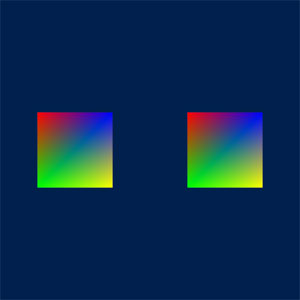
# 第2课 添加颜色

这节课介绍的内容是：

* 如何在顶点数据中包含颜色信息。
* 使用gl.drawArrays()和gl.drawElements()这两个绘图方法绘制gl.TRIANGLE\_STRIP和gl.TRIANGLES类型的图元。
* 如何在<script>标记中插入顶点着色器和片段着色器代码，并使用DOM API检索着色器。

最终效果的截图如下：



这节课介绍的内容基于第1课，下面的说明只包含与第1课不同的部分。

## 创建顶点缓冲

### 使用drawArrays()方法和TRIANGLE\_STRIP图元绘制左侧矩形

setupBuffer()函数的前半部分创建了左边的矩形的顶点坐标缓冲和顶点颜色缓冲：

// 创建左边矩形的顶点坐标缓冲

leftSquareVertexBuffer = gl.createBuffer();

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

var leftSquareVertices = [

-0.75, 0.25, 0.0, //v0

-0.75, -0.25, 0.0, //v1

-0.25, 0.25, 0.0, //v2

-0.25, -0.25, 0.0, //v3

];

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

leftSquareVertexBuffer.itemSize = 3;

leftSquareVertexBuffer.numberOfItems = 4;

左边的矩形是使用gl.drawArrays()方法进行绘制的，使用的图元类型为三角形带（TRIANGLE\_STRIP），因为如果像第1课那样使用三角形列表（TRIANGLES）图元类型的话，画一个矩形需要定义6个顶点数据，额外多出2个。

有关图元类型的介绍可参见<http://msdn.microsoft.com/en-us/library/bb147291(v=vs.85).aspx>，虽然是DirectX，但原理是相同的，有关使用三角形带的优点可参见[使用Triangle Strips提高性能](http://shiba.hpe.cn/jiaoyanzu/wuli/ShowArticle.aspx?articleId=606&classId=4)，虽然是XNA代码，但原理是相同的。

gl.drawArrays()方法根据启用的WebGLBuffer对象中的顶点数据，绘制由第1个参数定义的图元。启用的webGLBuffer对象绑定到gl.ARRAY\_BUFFER目标上。

这意味着在调用gl.drawArrays()之前，必须执行以下操作：

* 用gl.createBuffer()建立一个webGLBuffer对象。
* 用gl.bindBuffer()方法把WebGLBuffer对象绑定到gl.ARRAY\_BUFFER目标。
* 用gl.bufferData()方法把顶点数据载入到缓冲中。
* 用gl.enableVertexAttribArray()方法激活通用顶点属性。
* 调用gl.vertexAttribPointer()方法把顶点着色器的属性连接到webGLBuffer对象中的正确数据。

[gl.drawArrays()](http://msdn.microsoft.com/en-us/library/ie/dn302395(v=vs.85).aspx)方法的原型如下：

void drawArrays(GLenum mode,Glint first,GLsizei count);

下面介绍这些参数的意义：

* mode定义了所要渲染的图元的类型。它可以取gl.POINTS、gl.LINES、gl.LINE\_LOOP、 gl.LINE\_STRIP、gl.TRIANGLES、gl.TRIANGLE\_STRIP、gl.TRIANGLE\_FAN
* first参数定义顶点数据数组中的哪个索引用作第一个索引。
* count定义了需耍使用的顶点数。

归纳起来，这个方法的mode参数定义了要绘制的图元类型，count参数定义连续顶点的个数，用first参数定义了第一个顶点在数组中的索引位置。

gl.drawArrays()方法的设计要求表示图元的顶点必须按正确的顺序进行绘制。此矩形顶点的绘制顺序为V0 V1 V2 V3，TRIANGLE\_STRIP绘制的第一个三角形应该是逆时针绕行的（这是因为OpenGL为右手坐标系，因此顺时针绕行的三角形会因背面剔除而不显示），下一个三角形的前两个顶点即上一个三角形的后两个顶点，依此原理，如果以V3V2V1V0的顺序绘制也能正确显示。顶点的位置如下图所示：

V0

V1

V2

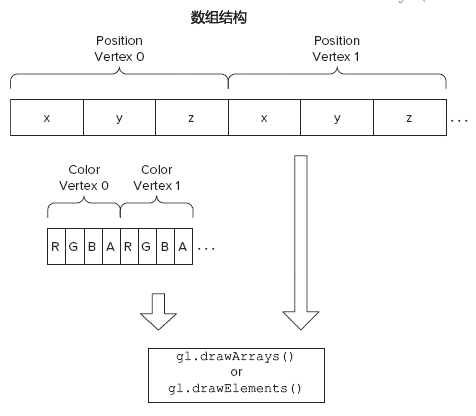
V3

在第1课中，顶点数据只包含顶点位置信息。然而，在实际的webGL应用程序中，顶点数据通常还包含更多的信息。除了顶点位置信息，顶点数据还包括顶点法线、顶点颜色和纹理坐标。

当顶点包含多种数据时，可以采用以下两种方法组织这些数据：

* 把每类顶点数据保存在WebGLBuffer对象的单独数组中，这意味着，除了顶点位置数组外，可能还有其他数组，如法线数组、颜色数组等。这通常称为**数组结构**。
* 把所有类型的数据都保存在WebGLBuffer对象的一个数组中。这意味着，需要把不同类型的数据交叉保存在同一个数组中。这通常称为**结构数组**。

一般来说，第一个方法（数组结构）是建立缓冲并把数据载入缓冲的最简单方法。每类顶点数据都有自己的顶点数组。下图是当顶点数据包含了位置信息（用坐标x、y、z）和颜色信息（RGBA格式，具有4个颜色分量且每个分量的类型为无符号字节）时的数组结构。



本教程使用的是第一种方法，第二种方法会在下一个教程中介绍。代码如下：

// 创建矩形的顶点颜色缓冲，右边的矩形共用这个颜色缓冲

squareVertexColorBuffer = gl.createBuffer();

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

var squareColors = [

1.0, 0.0, 0.0, 1.0, //v0

0.0, 1.0, 0.0, 1.0, //v1

0.0, 0.0, 1.0, 1.0, //v2

1.0, 1.0, 0.0, 1.0 //v3

];

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

squareVertexColorBuffer.itemSize = 4;

squareVertexColorBuffer.numberOfItems = 4;

在draw()方法中我们使用如下代码绘制左边的矩形：

// 绑定包含左边矩形顶点位置的缓冲

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

// 指定顶点位置在顶点数组的组织形式

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

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

// 绑定矩形顶点颜色的缓冲

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

// 指定顶点颜色在顶点数组的组织形式

gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,

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

// 绘制左边的矩形

gl.drawArrays(gl.TRIANGLE\_STRIP, 0, leftSquareVertexBuffer.numberOfItems);

### 使用drawElements()方法和TRIANGLES图元绘制右侧矩形

前面已经提到：如果使用gl.drawArrays方法和TRIANGLES图元绘制右侧的矩形，则需要定义6个顶点，有2个是重复的。这里我们可以使用gl.drawElements()方法和TRIANGLES图元，这样只需定义4个顶点即可，但需要额外定义一个元素数组缓冲。

drawArrays()和drawElements()的概念对应的就是DirectX中的Draw()和DrawIndexed()方法，因此drawElements()有时也被称为索引绘图，元素数组缓冲又被称为索引缓冲。在XNA中的相同概念可参见[5.3 使用索引移除冗余顶点](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=401&classId=4)和[5.4 使用顶点缓冲和索引缓冲将顶点和索引保存在显存中](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=402&classId=4)。

基于以上说明，setup()函数中建立右侧矩形缓冲的代码如下：

// 创建右边矩形的顶点坐标缓冲

rightSquareVertexBuffer = gl.createBuffer();

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

var rightSquareVertices = [

0.25, 0.25, 0.0, //v0

0.25, -0.25, 0.0, //v1

0.75, 0.25, 0.0, //v2

0.75, -0.25, 0.0, //v3

];

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

rightSquareVertexBuffer.itemSize = 3;

rightSquareVertexBuffer.numberOfItems = 4;

// 创建右边矩形的顶点索引缓冲

rightSquareElementBuffer=gl.createBuffer();

gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, rightSquareElementBuffer);

var indices = [0, 1, 2, 2, 1, 3];

gl.bufferData(gl.ELEMENT\_ARRAY\_BUFFER, new Uint16Array(indices), gl.STATIC\_DRAW);

rightSquareElementBuffer.numberOfItems = 6;

你可以参见上面的矩形图像理解索引的排列顺序，只要两个三角形是逆时针绕行的即可，因此按[2,0,1, 3, 2, 1]的顺序也能正常显示。

[gl.drawElements()](http://msdn.microsoft.com/en-us/library/ie/dn302396(v=vs.85).aspx)方法的原型如下：

void drawElements (Glenum mode, GLsizei count, GLenum type,Glintptr offset)

该方法的各个参数意义如下：

* mode定义了要渲染的图元的类型。
* count定义了绑定到gl.ELEMENT\_ARRAY\_BUFFER目标上的缓冲中的索引数。
* type定义了元素索引的类型，元素索引存储在绑定到gl.ELEMENT\_ARRAY\_BUFFER目标上的缓冲中。可以指定的类型为gl.UNSIGNED\_BYTE或者gI.UNSIGNED\_SHORT。
* offset定义绑定到gl.ELEMENT\_ARRAY\_BUFFER目标的缓冲中的偏移量，索引从此处开始。

在draw()方法中我们使用如下代码绘制右边的矩形：

// 绑定包含右边矩形顶点位置的缓冲

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

// 指定顶点位置在顶点数组的组织形式

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

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

// 由于右边矩形共用顶点颜色的缓冲，所以这步可以省略

// gl.bindBuffer(gl.ARRAY\_BUFFER, squareVertexColorBuffer);

// gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,

//squareVertexColorBuffer.itemSize, gl.FLOAT, false, 0, 0);

// 绑定右边矩形的索引缓冲

gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, rightSquareElementBuffer);

// 绘制右边的矩形

gl.drawElements(gl.TRIANGLES, rightSquareElementBuffer.numberOfItems, gl.UNSIGNED\_SHORT, 0);

## 加载Shader

在第1课中，顶点着色器和片段着色器都是以字符串的形式插入到JavaScript代码中，并用“+”运算符把字符串合并成着色器代码，如下所示：

var vertexShaderSource =

"attribute vec3 aVertexPosition; \n" +

"void main() { \n" +

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

"} \n";

var fragmentShaderSource =

"precision mediump float; \n" +

"void main() { \n" +

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

"} \n";

在这节课中，我们使用一个更简单的方法。下面是相同的顶点着色器代码，但是它插入<script>标记中，但并没有使用JavaScript的“+”运算符合并字符串。

<script id="shader-vs" type="x-shader/x-vertex">

attribute vec3 aVertexPosition;

attribute vec4 aVertexColor;

varying vec4 vColor;

void main() {

vColor = aVertexColor;

gl\_Position = vec4(aVertexPosition, 1.0);

}

</script>

<script id="shader-fs" type="x-shader/x-fragment">

precision mediump float;

varying vec4 vColor;

void main() {

gl\_FragColor = vColor;

}

</script>

对于这个小示例，这两种方法可能没有多大区别。但是，当我们遇到一个源代码很长的大型着色器时，则使用第二种方法会更加简洁，而且可读性会更好。

此外，由于WebGL的着色器是用OpenGL ES着色语言编写的，因此Internet上和很多文献中有很多的着色器示例都是用OpenGL ES2.0编写的，它们也可以用于webGL。也有一些着色器开发工具，如AMD公司的RenderMonkey，它可以输出OpenGL ES着色语言。如果使用第二个方法，则很容易在自己的webGL应用程序中，在<script>示记之间插入这些着色器。因此，在以后的课程中，我都将使用这种方法加载shader代码。

当在<script>标记之间包含着色器源代码而不是把它赋给一个JavaScript变量时，需要用一种方法检索它，并把它合并到一个Javascript字符串中，然后通过webGL API传入，代码如下：

function loadShaderFromDOM(id) {

var shaderScript = document.getElementById(id);

// 若未找到指定id的元素则退出

if (!shaderScript) {

return null;

}

// 遍历DOM元素的子节点，将shader代码重新构建为一个字符串

var shaderSource = "";

var currentChild = shaderScript.firstChild;

while (currentChild) {

if (currentChild.nodeType == 3) { // 3对应TEXT\_NODE

shaderSource += currentChild.textContent;

}

currentChild = currentChild.nextSibling;

}

var shader;

if (shaderScript.type == "x-shader/x-fragment") {

shader = gl.createShader(gl.FRAGMENT\_SHADER);

} else if (shaderScript.type == "x-shader/x-vertex") {

shader = gl.createShader(gl.VERTEX\_SHADER);

} else {

return null;

}

gl.shaderSource(shader, shaderSource);

gl.compileShader(shader);

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

alert(gl.getShaderInfoLog(shader));

return null;

}

return shader;

}

函数loadshaderFromDOM()需要接收一个id属性参数，它利用这个id属性在DOM树中找到相应的元素。例如，如果我们给插入着色器源代码的<script>标记设置id为shader-vs，则需要把shader-vs作为参数传给loadshaderFromDOM()函数。

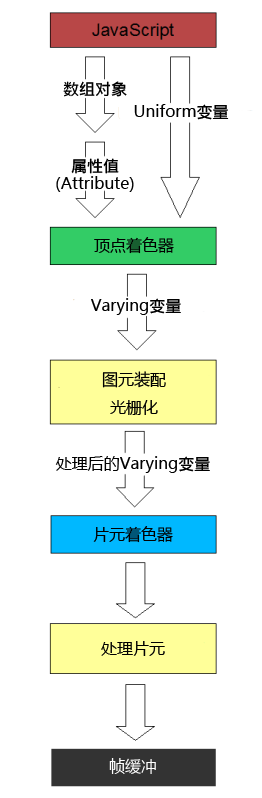
loadshaderFromDOM()使用DOM API函数并根据传入的id找到相应的元素，然后遍历这个元素的全部子元素，生成一个表示源代码的文本字符串。

最后，这个函数检查所找到元素的类型属性，并根据类型创建一个顶点着色器或片段着色器。它把着色器的源代码载入创建的着色器对象中，并对它进行编译。如果编译成功，把这个着色器对象返回给调用这个函数的程序。

## WebGL渲染管线

以下内容来自于[Lesson2添加颜色](http://www.hiwebgl.com/?p=133)。

在shader代码中还有一个与第1课不同地方：多了“varying vec4 vColor;”。为了理解这些改变的代码的原理，你需要先了解一下WebGL的渲染管线。这里有个流程图：

[](http://www.hiwebgl.com/wp-content/uploads/2011/08/webglprocess.png)

这个流程图用简单的形式表现出数据从Javascript的draw()函数中转换为像素显示在WebGL Canvas中的流程。这个流程图只给出了我们这节课需要用到的步骤，我们在以后的课程中还会为这个流程图添加更多细节。

从最高层开始，处理的过程是这样的：每次你调用类似于drawArrays的函数时，WebGL会处理你之前传递给它的数据，这些数据都是以属性（Attribute）（比如第1课中用到的顶点位置数组）和Uniform变量（我们用它来储存模型视图矩阵和投影矩阵，会在下一节课涉及）的形式存在的，然后WebGL会把这些数据传递给顶点着色器。

每次当相应顶点的属性建立完成后，都会调用一次顶点着色器；而Uniform变量，就像它的名字一样，在调用过程中并不发生任何改变，顶点着色器只是需要这些数据。然后顶点着色器把处理的结果储存在称为“Varying变量”（Varying Variable）的变量中。顶点着色器通常会输出一系列的Varying变量，其中有个特别的也是必须的变量，那就是gl\_Position，它储存着经过顶点着色器处理过的顶点坐标。

在顶点着色器处理完成之后，WebGL将这些Varying变量中描述的3D图形转换为2D图片，然后为图片中的每个像素调用片元着色器（这就是为什么在有些3D图形系统中你会听到他们把片元着色器称为“像素着色器”的原因了）。当然，确切的说是为那些非顶点位置的像素调用片元着色器，而在那些顶点位置上的像素则已经建立好了顶点。这个过程“填充”了各顶点间限定的空间，从而显示出一个可见的形状。片元着色器的作用是返回每个内插点的颜色，并储存在称为gl\_FragColor的Varying变量中。

当片元处理器工作完毕后，WebGL会再处理一下它输出的结果（我们在以后的课程中会讲到这个“再处理一下”的详细内容），然后放到Frame Buffer（帧缓冲）中，也就是最终显示在屏幕上的东西。

我们可以从顶点着色器中输出包括位置信息在内的一系列Varying变量，然后在片元着色器中重新获得他们。所以，我们要把颜色信息传递给顶点着色器，然后再直接输出为Varying变量，使片元着色器可以获取到它们。

这种方式下我们可以很容易填充渐变色。因为所有顶点着色器输出的Varying变量都在顶点间产生片元的时候被进行了线性插值，注意是所有Varying变量而不仅仅是位置信息。顶点间颜色信息的线性插值可以让我们制作平滑的渐变。