Tutorial 2 – Transformações Geométricas

1. Introdução

No domínio da Computação Gráfica, existem três tipos de transformações geométricas planas que nos permitem manipular os objetos geométricos na cena, sendo que estas podem ser combinadas para criar transformações mais complexas. Essas transformações são:

- **Translação**: consiste na deslocação de um objeto geométrico de uma posição inicial para outra posição. Em termos práticos, a translação dá-se pela soma de um vetor de deslocação aos vetores que representam cada um dos pontos geométricos que compõem o objeto. A translação é ilustrada na Figura 1, sendo definida pela Equação 1.

$$A = \begin{bmatrix} x_A & y_A & z_A & 1 \end{bmatrix},\ T = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$A' = T(dx, dy, dz) \cdot A$$

Equação 1 –Cálculo para a translação de um objeto geométrico

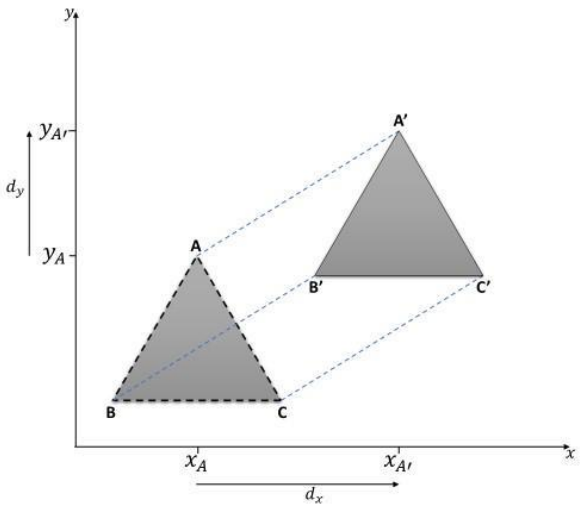


Figura 1 – Exemplificação de uma translação.

- **Variação de escala**: consiste na alteração das dimensões de um objeto geométrico em relação à origem através da multiplicação de cada um dos pontos que compõem o objeto geométrico por um fator de escala. Existem dois tipos de variação de escala: uniforme e não uniforme. Num escalamento uniforme, o fator de escala é o mesmo para todos os pontos fazendo com que a relação de aspeto e

  os ângulos do objeto se mantenham após a transformação. Já num escalamento não uniforme, o fator de escala não é o mesmo para todos os pontos, resultando num objeto com relação de aspeto e ângulos diferentes do objeto original. Os dois tipos de variação de escala são ilustrados na Figura 2, sendo que o seu cálculo é obtido através da Equação 2.

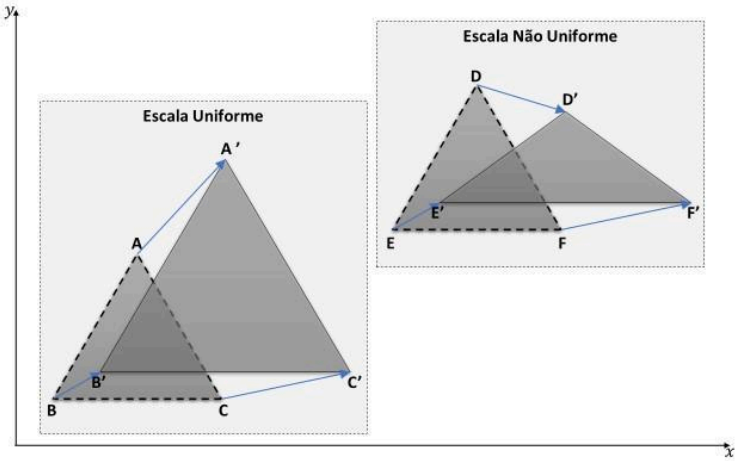


Figura 2 – Exemplificação de uma operação de escala uniforme (esquerda) e de uma operação de escala não uniforme (direita).

$$A = \begin{bmatrix} x_A & y_A & z_a & 1 \end{bmatrix}, \left(s_x, s_y, s_z\right) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$A' = S\left(s_x, s_y, s_z\right) \cdot A$$

Equação 2 –Cálculo para a operação de escala de um objeto geométrico

- **Rotação**: permite a rotação de o objeto em torno de um ponto mantendo a dimensão e relação de aspeto do objeto. Por defeito, a rotação é aplicada no sentido contrário ao da direção dos ponteiros do relógio. Este processo encontra-se ilustrado na Figura 3 e o seu cálculo definido nas Equação 3. Note-se que se deve considerar a matriz $R_x(\beta)$ para rotações no eixo do *x*, a matriz $R_y(\beta)$ para rotações no eixo do *y* e a matriz $R_z(\beta)$ para rotações no eixo do *z*.

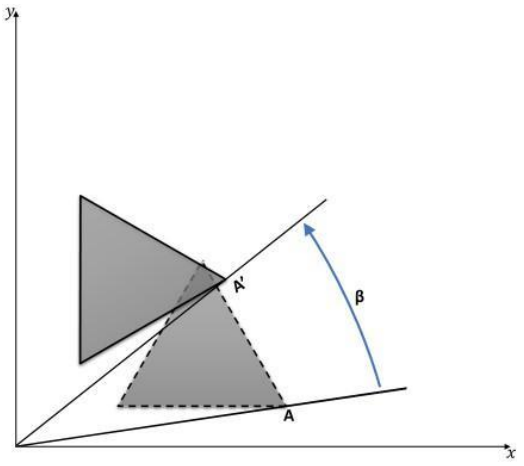


Figura 3 – Ilustração de uma operação de rotação.

$$A = \begin{bmatrix} x_A & y_A & z_A \end{bmatrix},$$

$$R_x(\beta) = [1 \quad 0 \quad 0 \quad 0\ 0 \quad \cos cos\ \beta \quad -sen\beta \quad 0\ 0 \quad sen\ \beta \quad \cos cos\ \beta \quad 0\ 0 \quad 0$$

,

$$R_y(\beta) = [\quad \cos cos\ \beta \quad 0 \quad sen\ \beta \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad -sen\ \beta \quad 0\beta \quad 0 \quad 0$$

,

$$R_z(\beta) = [\cos cos\ \beta \quad -sen\ \beta \quad 0 \quad 0\, sen\ \beta\beta \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1$$

$$A' = R(\beta)\cdot A$$

Equação 3 –Cálculo para a rotação de um objeto geométrico

1.1. Objetivos de aprendizagem

O objetivo deste capítulo é ensinar a utilizar as diferentes matrizes de transformação geométricas planas em *WebGL*. Para o efeito, irás aplicar as diferentes transformações geométricas ao triângulo que criaste no tutorial anterior, fazendo com que no final o triângulo fique a rodar sobre o eixo do *y*.

2. Transformações Geométricas com *WebGL*

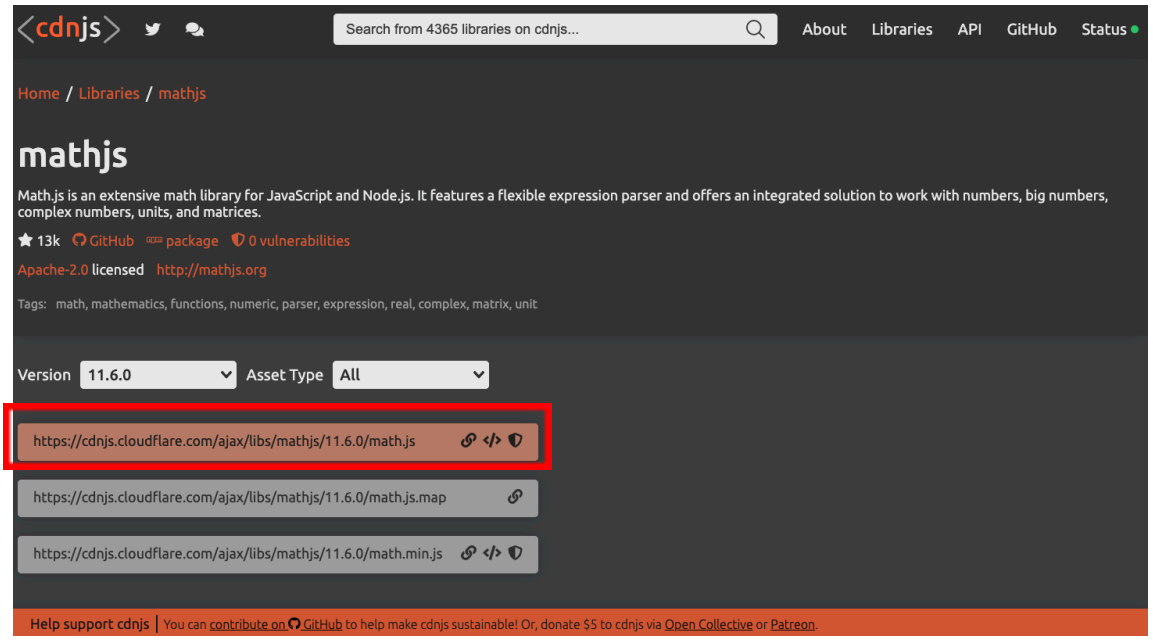
2.1. Ficheiros Necessários

Para facilitar a aprendizagem de WebGL vamos criar uma nova pasta para o segundo tutorial. Para isso dirige-te à pasta WebGL que criaste no Tutorial 1 e cria uma nova pasta com o nome "Tutorial 2". De seguida copia os ficheiros e pastas que criaste no tutorial anterior.

Com o Visual Studio Code, abre a pasta "Tutorial 2" e apaga todos os comentários (linhas que começam por //). Deste modo apenas terás os comentários deste tutorial e será mais fácil para ti perceberes o que está a ser feito. Depois de teres copiado os ficheiros e teres apagado todos os comentários vai ser necessário criar novos ficheiros e também descarregar uma biblioteca de matemática que te irá permitir fazer multiplicação de matrizes. Para isso segue os passos seguintes:

1. Dentro da pasta "JavaScript" adiciona um novo ficheiro com o nome "matrizes.js". Neste ficheiro é onde irás crias os métodos para a criação de matrizes de transformações geométricas.

2. Acede a este https://cdnjs.com/libraries/mathjs. e copia o link que está assinalado a vermelho na imagem abaixo. Isto vai garantir que estás a usar a última versão da biblioteca *mathjs*.



3. Agora, abre o ficheiro "***index.html***" e adiciona as seguintes linhas de código assinaladas a vermelho.

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            WebGL
        </title>
        <body onload="Start()">

            <script src="https://cdnjs.cloudflare.com/ajax/libs/mathjs/11.6.0/math.js"></script><!-- Biblioteca de matemática -->
            <script src="./JavaScript/matrizes.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>
```

1.2. Criar as Matrizes

Agora que já tens os ficheiros necessários vamos começar por criar as diferentes matrizes de transformações geométricas. Para isso abre o ficheiro que acabaste de criar com o nome "matrizes.js". Cada uma delas tem a respetiva matriz de transformação em coordenadas homogéneas 3D.

Matriz de translação

Vamos começar pela matriz de translação, para isso copia o código da imagem abaixo para o ficheiro "matrizes.js".

```
/**
 * @param {float} x Valor para translação no eixo do X
 * @param {float} y Valor para translação no eixo do Y
 * @param {float} z Valor para translação no eixo do Z
 * Devolve um 2D array com a matriz de translação pedida
 */
function CriarMatrizTranslacao(x,y,z)
{
    // Matriz de translação final
    return[
            [1, 0, 0, x],
            [0, 1, 0, y],
            [0, 0, 1, z],
            [0, 0, 0, 1]
    ];
}
```

A função acabaste de copiar devolve um **array de duas dimensões** com a matriz de translação tendo em conta os parâmetros de entrada. Os parâmetros de entrada indicam as unidades de deslocação em cada eixo.

Matriz de mudança de Escala

A segunda matriz que vamos criar é referente à matriz de modificação de escala. No mesmo ficheiro adiciona a função demonstrada na imagem abaixo.

```
17
18  /**
19   * @param {float} x Valor para escala no eixo do X
20   * @param {float} y Valor para escala no eixo do Y
21   * @param {float} z Valor para escala no eixo do Z
22   * Devolve um 2D array com a matriz de escala pedida
23   */
24  function CriarMatrizEscala (x, y, z)
25  {
26      // Matriz de escala final
27      return [
28          [x,  0,  0,  0],
29          [0,  y,  0,  0],
30          [0,  0,  z,  0],
31          [0,  0,  0,  1]
32      ];
33  }
34
```

Esta função devolve um **array de duas dimensões** com a matriz de modificação de escala tendo em conta os parâmetros de entrada. Os parâmetros de entrada indicam qual a rácio de que deve ser aplicada, isto quer dizer que se for passado o valor de 0.25 num dos eixos, esse eixo vai ser reduzido para ¼.

NOTA: Deves ter sempre em atenção que a escala normal é 1 unidade e não 0 unidades como o resto das transformações.

Matrizes de Rotação

Como deves saber, existem 3 matrizes de rotação, uma para cada um dos eixos (X, Y e Z). Copia o código das imagens seguintes para o ficheiro "**matrizes.js**".

```
34
35  /**
36   * @param {float} angulo Ângulo em graus para rodar no eixo do X
37   */
38  function CriarMatrizRotacaoX(angulo)
39  {
40      // Seno e cosseno são calculados em radianos, logo é necessário converter de graus
41      // para radianos utilizando a linha a baixo.
42      var radianos = angulo * Math.PI/180;
43
44      // Matriz final de Rotação no eixo do X
45      return [
46          [1,              0,                          0,                          0],
47          [0,              Math.cos(radianos),         -Math.sin(radianos),        0],
48          [0,              Math.sin(radianos),         Math.cos(radianos),         0],
49          [0,              0,                          0,                          1]
50      ];
52
53  /**
54   * @param {float} angulo Ângulo em graus para rodar no eixo do Y
55   */
56  function CriarMatrizRotacaoY(angulo)
57  {
58      // Seno e cosseno são calculados em radianos, logo é necessário converter de graus
59      // para radianos utilizando a linha a baixo.
60      var radianos = angulo * Math.PI/180;
61
62      // Matriz final de rotação no eixo do Y
63      return [
64          [Math.cos(radianos),    0,          Math.sin(radianos),     0],
65          [0,                     1,          0,                      0],
66          [-Math.sin(radianos),   0,          Math.cos(radianos),     0],
67          [0,                     0,          0,                      1]
68      ];
69  }
70
71  /**
72   * @param {float} angulo Ângulo em graus para rodar no eixo do Z
73   */
74  function CriarMatrizRotacaoZ(angulo)
75  {
76      // Seno e cosseno são calculados em radianos, logo é necessário converter de graus
77      // para radianos utilizando a linha a baixo.
78      var radianos = angulo * Math.PI/180;
79
80      // Matriz final de rotação no eixo do Z
81      return[
82          [Math.cos(radianos),    -Math.sin(radianos),    0,      0],
83          [Math.sin(radianos),    Math.cos(radianos),     0,      0],
84          [0,                     0,                      1,      0],
85          [0,                     0,                      0,      1]
86      ];
87  }
88
```

1.3. Aplicar Transformações Geométricas

Agora que já tens as diferentes funções que criam cada uma das transformações geométricas, vais aprendes a utilizá-las. Para utilizares estas funções é necessário fazeres modificações ao vertexShader para receber a matriz final de transformação. Modifica então o código do vertexShader que se encontra no ficheiro "**shaders.js**" para ficar igual à imagem seguinte adicionando o código assinalado a vermelho.

```
var codigoVertexShader = [
    'precision mediump float;'
    ,
    'attribute vec3 vertexPosition;',
    'attribute vec3 vertexColor;',
    ,
    'varying vec3 fragColor;',
    ,
    // Matrix de 4x4 que indica quais as transformações que devem ser
    // feitas a cada um dos vértices.
    'uniform mat4 transformationMatrix;',
    ,
    'void main(){',
    '    fragColor= vertexColor;',
    '    gl_Position = vec4(vertexPosition, 1.0) * transformationMatrix;',
    '}'
].join('\n');

```

De seguida, no ficheiro "**app.js**" adiciona as seguintes variáveis (tem em atenção que deve ser após a linha de código *`var gpuArrayBuffer = GL.createBuffer();`*).

```
var canvas = document.createElement('canvas');
canvas.width = window.innerWidth - 15;
canvas.height = window.innerHeight - 45;
var GL = canvas.getContext('webgl');
var vertexShader = GL.createShader(GL.VERTEX_SHADER);
var fragmentShader = GL.createShader(GL.FRAGMENT_SHADER);
var program = GL.createProgram();
var gpuArrayBuffer = GL.createBuffer();

// Variável que guarda a localização da variável 'transformationMatrix' do
// vertexShader.
var finalMatrixLocation;

// Variável que guarda a rotação que deve ser aplicada ao objeto.
var anguloDeRotacao = 0;

```

De seguida, na função com o nome "**SendDataToShaders()**", altera a função para ficar igual à imagem seguinte.

```
78
79  function SendDataToShaders() {
80      var vertexPositionAttributeLocation = GL.getAttribLocation(program, "vertexPosition");
81      var vertexColorAttributeLocation = GL.getAttribLocation(program, "vertexColor");
82
83      GL.vertexAttribPointer(
84          vertexPositionAttributeLocation,
85          3,
86          GL.FLOAT,
87          false,
88          6 * Float32Array.BYTES_PER_ELEMENT,
89          0 * Float32Array.BYTES_PER_ELEMENT
90      );
91
92      GL.vertexAttribPointer(
93          vertexColorAttributeLocation,
94          3,
95          GL.FLOAT,
96          false,
97          6 * Float32Array.BYTES_PER_ELEMENT,
98          3 * Float32Array.BYTES_PER_ELEMENT
99      );
100
101     GL.enableVertexAttribArray(vertexPositionAttributeLocation);
102     GL.enableVertexAttribArray(vertexColorAttributeLocation);
103
104     // Guarda a localização da variável 'transformationMatrix' do vertexShader
105     finalMatrixLocation = GL.getUniformLocation(program, 'transformationMatrix');
106
107     // Foi removido o código GL.useProgram(program); e GL.drawArrays(GL.TRIANGLES, 0, 3);
108 }
109
```

NOTA: Não te esqueças do fazer o que te é dito no último comentário da imagem a cima (Apagar as linhas GL.useProgram(progam); e GL.drawArrays(GL.TRIANGLES, 0, 3);)

Agora, vais criar uma nova função com o nome "**loop()**" que permite com que sejam gerados novos frames em permanência de forma a atualizar a cena e permitirá utilizar as matrizes que criaste anteriormente. Copia o seguinte código:

```
function loop ()
{
    // O código abaixo faz resize ao canvas de modo a ajustar-se ao tamanho
    // da página web.
    canvas.width = window.innerWidth - 15;
    canvas.height = window.innerHeight - 100;
    GL.viewport(0,0,canvas.width,canvas.height);

    // É necessário dizer que program vamos utilizar.
    GL.useProgram(program);

    // a cada frame é ncessário limpar os buffers de profundidade e de cor
    GL.clearColor(0.65, 0.65, 0.65, 1.0);
    GL.clear(GL.DEPTH_BUFFER_BIT | GL.COLOR_BUFFER_BIT);

    // Inicialização da variável que guarda a combinação de matrizes que
    // vão ser passadas para o vertexShader.
    var finalMatrix = [
        [1,0,0,0],
        [0,1,0,0],
        [0,0,1,0],
        [0,0,0,1]
    ];

    // A matriz final vai ser igual à multiplicação da matriz de escala com a matriz final.
    // Esta matriz faz uma modificação na escala de 0.25 unidades no eixo do X, 0.25 unidades
    // no eixo do Y e 0.25 unidades no eixo do Z. Quer isto dizer que o objeto irá ficar
    // 4 vezes mais pequeno, sendo que para um objeto ter uma escala normal
    // deverá ter 1 unidade em todos os eixos.
    finalMatrix = math.multiply(CriarMatrizEscala(0.25,0.25,0.25),finalMatrix);

        // A matriz final vai ser igual à multiplicação da matriz de rotação no eixo do Z
    // com a matriz final. Esta matriz faz uma rotação anguloDeRotação unidades
    // no eixo do Y.
    finalMatrix = math.multiply(CriarMatrizRotacaoY(anguloDeRotacao), finalMatrix);

    // A matriz final vai ser igual à multiplicação da matriz de translação com a matriz final.
    // Esta matriz faz uma translação de 0.5 unidades no eixo do X, 0.5 unidades
    // no eixo do Y e 0.0 unidades no eixo do Z.
    finalMatrix = math.multiply(CriarMatrizTranslacao(0.5,0.5, 0), finalMatrix);
```

*continua na próxima página.

```
    // Agora que já temos a matriz final de transformação, temos que converter de 2D array
    // para um array de uma dimensão. Para isso utilizamos o código a baixo.
    var newarray = [];
    for(i = 0; i< finalMatrix.length; i++)
    {
        newarray = newarray.concat(finalMatrix[i]);
    }

    // Depois de termos os array de uma dimensão temos que enviar essa matriz para
    // o vertexShader. Para isso utilizamos o código abaixo.
    GL.uniformMatrix4fv(finalMatrixLocation,false ,newarray);

    // Agora temos que mandar desenhar os triângulos
    GL.drawArrays(
        GL.TRIANGLES,
        0,
        3
    );

    // A cada frame é preciso atualizar o angulo de rotação.
    anguloDeRotacao +=1;

    // O código abaixo indica que no próximo frame tem que chamar
    // a função passada por parametro. No nosso caso é a mesma função
    // criando um loop de animação.
    requestAnimationFrame(loop);
}
```

Depois de copiares o código da função "**loop()**" vais à função "**Start()**" e adicionas a linha de código assinalada a vermelho na imagem abaixo.

```
179
180  function Start() {
181      PrepareCanvas();
182      PrepareShaders();
183      PrepareProgram();
184      PrepareTriangleData();
185      SendDataToShaders();
186
187      // Quando acabar de preparar tudo chama a função de loop.
188      // Ao chamar o loop vai criar um loop de animação.
189      loop();
190  }
```

Depois disto tudo, se abrires o ficheiro "**index.html**" no teu navegador deverás ver o triangulo que criaste no tutorial anterior a rodar sobre o vértice vermelho. Caso não consigas ver esse triângulo, vai à consola do navegar ver qual o erro que está a acontecer.

3. Transformações Geométricas com *ThreeJS*

Com WebGL, para podermos fazer as transformações geométricas, foi necessária a importação de uma biblioteca externa para fazer as operações com matrizes, implementar as matrizes de transformação das transformações geométrica para, por fim, poder aplicar transformações geométricas aos objetos na cena. Depois de tudo implementado, fizemos as seguintes operações com o triângulo:

- Aplicamos uma translação de 0.5 unidades no eixo do X e no eixo do Y;
- Aplicamos uma escala uniforme de 0.25 nos eixos do X, Y e Z;
- Criamos uma rotação contínua no eixo do Y de 1;

De seguida demonstraremos o mesmo processo, só que com recurso a *WebGL*.

3.1. Aplicar Transformações Geométricas

Primeiro, precisamos de criar uma função que permita à aplicação gráfica gerar novos frames de forma contínua – o equivalente à função *loop()* que usamos anteriormente. Para tal, copia o código da imagem abaixo:

```
function loop() {
    //função chamada para gerarmos um novo frame
    renderer.render(cena, camara);

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

Agora, precisamos de chamar a função de *loop()* na função *Start()*. Para isso, adiciona a seguinte linha de código no final da função *Start():*

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

Para aplicar a operação de translação de de 0.5 unidades no eixo do X e no eixo do Y, utiliza o primeiro bloco de código assinalado na imagem a vermelho. Para aplicar a escala uniforme de 0.25 unidades, copia o segundo bloco de código assinalado a vermelho na imagem.

```
var mesh = new THREE.Mesh(geometria, material);

//Definir a operação de translação no nosso triangulo
mesh.translateX(0.5);
mesh.translateY(0.5);

//Definir a operação de escala no nosso triangulo
mesh.scale.set(0.25,0.25,0.25);
```

Para definir a rotação, como é algo que queremos que seja feito a cada frame, temos que fazer essa atualização na nossa função de loop. Para tal, acrescenta o código assinalado na imagem a vermelho à função *loop()*.

```
function loop() {

    //Definir a rotação no eixo do Y.
    //Como o ThreeJS usa Radianos por defeito, temos que converter em graus usando a fórmula Math.PI/180 * GRAUS
    mesh.rotateY(Math.PI/180 * 1);

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

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

Agora, se testares a tua aplicação e esta não tiver erros, podes ver o triângulo a rodar mas que este acaba por desaparecer ao rodar. Isso está relacionado com o volume de visualização: o triângulo está a sair fora do alcance da câmara (iremos abordar com mais detalhe as câmaras no Tutorial 3). Para ultrapassar essa limitação, atualiza os parâmetros da tua câmara tal como ilustrado na imagem abaixo:

```
var camara = new THREE.OrthographicCamera(- 1, 1, 1, - 1, -10, 10);
```