18.3.9　图案

图案是用于填充和描画图形的重复图像。要创建新图案，可以调用 createPattern()方法并传入两个参数：一个 HTML <img>元素和一个表示该如何重复图像的字符串。第二个参数的值与 CSS 的 background-repeat 属性是一样的，包括"repeat"、"repeat-x"、"repeat-y"和"no-repeat"。比如：

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

记住，跟渐变一样，图案的起点实际上是画布的原点(0, 0)。将填充样式设置为图案，表示在指定位置而不是开始绘制的位置显示图案。以上代码执行的结果如图 18-13 所示。

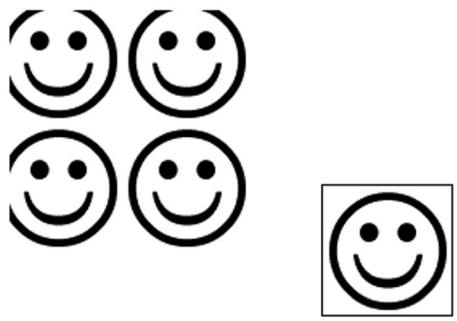

图　18-13

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

18.3.10　图像数据

2D 上下文中比较强大的一种能力是可以使用 getImageData()方法获取原始图像数据。这个方法接收 4 个参数：要取得数据中第一个像素的左上角坐标和要取得的像素宽度及高度。例如，要从(10, 5)开始取得 50 像素宽、50 像素高的区域对应的数据，可以这样写：

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

返回的对象是一个 ImageData 的实例。每个 ImageData 对象都包含 3 个属性：width、height 和 data，其中，data 属性是包含图像的原始像素信息的数组。每个像素在 data 数组中都由 4 个值表示，分别代表红、绿、蓝和透明度值。换句话说，第一个像素的信息包含在第 0 到第 3 个值中，比如：

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

这个数组中的每个值都在 0~255 范围内（包括 0 和 255）。对原始图像数据进行访问可以更灵活地操作图像。例如，通过更改图像数据可以创建一个简单的灰阶过滤器：

```
let drawing = document.getElementById("drawing");
// 确保浏览器支持<canvas>
if (drawing.getContext) {
  let context = drawing.getContext("2d"),
    image = document.images[0],
    imageData, data,
    i, len, average,
    red, green, blue, alpha;

  // 绘制图像
  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];

    // 取得 RGB 平均值
    average = Math.floor((red + green + blue) / 3);

    // 设置颜色，不管透明度
    data[i] = average;
    data[i+1] = average;
    data[i+2] = average;

  }

  // 将修改后的数据写回 ImageData 并应用到画布上显示出来
  imageData.data = data;
  context.putImageData(imageData, 0, 0);
}
```

这个例子首先在画布上绘制了一个图像，然后又取得了其图像数据。for 循环遍历了图像数据中的每个像素，注意每次循环都要给 i 加上 4。每次循环中取得红、绿、蓝的颜色值，计算出它们的平均值。然后再把原来的值修改为这个平均值，实际上相当于过滤掉了颜色信息，只留下类似亮度的灰度信息。之后将 data 数组重写回 imageData 对象。最后调用 putImageData()方法，把图像数据再绘制到画布上。结果就得到了原始图像的黑白版。

当然，灰阶过滤只是基于原始像素值可以实现的其中一种操作。要了解基于原始图像数据还可以实现哪些操作，可以参考 Ilmari Heikkinen 的文章“Making Image Filters with Canvas”。

> **注意** 只有在画布没有加载跨域内容时才可以获取图像数据。如果画布上绘制的是跨域内容，则尝试获取图像数据会导致 JavaScript 报错。

18.3.11 合成

2D 上下文中绘制的所有内容都会应用两个属性：globalAlpha 和 globalComposition Operation，其中，globalAlpha 属性是一个范围在 0~1 的值（包括 0 和 1），用于指定所有绘制内容的透明度，默

认值为 0。如果所有后来的绘制都需要使用同样的透明度，那么可以将 globalAlpha 设置为适当的值，执行绘制，然后再把 globalAlpha 设置为 0。比如：

```
// 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
context.globalAlpha = 0;
```

在这个例子中，蓝色矩形是绘制在红色矩形上面的。因为在绘制蓝色矩形前 globalAlpha 被设置成了 0.5，所以蓝色矩形就变成半透明了，从而可以透过它看到下面的红色矩形。

globalCompositionOperation 属性表示新绘制的形状如何与上下文中已有的形状融合。这个属性是一个字符串，可以取下列值。

- source-over：默认值，新图形绘制在原有图形上面。
- source-in：新图形只绘制出与原有图形重叠的部分，画布上其余部分全部透明。
- source-out：新图形只绘制出不与原有图形重叠的部分，画布上其余部分全部透明。
- source-atop：新图形只绘制出与原有图形重叠的部分，原有图形不受影响。
- destination-over：新图形绘制在原有图形下面，重叠部分只有原图形透明像素下的部分可见。
- destination-in：新图形绘制在原有图形下面，画布上只剩下二者重叠的部分，其余部分完全透明。
- destination-out：新图形与原有图形重叠的部分完全透明，原图形其余部分不受影响。
- destination-atop：新图形绘制在原有图形下面，原有图形与新图形不重叠的部分完全透明。
- lighter：新图形与原有图形重叠部分的像素值相加，使该部分变亮。
- copy：新图形将擦除并完全取代原有图形。
- xor：新图形与原有图形重叠部分的像素执行“异或”计算。

以上合成选项的含义很难用语言来表达清楚，只用黑白图像也体现不出所有合成的效果。下面来看一个例子：

```
// 绘制红色矩形
context.fillStyle = "#ff0000";
context.fillRect(10, 10, 50, 50);
// 设置合成方式
context.globalCompositeOperation = "destination-over";
// 绘制蓝色矩形
context.fillStyle = "rgba(0,0,255,1)";
context.fillRect(30, 30, 50, 50);
```

虽然后绘制的蓝色矩形通常会出现在红色矩形上面，但将 globalCompositeOperation 属性的值修改为"destination-over"意味着红色矩形会出现在蓝色矩形上面。

使用 globalCompositeOperation 属性时，一定记得要在不同浏览器上进行测试。不同浏览器在实现这些选项时可能存在差异。这些操作在 Safari 和 Chrome 中仍然有些问题，可以参考 MDN 文档上的 CanvasRenderingContext2D.globalCompositeOperation，比较它们与 IE 或 Firefox 渲染的差异。

18.4 WebGL

WebGL 是画布的 3D 上下文。与其他 Web 技术不同，WebGL 不是 W3C 制定的标准，而是 Khronos Group 的标准。根据官网描述，“Khronos Group 是非营利性、会员资助的联盟，专注于多平台和设备下并行计算、图形和动态媒体的无专利费开放标准”。Khronos Group 也制定了其他图形 API，包括作为浏览器中 WebGL 基础的 OpenGL ES 2.0。

OpenGL 这种 3D 图形语言很复杂，本书不会涉及过多相关概念。不过，要使用 WebGL 最好熟悉 OpenGL ES 2.0，因为很多概念可以照搬过来。

本节假设读者了解 OpenGL ES 2.0 的基本概念，并简单介绍 OpenGL ES 2.0 在 WebGL 中实现的部分。要了解关于 OpenGL 的更多信息，可以访问 OpenGL 网站。另外，推荐一个 WebGL 教程网站：Learn WebGL。

> **注意** 定型数组是在 WebGL 中执行操作的重要数据结构。第 6 章中讨论了定型数组。

18.4.1 WebGL 上下文

在完全支持的浏览器中，WebGL 2.0 上下文的名字叫`"webgl2"`，WebGL 1.0 上下文的名字叫`"webgl1"`。如果浏览器不支持 WebGL，则尝试访问 WebGL 上下文会返回 `null`。在使用上下文之前，应该先检测返回值是否存在：

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

// 确保浏览器支持<canvas>
if (drawing.getContext) {
  let gl = drawing.getContext("webgl");
  if (gl){
    // 使用 WebGL
  }
}
```

这里把 WebGL `context` 对象命名为 `gl`。大多数 WebGL 应用和例子遵循这个约定，因为 OpenGL ES 2.0 方法和值通常以`"gl"`开头。这样可以让 JavaScript 代码看起来更接近 OpenGL 程序。

18.4.2 WebGL 基础

取得 WebGL 上下文后，就可以开始 3D 绘图了。如前所述，因为 WebGL 是 OpenGL ES 2.0 的 Web 版，所以本节讨论的概念实际上是 JavaScript 所实现的 OpenGL 概念。

可以在调用 `getContext()`取得 WebGL 上下文时指定一些选项。这些选项通过一个参数对象传入，选项就是参数对象的一个或多个属性。

- `alpha`：布尔值，表示是否为上下文创建透明通道缓冲区，默认为 `true`。
- `depth`：布尔值，表示是否使用 16 位深缓冲区，默认为 `true`。
- `stencil`：布尔值，表示是否使用 8 位模板缓冲区，默认为 `false`。
- `antialias`：布尔值，表示是否使用默认机制执行抗锯齿操作，默认为 `true`。
- `premultipliedAlpha`：布尔值，表示绘图缓冲区是否预乘透明度值，默认为 `true`。
- `preserveDrawingBuffer`：布尔值，表示绘图完成后是否保留绘图缓冲区，默认为 `false`。建议在充分了解这个选项的作用后再自行修改，因为这可能会影响性能。

可以像下面这样传入 options 对象：

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

// 确保浏览器支持<canvas>
if (drawing.getContext) {

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

这些上下文选项大部分适合开发高级功能。多数情况下，默认值就可以满足要求。

如果调用 getContext()不能创建 WebGL 上下文，某些浏览器就会抛出错误。为此，最好把这个方法调用包装在 try/catch 块中：

```
Insert IconMargin     [download]let drawing = document.getElementById("drawing"),
  gl;

// 确保浏览器支持<canvas>
if (drawing.getContext) {
  try {
    gl = drawing.getContext("webgl");
  } catch (ex) {
    // 什么也不做
  }
  if (gl) {
    // 使用 WebGL
  } else {
    alert("WebGL context could not be created.");
  }
}
```

1. 常量

如果你熟悉 OpenGL，那么可能知道用于操作的各种常量。这些常量在 OpenGL 中的名字以 GL_开头。在 WebGL 中，context 对象上的常量则不包含 GL_前缀。例如，GL_COLOR_BUFFER_BIT 常量在 WebGL 中要这样访问 gl.COLOR_BUFFER_BIT。WebGL 以这种方式支持大部分 OpenGL 常量（少数常量不支持）。

2. 方法命名

OpenGL（同时也是 WebGL）中的很多方法会包含相关的数据类型信息。接收不同类型和不同数量参数的方法，会通过方法名的后缀体现这些信息。表示参数数量的数字（1~4）在先，表示数据类型的字符串（“f”表示浮点数，“i”表示整数）在后。比如，gl.uniform4f()的意思是需要 4 个浮点数值参数，而 gl.uniform3i()表示需要 3 个整数值参数。

还有很多方法接收数组，这类方法用字母“v”（vector）来表示。因此，gl.uniform3iv()就是要接收一个包含 3 个值的数组参数。在编写 WebGL 代码时，要记住这些约定。

3. 准备绘图

准备使用 WebGL 上下文之前，通常需要先指定一种实心颜色清除<canvas>。为此，要调用 clearColor()方法并传入 4 个参数，分别表示红、绿、蓝和透明度值。每个参数必须是 0~1 范围内的值，表示各个组件在最终颜色的强度。比如：

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

以上代码把清理颜色缓冲区的值设置为黑色，然后调用 clear()方法，这个方法相当于 OpenGL 中的 glClear()方法。参数 gl.COLOR_BUFFER_BIT 告诉 WebGL 使用之前定义的颜色填充画布。通常，所有绘图操作之前都需要先清除绘制区域。

4. 视口与坐标

绘图前还要定义 WebGL 视口。默认情况下，视口使用整个<canvas>区域。要改变视口，可以调用 viewport()方法并传入视口相对于<canvas>元素的 x、y 坐标及宽度和高度。例如，以下代码表示要使用整个<canvas>元素：

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


这个视口的坐标系统与网页中通常的坐标系统不一样。视口的 x 和 y 坐标起点(0, 0)表示<canvas>元素的左下角，向上、向右增长可以用点(width–1, height–1)定义（见图 18-14）。

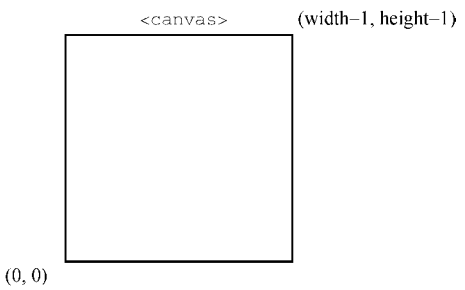

图 18-14

知道如何定义视口就可以只使用<canvas>元素的一部分来绘图。比如下面的例子：

```
// 视口是<canvas> 左下角四分之一区域
gl.viewport(0, 0, drawing.width/2, drawing.height/2);
// 视口是<canvas> 左上角四分之一区域
gl.viewport(0, drawing.height/2, drawing.width/2, drawing.height/2);
// 视口是<canvas> 右下角四分之一区域
gl.viewport(drawing.width/2, 0, drawing.width/2, drawing.height/2);
```

定义视口的坐标系统与视口中的坐标系统不一样。在视口中，坐标原点(0, 0)是视口的中心点。左下角是(–1, –1)，右上角是(1, 1)，如图 18-15 所示。

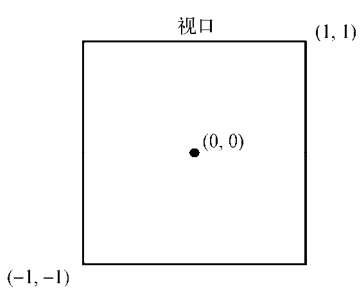

图 18-15

如果绘图时使用了视口外部的坐标，则绘制结果会被视口剪切。例如，要绘制的形状有一个顶点在(1, 2)，则视口右侧的图形会被切掉。

5. 缓冲区

在 JavaScript 中，顶点信息保存在定型数组中。要使用这些信息，必须先把它们转换为 WebGL 缓冲区。创建缓冲区要调用 `gl.createBuffer()`方法，并使用 `gl.bindBuffer()`方法将缓冲区绑定到 WebGL 上下文。绑定之后，就可以用数据填充缓冲区了。比如：

```
let 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()`将 `buffer` 设置为上下文的当前缓冲区。然后，所有缓冲区操作都在 `buffer` 上直接执行。因此，调用 `gl.bufferData()`虽然没有包含对 `buffer` 的直接引用，但仍然是在它上面执行的。上面最后一行代码使用一个 `Float32Array`（通常把所有顶点信息保存在 `Float32Array` 中）初始化了 `buffer`。如果想输出缓冲区内容，那么可以调用 `drawElements()`方法并传入 `gl.ELEMENT_ARRAY_BUFFER`。

`gl.bufferData()`方法的最后一个参数表示如何使用缓冲区。这个参数可以是以下常量值。

- `gl.STATIC_DRAW`：数据加载一次，可以在多次绘制中使用。
- `gl.STREAM_DRAW`：数据加载一次，只能在几次绘制中使用。
- `gl.DYNAMIC_DRAW`：数据可以重复修改，在多次绘制中使用。

除非是很有经验的 OpenGL 程序员，否则我们会对大多数缓冲区使用 `gl.STATIC_DRAW`。

缓冲区会一直驻留在内存中，直到页面卸载。如果不再需要缓冲区，那么最好调用 `gl.deleteBuffer()`方法释放其占用的内存：

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

6. 错误

与 JavaScript 多数情况下不同的是，在 WebGL 操作中通常不会抛出错误。必须在调用可能失败的方法后，调用 `gl.getError()`方法。这个方法返回一个常量，表示发生的错误类型。下面列出了这些常量。

- `gl.NO_ERROR`：上一次操作没有发生错误（0 值）。
- `gl.INVALID_ENUM`：上一次操作没有传入 WebGL 预定义的常量。
- `gl.INVALID_VALUE`：上一次操作需要无符号数值，但是传入了负数。
- `gl.INVALID_OPERATION`：上一次操作在当前状态下无法完成。
- `gl.OUT_OF_MEMORY`：上一次操作因内存不足而无法完成。
- `gl.CONTEXT_LOST_WEBGL`：上一次操作因外部事件（如设备掉电）而丢失了 WebGL 上下文。

每次调用 `gl.getError()`方法会返回一个错误值。第一次调用之后，再调用 `gl.getError()`可能会返回另一个错误值。如果有多个错误，则可以重复这个过程，直到 `gl.getError()`返回 `gl.NO_ERROR`。如果执行了多次操作，那么可以通过循环调用 `getError()`：

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

如果 WebGL 代码没有产出想要的输出结果，那么可以调用几次 `getError()`，这样有可能帮你找

到问题所在。

7. 着色器

着色器是 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**。顶点着色器用于把 3D 顶点转换为可以渲染的 2D 点。片段着色器用于计算绘制一个像素的正确颜色。WebGL 着色器的独特之处在于，它们不是 JavaScript 实现的，而是使用一种与 C 或 JavaScript 完全不同的语言 GLSL（OpenGL Shading Language）写的。

- **编写着色器**

GLSL 是一种类似于 C 的语言，专门用于编写 OpenGL 着色器。因为 WebGL 是 OpenGL ES 2 的实现，所以 OpenGL 中的着色器可以直接在 WebGL 中使用。这样也可以让桌面应用更方便地移植到 Web 上。

每个着色器都有一个 `main()`方法，在绘制期间会重复执行。给着色器传递数据的方式有两种：`attribute` 和 `uniform`。`attribute` 用于将顶点传入顶点着色器，而 `uniform` 用于将常量值传入任何着色器。`attribute` 和 `uniform` 是在 `main()`函数外部定义的。在值类型关键字之后是数据类型，然后是变量名。下面是一个简单的顶点着色器的例子：

```
// OpenGL 着色器语言
// 着色器，摘自 Bartek Drozdz 的文章“Get started with WebGL—draw a square”
attribute vec2 aVertexPosition;

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

这个顶点着色器定义了一个名为 `aVertexPosition` 的 `attribute`。这个 `attribute` 是一个包含两项的数组（数据类型为 `vec2`），代表 x 和 y 坐标。即使只传入了两个坐标，顶点着色器返回的值也会包含 4 个元素，保存在变量 `gl_Position` 中。这个着色器创建了一个新的包含 4 项的数组（`vec4`），缺少的坐标会补充上，实际上是把 2D 坐标转换为了 3D 坐标。

片段着色器与顶点着色器类似，只不过是通过 `uniform` 传入数据。下面是一个片段着色器的例子：

```
// OpenGL 着色器语言
// 着色器，摘自 Bartek Drozdz 的文章“Get started with WebGL—draw a square”
uniform vec4 uColor;

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

片段着色器必须返回一个值，保存到变量 `gl_FragColor` 中，这个值表示绘制时使用的颜色。这个着色器定义了一个 `uniform`，包含颜色的 4 个组件（`vec4`），保存在 `uColor` 中。从代码上看，这个着色器只是把传入的值赋给了 `gl_FragColor`。`uColor` 的值在着色器内不能改变。

> **注意** OpenGL 着色器语言比示例中的代码要复杂，详细介绍需要整本书的篇幅。因此，本节只是从使用 WebGL 的角度对这门语言做个极其简单的介绍。要了解更多信息，可以参考 Randi J. Rost 的著作《OpenGL 着色语言》。

- **创建着色器程序**

浏览器并不理解原生 GLSL 代码，因此 GLSL 代码的字符串必须经过编译并链接到一个着色器程序

中。为便于使用，通常可以使用带有自定义 type 属性的<script>元素把着色器代码包含在网页中。如果 type 属性无效，则浏览器不会解析<script>的内容，但这并不妨碍读写其中的内容：

```
<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>
```

然后可以使用 text 属性提取<script>元素的内容：

```
let vertexGlsl = document.getElementById("vertexShader").text,
    fragmentGlsl = document.getElementById("fragmentShader").text;
```

更复杂的 WebGL 应用可以动态加载着色器。重点在于要使用着色器，必须先拿到 GLSL 代码的字符串。

有了 GLSL 字符串，下一步是创建 shader 对象。为此，需要调用 gl.createShader()方法，并传入想要创建的着色器类型（gl.VERTEX_SHADER 或 gl.FRAGMENT_SHADER）。然后，调用 gl.shaderSource()方法把 GLSL 代码应用到着色器，再调用 gl.compileShader()编译着色器。下面是一个例子：

```
let vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexGlsl);
gl.compileShader(vertexShader);
let fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentGlsl);
gl.compileShader(fragmentShader);
```

这里的代码创建了两个着色器，并把它们保存在 vertexShader 和 fragmentShader 中。然后，可以通过以下代码把这两个对象链接到着色器程序：

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

第一行代码创建了一个程序，然后 attachShader()用于添加着色器。调用 gl.linkProgram()将两个着色器链接到了变量 program 中。链接到程序之后，就可以通过 gl.useProgram()方法让 WebGL 上下文使用这个程序了：

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

调用 gl.useProgram()之后，所有后续的绘制操作都会使用这个程序。

- **给着色器传值**

前面定义的每个着色器都需要传入一个值，才能完成工作。要给着色器传值，必须先找到要接收值的变量。对于 uniform 变量，可以调用 gl.getUniformLocation()方法。这个方法返回一个对象，

表示该 uniform 变量在内存中的位置。然后，可以使用这个位置来完成赋值。比如：

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

这个例子从 program 中找到 uniform 变量 uColor，然后返回了它的内存位置。第二行代码调用 gl.uniform4fv()方法给 uColor 传入了值。

给顶点着色器传值也是类似的过程。而要获得 attribute 变量的位置，可以调用 gl.getAttrib-Location()方法。找到变量的内存地址后，可以像下面这样给它传入值：

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


这里，首先取得 aVertexPosition 的内存位置，然后使用 gl.enableVertexAttribArray()来启用。最后一行代码创建了一个指向调用 gl.bindBuffer()指定的缓冲区的指针，并把它保存在 aVertexPosition 中，从而可以在后面由顶点着色器使用。

- **调试着色器和程序**

与 WebGL 中的其他操作类似，着色器操作也可能失败，而且是静默失败。如果想知道发生了什么错误，则必须手工通过 WebGL 上下文获取关于着色器或程序的信息。

对于着色器，可以调用 gl.getShaderParameter()方法取得编译之后的编译状态：

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

这个例子检查了 vertexShader 编译的状态。如果着色器编译成功，则调用 gl.getShaderParameter()会返回 true。如果返回 false，则说明编译出错了。此时，可以使用 gl.getShaderInfoLog()并传入着色器取得错误。这个方法返回一个字符串消息，表示问题所在。gl.getShaderParameter()和 gl.getShaderInfoLog()既可以用于顶点着色器，也可以用于片段着色器。

着色器程序也可能失败，因此也有类似的方法。gl.getProgramParameter()用于检测状态。最常见的程序错误发生在链接阶段，为此可以使用以下代码来检查：

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

与 gl.getShaderParameter()一样，gl.getProgramParameter()会在链接成功时返回 true，失败时返回 false。当然也有一个 gl.getProgramInfoLog()方法，可以在程序失败时获取错误信息。

这些方法主要在开发时用于辅助调试。只要没有外部依赖，在产品环境中就可以放心地删除它们。

- **GLSL 100 升级到 GLSL 300**

WebGL2 的主要变化是升级到了 GLSL 3.00 ES 着色器。这个升级暴露了很多新的着色器功能，包括 3D 纹理等在支持 OpenGL ES 3.0 的设备上都有的功能。要使用升级版的着色器，着色器代码的第一行必须是：

```
#version 300 es
```

这个升级需要一些语法的变化。

❑ 顶点 attribute 变量要使用 in 而不是 attribute 关键字声明。