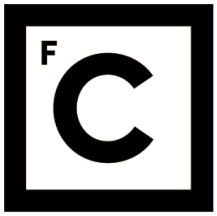
UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS



Ciências ULisboa

Relatório Computação Gráfica 2020/2021

Trabalho – WebGL

WebGL – Sample7

Grupo 08
André Costa nº52788
João Marto nº52818
Sara Graça nº52804

Parte 1 – Modificações na cena original

1.1 e 1.2

Criação do Cubo:

A sample7 já o criava apenas tivemos de retirar as texturas e substituímos por um vetor que continha cores como queríamos.

```
const faceColors = [
  [0.0,  1.0,  0.0,  1.0],    // Front face: green
  [0.0,  1.0,  0.0,  1.0],    // Back face: green
  [0.0,  1.0,  0.0,  1.0],    // Top face: green
  [0.0,  1.0,  0.0,  1.0],    // Bottom face: green
  [0.0,  1.0,  0.0,  1.0],    // Right face: green
  [0.0,  1.0,  0.0,  1.0],    // Left face: green
];
```

A função initBuffers passou a retornar os seguintes valores.

```
return {
  position: positionBuffer,
  normal: normalBuffer,
  indices: indexBuffer,
  color: colorBuffer,
```

Chegámos o cubo mais para a esquerda aplicando uma translação no eixo X para o lado negativo.

```
mat4.translate(modelViewMatrix,     // destination matrix
               modelViewMatrix,     // matrix to translate
               [-3.0, 0.0, -6.0]);  // amount to translate
```

Indica ao WebgGL como deve retornar a cor do color buffer num atributo vertexColor.

```
{
  const numComponents = 4;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.color);
  gl.vertexAttribPointer(
      programInfo.attribLocations.vertexColor,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexColor);
}
```

Criação da pirâmide:
Criamos um vetor com as posições dos triângulos que formam a pirâmide.

```
  // Front face
  0.0,  1.0,  0.0,
  -1.0, -1.0,  1.0,
  1.0, -1.0,  1.0,
  // Left face
  0.0,  1.0,  0.0,
  -1.0, -1.0, -1.0,
  -1.0, -1.0,  1.0,
  // Back face
  0.0,  1.0,  0.0,
  1.0, -1.0, -1.0,
  -1.0, -1.0, -1.0,

  // Base face
  -1.0, -1.0, -1.0,
  -1.0, -1.0, 1.0,
  1.0, -1.0, 1.0,

  // Base Face
  1.0, -1.0, -1.0,
  -1.0, -1.0, -1.0,
```

Criação do buffer para os vértices das posições e seleciona-lo para aplicar as respetivas operações.

```
const positionBuffer2 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer2);
```

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions2), gl.STATIC_DRAW);
```

É passada a lista de posições para o WebGL para construir o formato.

Aplicamos o mesmo método para as cores e para as normais.

```
const faceColors2 = [
  [1.0,  1.0,  0.0,  1.0],    // Front face: yellow
  [1.0,  1.0,  0.0,  1.0],    // Back face: yellow
  [1.0,  1.0,  0.0,  1.0],    // Top face: yellow
  [1.0,  1.0,  0.0,  1.0],    // Bottom face: yellow
  [1.0,  1.0,  0.0,  1.0],    // Right face: yellow
  [1.0,  1.0,  0.0,  1.0],    // Left face: yellow
];

// Convert the array of colors into a table for all the vertices.

var colors2 = [];
```

```
var vertexNormals2 = [
  // Front face
  0.0, 1.0,  2.0,
  0.0, 1.0,  2.0,
  0.0, 1.0,  2.0,
  // Left face
  -2.0, 1.0, 0.0,
  -2.0, 1.0, 0.0,
  -2.0, 1.0, 0.0,
  // Back face
  0.0, 1.0, -2.0,
  0.0, 1.0, -2.0,
  0.0, 1.0, -2.0,

  // Base face
  0.0, -1.0, 0.0,
  0.0, -1.0, 0.0,
  0.0, -1.0, 0.0,
```

```
const normalBuffer2 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer2);

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexNormals2),
              gl.STATIC_DRAW);
```

A função initBuffers passou a retornar mais estes valores.

```
position2: positionBuffer2,
normal2: normalBuffer2,
color2: colorBuffer2,
```

É criada uma matriz que define o centro do objeto e são aplicados os buffers.

```
//                                                PYRAMID
{
  const numComponents = 4;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.color2);
  gl.vertexAttribPointer(
      programInfo.attribLocations.vertexColor,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexColor);
}
```

```
{
  {
  const numComponents = 3;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.normal2);
  gl.vertexAttribPointer(
  programInfo.attribLocations.vertexNormal,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
    programInfo.attribLocations.vertexNormal);
  }
```

```
gl.useProgram(programInfo.program);

// Set the shader uniforms

gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatrix,
    false,
    modelViewMatrix1);
gl.uniformMatrix4fv(
    programInfo.uniformLocations.normalMatrix,
    false,
    normalMatrix1);


{
const vertexCount = 36;
const type = gl.UNSIGNED_SHORT;
const offset = 0;
gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
}
```

Na criação da esfera foi utilizado uma fórmula fornecida pela internet.

```
var Icosahedron3D = (function () {
  function Icosahedron3D(quality) {
      this._quality = quality;
      this._calculateGeometry();
  }
  Icosahedron3D.prototype._calculateGeometry = function () {
      this.Points = [];
      this.TriangleIndices = [];
      this._middlePointIndexCache = {};
      this._index = 0;
      var t = (1.0 + Math.sqrt(5.0)) / 2.0;
      this._addVertex(-1, t, 0);
      this._addVertex(1, t, 0);
      this._addVertex(-1, -t, 0);
      this._addVertex(1, -t, 0);
      this._addVertex(0, -1, t);
      this._addVertex(0, 1, t);
      this._addVertex(0, -1, -t);
      this._addVertex(0, 1, -t);
      this._addVertex(t, 0, -1);
      this._addVertex(t, 0, 1);
      this._addVertex(-t, 0, -1);
      this._addVertex(-t, 0, 1);
      this._addFace(0, 11, 5);
      this._addFace(0, 5, 1);
      this._addFace(0, 1, 7);
      this._addFace(0, 7, 10);
      this._addFace(0, 10, 11);
      this._addFace(1, 5, 9);
      this._addFace(5, 11, 4);
      this._addFace(11, 10, 2);
      this._addFace(10, 7, 6);
      this._addFace(7, 1, 8);
      this._addFace(3, 9, 4);
      this._addFace(3, 4, 2);
      this._addFace(3, 2, 6);
      this._addFace(3, 6, 8);
      this._addFace(3, 8, 9);
      this._addFace(4, 9, 5);
      this._addFace(2, 4, 11);
      this._addFace(6, 2, 10);
      this._addFace(8, 6, 7);
      this._addFace(9, 8, 1);
      this._refineVertices();
  };
```

```
Icosahedron3D.prototype._addVertex = function (x, y, z) {
    var length = Math.sqrt(x * x + y * y + z * z);
    this.Points.push({
        x: x / length,
        y: y / length,
        z: z / length
    });
    return this._index++;
};
Icosahedron3D.prototype._addFace = function (x, y, z) {
    this.TriangleIndices.push(x);
    this.TriangleIndices.push(y);
    this.TriangleIndices.push(z);
};
Icosahedron3D.prototype._refineVertices = function () {
    for (var i = 0; i < this._quality; i++) {
        var faceCount = this.TriangleIndices.length;
        for (var face = 0; face < faceCount; face += 3) {
            var x1 = this.TriangleIndices[face];
            var y1 = this.TriangleIndices[face + 1];
            var z1 = this.TriangleIndices[face + 2];
            var x2 = this._getMiddlePoint(x1, y1);
            var y2 = this._getMiddlePoint(y1, z1);
            var z2 = this._getMiddlePoint(z1, x1);
            this._addFace(x1, x2, z2);
            this._addFace(y1, y2, x2);
            this._addFace(z1, z2, y2);
            this._addFace(x2, y2, z2);
        }
    }
};
```

```
};
Icosahedron3D.prototype._getMiddlePoint = function (p1, p2) {
    var firstIsSmaller = p1 < p2;
    var smallerIndex = firstIsSmaller ? p1 : p2;
    var greaterIndex = firstIsSmaller ? p2 : p1;
    var key = (smallerIndex << 32) + greaterIndex;
    var p = this._middlePointIndexCache[key];
    if (p !== undefined)
        p;
    var point1 = this.Points[p1];
    var point2 = this.Points[p2];
    var middle = {
        x: (point1.x + point2.x) / 2.0,
        y: (point1.y + point2.y) / 2.0,
        z: (point1.z + point2.z) / 2.0,
    };
```

```
const positionBuffer3 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer3);

var icosahedron = new Icosahedron3D(3);
var positions3 = icosahedron.Points.reduce(function (a, b, i) { return i === 1 ? [a.x, a.y, a.z, b.x, b.y, b.z] : a.concat([b.x, b.y, b.z]); });
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions3), gl.STATIC_DRAW);
```

Para as normais foi utilizada a fórmula indicada pelo link do enunciado e o utilizado o mesmo método.

```
var stackCount = 500;
var sectorCount = 1000;
const vertexNormals3 = [];
var x = 0;
var y = 0;
var radius = 1;
var lengthInv = 1.0 / radius;
var stackStep = Math.PI / stackCount;
var sectorStep = 2 * Math.PI / sectorCount;
var sectorAngle, stackAngle;
for (var i = 0; i <= stackCount; ++i) {
  stackAngle = Math.PI / 2 - i * stackStep;
  xy = radius * Math.cos(stackAngle);
  z = radius * Math.sin(stackAngle);
  for (var j = 0; j <= sectorCount; ++j) {
    sectorAngle = j * sectorStep;
    x = xy * Math.cos(sectorAngle);
    y = xy * Math.sin(sectorAngle);
```

```
const normalBuffer3 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer3);
const colorBuffer3 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer3);
faceColors3 = [
  [0.0,  0.0,  1.0,  1.0],
  [0.0,  0.0,  1.0,  1.0],
  [0.0,  0.0,  1.0,  1.0],
  [0.0,  0.0,  1.0,  1.0],
  [0.0,  0.0,  1.0,  1.0],
  [0.0,  0.0,  1.0,  1.0],
];
var colors3 = []
for (var i = 0; i < positions3.length; i++) {
```

```
const indexBuffer3 = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer3);
var indices3 = icosahedron.TriangleIndices;
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices3), gl.STATIC_DRAW);
```

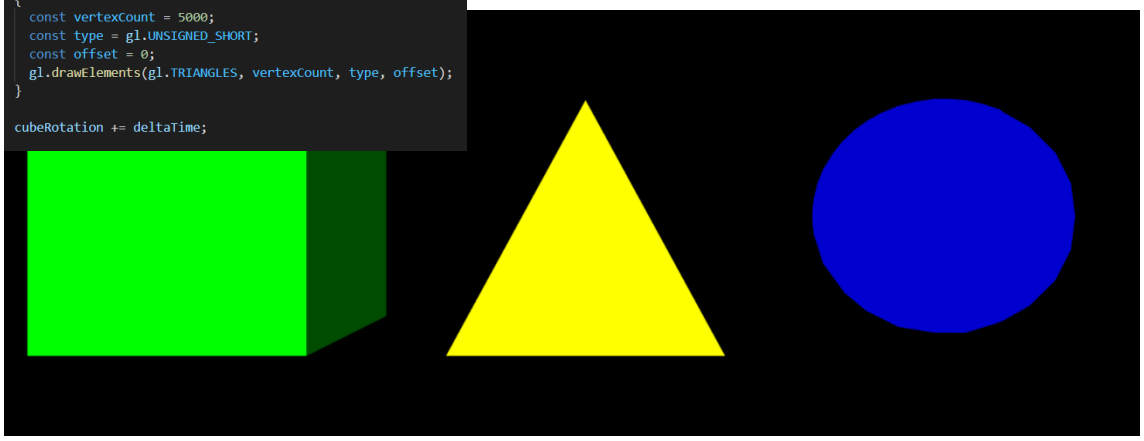
A função initBuffers passou a retornar mais estes valores.

```
position3: positionBuffer3,
color3: colorBuffer3,
indices3: indexBuffer3,
normal3: normalBuffer3,
```

É criada uma matriz que define o centro do objeto e são aplicados os buffers como efetuado para a pirâmide.

```
const modelViewMatrix2 = mat4.create();

// Now move the drawing position a bit to where we want to
// start drawing the square.

mat4.translate(modelViewMatrix2,     // destination matrix
               modelViewMatrix2,     // matrix to translate
               [3.0, 0.0, -6.0]);  // amount to translate
mat4.rotate(modelViewMatrix2,  // destination matrix
            modelViewMatrix2,  // matrix to rotate
            cubeRotation,     // amount to rotate in radians
            [0, 1, 0]);       // axis to rotate around (Z)
mat4.rotate(modelViewMatrix2,  // destination matrix
            modelViewMatrix2,  // matrix to rotate
            cubeRotation * .7,// amount to rotate in radians
            [0, 0, 0]);       // axis to rotate around (X)

const normalMatrix2 = mat4.create();
mat4.invert(normalMatrix2, modelViewMatrix2);
mat4.transpose(normalMatrix2, normalMatrix2);

// Tell WebGL how to pull out the positions from the position
// buffer into the vertexPosition attribute
{
  const numComponents = 3;
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indices3);

// Tell WebGL to use our program when drawing

gl.useProgram(programInfo.program);

// Set the shader uniforms

gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatrix,
    false,
    modelViewMatrix2);
gl.uniformMatrix4fv(
    programInfo.uniformLocations.normalMatrix,
    false,
    normalMatrix2);

{
  const vertexCount = 5000;
  const type = gl.UNSIGNED_SHORT;
  const offset = 0;
  gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
}

cubeRotation += deltaTime;
```

```
{
  const numComponents = 4;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.color3);
  gl.vertexAttribPointer(
      programInfo.attribLocations.vertexColor,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexColor);
}

// Tell WebGL how to pull out the normals from
// the normal buffer into the vertexNormal attribute.
{
  const numComponents = 3;
  const type = gl.FLOAT;
  const normalize = false;
  const stride = 0;
  const offset = 0;
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.normal3);
  gl.vertexAttribPointer(
      programInfo.attribLocations.vertexNormal,
      numComponents,
      type,
      normalize,
      stride,
      offset);
  gl.enableVertexAttribArray(
      programInfo.attribLocations.vertexNormal);
}
```



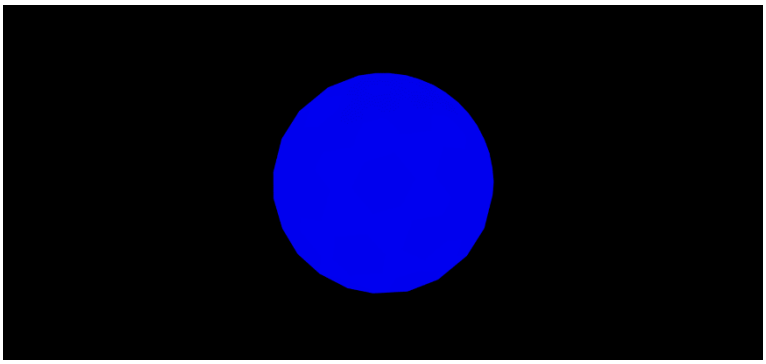
1.3

Foi feita uma rotação em torno do eixo Y da matriz de projeção que altera a câmara para apresentar a visão desejada.

```
mat4.translate(projectionMatrix, projectionMatrix, [-6,0,-7]);
mat4.rotate(projectionMatrix, projectionMatrix, -90 * Math.PI / 180, [0,1,0]);
```

Como temos Cubo, Pirâmide e Esfera e estamos à direita de todos apenas vemos a esfera.

Foi alterada também feita para o outro lado da cena por isso é

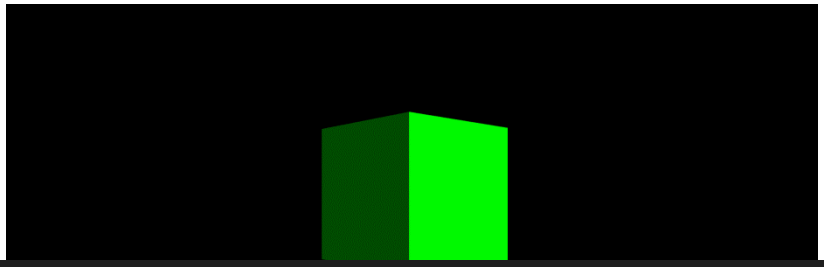


```
mat4.translate(projectionMatrix, projectionMatrix, [6,0,-10]);
mat4.rotate(projectionMatrix, projectionMatrix, 90 * Math.PI / 180, [0,1,0]);
```

apenas visível o cubo.

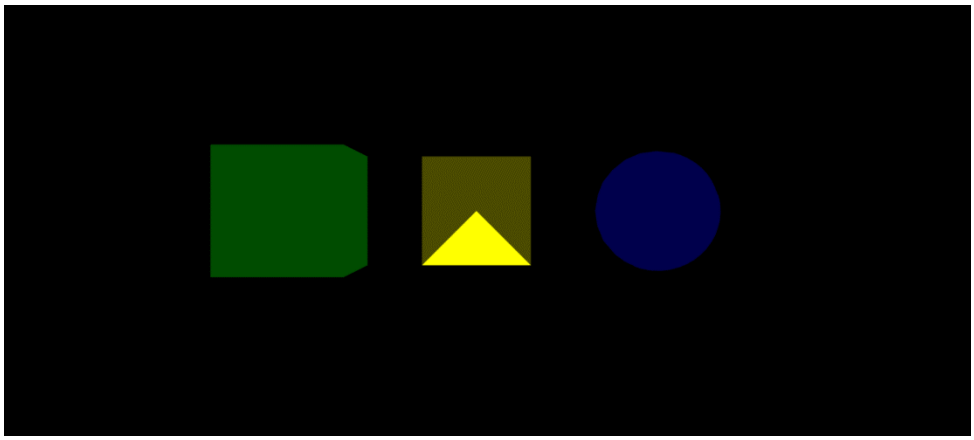
1.4

Foi feita uma



```
mat4.translate(projectionMatrix, projectionMatrix, [0,-6,-10]);
mat4.rotate(projectionMatrix, projectionMatrix, 90 * Math.PI / 180, [1,0,0]);
```

rotação em torno do X na matriz projeção para apresentar o topo da cena.



Parte 2 – Iluminação

2.1 – Implementar Phong Shading (Luz Ambiente + Luz Difusa)
Implementámos o método de **Phong Shading**, partindo do *sample 7* que já tinha o método de **Gouraud** implementado.
Em termos de **WebGl** a diferença é que **Gouraud** faz os seus cálculos nos vértices e interpola esses cálculos para cada pixel enquanto **Phong** faz os cálculos para cada pixel.

```
 // O main do nosso Vertex shader ficou assim:
   void main(void) {
     gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
     vTextureCoord = aTextureCoord;
     vNormal = uNormalMatrix * vec4(aVertexNormal, 1.0);
   }

// O nosso Fragment shader ficou mais complexo:
 const fsSource = `
   varying highp vec2 vTextureCoord;
   varying highp vec4 vNormal;
   uniform sampler2D uSampler;

   void main(void) {
     highp vec4 texelColor = texture2D(uSampler, vTextureCoord);

     highp vec3 ambientLight = vec3(0.3, 0.3, 0.3);
     highp vec3 directionalLightColor = vec3(1, 1, 1);
     highp vec3 directionalVector = normalize(vec3(0.85, 0.8, 0.75));

     highp float directional = max(dot(vNormal.xyz, directionalVector), 0.0);
     highp vec3 vLighting = ambientLight + (directionalLightColor * directional);
     gl_FragColor = vec4(texelColor.rgb * vLighting, texelColor.a);
   }
 `;
```

Os resultados foram muito semelhantes entre os dois métodos de **iluminação**, embora se notasse mais diferença se estivéssemos a visualizar objetos com um número de polígonos mais elevados, como por exemplo uma *esfera*.

Nas imagens seguintes conseguimos visualizar a diferença entre os dois métodos:

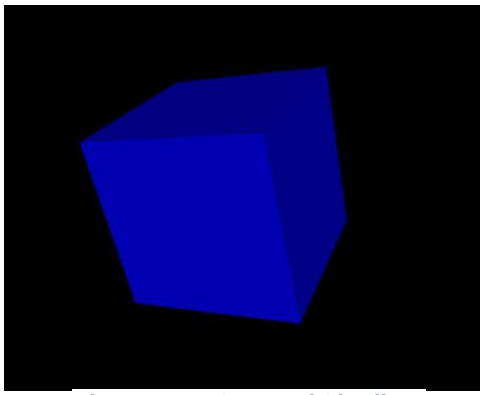


Figura 1.1.1 Gouraud Shading

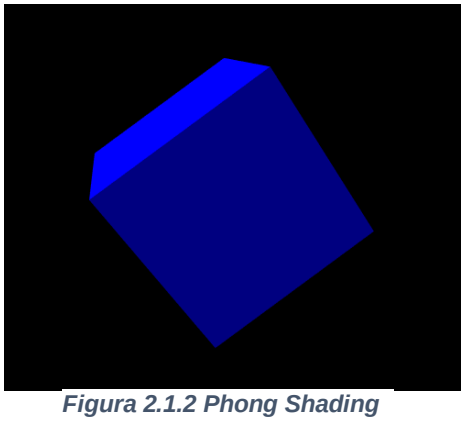


Figura 2.1.2 Phong Shading

Como podemos ver as diferenças são poucas, embora se note uma qualidade ligeiramente superior na **Figura 2**.

Para além disso podemos também visualizar as duas componentes do **Phong Shading**, a **luz ambiente** e a **luz difusa** separadamente:

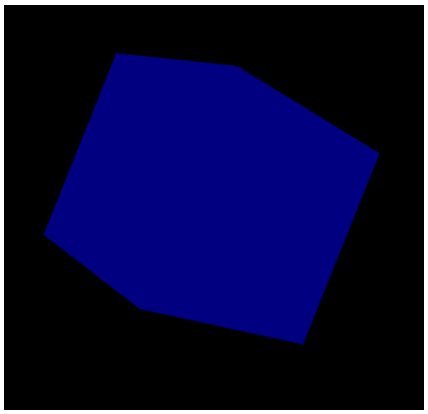


Figura 3.1.3 Phong Shading - Luz Ambiente

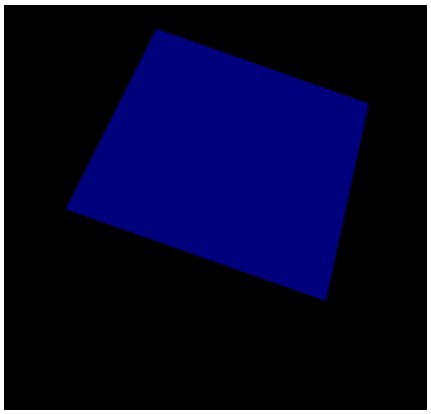


Figura 2.1.4 Phong Shading - Luz Difusa

Conseguimos então perceber que a **luz ambiente** ilumina o cubo na sua totalidade, enquanto a componente **difusa** apenas ilumina a face em que a luz incide (neste caso a face que estava no topo, pois a luz estava a vir de cima).

2.2 - Implementar Phong Shading (Luz Especular)

Implementámos agora a componente da **luz especular**. Para tal tivemos que saber a posição de visualização, para que pudéssemos fazer os nossos cálculos com a **direção de visualização** e a **direção da luz** em cada pixel. Utilizámos o *"shortcut"* **halfvector** para simplificar os cálculos, assim como aumentar a sua eficiência.

O nosso **vertex shader** ficou praticamente igual, logo apenas vamos apresentar o **fragment shader**:

```
// Fragment shader program
const fsSource = `
  varying highp vec2 vTextureCoord;
  varying highp vec4 vNormal;
  varying highp mat4 vModelViewMatrix;
  varying highp vec3 vPos;
  varying highp vec3 vViewPos;
  varying highp vec3 vLightPos;
  uniform sampler2D uSampler;

  void main(void) {
    highp vec4 texelColor = texture2D(uSampler, vTextureCoord);

    //Luz Ambiente
    highp vec3 ambientLight = 0.5 * vec3(1.0, 1.0, 1.0);

    //Luz Difusa
    highp vec3 directionalLightColor = vec3(1, 1, 1);
    highp vec3 directionalVector = vec3(0.0, 1.0, 0.0);
    highp float directional = max(dot(vNormal.xyz, normalize(directionalVector)), 0.0);

    //Luz Especular
    highp vec3 viewerPos = vViewPos;
    highp vec3 surfaceToLightDirection = normalize(-1.0 * directionalVector);
    highp vec3 surfaceToViewDirection = (vPos - viewerPos);
    highp vec3 halfVector = normalize(surfaceToLightDirection + surfaceToViewDirection);
    highp float specular = max(dot(vNormal.xyz, halfVector), 0.0);

    highp vec3 vLighting = ambientLight + (directionalLightColor * directional);

    gl_FragColor = vec4(texelColor.rgb * vLighting + (specular * 1.0), texelColor.a);
  }
`;
```

Comparação **Phong Shading completo** com exercício anterior:

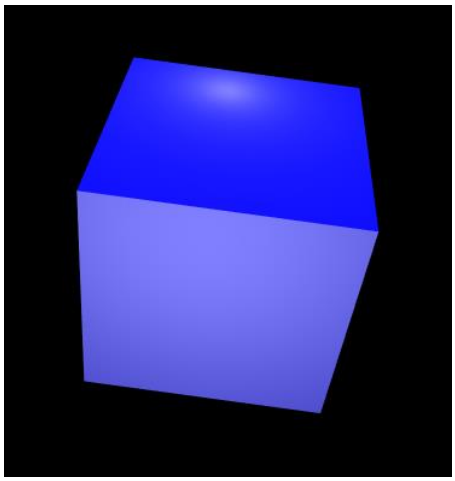


Figura 4.2.1 Phong Shading c/Luz Especular

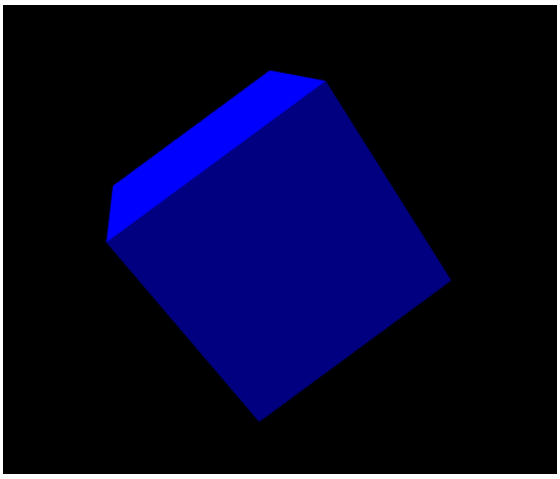


Figura 5.2.2 Phong Shading s/Luz Especular

Conseguimos claramente ver que o nosso cubo parece mais metálico e ao mesmo tempo menos baço.
A **componente especular** adiciona uma reflexão da luz que associamos a materiais metálicos, logo quão maior for o **coeficiente de especular** mais metálico e menos baço parece o nosso objeto.

2.3 – Comparação Luz Ambiente/Luz Difusa/Luz Especular

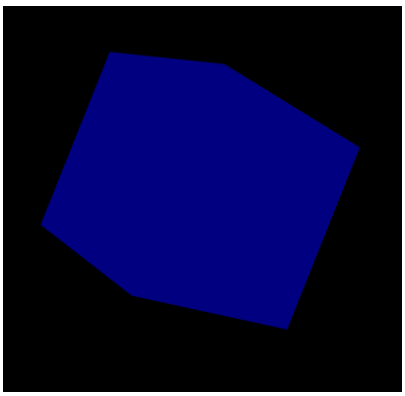


Figura 8.3.1 Luz Ambiente

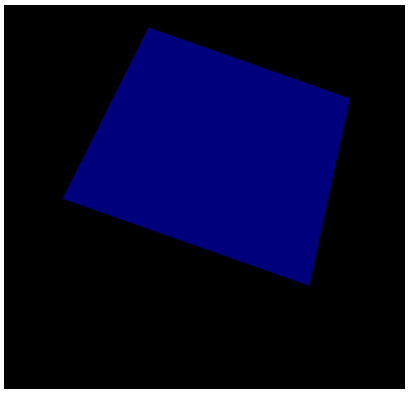


Figura 7.3.2 Luz Difusa

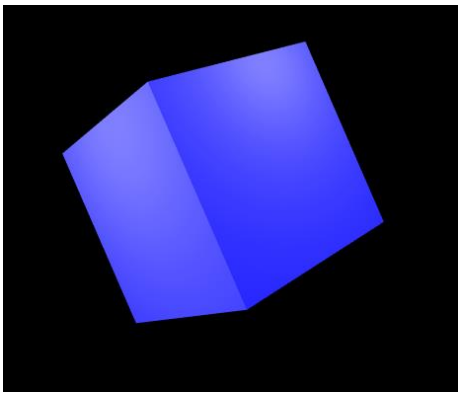


Figura 6.3.3 Luz Especular

2.4 – Comparação de coeficientes de Especularidade

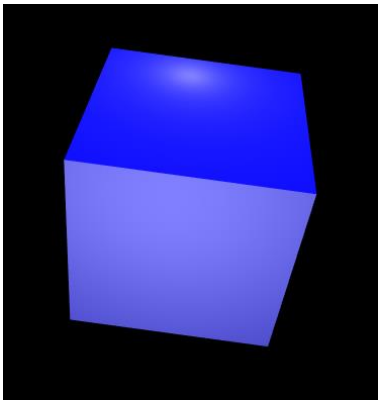


Figura 9.4.1 Coeficiente Baixo

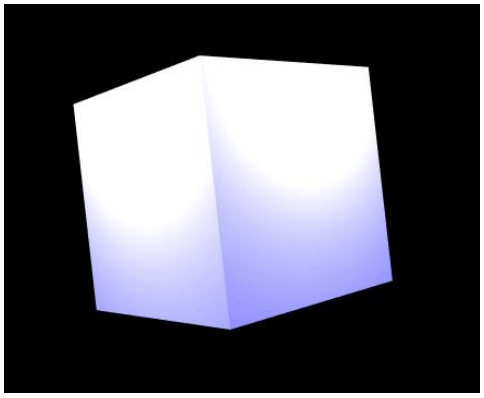


Figura 10.4.2 Coeficiente Alto

Podemos ver pelas duas imagens acima, sendo a da esquerda um cubo com um coeficiente de especularidade baixo e a da direita um cubo com um coeficiente de especularidade elevado, que o cubo da esquerda parece mais baço enquanto o da direita parece mais metálico. O aumento do coeficiente de especularidade causa um aumento na quantidade de brilho refletida pelo objeto (em cada pixel).

2.5 – Fonte de Luz Pontual

Por fim implementámos uma fonte de luz pontual. A diferença desta componente de iluminação para as outras é que emite luz em todas as direções, não apenas numa. Para o nosso programa isto significa que o nosso **fragment shader** vai ter que calcular o vetor de incidência de luz para cada pixel. Vai fazer isto a partir da posição da **fonte de luz pontual**, da posição de cada **fragmento** e calculando o vetor que vai do mesmo (do objeto) para a **fonte de luz** e utilizando-o como sendo a direção da fonte de luz.

Novamente, o nosso **vertex shader** pouco mudou. O nosso **fragment shader** ficou assim:

```
// Fragment shader program
const fsSource = `
  varying highp vec2 vTextureCoord;
  varying highp vec4 vNormal;
  varying highp mat4 vModelViewMatrix;
  varying highp vec3 vPos;
  varying highp vec3 vViewPos;
  varying highp vec3 vLightPos;
  uniform sampler2D uSampler;

  void main(void) {
    highp vec4 texelColor = texture2D(uSampler, vTextureCoord);

    //Luz Ambiente
    highp vec3 ambientLight = 0.5 * vec3(1.0, 1.0, 1.0);
```

```
      //Luz Difusa
      highp vec3 directionalLightColor = vec3(1, 1, 1);
      highp vec3 directionalVector = vec3(0.0, 1.0, 0.0);
      highp float directional = max(dot(vNormal.xyz, normalize(directionalVector)), 0.0);

      //Luz Especular
      highp vec3 viewerPos = vViewPos;
      highp vec3 surfaceToLightDirection = normalize(-1.0 * directionalVector);
      highp vec3 surfaceToViewDirection = (vPos - viewerPos);
      highp vec3 halfVector = normalize(surfaceToLightDirection + surfaceToViewDirection);
      highp float specular = max(dot(vNormal.xyz, halfVector), 0.0);

      //Luz Pontual
      highp vec3 pointLightColor = vec3(0.8, 0, 0);
      highp vec3 surfaceToLightDirectionPontual = normalize(vLightPos - vPos);
      highp float pointLighting = max(dot(surfaceToLightDirectionPontual, vNormal.xyz), 0.0);


      highp vec3 vLighting = ambientLight + (pointLightColor * pointLighting) + (directionalLigh
tColor * directional);

      gl_FragColor = vec4(texelColor.rgb * vLighting + (specular * 1.0), texelColor.a);
    }
  `;
```

Demos uma **cor vermelha** a esta **fonte de luz**.
O cubo que utilizámos como exemplo até agora é azul, significando que apenas emite luz azul, absorvendo a luz vermelha e verde. Por esta razão se quisermos visualizar a cor vermelha no cubo devemos mudar-lhe a cor. Neste caso foi-lhe aplicada uma textura para facilitar a visualização:

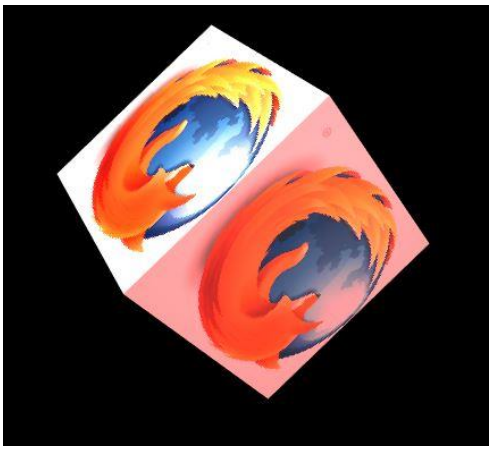


Figura 11.5.1 Luz Pontual vermelha

Parte 3 - Animação

3.1
Foi feita uma rotação da normal em torno do eixo Y para criar o efeito da luz andar em torno do cubo.

```
mat4.rotate(normalMatrix,
  normalMatrix,
  cubeRotation,
  [0, 1, 0]);
```

https://youtu.be/F0tIdeMCp8U

3.2
Foi feita uma rotação da câmara em torno do eixo Y, rodando a matriz modelViewMatrix em torno do eixo Y.

```
mat4.translate(modelViewMatrix, modelViewMatrix, [0,0,-10]);
mat4.rotate(modelViewMatrix, modelViewMatrix, cubeRotation, [0,1,0]);
```

Podemos reparar que a luz se mantém apenas numa face como seria de esperar se apenas movemos a câmara.
https://youtu.be/Vad_bzki4Qg

3.3
Para fazer o que era pedido, foi necessário mudar a ordem dos objetos sendo primeiro representada a pirâmide onde apenas aplicamos uma translação no Z assim como na câmara mas de diferentes valores.

```
mat4.translate(modelViewMatrix1,     // destination matrix
               modelViewMatrix1,     // matrix to translate
               [0.0, 0.0, -6.0]);  // amount to translate
mat4.rotate(modelViewMatrix1,  // destination matrix
            modelViewMatrix1,  // matrix to rotate
            cubeRotation,     // amount to rotate in radians
            [0, 0, 0]);       // axis to rotate around (Z)
mat4.rotate(modelViewMatrix1,  // destination matrix
            modelViewMatrix1,  // matrix to rotate
```

Depois é

```
mat4.translate(projectionMatrix, projectionMatrix, [0,0,-30]);
```

colocado o cubo onde aplicamos uma translação igual no Z e no eixo X para ficar na posição certa, de seguida aplicamos uma rotação na matriz de projeção que cria o efeito de orbita de um planeta em torno da pirâmide e aplicamos uma rotação em si mesmo para fazer o efeito de planeta a rodar em si próprio.

```
  mat4.translate(modelViewMatrix,     // destination matrix
                 modelViewMatrix,     // matrix to translate
                 [-10.0, 0.0, -6.0]);  // amount to translate
  mat4.rotate(modelViewMatrix,  // destination matrix
              modelViewMatrix,  // matrix to rotate
              cubeRotation,     // amount to rotate in radians
              [0, 0, 0]);       // axis to rotate around (Z)
  mat4.rotate(modelViewMatrix,  // destination matrix
```

```
mat4.rotate(projectionMatrix, projectionMatrix, cubeRotation, [0,1,0]);
mat4.rotate(modelViewMatrix, modelViewMatrix, cubeRotation, [0,1,0]);
```

Por fim aplicamos uma translação no eixo Z e no eixo X para meter a esfera no sítio correto para criar a órbita em volta do planeta. Depois metemos o centro de rotação no centro do planeta representado pelo cubo através de uma translação e criamos a rotação em torno do cubo com uma rotação na matriz de projeção.

```
mat4.translate(modelViewMatrix2,     // destination matrix
               modelViewMatrix2,     // matrix to translate
               [-1.0, 0.0, -4.0]);  // amount to translate
mat4.rotate(modelViewMatrix2,  // destination matrix
            modelViewMatrix2,  // matrix to rotate
            cubeRotation,     // amount to rotate in radians
            [0, 0, 0]);       // axis to rotate around (Z)
mat4.rotate(modelViewMatrix2,  // destination matrix
            modelViewMatrix2,  // matrix to rotate
            cubeRotation * .7,// amount to rotate in radians
```

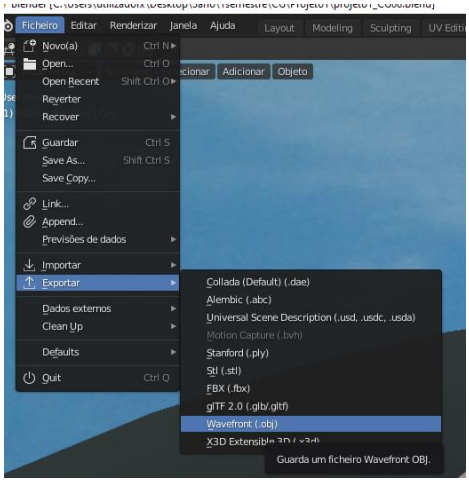
```
mat4.translate(projectionMatrix, projectionMatrix, [-10,0,-4]);

mat4.rotate(projectionMatrix, projectionMatrix, cubeRotation, [1,1,1]);
```

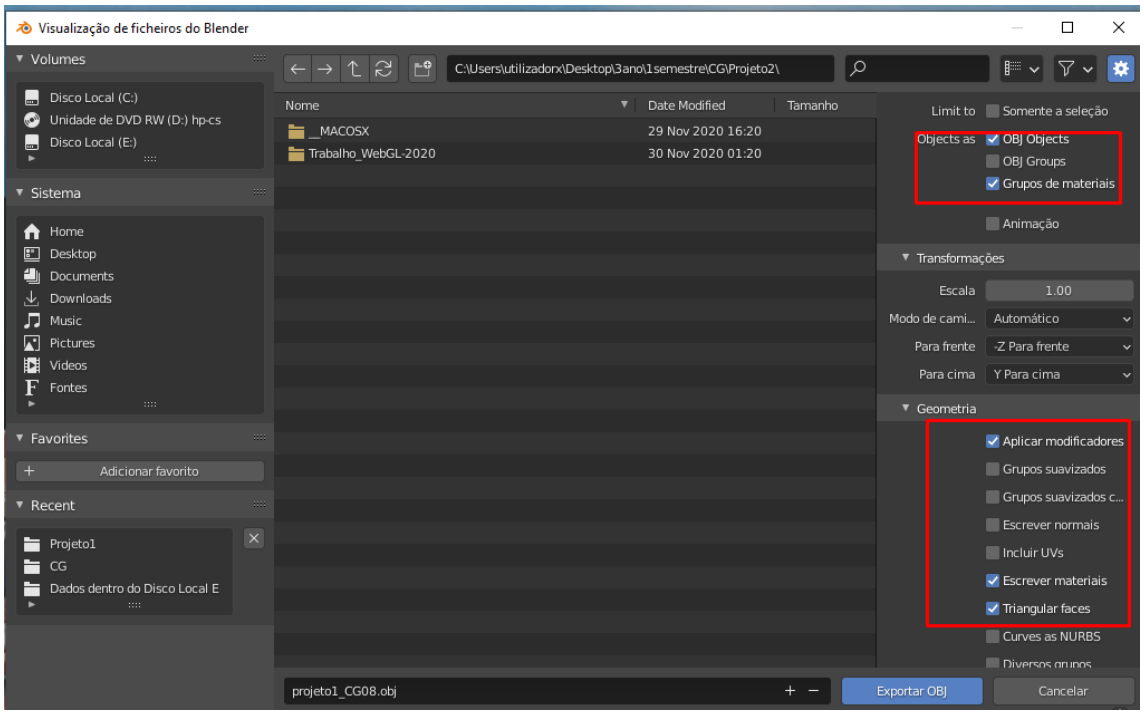
https://youtu.be/fW70hKfmbWc

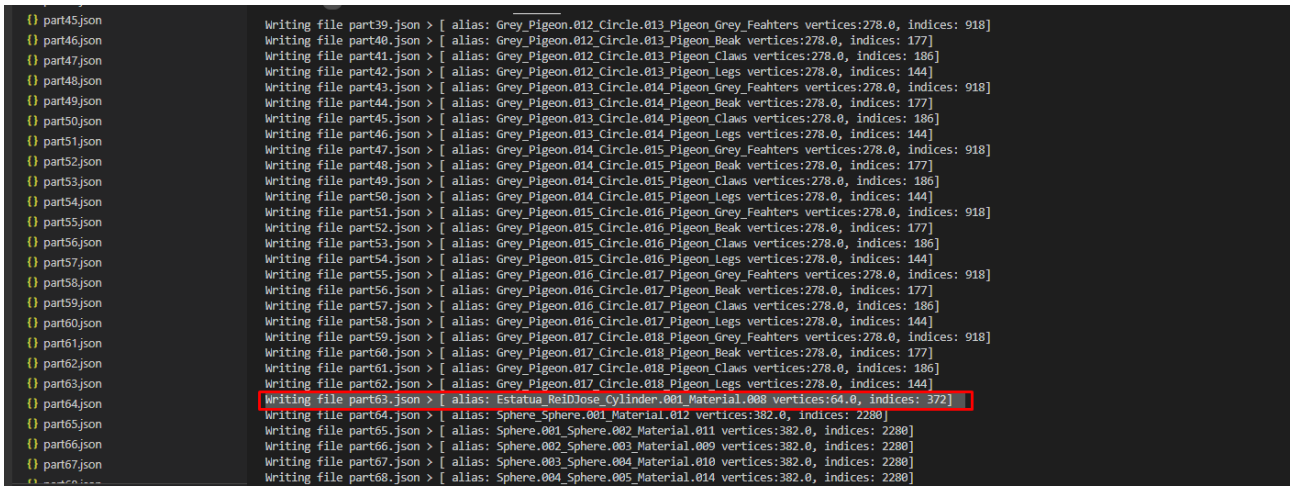
Parte 4 – Novos modelos de objetos

4.1

Seguimos o tutorial do professor passo a passo para transformar a estátua em json.







Depois foi feita uma mudança no código fornecido mais pormenorizadamente no OfficeStart.html
E afastada a câmara.

Contribuição de cada elemento

```
*/
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('part63.json','Object');
}
```

André Costa: Realizou a parte da iluminação (parte 2) e a parte do relatório que explica a sua resolução.
João Marto: Fez uma parte da realização das modificações na cena original, da animação e dos novos modelos de objetos (parte 1,3,4) e do relatório.
Sara Graça: Fez uma parte da realização das modificações na cena original, da animação e dos novos modelos de objetos (parte 1,3,4) e do relatório.