

Tutorial 3 - Câmara Virtual

1. Introdução

Em Computação Gráfica, o processo de visualização é obtido através de uma superfície de visualização onde é desenhada toda a informação relativa a um determinado volume de visualização da cena (*i. e.* espaço do ambiente virtual que é considerado pela câmara virtual). Dadas as características da superfície de visualização e a forma como é obtida, esta pode ser comparada de forma análoga ao modo de funcionamento de uma câmara convencional e, por isso, adota-se a denominação de câmara virtual.

A câmara virtual pode ser parametrizada, sendo que existem os dois seguintes tipos de volume de visualização:

- **Volume de visualização ortogonal:** consiste num paralelepípedo cujo raios projetores são paralelos, sendo assim a profundidade fixa e não existindo o conceito de distância à câmara (Figura 1, à esquerda);
- **Volume de visualização perspetivo:** pode ser associado a uma pirâmide invertida cujo vértice se localiza na posição da câmara. Para delimitar a pirâmide de modo que o volume de visualização não seja infinito, este tipo de volume de visualização especifica o plano de recorte anterior e o plano de corte posterior (Figura 1, à direita). Este espaço delimitado abrangido pelo volume de visualização é também conhecido por *frustum*. Este tipo de volume de visualização é definido ainda pelo campo de visão da câmara virtual (*field-of-view*) e a proporção do mesmo (relação entre a largura e altura da imagem da câmara virtual).

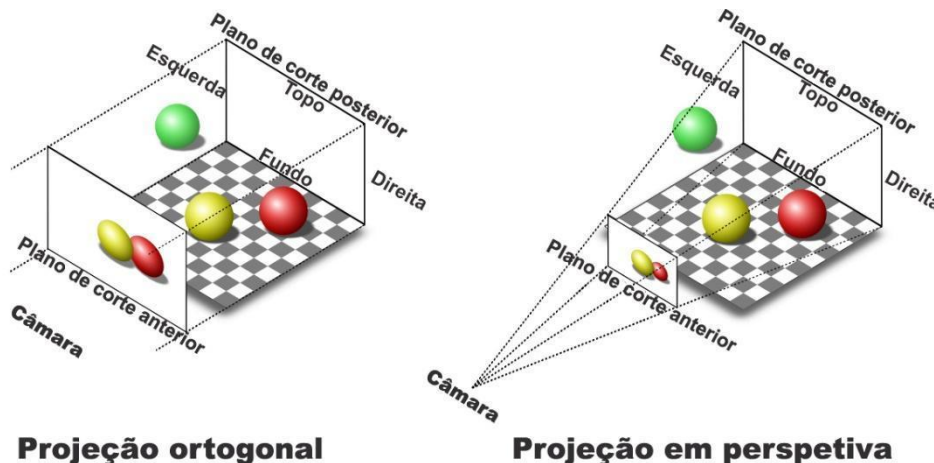


Figura 1 – Ilustração de volume de projeção ortogonal (esquerda) e perspetivo (direita) (adaptado de (Geo-F/X, 2017).

A configuração da câmara virtual é feita, essencialmente, através das seguintes matrizes:

- Matriz de visualização: permite movimentar ou rodar a câmara virtual;
- Matriz de projeção ortográfica: comprime tudo o que se encontra entre o plano anterior e o plano posterior dentro do volume de visualização;
- Matriz de projeção em perspetiva: comprime tudo o que se encontra entre o plano anterior e o plano posterior dentro do volume canónico (área que é renderizada) tendo em conta a distância entre a posição da câmara e a posição do objeto em coordenadas mundo (objetos mais próximos da câmara vão ser maiores que objetos mais longe da câmara);
- Matriz de *viewport*: transforma as coordenadas do *WebGL* (eixos x e y compreendidos entre -1 e 1) em coordenadas do dispositivo (eixos x e y compreendidos entre 0 e 1).

1.1. Objetivos de aprendizagem

Neste tutorial, irás aprender a criar as diferentes matrizes que constituem uma câmara virtual e como criar câmaras virtuais ortogonais e em perspetiva.

2. Câmaras Virtuais em WebGL

2.1. Ficheiros necessários

1. Assim como no tutorial anterior, vamos começar por criar uma nova pasta com o nome “Tutorial 3” e copiar os ficheiros e pastas que estão na pasta “Tutorial 2” para a pasta que acabaste de criar. Apaga todos os comentários (linhas que começam por //) que existem nesses ficheiros para perceberes melhor o que está a ser feito.
2. Agora abre o VSCode (Visual Studio Code), se aparecerem os ficheiros anteriores significa que ele guardou a pasta que estavas a trabalhar anteriormente e precisas de fechar essa pasta. Para fechares essa pasta vai a **File -> Close Folder** e serás levado para a página principal do VSCode. Selecciona “**Open Folder**” e selecciona a pasta “Tutorial 3” que criaste anteriormente.
3. Dentro da pasta “JavaScript” adiciona um novo ficheiro com o nome “camara.js”. Neste ficheiro é onde irás criar os métodos para a criação de matrizes relativas à câmara.

4. Agora, abre o ficheiro “*index.html*” e adiciona as seguintes linhas de código assinaladas a vermelho:

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>
5       WebGL
6     </title>
7   </head>
8   <body onload="Start()">
9     <script src="https://cdn.jsdelivr.net/npm/mathjs@14.0.1/math.js"></script><!-- Biblioteca de matemática -->
10    <script src="./JavaScript/matrizes.js"></script>
11    <!-- Script com as matrizes relativas à câmara -->
12    <script src="./JavaScript/camara.js"></script>
13    <script src="./JavaScript/shaders.js"></script>
14    <script src="./JavaScript/app.js"></script>
15    <p>
16      Estás a visualizar a aplicação gráfica baseada em <b>WebGL</b>.
17      <a href="threejs.html">Clica aqui para veres a aplicação baseada em ThreeJS.</a>
18    </p>
19  </body>
20 </html>

```

2.2. Criação das Matrizes da Câmara

Uma vez que já tens todos os ficheiros necessários, vamos começar por criar as diferentes matrizes pelas quais a câmara virtual é composta. Para isso abre o ficheiro “camara.js” que criaste no passo anterior e que se encontra dentro da pasta “JavaScript”.

Como foi falado na aula teórica, o processo de renderização da câmara envolve diferentes matrizes tais como: matriz de visualização, matrizes de projecção e matriz de *viewport*. A seguir vamos ver para que servem cada uma delas.

Matriz de visualização

A matriz de visualização é a matriz que permite movimentar ou rodar a câmara virtual. Vamos então criar a matriz de visualização. Para isso copia o código da imagem seguinte para o ficheiro “camara.js”.

```

1 /**
2  * Função de devolve a matriz de visualização
3  * @param {float[3]} rightVector Array direita da câmara
4  * @param {float[3]} upVector Array cima da câmara
5  * @param {float[3]} forwardVector Array frente da câmara
6  * @param {float[3]} centerPoint array com a posição da câmara em coordenadas mundo.
7  */
8 function MatrizDeVisualizacao(rightVector, upVector, forwardVector, centerPoint) {
9   return [
10     [rightVector[0], rightVector[1], rightVector[2], -math.multiply(rightVector, centerPoint)],
11     [upVector[0], upVector[1], upVector[2], -math.multiply(upVector, centerPoint)],
12     [forwardVector[0], forwardVector[1], forwardVector[2], -math.multiply(forwardVector, centerPoint)],
13     [0, 0, 0, 1]
14   ];
15 }
16

```

A função que acabaste de criar devolve um array de 2 dimensões com a matriz de visualização tendo em conta os parâmetros de entrada. Esses parâmetros de entrada são os vetores locais referentes à direita, cima e frente da câmara bem como a posição da câmara em coordenadas do mundo.

Matriz de Projeção (Ortográfica)

A matriz de projeção ortográfica é a matriz que comprime tudo o que se encontra entre o plano anterior e o plano posterior dentro do volume canónico (área que é renderizada). Para criares a matriz de projeção ortográfica, copia o código da imagem seguinte para o ficheiro “camara.js”.

```
16
17 /**
18  * Função que devolve a matriz de projeção Ortográfica
19  * @param {float} width Indica qual o comprimento da câmara que deve ser renderizada
20  * @param {float} height Indica qual a altura da câmara que deve ser renderizada
21  * @param {float} nearPlane Indica qual o plano de corte anterior da câmara
22  * @param {float} farPlane Indica qual o plano de corte posterior da câmara
23  */
24 function MatrizOrtografica(width, height, nearPlane, farPlane){
25     var matrizOrtografica = [
26         [1/width, 0, 0, 0],
27         [0, 1/height, 0, 0],
28         [0, 0, 1/((farPlane/2) - nearPlane), -nearPlane/((farPlane/2)-nearPlane)],
29         [0, 0, 0, 1]
30     ];
31
32     return math.multiply(matrizOrtografica, CriarMatrizTranslacao(0,0,-(nearPlane + farPlane / 2)));
33 }
34
```

A função que acabaste de criar devolve um array de 2 dimensões com a matriz de projeção ortográfica tendo em conta os parâmetros de entrada. Esses parâmetros de entrada são o comprimento da câmara, a altura da câmara, o plano de corte anterior e o plano de corte posterior, respetivamente.

NOTA: o *standard* de volumes canónicos tem os seus eixos X e Y compreendidos entre -1 e 1 e o eixo do Z compreendido entre 0 e 1. Isto não acontece em WebGL, uma vez que nesta tecnologia todos os eixos estão compreendidos entre -1 e 1. Foi então necessário ajustar esta matriz para comprimir os objetos entre 0 e 1 em vez de comprimir entre -1 e 1, daí a matriz ser diferente daquela dada na aula.

Matriz de Projeção (Perspetiva)

A matriz de projeção em perspetiva é a matriz que comprime tudo o que se encontra entre o plano anterior e o plano posterior dentro do volume canónico (área que é renderizada) tendo em conta a distância entre a posição da câmara e a posição do objeto em coordenadas mundo (objetos mais próximos da câmara vão ser maiores que objetos mais longe da câmara). Esta matriz tem também em conta qual a distância à qual a imagem vai ser renderizada. Para criares a matriz de projeção em perspetiva, copia o código da imagem seguinte para o ficheiro “camara.js”.

```

34
35 /**
36  * Função que devolve a matriz de projeção em perspectiva
37  * @param {float} distance Distância do centro que a imagem deve ser renderizada
38  * @param {float} width Indica qual o comprimento da câmara que deve ser renderizada
39  * @param {float} height Indica qual a altura da câmara que deve ser renderizada
40  * @param {float} nearPlane Indica qual o plano de corte anterior da câmara
41  * @param {float} farPlane Indica qual o plano de corte posterior da câmara
42  */
43 function MatrizPerspetiva (distance,width, height, nearPlane, farPlane)
44 {
45   ... return [
46     ... [distance/width, ... 0, ... 0, ... 0],
47     ... [0, ... distance/height, ... 0, ... 0],
48     ... [0, ... 0, ... farPlane / (farPlane - nearPlane), ... nearPlane*farPlane/(farPlane - nearPlane)],
49     ... [0, ... 0, ... 1, ... 0]
50   ];
51 }
52

```

A função que acabaste de criar devolve um array de 2 dimensões com a matriz de projeção em perspetiva tendo em conta os parâmetros de entrada. Esses parâmetros de entrada são a distância do plano ao qual os objetos serão projetados (para o mesmo tamanho de câmara, quanto maior a distância maior serão os objetos), o comprimento da câmara, a altura da câmara, o plano de corte anterior e o plano de corte posterior, respetivamente.

Matriz de Viewport

A matriz de *viewport* é a matriz que transforma as coordenadas do webGl (eixos X e Y compreendidos entre -1 e 1) em coordenadas do dispositivo (eixos X e Y compreendidos entre 0 e 1). Para criares a matriz de *viewport* copia o código da imagem seguinte para o ficheiro “camara.js”.

```

52
53 /**
54  * Função que devolve a matriz de Viewport
55  * @param {float} minX Valor mínimo do volume canónico no eixo do X
56  * @param {float} maxX Valor máximo do volume canónico no eixo do X
57  * @param {float} minY Valor mínimo do volume canónico no eixo do Y
58  * @param {float} maxY Valor máximo do volume canónico no eixo do Y
59  */
60 function MatrizViewport(minX, maxX, minY, maxY){
61   ... return [
62     ... [(maxX - minX)/2, ... 0, ... 0, ... (maxX + minX)/2],
63     ... [0, ... (maxY - minY)/2, ... 0, ... (maxY + minY)/2],
64     ... [0, ... 0, ... 1, ... 0],
65     ... [0, ... 0, ... 0, ... 1]
66   ];
67 }
68

```

A função que acabaste de criar devolve um array de duas dimensões com a matriz de *viewport* tendo em conta os parâmetros de entrada especificados. Estes parâmetros são o máximo e mínimo de cada eixo (X e Y entre -1 e 1).

2.3. Utilização das câmaras virtuais

Uma vez que agora tens todas as matrizes que são necessárias para para o bom funcionamento da câmara, é necessário que estas sejam aplicadas a todos os objetos. Para isso é necessário passar essas informações para dentro de cada Shader que exista dentro do programa. Vamos então adaptar o nosso *vertex shader* para receber estas informações. Abre o ficheiro com o nome “shaders.js” e altera o código do *vertex shader* para ficar igual à imagem abaixo.

```
1 var codigoVertexShader = `
2   precision mediump float;
3   ,
4   attribute vec3 vertexPosition;
5   attribute vec3 vertexColor;
6   ,
7   varying vec3 fragColor;
8   ,
9   uniform mat4 transformationMatrix;
10  uniform mat4 visualizationMatrix; // Matriz de Visualização
11  uniform mat4 projectionMatrix; // Matriz de Projeção
12  uniform mat4 viewportMatrix; // Matriz de Viewport
13  ,
14  void main(){
15    fragColor = vertexColor;
16    // Depois de transformação geométrica é necessário multiplicar pelas matrizes de visualização, projeção e viewport.
17    gl_Position = vec4(vertexPosition, 1.0) * transformationMatrix * visualizationMatrix * projectionMatrix * viewportMatrix;
18  }
19 `;
20
```

Agora que o *vertex shader* já está pronto para receber estas informações é necessário irmos ao nosso programa (*app.js*) e mandar essa mesma informação para o *shader*. Vamos então começar por criar três variáveis, uma por cada matriz criada no *vertex shader*, como mostra a imagem abaixo. Estas três variáveis vão guardar a posição de cada uma das respetivas variáveis que se encontram dentro do *vertex shader*.

```
11
12 // Localização da variável 'visualizationMatrix'
13 var visualizationMatrixLocation;
14
15 // Localização da variável 'projectionMatrix'
16 var projectionMatrixLocation;
17
18 // Localização da variável 'viewportMatrix'
19 var viewportMatrixLocation;
20
```

Agora que já temos onde guardar a localização de cada variável, é necessário ir buscar as respetivas posições. Para isso iremos utilizar a mesma função que utilizamos no tutorial anterior e que é demonstrada na imagem abaixo. Copia o código assinalado a vermelho na imagem abaixo para o fim da função “*SendDataToShaders()*”.

```

108 ...finalMatrixLocation = GL.getUniformLocation(program, 'transformationMatrix');
109
110 // Vai buscar qual o localização da variável 'visualizationMatrix' ao vertexShader
111 visualizationMatrixLocation = GL.getUniformLocation(program, 'visualizationMatrix');
112
113 // Vai buscar qual o localização da variável 'projectionMatrix' ao vertexShader
114 projectionMatrixLocation = GL.getUniformLocation(program, 'projectionMatrix');
115
116 // Vai buscar qual o localização da variável 'viewportMatrix' ao vertexShader
117 viewportMatrixLocation = GL.getUniformLocation(program, 'viewportMatrix');
118
119 }

```

Agora é necessário ir à função `loop()` e, utilizando as funções que criaste no início deste tutorial, criares uma câmara virtual. Para isso copia o código assinalado a vermelho das imagens seguintes para debaixo da variável *finalMatrix*, tal como a imagem ilustra.

```

var finalMatrix = [
    [1,0,0,0],
    [0,1,0,0],
    [0,0,1,0],
    [0,0,0,1]
];

//Comentamos esta linha para o triângulo voltar ao tamanho normal
//finalMatrix = math.multiply(CriarMatrizEscala(0.25,0.25,0.25),finalMatrix);

// Mantemos esta linha de código para perceber melhor a rotação em perspectiva
finalMatrix = math.multiply(CriarMatrizRotacaoY(anguloDeRotacao), finalMatrix);

// Comentamos esta linha para o triângulo "voltar" para o centro
//finalMatrix = math.multiply(CriarMatrizTranslacao(0.5,0.5, 0), finalMatrix);

//Criamos uma translação no eixo do Z para que o triângulo fique dentro do volume de visualização
finalMatrix = math.multiply(CriarMatrizTranslacao(0,0,1), finalMatrix);

var newarray = [];
for(i = 0; i< finalMatrix.length; i++)
{
    newarray = newarray.concat(finalMatrix[i]);
}

var visualizationMatrix = MatrizDeVisualizacao([1,0,0],[0,1,0],[0,0,1],[0,0,0]);
var newVisualizationMatrix = [];
for(i = 0; i < visualizationMatrix.length; i++){
    newVisualizationMatrix = newVisualizationMatrix.concat(visualizationMatrix[i]);
}

//Linha responsável pela criação da câmara em perspectiva com os parametros de distância=1,
// comprimento da camera de 4 unidades, altura de 3 unidades, plano anterior de 0.1 unidades e
// plano posterior de 100 unidades.
var projectionMatrix = MatrizPerspetiva(1, 4, 3, 0.1,100);

//Linha responsável pela criação da câmara ortográfica com os parametros de comprimento=4,
//altura=3, plano anterior de 0.1 unidade e plano posterior de 100 unidades
//var projectionMatrix = MatrizOrtografica(4,3,0.1,100);

//NOTA: Uma das linhas de criação tem que estar comentada para não haver conflito. Caso se queira mudar de câmara,
// basta descomentar uma câmara e comentar a outra. Tal como está, a camara em perspectiva é a câmara ativa.

```

**continua na próxima página.*


```
var newprojectionMatrix = [];
for(i = 0; i < projectionMatrix.length; i++){
    newprojectionMatrix = newprojectionMatrix.concat(projectionMatrix[i]);
}

// Utilizando a função MatrizViewport vamos passar os parametros do volume canônico do webGL.
// O volume canônico do webGL tem o valor de x, y e z compreendidos entre -1 e 1.
var viewportMatrix = MatrizViewport(-1,1,-1,1);
var newViewportMatrix = [];
for(i = 0; i < viewportMatrix.length; i++){
    newViewportMatrix = newViewportMatrix.concat(viewportMatrix[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);

GL.uniformMatrix4fv(visualizationMatrixLocation,false ,newVisualizationMatrix);
GL.uniformMatrix4fv(projectionMatrixLocation,false ,newprojectionMatrix);
GL.uniformMatrix4fv(viewportMatrixLocation,false ,newViewportMatrix);

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

Se testares agora o teu jogo, verás um triângulo a rodar em perspetiva.

3. Câmaras Virtuais em *Three.js*

O *ThreeJS* já tem todas as implementações das matrizes relativas à câmara implementadas pelo que podemos ir diretamente para a criação da nossa câmara em perspetiva. Para tal, adiciona as linhas de código assinaladas a vermelho:

```
var renderer = new THREE.WebGLRenderer();

//Linha responsável pela criação da câmara em perspetiva com os parametros de field of view 45,
// aspect ratio de 4 / 3, plano anterior de 0.1 unidades e plano posterior de 100 unidades.
var camaraPerspetiva = new THREE.PerspectiveCamera(45, 4/3, 0.1, 100);
```

Tal como fizemos anteriormente, vamos comentar as operações de escala e de translação que fizemos inicialmente ao triângulo para que ele apareça novamente ao centro com o tamanho inicial, tal como ilustra o código assinalado abaixo.


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

// Comentamos esta linha para o triângulo "voltar" para o centro
//mesh.translateX(0.5);
//mesh.translateY(0.5);

//Comentamos esta linha para o triângulo voltar ao tamanho normal
//mesh.scale.set(0.25,0.25,0.25);

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

//Variável relativa ao ângulo de rotacao
var anguloDeRotacao = 0;
```

Por fim, temos que atualizar a câmara que está a ser renderizada, alterando da câmara ortográfica para a câmara perspectiva. Para tal, atualiza as duas ocorrências de render, tal como mostra a imagem abaixo.

```
function Start(){
    // 0 código abaixo adiciona o triângulo que criamos anteriormente à cena.
    cena.add(mesh);

    //camaraPerspetiva.position.z = 3; //3 mais perto que 4

    render.render(cena, camaraPerspetiva);

    //Função para chamar a nossa função de loop
    requestAnimationFrame(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
    render.render(cena, camaraPerspetiva);

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

Se testares a tua aplicação, deverás conseguir ver o teu triângulo a rodar. Nota que o aspecto visual não é completamente igual ao aspecto visual da aplicação desenvolvida em WebGL. Isto acontece apenas porque as implementações das câmaras em ThreeJS não são completamente iguais às aquelas que definimos em WebGL.

4. Desafios

Sem apagar/modificar nenhuma linha de código resultante da realização dos Tutoriais 1 e 2, implementa as seguintes operações nas duas aplicações (WebGL e ThreeJS):

1. Aplica uma operação de escala, fazendo que o triângulo fique com 50% do tamanho do triângulo original.
2. Aplica uma translação ao triângulo de forma que ele fique a rodar posicionado do lado superior esquerdo e sem sair do canvas.
3. Faz com que o triângulo rode em dois eixos sem que saia do canvas.

ENTREGA

O trabalho deve ser submetido no **MOODLE** até dia **21/03/2025**. A submissão é individual e deve conter a pasta do tutorial com todos os ficheiros necessários à correta execução da aplicação. Existem **duas submissões** de devem ser efetuadas:

- **Submissão 1:** Projeto com o tutorial implementado
- **Submissão 2:** Projeto com o tutorial implementado e com os respetivos desafios