# 第4课 3D变换

本课要介绍的内容是：

* 使用glMatrix库处理顶点的变换；
* 使用栈保存模型视图矩阵；
* 如何设置常量顶点数据。

程序截图如下：



本例中我们会在地板上放置一张桌子，桌子上放一个盒子。

## 创建和绘制地板

在setupBuffers()方法中首先创建一个位于y=0平面、边长为10的地板的顶点缓冲。代码如下：

// 建立地板的顶点缓冲

floorVertexPositionBuffer = gl.createBuffer();

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

var floorVertexPosition = [

 5.0, 0.0, 5.0, //v0

 5.0, 0.0, -5.0, //v1

 -5.0, 0.0, 5.0, //v2

 -5.0, 0.0, -5.0]; //v3

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(floorVertexPosition),

 gl.STATIC\_DRAW);

floorVertexPositionBuffer.itemSize = 3;

floorVertexPositionBuffer.numberOfItems = 4;

绘制地板的方法drawFloor()代码如下：

function drawFloor(r, g, b, a) {

 // 禁用顶点属性数组，使用颜色常量设置地板的颜色

 gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);

 // 设置颜色

 gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, r, g, b, a);

 // 绘制地板

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

 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

 floorVertexPositionBuffer.itemSize,

 gl.FLOAT, false, 0, 0);

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

}

上述代码的前两行给地板所有顶点赋予相同的颜色，下面是具体说明。

### 使用顶点数组或常量顶点数据

在前面的代码中，顶点着色器中属性变量的输入值都来自于WebGLBuffer对象的一个顶点数组。顶点数组保存了几何对象中每个顶点专用的数据。为了让一个属性从一个顶点数组读取数据，需要进行以下设置：

* 用gl.enableVertexAttribArray()方法激活对应顶点着色器中的属性的通用属性索引。
* 创建一个WebGLBuffer对象，把它绑定到顶点缓冲上，并把顶点数据载入到顶点冲。
* 调用gl.vertexAttribPointer()方法，把顶点着色器中某个属性相对应的通用属性索引连接到绑定的webGLBUffer对象上。

然而，如果顶点数据对于一个图元的所有顶点都是常量，则不需要把这个值复制到顶点数组的每个元素中。

相反，我们可以禁用与希望传入常量数据的顶点着色器中这个属性相对应的通用属性索引，为此要调用gl.disableVertexAttribArray()方法。然后将所有顶点的数据设置为这个常量值。例如，为了给类型vec4的属性设置常量顶点数据，需要调用下面的方法：

gl.vertexAttrib4f(index, r,g,b,a)

在这条调用语句中，index参数是要给所有顶点设置的4个颜色分量。还有相应的方法设置3个、2个或1个浮点数：

gl.vertexAttrib3f(index, x,y,z)

gl.vertexAttrib2f(index, x,y)

gl.vertexAttrib1f(index, x)

下图说明了gl.enableVertexAttribArray()和gl.disableVertexAttribArray()方法的工作方式：



**gl.enableVertexAttribArray()和gl.disableVertexAttribArray()方法工作方式的概念图**

## 创建立方体的顶点缓冲

然后是创建一个边长为2的立方体的顶点缓冲，因为使用drawElements()进行绘制，所以还要设置索引缓冲。代码如下：

// 创建立方体的顶点缓冲

cubeVertexPositionBuffer = gl.createBuffer();

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

var cubeVertexPosition = [

 // 前表面

 1.0, 1.0, 1.0, //v0

 -1.0, 1.0, 1.0, //v1

 -1.0, -1.0, 1.0, //v2

 1.0, -1.0, 1.0, //v3

 // 后表面

 1.0, 1.0, -1.0, //v4

 -1.0, 1.0, -1.0, //v5

 -1.0, -1.0, -1.0, //v6

 1.0, -1.0, -1.0, //v7

 // 左表面

 -1.0, 1.0, 1.0, //v8

 -1.0, 1.0, -1.0, //v9

 -1.0, -1.0, -1.0, //v10

 -1.0, -1.0, 1.0, //v11

 // 右表面

 1.0, 1.0, 1.0, //12

 1.0, -1.0, 1.0, //13

 1.0, -1.0, -1.0, //14

 1.0, 1.0, -1.0, //15

 // 上表面

 1.0, 1.0, 1.0, //v16

 1.0, 1.0, -1.0, //v17

 -1.0, 1.0, -1.0, //v18

 -1.0, 1.0, 1.0, //v19

 // 下表面

 1.0, -1.0, 1.0, //v20

 1.0, -1.0, -1.0, //v21

 -1.0, -1.0, -1.0, //v22

 -1.0, -1.0, 1.0, //v23

];

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(cubeVertexPosition),

 gl.STATIC\_DRAW);

cubeVertexPositionBuffer.itemSize = 3;

cubeVertexPositionBuffer.numberOfItems = 24;

cubeVertexIndexBuffer = gl.createBuffer();

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

var cubeVertexIndices = [

 0, 1, 2, 0, 2, 3, // 前表面

 4, 6, 5, 4, 7, 6, // 后表面

 8, 9, 10, 8, 10, 11, // 左表面

 12, 13, 14, 12, 14, 15, // 右表面

 16, 17, 18, 16, 18, 19, // 上表面

 20, 22, 21, 20, 23, 22 // 下表面

];

gl.bufferData(gl.ELEMENT\_ARRAY\_BUFFER, new Uint16Array(cubeVertexIndices),

 gl.STATIC\_DRAW);

cubeVertexIndexBuffer.itemSize = 1;

cubeVertexIndexBuffer.numberOfItems = 36;

上一课中我们只使用了8个顶点就构建了一个立方体，但在后面的教程中每个面都需要法线，所以定义了24个顶点。可以参考下图理解索引值的顺序。



**立方体示意图，图中标示了一个正面三角形和一个背面三角形**

## 理解完整的变换流水线

如果你对模型变换、视图变换、模型视图变换、投影等重要概念己经有一个基本的理解（如果没有请参见[Direct3D 10教程5：3D变换](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=946&classId=5)或参考网上的其他），下面就可以讨论整个WebGL变换流水线的处理过程。下图说明了整个变换流水线的结构。



**变换流水线**

在图的左侧，用对象坐标表示的顶点保存在WebGLBuffer对象中，顶点着色器从这个对象读取数据，并将顶点坐标乘以模型视图矩阵，这样顶点就处于观察坐标系中。如果需要在顶点着色器中执行光照处理，则通常是在观察坐标系中进行的。

顶点着色器然后将顶点坐标乘以投影矩阵，得到裁剪坐标。顶点着色器将顶点坐标写入gl\_Position变量中，gl\_Position变量的值代表裁剪坐标。

在顶点着色器将顶点写入gl\_Position后，顶点要经过透视除法，即用第四分量w除裁剪坐标（x，y，z，w）得到归一化设备坐标。最后，归一化坐标通过视口变换映射到实际的屏幕坐标。

本例使用JavaScript库[glMatrix](http://glmatrix.net/)实现对矩阵的运算操作，glMatrix由Brandon Jone开发的，它主要是为WebGL设计的，支持3个元素的向量和3×3、4×4矩阵。因此需要在代码中添加三个全局变量：

var modelViewMatrix;

var projectionMatrix;

var modelViewMatrixStack;

并在setupShaders()函数的末尾进行初始化：

function setupShaders() {

…

 modelViewMatrix = mat4.create();

 projectionMatrix = mat4.create();

 modelViewMatrixStack = [];

}

其中，modelViewMatrixStack是用来保存模型视图矩阵的栈。

## 变换矩阵的入栈和出栈操作

本例中我们用立方体绘制一个桌子，即把立方体缩放为5个长方体（一个长方体用作桌面，4个长方体用作4个桌腿）。为了绘制这样一个桌子，需要执行以下操作：先从桌子的原点位置平移到桌面需要放置的位置，然后将立方体缩放为一个长方体，使其大小合适作为一个桌面，最后绘制这个桌面。接着，平移到第一个桌腿所在的位置，并将立方体缩放为合适的大小，使之像—个桌腿，然后进行绘制。接下来平移到第二个桌腿位置，用同样的方法绘制第二个桌腿。按同样的方法绘制第三、第四个桌腿。下图显示了按这种方法绘制桌子的草图：



**桌子的侧视草图**

用这样的方法绘制一个桌子是可行的，但其过程也相当麻烦。将第一个立方体缩放为桌面后，对第一个桌腿的平移变换必须考虑到这个缩放操作对之后创建的对象的影响。当需要绘制这样的组合对象时，实际上我们通常使用分层变换技术，将变换状态保存在层次上的某个位置。

如果我们用第二种方法绘制这个桌子，则从桌子的原点位置开始。在将桌面平移到合适位置之前，先将当前的模型视图矩阵保存起来，然后将立方体平移到桌面所在的位置，并把它缩放为一个像桌面的长方体，最后绘制桌面。

桌面绘制完成后，不是直接从当前桌面位置平移第一个桌腿，而是先恢复在桌面平移和缩放之前保存的模型视图矩阵。这意味着现在回到了桌面的原点位置，即模型视图矩阵没有缩放效果。

现在，我们再次保存当前模型视图矩阵，然后平移到第一个桌腿位置，将立方体缩放为像一个桌腿的长方体，并绘制它。当第一个桌腿绘制完成后，恢复保存的变换矩阵。然后用同样的方法继续绘制第二、第三、第四个桌腿。

保存这种层级结构的变换矩阵的最好方法是使用变换矩阵栈。当我们想要保存一个变换矩阵时，就将它推入到栈顶；当我们想恢复这个矩阵时，需要将它从栈顶弹出。

JavaScript的数据数据类型都有push()和pop()方法，利用这两个方法可以将数组当作栈来处理。推入模型视图矩阵栈或者从栈中弹出的辅助函数代码如下：

function pushModelViewMatrix() {

 var copyToPush = mat4.create(modelViewMatrix);

 modelViewMatrixStack.push(copyToPush);

}

function popModelViewMatrix() {

 if (modelViewMatrixStack.length == 0) {

 throw "popModelViewMatrix()出错 - 栈为空";

 }

 modelViewMatrix = modelViewMatrixStack.pop();

}

在webGL应用程序中，要保存模型视图矩阵时就调用pushModelViewMatrix()函数，然后在需要恢复这个矩阵时调用popModelViewMatrix()函数。

drawCube(r,g,b,a)用于绘制一个立方体。代码如下：

function drawCube(r, g, b, a) {

 // 禁用顶点属性数组，使用常量颜色设置立方体顶点的颜色

 gl.disableVertexAttribArray(shaderProgram.vertexColorAttribute);

 // 设置颜色

 gl.vertexAttrib4f(shaderProgram.vertexColorAttribute, r, g, b, a);

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

 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,

 cubeVertexPositionBuffer.itemSize,

 gl.FLOAT, false, 0, 0);

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

 gl.drawElements(gl.TRIANGLES, cubeVertexIndexBuffer.numberOfItems,

 gl.UNSIGNED\_SHORT, 0);

}

这个函数用（r,g,b,a）颜色绘制一个立方体，这个立方体还没变换之前的边长为2，然后，最终的显示效果要考虑到当前的模型视图变换矩阵。

画桌子的drawTable()函数的代码如下：

function drawTable() {

 // 绘制桌面

 pushModelViewMatrix();

 mat4.translate(modelViewMatrix, [0.0, 1.0, 0.0], modelViewMatrix);

 mat4.scale(modelViewMatrix, [2.0, 0.1, 2.0], modelViewMatrix);

 uploadModelViewMatrixToShader();

 // 绘制一个扁平的咖啡色立方体作为桌面

 drawCube(0.72, 0.53, 0.04, 1.0);

 popModelViewMatrix();

 // 绘制桌腿

 for (var i = -1; i <= 1; i += 2) {

 for (var j = -1; j <= 1; j += 2) {

 pushModelViewMatrix();

 mat4.translate(modelViewMatrix, [i \* 1.9, -0.1, j \* 1.9], modelViewMatrix);

 mat4.scale(modelViewMatrix, [0.1, 1.0, 0.1], modelViewMatrix);

 uploadModelViewMatrixToShader();

 drawCube(0.72, 0.53, 0.04, 1.0);

 popModelViewMatrix();

 }

 }

}

进入drawTable()函数后，第一条语句就是调用pushModelViewMatrix()函数，保存当前模型视图矩阵。然后沿y轴平移1个单位，到达我们想要放置桌面的位置。接着调用mat4.scale()将立方体沿x轴方向和z轴方向放大2倍，沿y轴方向放大0.1倍。然后调用uploadModelMatrixToshader()函数将这个模型视图矩阵上传到顶点着色器，再调用drawCube()函数绘制桌面。最后，调用popModelViewMatrix()函数将模型视图矩阵恢复为调用mat4.translate()和mat4.scale()方法之前的值。

桌面绘制完成后，接着是两个嵌套的循环，用于绘制四个桌腿。在循环体内，先用mat4.translate()方法将立方体平移到桌腿位置，然后缩放立方体，使它沿x轴方向和z轴方向放大0.1倍，沿y轴方向放大1倍，绘制每个桌腿后，调用popModelViewMatrix()函数恢复当前模型视图矩阵。

下图显示了本例中地板、桌子和盒子的位置及尺寸：



**地板、桌子和盒子的位置和尺寸**