objeto, e das suas caraterísticas – isto define o "shading".

Equacionando esta questão, através do modelo de iluminação de Phong, temos (2):

$$I = I_a K_a + I_p f_{at}[K_d(L \cdot N) + K_s(R \cdot V)^{n}] \quad \textbf{(2)}$$

onde a intensidade da cor incidente (I) depende da intensidade da luz ambiente (I_a), da intensidade de uma fonte de luz (I_p), da intensidade da luz refletida por difusão e por especular, da distância do foco de luz ao objeto (f_{at}) iluminado, a direção dos raios de incidência por reflexão (R) e a direção de visualização (V) – este produto interno entre os vetores R e V corresponde ao cosseno de θ. Neste modelo, a cor para cada fragmento é calculada no "Fragment Shader"

No shading Gouraud (método de interpolação de intensidades), para cada um dos vértices do polígono é considerado como normal à superfície, a média das normais dos polígonos que partilham o vértice. A

intensidade de cada vértice é obtida a partir da normal calculada, sendo a cor de um fragmento calculada no "Vertex Shader". É efetuada uma interpolação linear das cores sobre as faces.

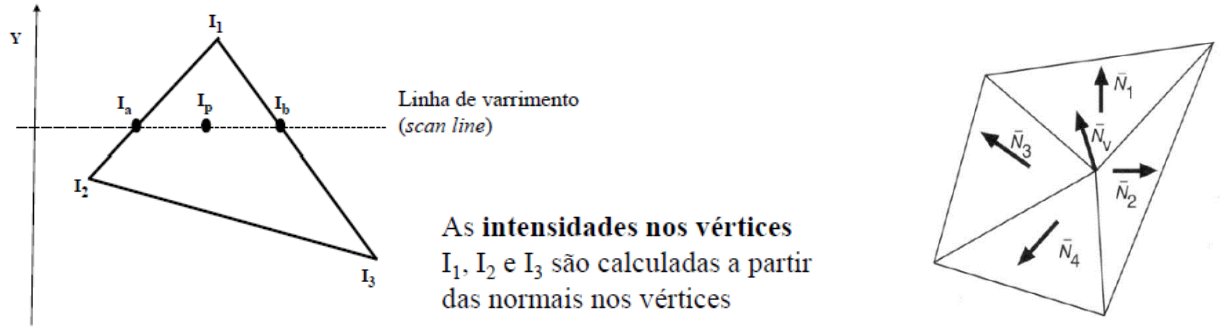


Figura 12 - Método de Interpolação de Intensidades (Gouraud Shading)

De modo a obter o Shading de Phong, foram efetuadas algumas alterações no código. Começando por definir as variáveis globais com os valores de atributos a utilizar no Shader, e alterar o código do Shader para corresponder ao Shading de Phong, tal como explicado anteriormente neste capítulo. Pesquisámos os valores dos atributos dos shaders, como as constantes para as cores, e atribuímos esses valores guardados às variáveis globais.

Estas operações podem ser observadas nas figuras que se seguem:

```
var shininess = 80.0; // Expoente Reflexão Especular - Alínea 2.4: Alteração deste valor
var kaVal = 1.0; // Coeficiente Reflexão da Luz Ambiente
var kdVal = 1.0; // Coeficiente Reflexão Difusa
var ksVal = 1.0; // Coef. Reflexão Especular - Alínea 2.4: Alteração deste valor

var lightPos = [3.0, 2.0, -1.0]; // Posição da luz

var ambientColor = [0.2, 0.1, 0.0];
var diffuseColor = [0.2, 1.0, 0.0];
var specularColor = [1.0, 1.0, 1.0];
```

Figura 13 - Shading de Phong: Valores dos coeficientes utilizados e cores de cada componente

```
    // Vertex shader program

    const vsSource = `
    attribute vec4 position;
    attribute vec3 normal;
    attribute vec2 texCoord;
    uniform mat4 projection, modelview, normalMat;
    varying vec3 normalInterp;
    varying vec3 vertPos;

    void main(){
      vec4 vertPos4 = modelview * position;
      vertPos = vec3(vertPos4) / vertPos4.w;
      normalInterp = vec3(normalMat * vec4(normal, 0.0));
      gl_Position = projection * vertPos4;
    }
  `;
```

Figura 14 - Shading de Phong: Vertex Shader

```
const fsSource = `
precision mediump float;
varying vec3 normalInterp;  // Surface normal
varying vec3 vertPos;       // Vertex position

uniform float Ka;   // Coeficiente de reflexão ambiente
uniform float Kd;   // Coeficiente de reflexão difusa
uniform float Ks;   // Coeficiente de reflexão especular
uniform float shininessVal; // Brilho

// Material color
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform vec3 lightPos; // Light position

void main() {
  vec3 N = normalize(normalInterp);
  vec3 L = normalize(lightPos - vertPos);

  // Lambert's cosine law
  float lambertian = max(dot(N, L), 0.0);
  float specular = 0.0;
  if(lambertian > 0.0) {
    vec3 R = reflect(-L, N);      // Reflected light vector
    vec3 V = normalize(-vertPos); // Vector to viewer
    // Compute the specular term
    float specAngle = max(dot(R, V), 0.0);
    specular = pow(specAngle, shininessVal);
  }
   gl_FragColor = vec4(Ka * ambientColor +
                       Kd * lambertian * diffuseColor
                       // + Ks * specular * specularColor // Alínea 2.2
                      , 1.0);

    // Para obter apenas a Componente Ambiente:
    // gl_FragColor = vec4(Ka * ambientColor, 1.0);

    // Apenas Componente Difusa:
    // gl_FragColor = vec4(Kd * lambertian * diffuseColor, 1.0);

    // Apenas Componente Especular:
    // gl_FragColor = vec4(Ks * specular * specularColor, 1.0);

}
`;
```

Figura 15 - Shading de Phong: Fragment Shader

```
// Atribuição de valores às variáveis definidas globalmente
// Setup Phong coeficients
gl.uniform1f(programInfo.uniformLocations.ka, kaVal);
gl.uniform1f(programInfo.uniformLocations.ks, ksVal);
gl.uniform1f(programInfo.uniformLocations.kd, kdVal);
gl.uniform1f(programInfo.uniformLocations.shine, shininess);

// Setup colors
gl.uniform3fv(programInfo.uniformLocations.ambientColor, ambientColor);
gl.uniform3fv(programInfo.uniformLocations.diffuseColor, diffuseColor);
gl.uniform3fv(programInfo.uniformLocations.specularColor, specularColor);

gl.uniform3fv(programInfo.uniformLocations.lightPos, lightPos)
```

Figura 16 - Shading de Phong: Atribuição de valores às variáveis definidas globalmente

Para obtermos o Shading Gouraud, foram efetuadas alterações semelhantes no código, exceto na parte dos Shaders, onde o foco principal deste Shader será o Vertex Shader, como podemos observar no código que se segue.

```
// Vertex shader program
// Implementação do Shading de Gouraud

const vsSource = `
attribute vec3 position;
attribute vec3 normal;
uniform mat4 projection, modelview, normalMat;
varying vec3 normalInterp;
varying vec3 vertPos;
uniform int mode;   // Rendering mode
uniform float Ka;   // Coeficiente de reflexão ambiente
uniform float Kd;   // Coeficiente de reflexão difusa
uniform float Ks;   // Coeficiente de reflexão especular
uniform float shininessVal; // Brilho
// Material color
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform vec3 lightPos; // Light position
varying vec4 color; //color

void main(){
  vec4 vertPos4 = modelview * vec4(position, 1.0);
  vertPos = vec3(vertPos4) / vertPos4.w;
  normalInterp = vec3(normalMat * vec4(normal, 0.0));
  gl_Position = projection * vertPos4;

  vec3 N = normalize(normalInterp);
  vec3 L = normalize(lightPos - vertPos);
  // Lambert's cosine law
  float lambertian = max(dot(N, L), 0.0);
  float specular = 0.0;
  if(lambertian > 0.0) {
    vec3 R = reflect(-L, N);      // Reflected light vector
    vec3 V = normalize(-vertPos); // Vector to viewer
    // Compute the specular term
    float specAngle = max(dot(R, V), 0.0);
    specular = pow(specAngle, shininessVal);
  }
  color = vec4(Ka * ambientColor +
               Kd * lambertian * diffuseColor
                // + Ks * specular * specularColor // Alínea 2.2
               , 1.0);
}
`;
```

```
// Fragment shader program

const fsSource = `
precision mediump float;

varying vec4 color;
void main() {
  gl_FragColor = color;
}
`;
```

Figura 17 - Shading Gouraud: Vertex Shader e Fragment Shader

Posto isto, comparando lado-a-lado as figuras obtidas através dos processos anteriores, temos o seguinte.

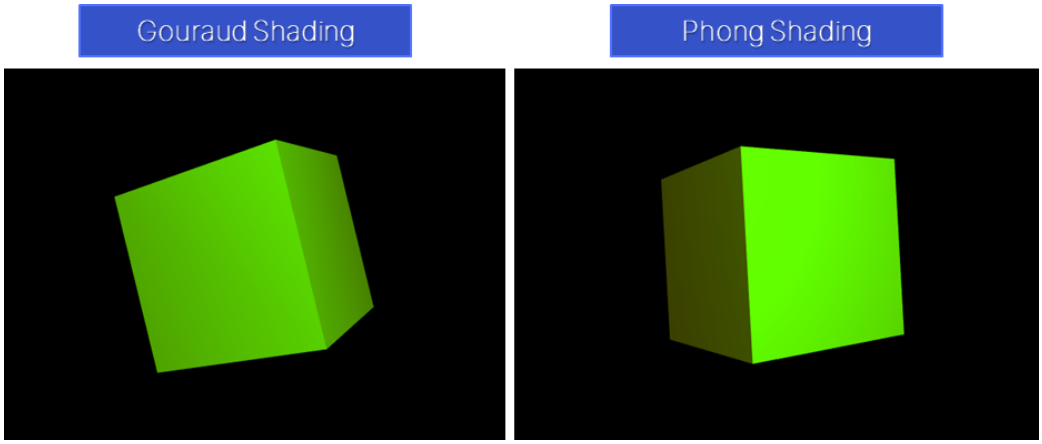


Figura 18 - Shading Gouraud e Shading de Phong (Componentes Ambiente e Direcional)

Nesta comparação, é visível a diferença entre os dois métodos de Shading aplicados no cubo. No primeiro, é visível a perda de "highlights" que deveriam estar presentes, sendo que este algoritmo apenas interpola cores a partir das cores dos vértices. Estes "highlights" são visíveis no segundo método, dado que este algoritmo calcula de modo explicito a cor em cada ponto da imagem.

Podemos concluir que o método de Phong é mais realista, elaborado, e melhorado em comparação com o método de Gouraud.

2.2 Implementação da componente de luz especular

A reflexão especular rege-se pela lei de reflexão: ângulo de incidência = ângulo de reflexão. Esta componente depende da posição do observador. No cálculo da intensidade da luz refletida, temos a seguinte fórmula aproximada:

$$I_S = I_P k_s [\cos(\alpha)]^n \quad \textbf{(3)}$$

Tal que I_P é a intensidade da fonte de luz, k_s é o coeficiente de reflexão especular (varia entre 0 e 1), α é o ângulo entre a direção de reflexão e a de visualização, e n é o expoente de reflexão especular (varia entre 1 e infinito).

Esta componente encontra-se implementada na alínea anterior, sob a forma de comentário no código de ambos. Nestas figuras, o coeficiente de reflexão especular é igual a 1.

```
71      // Lambert's cosine law
72      float lambertian = max(dot(N, L), 0.0);
73      float specular = 0.0;
74      if(lambertian > 0.0) {
75        vec3 R = reflect(-L, N);      // Reflected light vector
76        vec3 V = normalize(-vertPos); // Vector to viewer
77        // Compute the specular term
78        float specAngle = max(dot(R, V), 0.0);
79        specular = pow(specAngle, shininessVal);
80      }
81       gl_FragColor = vec4(Ka * ambientColor +
82                           Kd * lambertian * diffuseColor
83                           + Ks * specular * specularColor // Alínea 2.2
84                           , 1.0);
```

Figura 19 - Componente de luz especular

Para este ponto, foi utilizado um coeficiente de luz especular igual a 1. A luz especular é visível se como um ponto brilhante que surge nos objetos polidos quando iluminados. Os realces especulares são importantes na área da computação gráfica, dado fornecerem uma forte indicação visual para a forma de um objeto e a sua localização em relação às fontes de luz presentes na cena.

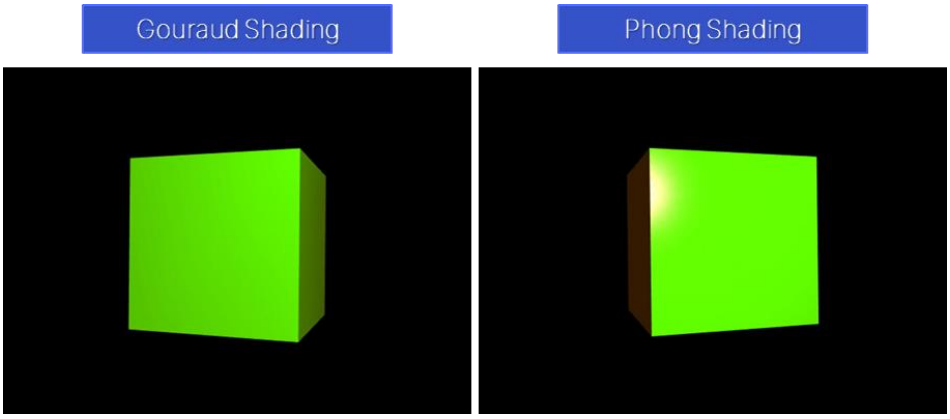


Figura 20 - Shading Gouraud e Shading de Phong (Componentes Ambiente, Direcional e Especular)

Com a implementação da luz especular, aumenta a qualidade da cena – o realismo acrescentado com a presença desta componente advém da perceção que temos da qualidade da superfície exposta à luz. O observador, inconscientemente, ganha perceção sobre a textura (mais ou menos rugosa) da superfície do objeto – possibilitando, deste modo, a representação de superfícies espelhadas. Retirando a componente especular, torna-se difícil a implementação deste efeito.

2.3 Visualização das diferentes componentes

Na alínea anterior obtivemos um cubo composto por 3 elementos distintos: a componente especular, difusa e ambiente. Estas componentes podem ser facilmente observadas no esquema que se segue.

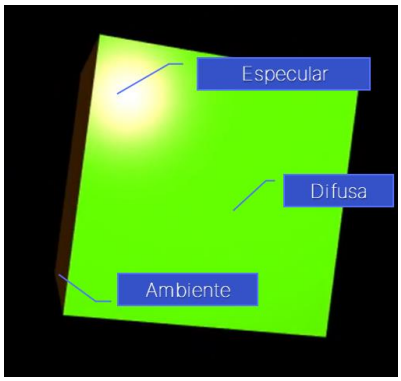


Figura 21 - Composição do cubo

De modo a obtermos as componentes anteriores separadamente, recorremos ao seguinte código, substituindo a função gl_FragColor, por cada uma das linhas desejadas, consoante o efeito pretendido.

```
81        gl_FragColor = vec4(Ka * ambientColor +
82                            Kd * lambertian * diffuseColor
83                            // + Ks * specular * specularColor // Alínea 2.2
84                           , 1.0);
85
86        // Para obter apenas a Componente Ambiente:
87        // gl_FragColor = vec4(Ka * ambientColor, 1.0);
88
89        // Apenas Componente Difusa:
90        // gl_FragColor = vec4(Kd * lambertian * diffuseColor, 1.0);
91
92        // Apenas Componente Especular:
93        // gl_FragColor = vec4(Ks * specular * specularColor, 1.0);
94
95    }
96  `;
```

Figura 22 - Obtenção das diferentes componentes

Os resultados obtidos em cada linha de código encontram-se apresentados na figura que se segue, onde podemos distinguir cada uma das componentes de modo isolado.

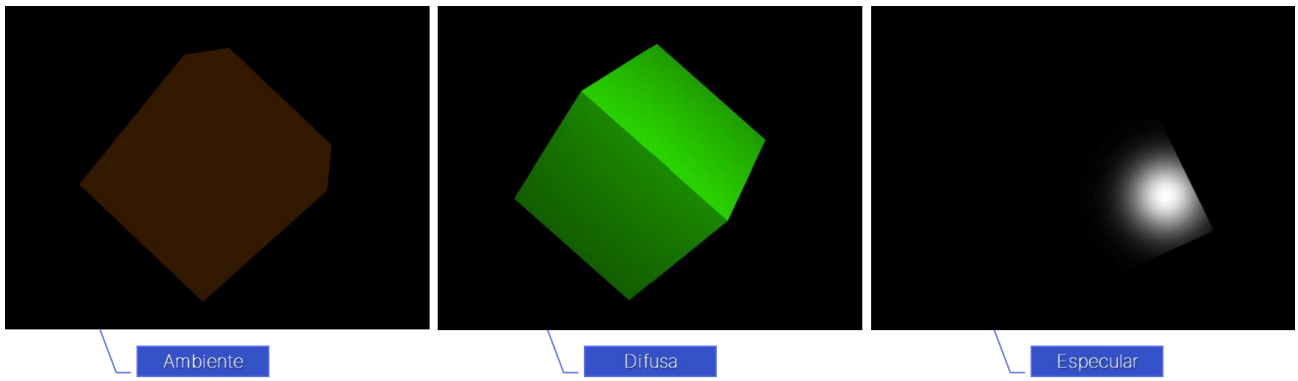


Figura 23 - Várias componentes (Ambiente, Difusa e Especular)

Da figura 23 podemos observa que a luz ambiente é aplicada de forma constante em todas as faces do objeto, independentemente a sua orientação. Este facto advém do princípio teórico de que a luz ambiente é uma luz não direcional, e que corresponde à combinação de todas as interações luminosas entre faces e objetos na cena, e, portanto, considera-se que vem de todas as direções.

Já a luz difusa tem em consideração a orientação de cada uma das faces do objeto. Dentro de cada uma das faces também é possível observar diferentes intensidades: a interpolação das cores. Quando referimos que depende da orientação da face estamos a dizer que é independente da posição do observador, uma vez que apenas considera a direção de incidência a normal à superfície dos objetos.

Quanto à luz especular, verificamos que na imagem à direita na figura 23 temos uma zona que corresponde à reflexão dos raios luminosos provenientes da fonte de luz. Este efeito de reflexão corresponde ao aumento da suavidade e capacidade de reflexão das superfícies dos objetos, ou seja, a suavidade de uma superfície é diretamente proporcional à dimensão da reflexão da luz (especular).

2.4 Variação dos valores dos coeficientes de especularidade

```
6    var ksVal = 1.0; // Alínea 2.4: Alteração deste valor
```

Figura 24 - Variável: Valor do coeficiente de reflexão especular

Como mencionado na alínea 2.2, inicialmente utilizamos um valor para o coeficiente de reflexão especular, que varia entre 0 e 1, igual a 1. Variando o valor deste coeficiente, obtemos os seguintes resultados:

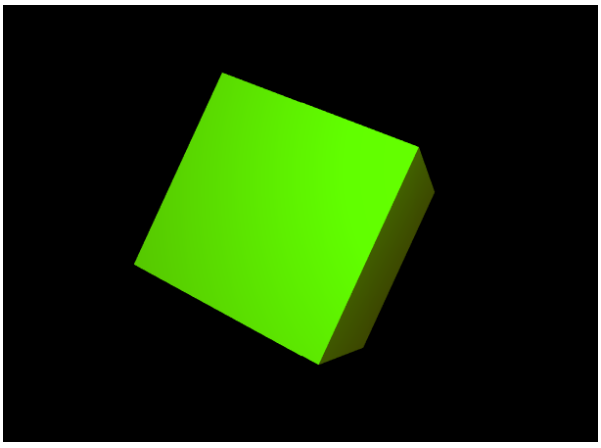


Figura 25 - KsVal = 0.2, Shininess = 80

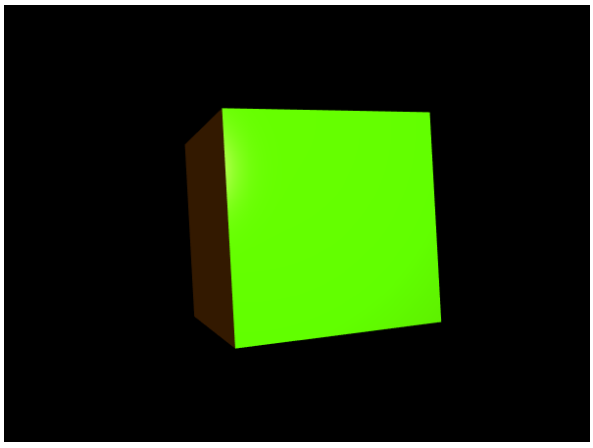


Figura 26 - KsVal = 0.8, Shininess = 80

É possível observar, no contraste da figura 25 com a figura 26, que quanto maior for o valor do coeficiente de reflexão especular, maior será a refletividade da superfície do cubo – sendo mais visível o "highlight" do ponto de luz especular presente na superfície do sólido, consoante o aumento deste coeficiente.

```
3    var shininess = 80.0;
```

Figura 27 Variável: Valor do expoente de reflexão especular

Na equação (3), referida na alínea 2.2, temos ainda a componente n, que é o expoente de reflexão especular (varia entre 1 e infinito). O fenómeno de reflexão é aproximado consoante o aumento do valor deste expoente quando queremos simular materiais mais polidos. Para um espelho perfeito, este valor seria infinito. No nosso código, esta componente é caracterizada pela variável "Shininess", que em ambos casos nas figuras anteriores, se encontra com um valor atribuído de 80, como é indicado na figura 27. Variando este valor, e mantendo o KsVal = 1, temos as figuras que se seguem:

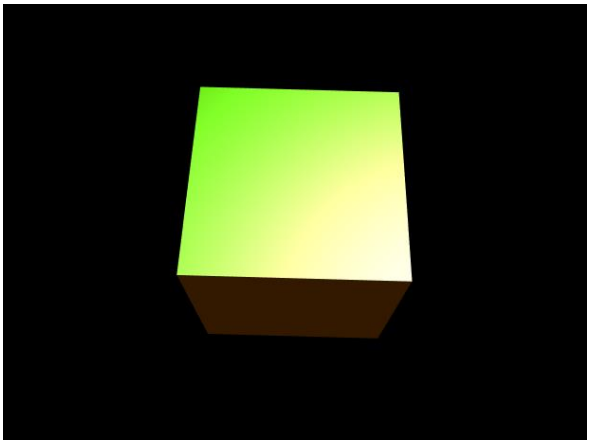


Figura 28 - KsVal = 1, Shininess = 2

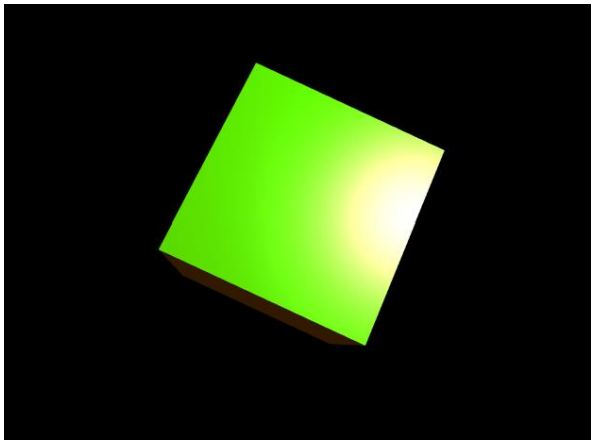


Figura 29 - KsVal = 1, Shininess = 20

Comparando estas figuras, podemos então afirmar que, quão maior for o valor desta componente, mais atenuado será o efeito de brilho refletido na superfície do cubo.

2.5 Implementação de uma fonte de luz pontual

Uma fonte de luz pontual localiza-se num ponto P, e emite luz em todas as direções com a mesma intensidade. Para esta alínea, foram efetuadas várias tentativas, não tendo sido nenhuma delas bem sucedida.

Questão 3: Animação

Para a realização deste exercício as questões 3.1 e 3.2 tiveram o *Sample 7* como base, enquanto que o 3.3 foi realizado com recurso ao script obtido no exercício 1.

3.1 Produção da animação da cena iluminada – rotação da luz direcional

Para garantir a movimentação da luz direcional em torno da cena, foi necessário alterar o valor das coordenadas de X e Z no vetor que define a sua direção. Este procedimento foi o que está assinalado na linha 53 do código (número 2 da figura 30).

Para o implementar foi necessário utilizar a fórmula de conversão de coordenadas esféricas para cartesianas. Assim, definiu-se um valor inicial para o angulo de rotação (ang_teta) e um raio, após o qual se definiram as coordenadas das componentes X e Z (x_light_dir e z_light_dir) a alterar no vetor da direção da

luz. Este ângulo de rotação é atualizado dentro da função drawScene com a adição do "deltaTime". Com esta atualização volta-se a calcular o valor das componentes X e Z (linhas 525 a 527 do número 4 da figura 30).

Após ter este sistema de atualização das componentes X e Z para a direção da luz, foi necessário atualizar o *vertex shader*, o *programInfo* e garantir que estes valores seriam atualizados para cada frame. Estes procedimentos foram implementados respetivamente pelas linhas de código 39 e 40, 97 e 98, e 521 e 522 (números 2, 3 e 4 da figura 30).

(1)

```
3   //Componentes X e Z da direção da luz
4   //Estes parametros serão indicados e alterados no vetor da direção da luz direcional
5   //Variam com o deta_time
6   let ang_teta = 0;
7   let raio = 10;
8   let x_light_dir = raio * Math.cos(ang_teta);
9   let z_light_dir = raio * Math.sin(ang_teta);
10
```

(2)

```
28    // Vertex shader program
29
30    const vsSource = `
31      attribute vec4 aVertexPosition;
32      attribute vec3 aVertexNormal;
33      attribute vec2 aTextureCoord;
34
35      uniform mat4 uNormalMatrix;
36      uniform mat4 uModelViewMatrix;
37      uniform mat4 uProjectionMatrix;
38
39      uniform float x_light_dir;
40      uniform float z_light_dir;
41
42      varying highp vec2 vTextureCoord;
43      varying highp vec3 vLighting;
44
45      void main(void) {
46        gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
47        vTextureCoord = aTextureCoord;
48
49        // Apply lighting effect
50
51        highp vec3 ambientLight = 0.5* vec3(1.0, 1.0, 1.0);
52        highp vec3 directionalLightColor = vec3(1, 1, 1);
53        highp vec3 directionalVector = normalize(vec3(x_light_dir, 0.0, z_light_dir));
54
55        highp vec4 transformedNormal = uNormalMatrix * vec4(aVertexNormal, 1.0);
56
57        highp float directional = max(dot(transformedNormal.xyz, directionalVector), 0.0);
58        vLighting = ambientLight + (directionalLightColor * directional);
59      }
60    `;
```

(3)

```
85    const programInfo = {
86      program: shaderProgram,
87      attribLocations: {
88        vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
89        vertexNormal: gl.getAttribLocation(shaderProgram, 'aVertexNormal'),
90        textureCoord: gl.getAttribLocation(shaderProgram, 'aTextureCoord'),
91      },
92      uniformLocations: {
93        projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
94        modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
95        normalMatrix: gl.getUniformLocation(shaderProgram, 'uNormalMatrix'),
96        uSampler: gl.getUniformLocation(shaderProgram, 'uSampler'),
97        x_light_dir: gl.getUniformLocation(shaderProgram, "x_light_dir"),
98        z_light_dir: gl.getUniformLocation(shaderProgram, "z_light_dir"),
99      }
100   };
```

(4)

```
516   ////////////////////////////////////////////////////////////////
517   // Questão 3.1
518
519
520   //Atribuição de valores
521   gl.uniform1f(programInfo.uniformLocations.x_light_dir, x_light_dir);
522   gl.uniform1f(programInfo.uniformLocations.z_light_dir, z_light_dir);
523
524   //Atualização de valores
525   ang_teta += deltaTime;
526   x_light_dir = raio * Math.cos(ang_teta);
527   z_light_dir = raio * Math.sin(ang_teta);
528 }
```

Figura 30 – Implementação da rotação da fonte de luz direcional

3.2 Produção da animação da cena original – movimento da câmara

Para a movimentação da câmara em torno da cena foi necessário introduzir movimentos na modelViewMatrix. Foi utilizada a função *lookAt* para garantir a rotação da câmara em torno do centro de representação em coordenadas (0, 0, 0).

Para o cálculo das coordenadas de onde se visualiza, também foram utilizadas as fórmulas de conversão de coordenadas esféricas em coordenadas cartesianas (linhas de código 411 a 413 da figura 31). Após o seu cálculo estas coordenadas foram indicadas para a função *lookAt* como posição do observador, e desta forma atingiu-se o efeito de rotação da câmara em torno do objeto.

A posição do observador é atualizada a cada frame com a variação do ângulo de observação em função do "cubeRotation".

```
397     // Set the drawing position to the "identity" point, which is
398     // the center of the scene.
399     const modelViewMatrix = mat4.create();
400
401     // Now move the drawing position a bit to where we want to
402     // start drawing the square.
403
404
405     // mat4.lookAt(modelViewMatrix, [15, 10, 5], [0, 0, 0], [0.0, 1.0, 0.0]);
406     // out - eye - center - up
407     raio_observacao = 20; //distancia da camara
408     ang_phi = 45; // corresponde à observação da câmara ao nível do plano XXYY
409     ang_teta = cubeRotation*-0.3; //o angulo de rotação é implementado com um contador que aumenta com a pasagem do tempo
410     //Cálculo de coordenadas cartesianas para implementar na função LookAt
411     const x = raio_observacao * Math.sin(ang_phi) * Math.cos(ang_teta);
412     const y = raio_observacao * Math.sin(ang_phi) * Math.sin(ang_teta);
413     const z = raio_observacao * Math.cos(ang_phi);
414     mat4.lookAt(modelViewMatrix, [x, z, y], [0, 0, 0], [0.0, 1.0, 0.0]);
415
```

Figura 31 – Implementação da rotação da câmara.

3.3 Implementação de movimentos nos objetos – sistema solar

Como referido na explicação do exercício 1.1, o grupo optou por criar funções geradores de *buffers* para cada objeto, e por esse motivo também teve de criar uma nova função (*draw*) para iterar na liste destes *buffers* e implementar os objetos na cena (figura 3). Por estes motivos foi possível implementar a solução desta questão apenas na função draw, na fase de criação das matrizes de composição de transformações geométricas para cada um deles (m1 - cubo, m2 - pirâmide, m3 - esfera).

A m2 relativa à pirâmide necessitou apenas da translação para se afastar da localização do observador (linhas de código 154 a 156 da figura 32).

Já a m1 do cubo além desta última translação, foi aplicada uma rotação em torno do eixo Y e uma translação em X. A aplicação destas matrizes dá-se por ordem inversa, ou seja, primeiro fazemos um afastamento em X de valor 10 que simboliza o raio até ao centro (pirâmide), aplicamos a rotação em torno do centro e com esse raio, e por fim garantimos que está afastado na mesma quantidade que a pirâmide do observador (linhas de código 148 a 151 da figura 32).

Por fim, para a esfera teríamos de garantir que esta além de acompanhar todos os movimentos do planeta (cubo), teríamos de assegurar mais uma rotação em torno deste com um raio menor. Assim, o que efetuamos foi garantir a m3 todos os movimentos de m2, mas acrescentamos mais uma rotação em torno do eixo Y e respetivo afastamento por uma translação em 4 unidades. Desta forma seguimos o mesmo princípio de que anteriormente, ou seja, a ordem de aplicação das transformações geométricas é por ordem inversa, devendo-se iniciar pela adição do raio de rotação da esfera sobre o cubo, a respetiva rotação em torno deste, e por fim todas as outras transformações geométricas sofridas pelo cubo (linhas de código 159 a 165 da figura 32).

Se fosse necessário acrescentar rotação a cada um dos objetos em torno dos próprios eixos, bastaria acrescentar nestas matrizes m1 a m3 uma rotação no fim, ou seja, a primeira transformação a aplicar.

```
var cubeRotation = 1.0;
var piramideRotation = 1.0;
var esferaRotation = 1.0;
```

(1)

```
// Esta função vai iterar na lista com os objetos a desenhar
// em cada uma dessas iterações aplica uma matriz composta pelos
// movimentos que queremos executar (translate, rotate, scale)
function draw(gl, programInfo, buffers, deltaTime){
    gl.clearColor(0.0, 0.0, 0.0, 1.0);  // Clear to black, fully opaque
    gl.clearDepth(1.0);                 // Clear everything
    gl.enable(gl.DEPTH_TEST);           // Enable depth testing
    gl.depthFunc(gl.LEQUAL);            // Near things obscure far things

    // Clear the canvas before we start drawing on it.

    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // Criar as martizes compostas, uma por objeto (translate, rotate, scale)
    // Matriz do cubo
    //let LAt = mat4.lookAt(mat4.create(), [0.0, 10.0, 0.0], [0.0, 0.0, 0.0], [1.0, 0.0, 0.0]);

    let m1 = mat4.create();
    mat4.translate(m1, m1,[0, 0.0, -36.0]);
    mat4.rotate(m1, m1, cubeRotation*0.5, [0, 1, 0]);
    mat4.translate(m1, m1,[10, 0.0, 0.0]);


    // Matriz da piramide
    let m2 = mat4.create();
    mat4.translate(m2, m2,[0.0, 0.0, -36.0]);
    //mat4.rotate(m2, m2, piramideRotation, [0, 1, 0]);

    // Matriz da esfera
    let m3 = mat4.create();
    mat4.translate(m3, m3,[0, 0.0, -36.0]);
    mat4.rotate(m3, m3, esferaRotation*0.5, [0, 1, 0]);
    mat4.translate(m3, m3,[10, 0.0, 0.0]);
    mat4.rotate(m3, m3, esferaRotation*2, [0, 1, 0]);
    mat4.translate(m3, m3,[4, 0.0, 0.0]);

    //lista de MVMatrix criada para cada objeto
    matrizes = [m1, m2, m3]

    // Estes valores têm de ser iguas aos das matrizes de translação m1,m2,m3
    let base_location = [[-4.5, 0.0, 0.0], [0.0, 0.0, 0.0], [4.5, 0.0, 0.0]]

    for (i = 0; i < buffers.length; i++) {
        //valor de vértices recebido de cada função que cria os objetos
        n_points = buffers[i]["indices_obj"].length

        // "matrizes[i]" é a MVMatrix criada para cada objeto
        drawScene(gl, programInfo, buffers[i], deltaTime, matrizes[i], n_points, base_location[i]);
        }
}
```

(2)

```
    // Update the rotation for the next draw

    cubeRotation += deltaTime;
    piramideRotation += deltaTime;
    esferaRotation += deltaTime;
}
```

(3)

Figura 32 – Implementação das matrizes de composição de transformações geométricas para cada objeto.

Questão 4: Novos modelos de objetos

4.1: Simplificação do modelo da estátua, exportação e conversão em .JSON

Devido à estrutura da nossa estátua realizada no projeto anterior, não foi necessário simplificar a mesma. Seguindo as instruções expostas na Figura 33, e com as opções selecionadas apresentadas na figura 34, obtivemos os ficheiros *estatua.obj* e *estatua.mtl*. Utilizando o script em python fornecido, convertemos estes 2 ficheiros no ficheiro *estatua.json* (inicialmente chamado *part1.json*), a partir do comando "python2 obj_parser.py estatua.obj estatua.mtl".

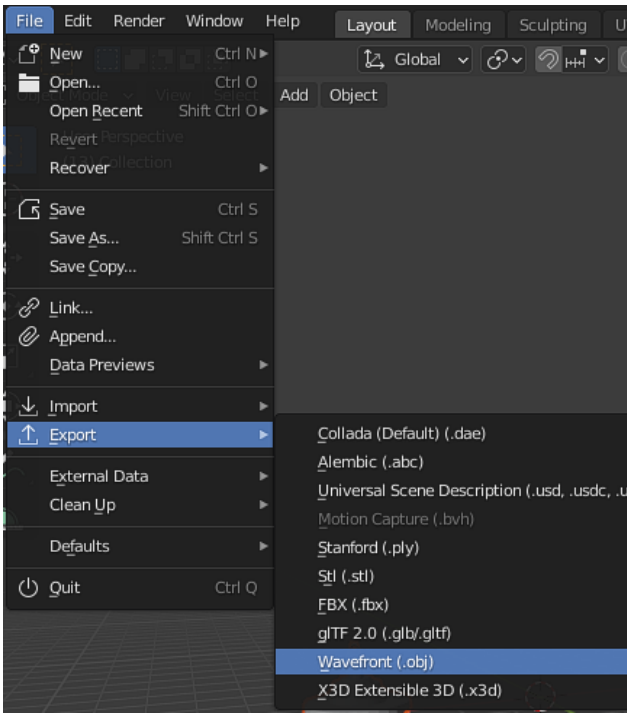


Figura 33 - Blender: Export > Wavefront (.obj)

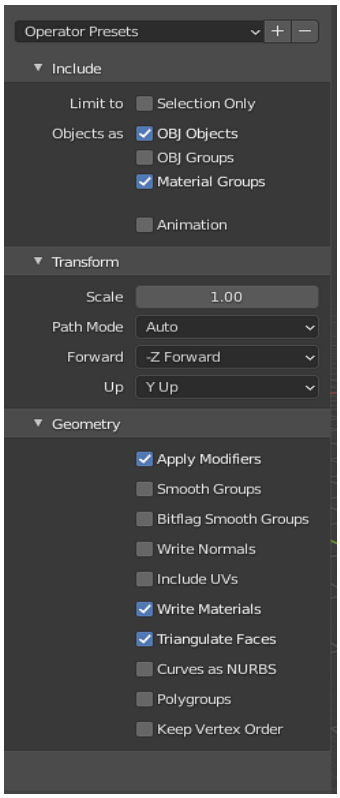


Figura 34 - Blender: Operator Presets

4.2: Adição dos objectos e da estátua à *Scene*

De forma a adicionar a estatua à cena criada no ficheiro OfficeStart.html fornecido (para a questão 4 decidimos escrever o script diretamente no ficheiro .html), apenas foi necessário acrescentar à função load() a expressão *"Scene.loadObject('estatua.json', Estatua)"* (como descrito na figura 35).

```
function load(){
    //Load the office building
    Scene.loadObjectByParts('models/geometry/Building/part','Office',758);
    //Load the ground
    Scene.loadObject('models/geometry/Building/plane.json','Plane');
    Scene.loadObject('models/geometry/Building/obj.json','Object');
    Scene.loadObject('estatua.json','Estatua')
}
```

Figura 35 - Função load() localizada no ficheiro OfficeStart.html

Para adicionar os objetos ao ficheiro officeStart, foi necessário adaptar o código que utilizámos nas questões anteriores, extraindo os vértices e os índices do cubo e da pirâmide localizados nas suas respetivas funções de *Init_buffers*, com a estrutura apresentada na figura 36.

```
let cubo = {
vertices: verticesCubo,
indices: indicesCubo,
diffuse: [1,0,0,1],
alias: 'Cubo'
};
let piramide = {
vertices: verticesPiramide,
indices: indicesPiramide,
diffuse: [0,0,1,1],
alias: 'piramide'
};
```

Figura 36 - Estrutura da pirâmide e do cubo passados à Scene

Para a esfera, foi necessário adotar uma estratégia diferente devido ao método utilizado nas questões anteriores: desenhar apenas uma porção da esfera na Scene. Como alternativa, foi utilizada a esfera presente na referência fornecida pelo guião do projeto (OpenGL Sphere, com o link exposto na Bibliografia), localizada no ficheiro *Sphere.js*.

Com todos os objetos inicializados, foi necessário criar uma função auxiliar para posicionar os objetos (apresentada na figura 37) na Scene de modo a não ficarem sobrepostos.

```
function changePosition(object, position, distance){

    position.forEach((pi,p) => {
        console.log(p,p+1,pi,position[p])
        object.vertices.forEach((i,j,a)=>{
            if (j%(p+1) == 0 && position[p] != 0){
                a[j] = i + position[p]*distance
            }
        })
    })
}
```

Figura 37 - Função auxiliar para mover os objetos na Scene

Por fim, com todos os objetos inicializados e posicionados, adicionamos os mesmos à Scene com a expressão "Scene.addObject(*{objeto}*)"(uma expressão por cada objeto), ficando com o produto final apresentado na figura 38.

Figura 38 - Scene final para a questão 4.2

Bibliografia

CLÁUDIO, Ana Paula, Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa. 2020. Apontamentos teóricos fornecidos no decorrer das aulas.

Stackoverflow, Multiply Matrices in Javascript. Consultado no mês de dezembro de 2020. – [Link]

IA725 – Computação Gráfica I. Modelos de Iluminação. 13 de maio de 2008. – [Link]

KEHRER, Johannes. Based on a WebGL applet by Prof. Thorsten Thormählen. Phong Shading (WebGL). – [Link]

Learnopengl, Camera. Consultado em 6 de dezembro de 2020. – [Link]

Keisan, Spherical to Cartesian Coordinates Calculator. Consultado em 6 de dezembro de 2020. – [Link]

Song Ho Ahn, OpenGL Sphere. Consultado em 8 de dezembro de 2020. – [Link]