Tutorial 1 – Desenvolvimento das primeiras aplicações gráficas

1. Introdução

O *WebGL* é uma API de baixo nível que permite desenvolver programação gráfica para a *web*. Como tal, o processo de renderização é feito numa página *HTML*, mais concretamente dentro do elemento *HTML canvas.* que permite desenhar informação no navegador de internet através de *JavaScript*. Para tal, nesse *canvas*, é necessário criar um objeto *WebGL* denominado de *context* que serve de interface com os comandos de renderização do *WebGL* e com o *buffer* de desenho.

Em computação gráfica, todos os elementos desenhados no ecrã são controlados por funções denominadas de *shaders*, as quais são responsáveis por processar todos os dados gráficos a serem desenhados. Existem dois tipos de *shaders*: o *vertex shader* e o *fragment shader*. O *vertex shader* é responsável pelo cálculo das posições dos vértices e pela rasterização de primitivas (por exemplo, pontos, linhas ou triângulos). Já o *fragment shader* é responsável pela computação da cor de cada pixel da primitiva que está a ser rasterizada. Aquando da utilização de *shaders*, existem quatro formas de receber dados:

- ***Buffers* e *Attributes***: os *buffers* são *arrays* de dados que são enviados para o GPU contendo diversa informação como posições, normais, coordenadas de texturas ou cores de vértices. Os *Attributes* especificam como (e de que forma) os dados do *buffer* são utilizados;

- ***Uniforms***: variáveis globais que são definidas antes de executar os *shaders*;

- ***Textures***: *arrays* de dados, geralmente baseados em dados de imagem;

- ***Varyings***: permitem passar dados entre o *vertex shader* e o *fragment shader*.

Em *WebGL*, a compilação dos *shaders* é feita através do objeto *program*.

1.1. Objetivos de aprendizagem

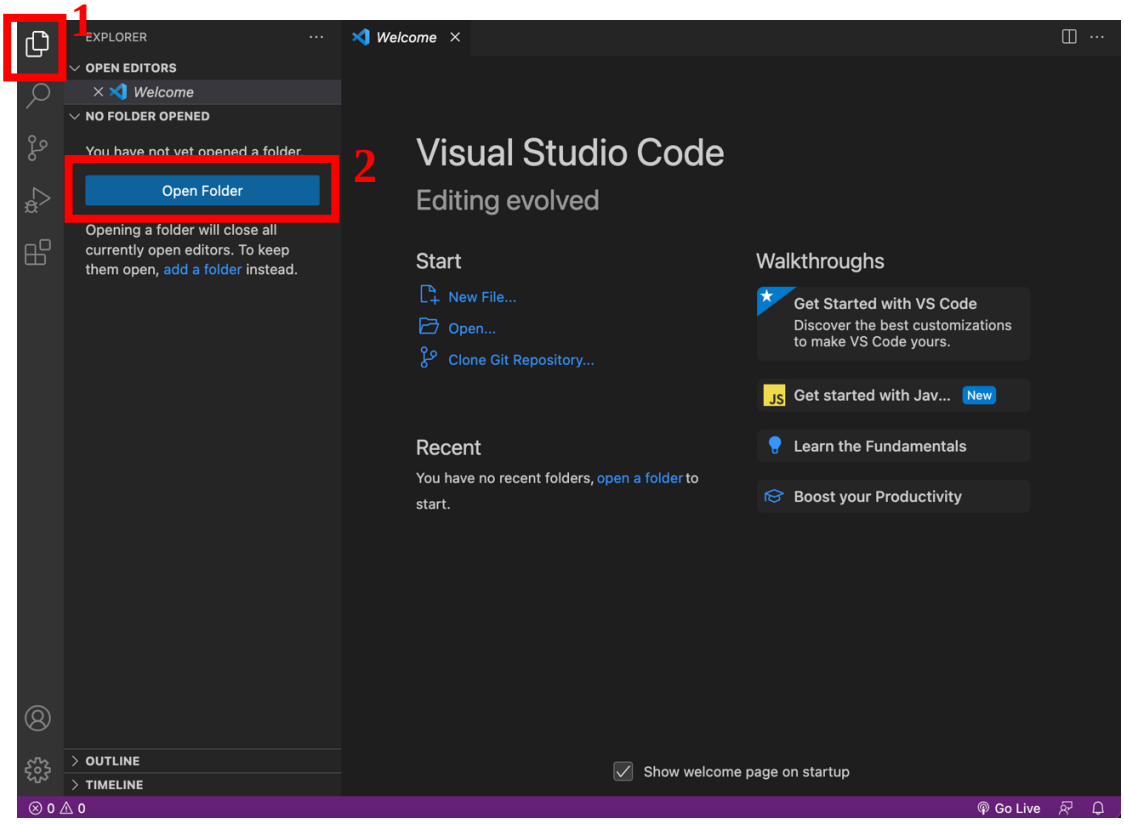
O objetivo deste capítulo é o de providenciar conhecimentos acerca dos recursos e das funções base necessárias para a criação aplicações baseadas na tecnologia *WebGL*. Mais concretamente, aprenderás a criar os ficheiros base ficheiros necessários para desenvolver uma aplicação baseada em *WebGL*, incluindo a criação de *canvas*, *shaders* e *programs*. Para além disso, aprenderás também a criar uma aplicação baseada em *Three.JS*.

2. Primeira aplicação gráfica baseada em *WebGL*

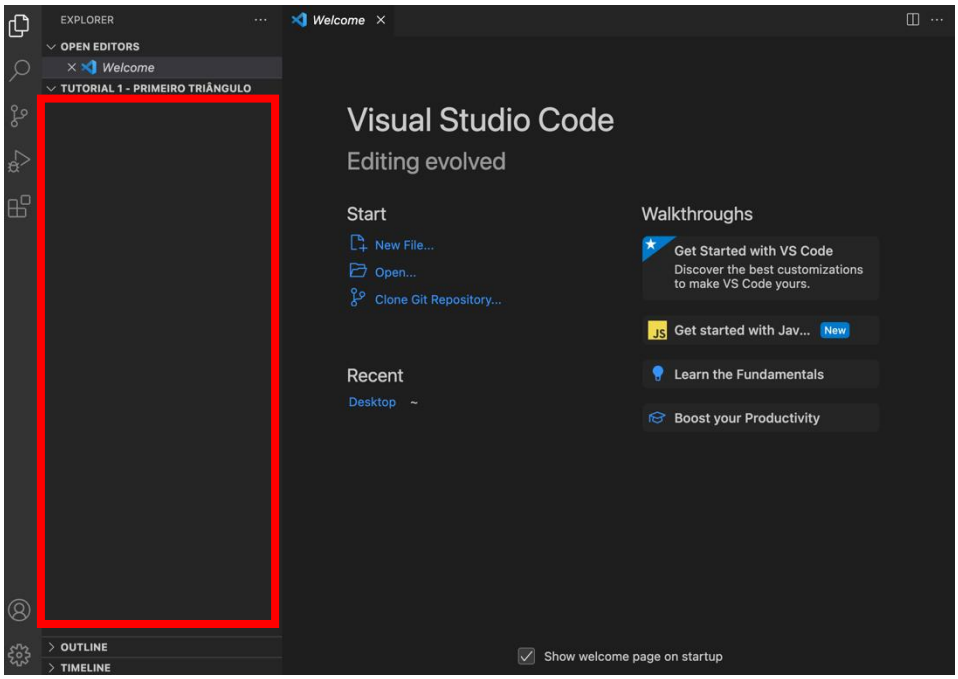
2.1. Ficheiros necessários

Agora que já tens o software necessário instalado no teu PC, vais criar todos os ficheiros necessários para desenvolveres a tua primeira aplicação gráfica baseada em *WebGL*. Para tal, segue os passos seguintes:

1. No *VSCode*, abre a pasta que criaste no tutorial anterior (provavelmente com o nome "WebGL - Tutoriais"). A imagem abaixo ilustra este processo. Depois de aberta, clica no segundo ícone junto do nome da pasta para criar uma nova pasta com o nome "*Tutorial 1 – Primeiro Triangulo*". Abre a pasta que acabaste de criar. Caso apareça algum aviso de segurança, confirma.



2. De seguida, carrega com o botão do lado esquerdo do rato na área que está assinalada a vermelho na figura abaixo. Nesse menu de contexto, clica com o botão direito do rato na pasta relativa ao Tutorial 1 e seleciona "*New File*", dando o nome de "*index.html*" ao ficheiro. Nessa mesma pasta, cria outro ficheiro, agora com o nome de "*threejs.html*". Agora faz o mesmo procedimento, mas, em vez de adicionares um ficheiro, cria uma pasta (seleciona "*New Folder*") e dá-lhe o nome de "*JavaScript*". Carregando com o botão do lado direito em cima da pasta que acabaste de criar vais criar dois ficheiros, um com o nome "*app.js*" e outro com o nome "*shaders.js*".



2.2.Criação da primeira aplicação baseada em WebGL

Agora que já tens o software necessário instalado no teu computador, vais adicionar código aos ficheiros criados para desenvolveres a tua primeira aplicação gráfica baseada em *WebGL*. Para isso, segue os passos seguintes:

1. Vamos começar pelo ficheiro "*index.html*" e criar uma simples página *web*. Assim, abre o ficheiro "*index.html*" e adiciona o seguinte código[1][2], salvando depois o ficheiro:

```
<!DOCTYPE html>
<html>
    <head>
        <title>
            WebGL
        </title>
    </head>
        <body onload="Start()">
            <!--Scripts que correrão na página -->
            <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>
</html>
```

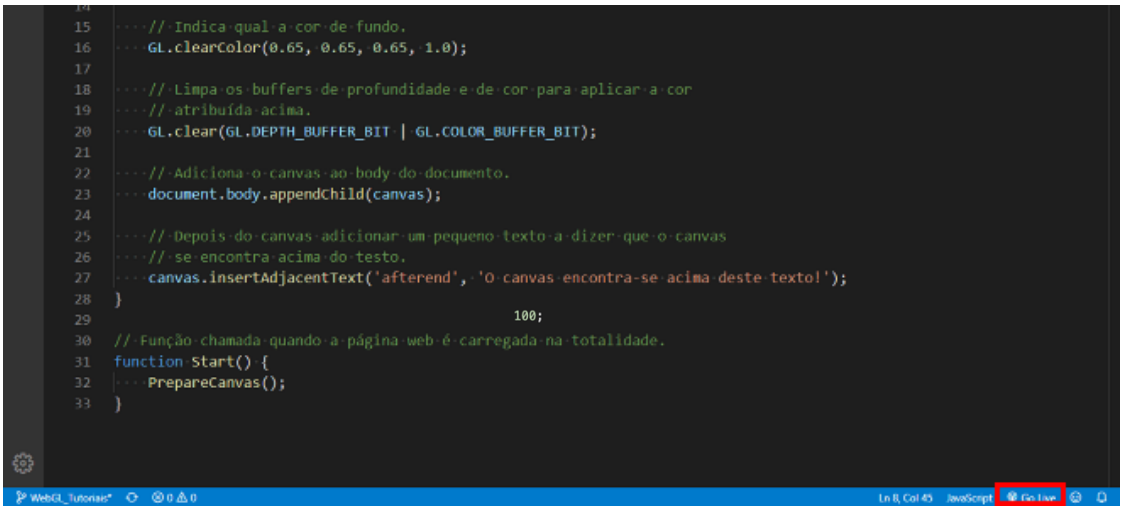
[1] No código, a *tag* "*body*" tem um parâmetro "*onload*", que indica qual a função que deverá chamar quando terminar de carregar a página.

[2] No código, a *tag* "*script*" indica que os scripts que se encontram no caminho passado pelo parâmetro "*src*" serão utilizados nesta página.

2. De seguida abre o ficheiro "*app.js*" e introduz o seguinte código e salva o teu ficheiro no final:

```
// A primeira coisa que é necessário é um elemento HTML do tipo canvas
var canvas = document.createElement('canvas');

// Em primeiro lugar temos que especificar qual o tamanho do canvas.
// O tamanho do canvas vai ser o tamanho da janela (window).
canvas.width = window.innerWidth - 15;
canvas.height = window.innerHeight - 100;

// Para podermos trabalhar sobre WebGl é necessário termos a Biblioteca Gráfica
// (GL significa Graphic Library)
var GL = canvas.getContext('webgl');

// Função responsável por preparar o canvas.
function PrepareCanvas() {

    // Indica qual a cor de fundo.
    GL.clearColor(0.65, 0.65, 0.65, 1.0);

    // Limpa os buffers de profundidade e de cor para aplicar a cor
    // atribuída acima.
    GL.clear(GL.DEPTH_BUFFER_BIT | GL.COLOR_BUFFER_BIT);

    // Adiciona o canvas ao body do documento.
    document.body.appendChild(canvas);

    // Depois do canvas adicionar um pequeno texto a dizer que o canvas
    // se encontra acima do testo.
    canvas.insertAdjacentText('afterend', 'O canvas encontra-se acima deste texto!');
}

// Função chamada quando a página web é carregada na totalidade.
function Start() {
    PrepareCanvas();
}

```

Agora que copiaste o código acima, podes ver a página *web*. Tens duas maneiras de ver a página web. Uma das formas é ires à pasta do tutorial e fazeres duplo clique no ficheiro "*index.html*". A segunda forma (e a mais aconselhada) é através do *plugin* que instalaste, carregando no botão com o nome "*Go Live*" assinalado a vermelho na figura abaixo (ou pelo atalho *Alt+L+O* ou *CMD+L+O*). Este *plugin* irá criar um servidor de HTTP local e, para além disso, quando guardares algum ficheiro, a página web recarregará automaticamente para refletir as alterações feitas. Saberás que o servidor está ligado quando aparecer "*Port:5500*" (ou outro número) no lugar do botão.



Ao abrir a página *web*, deverás ver uma caixa cinzenta e, no fundo da página, o texto a indicar que o *canvas* está acima desse texto. Caso não apareça nada, abre as ferramentas do programador no *browser* (p. ex., no *Chrome* carrega na tecla *F12* e na aba "*Console*") e vê se é identificado algum erro. Se aparecer algum erro, verifica a mensagem de erro e certifica-te que os passos anteriores estão corretos. Se não aparecer nada, recarrega a página só para confirmar que não existem erros.

3. Depois de teres testado a página *web* de forma a garantir que não tens erros, abre o ficheiro "*shaders.js*" no *VSCode* e copia o seguinte bloco de código:

```
// Código corresponte ao vertex shader
var codigoVertexShader = [
    'precision mediump float;' // indica qual a precisão do tipo float
    ,
    // Variável read-only do tipo vec3 que indicará a posição de um vértice
    'attribute vec3 vertexPosition;',
    // Variável read-only do tipo vec3 que indicará a cor de um vértice
    'attribute vec3 vertexColor;',
    ,
    // Variável que serve de interface entre o vertex shader e o fragment shader
    'varying vec3 fragColor;',
    ,

    'void main(){',
    // Dizemos ao fragment shader qual a cor do vértice.
    '    fragColor= vertexColor;',
    // gl_Position é uma variável própria do Shader que indica a posição do vértice.
    // Esta variável é do tipo vec4 e a variável vertexPosition é do tipo vec3.
    // Por esta razão temos que colocar 1.0 como último elemento.
    '    gl_Position = vec4(vertexPosition, 1.0);',
    '}'
].join('\n');

// Código corresponte ao fragment shader
var codigoFragmentShader = [
    'precision mediump float;'// indica qual a precisão do tipo float
    ,
    // Variável que serve de interface entre o vertex shader e o fragment shader
    'varying vec3 fragColor;',
    ,
    'void main(){',
    // gl_FragColor é uma variável própria do Shader que indica qual a cor do vértice
    // Esta variável é do tipo vec4 e a variável fragColor é do tipo vec3.
    // Por esta razão temos que colocar 1.0 como último elemento.
    '    gl_FragColor = vec4(fragColor, 1.0);',
    '}'
].join('\n');
```

4. O próximo passo é criar *shaders* que utilizem o código acima. Abre o ficheiro "*app.js*" e adiciona as variáveis relativas aos *shaders* debaixo da definição da variável "*GL*", tal como ilustrado na imagem abaixo.

```
// Para podermos trabalhar sobre WebGl é necessário termos a Biblioteca Gráfica
// (GL significa Graphic Library)
var GL = canvas.getContext('webgl');

// Criar o vertex shader. Este shader é chamado por cada vértice do objeto
// de modo a indicar qual a posição do vértice.
var vertexShader = GL.createShader(GL.VERTEX_SHADER);

// Criar o fragment shader. Este shader é chamado para todos os píxeis do objeto
// de modo a dar cor ao objeto.
var fragmentShader = GL.createShader(GL.FRAGMENT_SHADER);
```

5. Depois de criadas as variáveis, é necessário dizeres qual o código que os *shaders* irão utilizar. Para isso vais criar uma função para preparar os *shaders*. Antes da função "*Start()*", copia a função "PrepareShaders()" que se encontra abaixo:

```
40
41  // Função responsável por preparar os shaders.
42  function PrepareShaders()
43  {
44      // Atribui o código que está no ficheiro "shaders.js" ao vertexShader.
45      GL.shaderSource(vertexShader, codigoVertexShader);
46
47      // Atribui o código que está no ficheiro "shaders.js" ao fragmentShader.
48      GL.shaderSource(fragmentShader, codigoFragmentShader);
49
50      // Esta linha de código compila o shader passado por parâmetro.
51      GL.compileShader(vertexShader); // Compila o vertexShader.
52      GL.compileShader(fragmentShader); // Compila o fragmentShader.
53
54      // Depois de compilado os shaders é necessário verificar se ocurreu algum erro
55      // durante a compilação. Para o vertex shader usamos o código abaixo.
56      if(!GL.getShaderParameter(vertexShader, GL.COMPILE_STATUS)){
57          console.error("ERRO :: A compilação do vertex shader lançou uma excepção!",
58                        GL.getShaderInfoLog(vertexShader));
59      }
60
61      // Depois de compilado os shaders é necessário verificar se ocurreu algum erro
62      // durante a compilação. Para o fragment shader usamos o código abaixo.
63      if(!GL.getShaderParameter(fragmentShader, GL.COMPILE_STATUS)){
64          console.error("ERRO :: A compilação do fragment shader lançou uma excepção!",
65                        GL.getShaderInfoLog(fragmentShader));
66      }
67  }
```

6. Na função "*Start()*" adiciona a seguinte linha de código depois da chamada da função "*PrepareCanvas()"*:

```
69  // Função chamada quando a página web é carregada na totalidade.
70  function Start() {
71      PrepareCanvas();
72      PrepareShaders();
73  }
74
```

7. Os *shaders* por si só não são suficientes para que a GPU os utilize, sendo necessário criar programas que usem esses *shaders*. Para isso, vais criar uma variável que guarde o programa que irá usar os *shaders* utilizando o código assinalado a vermelho na imagem seguinte.

```
8   // Criar o vertex shader. Este shader é chamado por cada vértice do objeto
9   // de modo a indicar qual a posição do vértice.
10  var vertexShader = GL.createShader(GL.VERTEX_SHADER);
11
12  // Criar o fragment shader. Este shader é chamado para todos os píxeis do objeto
13  // de modo a dar cor ao objeto.
14  var fragmentShader = GL.createShader(GL.FRAGMENT_SHADER);
15
16  // Criar o programa que utilizará os shaders.
17  var program = GL.createProgram();
18
```

8. De seguida vais acrescentar uma função que tem como objetivo atribuir e verificar se o programa é válido para ser executado na GPU. Copia a seguinte função "*PrepareProgram()*":

```
68
69  // Função reponsável por preparar o Programa que irá correr sobre a GPU
70  function PrepareProgram(){
71      // Depois de teres os shaders criados e compilados é necessário dizeres ao program
72      // para utilizar esses mesmos shaders. Para isso utilizamos o código seguinte.
73      GL.attachShader(program, vertexShader);
74      GL.attachShader(program, fragmentShader);
75
76      // Agora que já atribuiste os shaders, é necessário dizeres à GPU que acabaste de
77      // configurar o program. Uma boa prática é verificar se existe algum erro no program
78      GL.linkProgram(program);
79      if(!GL.getProgramParameter(program, GL.LINK_STATUS)){
80          console.error("ERRO ::: O linkProgram lançou uma excepção!", GL.getProgramInfoLog(program));
81      }
82
83      // É boa prática verificar se o programa foi conectado corretamente e se pode ser
84      // utilizado.
85      GL.validateProgram(program);
86      if(!GL.getProgramParameter(program, GL.VALIDATE_STATUS)){
87          console.error("ERRO ::: A validação do program lançou uma excepção!", GL.getProgramInfoLog(program));
88      }
89
90      // Depois de tudo isto, é necessário dizer que queremos utilizar este program. Para isso
91      // utilizamos o seguinte código
92      GL.useProgram(program);
93  }
```

9. Na função "*Start()*" adiciona a seguinte linha de código depois da chamada da função *PrepareShaders()*:

```
96  // Função chamada quando a página web é carregada na totalidade.
97  function Start() {
98      PrepareCanvas();
99      PrepareShaders();
100     PrepareProgram();
101 }
```

10. Agora que já tens o programa a utilizar os *shaders*, é necessário criares uma variável que guarde o *buffer* da GPU para onde vais mandar os dados. Para isso utiliza o código assinalado a vermelho na imagem abaixo.

```
15
16  // Criar o programa que utilizará os shaders.
17  var program = GL.createProgram();
18
19  // Criar um buffer que está localizado na GPU para receber os pontos que
20  // os shaders irão utilizar.
21  var gpuArrayBuffer = GL.createBuffer();
22
```

11. Depois de criares o *buffer*, vais criar uma função que tem como objetivo guardar a posição (*x*,*y*,*z*) de cada vértice bem como a cor *RGB* de cada ponto. Para tal, copia a função "*PrepareTriangleData()*" que se segue:

```
// Função resposável por criar guardar a posição XYZ e cor RBG de cada um dos vértices do triângulo.
// Esta função é também responsável por copiar essa mesma informação para um buffer que se encontra na GPU.
function PrepareTriangleData(){
    // Variável que guardará os pontos de cada vértice (XYZ) bem como a cor de cada um deles (RGB)
    // Nesta variável, cada vértice é constituída por 6 elementos, lembra-te disso para os passos a seguir.
    // Outra coisa que te deves saber é que a área do canvas vai de -1 a 1 tanto em altura como em largura
    // com centro no meio da área do canvas. O código RBG tem valores compreendidos entre 0.0 e 1.0
    var triangleArray = [
        //  X      Y      Z      R      G      B
            -0.5,  -0.5,  0.0,   1.0,   0.0,   0.0,//  Vértice 1 da "imagem" ao lado ->      /\
             0.5,  -0.5,  0.0,   0.0,   1.0,   0.0,//  Vértice 2 da "imagem" ao lado ->     /  \
             0.0,   0.5,  0.0,   0.0,   0.0,   1.0 //  Vértice 3 da "imagem" ao lado ->    1----2
    ];

    // Esta linha de código indica à GPU que o gpuArrayBuffer é do tipo ARRAY_BUFFER
    GL.bindBuffer(GL.ARRAY_BUFFER,gpuArrayBuffer);

    // Esta linha de código copia o array que acabamos de criar (triangleArray)
    // para o buffer que está localizado na GPU (gpuArrayBuffer).
    GL.bufferData(
        // Tipo de buffer que estámos a utilizar.
        GL.ARRAY_BUFFER,
        // Dados que pretendemos passar para o buffer que se encontra na GPU.
        // Importante saber que no CPU os dados do tipo float utilizam 64bits mas a GPU só trabalha com
        // dados de 32bits. O JavaScript permite-nos converter floats de 64bits para floats de 32bits utilizando
        // a função a baixo.
        new Float32Array(triangleArray),
        // Este parâmetro indica que os dados que são passados não vão ser alterados dentro da GPU.
        GL.STATIC_DRAW
        );
}
```

12. Na função "*Start()*" adiciona a seguinte linha de código depois da chamada da função "*PrepareProgram()"*:

```
// Função chamada quando a página web é carregada na totalidade.
function Start() {
    PrepareCanvas();
    PrepareShaders();
    PrepareProgram();
    PrepareTriangleData();
}
```

13. De seguida, vais passar os dados que estão no *buffer* para o *vertex shader* de modo que estes sejam desenhados. Para isso cria uma função com o nome de "*SendDataToShaders()*" que é definida pelo seguinte bloco de código:

```
// Esta função é responsável por pegar na informação que se encontra no gpuArrayBuffer
// e atribuí-la ao vertex shader.
function SendDataToShaders(){
    // A primeira coisa que é necessário fazer é ir buscar a posição de cada umas das variáveis dos Shaders.
    // Se verificares o código dos shaders, é necessário passar informação para duas variáveis (vertexPosition
    // e vertexColor). Para isso vamos utilizar o código abaixo.
    var vertexPositionAttributeLocation = GL.getAttribLocation(program, "vertexPosition");
    var vertexColorAttributeLocation = GL.getAttribLocation(program, "vertexColor");

    // Esta função utiliza o último buffer que foi feito binding. Como podes ver pela função anterior
    // o último buffer ao qual foi feito bind foi o gpuArrayBuffer, logo ele vai buscar informação a esse
    // buffer e inserir essa informação no vertex shader. Vamos inserir os dados para a veriável vertexPosition.
    GL.vertexAttribPointer(
        // Localização da variável na qual pretendemos inserir a informação. No nosso caso a variável
        // "vertexPosition"
        vertexPositionAttributeLocation,
        // Este parâmetro indica o quantos elementos vão ser usados pela variável. No nosso caso, a variável
        // que irá utilizar estes valores é do tipo vec3 (XYZ) logo são 3 elementos.
        3,
        // Este parâmetro indica qual é o tipo dos objetos que estão nesse buffer. No nossa caso são FLOATs.
        GL.FLOAT,
        // Este parâmetro indica se os dados estão ou não normalizados. Para já este parametro pode ser false.
        false,
        // Este parametro indica qual o tamanho de objetos que constituem cada ponto do triângulo em bytes.
        // Cada ponto do triângulo é constituído por 6 valores (3 para posição X Y Z e 3 para a cor R G B) e
        // o array que está no buffer e do tipo Float32Array. Float32Array tem uma propriedade que indica
        // qual o número de bytes que cada elemento deste tipo usa. Basta multiplicar 3 pelo numero de
        //bytes de um elemento.
        6 * Float32Array.BYTES_PER_ELEMENT,
        // Este parâmetro indica quando elementos devem ser ignorados no ínicio para chegar aos valores que
        // pretendemos utilizar. No nosso caso queremos utilizar os primeiros 3 elementos. Este valor também
        // é em bytes logo multiplicamos pelo número de bytes de um Float32Array.
        0 * Float32Array.BYTES_PER_ELEMENT
    );

    // Agora utilizando o mesmo método acima, vamos inserir os dados na variável vertexColor.
    // Se prestares atenção nos parâmetros desta função, é bastante parecido ao método anterior, mudando apenas
    // a variável à qual prentendemos inserir os dados (vertexColor) e o último parâmetro (uma vez que agora
    // pretendemos ignorar os primeiros 3 valores que significam a posição de cada vértice)
    GL.vertexAttribPointer(
        // Localização da variável na qual pretendemos inserir a informação. No nosso caso a variável
        // "vertexPosition"
        vertexColorAttributeLocation,
        // Este parâmetro indica o quantos elementos vão ser usados pela variável. No nosso caso, a variável
        // que irá utilizar estes valores é do tipo vec3 (XYZ) logo são 3 elementos.
        3,
        // Este parâmetro indica qual é o tipo dos objetos que estão nesse buffer. No nossa caso são FLOATs.
        GL.FLOAT,
        // Este parâmetro indica se os dados estão ou não normalizados. Para já este parametro pode ser false.
        false,
        // Este parametro indica qual o tamanho de objetos que constituem cada ponto do triângulo em bytes.
        // Cada ponto do triângulo é constituído por 6 valores (3 para posição X Y Z e 3 para a cor R G B) e
        // o array que está no buffer e do tipo Float32Array. Float32Array tem uma propriedade que indica
        // qual o número de bytes que cada elemento deste tipo usa. Basta multiplicar 3 pelo numero de
        //bytes de um elemento.
        6 * Float32Array.BYTES_PER_ELEMENT,
        // Este parâmetro indica quando elementos devem ser ignorados no ínicio para chegar aos valores que
        // pretendemos utilizar. No nosso caso queremos utilizar os primeiros 3 elementos. Este valor também
        // é em bytes logo multiplicamos pelo número de bytes de um Float32Array.
        3 * Float32Array.BYTES_PER_ELEMENT
    );

    // Agora é necessário ativar os atributos que vão ser utilizados e para isso utilizamos a linha seguinte.
    // Temos de fazer isso para cada uma das variáveis que pretendemos utilizar.
    GL.enableVertexAttribArray(vertexPositionAttributeLocation);
    GL.enableVertexAttribArray(vertexColorAttributeLocation);

    // Indica que vais utilizar este programa
    GL.useProgram(program);
```

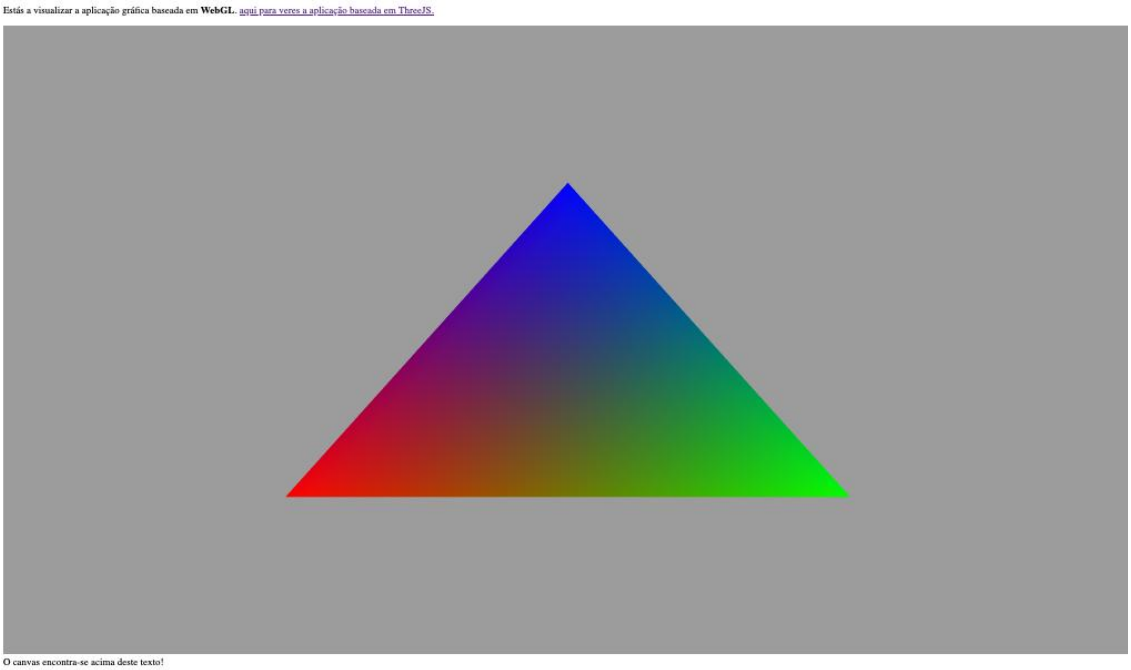
(continua na próxima página)

```
// Indica à GPU que pode desenhar
GL.useProgram(program);

GL.drawArrays(
    GL.TRIANGLES,   //Parâmetro que indica o tipo de objetos que pretendes desenhar
    0,              // Qual o primeiro elemento que deve ser desenhado (elemento na posição 0)
    3               //Quantos elementos devem ser desenhados
    );

}
```

14. Por último, deves adicionar uma chamada da função *SendDataToShaders()* à função de "*Start()*" como mostra a imagem abaixo.

```
// Função chamada quando a página web é carregada na totalidade.
function Start() {
    PrepareCanvas();
    PrepareShaders();
    PrepareProgram();
    PrepareTriangleData();
    SendDataToShaders();
}
```

De forma a conferires o resultado final, vai ao teu browser e atualiza a página (caso estejas a utilizar o *Live server*, não será necessário atualizar a página pois a aplicação atualiza a página em tempo real). O resultado final deverá ser semelhante ao da imagem abaixo.



3. Primeira aplicação gráfica baseada em *ThreeJS*

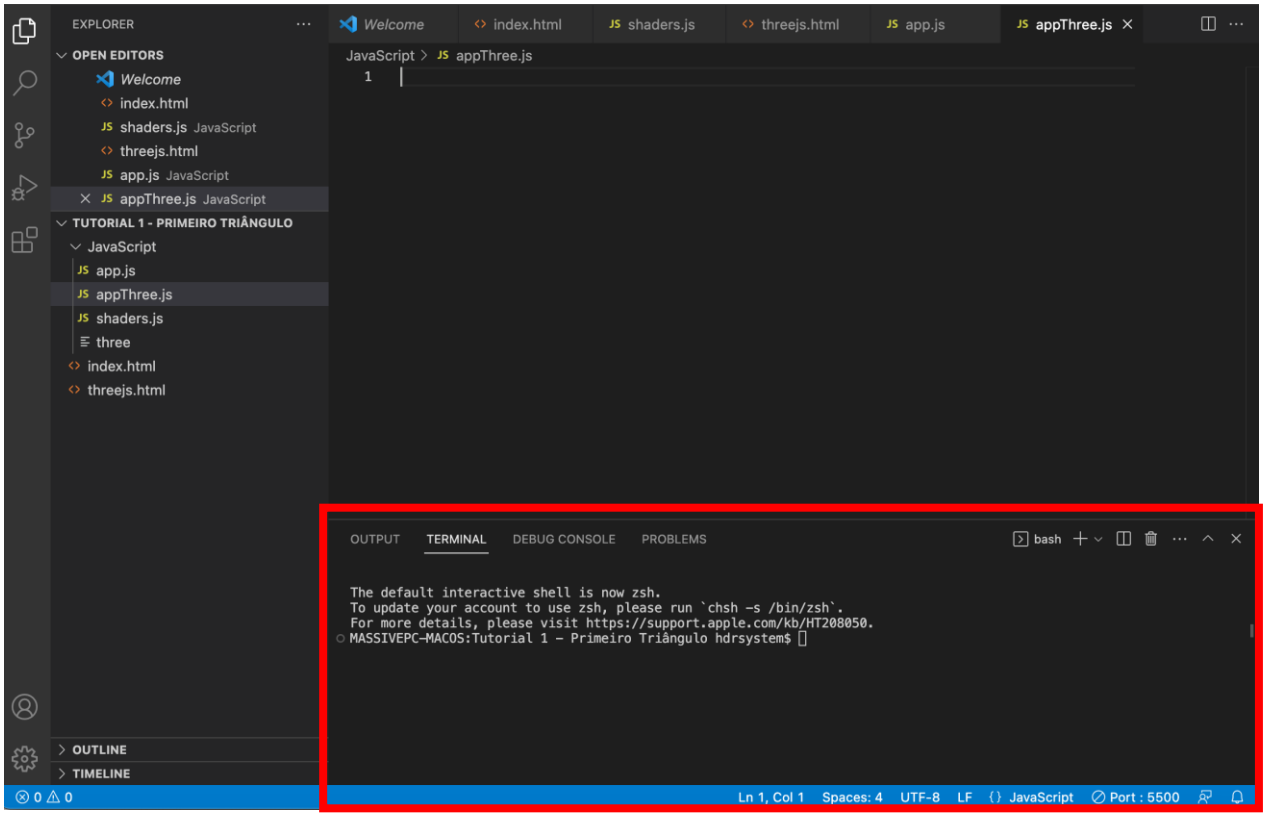
Na seção anterior, criaste a tua primeira aplicação gráfica com recurso à linguagem de baixo nível WebGL. No entanto, existem bibliotecas que permitem simplificar a programação em WebGL, permitindo assim uma maior facilidade no desenvolvimento de aplicações. Esta seção foca-se na utilização de uma das mais conhecidas bibliotecas de *WebGL*, a biblioteca *ThreeJS*.

3.1. Ficheiros necessários

1. Dentro da pasta relativa ao Tutorial 1, cria o ficheiro "*threejs.html*".

2. Dentro da pasta "*JavaScript*", cria o ficheiro "*appThree.js*"

Neste ponto, já tens todos os ficheiros necessários para utilizares a biblioteca *ThreeJS*, mas ainda não tens instalado o *IntelliSense* (que te confere autocomplete/ajudas). Para instalares o *IntelliSense*, terás de te dirigir ao terminal do VSCode (assinalado a vermelho na Figura 2). Caso o terminal não esteja aberto, vai ao menu Terminal → New Terminal ou através do atalho CTRL + SHIFT + Ç.



3. De seguida escreve os seguintes comandos[3]:
   - *npm install typings --global* (para instalares um plugin, se usares Mac ou Linux coloca sudo antes do termo npm)
   - *typings init* (para inicializares o plugin)[4]
   - *typings install dt~three --save --global* (instala a biblioteca de ajuda do *ThreeJS*, se usares Mac ou Linux coloca *sudo* antes do termo *npm*). *Nota*: se der erro, usa o comando *typings install three --save –global*

4. No final da execução destes comandos, deverás adicionar um novo ficheiro à raiz do projeto com o nome "*jsconfig.json*" e copiar o seguinte código para esse novo ficheiro:

```
{
    "typeAcquisition": {
        "include": [
            "three"
        ]
    }
}
```

A partir deste momento já deverás ter *IntelliSense* para te ajudar a escrever código sobre a biblioteca *ThreeJS.*

3.2. Criação da primeira aplicação baseada em *ThreeJS*

1. Abre o ficheiro "threejs. html" localizado na raiz da pasta relativa ao Tutorial 1 e transcreve o seguinte código:

[3] não copies o que está entre parêntesis "()", isto apenas serve para explicar o que faz cada comando e, no final de cada comando, pressiona *enter*).

[4] Se ocorrer erros na execução deste comando, abre a powershell com permissões de administrador e executa o seguinte comando: Set-ExecutionPolicy RemoteSigned . Tenta novamente e, quando acabares de correr os comandos do ponto 4 do tutorial, executa o comando Set-ExecutionPolicy Restricted de forma que o computador não fique exposto.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>
            WebGL - Three.js
        </title>
        <script type="importmap">
            {
              "imports": {
                "three": "https://cdn.jsdelivr.net/npm/three@0.173.0/build/three.module.js",
                "three/addons/": "https://cdn.jsdelivr.net/npm/three@0.173.0/examples/jsm/"
              }
            }
          </script>
          <script src="./JavaScript/appThree.js" type="module"></script>
    </head>
    <body>
        <p>
            Estás a visualizar a aplicação gráfica baseada em <b>ThreeJS</b>.
            <a href="index.html">Clica aqui para veres a aplicação baseada em WebGL.</a>
        </p>
    </body>
</html>
```

2. Abre o ficheiro "*appThree.js*" e transcreve o seguinte código:

```javascript
// Importa a biblioteca ThreeJS baseado no ImportMap do ficheiro threejs.html
import * as THREE from 'three';

// Indica ao documento HTML que quando acabar de carregar todo o seu conteúdo
// deve chamar a função "Start".
document.addEventListener('DOMContentLoaded', Start);

// Em three.js tudo é baseado em cenas e camaras. Cada cena contém os objetos que a ela pertencem.
// Podem existir diferentes camaras mas apenas uma é rederizada.
// As linhas de código abaixo criar uma cena, uma camara e um render em WebGL.
//Vamos ver nas próximas aulas os diferentes tipos de câmara, por agora usamos a camara ortográfica.
// Este último é o que vai renderizar a imagem tendo em conta a camâma e a cena.
var cena = new THREE.Scene();
var camara = new THREE.OrthographicCamera(- 1, 1, 1, - 1, 0, 10);
var renderer = new THREE.WebGLRenderer();

// O código abaixo indica ao render qual o tamanho da janela de visualização
renderer.setSize(window.innerWidth -15, window.innerHeight-80);

// O código abaixo indica ao render qual a cor de fundo da janela de visualização
renderer.setClearColor(0xaaaaaa);

// O código abaixo adiciona o render ao body do documento html para que este possa ser visto.
document.body.appendChild(renderer.domElement);

// Para criarmos um objeto precisamos sempre de uma geometria e um material, o primeiro é
// responsável por definir a geometria (ou vértices de cada ponto), e o segundo é responsável
// por dizer qual o material que o objeto irá usar.

// Para criar um triângulo, é necessário criar a geometria para isso utilizamos o código abaixo indicando
// quais as posições de cada um dos vértices do triângulo.
var geometria = new THREE.BufferGeometry();
var vertices = new Float32Array( [
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0,
    0.0, 0.5, 0.0
] );
```

(continua na próxima página)

```
35
36  //De forma a definir a cor para cada um dos vértices, temos que criar uma matriz com os valores RGB para
37  //cada um deles
38  const cores = new Float32Array( [
39      1.0, 0.0,  0.0,
40      0.0, 1.0,  0.0,
41      0.0, 0.0,  1.0,
42  ] );
43
44  // itemSize = 3 pois são 3 valores (componentes x,y,z para a posição e RGB para a cor) por vértice
45  geometria.setAttribute( 'position', new THREE.BufferAttribute( vertices, 3 ) );
46  geometria.setAttribute( 'color', new THREE.BufferAttribute(new Float32Array(cores), 3));
47
48  // É necessário também criar o material, para este caso vamos utilizar um material básico e
49  // dentro desse material básico (que representa uma cor) ativamos o parametro vertexColors
50  // para assumir a matriz que criamos com os pontos RBG como as cores a aplicar
51  var material = new THREE.MeshBasicMaterial({vertexColors: true});
52
53  // No final, quando já tens a geometria e o material, é necessário criares uma mesh
54  // com os dados da geometria e do material. A Mesh é o componente necessário para
55  // poderes fazer as diferentes transformações ao objeto.
56  var mesh = new THREE.Mesh(geometria, material );
57
58  // Função chamada quando a página HTML acabar de carregar e é responsável por configurar
59  // a cena para a primeira renderização.
60  function Start(){
61      // O código abaixo adiciona o triangulo que criamos anteriormente à cena.
62      cena.add(mesh);
63
64      renderer.render(cena, camara);
65  }
```

De forma a conferires o resultado final, vai ao teu browser e atualiza a página (caso estejas a utilizar o *Live server*, não será necessário atualizar a página pois a aplicação atualiza a página em tempo real) e clica no link para ver a versão ThreeJS. Como podes verificar, os resultados são idênticos nas duas páginas. No entanto, por ser uma biblioteca baseada em WebGL, permite a criação de aplicações de uma forma muito mais expedita.