1.什么是 webgl?

WebGL 简单来说就是浏览器原生支持的一套 3D 绘制 API。在 WebGL 出现前，想要在浏览器中绘制 3D 场景，是一件很繁琐的事情，没有统一的规范，一切都得靠插件。伴随着 HTML5 的到来，WebGL1.0, WebGL2.0 规范相继推出，各大浏览器厂商纷纷开始实现 WebGL 接口。到目前为止，主流浏览器的最新版本都已经支持 WebGL，譬如 Chrome，Firefox，Opera，Safari。

2.WebGL 和 OpenGL 的异同

WebGL 和 OpenGL 都带一个 GL 后缀，看起来很有关系，实际上关系很大。OpenGL 有一个功能子集 OpenGL ES，OpenGL ES 由一系列的 C 函数组成，主要用在移动设备上。WebGL 则是使用了 Javascript 将 OpenGL ES 进行了封装，你可以使用 Javascript 代替 C 在浏览器中进行 OpenGL ES 的编程。WebGL1.0 是基于 OpenGL ES2.0 的封装，WebGL2.0 是基于 OpenGL ES3.0 的封装。下面是浏览器对 WebGL1.0 和 WebGL2.0 支持的状况。WebGL1.0 已经被广泛支持，WebGL2.0 还有待提高。

Desktop Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	9	(Yes)	4.0 (2.0)	11	12	5.1
WebGL 2	56	No support	51 (51)	No support	43	No support

这是链接 Khronos WebGL （里面有 WebGL 2.0 的链接）的网址：

https://www.khronos.org/webgl/

第一步我们肯定要知道你所用的浏览器是不是支持 WebGL 2.0。点击下列网址，你就会知道，而且会有挺多有意思的事情。

WebGL 2.0 支持测试链接：http://webglreport.com/?v=2

写在前面

Canvas 绘制二维位图是通过 getContext('2d')获取类似画笔的东西在其中绘制，而这里是绘制三维位图，通过 getContext('webgl')来获取这只神奇的笔。

绘图过程

canvas 2D 绘制过程

页面添加 canvas 元素 -> 获取 2D 画笔 -> 设置好画笔等 -> 清空屏幕（如果需要） -> 绘制（相应 API）

canvas WebGL 绘制过程

页面添加 canvas 元素 -> 获取 3D 画笔 -> 设置好着色器等 -> 清空屏幕(如果需要）-> 绘制（相应 API）

可以看出来，主要是【设置好着色器等】这一步和之前的绘制方法不一样，因此，我们先来说明一下，什么是着色器?

着色器

绘制三维图形的时候（以绘制一个点为例），你需要告诉他点的位置、尺寸和颜色，当你设置好这些以后就类似 2D 设置好画笔了，就可以绘制了，而这三个的设置就是设置着色器的过程。

WebGL 需要两种着色器

- 顶点着色器（Vertex shader）：顶点着色器是用来描述顶点特性（如位置、颜色等）的程序。顶点(Vertex)是指二维或三维空间的一个点，比如二维或三维空间线与线之间的交叉点或者端点。
- 片元着色器（Fragment shader）：进行逐片元处理过程（如光照等）的程序。片元(fragment)是一个 WebGL 的术语，你可以将其理解成像素。

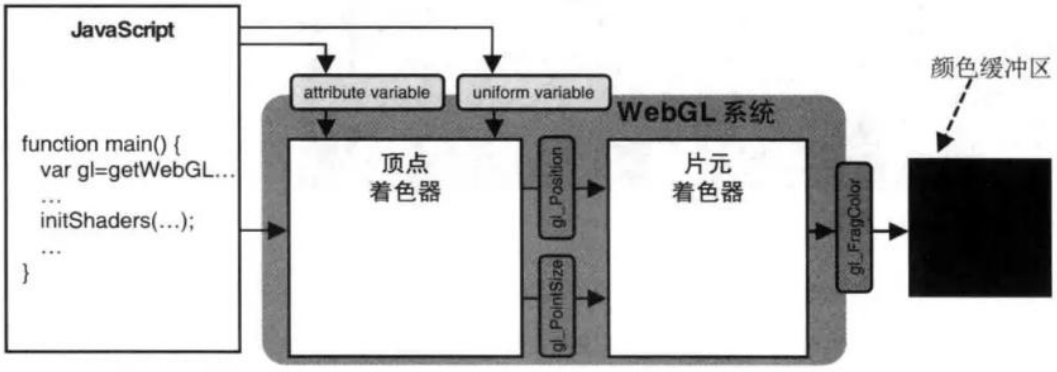
着色器语言使用的是 GLSL ES 语言，所以 javascript 需要将之存放在字符串中，等待调用编译

着色器变量类型

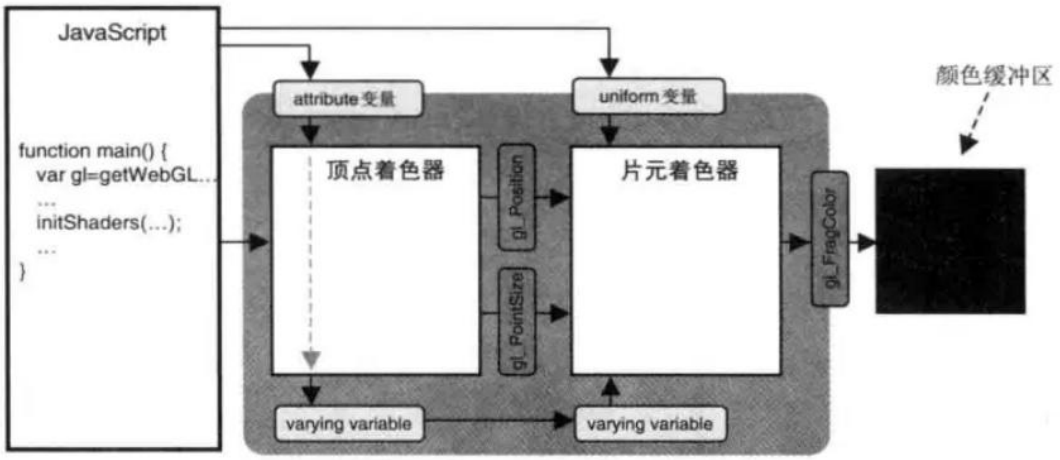
attribute 变量：属性，传输与顶点相关的数据；（顶点着色器）

uniform 变量：全局变量，传输与所有顶点都相同（或与顶点无关）的数据；（片元着色器）

varying 变量：可变量，从顶点着色器向片元着色器传输数据。（片元着色器）



向顶点着色器传输数据的两种方式



向片元着色器传输数据的两种方式

只有顶点着色器才能使用 attribute 变量，使用片元着色器时，就需要使用 uniform 变量和 varying 变量。

着色器中包含几个内置变量：**gl_Position, gl_PointSize, gl_FragColor**。

着色器语言中涉及到 vec4 的数据类型，此数据类型是一个思维浮点数组，所以其值不可以是整形如(1,1,1,1)，**正确应为：(1.0,1.0,1.0,1.0)**

// 顶点着色器

VSHADER_SOURCE =

```
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '  gl_PointSize = 10.0;\n' +
  '}\n';
// 片元着色器
var FSHADER_SOURCE =
  'uniform vec4 u_FragColor;\n' +
  'void main() {\n' +
  '  gl_FragColor = u_FragColor;\n' +
  '}\n';
```

- gl_Position: 为一种 vec4 类型的变量，且必须被赋值。四维坐标矢量，我们称之为齐次坐标，即(x,y,z,w)等价于三维左边(x/w,y/w,z/w)，w 相当于深度，没有特殊要求设置为 1.0 即可。
- gl_PointSize：表示顶点的尺寸，也是浮点数，为非必填项，如果不填则默认显示为 1.0。
- gl_FragColor：该变量为片元着色器唯一的内置变量，表示其颜色，也是一个 vec4 类型变量，分别代表(R,G,B,A)，不过颜色范围是从 0.0-1.0 对应 Javascript 中的#00-#FF。

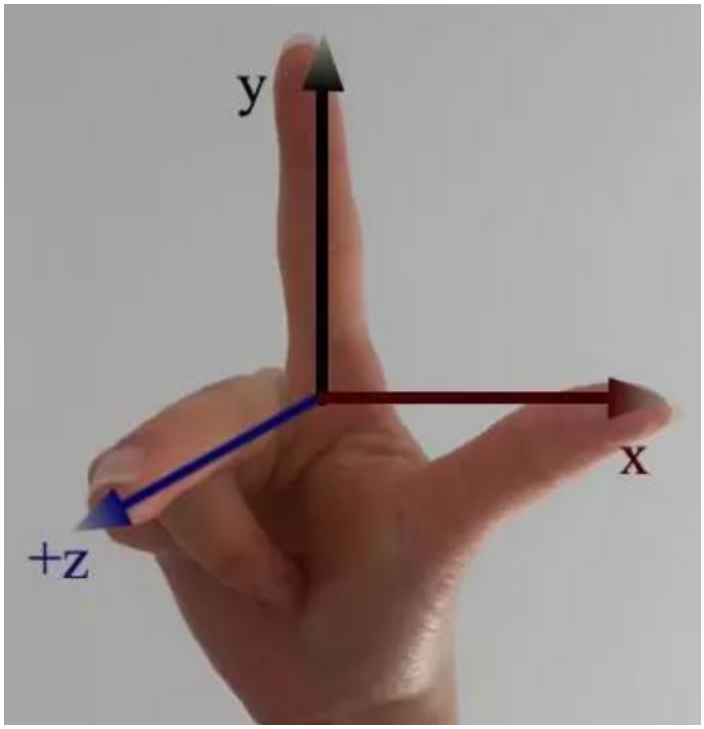
使用着色器

让我们来看看如何把着色器代码编译并且使用起来

着色器代码需要载入到一个程序中，webgl 使用此程序才能调用着色器。

```
var program = gl.createProgram();// 创建顶点着色器
var vShader = gl.createShader(gl.VERTEX_SHADER);// 创建片元着色器
var fShader = gl.createShader(gl.FRAGMENT_SHADER);// shader 容器与着色器绑定
gl.shaderSource(vShader, VSHADER_SOURCE);
gl.shaderSource(fShader, FSHADER_SOURCE);// 将 GLSE 语言编译成浏览器可用代码
gl.compileShader(vShader);
gl.compileShader(fShader);// 将着色器添加到程序上
gl.attachShader(program, vShader);
gl.attachShader(program, fShader);// 链接程序，在链接操作执行以后，可以任意修改 shader 的源代码，
对 shader 重新编译不会影响整个程序，除非重新链接程序
gl.linkProgram(program);// 加载并使用链接好的程序
gl.useProgram(program);
```

有了着色器我们就可以着手去绘制图像了，既然绘制 3D 图形，必然会有对应的三维坐标系，WebGL 采用右手坐标系，如图所示：



绘制一个简单的点

下面，我们通过绘制一个简单的点为例来按照上面的过程来绘制，感受一下。

页面添加 canvas 元素

首先在页面中添加 canvas 标签：

```
<canvas id='webgl' width='400' height='400'></canvas>
```

获取 3D 画笔，创建 webGL 对象

获取 3D 画笔也 2D 的一样，只不过参数有点改变：不同浏览器生命 WebGL 对象方式有所区别，虽然大部分浏览器都支持 experimental-webgl，而且以后会变成 webgl，所以创建时做一下兼容处理

```
var canvas = document.getElementById('webgl');
var gl =  canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

设置好着色器等

WebGL 依赖一种新的称为着色器（shader）的绘图机制。着色器提供了灵活且强大的绘制二维或三维图形的方法，所有 WebGL 必须使用它。着色器不仅强大，而且更复杂，仅仅通过一条简单的绘图指令是不能操作它的。

着色器的设置会有点麻烦，其实着色器就是二段字符串，先定义好，然后使其生效。

下面是大致过程：

1.定义着色器字符串 -> 2.创建着色器对象并绑定好对应的着色器字符串 -> 3.创建着色器程序并添加前面创建的着色器对象 -> 4.把着色器程序链接成一个完整的程序并使用他

说明：上面的过程看起来有点昏，是因为 3D 的绘制比较复杂，有很多新概念的东西，需要慢慢去适应。

下面看看具体代码：

【1. 定义着色器字符串】

顶点着色器有二个固定的属性 gl_Position 和 gl_PointSize，分别表示位置和尺寸：

```
<!-- 顶点着色器 -->
<script type='x-shader/x-vertex' id='vs-shader'>
```

```
    void main(){
        gl_Position=vec4(0.0,0.0,0.0,1.0); //位置
        gl_PointSize=100.0;   //尺寸
    }
    </script>
```

片元着色器有一个固定的属性 gl_FragColor 表示颜色：

```
<!-- 片元着色器 -->
<script type='x-shader/x-fragment' id='fs-shader'>
    void main(){
        gl_FragColor=vec4(1.0,1.0,0.0,1.0);
}
</script>
```

这样，二段字符串就准备好了，也就是二个着色器字符串好了（为了方便定义在标签中，后续可以看见，他们就是二个字符串）。

【2.创建着色器对象并绑定好对应的着色器字符串】

具体在代码中备注，看下面代码：

```
//初始化着色器（上面第一步的其实就是这样在用，这里直接写字符串也可以，只不过上面的写法更方便编辑）
var vs_source = document.getElementById('vs-shader').innerHTML,
    fs_source = document.getElementById('fs-shader').innerHTML;
```

```
// 创建顶点着色器对象
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
// shader 容器与着色器绑定，绑定资源
gl.shaderSource(vertexShader, vs_source);
// 编译着色器，将 GLSE 语言编译成浏览器可用代码
gl.compileShader(vertexShader);

// 创建片段着色器对象
var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
// shader 容器与着色器绑定，绑定资源
gl.shaderSource(fragmentShader, fs_source);
// 编译着色器，将 GLSE 语言编译成浏览器可用代码
gl.compileShader(fragmentShader);
```

【3.创建着色器程序并添加前面创建的着色器对象】

到这里，二个着色器就准备好了，可是需要程序使用，还需要变成着色器程序，

因为着色器代码需要载入到一个程序中，webgl 使用此程序才能调用着色器，

看代码：

```
// 创建一个着色器程序
var glProgram = gl.createProgram();
// 把前面创建的二个着色器对象添加到着色器程序中
gl.attachShader(glProgram, vertexShader);
```

gl.attachShader(glProgram, fragmentShader);

【4. 把着色器程序链接成一个完整的程序并使用他】

此时，着色器程序也好了，我们需要指定使用他：

// 把着色器程序链接成一个完整的程序

//链接程序，在链接操作执行以后，可以任意修改 shader 的源代码，对 shader 重新编译不会影响整个程序，除非重新链接程序

gl.linkProgram(glProgram);

//加载并使用链接好的程序

gl.useProgram(glProgram);

这样，着色器就设置好了，当然，这是最简单的着色器，具体还有很多操作，后续会说明，先理解这些基本的吧。

清空屏幕（如果需要）

// 设置清空颜色

gl.clearColor(0.5, 0.5, 0.5, 1.0);// 用设置的颜色清空屏幕（参数代表的是清除颜色缓冲）

gl.clear(gl.COLOR_BUFFER_BIT);

绘制（相应 API）

最后一步绘制点：

gl.drawArrays(**gl**.POINTS，0，1);

drawArrays 就类似 2D 中的绘图方法，第一个参数代表绘制一个点，第二个参数代表从第一个点开始，第三个参数代码绘制个数一个。

你可能好奇他从哪里知道点的位置的？是的，你开始不是定义了顶点着色器吗？就是他。

WebGL 的坐标系

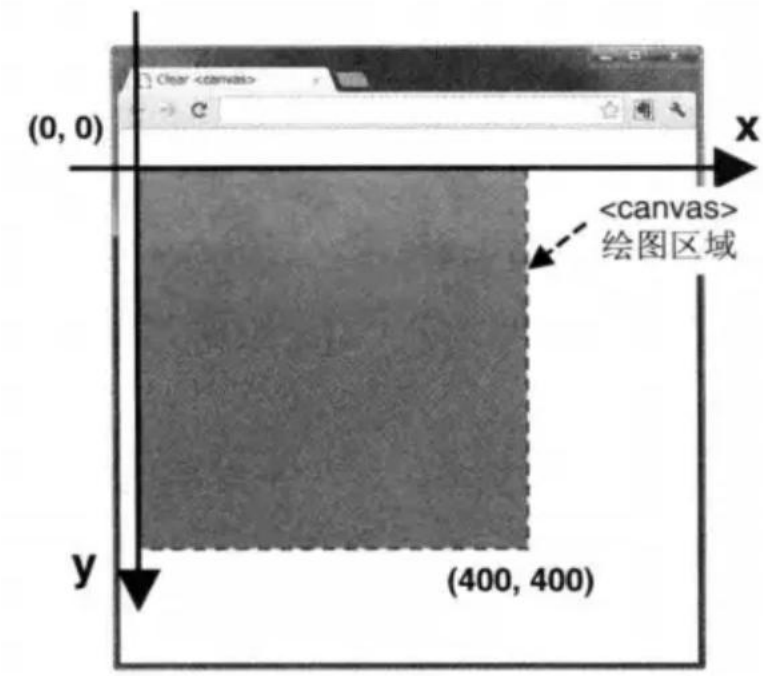
因为 WebGL 的坐标系与实际页面中的坐标系是不同的，如下图，普通 canvas 坐标系与正常的浏览器像素值相同，但 WebGL 中的坐标系是以整个 WebGL 中心点为（0.0，0.0），而且坐标的精确度为小数点后一位。坐标系对比如下图所示：

<canvas>坐标系

坐标原点：<canvas>绘图区域的左上角。

x 轴：水平向右为正。

y 轴：垂直向下为正。



WebGL 坐标系（坐标区间 -1.0 ~ 1.0）

坐标原点：<canvas>绘图区域的中心。

x 轴：水平向右为正。

y 轴：垂直向上为正。

由于 WebGL 处理的是三维图形，所以它使用三维坐标系统（笛卡尔坐标系），具有 X 轴、Y 轴和 Z 轴。三维坐标系统很容易理解，因为我们的世界也是三维的：具有宽度、高度和长度。在任何坐标系统中，轴的方向都非常重要。通常，在 WebGL 中，当你面向计算机屏幕时，X 轴是水平的（正方向为右），Y 轴是垂直的（正方向为下），而 Z 轴垂直于屏幕（正方向为外），如图 2.16 左所示。观察者的眼睛位于原点 (0.0, 0.0, 0.0) 处，视线则是沿着 Z 轴的负方向，从你指向屏幕（图 2.16 右）。这套坐标系又被称为**右手坐标系** (right-handed coordinate system)，因为可以用右手来表示，如图 2.17 所示。默认情况下 WebGL 使用右手坐标系，右手坐标系也会贯穿本书始终。然而，事实远远比这复杂。实际上，WebGL 本身既不是右手坐标系，又不是左手坐标系的。附录 D“WebGL/

image.png

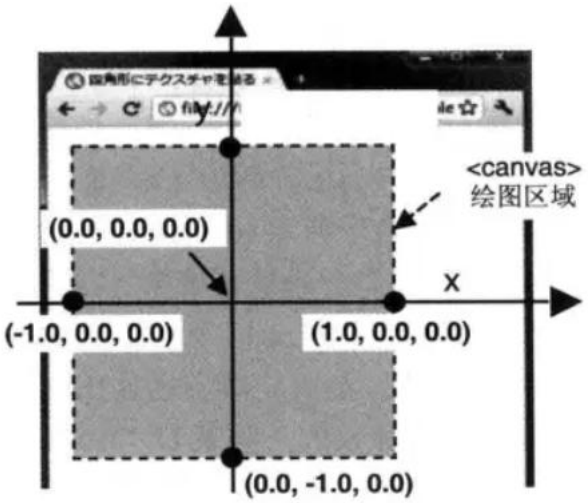
如图所示，WebGL 的坐标系和 <canvas> 绘图区的坐标系不同，需要将前者映射到后者。默认情况下，如图 2.18 所示，WebGL 坐标与 <canvas> 坐标的对应关系如下。

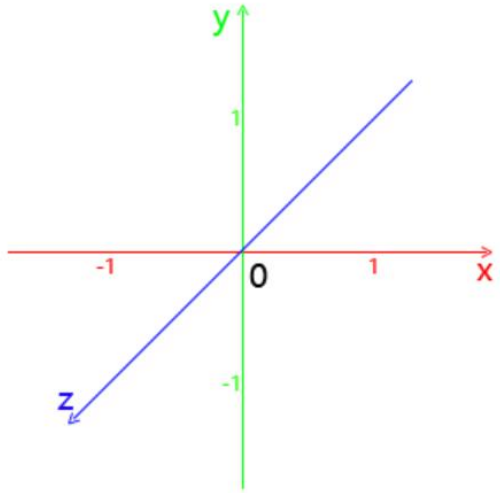
- <canvas> 的中心点：(0.0, 0.0, 0.0)
- <canvas> 的上边缘和下边缘：(−1.0, 0.0, 0.0) 和 (1.0, 0.0, 0.0)
- <canvas> 的左边缘和右边缘：(0.0, −1.0, 0.0) 和 (0.0, 1.0, 0.0)

WebGL 使用的是正交右手坐标系，且每个方向都有可使用的值的区间，超出该矩形区间的图像不会绘制：

- x 轴最左边为-1，最右边为 1；
- y 轴最下边为-1，最上边为 1；
- z 轴朝向你的方向最大值为 1，远离你的方向最大值为-1；

注：这些值与 Canvas 的尺寸无关，无论 Canvas 的长宽比是多少，WebGL 的区间值都是一致的。



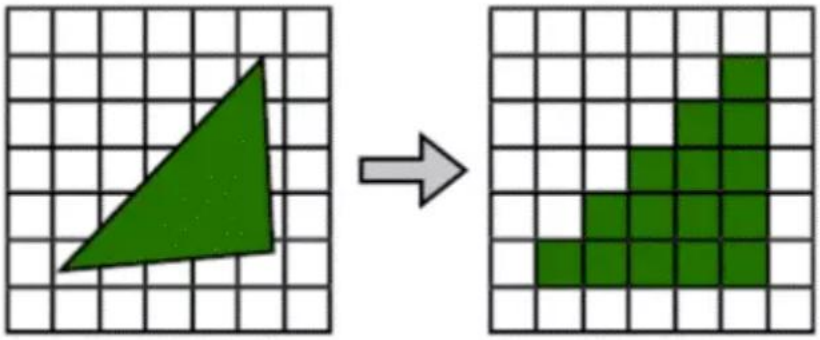



```
<canvas id="glcanvas" style="width: 700px; height: 500px;"> //错误

<canvas id="glcanvas" width="700" height="500"> //正确的方式
```

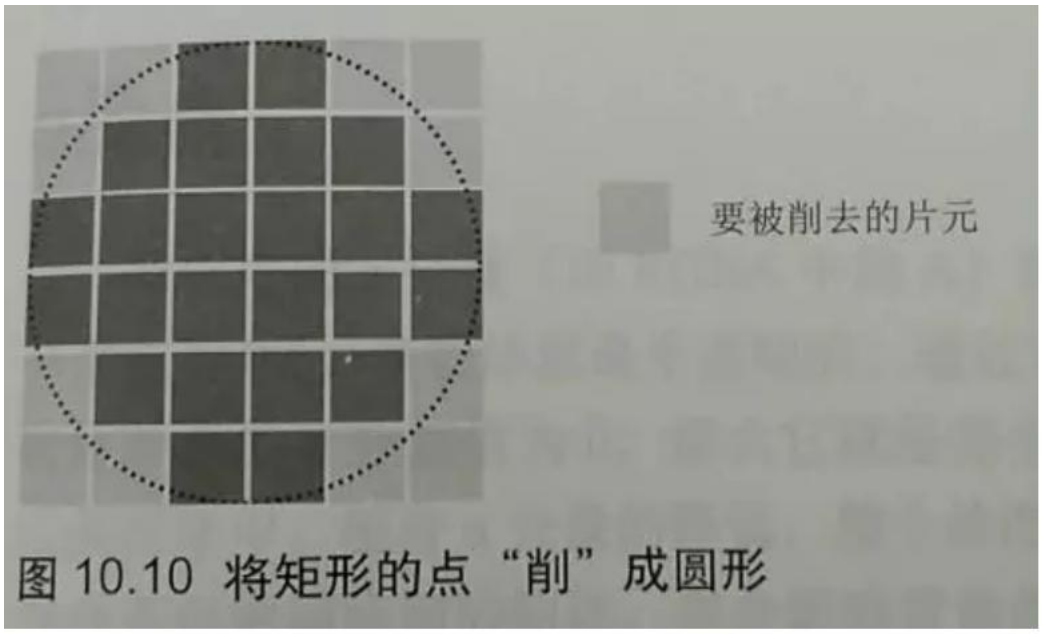
WebGL 是未提供绘制圆点的方法，那么如何绘制一个圆呢?

我们首先来了解一下 WebGL 的渲染机制，顶点着色器的信息在传递给 OpenGL 底层绘制的时候，会先进行光栅化，也就是把点转化成对应的像素。然后在片元着色器会逐个点进行渲染，最终就达到了视觉看到的效果。



光栅化

点也是一样，会将点转变成多个像素点，所以要变成圆点，需要如下的方式：



绘制圆点方式

我们需要在片元着色器中来处理，将非原型区域的像素点，不用片元着色器来绘制，着色器需要判断距离圆点的位置超过 0.5 的话，就忽略此片元点，最终就会出现一个圆点的效果。

```
var FSHADER_SOURCE = `
 #ifdef GL_ES
```

```
      precision mediump float;
#endif
void main() {
  float d = distance(gl_PointCoord, vec2(0.5,0.5));
    if(d < 0.5){
      gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }else{ discard;} }`;
```

成功绘制一个圆了

第二章：传递点的位置、大小和色彩

接着前面的文章开始学习，前面我们画了一个点，不过点的位置、大小和颜色都是在着色器中写死的，这一次，我们通过 js 代码传递给着色器。

传递位置

着色器修改

首先让我来介绍 2 个变量，我们需要借助这 2 个变量搭建的桥梁才能使 JavaScript 与 GLSL ES 之间进行沟通。

- attribute： 用于顶点着色器（Vertex Shader）传值时使用。
- uniform：可用于顶点着色器（Vertex Shader）与片元着色器（Fragment Shader）使用。

由于现在的点是传递进去的，不是写死的，应该是一个变量，因此我们修改成如下，将顶点动态化：

```
<!-- 顶点着色器 -->
<script type='x-shader/x-vertex' id='shader-vs'>
    attribute vec4 a_Position;
    void main(){
        gl_Position=a_Position;
        gl_PointSize=100.0;
}
</script>
```

其中 attribute 代表的是不同顶点的不同信息，后面还会看见 uniform 代表所有顶点一致的处理信息，而 varying 是顶点着色器和片元着色器桥梁，二者间传递数据。一句话，这三个就是标明变量的地位。

不过需要注意的是，attribute 只可以在顶点着色器中使用。

再看 vec4 这是一种类型，表示四个浮点数，这里声明类型。

我们定义了一个变量，返回赋值给位置，因此下一步，我们需要给这个变量赋值。

给着色器中定义的变量传递数据

在使用完整的程序后 ， 使用这个完整的程序 添加

使用 WebGL 来获取对应参数的存储地址地址 ；

```
// 获取 attribute 变量的存储位置 ，返回对应的地址信息
```

```
var a_Position = gl.getAttribLocation(glProgram, 'a_Position');
```

然后给变量赋值

方式一：

gl.vertexAttrib3f(aPosition, 1.0, 1.0, 0.0);

方式二：

```
//或者使用 Float32Array 来传参
var p = new Float32Array([1.0, 1.0, 1.0]);
gl.vertexAttrib3fv(aPosition, p);
```

注意：vertexAttrib3fv 这个函数是典型的 GLSL 语法命名规范，

vertexAttrib 函数功能，3：对应需要传 3 个参数，或者是几维向量，

f：表示参数是 float 类型，

v：表示传如的为一个 vector 变量。

也就是说对应设置顶点着色器的函数有一下几种功能

```
void gl.vertexAttrib1f(index, v0);
void gl.vertexAttrib2f(index, v0, v1);
void gl.vertexAttrib3f(index, v0, v1, v2);
void gl.vertexAttrib4f(index, v0, v1, v2, v3);
void gl.vertexAttrib1fv(index, value);
void gl.vertexAttrib2fv(index, value);
```

```
void gl.vertexAttrib3fv(index, value);
void gl.vertexAttrib4fv(index, value);
```

传递大小

着色器

还是一样，我们需要把原本写死的大小 定义为一个变量，赋值：

```
<!-- 顶点着色器 -->
<script type='x-shader/x-vertex' id='shader-vs'>
    attribute vec4 a_Position;
    attribute float a_PointSize;
    void main(){
        gl_Position=a_Position;
        gl_PointSize=a_PointSize;
}
</script>
```

给着色器中定义的变量传递数据

```
//【传递点的大小】
var a_PointSize=gl.getAttribLocation(glProgram,'a_PointSize');
gl.vertexAttrib1f(a_PointSize,30.0);
```

传递颜色

着色器

```
<!-- 片元着色器 -->
<script type='x-shader/x-fragment' id='shader-fs'>
    precision mediump float;
    uniform vec4 u_FragColor;
    void main(){
        gl_FragColor=u_FragColor;
}
</script>
```

由于 attribute 值可以用在顶点着色器，因此这里定义使用 uniform。

【precision mediump float;】这句话是修改精度的，必须有，除了 mediump 这种中等的精度，还有 lowp 和 highp。

给着色器中定义的变量传递数据

```
// 【传递点的颜色】
var u_FragColor=gl.getUniformLocation(glProgram,'u_FragColor');
gl.uniform4f(u_FragColor,1.0,0.0,0.0,1.0);
```

由于变量的性质改变了，因此获取变量的存储位置和传递数据的方法也要使用对应 api，不过大体差不多。

第三章：更灵活的数据传递和绘制图形

绘制三个点

前一篇文章我们绘制了一个点，现在想绘制三个点，怎么办?

当然，你可以按照绘制一个点的方法，绘制三次，不过我们这里来说说如何一次绘制三个点。

在前面代码的基础上，我们只需要修改传递点的个数，然后修改绘画方法就可以了。

传递三个点

之前【传递点的位置】处的代码的传递方法只可以传递一个点，如果想传递三个点，需要使用缓冲区。

因此，**我们第一步要创建一个缓冲区，并且把点的数据写入缓冲区，**代码如下（注释调原来【传递点的位置】的代码，在相同位置添加）：

```
//1.创建缓冲区对象
var vertexBuffer = gl.createBuffer();
// 2.绑定缓冲区对象（表明了缓冲区对象的用途）
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
// 3.向缓冲区对象中写入数据
var tempData=new Float32Array([
    0.0, 0.5,
    -0.5, -0.5,
    -0.5, 0.5
]);
```

gl.bufferData(gl.ARRAY_BUFFER, tempData, gl.STATIC_DRAW);

这样，缓冲区就准备好了，接着，把缓冲区和着色器联系起来：

// 4.获取变量存储位置

var a_Position = gl.getAttribLocation(glProgram, 'a_Position');

// 5.把缓冲区对象分配给 a_Position 变量

gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);

// 6.连接缓冲区对象和 a_Position 变量

gl.enableVertexAttribArray(a_Position);

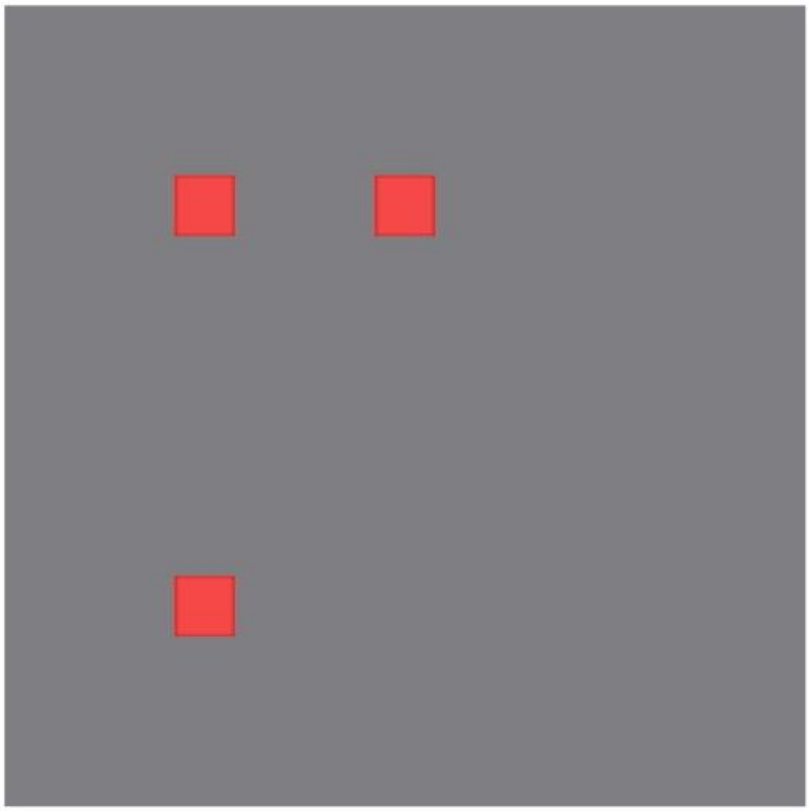
这样，点的位置就传递好了。

修改绘图方法

现在，修改一下绘图方法：

gl.drawArrays(gl.POINTS, 0, 3);

刷新页面，应该可以看见下图



绘制三个不同大小的点

数据修改

首先，我们要在缓冲区或准备一下数据，表示每个点的大小，对应代码修改如下：

```
var tempData=new Float32Array([
    0.0, 0.5,   10.0,
    -0.5, -0.5, 20.0,
    -0.5, 0.5,30.0
]);
//这个是每个数据大小，辅助用的
var FSIZE = tempData.BYTES_PER_ELEMENT;
```

传递点位置的方法修改

由于数据重新调整了，缓冲区分配给点位置的变量也要对应调整：

```
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE*3, 0);
```

不清楚的可以去百度一个 vertexAttribPointer 这个方法，后面我们会统一说明一些常用方法，这里就不说了。

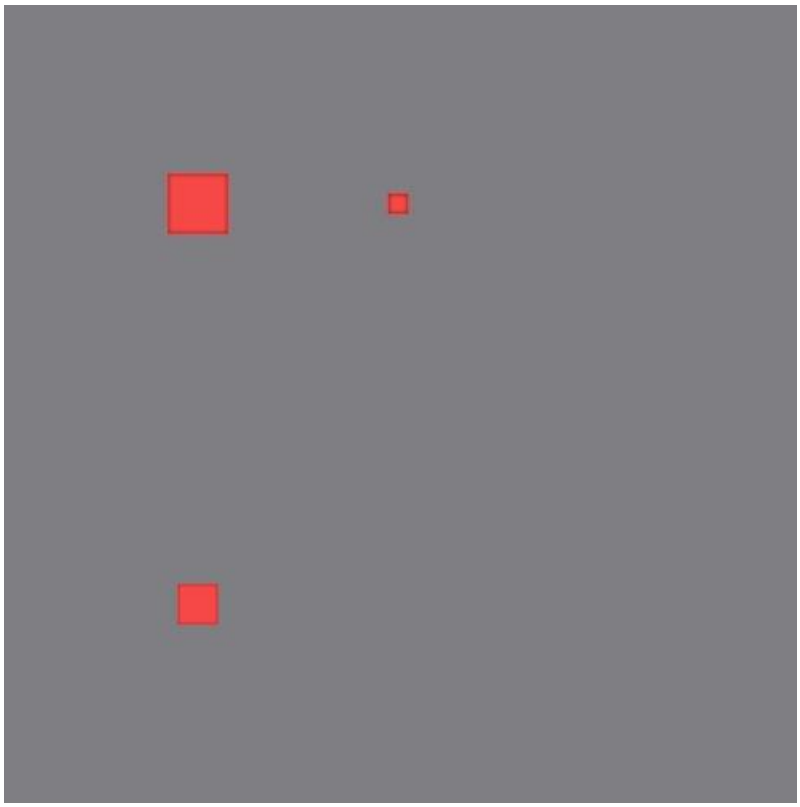
现在，你刷新页面，应该和之前的一样。

传递点大小的方法修改

和上面类似，之前传递一个点大小，现在传递三个就可以了：

```
//【传递点的大小】
var a_PointSize=gl.getAttribLocation(glProgram,'a_PointSize');
//之前传递一个的方法注释调了// gl.vertexAttrib1f(a_PointSize,30.0);
gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, FSIZE*3, FSIZE*2);
gl.enableVertexAttribArray(a_PointSize);
```

现在刷新页面，应该就可以了：



绘制三角形

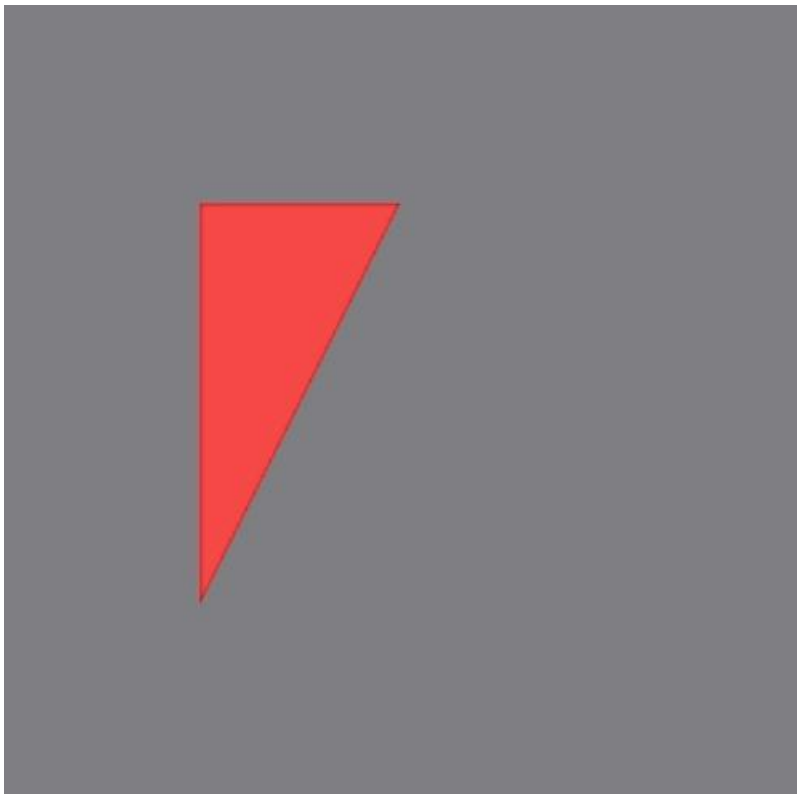
普通三角形

到目前为止，传递多个点应该没有问题了吧，如果想绘制复杂图形，调用对应 api 就可以了，比如绘制三角形，修改最后一个绘制方法（绘制三角形的话，点的大小就没有意义了，可以删除，留着也没事）：

```
// gl.drawArrays(gl.POINTS, 0, 3);

gl.drawArrays(gl.TRIANGLES, 0, 3);
```

结果如下：



彩色三角形

既然位置可以传递多个，是不是颜色也可以，是的，也可以。

同样的，我们先添加颜色数据：

```
var tempData=new Float32Array([
    0.0, 0.5,   10.0,1.0,0.0,1.0,
    -0.5, -0.5, 20.0,0.0,1.0,0.0,
    -0.5, 0.5,30.0,0.0,0.0,1.0
]);
```

修改缓冲区绑定位置的方法：

```
gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE*6, 0);
```

由于要传递三个点，uniform 只可以传递一致的数据，因此我们需要用 attribute，可是 attribute 有只可以用在顶点着色器，因此，需要先传递改顶点着色器，然后借助 varying 传递给片元着色器，着色器修改如下（点的大小代码就没有删除了）

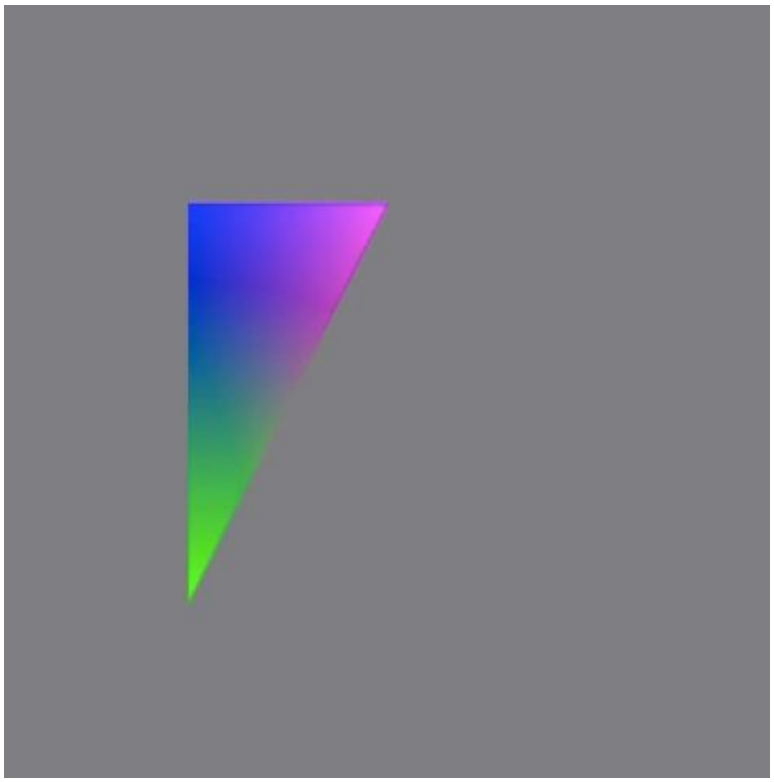
```
<!-- 顶点着色器 -->
<script type='x-shader/x-vertex' id='shader-vs'>
    attribute vec4 a_Position;
    attribute float a_PointSize;
    attribute vec4 a_Color;
    varying vec4 v_FragColor;
    void main(){
        gl_Position=a_Position;
        gl_PointSize=a_PointSize;
        v_FragColor=a_Color;
}
</script>
<!-- 片元着色器 -->
<script type='x-shader/x-fragment' id='shader-fs'>
    precision lowp float;
    varying vec4 v_FragColor;
    void main(){
        gl_FragColor=v_FragColor;
}
```

</script>

然后，和点的位置一样，修改传递色彩的代码：

```
// 【传递点的颜色】
var a_Color = gl.getAttribLocation(glProgram, 'a_Color');
// gl.uniform4f(u_FragColor,1.0,0.0,0.0,1.0);
gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 6, FSIZE * 3);
gl.enableVertexAttribArray(a_Color);
```

显示效果如下：



WebGL 的 3D 框架比较 Three.js 和 ThingJS

随着 flash 的没落，浏览器的原生能力的兴起。在 3D 方面 WebGL 不管从功能还是性能方面都在逐渐加强。2D 应用变为 3D 应用的需求也越来越强烈。 win10 的画图板支持 3D 图片，2d 工具 photoshop 也开始逐步集成了 3D 工具。

官网链接

面就基于 WebGL 技术探讨一下现在的两款 3D 框架。

Threejs(http://threejs.org/)

目前最流行的开源 3D 框架，2009 年 4 月诞生，2005 年 adobe 收购了 macromedia 的 flash 产品，2008，2009 年正是 flash 如日中天之时,threejs 也识时务的选择了 flash 的 ActionScript 平台，后来 flash 没落之后选择了 WebGL。

ThingJS(http://thingjs.com/)

新兴的 3D 框架，2018 年诞生，是针对物联网领域的 JavaScript 3D Library。它是由在 3D 领域经营多年的优锘科技公司研发，旨在简化 3D 应用开发。

设计角度

WebGL 可以处理 3D 图像，听起来是非常高兴的一件事，但是 WebGL 实在是太底层了，WebGl 解决是如何再画布上画图的问题，怎么画点，线，面，怎么上色，怎么贴图，怎么

处理光线，视角转动之后怎么换算绘制等等。这些对于一个做 3D 应用的开发者来说要学的东西太多了。

Threejs 库的出现解决了底层的渲染细节和复杂的数据结构，终于将复杂的底层细节抽象出来，使得大家开发 3d 应用更容易了一些。和很多开发者交流 threejs 都是他们首次接触的 WebGL 3D 库，并能很容易的就能开始做一些实验。

但是使用 Threejs 开发应用还是门槛很高，但就一个加载模型，调光，选择模型弹框的功能，就能干出 Threejs 上百行代码。同时还有很多复杂的 3D 概念需要理解。

这时就需要 ThingJS 了。

ThingJS 是更为上层的抽象，不用关心，渲染，mesh，光线等复杂概念。它抽象是一个个具体的模型，ThingJS 封装了对模型交互事件的各种 api，比如单击，左键，鼠标滑过等，ThingJS 封装了对模型的操作，例如移动，放大缩小，上色，勾边，甚至开门，ThingJS 还封装了模型的层次关系，例如物体是放在某个房间里的，房间又在某个楼层，楼层又是某个大楼的。大楼在园区里。

编码对比

这里仅仅从 3D 模型加载这个小点进行对比说明。更多内容大家可参考各自的网站 www.three.org 和 www.thingjs.com 进行详细对比。

three 的模型加载

```
function load3DModel(){
    /
       1、collada 是一种基于 XML 的 3D 模型交互方案，简单来说，就是一种 3D 模型可以通过 collada 转换成另一种 3D 模型，
       从而，各种 3D 模型都可以通过 collada 转换成 web 支持的 3D 模型。
       2、。dae 是一个钟 3D 模型的格式
       3、加载时注意浏览器同源策略的限制
     /
    var loader = new THREE.ColladaLoader();
    loader.load( "./model/avatar.dae", function ( collada ) {
        //找到模型中需要的对象。将相机看向这个对象是为了让这个对象显示在屏幕中心
        collada.scene.traverse( function ( child ) {
            if ( child instanceof THREE.SkinnedMesh ) {
                modelObj = child;
                camera.lookAt( child.position );
            }
        } );
        //将模型的场景加入到整体的场景
         modelObj.material.opacity = 0.8;
        scene.add( collada.scene ); //每个模型都要添加到场景

        //显示出模型的骨骼的代码,不需要可删去
        var helper = new THREE.SkeletonHelper( modelObj );
        helper.material.linewidth = 3;
        scene.add( helper );
    } );
}
```

ThingJS 的模型加载

```
var app = new THING.App({
  container: 'div3d',
  url: 'https://speech.uinnova.com/static/models/building'
});
```

只关注场景再页面的 div 的 id 和场景存放的地址，所有的细节 ThingJS 都处理好了。

场景加载完之后便可从场景获得加载内容，并进行交互应用开发。

```
    // 获取建筑对象
```

```
var building = app.buildings[0];
// 打印建筑中所有的楼层
building.floors.forEach(function(floor) {
      console.log('Floor: ' + floor.id);
});
// 获取室外对象
var outdoors = app.outdoors;
// 打印室外所有物体
outdoors.things.forEach(function(thing) {
   console.log('Thing: ' + thing.id);
}
```

多么完美的封装方式。更多细节可以到 www.thingjs.com 查看

总结

three.js(www.three.org)和 ThingJS(www.thingjs.com)都是 JavaScript 3D Library，都对 webGL 的 3D 处理能力进行了封装，但是 three.js 更偏三维技术底层，适用于 3D 爱好者学习 3D 技术；ThingJS 更偏物联网应用功能开发，重在开发效率，降低开发成本，适合于使用 3D 技术做项目的开发者

ThingJS 使用

ThingJS，注意不是 Three.js，是比 Three.js 封装度更高得 3D 框架，但不是开源的，使用得付费。也不是特别成熟，仅调研一波。

1、场景导入

- 控制台 新建项目保存
- 下载 CampusBuilder ，创建一个场景并导出为 Thingjs 场景包 B，如 hmf.tjs 格式
- 控制台 "我的资源-我的场景-上传场景"上传导出的场景包 B
- 打开在线开发平台，通过 "资源-场景资源" 引入场景包 B

2、模型制作

提供三种方式制作模型

- 打开在线开发平台，通过 "资源-模型资源" 搜索需要的模型
- 在资源中心，可找到很多付费和免费的模型，然后通过 url 创建
- 个人 obj 模型，将 obj 资源按照以下格式打成 zip 包

然后在 CampusBuilder 里的 DIY 模型库 tab 里点击上传资源

3、常用操作

3.1.创建对象

```
/**
* 说明：创建一个 Box，并在 app 的 update 事件中旋转 Box，摄像机看 Box
*/
var app = new THING.App();

// 创建 Box
var box = app.create({
    type: 'Box',
    position: [-4, 0, 0],
});

// update 事件
app.on('update', function () {
    box.rotateY(30 * app.deltaTime * 0.001); // 箱子自转
});

// 看 Box
app.camera.lookAt(box);
```

参数含义

创建对象后，设置其世界坐标即可展示在场景中。注意要设置其父元素，否则会直接挂到 app 下

3.2 获取对象

主要通过 query 方式以及层级关系查询对象，query 支持 id，类，属性，正则查询等。查询结果返回的是一个 Selector 对象，查询结果可以相加、排除，也可以直接绑定事件，或一些批量操作

```
// 查询 id 是 100 的对象

app.query("#100")[0];

// 查询名称（name）是 car01 的对象

app.query("car01");

// 查询物体类是 Thing 的对象

app.query(".Thing");
```

```
//有物体类型属性的，无论值是什么

app.query("[alarm]");

//查询物体类型属性是粮仓的对象

app.query("[报警=normal]");

app.query('["userData/物体类型"="粮仓"]');

// 查询 levelNum 属性大于 2 的对象,目前支持 <= ，< ，= ，> ，>=

app.query("[levelNum>2]");

// 正则表达式（RegExp）对象，目前只是对名称(name)属性值进行正则匹配

app.query(/car/);// 上例等同于 var reg=new RegExp('car');

app.query(reg);
```

3.3 控制对象

可控制的有：
1、物体显示与隐藏 visible；
2、物体移动 position，localPosition，translate；
3、物体旋转 angles，localAngles，rotateX，rotateY，rotate Y
4、缩放 scale
5、位移、旋转、缩放动画 moveTo，rotateTo，scaleTo
6、CSS 属性控制，如透明度 opacity、描边 outline、颜色 color、进出场动画 fadeIn fadeOut
7、连接操作：通过 add 接口可以添加子元素，添加子元素时，子物体的世界位置不发生变化，并保持那一刻与父物体的相对位置关系进行移动

3.4 鼠标交互

1、picker，结合 mouseenter 和 mouseleave 事件实现鼠标悬浮选中效果

```
// 鼠标拾取物体显示边框
    app.on(THING.EventType.MouseEnter, '.Thing', function(ev) {
        ev.object.style.outlineColor = '#FF0000';
    });
    // 鼠标离开物体边框取消
    app.on(THING.EventType.MouseLeave, '.Thing', function(ev) {
        ev.object.style.outlineColor = null;
    });
    // 每一帧判断拾取的物体是否发生变化
    app.on('update', function () {
        if (app.picker.isChanged()) {
            console.clear();
            // 打印当前被 pick 的物体
            if (app.picker.objects[0]) {
                console.log('当前拾取的物体 ' + app.picker.objects[0].name);
            }
            // 打印之前被 pick 的物体
            if (app.picker.previousObjects[0]) {
                console.log('之前拾取的物体 ' + app.picker.previousObjects[0].name);
            }
        }
    });
```

2、selection，鼠标点击选中效果，这个跟 click 事件不一样，selection 相当于在内部给你保存了选中的所有物体

3.5 摄像机

摄像机包含两个重要的位置参数：镜头位置 position 和被拍摄物体的位置 target (又叫目标点)

1、设置摄像机位置 position,target 和 object 二选一

```
// 直接设置

app.camera.position = [0, 20, 20]; // 镜头位置

app.camera.target = [-30, 10, 0]; // 目标点位置// fit 方法设置

app.camera.fit({

  position: [100, 100, 100],

  target: [0, 0, 0]});//设置摄像机到物体的“最佳看点”

app.camera.fit(obj);

//当不传参数时，设置摄像机到当前整个场景下的“最佳看点”

app.camera.fit();// 自定义设置

app.camera.fit({

  'object': obj,

  'xAngle': 60,  //绕物体自身 X 轴旋转角度

  'yAngle': 30,  //绕物体自身 Y 轴旋转角度
```

```
  'radiusFactor':3,   //物体包围球半径的倍数

});
```

2. lookAt 设置相机观察的物体

```
//摄像机一直“盯着”[0,0,0]点看

app.camera.lookAt([0, 0, 0]); ////摄像机一直“盯着”某物体看

var obj = app.query("car01")[0];

app.camera.lookAt(obj);

//取消摄影机一直盯着物体看

app.camera.lookAt(null);
```

3、flyTo 让摄像机从当前位置，飞行到将要设置的位置

```
//以 Quartic.In 的插值方式 让飞行速度渐增

app.camera.flyTo({

  position: [0, 20, 20],

  target: [-30, 10, 0],

  time: 3 * 1000,

  lerpType: THING.LerpType.Quartic.In});

//自定义飞到物体的摄像机位置参数（同 fit）
```

```
app.camera.flyTo({
  object: obj,
  xAngle: 30, //绕物体自身 X 轴旋转角度
  yAngle: 60, //绕物体自身 Y 轴旋转角度
  radiusFactor: 3, //物体包围盒半径的倍数
  time: 5 * 1000,
  complete: function() {
    console.log("飞行结束");
  }});
```

4、routateAround 设置相机环绕某点飞行

```
//环绕[0,0,0]点旋转 180 度，5s 转完
app.camera.rotateAround({
    target: [0,0,0],//环绕的坐标点
    time: 5*1000,//环绕飞行的时间
    yRotateAngle ： 180,//环绕 y 轴飞行的旋转角度
    complete:function(){
        console.log('结束环绕飞行');
```

```
    }

});
```

5、followObject 设置相机跟随物体

app.camera.followObject(obj);

6、move()，zoom()，rotateY()，rotateX()来控制摄像机的移动、缩放、旋转

```
//摄像机水平移动 10m

app.camera.move(10, 0);

//摄像机垂直移动 10m

app.camera.move(0, 10);

//摄像机向前推进 10m

app.camera.zoom(10);

//设置摄像机 target 为圆心转在水平方向上旋转的夹角增量

app.camera.rotateY(20);

// 设置摄像机 target 为圆心转在竖直方向上旋转的夹角增量

app.camera.rotateX(20);
```

3.6 界面元素

1、3D 元素 Marker,Webview，会随着缩放进大远小

```
app.create({
    type: "Marker",
    offset: [0, 2, 0],
    size: [4, 4],
    url: "https://thingjs.com/static/images/warning1.png",
    parent: app.query("car01")[0]
});
```

2、2D 元素 UIAnchor，需要自己写 html

```
var uiAnchor = app.create({
    type: "UIAnchor",
    parent: app.query("car02")[0],
    element: document.getElementById("XXXX"),
    localPosition: [0, 2, 0],
    pivot: [0.5, 1]});
uiAnchor.destroy();
uiAnchor.visible = true / false;
```

3、快捷界面库，一些 panel 控件，也可以绑定到模型上

```
// 创建 UIAnchor 面板
var ui = app.create({
```

```
    type: 'UIAnchor',   // 类型
    parent: obj,    // 父节点设置
    element: panel.domElement,   // 要绑定的页面的 element 对象
    localPosition: [0, 0, 0],   // 设置 localPosition 为[0, 0, 0]
    pivot: [-0.15, 1.8]   // 指定页面的哪个点放到 localPosition 位置上
});
```

3.7 数据对接

1、ajax，在 ThingJS 在线开发环境中，内置了 JQuery 库，可以直接使用 JQurey 封装的 Ajax 方法进行数据对接

```
$.ajax({
    'url': "http://3dmmd.cn:83/getMonitorDataById", //Ajax 请求服务的地址
    'type': "GET", //请求方式 "POST" 或 "GET"，默认为 "GET"
    'dataType': "json",     //服务返回的数据类型，推荐使用标准 JSON 数据格式
    //发送到服务器的数据
    'data': { 'id': 89757 },
    //请求成功后的回调函数
    'success': function (data) {
        console.log(data);
        // 处理返回的数据
    },
    //请求失败时调用的函数 有以下三个参数：XMLHttpRequest 对象、错误信息、（可选）捕获的异常对象
```

```
    'error': function (xhr, status, error) {
        console.log(xhr);
    },});
```

2、Websocket

```
// 创建一个 WebSocket 连接
var webSocket = new WebSocket('ws://3dmmd.cn:82');// 建立 websocket 连接成功触发 open 事件
webSocket.onopen = function () {
    console.log("websocket 服务器连接成功...");};
// 接收服务端数据 触发 message 事件
webSocket.onmessage = function (ev) {
    console.log("websocket 接收到的数据：" + ev.data);};// 关闭连接后 触发 close 事件
webSocket.onclose = function (evt) {
    console.log("websocket 关闭...");};// 通信发生错误时 触发 error 事件
webSocket.onerror = function () {
    console.log('发生错误')}var dataObj = { 'id': 89785 };// send 数据类型可以是 字符串 或 二进制对象（Blob 对象、ArrayBuffer 对象）
webSocket.send(JSON.stringify(dataObj));
```

4、一些概念

资源：场景资源（场景+模型） + 页面资源 + 全景图资源

层级

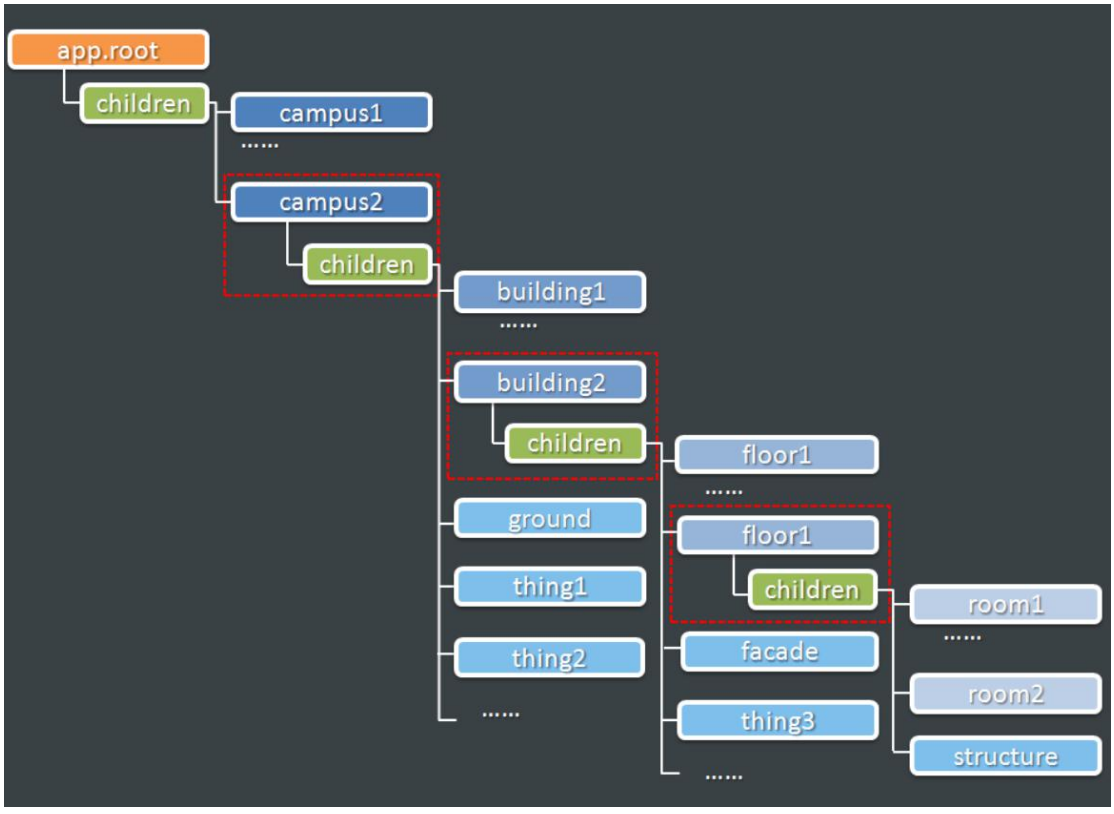
1、层级可以让我们方便管理和查询到场景中物体 && 批量操作物体

2、可以通过 app.level.change(obj) 切换到对应层级，app.level.back() 返回

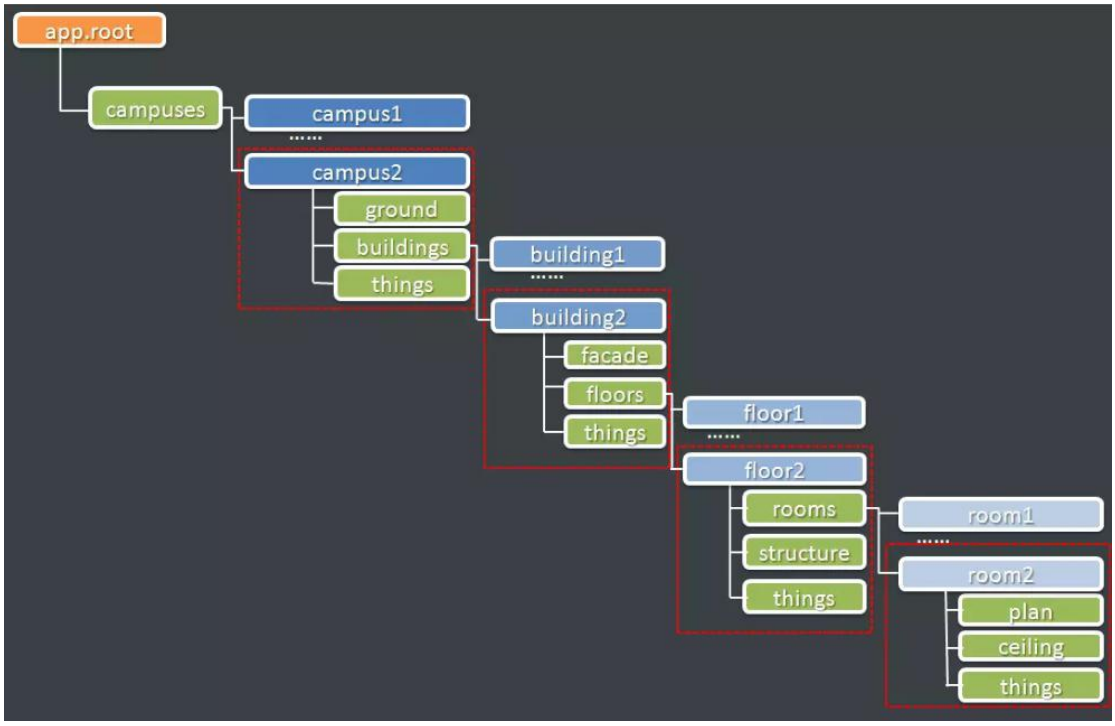
3、有两套层级体系：父子树和分类对象属性树

父子树，通过 children 连接层级：root => campus => building，ground，thing

父子树



分类对象属性树：每个对象都内置了一些属性，如 root.campuses，campus.ground，campus.buildings，campus.things，building.facade，building.floors，building.things，floor.rooms，floor.things



分类对象属性树

4、当进入层级时会触发 EnterLevel 事件。

当退出层级时会触发 LeaveLevel 事件。

当默认的层级切换飞行结束后，会触发 THING.EventType.LevelFlyEnd 事件

坐标系

1、世界坐标系

2、父级坐标系

3、自身坐标系

基本套路

- 加载场景
- 创建面板
- load 事件中处理逻辑

```
// 加载场景
var app = new THING.App({
    // 场景地址
    "url": "models/silohouse",
    // 天空盒
    "skyBox": "Universal"
});
// load 处理
app.on('load', function (ev) {
    // 获取粮仓
    siloHouse = app.query("[物体类型=粮仓]");
    // 添加粮仓自定义属性 monitorData，用来存储监控信息
    siloHouse.forEach(function (obj) {
        obj.monitorData = {};
    });

    // 创建开关控件
    createSwitchControl();
```

```
});
```