Tutorial 5 – Texturização

1. Introdução

De forma a aumentar o realismo dos objetos 3D, é possível fazer uso de imagens e aplicá-las às superfícies dos objetos geométricos. Este processo é denominado de texturização, sendo que as imagens utilizadas são designadas de texturas. À semelhança dos objetos 3D, as texturas também têm um espaço de coordenadas, mas de duas dimensões (*UV*) com valores entre 0 e 1 em que as coordenadas das texturas são designadas como *texels*.

Para aplicar uma textura a um objeto, é necessário mapear cada vértice do objeto 3D a um ponto da textura que tem como referência o referencial *UV*. Tendo a Figura 1 como exemplo, cada ponto da geometria final tem uma representação no referencial *UV*. O *fragment shader* é responsável por mapear cada ponto da textura na geometria.

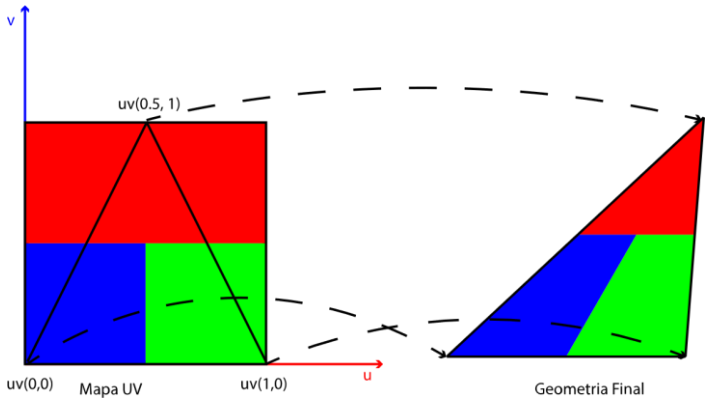


Figura 1 – Mapeamento de texturas

2.1. Objetivos de aprendizagem

Com este tutorial, irás aprender a texturizar um cubo no espaço de coordenadas *UV*. Para tal, irás aprender a criar um buffer para a textura, aprender a configurar a *GPU* com parâmetros base e como enviar essa mesma textura para a *GPU*.

2. Texturização em WebGL

2.2. Ficheiros necessários

1. À semelhança dos tutoriais anteriores, cria uma pasta com o nome "*Tutorial 5*" e copia para dentro dessa pasta todos os ficheiros do tutorial anterior. Apaga todos os comentários existentes no código para uma melhor organização e compreensão da realização deste tutorial.

2. Dirige-te à pasta "*Tutorial 5*" e cria uma pasta com o nome "*Images*".

3. Guarda a imagem "*boxImage*" que descarregaste juntamente com o tutorial na pasta "*Images*" que se encontra dentro da pasta "*Tutorial 5*".

2.3. Texturização

1. Abre o VSCode e, no programa, abre o ficheiro "*index.html*" localizado na pasta "*Tutorial 5*"

2. Importa a imagem que descarregaste com recurso ao código assinalado a vermelho:

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            WebGL
        </title>
        <body onload="Start()">
            <!--
            A linha seguinte adiciona a imagem que descarregaste ao documento HTML,
            dá-lhe um id para ser mais fácil a sua procura e ainda dá-lhe o comprimento
            e altura de 0 unidades para não aparecer quando a página é carregada.
            -->
            <img id="boxImage" src="./Images/boxImage.jpeg" width="0" height="0"></img>
            <script src="https://cdnjs.cloudflare.com/ajax/libs/mathjs/11.6.0/math.js"></script>
            <script src="./JavaScript/matrizes.js"></script>
            <script src="./JavaScript/camara.js"></script>
            <script src="./JavaScript/shaders.js"></script>
            <script src="./JavaScript/app.js"></script>
            <p>
                Estás a visualizar a aplicação gráfica baseada em <b>WebGL</b>.
                <a href="threejs.html">Clica aqui para veres a aplicação baseada em ThreeJS.</a>
            </p>
        </body>
    </head>
</html>
```

3. Nos tutoriais anteriores estavam a utilizar código *RGB* para cada um dos vértices do cubo. Agora, ao aplicar a textura ao cubo criado, iremos utilizar o referencial *UV* de forma a mapear a textura ao cubo. Para isso, abre o ficheiro "*shaders.js*" e altera o *vertex shader* tal como se segue:

```
var codigoVertexShader = [
    'precision mediump float;'
    ,
    'attribute vec3 vertexPosition;',
    // Uma vez que agora vamos receber coordenadas UV já não vamos precisar da linha
    // 'attribute vec3 vertexColor;', e vamos substituí-la pela linha abaixo
    'attribute vec2 texCoords;',
    ,
    // Pela mesma explicação da variável acima, agora já nao vamos precisar da linha
    // 'varying vec3 fragColor;', e vamos substituí-la pela linha abaixo.
    'varying vec2 fragTexCoords;',
    ,
    'uniform mat4 transformationMatrix;',
    'uniform mat4 visualizationMatrix;',
    'uniform mat4 projectionMatrix;',
    'uniform mat4 viewportMatrix;',
    ,
    'void main(){',
    // Uma vez que agora não estamos a utilizar cores, temos que adaptar a linha abaixo
    // para que o fragmentShader receba as coordenadas UV.
    '    fragTexCoords = texCoords;',
    '    gl_Position = vec4(vertexPosition, 1.0) * transformationMatrix * visualizationMatrix * projectionMatrix * viewportMatrix;',
    '}'
].join('\n');

```

4. Para além de alterar o *vertex shader* temos também que alterar o *fragment shader* para receber as informações que provêm do *vertex shader*. Para isso altera o código do *fragment shader* da seguinte forma:

```
25
26  var codigoFragmentShader = [
27      'precision mediump float;'
28      ,
29      // Uma vez que agora vamos receber coordenadas UV já não vamos precisar da linha
30      // 'varying vec3 fragColor;', e vamos substituí-la pela linha abaixo
31      'varying vec2 fragTexCoords;',
32      ,
33      // Vamos adicionar uma variável que guarde a texture que vai ser utilizada para
34      // aplicar ao cubo.
35      'uniform sampler2D sampler;',
36      ,
37      'void main(){',
38      // Uma vez que temos as coordenadas UV temos que utilizar a função abaixo para ir buscar
39      // a cor de cada pixel à textura fornecida.
40      '    gl_FragColor = texture2D(sampler, fragTexCoords);',
41      '}'
42  ].join('\n');
```

5. Agora, é necessário alterar o nosso programa para passar essas informações ao shader. Primeiro, cria uma variável que guarde a textura, para isso transcreve o código abaixo para o ficheiro "*app.js*", onde são declaradas outras variáveis globais.

```
17
18  // Variável que guarda na memória da GPU a Textura que será utilizada.
19  var boxTexture = GL.createTexture();
20
```

6. Agora, na função *PrepareTriangleData()*, onde definimos cada vértice e a cor desse mesmo vértice, teremos que adaptar esse mesmo *array* para que, em vez de receber a cor, receba as coordenadas *UV* da textura. Para isso adapta o *array* para ficar igual ao seguinte:

```
    vertexPosition = [
    // Em vez de termos 3 valores para as cores RGB vamos ter apenas
    // 2 valores, que são as coordenadas UV
    //  X,      Y,      Z,      U,      V
    //  Frente
        0,      0,      0,      0,      0,
        0,      1,      0,      0,      1,
        1,      1,      0,      1,      1,
        1,      0,      0,      1,      0,

    //  Direita
        1,      0,      0,      0,      0,
        1,      1,      0,      1,      0,
        1,      1,      1,      1,      1,
        1,      0,      1,      0,      1,

    //  Trás
        1,      0,      1,      1,      0,
        1,      1,      1,      1,      1,
        0,      1,      1,      0,      1,
        0,      0,      1,      0,      0,

    // Esquerda
        0,      0,      1,      0,      1,
        0,      1,      1,      1,      1,
        0,      1,      0,      1,      0,
        0,      0,      0,      0,      0,

    // Cima
        0,      1,      0,      0,      0,
        0,      1,      1,      0,      1,
        1,      1,      1,      1,      1,
        1,      1,      0,      1,      0,

    //  Baixo
        1,      0,      0,      0,      0,
        1,      0,      1,      0,      1,
        0,      0,      1,      0,      1,
        0,      0,      0,      0,      0
    ];
```

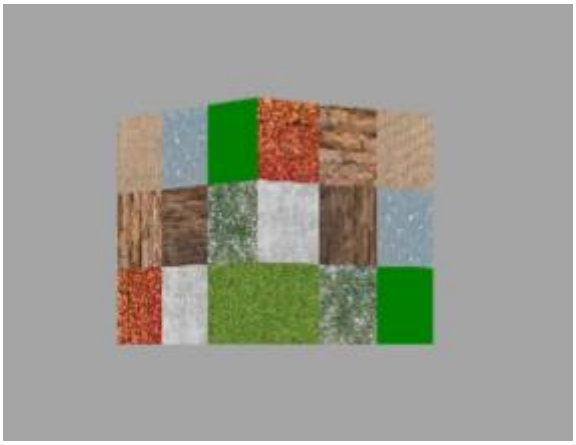
7. Agora que já tens a informação *UV* para cada vértice, é necessário configurar a *GPU* e dar-lhe a imagem que queremos que ela use para texturizar o cubo. Para isso copia o código abaixo para o fim da função *PrepareTriangleData()*:

```

    // Agora é necessário configurar os parâmetros da GPU.
    // Primeiro temos que fazer bind à textura.
    GL.bindTexture(GL.TEXTURE_2D, boxTexture);

    // Parâmetros necessários para a GPU saber como interpretar a rasterização
    // Faz Clamp à borda no eixo do U
    GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_WRAP_S, GL.CLAMP_TO_EDGE);

    // Faz Clamp à borda no eixo do V
    GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_WRAP_T, GL.CLAMP_TO_EDGE);

    // As duas linhas a baixo indicam como deve ser escalada a textura tanto
    // para diminuí-la (TEXTURE_MIN_FILTER) como para aumentá-la (TEXTURE_MAG_FILTER)
    GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MIN_FILTER, GL.LINEAR);
    GL.texParameteri(GL.TEXTURE_2D, GL.TEXTURE_MAG_FILTER, GL.LINEAR);

    // Passamos a imagem que está no documento "escondida" através do id da imagem
    GL.texImage2D(
        GL.TEXTURE_2D, // Tipo da Textura
        0,  // Detalhe da imagem (0 é o valor por defeito)
        GL.RGBA, // Tipo de imagem
        GL.RGBA, // Tipo de textura que vai ser aplicada a imagem
        GL.UNSIGNED_BYTE, // Tipo de valores da textura
        document.getElementById('boxImage') // Imagem que deve ser passada para o sampler
    );
}
```

8. Depois disso, é necessário corrigir a transferência da informação para a *GPU* uma vez que ela ainda está a utilizar o *array* como se ainda estivesse com as cores. Altera o código da função *SendDataToShaders()* para ficar igual à código da imagem abaixo.

```
function SendDataToShaders() {
    var vertexPositionAttributeLocation = GL.getAttribLocation(program, "vertexPosition");
    // Agora em vez de irmos buscar a localização da variável "vertexColor" vamos buscar
    // a localização da variável "texCoords"
    var texCoordAttributeLocation = GL.getAttribLocation(program, 'texCoords')

    GL.vertexAttribPointer(
        vertexPositionAttributeLocation,
        3,
        GL.FLOAT,
        false,
        // Agora o conjunto apenas tem 5 valores.
        5 * Float32Array.BYTES_PER_ELEMENT,
        0 * Float32Array.BYTES_PER_ELEMENT
    );

    GL.vertexAttribPointer(
        // Mudar a variável de vertexColorAttributeLocation
        // para texCoordAttributeLocation
        texCoordAttributeLocation,
        // Agora só enviámos um conjunto de 2 valores para a variável
        // texCoords
        2,
        GL.FLOAT,
        false,
        // Agora o conjunto apenas tem 5 valores.
        5 * Float32Array.BYTES_PER_ELEMENT,
        3 * Float32Array.BYTES_PER_ELEMENT
    );

    GL.enableVertexAttribArray(vertexPositionAttributeLocation);
    // Substituímos "vertexColorAttributeLocation" por "texCoordAttributeLocation"
    GL.enableVertexAttribArray(texCoordAttributeLocation);

    finalMatrixLocation = GL.getUniformLocation(program, 'transformationMatrix');
    visualizationMatrixLocation = GL.getUniformLocation(program, 'visualizationMatrix');
    projectionMatrixLocation = GL.getUniformLocation(program, 'projectionMatrix');
    viewportMatrixLocation = GL.getUniformLocation(program, 'viewportMatrix');

}
```

Agora já tens tudo pronto para abrires a página *web* e veres o cubo texturizado com a imagem que descarregamos no início deste tutorial.



3.Texturização em ThreeJS

3.1. Ficheiros necessários

1. Tal como fizemos na versão WebGL, é necessário carregar a textura para uma variável. Depois disso, precisamos de criar um material que faça uso dessa mesma textura. Para isso, no ficheiro appThree.js, adiciona o seguinte código:

```
//Primeiro, temos que carregar a textura para uma variável através do método TextureLoader()
var textura = new THREE.TextureLoader().load( './Images/boxImage.jpeg');
var materialTextura = new THREE.MeshBasicMaterial( { map: textura } );
```

2. Após importar a textura, temos que atualizar a nossa mesh para usar esse mesmo material. Assim, atualiza a linha de código

```
meshCubo = new THREE.Mesh( geometriaCubo, materialCubo );
```

para:

```
meshCubo = new THREE.Mesh( geometriaCubo, materialTextura );
```

3. Se executares a tua aplicação neste momento, deverás ver o cubo a rodar com a textura aplicada. No entanto, para o desafio, será necessário fazer o mapeamento *uv*. Para isso, temos que usar o atributo *uv* da geometria e associar os valores *uv* da textura a utilizar a cada um dos vértices do cubo. Para tal, copia o código da imagem abaixo para depois da criação da variável *materialTextura e geometriaCubo*.

```
//Criação de variável que vai conter a informação de mapeamento uv
var uvAttribute = geometriaCubo.getAttribute('uv');

//Atruibuição de posição uv a cada um dos vértices da geometria com recurso à função
// .setXY(índice do vértice, coordenada u, coordenada v)
uvAttribute.setXY(0,1,1);
uvAttribute.setXY(1,0,1);
uvAttribute.setXY(2,1,0);
uvAttribute.setXY(3,0,0);

uvAttribute.setXY(4,1,1);
uvAttribute.setXY(5,0,1);
uvAttribute.setXY(6,1,0);
uvAttribute.setXY(7,0,0);

uvAttribute.setXY(8,1,1);
uvAttribute.setXY(9,0,1);
uvAttribute.setXY(10,1,0);
uvAttribute.setXY(11,0,0);

uvAttribute.setXY(12,1,1);
uvAttribute.setXY(13,0,1);
uvAttribute.setXY(14,1,0);
uvAttribute.setXY(15,1,0);

uvAttribute.setXY(16,1,1);
uvAttribute.setXY(17,0,1);
uvAttribute.setXY(18,1,0);
uvAttribute.setXY(19,0,0);

uvAttribute.setXY(20,1,1);
uvAttribute.setXY(21,0,1);
uvAttribute.setXY(22,1,0);
uvAttribute.setXY(23,0,0);
```

4. Se executares a aplicação, deverás ter um resultado muito semelhante ao que tinhas antes de executar o passo 4. No entanto, agora podes configurar facilmente o mapeamento *uv*.

4.Desafios

Desafio 1. **Para a versão *WebGL***, configura o teu cubo para que cada uma das faces laterais corresponda a cada uma das diferentes imagens da textura, como ilustra abaixo[1].

[1] *Dica*: se tiveres dificuldades em encontrar os valores para mapear a textura corretamente para cada uma das faces, desenhar o cubo e fazer essa correspondência primeiro no papel pode ser muito útil.

Desafio 2. **Para a versão *Three.JS***, configura o teu cubo para que cada uma das faces laterais corresponda a diferentes imagens da textura, como ilustra abaixo.

ENTREGA

O trabalho deve ser submetido no **MOODLE** até dia **04/04/2025.**

A submissão é individual e deve conter todos os ficheiros necessários à correta execução da aplicação.