Tutorial 4 – Objetos 3D

1. Introdução

Em *WebGL*, os objetos 3D são obtidos através de uma malha de triângulos interligados (denominada de *mesh*). Cada triângulo é definido através de um conjunto de vértices no espaço. Cada vértice tem uma coordenada *(x,y,z)* associada que, opcionalmente, pode ter também propriedades adicionais associadas tais como cor ou coordenada de textura. A Figura 1 ilustra um cubo onde a tracejado são identificados os limites dos triângulos em cada uma das faces que compõe o cubo.

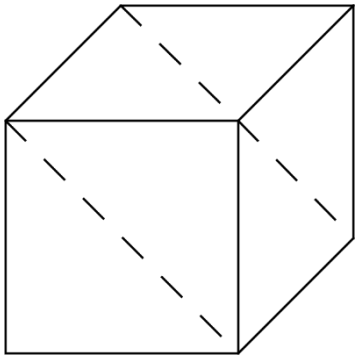

Figura 1 – Ilustração dos triângulos nas diferentes faces do cubo.

Existem duas maneiras de representar um triângulo: declarando os pontos pelo sentido de rotação dos ponteiros do relógio ou pelo sentido contrário à rotação dos ponteiros do relógio. Este fator determina se o triângulo faz parte da face traseira ou da face frontal. A regra é que os pontos declarados no sentido contrário aos ponteiros do relógio pertencem a face frontal, e os pontos declarados no sentido dos ponteiros do relógio fazem parte da face traseira.

Por defeito, cada triângulo é renderizado dos dois lados. No caso do cubo apresentado na Figura 1, apesar de ambos os lados serem renderizados, o lado dos triângulos que fica voltado para dentro do cubo nunca será visível. Embora, em cenas simples como esta, este fato não tenha grandes implicações, em cenas complexas pode gerar uma grande carga matemática para a *GPU* e comprometer a performance da cena. Para otimizar o processo de renderização nessas situações, o *OpenGL* recorre à técnica de *Culling*.

A técnica de *Culling* permite renderizar apenas as faces que estão voltadas para a câmara virtual. Para tal, é computada se a direção da normal (i. e., orientação da superfície do objeto geométrico) está voltada para a câmara virtual, sendo que a normal é calculada pela ordem através da qual os vértices do polígono são desenhados.

1.1. Objetivos de aprendizagem

Neste tutorial vais aprender a criar uma *mesh* de forma a obter um objeto 3D (cubo) através da definição de vértices e de índices de vértices. Para além disso, irás aprender a definir que faces da *mesh* são renderizadas (frontal, posterior ou ambas).

2.Objetos 3D em WebGL

2.1. Ficheiros necessários

À semelhança dos tutoriais anteriores, cria uma pasta com o nome "*Tutorial 4*" 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.

NOTA: Este tutorial pressupõe a continuação do tutorial 3 sem os desafios (ou seja, com o triângulo a rodar no meio do ecrã).

2.2. Criação do objeto 3D

Tal como referido antetiormente, a *mesh* dos objetos é constituída apenas por triângulos e o *WebGL* não é exceção. Para fazeres um cubo, é necessário entenderes que cada face do cubo tem de ser dividida em dois triângulos (tal como ilustra a Figura 1).

1. Para iniciar a criação do cubo, abre o ficheiro "*app.js*" e adiciona as três variáveis seguintes:

```
14
15  // Variável que irá guardar a posição dos vértices
16  var vertexPosition;
17
18  // Variável que irá guardar o conjunto de vértices que constituem cada triângulo
19  var vertexIndex;
20
21  // Buffer que irá guardar todos o conjunto de vértices na GPU
22  var gpuIndexBuffer = GL.createBuffer();
23
```

2. Na função *PrepareTriangleData()*, apaga as linhas que definem a variável *triangleArray* e cria uma nova variável da seguinte forma[1]:

[1] Cada linha da variável vertexPosition é um vértice do cubo. Se prestares atenção vais ver que todos os pontos se repetem três vezes, uma por cada face do cubo. Tem isto em mente pois vai ser importante para o próximo tutorial.

```
function PrepareTriangleData() {
    // Foi removido o array que continha os vertices do triângulo.
    // Assim como o array anterior, este novo array vai ter os diferentes posições de cada ponto
    // bem como o código de cores RGB de cada ponto.
    vertexPosition = [
    //  X,    Y,    Z,    R,    G,    B

    //  Frente
        0,    0,    0,    0,    0,    0,
        0,    1,    0,    0,    1,    0,
        1,    1,    0,    1,    1,    0,
        1,    0,    0,    1,    0,    0,

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

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

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

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

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

```

3. Agora é necessário construir um novo *array* com os conjuntos de pontos pelos quais cada triângulo é constituído. A Figura 2 ilustra o referencial sobre o qual vai ser construído o cubo. Imediatamente abaixo do *array* de pontos que transcreveste anteriormente, introduz o seguinte código:

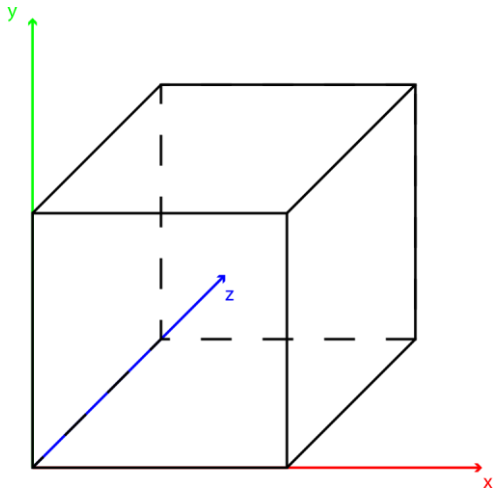

Figura 2 – Ilustração do cubo a construir sob o referencial

```
    // Array que guarda qual os indices do array anterior que constituem cada triângulo.
    // De relembrar que cada lado do cubo é constituído por dois triângulos, por exemplo:
    // a primeira linha é "0, 1, 2" isto significa que um dos triângulos da face da frente é
    // formado pela 1ª, 2ª e 3ª linha (de relembrar que o indice do primeiro elemento
    // num array é "0").
    vertexIndex =[
        //Frente
        0, 2, 1,
        0, 3, 2,

        //Direita
        4, 6, 5,
        4, 7, 6,

        //Trás
        8, 10, 9,
        8, 11, 10,

        //Esquerda
        12, 14, 13,
        12, 15, 14,

        //Cima
        16, 18, 17,
        16, 19, 18,

        //Baixo
        20, 22, 21,
        20, 23, 22
    ];

```

4. Agora que temos o *array* que define quais vértices de cada triângulo, é necessário passar essa informação para a *GPU*. Para isso, ainda na função PrepareTriangleData(), vais atualizar o teu código tal como mostrado abaixo. O primeiro texto assinalado a vermelho é apenas para mudares o nome da variável que deve encontrar-se imediatamente abaixo do *array* que acabaste de criar. O segundo texto assinalado é o texto que tens de copiar.

```
    GL.bindBuffer(GL.ARRAY_BUFFER, gpuArrayBuffer);

    GL.bufferData(
        GL.ARRAY_BUFFER,
        new Float32Array(vertexPosition), // Não esquecer que agora é uma nova variável
        GL.STATIC_DRAW
    );

    // Voltamos a fazer bind ao novo buffer que acabamos de criar dizendo que o buffer agora
    // é de ELEMENT_ARRAY_BUFFER.
    GL.bindBuffer(GL.ELEMENT_ARRAY_BUFFER, gpuIndexBuffer);
    // Passamos os dados relativos ao indices de cada triângulo
    GL.bufferData(
        GL.ELEMENT_ARRAY_BUFFER, // Indica que os dados são do tipo ELEMENT_ARRAY_BUFFER
        new Uint16Array(vertexIndex), // Agora os valores são do tipo Unsigned int 16
        GL.STATIC_DRAW
    ); // Os valores são estáticos e não irão mudar ao longo do tempo
}

```

5. De seguida vamos atualizar a primeira operação de translação para que o cubo fique no centro do ecrã e dentro do volume de visualização:

```
// Atualizamos esta linha para o triângulo "voltar" para o centro e criamos
// uma translação no eixo do Z para que o triângulo fique dentro do volume de visualização
// versão anterior: finalMatrix = math.multiply(CriarMatrizTranslacao(0.5,0.5, 0), finalMatrix);
finalMatrix = math.multiply(CriarMatrizTranslacao(0,0,2), finalMatrix);
```

6. Uma vez que vamos desenhar os pontos através dos seus índices, não podemos utilizar a função *drawArrays()*. Para desenhar esses pontos vamos à função *loop()* e, onde temos a função *drawArrays()* e substituímos essa mesma função pela função *drawElements()*:

```
237
238     // Agora, em vez de chamar-mos a função de drawArray, vamos chamar a função drawElements.
239     // Esta função permite-nos utilizar vertexIndex para dizermos quais são os elementos que
240     // constituem os triângulos.
241     GL.drawElements(
242         GL.TRIANGLES, // Queremos desenhar na mesma triângulos
243         vertexIndex.length, // O número elementos que vão ser desenhados
244         GL.UNSIGNED_SHORT, // Qual o tipo de elementos
245         0 // Qual o offset para o primeiro elemento a ser desenhado
246     );
247
```

7. Se abrires agora a página *index.html*, vais ver já tens o cubo a rodar. Irás notar que tem o aspecto um pouco estranho. Isto acontece porque os triângulos estão a ser renderizados de um dos lados. É preciso definir que o lado que tem que ser renderizado é aquele que está virado para a câmara. Para ultrapassar isso, vai à função *PrepareCanvas()* e a seguir à linha de código

```
GL.clear(GL.DEPTH_BUFFER_BIT || GL.COLOR_BUFFER_BIT);
```

adiciona o seguinte código:

```
28
29      // Permite o teste de profundidade
30      GL.enable(GL.DEPTH_TEST);
31
32      // Permite visualizar apenas os triângulos que tiverem as normais viradas para a câmara.
33      // As normais são calculadas através da ordem pela qual os triângulos forem desenhados.
34      // No sentido contrário ao ponteiro do relógio a normal vai estar virada para a câmara,
35      // caso contrário a normal vai estar a apontar na mesma direção que a câmara logo não sera visualizada.
36      GL.enable(GL.CULL_FACE);
37
```

8. Para perceberes a diferença, dirige-te à variável que define os índices dos triângulos (*vertexIndex*) e altera o valor da primeira linha de *(0,2,1)* para *(0, 1, 2)*. Se abrires agora a página *web*, verás que um dos triângulos para parte frontal não é renderizada, assim como nenhuma das faces interiores do cubo quando vistas através desse mesmo triângulo. Volta a colocar o valor da linha que alteraste a *(0,2,1)*.

3.Objetos 3D em ThreeJS

3.1. Criação do objeto 3D

Em ThreeJS, existem diferentes primitivas que permitem a criação mais expedita de objetos geométricos ou objetos 3D tais como, por exemplo, planos, círculos, cubos, esferas, cones, tetraedros, etc. Podes encontrar uma listagem completa destas primitivas em https://threejs.org/manual/#en/primitives

À semelhança do que fizemos para WebGL, vamos criar um cubo, o material e a mesh, mas desta vez vamos fazê-lo com recurso às primitivas ThreeJS. Para tal, copia o código na imagem abaixo para antes da função *Start()*.

```
//Criação da geometria de um cubo, com os parâmetros de largura, altura e profundidade de 1 unidade
var geometriaCubo = new THREE.BoxGeometry( 1, 1, 1);

//Criação do material básico que vai permitir configurar o aspeto das faces do cubo
//Neste caso, ativamos a propriedade vertexColors para que possamos definir as cores dos vértices
var materialCubo = new THREE.MeshBasicMaterial( { vertexColors: true} );

//Definição das cores dos vértices do cubo
 const vertexColorsCubo = new Float32Array( [
    1.0, 0.0,  0.0,
    0.0, 1.0,  0.0,
    0.0, 0.0,  1.0,
    0.0, 0.0,  0.0,

    1.0, 0.0,  0.0,
    0.0, 0.0,  0.0,
    0.0, 0.0,  1.0,
    0.0, 1.0,  0.0,

    0.0, 0.0,  1.0,
    0.0, 1.0,  0.0,
    0.0, 0.0,  0.0,
    1.0, 0.0,  0.0,

    0.0, 1.0,  0.0,
    0.0, 0.0,  1.0,
    1.0, 0.0,  0.0,
    0.0, 0.0,  0.0,

    0.0, 0.0,  0.0,
    1.0, 0.0,  0.0,
    0.0, 1.0,  0.0,
    0.0, 0.0,  1.0,

    0.0, 1.0,  0.0,
    1.0, 0.0,  0.0,
    0.0, 0.0,  1.0,
    0.0, 0.0,  0.0,
] );

 //Associar o array com as cores dos vértices à propriedade de cor da geometria
 geometriaCubo.setAttribute( 'color', new THREE.Float32BufferAttribute( vertexColorsCubo, 3 ) );

// Após criar a geometria e o material, criamos a mesh com os dados da geometria e do material.
var meshCubo = new THREE.Mesh( geometriaCubo, materialCubo );
```

Tal como fizemos com o triângulo, vamos posicionar o cubo de forma a ficar dentro do volume de visualização. Antes da função *Start()* adiciona o seguinte código:

```
//Criamos uma translação no eixo do Z para que o triângulo fique dentro do volume de visualização
meshCubo.translateZ(-6.0);
```

Agora, na função Start(), vamos remover o triângulo da cena comentando a linha que é responsável por adicionar o triângulo à cena na função e acrescentar uma linha de código que nos vai permitir adicionar o cubo à cena tal como destacado a vermelho na imagem abaixo.

```
function Start(){
    // Comentamos esta linha para o triângulo não ser adicionado
    //cena.add(mesh);

    //Adicionamos esta linha para adicionar o cubo à cena
    cena.add( meshCubo );

    renderer.render(cena, camaraPerspetiva);

    //Função para chamar a nossa função de loop
    requestAnimationFrame(loop);
}
```

Por fim, colocamos o nosso cubo a rodar como fizemos com o triângulo. Para isso, atualizamos a função *loop()* como indica a imagem abaixo.

```
function loop() {

    // Comentamos a linha que faz o triângulo rodar pois já não precisamos dela
    //mesh.rotateY(Math.PI/180 * 1);

    //Tal como fizemos inicialmente com o triângulo, vamos colocar o cubo a rodar no eixo do Y
    meshCubo.rotateY(Math.PI/180 * 1);

    //função chamada para gerarmos um novo frame
    renderer.render(cena, camaraPerspetiva);

    //função chamada para executar de novo a função loop de forma a gerar o frame seguinte
    requestAnimationFrame(loop);
}
```

Agora, se executares a tua aplicação, deverás ver o cubo com as faces coloridas e a rodar. Nota que o ponto de referência para a rotação é diferente entre o ThreeJS e o WebGL, numa solução o cubo gira por si mesmo, no outro roda em relação ao centro do nosso referencial.