



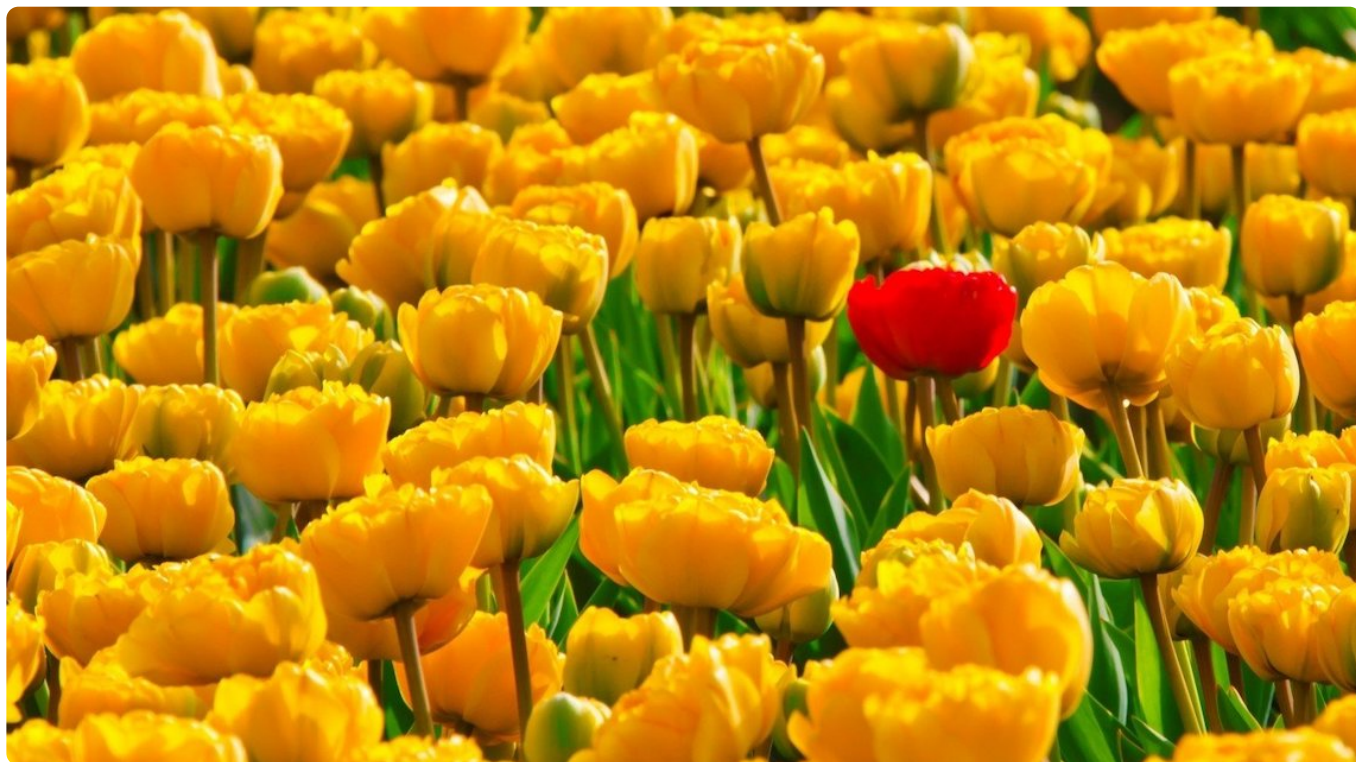
下载APP



31 | 针对海量数据，如何优化性能？

2020-09-09 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 11:54 大小 10.91M



你好，我是月影。

前两节课，我们一起学习了 Canvas2D 和 WebGL 性能优化的一些基本原则和处理方法。在正确运用这些方法后，我们能让渲染性能达到较高的程度，满足我们项目的需要。

不过，在数据量特别多的时候，我们会遇到些特殊的渲染需求，比如，要在一个地图上标记非常多的地理位置点（数千到数万），或者在地图上同时需要渲染几万条黑客攻击和防御数据。这些需求可能超过了常规优化手段所能达到的层次，需要我们针对数据和渲染的特点进行性能优化。

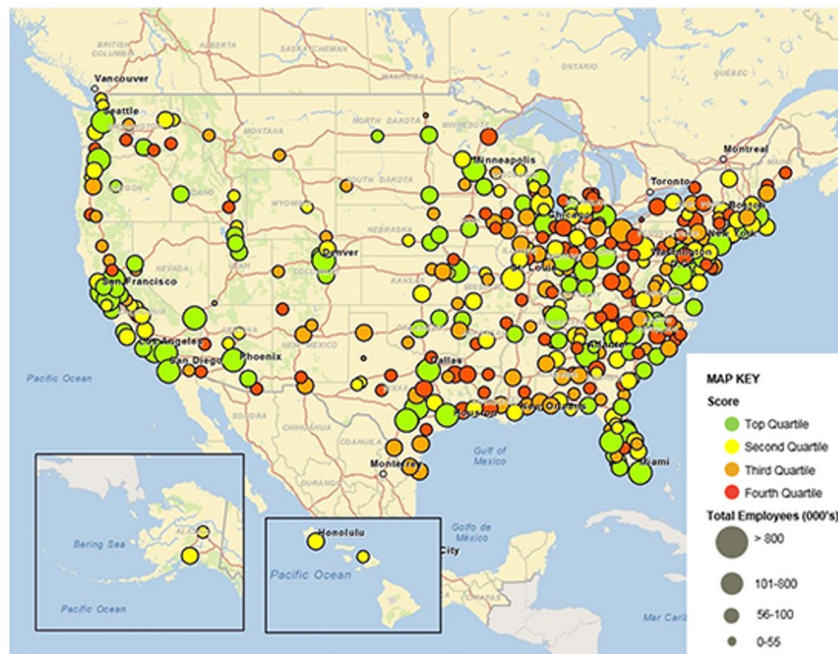


今天，我通过渲染动态地理位置的例子，来和你说说如何对特殊渲染需求迭代优化。不过，我今天用到特殊优化手段，只是一种具体的方法和手段，你可以借鉴他去理解思路，

但千万不要陷入到思维定式中。因为解决这些特殊渲染需求，并没有固定的路径或方法，它是一个需要迭代优化的过程，需要我们对 WebGL 的渲染机制非常了解，并深入思考，才能创造出最适合的方法来。在我们实际的工作里，还有许多其他的方法可以使用，你一定要根据自己的实际情况随机应变。

渲染动态的地理位置

我先来看今天要实现的例子。在地图可视化应用中，渲染地理位置信息是一类常见的需求，例如在这张地图上，我们就用许多不同颜色的小圆点标注出了美国一些不同的地区。



如果我们要实现这些静态的标准点，方法其实很简单，用 Canvas2D 或者 WebGL 都可以轻松实现。就算点数量比较多也没关系，因为一次性渲染对性能影响也不会很大。不过，如果我们想让圆点运动起来，比如，做出一种闪烁或者呼吸灯的效果，那我们就要考虑点的数量对性能的影响了。

那面对这一类特殊的渲染的需求，我们该怎么办呢？下面，我们先用常规的做法来实现，然后在这个方法上不断迭代优化。为了方便你理解，我就不绘制地图了，只绘制这些随机的小圆点。

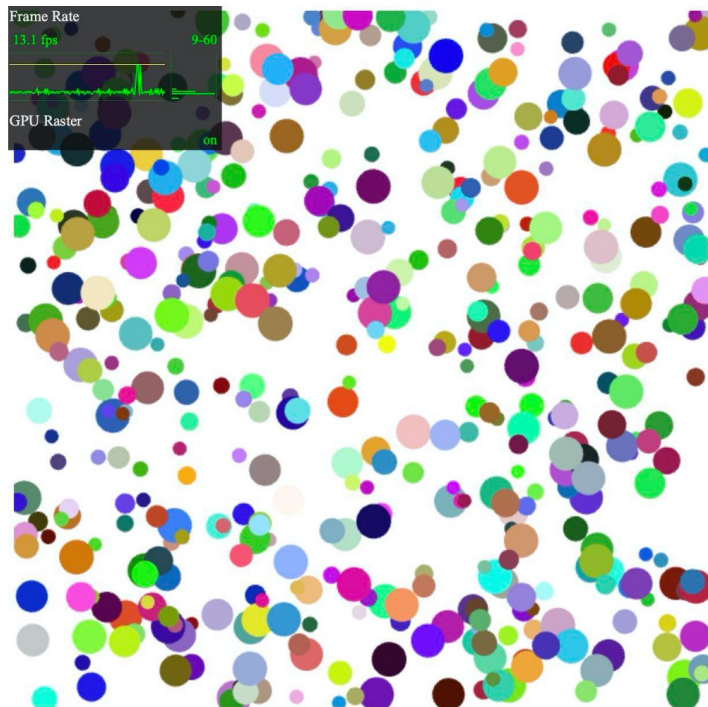
最简单的做法当然是一个一个圆绘制上去，也就是先创建圆的几何顶点数据，然后对每个圆设置不同的参数来分别绘制。实现代码如下：

```
1  const canvas = document.querySelector('canvas');
2  const renderer = new GLRenderer(canvas);
3
4  const vertex = `
5    attribute vec2 a_vertexPosition;
6    uniform vec2 xy;
7    uniform float uTime;
8    uniform float bias;
9
10   void main() {
11     vec3 pos = vec3(a_vertexPosition, 1);
12     float scale = 0.7 + 0.3 * sin(6.28 * bias + 0.003 * uTime);
13     mat3 m = mat3(
14       scale, 0, 0,
15       0, scale, 0,
16       xy, 1
17     );
18     gl_Position = vec4(m * pos, 1);
19   }
20 `;
21
22 const fragment = `
23   #ifdef GL_ES
24   precision highp float;
25   #endif
26
27   uniform vec4 u_color;
28
29   void main() {
30     gl_FragColor = u_color;
31   }
32 `;
33 const program = renderer.compileSync(fragment, vertex);
34 renderer.useProgram(program);
35
36 function circle(radius = 0.05) {
37   const delta = 2 * Math.PI / 32;
38   const positions = [];
39   const cells = [];
40   for(let i = 0; i < 32; i++) {
41     const angle = i * delta;
42     positions.push([radius * Math.sin(angle), radius * Math.cos(angle)]);
43     if(i > 0 && i < 31) {
44       cells.push([0, i, i + 1]);
45     }
46   }
47   return {positions, cells};
48 }
49
50 const COUNT = 500;
```

```
51 function init() {
52   const meshData = [];
53   const {positions, cells} = circle();
54   for(let i = 0; i < COUNT; i++) {
55     const x = 2 * Math.random() - 1;
56     const y = 2 * Math.random() - 1;
57     const rotation = 2 * Math.PI * Math.random();
58     const uniforms = {};
59
60     uniforms.u_color = [
61       Math.random(),
62       Math.random(),
63       Math.random(),
64       1];
65
66     uniforms.xy = [
67       2 * Math.random() - 1,
68       2 * Math.random() - 1,
69     ];
70
71     uniforms.bias = Math.random();
72
73     meshData.push({
74       positions,
75       cells,
76       uniforms,
77     });
78   }
79   renderer.uniforms.uTime = 0;
80   renderer.setMeshData(meshData);
81 }
82 init();
83
84 function update(t) {
85   renderer.uniforms.uTime = t;
86   renderer.render();
87   requestAnimationFrame(update);
88 }
89
90 update(0);
```

上面的代码非常简单，关键思路就是用 circle 生成顶点信息，然后对每个需要绘制的圆应用 circle 顶点信息，并设置不同的 unifom 参数，最后在 shader 中根据参数进行绘制就可以了。

不过如果我们这么做的话，整体的性能就会非常低，比如在绘制 500 个圆的时候，浏览器的帧率就掉到十几 fps 了。那我们该怎么优化呢？



优化大数据渲染的常见方法

我们通过前面的学习已经知道渲染次数和每次渲染的顶点计算次数是影响渲染性能的因素，所以优化大数据渲染的思路方向自然就是减少渲染次数和减少几何体顶点数了。

1. 使用批量渲染优化

在学完 Canvas 和 WebGL 的性能优化之后，我们知道，在绘制大量同种几何图形的时候，通过减少渲染次数来提升性能最好的做法是直接使用批量渲染。针对今天的例子也是一样，我们稍微修改一下上面的代码，用实例渲染来代替逐个渲染，代码如下：

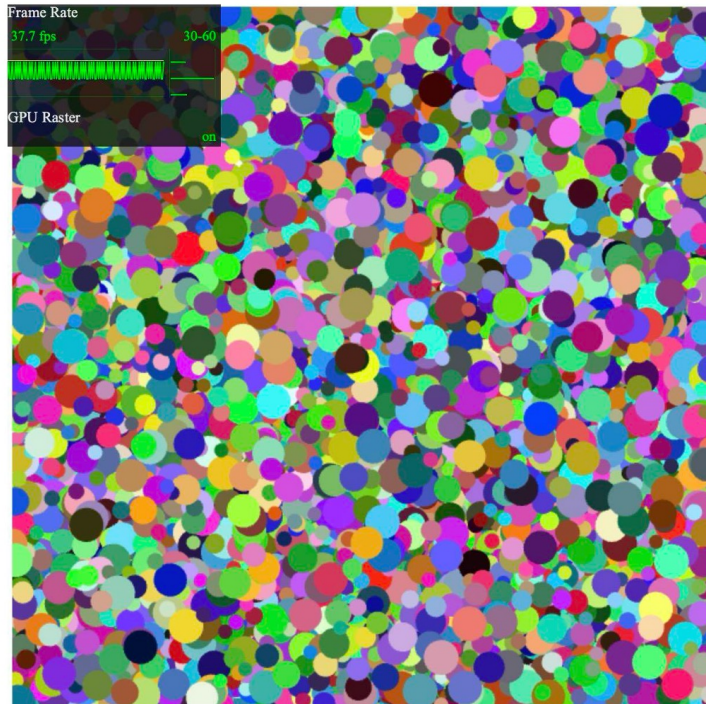
复制代码

```
1  const canvas = document.querySelector('canvas');
2  const renderer = new GLRenderer(canvas);
3
4  const vertex = `
5    attribute vec2 a_vertexPosition;
6    attribute vec4 color;
7    attribute vec2 xy;
8    attribute float bias;
9    uniform float uTime;
10
11    varying vec4 vColor;
12
13    void main() {
14        vec3 pos = vec3(a_vertexPosition, 1);
15        float scale = 0.7 + 0.3 * sin(6.28 * bias + 0.003 * uTime);
```

```
16     mat3 m = mat3(
17         scale, 0, 0,
18         0, scale, 0,
19         xy, 1
20     );
21     vColor = color;
22     gl_Position = vec4(m * pos, 1);
23 }
24 `;
25
26 const fragment = `
27 #ifdef GL_ES
28 precision highp float;
29 #endif
30
31
32 varying vec4 vColor;
33
34 void main() {
35     gl_FragColor = vColor;
36 }
37 `;
38 const program = renderer.compileSync(fragment, vertex);
39 renderer.useProgram(program);
40
41 function circle(radius = 0.05) {
42     const delta = 2 * Math.PI / 32;
43     const positions = [];
44     const cells = [];
45     for(let i = 0; i < 32; i++) {
46         const angle = i * delta;
47         positions.push([radius * Math.sin(angle), radius * Math.cos(angle)]);
48         if(i > 0 && i < 31) {
49             cells.push([0, i, i + 1]);
50         }
51     }
52     return {positions, cells};
53 }
54
55 const COUNT = 200000;
56 function init() {
57     const {positions, cells} = circle();
58     const colors = [];
59     const pos = [];
60     const bias = [];
61     for(let i = 0; i < COUNT; i++) {
62         const x = 2 * Math.random() - 1;
63         const y = 2 * Math.random() - 1;
64         const rotation = 2 * Math.PI * Math.random();
65
66         colors.push([
67             Math.random(),
```

```
68     Math.random(),
69     Math.random(),
70     1
71   ]);
72
73   pos.push([
74     2 * Math.random() - 1,
75     2 * Math.random() - 1
76   ]);
77
78   bias.push(
79     Math.random()
80   );
81 }
82
83 renderer.uniforms.uTime = 0;
84 renderer.setMeshData({
85   positions,
86   cells,
87   instanceCount: COUNT,
88   attributes: {
89     color: {data: [...colors], divisor: 1},
90     xy: {data: [...pos], divisor: 1},
91     bias: {data: [...bias], divisor: 1},
92   },
93 });
94 }
95 init();
96
97 function update(t) {
98   renderer.uniforms.uTime = t;
99   renderer.render();
100   requestAnimationFrame(update);
101 }
102
103 update(0);
```

你可以比较一下上面的代码和前一个例子代码的差异，这里我们使用实例渲染将之前的 uniform 变量替换成 attribute 变量，其他的逻辑几乎不变。我们这么做了之后，即使渲染 100000 个点，浏览器的帧率也能达到 30fps 以上，性能提升了超过 2000 倍！



之所以批量渲染的性能比逐个渲染要高得多，是因为我们通过减少绘制次数，大大减少了 JavaScript 与 WebGL 底层交互的时间。不过，使用批量渲染绘制 20000 个点，就达到我们的性能极限了吗？显然没有，我们还可以运用其他的优化手段。

2. 使用点图元优化

绘制规则的图形，我们还可以使用点图元。还记得吗？我们说过 WebGL 的基本图元包括点、线、三角形等等。前面我们绘制圆的时候，都是用 circle 函数生成三角网格，然后通过三角形绘制的。这样，我们绘制一个圆需要许多顶点。但实际上，这种简单的图形，我们还可以直接采用点图元。

点图元造型

在 WebGL 中，点图元是最简单的图元，它用来显示画布上的点。在顶点着色器里，我们可以设置 `gl_PointSize` 来改变点图元的大小，所以我们就可以用点图元来表示一个矩形。我们看下面这个例子。

📄 复制代码

```
1 const canvas = document.querySelector('canvas');
2 const renderer = new GLRenderer(canvas);
3
4 const vertex = `
5     attribute vec2 a_vertexPosition;
```



```
6   uniform vec2 uResolution;
7
8   void main() {
9       gl_PointSize = 0.2 * uResolution.x;
10      gl_Position = vec4(a_vertexPosition, 1, 1);
11  }
12 `;
13
14 const fragment = `
15     #ifdef GL_ES
16     precision highp float;
17     #endif
18
19     void main() {
20         gl_FragColor = vec4(0, 0, 1, 1);
21     }
22 `;
23 const program = renderer.compileSync(fragment, vertex);
24 renderer.useProgram(program);
25
26 renderer.uniforms.uResolution = [canvas.width, canvas.height];
27 renderer.setMeshData({
28     mode: renderer.gl.POINTS,
29     positions: [[0, 0]],
30 });
31
32 renderer.render();
```

如上面代码所示，我们将 meshData 的 mode 设为 gl.POINTS，只绘制一个点 (0, 0)。在顶点着色器中，我们通过 gl_PointSize 来设置顶点的大小。由于 gl_PointSize 的单位是像素，所以我们需要传一个画布宽高 uResolution 进去，然后将 gl_Position 设为 0.2 * uResolution，这就让这个点的大小设为画布的 20%，最终在画布上就呈现出一个蓝色矩形。



注意，这里你可以回顾一下之前我们采用的常规的方法绘制的矩形，我们是将矩形剖分为两个三角形，然后用填充三角形来绘制的。而这里，我们用点图元，好处是我们只需要一个顶点就可以绘制，而不需要用四个顶点、两个三角形来填充。

现在，我们通过点图元，改变 `gl_PointSize` 绘制出了矩形，那怎么才能绘制出其他图形呢？实际上，答案在我们前面学过的课程里——使用距离场和造型函数。

在上面例子的基础上，我们修改一下顶点着色器和片元着色器。具体代码如下：

首先是顶点着色器代码。

 复制代码

```
1 attribute vec2 a_vertexPosition;
2
3 uniform vec2 uResolution;
4 varying vec2 vResolution;
5 varying vec2 vPos;
6
7 void main() {
8     gl_PointSize = 0.2 * uResolution.x;
9     vResolution = uResolution;
10    vPos = a_vertexPosition;
11    gl_Position = vec4(a_vertexPosition, 1, 1);
12 }
```

然后是片元着色器代码。


[复制代码](#)

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vResolution;
6  varying vec2 vPos;
7
8  void main() {
9      vec2 st = gl_FragCoord.xy / vResolution;
10     st = 2.0 * st - 1.0;
11     float d = distance(st, vPos);
12     d = 1.0 - smoothstep(0.195, 0.2, d);
13     gl_FragColor = d * vec4(0, 0, 1, 1);
14 }
```

经过前面课程的学习，你应该对造型函数的实现原理比较熟悉了，这里我们就是通过计算到圆心的距离得出距离场，然后通过 smoothstep 将一定距离内的图形绘制出来，这样就得到一个蓝色的圆。



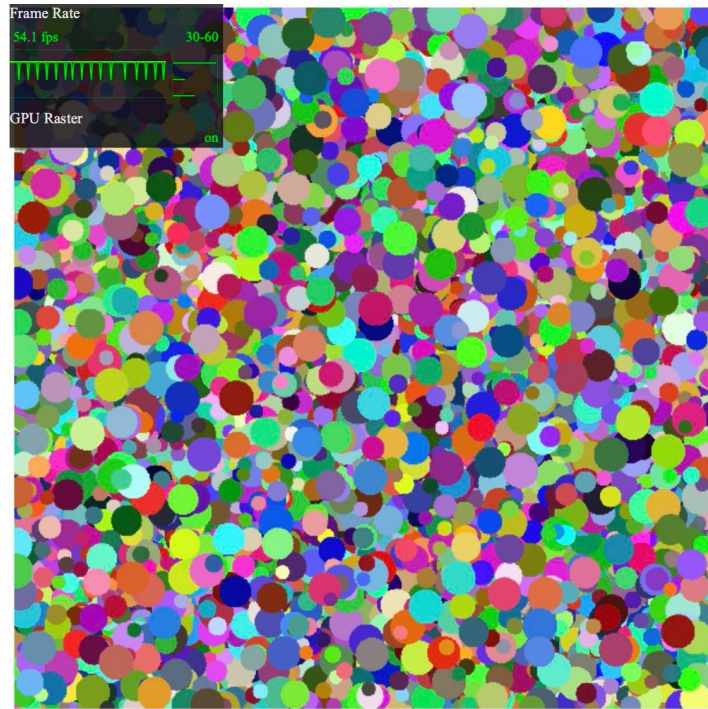
用这样的思路呢，我们就可以得到新的绘制大量圆的方法了。这种思路实现圆的代码如下：

 复制代码

```
1  const canvas = document.querySelector('canvas');
2  const renderer = new GLRenderer(canvas);
3
4  const vertex = `
5      attribute vec2 a_vertexPosition;
6      attribute vec4 color;
7      attribute float bias;
8
9      uniform float uTime;
10     uniform vec2 uResolution;
11
12     varying vec4 vColor;
13     varying vec2 vPos;
14     varying vec2 vResolution;
15     varying float vScale;
16
17     void main() {
18         float scale = 0.7 + 0.3 * sin(6.28 * bias + 0.003 * uTime);
19         gl_PointSize = 0.05 * uResolution.x * scale;
20         vColor = color;
21         vPos = a_vertexPosition;
22         vResolution = uResolution;
23         vScale = scale;
24         gl_Position = vec4(a_vertexPosition, 1, 1);
25     }
26 `;
27
28
29 const fragment = `
30     #ifdef GL_ES
31     precision highp float;
32     #endif
33
34     varying vec4 vColor;
35     varying vec2 vPos;
36     varying vec2 vResolution;
37     varying float vScale;
38
39     void main() {
40         vec2 st = gl_FragCoord.xy / vResolution;
41         st = 2.0 * st - vec2(1);
42         float d = step(distance(vPos, st), 0.05 * vScale);
43         gl_FragColor = d * vColor;
44     }
45 `;
46 const program = renderer.compileSync(fragment, vertex);
47 renderer.useProgram(program);
48
49 const COUNT = 200000;
50 function init() {
```

```
51  const colors = [];  
52  const pos = [];  
53  const bias = [];  
54  for(let i = 0; i < COUNT; i++) {  
55      const x = 2 * Math.random() - 1;  
56      const y = 2 * Math.random() - 1;  
57      const rotation = 2 * Math.PI * Math.random();  
58  
59      colors.push([  
60          Math.random(),  
61          Math.random(),  
62          Math.random(),  
63          1  
64      ]);  
65  
66      pos.push([  
67          2 * Math.random() - 1,  
68          2 * Math.random() - 1  
69      ]);  
70  
71      bias.push(  
72          Math.random()  
73      );  
74  }  
75  
76  renderer.uniforms.uTime = 0;  
77  renderer.uniforms.uResolution = [canvas.width, canvas.height];  
78  
79  renderer.setMeshData({  
80      mode: renderer.gl.POINTS,  
81      enableBlend: true,  
82      positions: pos,  
83      attributes: {  
84          color: {data: [...colors]},  
85          bias: {data: [...bias]},  
86      },  
87  });  
88  }  
89  init();  
90  
91  
92  function update(t) {  
93      renderer.uniforms.uTime = t;  
94      renderer.render();  
95      requestAnimationFrame(update);  
96  }  
97  
98  update(0);
```


可以看到，我们没有采用前面那样通过 `circle` 函数来生成圆的顶点数据，而是直接使用 `gl.POINTS` 来绘制，并在着色器中用距离场和造型函数来画圆。这么做之后，我们大大减少了顶点的运算，原先我们每绘制一个圆，需要 32 个顶点、30 个三角形，而现在用一个点就解决了问题。这样一来，就算我们要渲染 200000 个点，帧率也可以保持在 50fps 以上，性能又提升了超过一倍！



其他方法

这里举上面的这个例子，主要是想说明一个问题：即使是使用 WebGL，不同的渲染方式，性能的差别也会很大，甚至会达到数千倍的差别。因此，在可视化业务中，我们一定要学会**根据不同的应用场景来有针对性地进行优化**。说起来简单，要做到这一点并不容易，你需要对 WebGL 本身非常熟悉，而且对于 GPU 的使用、渲染管线等基本原理有着比较深刻的理解。这不是一朝一夕可以做到的，需要持续不断地学习和积累。

就像有些同学使用绘图库 ThreeJS 或者 SpriteJS 来绘图的时候，做出来的应用性能很差，就会怀疑是图形库本身的问题。实际上，这些问题很可能不是库本身的问题，而是我们使用方法上的问题。换句话说，是我们使用的绘图方式并不是最适用于当前的业务场景。而 ThreeJS、SpriteJS 这些通用的绘图库，也并不会自己针对特定场景来优化。

因此，单纯使用图形库，我们绘制出来的图形就没法真正达到性能极致。也正是因为这个原因，我没有把这门课程的重点放在库的 API 的使用上，而是深入到图形渲染的底层原

理。只有掌握了这些，你才能真正学会如何驾驭图形库，做出高性能的可视化解决方案来。

针对场景的性能优化方法其实非常多，我刚才讲的也只是几种典型的情况。为了帮助你在实战中慢慢领悟，我再举几个例子。不过我要提前说一下，我不会具体去讲这些例子的代码，只会重点强调常用的思路。学会这些方法之后，你再在实践中慢慢应用和体会就会容易很多了。

1. 使用后期处理通道优化

我们已经学习过使用后期处理通道的基本方法。实际上，后期处理通道十分强大，它最重要的特性就是可以把各种数据存储在纹理图片中。这样在迭代处理的时候，我们就可以用 GPU 将这些数据并行地读取和处理，从而达到非常高效地渲染。

这里是一个 OGL 官网上 [例子](#)，它就是用后期处理通道实现了粒子流的效果。这样的效果，在其他图形系统中，或者 WebGL 不使用后期处理通道是不可能做到的。



这里的具体实现比较复杂，但其中最关键的一点是，我们要将每个像素点的速度值保存到纹理图片中，然后利用 GPU 并行计算的能力，对每个像素点同时进行处理。

2. 使用 GPGPU 优化

还有一种优化思路和后期处理通道很像，刚好 OGL 官网也有这个例子，它使用了一种叫做 GPGPU 的方式，也叫做通用 GPU 方式，就是把每个粒子的速度保存到纹理图片里，实现同时渲染几万个粒子并产生运动的效果。



3. 使用服务端渲染优化

最后一种优化思路，是从我之前做过的一个可视化项目中提取出来的。当时，我需要渲染数十万条历史数据的记录，如果单纯在前端渲染，性能会成为瓶颈。但由于这些数据都是历史数据，因此针对这个场景我们可以在服务端进行渲染，然后直接将渲染后的图片输出给前端。

要使用服务端渲染，我们可以使用 [Node-canvas-webgl](#) 这个库，它可以在 Node.js 中启动一个 Canvas2D 和 WebGL 环境，这样我们就可以在服务端进行渲染，然后再将结果缓存起来直接提供给客户端。

要点总结

这节课我们主要讲了针对业务的不同应用场景进行性能优化的思路和方法。尤其是在海量数据的情况下，特定优化手段显得十分重要，甚至有可能产生上千倍的性能差距。

结合今天的例子，对于要绘制大量圆形的场景，我们用常规的处理方法，渲染 500 个元素都比较吃力，一旦我们使用了批量渲染，就可以把性能一下子提升两千倍以上，能够轻松渲染 10 万个元素，如果我们再使用点图元结合造型函数的方法，就能轻松渲染 20 万个以上的元素了。像这样的优化方法，需要我们理解业务场景，并对 WebGL 和 GPU、渲染管线等有深入的理解，再在项目实践中慢慢积累。

除此以外，我们还简单介绍了其他的一些优化手段，包括使用后期处理通道、使用 GPGPU 和使用服务端渲染。你可以结合我给出的例子去深入理解。

小试牛刀

今天，我们使用点图元结合造型函数的思路，绘制了正方形和圆，你还能绘制出其他不同图形吗？比如圆、正方形、菱形、花瓣、苹果或葫芦等等。

欢迎在留言区和我讨论，分享你的答案和思考，也欢迎你把这节课分享给你的朋友，我们下节课见！

源码

[🔗 课程中完整示例代码](#)

推荐阅读

[1] [🔗 Post Fluid Distortion](#)

[2] [🔗 GPGPU Particles \(General-Purpose computing on Graphics Processing Units\)](#)

[3] [🔗 Node-canvas-webgl](#)

提建议

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 怎么给WebGL绘制加速？

下一篇 32 | 数据之美：如何选择合适的方法对数据进行可视化处理？

精选留言 (1)

写留言



蛮好蛮开心

2020-09-11

厉害👍

展开

