



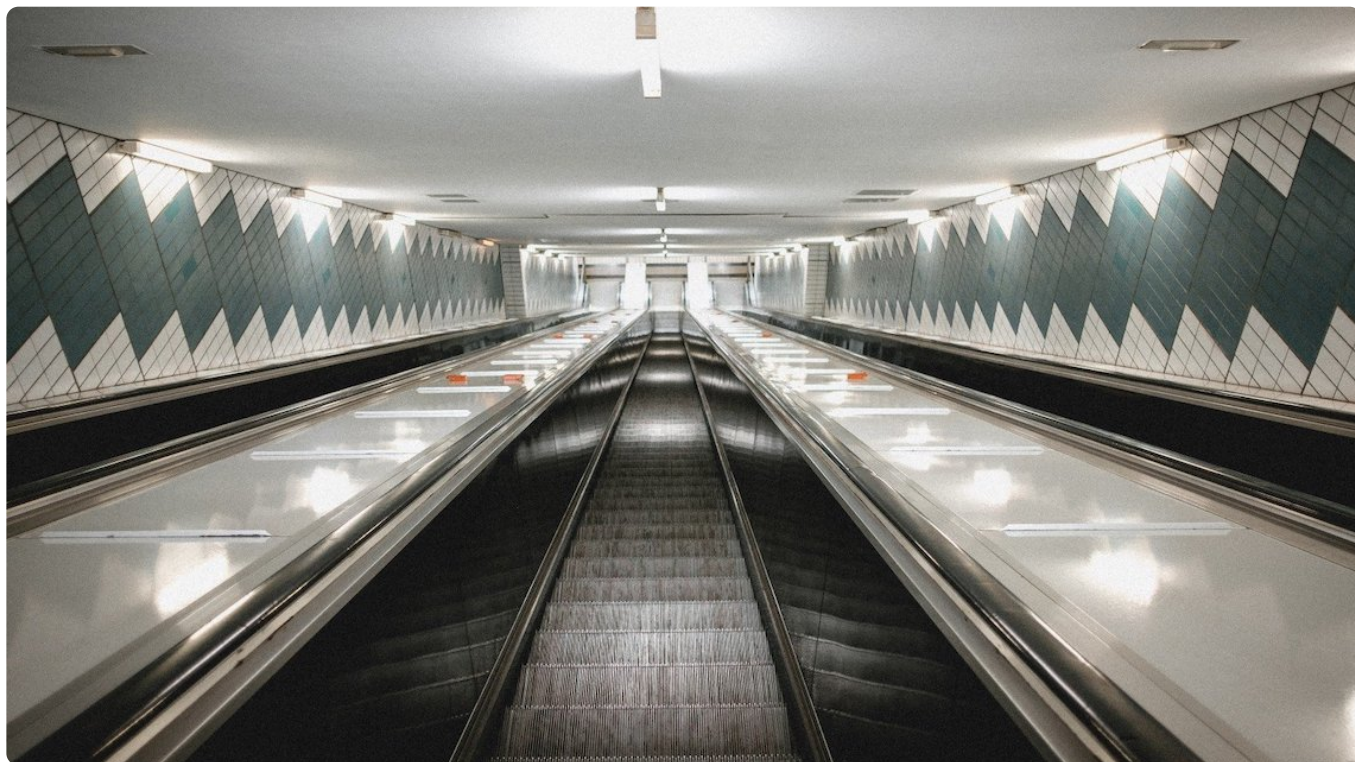
下载APP



## 30 | 怎么给WebGL绘制加速?

2020-08-31 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 15:29 大小 14.19M



你好，我是月影。这节课，我们一起来讨论 WebGL 的性能优化。

WebGL 因为能够直接操作 GPU，性能在各个图形系统中是最好的，尤其是当渲染的元素特别多的时候，WebGL 的性能优势越明显。但是，WebGL 整体性能好，并不意味着我们用 WebGL 就能写出高性能代码来。如果使用方式不当，也不能充分发挥 WebGL 在性能方面的优势。

这节课，我们就重点来说说，怎么充分发挥 WebGL 的优势，让它保持高性能。



### 尽量发挥 GPU 的优势

首先，我们来想一个问题，WebGL 的优势是什么？没错，我强调过很多遍，就是直接操作 GPU。因此，我们只有尽量发挥出 GPU 的优势，才能让 WebGL 保持高性能。但这一点是很多习惯用 SVG、Canvas 的 WebGL 初学者，最容易忽视的。

为了让你体会到发挥 GPU 优势的重要性，我们先来看一个没有进行任何优化的绘图例子，再对它进行优化。

## 常规绘图方式的性能瓶颈

假设，我们要在一个画布上渲染 3000 个不同颜色的、位置随机的三角形，并且让每个三角形的旋转角度也随机。

常规的实现方法当然是用 JavaScript 来创建随机三角形的顶点，然后依次渲染。我在创建随机三角形顶点的时候，是使用向量角度旋转的方法创建了正三角形，我想这个方法你应该也不会陌生。

[复制代码](#)

```
1 function randomTriangle(x = 0, y = 0, rotation = 0.0, radius = 0.1) {
2   const a = rotation,
3     b = a + 2 * Math.PI / 3,
4     c = a + 4 * Math.PI / 3;
5
6   return [
7     [x + radius * Math.sin(a), y + radius * Math.cos(a)],
8     [x + radius * Math.sin(b), y + radius * Math.cos(b)],
9     [x + radius * Math.sin(c), y + radius * Math.cos(c)],
10  ];
11 }
```

然后，我在下面代码的 for 循环中依次渲染每个三角形。

[复制代码](#)

```
1 const COUNT = 3000;
2 function render() {
3   for(let i = 0; i < COUNT; i++) {
4     const x = 2 * Math.random() - 1;
5     const y = 2 * Math.random() - 1;
6     const rotation = 2 * Math.PI * Math.random();
7
8     renderer.uniforms.u_color = [
9       Math.random(),
```

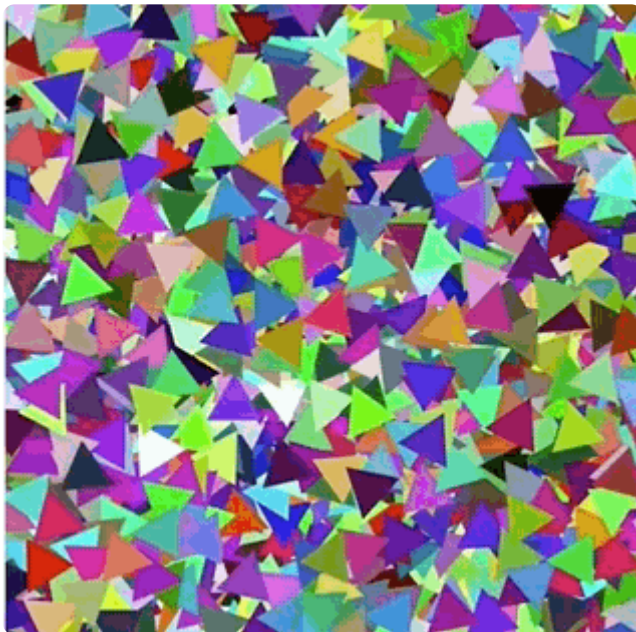
```
10     Math.random(),
11     Math.random(),
12     1];
13
14     const positions = randomTriangle(x, y, rotation);
15     renderer.setMeshData([[
16         positions,
17     ]]);
18
19     renderer._draw();
20 }
21 requestAnimationFrame(render);
22 }
23
24 render();
```

这里，我们只给着色器传入了一个颜色参数，其他的运算都是在 JavaScript 中完成的，所以对应的着色器代码非常简单。代码如下：

[📄 复制代码](#)

```
1 // 顶点着色器
2 attribute vec2 a_vertexPosition;
3
4 void main() {
5     gl_Position = vec4(a_vertexPosition, 1, 1);
6 }
7
8 // 片元着色器
9 #ifdef GL_ES
10 precision highp float;
11 #endif
12
13 uniform vec4 u_color;
14
15 void main() {
16     gl_FragColor = u_color;
17 }
```

这样，我们就完成了渲染 3000 个随机三角形的功能，效果如下：




你会发现，这样实现的图形性能很一般，因为 3000 个三角形渲染在普通笔记本电脑上只有 20fps，这大概和 Canvas2D 渲染出来的性能差不多，可以说完全没能发挥出 WebGL 应有的优势。

那我们应该对哪些点进行优化，从而**尽量发挥出 GPU 的优势**呢？

## 减少 CPU 计算次数


首先，我们可以不用生成这么多个三角形。根据前面学过的知识，我们可以创建一个正三角形，然后通过视图矩阵的变化来实现绘制多个三角形，而视图矩阵可以放在顶点着色器中计算。这样，我们就只要在渲染每个三角形的时候更新视图矩阵就行了。

具体来说就是，我们直接生成一个正三角形顶点，并设置数据到缓冲区。

 复制代码


```
1  const alpha = 2 * Math.PI / 3;
2  const beta = 2 * alpha;
3
4  renderer.setMeshData({
5    positions: [
6      [0, 0.1],
7      [0.1 * Math.sin(alpha), 0.1 * Math.cos(alpha)],
8      [0.1 * Math.sin(beta), 0.1 * Math.cos(beta)],
9    ],
10  });
```

然后，我们用随机坐标和角度更新每个三角形的 modelMatrix 数据。

 复制代码

```
1  const COUNT = 3000;
2  function render() {
3    for(let i = 0; i < COUNT; i++) {
4      const x = 2 * Math.random() - 1;
5      const y = 2 * Math.random() - 1;
6      const rotation = 2 * Math.PI * Math.random();
7
8      renderer.uniforms.modelMatrix = [
9        Math.cos(rotation), -Math.sin(rotation), 0,
10       Math.sin(rotation), Math.cos(rotation), 0,
11       x, y, 1,
12     ];
13
14     renderer.uniforms.u_color = [
15       Math.random(),
16       Math.random(),
17       Math.random(),
18       1];
19
20     renderer._draw();
21   }
22   requestAnimationFrame(render);
23 }
24
25 render();
```

而位置和角度的计算，我们放到顶点着色器内完成，代码如下：

 复制代码

```
1  attribute vec2 a_vertexPosition;
2
3  uniform mat3 modelMatrix;
4
5  void main() {
6    vec3 pos = modelMatrix * vec3(a_vertexPosition, 1);
7    gl_Position = vec4(pos, 1);
8  }
```

这么做了之后，三角形渲染的 fps 会略有提升，因为我们通过在顶点着色器中并行矩阵运算减少了顶点计算的次数。不过，这个性能提升在最新的 chrome 浏览器下可能并不明



显，因为现在浏览器的 JavaScript 引擎的运算速度很快，尽管将顶点计算放到顶点着色器中进行了，性能差别也很微小。但不管怎么样，这种方法依然是可以提升性能的。

## 静态批量绘制（多实例绘制）


那有没有办法更大程度地提升性能呢？当然是有的。实际上，对于需要重复绘制的图形，最好的办法是使用批量绘制。重复图形的批量绘制，在 WebGL 中也叫做**多实例绘制** (Instanced Drawing)，它是一种减少绘制次数的技术。

在 WebGL 中，一个几何图形一般需要一次渲染，如果我们要绘制多个图形的话，因为每个图形的顶点、颜色、位置等属性都不一样，所以我们只能一一渲染，不能一起渲染。但是，如果几何图形的顶点数据都相同，颜色、位置等属性就都可以在着色器计算，那么我们就可以使用 WebGL 支持的多实例绘制方式，一次性地把所有的图形都渲染出来。

多实例绘制的代码，其实我们在第 28 课里已经见过了。这里，我们再看一个例子，帮你加深印象。

首先，我们也是创建三角形顶点数据，然后使用多实例绘制的方式传入数据。因为 gl-renderer 中已经封装好了多实例绘制的方法，我们只需要传入 instanceCount 表示要绘制的图形数量即可。在原生的 WebGL 中使用多实例绘制会稍微复杂一点，我们一般不会这么做，但如果你想要尝试一下，可以参考 [这篇文章](#)。

使用多实例绘制的代码如下：

 复制代码

```
1  const alpha = 2 * Math.PI / 3;
2  const beta = 2 * alpha;
3
4  const COUNT = 3000;
5  renderer.setMeshData({
6    positions: [
7      [0, 0.1],
8      [0.1 * Math.sin(alpha), 0.1 * Math.cos(alpha)],
9      [0.1 * Math.sin(beta), 0.1 * Math.cos(beta)],
10   ],
11   instanceCount: COUNT,
12   attributes: {
13     id: {data: [...new Array(COUNT).keys()], divisor: 1},
14   },
15 });
```

这样，我们就只需要每帧渲染一次就可以了。为了能在顶点着色器中完成图形的位置和颜色计算，我们传入了时间 `uTime` 参数。代码如下：

[复制代码](#)

```
1 function render(t) {
2   renderer.uniforms.uTime = t;
3   renderer.render();
4   requestAnimationFrame(render);
5 }
6
7 render(0);
```

对应的顶点着色器如下：

[复制代码](#)

```
1 attribute vec2 a_vertexPosition;
2 attribute float id;
3
4 uniform float uTime;
5
6 highp float random(vec2 co) {
7   highp float a = 12.9898;
8   highp float b = 78.233;
9   highp float c = 43758.5453;
10  highp float dt= dot(co.xy ,vec2(a,b));
11  highp float sn= mod(dt,3.14);
12  return fract(sin(sn) * c);
13 }
14
15 varying vec3 vColor;
16
17 void main() {
18   float t = id / 10000.0;
19   float alpha = 6.28 * random(vec2(uTime, 2.0 + t));
20   float c = cos(alpha);
21   float s = sin(alpha);
22
23   mat3 modelMatrix = mat3(
24     c, -s, 0,
25     s, c, 0,
26     2.0 * random(vec2(uTime, t)) - 1.0, 2.0 * random(vec2(uTime, 1.0 + t)) - 1
27   );
28   vec3 pos = modelMatrix * vec3(a_vertexPosition, 1);
29   vColor = vec3(
```

```
30     random(vec2(uTime, 4.0 + t)),
31     random(vec2(uTime, 5.0 + t)),
32     random(vec2(uTime, 6.0 + t))
33 );
34 gl_Position = vec4(pos, 1);
35 }
```

我们这么做了之后，每一帧的实际渲染次数（即 WebGL 执行 drawElements 的次数）从原来的 3000 减少到了只有 1 次，而且计算都放到着色器里，利用 GPU 并行处理了，因此性能提升了 3000 倍。而且，现在不要说 3000，哪怕是 6000 个三角形，帧率都可以轻松达到 60fps 了，是不是很厉害？

## 动态批量绘制

可是，我又要给你泼一盆冷水了。虽然在绘制大量图形的时候，使用多实例绘制是一种非常好的方式，但是多实例渲染也有局限性，那就是只能在绘制相同的图形时使用。

不过，如果是绘制不同的几何图形，只要它们使用同样的着色器程序，而且没有改变 uniform 变量，我们也还是可以将顶点数据先合并再渲染，以减少渲染次数。

这么说你可能还不太理解，我们一起来看一个例子。假设，我们现在不只显示正三角形，而是显示随机的正三角形、正方形和正五边形。最常规的实现方式和前面显示随机正三角形的例子类似，我们只要修改一下顶点生成的函数，根据不同的边数生成对应的正多边形就可以了。代码如下：

 复制代码

```
1 function randomShape(x = 0, y = 0, edges = 3, rotation = 0.0, radius = 0.1) {
2     const a0 = rotation;
3     const delta = 2 * Math.PI / edges;
4     const positions = [];
5     const cells = [];
6     for(let i = 0; i < edges; i++) {
7         const angle = a0 + i * delta;
8         positions.push([x + radius * Math.sin(angle), y + radius * Math.cos(angle)
9         if(i > 0 && i < edges - 1) {
10             cells.push([0, i, i + 1]);
11         }
12     }
13     return {positions, cells};
14 }
```



这样，我们就可以随机生成三、四、五、六边形，代码如下：

[复制代码](#)

```
1 const {positions, cells} = randomShape(x, y, 3 + Math.floor(4 * Math.random()))
2 renderer.setMeshData([
3   positions,
4   cells,
5 ]);
```

不过这个例子的性能就更差了，渲染完 3000 个图形之后只有大概 5fps。当然这是正常的，因为正四边形、正五边形、正六边形每个分别要用 2、3、4 个三角形，所以虽然要绘制 3000 个图形，但我们实际绘制的三角形数量要远多于 3000 个。

而且，因为这些图形的形状不同，所以我们就不能使用多实例绘制的方式了。这个时候，我们又该如何优化呢？

我们依然可以将顶点合并起来绘制。因为每个图形都是由顶点（positions）和索引（cells）构成的，所以我们可以批量创建图形，将这些图形的顶点和索引全部合并起来。

[复制代码](#)

```
1 function createShapes(count) {
2   const positions = new Float32Array(count * 6 * 3); // 最多6边形
3   const cells = new Int16Array(count * 4 * 3); // 索引数等于3倍顶点数-2
4
5   let offset = 0;
6   let cellsOffset = 0;
7   for(let i = 0; i < count; i++) {
8     const edges = 3 + Math.floor(4 * Math.random());
9     const delta = 2 * Math.PI / edges;
10
11     for(let j = 0; j < edges; j++) {
12       const angle = j * delta;
13       positions.set([0.1 * Math.sin(angle), 0.1 * Math.cos(angle), i], (offset
14       if(j > 0 && j < edges - 1) {
15         cells.set([offset, offset + j, offset + j + 1], cellsOffset);
16         cellsOffset += 3;
17       }
18     }
19     offset += edges;
20   }
21   return {positions, cells};
22 }
23
```

如上面代码所示，我们首先创建两个类型数组 `positions` 和 `cells`，我们可以假定所有的图形都是正六边形，算出要创建的类型数组的总长度。注意，这里我们用的是三维顶点而不是二维顶点，这并不是说我们要绘制的图形是 3D 图形，而是我们使用 `z` 轴来保存当前图形的 `id`，提供给着色器中的伪随机函数使用。

计算顶点的方式和前面一样，都用的是向量旋转的方法。值得注意的是，在计算索引的时候，我们只要将之前已经算过的几何图形顶点总数记录下来，保存到 `offset` 变量里，从 `offset` 值开始计算就可以了。

最终，`createShapes` 函数会返回一个包含几万个顶点和索引的几何体数据，然后我们把它一次性渲染出来就行了。

[复制代码](#)

```
1  const {positions, cells} = createShapes(COUNT);
2
3  renderer.setMeshData([
4    positions,
5    cells,
6  ]);
7
8  function render(t) {
9    renderer.uniforms.uTime = t;
10   renderer.render();
11   requestAnimationFrame(render);
12 }
13
14 render(0);
```

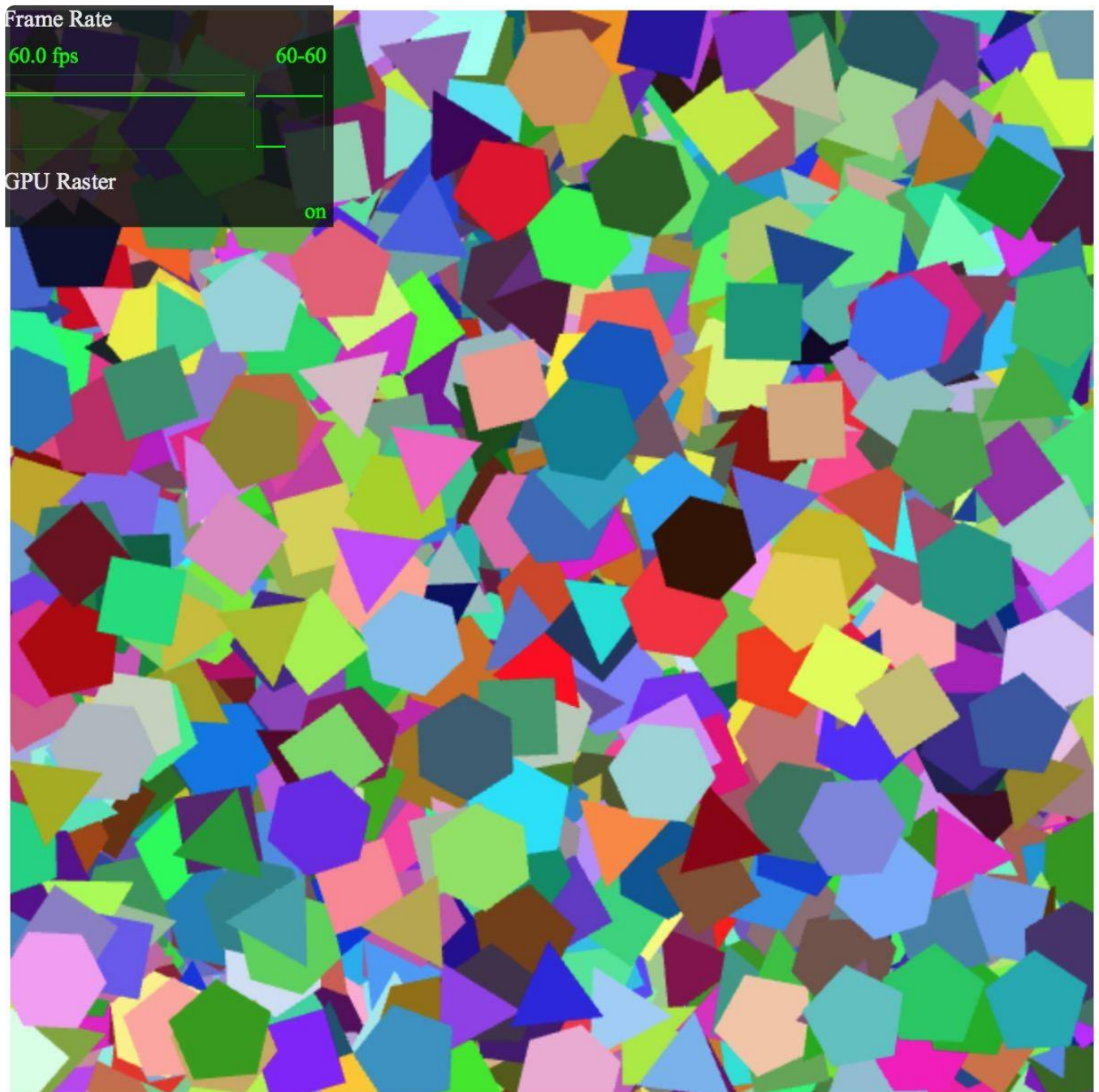
因为，对应的顶点着色器代码，与我们前面用多实例绘制的三角形例子差不多，只有一些微小的改动，所以你可以对比着看一下，加深理解。

[复制代码](#)

```
1  attribute vec3 a_vertexPosition;
2  uniform float uTime;
3
4  highp float random(vec2 co) {
5    highp float a = 12.9898;
6    highp float b = 78.233;
7    highp float c = 43758.5453;
```

```
8   highp float dt= dot(co.xy ,vec2(a,b));
9   highp float sn= mod(dt,3.14);
10  return fract(sin(sn) * c);
11 }
12
13 varying vec3 vColor;
14
15 void main() {
16     vec2 pos = a_vertexPosition.xy;
17     float t = a_vertexPosition.z / 10000.0;
18
19     float alpha = 6.28 * random(vec2(uTime, 2.0 + t));
20     float c = cos(alpha);
21     float s = sin(alpha);
22
23     mat3 modelMatrix = mat3(
24         c, -s, 0,
25         s, c, 0,
26         2.0 * random(vec2(uTime, t)) - 1.0, 2.0 * random(vec2(uTime, 1.0 + t)) - 1
27     );
28     vColor = vec3(
29         random(vec2(uTime, 4.0 + t)),
30         random(vec2(uTime, 5.0 + t)),
31         random(vec2(uTime, 6.0 + t))
32     );
33     gl_Position = vec4(modelMatrix * vec3(pos, 1), 1);
34 }
```

采用动态批量绘制之后，之前不到 5fps 的帧率，就被我们轻松提升到了 60fps。



批量渲染几乎是 WebGL 绘制最大的优化手段，因为它充分发挥了 GPU 的优势，所以能极大地提升性能。因此，在实际的 WebGL 项目中，如果我们遇到性能瓶颈，第一步就是要看看绘制的几何图形有哪些是可以批量渲染的，如果能批量渲染的，要尽量采用批量渲染，以减少一帧中的绘制次数。

不过批量渲染也有局限性，如果我们绘制的图形必须要用到不同的 WebGLProgram，或者每个图形要用到不同的 uniform 变量，那么它们就无法合并渲染。因此，我们在设计程序的时候，要尽量避免 WebGLProgram 切换，以及 uniform 的修改。

另外，在前面两个例子中，我们将 id 传入着色器，然后根据 id 在着色器中用伪随机函数计算位置和颜色。这样的好处自然是渲染起来特别快，但坏处是这些数据是在着色器中计

算出来的，如果我们想从 JavaScript 中拿到一些有用信息，比如，图形的位置、颜色等等，就很难拿到了。

因此，如果业务中需要用到这些信息，我们就不能将它们放在着色器中计算。当然，我们可以通过 JavaScript 来计算位置和颜色信息，然后把它们写到 attribute 中。不过这样的话，我们使用的内存消耗就会增加一些，而且用 JavaScript 计算这些值的过程会比在着色器中略慢。当然这也是因为项目需求不得不做出的选择。

## 其他优化手段

好了，对性能影响最大的批量绘制我们讲完了。其实，还有两个因素对性能也有影响，分别是透明与反锯齿和 Shader 效率。下面，我也简单介绍一下。由于这些因素影响性能的原理相对比较简单，我就不举例来说了，你可以自己实践一下来加深理解。

## 透明度与反锯齿

首先，是透明与反锯齿。在 WebGL 中，我们要处理半透明图形，可以开启混合模式 (Blending Mode) 让透明度生效。只有这样，WebGL 才会根据 Alpha 通道值和图形的层叠关系正确渲染并合成出叠加的颜色值。开启混合模式的代码如下：

```
1 gl.enable(gl.BLEND);
```

[复制代码](#)

不过，混合颜色本身有计算量，所以开启混合模式会造成一定的性能开销。因此，如果不需要处理半透明图形，我们尽量不开启混合模式，这样性能好就会更好一些。

此外，WebGL 本身对图形有反锯齿的优化，反锯齿可以避免图形边缘在绘制时出现锯齿，当然反锯齿本身也会带来性能开销。因此，如果对反锯齿的要求不高，我们在获取 WebGL 上下文时，关闭反锯齿设置也能减少开销、提升渲染性能。

```
1 const gl = canvas.getContext('webgl', {antiAlias: false}); //不消除反锯齿
```

[复制代码](#)

## Shader 的效率




最后，Shader 的效率也是我们在使用 WebGL 时需要注意的。我们前面说过，为了尽可能合并数据，动态批量绘制图形，我们要求图形尽量使用同一个 WebGLProgram，并且避免在绘制过程中切换 WebGLProgram。

但如果不同图形的绘制都使用同一个 WebGLProgram，这也会造成着色器本身的代码逻辑复杂，从而影响 Shader 的效率。最好的解决办法就是尽可能拆分不同的着色器代码，然后在绘制过程中根据不同元素进行切换。所以，批量绘制和简化 WebGLProgram 是一对矛盾，我们只能对两者进行取舍，尽可能让性能达到最优。

另外，shader 代码不同于常规的 JavaScript 代码，它最大的特性是并行计算，因此处理逻辑的过程与普通的代码不同。


那不同在哪儿呢？我们先来看一个常规的 JavaScript 代码。

 复制代码

```
1 if(Math.random() > 0.5) {  
2   do something  
3 } else {  
4   do something else  
5 }
```

我们都知道，如果 if 语句中的条件值为 true，那么第一个分支被执行，否则第二个分支被执行，这两个分支是不能同时被执行的。

但如果是 Shader 中的代码，情况就完全不同了。


 复制代码

```
1 if(random(st) > 0.5) {  
2   gl_FragColor = vec4(1)  
3 } else {  
4   gl_FragColor = vec4(0)  
5 }
```

无论是 if 还是 else 分支，在 glsl 中都会被执行，最终的值则根据条件表达式结果不同取不同分支计算的结果。



之所以会这样，就是因为 GPU 是并行计算的，也就是说并行执行大量 glsl 程序，但是每个子程序并不知道其他子程序的执行结果，所以最优的办法就是事先计算好 if 和 else 分支中的结果，再根据不同子程序的条件返回对应的结果。因此，if 语句必然要同时执行两个分支，但这样就会造成性能上一定的损耗，解决这个问题的办法是尽可能不用 if 语句。比如，对上面的代码，我们不用 if 语句，而是用 step 函数来解决问题，这样性能就会好一些。代码如下：

 复制代码

```
1 gl_FragColor = vec4(1) * step(random(st), 0.5);
```

此外，一些耗时的计算，比如开平方、反正切、反余弦等等，我们的优化原则也是能避免就尽可能避免，多使用简单的加法和乘法，这样就能保证着色器的高效率运行了。

## 要点总结

今天，我们重点讲了优化 WebGL 绘制性能的核心原则。

虽然 WebGL 是图形系统中渲染性能最高的，但如果我们不够了解 GPU，不对它进行有效的优化，就不能很好地发挥出 WebGL 的高性能优势。

用一句话总结，WebGL 的性能优化原则就是尽量发挥出 GPU 的优势。核心原则有两个：首先，我们尽量减少 CPU 计算次数，把能放在 GPU 中计算的部分放在 GPU 中并行计算；其次，也是更重要的，我们应该减少每一帧的绘制次数。

对应的优化方法也有两个：一是如果我们要绘制大量相同的图形，可以利用多实例渲染来实现静态批量绘制；二是如果绘制的图形不同，但是采用的 WebGL 程序相同、以及 uniform 的值没有改变，那我们可以人为合并顶点并进行渲染。减少绘制次数一般来说对性能会有比较明显的提升。

除此之外，我们还可以在不需要处理透明度的时候不启用混合模式，在不需要抗锯齿的时候关闭抗锯齿功能，它们都能减少性能开销。以及，我们还要注意 Shader 的效率，尽量用函数代替分支，避免一些耗时的计算，多使用简单的加法和乘法，这样能够保证着色器高效运行。

总的来说，性能优化是一个非常复杂的问题，我们应该结合实际项目的需求、数据的特征、技术方案等等综合考虑，最终才能得出最适合的方案。在实际项目中，无论你是直接用原生的 WebGL，还是使用 OGL、SpriteJS 或者 ThreeJS，大体的优化思路肯定离不开我前面总结的这些点。但怎么既恰到好处的优化，又保持性能与产品功能、开发效率以及扩展性的平衡，就需要我们通不断积累项目经验，才能慢慢做到最好啦。

## 小试牛刀

在前面的例子中，我们把位置和颜色的计算都放在了着色器中。这有利有弊，如果让你来重构代码，你能做到既兼顾性能，又能满足我们从 JavaScript 中拿到几何体位置和颜色的需求吗？如果可以，就快把你的解决方案写好分享出来吧。

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

[🔗 WebGL2 系列之实例数组 \(Instanced Arrays\)](#)

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 29 | 怎么给Canvas绘制加速？

下一篇 31 | 针对海量数据，如何优化性能？

## 精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。