# 第7课 键盘输入

本节课要介绍的内容是：

* 如何响应用户的键盘输入；
* 使用不同的纹理包装模式。

程序截图如下：



为了在WebGL应用程序中处理用户的输入，需要掌握JavaScript的事件处理机制。事件处理的基本思想是当浏览器中发生了某个重要的事情时，浏览器就创建并发送一个事件。用户的应用程序可以用事件处理程序侦听这些事件。

浏览器可以创建很多不同类型的事件。与用户输入有关的事件有许多。比较有用的事件有：

* keydown
* keypress
* keyup
* mousedown
* mouseup
* mousemove

处理Web浏览器的事件有两种主要方法：

* DOM Level 0基本事件处理
* DOM Level 2高级事件处理

我们的示例使用的是DOM Level 2高级事件处理。

## DOM Level 2高级事件处理

我们可以用addEventListener()方法注册一个元素的事件侦听程序。对于DOM Level 0模型，事件发送给在其上发生此事件的文档元素。如果这个元素已注册了一个事件处理程序，就执行这个事件处理程序。

DOM Level 2模型要复杂得多。它有一个事件传播过程，此过程有3个阶段：

* 事件捕获阶段
* 执行位于目标结点上的处理程序
* 事件冒泡阶段

如下图所示：



**DOM Level 2模型的事件传播过程**

在第一个阶段，即事件捕获阶段，事件从DOM树的顶端结点开始，即从文档对象开始，然后往下向目标点传播。如果目标点的任何祖先结点已经注册一个事件处理程序并激活了它的捕获功能，就会在事件的捕获阶段执行这个处理程序。调用addEventListener()函数并把它的最后个参数设置true，就可以注册一个事件处理程序并激活它的捕获功能。

第二个阶段执行位于目标结点上的的事件处理程序，这与DOM Level 0事件模型的行为相似。

第三个阶段（最后一个阶段）是事件冒泡阶段。从某种意义上讲，它与捕获阶段相反。在事件冒泡阶段，事件向上传播直到DOM树的顶部，即到文档对象为止。需要指出的是，并非所有事件都向上冒泡。然而在一般情况下，低层事件，如mousedown、mouseup和mousemove，都要向上冒泡。在冒泡阶段，任何注册在目标结点的祖先结点上的事件处理程序都会被触发。

在我们的示例中，第三个参数设置false，这是因为我们对捕获阶段不感兴趣。

## 键盘输入

键盘输入的事件处理程序因浏览器供应商和操作系统不同而有所区别。但是是有些基本的内容对于绝大多数浏览器具有共性。一般而言，当按下一个字母数字键时，会引发3个键盘事件：

* keydown
* keypress
* keyup

当按下一个字母数字键时，首先引发一个keydown事件，紧随其后是keypress事件。当释放这个键时，引发一个keyup事件。

实际上，keydown和keyup事件不同于keypress事件。如果我们能够分清键和字符的区别，则比较容易理解它们之间的差别。我们可以把一个键看成设备上的物理键，而字符是我们按下一个物理键后显示的符号（如A、a、b、@、$、%等）。

一般而言，keydown和keyup事件表示物理键被按下或释放，而keypress事件代表输入的字符。当用户想知道按下哪个键或按下的键对应哪个字符时，则键盘事件的两个属性会令用户感兴趣，它们是：

* keyCode——它表示一个虚拟键码（简称虚拟码），给我们提供了用户按下哪个键的信息。虚拟键码对应键的大写字母版本的ASCII码。例如，如果按下标有A的键，则生成65的虚拟键码，而不管是否按下大写字母锁定键。
* charCode——代表最后得到的字符的ASCII码。

当利用键盘输入控制WebGL场景中的对象时，更重要的是要知道按下的是哪个键，而不是这个键对应的字符。例如，在一场赛车游戏中，通常我们需要知道用户按的是油门还是刹车。在飞行模拟游戏中，我们需要知道用户是否按发射火箭的键。

由前所述，在WebGL应用程序中，对于keydown和keyup事件，应该使用它们的keyCode属性。

因此在startup()函数中注册键盘事件：

function startup() {

 document.addEventListener('keydown', handleKeyDown, false);

 document.addEventListener('keyup', handleKeyUp, false);

 document.addEventListener('keypress', handleKeyPress, false);

}

## 处理单键按下和多键同时按下事件

有时我们希望能够在自己的应用程序中同时使用两个键或多个键组合实现某个功能。这在游戏中很常见。

假设我们正在用WebGL设计一款赛车游戏，我们希望用个键表示油门，用其他两个键控制方向盘。更进一步假设这款赛车游戏很不一般，它允许游戏玩家向对手发射导弹，因此还需要用另一个键发射导弹。此外，游戏的开发人员可能还希望游戏玩家在操控方向盘的同时能够按下油门，或者能够发射导弹。

下面这段代码中，事件处理程序handleKeyDown()是为keydown事件注册的，handleKeyUp()为keyup事件注册的，handleKeyPress()是为keyfress事件注册的。

在handleKeyDown()函数中，调用个静态方法StringfromCharCode()，把keyCode的ASCII码转换为一个字符。直接把keyCode值与R的ASCII码（它为82）相比较也是完全可以的，但是利用String.fromChar- Code()，我们在读写这段代码时就不需要知道R的ASCII值。

为了能够处理游戏玩家同时按下几个键的情况，开发人员需要记住哪些键会被同时按下。这只需定义一个变量listOfPressedKeys保存这些键的值，然后这个列表用在handlePressedDownKeys()函数中。一般而言，这个函数需要每帧调用一次，通常是在场景绘制之前调用。在这个函数中，要检查当前哪几个键被同时按下，根据检查结果采取不同的操作。

完整的代码如下：

var listOfPressedKeys = {}; // 保存目前已按下的键的对象

var infoWrapS; // 纹理s方向的包装模式

var infoWrapT; // 纹理t方向的包装模式

var currentWrapS= 0;

var currentWrapT= 0;

function handleKeyDown(event) {

 listOfPressedKeys[event.keyCode] = true;

 // 按R键恢复初始状态

 if (String.fromCharCode(event.keyCode) == "R") {

 xRot = 0;

 yRot = 0;

 xSpeed = 0;

 ySpeed = 0;

 z = -2.5;

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

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.REPEAT);

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.REPEAT);

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

 }

 if (String.fromCharCode(event.keyCode) == "S") {

 currentWrapS += 1;

 if (currentWrapS == 3) {

 currentWrapS = 0;

 }

 var info;

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

 if (currentWrapS == 0) {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.REPEAT);

 info = "REPEAT";

 }

 if (currentWrapS == 1) {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.MIRRORED\_REPEAT);

 info = "MIRRORED\_REPEAT";

 }

 if (currentWrapS == 2)

 {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.CLAMP\_TO\_EDGE);

 info = "CLAMP\_TO\_EDGE";

 }

 infoWrapS.innerHTML = info;

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

 }

 if (String.fromCharCode(event.keyCode) == "T") {

 currentWrapT += 1;

 if (currentWrapT == 3) {

 currentWrapT = 0;

 }

 var info;

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

 if (currentWrapT == 0) {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.REPEAT);

 info = "REPEAT";

 }

 if (currentWrapT == 1) {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.MIRRORED\_REPEAT);

 info = "MIRRORED\_REPEAT";

 }

 if (currentWrapT == 2) {

 gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.CLAMP\_TO\_EDGE);

 info = "CLAMP\_TO\_EDGE";

 }

 infoWrapT.innerHTML = info;

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

 }

}

function handleKeyUp(event) {

 listOfPressedKeys[event.keyCode] = false;

}

function handleKeyPress(event) {}

function handlePressedDownKeys() {

 if (listOfPressedKeys[33]) {

 // Page Up

 z -= 0.05;

 }

 if (listOfPressedKeys[34]) {

 // Page Down

 z += 0.05;

 }

 if (listOfPressedKeys[37]) {

 // 向左箭头

 ySpeed -= 1;

 }

 if (listOfPressedKeys[39]) {

 // 向右箭头

 ySpeed += 1;

 }

 if (listOfPressedKeys[38]) {

 // 向上箭头

 xSpeed -= 1;

 }

 if (listOfPressedKeys[40]) {

 // 向下箭头

 xSpeed += 1;

 }

}

## 纹理坐标包装模式

在上面的代码中，我们通过按下“S”或“T”键可以切换纹理包装模式。

通常纹理的左下角的坐标为(0,0)，右上角的坐标为(1,1)。但坐标可以取超过这个范围的值。例如，右上角坐标取为(3,3)，则几何体会出现3×3张相同的纹理。

在本例的代码中，这种效果不是通过定义顶点的纹理坐标实现的，而是在顶点着色器中实现的，两种方法都能够获得相同的效果。顶点着色器中的main()代码如下：

void main(void) {

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

 // 乘以3将纹理左右平铺三次，减1是让纹理位于中央

 vTextureCoords = (aTextureCoords \* 3.0) - vec2(1.0, 1.0);

}

利用gl.texParameter()方法，可以分别为s方和t方向定义纹理的包装模式。这个方法的第二个参数可以取gl.TEXTURE\_WRAP\_S或gl.TEXTURE\_WRAP\_T两个值之一。该方法的第三个参数定义包装模式，它们是：

* gl.REPEAT
* gl.MIRRORED\_REPEAT
* gl.CLAMP\_TO\_EDGE

如果没有显示地定义一个包装模式，则WebGL的默认包装模式就是gl.REPEAT。在这个模式中，WebGL在纹理坐标的每个整数边界上重复这个纹理。

gl.MIRRORED\_REPEAT包装模式与gl.REPEAT包装模式相似，但当纹理坐标的整数部分是偶数时，它镜像原来的图像。这种模式常用来在一个表面上产生无缝的平铺效果。

应用gl.CLAMP\_TO\_EDGE包装模式时，所有纹理坐标都嵌在[0,1]范围内。在这个范围之外的纹理坐标将由纹理的最接近边界采样得到。

三种包装方式的程序截图如下：



**从左至右分别是gl.REPEAT、gl.MIRRORED\_REPEAT、gl.CLAMP\_TO\_EDGE**