- 使用 varying 关键字为顶点或片段着色器声明的变量，现在必须根据相应着色器的行为改为使用 in 或 out。
- 预定义的输出变量 gl_FragColor 没有了，片段着色器必须为颜色输出声明自己的 out 变量。
- 纹理查找函数 texture2D 和 textureCube 统一成了一个 texture 函数。

8. 绘图

WebGL 只能绘制三种形状：点、线和三角形。其他形状必须通过这三种基本形状在 3D 空间的组合来绘制。WebGL 绘图要使用 drawArrays()和 drawElements()方法，前者使用数组缓冲区，后者则操作元素数组缓冲区。

drawArrays()和 drawElements()的第一个参数都表示要绘制形状的常量。下面列出了这些常量。

- gl.POINTS：将每个顶点当成一个点来绘制。
- gl.LINES：将数组作为一系列顶点，在这些顶点间绘制直线。每个顶点既是起点也是终点，因此数组中的顶点必须是偶数个才能开始绘制。
- gl.LINE_LOOP：将数组作为一系列顶点，在这些顶点间绘制直线。从第一个顶点到第二个顶点绘制一条直线，再从第二个顶点到第三个顶点绘制一条直线，以此类推，直到绘制到最后一个顶点。此时再从最后一个顶点到第一个顶点绘制一条直线。这样就可以绘制出形状的轮廓。
- gl.LINE_STRIP:类似于 gl.LINE_LOOP,区别在于不会从最后一个顶点到第一个顶点绘制直线。
- gl.TRIANGLES：将数组作为一系列顶点，在这些顶点间绘制三角形。如不特殊指定，每个三角形都分开绘制，不共享顶点。
- gl.TRIANGLES_STRIP：类似于 gl.TRIANGLES，区别在于前 3 个顶点之后的顶点会作为第三个顶点与其前面的两个顶点构成三角形。例如，如果数组中包含顶点 *A*、*B*、*C*、*D*，那么第一个三角形使用 *ABC*，第二个三角形使用 *BCD*。
- gl.TRIANGLES_FAN：类似于 gl.TRIANGLES，区别在于前 3 个顶点之后的顶点会作为第三个顶点与其前面的顶点和第一个顶点构成三角形。例如，如果数组中包含顶点 *A*、*B*、*C*、*D*，那么第一个三角形使用 *ABC*，第二个三角形使用 *ACD*。

以上常量可以作为 gl.drawArrays()方法的第一个参数，第二个参数是数组缓冲区的起点索引，第三个参数是数组缓冲区包含的顶点集合的数量。以下代码使用 gl.drawArrays()在画布上绘制了一个三角形：

```
// 假设已经使用本节前面的着色器清除了视口
// 定义 3 个顶点的 x 坐标和 y 坐标
let vertices = new Float32Array([ 0, 1, 1, -1, -1, -1 ]),
    buffer = gl.createBuffer(),
    vertexSetSize = 2,
    vertexSetCount = vertices.length/vertexSetSize,
    uColor,
    aVertexPosition;
// 将数据放入缓冲区
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
// 给片段着色器传入颜色
uColor = gl.getUniformLocation(program, "uColor");
gl.uniform4fv(uColor, [ 0, 0, 0, 1 ]);
// 把顶点信息传给着色器
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);
```

这个例子定义了一个 Float32Array 变量，它包含 3 组两个点的顶点。完成计算的关键是跟踪顶点大小和数量。将 vertexSetSize 的值指定为 2，再计算出 vertexSetCount。顶点信息保存在了缓冲区。然后把颜色信息传给片段着色器。

接着给顶点着色器传入顶点集的大小，以及表示顶点坐标数值类型的 gl.FlOAT。第四个参数是一个布尔值，表示坐标不是标准的。第五个参数是**步长值**（stride value），表示跳过多个数组元素取得下一个值。除非真要跳过一些值，否则就向这里传入 0 即可。最后一个参数是起始偏移量，这里的 0 表示从第一个数组元素开始。

最后一步是使用 gl.drawArrays()把三角形绘制出来。通过把第一个参数指定为 gl.TRIANGLES，就可以从(0, 1)到(1, –1)再到(–1, –1)绘制一个三角形，并填充传给片段着色器的颜色。第二个参数表示缓冲区的起始偏移量，最后一个参数是要读取的顶点数量。以上绘图操作的结果如图 18-16 所示。

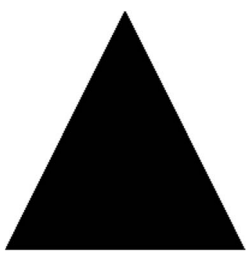

图 18-16

通过改变 gl.drawArrays()的第一个参数，可以修改绘制三角形的方式。图 18-17 展示了修改第一个参数之后的两种输出。

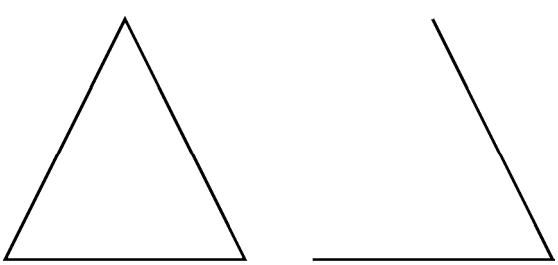

图 18-17

9. 纹理

WebGL 纹理可以使用 DOM 中的图片。可以使用 gl.createTexture()方法创建新的纹理，然后再将图片绑定到这个纹理。如果图片还没有加载，则可以创建一个 Image 对象来动态加载。图片加载完成后才能初始化纹理，因此在图片的 load 事件之后才能使用纹理。比如：

```
let 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);

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

除了使用 DOM 图片，这些步骤跟在 OpenGL 中创建纹理是一样的。最大的区别在于使用 `gl.pixelStorei()`设置了像素存储格式。常量 `gl.UNPACK_FLIP_Y_WEBGL` 是 WebGL 独有的，在基于 Web 加载图片时通常要使用。原因在于 GIF、JPEG 和 PNG 图片使用的坐标系统与 WebGL 内部的坐标系统不一样。如果不使用这个标志，图片就会倒过来。

用于纹理的图片必须跟当前页面同源，或者是来自启用了跨源资源共享（CORS，Cross-Origin Resource Sharing）的服务器上。

> **注意**　纹理来源可以是图片、通过<video>元素加载的视频，甚至是别的<canvas>元素。视频同样受跨源限制。

10. 读取像素

与 2D 上下文一样，可以从 WebGL 上下文中读取像素数据。读取像素的 `readPixels()`方法与 OpenGL 中的方法有同样的参数，只不过最后一个参数必须是定型数组。像素信息是从帧缓冲区读出来并放到这个定型数组中的。`readPixels()`方法的参数包括 *x* 和 *y* 坐标、宽度、高度、图像格式、类型和定型数组。前 4 个参数用于指定要读取像素的位置。图像格式参数几乎总是 `gl.RGBA`。类型参数指的是要存储在定型数组中的数据类型，有如下限制：

- 如果这个类型是 `gl.UNSIGNED_BYTE`，则定型数组必须是 `Uint8Array`；
- 如果这个类型是 `gl.UNSIGNED_SHORT_5_6_5`、`gl.UNSIGNED_SHORT_4_4_4_4` 或 `gl.UNSIGNED_SHORT_5_5_5_1`，则定型数组必须是 `Uint16Array`。

下面是一个调用 `readPixels()`方法的例子：

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

以上代码读取了帧缓冲区中 25 像素×25 像素大小的区域，并把读到的像素信息保存在 `pixels` 数组中，其中每个像素的颜色在这个数组中都以 4 个值表示，分别代表红、绿、蓝和透明度值。每个数组值的取值范围是 0~255（包括 0 和 255）。别忘了先按照预期存储的数据量初始化定型数组。

在浏览器绘制更新后的 WebGL 图像之前调用 `readPixels()`没有问题。而在绘制完成后，帧缓冲区会恢复到其初始清除状态，此时调用 `readPixels()`会得到与清除状态一致的像素数据。如果想在绘制之后读取像素，则必须使用前面讨论过的 `preserveDrawingBuffer` 选项初始化 WebGL 上下文：

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

设置这个标志可以强制帧缓冲区在下一次绘制之前保持上一次绘制的状态。这个选项可能会影响性能，因此尽量不要使用。

18.4.3 WebGL1 与 WebGL2

WebGL1 代码几乎完全与 WebGL2 兼容。在使用 WebGL2 上下文时，唯一可能涉及修改代码以保证兼容性的就是扩展。在 WebGL2 中，很多扩展都变成了默认功能。

例如，要在 WebGL1 中使用绘制缓冲区，需要先测试相应扩展后再使用：

```
let ext = gl.getExtension('WEBGL_draw_buffers');

if (!ext) {
  // 没有扩展的代码
} else {
  ext.drawBuffersWEBGL([...])
}
```


而在 WebGL2 中，这里的检测代码就不需要了，因为这个扩展已经直接暴露在上下文对象上了：

```
gl.drawBuffers([...]);
```

以下特性都已成为 WebGL2 的标准特性：

- ANGLE_instanced_arrays
- EXT_blend_minmax
- EXT_frag_depth
- EXT_shader_texture_lod
- OES_element_index_uint
- OES_standard_derivatives
- OES_texture_float
- OES_texture_float_linear
- OES_vertex_array_object
- WEBGL_depth_texture
- WEBGL_draw_buffers
- Vertex shader texture access

> **注意** 要了解 WebGL 更新的内容，可以参考 WebGL2Fundamentals 网站上的文章“WebGL2 from WebGL1”。

18.5 小结

requestAnimationFrame 是简单但实用的工具，可以让 JavaScript 跟进浏览器渲染周期，从而更加有效地实现网页视觉动效。

HTML5 的<canvas>元素为 JavaScript 提供了动态创建图形的 API。这些图形需要使用特定上下文绘制，主要有两种。第一种是支持基本绘图操作的 2D 上下文：

- 填充和描绘颜色及图案
- 绘制矩形
- 绘制路径
- 绘制文本
- 创建渐变和图案

第二种是 3D 上下文，也就是 WebGL。WebGL 是浏览器对 OpenGL ES 2.0 的实现。OpenGL ES 2.0 是游戏图形开发常用的一个标准。WebGL 支持比 2D 上下文更强大的绘图能力，包括：

- 用 OpenGL 着色器语言（GLSL）编写顶点和片段着色器；
- 支持定型数组，限定数组中包含数值的类型；
- 创建和操作纹理。

目前所有主流浏览器的较新版本都已经支持<canvas>标签。

第 19 章
表单脚本

本章内容
- ❑ 理解表单基础
- ❑ 文本框验证与交互
- ❑ 使用其他表单控件

JavaScript 较早的一个用途是承担一部分服务器端表单处理的责任。虽然 Web 和 JavaScript 都已经发展了很多年，但 Web 表单的变化不是很大。由于不能直接使用表单解决问题，因此开发者不得不使用 JavaScript 既做表单验证，又用于增强标准表单控件的默认行为。

19.1 表单基础

Web 表单在 HTML 中以<form>元素表示，在 JavaScript 中则以 HTMLFormElement 类型表示。HTMLFormElement 类型继承自 HTMLElement 类型，因此拥有与其他 HTML 元素一样的默认属性。不过，HTMLFormElement 也有自己的属性和方法。

- ❑ acceptCharset：服务器可以接收的字符集，等价于 HTML 的 accept-charset 属性。
- ❑ action：请求的 URL，等价于 HTML 的 action 属性。
- ❑ elements：表单中所有控件的 HTMLCollection。
- ❑ enctype：请求的编码类型，等价于 HTML 的 enctype 属性。
- ❑ length：表单中控件的数量。
- ❑ method：HTTP 请求的方法类型，通常是"get"或"post"，等价于 HTML 的 method 属性。
- ❑ name：表单的名字，等价于 HTML 的 name 属性。
- ❑ reset()：把表单字段重置为各自的默认值。
- ❑ submit()：提交表单。
- ❑ target：用于发送请求和接收响应的窗口的名字，等价于 HTML 的 target 属性。

有几种方式可以取得对<form>元素的引用。最常用的是将表单当作普通元素为它指定一个 id 属性，从而可以使用 getElementById()来获取表单，比如：

```
let form = document.getElementById("form1");
```

此外，使用 document.forms 集合可以获取页面上所有的表单元素。然后，可以进一步使用数字索引或表单的名字（name）来访问特定的表单。比如：

```
// 取得页面中的第一个表单
let firstForm = document.forms[0];

// 取得名字为"form2"的表单
let myForm = document.forms["form2"];
```

较早的浏览器，或者严格向后兼容的浏览器，也会把每个表单的 name 作为 document 对象的属性。例如，名为"form2"的表单可以通过 document.form2 来访问。不推荐使用这种方法，因为容易出错，而且这些属性将来可能会被浏览器删除。

注意，表单可以同时拥有 id 和 name，而且两者可以不相同。

19.1.1　提交表单

表单是通过用户点击提交按钮或图片按钮的方式提交的。提交按钮可以使用 type 属性为"submit"的<input>或<button>元素来定义，图片按钮可以使用 type 属性为"image"的<input>元素来定义。点击下面例子中定义的所有按钮都可以提交它们所在的表单：

```
<!-- 通用提交按钮 -->
<input type="submit" value="Submit Form">

<!-- 自定义提交按钮 -->
<button type="submit">Submit Form</button>

<!-- 图片按钮 -->
<input type="image" src="graphic.gif">
```

如果表单中有上述任何一个按钮，且焦点在表单中某个控件上，则按回车键也可以提交表单。（textarea 控件是个例外，当焦点在它上面时，按回车键会换行。）注意，没有提交按钮的表单在按回车键时不会提交。

以这种方式提交表单会在向服务器发送请求之前触发 submit 事件。这样就提供了一个验证表单数据的机会，可以根据验证结果决定是否真的要提交。阻止这个事件的默认行为可以取消提交表单。例如，下面的代码会阻止表单提交：

```
let form = document.getElementById("myForm");

form.addEventListener("submit", (event) => {
  // 阻止表单提交
  event.preventDefault();
});
```

调用 preventDefault()方法可以阻止表单提交。通常，在表单数据无效以及不应该发送到服务器时可以这样处理。

当然，也可以通过编程方式在 JavaScript 中调用 submit()方法来提交表单。可以在任何时候调用这个方法来提交表单，而且表单中不存在提交按钮也不影响表单提交。下面是一个例子：

```
let form = document.getElementById("myForm");

// 提交表单
form.submit();
```

通过 submit()提交表单时，submit 事件不会触发。因此在调用这个方法前要先做数据验证。

表单提交的一个最大的问题是可能会提交两次表单。如果提交表单之后没有什么反应，那么没有耐心的用户可能会多次点击提交按钮。结果是很烦人的（因为服务器要处理重复的请求），甚至可能造成损失（如果用户正在购物，则可能会多次下单）。解决这个问题主要有两种方式：在表单提交后禁用提交按钮，或者通过 onsubmit 事件处理程序取消之后的表单提交。

19.1.2 重置表单

用户单击重置按钮可以重置表单。重置按钮可以使用 type 属性为"reset"的<input>或<button>元素来创建，比如：

```
<!-- 通用重置按钮 -->
<input type="reset" value="Reset Form">

<!-- 自定义重置按钮 -->
<button type="reset">Reset Form</button>
```

这两种按钮都可以重置表单。表单重置后，所有表单字段都会重置回页面第一次渲染时各自拥有的值。如果字段原来是空的，就会变成空的；如果字段有默认值，则恢复为默认值。

用户单击重置按钮重置表单会触发 reset 事件。这个事件为取消重置提供了机会。例如，以下代码演示了如何阻止重置表单：

```
let form = document.getElementById("myForm");

form.addEventListener("reset", (event) => {
  event.preventDefault();
});
```

与表单提交一样，重置表单也可以通过 JavaScript 调用 reset()方法来完成，如下面的例子所示：

```
let form = document.getElementById("myForm");

// 重置表单
form.reset();
```

与 submit()方法的功能不同，调用 reset()方法会像单击了重置按钮一样触发 reset 事件。

> **注意** 表单设计中通常不提倡重置表单，因为重置表单经常会导致用户迷失方向，如果意外触发则会令人感到厌烦。实践中几乎没有重置表单的需求。一般来说，提供一个取消按钮，让用户点击返回前一个页面，而不是恢复表单中所有的值来得更直观。

19.1.3 表单字段

表单元素可以像页面中的其他元素一样使用原生 DOM 方法来访问。此外，所有表单元素都是表单 elements 属性（元素集合）中包含的一个值。这个 elements 集合是一个有序列表，包含对表单中所有字段的引用，包括所有<input>、<textarea>、<button>、<select>和<fieldset>元素。elements 集合中的每个字段都以它们在 HTML 标记中出现的次序保存，可以通过索引位置和 name 属性来访问。以下是几个例子：

```
let form = document.getElementById("form1");

// 取得表单中的第一个字段
let field1 = form.elements[0];

// 取得表单中名为"textbox1"的字段
let field2 = form.elements["textbox1"];

// 取得字段的数量
let fieldCount = form.elements.length;
```

如果多个表单控件使用了同一个 name，比如像单选按钮那样，则会返回包含所有同名元素的 HTMLCollection。比如，来看下面的 HTML 代码片段：

```
<form method="post" id="myForm">
  <ul>
    <li><input type="radio" name="color" value="red">Red</li>
    <li><input type="radio" name="color" value="green">Green</li>
    <li><input type="radio" name="color" value="blue">Blue</li>
  </ul>
</form>
```

这个 HTML 中的表单有 3 个单选按钮的 name 是"color"，这个名字把它们联系在了一起。在访问 elements["color"]时，返回的 NodeList 就包含这 3 个元素。而在访问 elements[0]时，只会返回第一个元素。比如：

```
let form = document.getElementById("myForm");

let colorFields = form.elements["color"];
console.log(colorFields.length);  // 3

let firstColorField = colorFields[0];
let firstFormField = form.elements[0];
console.log(firstColorField === firstFormField);   // true
```

以上代码表明，使用 form.elements[0]获取的表单的第一个字段就是 form.elements["color"]中包含的第一个元素。

> **注意**　也可以通过表单属性的方式访问表单字段，比如 form[0]这种使用索引和 form["color"]这种使用字段名字的方式。访问这些属性与访问 form.elements 集合是一样的。这种方式是为向后兼容旧版本浏览器而提供的，实际开发中应该使用 elements。

1. 表单字段的公共属性

除<fieldset>元素以外，所有表单字段都有一组同样的属性。由于<input>类型可以表示多种表单字段，因此某些属性只适用于特定类型的字段。除此之外的属性可以在任何表单字段上使用。以下列出了这些表单字段的公共属性和方法。

- disabled：布尔值，表示表单字段是否禁用。
- form：指针，指向表单字段所属的表单。这个属性是只读的。
- name：字符串，这个字段的名字。
- readOnly：布尔值，表示这个字段是否只读。
- tabIndex：数值，表示这个字段在按 Tab 键时的切换顺序。
- type：字符串，表示字段类型，如"checkbox"、"radio"等。
- value：要提交给服务器的字段值。对文件输入字段来说，这个属性是只读的，仅包含计算机上某个文件的路径。

这里面除了 form 属性以外，JavaScript 可以动态修改任何属性。来看下面的例子：

```
let form = document.getElementById("myForm");
let field = form.elements[0];

// 修改字段的值
```

```
field.value = "Another value";

// 检查字段所属的表单
console.log(field.form === form);   // true

// 给字段设置焦点
field.focus();

// 禁用字段
field.disabled = true;

// 改变字段的类型（不推荐，但对<input>来说是可能的）
field.type = "checkbox";
```

这种动态修改表单字段属性的能力为任何时候以任何方式修改表单提供了方便。举个例子，Web 表单的一个常见问题是用户常常会点击两次提交按钮。在涉及信用卡扣款的情况下，这是个严重的问题，可能会导致重复扣款。对此，常见的解决方案是第一次点击之后禁用提交按钮。可以通过监听 submit 事件来实现。比如下面这个例子：


```
// 避免多次提交表单的代码
let form = document.getElementById("myForm");

form.addEventListener("submit", (event) => {
  let target = event.target;

  // 取得提交按钮
  let btn = target.elements["submit-btn"];

  // 禁用提交按钮
  btn.disabled = true;
});
```

以上代码在表单的 submit 事件上注册了一个事件处理程序。当 submit 事件触发时，代码会取得提交按钮，然后将其 disabled 属性设置为 true。注意，这个功能不能通过直接给提交按钮添加 onclick 事件处理程序来实现，原因是不同浏览器触发事件的时机不一样。有些浏览器会在触发表单的 submit 事件前先触发提交按钮的 click 事件，有些浏览器则会后触发 click 事件。对于先触发 click 事件的浏览器，这个按钮会在表单提交前被禁用，这意味着表单就不会被提交了。因此最好使用表单的 submit 事件来禁用提交按钮。但这种方式不适用于没有使用提交按钮的表单提交。如前所述，只有提交按钮才能触发 submit 事件。

type 属性可以用于除<fieldset>之外的任何表单字段。对于<input>元素，这个值等于 HTML 的 type 属性值。对于其他元素，这个 type 属性的值按照下表设置。

描　述	示例 HTML	类型的值
单选列表	<select>...</select>	"select-one"
多选列表	<select multiple>...</select>	"select-multiple"
自定义按钮	<button>...</button>	"submit"
自定义非提交按钮	<button type="button">...</button>	"button"
自定义重置按钮	<button type="reset">...</button>	"reset"
自定义提交按钮	<button type="submit">...</button>	"submit"