



# WebGL中文版

---

极客学院出版

# 前言

---

WebGL 是一种 3D 绘图标准，这种绘图技术标准允许把 JavaScript 和 OpenGL ES 2.0 结合在一起，通过增加 OpenGL ES 2.0 的一个 JavaScript 绑定，WebGL 可以为 HTML5 Canvas 提供硬件 3D 加速渲染，这样 Web 开发人员就可以借助系统显卡来在浏览器里更流畅地展示 3D 场景和模型了，还能创建复杂的导航和数据视觉化。

本教程是 WebGL 的最新版介绍，文章区别于网上其他同类的内容，内容全面且新颖，旨在通过简洁的介绍，让读者明白 WebGL 是如何使用的。

## 适用人群

开发 3D 模型和场景的程序员。

## 学习前提

WebGL 涉及 JavaScript、HTML 代码，所以需要你掌握以上技能，如果你对 2D、3D 绘图有一定的了解，那么本教程学习起来将会很容易。

英文出处：<http://webglfundamentals.org/>

# 目录

---

前言 .....	1
第 1 章    WebGL 基础 .....	4
WebGL 基本原理 .....	5
WebGL 是如何工作的 .....	11
WebGL 着色器和 GLSL .....	20
第 2 章    图像处理 .....	28
WebGL 图像处理 .....	29
WebGL 图像处理（续） .....	36
第 3 章    2D 转换、旋转、伸缩、矩阵 .....	41
WebGL 2D 图像转换 .....	42
WebGL 2D 图像旋转 .....	47
WebGL 2D 图像伸缩 .....	52
WebGL 2D 矩阵 .....	54
第 4 章    3D .....	66
WebGL 正交 3D .....	67
WebGL 3D 透视 .....	80
WebGL 3D 摄像机 .....	89
第 5 章    结构与组织 .....	98
WebGL – 更少的代码，更多的乐趣 .....	99
WebGL 绘制多个东西 .....	110
WebGL 场景图 .....	120
第 6 章    文本 .....	133
WebGL 文本 HTML .....	134

WebGL 文本 Canvas 2D .....	141
WebGL 文本 纹理 .....	144
WebGL 文本 使用字符纹理.....	157



## WebGL 基础



## WebGL 基本原理

---

WebGL 的出现使得在浏览器上面实时显示 3D 图像成为可能，WebGL 本质上是基于光栅化的 API，而不是基于 3D 的 API。

WebGL 只关注两个方面，即投影矩阵的坐标和投影矩阵的颜色。使用 WebGL 程序的任务就是实现具有投影矩阵坐标和颜色的 WebGL 对象即可。可以使用“着色器”来完成上述任务。顶点着色器可以提供投影矩阵的坐标，片段着色器可以提供投影矩阵的颜色。

无论要实现的图形尺寸有多大，其投影矩阵的坐标的范围始终是从 -1 到 1。下面是一个关于实现 WebGL 对象的一个简单例子。

```
// Get A WebGL context
var canvas = document.getElementById("canvas");
var gl = canvas.getContext("experimental-webgl");

// setup a GLSL program
var program = createProgramFromScripts(gl, ["2d-vertex-shader", "2d-fragment-shader"]);
gl.useProgram(program);

// look up where the vertex data needs to go.
var positionLocation = gl.getAttribLocation(program, "a_position");

// Create a buffer and put a single clip-space rectangle in
// it (2 triangles)
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array([
    -1.0, -1.0,
    1.0, -1.0,
    -1.0, 1.0,
    1.0, 1.0,
    1.0, -1.0,
    1.0, 1.0]),
  gl.STATIC_DRAW);
gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);

// draw
gl.drawArrays(gl.TRIANGLES, 0, 6);
```

下面是两个着色器。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

void main() {
    gl_Position = vec4(a_position, 0, 1);
}
</script>

<script id="2d-fragment-shader" type="x-shader/x-fragment">
void main() {
    gl_FragColor = vec4(0, 1, 0, 1); // green
}
</script>
```

它将绘出一个绿色的长方形来填充整个画板。



后面内容还会更精彩，我们继续：-P

我们再次降调一下，无论画板尺寸多大，投影矩阵坐标的范围只会在  $-1$  到  $1$  之间。从上面的例子中，我们可以看出我们只是将位置信息直接写在了程序里。因为位置信息已经在投影矩阵中，所以并没有其他额外的工作要做。如果想实现 3D 的效果，那么可以使用着色器来将 3D 转换为投影矩阵，这是因为 WebGL 是基于光栅的 API。

对于 2D 的图像，也许会使用像素而不是投影矩阵来表述尺寸，那么这里我们就更改这里的着色器，使得我们实现的矩形可以以像素的方式来度量，下面是新的顶点着色器。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
```

```

uniform vec2 u_resolution;

void main() {
    // convert the rectangle from pixels to 0.0 to 1.0
    vec2 zeroToOne = a_position / u_resolution;

    // convert from 0->1 to 0->2
    vec2 zeroToTwo = zeroToOne * 2.0;

    // convert from 0->2 to -1->+1 (clipSpace)
    vec2 clipSpace = zeroToTwo - 1.0;

    gl_Position = vec4(clipSpace, 0, 1);
}
</script>

```

下面我们将我们的数据从投影矩阵改为像素。

```

// set the resolution
var resolutionLocation = gl.getUniformLocation(program, "u_resolution");
gl.uniform2f(resolutionLocation, canvas.width, canvas.height);

// setup a rectangle from 10,20 to 80,30 in pixels
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
    10, 20,
    80, 20,
    10, 30,
    10, 30,
    80, 20,
    80, 30]), gl.STATIC_DRAW);

```





上面例子矩阵位于底部边框，下面我们让它位于左上边框：

修改代码如下：

```
gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
```

效果如下图：



下面我们将上述关于矩阵的实现写成函数以便可以以函数调用的方式来实现不同尺寸的矩阵。然而，这里的颜色应该是可变的。

首先，我们为片段着色器设计一个关于颜色的输入。

```

<script id="2d-fragment-shader" type="x-shader/x-fragment">
precision mediump float;

uniform vec4 u_color;

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

```

下面是实现绘画 50 个尺寸和颜色均随机的矩阵的代码。

```

var colorLocation = gl.getUniformLocation(program, "u_color");
...
// Create a buffer
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.enableVertexAttribArray(positionLocation);
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);

// draw 50 random rectangles in random colors
for (var ii = 0; ii < 50; ++ii) {
// Setup a random rectangle
setRectangle(
gl, randomInt(300), randomInt(300), randomInt(300), randomInt(300));

// Set a random color.
gl.uniform4f(colorLocation, Math.random(), Math.random(), Math.random(), 1);

// Draw the rectangle.
gl.drawArrays(gl.TRIANGLES, 0, 6);
}
}

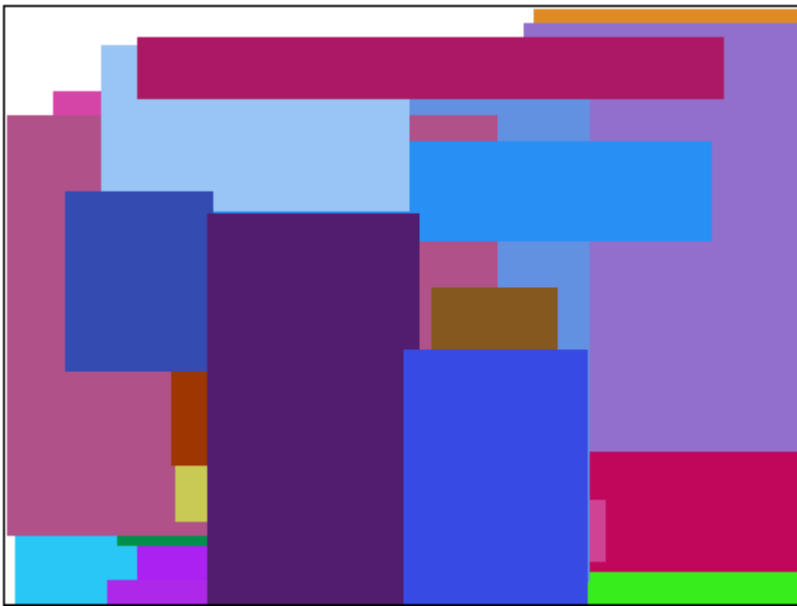
// Returns a random integer from 0 to range - 1.
function randomInt(range) {
    return Math.floor(Math.random() * range);
}

// Fills the buffer with the values that define a rectangle.
function setRectangle(gl, x, y, width, height) {
    var x1 = x;
    var x2 = x + width;
    var y1 = y;
    var y2 = y + height;
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([

```

```
x1, y1,  
x2, y1,  
x1, y2,  
x1, y2,  
x2, y1,  
x2, y2]), gl.STATIC_DRAW);  
}
```

下面就是实现出来的效果。



到此可以看出 WebGL 实质上是一种轻量级的 API。但是，它可以实现较为复杂的 3D 效果，其复杂性由程序定制。WebGL API 本身是 2D 的且相对比较简单。

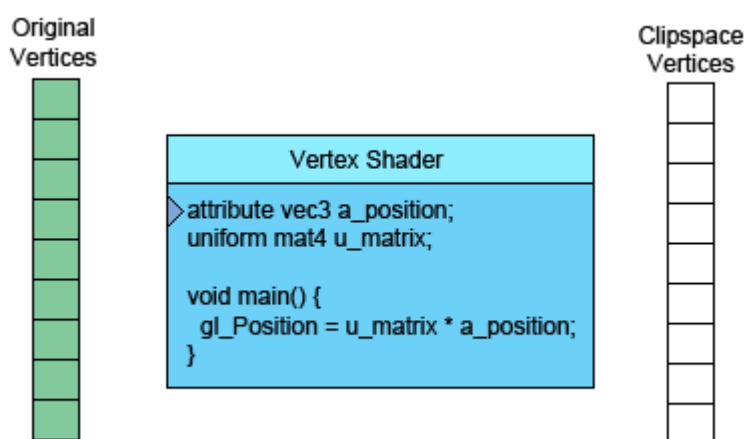
## WebGL 是如何工作的

这部分是上一节[WebGL 基本原理](#)的延续。在继续之前，我们需要讨论 WebGL 和 GPU 是如何运作的。GPU 有两个基础任务，第一个就是将点处理为投影矩阵。第二部分就是基于第一部分将相应的像素点描绘出来。

当用户调用

```
gl.drawArrays(gl.TRIANGLE, 0, 9);
```

这里的 9 就意味着“处理 9 个顶点”，所以就有 9 个顶点需要被处理。



上图左侧的是用户自己提供的数据。定点着色器就是用户在 GLSL 中写的函数。处理每个定点时，均会被调用一次。用户可以将投影矩阵的值存储在特定的变量 `gl_Position` 中。GPU 会处理这些值，并将他们存储在其内部。

假设用户希望绘制三角形 `TRIANGLES`，那么每次绘制时，上述的第一部分就会产生三个顶点，然后 GPU 会使用他们来绘制三角形。首先 GPU 会将三个顶点对应的像素绘制出来，然后将三角形光栅化，或者说是使用像素点绘制出来。对每一个像素点，GPU 都会调用用户定义的片段着色器来确定该像素点该涂成什么颜色。当然，用户定义的片段着色器必须在 `gl_FragColor` 变量中设置对应的值。

我们例子中的片段着色器中并没有存储每一个像素的信息。我们可以在其中存储更丰富的信息。我们可以为每一个值定义不同的意义从定点着色器传递到片段着色器。

作为一个简单的例子，我们将直接计算出来的投影矩阵坐标从定点着色器传递给片段着色器。

我们将绘制一个简单的三角形。我们在[上个例子](#)的基础上更改一下。

```
// Fill the buffer with the values that define a triangle.
function setGeometry(gl) {
```

```
gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array([
    0, -100,
    150, 125,
    -175, 100]),
  gl.STATIC_DRAW);
}
```

然后，我们绘制三个顶点。

```
// Draw the scene.
function drawScene() {
  ...
  // Draw the geometry.
  gl.drawArrays(gl.TRIANGLES, 0, 3);
}
```

然后，我们可以在顶点着色器中定义变量来将数据传递给片段着色器。

```
varying vec4 v_color;
...
void main() {
  // Multiply the position by the matrix.
  gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);

  // Convert from clip space to color space.
  // Clip space goes -1.0 to +1.0
  // Color space goes from 0.0 to 1.0
  v_color = gl_Position * 0.5 + 0.5;
}
```

然后，我们在片段着色器中声明相同的变量。

```
precision mediump float;

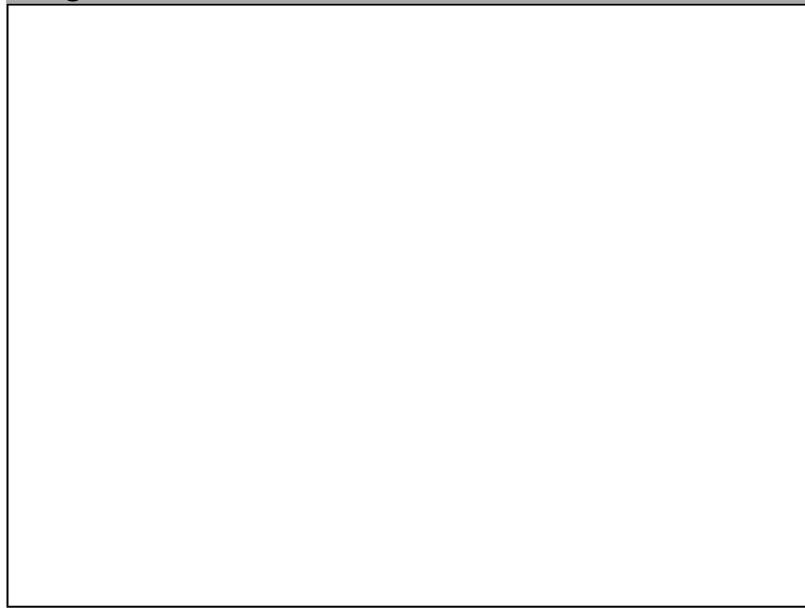
varying vec4 v_color;

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

WebGL 将会连接顶点着色器中的变量和片段着色器中的相同名字和类型的变量。

下面是可以交互的版本。

Drag sliders to translate, rotate, and scale.



移动、缩放或旋转这个三角形。注意由于颜色是从投影矩阵计算而来，所以，颜色并不会随着三角形的移动而一直一样。他们完全是根据背景色设定的。

现在我们考虑下面的内容。我们仅仅计算三个顶点。我们的顶点着色器被调用了三次，因此，仅仅计算了三个颜色。而我们的三角形可以有好多颜色，这就是为何被称为 *varying*。

WebGL 使用了我们为每个定点计算的三个值，然后将三角形光栅化。对于每一个像素，都会使用被修改过的值来调用片段着色器。

基于上述例子，我们以三个顶点开始。

Vertices	
0	-100
150	125
-175	100

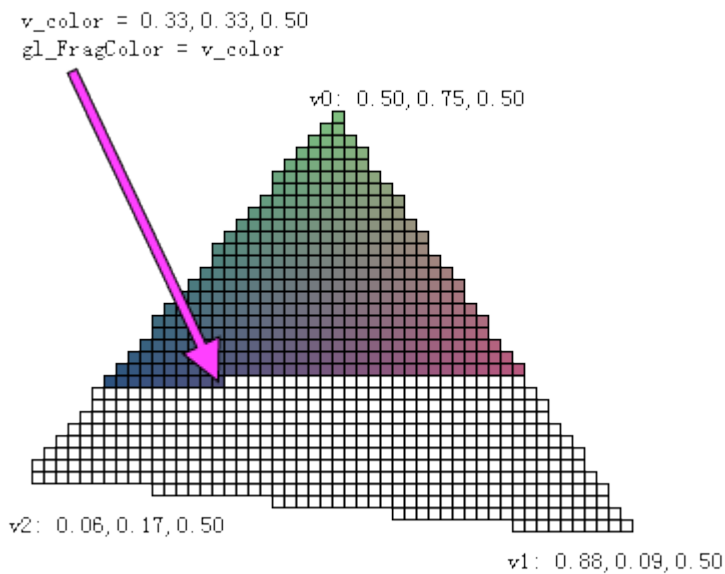
我们的顶点着色器会引用矩阵来转换、旋转、缩放和转化为投影矩阵。转换、旋转和缩放的默认值是转换为200,150，旋转为 0，缩放为 1,1，所以实际上只进行转换。我们的后台缓存是 400x300。我们的顶点矩阵应用矩阵然后计算下面的三个投影矩阵顶点。

values written to gl_Position	
0.000	0.660
0.750	-0.830
-0.875	-0.660

同样也会将这些转换到颜色空间上，然后将他们写到我们声明的多变变量 `v_color`。

values written to v_color		
0.5000	0.750	0.5
0.8750	0.915	0.5
0.0625	0.170	0.5

这三个值会写回到 v\_color，然后它会被传递到片段着色器用于每一个像素进行着色。



v\_color 被修改为 v0, v1 和 v2 三个值中的一个。

我们也可以在顶点着色器中存储更多的数据以便往片段着色器中传递。所以，对于以两种颜色绘制包含两个三角形的矩形的例子。为了实现这个例子，我们需要往顶点着色器中附加更多的属性，以便传递更多的数据，这些数据会直接传递到片段着色器中。

```
attribute vec2 a_position;
attribute vec4 a_color;
...
varying vec4 v_color;

void main() {
    ...
    // Copy the color from the attribute to the varying.
    v_color = a_color;
}
```

我们现在需要使用 WebGL 颜色相关的功能。

```
// look up where the vertex data needs to go.
var positionLocation = gl.getAttribLocation (program, "a_position");
```

```

var colorLocation = gl.getAttribLocation(program, "a_color");
...
// Create a buffer for the colors.
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.enableVertexAttribArray(colorLocation);
gl.vertexAttribPointer(colorLocation, 4, gl.FLOAT, false, 0, 0);

// Set the colors.
setColors(gl);

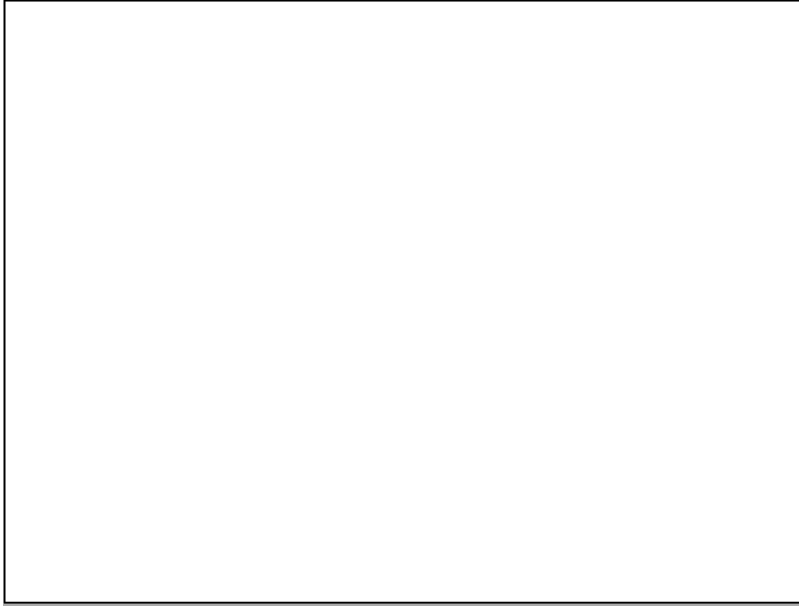
// Fill the buffer with colors for the 2 triangles
// that make the rectangle.
function setColors(gl) {
  // Pick 2 random colors.
  var r1 = Math.random();
  var b1 = Math.random();
  var g1 = Math.random();
  var r2 = Math.random();
  var b2 = Math.random();
  var g2 = Math.random();
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array(
      [ r1, b1, g1, 1,
        r1, b1, g1, 1,
        r1, b1, g1, 1,
        r2, b2, g2, 1,
        r2, b2, g2, 1,
        r2, b2, g2, 1]),
    gl.STATIC_DRAW);
} `

```

下面是结果。



### Drag sliders to translate, rotate, and scale.

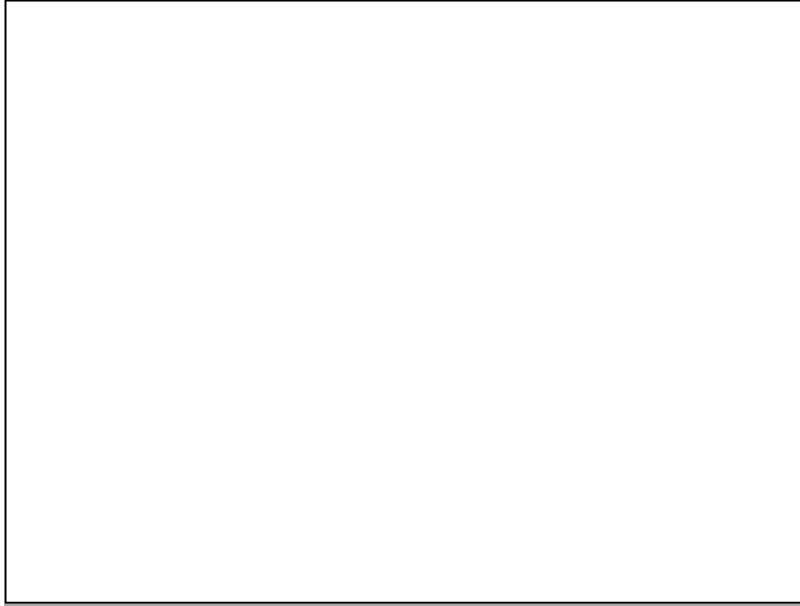


注意，在上面的例子中，有两个顶点颜色的三角形。我们仍将要传递的值存储在多变变量中，所以，该变量会相关三角形区域内改变。我们只是对于每个三角形的三个顶点使用了相同的颜色。如果我们使用了不同的颜色，我们可以看到整个渲染过程。

```
/ Fill the buffer with colors for the 2 triangles
// that make the rectangle.
function setColors(gl) {
  // Make every vertex a different color.
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array(
      [ Math.random(), Math.random(), Math.random(), 1,
        Math.random(), Math.random(), Math.random(), 1,
        Math.random(), Math.random(), Math.random(), 1,
        Math.random(), Math.random(), Math.random(), 1,
        Math.random(), Math.random(), Math.random(), 1,
        Math.random(), Math.random(), Math.random(), 1]),
    gl.STATIC_DRAW);
}
```

现在我们看一下被渲染后的多变变量。

Drag sliders to translate, rotate, and scale.



从定点着色器往片段着色器可以传递更多更丰富的数据。如果我们来检验[图像处理示例](#)，就会发现在纹理坐标中会传递更多的属性。

## 缓存和属性指令究竟做了什么？

缓存是获取定点和顶点相关数据到 GPU 中的方法。`gl.createBuffer` 用于创建缓存。`gl.bindBuffer` 方法用于将缓存激活来处于准备工作的状态。`gl.bufferData` 方法可以将数据拷贝到缓存中。

一旦，数据到了缓存中，就需要告诉 WebGL 如何从里面错去数据，并将它提供给顶点着色器以给相应的属性赋值。

为了实现这个功能，首先我们需要 WebGL 提供一个属性存储位置。下面是示例代码。

```
// look up where the vertex data needs to go.
var positionLocation = gl.getAttribLocation(program, "a_position");
var colorLocation = gl.getAttribLocation(program, "a_color");
```

一旦我们知道了对应的属性，我们可以触发两个指令。

```
gl.enableVertexAttribArray(location);
```

这个指令会告诉 WebGL 我们希望将缓存中的数据赋值给一个变量。

```
gl.vertexAttribPointer(
  location,
  numComponents,
  typeOfData,
  normalizeFlag,
```

```
strideToNextPieceOfData,
offsetIntoBuffer);
```

这个指令会告诉 WebGL 会从缓存中获取数据，这个缓存会与 `gl.bindBuffer` 绑定。每个顶点可以有 1 到 4 个部件，数据的类型可以是 `BYTE`, `FLOAT`, `INT`, `UNSIGNED_SHORT` 等。跳跃意味着从数据的这片到那片会跨越多少个字节。跨越多远会以偏移量的方式存储在缓存中。

部件的数目一般会是 1 到 4。

如果每个数据类型仅使用一个缓存，那么跨越和偏移量都会是 0。跨越为 0 意味着“使用一个跨越连匹配类型和尺寸”。偏移量为 0 意味着是在缓存的开头部分。将这个值赋值为除 0 之外其他的值会实现更为灵活的功能。虽然在性能方面它有些优势，但是并不值得搞得很复杂，除非程序员希望将 WebGL 运用到极致。

我希望到此就可以将缓冲区和属性相关内容已经介绍清楚了。

下面我们学习[着色器和 GLSL](#)。

## 什么是顶点属性指针 `vertexAttribPointer` 的规范化标志 `normalizeFlag` ?

规范化标志应用于非浮点指针类型。如果该值置为 `false`, 就意味着该值就会被翻译为类型。 `BYTE` 的标示范围是 -128 到 127。 `UNSIGNED_BYTE` 范围是 0 到 255, `SHORT` 是从 -32768 到 32767。

如果将规范化标志置为 `true`, 那么 `BYTE` 的标示范围将为变为 -1.0 到 +1.0, `UNSIGNED_BYTE` 将会变为 0.0 到 +1.0, 规范化后的 `SHORT` 将会变为 -1.0 到 +1.0, 它将有比 `BYTE` 更高的精确度。

标准化数据最通用的地方就是用于颜色。大部分时候，颜色范围为 0.0 到 1.0 红色、绿色和蓝色需要个浮点型的值来表示，alpha 需要 16 字节来表示顶点的每个颜色。如果要实现更为复杂的图形，可以增加更多的字节。相反的，程序可以将颜色转为 `UNSIGNED_BYTE` 类型，这个类型使用 0 表示 0.0，使用 255 表示 1.0。那么仅需要 4 个字节来表示顶点的每个颜色，这将节省 75% 的存储空间。

我们按照下面的方式来更改我们的代码。当我们告诉 WebGL 如何获取颜色。

```
gl.vertexAttribPointer(colorLocation, 4, gl.UNSIGNED_BYTE, true, 0, 0);
```

我们可以使用下面的代码来填充我们的缓冲区。

```
// Fill the buffer with colors for the 2 triangles
// that make the rectangle.
function setColors(gl) {
  // Pick 2 random colors.
  var r1 = Math.random() * 256; // 0 to 255.99999
  var b1 = Math.random() * 256; // these values
```

```
var g1 = Math.random() * 256; // will be truncated
var r2 = Math.random() * 256; // when stored in the
var b2 = Math.random() * 256; // Uint8Array
var g2 = Math.random() * 256;

gl.bufferData(
  gl.ARRAY_BUFFER,
  new Uint8Array( // Uint8Array
    [ r1, b1, g1, 255,
      r1, b1, g1, 255,
      r1, b1, g1, 255,
      r2, b2, g2, 255,
      r2, b2, g2, 255,
      r2, b2, g2, 255]),
  gl.STATIC_DRAW);
}
```

下面是个例子。

Drag sliders to translate, rotate, and scale.



## WebGL 着色器和 GLSL

---

我们已经讨论了一些关于着色器和 GLSL 的相关内容，但是仍然介绍的不是很仔细。可以通过一个例子来更清楚地了解的更清楚。

正如 [WebGL 是如何工作](#)中讲到的，每次绘制，都需要两个着色器，分别是顶点着色器和片段着色器。每个着色器都是一个函数。顶点着色器和片段着色器都是链接在程序中的。一个典型的 WebGL 程序都会包含很多这样的着色器程序。

### 顶点着色器

顶点着色器的任务就是产生投影矩阵的坐标。其形式如下：

```
void main() {  
    gl_Position = doMathToMakeClipspaceCoordinates  
}
```

每一个顶点都会调用你的着色器。每次调用程序都需要设置特定的全局变量 `gl_Position` 来表示投影矩阵的坐标。

顶点着色器需要数据，它以下面三种方式来获取这些数据。

- 属性（从缓冲区中获取数据）
- 一致变量（每次绘画调用时都保持一致的值）
- 纹理（从像素中得到的数据）

#### 属性

最常用的方式就是使用缓存区和属性。[WebGL 如何工作](#)中已经涵盖了缓冲区和属性。

程序可以以下面的方式创建缓存区。

```
var buf = gl.createBuffer();
```

在这些缓存中存储数据。

```
gl.bindBuffer(gl.ARRAY_BUFFER, buf);  
gl.bufferData(gl.ARRAY_BUFFER, someData, gl.STATIC_DRAW);
```

于是，给定一个着色器程序，程序可以去查找属性的位置。

```
var positionLoc = gl.getAttribLocation(someShaderProgram, "a_position");
```

下面告诉 WebGL 如何从缓存区中获取数据并存储到属性中。

```
// turn on getting data out of a buffer for this attribute
gl.enableVertexAttribArray(positionLoc);

var numComponents = 3; // (x, y, z)
var type = gl.FLOAT;
var normalize = false; // leave the values as they are
var offset = 0; // start at the beginning of the buffer
var stride = 0; // how many bytes to move to the next vertex
// 0 = use the correct stride for type and numComponents

gl.vertexAttribPointer(positionLoc, numComponents, type, false, stride, offset);
```

在 [WebGL 基本原理](#) 中我们展示了我们可以在着色器中附加一些逻辑，然后将值直接传递。

```
attribute vec4 a_position;

void main() {
    gl_Position = a_position;
}
```

如果我们可以将投影矩阵放入我们的缓存区中，它就会开始运作。

属性可以使用 float, vec2, vec3, vec4, mat2, mat3 和 mat4 作为类型。

### 一致性变量

对于顶点着色器，一致性变量就是在绘画每次调用时，在着色器中一直保持不变的值。下面是一个往顶点中添加偏移量着色器的例子。

```
attribute vec4 a_position;
uniform vec4 u_offset;

void main() {
    gl_Position = a_position + u_offset;
}
```

下面，我们需要对每一个顶点都偏移一定量。首先，我们需要先找到一致变量的位置。

```
var offsetLoc = gl.getUniformLocation(someProgram, "u_offset");
```

然后，我们在绘制前需要设置一致性变量。

```
gl.uniform4fv(offsetLoc, [1, 0, 0, 0]); // offset it to the right half the screen
```

一致性变量可以有很多种类型。对每一种类型都可以调用相应的函数来设置。

```
gl.uniform1f(floatUniformLoc, v); // for float
gl.uniform1fv(floatUniformLoc, [v]); // for float or float array
gl.uniform2f(vec2UniformLoc, v0, v1); // for vec2
gl.uniform2fv(vec2UniformLoc, [v0, v1]); // for vec2 or vec2 array
gl.uniform3f(vec3UniformLoc, v0, v1, v2); // for vec3
gl.uniform3fv(vec3UniformLoc, [v0, v1, v2]); // for vec3 or vec3 array
gl.uniform4f(vec4UniformLoc, v0, v1, v2, v4); // for vec4
gl.uniform4fv(vec4UniformLoc, [v0, v1, v2, v4]); // for vec4 or vec4 array

gl.uniformMatrix2fv(mat2UniformLoc, false, [ 4x element array ]) // for mat2 or mat2 array
gl.uniformMatrix3fv(mat3UniformLoc, false, [ 9x element array ]) // for mat3 or mat3 array
gl.uniformMatrix4fv(mat4UniformLoc, false, [ 17x element array ]) // for mat4 or mat4 array

gl.uniform1i(intUniformLoc, v); // for int
gl.uniform1iv(intUniformLoc, [v]); // for int or int array
gl.uniform2i(ivec2UniformLoc, v0, v1); // for ivec2
gl.uniform2iv(ivec2UniformLoc, [v0, v1]); // for ivec2 or ivec2 array
gl.uniform3i(ivec3UniformLoc, v0, v1, v2); // for ivec3
gl.uniform3iv(ivec3UniformLoc, [v0, v1, v2]); // for ivec3 or ivec3 array
gl.uniform4i(ivec4UniformLoc, v0, v1, v2, v4); // for ivec4
gl.uniform4iv(ivec4UniformLoc, [v0, v1, v2, v4]); // for ivec4 or ivec4 array

gl.uniform1i(sampler2DUniformLoc, v); // for sampler2D (textures)
gl.uniform1iv(sampler2DUniformLoc, [v]); // for sampler2D or sampler2D array

gl.uniform1i(samplerCubeUniformLoc, v); // for samplerCube (textures)
gl.uniform1iv(samplerCubeUniformLoc, [v]); // for samplerCube or samplerCube array
```

一般类型都有 `bool`, `bvec2`, `bvec3` 和 `bvec4`。他们相应的调用函数形式为 `gl.uniform?f?` 或 `gl.uniform?i?`。

可以一次性设置数组中的所有一致性变量。比如：

```
// in shader
uniform vec2 u_someVec2[3];

// in JavaScript at init time
var someVec2Loc = gl.getUniformLocation(someProgram, "u_someVec2");

// at render time
gl.uniform2fv(someVec2Loc, [1, 2, 3, 4, 5, 6]); // set the entire array of u_someVec3
```

但是，如果程序希望单独设置数组中的成员，那么必须单个的查询每个成员的位置。

```
// in JavaScript at init time
var someVec2Element0Loc = gl.getUniformLocation(someProgram, "u_someVec2[0]");
var someVec2Element1Loc = gl.getUniformLocation(someProgram, "u_someVec2[1]");
var someVec2Element2Loc = gl.getUniformLocation(someProgram, "u_someVec2[2]");

// at render time
gl.uniform2fv(someVec2Element0Loc, [1, 2]); // set element 0
gl.uniform2fv(someVec2Element1Loc, [3, 4]); // set element 1
gl.uniform2fv(someVec2Element2Loc, [5, 6]); // set element 2
```

类似的，可以创建一个结构体。

```
struct SomeStruct {
    bool active;
    vec2 someVec2;
};
uniform SomeStruct u_someThing;
```

程序可以单独的查询每一个成员。

```
var someThingActiveLoc = gl.getUniformLocation(someProgram, "u_someThing.active");
var someThingSomeVec2Loc = gl.getUniformLocation(someProgram, "u_someThing.someVec2");
```

顶点着色器中的纹理

参考[片段着色器中的纹理](#)

## 片段着色器

片段着色器的任务就是为当前被栅格化的像素提供颜色。它通常以下面的方式呈现出来。

```
precision mediump float;

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

片段着色器对每一个像素都会调用一次。每次调用都会设置全局变量 `gl_FragColor` 来设置一些颜色。

片段着色器需要存储获取数据，通常有下面这三种方式。

- 一致变量（每次绘制像素点时都会调用且一直保持一致）
- 纹理（从像素中获取数据）
- 多变变量（从定点着色器中传递出来且被栅格化的值）



## 片段着色器中的一致变量

参考[顶点着色器中的一致变量](#)。

## 片段着色器中的纹理

我们可以从纹理中获取值来创建 `sampler2D` 一致变量, 然后使用 GLSL 函数 `texture2D` 来从中获取值。

```
precision mediump float;

uniform sampler2D u_texture;

void main() {
    vec2 texcoord = vec2(0.5, 0.5) // get a value from the middle of the texture
    gl_FragColor = texture2D(u_texture, texcoord);
}
```

从纹理中提取的值是要[依据很多设置](#)的。最基本的, 我们需要创建并在纹理中存储值。比如,

```
var tex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, tex);
var level = 0;
var width = 2;
var height = 1;
var data = new Uint8Array([255, 0, 0, 255, 0, 255, 0, 255]);
gl.texImage2D(gl.TEXTURE_2D, level, gl.RGBA, width, height, 0, gl.RGBA, gl.UNSIGNED_BYTE, data);
```

然后, 在着色器程序中查询一致变量的位置。

```
var someSamplerLoc = gl.getUniformLocation(someProgram, "u_texture");
```

WebGL 需要将它绑定到纹理单元中。

```
var unit = 5; // Pick some texture unit
gl.activeTexture(gl.TEXTURE0 + unit);
gl.bindTexture(gl.TEXTURE_2D, tex);
```

然后告知着色器哪个单元会被绑定到纹理中。

```
gl.uniform1i(someSamplerLoc, unit);
```

## 多变变量

多变变量是从顶点着色器往片段着色器中传递的值, 这些在[WebGL 如何工作的](#)已经涵盖了。

为了使用多变变量，需要在顶点着色器和片段着色器中均设置匹配的多变变量。我们在顶点着色器中设置多变变量。当 WebGL 绘制像素时，它会栅格化该值，然后传递到片段着色器中相对应的片段着色器。

### 顶点着色器

```
attribute vec4 a_position;

uniform vec4 u_offset;

varying vec4 v_positionWithOffset;

void main() {
    gl_Position = a_position + u_offset;
    v_positionWithOffset = a_position + u_offset;
}
```

### 片段着色器

```
precision mediump float;

varying vec4 v_positionWithOffset;

void main() {
    // convert from clipspace (-1 <-> +1) to color space (0 -> 1).
    vec4 color = v_positionWithOffset * 0.5 + 0.5;
    gl_FragColor = color;
}
```

上面的例子是无关紧要的例子。它一般不会直接复制投影矩阵的值到片段着色器中。

## GLSL

GLSL是图像库着色器语言的简称。语言着色器就是被写在这里。它具有一些 JavaScript 中不存在的独特的特性。它用于实现一些逻辑来渲染图像。比如，它可以创建类似于 `vec2`，`vec3` 和 `vec4` 分别表示2、3、4个值。类似的，`mat2`，`mat3` 和 `mat4` 来表示 2x2,3x3,4x4 的矩阵。可以实现 `vec` 来乘以一个标量。

```
vec4 a = vec4(1, 2, 3, 4);
vec4 b = a * 2.0;
// b is now vec4(2, 4, 6, 8);
```

类似的，可以实现矩阵的乘法和矩阵的向量乘法。

```
mat4 a = ???
mat4 b = ???
mat4 c = a * b;

vec4 v = ???
vec4 y = c * v;
```

也可以选择 vec 的部分，比如，vec4

```
vec4 v;
```

- v.x 等价于 v.s, v.r, v[0]
- v.y 等价于 v.t, v.g, v[1]
- v.z 等价于 v.p, v.b, v[2]
- v.w 等价于 v.q, v.a, v[3]

可以调整 vec 组件意味着可以交换或重复组件。

```
v.yyyy
```

这等价于

```
vec4(v.y, v.y, v.y, v.y)
```

类似的

```
v.bgra
```

等价于

```
vec4(v.b, v.g, v.r, v.a)
```

当创建一个 vec 或一个 mat 时，程序可以一次操作多个部分。比如，

```
vec4(v.rgb, 1)
```

这等价于

```
vec4(v.r, v.g, v.b, 1)
```

你可能意识到 GLSL 是一种很严格类型的语言。

```
float f = 1; // ERROR 1 is an int. You can't assign an int to a float
```

正确的方式如下：

```
float f = 1.0; // use float
float f = float(1) // cast the integer to a float
```

上面例子的 `vec4(v.rgb, 1)` 并不会对 1 进行混淆，这是因为 `vec4` 是类似于 `float (1)`。

GLSL 是内置函数的分支.可以同时操作多个组件。比如，

```
T sin(T angle)
```

这意味着 T 可以是 `float`，`vec2`，`vec3` 或 `vec4`。如果用户在 `vec4` 中传递数据。也就是说 v 是 `vec4`，

```
vec4 s = sin(v);
```

折等价于

```
vec4 s = vec4(sin(v.x), sin(v.y), sin(v.z), sin(v.w));
```

有时候，一个参数是 `float`，另一个是 T。这意味着 `float` 将应用到所有的部件。比如，如果 `v1`，`v2` 是 `vec4`，`f` 是 `float`，然后

```
vec4 m = mix(v1, v2, f);
```

这等价于

```
vec4 m = vec4(
    mix(v1.x, v2.x, f),
    mix(v1.y, v2.y, f),
    mix(v1.z, v2.z, f),
    mix(v1.w, v2.w, f));
```

可以在 [WebGL 参考目录](#)的最后一页查看 GLSL 函数。如果你希望查看一些仔细的可以查看 [GLSL 规范](#)。

## 混合起来看

WebGL 是关于创建各种着色器的，将数据存储在这些着色器上，然后调用 `gl.drawArrays` 或 `gl.drawElements` 来使 WebGL 处理顶点通过为每个顶点调用当前的顶点着色器，然后为每一个像素调用当前的片段着色器。

实际上，着色器的创建需要写几行代码。因为，这些代码在大部分 WebGL 程序中的是一样的，又因为一旦写了，可以几乎可以忽略他们如何编译 GLSL 着色器和链接成一个着色器程序。



图像处理



## WebGL 图像处理

---

在 WebGL 中图像处理是很简单的，多么简单？

为了在 WebGL 中绘制图像，我们需要使用纹理。类似于当渲染代替像素时，WebGL 会需要操作投影矩阵的坐标，WebGL 读取纹理时需要获取纹理坐标。纹理坐标范围是从 0.0 到 1.0。

因为我们仅需要绘制由两个三角形组成的矩形，我们需要告诉 WebGL 在矩阵中纹理对应的那个点。我们可以使用特殊的被称为多变量，会将这些信息从顶点着色器传递到片段着色器。WebGL 将会插入这些值，这些值会在顶点着色器中，当对每个像素绘制时均会调用片段着色器。

我们需要在纹理坐标传递过程中添加更多的信息，然后将他们传递到片段着色器中。

```
attribute vec2 a_texCoord;
...
varying vec2 v_texCoord;

void main() {
    ...
    // pass the texCoord to the fragment shader
    // The GPU will interpolate this value between points
    v_texCoord = a_texCoord;
}
```

然后，我们提供一个片段着色器来查找颜色纹理。

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">
precision mediump float;

// our texture
uniform sampler2D u_image;

// the texCoords passed in from the vertex shader.
varying vec2 v_texCoord;

void main() {
    // Look up a color from the texture.
    gl_FragColor = texture2D(u_image, v_texCoord);
}
</script>
```

最后，我们需要加载一个图片，然后创建一个问题，将该图片传递到纹理里面。因为，是在浏览器里面显示，所以图片是异步加载，所以我们安置我们的代码来等待纹理的加载。一旦，加载完成就可以绘制。

```
function main() {
  var image = new Image();
  image.src = "http://someimage/on/our/server"; // MUST BE SAME DOMAIN!!!
  image.onload = function() {
    render(image);
  }
}

function render(image) {
  ...
  // all the code we had before.
  ...
  // look up where the texture coordinates need to go.
  var texCoordLocation = gl.getAttribLocation(program, "a_texCoord");

  // provide texture coordinates for the rectangle.
  var texCoordBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
    0.0, 0.0,
    1.0, 0.0,
    0.0, 1.0,
    0.0, 1.0,
    1.0, 0.0,
    1.0, 1.0]), gl.STATIC_DRAW);
  gl.enableVertexAttribArray(texCoordLocation);
  gl.vertexAttribPointer(texCoordLocation, 2, gl.FLOAT, false, 0, 0);

  // Create a texture.
  var texture = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, texture);

  // Set the parameters so we can render any size image.
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

  // Upload the image into the texture.
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
  ...
}
```

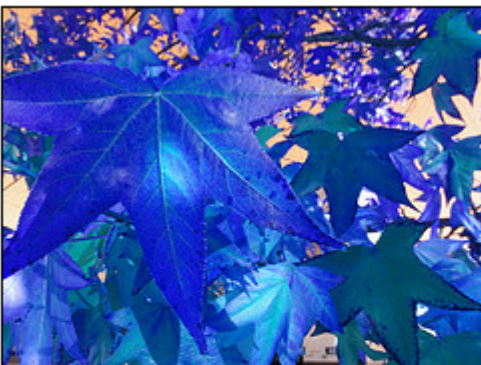
如下是 WebGL 渲染出来的图像。



下面我们对这个图片进行一些操作，来交换图片中的红色和蓝色。

```
...  
gl_FragColor = texture2D(u_image, v_texCoord).bgra;  
...
```

现在红色和蓝色已经被交换了。效果如下：



假如我们想做一些图像处理，那么我们可以看一下其他像素。自从 WebGL 引用纹理的纹理坐标从 0.0 到 1.0。然后，我们可以计算移动的多少个像素  $\text{onePixel} = 1.0 / \text{textureSize}$ 。



这里有个片段着色器来平均纹理中每个像素的左侧和右侧的像素。

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">
precision mediump float;

// our texture
uniform sampler2D u_image;
uniform vec2 u_textureSize;

// the texCoords passed in from the vertex shader.
varying vec2 v_texCoord;

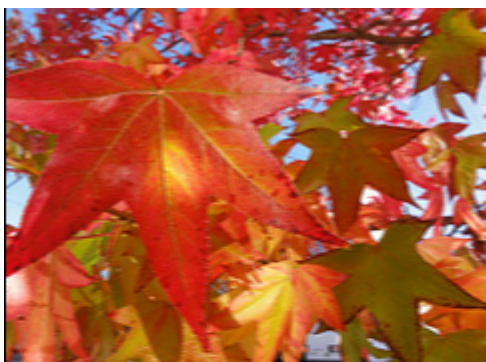
void main() {
    // compute 1 pixel in texture coordinates.
    vec2 onePixel = vec2(1.0, 1.0) / u_textureSize;

    // average the left, middle, and right pixels.
    gl_FragColor = (
        texture2D(u_image, v_texCoord) +
        texture2D(u_image, v_texCoord + vec2(onePixel.x, 0.0)) +
        texture2D(u_image, v_texCoord + vec2(-onePixel.x, 0.0))) / 3.0;
}
</script>
```

然后,我们需要通过 JavaScript 传递出纹理的大小。

```
...
var textureSizeLocation = gl.getUniformLocation(program, "u_textureSize");
...
// set the size of the image
gl.uniform2f(textureSizeLocation, image.width, image.height);
...
```

比较上述两个图片



现在，我们知道如何让使用像素卷积内核做一些常见的图像处理。这里，我们会使用 3x3 的内核。卷积内核就是一个 3x3 的矩阵，矩阵中的每个条目代表有多少像素渲染。然后，我们将这个结果除以内核的权重或 1.0。[这里是一个非常好的参考文章](#)。[这里有另一篇文章显示出一些实际代码，它是使用 C++ 写的](#)。

在我们的例子中我们要在着色器中做这样工作，这里是一个新的片段着色器。

```
<script id="2d-fragment-shader" type="x-shader/x-fragment">
precision mediump float;

// our texture
uniform sampler2D u_image;
uniform vec2 u_textureSize;
uniform float u_kernel[9];
uniform float u_kernelWeight;

// the texCoords passed in from the vertex shader.
varying vec2 v_texCoord;

void main() {
    vec2 onePixel = vec2(1.0, 1.0) / u_textureSize;
    vec4 colorSum =
    texture2D(u_image, v_texCoord + onePixel * vec2(-1, -1)) * u_kernel[0] +
    texture2D(u_image, v_texCoord + onePixel * vec2( 0, -1)) * u_kernel[1] +
    texture2D(u_image, v_texCoord + onePixel * vec2( 1, -1)) * u_kernel[2] +
    texture2D(u_image, v_texCoord + onePixel * vec2(-1,  0)) * u_kernel[3] +
    texture2D(u_image, v_texCoord + onePixel * vec2( 0,  0)) * u_kernel[4] +
    texture2D(u_image, v_texCoord + onePixel * vec2( 1,  0)) * u_kernel[5] +
    texture2D(u_image, v_texCoord + onePixel * vec2(-1,  1)) * u_kernel[6] +
    texture2D(u_image, v_texCoord + onePixel * vec2( 0,  1)) * u_kernel[7] +
```

```
texture2D(u_image, v_texCoord + onePixel * vec2( 1, 1)) * u_kernel[8] ;

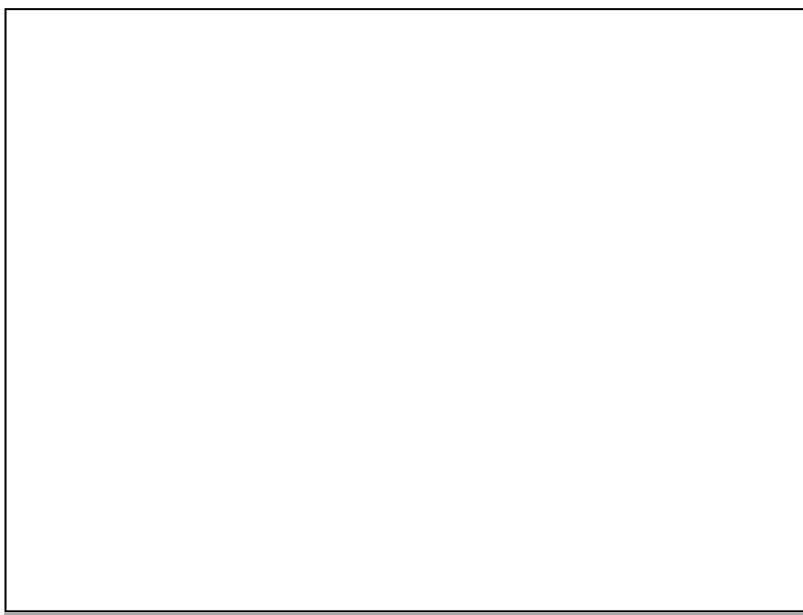
// Divide the sum by the weight but just use rgb
// we'll set alpha to 1.0
gl_FragColor = vec4((colorSum / u_kernelWeight).rgb, 1.0);
}
</script>
```

在 JavaScript 中，我们需要提供一个卷积内核和它的权重。

```
function computeKernelWeight(kernel) {
  var weight = kernel.reduce(function(prev, curr) {
    return prev + curr;
  });
  return weight <= 0 ? 1 : weight;
}

...
var kernelLocation = gl.getUniformLocation(program, "u_kernel[0]");
var kernelWeightLocation = gl.getUniformLocation(program, "u_kernelWeight");
...
var edgeDetectKernel = [
  -1, -1, -1,
  -1, 8, -1,
  -1, -1, -1
];
gl.uniform1fv(kernelLocation, edgeDetectKernel);
gl.uniform1f(kernelWeightLocation, computeKernelWeight(edgeDetectKernel));
...
```

我们在列表框内选择不同的内核。



我们希望通过这篇文章讲解，能够让你觉得使用 WebGL 做图像处理很简单。下面，我们将讲解[如何在一个图像上应用更多的效果](#)。

## WebGL 图像处理（续）

这篇文章是 [WebGL 图像处理](#) 内容扩展。

下一个关于图像处理的显著问题就是如何应用多重效果？读者当然可以尝试着写一下着色器。生成一个 UI 来让用户使用不同的着色器选择他们希望的效果。这通常是不太可能的，因为这个技术通常需要[实时的渲染效果](#)。

一种比较灵活的方式是使用两种或更多的纹理和渲染效果来交替渲染,每次应用一个效果，然后反复应用。

```
Original Image -> [Blur]-> Texture 1
Texture 1 -> [Sharpen] -> Texture 2
Texture 2 -> [Edge Detect] -> Texture 1
Texture 1 -> [Blur]-> Texture 2
Texture 2 -> [Normal] -> Canvas
```

要做到这一点，就需要创建帧缓存区。在 WebGL 和 OpenGL 中，帧缓存区实际上是一个非常不正式的名称。WebGL/OpenGL 中的帧缓存实际上仅仅是一些状态的集合，而不是真正的缓存。但是，每当一种纹理到达帧缓存，我们就会渲染出这种纹理。

首先让我们把[旧的纹理创建代码](#)写成一个函数。

```
function createAndSetupTexture(gl) {
  var texture = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, texture);

  // Set up texture so we can render any size image and so we are
  // working with pixels.
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

  return texture;
}

// Create a texture and put the image in it.
var originalImageTexture = createAndSetupTexture(gl);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
```

然后，我们使用这两个函数来生成两种问题，并且附在两个帧缓存中。

```
// create 2 textures and attach them to framebuffers.
var textures = [];
```

```

var framebuffers = [];
for (var ii = 0; ii < 2; ++ii) {
var texture = createAndSetupTexture(gl);
textures.push(texture);

// make the texture the same size as the image
gl.texImage2D(
gl.TEXTURE_2D, 0, gl.RGBA, image.width, image.height, 0,
gl.RGBA, gl.UNSIGNED_BYTE, null);

// Create a framebuffer
var fbo = gl.createFramebuffer();
framebuffers.push(fbo);
gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);

// Attach a texture to it.
gl.framebufferTexture2D(
gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture, 0);
}

```

现在，我们生成一些核的集合，然后存储到列表里来应用。

```

// Define several convolution kernels
var kernels = {
normal: [
0, 0, 0,
0, 1, 0,
0, 0, 0
],
gaussianBlur: [
0.045, 0.122, 0.045,
0.122, 0.332, 0.122,
0.045, 0.122, 0.045
],
unsharpen: [
-1, -1, -1,
-1, 9, -1,
-1, -1, -1
],
emboss: [
-2, -1, 0,
-1, 1, 1,
0, 1, 2
]
};

```

```
// List of effects to apply.
var effectsToApply = [
"gaussianBlur",
"emboss",
"gaussianBlur",
"unsharpen"
];
```

最后，我们应用每一个，然后交替渲染。

```
// start with the original image
gl.bindTexture(gl.TEXTURE_2D, originalImageTexture);

// don't y flip images while drawing to the textures
gl.uniform1f(flipYLocation, 1);

// loop through each effect we want to apply.
for (var ii = 0; ii < effectsToApply.length; ++ii) {
// Setup to draw into one of the framebuffers.
setFramebuffer(framebuffers[ii % 2], image.width, image.height);

drawWithKernel(effectsToApply[ii]);

// for the next draw, use the texture we just rendered to.
gl.bindTexture(gl.TEXTURE_2D, textures[ii % 2]);
}

// finally draw the result to the canvas.
gl.uniform1f(flipYLocation, -1); // need to y flip for canvas
setFramebuffer(null, canvas.width, canvas.height);
drawWithKernel("normal");

function setFramebuffer(fbo, width, height) {
// make this the framebuffer we are rendering to.
gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);

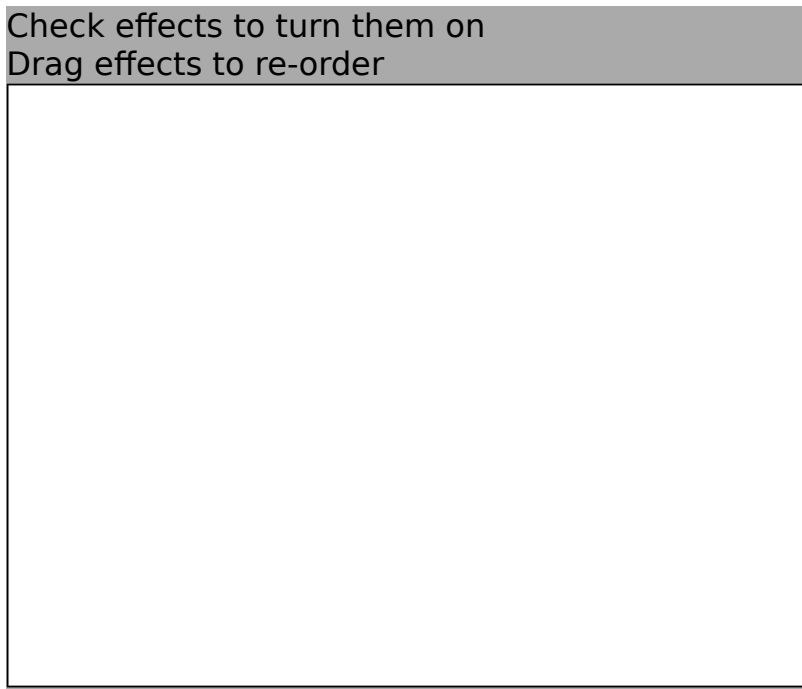
// Tell the shader the resolution of the framebuffer.
gl.uniform2f(resolutionLocation, width, height);

// Tell webgl the viewport setting needed for framebuffer.
gl.viewport(0, 0, width, height);
}

function drawWithKernel(name) {
// set the kernel
gl.uniform1fv(kernelLocation, kernels[name]);
```

```
// Draw the rectangle.
gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```

下面是更灵活的 UI 的可交互版本。勾选相应的效果即可检查效果。



以空值调用 `gl.bindFramebuffer` 即可告诉 WebGL 程序希望渲染到画板而不是帧缓存中的纹理。

WebGL 不得不将投影矩阵转换为像素。这是基于 `gl.viewport` 的设置。当我们初始化 WebGL 的时候，`gl.viewport` 的设置默认为画板的尺寸。因为，我们会将帧缓存渲染为不同的尺寸，所以画板需要设置合适的视图。

最后，在[原始例子](#)中，当需要渲染的时候，我们会翻转 Y 坐标。这是因为 WebGL 会以 0 来显示面板。0 表示是左侧底部的坐标，这不同于 2D 图像的顶部左侧的坐标。当渲染为帧缓存时就不需要了。这是因为帧缓存并不会显示出来。其部分是顶部还是底部是无关紧要的。所有重要的就是像素 0, 0 在帧缓存里就对应着 0。为了解决这一问题，我们可以通过是否在着色器中添加更多输入信息的方法来设置是否快读交替。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
...
uniform float u_flipY;
...

void main() {
...
gl_Position = vec4(clipSpace * vec2(1, u_flipY), 0, 1);
...
}
```



```
}  
</script>
```

当我们渲染的时候，就可以设置它。

```
var flipYLocation = gl.getUniformLocation(program, "u_flipY");  
...  
// don't flip  
gl.uniform1f(flipYLocation, 1);  
...  
// flip  
gl.uniform1f(flipYLocation, -1);
```

在这个简单的例子中，通过使用单个 GLSL 程序可以实现多个效果。

如果你想做完整的图像处理你可能需要许多 GLSL 程序。一个程序实现色相、饱和度和亮度调整。另一个实现亮度和对比度。一个实现反相，另一个用于调整水平。你可能需要更改代码以更新 GLSL 程序和更新特定程序的参数。我本来考虑写出这个例子，但这是一个练习，所以最好留给读者自己实现，因为多个 GLSL 项目中每一种方法都有自己的参数，可能意味着需要一些重大重构，这很可能导致成为意大利面条似的大混乱。

我希望从这里和前面的示例中可以看出 WebGL 似乎更平易近人，我希望从 2D 方面入手，以有助于使 WebGL 更容易理解。



3



2D 转换、旋转、伸缩、矩阵



## WebGL 2D 图像转换

---

在学习 3D 相关知识之前，请首先看看 2D 的知识。请保持耐心。这篇文章某些人看起来可能非常简单，但是我们将要讲解的知识是建立在前几篇的文章的基础之上。如果你没有阅读过，我建议你至少阅读第一章之后再回到这里继续学习。

Translation 指的是一些奇特的数学名称，它的基本意思是“移动”某物。它同样适用于将一个句子从英文“移动”成为日语这一说法，但是此处我们谈论的是几何中的移动。通过使用以 [the first post](#) 结尾的代码，你可以仅仅通过修改 `setRectangle` 距离右边的的值来使矩形移动。如下是一个基于我们[初始示例](#)的代码：

```
// First lets make some variables
// to hold the translation of the rectangle
var translation = [0, 0];

// then let's make a function to
// re-draw everything. We can call this
// function after we update the translation.

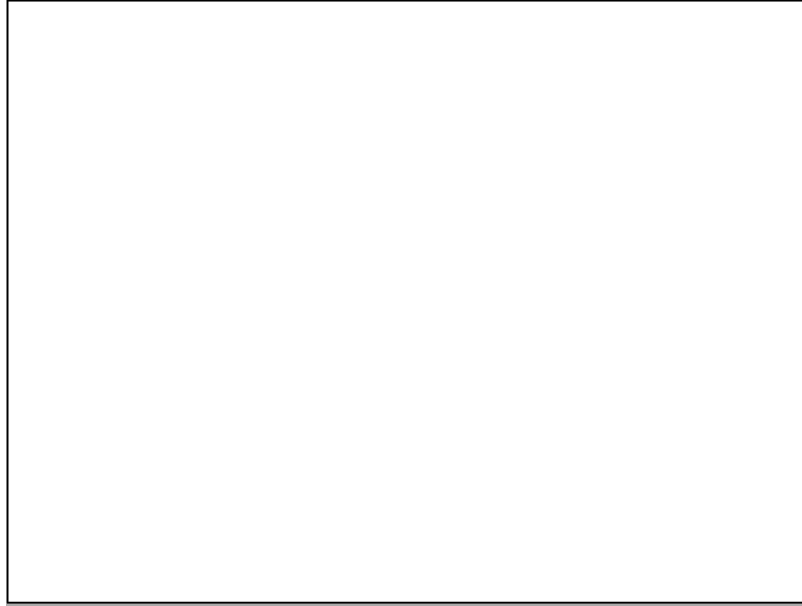
// Draw a the scene.
function drawScene() {
  // Clear the canvas.
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Setup a rectangle
  setRectangle(gl, translation[0], translation[1], width, height);

  // Draw the rectangle.
  gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```

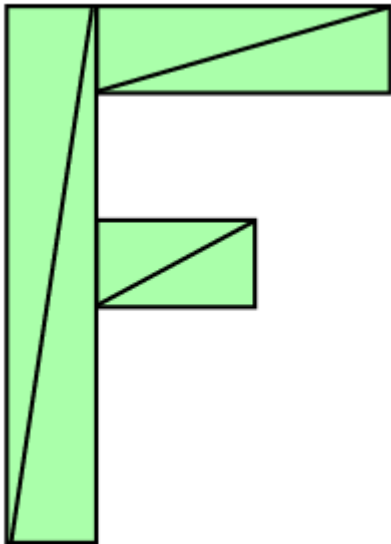
在上面的例子中，我在界面中放置了两个可滑动栏，你可以通过滑动按钮来修改 `translation[0]` 和 `translation[1]` 的值，而且在这个两个值发生修改时调用 `drawScene` 函数对界面进行更新。拖动滑动条对矩阵进行移动。

Drag sliders to translate.



到此处你已经觉得很不错。然而，现在假设我们想要利用相同的操作，但是处理更复杂的图形，那么该如何实现了。

假设我们想要画一个包含 6 个三角形的 'F' 形状，如下所示：



图片 3.1 F

如下是我们将要使用的改变 setRectangle 值的代码：

```
// Fill the buffer with the values that define a letter 'F'.  
function setGeometry(gl, x, y) {  
  var width = 100;  
  var height = 150;  
  var thickness = 30;
```

```

gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array([
    // left column
    x, y,
    x + thickness, y,
    x, y + height,
    x, y + height,
    x + thickness, y,
    x + thickness, y + height,

    // top rung
    x + thickness, y,
    x + width, y,
    x + thickness, y + thickness,
    x + thickness, y + thickness,
    x + width, y,
    x + width, y + thickness,

    // middle rung
    x + thickness, y + thickness * 2,
    x + width * 2 / 3, y + thickness * 2,
    x + thickness, y + thickness * 3,
    x + thickness, y + thickness * 3,
    x + width * 2 / 3, y + thickness * 2,
    x + width * 2 / 3, y + thickness * 3]),
  gl.STATIC_DRAW);
}

```

你会发现画出来的图形伸缩比例不是很好。如果你想画出有几百或者几千条线组成的几何图形，我们就需要编写一些相当复杂的代码。在上面的代码中，每次用 JavaScript 就需要更新所有的点。

有一种更简单的方式。仅仅只需要更新几何图形，接着修改渲染器部分。

如下是渲染器部分：

```

<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform vec2 u_translation;

void main() {
  // Add in the translation.
  vec2 position = a_position + u_translation;

```

```
// convert the rectangle from pixels to 0.0 to 1.0
vec2 zeroToOne = position / u_resolution;
...
```

接着我们将会稍微重构下代码。我们仅仅需要设置几何图形一次。

```
// Fill the buffer with the values that define a letter 'F'.
function setGeometry(gl) {
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      // left column
      0, 0,
      30, 0,
      0, 150,
      0, 150,
      30, 0,
      30, 150,

      // top rung
      30, 0,
      100, 0,
      30, 30,
      30, 30,
      100, 0,
      100, 30,

      // middle rung
      30, 60,
      67, 60,
      30, 90,
      30, 90,
      67, 60,
      67, 90]),
    gl.STATIC_DRAW);
}
```

在实现我们想要的移动之前需要更新下 `u_translation` 变量的值。

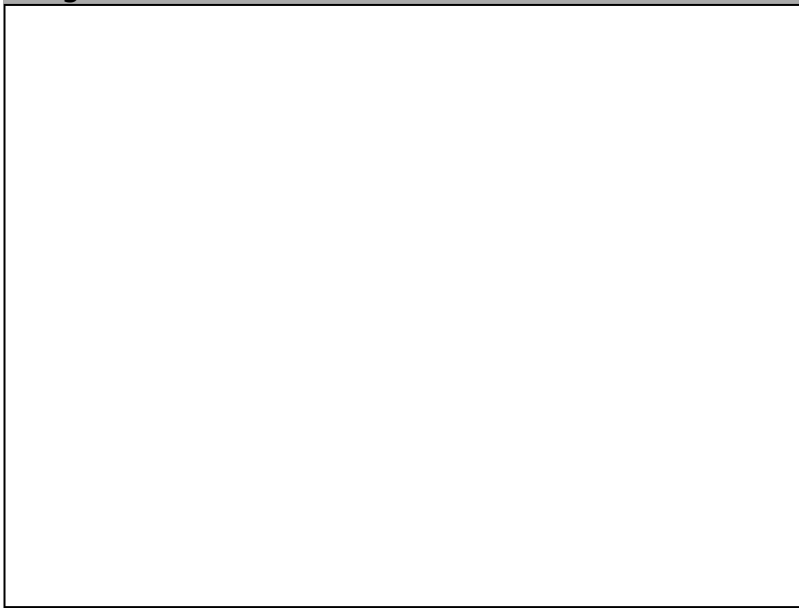
```
...
var translationLocation = gl.getUniformLocation(
  program, "u_translation");
...
// Set Geometry.
setGeometry(gl);
..
```

```
// Draw scene.  
function drawScene() {  
  // Clear the canvas.  
  gl.clear(gl.COLOR_BUFFER_BIT);  
  
  // Set the translation.  
  gl.uniform2fv(translationLocation, translation);  
  
  // Draw the rectangle.  
  gl.drawArrays(gl.TRIANGLES, 0, 18);  
}
```

注意 `setGeometry` 只是被调用一次。在 `drawScene` 中不需要。

如下是一个示例。同样，你可以通过拖动滑动条来更新图形的位置。

**Drag sliders to translate.**



现在，当我们绘制 WebGL 图像就包括要实现上面所有的事。我们所作的所有事指的是设置移动变量接着调用函数进行绘制。即使我们的几何图形包含成千上万的点，main 代码仍然是相同的。

## WebGL 2D 图像旋转

---

首先我要承认下我不是很清楚如何讲解这个让它看起来更容易理解，但是，不管怎么样，我想尽力尝试下。首先，我想介绍下什么叫做“单位圆”。你如果还记得高中数学中(不要睡着了!)一个圆又一个半径。半径指的是从圆心到圆的圆边的距离。单位圆指的是它的半径是 1.0。

如下是一个单位圆：

### Drag Circle

打开上面的链接之后，你可以拖动圆环上面的小圆，接着 X 和 Y 的值也会随之发生变化。这个左边值表示的是圆环上的点。在圆上的最高点处，Y 为 1 和 X 为 0。在最右的位置时 X 为 1 和 Y 为 0。

如果你还记得基础的三年级数学，把某个数乘以 1 以后结果仍然是该数。那么  $123 * 1 = 123$ 。相当基础对吧？那么半径为 1.0 的单位圆也是一种形式的 1。它是一种旋转的 1。因此你可以将这个单位圆与某物相乘，它执行的操作和乘以 1 类似，除了一些奇异的事情发生改变这种方式。

我们将从单位圆上得到任何点的 X 和 Y 值，接着将他们乘以[上一节示例](#)中的几何图形。

如下是更新渲染器：

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform vec2 u_translation;
uniform vec2 u_rotation;
```



```

void main() {
    // Rotate the position
    vec2 rotatedPosition = vec2(
        a_position.x * u_rotation.y + a_position.y * u_rotation.x,
        a_position.y * u_rotation.y - a_position.x * u_rotation.x);

    // Add in the translation.
    vec2 position = rotatedPosition + u_translation;
}

```

接着修改 JavaScript 代码，这样我们就可以传递上面的两个参数：

```

...
var rotationLocation = gl.getUniformLocation(program, "u_rotation");
...
var rotation = [0, 1];
..
// Draw the scene.
function drawScene() {
    // Clear the canvas.
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Set the translation.
    gl.uniform2fv(rotationLocation, translation);

    // Set the rotation.
    gl.uniform2fv(rotationLocation, rotation);

    // Draw the rectangle.
    gl.drawArrays(gl.TRIANGLES, 0, 18);
}

```

如下是代码运行的结果。拖动单位圆上的小环使图形进行旋转或者拖动滑动条使图形进行移动。

Drag sliders to translate.  
Drag circle to rotate.

为什么上面的代码能够起作用？首先，让我们看下数学公式：

```
rotatedX = a_position.x * u_rotation.y + a_position.y * u_rotation.x;
rotatedY = a_position.y * u_rotation.y - a_position.x * u_rotation.x;
```

假设你有一个矩形，并且你想旋转它。在你把它旋转到右上角 (3.0, 9.0) 这个位置之前。我们先在单位圆中选择一个从 12 点钟的位置顺时针偏移 30 度的点。

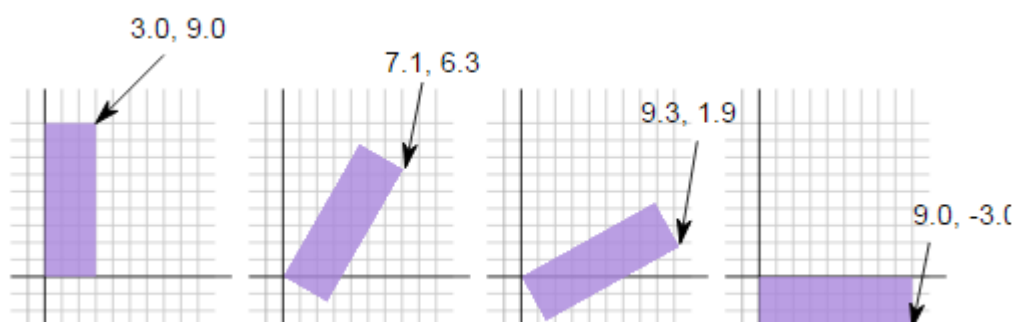
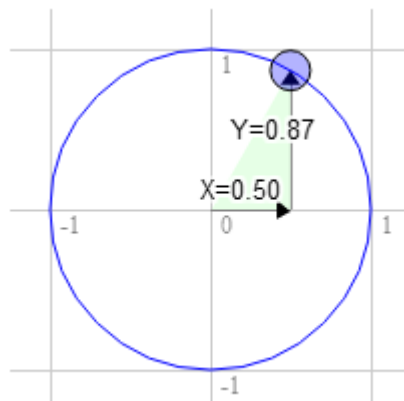
在圆上那个位置的点的坐标为 0.50 和 0.87：

```
3.0 * 0.87 + 9.0 * 0.50 = 7.1
9.0 * 0.87 - 3.0 * 0.50 = 6.3
```

那刚刚好是我们需要的位置：

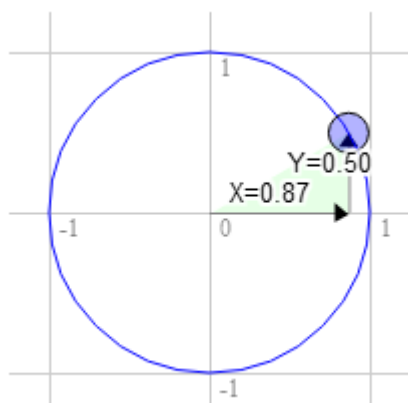
图片

3.2 30  
degree  
s clock  
wise fr  
om 12  
o'clock



图片 3.3 rotate rectangle

旋转 60 度和上面的操作一样：



图片 3.4 rotate 60 degrees

圆上面的位置的坐标是 0.87 和 0.50:

$$\begin{aligned} 3.0 * 0.50 + 9.0 * 0.87 &= 9.3 \\ 9.0 * 0.50 - 3.0 * 0.87 &= 1.9 \end{aligned}$$

你可以发现当我们顺时针向右旋转那个点时，X 的值变得更大而 Y 的值在变小。如果接着旋转超过 90 度，X 的值将再次变小而 Y 的值将变得更大。这个形式就能够达到旋转的目的。

圆环上的那些点还有另外一个名称。他们被称作为 sine 和 cosine。因此，对任意给定的角度，我们就只需查询它所对应的 sine 和 cosine 值：

```
function printSineAndCosineForAnyAngle(angleInDegrees) {
  var angleInRadians = angleInDegrees * Math.PI / 180;
  var s = Math.sin(angleInRadians);
  var c = Math.cos(angleInRadians);
  console.log("s = " + s + " c = " + c);
}
```

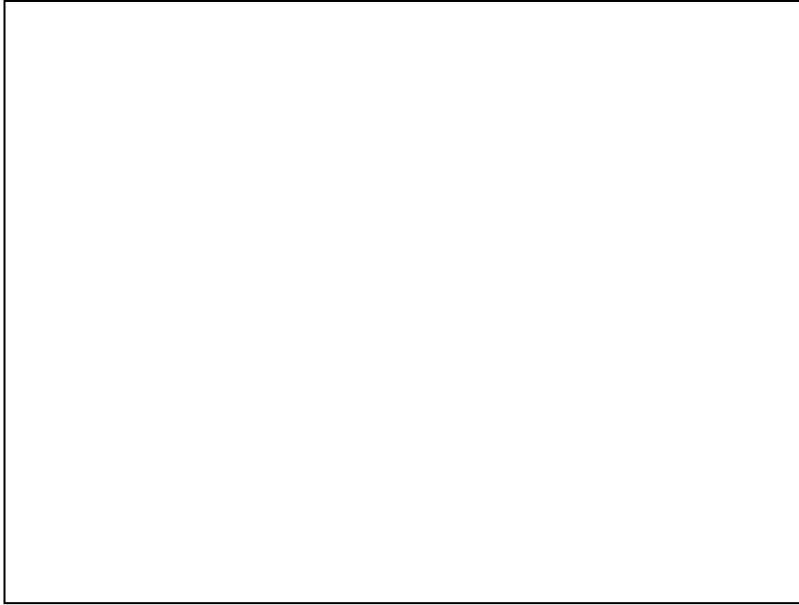
如果你把上面的代码复制粘贴到 JavaScript 控制台中，接着输入 `printSineAndCosineForAnyAngle(30)`，接着你会看到输出 `s = 0.49 c = 0.87`（注意：这个数字是近似值。）

如果你把上面的代码整合在一起的话，你就可以将你的几何体按照你想要的任何角度进行旋转。仅仅只需要将你需要旋转的角度值传给 sine 和 cosine 就可以了。

```
...
var angleInRadians = angleInDegrees * Math.PI / 180;
rotation[0] = Math.sin(angleInRadians);
rotation[1] = Math.cos(angleInRadians);
```

如下是设置一个角度旋转的版本。拖动滑动条旋转或者移动。

Drag sliders to translate & rotate.



## WebGL 2D 图像伸缩

图像伸缩和[转换](#)一样简单。我们只需对需要变换的点乘以我们想要的比例。如下是从[以前的代码](#)改变而来的。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform vec2 u_translation;
uniform vec2 u_rotation;
uniform vec2 u_scale;

void main() {
    // Scale the position
    vec2 scaledPosition = a_position * u_scale;

    // Rotate the position
    vec2 rotatedPosition = vec2(
        scaledPosition.x * u_rotation.y + scaledPosition.y * u_rotation.x,
        scaledPosition.y * u_rotation.y - scaledPosition.x * u_rotation.x);

    // Add in the translation.
    vec2 position = rotatedPosition + u_translation;
```

接着当我们需要绘图时添加必要的 JavaScript 代码来设置伸缩比例。

```
...
var scaleLocation = gl.getUniformLocation(program, "u_scale");
...
var scale = [1, 1];
...
// Draw the scene.
function drawScene() {
    // Clear the canvas.
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Set the translation.
    gl.uniform2fv(translationLocation, translation);

    // Set the rotation.
    gl.uniform2fv(rotationLocation, rotation);

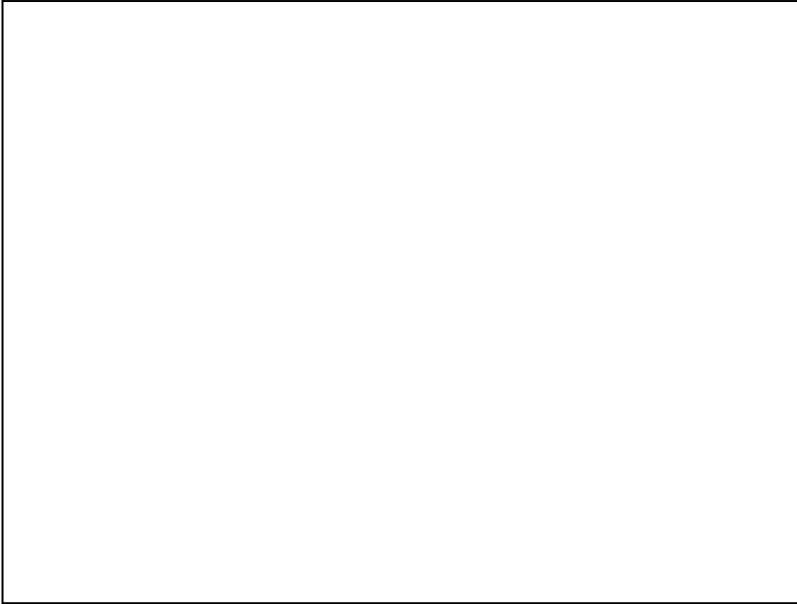
    // Set the scale.
```

```
gl.uniform2fv(scaleLocation, scale);

// Draw the rectangle.
gl.drawArrays(gl.TRIANGLES, 0, 18);
}
```

现在我们就可以通过拖动滑动条对图像进行伸缩变换。

Drag sliders to translate, rotate, and scale.

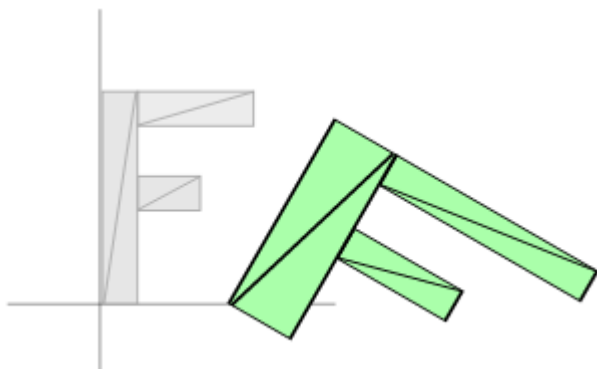


需要注意的一件事是如果设置伸缩比例为负值，那么几何图形就会发生翻转。

希望这相邻的三篇文章能够帮助你理解图形转换，旋转和伸缩。接下来我们将讲解拥有魔力的矩阵，它能够很容易的将这三种操作集合在一起，而且通常是更有用的形式。

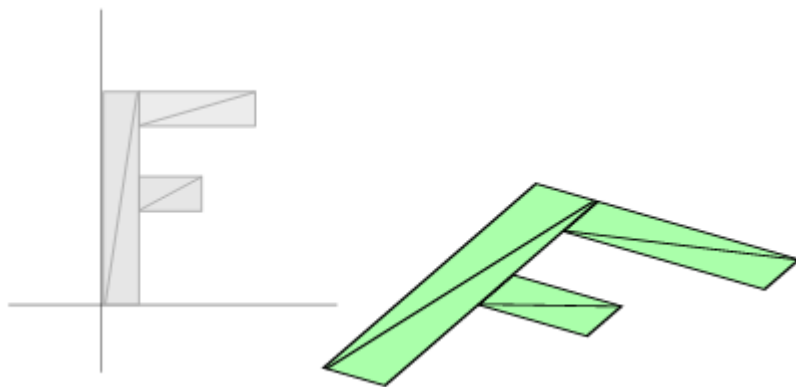
## WebGL 2D 矩阵

在前面三篇文章中我们讲解了如何平移几何图形，如何旋转几何图形，如何伸缩变换图形。平移，旋转和伸缩都被认为是一种变化类型。每一种变化都需要改变渲染器，而且他们依赖于操作的顺序。在[前面的例子](#)中我们进行了伸缩，旋转和平移操作。如果他们执行操作的顺序改变将会得到不同的结果。例如 XY 伸缩变换为 2, 1，旋转 30%，接着平移变换 100, 0。



图片 3.5 transformation1

如下是平移 100, 0，旋转 30%，接着伸缩变换 2, 1。



图片 3.6 transformation2

结果是完全不同的。更糟糕的是，如果我们需要得到的是第二个示例的效果，就必须编写一个不同的渲染器，按照我们想要的执行顺序进行平移，旋转和伸缩变换。

然而，有些比我聪明的人利用数学中的矩阵能够解决上面这个问题。对于 2d 图形，使用一个 3X3 的矩阵。3X3 的矩阵类似了 9 宫格。

1.0	2.0	3.0
4.0	5.0	6.0
7.0	8.0	9.0

图片 3.7 9 boxes

数学中的操作是与列相乘然后把结果加在一起。一个位置有两个值，用  $x$  和  $y$  表示。但是为了实现这个需要三个值，因为我们对第三个值设为 1。

在上面的例子中就变成了：

$$\begin{aligned}
 \text{newX} &= x * \begin{array}{|c|} \hline 1.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 4.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline 7.0 \\ \hline \end{array} \\
 \text{newY} &= x * \begin{array}{|c|} \hline 2.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 5.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline 8.0 \\ \hline \end{array} \\
 \text{extra} &= x * \begin{array}{|c|} \hline 3.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 6.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline 9.0 \\ \hline \end{array}
 \end{aligned}$$

图片 3.8 matrix transformation

对于上面的处理你也许会想“这样处理的原因在哪里”。假设要执行平移变换。我们将想要执行的平移的总量为  $t_x$  和  $t_y$ 。构造如下的矩阵：

1.0	0.0	0.0
0.0	1.0	0.0
$t_x$	$t_y$	1.0

图片 3.9 matrix

接着进行计算：

$$\begin{aligned}
 \text{newX} &= x * \begin{array}{|c|} \hline 1.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline t_x \\ \hline \end{array} \\
 \text{newY} &= x * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 1.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline t_y \\ \hline \end{array} \\
 \text{extra} &= x * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 &\quad y * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 &\quad 1 * \begin{array}{|c|} \hline 1.0 \\ \hline \end{array}
 \end{aligned}$$

图片 3.10 multiply matrix

如果你还记得代数学，就可以那些乘积结果为零的位置。乘以 1 的效果相当于什么都没做，那么将计算简化看看发生了什么：



$$\begin{array}{lcl}
 \text{newX} = x & \begin{array}{|c|} \hline \text{tx} \\ \hline \end{array} & + \\
 & \begin{array}{|c|} \hline \text{tx} \\ \hline \end{array} & \\
 \text{newY} = y & \begin{array}{|c|} \hline \text{ty} \\ \hline \end{array} & + \\
 & \begin{array}{|c|} \hline \text{ty} \\ \hline \end{array} & \\
 \text{extra} = & \begin{array}{|c|} \hline 1 \\ \hline \end{array} & 
 \end{array}$$

图片 3.11 simplify result

或者更简洁的方式：

```
newX = x + tx;
newY = y + ty;
```

extra 变量我们并不用在意。这个处理和我们在平移中编写的代码惊奇的相似。

同样地，让我们看看旋转。正如在旋转那篇中指出当我们想要进行旋转的时候，我们只需要角度的 sine 和 cosine 值。

```
s = Math.sin(angleToRotateInRadians);
c = Math.cos(angleToRotateInRadians);
```

构造如下的矩阵：

c	-s	0.0
s	c	0.0
0.0	0.0	1.0

图片 3.12 rotate\_matrix

执行上面的矩阵操作：

$$\begin{array}{lcl}
 \text{newX} = x * \begin{array}{|c|} \hline c \\ \hline \end{array} + & \text{newY} = x * \begin{array}{|c|} \hline -s \\ \hline \end{array} + & \text{extra} = x * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 y * \begin{array}{|c|} \hline s \\ \hline \end{array} + & y * \begin{array}{|c|} \hline c \\ \hline \end{array} + & y * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} + \\
 1 * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} & 1 * \begin{array}{|c|} \hline 0.0 \\ \hline \end{array} & 1 * \begin{array}{|c|} \hline 1.0 \\ \hline \end{array}
 \end{array}$$

图片 3.13 rotate\_apply\_matrix

将得到 0 和 1 结果部分用黑色块表示了。

$$\begin{array}{lcl}
 \text{newX} = x * \begin{array}{|c|} \hline c \\ \hline \end{array} + & \text{newY} = x * \begin{array}{|c|} \hline -s \\ \hline \end{array} + & \text{extra} = \\
 y * \begin{array}{|c|} \hline s \\ \hline \end{array} + & y * \begin{array}{|c|} \hline c \\ \hline \end{array} + & \begin{array}{|c|} \hline 1 \\ \hline \end{array} \\
 \begin{array}{|c|} \hline \text{tx} \\ \hline \end{array} & \begin{array}{|c|} \hline \text{ty} \\ \hline \end{array} & 
 \end{array}$$

图片 3.14 rotate\_matrix\_result

同样可以简化计算：

```
newX = x * c + y * s;
newY = x * -s + y * c;
```

上面处理的结果刚好和[旋转例子](#)效果一样。

最后是伸缩变换。称两个伸缩变换因子为  $sx$  和  $sy$ 。

构造如下的矩阵：

$sx$	0.0	0.0
0.0	$sy$	0.0
0.0	0.0	1.0

图片 3.15 scale\_matrix

进行矩阵操作会得到如下：

$$\begin{aligned} newX = & x * \begin{bmatrix} sx \\ 0.0 \\ 0.0 \end{bmatrix} + \\ & y * \begin{bmatrix} 0.0 \\ sy \\ 0.0 \end{bmatrix} + \\ & 1 * \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} newY = & x * \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + \\ & y * \begin{bmatrix} 0.0 \\ sy \\ 0.0 \end{bmatrix} + \\ & 1 * \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} extra = & x * \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + \\ & y * \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + \\ & 1 * \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix} \end{aligned}$$

图片 3.16 scale apply matrix

实际需要计算：

$$newX = x * \begin{bmatrix} sx \\ 0.0 \\ 0.0 \end{bmatrix} +$$

$$newY = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + y * \begin{bmatrix} 0.0 \\ sy \\ 0.0 \end{bmatrix} +$$

$$extra = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix} + 1 * \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix}$$

图片 3.17 scale matrix result

简化为：

```
newX = x * sx;
newY = y * sy;
```

和我们以前讲解的[伸缩示例](#)是一样的。

到了这里，我坚信你仍然在思考，这样处理之后了？有什么意义。看起来好象它只是做了和我们以前一样的事。

接下来就是魔幻的地方。已经被证明了我们可以将多个矩阵乘在一起，接着一次执行完所有的变换。假设有函数 `matrixMultiply`，它带两个矩阵做参数，将他们俩相乘，返回乘积结果。

为了让上面的做法更清楚，于是编写如下的函数构建一个用来平移，旋转和伸缩的矩阵：

```
function makeTranslation(tx, ty) {
  return [
    1, 0, 0,
    0, 1, 0,
    tx, ty, 1
  ];
}

function makeRotation(angleInRadians) {
  var c = Math.cos(angleInRadians);
  var s = Math.sin(angleInRadians);
  return [
    c, -s, 0,
    s, c, 0,
    0, 0, 1
  ];
}

function makeScale(sx, sy) {
  return [
    sx, 0, 0,
    0, sy, 0,
    0, 0, 1
  ];
}
```

接下来，修改渲染器。以往的渲染器是如下的形式：

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform vec2 u_translation;
uniform vec2 u_rotation;
uniform vec2 u_scale;

void main() {
  // Scale the positon
```

```

vec2 scaledPosition = a_position * u_scale;

// Rotate the position
vec2 rotatedPosition = vec2(
scaledPosition.x * u_rotation.y + scaledPosition.y * u_rotation.x,
scaledPosition.y * u_rotation.y - scaledPosition.x * u_rotation.x);

// Add in the translation.
vec2 position = rotatedPosition + u_translation;

...

```

新的渲染器将会变得更简单：

```

<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform vec2 u_resolution;
uniform mat3 u_matrix;

void main() {
// Multiply the position by the matrix.
vec2 position = (u_matrix * vec3(a_position, 1)).xy;

...

```

如下是我们使用它的方式：

```

// Draw the scene.
function drawScene() {
// Clear the canvas.
gl.clear(gl.COLOR_BUFFER_BIT);

// Compute the matrices
var translationMatrix = makeTranslation(translation[0], translation[1]);
var rotationMatrix = makeRotation(angleInRadians);
var scaleMatrix = makeScale(scale[0], scale[1]);

// Multiply the matrices.
var matrix = matrixMultiply(scaleMatrix, rotationMatrix);
matrix = matrixMultiply(matrix, translationMatrix);

// Set the matrix.
gl.uniformMatrix3fv(matrixLocation, false, matrix);

// Draw the rectangle.
gl.drawArrays(gl.TRIANGLES, 0, 18);
}

```

如下是使用新的代码的示例。平移，旋转和伸缩滑动条是一样的。但是他们在渲染器上应用的更简单。

Drag sliders to translate, rotate, and scale.

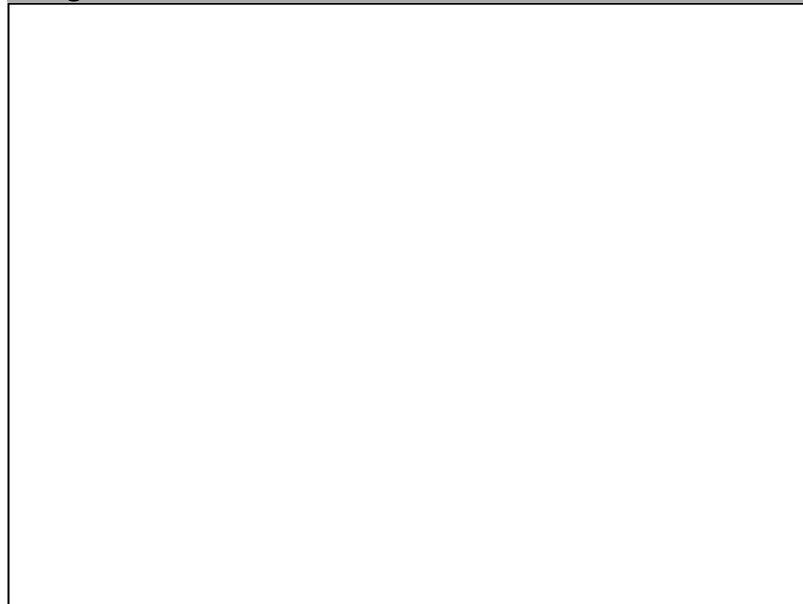


此时，你仍然会问，之后了？这个看起来并没有方便多少。然而，此时如果你想改变执行的顺序，就不再需要编写一个新的渲染器了。我们仅仅只需要改变数序公式。

```
...  
// Multiply the matrices.  
var matrix = matrixMultiply(translationMatrix, rotationMatrix);  
matrix = matrixMultiply(matrix, scaleMatrix);  
...
```

如下是新版本:

Drag sliders to translate, rotate, and scale.



能够按照这种方式执行矩阵操作是特别重要的，特别是对于层级动画的实现比如身体上手臂的，在一个星球上看月球同时在围绕着太阳旋转，或者数上的树枝等都是很重要的。举一个简单的层级动画例子，现在想要绘制 5 次 ‘F’，但是每次绘制是从上一个 ‘F’ 开始的。

```
// Draw the scene.
function drawScene() {
    // Clear the canvas.
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Compute the matrices
    var translationMatrix = makeTranslation(translation[0], translation[1]);
    var rotationMatrix = makeRotation(angleInRadians);
    var scaleMatrix = makeScale(scale[0], scale[1]);

    // Starting Matrix.
    var matrix = makeIdentity();

    for (var i = 0; i < 5; ++i) {
        // Multiply the matrices.
        matrix = matrixMultiply(matrix, scaleMatrix);
        matrix = matrixMultiply(matrix, rotationMatrix);
        matrix = matrixMultiply(matrix, translationMatrix);

        // Set the matrix.
        gl.uniformMatrix3fv(matrixLocation, false, matrix);

        // Draw the geometry.
        gl.drawArrays(gl.TRIANGLES, 0, 18);
    }
}
```

为了实现这个，我们要编写自己的函数 `makeIdentity`，这个函数返回单位矩阵。单位矩阵实际上表示的类似于 1.0 的矩阵，如果一个矩阵乘以单位矩阵，那么得到的还是原先那个矩阵。就如：

$$X * 1 = X$$

同样：

$$\text{matrixX} * \text{identity} = \text{matrixX}$$

如下是构造单位矩阵的代码：

```
function makeIdentity() {
    return [
        1, 0, 0,
```

```

    0, 1, 0,
    0, 0, 1
  ];
}
```

如下是 5 个 F：

Drag sliders to translate, rotate, and scale.



再来一个示例，在前面示例中，‘F’ 旋转总是绕左上角。这是因为我们使用的数学方法总是围着源点旋转，并且 ‘F’ 的左上角就是原点，(0, 0)。

但是现在，因为我们能够使用矩阵，那么就可以选择变化的顺序，可以在执行其他的变换之前先移动原点。

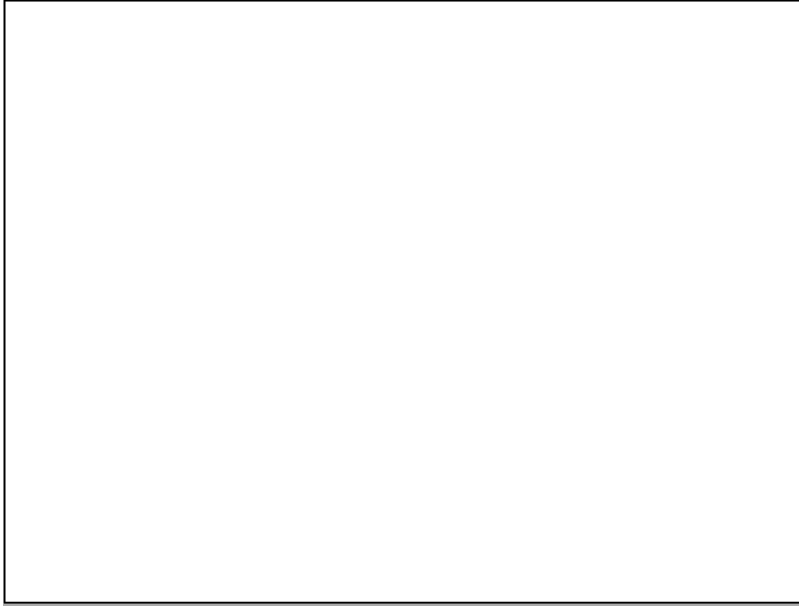
```

// make a matrix that will move the origin of the 'F' to its center.
var moveOriginMatrix = makeTranslation(-50, -75);
...

// Multiply the matrices.
var matrix = matrixMultiply(moveOriginMatrix, scaleMatrix);
matrix = matrixMultiply(matrix, rotationMatrix);
matrix = matrixMultiply(matrix, translationMatrix);
```

如下所示，注意 F 可以围着中心点进行旋转和伸缩。

### Drag sliders to translate, rotate, and scale.



使用如上的方法，你可以围着任何点进行旋转或者伸缩。现在你就明白了 Photoshop 或者 Flash 中实现绕某点旋转的原理。

让我们学习更深入点。如果你回到本系列的第一篇文章 [WebGL 基本原理](#)，你也许还记得我们编写的渲染器的代码中将像素转换成投影空间，如下所示：

```
...
// convert the rectangle from pixels to 0.0 to 1.0
vec2 zeroToOne = position / u_resolution;

// convert from 0->1 to 0->2
vec2 zeroToTwo = zeroToOne * 2.0;

// convert from 0->2 to -1->+1 (clipSpace)
vec2 clipSpace = zeroToTwo - 1.0;

gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
```

如果你现在反过来看下每一步，第一步，“将像素转换成 0.0 变成 1.0”，其实是一个伸缩操作。第二步同样是伸缩变换。接下来是平移变换，并且 Y 的伸缩因子是 -1。我们可以通过将该矩阵传给渲染器实现上面的所有操作。可以构造二维伸缩矩阵，其中一个伸缩因子设置为 1.0/分辨率，另外一个伸缩因子设置为 2.0，第三个使用 -1.0，-1.0 来进行移动，并且第四个设置伸缩因子 Y 为 -1，接着将他们乘在一起，然而，因为数学是很容易的，我们仅仅只需编写一个函数，能够直接将给定的分辨率转换成投影矩阵。

```
function make2DProjection(width, height) {
  // Note: This matrix flips the Y axis so that 0 is at the top.
  return [
    2 / width, 0, 0,
```



```

    0, -2 / height, 0,
    -1, 1, 1
  ];
}

```

现在我们能进一步简化渲染器。如下是完整的顶点渲染器。

```

<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform mat3 u_matrix;

void main() {
  // Multiply the position by the matrix.
  gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);
}
</script>

```

在 JavaScript 中我们需要与投影矩阵相乘。

```

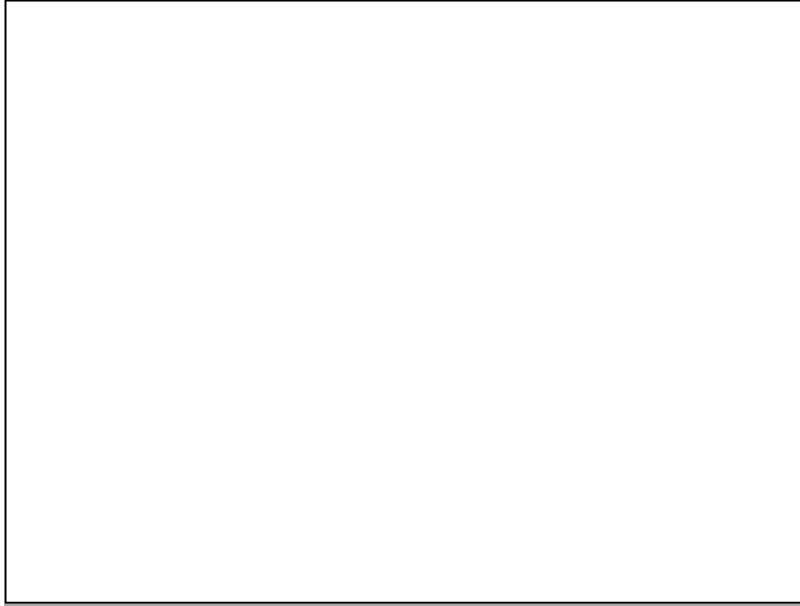
// Draw the scene.
function drawScene() {
  ...
  // Compute the matrices
  var projectionMatrix = make2DProjection(
    canvas.clientWidth, canvas.clientHeight);
  ...

  // Multiply the matrices.
  var matrix = matrixMultiply(scaleMatrix, rotationMatrix);
  matrix = matrixMultiply(matrix, translationMatrix);
  matrix = matrixMultiply(matrix, projectionMatrix);
  ...
}

```

我们也移出了设置分辨率的代码。最后一步，通过使用数学矩阵就将原先需要 6-7 步操作复杂的渲染器变成仅仅只需要 1 步操作的更简单的渲染器。

Drag sliders to translate, rotate, and scale.



希望这篇文章能够让你理解矩阵数学。接下来会讲解 [3D](#) 空间的知识。在 3D 中矩阵数学遵循同样的规律和使用方式。从 2D 开始讲解是希望让知识简单易懂。



T



3D



## WebGL 正交 3D

---

在上一篇文章中，我们就学习过二维矩阵是如何进行工作的。我们谈到的平移，旋转，缩放，甚至像素到剪辑空间的投影都可以通过一个矩阵和一些神奇的矩阵数学来完成。做三维只是一个从那里向前的一小步。

在我们以前的二维的例子中，我们有二维点  $(x, y)$ ，我们乘以一个  $3 \times 3$  的矩阵。做三维我们需要三维点  $(x, y, z)$  和一个  $4 \times 4$  矩阵。

让我们看最后一个例子，把它改为三维，我们将再次使用一个 F，但这一次的三维 'F'。

我们需要做的第一件事就是改变顶点着色来处理三维，这里是旧的着色。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;

uniform mat3 u_matrix;

void main() {
    // Multiply the position by the matrix.
    gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);
}
</script>
```

下面是新的

```
<script id="3d-vertex-shader" type="x-shader/x-vertex">
attribute vec4 a_position;

uniform mat4 u_matrix;

void main() {
    // Multiply the position by the matrix.
    gl_Position = u_matrix * a_position;
}
</script>
```

它甚至更简单！

然后我们需要提供三维数据。

...

```
gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false, 0, 0);
```

```

...

// Fill the buffer with the values that define a letter 'F'.
function setGeometry(gl) {
  gl.bufferData(
    gl.ARRAY_BUFFER,
    new Float32Array([
      // left column
      0, 0, 0,
      30, 0, 0,
      0, 150, 0,
      0, 150, 0,
      30, 0, 0,
      30, 150, 0,

      // top rung
      30, 0, 0,
      100, 0, 0,
      30, 30, 0,
      30, 30, 0,
      100, 0, 0,
      100, 30, 0,

      // middle rung
      30, 60, 0,
      67, 60, 0,
      30, 90, 0,
      30, 90, 0,
      67, 60, 0,
      67, 90, 0]),
    gl.STATIC_DRAW);
}

```

接下来，我们需要把所有的矩阵函数从二维变到三维

这是 `makeTranslation`，`makeRotation` 和 `makeScale` 的二维（前面的）版本

```

function makeTranslation(tx, ty) {
  return [
    1, 0, 0,
    0, 1, 0,
    tx, ty, 1
  ];
}

```

```
function makeRotation(angleInRadians) {
  var c = Math.cos(angleInRadians);
  var s = Math.sin(angleInRadians);
  return [
    c, -s, 0,
    s, c, 0,
    0, 0, 1
  ];
}

function makeScale(sx, sy) {
  return [
    sx, 0, 0,
    0, sy, 0,
    0, 0, 1
  ];
}
```

这是更新的三维版本。

```
function makeTranslation(tx, ty, tz) {
  return [
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    tx, ty, tz, 1
  ];
}

function makeXRotation(angleInRadians) {
  var c = Math.cos(angleInRadians);
  var s = Math.sin(angleInRadians);

  return [
    1, 0, 0, 0,
    0, c, s, 0,
    0, -s, c, 0,
    0, 0, 0, 1
  ];
};

function makeYRotation(angleInRadians) {
  var c = Math.cos(angleInRadians);
  var s = Math.sin(angleInRadians);

  return [
```

```

    c, 0, -s, 0,
    0, 1, 0, 0,
    s, 0, c, 0,
    0, 0, 0, 1
  ];
};

function makeZRotation(angleInRadians) {
  var c = Math.cos(angleInRadians);
  var s = Math.sin(angleInRadians);
  return [
    c, s, 0, 0,
    -s, c, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
  ];
}

function makeScale(sx, sy, sz) {
  return [
    sx, 0, 0, 0,
    0, sy, 0, 0,
    0, 0, sz, 0,
    0, 0, 0, 1,
  ];
}

```

注意，我们现在有3个旋转函数。在二维中我们只需要一个旋转函数，因为我们只需要绕 Z 轴旋转。现在虽然做三维我们也希望能够绕 X 轴和 Y 轴旋转。你可以从中看出，它们都非常相似。如果我们让它们工作，你会看到它们像以前一样简化

Z 旋转

<

p align="center"> newX = x \* c + y \* s;

<

p align="center"> newY = x \* -s + y \* c;

Y 旋转

<

p align="center"> newX = x \* c + z \* s;

&lt;

```
p align="center"> newZ = x * -s + z * c;
```

X 旋转

&lt;

```
p align="center"> newY = y * c + z * s;
```

&lt;

```
p align="center"> newZ = y * -s + z * c;
```

它给你这些旋转。

我们还需要更新投影函数。这是旧的

```
function make2DProjection(width, height) {
  // Note: This matrix flips the Y axis so 0 is at the top.
  return [
    2 / width, 0, 0,
    0, -2 / height, 0,
    -1, 1, 1
  ];
}
```

它从像素转换到剪辑空间。作为我们扩展到三维的第一次尝试，让我们试一下

```
function make2DProjection(width, height, depth) {
  // Note: This matrix flips the Y axis so 0 is at the top.
  return [
    2 / width, 0, 0, 0,
    0, -2 / height, 0, 0,
    0, 0, 2 / depth, 0,
    -1, 1, 0, 1,
  ];
}
```

就像我们需要把 X 和 Y 从像素转换到剪辑空间，对于 Z 我们需要做同样的事情。在这种情况下，我们制作 Z 空间像素单元。我会把一些类似 *width* 的值传入 *depth*，所以我们的空间宽度为 0 到宽度像素，高为 0 到高度像素，但深度是  $-\text{depth} / 2$  到  $+\text{depth} / 2$ 。

最后，我们需要更新计算矩阵的代码。



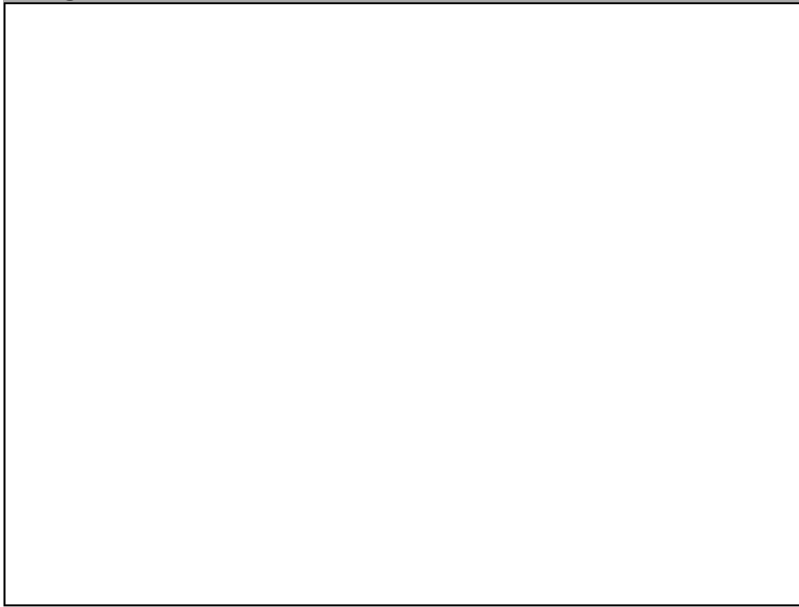
```
// Compute the matrices
var projectionMatrix =
    make2DProjection(canvas.clientWidth, canvas.clientHeight, 400);
var translationMatrix =
    makeTranslation(translation[0], translation[1], translation[2]);
var rotationXMatrix = makeXRotation(rotation[0]);
var rotationYMatrix = makeYRotation(rotation[1]);
var rotationZMatrix = makeZRotation(rotation[2]);
var scaleMatrix = makeScale(scale[0], scale[1], scale[2]);

// Multiply the matrices.
var matrix = matrixMultiply(scaleMatrix, rotationZMatrix);
matrix = matrixMultiply(matrix, rotationYMatrix);
matrix = matrixMultiply(matrix, rotationXMatrix);
matrix = matrixMultiply(matrix, translationMatrix);
matrix = matrixMultiply(matrix, projectionMatrix);

// Set the matrix.
gl.uniformMatrix4fv(matrixLocation, false, matrix);
```

下面是例子。

**Drag sliders to translate, rotate, and scale.**



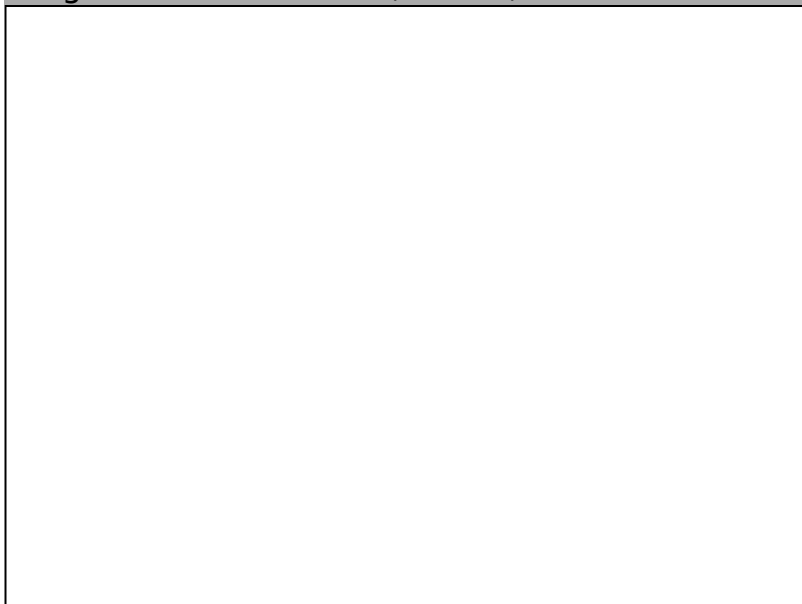
我们的第一个问题是几何体是一个平面的 F，这使得很难看到任何 3D。为了解决这个问题让我们将几何体扩展到三维。我们目前的 F 由 3 个矩形组成，每个有 2 个三角形。为使其成为三维将需要 16 个矩形。太多了就不列举在这里了。16 个矩形，每个有 2 个三角形，每个三角形有 3 个顶点，一共是 96 个顶点。如果你想看到所有的视图来源的样本。

我们必须画更多的顶点，因此

```
// Draw the geometry.
gl.drawArrays(gl.TRIANGLES, 0, 16 * 6);
```

这是这个版本

Drag sliders to translate, rotate, and scale.



移动滚动条，很难说它是 3D 的。让我们尝试一种不同的颜色着色每个矩形。要做到这一点，我们将为我们的顶点着色添加另一个属性和 varying 来把它从顶点着色传到到片段着色。

这里是新的顶点着色

```
<script id="3d-vertex-shader" type="x-shader/x-vertex">
attribute vec4 a_position;
attribute vec4 a_color;

uniform mat4 u_matrix;

varying vec4 v_color;

void main() {
    // Multiply the position by the matrix.
    gl_Position = u_matrix * a_position;

    // Pass the color to the fragment shader.
    v_color = a_color;
}
</script>
```

我们需要在片段着色器中使用颜色

```

<script id="3d-vertex-shader" type="x-shader/x-fragment">
precision mediump float;

// Passed in from the vertex shader.
varying vec4 v_color;

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

```

我们需要查找提供颜色的 location，然后设置另一个缓冲和属性给它的颜色。

```

...
var colorLocation = gl.getAttribLocation(program, "a_color");

...
// Create a buffer for colors.
var buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.enableVertexAttribArray(colorLocation);

// We'll supply RGB as bytes.
gl.vertexAttribPointer(colorLocation, 3, gl.UNSIGNED_BYTE, true, 0, 0);

// Set Colors.
setColors(gl);

...
// Fill the buffer with colors for the 'F'.

function setColors(gl) {
    gl.bufferData(
        gl.ARRAY_BUFFER,
        new Uint8Array([
            // left column front
            200, 70, 120,
            200, 70, 120,
            200, 70, 120,
            200, 70, 120,
            200, 70, 120,
            200, 70, 120,
            200, 70, 120,

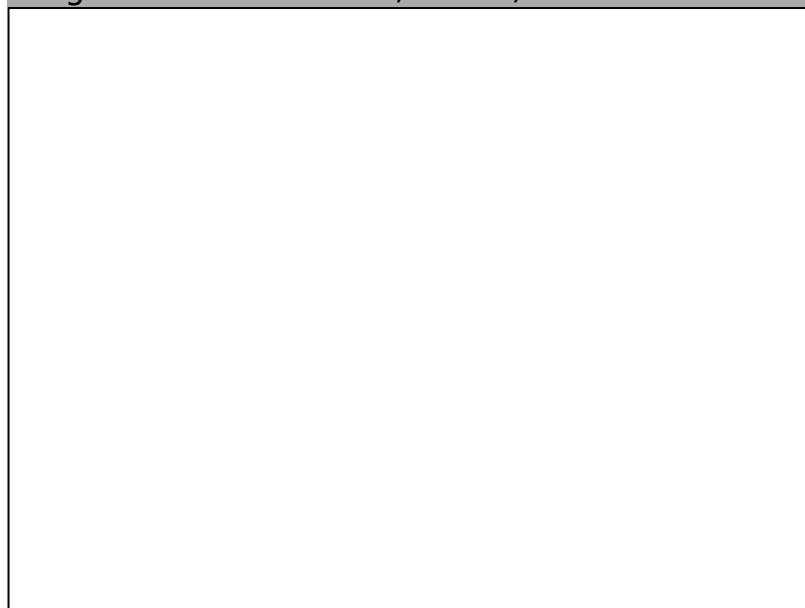
            // top rung front
            200, 70, 120,
            200, 70, 120,

```

```
...  
...  
gl.STATIC_DRAW);  
}
```

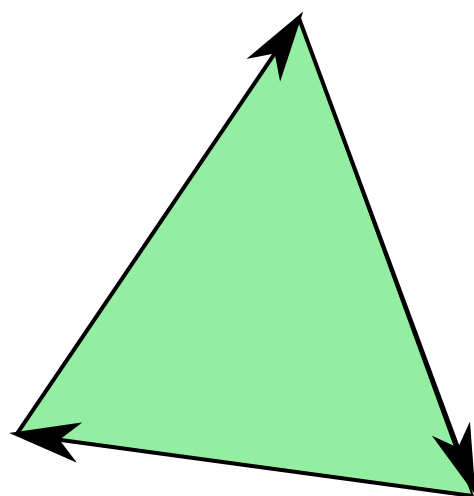
现在我们得到这个。

Drag sliders to translate, rotate, and scale.

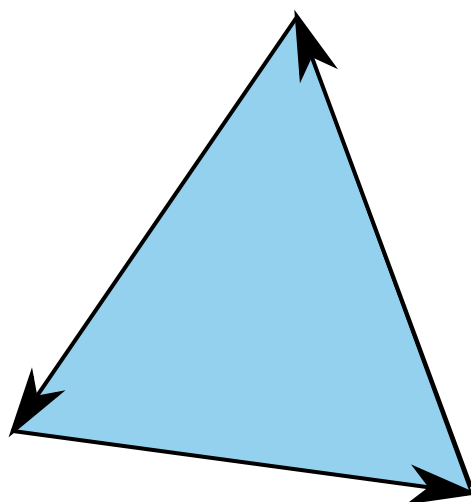


哦，那是什么？好了，它把三维 ‘F’ 的各个部分，前面，后面，边等等按它们出现在我们的几何体里的顺序画。这并不能给我们带来预期的结果，因为有时在后面的那些要在前面的那些之后画。

在 WebGL 中的三角形有前向和后向的概念。一个正向的三角形的顶点以顺时针方向画。后向的三角形的顶点按逆时针方向画。



Clockwise  
Front Facing  
Triangle



Counter Clockwise  
Back Facing  
Triangle

WebGL也只有向前或向后画三角形的能力。我们可以使用这个功能

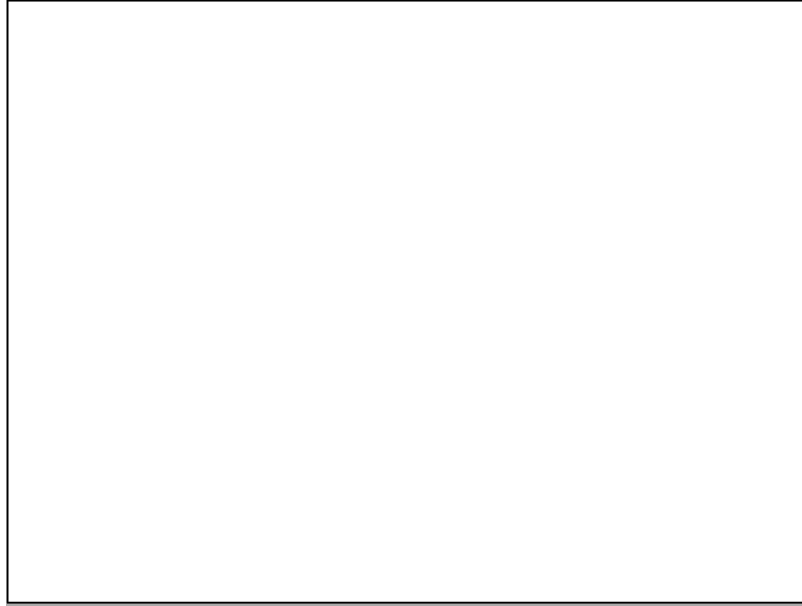
```
gl.enable(gl.CULL_FACE);
```

我们只做了一次，就在我们程序的开始。这个功能开启后，WebGL 的默认“culling”后向三角形。在这里“culling”是“not drawing”的意思。

注意到就 WebGL 而言，一个三角形是顺时针或逆时针取决于剪辑空间中该三角形的顶点。换句话说，WebGL 通过你在顶点着色器的顶点上应用的数学函数辨别三角形是前面或后面。这意味着例如在 X 轴按 -1 度量的顺时针三角形成为一个逆时针或顺时针三角形三角形旋转180度成为一个逆时针三角形。因为我们使 CULL\_FACE 失去作用，我们就能看到顺时针（前）和逆时针（回）三角形。现在我们打开它，任何时候一个前置三角形因为缩放或旋转或是什么原因左右翻转，WebGL就不会画。这是一件好事，因为当你在三维空间里的时候，你通常想要的是正面面对你的那个三角形。

CULL\_FACE 打开后，这是我们得到的

Drag sliders to translate, rotate, and scale.



嘿！所有的三角形都去哪儿了？事实证明，它们中的许多都面朝着错误的方向。旋转它，当你看向另一边是你会看到他们出现了。幸运的是这很容易解决。我们只看哪一个是朝后的，并且交换2个顶点。例如，如果一个朝后的三角形是

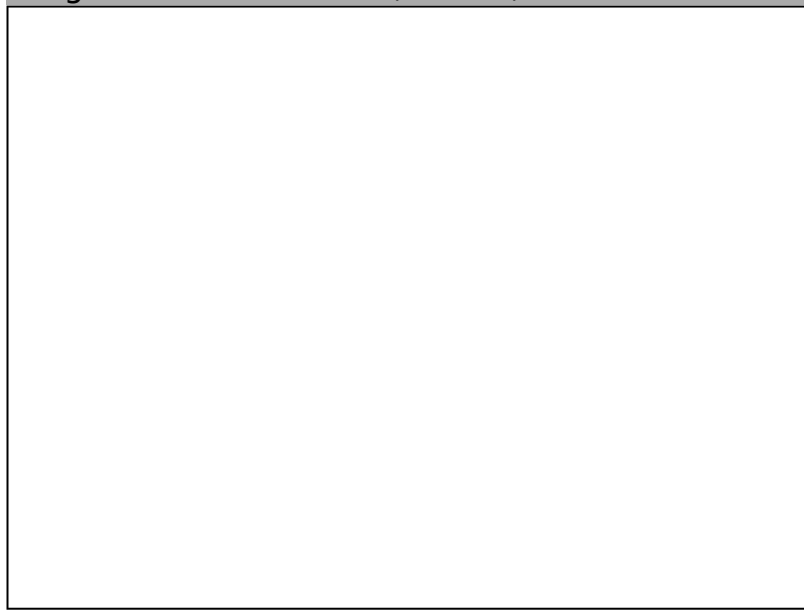
```
1, 2, 3,
40, 50, 60,
700, 800, 900,
```

我们只是翻转最后两个顶点以使它朝前。

```
1, 2, 3,
700, 800, 900,
40, 50, 60,
```

这样处理所有的后向三角形让我们得到

Drag sliders to translate, rotate, and scale.



这更接近，但仍然有一个问题。甚至所有三角形面对在正确的方向，后向三角形被剔除，我们还有应该在后面的三角形被画在前面三角形应该被画的地方。

进入深度缓冲区。

深度缓冲，有时被称为一个 Z-缓冲，是一个 *depth* 像素的矩形区域，每个彩色像素的深度像素用于形成图像。随着 WebGL 绘制每个彩色像素它还可以画一个深度像素。它做这些都基于我们从顶点着色器返回的 Z 的值。就像我们不得不转换剪辑空间对 X 和 Y，我们在剪辑空间也对 Z 做同样处理或（-1 至 +1）。该值，然后转换成一个深度空间值（0 至 1）。在 WebGL 绘制一个彩色像素之前它会检查相应的深度像素。如果要画的像素深度值大于相应的深度像素 WebGL 就不画新的彩色像素。否则，它从你的片段着色器绘制颜色的新的彩色像素，并且根据新的深度值绘制深度像素。这意味着，在其他像素后面的像素不会被绘制。

我们可以打开这个功能就像打开 culling 一样简单，用下面语句

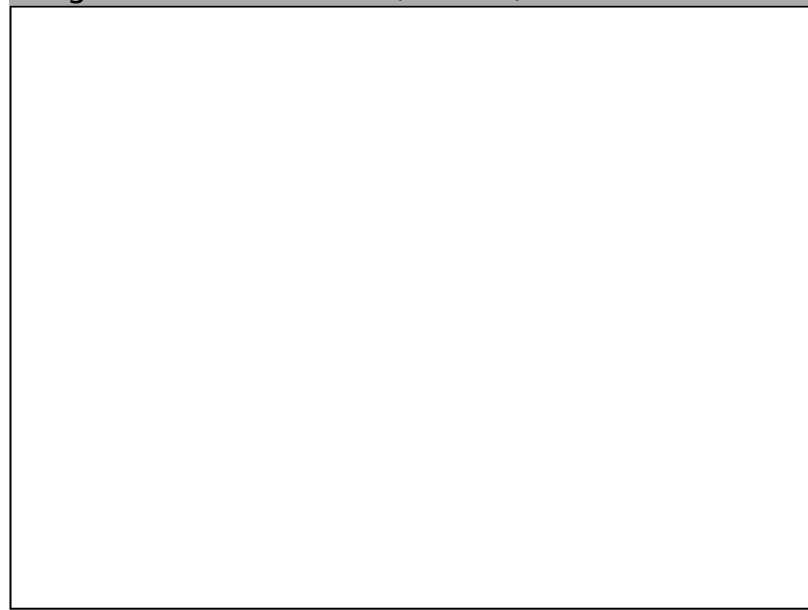
```
gl.enable(gl.DEPTH_TEST);
```

我们还需要在我们开始绘画之前清除深度缓冲回到 1.0。

```
// Draw the scene.
function drawScene() {
  // Clear the canvas AND the depth buffer.
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  ...
}
```

现在我们得到

Drag sliders to translate, rotate, and scale.



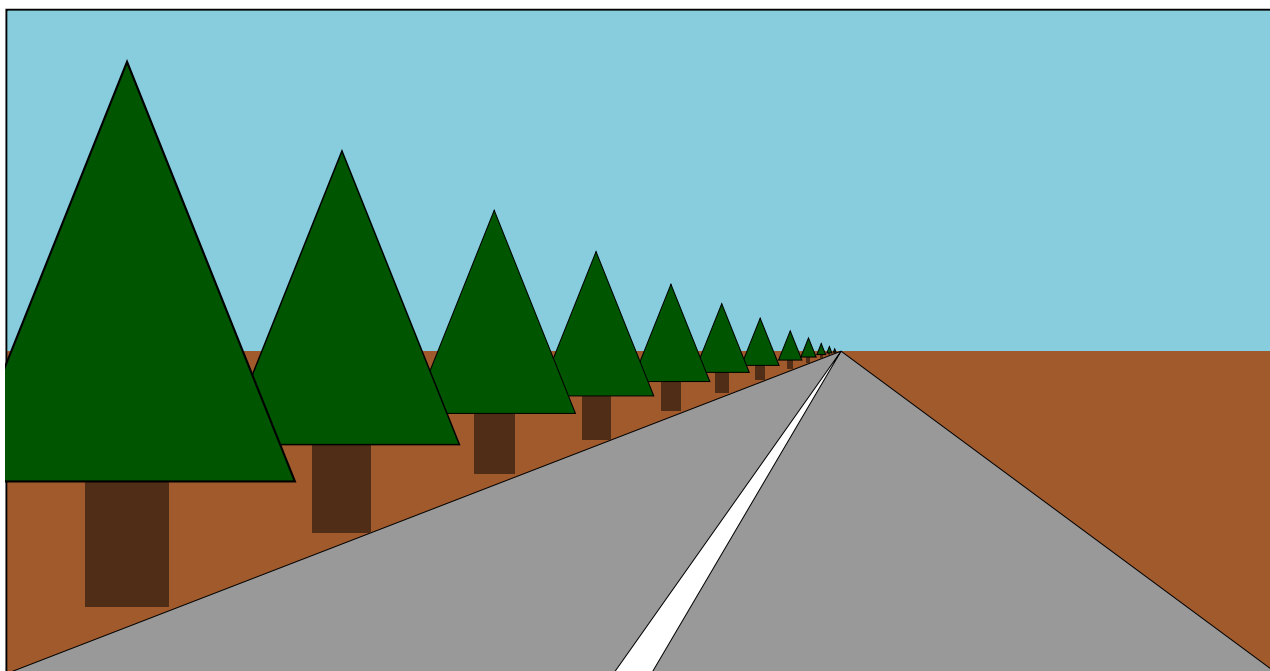
这是一个 3D!



## WebGL 3D 透视

在上一篇文章中，我们就学习过如何做三维，但三维没有任何透视。它是利用一个所谓的“正交”的观点，它固然有其用途，但这通常不是人们说“3D”时他们想要的。

相反，我们需要补充透视。只不过什么是透视？它基本上是一种事物越远显得更小的特征。



看上面的例子，我们看到越远的东西被画得越小。鉴于我们目前样品的一个让较远的物体显得更小简单的方法就是将 clip space X 和 Y 除以 Z。

可以这样想：如果你有一条从 (10, 15) 到 (20, 15) 的线段，10 个单位长。在我们目前的样本中，它将绘制 10 像素长。但是如果我们除以 Z，例如例子中如果是 Z 是 1

```
10 / 1 = 10  
20 / 1 = 20  
abs(10-20) = 10
```

这将是 10 像素，如果 Z 是 2，则有

```
10 / 2 = 5  
20 / 2 = 10  
abs(5 - 10) = 5
```

5 像素长。如果 Z = 3，则有

```
10 / 3 = 3.333
20 / 3 = 6.666
abs(3.333 - 6.666) = 3.333
```

你可以看到，随着  $Z$  的增加，随着它变得越来越远，我们最终会把它画得更小。如果我们在 clip space 中除，我们可能得到更好的结果，因为  $Z$  将是一个较小的数字（ $-1$  到  $+1$ ）。如果在除之前我们加一个 `fudgeFactor` 乘以  $Z$ ，对于一个给定的距离我们可以调整事物多小。

让我们尝试一下。首先让我们在乘以我们的“`fudgefactor`”后改变顶点着色器除以  $Z$ 。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
...
uniform float u_fudgeFactor;
...
void main() {
    // Multiply the position by the matrix.
    vec4 position = u_matrix * a_position;

    // Adjust the z to divide by
    float zToDivideBy = 1.0 + position.z * u_fudgeFactor;

    // Divide x and y by z.
    gl_Position = vec4(position.xy / zToDivideBy, position.zw);
}
</script>
```

注意，因为在 clip space 中  $Z$  从  $-1$  到  $+1$ ，我加 1 得到 `zToDivideBy` 从 0 到  $+2 * \text{fudgeFactor}$

我们也需要更新代码，让我们设置 `fudgeFactor`。

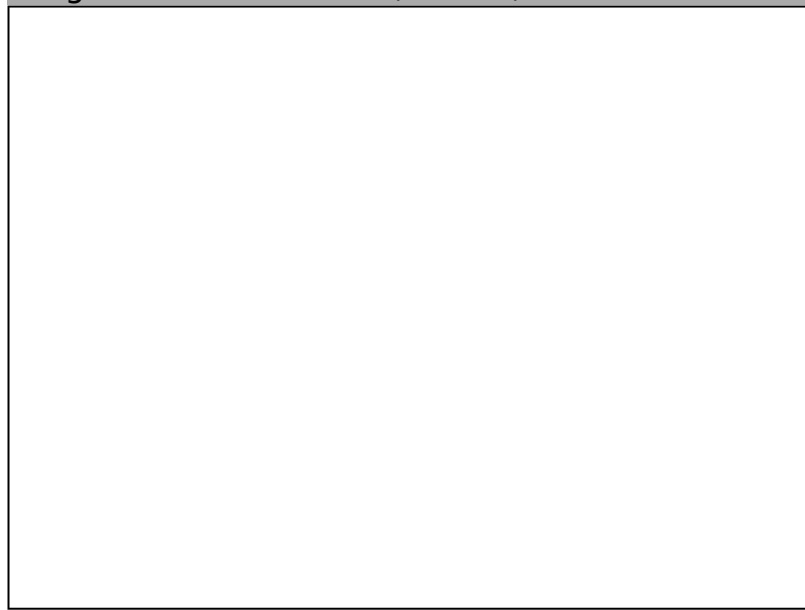
```
...
var fudgeLocation = gl.getUniformLocation(program, "u_fudgeFactor");

...
var fudgeFactor = 1;
...
function drawScene() {
    ...
    // Set the fudgeFactor
    gl.uniform1f(fudgeLocation, fudgeFactor);

    // Draw the geometry.
    gl.drawArrays(gl.TRIANGLES, 0, 16 * 6);
```

下面是结果。

Drag sliders to translate, rotate, and scale.



如果没有明确的把 “fudgefactor” 从 1 变化到 0 来看事物看起来像什么样子在我们除以 Z 之前。



WebGL 在我们的顶点着色器中把 X, Y, Z, W 值分配给 `gl_Position` 并且自动除以 W。

我们可以证明通过改变着色这很容易实现，而不是自己做除法，在 `gl_Position.w` 中加 `zToDivideBy`。

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
...
uniform float u_fudgeFactor;
...
void main() {
    // Multiply the position by the matrix.
    vec4 position = u_matrix * a_position;

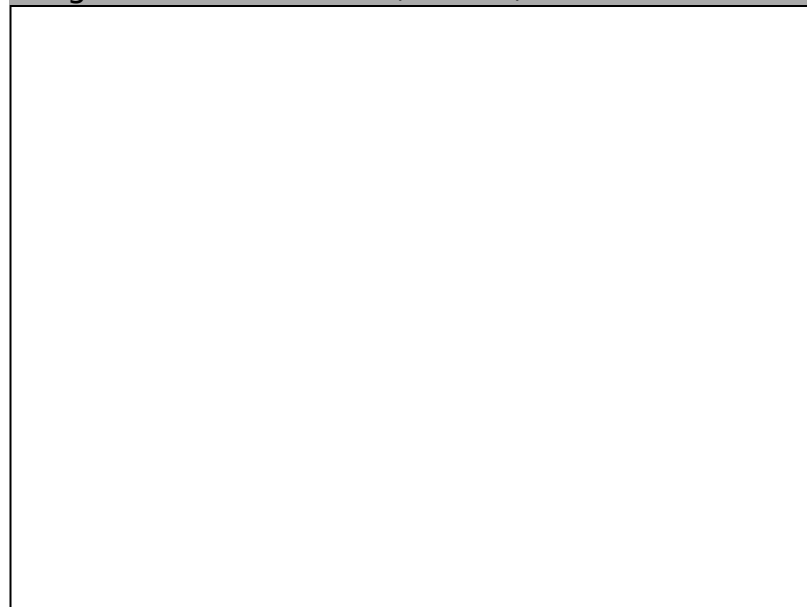
    // Adjust the z to divide by
    float zToDivideBy = 1.0 + position.z * u_fudgeFactor;

    // Divide x, y and z by zToDivideBy
    gl_Position = vec4(position.xyz, zToDivideBy);
}
```

```
}
</script>
```

看看这是完全相同的。

Drag sliders to translate, rotate, and scale.



为什么有这样一个事实：WebGL 自动除以 W？因为现在，使用更多维的矩阵，我们可以使用另一个矩阵复制 z 到 w。

矩阵如下

```
1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, 1,
0, 0, 0, 0,
```

将复制 z 到 w.你可以看看这些列如下

```
x_out = x_in * 1 +
      y_in * 0 +
      z_in * 0 +
      w_in * 0;
```

```
y_out = x_in * 0 +
      y_in * 1 +
      z_in * 0 +
      w_in * 0;
```

```
z_out = x_in * 0 +
      y_in * 0 +
      z_in * 1 +
```

```

    w_in * 0 ;

w_out = x_in * 0 +
    y_in * 0 +
    z_in * 1 +
    w_in * 0 ;

```

简化后如下

```

x_out = x_in;
y_out = y_in;
z_out = z_in;
w_out = z_in;

```

我们可以加 1 我们之前用的这个矩阵，因为我们知道  $w_{in}$  总是 1.0。

```

1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, 1,
0, 0, 0, 1,

```

这将改变  $W$  计算如下

```

w_out = x_in * 0 +
    y_in * 0 +
    z_in * 1 +
    w_in * 1 ;

```

因为我们知道  $w_{in} = 1.0$  所以就有

```

w_out = z_in + 1;

```

最后我们可以将 `fudgeFactor` 加到矩阵，矩阵如下

```

1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, fudgeFactor,
0, 0, 0, 1,

```

这意味着

```

w_out = x_in * 0 +
    y_in * 0 +
    z_in * fudgeFactor +
    w_in * 1 ;

```

简化后如下

```
w_out = z_in * fudgeFactor + 1;
```

所以，让我们再次修改程序只使用矩阵。

首先让我们放回顶点着色器。这很简单

```
<script id="2d-vertex-shader" type="x-shader/x-vertex">
uniform mat4 u_matrix;

void main() {
    // Multiply the position by the matrix.
    gl_Position = u_matrix * a_position;
    ...
}
</script>
```

接下来让我们做一个函数使  $Z \rightarrow W$  矩阵。

```
function makeZToWMatrix(fudgeFactor) {
    return [
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, fudgeFactor,
        0, 0, 0, 1,
    ];
}
```

我们将更改代码，以使用它。

```
...
// Compute the matrices
var zToWMatrix =
    makeZToWMatrix(fudgeFactor);

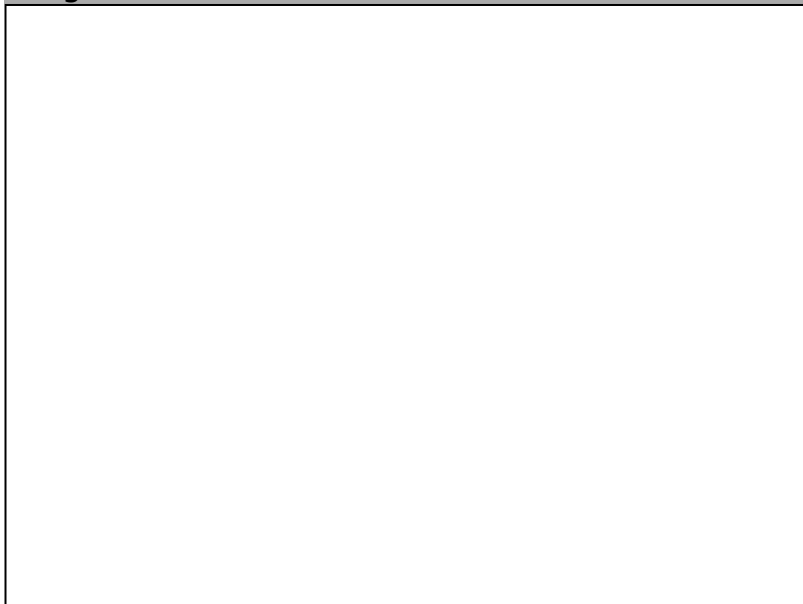
...

// Multiply the matrices.
var matrix = matrixMultiply(scaleMatrix, rotationZMatrix);
matrix = matrixMultiply(matrix, rotationYMatrix);
matrix = matrixMultiply(matrix, rotationXMatrix);
matrix = matrixMultiply(matrix, translationMatrix);
matrix = matrixMultiply(matrix, projectionMatrix);
matrix = matrixMultiply(matrix, zToWMatrix);

...
```

注意，这一次也是完全相同的。

Drag sliders to translate, rotate, and scale.



以上基本上是向你们展示，除以  $Z$  给了我们透视图，WebGL 方便地为我们除以  $Z$ 。

但是仍然有一些问题。例如如果你设置  $Z$  到  $-100$  左右，你会看到类似下面的动画。

**z: -30**



发生了什么事？为什么  $F$  消失得很早？就像 WebGL 剪辑  $X$  和  $Y$  或  $+1$  到  $-1$  它也剪辑  $Z$ 。这里看到的的就是  $Z < -1$  的地方。

我可以详细了解如何解决它，但你可以以我们做二维投影相同的方式来[得到它](#)。我们需要利用  $Z$ ，添加一些数量和测量一定量，我们可以做任何我们想要得到的  $-1$  到  $1$  的映射范围。

真正酷的事情是所有这些步骤可以在 1 个矩阵内完成。甚至更好的，我们来决定一个 `fieldOfView` 而不是一个 `frustum` `Factor`，并且计算正确的值来做这件事。

```

ect, near, far) {
dians);

```

我们在 `clipSpace` 中它会做数学运算，因此我们可以通过它假定有一个 `eye` 或 `camera` 在原点  $(0, 0, 0)$  并它在 `zNear` 的物体在  $z = -1$  结束以及在 `zNear` 的物体和  $y = 1$  结束。计算 `X` 所使用的只是乘以传入的 `aspect` 后，它计算出在 `Z` 区域物体的规模，因此在 `zFar` 的物体



形状像四面锥的立方体旋转称为“截锥”。矩阵在截锥内占空间并且转换到 clip space。zNear 定义夹在前面的物体，zfar定义夹在后面的物体。设置 zNear 为23你会看到旋转的立方体的前面得到裁剪。设置 zFar 为24你会看到立方体的后面得到剪辑。

只剩下一个问题。这个矩阵假定有一个视角在 0, 0, 0 并假定它在Z轴负方向，Y的正方向。我们的矩阵到目前为止已经以不同的方式解决问题。为了使它工作，我们需要我们的对象在前面的视图。

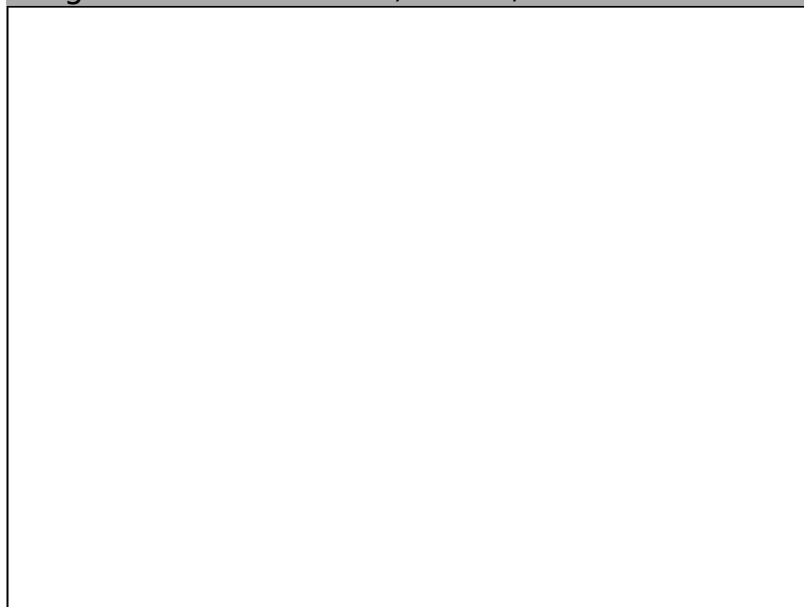
我们可以通过移动我们的 F 做到。我们在 ( 45, 150, 0 ) 绘图。让我们将它移到 ( 0, 150, - 360 )

现在，要想使用它，我们只需要用对 makePerspective 的调用取代对make2DProjection 旧的调用

```
var aspect = canvas.clientWidth / canvas.clientHeight;
var projectionMatrix =
    makePerspective(fieldOfViewRadians, aspect, 1, 2000);
var translationMatrix =
    makeTranslation(translation[0], translation[1], translation[2]);
var rotationXMatrix = makeXRotation(rotation[0]);
var rotationYMatrix = makeYRotation(rotation[1]);
var rotationZMatrix = makeZRotation(rotation[2]);
var scaleMatrix = makeScale(scale[0], scale[1], scale[2]);
```

结果如下

Drag sliders to translate, rotate, and scale.



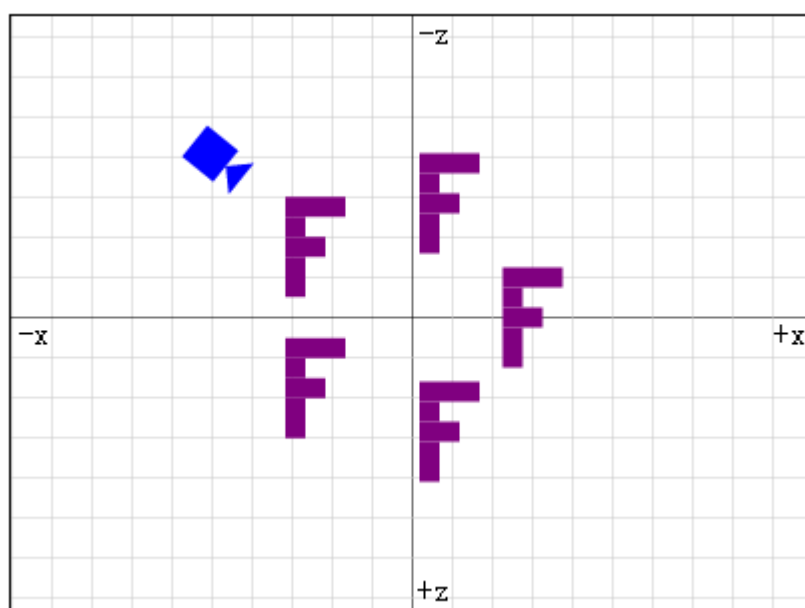
我们回到了一个矩阵乘法，我们得到两个领域的视图，我们可以选择我们的 z 空间。受篇幅限制我们没有做。下一节，[摄像机](#)。

## WebGL 3D 摄像机

在过去的章节里我们将 F 移动到截锥的前面，因为 `makePerspective` 函数从原点  $(0, 0, 0)$  度量它，并且截锥的对象从  $-zNear$  到  $-zFar$  都在它前面。

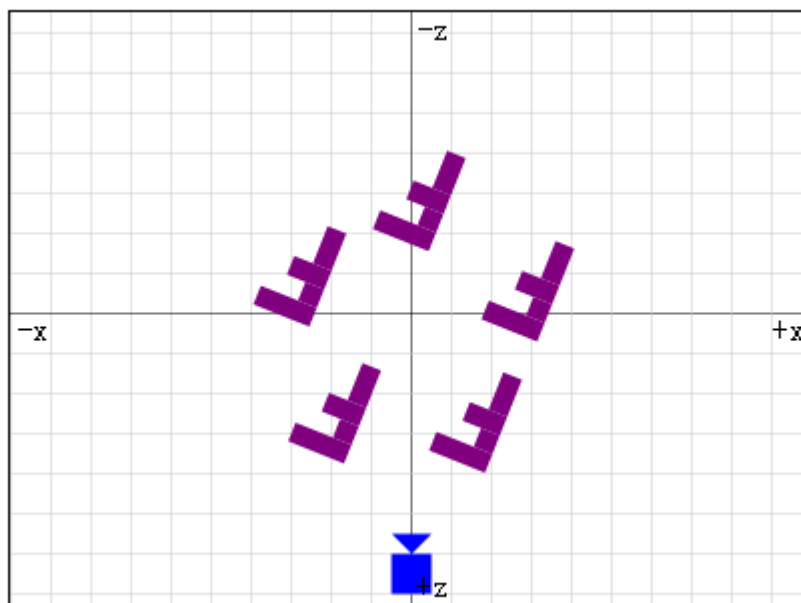
视点前面移动的物体似乎没有正确的方式去做吗？在现实世界中，你通常会移动你的相机来给建筑物拍照。

将摄像机移动到对象前



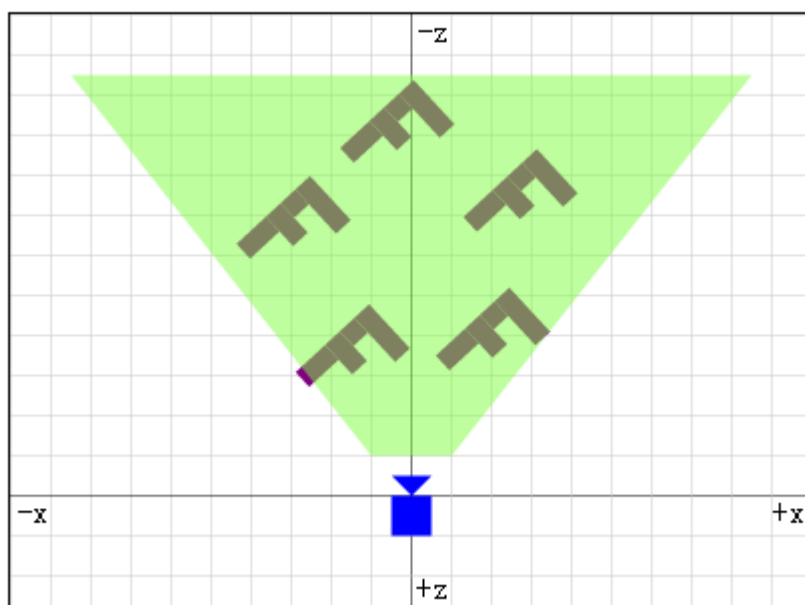
你通常不会将建筑移动到摄像机前。

将对象移动到摄像机前



但在我们最后一篇文章中，我们提出了一个投影，这就需要物体在 Z 轴的原点前面。为了实现它，我们想做的是把摄像机移动到原点，然后把所有的其它物体都移动恰当的距离，所以它相对于摄像机仍然是在同一个地方。

将对象移动到视图



我们需要有效地将现实中的物体移动到摄像机的前面。能达到这个目的的最简单的方法是使用“逆”矩阵。一般情况下的逆矩阵的计算是复杂的，但从概念上讲，它是容易的。逆是你用来作为其他数值的对立的值。例如，123

的是相反数是  $-123$ 。缩放比例为5的规模矩阵的逆是  $1/5$  或  $0.2$ 。在  $X$  域旋转  $30^\circ$  的矩阵的逆是一个在  $X$  域旋转  $-30^\circ$  的矩阵。

直到现在我们已使用了平移，旋转和缩放来影响我们的 'F' 的位置和方向。把所有的矩阵相乘后，我们有一个单一的矩阵，表示如何将“F”以我们希望的大小和方向从原点移动到相应位置。使用摄像机我们可以做相同的事情。一旦我们的矩阵告诉我们如何从原点到我们想要的位置移动和旋转摄像机，我们就可以计算它的逆，它将给我们一个矩阵来告诉我们如何移动和旋转其它一切物体的相对数量，这将有效地使摄像机在点  $(0, 0, 0)$ ，并且我们已经将一切物体移动到它的前面。

让我们做一个有一圈 'F' 的三维场景，就像上面的图表那样。

下面是实现代码。

```
var numFs = 5;
var radius = 200;

// Compute the projection matrix
var aspect = canvas.clientWidth / canvas.clientHeight;
var projectionMatrix =
    makePerspective(fieldOfViewRadians, aspect, 1, 2000);

// Draw 'F's in a circle
for (var ii = 0; ii < numFs; ++ii) {
    var angle = ii * Math.PI * 2 / numFs;

    var x = Math.cos(angle) * radius;
    var z = Math.sin(angle) * radius;
    var translationMatrix = makeTranslation(x, 0, z);

    // Multiply the matrices.
    var matrix = translationMatrix;
    matrix = matrixMultiply(matrix, projectionMatrix);

    // Set the matrix.
    gl.uniformMatrix4fv(matrixLocation, false, matrix);

    // Draw the geometry.
    gl.drawArrays(gl.TRIANGLES, 0, 16 * 6);
}
```

就在我们计算出我们的投影矩阵之后，我们就可以计算出一个就像上面的图表中显示的那样围绕‘F’旋转的摄像机。

```
// Compute the camera's matrix
var cameraMatrix = makeTranslation(0, 0, radius * 1.5);
cameraMatrix = matrixMultiply(
    cameraMatrix, makeYRotation(cameraAngleRadians));
```

然后，我们根据相机矩阵计算“视图矩阵”。“视图矩阵”是将一切物体移动到摄像机相反的位置，这有效地使摄像机相对于一切物体就像在原点（0,0,0）。

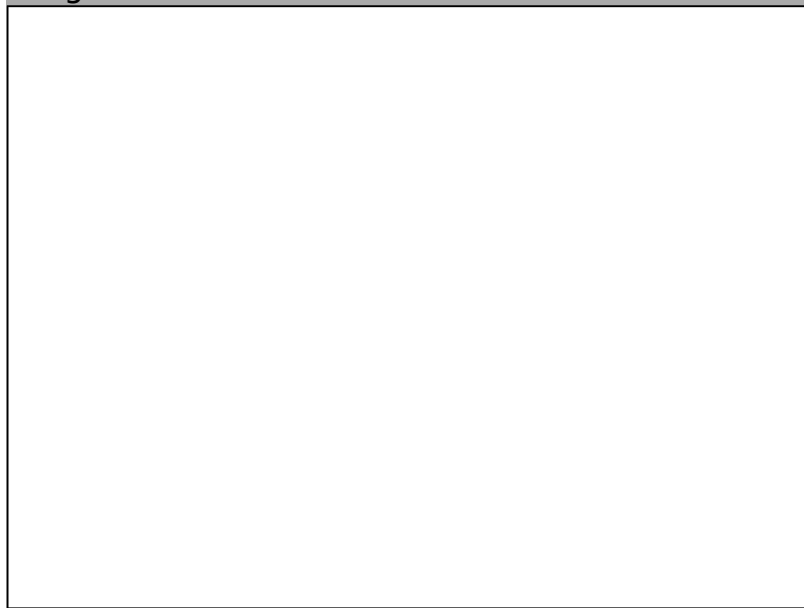
```
// Make a view matrix from the camera matrix.
var viewMatrix = makeInverse(cameraMatrix);
```

最后我们需要应用视图矩阵来计算每个‘F’的矩阵

```
// Multiply the matrices.
var matrix = translationMatrix;
matrix = matrixMultiply(matrix, viewMatrix); // <== added
matrix = matrixMultiply(matrix, projectionMatrix);
```

一个摄像机可以绕着一圈“F”。拖动 cameraAngle 滑块来移动摄像机。

**Drag slider to move camera.**



这一切都很好，但使用旋转和平移来移动一个摄像头到你想要的地方，并且指向你想看到的地方并不总是很容易。例如如果我们想要摄像机总是指向特定的‘F’就要进行一些非常复杂的数学计算来决定当摄像机绕‘F’圈旋转的时候如何旋转摄像机来指向那个‘F’。

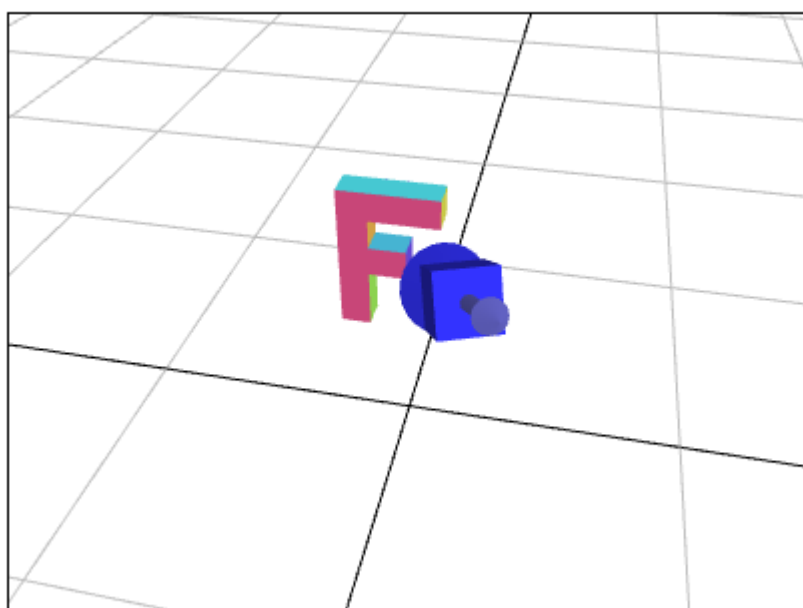
幸运的是，有一个更容易的方式。我们可以决定摄像机在我们想要的地方并且可以决定它指向什么，然后计算矩阵，这个矩阵可以将把摄像机放到那里。基于矩阵的工作原理这非常容易实现。

首先，我们需要知道我们想要摄像机在什么位置。我们将称之为 CameraPosition。然后我们需要了解我们看过或瞄准的物体的位置。我们将把它称为 target。如果我们将 CameraPosition 减去 target 我们将得到一个向

量，它指向从摄像头获取目标的方向。让我们称它为 **zAxis**。因为我们知道摄像机指向  $-Z$  方向，我们可以从另一方向做减法  $\text{cameraPosition} - \text{target}$ 。我们将结果规范化，并直接复制到  $z$  区域矩阵。

```
+-----+-----+-----+-----+
|   |   |   |   |
+-----+-----+-----+-----+
|   |   |   |   |
+-----+-----+-----+-----+
|Zx|Zy|Zz|   |
+-----+-----+-----+-----+
|   |   |   |   |
+-----+-----+-----+-----+
```

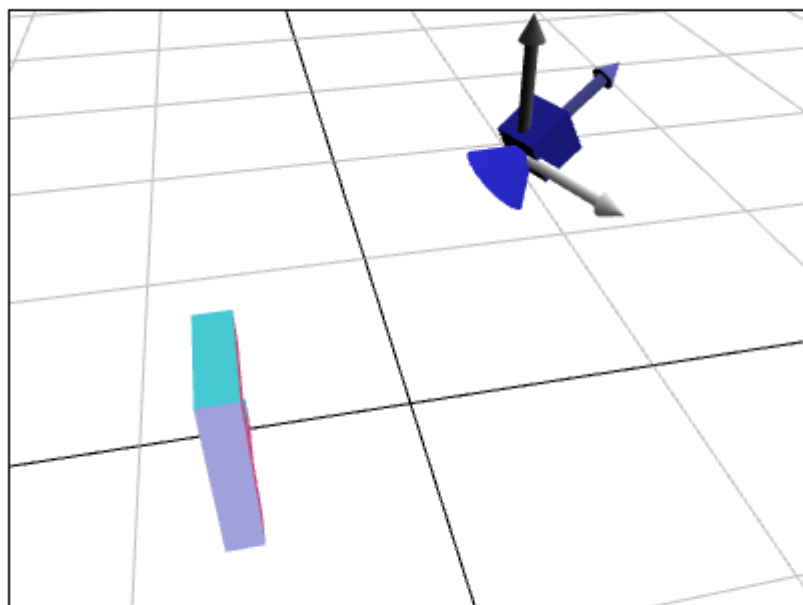
这部分矩阵表示的是  $Z$  轴。在这种情况下，是摄像机的  $Z$  轴。一个向量的标准化意味着它代表了 1.0。如果你回到二维旋转的文章，在哪里我们谈到了如何与单位圆以及二维旋转，在三维中我们需要单位球面和一个归一化的向量来代表在单位球面上一点。



虽然没有足够的信息。只是一个单一的向量给我们一个点的单位范围内，但从这一点到东方的东西？我们需要把矩阵的其他部分填好。特别的  $X$  轴和  $Y$  轴类零件。我们知道这 3 个部分是相互垂直的。我们也知道，“一般”我们不把相机指向。因为，如果我们知道哪个方向是向上的，在这种情况下  $(0,1,0)$ ，我们可以使用一种叫做“跨产品和“计算  $X$  轴和  $Y$  轴的矩阵。

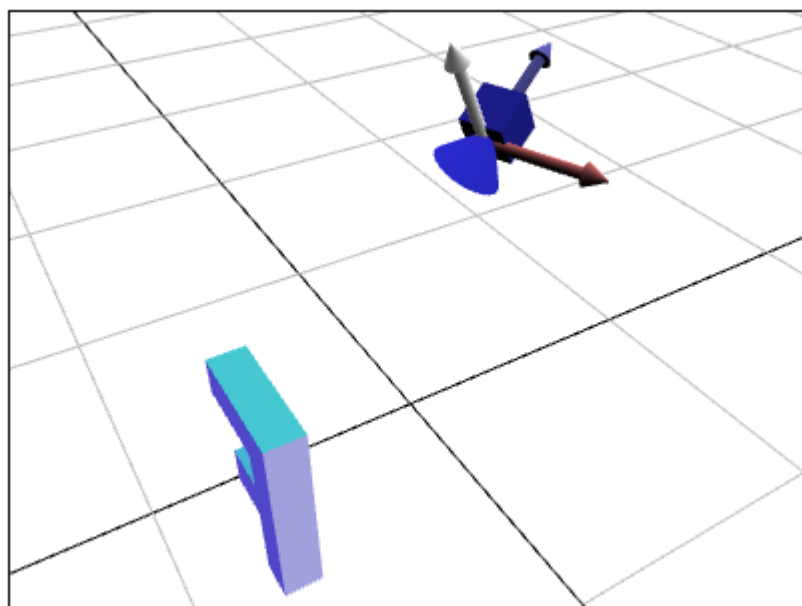
我不知道一个跨产品意味着在数学方面。我所知道的是，如果你有 2 个单位向量和计算的交叉产品，你会得到一个向量，是垂直于这 2 个向量。换句话说，如果你有一个向量指向东南方，和一个向量指向上，和你计算交叉

产品，你会得到一个向量指向北西或北东自这2个向量，purpendicular 到东南亚和。根据你计算交叉产品的顺序，你会得到相反的答案。



$$\text{zAxis cross up} = \text{xAxis}$$

现在，我们有 xAxis，我们可以通过 zAxis 和 xAxis 得到摄像机的 yAxis



现在我们所要做的就是将 3 个轴插入一个矩阵。这使得矩阵可以指向物体，从 cameraPosition 指向 target。我们只需要添加 position

```

+-----+-----+-----+-----+
| Xx | Xy | Xz | 0 | <- x axis
+-----+-----+-----+-----+
| Yx | Yy | Yz | 0 | <- y axis
+-----+-----+-----+-----+
| Zx | Zy | Zz | 0 | <- z axis
+-----+-----+-----+-----+
| Tx | Ty | Tz | 1 | <- camera position
+-----+-----+-----+-----+

```

下面是用来计算 2 个向量的交叉乘积的代码。

```

function cross(a, b) {
  return [a[1] * b[2] - a[2] * b[1],
    a[2] * b[0] - a[0] * b[2],
    a[0] * b[1] - a[1] * b[0]];
}

```

这是减去两个向量的代码。

```

function subtractVectors(a, b) {
  return [a[0] - b[0], a[1] - b[1], a[2] - b[2]];
}

```

这里是规范化一个向量（使其成为一个单位向量）的代码。

```

function normalize(v) {
  var length = Math.sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  // make sure we don't divide by 0.
  if (length > 0.00001) {
    return [v[0] / length, v[1] / length, v[2] / length];
  } else {
    return [0, 0, 0];
  }
}

```

下面是计算一个 "lookAt" 矩阵的代码。

```

function makeLookAt(cameraPosition, target, up) {
  var zAxis = normalize(
    subtractVectors(cameraPosition, target));
  var xAxis = cross(up, zAxis);
  var yAxis = cross(zAxis, xAxis);

  return [
    xAxis[0], xAxis[1], xAxis[2], 0,

```



```

    yAxis[0], yAxis[1], yAxis[2], 0,
    zAxis[0], zAxis[1], zAxis[2], 0,
    cameraPosition[0],
    cameraPosition[1],
    cameraPosition[2],
    1;
}

```

这是我们如何使用它来使相机随着我们移动它指向在一个特定的 ‘F’ 的。

```

...

// Compute the position of the first F
var fPosition = [radius, 0, 0];

// Use matrix math to compute a position on the circle.
var cameraMatrix = makeTranslation(0, 50, radius * 1.5);
cameraMatrix = matrixMultiply(
    cameraMatrix, makeYRotation(cameraAngleRadians));

// Get the camera's position from the matrix we computed
cameraPosition = [
    cameraMatrix[12],
    cameraMatrix[13],
    cameraMatrix[14]];

var up = [0, 1, 0];

// Compute the camera's matrix using look at.
var cameraMatrix = makeLookAt(cameraPosition, fPosition, up);

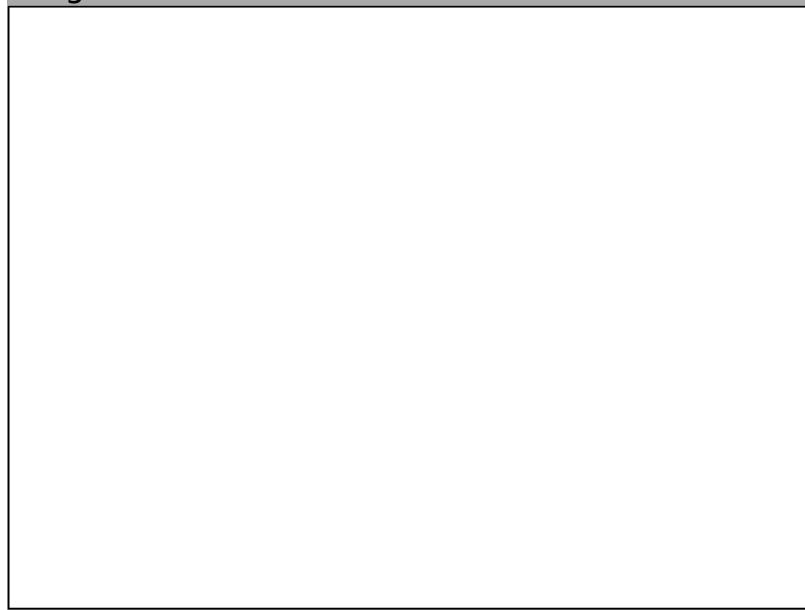
// Make a view matrix from the camera matrix.
var viewMatrix = makeInverse(cameraMatrix);

...

```

下面是结果。

Drag slider to move camera.



拖动滑块，注意到相机追踪一个 ‘F’ 。

请注意，您可以不只对摄像机使用 “lookAt” 函数。共同的用途是使一个人物的头跟着某人。使小塔瞄准一个目标。使对象遵循一个路径。你计算目标的路径。然后你计算出目标在未来几分钟在路径的什么地方。把这两个值放进你的 lookAt 函数，你会得到一个矩阵，使你的对象跟着路径并且朝向路径。



结构与组织



## WebGL – 更少的代码，更多的乐趣

---

WebGL 程序要求你编写必须编译和链接的着色器程序，然后你需要查看对于这些着色器程序的输入的位置。这些输入被称为制服和属性，同时用来查找它们的位置的代码可能是冗长而乏味的。

假设我们已经有了[用来编译和链接着色器程序的 WebGL 代码的典型样本](#)。下面给出了一组着色器。

顶点着色器：

```
uniform mat4 u_worldViewProjection;
uniform vec3 u_lightWorldPos;
uniform mat4 u_world;
uniform mat4 u_viewInverse;
uniform mat4 u_worldInverseTranspose;

attribute vec4 a_position;
attribute vec3 a_normal;
attribute vec2 a_texcoord;

varying vec4 v_position;
varying vec2 v_texCoord;
varying vec3 v_normal;
varying vec3 v_surfaceToLight;
varying vec3 v_surfaceToView;

void main() {
    v_texCoord = a_texcoord;
    v_position = (u_worldViewProjection * a_position);
    v_normal = (u_worldInverseTranspose * vec4(a_normal, 0)).xyz;
    v_surfaceToLight = u_lightWorldPos - (u_world * a_position).xyz;
    v_surfaceToView = (u_viewInverse[3] - (u_world * a_position)).xyz;
    gl_Position = v_position;
}
```

片段着色器：

```
precision mediump float;

varying vec4 v_position;
varying vec2 v_texCoord;
varying vec3 v_normal;
varying vec3 v_surfaceToLight;
varying vec3 v_surfaceToView;
```

```

uniform vec4 u_lightColor;
uniform vec4 u_ambient;
uniform sampler2D u_diffuse;
uniform vec4 u_specular;
uniform float u_shininess;
uniform float u_specularFactor;

vec4 lit(float l, float h, float m) {
    return vec4(1.0,
        max(l, 0.0),
        (l > 0.0) ? pow(max(0.0, h), m) : 0.0,
        1.0);
}

void main() {
    vec4 diffuseColor = texture2D(u_diffuse, v_texCoord);
    vec3 a_normal = normalize(v_normal);
    vec3 surfaceToLight = normalize(v_surfaceToLight);
    vec3 surfaceToView = normalize(v_surfaceToView);
    vec3 halfVector = normalize(surfaceToLight + surfaceToView);
    vec4 litR = lit(dot(a_normal, surfaceToLight),
        dot(a_normal, halfVector), u_shininess);
    vec4 outColor = vec4((
        u_lightColor * (diffuseColor * litR.y + diffuseColor * u_ambient +
        u_specular * litR.z * u_specularFactor)).rgb,
        diffuseColor.a);
    gl_FragColor = outColor;
}

```

你最终不得不像以下这样编写代码，来对所有要绘制的各种各样的值进行查找和设置。

```

// At initialization time
var u_worldViewProjectionLoc = gl.getUniformLocation(program, "u_worldViewProjection");
var u_lightWorldPosLoc = gl.getUniformLocation(program, "u_lightWorldPos");
var u_worldLoc = gl.getUniformLocation(program, "u_world");
var u_viewInverseLoc = gl.getUniformLocation(program, "u_viewInverse");
var u_worldInverseTransposeLoc = gl.getUniformLocation(program, "u_worldInverseTranspose");
var u_lightColorLoc = gl.getUniformLocation(program, "u_lightColor");
var u_ambientLoc = gl.getUniformLocation(program, "u_ambient");
var u_diffuseLoc = gl.getUniformLocation(program, "u_diffuse");
var u_specularLoc = gl.getUniformLocation(program, "u_specular");
var u_shininessLoc = gl.getUniformLocation(program, "u_shininess");
var u_specularFactorLoc = gl.getUniformLocation(program, "u_specularFactor");

var a_positionLoc = gl.getAttribLocation(program, "a_position");

```

```

var a_normalLoc= gl.getAttribLocation(program, "a_normal");
var a_texCoordLoc = gl.getAttribLocation(program, "a_texcoord");

// At init or draw time depending on use.
var someWorldViewProjectionMat = computeWorldViewProjectionMatrix();
var lightWorldPos = [100, 200, 300];
var worldMat = computeWorldMatrix();
var viewInverseMat = computeInverseViewMatrix();
var worldInverseTransposeMat = computeWorldInverseTransposeMatrix();
var lightColor = [1, 1, 1, 1];
var ambientColor = [0.1, 0.1, 0.1, 1];
var diffuseTextureUnit = 0;
var specularColor = [1, 1, 1, 1];
var shininess = 60;
var specularFactor = 1;

// At draw time
gl.useProgram(program);

// Setup all the buffers and attributes
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.enableVertexAttribArray(a_positionLoc);
gl.vertexAttribPointer(a_positionLoc, positionNumComponents, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.enableVertexAttribArray(a_normalLoc);
gl.vertexAttribPointer(a_normalLoc, normalNumComponents, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
gl.enableVertexAttribArray(a_texcoordLoc);
gl.vertexAttribPointer(a_texcoordLoc, texcoordNumComponents, gl.FLOAT, 0, 0);

// Setup the textures used
gl.activeTexture(gl.TEXTURE0 + diffuseTextureUnit);
gl.bindTexture(gl.TEXTURE_2D, diffuseTexture);

// Set all the uniforms.
gl.uniformMatrix4fv(u_worldViewProjectionLoc, false, someWorldViewProjectionMat);
gl.uniform3fv(u_lightWorldPosLoc, lightWorldPos);
gl.uniformMatrix4fv(u_worldLoc, worldMat);
gl.uniformMatrix4fv(u_viewInverseLoc, viewInverseMat);
gl.uniformMatrix4fv(u_worldInverseTransposeLoc, worldInverseTransposeMat);
gl.uniform4fv(u_lightColorLoc, lightColor);
gl.uniform4fv(u_ambientLoc, ambientColor);
gl.uniform1i(u_diffuseLoc, diffuseTextureUnit);

```

```
gl.uniform4fv(u_specularLoc, specularColor);
gl.uniform1f(u_shininessLoc, shininess);
gl.uniform1f(u_specularFactorLoc, specularFactor);

gl.drawArrays(...);
```

这是大量的输入。

这里有许多方法可以用来简化它。其中一项建议是要求 WebGL 告诉我们所有的制服和位置，然后设置函数，来帮助我们建立它们。然后我们可以通过 JavaScript 对象来使设置我们的设置更加容易。如果还是不清楚，我们的代码将会跟以下代码类似

```
// At initiation time
var uniformSetters = createUniformSetters(gl, program);
var attribSetters = createAttributeSetters(gl, program);

var attribs = {
  a_position: { buffer: positionBuffer, numComponents: 3, },
  a_normal: { buffer: normalBuffer, numComponents: 3, },
  a_texcoord: { buffer: texcoordBuffer, numComponents: 2, },
};

// At init time or draw time depending on use.
var uniforms = {
  u_worldViewProjection: computeWorldViewProjectionMatrix(...),
  u_lightWorldPos: [100, 200, 300],
  u_world: computeWorldMatrix(),
  u_viewInverse: computeInverseViewMatrix(),
  u_worldInverseTranspose: computeWorldInverseTransposeMatrix(),
  u_lightColor:[1, 1, 1, 1],
  u_ambient: [0.1, 0.1, 0.1, 1],
  u_diffuse: diffuseTexture,
  u_specular: [1, 1, 1, 1],
  u_shininess: 60,
  u_specularFactor:1,
};

// At draw time
gl.useProgram(program);

// Setup all the buffers and attributes
setAttributes(attribSetters, attribs);

// Set all the uniforms and textures used.
setUniforms(uniformSetters, uniforms);
```

```
gl.drawArrays(...);
```

这对于我来说，看起来是很多的更小，更容易，更少的代码。

你甚至可以使用多个 JavaScript 对象，如果那样适合你的话。如下所示

```
// At initiation time
var uniformSetters = createUniformSetters(gl, program);
var attribSetters = createAttributeSetters(gl, program);

var attribs = {
  a_position: { buffer: positionBuffer, numComponents: 3, },
  a_normal: { buffer: normalBuffer, numComponents: 3, },
  a_texcoord: { buffer: texcoordBuffer, numComponents: 2, },
};

// At init time or draw time depending
var uniformsThatAreTheSameForAllObjects = {
  u_lightWorldPos: [100, 200, 300],
  u_viewInverse: computeInverseViewMatrix(),
  u_lightColor:[1, 1, 1, 1],
};

var uniformsThatAreComputedForEachObject = {
  u_worldViewProjection: perspective(...),
  u_world: computeWorldMatrix(),
  u_worldInverseTranspose: computeWorldInverseTransposeMatrix(),
};

var objects = [
  { translation: [10, 50, 100],
  materialUniforms: {
    u_ambient: [0.1, 0.1, 0.1, 1],
    u_diffuse: diffuseTexture,
    u_specular: [1, 1, 1, 1],
    u_shininess: 60,
    u_specularFactor:1,
  },
  },
  { translation: [-120, 20, 44],
  materialUniforms: {
    u_ambient: [0.1, 0.2, 0.1, 1],
    u_diffuse: someOtherDiffuseTexture,
    u_specular: [1, 1, 0, 1],
    u_shininess: 30,
```



```

    u_specularFactor:0.5,
  },
  {
    { translation: [200, -23, -78],
materialUniforms: {
  u_ambient: [0.2, 0.2, 0.1, 1],
  u_diffuse: yetAnotherDiffuseTexture,
  u_specular: [1, 0, 0, 1],
  u_shininess: 45,
  u_specularFactor:0.7,
},
},
];

// At draw time
gl.useProgram(program);

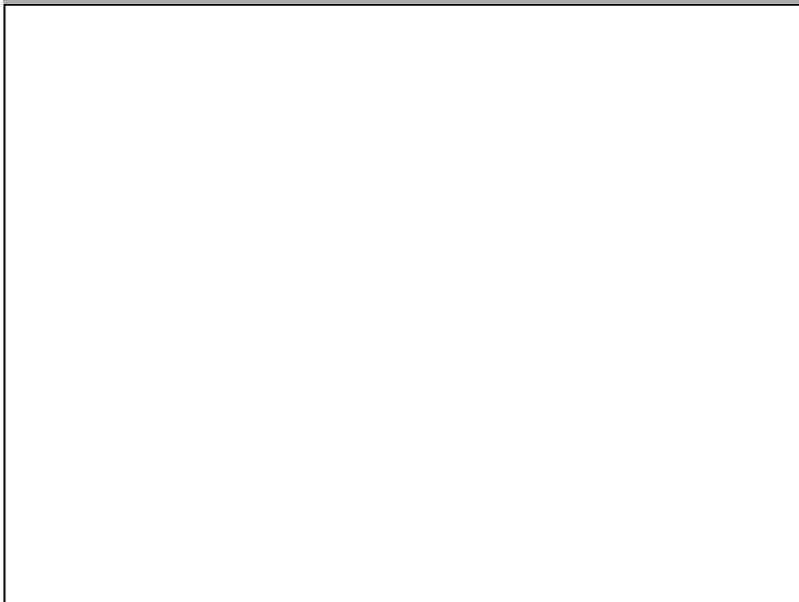
// Setup the parts that are common for all objects
setAttributes(attribSetters, attribs);
setUniforms(uniformSetters, uniformThatAreTheSameForAllObjects);

objects.forEach(function(object) {
  computeMatricesForObject(object, uniformsThatAreComputedForEachObject);
  setUniforms(uniformSetters, uniformThatAreComputedForEachObject);
  setUniforms(uniformSetters, object.materialUniforms);
  gl.drawArrays(...);
});

```

这里有一个使用这些帮助函数的例子

**Uses a few utility functions so there's much less code.**



让我们向前更进一小步。在上面代码中，我们设置了一个拥有我们创建的缓冲区的变量 `attrs`。在代码中不显示设置这些缓冲区的代码。例如，如果你想要设置位置，法线和纹理坐标，你可能会需要这样的代码

```
// a single triangle
var positions = [0, -10, 0, 10, 10, 0, -10, 10, 0];
var texcoords = [0.5, 0, 1, 1, 0, 1];
var normals = [0, 0, 1, 0, 0, 1, 0, 0, 1];

var positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

var texcoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texcoords), gl.STATIC_DRAW);

var normalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals), gl.STATIC_DRAW);
```

看起来像一种我们也可以简化的模式。

```
// a single triangle
var arrays = {
  position: { numComponents: 3, data: [0, -10, 0, 10, 10, 0, -10, 10, 0], },
  texcoord: { numComponents: 2, data: [0.5, 0, 1, 1, 0, 1], },
  normal: { numComponents: 3, data: [0, 0, 1, 0, 0, 1, 0, 0, 1], },
};

var bufferInfo = createBufferInfoFromArrays(gl, arrays);
```

更短！现在我们可以渲染时间这样做

```
// Setup all the needed buffers and attributes.
setBuffersAndAttributes(gl, attribSetters, bufferInfo);

...

// Draw the geometry.
gl.drawArrays(gl.TRIANGLES, 0, bufferInfo.numElements);
```

如下所示

Uses a few utility functions so there's much less code.



如果我们有 `indices`，这可能会奏效。`setAttribsAndBuffers` 将会设置所有的属性，同时用你的 `indices` 来设置 `ELEMENT-ARRAY-BUFFER`。所以你可以调用 `gl.drawElements`。

```
// an indexed quad
var arrays = {
  position: { numComponents: 3, data: [0, 0, 0, 10, 0, 0, 0, 10, 0, 10, 10, 0], },
  texcoord: { numComponents: 2, data: [0, 0, 0, 1, 1, 0, 1, 1], },
  normal: { numComponents: 3, data: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1], },
  indices: { numComponents: 3, data: [0, 1, 2, 1, 2, 3], },
};

var bufferInfo = createBufferInfoFromTypedArray(gl, arrays);
```

同时在渲染时，我们可以调用 `gl.drawElements`，而不是 `gl.drawArrays`。

```
// Setup all the needed buffers and attributes.
setBuffersAndAttributes(gl, attribSetters, bufferInfo);

...

// Draw the geometry.
gl.drawElements(gl.TRIANGLES, bufferInfo.numElements, gl.UNSIGNED_SHORT, 0);
```

如下所示

Uses a few utility functions so there's much less code.



createBufferInfoFromArrays 基本上使一个对象与如下代码相似

```
bufferInfo = {
  numElements: 4, // or whatever the number of elements is
  indices: WebGLBuffer, // this property will not exist if there are no indices
  attribs: {
    a_position: { buffer: WebGLBuffer, numComponents: 3, },
    a_normal: { buffer: WebGLBuffer, numComponents: 3, },
    a_texcoord: { buffer: WebGLBuffer, numComponents: 2, },
  },
};
```

同时 `setBuffersAndAttributes` 使用这个对象来设置所有的缓冲区和属性。

最后我们可以进展到我之前认为可能太远的地步。给出的 `position` 几乎总是拥有 3 个组件 (x, y, z)，同时 `texcoords` 几乎总是拥有 2 个组件，`indices` 几乎总是有 3 个组件，同时 `normals` 总是有 3 个组件，我们就可以让系统来猜想组件的数量。

```
// an indexed quad
var arrays = {
  position: [0, 0, 0, 10, 0, 0, 0, 10, 0, 10, 10, 0],
  texcoord: [0, 0, 0, 1, 1, 0, 1, 1],
  normal: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
  indices: [0, 1, 2, 1, 2, 3],
};
```

以下是另一个版本。

Uses a few utility functions so there's much less code.

我不确认我个人喜欢那种版本。我可能猜测出错，因为它可能猜错。例如，我可能选择在我的 texcoord 属性中添加额外的一组结构坐标，然后它会猜测 2，是错误的。当然，如果它猜错了，你可以像以上示例中那样指定个数。我想我担心的，如果猜测代码改变了人们的日常的情况可能会打破。全都取决于你。一些人喜欢让东西尽量像他们考虑的那样简单。

我们为什么不在着色器程序中查看这些属性来得出组件的数量？那是因为，从一个缓冲区中提供 3 组件 (x, y, z)，但是在着色器中使用 `vec4` 是非常普遍的。对于属性 WebGL 会自动设置 `w = 1`。但是那意味着，我们不可能很容易的就知道用户的意图，因为他们在他们的着色器中声明的可能与他们提供的组件的数量不匹配。

如果想要寻找更多的模式，如下所示

```
var program = createProgramFromScripts(gl, ["vertexshader", "fragmentshader"]);
var uniformSetters = createUniformSetters(gl, program);
var attribSetters = createAttributeSetters(gl, program);
```

让我们将上述代码简化成如下代码

```
var programInfo = createProgramInfo(gl, ["vertexshader", "fragmentshader"]);
```

它将返回与下面代码类似的东西

```
programInfo = {
  program: WebGLProgram, // program we just compiled
  uniformSetters: ..., // setters as returned from createUniformSetters,
  attribSetters: ..., // setters as returned from createAttribSetters,
}
```

那是另一个更小的简化。在我们开始使用多个程序时，它将会派上用场，因为它自动保持与它们的相关联的程序的设定。

Uses a few utility functions so there's much less code.



无论如何，这是我想要编写我自己的 WebGL 程序的风格。在这些教程中的课程中，尽管我已经感觉到我需要使用标准的 `verbose` 方法，这样人们就不会对 WebGL 是什么和什么是我自己的风格感到困惑。在一些点上，尽管显示所有的可以获取这些点的方式的步骤，所以继续学习这些课程的可以在这种风格中被使用。

在你自己的代码中随便使用这种风格。 `createUniformSetters`， `createAttributeSetters`， `createBufferInfoFromArrays`， `setUniforms` 和 `setBuffersAndAttributes` 这些函数都包含在 [webgl-utils.js](#) 文件中，可以在所有的例子中使用。如果你想要一些更多有组织的东西，可以查看 [TWGL.js](#)。

## WebGL 绘制多个东西

在 WebGL 中第一次得到东西后最常见的问题之一是，我怎样绘制多个东西。

除了少数例外情况，首先要意识到的东西是，WebGL 就像某人写的包含某个函数，而不是向函数中传递大量参数，相反，你有一个独自的函数来绘制东西，同时有 70 + 函数来为一个函数设置状态。因此，例如假设你有一个可以绘制一个圆的函数。你可以像如下一样编写程序

```
function drawCircle(centerX, centerY, radius, color) { ... }
```

或者你可以像如下一样编写代码

```
var centerX;
var centerY;
var radius;
var color;

function setCenter(x, y) {
  centerX = x;
  centerY = y;
}

function setRadius(r) {
  radius = r;
}

function setColor(c) {
  color = c;
}

function drawCircle() {
  ...
}
```

WebGL 以第二种方式工作。函数，诸如 `gl.createBuffer`，`gl.bufferData`，`gl.createTexture` 和 `gl.texImage2D`，让你可以上传缓冲区（顶点）和质地（颜色，等等）数据到 WebGL。`gl.createProgram`，`gl.createShader`，`gl.compileProgram` 和 `gl.linkProgram` 让你可以创建你的 GLSL 着色器。当 `gl.drawArrays` 或者 `gl.drawElements` 函数被调用时，几乎所有的 WebGL 的其余函数都正在设置要被使用的全局变量或者状态。

我们知道，这个典型的 WebGL 程序基本上遵循这个结构。

在初始化时

- 创建所有的着色器和程序
- 创建缓冲区和上传顶点数据
- 创建质地和上传质地数据

在渲染时

- 清除并且设置视区和其他全局状态（启用深度测试，开启扑杀，等等）
- 对于你想要绘制的每一件事
  - 为你想要书写的程序调用 `gl.useProgram`
  - 为你想要绘制的东西设置属性
  - 对于每个属性调用 `gl.bindBuffer` , `gl.vertexAttribPointer` , `gl.enableVertexAttribArray` 函数
  - 为你想要绘制的东西设置制服
  - 为每一个制服调用 `gl.uniformXXX`
  - 调用 `gl.activeTexture` 和 `gl.bindTexture` 来为质地单元分配质地
  - 调用 `gl.drawArrays` 或者 `gl.drawElements`

这就是最基本的。怎样组织你的代码来完成这一任务取决于你。

一些事情诸如上传质地数据（甚至顶点数据）可能会异步的发生，因为你需要等待他们在网上下载完。

让我们来做一个简单的应用程序来绘制 3 种东西。一个立方体，一个球体和一个圆锥体。

我不会去详谈如何计算立方体，球体和圆锥体的数据。假设我们有函数来创建它们，然后我们返回[在之前篇章中介绍的 `bufferInfo` 对象](#)。

所以这里是代码。我们的着色器，与从我们的[角度看示例](#)的一个简单着色器相同，除了我们已经通过添加另外一个 `u_colorMult` 来增加顶点颜色。

```
// Passed in from the vertex shader.
varying vec4 v_color;

uniform vec4 u_colorMult;

void main() {
    gl_FragColor = v_color * u_colorMult;
}
```



在初始化时

```
// Our uniforms for each thing we want to draw
var sphereUniforms = {
  u_colorMult: [0.5, 1, 0.5, 1],
  u_matrix: makeIdentity(),
};
var cubeUniforms = {
  u_colorMult: [1, 0.5, 0.5, 1],
  u_matrix: makeIdentity(),
};
var coneUniforms = {
  u_colorMult: [0.5, 0.5, 1, 1],
  u_matrix: makeIdentity(),
};

// The translation for each object.
var sphereTranslation = [ 0, 0, 0];
var cubeTranslation   = [-40, 0, 0];
var coneTranslation   = [ 40, 0, 0];
```

在绘制时

```
var sphereXRotation = time;
var sphereYRotation = time;
var cubeXRotation   = -time;
var cubeYRotation   = time;
var coneXRotation   = time;
var coneYRotation   = -time;

// ----- Draw the sphere -----

gl.useProgram(programInfo.program);

// Setup all the needed attributes.
setBuffersAndAttributes(gl, programInfo.attribSetters, sphereBufferInfo);

sphereUniforms.u_matrix = computeMatrix(
  viewMatrix,
  projectionMatrix,
  sphereTranslation,
  sphereXRotation,
  sphereYRotation);

// Set the uniforms we just computed
```

```

setUniforms(programInfo.uniformSetters, sphereUniforms);

gl.drawArrays(gl.TRIANGLES, 0, sphereBufferInfo.numElements);

// ----- Draw the cube -----

// Setup all the needed attributes.
setBuffersAndAttributes(gl, programInfo.attribSetters, cubeBufferInfo);

cubeUniforms.u_matrix = computeMatrix(
viewMatrix,
projectionMatrix,
cubeTranslation,
cubeXRotation,
cubeYRotation);

// Set the uniforms we just computed
setUniforms(programInfo.uniformSetters, cubeUniforms);

gl.drawArrays(gl.TRIANGLES, 0, cubeBufferInfo.numElements);

// ----- Draw the cone -----

// Setup all the needed attributes.
setBuffersAndAttributes(gl, programInfo.attribSetters, coneBufferInfo);

coneUniforms.u_matrix = computeMatrix(
viewMatrix,
projectionMatrix,
coneTranslation,
coneXRotation,
coneYRotation);

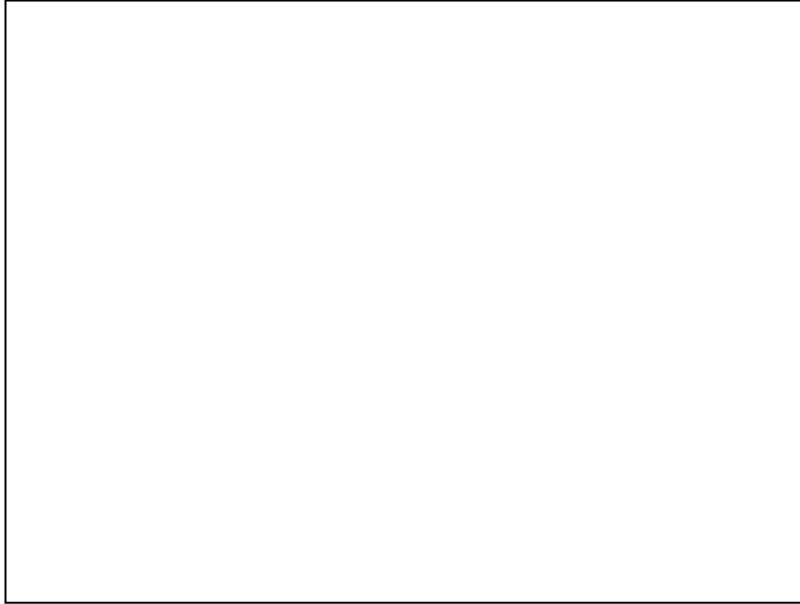
// Set the uniforms we just computed
setUniforms(programInfo.uniformSetters, coneUniforms);

gl.drawArrays(gl.TRIANGLES, 0, coneBufferInfo.numElements);

```

如下所示

## Draw multiple objects manually



需要注意的一件事情是，因为我们只有一个着色器程序，我们仅调用了 `gl.useProgram` 一次。如果我们有不同的着色器程序，你需要在使用每个程序之前调用 `gl.useProgram`。

这是另外一个值得去简化的地方。这里结合了 3 个主要的有效的东西。

1. 一个着色器程序（同时还有它的制服和属性 信息/设置）
2. 你想要绘制的东西的缓冲区和属性
3. 制服需要用给出的着色器来绘制你想要绘制的东西

所以，一个简单的简化可能会绘制出一个数组的东西，同时在这个数组中将 3 个东西放在一起。

```
var objectsToDraw = [  
  {  
    programInfo: programInfo,  
    bufferInfo: sphereBufferInfo,  
    uniforms: sphereUniforms,  
  },  
  {  
    programInfo: programInfo,  
    bufferInfo: cubeBufferInfo,  
    uniforms: cubeUniforms,  
  },  
  {  
    programInfo: programInfo,  
    bufferInfo: coneBufferInfo,  
    uniforms: coneUniforms,  
  },  
];
```

在绘制时，我们仍然需要更新矩阵

```
var sphereXRotation = time;
var sphereYRotation = time;
var cubeXRotation  = -time;
var cubeYRotation  = time;
var coneXRotation  = time;
var coneYRotation  = -time;

// Compute the matrices for each object.
sphereUniforms.u_matrix = computeMatrix(
viewMatrix,
projectionMatrix,
sphereTranslation,
sphereXRotation,
sphereYRotation);

cubeUniforms.u_matrix = computeMatrix(
viewMatrix,
projectionMatrix,
cubeTranslation,
cubeXRotation,
cubeYRotation);

coneUniforms.u_matrix = computeMatrix(
viewMatrix,
projectionMatrix,
coneTranslation,
coneXRotation,
coneYRotation);
```

但是这个绘制代码现在只是一个简单的循环

```
// ----- Draw the objects -----

objectsToDraw.forEach(function(object) {
  var programInfo = object.programInfo;
  var bufferInfo  = object.bufferInfo;

  gl.useProgram(programInfo.program);

  // Setup all the needed attributes.
  setBuffersAndAttributes(gl, programInfo.attribSetters, bufferInfo);

  // Set the uniforms.
```

```
setUniforms(programInfo.uniformSetters, object.uniforms);

// Draw
gl.drawArrays(gl.TRIANGLES, 0, bufferInfo.numElements);
});
```

这可以说是大多数 3D 引擎的主渲染循环都存在的。一些代码所在的地方或者是代码决定将什么放入 `objectsToDraw` 的列表中，基本上是这样。

### Draw multiple objects using a list



这里有几个基本的优化。如果这个我们想要绘制东西的程序与我们已经绘制东西的之前的程序一样，就不需要重新调用 `gl.useProgram` 了。同样，如果我们现在正在绘制的与我们之前已经绘制的东西有相同的形状 / 几何 / 顶点，就不需要再次设置上面的东西了。

所以，一个很简单的优化会与以下代码类似

```
var lastUsedProgramInfo = null;
var lastUsedBufferInfo = null;

objectsToDraw.forEach(function(object) {
  var programInfo = object.programInfo;
  var bufferInfo = object.bufferInfo;
  var bindBuffers = false;

  if (programInfo !== lastUsedProgramInfo) {
    lastUsedProgramInfo = programInfo;
    gl.useProgram(programInfo.program);

    // We have to rebind buffers when changing programs because we
    // only bind buffers the program uses. So if 2 programs use the same
```

```

// bufferInfo but the 1st one uses only positions the when the
// we switch to the 2nd one some of the attributes will not be on.
bindBuffers = true;
}

// Setup all the needed attributes.
if (bindBuffers || bufferInfo != lastUsedBufferInfo) {
lastUsedBufferInfo = bufferInfo;
setBuffersAndAttributes(gl, programInfo.attribSetters, bufferInfo);
}

// Set the uniforms.
setUniforms(programInfo.uniformSetters, object.uniforms);

// Draw
gl.drawArrays(gl.TRIANGLES, 0, bufferInfo.numElements);
});

```

这次让我们来绘制更多的对象。与之前的仅仅 3 个东西不同，让我们做一系列的东西来绘制更大的东西。

```

// put the shapes in an array so it's easy to pick them at random
var shapes = [
  sphereBufferInfo,
  cubeBufferInfo,
  coneBufferInfo,
];

// make 2 lists of objects, one of stuff to draw, one to manipulate.
var objectsToDraw = [];
var objects = [];

// Uniforms for each object.
var numObjects = 200;
for (var ii = 0; ii < numObjects; ++ii) {
  // pick a shape
  var bufferInfo = shapes[rand(0, shapes.length) | 0];

  // make an object.
  var object = {
uniforms: {
  u_colorMult: [rand(0, 1), rand(0, 1), rand(0, 1), 1],
  u_matrix: makeIdentity(),
},
translation: [rand(-100, 100), rand(-100, 100), rand(-150, -50)],
xRotationSpeed: rand(0.8, 1.2),
yRotationSpeed: rand(0.8, 1.2),

```

```

};
objects.push(object);

// Add it to the list of things to draw.
objectsToDraw.push({
programInfo: programInfo,
bufferInfo: bufferInfo,
uniforms: object.uniforms,
});
}

```

在渲染时

```

// Compute the matrices for each object.
objects.forEach(function(object) {
  object.uniforms.u_matrix = computeMatrix(
    viewMatrix,
    projectionMatrix,
    object.translation,
    object.xRotationSpeed * time,
    object.yRotationSpeed * time);
});

```

然后使用上面的循环绘制对象

### Draw multiple objects by list with optimization checks

你也可以通过 `programInfo` 和 / 或者 `bufferInfo` 来对列表进行排序，以便优化开始的更加频繁。大多数游戏引擎都是这样做。不幸的是它不是那么简单。如果你现在正在绘制的任何东西都不透明，然后你可以只排序。但是，一旦你需要绘制半透明的东西，你就需要以特定的顺序来绘制它们。大多数 3D 引擎都通过有 2 个或者更多的要绘制的对象的列表来处理这个问题。不透明的东西有一个列表。透明的东西有另外一个列表。不透明的列表

按程序和几何来排序。透明的列表按深度排序。对于其他东西，诸如覆盖或后期处理效果，还会有其他单独的列表。

[这里有一个已经排好序的例子](#)。在我的机器上，我得到了未排序的 ~31 fps 和排好序的 ~37.发现几乎增长了 20 %。但是，它是在最糟糕的案例和最好的案例相比较下，大多数的程序将会做的更多，因此，它可能对于所有情况来说不值得考虑，但是最特别的案例值得考虑。

注意到你不可能仅仅使用任何着色器来仅仅绘制任何几何是非常重要的。例如，一个需要法线的着色器在没有法线的几何情况下将不会起作用。同样，一个组要质地的着色器在没有质地时将不会工作。

选择一个像 [Three.js](#) 的 3D 库是很重要的，这是众多原因之一，因为它会为你处理所有这些东西。你创建了一些几何，你告诉 three.js 你想让它怎样呈现，它会在运行时产生着色器来处理你需要的东西。几乎所有的 3D 引擎都将它们从 Unity3D 到虚幻的 Crytek 源。一些离线就可以生成它们，但是最重要的事是意识到是它们生成了着色器。

当然，你正在读这些文章的原因，是你想要知道接下来将会发生什么。你自己写任何东西都是非常好且有趣的。意识到 [WebGL 是超级低水平的](#)是非常重要的，因此如果你想要自己做，这里有许多你可以做的工作，这经常包括写一个着色器生成器，因为不同的功能往往需要不同的着色器。

你将会注意到我并没有在循环中放置 `computeMatrix`。那是因为呈现应该与计算矩阵分开。从[场景图和我们另一篇文章中读到的内容](#)，计算矩阵是非常常见的。

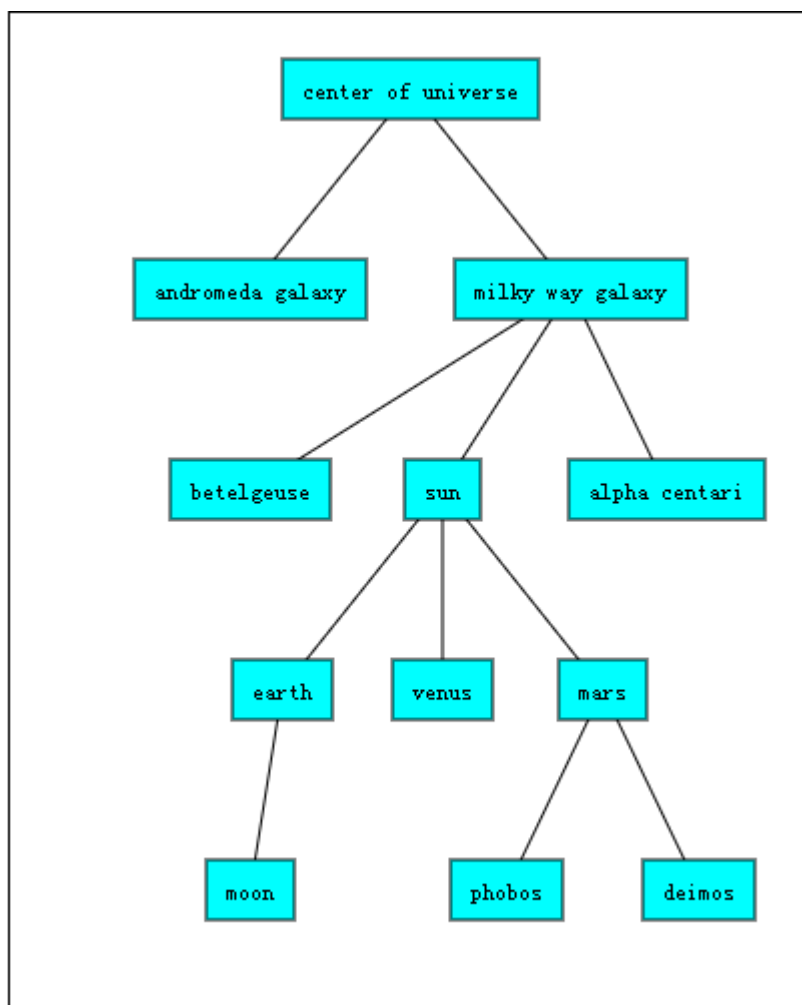
现在，我们已经有了一个绘制多对象的框架，[让我们来绘制一些文本](#)。



## WebGL 场景图

我很肯定一些 CS 大师或者图形大师会给我们讲很多东西，但是...一个场景图通常是一个树结构，在这个树结构中的每个节点都生成一个矩阵...嗯，这并不是一个非常有用的定义。也许讲一些例子会非常有用。

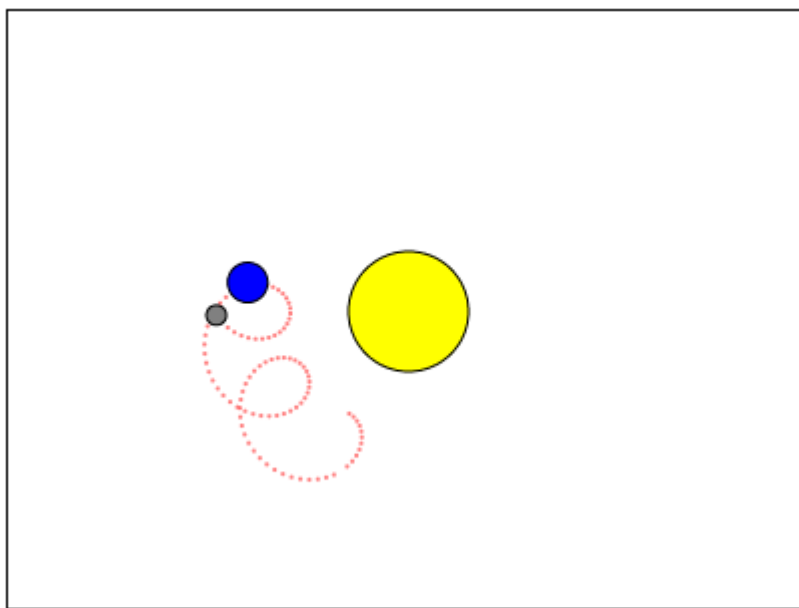
大多数的 3D 引擎都使用一个场景图。你在场景图中放置你想要在场景图中出现的东西。引擎然后按场景图行进，同时计算出需要绘制的一系列东西。场景图都是有层次感的，例如，如果你想要去制作一个宇宙模拟图，你可能需要一个图与下面所示的图相似



一个场景图的意义是什么？一个场景图的 #1 特点是它为矩阵提供了一个父母子女关系，正如[我们在二维矩阵数学中讨论的](#)。因此，例如在一个简单的宇宙中（但是不是实际的）模拟星星（孩子），随着它们的星系移动（父母）。同样，一个月亮（孩子）随着行星移动，如果你移动了地球，月亮会跟着它一起移动。如果你移动一个星系，在这个星系中的所有的星星也会随着它一起移动。在上面的图中拖动名称，希望你可以看到它们之间的关系。

如果你回到[二维矩阵数学](#)，你可能会想起我们将大量矩阵相乘来达到转化，旋转和缩放对象。一个场景图提供了一个结构来帮助决定要将哪个矩阵数学应用到对象上。

通常，在一个场景图中的每个节点都表示一个局部空间。给出了正确的矩阵数学，在这个局部空间的任何东西都可以忽略在他上面的任何东西。用来说明同一件事的另一种方式是月亮只关心绕地球轨道运行。它不关心绕太阳的轨道运行。没有场景图结构，你需要做更多的复杂数学，来计算怎样才可以得到月亮绕太阳的轨道，因为它绕太阳的轨道看起来像这样



使用场景图，你可以将月球看做是地球的孩子，然后简单的绕地球转动。场景图很注意地球围绕太阳转的事实。它是通过节点和它走的矩阵相乘来完成的。

```
worldMatrix = greatGrandParent * grandParent * parent * self(localMatrix)
```

在具体的条款中，我们的宇宙模型可能是

```
worldMatrixForMoon = galaxyMatrix * starMatrix * planetMatrix * moonMatrix;
```

我们可以使用一个有效的递归函数来非常简单的完成这些

```
function computeWorldMatrix(currentNode, parentWorldMatrix) {
  // compute our world matrix by multiplying our local matrix with
  // our parent's world matrix.
  var worldMatrix = matrixMultiply(currentNode.localMatrix, parentWorldMatrix);

  // now do the same for all of our children
  currentNode.children.forEach(function(child) {
    computeWorldMatrix(child, worldMatrix);
  });
}
```

```
});
}
```

这将会给我们引进一些在 3D 场景图中非常常见的术语。

- `localMatrix`：当前节点的本地矩阵。它在原点转换它和在局部空间它的孩子。
- `worldMatrix`：对于给定的节点，它需要获取那个节点的局部空间的东西，同时将它转换到场景图的根节点的空间。或者，换句话说，将它置于世界中。如果我们为月球计算世界矩阵，我们将会得到上面我们看到的轨道。

制作场景图非常简单。让我们定义一个简单的 `节点` 对象。还有无数个方式可以组织场景图，我不确定哪一种方式是最好的。最常见的是有一个可以选择绘制东西的字段。

```
var node = {
  localMatrix: ..., // the "local" matrix for this node
  worldMatrix: ..., // the "world" matrix for this node
  children: [], // array of children
  thingToDraw: ??, // thing to draw at this node
};
```

让我们来做一个太阳系场景图。我不准备使用花式纹理或者类似的东西，因为它会使例子变的混乱。首先让我们来制作一些功能来帮助管理这些节点。首先我们将做一个节点类

```
var Node = function() {
  this.children = [];
  this.localMatrix = makeldentity();
  this.worldMatrix = makeldentity();
};
```

我们给出一种设置一个节点的父母的方式

```
Node.prototype.setParent = function(parent) {
  // remove us from our parent
  if (this.parent) {
    var ndx = this.parent.children.indexOf(this);
    if (ndx >= 0) {
      this.parent.children.splice(ndx, 1);
    }
  }

  // Add us to our new parent
  if (parent) {
    parent.children.append(this);
  }
}
```

```
this.parent = parent;
};
```

这里，这里的代码是从基于它们的父子关系的本地矩阵计算世界矩阵。如果我们从父母和递归访问它孩子开始，我们可以计算它们的世界矩阵。

```
Node.prototype.updateWorldMatrix = function(parentWorldMatrix) {
  if (parentWorldMatrix) {
    // a matrix was passed in so do the math and
    // store the result in `this.worldMatrix`.
    matrixMultiply(this.localMatrix, parentWorldMatrix, this.worldMatrix);
  } else {
    // no matrix was passed in so just copy.
    copyMatrix(this.localMatrix, this.worldMatrix);
  }

  // now process all the children
  var worldMatrix = this.worldMatrix;
  this.children.forEach(function(child) {
    child.updateWorldMatrix(worldMatrix);
  });
};
```

让我们仅仅做太阳，地球，月亮，来保持场景图简单。当然我们会使用假的距离，来使东西适合屏幕。我们将只使用一个单球体模型，然后太阳为淡黄色，地球为蓝 - 淡绿色，月球为淡灰色。如果你对 `drawInfo`，`bufferInfo` 和 `programInfo` 并不熟悉，你可以[查看前一篇文章](#)。

```
// Let's make all the nodes
var sunNode = new Node();
sunNode.localMatrix = makeTranslation(0, 0, 0); // sun at the center
sunNode.drawInfo = {
  uniforms: {
    u_colorOffset: [0.6, 0.6, 0, 1], // yellow
    u_colorMult: [0.4, 0.4, 0, 1],
  },
  programInfo: programInfo,
  bufferInfo: sphereBufferInfo,
};

var earthNode = new Node();
earthNode.localMatrix = makeTranslation(100, 0, 0); // earth 100 units from the sun
earthNode.drawInfo = {
  uniforms: {
    u_colorOffset: [0.2, 0.5, 0.8, 1], // blue-green
    u_colorMult: [0.8, 0.5, 0.2, 1],
```

```

},
programInfo: programInfo,
bufferInfo: sphereBufferInfo,
};

var moonNode = new Node();
moonNode.localMatrix = makeTranslation(20, 0, 0); // moon 20 units from the earth
moonNode.drawInfo = {
  uniforms: {
    u_colorOffset: [0.6, 0.6, 0.6, 1], // gray
    u_colorMult: [0.1, 0.1, 0.1, 1],
  },
  programInfo: programInfo,
  bufferInfo: sphereBufferInfo,
};

```

现在我们已经得到了节点，让我们来连接它们。

```

// connect the celestial objects
moonNode.setParent(earthNode);
earthNode.setParent(sunNode);

```

我们会再一次做一个对象的列表和一个要绘制的对象的列表。

```

var objects = [
  sunNode,
  earthNode,
  moonNode,
];

var objectsToDraw = [
  sunNode.drawInfo,
  earthNode.drawInfo,
  moonNode.drawInfo,
];

```

在渲染时，我们将会通过稍微旋转它来更新每一个对象的本地矩阵。

```

// update the local matrices for each object.
matrixMultiply(sunNode.localMatrix, makeYRotation(0.01), sunNode.localMatrix);
matrixMultiply(earthNode.localMatrix, makeYRotation(0.01), earthNode.localMatrix);
matrixMultiply(moonNode.localMatrix, makeYRotation(0.01), moonNode.localMatrix);

```

现在，本地矩阵都更新了，我们会更新所有的世界矩阵。

```

sunNode.updateWorldMatrix();

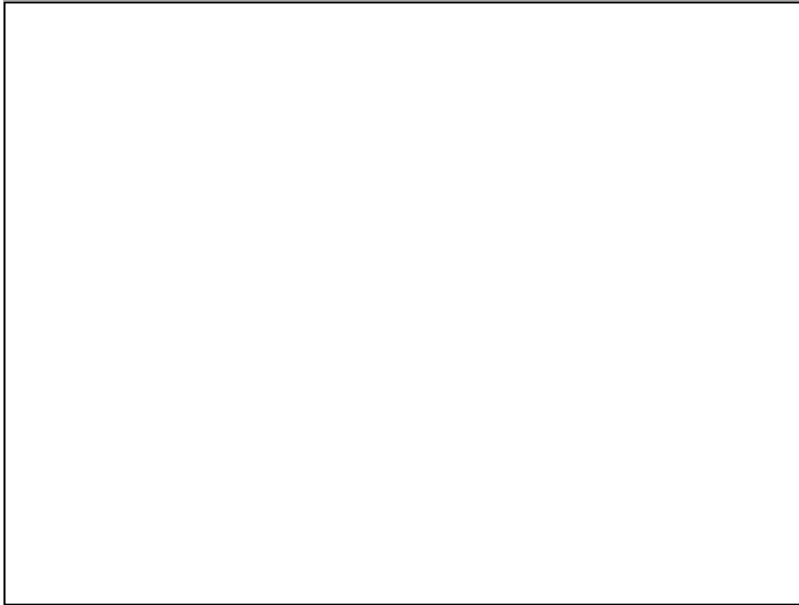
```

最后，我们有了世界矩阵，我们需要将它们相乘来为每个对象获取一个[世界观投射矩阵](#)。

```
// Compute all the matrices for rendering
objects.forEach(function(object) {
  object.drawInfo.uniforms.u_matrix = matrixMultiply(object.worldMatrix, viewProjectionMatrix);
});
```

渲染是[我们在上一篇文章中看到的相同的循环](#)。

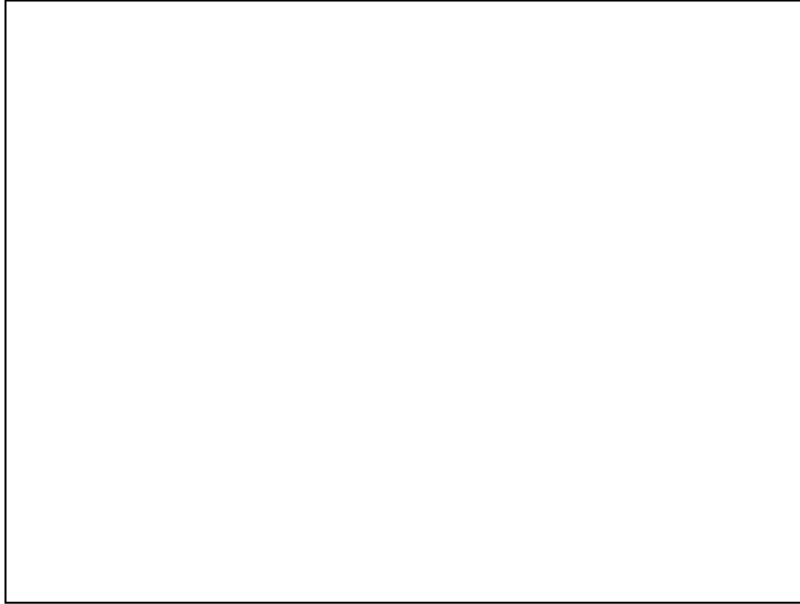
### Simulating a solar system using a scene graph.



你将会注意到所有的行星都是一样的尺寸。我们试着让地球更大点。

```
earthNode.localMatrix = matrixMultiply(
  makeScale(2, 2, 2), // make the earth twice as large
  makeTranslation(100, 0, 0)); // earth 100 units from the sun
```

### Simulating a solar system using a scene graph.



哦。月亮也越来越大。为了解决这个问题，我们可以手动的缩小月亮。但是一个更好的解决方法是在我们的场景中增加更多的节点。而不仅仅是如下图所示。

```
sun
|
earth
|
moon
```

我们将改变它为

```
solarSystem
||
| sun
|
earthOrbit
||
| earth
|
moonOrbit
|
moon
```

这将会使地球围绕太阳系旋转，但是我们可以单独的旋转和缩放太阳，它不会影响地球。同样，地球与月球可以单独旋转。让我们给 `太阳系`，`地球轨道` 和 `月球轨道` 设置更多的节点。

```
var solarSystemNode = new Node();
var earthOrbitNode = new Node();
earthOrbitNode.localMatrix = makeTranslation(100, 0, 0); // earth orbit 100 units from the sun
```

```
var moonOrbitNode = new Node();
moonOrbitNode.localMatrix = makeTranslation(20, 0, 0); // moon 20 units from the earth
```

这些轨道距离已经从旧的节点移除

```
var earthNode = new Node();
earthNode.localMatrix = matrixMultiply(
  makeScale(2, 2, 2), // make the earth twice as large
  makeTranslation(100, 0, 0)); // earth 100 units from the sun
earthNode.localMatrix = makeScale(2, 2, 2); // make the earth twice as large

var moonNode = new Node();
moonNode.localMatrix = makeTranslation(20, 0, 0); // moon 20 units from the earth
```

现在连接它们，如下所示

```
// connect the celestial objects
sunNode.setParent(solarSystemNode);
earthOrbitNode.setParent(solarSystemNode);
earthNode.setParent(earthOrbitNode);
moonOrbitNode.setParent(earthOrbitNode);
moonNode.setParent(moonOrbitNode);
```

同时，我们只需要更新轨道

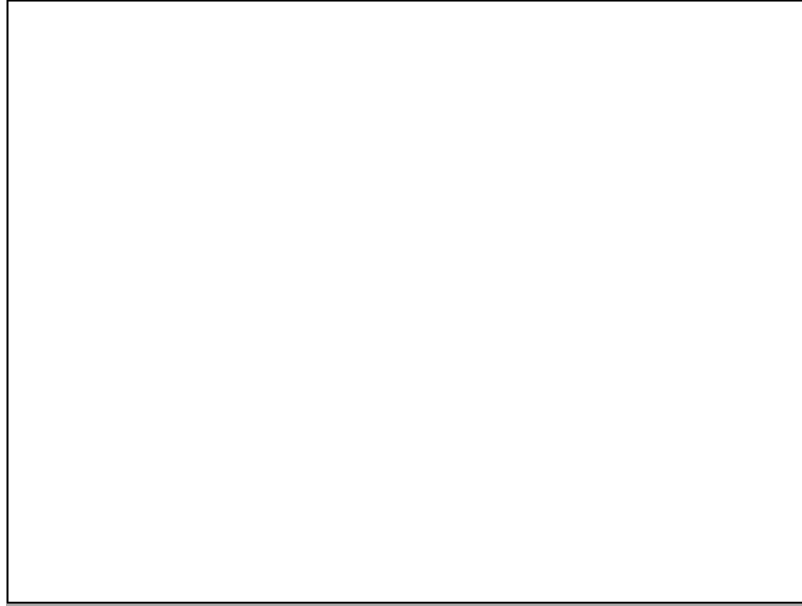
```
// update the local matrices for each object.
matrixMultiply(sunNode.localMatrix, makeYRotation(0.01), sunNode.localMatrix);
matrixMultiply(earthNode.localMatrix, makeYRotation(0.01), earthNode.localMatrix);
matrixMultiply(moonNode.localMatrix, makeYRotation(0.01), moonNode.localMatrix);
matrixMultiply(earthOrbitNode.localMatrix, makeYRotation(0.01), earthOrbitNode.localMatrix);
matrixMultiply(moonOrbitNode.localMatrix, makeYRotation(0.01), moonOrbitNode.localMatrix);

// Update all world matrices in the scene graph
sunNode.updateWorldMatrix();
solarSystemNode.updateWorldMatrix();
```

现在你可以看到地球是两倍大小，而月球不会。



## Simulating a solar system using a scene graph.



你可能还会注意到太阳和地球不再旋转到位。它们现在是无关的。

让我们调整更多的东西。

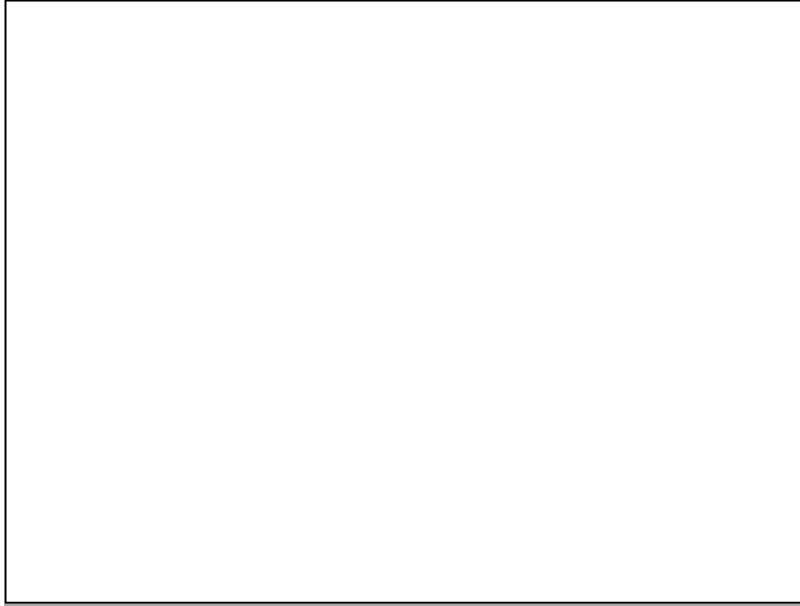
```
sunNode.localMatrix = makeTranslation(0, 0, 0); // sun at the center
sunNode.localMatrix = makeScale(5, 5, 5);

...

moonNode.localMatrix = makeScale(0.4, 0.4, 0.4);

...
// update the local matrices for each object.
matrixMultiply(earthOrbitNode.localMatrix, makeYRotation(0.01), earthOrbitNode.localMatrix);
matrixMultiply(moonOrbitNode.localMatrix, makeYRotation(0.01), moonOrbitNode.localMatrix);
// spin the sun
matrixMultiply(sunNode.localMatrix, makeYRotation(0.005), sunNode.localMatrix);
// spin the earth
matrixMultiply(earthNode.localMatrix, makeYRotation(0.05), earthNode.localMatrix);
// spin the moon
matrixMultiply(monNode.localMatrix, makeYRotation(-0.01), moonNode.localMatrix);
```

## Simulating a solar system using a scene graph.



目前我们有一个 `localMatrix`，我们在每一帧都修改它。但是有一个问题，即在每一帧中我们数学都将收集一点错误。有许多可以解决这种被称为邻位的正常化矩阵的数学的方式，但是，甚至是它都不总是奏效。例如，让我们想象我们缩减零。让我们为一个值 `x` 这样做。

```
x = 246; // frame #0, x = 246

scale = 1;
x = x * scale // frame #1, x = 246

scale = 0.5;
x = x * scale // frame #2, x = 123

scale = 0;
x = x * scale // frame #3, x = 0

scale = 0.5;
x = x * scale // frame #4, x = 0 OOPS!

scale = 1;
x = x * scale // frame #5, x = 0 OOPS!
```

我们失去了我们的值。我们可以通过添加其他一些从其他值更新矩阵的类来解决它。让我们通过拥有一个 `source` 来改变 `Node` 的定义。如果它存在，我们会要求 `source` 给出我们一个本地矩阵。

```

var Node = function(source) {
  this.children = [];
  this.localMatrix = makeIdentity();
  this.worldMatrix = makeIdentity();
  this.source = source;
};

Node.prototype.updateWorldMatrix = function(matrix) {

  var source = this.source;
  if (source) {
    source.getMatrix(this.localMatrix);
  }

  ...

```

现在我们来创建一个源。一个常见的源是那些提供转化，旋转和缩放的，如下所示。

```

var TRS = function() {
  this.translation = [0, 0, 0];
  this.rotation = [0, 0, 0];
  this.scale = [1, 1, 1];
};

TRS.prototype.getMatrix = function(dst) {
  dst = dst || new Float32Array(16);
  var t = this.translation;
  var r = this.rotation;
  var s = this.scale;

  // compute a matrix from translation, rotation, and scale
  makeTranslation(t[0], t[1], t[2], dst);
  matrixMultiply(makeXRotation(r[0]), dst, dst);
  matrixMultiply(makeYRotation(r[1]), dst, dst);
  matrixMultiply(makeZRotation(r[2]), dst, dst);
  matrixMultiply(makeScale(s[0], s[1], s[2]), dst, dst);
  return dst;
};

```

我们可以像下面一样使用它

```

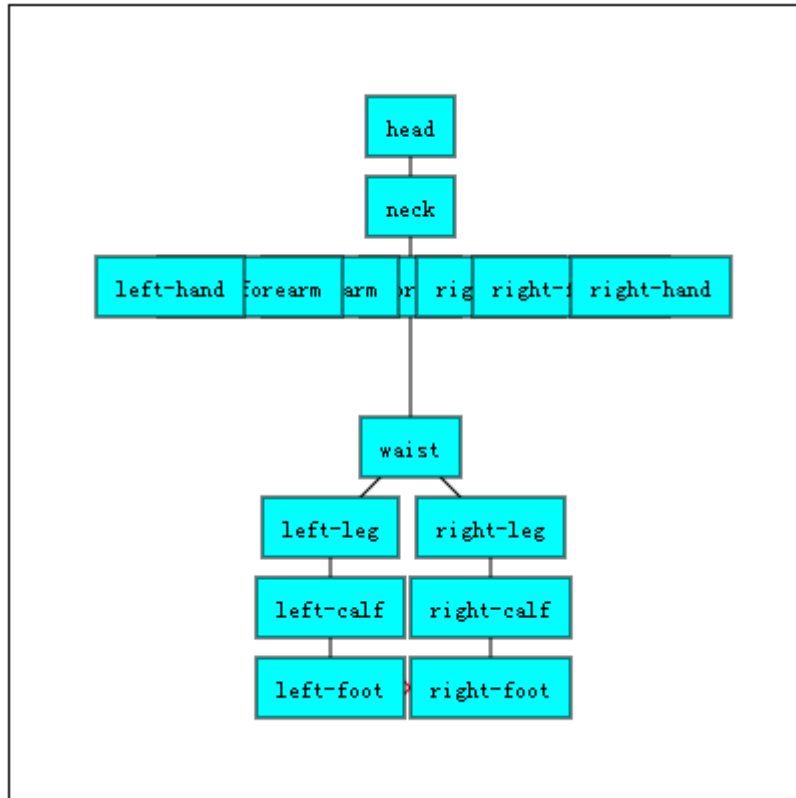
// at init time making a node with a source
var someTRS = new TRS();
var someNode = new Node(someTRS);

// at render time
someTRS.rotation[2] += elapsedTime;

```

现在没有问题了，因为我们每次都重新创建矩阵。

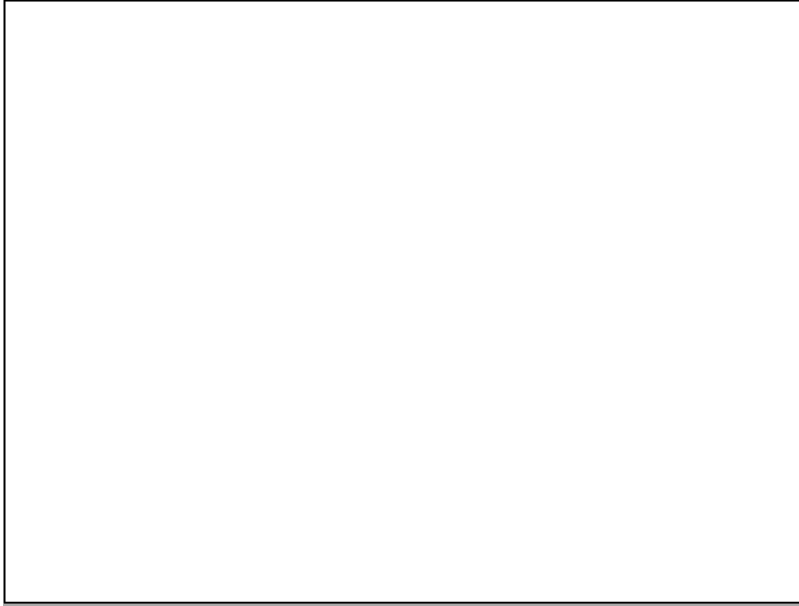
你可能会想，我没做一个太阳系，所以这样的意义何在？好吧，如果你想要去动画一个人，你可能会有一个跟下面所示一样的场景图。



为手指和脚趾添加多少关节全部取决于你。你有的关节越多，它用于计算动画的力量越多，同时它为所有的关节提供的动画数据越多。像虚拟战斗机的旧游戏大约有 15 个关节。在 2000 年代早期至中期，游戏有 30 到 70 个关节。如果你为每个手都设置关节，在每个手中至少有 20 个，所以两只手是 40 个关节。许多想要动画手的游戏都把大拇指处理为一个，其他的四个作为一个大的手指处理，以节省时间（所有的 CPU/GPU 和艺术家的时间）和内存。

不管怎样，这是一个我组件在一起的块人。它为上面提到的每个节点使用 `TRS` 源。艺术程序员和动画程序员万岁。

### A human hierarchy using a scene graph



如果你查看一下，几乎所有的 3D 图书馆，你都会发现一个与下图类似的场景图。



T



文本



## WebGL 文本 HTML

---

“在 WebGL 中如何绘制文本”是一个我们常见的问题。那么第一件事就是我们要问自己绘制文本的目的何在。现在有一个浏览器，浏览器用来显示文本。所以你的第一个答案应该是如何使用 HTML 来显示文本。

让我们从最简单的例子开始：你只是想在你的 WebGL 上绘制一些文本。我们可以称之为一个文本覆盖。基本上这是停留在同一个位置的文本。

简单的方法是构造一些 HTML 元素，使用 CSS 使它们重叠。

例如：先构造一个容器，把画布和一些 HTML 元素重叠放置在容器内部。

```
<div class="container">
  <canvas id="canvas" width="400" height="300"></canvas>
  <div id="overlay">
    <div>Time: <span id="time"></span></div>
    <div>Angle: <span id="angle"></span></div>
  </div>
</div>
```

接下来设置 CSS，以达到画布和 HTML 重叠的目的。

```
.container {
position: relative;
}
#overlay {
position: absolute;
left: 10px;
top: 10px;
}
```

现在按照初始化和创建时间查找这些元素，或者查找你想要改变的区域。

```
// look up the elements we want to affect
var timeElement = document.getElementById("time");
var angleElement = document.getElementById("angle");

// Create text nodes to save some time for the browser.
var timeNode = document.createTextNode("");
var angleNode = document.createTextNode("");

// Add those text nodes where they need to go
```

```
timeElement.appendChild(timeNode);
angleElement.appendChild(angleNode);
```

最后在渲染时更新节点。

```
function drawScene() {
  ...


  // convert rotation from radians to degrees
  var angle = radToDeg(rotation[1]);

  // only report 0 - 360
  angle = angle % 360;

  // set the nodes
  angleNode.nodeValue = angle.toFixed(0); // no decimal place
  timeNode.nodeValue = clock.toFixed(2); // 2 decimal places
```

这里有一个例子：

### HTML Text Overlay



Time:  
Angle:

注意为了我想改变的部分，我是如何把 spans 置入特殊的 div 内的。在这里我做一个假设，这比只使用 div 而没有 spans 速度要快，类似的有：

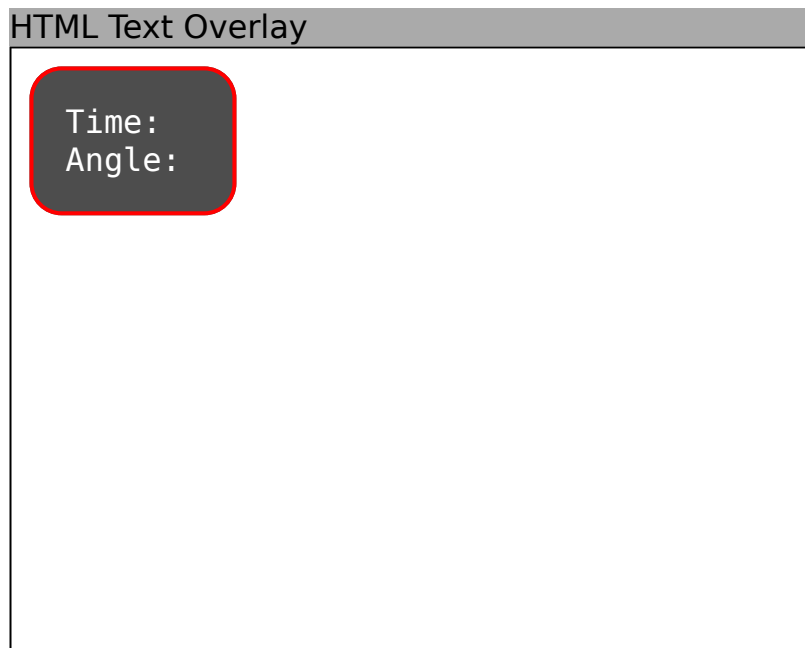
```
timeNode.value = "Time " + clock.toFixed(2);
```

另外，我们可以使用文本节点，通过调用 `node = document.createTextNode()` 和 `laterNode.node = someMsg`。我们也可以使用 `someElement.innerHTML = someHTML`。这将会更加灵活，虽然这样可能会稍微慢一些，但是您却可以插入任意的 HTML 字符串，因为你每次设置它，浏览器都不得不创建和销毁节点。这对你来说更加方便。



不采用叠加技术很重要的一点是，WebGL 在浏览器中运行。要记得在适当的时候使用浏览器的特征。大量的 OpenGL 程序员习惯于从一开始就 100% 靠他们自己实现应用的每一部分自己，因为 WebGL 要在一个已经有很多特征的浏览器上运行。使用它们有很多好处。例如使用 CSS 样式，你可以很容易就覆盖一个有趣的风格。

这里有一个相同的例子，但是添加了一些风格。背景是圆形的，字母的周围有光晕。这儿有一个红色的边界。你可以使用 HTML 免费得到所有。



我们要讨论的下一件最常见的事情是文本相对于你渲染的东西的位置。我们也可以在 HTML 中做到这一点。

在本例中，我们将再次构造一个画布容器和另一个活动的 HTML 容器。

```
<div class="container">
  <canvas id="canvas" width="400" height="300"></canvas>
  <div id="divcontainer"></div>
</div>
```

我们将设置 CSS

```
.container {
  position: relative;
  overflow: none;
}

#divcontainer {
  position: absolute;
  left: 0px;
  top: 0px;
  width: 400px;
```

```

height: 300px;
z-index: 10;
overflow: hidden;

}

.floating-div {
position: absolute;
}

```

相对于第一个父类位置 `position: relative` 或者 `position: absolute` 风格, `position: absolute`; 部分使 `#divcontainer` 放置于绝对位置。在本例中, 画布和 `#divcontainer` 都在容器内。

`left: 0px; top: 0px` 使 `#divcontainer` 结合一切。`z-index: 10` 使它浮在画布上。`overflow: hidden` 让它的子类被剪除。

最后, `floating-div` 将使用我们创建的可移式 `div`。

现在我们需要查找 `div` 容器, 创建一个 `div`, 将 `div` 附加到容器。

```

// look up the divcontainer
var divContainerElement = document.getElementById("divcontainer");

// make the div
var div = document.createElement("div");

// assign it a CSS class
div.className = "floating-div";

// make a text node for its content
var textNode = document.createTextNode("");
div.appendChild(textNode);

// add it to the divcontainer
divContainerElement.appendChild(div);

```

现在, 我们可以通过设置它的风格定位 `div`。

```

div.style.left = Math.floor(x) + "px";
div.style.top = Math.floor(y) + "px";
textNode.nodeValue = clock.toFixed(2);

```

下面是一个例子, 我们只需要限制 `div` 的边界。

## HTML Text Divs

下一步，我们想在 3D 场景中设计它相对于某些事物的位置。我们该如何做？当我们[透视投影覆盖](#)，我们如何做实际上就是我们请求 GPU 如何做。

通过这个例子我们学习了如何使用模型，如何复制它们，以及如何应用一个投影模型将他们转换成 clip space。然后我们讨论着色器的内容，它在本地空间复制模型，并将其转换成 clip space。我们也可以在 JavaScript 中做所有的这些。然后我们可以增加 clip space(-1 到 +1) 到像素和使用 div 位置。

```
gl.drawArrays(...);

// We just got through computing a matrix to draw our
// F in 3D.

// choose a point in the local space of the 'F'.
// X Y Z W
var point = [100, 0, 0, 1]; // this is the front top right corner

// compute a clip space position
// using the matrix we computed for the F
var clipSpace = matrixVectorMultiply(point, matrix);

// divide X and Y by W just like the GPU does.
clipSpace[0] /= clipSpace[3];
clipSpace[1] /= clipSpace[3];

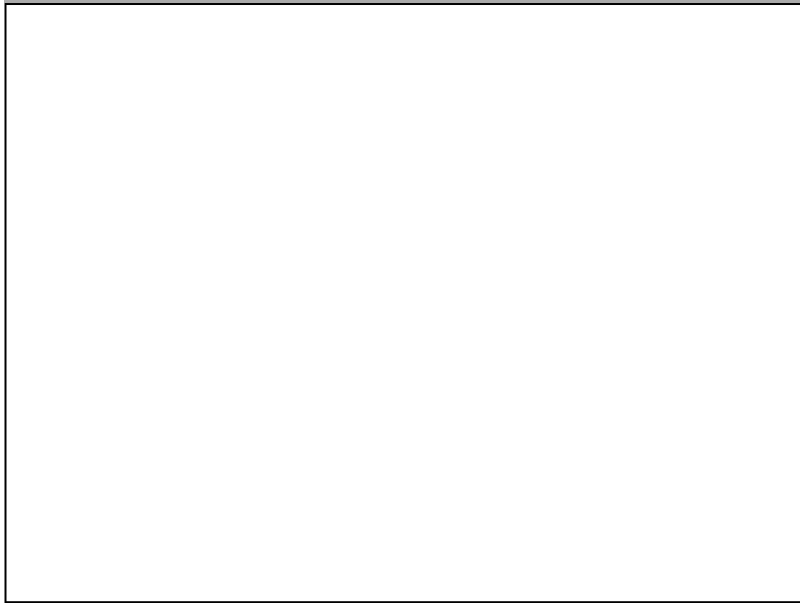
// convert from clip space to pixels
var pixelX = (clipSpace[0] * 0.5 + 0.5) * gl.canvas.width;
var pixelY = (clipSpace[1] * -0.5 + 0.5) * gl.canvas.height;

// position the div
```

```
div.style.left = Math.floor(pixelX) + "px";  
div.style.top = Math.floor(pixelY) + "px";  
textNode.nodeValue = clock.toFixed(2);
```

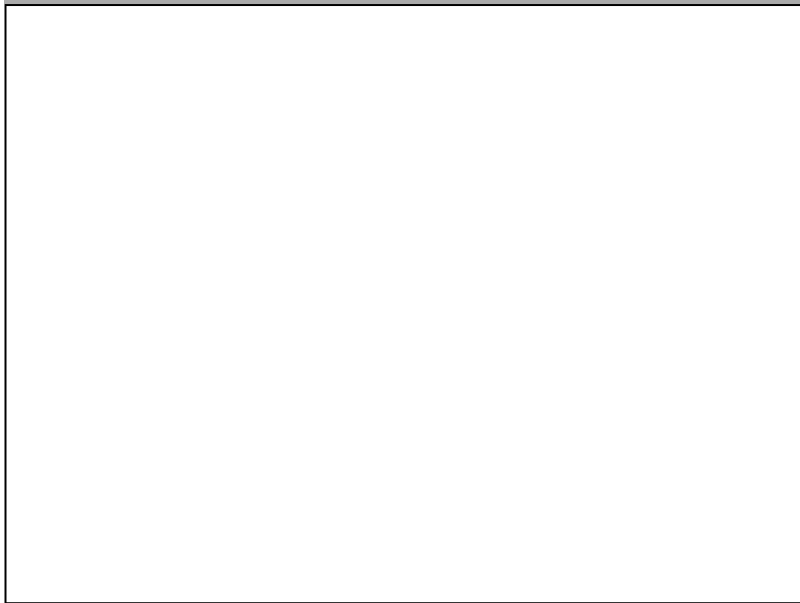
wahlah, 我们 div 的左上角和 F 的右上角完全符合。

### HTML Text Divs



当然, 如果你想要更多的文本构造更多的 div, 如下:

### HTML Text Divs



你可以查看最后一个例子的源代码来观察细节。一个重要的点是我猜测从 DOM 创建、添加、删除 HTML 元素是缓慢的, 所以上面的示例用来创建它们, 将它们保留在周围。它将任何未使用的都隐藏起来, 而不是把他们从 DOM 删除。你必须明确知道是否可以更快。这只是我选择的方法。

希望这已经清楚地说明了如何使用 HTML 制造文本。[接下来, 我们将介绍如何使用 Canvas2D 制造文本。](#)

问题? [可以在 stackoverflow 提问](#)。问题/缺陷? [可以在 github 创建一个话题](#)。

## WebGL 文本 Canvas 2D

相对于使用 HTML 元素制作文本我们还可以使用另一种使用 2D 上下文的画布。不需要分析，我们就可以做一个猜测，这将比使用 DOM 快。当然也会变得相对不灵活。你可能不能得到所有的 CSS 样式。但是，这儿没有 HTML 元素可以创建和跟踪。

和前边其他的例子类似，让我们来构造一个容器，但这一次我们将在其中放置两个画布。

```
<div class="container">
  <canvas id="canvas" width="400" height="300"></canvas>
  <canvas id="text" width="400" height="300"></canvas>
</div>
```

接下来设置 CSS，以使画布和 HTML 重叠

```
.container {
position: relative;
}

#text {
position: absolute;
left: 0px;
top: 0px;
z-index: 10;
}
```

现在按照初始化时间查找文本画布，并为之创建一个 2D 上下文。

```
// look up the text canvas.
var textCanvas = document.getElementById("text");

// make a 2D context for it
var ctx = textCanvas.getContext("2d");
```

当绘图时，就像 WebGL，我们需要清除 2d 画布的每一帧。

```
function drawScene() {
...

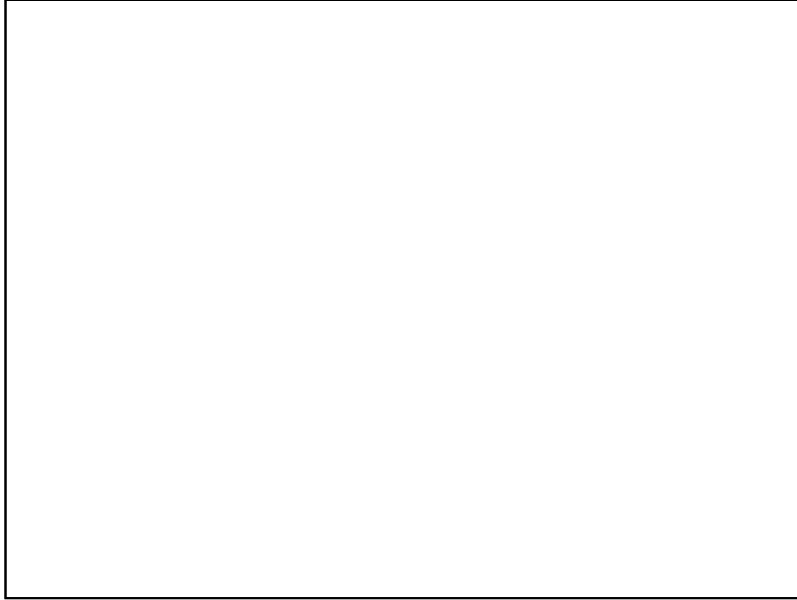
// Clear the 2D canvas
ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
```

然后我们就调用 `fillText` 绘制文本

```
ctx.fillText(someMsg, pixelX, pixelY);
```

下面有一个例子：

### HTML Text Canvas2D



为什么这个文本更小呢？因为这是 canvas2d 默认的尺寸。如果你想要其它的尺寸，可以[查看 canvas2d api](#)。

使用 canvas2d 的另一个原因是用它很容易绘制其他的事物。例如让我们来添加一个箭头：

```
// draw an arrow and text.

// save all the canvas settings
ctx.save();

// translate the canvas origin so 0, 0 is at
// the top front right corner of our F
ctx.translate(pixelX, pixelY);

// draw an arrow
ctx.beginPath();
ctx.moveTo(10, 5);
ctx.lineTo(0, 0);
ctx.lineTo(5, 10);
ctx.moveTo(0, 0);
ctx.lineTo(15, 15);
ctx.stroke();

// draw the text.
ctx.fillText(someMessage, 20, 20);
```

```
// restore the canvas to its old settings.  
ctx.restore();
```

这里我们利用 canvas2d 的翻译功能，所以画箭头时，我们不需要做任何额外的工作。我们只是假装在原点开始绘制，翻译负责移动原点到 F 的角落。

## HTML Text Canvas2D



封面使用了 2D 画布。[查看 canvas2d API](#) 的更多想法。[接下来，我们将实际在 WebGL 呈现文本。](#)

问题？[可以再 stackoverflow 提问。](#)

问题/缺陷？[可以在 github 上创建一个话题。](#)



## WebGL 文本 纹理

在上一篇文章中我们学习了在 [WebGL 场景中如何使用一个 2D 画布绘制文本](#)。这个技术可以工作且很容易做到，但它有一个限制，即文本不能被其他的 3D 对象遮盖。要做到这一点，我们实际上需要在 WebGL 中绘制文本。

最简单的方法是绘制带有文本的纹理。例如你可以使用 photoshop 或其他绘画程序，来绘制带有文本的一些图像。



然后我们构造一些平面几何并显示它。这实际上是一些游戏中构造所有的文本的方式。例如 Locoroco 只有大约 270 个字符串。它本地化成 17 种语言。我们有一个包含所有语言的 Excel 表和一个脚本，该脚本将启动 Photoshop 并生成纹理，每个纹理都对应一种语言里的一个消息。

当然你也可以在运行时生成纹理。因为在浏览器中 WebGL 是依靠画布 2d api 来帮助生成纹理的。

我们来看[上一篇文章](#)的例子，在其中添加一个函数：用文本填补一个 2D 画布。

```
var textCtx = document.createElement("canvas").getContext("2d");

// Puts text in center of canvas.
function makeTextCanvas(text, width, height) {
```

```

textCtx.canvas.width = width;
textCtx.canvas.height = height;
textCtx.font = "20px monospace";
textCtx.textAlign = "center";
textCtx.textBaseline = "middle";
textCtx.fillStyle = "black";
textCtx.clearRect(0, 0, textCtx.canvas.width, textCtx.canvas.height);
textCtx.fillText(text, width / 2, height / 2);
return textCtx.canvas;
}

```

现在我们需要在 WebGL 中绘制 2 个不同东西：“F”和文本，我想切换到[使用一些前一篇文章中所描述的辅助函数](#)。如果你还不清楚 `programInfo`, `bufferInfo` 等，你需要浏览那篇文章。

现在，让我们创建一个“F”和四元组单元。

```

// Create data for 'F'
var fBufferInfo = primitives.create3DVertexBufferInfo(gl);
// Create a unit quad for the 'text'
var textBufferInfo = primitives.createPlaneBufferInfo(gl, 1, 1, 1, 1, makeXRotation(Math.PI / 2));

```

一个四元组单元是一个 1 单元大小的四元组(方形)，中心在原点。`createPlaneBufferInfo` 在 xz 平面创建一个平面。我们通过一个矩阵旋转它，就得到一个 xy 平面四元组单元。

接下来创建 2 个着色器：

```

// setup GLSL programs
var fProgramInfo = createProgramInfo(gl, ["3d-vertex-shader", "3d-fragment-shader"]);
var textProgramInfo = createProgramInfo(gl, ["text-vertex-shader", "text-fragment-shader"]);

```

创建我们的文本纹理：

```

// create text texture.
var textCanvas = makeTextCanvas("Hello!", 100, 26);
var textWidth = textCanvas.width;
var textHeight = textCanvas.height;
var textTex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, textTex);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, textCanvas);
// make sure we can render it even if it's not a power of 2
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);

```

为“F”和文本设置 uniforms：

```
var fUniforms = {
  u_matrix: makeldentity(),
};

var textUniforms = {
  u_matrix: makeldentity(),
  u_texture: textTex,
};
```

当我们计算 F 的矩阵时，保存 F 的矩阵视图：

```
var matrix = makeldentity();
matrix = matrixMultiply(matrix, preTranslationMatrix);
matrix = matrixMultiply(matrix, scaleMatrix);
matrix = matrixMultiply(matrix, rotationZMatrix);
matrix = matrixMultiply(matrix, rotationYMatrix);
matrix = matrixMultiply(matrix, rotationXMatrix);
matrix = matrixMultiply(matrix, translationMatrix);
matrix = matrixMultiply(matrix, viewMatrix);
var fViewMatrix = copyMatrix(matrix); // remember the view matrix for the text
matrix = matrixMultiply(matrix, projectionMatrix);
```

像这样绘制 F：

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

setBuffersAndAttributes(gl, fProgramInfo.attribSetters, fBufferInfo);

copyMatrix(matrix, fUniforms.u_matrix);
setUniforms(fProgramInfo.uniformSetters, fUniforms);

// Draw the geometry.
gl.drawElements(gl.TRIANGLES, fBufferInfo.numElements, gl.UNSIGNED_SHORT, 0);
```

文本中我们只需要知道 F 的原点位置，我们还需要测量和单元四元组相匹配的纹理尺寸。最后，我们需要多种投影矩阵。

```
// scale the F to the size we need it.
// use just the view position of the 'F' for the text
var textMatrix = makeldentity();
textMatrix = matrixMultiply(textMatrix, makeScale(textWidth, textHeight, 1));
textMatrix = matrixMultiply(
  textMatrix,
  makeTranslation(fViewMatrix[12], fViewMatrix[13], fViewMatrix[14]));
textMatrix = matrixMultiply(textMatrix, projectionMatrix);
```

然后渲染文本

```
// setup to draw the text.
gl.useProgram(textProgramInfo.program);

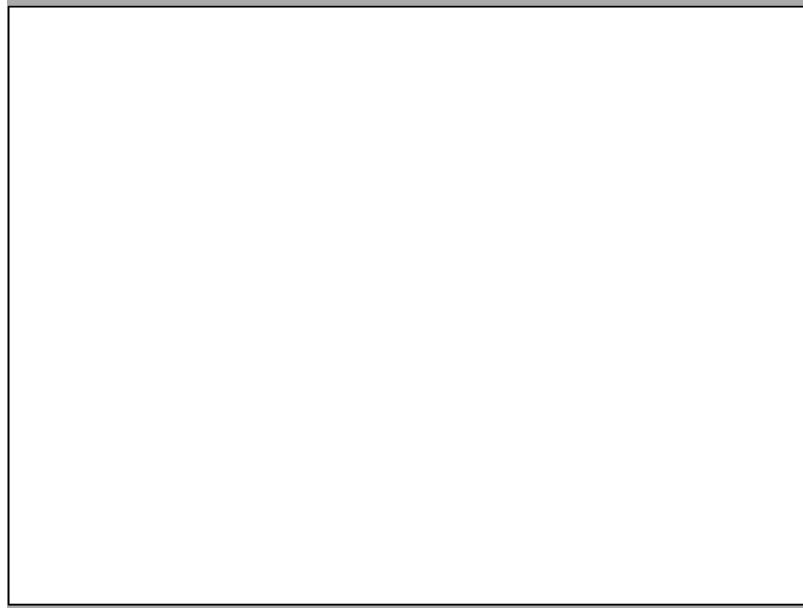
setBuffersAndAttributes(gl, textProgramInfo.attribSetters, textBufferInfo);

copyMatrix(textMatrix, textUniforms.u_matrix);
setUniforms(textProgramInfo.uniformSetters, textUniforms);

// Draw the text.
gl.drawElements(gl.TRIANGLES, textBufferInfo.numElements, gl.UNSIGNED_SHORT, 0);
```

即：

### HTML Text Textures



你会发现有时候我们文本的一部分遮盖了我们 Fs 的一部分。这是因为我们绘制一个四元组。画布的默认颜色是透明的黑色(0,0,0,0)和我们在四元组中使用这种颜色绘制。我们也可以混合像素。

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

根据混合函数，将源像素(这个颜色取自片段着色器)和 目的像素(画布颜色)结合在一起。在混合函数中，我们为源像素设置：SRC\_ALPHA，为目的像素设置：ONE\_MINUS\_SRC\_ALPHA。

```
result = dest * (1 - src_alpha) + src * src_alpha
```

举个例子，如果目的像素是绿色的 0,1,0,1 和源像素是红色的 1,0,0,1，如下：

```
src = [1, 0, 0, 1]
dst = [0, 1, 0, 1]
src_alpha = src[3] // this is 1
result = dst * (1 - src_alpha) + src * src_alpha

// which is the same as
result = dst * 0 + src * 1

// which is the same as
result = src
```

对于纹理的部分内容，使用透明的黑色 0,0,0,0

```
src = [0, 0, 0, 0]
dst = [0, 1, 0, 1]
src_alpha = src[3] // this is 0
result = dst * (1 - src_alpha) + src * src_alpha

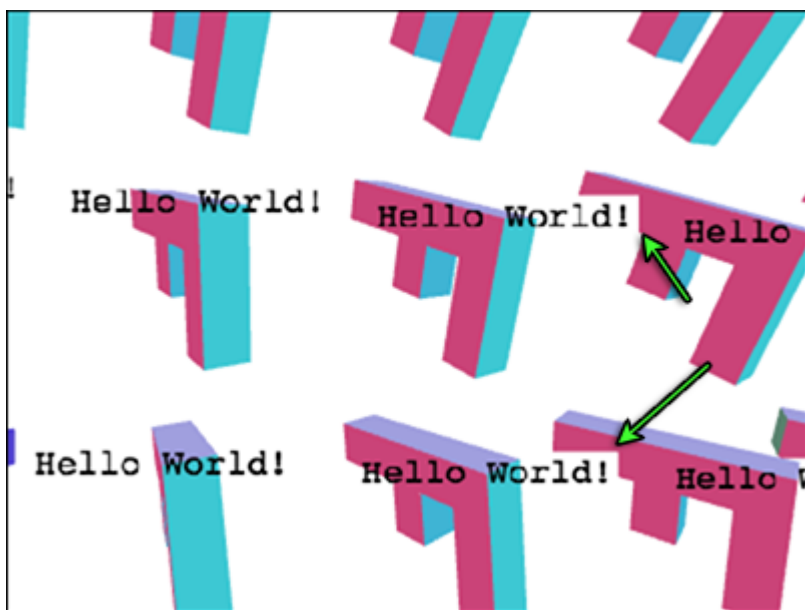
// which is the same as
result = dst * 1 + src * 0

// which is the same as
result = dst
```

这是启用了混合的结果。



你可以看到尽管它还不完美，但它已经更好了。如果你仔细看，有时能看到这个问题



发生什么事情了？我们正在绘制一个 F 然后是它的文本，然后下一个 F 的重复文本。所以当我们绘制文本时，我们仍然需要一个深度缓冲，即使混合了一些像素来保持背景颜色，深度缓冲仍然需要更新。当我们绘制下一个 F，如果 F 的部分是之前绘制文本的一些像素，他们就不会再绘制。

我们刚刚遇到的最困难的问题之一，在 GPU 上渲染 3D。透明度也存在问题。

针对几乎所有透明呈现问题，最常见的解决方案是先画出所有不透明的东西，之后，按中心距的排序，绘制所有的透明的东西，中心距的排序是在深度缓冲测试开启但深度缓冲更新关闭的情况下得出的。

让我们先单独绘制透明材料(文本)中不透明材料(Fs)的部分。首先，我们要声明一些来记录文本的位置。

```
var textPositions = [];
```

在循环中渲染记录位置的 Fs

```
matrix = matrixMultiply(matrix, viewMatrix);
var fViewMatrix = copyMatrix(matrix); // remember the view matrix for the text
textPositions.push([matrix[12], matrix[13], matrix[14]]); // remember the position for the text
```

在我们绘制 “F” s之前，我们禁用混合并打开写深度缓冲

```
gl.disable(gl.BLEND);
gl.depthMask(true);
```

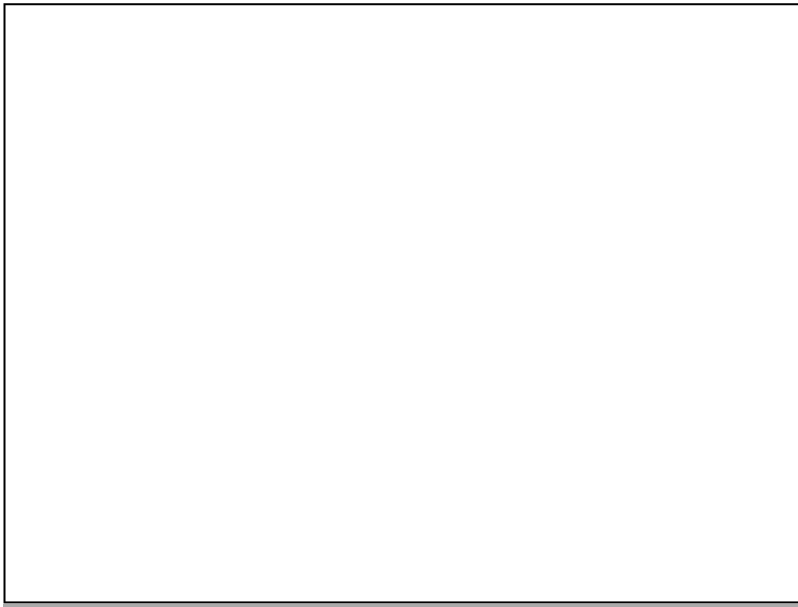
绘制文本时，我们将打开混合并关掉写作深度缓冲

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.depthMask(false);
```

然后在我们保存的所有位置绘制文本

```
textPositions.forEach(function(pos) {  
    // draw the text  
    // scale the F to the size we need it.  
    // use just the position of the 'F' for the text  
    var textMatrix = makeIdentity();  
    textMatrix = matrixMultiply(textMatrix, makeScale(textWidth, textHeight, 1));  
    textMatrix = matrixMultiply(textMatrix, makeTranslation(pos[0], pos[1], pos[2]));  
    textMatrix = matrixMultiply(textMatrix, projectionMatrix);  
  
    // setup to draw the text.  
    gl.useProgram(textProgramInfo.program);  
  
    setBuffersAndAttributes(gl, textProgramInfo.attribSetters, textBufferInfo);  
  
    copyMatrix(textMatrix, textUniforms.u_matrix);  
    setUniforms(textProgramInfo.uniformSetters, textUniforms);  
  
    // Draw the text.  
    gl.drawElements(gl.TRIANGLES, textBufferInfo.numElements, gl.UNSIGNED_SHORT, 0);  
});
```

现在启动：



请注意我们没有像我上面提到的那样分类。在这种情况下，因为我们绘制大部分是不透明文本，所以即使排序也没有明显差异，所以就省去了这一步骤，节省资源用于其他文章。

另一个问题是文本的“F”总是交叉。实际上这个问题没有一个具体的解决方案。如果你正在构造一个 MMO，希望每个游戏者的文本总是出现在你试图使文本出现的顶部。只需要将之转化为一些单元 +Y，足以确保它总是位于游戏者之上。

你也可以使之向 camera 移动。在这里我们这样做只是为了好玩。因为“pos”是在坐标系中，意味着它是相对于眼(在坐标系中即：0,0,0)。所以如果我们使之标准化，我们可以得到一个单位向量，这个向量的指向是从原点到某一点，我们可以乘一定数值将文本特定数量的单位靠近或远离眼。

```
// because pos is in view space that means it's a vector from the eye to
// some position. So translate along that vector back toward the eye some distance
var fromEye = normalize(pos);
var amountToMoveTowardEye = 150; // because the F is 150 units long
var viewX = pos[0] - fromEye[0] * amountToMoveTowardEye;
var viewY = pos[1] - fromEye[1] * amountToMoveTowardEye;
var viewZ = pos[2] - fromEye[2] * amountToMoveTowardEye;

var textMatrix = makeIdentity();
textMatrix = matrixMultiply(textMatrix, makeScale(textWidth, textHeight, 1));
textMatrix = matrixMultiply(textMatrix, makeTranslation(viewX, viewY, viewZ));
textMatrix = matrixMultiply(textMatrix, projectionMatrix);
```

即：



你还可能会注意到一个字母边缘问题。





这里的问题是 Canvas2D api 只引入了自左乘 alpha 值。当我们上传内容到试图 unpremultiply 的纹理 WebGL，它就不能完全做到，这是因为自左乘 alpha 会失真。

为了解决这个问题，使 WebGL 不会 unpremultiply：

```
gl.pixelStorei(gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL, true);
```

这告诉 WebGL 支持自左乘 alpha 值到 `gl.texImage2D` 和 `gl.texSubImage2D`。如果数据传递给 `gl.texImage2D` 已经自左乘，就像 canvas2d 数据，那么 WebGL 就可以通过。

我们还需要改变混合函数

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.blendFunc(gl.ONE, gl.ONE_MINUS_SRC_ALPHA);
```

老方法是源色乘以 alpha。这是 `SRC_ALPHA` 意味着什么。但是现在我们的纹理数据已经被乘以其 alpha。这是 `premultiplied` 意味着什么。所以我们不需要 GPU 做乘法。将其设置为 `ONE` 意味着乘以 1。

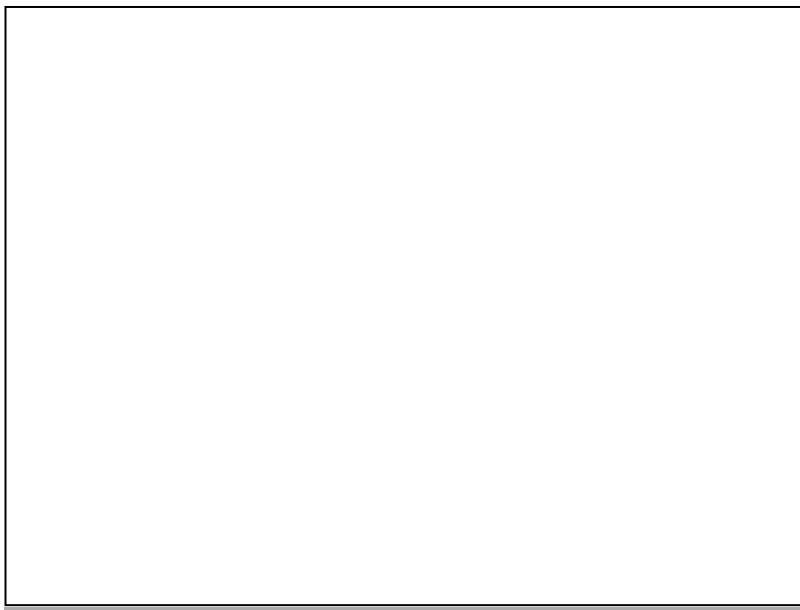


边缘现在没有了。

如果你想保持文本在一种固定大小，但仍然正确？那么，如果你还记得[透视文章](#)中透视矩阵以  $-Z$  调整我们的对象使其在距离上更小。所以，我们可以以  $-Z$  倍数调整以达到我们想要的规模作为补偿。

```
...
// because pos is in view space that means it's a vector from the eye to
// some position. So translate along that vector back toward the eye some distance
var fromEye = normalize(pos);
var amountToMoveTowardEye = 150; // because the F is 150 units long
var viewX = pos[0] - fromEye[0] * amountToMoveTowardEye;
var viewY = pos[1] - fromEye[1] * amountToMoveTowardEye;
var viewZ = pos[2] - fromEye[2] * amountToMoveTowardEye;
var desiredTextScale = -1 / gl.canvas.height; // 1x1 pixels
var scale = viewZ * desiredTextScale;

var textMatrix = makeIdentity();
textMatrix = matrixMultiply(textMatrix, makeScale(textWidth * scale, textHeight * scale, 1));
textMatrix = matrixMultiply(textMatrix, makeTranslation(viewX, viewY, viewZ));
textMatrix = matrixMultiply(textMatrix, projectionMatrix);
...
```



如果你想在每个 F 中绘制不同文本，你应该为每个 F 构造一个新纹理，为每个 F 更新文本模式。

```
// create text textures, one for each F
var textTextures = [
  "anna", // 0
  "colin", // 1
  "james", // 2
  "danny", // 3
  "kalin", // 4
```

```

"hiro", // 5
"eddie", // 6
"shu", // 7
"brian", // 8
"tami", // 9
"rick", // 10
"gene", // 11
"natalie", // 12,
"evan", // 13,
"sakura", // 14,
"kai", // 15,
].map(function(name) {
  var textCanvas = makeTextCanvas(name, 100, 26);
  var textWidth = textCanvas.width;
  var textHeight = textCanvas.height;
  var textTex = gl.createTexture();
  gl.bindTexture(gl.TEXTURE_2D, textTex);
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, textCanvas);
  // make sure we can render it even if it's not a power of 2
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  return {
    texture: textTex,
    width: textWidth,
    height: textHeight,
  };
});

```

然后在呈现时选择一个纹理

```

textPositions.forEach(function(pos, ndx) {

  +// select a texture
  +var tex = textTextures[ndx];

  // scale the F to the size we need it.
  // use just the position of the 'F' for the text
  var textMatrix = makeIdentity();
  *textMatrix = matrixMultiply(textMatrix, makeScale(tex.width, tex.height, 1));

```

并在绘制前为纹理设置统一结构

```

textUniforms.u_texture = tex.texture;

```



我们一直用黑色绘制到画布上的文本。这比用白色呈现文本更有用。然后我们再增加文本的颜色，以便得到我们想要的任何颜色。

首先我们改变文本材质，通过复合一个颜色

```
varying vec2 v_texcoord;

uniform sampler2D u_texture;
uniform vec4 u_color;

void main() {
    gl_FragColor = texture2D(u_texture, v_texcoord) * u_color;
}
```

当我们绘制文本到画布上时使用白色

```
textCtx.fillStyle = "white";
```

然后我们添加一些其他颜色

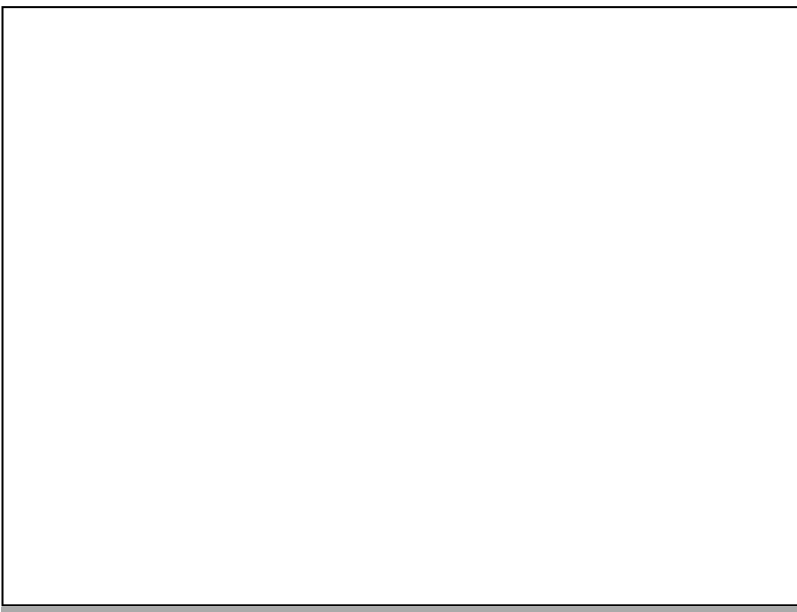
```
// colors, 1 for each F
var colors = [
    [0.0, 0.0, 0.0, 1], // 0
    [1.0, 0.0, 0.0, 1], // 1
    [0.0, 1.0, 0.0, 1], // 2
    [1.0, 1.0, 0.0, 1], // 3
    [0.0, 0.0, 1.0, 1], // 4
    [1.0, 0.0, 1.0, 1], // 5
    [0.0, 1.0, 1.0, 1], // 6
    [0.5, 0.5, 0.5, 1], // 7
```

```
[0.5, 0.0, 0.0, 1], // 8  
[0.0, 0.0, 0.0, 1], // 9  
[0.5, 5.0, 0.0, 1], // 10  
[0.0, 5.0, 0.0, 1], // 11  
[0.5, 0.0, 5.0, 1], // 12,  
[0.0, 0.0, 5.0, 1], // 13,  
[0.5, 5.0, 5.0, 1], // 14,  
[0.0, 5.0, 5.0, 1], // 15,  
];
```

在绘制时选择一个颜色

```
// set color uniform  
textUniforms.u_color = colors[ndx];
```

结果如下：



这个技术实际上是大多数浏览器使用 GPU 加速时的技术。他们用 HTML 的内容和你应用的各种风格生成纹理，只要这些内容没有改变，他们就可以在滚动时再次渲染纹理。当然，如果你一直都在更新那么这技术可能会有点慢，因为重新生成纹理并更新它对于 GPU 来说是一个相对缓慢的操作。

## WebGL 文本 使用字符纹理

---

在上一篇文章中我们复习了[在 WebGL 场景中如何使用纹理绘制文本](#)。技术是很常见的，对一些事物也是极重要的，例如在多人游戏中你想在一个头像上放置一个名字。同时这个名字也不能影响它的完美性。

比方说你想呈现大量的文本，这需要经常改变 UI 之类的事物。[前一篇文章](#)给出的最后一个例子中，一个明显的解决方案是给每个字母加纹理。我们来尝试一下改变上一个例子。

```
var names = [
  "anna", // 0
  "colin", // 1
  "james", // 2
  "danny", // 3
  "kalin", // 4
  "hiro", // 5
  "eddie", // 6
  "shu", // 7
  "brian", // 8
  "tami", // 9
  "rick", // 10
  "gene", // 11
  "natalie", // 12,
  "evan", // 13,
  "sakura", // 14,
  "kai", // 15,
];

// create text textures, one for each letter
var textTextures = [
  "a", // 0
  "b", // 1
  "c", // 2
  "d", // 3
  "e", // 4
  "f", // 5
  "g", // 6
  "h", // 7
  "i", // 8
  "j", // 9
  "k", // 10
  "l", // 11
  "m", // 12,
```

```

"n",// 13,
"o",// 14,
"p",// 14,
"q",// 14,
"r",// 14,
"s",// 14,
"t",// 14,
"u",// 14,
"v",// 14,
"w",// 14,
"x",// 14,
"y",// 14,
"z",// 14,
].map(function(name) {
  var textCanvas = makeTextCanvas(name, 10, 26);

```

相对于为每个名字呈现一个四元组，我们将为每个名字的每个字母呈现一个四元组。

```

// setup to draw the text.
// Because every letter uses the same attributes and the same program
// we only need to do this once.
gl.useProgram(textProgramInfo.program);
setBuffersAndAttributes(gl, textProgramInfo.attribSetters, textBufferInfo);

textPositions.forEach(function(pos, ndx) {
  var name = names[ndx];
  // for each letter
  for (var ii = 0; ii < name.length; ++ii) {
    var letter = name.charCodeAt(ii);
    var letterNdx = letter - "a".charCodeAt(0);
    // select a letter texture
    var tex = textTextures[letterNdx];

    // use just the position of the 'F' for the text

    // because pos is in view space that means it's a vector from the eye to
    // some position. So translate along that vector back toward the eye some distance
    var fromEye = normalize(pos);
    var amountToMoveTowardEye = 150; // because the F is 150 units long
    var viewX = pos[0] - fromEye[0] * amountToMoveTowardEye;
    var viewY = pos[1] - fromEye[1] * amountToMoveTowardEye;
    var viewZ = pos[2] - fromEye[2] * amountToMoveTowardEye;
    var desiredTextScale = -1 / gl.canvas.height; // 1x1 pixels
    var scale = viewZ * desiredTextScale;

    var textMatrix = makeIdentity();

```

```

textMatrix = matrixMultiply(textMatrix, makeTranslation(ii, 0, 0));
textMatrix = matrixMultiply(textMatrix, makeScale(tex.width * scale, tex.height * scale, 1));
textMatrix = matrixMultiply(textMatrix, makeTranslation(viewX, viewY, viewZ));
textMatrix = matrixMultiply(textMatrix, projectionMatrix);

// set texture uniform
textUniforms.u_texture = tex.texture;
copyMatrix(textMatrix, textUniforms.u_matrix);
setUniforms(textProgramInfo.uniformSetters, textUniforms);

// Draw the text.
gl.drawElements(gl.TRIANGLES, textBufferInfo.numElements, gl.UNSIGNED_SHORT, 0);
}
});

```

你可以看到它是如何工作的：



不幸的是它很慢。下面的例子：单独绘制 73 个四元组，还看不出来差别。我们计算 73 个矩阵和 292 个矩阵倍数。一个典型的 UI 可能有 1000 个字母要显示。这是众多工作可以得到一个合理的帧速率的方式。

解决这个问题通常的方法是构造一个纹理图谱，其中包含所有的字母。我们讨论[给立方体的 6 面加纹理](#)时，复习了纹理图谱。

下面的代码构造了字符的纹理图谱。

```

function makeGlyphCanvas(ctx, maxWidthOfTexture, heightOfLetters, baseLine, padding, letters) {
    var rows = 1; // number of rows of glyphs
    var x = 0; // x position in texture to draw next glyph
    var y = 0; // y position in texture to draw next glyph

```



```

var glyphInfos = { // info for each glyph
};

// Go through each letter, measure it, remember its width and position
for (var ii = 0; ii < letters.length; ++ii) {
var letter = letters[ii];
var t = ctx.measureText(letter);
// Will this letter fit on this row?
if (x + t.width + padding > maxWidthOfTexture) {
    // so move to the start of the next row
    x = 0;
    y += heightOfLetters;
    ++rows;
}
// Remember the data for this letter
glyphInfos[letter] = {
    x: x,
    y: y,
    width: t.width,
};
// advance to space for next letter.
x += t.width + padding;
}

// Now that we know the size we need set the size of the canvas
// We have to save the canvas settings because changing the size
// of a canvas resets all the settings
var settings = saveProperties(ctx);
ctx.canvas.width = (rows == 1) ? x : maxWidthOfTexture;
ctx.canvas.height = rows * heightOfLetters;
restoreProperties(settings, ctx);

// Draw the letters into the canvas
for (var ii = 0; ii < letters.length; ++ii) {
var letter = letters[ii];
var glyphInfo = glyphInfos[letter];
var t = ctx.fillText(letter, glyphInfo.x, glyphInfo.y + baseLine);
}

return glyphInfos;
}

```

现在我们试试看：

```

var ctx = document.createElement("canvas").getContext("2d");
ctx.font = "20px sans-serif";

```

```

ctx.fillStyle = "white";
var maxTextureWidth = 256;
var letterHeight = 22;
var baseline = 16;
var padding = 1;
var letters = "0123456789.abcdefghijklmnopqrstuvwxyz";
var glyphInfos = makeGlyphCanvas(
  ctx,
  maxTextureWidth,
  letterHeight,
  baseline,
  padding,
  letters);

```

结果如下

---

现在，我们已经创建了一个我们需要使用的字符纹理。看看效果怎样，我们为每个字符建四个顶点。这些顶点将使用纹理坐标来选择特殊的字符。

给定一个字符串，来建立顶点：

```

function makeVerticesForString(fontInfo, s) {
  var len = s.length;
  var numVertices = len * 6;
  var positions = new Float32Array(numVertices * 2);
  var texcoords = new Float32Array(numVertices * 2);
  var offset = 0;
  var x = 0;
  for (var ii = 0; ii < len; ++ii) {
    var letter = s[ii];
    var glyphInfo = fontInfo.glyphInfos[letter];
    if (glyphInfo) {
      var x2 = x + glyphInfo.width;
      var u1 = glyphInfo.x / fontInfo.textureWidth;
      var v1 = (glyphInfo.y + fontInfo.letterHeight) / fontInfo.textureHeight;
      var u2 = (glyphInfo.x + glyphInfo.width) / fontInfo.textureWidth;
      var v2 = glyphInfo.y / fontInfo.textureHeight;

      // 6 vertices per letter
      positions[offset + 0] = x;
      positions[offset + 1] = 0;
      texcoords[offset + 0] = u1;

```

```

texcoords[offset + 1] = v1;

positions[offset + 2] = x2;
positions[offset + 3] = 0;
texcoords[offset + 2] = u2;
texcoords[offset + 3] = v1;

positions[offset + 4] = x;
positions[offset + 5] = fontInfo.letterHeight;
texcoords[offset + 4] = u1;
texcoords[offset + 5] = v2;

positions[offset + 6] = x;
positions[offset + 7] = fontInfo.letterHeight;
texcoords[offset + 6] = u1;
texcoords[offset + 7] = v2;

positions[offset + 8] = x2;
positions[offset + 9] = 0;
texcoords[offset + 8] = u2;
texcoords[offset + 9] = v1;

positions[offset + 10] = x2;
positions[offset + 11] = fontInfo.letterHeight;
texcoords[offset + 10] = u2;
texcoords[offset + 11] = v2;

x += glyphInfo.width;
offset += 12;
} else {
    // we don't have this character so just advance
    x += fontInfo.spaceWidth;
}
}

// return ArrayBufferViews for the portion of the TypedArrays
// that were actually used.
return {
arrays: {
    position: new Float32Array(positions.buffer, 0, offset),
    texcoord: new Float32Array(texcoords.buffer, 0, offset),
},
numVertices: offset / 2,
};
}

```

为了使用它，我们手动创建一个 `bufferInfo`。(如果你已经不记得了，可以查看前面的文章：[bufferInfo 是什么](#))。

```
// Maunally create a bufferInfo
var textBufferInfo = {
  attribs: {
    a_position: { buffer: gl.createBuffer(), numComponents: 2, },
    a_texcoord: { buffer: gl.createBuffer(), numComponents: 2, },
  },
  numElements: 0,
};
```

使用 `bufferInfo` 中的字符创建画布的 `fontInfo` 和纹理：

```
var ctx = document.createElement("canvas").getContext("2d");
ctx.font = "20px sans-serif";
ctx.fillStyle = "white";
var maxTextureWidth = 256;
var letterHeight = 22;
var baseline = 16;
var padding = 1;
var letters = "0123456789.,abcdefghijklmnopqrstuvwxyz";
var glyphInfos = makeGlyphCanvas(
  ctx,
  maxTextureWidth,
  letterHeight,
  baseline,
  padding,
  letters);
var fontInfo = {
  glyphInfos: glyphInfos,
  letterHeight: letterHeight,
  baseline: baseline,
  spaceWidth: 5,
  textureWidth: ctx.canvas.width,
  textureHeight: ctx.canvas.height,
};
```

然后渲染我们将更新缓冲的文本。我们也可以构成动态的文本：

```
textPositions.forEach(function(pos, ndx) {

  var name = names[ndx];
  var s = name + ":" + pos[0].toFixed(0) + "," + pos[1].toFixed(0) + "," + pos[2].toFixed(0);
  var vertices = makeVerticesForString(fontInfo, s);
```

```

// update the buffers
textBufferInfo.attribs.a_position.numComponents = 2;
gl.bindBuffer(gl.ARRAY_BUFFER, textBufferInfo.attribs.a_position.buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices.arrays.position, gl.DYNAMIC_DRAW);
gl.bindBuffer(gl.ARRAY_BUFFER, textBufferInfo.attribs.a_texcoord.buffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices.arrays.texcoord, gl.DYNAMIC_DRAW);

setBuffersAndAttributes(gl, textProgramInfo.attribSetters, textBufferInfo);

// use just the position of the 'F' for the text
var textMatrix = makeIdentity();
// because pos is in view space that means it's a vector from the eye to
// some position. So translate along that vector back toward the eye some distance
var fromEye = normalize(pos);
var amountToMoveTowardEye = 150; // because the F is 150 units long
textMatrix = matrixMultiply(textMatrix, makeTranslation(
pos[0] - fromEye[0] * amountToMoveTowardEye,
pos[1] - fromEye[1] * amountToMoveTowardEye,
pos[2] - fromEye[2] * amountToMoveTowardEye));
textMatrix = matrixMultiply(textMatrix, projectionMatrix);

// set texture uniform
copyMatrix(textMatrix, textUniforms.u_matrix);
setUniforms(textProgramInfo.uniformSetters, textUniforms);

// Draw the text.
gl.drawArrays(gl.TRIANGLES, 0, vertices.numVertices);
});

```

即：



这是使用字符纹理集的基本技术。可以添加一些明显的东西或方式来改进它。

- 重用相同的数组。

目前，每次被调用时，`makeVerticesForString` 就会分配新的 32 位浮点型数组。这最终可能会导致垃圾收集出现问题。重用相同的数组可能会更好。如果不是足够大，你也放大数组，但是保留原来的大小。

- 添加支持回车

当生成顶点时，检查 `\n` 是否存在从而实现换行。这将使文本分隔段落更容易。

- 添加对各种格式的支持。

如果你想文本居中，或调整你添加的一切文本的格式。

- 添加对顶点颜色的支持。

你可以为文本的每个字母添加不同的颜色。当然你必须决定如何指定何时改变颜色。

这里不打算涉及的另一大问题是：纹理大小有限，但字体实际上是无限的。如果你想支持所有的 unicode，你就必须处理汉语、日语和阿拉伯语等其他所有语言，2015 年在 unicode 有超过 110000 个符号！你不可能在纹理中适配所有这些，也没有足够的空间供你这样做。

操作系统和浏览器 GPU 加速处理这个问题的方式是：通过使用一个字符纹理缓存实现。上面的实现他们是把纹理处理成纹理集，但他们为每个 glyph 布置一个固定大小的区域，保留纹理集中最近使用的符号。如果需要绘制一个字符，而这个字符不在纹理集中，他们就用他们需要的这个新的字符取代最近最少使用的一个。当然如果他们即将取代的字符仍被有待绘制的四元组引用，他们需要绘制他们之前所取代的字符。

虽然我不推荐它，但是还有另一件事你可以做，将这项技术和[以前的技术](#)结合在一起。你可以直接渲染另一种纹理的符号。当然 GPU 加速画布已经这样做了，你可能没有自己动手的理由。

另一种在 WebGL 中绘制文本的方法实际上是使用了 3D 文本。在上面所有的例子中“F”是一个 3D 的字母。你已经为每个字母都构成了一个相应的 3D 字符。3D 字母常见于标题和电影标志，此外的用处就少了。

我希望在 WebGL 这可以覆盖文本。

# 极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/webgl/>