Aula prática 1

Objetivos

Familiarização com o ambiente de desenvolvimento

Visto irmos usar WebGL, os problemas a resolver consistem em pequenas aplicações que correrão num browser, escritas em HTML e Javascript. Embora não seja totalmente necessário, recomenda-se o uso do editor de texto Brackets (brackets.io), o qual suporta "live preview" das páginas em construção.

Também poderá usar qualquer outro editor de texto com o qual se sinta familiarizado. A nossa tarefa consistirá, na maior parte dos casos, em editar ficheiros HTML e javascript, com especial ênfase para este último tipo.

Organização das pastas e ficheiros

Para uniformização, a organização dos ficheiros dos nossos exemplos/projetos terá as seguintes características, supondo a existência duma pasta raiz, de nome `CGI`:

- Uma pasta denominada `Common` onde serão colocadas as bibliotecas auxiliares que necessitamos para os nossos programas.
- Uma pasta por cada exemplo de programa ou projeto

```
CGI
|
+--Common
|   |
|   +-- webgl-utils.js
|   +-- initShaders.js
|   +-- MV.js
|
+--ex01-triangle
|   |
|   +-- triangle.html
|   +-- triangle.js
|
+--tp1
|
...
```

O conteúdo da pasta `Common` será, para já, o seguinte:

- `webgl-utils.js` - biblioteca para inicialização do contexto WebGL e associação ao canvas
- `initShaders.js` - biblioteca para carregar/compilar e ligar os shaders em programas
- `MV.js` - biblioteca matemática para operações vetoriais/matriciais usadas frequentemente

O conteúdo da pasta específica do exemplo/projeto em questão, neste caso `ex01-triangle`, terá obrigatoriamente dois ficheiros:

- um ficheiro HTML com a página do programa, por exemplo `triangle.html`
- um ficheiro javascript com o código da aplicação, por exemplo `triangle.js`

Para ilustração poderá navegar até à página de programas de exemplo para ver uma concretização da organização proposta.

Recrie a estrutura de pastas acima copiando os ficheiros usando os links nesta página. Experimente abrir o programa de exemplo diretamente a partir do "file system" e a partir do Live Preview do brackets (caso o tenha instalado).

Ficheiro do documento (Triangle.html)

Como já foi referido, iremos necessitar dum ficheiro *.html* que representa o "documento" a carregar pelo browser para executar o nosso programa. No essencial, os nossos documentos .html terão uma área de desenho (um canvas) e alguns elementos de interface, tais como sliders, botões, caixas de texto, etc. Estes últimos elementos serão utilizados para materializar a interface gráfica da nossa aplicação.

Vamos começar por analisar a estrutura do ficheiro `triangle.html`:

```
<!DOCTYPE html>
<html>
<head>
    <script ...>
    ...
    </script>
    <script ...>
    ...
    </script>
    ...
</head>
<body>
    <title>Triangle</title>
    <canvas id="gl-canvas" width="512" height="512">
        Oops... your browser doesn't support the HTML5 canvas element"
    </canvas>
</body>
</html>
```

O conteúdo do documento, delimitado pelas tags `<html>...</html>` está dividido em duas secções:

- Uma zona de cabeçalho, delimitada por `<head>...</head>`
- Uma zona relativa ao corpo (conteúdo útil) da página e delimitada por `<body>...</body>`

Corpo do documento - Conteúdo efetivo da página

Embora possa parecer pouco natural, iremos começar por descrever os componentes desta secção do documento, deixando a discussão da zona de cabeçalho para depois.

O corpo do documento do nosso programa consiste num título, o qual aparecerá na barra de título da janela, bem como de uma zona de desenho, de dimensão 512x512 pixels. Esta área de desenho consiste num elemento `<canvas>`. O texto no seu interior apenas será mostrado se o browser não suportar WebGL, caso contrário, o nosso programa encarregar-se-á de desenhar os gráficos da aplicação na área do canvas, ocultando a mensagem do seu interior.

```
<body>
    <title>Triangle</title>
    <canvas id="gl-canvas" width="512" height="512">
        Oops... your browser doesn't support the HTML5 canvas element"
    </canvas>
</body>
```

E é tudo quanto ao documento! Agora, vamos analisar o código da aplicação, que nos revelará o que ela efetivamente faz.

Zona de cabeçalho - Carregamento de recursos adicionais

Na zona de cabeçalho é onde se indicam os recursos adicionais de que a página necessita. Neste caso, podemos encontrar duas variantes dum mesmo tipo de recurso: `<script>`. Por um lado podemos encontrar pedaços de código cuja finalidade é a de serem executados pelo processador gráfico. Estes scripts são aquilo a que designamos por *shaders*, sendo um destinado a ser aplicado a coordenadas no espaço (vértices) e o outro aplicado a pixels na imagem que se vai formando (fragmentos). As designações comuns para estes scripts são *vertex shader* e *fragment shader*:

Vertex shader

```
<script id="vertex-shader" type="x-shader/x-vertex">
attribute vec4 vPosition;
void main(){
    gl_Position = vPosition;
}
</script>
```

O exemplo de vertex shader acima ilustrado limita-se a propagar pelo pipeline gráfico o vértice que recebeu. Ou seja, trata-se dum programa que produz como output exactamente aquilo que recebeu de input.

Fragment shader

```
<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;
void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
</script>
```

O *fragment shader* acima limita-se a definir que a cor final do pixel a ser pintado é a cor vermelha.

Carregamento das bibliotecas auxiliares javascript

Para além destes dois scripts acima e que estão efectivamente embebidos no próprio documento .html, existe ainda a possibilidade de carregar scripts alojados em ficheiros separados. São exemplo disso os três ficheiros javascript que são carregados pelas seguintes linhas do documento .html:

```
...
<script type="text/javascript" src="../Common/webgl-utils.js"></script>
<script type="text/javascript" src="../Common/initShaders.js"></script>
<script type="text/javascript" src="../Common/MV.js"></script>
...
```

Todos os nossos programas irão ter estes três ficheiros carregados de início, ficando as funções e tipos de dados neles definidos disponíveis para uso no nosso programa.

Carregamento do código específico do programa

Após o carregamento do código javascript relativo às bibliotecas que necessitamos, é altura de carregar o código específico de cada aplicação:

```
...
<script type="text/javascript" src="triangle.js"></script>
...
```

Neste caso o código relativo ao nosso programa, e que é responsável por mostrar um triângulo vermelho no ecrã, está guardado no ficheiro `triangle.js`.

Ficheiro da aplicação (Triangle.js)

O nosso programa está armazenado precisamente neste ficheiro. Nele podemos encontrar duas funções sem as quais nada funcionaria. Por um lado temos a função de inicialização do programa, a qual é invocada assim que o browser acabar de efectuar o carregamento da página respetiva:

```
window.onload = function init() {
    ...
}
```

A outra função igualmente importante é a função responsável pelo desenho produzido pela aplicação. Esta função é, normalmente, chamada várias vezes por segundo e será ela que desenhará os gráficos na zona do documento ocupada pelo canvas:

```
function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, 3);
}
```

Neste exemplo simples e sem animações, ela é chamada uma única vez, mesmo no final da função `init()`. Para além de limpar o canvas, é desenhado um triângulo. Repare-se que a chamada da função para desenhar triângulos não contém qualquer coordenadas dos vértices.

Olhemos agora para dentro da função de inicialização, invocada pelo browser assim que a página (e todos os scripts referidos) forem carregados.

De início, coloca-se na variável `canvas` o elemento da página html com o id `gl-canvas`. Neste caso, é o canvas que já foi referido anteriormente e que tem 512x512 pixels de dimensão:

```
window.onload = function init() {
    var canvas = document.getElementById("gl-canvas");
    gl = WebGLUtils.setupWebGL(canvas);
    if(!gl) { alert("WebGL isn't available"); }
    ...
```

De seguida o programa invoca uma função auxiliar definida no ficheiro `webgl-utils.js` e cuja intuito é o de inicializar o webgl por forma a que o output gráfico seja dirijido para a área da página ocupada pelo canvas. Deste ponto do programa em diante, a variável `gl` refere-se ao contexto do webgl e permite aceder a todas as suas funções.

Segue-se a declaração dos vértices do nosso triângulo. Para ajudar a visualizar o triângulo convém referir que o sistema de eixos do webgl tem o eixo x na horizontal, e o y na vertical. A área de desenho corresponde a [-1,1]x[-1,1].

```
    ...
    // Three vertices
    var vertices = [
        vec2(-0.5, -0.5),
        vec2(0.5, -0.5)
        vec2(0.0, 0.5),
    ];
    ...
```

A chamada da função viewport define a zona retangular onde o output gráfico irá ser mostrado. À partida poderá usar-se toda a área do canvas, mas tal não é obrigatório, bastando para tal indicar outros limites na chamada daquela função. Neste caso está a ser usado todo o retângulo do canvas:

```
    ...
    // Configure WebGL
    gl.viewport(0,0,canvas.width, canvas.height);
    gl.clearColor(1.0, 1.0, 1.0, 1.0);
    ...
```

O próximo passo é a criação dum programa GLSL (a linguagem usada para programar o pipeline gráfico). Um programa GLSL é composto por um shader - vertex shader - que será responsável por processar cada vértice das primitivas gráficas que forem usadas (linhas, triângulos, pontos, ...), bem como por um outro - fragment shader - invocado para estabelecer a cor de cada pixel a ser desenhado no ecrã.

No exemplo concreto desta aplicação, a qual desenha apenas um triângulo pintado a vermelho no ecrã, o vertex shader será chamado para processar cada um dos vértices do triângulo, enquanto o *fragment shader* será invocado para cada pixel no interior do triângulo. Tipicamente, o número de vértices numa cena é inferior ao número de pixels a pintar, embora possa haver situações em que tal não acontece. A chamada da função `initShaders()` devolve o programa GLSL formado pelos dois shaders referidos nos 2º e 3º parâmetros. Repare que esses são os nomes dados aos `id`'s dos respetivos scripts na página html. Assim, podemos ter múltiplos shaders nas nossas aplicações, visto cada um deles ser referido por um id próprio.

```
    ...
    // load shaders and initialize attribute buffers
    var program = initShaders(gl, "vertex-shader", "fragment-shader");
    gl.useProgram(program);
    ...
```

A parte do código que se segue trata de enviar as coordenadas dos vértices do triângulo para a memória do processador gráfico. Os detalhes da operação serão explicados posteriormente, mas envolvem a criação dum buffer no GPU, a ativação desse mesmo buffer e o seu preenchimento com dados.

```
    ...
    // Load the data into the GPU
    var bufferId = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
    ...
```

Por fim, uma vez transferidos os dados para a memória do GPU, torna-se necessário ligar esses mesmos dados às variáveis referidas no programa GLSL (quer no vertex shader, quer no frament shader). Mais tarde voltaremos a analisar isto:

```
    ...
    // Associate our shader variables with our data buffer
    var vPosition = gl.getAttribLocation(program, "vPosition");
    gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(vPosition);
    ...
```

Exercícios propostos

Nota: Para cada exercício crie uma nova pasta e comece por copiar para lá os dois ficheiros do exemplo inicial (triangle.js e triangle.html) renomeando-os da forma que achar mais conveniente para cada um dos exercícios.

Aula:

Exercício 1.1 - Experimente mudar a cor do preenchimento do triângulo

Exercício 1.2 - Experimente mudar a cor de fundo do canvas

Exercício1.3 - Experimente desenhar um quadrado em vez dum triângulo (nota: o webgl apenas suporta triângulos...)

Exercício 1.4 - Que acontece às figuras se mudar as dimensões do canvas para outros formatos?

Exercício 1.5 - Desenhe apenas a fronteira da figura, usando segmentos de reta.

Exercício 1.6 - Desenhe o interior duma cor e a fronteira de outra cor. (sugestão: crie um frament shader adicional)

TPC:

Exercício 1.7 - Faça com que a figura se desloque ao longo do tempo (sugestão: fazer a animação no vertex shader. Ver variáveis uniformes)

Exercício 1.8 - Desenhe um programa que gere 10.000 pequenos triângulos aleatórios. Desses, apenas estarão visíveis 500 em cada momento. Em cada instante será visualizado um novo triângulo, deixando de ver-se o triângulo que estava visível há mais tempo.