# 第3课 绘制一个立方体

这节课要介绍的内容是：

* 使用**结构数组**构建一个包含顶点颜色的立方体；
* 最简单的3D变换。

实现的效果其实就是[Direct3D 10教程4：3D空间](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=945&classId=5)中的效果，不过只有一半，只绘制了一个彩色立方体，并没有实现旋转的动画，动画会在下一节课中介绍。

程序截图如下：



在上一节课提到：当顶点包含多种数据时，可以采用两种方法组织这些数据。上节课用的是第一种方法：**数组结构**，而这节课使用第二种方法：**结构数组**。为了理解这种方法，首先需要理解类型化数组的概念。

## 类型化数组

在前面的例子中我们已经使用了如下代码创建顶点缓冲：

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);

代码中的Float32Array到底是什么？为什么要用这个类型？

在C和C++等程序设计语言中，需要处理二进制数据的情形并不少见。在这些语言中完全支持二进制数据的处理。但是在Javascript语言中，二进制数据的处理并不常见，因此JavaScript语言并没有内置二进制数据的处理功能。为此，专门为WebGL引入了新数据类型ArrayBuffer，也被称为类型化数组（typed array），它提供一个比较有效的二进制处理方法。

类型化数组的详细介绍可见<http://www.khronos.org/registry/typedarray/specs/latest/>。

### 缓冲与视图

为了处理二进制数据，类型化数组规范定义了缓冲和一个或多个缓冲视图等概念。缓冲是一个固定长度的二进制数据存储区，由类型ArrayBuffer（数组缓冲）表示。例如，用下面的代码创建—个8字节的缓冲：

var buffer = new ArrayBuffer(8);

执行这条语句后，我们就得到一个8字节的缓冲。但是，无法直接对这个缓冲中的数据进行处理。为此需要创建ArrayBuffer的一个视图。从ArrayBuffer可以创建若干个不同的视图。用下面的语句从名为buffer的的数组缓冲创建一个Float32Array视图：

var viewFloat32 = new Float32Array(buffer)

如果需要，也可以给同一个缓冲创建更多的视图。例如，我们可以同一个缓冲创建另外两个视图Uint16Array和Uint8Array：

var viewUint16 = new Uint16Array(buffer);

var viewUint8 = new Uint8Array(buffer);

如果创建了一个ArrayBuffer，并且以它为基础又创建了一个Float32Array、一个Uint16Array、一个Uint8Array视图，我们就得到如下图所示的对应关系。从中可以看出，在不同的视图中可以用不同的索引访问数组缓冲中的不同字节。例如，viewFloat32[0]代表ArrayBuffer中的字节0～3，它表示一个32位的浮点数。viewUint[2]表示数组缓冲中第4和第5个字节，它是一个16位的无符号整数。



**8字节的数组缓冲与3个不同视图之间的对应关系**

### WebGL支持的视图类型

下表列出了webGL中的视图类型以及相应的大小：

| **视图类型** | **大小（字节）** | **说明** | **C对应类型** |
| --- | --- | --- | --- |
| Int8Array | 1 | 8位有符号整数 | signed char |
| Uint8Array | 1 | 8位无符号整数 | unsigned char |
| Uint8ClampedArray | 1 | 8位无符号整数(截取) | unsigned char |
| Int16Array | 2 | 16位有符号整数 | short |
| Uint16Array | 2 | 16位无符号整数 | unsigned short |
| Int32Array | 4 | 32位有符号整数 | int |
| Uint32Array | 4 | 32位无符号整数 | unsigned int |
| Float32Array | 4 | 32位IEEE浮点数 | float |
| Float64Array | 8 | 64位IEEE浮点数 | double |

所有的视图类型都使用相同的构造函数，具有相同的属性、常量和方法。

例如，在第1课的例子中，我们先创建一个缓冲，然后显式地创建一个视图。如下所示：

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);

这个示例使用Float32Array的构造函数，它以JavaScript数组为参数。

这个构造函数创建一个新的ArrayBuffer对象，且它的空间足以存放JavaScript数组triangleVertices的内容。但是，这段代码也直接建立一个Float32Array视图，此视图绑定到这个缓冲上。triangleVertices中的顶点数据上传到ArrayBuffer中。下图显示了Float32Array和相应的ArrayBuffer。



**Float32Array和相应的ArrayBuffer**

其实视图的概念就类似于DirectX中的资源视图（resource view）：资源视图可以让一个资源绑定到图形管线的某个阶段。可以将资源视图看成C中的类型转换，C中的一块原始内存（raw memory）可以被转换为任意数据结构，我们可以将一块内存转换为整数数组，浮点数数组，结构，结构数组等。如果我们不知道原始内存的类型，那么它对我们来说用处不大。Direct3D 11的资源视图工作原理类似，例如一张2D纹理就类似于一块原始内存，就是一种原始基础资源，有了这个原始资源，我们就可以创建不同的资源视图将这个纹理以不同的格式绑定到图形管线的不同阶段，而不同的格式可以是要绘制的渲染目标，接收深度信息的深度模板缓冲，或者也可以是一个纹理资源。C中的类型转换可以以不同方式使用一块内存，而在Direct3D 11中是资源视图进行类似的操作。

## 为提高性能交叉存放顶点数据

理解了以上知识，让我们来看一下结构数组。

本节课我们要绘制一个带顶点颜色的立方体，我们要把所有类型的数据交叉都保存在一个数组中，这通常称为结构数组。

如果使用结构数组，则在建立缓冲和载入数据时需要更多的操作。下图展示了如何把位置数据和颜色数据交叉保存在同一个顶点数组中。



**交叉顶点数组（结构数组）工作方式的概念图**

从性能角度来看，把顶点数据按交叉模式保存在一个结构数组中是首选的顶点数据组织方式。这是因为它提供顶点数据的更好的内存局部性。当顶点着色器需要某个顶点的位置数据时，在同一个时刻，它也可能需要同一个顶点的法线数据和纹理数据。如果把这些数据保存在内存相近位置，则读取其中一个数据很可能同时读取同一个块中的其他数据，因此当顶点着色器需要它们时，它们已经出现在前期变换顶点缓存中。

理解结构数组的原理后，使用结构数组实际上并不困难。但是第一次使用时确实需要一点技巧。建立结构数组的代码在setupBuffer()函数中。

### 为交叉顶点数据设置缓冲

首先用gl.createBuffer()方法创建一个WebGLBuffer对象，并用gl.bindBuffer()方法把它绑定到目标缓冲桑。然后使用标准的JavaScript数组定义顶点数据。先是3个位置元素（x、y、z），然后是4个颜色元素（r、g、b、r），接下来又是3个位置元素和4个颜色元素。然后从这个JavaScript数组读取数据并载入一个类型化数组中。

前面当我们需要把顶点数据载入到WebGLBuffer对象时，需要创建一个类型化数组，并直接以一个JavaScript数组作为输入源。在这个例子中，情况稍微复杂。本例创建的WebGLBuffer对象的结构如下图所示：



**交叉顶点数组的格式**

首先是3个位置元素，每个元素占4个字节，共12个字节，这同时也是第一个颜色分量相对于缓冲的起始位置的偏移量。每个颜色分量占一个字节。一个顶点包含3个位置分量和4个颜色分量。代码如下：

cubeVertexBuffer = gl.createBuffer();

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

var vertices = [

 // 前方平面四个顶点

 -0.5, -0.5, 0.5, 0, 0, 255, 255, //v0

 -0.5, 0.5, 0.5, 0, 255, 0, 255, //v1

 0.5, -0.5, 0.5, 0, 255, 255, 255, //v2

 0.5, 0.5, 0.5, 255, 0, 0, 255, //v3

 // 后方平面四个顶点

 -0.5, -0.5, -0.5, 255, 0, 255, 255, //v4

 -0.5, 0.5, -0.5, 255, 255, 0.0, 255, //v5

 0.5, -0.5, -0.5, 255, 255, 255, 255, //v6

 0.5, 0.5, -0.5, 0, 0, 0, 255 //v7

];

顶点的位置坐标可以用下图表示：



一个顶点的数据大小可以由以下代码求得：

// 顶点数量为8

var numberOfVertices = 8;

// 计算一个顶点元素的字节数量，(x,y,z)位置为12个字节，(r,g,b,a)颜色为4个字节，此处为16

var vertexSizeInBytes = 3 \* Float32Array.BYTES\_PER\_ELEMENT +

 4 \* Uint8Array.BYTES\_PER\_ELEMENT;

// 计算一个顶点元素的浮点数数量，此处为4

var vertexSizeInFloats = vertexSizeInBytes / Float32Array.BYTES\_PER\_ELEMENT;

得到这些值后，分配一个ArrayBuffer，保存实际的顶点数据。为了访问位置元素，要把Float32Array映射到这个ArrayBuffer，为了访问颜色元素，还需把Uint8Array映射到这个ArrayBuffer。代码如下：

// 根据上面算出的大小创建一个数组缓冲，此处这个大小为128个字节

var buffer = new ArrayBuffer(numberOfVertices \* vertexSizeInBytes);

// 新建一个Float32Array视图，用于访问位置数据

var positionView = new Float32Array(buffer);

// 新建一个Uint8Array视图， 用于访问颜色数据

var colorView = new Uint8Array(buffer);

然后是一个循环，用于读取JavaScript数组的每个元素值并填充ArrayBuffer：

// 将JavaScript数组的数据填充到ArrayBuffer中

var positionOffsetInFloats = 0;

var colorOffsetInBytes = 12;

var k = 0; // JavaScript数组中的索引

for (var i = 0; i < numberOfVertices; i++) {

 positionView[positionOffsetInFloats] = vertices[k]; // x

 positionView[1 + positionOffsetInFloats] = vertices[k + 1]; // y

 positionView[2 + positionOffsetInFloats] = vertices[k + 2]; // z

 colorView[colorOffsetInBytes] = vertices[k + 3]; // R

 colorView[1 + colorOffsetInBytes] = vertices[k + 4]; // G

 colorView[2 + colorOffsetInBytes] = vertices[k + 5]; // B

 colorView[3 + colorOffsetInBytes] = vertices[k + 6]; // A

 positionOffsetInFloats += vertexSizeInFloats;

 colorOffsetInBytes += vertexSizeInBytes;

 k += 7;

}

变量positionOffsetInFloats表示Float32Array的位置，这个位置就是每个顶点的坐标x分量写入的位置。第一次迭代循环时，它的值为0。然后每次迭代增加vertexSizelnFloats（值为4）。变量coloIOffsetlnBytes是红色分量的Uint8Array中的位置。第一次迭代循环时，它的值为12，然后每次迭代时，它的值增加vertexSizelnBytes（值为16）。

### 基于交叉顶点数据绘图

为交叉顶点数据创建了正确的缓冲后，现在需要告诉WebGL其中的数据是如何组织，然后根据这些数据进行绘图。在draw()方法中的代码如下：

// 绑定同时包含顶点位置和颜色信息的缓冲

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

// 描述顶点位置在数组中的组织形式

gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

 cubeVertexBuffer.positionSize, gl.FLOAT, false, 16, 0);

// 描述顶点颜色在数组中的组织形式

gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,

 cubeVertexBuffer.colorSize, gl.UNSIGNED\_BYTE, true, 16, 12);

// 绑定索引缓冲

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

gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numItems, gl.UNSIGNED\_SHORT, 0);

首先需要绑定包含交叉顶点数据的缓冲。然后针对顶点着色器中的每个属性，用gl.[vertexAttribpointer()](http://msdn.microsoft.com/en-us/library/ie/dn302460%28v%3Dvs.85%29.aspx)方法连接到绑定缓冲中的数据上。在上述这段代码中，第一次调用vertexAttribpointer()是指示WebGL绑定的WebGLBuffer对象中的位置数据是如何组织的。这个方法中各个参数的意义如下：

* 第一个参数告诉WebGL，绑定的缓冲应该用作哪个通用属性索引的输入。这里定义的索引对应于顶点着色器中的aVertexPosition属性。
* 第二个参数定义每个位置信息要用多少个元素来表示。位置信息由x、y、z三个分量组成，因此triangleVertexBuffer.positionSize的值设置为3。
* 第三个参数设置为gl.FLOAT，它表示位置元素是浮点类型的值。
* 第四个参数是规范化标志。它说明了如何处理非浮点类型的数据项。由于位置数据是浮点类型的，因此这个标志是否设置为false对位置数据并不重要。
* 第五个参数是步长。它定义了一个元素的开始位置与下一个同类型元素的开始位置之间的间隔。本例中的步长为16字节（3个位置分量共12个字节，1个颜色分量1字节）。
* 第六个参数是本调用定义的顶点数据类型中第一个元素的偏移值。由于位置数据定义在颜色数据之前，因此位置的偏移值为0。

在第二次调用vertexAttribpointer()中，第三个参数设置为gl.UNSIGNED\_BYTE，因为每个颜色分量用ArrayBuffer中的一个字节来表示。

第四个参数（规范化标志）现在很重要，因为现在的顶点数据是gl.UNSIGNED\_BYTE而非gl.FLOAT。所有供顶点着色器处理的顶点属性都是单精度浮点数，如果传入的顶点数据是其他类型，则在顶点着色器处理这些数据之前需要把它们转换为浮点数。

最后一个参数设置为12，表示第一个颜色分量的偏移值，因为它前面有3个浮点位置元素。

## 3D变换

在前面的教程中，几何体的顶点位置都是直接赋予裁剪坐标，本例中使用的是模型坐标，要将模型坐标转换到屏幕坐标，需要使用矩阵进行坐标转换，具体的细节请参见[Direct3D 10教程4：3D空间](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=945&classId=5)。

DirectX使用的HLSL与OpenGL使用的GLSL有少许区别：HLSL矩阵使用行主序，矩阵乘法为右乘，例如顶点变换在HLSL中写成position\*worldMatrix\*viewMatrix\*projectionMatrix，而在GLSL中矩阵乘法为左乘，顶点变换写成projectionMatrix\*modelViewMatrix\*position，而且在OpenGL中将世界矩阵和视矩阵合并为一个模型视图矩阵。

在绘制立方体之前，我们必须建立两个矩阵。首先需要一个模型视图矩阵定义立方体在3D坐标系中相对于相机的位置。

第二个矩阵是投影矩阵，用于将相机空间中模型的3D坐标转换为视口的2D坐标。

在JavaScript内部并没有支持矩阵或向量的运算，与矩阵或向量接近的数据类型是内置的数组对象。

因此我们在代码中draw()方法中手动设置这两个向量：

// 设置模型视图矩阵

var modelViewMatrix = new Float32Array(

 [1, 0, 0, 0,

 0, 1, 0, 0,

 0, 0, 1, 0,

 0, -1, -5, 1]);

// 设置投影矩阵

var projectionMatrix = new Float32Array(

 [2.41421, 0, 0, 0,

 0, 2.41421, 0, 0,

 0, 0, -1.002002, -1,

0, 0, -0.2002002, 0]);

当然，不太有可能手动写出这两个矩阵，在下一节课会介绍用第三方的glMatrix库实现矩阵和向量的计算。

然后用gl.uniformMatrix4fv()方法将这两个矩阵上传到GPU中的顶点着色器：

gl.uniformMatrix4fv(shaderProgram.pMatrixUniform, false, projectionMatrix);

gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform, false, modelViewMatrix);

顶点着色器的代码也需要包含这两个矩阵：

attribute vec3 aVertexPosition;

attribute vec4 aVertexColor;

uniform mat4 uMVMatrix;

uniform mat4 uPMatrix;

varying vec4 vColor;

void main(void) {

 gl\_Position = uPMatrix \* uMVMatrix \* vec4(aVertexPosition, 1.0);

 vColor = aVertexColor;

}

最后别忘了用以下代码开启深度缓冲，否则被前面的物体挡住的后面的物体也会被绘制，图像会一团糟。

gl.enable(gl.DEPTH\_TEST);