# 第6课 2D纹理贴图

## 载入2D纹理

WebGL中最常见的纹理格式就是2D纹理，2D纹理的最基本形式就是使用单幅图像作为纹理。为了把2D纹理应用于几何对象，首先需要载入纹理。就是将普通图像文件的纹理载入到纹理对象中，然后将这幅图像作为2D纹理的输入数据。图像文件可以是PNG、JPEG或GIF格式。

### 创建WebGLTexture对象

在WebGL中使用纹理的第一个步骤是为每个纹理创建一个WebGLTexture对象。创建纹理对象要使用下面的代码：

var texture = gl.createTexture()

WebGLTexture是个容器对象，它可以作为纹理的引用，通过它访问与此纹理有关的处理参数。

还有个方法可以显式地删除一个WebGLTexture对象。例如，如果要删除一个名为Texture的纹理对象，则可以使用以下的代码：

gl.deleteTexture(texture);

注意，当结束使用纹理时，并不需要调用gl.deleteTexture()方法。JavaScript垃圾收集在销毁WebGLTexture对象时会自动删除相应的纹理对象。这个方法只是给用户提供更灵活的控制权，控制何时销毁纹理对象。

### 绑定纹理

对新创建的纹理对象做任何操作之前，首先需要把它绑定为当前纹理对象。例如为了把一个名为texture的纹理对象绑定为一个2D纹理对象，要用下面的代码：

gl.bindTexture(gl.TEXTURE\_2D, texture);

调用gl.bindTexture()可告诉WebGL，这就是从现在起需要操作的纹理对象。这个方法与WebGLTexture纹理对象的关系就如同gl.bindBuffer()方法WebGLBuffer缓冲对象的关系。

### 载入图像数据

绑定了纹理对象后，我们就可以把图像数据载入纹理对象中，即把纹理数据传到GPU（或GPU可以访问的内存）。把纹理数据上传给GPU要使用gl.texlmage2D()方法，我们等会详细介绍，这个方法可以接受各种不同格式的纹理数据，但是当纹理是普通图像文件（PNG、GIF或JPEG）时，则它通常可以接受一个HTML DOM类型Image对象。因此，在调用gl.texlmage2D()方法之前，需要把数据保存在一个image对象中。

一个Image对象是由HTML文档中的<img>标记显式创建得到的元素。但是调用下面的函数也可以显创建一个image对象：

image = new Image();

新建的Image对象是一个空对象，还没有载入任何图像数据。为了把图像数据载入image对象中，需要把Image对象的src属性设置为需要载入图像的URL。只要把图像的URL赋给这个src属性，系统就会按异步方式载入这个图像。

为了知道图像何时载入结束，可以使用onload事件。当图像载入结束时会立刻引发这个事件。下面这个代码创建一个Image对象，并载入图像数据：

function setupTexture() {

myTexture = gl.createTexture();

myTexture.image = new Image();

myTexture.image.onload = function () {

textureFinishedLoading(myTexture)

}

myTexture.image.src = "webgl.gif";

}

载入图像且onload事件触发匿名函数时，调用textureFinishedLoading()函数把处理过程转入下个步骤。在下个步骤中把图像数据上传给GPU。

在用JavaScript代码把图像数据载入一个Image对象时，一个常用的方法是把onload事件处理程序赋给Image对象。当我们想把一个Image对象作为WebGL中某个纹理的输入数据时，就是使用这种方法。

**注意**：纹理大小必须是2的n次方。在学习如何载入图像数据时，必须知道图像可接受的大小。开发人员经常选择宽度和高度都是2的n次方的图像（即图像的宽度和高度为1、2、4、8、16、32、64、128等值）。采用这种方法的理由之一是老式的GPU只支持纹理的宽度和高度都是2的n次方。桌面OpenGL 2.0及之后的版本实际上可以支持非2的n次方（NPOT）的纹理。在OpenGL ES2.0和WebGL中，允许使用NPOT纹理，但是存在以下限制：

* 如果使用NPOT纹理，则不能使用Mip映射贴图。
* 唯一允许使用的纹理包装模式是gl.CLAMP\_TO\_EDGE。

### 将纹理上传到GPU

为了把纹理上传到GPU，需要调用gl.[texImage2D()](http://www.khronos.org/registry/webgl/specs/latest/1.0/#TEXIMAGE2D_HTML)方法。这个方法有多个不同的版本，

它们各有不同的参数，具体用法取决于用作纹理的数据类型。主要的3个原型如下所示：

void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLImageElement image) /\* May throw DOMException \*/

void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLCanvasElement canvas) /\* May throw DOMException \*/

void texImage2D(GLenum target, GLint level, GLenum internalformat, GLenum format, GLenum type, HTMLVideoElement video) /\* May throw DOMException \*/

对于第一个版本，纹理数据是一个HTML DOM类型的image对象。第二个版本接受一个HTML5画布元素作为纹理的输入数据。最后一个版本接受一个视频元素作为纹理的输入数据。

接受HTML Image对象的版本可能会是我们最经常使用的方法。为了将Image对象上传到GPU，要像下面的代码那样调用gl.texImage2D()方法：

gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, texture.image);

第1个参数表示目标是一个2D纹理，第2个参数指定了Mip映射级别，这里必须把这个参数设置为0。

第3个参数表示内部格式，第4个参数表示格式，在WebGL中，这两个参数必须相同。在本例中，用gl.RGBA表示此纹理的每个纹素都有红、绿、蓝和alpha通道这4个分量。

第5个参数定义了每个纹素数据的存储类型。用gI.UNSIGNEDBYTE表示用一个字节保存红、绿、蓝和alpha的每个通道信息。这意味着，每个纹理需要占用4个字节的内存。

最后一个参数表示HTML DOM类型的Image对象，此对象己载入图像数据。从这个函数的原型可以看出，这个参数也可以是HTML5画布元素或视频元素。

### 定义纹理参数

有几个参数需要用户来设置，它们会影响绘图期间的纹理处理。这些参数都是用gl.texParameteri()方法设置的。

这个方法的原型如下：

void texParameteri(GLenum target, GLenum pname, GLint param)

该方法的参数可以取以下值：

* target可以取gl.TEXTURE\_2D或gl.TEXTURE\_CUBE\_MAP。
* pname定义我们想要设置的目标参数。它可以取以下值（前两个值对应纹理的过滤方式，后两个值对应纹理的包装方式。至于什么是纹理过滤方式和纹理包装方式，可自己到网上查找，不再具体展开，最基本的概念可参见[XNA Shader编程教程3.1－纹理](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=336&classId=4)）：
  + gI.TEXTURE\_MAG\_FILTER
  + gI.TEXTURE\_MIN\_FILTER
  + gl.TEXTURE\_WRAP\_S
  + gl.TEXTURE\_WRAP\_T
* param定义了目标参数的值。Param可以设置的值取决于第一个参数pname的定义。如果pname取gl.TEXTURE\_MAG\_FILTER，则param可以取以下值之一：
  + gl.NEAREST
  + gl.LINEAR
* 如果pname是gl.TEXTURE\_MIN\_FILTFR，则param可以取以下值之一：
  + gl.NEARFST
  + gl.LINEAR
  + gl.NEAREST\_MIMAP\_NEAREST
  + gI.LINEAR\_MIMAP\_NEAREST
  + gI.NEAREST\_MIMAP\_LINEAR
  + gl.LINEAR\_MIMAP\_LINEAR
* 如果pname是gl.TEXTURE\_WRAP\_S或gI.TEXTURE\_WRAP\_T。则param可以取以下值之一：
  + gl.REPEAT
  + gl.CLMAP\_TO\_EAGE
  + gI.MIRROR\_RFPEAT

### 理解载入纹理的完整过程

现在分析一个完整的源代码段，它创建一个纹理对象和一个Image对象，然后把图像数据载入到image对象中。

var myTexture;

function setupTexture() {

myTexture = gl.createTexture();

myTexture.image = new Image();

myTexture.image.onload = function () {

textureFinishedLoading(myTexture)

}

myTexture.image.src = "wall.png";

}

function textureFinishedLoading(texture) {

gl.bindTexture(gl.TEXTURE\_2D, texture);

gl.pixelStorei(gl.UNPACK\_FLIP\_Y\_WEBGL, true);

gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, texture.image);

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.NEAREST);

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.NEAREST);

gl.bindTexture(gl.TEXTURE\_2D, null);

}

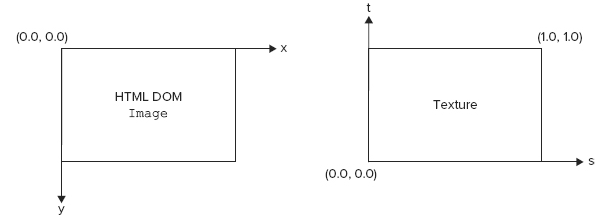
运行这段代码时，首先执行serupTextures()，我们会在startup()方法中调用它。在这个函数调用中，用gl.createTexture()方法创建一个纹理对象。然后创建一个Image对象用于保存纹理数据。图像载入结束时会触发Image对象的onload事件，并且以texture对象为参数调用textureFinishedLoading()函数。在这个函数中，第一件事是调用gl.bindTexture()方法将此纹理对象绑定为当前纹理。然后，在把此纹理数据载入纹理对象之前，调用一个前面还没有介绍过的方法，如下所示：

gl pixelStorei(gl.UNPACK\_FLIP\_Y\_WEBGL,true);

gl.pixelStorei()方法会影响后续对gl.texImage2D()的调用方式。如果gl.pixelStorei()的第一个参数设置为gl.UNPACK\_FLIP\_Y\_WEBGL，第二个参数设置为true，则用gl.texlmage2D()函数读取纹理图像时，图像会绕水平轴翻转。

翻转图像的原因是在WebGL中纹理使用的坐标系（所有OpenGL版本都如此）不同于Image对象使用的坐标系。在WebGL中，纹理的原点(00,00)在纹理的左下角，水平轴向右方，垂直轴指向上方。

对于Image对象，原点(0.0,0.0)位于图像的左上角，水平轴指向右方，但是垂直轴指自下方。下图说明了这两个坐标系。左边是HTML DOM类型的Image对象使用的坐标系，右边是WebGL纹理使用的坐标系。



**HTML DOM Image对象和WebGL纹理的坐标系**

定义了纹理数据是否需要翻转之后，调用gl.texImage2D()把纹理数据上传到GPU。最后两次调用gl.texParameteri()方法，参数分别设置为gl.TEXTURE\_MAG\_FILTER和gl.IEXTURE\_MIN\_FILTER，如下所示：

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.NEAREST);

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.NEAREST);

以上两条调用语句定义了当前纹理的纹素与屏幕上的像素没有一对一映射关系时纹理的映射方式。第一条调用语句定义纹理放大时（或伸展时）的纹理映射方式，第二条调用语句定义了纹理缩小时（或收缩时）的纹理映射方式。

## 在缓冲中定义纹理坐标

要决定几何对象的各个部分与纹理的各个部分之间的关系，需要给几何对象的顶点设置纹理坐标。代码在setupBuffers()方法中：

var cubeVertexBuffer;

var cubeVertexIndexBuffer;

function setupBuffers() {

cubeVertexBuffer = gl.createBuffer();

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

vertices = [

// 前表面

-0.5, -0.5, 0.5, 0.0, 0.0,

0.5, -0.5, 0.5, 1.0, 0.0,

0.5, 0.5, 0.5, 1.0, 1.0,

-0.5, 0.5, 0.5, 0.0, 1.0,

// 后表面

-0.5, -0.5, -0.5, 1.0, 0.0,

-0.5, 0.5, -0.5, 1.0, 1.0,

0.5, 0.5, -0.5, 0.0, 1.0,

0.5, -0.5, -0.5, 0.0, 0.0,

// 上表面

-0.5, 0.5, -0.5, 0.0, 1.0,

-0.5, 0.5, 0.5, 0.0, 0.0,

0.5, 0.5, 0.5, 1.0, 0.0,

0.5, 0.5, -0.5, 1.0, 1.0,

// 下表面

-0.5, -0.5, -0.5, 1.0, 1.0,

0.5, -0.5, -0.5, 0.0, 1.0,

0.5, -0.5, 0.5, 0.0, 0.0,

-0.5, -0.5, 0.5, 1.0, 0.0,

// 右表面

0.5, -0.5, -0.5, 1.0, 0.0,

0.5, 0.5, -0.5, 1.0, 1.0,

0.5, 0.5, 0.5, 0.0, 1.0,

0.5, -0.5, 0.5, 0.0, 0.0,

// 左表面

-0.5, -0.5, -0.5, 0.0, 0.0,

-0.5, -0.5, 0.5, 1.0, 0.0,

-0.5, 0.5, 0.5, 1.0, 1.0,

-0.5, 0.5, -0.5, 0.0, 1.0,

];

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

// 顶点数量为24

var numberOfVertices = 24;

// 计算一个顶点元素的字节数量，(x,y,z)位置为12个字节，(u,v)纹理坐标为8个字节，此处为20

var vertexSizeInBytes = 20;

// 根据上面算出的大小创建一个数组缓冲，此处这个大小为480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Float32Array视图， 用于访问纹理坐标

var texcoordView = new Float32Array(buffer);

var index = 0; // JavaScript数组中的索引

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

positionView[index] = vertices[index]; // x

positionView[1 + index] = vertices[index + 1]; // y

positionView[2 + index] = vertices[index + 2]; // z

texcoordView[3 + index] = vertices[index + 3]; // s

texcoordView[4 + index] = vertices[index + 4]; // t

index += 5;

}

cubeVertexBuffer.positionSize = 3;

cubeVertexBuffer.texcoordSize = 2;

cubeVertexIndexBuffer = gl.createBuffer();

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

var cubeVertexIndices = [

0, 1, 2, 0, 2, 3, // 前表面

4, 5, 6, 4, 6, 7, // 后表面

8, 9, 10, 8, 10, 11, // 上表面

12, 13, 14, 12, 14, 15, // 下表面

16, 17, 18, 16, 18, 19, // 右表面

20, 21, 22, 20, 22, 23 // 左表面

];

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

cubeVertexIndexBuffer.itemSize = 1;

cubeVertexIndexBuffer.numItems = 36;

}

以上代码类似于[第3课 绘制一个立方体](http://shiba.hpe.cn/jiaoyanzu/wuli/ShowArticle.aspx?articleId=1250)，也使用了交叉数组保存顶点的数据，只不过将顶点颜色的值替换为顶点纹理坐标的值。

对应的绘制代码在draw()中：

function draw() {

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, 20, 0);

// 描述顶点纹理坐标在数组中的组织形式

gl.vertexAttribPointer(shaderProgram.textureCoordAttribute, cubeVertexBuffer.texcoordSize, gl.FLOAT, false, 20, 12);

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

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

}

## 着色器中的纹理处理

代码如下：

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

attribute vec3 aVertexPosition;

attribute vec2 aTextureCoords;

uniform mat4 uMVMatrix;

uniform mat4 uPMatrix;

varying vec2 vTextureCoords;

void main(void) {

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

vTextureCoords = aTextureCoords;

}

</script>

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

precision mediump float;

varying vec2 vTextureCoords;

uniform sampler2D uSampler;

void main(void) {

gl\_FragColor = texture2D(uSampler, vTextureCoords);

}

</script>

从这段代码可以看出，用来保存纹理坐标的信息在一个vec2类型的aTextureCoords属性中，此外还有一个名为vTextureCoords 的可变变量，它把纹理坐标传递给片段着色器。

在片段着色器中，读取可变变量vTextureCoords 的值，然后从纹理对象读取纹素，并声明一个类型为sampler2D的特殊统一体变量uSampler。这个采样器用来表示需要访问哪个纹理图像单元。

采样器统一体的值来自JavaScript代码，用gl.uniformli()方法设置。它的值必须与绑定纹理的纹理图像单元相对应。例如，假设有纹理绑定到纹理图像单元gl.TEXTURE0上，则要用gl.uniformli()方法把uSampler统一体变量设置为0。即需要在draw()方法中添加以下代码：

function draw() {

gl.activeTexture(gl.TEXTURE0);

gl.bindTexture(gl.TEXTURE\_2D, myTexture);

gl.uniform1i(shaderProgram.samplerUniform, 0);

}