

第 15 章

使用 Canvas 绘图

本章内容

- 理解<canvas>元素
- 绘制简单的 2D 图形
- 使用 WebGL 绘制 3D 图形

不用说，HTML5 添加的最受欢迎的功能就是<canvas>元素。这个元素负责在页面中设定一个区域，然后就可以通过 JavaScript 动态地在这个区域中绘制图形。<canvas>元素最早是由苹果公司推出的，当时主要用在其 Dashboard 微件中。很快，HTML5 加入了这个元素，主流浏览器也迅速开始支持它。IE9+、Firefox 1.5+、Safari 2+、Opera 9+、Chrome、iOS 版 Safari 以及 Android 版 WebKit 都在某种程度上支持<canvas>。

与浏览器环境中的其他组件类似，<canvas>由几组 API 构成，但并非所有浏览器都支持所有这些 API。除了具备基本绘图能力的 2D 上下文，<canvas>还建议了一个名为 WebGL 的 3D 上下文。目前，支持该元素的浏览器都支持 2D 上下文及文本 API，但对 WebGL 的支持还不够好。由于 WebGL 还是实验性的，因此要得到所有浏览器支持还需要很长一段时间。Firefox 4+和 Chrome 支持 WebGL 规范的早期版本，但一些老版本的浏览器，比如 Windows XP，由于缺少必要的绘图驱动程序，即便安装了这两款浏览器也无济于事。

15.1 基本用法

要使用<canvas>元素，必须先设置其 width 和 height 属性，指定可以绘图的区域大小。出现在开始和结束标签中的内容是后备信息，如果浏览器不支持<canvas>元素，就会显示这些信息。下面就是<canvas>元素的例子。

```
<canvas id="drawing" width=" 200" height="200">A drawing of something.</canvas>
```

与其他元素一样，<canvas>元素对应的 DOM 元素对象也有 width 和 height 属性，可以随意修改。而且，也能通过 CSS 为该元素添加样式，如果不添加任何样式或者不绘制任何图形，在页面中是看不到该元素的。

要在这块画布（canvas）上绘图，需要取得绘图上下文。而取得绘图上下文对象的引用，需要调用 getContext() 方法并传入上下文的名字。传入"2d"，就可以取得 2D 上下文对象。

```
var drawing = document.getElementById("drawing");  
  
//确定浏览器支持<canvas>元素  
if (drawing.getContext){
```

```
var context = drawing.getContext("2d");  
//更多代码  
}
```

在使用<canvas>元素之前，首先要检测 `getContext()` 方法是否存在，这一步非常重要。有些浏览器会为 HTML 规范之外的元素创建默认的 HTML 元素对象^①。在这种情况下，即使 `drawing` 变量中保存着一个有效的元素引用，也检测不到 `getContext()` 方法。

使用 `toDataURL()` 方法，可以导出在<canvas>元素上绘制的图像。这个方法接受一个参数，即图像的 MIME 类型格式，而且适合用于创建图像的任何上下文。比如，要取得画布中的一幅 PNG 格式的图像，可以使用以下代码。



```
var drawing = document.getElementById("drawing");  
  
//确定浏览器支持<canvas>元素  
if (drawing.getContext){  
  
    //取得图像的数据 URI  
    var imgURI = drawing.toDataURL("image/png");  
  
    //显示图像  
    var image = document.createElement("img");  
    image.src = imgURI;  
    document.body.appendChild(image);  
}
```

2DDataUriExample01.htm

默认情况下，浏览器会将图像编码为 PNG 格式（除非另行指定）。Firefox 和 Opera 也支持基于 "image/jpeg" 参数的 JPEG 编码格式。由于这个方法是后来才追加的，所以支持<canvas>的浏览器也是在较新的版本中才加入了对它的支持，比如 IE9、Firefox 3.5 和 Opera 10。



如果绘制到画布上的图像源自不同的域，`toDataURL()` 方法会抛出错误。本章后面还将介绍更多相关内容。

15.2 2D 上下文

使用 2D 绘图上下文提供的方法，可以绘制简单的 2D 图形，比如矩形、弧线和路径。2D 上下文的坐标开始于<canvas>元素的左上角，原点坐标是(0,0)。所有坐标值都基于这个原点计算，x 值越大表示越靠右，y 值越大表示越靠下。默认情况下，`width` 和 `height` 表示水平和垂直两个方向上可用的像素数目。

15.2.1 填充和描边

2D 上下文的两种基本绘图操作是填充和描边。填充，就是用指定的样式（颜色、渐变或图像）填充图形；描边，就是只在图形的边缘画线。大多数 2D 上下文操作都会细分为填充和描边两个操作，而

^① 假设你想在 Firefox 3 中使用<canvas>元素。虽然浏览器会为该标签创建一个 DOM 对象，而且也可以引用它，但这个对象中并没有 `getContext()` 方法。（据作者回复）

操作的结果取决于两个属性：fillStyle 和 strokeStyle。

这两个属性的值可以是字符串、渐变对象或模式对象，而且它们的默认值都是"#000000"。如果为它们指定表示颜色的字符串值，可以使用 CSS 中指定颜色值的任何格式，包括颜色名、十六进制码、rgb、rgba、hsl 或 hsla。举个例子：

```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){


    var context = drawing.getContext("2d");
    context.strokeStyle = "red";
    context.fillStyle = "#0000ff";
}
```

以上代码将 strokeStyle 设置为 red (CSS 中的颜色名)，将 fillStyle 设置为#0000ff (蓝色)。然后，所有涉及描边和填充的操作都将使用这两个样式，直至重新设置这两个值。如前所述，这两个属性的值也可以是渐变对象或模式对象。本章后面会讨论这两种对象。

15.2.2 绘制矩形

矩形是唯一一种可以直接在 2D 上下文中绘制的形状。与矩形有关的方法包括 fillRect()、strokeRect() 和 clearRect()。这三个方法都能接收 4 个参数：矩形的 x 坐标、矩形的 y 坐标、矩形宽度和矩形高度。这些参数的单位都是像素。

首先，fillRect() 方法在画布上绘制的矩形会填充指定的颜色。填充的颜色通过 fillStyle 属性指定，比如：



```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){

    var context = drawing.getContext("2d");

    /*
     * 根据 Mozilla 的文档
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
     */

    //绘制红色矩形
    context.fillStyle = "#ff0000";
    context.fillRect(10, 10, 50, 50);

    //绘制半透明的蓝色矩形
    context.fillStyle = "rgba(0,0,255,0.5)";
    context.fillRect(30, 30, 50, 50);
}
```

2DFillRectExample01.htm

以上代码首先将 fillStyle 设置为红色，然后从(10,10)处开始绘制矩形，矩形的宽和高均为 50 像素。然后，通过 rgba() 格式再将 fillStyle 设置为半透明的蓝色，在第一个矩形上面绘制第二个矩

形。结果就是可以透过蓝色的矩形看到红色的矩形（见图 15-1）。

`strokeRect()` 方法在画布上绘制的矩形会使用指定的颜色描边。描边颜色通过 `strokeStyle` 属性指定。比如：

```
var drawing = document.getElementById("drawing");  
  
// 确定浏览器支持<canvas>元素  
if (drawing.getContext){  
  
    var context = drawing.getContext("2d");  
  
    /*  
     * 根据 Mozilla 的文档  
     * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage  
     */  
  
    // 绘制红色描边矩形  
    context.strokeStyle = "#ff0000";  
    context.strokeRect(10, 10, 50, 50);  
  
    // 绘制半透明的蓝色描边矩形  
    context.strokeStyle = "rgba(0,0,255,0.5)";  
    context.strokeRect(30, 30, 50, 50);  
}
```

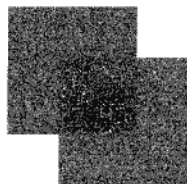


图 15-1

[2DStrokeRectExample01.htm](#)

以上代码绘制了两个重叠的矩形。不过，这两个矩形都只有框线，内部并没有填充颜色（见图 15-2）。

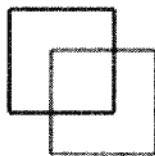


图 15-2



描边线条的宽度由 `linewidth` 属性控制，该属性的值可以是任意整数。另外，通过 `lineCap` 属性可以控制线条末端的形状是平头、圆头还是方头（“butt”、“round”或“square”），通过 `lineJoin` 属性可以控制线条相交的方式是圆交、斜交还是斜接（“round”、“bevel”或“miter”）。

最后，`clearRect()` 方法用于清除画布上的矩形区域。本质上，这个方法可以把绘制上下文中的某一矩形区域变透明。通过绘制形状然后再清除指定区域，就可以生成有意思的效果，例如把某个形状切掉一块。下面看一个例子。

```
var drawing = document.getElementById("drawing");  
  
// 确定浏览器支持<canvas>元素  
if (drawing.getContext){  
  
    var context = drawing.getContext("2d");
```



```
/*
 * 根据 Mozilla 的文档
 * http://developer.mozilla.org/en/docs/Canvas_tutorial:Basic_usage
 */

//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//绘制半透明的蓝色矩形
context.fillStyle = "rgba(0,0,255,0.5)";
context.fillRect(30, 30, 50, 50);

//在两个矩形重叠的地方清除一个小矩形
context.clearRect(40, 40, 10, 10);
}
```

2DClearRectExample01.htm

如图 15-3 所示, 两个填充矩形重叠在一起, 而重叠的地方又被清除了一个小矩形区域。



图 15-3

15.2.3 绘制路径

2D 绘制上下文支持很多在画布上绘制路径的方法。通过路径可以创造出复杂的形状和线条。要绘制路径, 首先必须调用 `beginPath()` 方法, 表示要开始绘制新路径。然后, 再通过调用下列方法来实际地绘制路径。

- `arc(x, y, radius, startAngle, endAngle, counterclockwise)`: 以 (x, y) 为圆心绘制一条弧线, 弧线半径为 `radius`, 起始和结束角度 (用弧度表示) 分别为 `startAngle` 和 `endAngle`。最后一个参数表示 `startAngle` 和 `endAngle` 是否按逆时针方向计算, 值为 `false` 表示按顺时针方向计算。
- `arcTo(x1, y1, x2, y2, radius)`: 从上一点开始绘制一条弧线, 到 $(x2, y2)$ 为止, 并且以给定的半径 `radius` 穿过 $(x1, y1)$ 。
- `bezierCurveTo(c1x, c1y, c2x, c2y, x, y)`: 从上一点开始绘制一条曲线, 到 (x, y) 为止, 并且以 $(c1x, c1y)$ 和 $(c2x, c2y)$ 为控制点。
- `lineTo(x, y)`: 从上一点开始绘制一条直线, 到 (x, y) 为止。
- `moveTo(x, y)`: 将绘图游标移动到 (x, y) , 不画线。
- `quadraticCurveTo(cx, cy, x, y)`: 从上一点开始绘制一条二次曲线, 到 (x, y) 为止, 并且以 (cx, cy) 作为控制点。
- `rect(x, y, width, height)`: 从点 (x, y) 开始绘制一个矩形, 宽度和高度分别由 `width` 和 `height` 指定。这个方法绘制的是矩形路径, 而不是 `strokeRect()` 和 `fillRect()` 所绘制的独立的形状。

创建了路径后, 接下来有几种可能的选择。如果想绘制一条连接到路径起点的线条, 可以调用 `closePath()`。如果路径已经完成, 你想用 `fillStyle` 填充它, 可以调用 `fill()` 方法。另外, 还可以调用 `stroke()` 方法对路径描边, 描边使用的是 `strokeStyle`。最后还可以调用 `clip()`, 这个方法可以在路径上创建一个剪切区域。

下面看一个例子，即绘制一个不带数字的时钟表盘。



```
var drawing = document.getElementById("drawing");  
  
// 确定浏览器支持<canvas>元素  
if (drawing.getContext){  
  
    var context = drawing.getContext("2d");  
  
    // 开始路径  
    context.beginPath();  
  
    // 绘制外圆  
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);  
  
    // 绘制内圆  
    context.moveTo(194, 100);  
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);  
  
    // 绘制分针  
    context.moveTo(100, 100);  
    context.lineTo(100, 15);  
  
    // 绘制时针  
    context.moveTo(100, 100);  
    context.lineTo(35, 100);  
  
    // 描边路径  
    context.stroke();  
}
```

[2DPathExample01.htm](#)

这个例子使用 `arc()` 方法绘制了两个圆形：一个外圆和一个内圆，构成了表盘的边框。外圆的半径是 99 像素，圆心位于点(100,100)，也是画布的中心点。为了绘制一个完整的圆形，我们从 0 弧度开始，绘制 2π 弧度（通过 `Math.PI` 来计算）。在绘制内圆之前，必须把路径移动到外圆上的某一点，以避免绘制出多余的线条。第二次调用 `arc()` 使用了小一点的半径，以便创造边框的效果。然后，组合使用 `moveTo()` 和 `lineTo()` 方法来绘制时针和分针。最后一步是调用 `stroke()` 方法，这样才能把图形绘制到画布上，如图 15-4 所示。

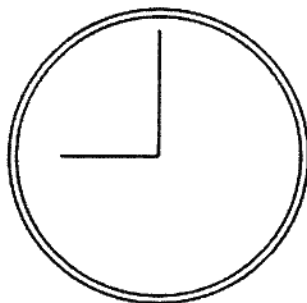


图 15-4

在 2D 绘图上下文中，路径是一种主要的绘图方式，因为路径能为要绘制的图形提供更多控制。由于路径的使用很频繁，所以就有一个名为 `isPointInPath()` 的方法。这个方法接收 `x` 和 `y` 坐标作为参数，用于在路径被关闭之前确定画布上的某一点是否位于路径上，例如：

```
if (context.isPointInPath(100, 100)){  
    alert("Point (100, 100) is in the path.");  
}
```

2D 上下文中的路径 API 已经非常稳定，可以利用它们结合不同的填充和描边样式，绘制出非常复杂的图形来。

15.2.4 绘制文本

文本与图形总是如影随形。为此，2D 绘图上下文也提供了绘制文本的方法。绘制文本主要有两个方法：`fillText()` 和 `strokeText()`。这两个方法都可以接收 4 个参数：要绘制的文本字符串、`x` 坐标、`y` 坐标和可选的最大像素宽度。而且，这两个方法都以下列 3 个属性为基础。

- `font`: 表示文本样式、大小及字体，用 CSS 中指定字体的格式来指定，例如 "10px Arial"。
- `textAlign`: 表示文本对齐方式。可能的值有 "start"、"end"、"left"、"right" 和 "center"。建议使用 "start" 和 "end"，不要使用 "left" 和 "right"，因为前两者的意思更稳妥，能同时适合从左到右和从右到左显示（阅读）的语言。
- `textBaseline`: 表示文本的基线。可能的值有 "top"、"hanging"、"middle"、"alphabetic"、"ideographic" 和 "bottom"。

这几个属性都有默认值，因此没有必要每次使用它们都重新设置一遍值。`fillText()` 方法使用 `fillStyle` 属性绘制文本，而 `strokeText()` 方法使用 `strokeStyle` 属性为文本描边。相对来说，还是使用 `fillText()` 的时候更多，因为该方法模仿了在网页中正常显示文本。例如，下面的代码在前一节创建的表盘上方绘制了数字 12：

```
context.font = "bold 14px Arial";  
context.textAlign = "center";  
context.textBaseline = "middle";  
context.fillText("12", 100, 20);
```

2D TextExample01.htm

结果如图 15-5 所示。

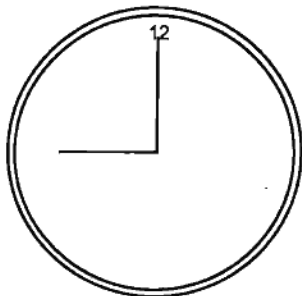


图 15-5

因为这里把 `textAlign` 设置为 "center", 把 `textBaseline` 设置为 "middle", 所以坐标 (100,20) 表示的是文本水平和垂直中点的坐标。如果将 `textAlign` 设置为 "start", 则 x 坐标表示的是文本左端的位置 (从左到右阅读的语言); 设置为 "end", 则 x 坐标表示的是文本右端的位置 (从左到右阅读的语言)。例如:

```
// 正常
context.font = "bold 14px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("12", 100, 20);

// 起点对齐
context.textAlign = "start";
context.fillText("12", 100, 40);

// 终点对齐
context.textAlign = "end";
context.fillText("12", 100, 60);
```

[2DTextExample02.htm](#)

这一回绘制了三个字符串 "12", 每个字符串的 x 坐标值相同, 但 `textAlign` 值不同。另外, 后两个字符串的 y 坐标依次增大, 以避免相互重叠。结果如图 15-6 所示。

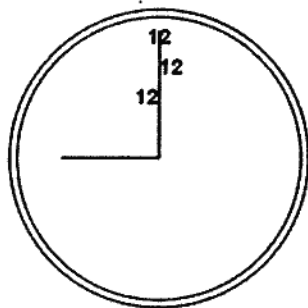


图 15-6

表盘中的分针恰好位于正中间, 因此文本的水平对齐方式如何变化也能够一目了然。类似地, 修改 `textBaseline` 属性的值可以调整文本的垂直对齐方式: 值为 "top", y 坐标表示文本顶端; 值为 "bottom", y 坐标表示文本底端; 值为 "hanging"、"alphabetic" 和 "ideographic", 则 y 坐标分别指向字体的特定基线坐标。

由于绘制文本比较复杂, 特别是需要把文本控制在某一区域中的时候, 2D 上下文提供了辅助确定文本大小的方法 `measureText()`。这个方法接收一个参数, 即要绘制的文本; 返回一个 `TextMetrics` 对象。返回的对象目前只有一个 `width` 属性, 但将来还会增加更多度量属性。

`measureText()` 方法利用 `font`、`textAlign` 和 `textBaseline` 的当前值计算指定文本的大小。比如, 假设你想在一个 140 像素宽的矩形区域中绘制文本 `Hello world!`, 下面的代码从 100 像素的字体大小开始递减, 最终会找到合适的字体大小。



```
var fontSize = 100;
context.font = fontSize + "px Arial";

while(context.measureText("Hello world!").width > 140){
    fontSize--;
    context.font = fontSize + "px Arial";
}

context.fillText("Hello world!", 10, 10);
context.fillText("Font size is " + fontSize + "px", 10, 50);
```

15

2DTextExample03.htm

前面提到过, `fillText` 和 `strokeText()` 方法都可以接收第四个参数, 也就是文本的最大像素宽度。不过, 这个可选的参数尚未得到所有浏览器支持 (最早支持它的是 Firefox 4)。提供这个参数后, 调用 `fillText()` 或 `strokeText()` 时如果传入的字符串大于最大宽度, 则绘制的文本字符的高度正确, 但宽度会收缩以适应最大宽度。图 15-7 展示了这个效果。

字体大小为26像素

图 15-7

绘制文本还是相对比较复杂的操作, 因此支持<canvas>元素的浏览器也并未完全实现所有与绘制文本相关的 API。

15.2.5 变换

通过上下文的变换, 可以把处理后的图像绘制到画布上。2D 绘制上下文支持各种基本的绘制变换。创建绘制上下文时, 会以默认值初始化变换矩阵, 在默认的变换矩阵下, 所有处理都按描述直接绘制。为绘制上下文应用变换, 会导致使用不同的变换矩阵应用处理, 从而产生不同的结果。

可以通过如下方法来修改变换矩阵。

- ❑ `rotate(angle)`: 围绕原点旋转图像 `angle` 弧度。
- ❑ `scale(scaleX, scaleY)`: 缩放图像, 在 `x` 方向乘以 `scaleX`, 在 `y` 方向乘以 `scaleY`。 `scaleX` 和 `scaleY` 的默认值都是 1.0。
- ❑ `translate(x, y)`: 将坐标原点移动到 `(x, y)`。执行这个变换之后, 坐标 `(0,0)` 会变成之前由 `(x, y)` 表示的点。
- ❑ `transform(m1_1, m1_2, m2_1, m2_2, dx, dy)`: 直接修改变换矩阵, 方式是乘以如下矩阵。

| | | |
|-------------------|-------------------|-----------------|
| <code>m1_1</code> | <code>m1_2</code> | <code>dx</code> |
| <code>m2_1</code> | <code>m2_2</code> | <code>dy</code> |
| 0 | 0 | 1 |
- ❑ `setTransform(m1_1, m1_2, m2_1, m2_2, dx, dy)`: 将变换矩阵重置为默认状态, 然后再调用 `transform()`。

变换有可能很简单, 但也可能很复杂, 这都要视情况而定。比如, 就拿前面例子中绘制表针来说, 如果把原点变换到表盘的中心, 然后再绘制表针就容易多了。请看下面的例子。



```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){
```

```
var context = drawing.getContext("2d");

//开始路径
context.beginPath();

//绘制外圆
context.arc(100, 100, 99, 0, 2 * Math.PI, false);

//绘制内圆
context.moveTo(194, 100);
context.arc(100, 100, 94, 0, 2 * Math.PI, false);

//变换原点
context.translate(100, 100);

//绘制分针
context.moveTo(0, 0);
context.lineTo(0, -85);

//绘制时针
context.moveTo(0, 0);
context.lineTo(-65, 0);

//描边路径
context.stroke();
}
```

2DTransformExample01.htm

把原点变换到时钟表盘的中心点(100,100)后,在同一方向上绘制线条就变成了简单的数学问题了。所有数学计算都基于(0,0),而不是(100,100)。还可以更进一步,像下面这样使用 `rotate()` 方法旋转时钟的表针。



```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){

    var context = drawing.getContext("2d");

    //开始路径
    context.beginPath();

    //绘制外圆
    context.arc(100, 100, 99, 0, 2 * Math.PI, false);

    //绘制内圆
    context.moveTo(194, 100);
    context.arc(100, 100, 94, 0, 2 * Math.PI, false);

    //变换原点
    context.translate(100, 100);

    //旋转表针
    context.rotate(1);

    //绘制分针
```

```
context.moveTo(0,0);  
context.lineTo(0, -85);  
  
//绘制时针  
context.moveTo(0, 0);  
context.lineTo(-65, 0);  
  
//描边路径  
context.stroke();  
}
```

2DTransformExample01.htm

因为原点已经变换到了时钟表盘的中心点，所以旋转也是以该点为圆心的。结果就像是表针真地被固定在表盘中心一样，然后向右旋转了一定角度。结果如图 15-8 所示。

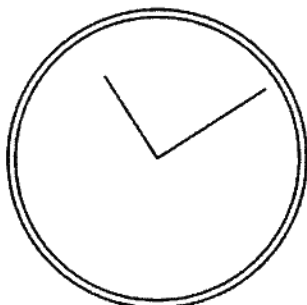


图 15-8

无论是刚才执行的变换，还是 `fillStyle`、`strokeStyle` 等属性，都会在当前上下文中一直有效，除非再对上下文进行什么修改。虽然没有什么办法把上下文中的一切都重置回默认值，但有两个方法可以跟踪上下文的状态变化。如果你知道将来还要返回某组属性与变换的组合，可以调用 `save()` 方法。调用这个方法后，当时的所有设置都会进入一个栈结构，得以妥善保管。然后可以对上下文进行其他修改。等想要回到之前保存的设置时，可以调用 `restore()` 方法，在保存设置的栈结构中向前返回一级，恢复之前的状态。连续调用 `save()` 可以把更多设置保存到栈结构中，之后再连续调用 `restore()` 则可以一级一级返回。下面来看一个例子。



```
context.fillStyle = "#ff0000";  
context.save();  
  
context.fillStyle = "#00ff00";  
context.translate(100, 100);  
context.save();  
  
context.fillStyle = "#0000ff";  
context.fillRect(0, 0, 100, 200); //从点(100,100)开始绘制蓝色矩形  
  
context.restore();  
context.fillRect(10, 10, 100, 200); //从点(110,110)开始绘制绿色矩形  
  
context.restore();
```

```
context.fillRect(0, 0, 100, 200); //从点(0,0)开始绘制红色矩形
```

[2DSaveRestoreExample01.htm](#)

首先, 将 `fillStyle` 设置为红色, 并调用 `save()` 保存上下文状态。接下来, 把 `fillStyle` 修改为绿色, 把坐标原点变换到(100,100), 再调用 `save()` 保存上下文状态。然后, 把 `fillStyle` 修改为蓝色并绘制蓝色的矩形。因为此时的坐标原点已经变了, 所以矩形的左上角坐标实际上是(100,100)。然后调用 `restore()`, 之后 `fillStyle` 变回了绿色, 因而第二个矩形就是绿色。之所以第二个矩形的起点坐标是(110,110), 是因为坐标位置的变换仍然起作用。再调用一次 `restore()`, 变换就被取消了, 而 `fillStyle` 也返回了红色。所以最后一个矩形是红色的, 而且绘制的起点是(0,0)。

需要注意的是, `save()` 方法保存的只是对绘图上下文的设置和变换, 不会保存绘图上下文的内容。

15.2.6 绘制图像

2D 绘图上下文内置了对图像的支持。如果你想把一幅图像绘制到画布上, 可以使用 `drawImage()` 方法。根据期望的最终结果不同, 调用这个方法时, 可以使用三种不同的参数组合。最简单的调用方式是传入一个 HTML `` 元素, 以及绘制该图像的起点的 `x` 和 `y` 坐标。例如:



```
var image = document.images[0];  
context.drawImage(image, 10, 10);
```

[2DDrawImageExample01.htm](#)

这两行代码取得了文档中的第一幅图像, 然后将它绘制到上下文中, 起点为(10,10)。绘制到画布上的图像大小与原始大小一样。如果你想改变绘制后图像的大小, 可以再多传入两个参数, 分别表示目标宽度和目标高度。通过这种方式来缩放图像并不影响上下文的变换矩阵。例如:

```
context.drawImage(image, 50, 10, 20, 30);
```

[2DDrawImageExample01.htm](#)

执行代码后, 绘制出来的图像大小会变成 20×30 像素。

除了上述两种方式, 还可以选择把图像中的某个区域绘制到上下文中。`drawImage()` 方法的这种调用方式总共需要传入 9 个参数: 要绘制的图像、源图像的 `x` 坐标、源图像的 `y` 坐标、源图像的宽度、源图像的高度、目标图像的 `x` 坐标、目标图像的 `y` 坐标、目标图像的宽度、目标图像的高度。这样调用 `drawImage()` 方法可以获得最多的控制。例如:

```
context.drawImage(image, 0, 10, 50, 50, 0, 100, 40, 60);
```

[2DDrawImageExample01.htm](#)

这行代码只会把原始图像的一部分绘制到画布上。原始图像的这一部分的起点为(0,10), 宽和高都是 50 像素。最终绘制到上下文中的图像的起点是(0,100), 而大小变成了 40×60 像素。

这种调用方式可以创造出很有意思的效果, 如图 15-9 所示。



图 15-9

除了给 `drawImage()` 方法传入 HTML `` 元素外, 还可以传入另一个 `<canvas>` 元素作为其第一个参数。这样, 就可以把另一个画布内容绘制到当前画布上。

结合使用 `drawImage()` 和其他方法, 可以对图像进行各种基本操作。而操作的结果可以通过 `toDataURL()` 方法获得^①。不过, 有一个例外, 即图像不能来自其他域。如果图像来自其他域, 调用 `toDataURL()` 会抛出一个错误。打个比方, 假如位于 `www.example.com` 上的页面绘制的图像来自于 `www.wrox.com`, 那当前上下文就会被认为“不干净”, 因而会抛出错误。

15.2.7 阴影

2D 上下文会根据以下几个属性的值, 自动为形状或路径绘制出阴影。

- `shadowColor`: 用 CSS 颜色格式表示的阴影颜色, 默认为黑色。
- `shadowOffsetX`: 形状或路径 *x* 轴方向的阴影偏移量, 默认为 0。
- `shadowOffsetY`: 形状或路径 *y* 轴方向的阴影偏移量, 默认为 0。
- `shadowBlur`: 模糊的像素数, 默认 0, 即不模糊。

这些属性都可以通过 `context` 对象来修改。只要在绘制前为它们设置适当的值, 就能自动产生阴影。例如:



```
var context = drawing.getContext("2d");

//设置阴影
context.shadowOffsetX = 5;
context.shadowOffsetY = 5;
context.shadowBlur = 4;
context.shadowColor = "rgba(0, 0, 0, 0.5)";

//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
```

① 请读者注意, 虽然本章至今一直在讨论 2D 绘图上下文, 但 `toDataURL()` 是 `Canvas` 对象的方法, 不是上下文对象的方法。

```
context.fillRect(30, 30, 50, 50);
```

2DFillRectShadowExample01.htm

两个矩形的阴影样式相同，结果如图 15-10 所示。

不同浏览器对阴影的支持有一些差异。IE9、Firefox 4 和 Opera 11 的行为最为规范，其他浏览器多多少少会有一些奇怪的现象，甚至根本不支持阴影。Chrome（直至第 10 版）不能正确地给描边的形状应用实心阴影。Chrome 和 Safari（直至第 5 版）在为带透明像素的图像应用阴影时也会有问题：不透明部分的下方本来是应有阴影的，但此时则一概不见了。Safari 也不能给渐变图形应用阴影，其他浏览器都可以。

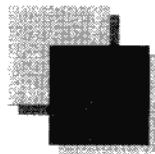


图 15-10

15.2.8 渐变

渐变由 CanvasGradient 实例表示，很容易通过 2D 上下文来创建和修改。要创建一个新的线性渐变，可以调用 createLinearGradient() 方法。这个方法接收 4 个参数：起点的 x 坐标、起点的 y 坐标、终点的 x 坐标、终点的 y 坐标。调用这个方法后，它就会创建一个指定大小的渐变，并返回 CanvasGradient 对象的实例。

创建了渐变对象后，下一步就是使用 addColorStop() 方法来指定色标。这个方法接收两个参数：色标位置和 CSS 颜色值。色标位置是一个 0（开始的颜色）到 1（结束的颜色）之间的数字。例如：



```
var gradient = context.createLinearGradient(30, 30, 70, 70);  
  
gradient.addColorStop(0, "white");  
gradient.addColorStop(1, "black");
```

2DFillRectGradientExample01.htm

此时，gradient 对象表示的是一个从画布上点(30,30)到点(70,70)的渐变。起点的色标是白色，终点的色标是黑色。然后就可以把 fillStyle 或 strokeStyle 设置为这个对象，从而使用渐变来绘制形状或描边：

```
//绘制红色矩形  
context.fillStyle = "#ff0000";  
context.fillRect(10, 10, 50, 50);  
  
//绘制渐变矩形  
context.fillStyle = gradient;  
context.fillRect(30, 30, 50, 50);
```

2DFillRectGradientExample01.htm

为了让渐变覆盖整个矩形，而不是仅应用到矩形的一部分，矩形和渐变对象的坐标必须匹配才行。以上代码会得到如图 15-11 所示的结果。

如果没有把矩形绘制到恰当的位置，那可能就只会显示部分渐变效果。例如：

```
context.fillStyle = gradient;
```



图 15-11


```
context.fillRect(50, 50, 50, 50);
```

[2DFillRectGradientExample02.htm](#)

这两行代码执行后得到的矩形只有左上角稍微有一点白色。这主要是因为矩形的起点位于渐变的中点位置，而此时渐变差不多已经结束了。由于渐变不重复，所以矩形的大部分区域都是黑色。确保渐变与形状对齐非常重要，有时候可以考虑使用函数来确保坐标合适。例如：

```
function createRectLinearGradient(context, x, y, width, height){  
    return context.createLinearGradient(x, y, x+width, y+height);  
}
```

[2DFillRectGradientExample03.htm](#)

这个函数基于起点的 x 和 y 坐标以及宽度和高度值来创建渐变对象，从而让我们可以在 `fillRect()` 中使用相同的值。

```
var gradient = createRectLinearGradient(context, 30, 30, 50, 50);  
  
gradient.addColorStop(0, "white");  
gradient.addColorStop(1, "black");  
  
//绘制渐变矩形  
context.fillStyle = gradient;  
context.fillRect(30, 30, 50, 50);
```

[2DFillRectGradientExample03.htm](#)

使用画布的时候，确保坐标匹配很重要，也需要一些技巧。类似 `createRectLinearGradient()` 这样的辅助方法可以让控制坐标更容易一些。

要创建径向渐变（或放射渐变），可以使用 `createRadialGradient()` 方法。这个方法接收 6 个参数，对应着两个圆的圆心和半径。前三个参数指定的是起点圆的圆心（ x 和 y ）及半径，后三个参数指定的是终点圆的圆心（ x 和 y ）及半径。可以把径向渐变想象成一个长圆桶，而这 6 个参数定义的正是这个桶的两个圆形开口的位置。如果把一个圆形开口定义得比另一个小一些，那这个圆桶就变成了圆锥体，而通过移动每个圆形开口的位置，就可达到像旋转这个圆锥体一样的效果。

如果想从某个形状的中心点开始创建一个向外扩散的径向渐变效果，就要将两个圆定义为同心圆。比如，就拿前面创建的矩形来说，径向渐变的两个圆的圆心都应该在 (55,55)，因为矩形的区域是从 (30,30) 到 (80,80)。请看代码：

```
var gradient = context.createRadialGradient(55, 55, 10, 55, 55, 30);  
  
gradient.addColorStop(0, "white");  
gradient.addColorStop(1, "black");  
  
//绘制红色矩形  
context.fillStyle = "#ff0000";  
context.fillRect(10, 10, 50, 50);  
  
//绘制渐变矩形  
context.fillStyle = gradient;  
context.fillRect(30, 30, 50, 50);
```

[2DFillRectGradientExample04.htm](#)

运行代码，会得到如图 15-12 所示的结果。

因为创建比较麻烦，所以径向渐变并不那么容易控制。不过，一般来说，让起点圆和终点圆保持为同心圆的情况比较多，这时候只要考虑给两个圆设置不同的半径就好了。

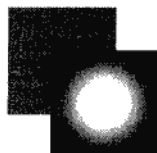


图 15-12

15.2.9 模式

模式其实就是重复的图像，可以用来填充或描边图形。要创建一个新模式，可以调用 `createPattern()` 方法并传入两个参数：一个 HTML `` 元素和一个表示如何重复图像的字符串。其中，第二个参数的值与 CSS 的 `background-repeat` 属性值相同，包括 `"repeat"`、`"repeat-x"`、`"repeat-y"` 和 `"no-repeat"`。看一个例子。



```
var image = document.images[0],  
    pattern = context.createPattern(image, "repeat");  
  
//绘制矩形  
context.fillStyle = pattern;  
context.fillRect(10, 10, 150, 150);
```

2DFillRectPatternExample01.htm

需要注意的是，模式与渐变一样，都是从画布的原点(0,0)开始的。将填充样式 (`fillStyle`) 设置为模式对象，只表示在某个特定的区域内显示重复的图像，而不是要从某个位置开始绘制重复的图像。上面的代码会得到如图 15-13 所示的结果。

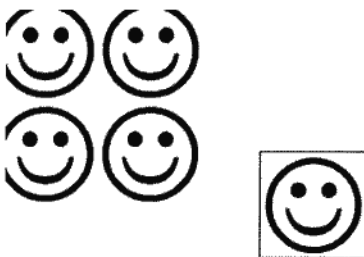


图 15-13

`createPattern()` 方法的第一个参数也可以是一个 `<video>` 元素，或者另一个 `<canvas>` 元素。

15.2.10 使用图像数据

2D 上下文的一个明显的长处就是，可以通过 `getImageData()` 取得原始图像数据。这个方法接收 4 个参数：要取得其数据的画面区域的 `x` 和 `y` 坐标以及该区域的像素宽度和高度。例如，要取得左上角坐标为(10,5)、大小为 50×50 像素的区域的图像数据，可以使用以下代码：

```
var imageData = context.getImageData(10, 5, 50, 50);
```

这里返回的对象是 `ImageData` 的实例。每个 `ImageData` 对象都有三个属性：`width`、`height` 和 `data`。其中 `data` 属性是一个数组，保存着图像中每一个像素的数据。在 `data` 数组中，每一个像素用

4 个元素来保存，分别表示红、绿、蓝和透明度值。因此，第一个像素的数据就保存在数组的第 0 到第 3 个元素中，例如：

```
var data = imageData.data,  
    red = data[0],  
    green = data[1],  
    blue = data[2],  
    alpha = data[3];
```

数组中每个元素的值都介于 0 到 255 之间（包括 0 和 255）。能够直接访问到原始图像数据，就能够以各种方式来操作这些数据。例如，通过修改图像数据，可以像下面这样创建一个简单的灰阶过滤器。

```
var drawing = document.getElementById("drawing");  
  
// 确定浏览器支持<canvas>元素  
if (drawing.getContext){  
  
    var context = drawing.getContext("2d"),  
        image = document.images[0],  
        imageData = data,  
        i, len, average,  
        red, green, blue, alpha;  
  
    // 绘制原始图像  
    context.drawImage(image, 0, 0);  
  
    // 取得图像数据  
    imageData = context.getImageData(0, 0, image.width, image.height);  
    data = imageData.data;  
  
    for (i=0, len=data.length; i < len; i+=4){  
  
        red = data[i];  
        green = data[i+1];  
        blue = data[i+2];  
        alpha = data[i+3];  
  
        // 求得 rgb 平均值  
        average = Math.floor((red + green + blue) / 3);  
  
        // 设置颜色值，透明度不变  
        data[i] = average;  
        data[i+1] = average;  
        data[i+2] = average;  
  
    }  
  
    // 回写图像数据并显示结果  
    imageData.data = data;  
    context.putImageData(imageData, 0, 0);  
}
```

2DImageDataExample01.htm

这个例子首先在画面上绘制了一幅图像，然后取得了原始图像数据。其中的 for 循环遍历了图像数据中的每一个像素。这里要注意的是，每次循环控制变量 i 都递增 4。在取得每个像素的红、绿、蓝颜

色值后, 计算出它们的平均值。再把这个平均值设置为每个颜色的值, 结果就是去掉了每个像素的颜色, 只保留了亮度接近的灰度值 (即彩色变黑白)。在把 data 数组回写到 imageData 对象后, 调用 putImageData() 方法把图像数据绘制到画布上。最终得到了图像的黑白版。

当然, 通过操作原始像素值不仅能实现灰阶过滤, 还能实现其他功能。要了解通过操作原始图像数据实现过滤器的更多信息, 请参考 Ilmari Heikkinen 的文章 “Making Image Filters with Canvas” (基于 Canvas 的图像过滤器): <http://www.html5rocks.com/en/tutorials/canvas/imagefilters/>。



只有在画布“干净”的情况下 (即图像并非来自其他域), 才可以取得图像数据。如果画布“不干净”, 那么访问图像数据时会导致 JavaScript 错误。

15.2.11 合成

还有两个会应用到 2D 上下文中所有绘制操作的属性: globalAlpha 和 globalCompositionOperation。其中, globalAlpha 是一个介于 0 和 1 之间的值 (包括 0 和 1), 用于指定所有绘制的透明度。默认值为 0。如果所有后续操作都要基于相同的透明度, 就可以先把 globalAlpha 设置为适当值, 然后绘制, 最后再把它设置回默认值 0。下面来看一个例子。



```
//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//修改全局透明度
context.globalAlpha = 0.5;

//绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);

//重置全局透明度
context.globalAlpha = 0;
```

[2DGlobalAlphaExample01.htm](#)

在这个例子中, 我们把蓝色矩形绘制到了红色矩形上面。因为在绘制蓝色矩形前, globalAlpha 已经被设置为 0.5, 所以蓝色矩形会呈现半透明效果, 透过它可以看到下面的红色矩形。

第二个属性 globalCompositionOperation 表示后绘制的图形怎样与先绘制的图形结合。这个属性的值是字符串, 可能的值如下。

- ☐ source-over (默认值): 后绘制的图形位于先绘制的图形上方。
- ☐ source-in: 后绘制的图形与先绘制的图形重叠的部分可见, 两者其他部分完全透明。
- ☐ source-out: 后绘制的图形与先绘制的图形不重叠的部分可见, 先绘制的图形完全透明。
- ☐ source-atop: 后绘制的图形与先绘制的图形重叠的部分可见, 先绘制图形不受影响。
- ☐ destination-over: 后绘制的图形位于先绘制的图形下方, 只有之前透明像素下的部分才可见。
- ☐ destination-in: 后绘制的图形位于先绘制的图形下方, 两者不重叠的部分完全透明。
- ☐ destination-out: 后绘制的图形擦除与先绘制的图形重叠的部分。
- ☐ destination-atop: 后绘制的图形位于先绘制的图形下方, 在两者不重叠的地方, 先绘制的

图形会变透明。

- **lighter**: 后绘制的图形与先绘制的图形重叠部分的值相加, 使该部分变亮。
- **copy**: 后绘制的图形完全替代与之重叠的先绘制图形。
- **xor**: 后绘制的图形与先绘制的图形重叠的部分执行“异或”操作。

这个合成操作实际上用语言或者黑白图像是很难说清楚的。要了解每个操作的具体效果, 请参见 https://developer.mozilla.org/samples/canvas-tutorial/6_1_canvas_composite.html。推荐使用 IE9+ 或 Firefox 4+ 访问前面的网页, 因为这两款浏览器对 Canvas 的实现最完善。下面来看一个例子。



```
//绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);

//设置合成操作
context.globalCompositeOperation = "destination-over";

//绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

2DGlobalCompositeOperationExample01.htm

如果不修改 `globalCompositeOperation`, 那么蓝色矩形应该位于红色矩形之上。但把 `globalCompositeOperation` 设置为 "destination-over" 之后, 红色矩形跑到了蓝色矩形上面。

在使用 `globalCompositeOperation` 的情况下, 一定要多测试一些浏览器。因为不同浏览器对这个属性的实现仍然存在较大的差别。Safari 和 Chrome 在这方面还有问题, 至于有什么问题, 大家可以比较在打开上述页面的情况下, IE9+ 和 Firefox 4+ 与它们有什么差异。

15.3 WebGL

WebGL 是针对 Canvas 的 3D 上下文。与其他 Web 技术不同, WebGL 并不是 W3C 制定的标准, 而是由 Khronos Group 制定的。其官方网站是这样介绍的: “Khronos Group 是一个非盈利的由会员资助的协会, 专注于为并行计算以及各种平台和设备上的图形及动态媒体制定无版税的开放标准。” Khronos Group 也设计了其他图形处理 API, 比如 OpenGL ES 2.0。浏览器中使用的 WebGL 就是基于 OpenGL ES 2.0 制定的。

OpenGL 等 3D 图形语言是非常复杂的, 本书不可能介绍其中每一个概念。熟悉 OpenGL ES 2.0 的读者可能会觉得 WebGL 更好理解一些, 因为好多概念是相通的。

本节将适当地介绍 OpenGL ES 2.0 的一些概念, 尽力解释其中的某些部分在 WebGL 中的实现。要全面了解 OpenGL, 请访问 www.opengl.org。要全面学习 WebGL, 请参考 www.learningwebgl.com, 其中包含非常棒的系列教程^①。

15.3.1 类型化数组

WebGL 涉及的复杂计算需要提前知道数值的精度, 而标准的 JavaScript 数值无法满足需要。为此,

^① 中文翻译版请参考 <http://www.hiwebgl.com/?p=42>。

WebGL 引入了一个概念，叫类型化数组（typed arrays）。类型化数组也是数组，只不过其元素被设置为特定类型的值。

类型化数组的核心就是一个名为 `ArrayBuffer` 的类型。每个 `ArrayBuffer` 对象表示的只是内存中指定的字节数，但不会指定这些字节用于保存什么类型的数据。通过 `ArrayBuffer` 所能做的，就是为了将来使用而分配一定数量的字节。例如，下面这行代码会在内存中分配 20B。

```
var buffer = new ArrayBuffer(20);
```

创建了 `ArrayBuffer` 对象后，能够通过该对象获得的信息只有它包含的字节数，方法是访问其 `byteLength` 属性：

```
var bytes = buffer.byteLength;
```

虽然 `ArrayBuffer` 对象本身没有多少可说的，但对 WebGL 而言，使用它是极其重要的。而且，在涉及视图的时候，你才会发现它原来还是很有意思的。

1. 视图

使用 `ArrayBuffer`（数组缓冲器类型）的一种特别的方式就是用它来创建数组缓冲器视图。其中，最常见的视图是 `DataView`，通过它可以选 `ArrayBuffer` 中一小段字节。为此，可以在创建 `DataView` 实例的时候传入一个 `ArrayBuffer`、一个可选的字节偏移量（从该字节开始选择）和一个可选的要选择的字节数。例如：

```
//基于整个缓冲器创建一个新视图
var view = new DataView(buffer);

//创建一个始于字节 9 的新视图
var view = new DataView(buffer, 9);

//创建一个从字节 9 开始到字节 18 的新视图
var view = new DataView(buffer, 9, 10);
```

实例化之后，`DataView` 对象会把字节偏移量以及字节长度信息分别保存在 `byteOffset` 和 `byteLength` 属性中。

```
alert(view.byteOffset);
alert(view.byteLength);
```

通过这两个属性可以在以后方便地了解视图的状态。另外，通过其 `buffer` 属性也可以取得数组缓冲器。

读取和写入 `DataView` 的时候，要根据实际操作的数据类型，选择相应的 `getter` 和 `setter` 方法。下表列出了 `DataView` 支持的数据类型以及相应的读写方法。

| 数据类型 | getter | setter |
|----------|--|---|
| 有符号8位整数 | <code>getInt8(byteOffset)</code> | <code>setInt8(byteOffset, value)</code> |
| 无符号8位整数 | <code>getUint8(byteOffset)</code> | <code>setUint8(byteOffset, value)</code> |
| 有符号16位整数 | <code>getInt16(byteOffset, littleEndian)</code> | <code>setInt16(byteOffset, value, littleEndian)</code> |
| 无符号16位整数 | <code>getUint16(byteOffset, littleEndian)</code> | <code>setUint16(byteOffset, value, littleEndian)</code> |
| 有符号32位整数 | <code>getInt32(byteOffset, littleEndian)</code> | <code>setInt32(byteOffset, value, littleEndian)</code> |

(续)

| 数据类型 | getter | setter |
|----------|---|--|
| 无符号32位整数 | <code>getUint32(byteOffset, littleEndian)</code> | <code>setUint32(byteOffset, value, littleEndian)</code> |
| 32位浮点数 | <code>getFloat32(byteOffset, littleEndian)</code> | <code>setFloat32(byteOffset, value, littleEndian)</code> |
| 64位浮点数 | <code>getFloat64(byteOffset, littleEndian)</code> | <code>setFloat64(byteOffset, value, littleEndian)</code> |

15

所有这些方法的第一个参数都是一个字节偏移量，表示要从哪个字节开始读取或写入。不要忘了，要保存有些数据类型的数据，可能需要不止 1B。比如，无符号 8 位整数要用 1B，而 32 位浮点数则要用 4B。使用 `DataView`，就需要你自己来管理这些细节，即要明确知道自己的数据需要多少字节，并选择正确的读写方法。例如：



```
var buffer = new ArrayBuffer(20),  
    view = new DataView(buffer),  
    value;  
  
view.setUint16(0, 25);  
view.setUint16(2, 50); // 不能从字节 1 开始，因为 16 位整数要用 2B  
value = view.getUint16(0);
```

[DataViewExample01.htm](#)

以上代码把两个无符号 16 位整数保存到了数组缓冲器中。因为每个 16 位整数要用 2B，所以保存第一个数的字节偏移量为 0，而保存第二个数的字节偏移量为 2。

用于读写 16 位或更大数值的方法都有一个可选的参数 `littleEndian`。这个参数是一个布尔值，表示读写数值时是否采用小端字节序（即将数据的最低有效位保存在低内存地址中），而不是大端字节序（即将数据的最低有效位保存在高内存地址中）。如果你也不确定应该使用哪种字节序，那不用管它，就采用默认的大端字节序方式保存即可。

因为在这里使用的是字节偏移量，而非数组元素数，所以可以通过几种不同的方式来访问同一字节。例如：



```
var buffer = new ArrayBuffer(20),  
    view = new DataView(buffer),  
    value;  
  
view.setUint16(0, 25);  
value = view.getInt8(0);  
  
alert(value); // 0
```

[DataViewExample02.htm](#)

在这个例子中，数值 25 以 16 位无符号整数的形式被写入，字节偏移量为 0。然后，再以 8 位有符号整数的方式读取该数据，得到的结果是 0。这是因为 25 的二进制形式的前 8 位（第一个字节）全部是 0，如图 15-14 所示。

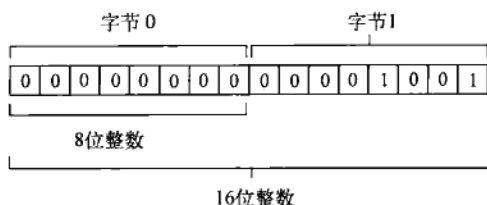


图 15-14

可见,虽然 DataView 能让我们在字节级别上读写数组缓冲器中的数据,但我们必须自己记住要将数据保存到哪里,需要占用多少字节。这样一来,就会带来很多工作量,因此类型化视图也就应运而生。

2. 类型化视图

类型化视图一般也被称为类型化数组,因为它们除了元素必须是某种特定的数据类型外,与常规的数组无异。类型化视图也分几种,而且它们都继承了 DataView。

- Int8Array: 表示 8 位二补整数。
- Uint8Array: 表示 8 位无符号整数。
- Int16Array: 表示 16 位二补整数。
- Uint16Array: 表示 16 位无符号整数。
- Int32Array: 表示 32 位二补整数。
- Uint32Array: 表示 32 位无符号整数。
- Float32Array: 表示 32 位 IEEE 浮点值。
- Float64Array: 表示 64 位 IEEE 浮点值。

每种视图类型都以不同的方式表示数据,而同一数据视选择的类型不同有可能占用一或多字节。例如,20B 的 ArrayBuffer 可以保存 20 个 Int8Array 或 Uint8Array,或者 10 个 Int16Array 或 Uint16Array,或者 5 个 Int32Array、Uint32Array 或 Float32Array,或者 2 个 Float64Array。

由于这些视图都继承自 DataView,因而可以使用相同的构造函数参数来实例化。第一个参数是要使用 ArrayBuffer 对象,第二个参数是作为起点的字节偏移量(默认为 0),第三个参数是要包含的字节数。三个参数中只有第一个是必需的。下面来看几个例子。

```
//创建一个新数组,使用整个缓冲器
var int8s = new Int8Array(buffer);

//只使用从字节 9 开始的缓冲器
var int16s = new Int16Array(buffer, 9);

//只使用从字节 9 到字节 18 的缓冲器
var uint16s = new Uint16Array(buffer, 9, 10);
```

能够指定缓冲器中可用的字节段,意味着能在同一个缓冲器中保存不同类型的数值。比如,下面的代码就是在缓冲器的开头保存 8 位整数,而在其他字节中保存 16 位整数。

```
//使用缓冲器的一部分保存 8 位整数,另一部分保存 16 位整数
var int8s = new Int8Array(buffer, 0, 10);
var uint16s = new Uint16Array(buffer, 11, 10);
```

每个视图构造函数都有一个名为 BYTES_PER_ELEMENT 的属性,表示类型化数组的每个元素需要多少字节。因此,Uint8Array.BYTES_PER_ELEMENT 就是 1,而 Float32Array.BYTES_PER_ELEMENT

则为 4。可以利用这个属性来辅助初始化。

```
//需要 10 个元素空间
var int8s = new Int8Array(buffer, 0, 10 * Int8Array.BYTES_PER_ELEMENT);

//需要 5 个元素空间
var uint16s = new Uint16Array(buffer, int8s.byteOffset + int8s.byteLength,
    5 * Uint16Array.BYTES_PER_ELEMENT);
```

以上代码基于同一个数组缓冲器创建了两个视图。缓冲器的前 10B 用于保存 8 位整数，而其他字节用于保存无符号 16 位整数。在初始化 Uint16Array 的时候，使用了 Int8Array 的 byteOffset 和 byteLength 属性，以确保 uint16s 开始于 8 位数据之后。

如前所述，类型化视图的目的在于简化对二进制数据的操作。除了前面看到的优点之外，创建类型化视图还可以不用首先创建 ArrayBuffer 对象。只要传入希望数组保存的元素数，相应的构造函数就可以自动创建一个包含足够字节数的 ArrayBuffer 对象，例如：

```
//创建一个数组保存 10 个 8 位整数 (10 字节)
var int8s = new Int8Array(10);

//创建一个数组保存 10 个 16 位整数 (20 字节)
var int16s = new Int16Array(10);
```

另外，也可以把常规数组转换为类型化视图，只要把常规数组传入类型化视图的构造函数即可：

```
//创建一个数组保存 5 个 8 位整数 (10 字节)
var int8s = new Int8Array([10, 20, 30, 40, 50]);
```

这是用默认值来初始化类型化视图的最佳方式，也是 WebGL 项目中最常用的方式。

以这种方式来使用类型化视图，可以让它们看起来更像 Array 对象，同时也能确保在读写信息的时候使用正确的数据类型。

使用类型化视图时，可以通过方括号语法访问每一个数据成员，可以通过 length 属性确定数组中有多少元素。这样，对类型化视图的迭代与对 Array 对象的迭代就是一样的了。

```
for (var i=0, len=int8s.length; i < len; i++){
    console.log("Value at position " + i + " is " + int8s[i]);
}
```

当然，也可以使用方括号语法为类型化视图的元素赋值。如果为相应元素指定的字节数放不下相应的值，则实际保存的值是最大可能值的模。例如，无符号 16 位整数所能表示的最大数值是 65535，如果你想保存 65536，那实际保存的值是 0；如果你想保存 65537，那实际保存的值是 1，依此类推。

```
var uint16s = new Uint16Array(10);
uint16s[0] = 65537;
alert(uint16s[0]); //1
```

数据类型不匹配时不会抛出错误，所以你必须自己保证所赋的值不会超过相应元素的字节限制。

类型化视图还有一个方法，即 subarray()，使用这个方法可以基于底层数组缓冲器的子集创建一个新视图。这个方法接收两个参数：开始元素的索引和可选的结束元素的索引。返回的类型与调用该方法的视图类型相同。例如：

```
var uint16s = new Uint16Array(10),
    sub = uint16s.subarray(2, 5);
```

在以上代码中, sub 也是 Uint16Array 的一个实例, 而且底层与 uint16s 都基于同一个 ArrayBuffer。通过大视图创建小视图的主要好处就是, 在操作大数组中的一部分元素时, 无需担心意外修改了其他元素。

类型化数组是 WebGL 项目中执行各种操作的重要基础。

15.3.2 WebGL 上下文

目前, 在支持的浏览器中, WebGL 的名字叫 "experimental-webgl", 这是因为 WebGL 规范仍然未制定完成。制定完成后, 这个上下文的名字就会变成简单的 "webgl"。如果浏览器不支持 WebGL, 那么取得该上下文时会返回 null。在使用 WebGL 上下文时, 务必先检测一下返回值。



```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl");
    if (gl){
        //使用 WebGL
    }
}
```

WebGLExample01.htm

一般都把 WebGL 上下文对象命名为 gl。大多数 WebGL 应用和示例都遵守这一约定, 因为 OpenGL ES 2.0 规定的方法和值通常都以 "gl" 开头。这样做也可以保证 JavaScript 代码与 OpenGL 程序更相近。

取得了 WebGL 上下文之后, 就可以开始 3D 绘图了。如前所述, WebGL 是 OpenGL ES 2.0 的 Web 版, 因此本节讨论的概念实际上就是 OpenGL 概念在 JavaScript 中的实现。

通过给 getContext() 传递第二个参数, 可以为 WebGL 上下文设置一些选项。这个参数本身是一个对象, 可以包含下列属性。

- ❑ alpha: 值为 true, 表示为上下文创建一个 Alpha 通道缓冲区; 默认值为 true。
- ❑ depth: 值为 true, 表示可以使用 16 位深缓冲区; 默认值为 true。
- ❑ stencil: 值为 true, 表示可以使用 8 位模板缓冲区; 默认值为 false。
- ❑ antialias: 值为 true, 表示将使用默认机制执行抗锯齿操作; 默认值为 true。
- ❑ premultipliedAlpha: 值为 true, 表示绘图缓冲区有预乘 Alpha 值; 默认值为 true。
- ❑ preserveDrawingBuffer: 值为 true, 表示在绘图完成后保留绘图缓冲区; 默认值为 false。

建议确实有必要的环境下再开启这个值, 因为可能影响性能。

传递这个选项对象的方式如下:



```
var drawing = document.getElementById("drawing");

//确定浏览器支持<canvas>元素
if (drawing.getContext){

    var gl = drawing.getContext("experimental-webgl", { alpha: false});
    if (gl){
        //使用 WebGL
    }
}
```

```
}  
}
```

WebGLEExample01.htm

大多数上下文选项只在高级技巧中使用。很多时候，各个选项的默认值就能满足我们的要求。

如果 `getContext()` 无法创建 WebGL 上下文，有的浏览器会抛出错误。为此，最好把调用封装到一个 `try-catch` 块中。

```
Insert IconMargin      [download]var drawing = document.getElementById("drawing"),  
    gl;  
  
//确定浏览器支持<canvas>元素  
if (drawing.getContext){  
    try {  
        gl = drawing.getContext("experimental-webgl");  
    } catch (ex) {  
        //什么也不做  
    }  
    if (gl){  
        //使用 WebGL  
    } else {  
        alert("WebGL context could not be created.");  
    }  
}
```

WebGLEExample01.htm

1. 常量

如果你熟悉 OpenGL，那肯定会对各种操作中使用非常多的常量印象深刻。这些常量在 OpenGL 中都带前缀 `GL_`。在 WebGL 中，保存在上下文对象中的这些常量都没有 `GL_` 前缀。比如说，`GL_COLOR_BUFFER_BIT` 常量在 WebGL 上下文中就是 `gl.COLOR_BUFFER_BIT`。WebGL 以这种方式支持大多数 OpenGL 常量（有一部分常量是不支持的）。

2. 方法命名

OpenGL（以及 WebGL）中的很多方法都试图通过名字传达有关数据类型的信息。如果某方法可以接收不同类型及不同数量的参数，看方法名的后缀就可以知道。方法名的后缀会包含参数个数（1 到 4）和接收的数据类型（`f` 表示浮点数，`i` 表示整数）。例如，`gl.uniform4f()` 意味着要接收 4 个浮点数，而 `gl.uniform3i()` 则表示要接收 3 个整数。

也有很多方法接收数组参数而非一个个单独的参数。这样的方法其名字中会包含字母 `v`（即 `vector`，矢量）。因此，`gl.uniform3iv()` 可以接收一个包含 3 个值的整数数组。请大家记住以上命名约定，这样对理解后面关于 WebGL 的讨论很有帮助。

3. 准备绘图

在实际操作 WebGL 上下文之前，一般都要使用某种实色清除 `<canvas>`，为绘图做好准备。为此，首先必须使用 `clearColor()` 方法来指定要使用的颜色值，该方法接收 4 个参数：红、绿、蓝和透明度。每个参数必须是一个 0 到 1 之间的数值，表示每种分量在最终颜色中的强度。来看下面的例子。

```
gl.clearColor(0,0,0,1); //black  
gl.clear(gl.COLOR_BUFFER_BIT);
```

WebGLEExample01.htm



以上代码把清理颜色缓冲区的值设置为黑色,然后调用了 `clear()` 方法,这个方法与 OpenGL 中的 `glClear()` 等价。传入的参数 `gl.COLOR_BUFFER_BIT` 告诉 WebGL 使用之前定义的颜色来填充相应区域。一般来说,都要先清理缓冲区,然后再执行其他绘图操作。

4. 视口与坐标

开始绘图之前,通常要先定义 WebGL 的视口 (viewport)。默认情况下,视口可以使用整个 `<canvas>` 区域。要改变视口大小,可以调用 `viewport()` 方法并传入 4 个参数: (视口相对于 `<canvas>` 元素的) `x` 坐标、`y` 坐标、宽度和高度。例如,下面的调用就使用了 `<canvas>` 元素:

```
gl.viewport(0, 0, drawing.width, drawing.height);
```

视口坐标与我们通常熟悉的网页坐标不一样。视口坐标的原点 $(0,0)$ 在 `<canvas>` 元素的左下角, `x` 轴和 `y` 轴的正方向分别是向右和向上,可以定义为 $(width-1, height-1)$, 如图 15-15 所示。

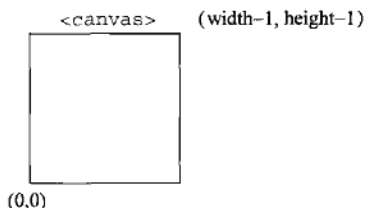


图 15-15

知道怎么定义视口大小,就可以只在 `<canvas>` 元素的部分区域中绘图。来看下面的例子。

```
//视口是<canvas>左下角的四分之一区域  
gl.viewport(0, 0, drawing.width/2, drawing.height/2);  
  
//视口是<canvas>左上角的四分之一区域  
gl.viewport(0, drawing.height/2, drawing.width/2, drawing.height/2);  
  
//视口是<canvas>右下角的四分之一区域  
gl.viewport(drawing.width/2, 0, drawing.width/2, drawing.height/2);
```

另外,视口内部的坐标系与定义视口的坐标系也不一样。在视口内部,坐标原点 $(0,0)$ 是视口的中心点,因此视口左下角坐标为 $(-1,-1)$,而右上角坐标为 $(1,1)$,如图 15-16 所示。

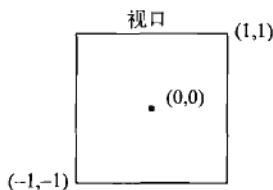


图 15-16

如果在视口内部绘图时使用视口外部的坐标,结果可能会被视口剪切。比如,要绘制的形状有一个顶点在 $(1,2)$,那么该形状在视口右侧的部分会被剪切掉。

5. 缓冲区

顶点信息保存在 JavaScript 的类型化数组中,使用之前必须转换到 WebGL 的缓冲区。要创建缓冲区,

可以调用 `gl.createBuffer()`，然后使用 `gl.bindBuffer()` 绑定到 WebGL 上下文。这两步做完之后，就可以用数据来填充缓冲区了。例如：

```
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([0, 0.5, 1]), gl.STATIC_DRAW);
```

调用 `gl.bindBuffer()` 可以将 `buffer` 设置为上下文的当前缓冲区。此后，所有缓冲区操作都直接在 `buffer` 中执行。因此，调用 `gl.bufferData()` 时不需要明确传入 `buffer` 也没有问题。最后一行代码使用 `Float32Array` 中的数据初始化了 `buffer`（一般都是用 `Float32Array` 来保存顶点信息）。如果想使用 `drawElements()` 输出缓冲区的内容，也可以传入 `gl.ELEMENT_ARRAY_BUFFER`。

`gl.bufferData()` 的最后一个参数用于指定使用缓冲区的方式，取值范围是如下几个常量。

- ❑ `gl.STATIC_DRAW`：数据只加载一次，在多次绘图中使用。
- ❑ `gl.STREAM_DRAW`：数据只加载一次，在几次绘图中使用。
- ❑ `gl.DYNAMIC_DRAW`：数据动态改变，在多次绘图中使用。

如果不是非常有经验的 OpenGL 程序员，多数情况下将缓冲区使用方式设置为 `gl.STATIC_DRAW` 即可。

在包含缓冲区的页面重载之前，缓冲区始终保留在内存中。如果你不想要某个缓冲区了，可以直接调用 `gl.deleteBuffer()` 释放内存：

```
gl.deleteBuffer(buffer);
```

6. 错误

JavaScript 与 WebGL 之间的一个最大的区别在于，WebGL 操作一般不会抛出错误。为了知道是否有错误发生，必须在调用某个可能出错的方法后，手工调用 `gl.getError()` 方法。这个方法返回一个表示错误类型的常量。可能的错误常量如下。

- ❑ `gl.NO_ERROR`：上一次操作没有发生错误（值为 0）。
- ❑ `gl.INVALID_ENUM`：应该给方法传入 WebGL 常量，但却传错了参数。
- ❑ `gl.INVALID_VALUE`：在需要无符号数的地方传入了负值。
- ❑ `gl.INVALID_OPERATION`：在当前状态下不能完成操作。
- ❑ `gl.OUT_OF_MEMORY`：没有足够的内存完成操作。
- ❑ `gl.CONTEXT_LOST_WEBGL`：由于外部事件（如设备断电）干扰丢失了当前 WebGL 上下文。

每次调用 `gl.getError()` 方法返回一个错误值。第一次调用后，后续对 `gl.getError()` 的调用可能会返回另一个错误值。如果发生了多个错误，需要反复调用 `gl.getError()` 直至它返回 `gl.NO_ERROR`。在执行了很多操作的情况下，最好通过一个循环来调用 `getError()`，如下所示：

```
var errorCode = gl.getError();
while(errorCode){
    console.log("Error occurred: " + errorCode);
    errorCode = gl.getError();
}
```

如果 WebGL 脚本输出不正确，那在脚本中放几行 `gl.getError()` 有助于找出问题所在。

7. 着色器

着色器（shader）是 OpenGL 中的另一个概念。WebGL 中有两种着色器：顶点着色器和片段（或像素）着色器。顶点着色器用于将 3D 顶点转换为需要渲染的 2D 点。片段着色器用于准确计算要绘制的

每个像素的颜色。WebGL 着色器的独特之处也是其难点在于，它们并不是用 JavaScript 写的。这些着色器是使用 GLSL (OpenGL Shading Language, OpenGL 着色语言) 写的，GLSL 是一种与 C 和 JavaScript 完全不同的语言。

8. 编写着色器

GLSL 是一种类 C 语言，专门用于编写 OpenGL 着色器。因为 WebGL 是 OpenGL ES 2.0 的实现，所以 OpenGL 中使用的着色器可以直接在 WebGL 中使用。这样就方便了将桌面图形应用移植到浏览器中。

每个着色器都有一个 `main()` 方法，该方法在绘图期间会重复执行。为着色器传递数据的方式有两种：Attribute 和 Uniform。通过 Attribute 可以向顶点着色器中传入顶点信息，通过 Uniform 可以向任何着色器传入常量值。Attribute 和 Uniform 在 `main()` 方法外部定义，分别使用关键字 `attribute` 和 `uniform`。在这两个值类型关键字之后，是数据类型和变量名。下面是一个简单的顶点着色器的例子。



```
//OpenGL 着色语言
//着色器，作者 Bartek Drozd, 摘自他的文章
//http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
attribute vec2 aVertexPosition;

void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
```

WebGLExample02.htm

这个顶点着色器定义了一个名为 `aVertexPosition` 的 Attribute，这个 Attribute 是一个数组，包含两个元素（数据类型为 `vec2`），表示 `x` 和 `y` 坐标。即使只接收到两个坐标，顶点着色器也必须把一个包含四方面信息的顶点赋值给特殊变量 `gl_Position`。这里的着色器创建了一个新的包含四个元素的数组（`vec4`），填补缺失的坐标，结果是把 2D 坐标转换成了 3D 坐标。

除了只能通过 Uniform 传入数据外，片段着色器与顶点着色器类似。以下是片段着色器的例子。

```
//OpenGL 着色语言
//着色器，作者 Bartek Drozd, 摘自他的文章
//http://www.netmagazine.com/tutorials/get-started-webgl-draw-square
uniform vec4 uColor;

void main() {
    gl_FragColor = uColor;
}
```

WebGLExample02.htm


片段着色器必须返回一个值，赋给变量 `gl_FragColor`，表示绘图时使用的颜色。这个着色器定义了一个包含四方面信息（`vec4`）的统一的颜色 `uColor`。从以上代码看，这个着色器除了把传入的值赋给 `gl_FragColor` 什么也没做。`uColor` 的值在这个着色器内部不能改变。



OpenGL 着色语言比这里看到的还要复杂。专门讲解这门语言的书有很多，本节只是从辅助使用 WebGL 的角度简要介绍一下该语言。要了解更多信息，请参考 Randi J. Rost 编著的 *OpenGL Shading Language* (Addison-Wesley, 2006)。

9. 编写着色器程序

浏览器不能理解 GLSL 程序，因此必须准备好字符串形式的 GLSL 程序，以便编译并链接到着色器程序。为便于使用，通常是把着色器包含在页面的<script>标签内，并为该标签指定一个自定义的 type 属性。由于无法识别 type 属性值，浏览器不会解析<script>标签中的内容，但这不影响你读写其中的代码。例如：



```
<script type="x-webgl/x-vertex-shader" id="vertexShader">
attribute vec2 aVertexPosition;

void main() {
    gl_Position = vec4(aVertexPosition, 0.0, 1.0);
}
</script>
<script type="x-webgl/x-fragment-shader" id="fragmentShader">
uniform vec4 uColor;

void main() {
    gl_FragColor = uColor;
}
</script>
```

WebGLExample02.htm

然后，可以通过 text 属性提取出<script>元素的内容：

```
var vertexGls1 = document.getElementById("vertexShader").text;
fragmentGls1 = document.getElementById("fragmentShader").text;
```

复杂一些的 WebGL 应用可能会通过 Ajax（详见第 21 章）动态加载着色器。而使用着色器的关键是要有字符串形式的 GLSL 程序。

取得了 GLSL 字符串之后，接下来就是创建着色器对象。要创建着色器对象，可以调用 gl.createShader() 方法并传入要创建的着色器类型（gl.VERTEX_SHADER 或 gl.FRAGMENT_SHADER）。编译着色器使用的是 gl.compileShader()。请看下面的例子。

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexGls1);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentGls1);
gl.compileShader(fragmentShader);
```

WebGLExample02.htm

以上代码创建了两个着色器，并将它们分别保存在 vertexShader 和 fragmentShader 中。而使用下列代码，可以把这两个对象链接到着色器程序中。

```
var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

WebGLExample02.htm

第一行代码创建了程序,然后调用 `attachShader()` 方法又包含了两个着色器。最后调用 `gl.linkProgram()` 则把两个着色器封装到了变量 `program` 中。链接完程序之后,就可以通过 `gl.useProgram()` 方法通知 WebGL 使用这个程序了。

```
gl.useProgram(program);
```

调用 `gl.useProgram()` 方法后,所有后续的绘图操作都将使用这个程序。

10. 为着色器传入值

前面定义的着色器都必须接收一个值才能工作。为了给着色器传入这个值,必须先找到要接收这个值的变量。对于 Uniform 变量,可以使用 `gl.getUniformLocation()`,这个方法返回一个对象,表示 Uniform 变量在内存中的位置。然后可以基于变量的位置来赋值。例如:

```
var uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [0, 0, 0, 1]);
```

[WebGLExample02.htm](#)

第一行代码从 `program` 中找到 Uniform 变量 `uColor`,返回了它在内存中的位置。第二行代码使用 `gl.uniform4fv()` 给 `uColor` 赋值。

对于顶点着色器中的 Attribute 变量,也是差不多的赋值过程。要找到 Attribute 变量在内存中的位置,可以调用 `gl.getAttribLocation()`。取得了位置之后,就可以像下面这样赋值了:

```
var aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, itemSize, gl.FLOAT, false, 0, 0);
```

[WebGLExample02.htm](#)

在此,我们取得了 `aVertexPosition` 的位置,然后又通过 `gl.enableVertexAttribArray()` 启用它。最后一行创建了指针,指向由 `gl.bindBuffer()` 指定的缓冲区,并将其保存在 `aVertexPosition` 中,以便顶点着色器使用。

11. 调试着色器和程序

与 WebGL 中的其他操作一样,着色器操作也可能会失败,而且也是静默失败。如果你想知道着色器或程序执行中是否发生了错误,必须亲自询问 WebGL 上下文。

对于着色器,可以在操作之后调用 `gl.getShaderParameter()`,取得着色器的编译状态:

```
if (!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
    alert(gl.getShaderInfoLog(vertexShader));
}
```

[WebGLExample02.htm](#)

这个例子检测了 `vertexShader` 的编译状态。如果着色器编译成功,调用 `gl.getShaderParameter()` 会返回 `true`。如果返回的是 `false`,说明编译期间发生了错误,此时调用 `gl.getShaderInfoLog()` 并传入相应的着色器就可以取得错误消息。错误消息就是一个表示问题所在的字符串。无论是顶点着色器,还是片段着色器,都可以使用 `gl.getShaderParameter()` 和 `gl.getShaderInfoLog()` 方法。

程序也可能会执行失败，因此也有类似的方法——`gl.getProgramParameter()`，可以用来检测执行状态。最常见的程序失败发生在链接过程中，要检测链接错误，可以使用下列代码。

```
if (!gl.getProgramParameter(program, gl.LINK_STATUS)){  
    alert(gl.getProgramInfoLog(program));  
}
```

WebGLExample02.htm

与 `gl.getShaderParameter()` 类似，`gl.getProgramParameter()` 返回 `true` 表示链接成功，返回 `false` 表示链接失败。同样，也有一个 `gl.getProgramInfoLog()` 方法，用于捕获程序失败的消息。

以上介绍的这些方法主要在开发过程中用于调试。只要没有依赖外部代码，就可以放心地把它们从产品代码中删除。

12. 绘图

WebGL 只能绘制三种形状：点、线和三角。其他所有形状都是由这三种基本形状合成之后，再绘制到三维空间中的。执行绘图操作要调用 `gl.drawArrays()` 或 `gl.drawElements()` 方法，前者用于数组缓冲区，后者用于元素数组缓冲区。

`gl.drawArrays()` 或 `gl.drawElements()` 的第一个参数都是一个常量，表示要绘制的形状。可取值的常量范围包括以下这些。

- ❑ `gl.POINTS`：将每个顶点当成一个点来绘制。
- ❑ `gl.LINES`：将数组当成一系列顶点，在这些顶点间画线。每个顶点既是起点也是终点，因此数组中必须包含偶数个顶点才能完成绘制。
- ❑ `gl.LINE_LOOP`：将数组当成一系列顶点，在这些顶点间画线。线条从第一个顶点到第二个顶点，再从第二个顶点到第三个顶点，依此类推，直至最后一个顶点。然后再从最后一个顶点到第一个顶点画一条线。结果就是一个形状的轮廓。
- ❑ `gl.LINE_STRIP`：除了不画最后一个顶点与第一个顶点之间的线之外，其他与 `gl.LINE_LOOP` 相同。
- ❑ `gl.TRIANGLES`：将数组当成一系列顶点，在这些顶点间绘制三角形。除非明确指定，每个三角形都单独绘制，不与其他三角形共享顶点。
- ❑ `gl.TRIANGLES_STRIP`：除了将前三个顶点之后的顶点当作第三个顶点与前两个顶点共同构成一个新三角形外，其他都与 `gl.TRIANGLES` 相同。例如，如果数组中包含 A、B、C、D 四个顶点，则第一个三角形连接 ABC，而第二个三角形连接 BCD。
- ❑ `gl.TRIANGLES_FAN`：除了将前三个顶点之后的顶点当作第三个顶点与前一个顶点及第一个顶点共同构成一个新三角形外，其他都与 `gl.TRIANGLES` 相同。例如，如果数组中包含 A、B、C、D 四个顶点，则第一个三角形连接 ABC，而第二个三角形连接 ACD。

`gl.drawArrays()` 方法接收上面列出的常量中的一个作为第一个参数，接收数组缓冲区中的起始索引作为第二个参数，接收数组缓冲区中包含的顶点数（点的集合数）作为第三个参数。下面的代码使用 `gl.drawArrays()` 在画布上绘制了一个三角形。

```
//假设已经使用本节前面定义的着色器清除了视口
```

```
//定义三个顶点以及每个顶点的 x 和 y 坐标
```




```
var vertices = new Float32Array([ 0, 1, 1, -1, -1, -1 ]),
    buffer = gl.createBuffer(),
    vertexSetSize = 2,
    vertexSetCount = vertices.length/vertexSetSize,
    uColor, aVertexPosition;

//把数据放到缓冲区
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

//为片段着色器传入颜色值
uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [ 0, 0, 0, 1 ]);

//为着色器传入顶点信息
aVertexPosition = gl.getAttribLocation(program, "aVertexPosition");
gl.enableVertexAttribArray(aVertexPosition);
gl.vertexAttribPointer(aVertexPosition, vertexSetSize, gl.FLOAT, false, 0, 0);

//绘制三角形
gl.drawArrays(gl.TRIANGLES, 0, vertexSetCount);
```

WebGLExample02.htm

这个例子定义了一个 `Float32Array`，包含三组顶点（每个顶点由两点表示）。这里关键是要知道顶点的大小及数量，以便将来计算时使用。把 `vertexSetSize` 设置为 2 之后，就可以计算出 `vertexSetCount` 的值。把顶点的信息保存在缓冲区中后，又把颜色信息传给了片段着色器。

接下来，给顶点着色器传入顶点大小以及 `gl.FLOAT`，后者表示顶点坐标是浮点数。传入的第四个参数是一个布尔值，`false` 在此表示坐标不是标准化的。第五个参数是步长值（stride value），表示取得下一个值的时候，要跳过多少个数组元素。除非你真需要跳过数组元素，否则传入 0 即可。最后一个参数是起点偏移量，值为 0 表示从第一个元素开始。

最后一步就是使用 `gl.drawArrays()` 绘制三角形。传入 `gl.TRIANGLES` 作为第一个参数，表示在 (0,1)、(1,-1) 和 (-1,-1) 点之间绘制三角形，并使用传给片段着色器的颜色来填充它。第二个参数是缓冲区中的起点偏移量，最后一个参数是要读取的顶点总数。这次绘图操作的结果如图 15-17 所示。

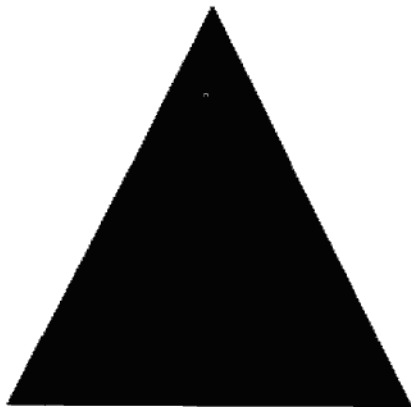


图 15-17

通过修改 `gl.drawArrays()` 的第一个参数，可以修改绘制三角形的方式。图 15-18 展示了传入不同的参数后可能得到的结果。

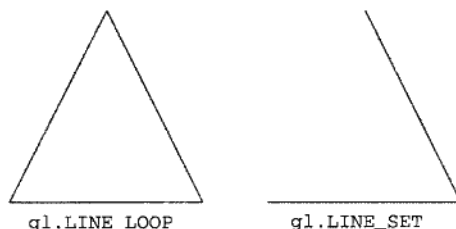


图 15-18

13. 纹理

WebGL 的纹理可以使用 DOM 中的图像。要创建一个新纹理，可以调用 `gl.createTexture()`，然后再将一幅图像绑定到该纹理。如果图像尚未加载到内存中，可能需要创建一个 `Image` 对象的实例，以便动态加载图像。图像加载完成之前，纹理不会初始化，因此，必须在 `load` 事件触发后才能设置纹理。例如：

```
var image = new Image(),
    texture;
image.src = "smile.gif";
image.onload = function(){
    texture = gl.createTexture();
    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, image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);

    //清除当前纹理
    gl.bindTexture(gl.TEXTURE_2D, null);
}
```

除了使用 DOM 中的图像之外，以上步骤与在 OpenGL 中创建纹理的步骤相同。最大的差异是使用 `gl.pixelStorei()` 设置像素存储格式。`gl.UNPACK_FLIP_Y_WEBGL` 是 WebGL 独有的常量，在加载 Web 中的图像时，多数情况下都必须使用这个常量。这主要是因为 GIF、JPEG 和 PNG 图像与 WebGL 使用的坐标系不一样，如果没有这个标志，解析图像时就会发生混乱。

用作纹理的图像必须与包含页面来自同一个域，或者是保存在启用了 CORS (Cross-Origin Resource Sharing, 跨域资源共享) 的服务器上。第 21 章将讨论 CORS。



图像、加载到 `<video>` 元素中的视频，甚至其他 `<canvas>` 元素都可以用作纹理。
跨域资源限制同样适用于视频。

14. 读取像素

与 2D 上下文类似，通过 WebGL 上下文也能读取像素值。读取像素值的方法 `readPixels()` 与 OpenGL 中的同名方法只有一点不同，即最后一个参数必须是类型化数组。像素信息是从帧缓冲区读取

的, 然后保存在类型化数组中。readPixels()方法的参数有: x、y、宽度、高度、图像格式、数据类型和类型化数组。前 4 个参数指定读取哪个区域中的像素。图像格式参数几乎总是 gl.RGBA。数据类型参数用于指定保存在类型化数组中的数据的类型, 但有以下限制。

- ❑ 如果类型是 gl.UNSIGNED_BYTE, 则类型化数组必须是 Uint8Array。
- ❑ 如果类型是 gl.UNSIGNED_SHORT_5_6_5、gl.UNSIGNED_SHORT_4_4_4_4 或 gl.UNSIGNED_SHORT_5_5_5_1, 则类型化数组必须是 Uint16Array。

下面是一个简单的例子。

```
var pixels = new Uint8Array(25*25);  
gl.readPixels(0, 0, 25, 25, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

以上代码从帧缓冲区中读取了 25×25 像素的区域, 将读取到的像素信息保存到了 pixels 数组中。其中, 每个像素的颜色由 4 个数组元素表示, 分别代表红、绿、蓝和透明度。每个数组元素的值介于 0 到 255 之间 (包含 0 和 255)。不要忘了根据返回的数据大小初始化类型化数组。

在浏览器绘制更新的 WebGL 图像之前调用 readPixels() 不会有什么意外。绘制发生后, 帧缓冲区会恢复其原始的干净状态, 而调用 readPixels() 返回的像素数据反映的就是清除缓冲区后的状态。如果你想在绘制发生后读取像素数据, 那在初始化 WebGL 上下文时必须传入适当的 preserveDrawingBuffer 选项 (前面讨论过)。

```
var gl = drawing.getContext("experimental-webgl", { preserveDrawingBuffer: true; });
```

设置这个标志的意思是让帧缓冲区在下次绘制之前, 保留其最后的状态。这个选项会导致性能损失, 因此能不用最好不要用。

15.3.3 支持

Firefox 4+ 和 Chrome 都实现了 WebGL API。Safari 5.1 也实现了 WebGL, 但默认是禁用的。WebGL 比较特别的地方在于, 某个浏览器的某个版本实现了它, 并不一定意味着就真能使用它。某个浏览器支持 WebGL, 至少意味着两件事: 首先, 浏览器本身必须实现了 WebGL API; 其次, 计算机必须升级显示驱动程序。运行 Windows XP 等操作系统的一些老机器, 其驱动程序一般都不是最新的。因此, 这些计算机中的浏览器都会禁用 WebGL。从稳妥的角度考虑, 在使用 WebGL 之前, 最好检测其是否得到了支持, 而不是只检测特定的浏览器版本。

大家别忘了, WebGL 还是一个正在制定和发展中的规范。不管是函数名、函数签名, 还是数据类型, 都有可能改变。可以说, WebGL 目前只适合实验性地学习, 不适合真正开发和应用。

15.4 小结

HTML5 的 <canvas> 元素提供了一组 JavaScript API, 让我们可以动态地创建图形和图像。图形是在一个特定的上下文中创建的, 而上下文对象目前有两种。第一种是 2D 上下文, 可以执行原始的绘图操作, 比如:

- ❑ 设置填充、描边颜色和模式
- ❑ 绘制矩形
- ❑ 绘制路径

- 绘制文本
- 创建渐变和模式

第二种是 3D 上下文, 即 WebGL 上下文。WebGL 是从 OpenGL ES 2.0 移植到浏览器中的, 而 OpenGL ES 2.0 是游戏开发人员在创建计算机图形图像时经常使用的一种语言。WebGL 支持比 2D 上下文更丰富和更强大的图形图像处理能力, 比如:

- 用 GLSL (OpenGL Shading Language, OpenGL 着色语言) 编写的顶点和片段着色器
- 支持类型化数组, 即能够将数组中的数据限定为某种特定的数值类型
- 创建和操作纹理

目前, 主流浏览器的较新版本大都已经支持<canvas>标签。同样地, 这些版本的浏览器基本上也都支持 2D 上下文。但对于 WebGL 而言, 目前还只有 Firefox 4+ 和 Chrome 支持它。