# 第1课 绘制一个三角形

本节课想要实现与[Direct3D 11教程2：绘制一个三角形](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=927&classId=5)相同的效果——在屏幕上绘制了一个黄色的三角形，代码来自于《WebGL高级编程》中的Listing-2-1-Basic-WebGL-Example。程序截图如下：



我比较熟悉用C#编写XNA程序，对用C++编写DirectX程序也略有研究，Shader语言用的是HLSL，对于WebGL中使用的OpenGL ES 2.0及其对应的Shader语言使用的GLSL比较陌生。不过这并不要紧，绝大部分图形学原理都是相同的，只是语法稍有不同罢了。

编写WebGL应用程序时，需要用源代码编写顶点着色器和片段着色器。但是在它们由GPU用作着色器程序之前，需要经过运行时的编译和链接。这正是设置和绘制这个简单的示例也需要好几个步骤的原因。创建一个基本的WebGL应用程序的基本步骤是：

* 创建WebGL上下文
* 创建顶点缓冲
* 创建顶点着色器和片段着色器
* 编译着色器
* 创建程序对象和链接着色器
* 绘制场景

以上这些步骤，如果你曾经研究过DirectX编程的话，会发现并不陌生，只不过在HLSL中，片段着色器被称为像素着色器。下面对每一步进行详细的说明。

## 创建WebGL上下文

在代码底部你会看到如下的HTML代码：

<body onload="startup();">

 <canvas id="Lesson1" width="500" height="500"></canvas>

</body>

整个Demo页面中只有这是一段完整的页面body部分的代码，其余的都是Javascript代码。显然我们可以在<body>中插入其他的HTML代码，以使WebGL图形可以嵌入到普通的网页中。<canvas>标签就是3D图形的部分。Canvas是HTML5中的新概念，它支持使用Javascript来绘制2D图形和通过WebGL来绘制3D图形。我们只需要简单的指定canvas标签中的布局的属性，然后把所有建立WebGL的代码都放到一个叫做startup()的Javascript函数中，它在页面加载后被调用使你可以看到绘制的图形。

startup()方法的第一件事情是用document.getElementById()获得页面中<canvas>标记的一个引用：

function startup() {

 canvas = document.getElementById("Lesson1");

 gl = createGLContext(canvas);

获得画布的引用后，就以它为参数调用函数createGLContext。在这个函数中，通过用一个标准的上下文名称调用方法canvas.getContext()来创建WebGLRenderingContext对象。代码如下：

function createGLContext(canvas) {

 var names = ["webgl", "experimental-webgl"];

 var context = null;

 for (var i=0; i < names.length; i++) {

 try {

 context = canvas.getContext(names[i]);

 } catch(e) {}

 if (context) {

 break;

 }

 }

 if (context) {

 context.viewportWidth = canvas.width;

 context.viewportHeight = canvas.height;

 } else {

 alert("Failed to create WebGL context!");

 }

 return context;

}

在WebGL规范的演变过程中，使用“experimental-webgl”上下文名称作为WebGLRenderingContext的临时名称，而在规范最终定稿中使用“webgl”名称作为它的正式名称，因此，为了增加用户浏览器能理解这个上下文名称的几率，使用names数组同时使用了这两个名称。

WebGLRenderingContext是一个提供全部WebGL API调用的接口，我们把它保存在一个名为gl的变量中。

## 创建顶点缓冲

本课中只需要保存三角形顶点位置的缓冲，在名为setupBuffer()的函数中建立和设置缓冲：

function setupBuffers() {

 vertexBuffer = gl.createBuffer();

 gl.bindBuffer(gl.ARRAY\_BUFFER, vertexBuffer);

 var triangleVertices = [

 0.0, 0.5, 0.0,

 -0.5, -0.5, 0.0,

 0.5, -0.5, 0.0

 ];

 gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(triangleVertices), gl.STATIC\_DRAW);

 vertexBuffer.itemSize = 3;

 vertexBuffer.numberOfItems = 3;

}

这个函数先调用gl.createBuffer()函数，创建一个webGLBuffer对象，并把它赋给一个全局变量vertexBuffer，然后把新建的webGLBuffer对象绑定为当前的数组缓冲对象。这相当于告诉webGL，从现在开始，这个缓冲对象就是它要使用的对象。三角形的顶点定义在一个名为triangleVertices的JavaScript数组中。

本课中为了保持简单，三角形的三个顶点直接被赋予剪裁过的坐标，无需在顶点着色器中进行矩阵变换操作。而剪裁坐标系的原点在（0，0），位于视口的中央。x轴沿着水平方向指向右，y轴方向竖直向上，z轴方向从屏幕指向用户，即用的是右手坐标系。这三个轴的坐标范围都是从-1到+1，视口的左下角的坐标为x=-1和y=-1，右上角的坐标为x=1和y=1，由于本课绘制的是一个2D图形，所有顶点的z值都为0，因此这个三角形绘制在z=0的xy平面上。

接着根据包含顶点的Javascript数组创建一个Float32Array对象，它用于把顶点数据传递给WebGL。

调用gl.bufferData()方法，把顶点数据写入当前绑定的webGLBuffer对象中。这个调用告诉webGL哪些数据保存在用gl.createBuffer()创建的缓冲对象中。

setupBuffer()函数最后执行的操作是给vertexBuffer对象添加两个新属性并给它们设置适当的初始值。第一个是itemSize属性，它定义了每个属性有多少个分量。第二个是numberOfItems属性，它定义在这个缓冲中的项項或顶点的个数。绘制场景时需要用到这两个属性属性的值。

## 创建顶点着色器和片段着色器

所有的WebGL程序都必须有一个顶点着色器和一个片段着色器。setupShaders()函数以字符串的形式包含了顶点着色器和片段着色器的源代码。本课中的着色器代码可以说是最简单的了。首先分析顶点着色器：

var vertexShaderSource =

 "attribute vec3 aVertexPosition; \n" +

 "void main() { \n" +

 " gl\_Position = vec4(aVertexPosition, 1.0); \n" +

 "} \n";

这里把顶点着色器的源代码以字符串的形式赋给JavaScript变量vertexShaderSource。其中的“+”运算符是Javascript中的字符串连接运算符，它把每行的源代码合并成一个字符串，对于本例这样的小程序，这样做处理是行得通的。但是如果面对一个比较复杂的着色器，则应该用<script>标记在全局级定义一个着色器，然后通过DOM API引用它们的源代码，这样做会更加方便，会在下一课中介绍这种方法，而且以后都会使用DOM API的方式访问着色器。

顶点着色器的第一行代码定义了一个类型为vec3、 名为aVertexPosition的变量。vec3表示一个包含三个分量的向量。在vec3类型之前还指定了此变量是属性。属性是一些特殊的输入变量，利用它们把顶点数据从WebGL API传送到顶点着色器。

在本例中，利用aVertexPosition这个属性把每个顶点的位置传送给顶点着色器，WebGL根据这些位置绘制一个三角形。为了使顶点顺利通过API并最终到达aVertexPosition属性中，还需要为顶点设置缓冲区，并将缓冲区绑定到aVertexPosition属性上。这两个步骤由后面给出的setupBuffers()和draw()函数来实现。

顶点着色器的下一条语句声明main()函数，它是执行顶点着色器的入口点。main()函数的内容非常简单，它只是把输入的一个顶点位置赋给一个名为gL\_Position的变量。所有顶点着色器都必须给这个预定义变量赋一个值。当顶点着色器结束处理一个顶点时，这个变量保存了它的位置，并将其传递给Web GL流水线的下一个阶段。

本课中的片段着色器也非常简单，具体代码如下：

var fragmentShaderSource =

 "precision mediump float; \n" +

 "void main() { \n" +

 " gl\_FragColor = vec4(1.0, 1.0, 0.0, 1.0); \n" +

 "} \n";

这个片段着色器也用一个“+”运算符把各行连接成一个字符串。第一行用一个精度限定符声明片声明片段着色器中浮点数的精度，这里使用中等精度。

在片段着色器中，main()函数定义了入口点，用vec4定义黄色，并把这个颜色保存在内置的gl\_FragColor变量中。这个内置变量是一个包含四个分量的向量，它以RGBA格式保存了片段着色器的输出颜色。

## 编译着色器

为了创建一个可以载入到GPU中且能够绘制几何图形的WebGL着色器，首先需要创建一个着色器对象，并把源代码载入到这个对象中，然后编译、链接这个着色器。

自定义的辅助函数loadShader()创建一个项点着色器或片段着色器，这要取决于传送给这个函数的参数type的值。参数type可以设置为gl.VERTEX\_SHADER或gl.FRAGMENT\_SHADER。函数loadShader()的代码如下所示：

function loadShader(type, shaderSource) {

 var shader = gl.createShader(type);

 gl.shaderSource(shader, shaderSource);

 gl.compileShader(shader);

 if (!gl.getShaderParameter(shader, gl.COMPILE\_STATUS)) {

 alert("Error compiling shader" + gl.getShaderInfoLog(shader));

 gl.deleteShader(shader);

 return null;

 }

 return shader;

}

在这段代码中，首先用gl.createShader()方法创建一个着色器对象。这个方法的参数决定了希望创建的着色器的类型，它可以取gl.VERTEX\_SHADER或gl.FRAGMENT\_SHADER。然后用gl.shaderSource()方法把源代码载入到着色器对象中。gl.shaderSource()方法的第一个参数表示己经创建的着色器对象，第二个参数表示着色器的源代码。

载入了源代码后，调用gL.compileShader()方法编译着色器，然后用gl.getShaderParameter()方法检查编译的状态。如果编译出现错误，则向用户发出一条JavaScript警告消息，并删除这个着色器对象，否则返回编译好的着色器。

## 创建程序对象和链接着色器

setupShaders()函数的第二部分创建一个程序对象，并把编译好的顶点着色器和片段着色器插入到这个对象中，然后把它们链接到一个webGL可以使用的着色器程序。下面是源代码：

 shaderProgram = gl.createProgram();

 gl.attachShader(shaderProgram, vertexShader);

 gl.attachShader(shaderProgram, fragmentShader);

 gl.linkProgram(shaderProgram);

 if (!gl.getProgramParameter(shaderProgram, gl.LINK\_STATUS)) {

 alert("Failed to setup shaders");

 }

 gl.useProgram(shaderProgram);

 shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aVertexPosition");

 }

为了创建着色器程序对象，需要调用一个名为gL.createProgram()的方法，并调用gl.attachShader()方法把编译过的顶点着色器和片段着色器附加到这个程序对象中。然后，调用 gl.linkProgram()方法执行链接操作。如果链接成功，就得到一个程序对象，并且可以调用gl.useProgram()方法，告诉WebGL引擎可以用这个程序对象绘制图形。

在链接之后，WebGL实现把顶点着色器使用的属性绑定到通用属性索引上。WebGL实现已为顶点的属性分配了固定数目的“插槽”，通用属性索引就是其中某个插槽的标识符。

在顶点着色器中必须知道每一个属性对应的通用属性索引，因为在绘制过程中，就是利用这个索引把包含顶点数据的缓冲与顶点着色器中的属性建立起正确的关联。我们可以由webGL引擎自己决定某个属性要使用哪个索引。在链接完成后，用gl.getAttribLocation()方法得到得到aVertexPosition属性绑定的索引。

把这个索引保存在shaderProgram对象中作为它的一个新属性，并命名为vertexPostionAttribute。在Javascript中，一个对象实际上只是一个哈希映射。只要给一个对象的新属性赋一个值就可以创建这个属性。因此，shaderProgram对象并没有一个预先定义的名为vertexPositionAttnbute的属性，但是给它赋一个值就可以创建这个属性。后面在draw()函数中，通过保存在这个属性中的索引把包含顶点数据的缓冲绑定到顶点着色器中的 aVertexPosition属性上。

## 绘制场景

我们把场景的实际绘制代码放在一个名为draw()的函数中：

function draw() {

 gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

 gl.clear(gl.COLOR\_BUFFER\_BIT);

 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

 vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);

 gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

 gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);

}

在这个函数中，首先定义一个视口。视口定义了最终绘制的场景在绘制缓冲中的位置。在创里WebGL上下文后，程序把视口初始化为一个原点在(0，0)位置的矩形，矩形的宽度和高象为画布的宽度和高度。这意味着调用gl.viewPort()方法实际上不会影响本课的任何对象。之所以插入这条语句，是因为它是webGL的一个基本方法，你应该熟悉它的用法。

在draw()方法通过参数gl.COLOR\_BUFFER\_BIT指示WebGL把颜色缓冲清除为事先用gl.clearColor()函数定义的颜色。本课把深蓝色定义为清除颜色。

在setupBuffer函数中，已经创建了一个WebGLBuffer对象并通过gl.bufferData()方法绑定到gl.ARRAY\_BUFFER目标上。通过gL.bufferData()方法把顶点数据发送到绑定的缓冲。但是至此还没有告诉webGL，顶点着色器中的哪个属性接受来自绑定的缓冲对象的输入数据。在本例中，顶点着色器只有一个缓冲对象和一个属性，但是，通常总是有几个缓存和属性，因此需要为它们创建连接。

WebGL方法gl.vertexAttribPointer()把刚刚绑定到gl.ARRAY\_BUFFER目标上的WebGLBuffer对象赋给一个顶点属性，后者作为一个索引传递给此方法的第一个参数。这个方法的第二个参数表示每个属性的大小或分量数。在本课中，每个属性的分量是3（因为每个顶点位置用x，y，z坐标表示），在setupBuffer()方法中已经存储该值，将其作为vertexBuffer对象的一个属性（即itemSize）。

第三个参数表示要把顶点缓冲对象中的值当作浮点数。如果我们传入的数据不是浮点数，则在顶点着色器中使用它之前，必须将其转换为浮点数。第四个参数是规范化标志，它表示是否把非浮点数转化为浮点数。在本例中，缓冲中的数据都是浮点数，因此不需要使用这个参数，第五个参数称为步幅（stride）。当这个参数取值0时，表示数据在内存中顺序存放。第六个参数，表示缓冲中的偏移量。由于数据从缓冲的开始位置存放，因此这个参数也设置为0。