# 第8课 鼠标输入

本节课要介绍的内容是：

* 鼠标控制；
* 纹理过滤模式。

程序截图如下：



上一节课我们已经介绍了如何在WebGL应用程序处理键盘输入，这节课介绍鼠标输入处理，它是WebGL应用程序获取用户的另一个重要方法。虽然存在几个不同的鼠标事件，但是有3个事件在WebGL应用程序中特别有用：

* mousemove
* mousedown
* mouseup

当把鼠标移动到某个对象之上时就会引发mousemove事件，在某个元素之上按下鼠标按键时就会引发mousedown事件，在某个对象之上释放鼠标按键时就会引发mouseup事件。这3个事件都包含几个非常有用的属性。这三个事件都有clientX和clientY属性，这两个属性包含了鼠标指针相对与浏览器视口左上角的位置信息。

mousedown和mouseup也包含一个button属性，它决定鼠标上的哪个按键被按下或释放。如果button值为0，则表示此事件与鼠标的左按键有关；如果button值为1，则此事件与鼠标的中间按键有关；如果button值为2，则事件与鼠标的右按键有关。

要使用鼠标事件，首先需要在startup()方法中为这些事件注册—个事件处理程序，如下面几行代码所示：

canvas.addEventListener('mousemove', handleMouseMove, false);

canvas.addEventListener('mousedown', handleMouseDown, false);

canvas.addEventListener('mouseup', handleMouseUp, false);

下面这段代码包含3个函数，它们是这3个事件的处理程序。

var cameraZ = -2.5;

var mouseDown = false;

var lastMouseX = null;

var lastMouseY = null;

var cubeRotationMatrix = mat4.create();

mat4.identity(cubeRotationMatrix);

function handleMouseMove(event) {

if (!mouseDown) {

return;

}

var newX = event.clientX;

var newY = event.clientY;

var deltaX = newX - lastMouseX;

var newRotationMatrix = mat4.create();

mat4.identity(newRotationMatrix);

mat4.rotate(newRotationMatrix, deltaX / 400, [0, 1, 0]);

var deltaY = newY - lastMouseY;

mat4.rotate(newRotationMatrix, deltaY / 400, [1, 0, 0]);

mat4.multiply(newRotationMatrix, cubeRotationMatrix, cubeRotationMatrix);

lastMouseX = newX

lastMouseY = newY;

}

function handleMouseDown(event) {

mouseDown = true;

lastMouseX = event.clientX;

lastMouseY = event.clientY;

}

function handleMouseUp(event) {

mouseDown = false;

}

上面的代码中，我们从event属性中获取鼠标的当前位置，并进一步计算了出当前帧鼠标在X、Y方向上的移动距离，然后根据这个距离确定屏幕上立方体的旋转角度。最后的效果就是可以按住鼠标键不放左右拖动就可以旋转立方体。

## 处理纹理过滤

在这个示例中，我还使用了jquery.ui创建了一个ui界面，通过按钮可以改变纹理的过滤模式。

// jquery代码，改变纹理过滤模式

$(function () {

$("#slider-z").slider(

{ value: 2.5, min: 1, max: 12, step: 0.1, slide: function () { updateCameraPosition() }, change: function () { updateCameraPosition() } }

);

$('#opt-mag-filter').buttonset();

$('#opt-min-filter').buttonset();

$('#mag-nearest').click(function () {

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

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

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

});

$('#mag-linear').click(function () {

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

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

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

});

$('#mag-linear').click(function () {

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

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

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

});

$('#min-nearest').click(function () {

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

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

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

});

$('#min-linear').click(function () {

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

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

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

});

$('#min-nearest-mipmap-nearest').click(function () {

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

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

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

});

$('#min-linear-mipmap-nearest').click(function () {

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

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR\_MIPMAP\_NEAREST);

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

});

$('#min-nearest-mipmap-linear').click(function () {

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

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

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

});

$('#min-linear-mipmap-linear').click(function () {

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

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR\_MIPMAP\_LINEAR);

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

});

});

function updateCameraPosition() {

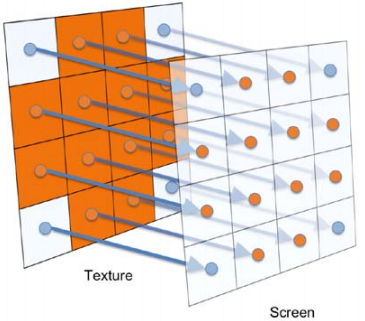
var z = $('#slider-z').slider("value");

cameraZ = -z;

$('#slider-z-value').html(z);

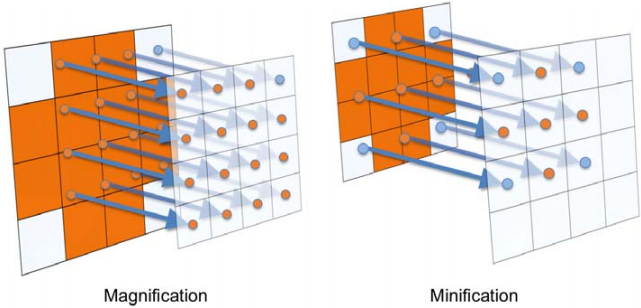
}

那什么是纹理过滤模式？假设有一张大小为256×256的纹理，再假设将此纹理应用于一个正方形。这个正方形投影到屏幕上，它的大小与纹理的大小一样。则在这个情形下，应用纹理后的正方形看起来与原图非常相似。



在一般的3D场景中，对象离相机距离有时很近，有时很远。根据透视投影，这意味着当对象离照相机较近时，其在屏幕上的投影比较大；当对象离照相机较远时，其在屏幕上的投影较小。

如果投影对象在屏幕上占据的区域大于纹理原始图像的大小，则需要拉伸纹理，这就是所谓的纹理伸展（**magnification**）。与此相反，如果投影对象在屏幕上占据的区域小于纹理原始图像的大小，则需要减小纹理的大小，这就是所谓的纹理收缩（**minification**）。



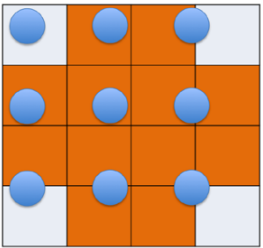
我们用纹理过滤表示纹理伸展或收缩时计算片段颜色的一般过程。调用gl.texParameteri()方法可以控制纹理的过滤方式，原型如下：

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

为了定义纹理的伸展过滤器，需要把这个方法的第二个参数设置为gl.TEXTURE\_MAG\_FILTER；为了定义纹理的收缩过滤器，需要把它的第二个参数设置为gl.TEXTURE\_MIN\_FILTER。

### 纹理伸展

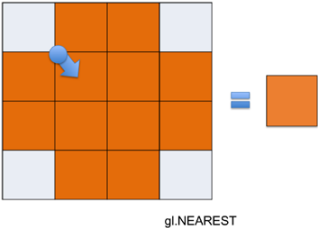
纹理伸展时一个纹素对于屏幕上的几个像素。下图说明了使用伸展过滤模式时像素与纹素之间的关系，其中桔色、白色方块表示纹素，蓝色圆点表示像素。纹理稍微进行了一点伸展，你会发现左下角、右上角和右下角的三个像素落在了不同的颜色区域，那么，这几个像素的颜色究竟应该是桔色还是白色呢？



纹理的过滤过程实质上就是确定像素（实际上就是片段）颜色的过程。伸展纹理时，可以选择两种过滤模式。最基本的过滤模式是最近相邻模式，用gl.NEAREST名称指定。用以下代码把伸展过滤设置为最近相邻模式：

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

最近相邻过滤模式表示把最接近纹素的颜色值赋给每个纹理坐标。这种过滤模式速度快，因为每个纹理坐标只需要使用一个纹素作为输入。



如果纹理伸展很多，这种过滤模式会产生块状效果，这有时也称为像素化效果。这是因为许多像素落在同一个纹素中，因此许多像素显示同一种颜色。

伸展过滤的另一个过滤模式称为线性过滤或双线性过滤，用gl.LINEAR名称定义。根据线性过滤模式，每个纹理坐标的颜色不是来自单个纹素，而是它周围的4个纹素的加权平均值。



为了把伸展过滤设置为线性过滤模式，需要调用gl.texParameteri()方法，如下所示：

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

线性过滤不会像最近相邻过滤模式那样产生块状效果，但线性过滤的特点是，当纹理拉伸时，会产生模糊效果。

### 纹理收缩

纹理收缩时几个纹素对应于屏幕上的一个像素。这意味着，几个纹素影响屏幕上一个像素的颜色，但是实际上，很难确定影响每个像素的全部纹素。

对于纹理收缩，与纹理伸展一样，也可以选择同样的两个过滤模式：最近相邻过滤和线性过滤。

用下面的调用语句设置纹理收缩的最近相邻过滤模式：

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

最近相邻过滤模式的颜色计算过程与纹理伸展的最近相邻过滤模式相同。纹理坐标的颜色由最接近纹素的颜色决定。当每个像素对应的区域覆盖很多纹素时，这种过滤模式会产生锯齿现象。

纹理收缩可以使用的另一个基本过滤模式是线性过滤。下面的代码为收缩纹理设置线性过滤模式：

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

纹理收缩线性过滤与纹理伸展线性过滤的处理过程完全相同。将离纹理坐标最近的4个纹素的加权平均值作为此纹理坐标的颜色。线性过滤得到的效果略好于最近相邻过滤，但是当一个像素对应不止4个纹素时，这种过滤模式也会产生锯齿问题。

### Mip映射纹理

当一个像素对应很多纹素时，不管是最近相邻过滤还是线性过滤都会产生锯齿问题。这个问题的一个解决方法是使用一个较小尺寸的纹理，这样每个像素不会对应很多纹素。但是对于场景中靠近观察者的对象，为了避免出现纹理伸展现象，你可能希望纹理具有较高的分辨率，纹理伸展会使纹理对象看起来有块状效果或模糊效果。

Mip映射是这个问题的解决方法。除了基础纹理（即零级纹理）外，Mip映射使用一系列的较小纹理。每个新纹理都是系列中前一个纹理的一半大小。这一系列纹理形成一个纹理链，即为Mip映射纹理链。与只应用基础纹理的情形相比，一个完整的Mip映射链大约要多占用1/3的内存空间。

纹理不必是正方形，但是纹理链上的纹理必须一直继续到最后一个纹理的大小为1×1为止。作为一个示例，Mip映射纹理链可以包含大小为256×256、128×128、64×64、32×32、16×16，8×8、4×4、2×2、1×1的纹理。下图显示了Mip映射链的图像。



在WebGL中应用Mip映射纹理的最简单方法是让WebGL自动生成Mip映射纹理链。我们只需要载入基础纹理作为零级纹理，然后调用gl.generateMipmap()方法，WebGL会自动生成整个Mip映射纹理链。以下是生成一个2D纹理的典型代码：

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

gl.generateMipmap(gl.TEXTURE\_2D);

另一个方法是用一个图像编辑工具或专门为生成Mip映射纹理而设计的纹理工具，离线生成Mip映射链上的全部纹理图像。然后，在生成Mip映射链上的全部图像后，用gl.texImage2D()方法载入每幅图像，这个方法的第二个参数定义了图像的Mip映射级。以下代码载入Mip映射0级至Mip映射4级：

gl.texImage2D(gl.TEXTURE\_2D,0,gl.RGBA,gl.RGBA,gl.UNSIGNED\_BYTE,imageLevel\_0) ;

gl.texImage2D(gl.TEXTURE\_2D,0,gl.RGBA,gl.RGBA,gl.UNSIGNED\_BYTE,imageLevel\_1) ;

gl.texImage2D(gl.TEXTURE\_2D,0,gl.RGBA,gl.RGBA,gl.UNSIGNED\_BYTE,imageLevel\_2) ;

gl.texImage2D(gl.TEXTURE\_2D,0,gl.RGBA,gl.RGBA,gl.UNSIGNED\_BYTE,imageLevel\_3) ;

gl.texImage2D(gl.TEXTURE\_2D,0,gl.RGBA,gl.RGBA,gl.UNSIGNED\_BYTE,imageLevel\_4) ;

可以看出，第二个方法需要较多的操作。但是，如果对Mip映射链上的低分辨率图像有特别的要求，则会知道这种方法是很有用的。

### 如何选择纹理过滤模式

当我们为自己的纹理创建一个完整的Mip映射纹理链后，除了gl.NEAREST和gl.LINEAR这两个普通的过滤模式外，我们还可以为纹理收缩选择4种新过滤模式。下面列出所有可以使用的过滤模式并对它们逐一作详细的介绍：

* gl.NEAREST——将最近相邻过滤模式应用于基础纹理，这是指使用最接近纹素的颜色作为纹理坐标的颜色。
* gl.LINEAR——将线性过滤（也称为双线性过滤）应用于几次基拙纹理，这是指纹理坐标的颜色是由相邻4个纹素的颜色的加权平均值得到的。
* gl.NEAREST\_MIPMAP\_NEAREST——选择最近的Mip映射级，并且在这个Mip映射级中使用最近相邻过滤。
* gl.NEAREST\_MIPMAP\_LINEAR——选择两个最近的Mip映射级，然后在每个Mip映射级中使用最近相邻过滤得到两个中间结果，对这两个中间结果进行线性插值得到纹理坐标的最终颜色。
* gl.LINEAR\_MIPMAP\_NEAREST——选择最近的Mip映射级，然后在每个Mip映射级中选择线性过滤。
* gl.LINEAR\_MIPMAP\_LINEAR——选择两个最近的Mip映射级，在这两个Mip映射级中，用线性过滤得到两个中间结果，对这两个结果使用线性插值，得到最终值。这个过滤模式在所有过滤中生成最好的过滤效果。有时也称为三线性过滤。

可以看出，4个Mip映射纹理过滤模式的名称都采用gl.A\_MIPMAP\_B的形式，其中A和B为NEAREST或LINEAR。这里的A定义在一种Mip映射级中使用纹素的方式。如果A为NEAREST，则表示在Mip映射级中使用最近相邻过滤。如果A为LINEAR，则表示在Mip映射级中使用线性插值计算。

这里的B规定是使用单个Mip映射级还是使用两个最近的Mip映射级。如果B是NEAREST，则使用最近的Mip映射级。如果B是LINEAR，则表示使用两个最近的Mip映射级，在这两个Mip映射级中，由A规定过滤模式。然后把两个Mip映射级的结果进行线性插值得到最终的结果。

虽然使用Mip映射纹理的主要理由是它的视觉效果，但是使用这种纹理还可以得到更好的性能。原因是，当使用Mip映射纹理时，一般情形下缓存的利用率会比较高，因为使用纹理收缩并使用比较“粗糙”的纹理时，纹素的读取往往发生在内存中比较靠近的单元中。

如果不用Mip映射纹理，则在收缩纹理时，需要从内存中相隔很远的单元读取纹素。这会降低需要读取的纹素已经出现在靠近GPU的缓存中的可能性。

如果用gl.LINEAR\_MIPMAP\_LINEAR参数定义三线性过滤模式，则比起gl．LINEAR\_MIPMAP\_NEAREST等过滤模式，GPU需要执行更多的运算，需要从内存中读取更多的纹素。最好在不同的设备上测试不同的过滤模式，选择一种性能和质量都比较好的过滤模式。