



下载APP



## 17 | 如何使用后期处理通道增强图像效果？

2020-07-31 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 13:14 大小 12.13M



你好，我是月影。

前面几节课，我们学习了利用向量和矩阵公式，来处理像素和生成纹理的技巧，但是这些技巧都有一定的局限性：每个像素是彼此独立的，不能共享信息。

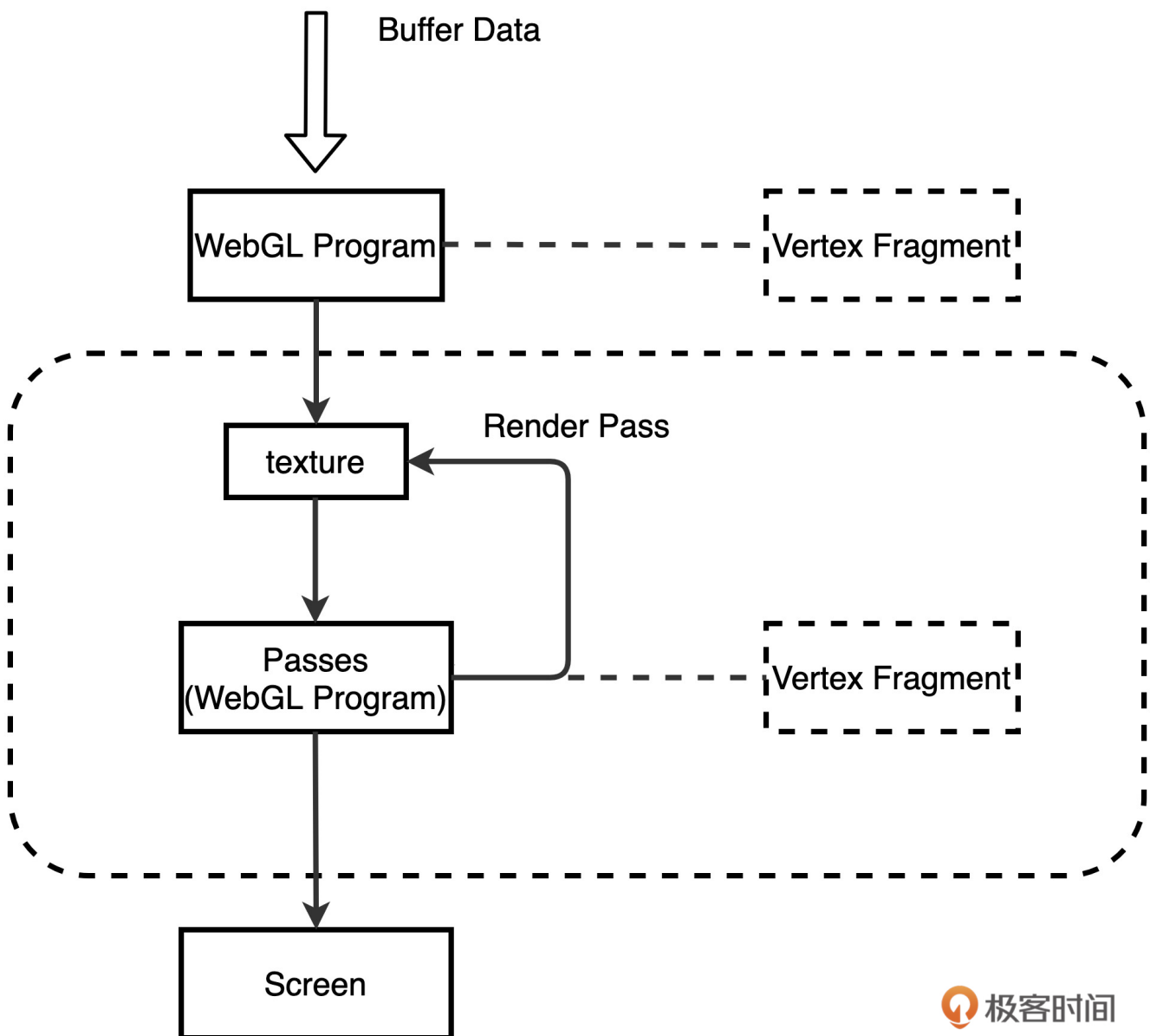
为什么这么说呢？因为 GPU 是并行渲染的，所以在着色器的执行中，每个像素的着色都是同时进行的。这样一来，我们就不能获得某一个像素坐标周围坐标点的颜色信息，也不能获得要渲染图像的全局信息。



这会导致什么问题呢？如果我们要实现与周围像素点联动的效果，比如给生成的纹理添加平滑效果滤镜，就不能直接通过着色器的运算来实现了。

因此，在 WebGL 中，像这样不能直接通过着色器运算来实现的效果，我们需要使用其他的办法来实现，其中一种办法就是使用**后期处理通道**。所谓后期处理通道，是指将渲染出来的图像作为纹理输入给新着色器处理，是一种二次加工的手段。这么一来，虽然我们不能从当前渲染中获取周围的像素信息，却可以从纹理中获取任意 uv 坐标下的像素信息，也就相当于可以获取任意位置的像素信息了。

使用后期处理通道的一般过程是，我们先正常地将数据送入缓冲区，然后执行 WebGLProgram。只不过，在执行了 WebGLProgram 之后，我们要将输出的结果再作为纹理，送入另一个 WebGLProgram 进行处理，这个过程可以进行一次，也可以循环多次。最后，经过两次 WebGLProgram 处理之后，我们再输出结果。



你可以先仔细看看这张流程总结图，加深一下印象。接下来，我会结合这个过程，说说怎么用后期处理通道，来实现 Blur 滤镜、辉光效果和烟雾效果，这样你就能理解得更深刻了。


首先，我们来实现 Blur 滤镜。

## 如何用后期处理通道实现 Blur 滤镜？

其实在第 11 节课中，我们已经在 Canvas2D 中实现了 Bblur 滤镜（高斯模糊的平滑效果滤镜），但 Canvas2D 实现滤镜的性能不佳，尤其是在图片较大，需要大量计算的时候。

而在 WebGL 中，我们可以通过后期处理来实现高性能的 Blur 滤镜。下面，我就以给随机三角形图案加 Blur 滤镜为例，来说说具体的操作。

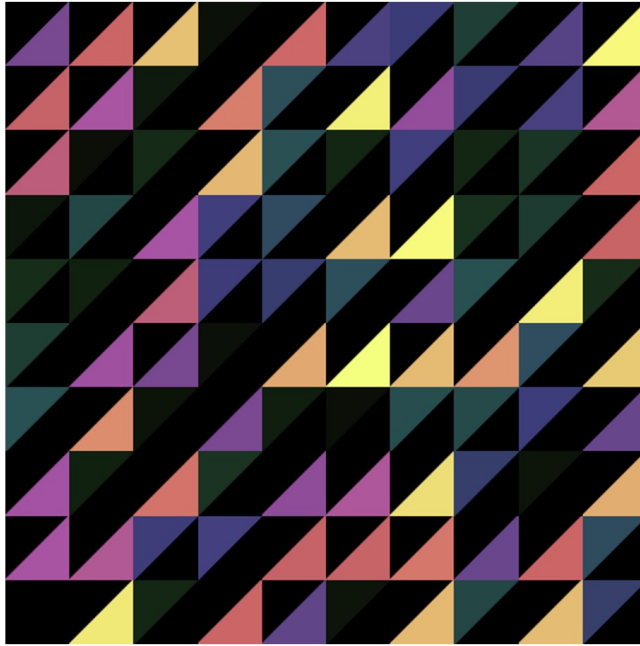
首先，我们实现一个绘制随机三角形图案的着色器。代码如下：

 复制代码

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  float line_distance(in vec2 st, in vec2 a, in vec2 b) {
6      vec3 ab = vec3(b - a, 0);
7      vec3 p = vec3(st - a, 0);
8      float l = length(ab);
9      return cross(p, normalize(ab)).z;
10 }
11
12 float seg_distance(in vec2 st, in vec2 a, in vec2 b) {
13     vec3 ab = vec3(b - a, 0);
14     vec3 p = vec3(st - a, 0);
15     float l = length(ab);
16     float d = abs(cross(p, normalize(ab)).z);
17     float proj = dot(p, ab) / l;
18     if(proj >= 0.0 && proj <= l) return d;
19     return min(distance(st, a), distance(st, b));
20 }
21
22 float triangle_distance(in vec2 st, in vec2 a, in vec2 b, in vec2 c) {
23     float d1 = line_distance(st, a, b);
24     float d2 = line_distance(st, b, c);
25     float d3 = line_distance(st, c, a);
26
27     if(d1 >= 0.0 && d2 >= 0.0 && d3 >= 0.0 || d1 <= 0.0 && d2 <= 0.0 && d3 <= 0.
```

```
28     return -min(abs(d1), min(abs(d2), abs(d3))); // 内部距离为负
29 }
30
31 return min(seg_distance(st, a, b), min(seg_distance(st, b, c), seg_distance(
32 }
33
34 float random (vec2 st) {
35     return fract(sin(dot(st.xy,
36                     vec2(12.9898,78.233))) *
37         43758.5453123);
38 }
39
40 vec3 hsb2rgb(vec3 c){
41     vec3 rgb = clamp(abs(mod(c.x*6.0+vec3(0.0,4.0,2.0), 6.0)-3.0)-1.0, 0.0, 1.0)
42     rgb = rgb * rgb * (3.0 - 2.0 * rgb);
43     return c.z * mix(vec3(1.0), rgb, c.y);
44 }
45
46 varying vec2 vUv;
47
48 void main() {
49     vec2 st = vUv;
50     st *= 10.0;
51     vec2 i_st = floor(st);
52     vec2 f_st = 2.0 * fract(st) - vec2(1);
53     float r = random(i_st);
54     float sign = 2.0 * step(0.5, r) - 1.0;
55
56     float d = triangle_distance(f_st, vec2(-1), vec2(1), sign * vec2(1, -1));
57     gl_FragColor.rgb = (smoothstep(-0.85, -0.8, d) - smoothstep(0.0, 0.05, d)) *
58     gl_FragColor.a = 1.0;
59 }
```

这个着色器绘制出的效果如下图：




接着就是重点了，我们要使用后期处理通道对它进行高斯模糊。

首先，我们需要准备另一个着色器：blurFragment。通过它，我们能将第一次渲染后生成的纹理 tMap 内容给显示出来。

复制代码

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vUv;
6  uniform sampler2D tMap;
7
8  void main() {
9      vec4 color = texture2D(tMap, vUv);
10     gl_FragColor.rgb = color.rgb;
11     gl_FragColor.a = color.a;
12 }
```

然后，我们要修改 JavaScript 代码，把渲染分为两次。第一次渲染时，我们启用 program 程序，但不直接把图形输出到画布上，而是输出到一个帧缓冲对象（Frame Buffer Object）上。第二次渲染时，我们再启用 blurProgram 程序，将第一次渲染完成的纹理（fbo.texture）作为 blurFragment 的 tMap 变量，这次的输出绘制到画布上。代码如下：

 复制代码

```
1  ...
2
3  renderer.useProgram(program);
4
5  renderer.setMeshData([ {
6      positions: [
7          [-1, -1],
8          [-1, 1],
9          [1, 1],
10         [1, -1],
11     ],
12     attributes: {
13         uv: [
14             [0, 0],
15             [0, 1],
16             [1, 1],
17             [1, 0],
18         ],
19     },
20     cells: [[0, 1, 2], [2, 0, 3]],
21 }]);
22
23 const fbo = renderer.createFBO();
24 renderer.bindFBO(fbo);
25 renderer.render();
26 renderer.bindFBO(null);
27
28 const blurProgram = renderer.compileSync(blurFragment, vertex);
29 renderer.useProgram(blurProgram);
30 renderer.setMeshData(program.meshData);
31 renderer.uniforms.tMap = fbo.texture;
32 renderer.render();
```

其中，`renderer.createFBO` 是创建帧缓冲对象，`bindFBO` 是绑定帧缓冲对象。为了方便调用，我在这里通过 `gl-renderer` 做了一层简单的封装。

那经过两次渲染之后，我们运行程序输出的结果和之前输出的并不会有什么区别。因为第二次渲染只不过是第一次渲染到帧缓冲的结果原封不动地输出到画布上了。

接下来，我们修改 `blurFragment` 的代码，在其中添加高斯模糊的代码。代码如下：

 复制代码

```
1  #ifdef GL_ES
2  precision highp float;
```

```
3  #endif
4
5  varying vec2 vUv;
6  uniform sampler2D tMap;
7  uniform int axis;
8
9  void main() {
10     vec4 color = texture2D(tMap, vUv);
11
12     // 高斯矩阵的权重值
13     float weight[5];
14     weight[0] = 0.227027;
15     weight[1] = 0.1945946;
16     weight[2] = 0.1216216;
17     weight[3] = 0.054054;
18     weight[4] = 0.016216;
19
20     // 每一个相邻像素的坐标间隔, 这里的512可以用实际的Canvas像素宽代替
21     float tex_offset = 1.0 / 512.0;
22
23     vec3 result = color.rgb;
24     result *= weight[0];
25     for(int i = 1; i < 5; ++i) {
26         float f = float(i);
27         if(axis == 0) { // x轴的高斯模糊
28             result += texture2D(tMap, vUv + vec2(tex_offset * f, 0.0)).rgb * weight[i];
29             result += texture2D(tMap, vUv - vec2(tex_offset * f, 0.0)).rgb * weight[i];
30         } else { // y轴的高斯模糊
31             result += texture2D(tMap, vUv + vec2(0.0, tex_offset * f)).rgb * weight[i];
32             result += texture2D(tMap, vUv - vec2(0.0, tex_offset * f)).rgb * weight[i];
33         }
34     }
35
36     gl_FragColor.rgb = result.rgb;
37     gl_FragColor.a = color.a;
38 }
39
```

因为高斯模糊有两个方向, x 和 y 方向, 所以我们至少要执行两次渲染, 一次对 x 轴, 另一次对 y 轴。如果想要达到更好的效果, 我们还可以执行多次渲染。

那我们就以分别对 x 轴和 y 轴执行 2 次渲染为例, 修改后的 JavaScript 代码如下:

[复制代码](#)

```
1  // 创建两个FBO对象交替使用
2  const fbo1 = renderer.createFBO();
3  const fbo2 = renderer.createFBO();
```



```
4 // 第一次，渲染原始图形
5 renderer.bindFBO(fbo1);
6 renderer.render();
7
8 // 第二次，对x轴高斯模糊
9 renderer.useProgram(blurProgram);
10 renderer.setMeshData(program.meshData);
11 renderer.bindFBO(fbo2);
12 renderer.uniforms.tMap = fbo1.texture;
13 renderer.uniforms.axis = 0;
14 renderer.render();
15
16 // 第三次，对y轴高斯模糊
17 renderer.useProgram(blurProgram);
18 renderer.bindFBO(fbo1);
19 renderer.uniforms.tMap = fbo2.texture;
20 renderer.uniforms.axis = 1;
21 renderer.render();
22
23 // 第四次，对x轴高斯模糊
24 renderer.useProgram(blurProgram);
25 renderer.bindFBO(fbo2);
26 renderer.uniforms.tMap = fbo1.texture;
27 renderer.uniforms.axis = 0;
28 renderer.render();
29
30 // 第五次，对y轴高斯模糊
31 renderer.useProgram(blurProgram);
32 renderer.bindFBO(null);
33 renderer.uniforms.tMap = fbo2.texture;
34 renderer.uniforms.axis = 1;
35 renderer.render();
36
```

在上面的代码中，我们创建了两个 FBO 对象，然后将它们交替使用。我们一共进行 5 次绘制，先对原始图片执行 1 次渲染，再进行 4 次后期处理。

这里啊，我还要告诉你一个小技巧。本来啊，在执行的这 5 次绘制中，前四次都是输出到帧缓冲对象，所以我们至少需要 4 个 FBO 对象。但是，由于我们可以交替使用 FBO 对象，也就是可以把用过的对象重复使用。因此，无论需要绘制多少次，我们都只要创建两个对象就可以，也就节约了内存。

最终，我们就能通过后期处理通道实现 Blur 滤镜，给三角形图案加上模糊的效果了。渲染结果如下：





## 如何用后期处理通道实现辉光效果？

在上面这个例子中，我们是对所有元素进行高斯模糊的。那除此之外，我们还可以对特定元素进行高斯模糊。在可视化和游戏开发中，就常用这种技巧来实现元素的“辉光”效果。比如，下面这张图就是用辉光效果实现的浩瀚宇宙背景。



全局辉光效果（图片来源于知乎：H光大小姐）

那类似这样的辉光效果该怎么实现呢？我们可以在前面添加高斯模糊例子的基础上进行修改。首先，我们给 blurFragment 加了一个关于亮度的滤镜，将颜色亮度大于 filter 值的三角形过滤出来添加高斯模糊。修改后的代码如下。

[复制代码](#)

```
1 uniform float filter;
2
3 void main() {
4     vec4 color = texture2D(tMap, vUv);
5     float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722));
6     brightness = step(filter, brightness);
7
8     // 高斯矩阵的权重值
9     float weight[5];
10    weight[0] = 0.227027;
11    weight[1] = 0.1945946;
12    weight[2] = 0.1216216;
13    weight[3] = 0.054054;
14    weight[4] = 0.016216;
15
16    // 每一个相邻像素的坐标间隔，这里的512可以用实际的Canvas像素宽代替
17    float tex_offset = 1.0 / 512.0;
18
19    vec3 result = color.rgb;
20    result *= weight[0];
21    for(int i = 1; i < 5; ++i) {
22        float f = float(i);
23        if(axis == 0) { // x轴的高斯模糊
24            result += texture2D(tMap, vUv + vec2(tex_offset * f, 0.0)).rgb * weight[
25            result += texture2D(tMap, vUv - vec2(tex_offset * f, 0.0)).rgb * weight[
26        } else { // y轴的高斯模糊
27            result += texture2D(tMap, vUv + vec2(0.0, tex_offset * f)).rgb * weight[
28            result += texture2D(tMap, vUv - vec2(0.0, tex_offset * f)).rgb * weight[
29        }
30    }
31
32    gl_FragColor.rgb = brightness * result.rgb;
33    gl_FragColor.a = color.a;
34 }
35
```

然后，我们再增加一个 bloomFragment 着色器，用来做最后的效果混合。这里，我们会用到一个叫做 [Tone Mapping](#)（色调映射）的方法。这个方法就比较复杂了，在这里，你只要知道它可以将对比度过大的图像色调映射到合理的范围内就可以了，其他的内容你可以在课后看一下我给出的参考链接。

这个着色器的代码如下：

[复制代码](#)

```
1  #ifdef GL_ES
2      precision highp float;
3  #endif
4
5  uniform sampler2D tMap;
6  uniform sampler2D tSource;
7
8  varying vec2 vUv;
9
10 void main() {
11     vec3 color = texture2D(tSource, vUv).rgb;
12     vec3 bloomColor = texture2D(tMap, vUv).rgb;
13     color += bloomColor;
14     // tone mapping
15     float exposure = 2.0;
16     float gamma = 1.3;
17     vec3 result = vec3(1.0) - exp(-color * exposure);
18     // also gamma correct while we're at it
19     if(length(bloomColor) > 0.0) {
20         result = pow(result, vec3(1.0 / gamma));
21     }
22     gl_FragColor.rgb = result;
23     gl_FragColor.a = 1.0;
24 }
```

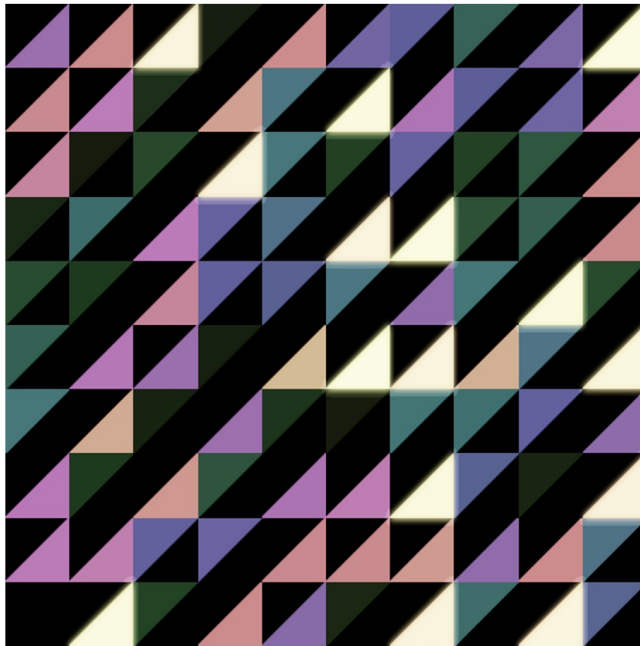
最后，我们修改 JavaScript 渲染的逻辑，添加新的后期处理规则。这里，我们要使用三个 FBO 对象，因为第一个 FBO 对象在渲染原始图形之后，还要在混合效果时使用，后两个对象是用来交替使用完成高斯模糊的。最后，我们再将原始图形和高斯模糊的结果进行效果混合就可以了。修改后的代码如下：

[复制代码](#)

```
1  // 创建三个FBO对象，fbo1和fbo2交替使用
2  const fbo0 = renderer.createFBO();
3  const fbo1 = renderer.createFBO();
4  const fbo2 = renderer.createFBO();
5
6  // 第一次，渲染原始图形
7  renderer.bindFBO(fbo0);
8  renderer.render();
9
10 // 第二次，对x轴高斯模糊
11 renderer.useProgram(blurProgram);
```

```
12 renderer.setMeshData(program.meshData);
13 renderer.bindFBO(fbo2);
14 renderer.uniforms.tMap = fbo0.texture;
15 renderer.uniforms.axis = 0;
16 renderer.uniforms.filter = 0.7;
17 renderer.render();
18
19 // 第三次, 对y轴高斯模糊
20 renderer.useProgram(blurProgram);
21 renderer.bindFBO(fbo1);
22 renderer.uniforms.tMap = fbo2.texture;
23 renderer.uniforms.axis = 1;
24 renderer.uniforms.filter = 0;
25 renderer.render();
26
27 // 第四次, 对x轴高斯模糊
28 renderer.useProgram(blurProgram);
29 renderer.bindFBO(fbo2);
30 renderer.uniforms.tMap = .texture;
31 renderer.uniforms.axis = 0;
32 renderer.uniforms.filter = 0;
33 renderer.render();
34
35 // 第五次, 对y轴高斯模糊
36 renderer.useProgram(blurProgram);
37 renderer.bindFBO(fbo1);
38 renderer.uniforms.tMap = fbo2.texture;
39 renderer.uniforms.axis = 1;
40 renderer.uniforms.filter = 0;
41 renderer.render();
42
43 // 第六次, 叠加辉光
44 renderer.useProgram(bloomProgram);
45 renderer.setMeshData(program.meshData);
46 renderer.bindFBO(null);
47 renderer.uniforms.tSource = fbo0.texture;
48 renderer.uniforms.tMap = fbo1.texture;
49 renderer.uniforms.axis = 1;
50 renderer.uniforms.filter = 0;
51
```

这样渲染之后, 我就能让三角形图案中几个比较亮的三角形, 产生一种微微发光的效果。渲染效果如下:



局部辉光效果示意图

这样，我们就实现了最终的局部辉光效果。实现它的关键，就是在高斯模糊原理的基础上，将局部高斯模糊的图像与原始图像叠加。

## 如何用后期处理通道实现烟雾效果？

除了模糊和辉光效果之外，后期处理通道还经常用来实现烟雾效果。接下来，我们就实现一个小圆的烟雾效果。具体的实现过程主要分为两步：第一步和前面两个例子一样，我们通过创建一个 shader，画出一个简单的圆。第二步，我们对这个圆进行后期处理，不过这次的处理方法就和实现辉光不同了。

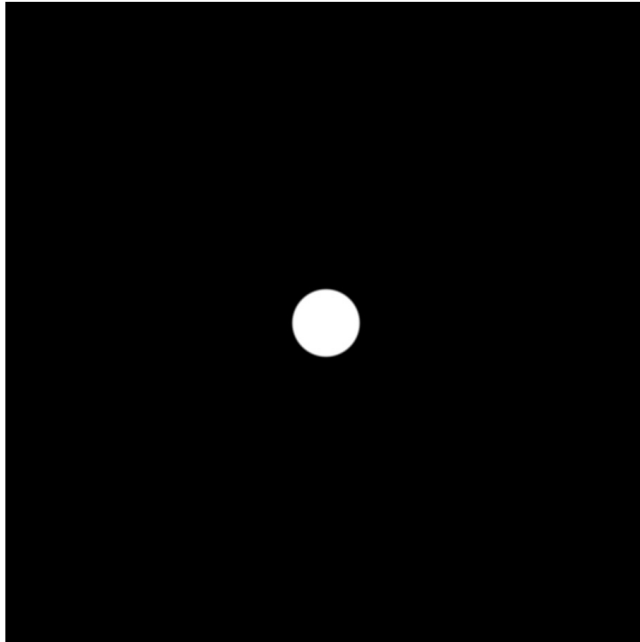
下面，我们就一起来看。

首先，我们创建一个简单的 shader，也就是使用距离场在画布上画一个圆。这个 shader 我们非常熟悉，具体的代码如下：

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vUv;
6
7  void main() {
```

[复制代码](#)

```
8   vec2 st = vUv - vec2(0.5);
9   float d = length(st);
10  gl_FragColor.rgb = vec3(1.0 - smoothstep(0.05, 0.055, d));
11  gl_FragColor.a = 1.0;
12 }
```



接着，我们修改一下 shader 代码，增加 uTime、tMap 这两个变量，代码如下所示。其中，uTime 用来控制图像随时间变化，而 tMap 是我们用来做后期处理的变量。

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vUv;
6  uniform sampler2D tMap;
7  uniform float uTime;
8
9  void main() {
10   vec3 smoke = vec3(0);
11   if(uTime <= 0.0) {
12     vec2 st = vUv - vec2(0.5);
13     float d = length(st);
14     smoke = vec3(1.0 - smoothstep(0.05, 0.055, d));
15   }
16   vec3 diffuse = texture2D(tMap, vUv).rgb;
17   gl_FragColor.rgb = diffuse + smoke;
18   gl_FragColor.a = 1.0;
```

[复制代码](#)

```
19 }
```

然后，我们依然创建两个 FBO，用它们交替进行绘制。最后，我们把绘制的内容输出到画布上。这里，我使用了一个 if 语句，根据绘制过程判断初始绘制还是后续的叠加过程，就能把着色器合并成一个。这样一来，不管是输出到画布还是 FBO，我们使用同一个 program 就可以了。

[复制代码](#)

```
1  const fbo = {
2    readFBO: renderer.createFBO(),
3    writeFBO: renderer.createFBO(),
4    get texture() {
5      return this.readFBO.texture;
6    },
7    swap() {
8      const tmp = this.writeFBO;
9      this.writeFBO = this.readFBO;
10     this.readFBO = tmp;
11   },
12 };
13
14 function update(t) {
15   // 输出到画布
16   renderer.bindFBO(null);
17   renderer.uniforms.uTime = t / 1000;
18   renderer.uniforms.tMap = fbo.texture;
19   renderer.render();
20   // 同时输出到FBO
21   renderer.bindFBO(fbo.writeFBO);
22   renderer.uniforms.tMap = fbo.texture;
23   // 交换读写缓冲以便下一次写入
24   fbo.swap();
25   renderer.render();
26   requestAnimationFrame(update);
27 }
28 update(0);
```

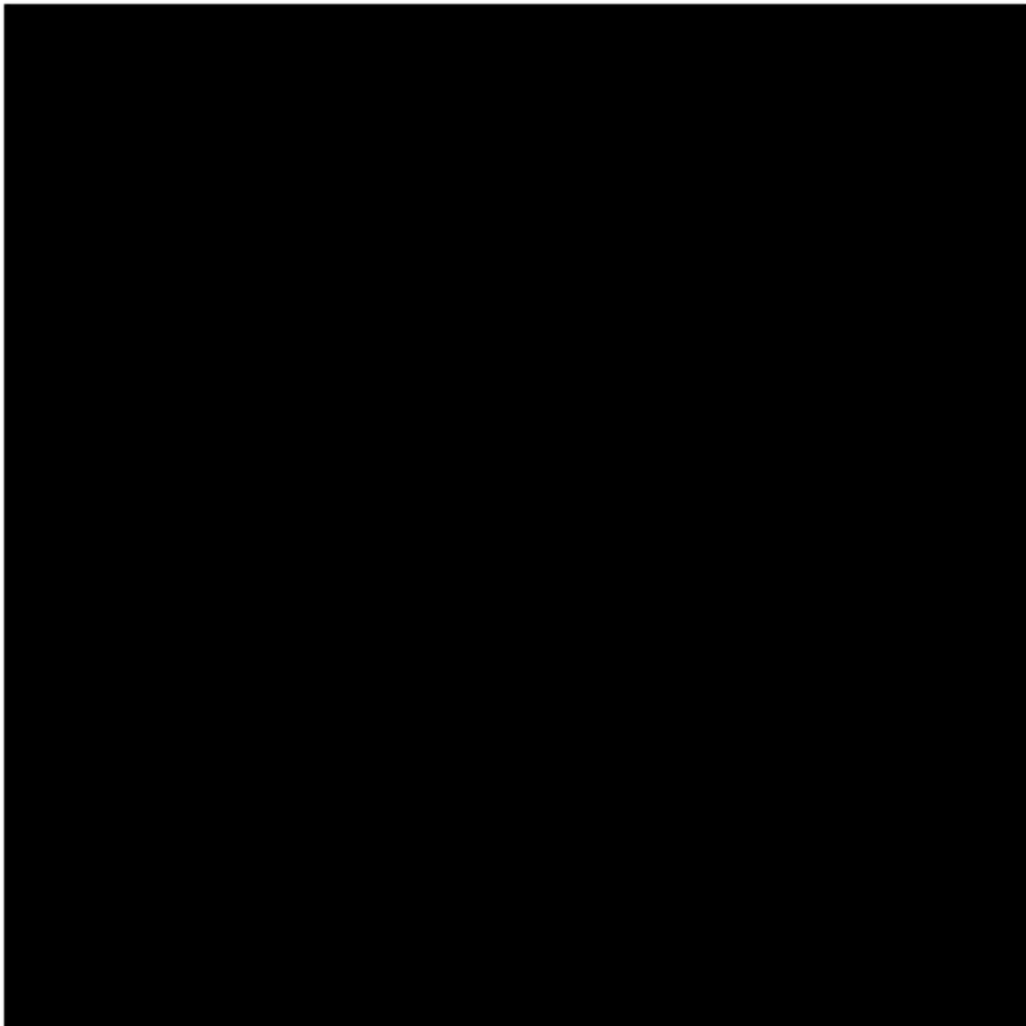
你会发现，上面的代码执行以后输出的画面并没有什么变化。这是为什么呢？因为我们第一次渲染时，也就是当 uTime 为 0 的时候，我们直接画了一个圆。而当我们从上一次绘制的纹理中获取信息，重新渲染时，因为每次获取的纹理图案都是不变的，所以现在的画面依然是静止的圆。



如果我们想让这个图动起来，比如说让它向上升，那么我们只要在每次绘制的时候，改变一下采样的 y 坐标，就是每次从 tMap 取样时取当前纹理坐标稍微下方一点的像素点就可以了。具体的操作代码如下：

[复制代码](#)

```
1 void main() {
2     vec3 smoke = vec3(0);
3     if(uTime <= 0.0) {
4         vec2 st = vUv - vec2(0.5);
5         float d = length(st);
6         smoke = vec3(1.0 - smoothstep(0.05, 0.055, d));
7     }
8     vec2 st = vUv;
9     st.y -= 0.001;
10    vec3 diffuse = texture2D(tMap, st).rgb;
11    gl_FragColor.rgb = diffuse + smoke;
12    gl_FragColor.a = 1.0;
13 }
14
```



不过，由于纹理采样精度的问题，我们得到的上升圆还会有一个扩散的效果。不过这没有关系，它不影响我们接下来要实现的烟雾效果。

接下来，我们需要构建一个烟雾的扩散模型，也就是以某个像素位置以及周边像素的纹理颜色来计算新的颜色值。为了方便你理解，我就以一个 5\*5 的画布为例来详细说说。假设，这个画布只有中心五个位置的颜色是纯白（1.0），周围都是黑色，如下图所示。

		1.0		
	1.0	1.0	1.0	
		1.0		

在这个扩散模型中，每个格子到下一时刻的颜色变化量，等于它周围四个格子的颜色值之和减去它自身颜色值的 4 倍，乘以扩散系数。

假设扩散系数是常量 0.1，那么第一轮每一格的颜色值我在表格上标出来了，如下图所示。

		<div>+0.1</div>		
	<div>+0.2</div>	<div>-0.3 1.0</div>	<div>+0.2</div>	
<div>+0.1</div>	<div>-0.3 1.0</div>	<div>+0 1.0</div>	<div>-0.3 1.0</div>	<div>+0.1</div>
	<div>+0.2</div>	<div>-0.3 1.0</div>	<div>+0.2</div>	
		<div>+0.1</div>		

可以看到，上图中有三种颜色的格子，分别是红色、蓝色和绿色。下面，我们直接来看颜色值的计算过程。

首先是中间红色的那个格子。因为它四周的格子颜色都是 1.0，所以它的颜色变化量是： $0.1 * ((1.0 + 1.0 + 1.0 + 1.0) - 4 * 1.0) = 0$ ，那么下一帧的颜色值还是 1.0 不变。

其次，红格子周围的四个蓝色格子。它们下一帧的颜色变化量为： $0.1 * ((1.0 + 0 + 0 + 0) - 4 * 1.0) = -0.3$ ，那么它们下一帧的颜色值都要减去 0.3 就是 0.7。

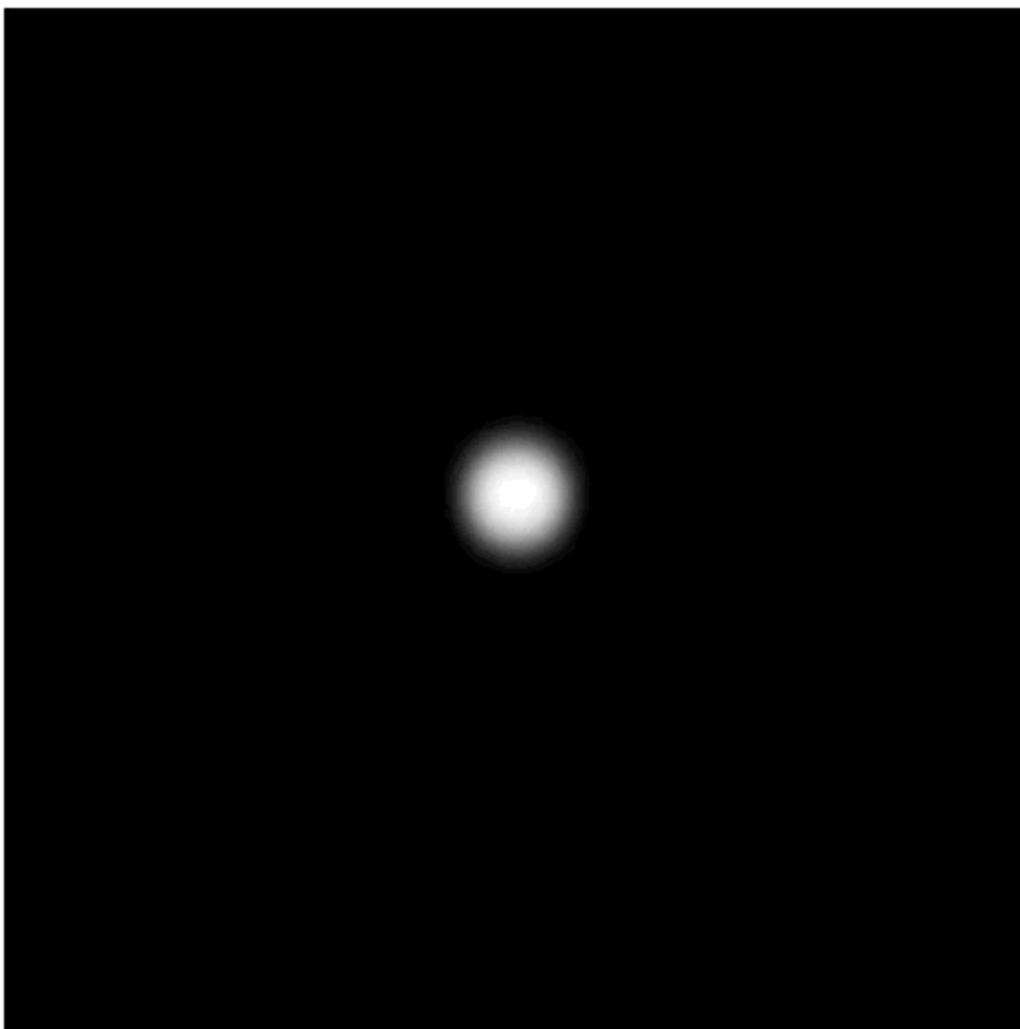
最后，在计算绿色格子下一帧的颜色值时，要分为两种情况。

第一种，当要计算的绿色格子和两个蓝色格子相邻的时候，颜色变化量为： $0.1 * ((1.0 + 1.0 + 0 + 0) - 4 * 0) = 0.2$ ，所以绿格子下一帧的颜色值变为 0.2。

第二种，当这个绿色格子只和一个蓝色格子相邻的时候，颜色变化量为 0.1，那么绿格子下一帧的颜色值就变为 0.1。

就这样，我们把每一帧颜色按照这个规则不断迭代下去，就能得到一个烟雾扩散效果了。那我们下一步就是把它实现到 Shader 中，不过，在 Fragment Shader 中添加扩散模型的时候，为了让这个烟雾效果，能上升得更明显，我稍稍修改了一下扩散公式的权重，让它向上的幅度比较大。

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  varying vec2 vUv;
6  uniform sampler2D tMap;
7  uniform float uTime;
8
9  void main() {
10     vec3 smoke = vec3(0);
11     if(uTime <= 0.0) {
12         vec2 st = vUv - vec2(0.5);
13         float d = length(st);
14         smoke = vec3(step(d, 0.05));
15         // smoke = vec3(1.0 - smoothstep(0.05, 0.055, d));
16     }
17
18     vec2 st = vUv;
19
20     float offset = 1.0 / 256.0;
21     vec3 diffuse = texture2D(tMap, st).rgb;
22
23     vec4 left = texture2D(tMap, st + vec2(-offset, 0.0));
24     vec4 right = texture2D(tMap, st + vec2(offset, 0.0));
25     vec4 up = texture2D(tMap, st + vec2(0.0, -offset));
26     vec4 down = texture2D(tMap, st + vec2(0.0, offset));
27
28     float diff = 8.0 * 0.016 * (
29         left.r +
30         right.r +
31         down.r +
32         2.0 * up.r -
33         5.0 * diffuse.r
34     );
35
36     gl_FragColor.rgb = (diffuse + diff) + smoke;
37     gl_FragColor.a = 1.0;
38 }
39
```



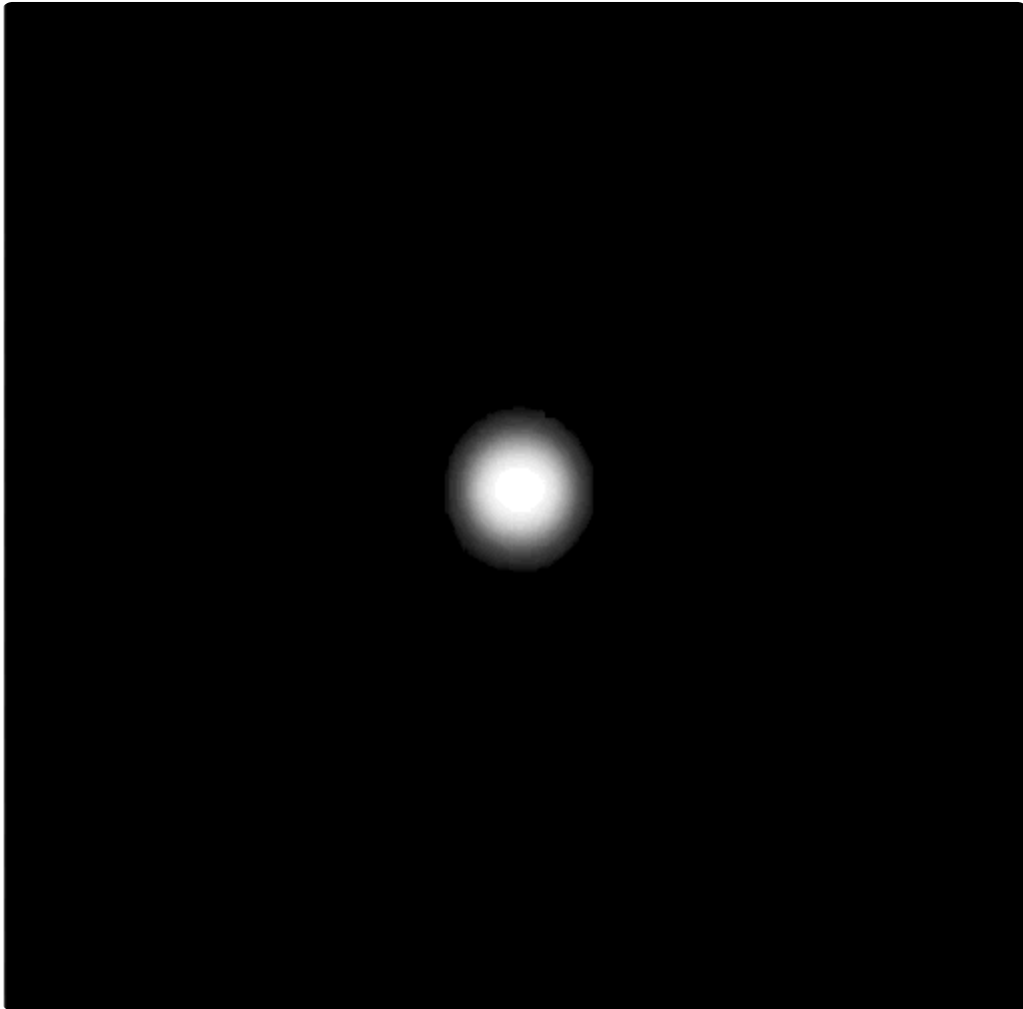
你会发现，这个效果还不是特别真实。那为了达到更真实的烟雾效果，我们还可以在扩散函数上增加一些噪声，代码如下：

[复制代码](#)

```
1 void main() {
2     vec3 smoke = vec3(0);
3     if(uTime <= 0.0) {
4         vec2 st = vUv - vec2(0.5);
5         float d = length(st);
6         smoke = vec3(step(d, 0.05));
7         // smoke = vec3(1.0 - smoothstep(0.05, 0.055, d));
8     }
9
10    vec2 st = vUv;
11
12    float offset = 1.0 / 256.0;
13    vec3 diffuse = texture2D(tMap, st).rgb;
14
15    vec4 left = texture2D(tMap, st + vec2(-offset, 0.0));
16    vec4 right = texture2D(tMap, st + vec2(offset, 0.0));
17    vec4 up = texture2D(tMap, st + vec2(0.0, -offset));
18    vec4 down = texture2D(tMap, st + vec2(0.0, offset));
```

```
19 float rand = noise(st + 5.0 * uTime);
20 float diff = 8.0 * 0.016 * (
21     (1.0 + rand) * left.r +
22     (1.0 - rand) * right.r +
23     down.r +
24     2.0 * up.r -
25     5.0 * diffuse.r
26 );
27
28 gl_FragColor.rgb = (diffuse + diff) + smoke;
29 gl_FragColor.a = 1.0;
30 }
31
```

这样，我们最终实现的效果看起来就会更真实一些：

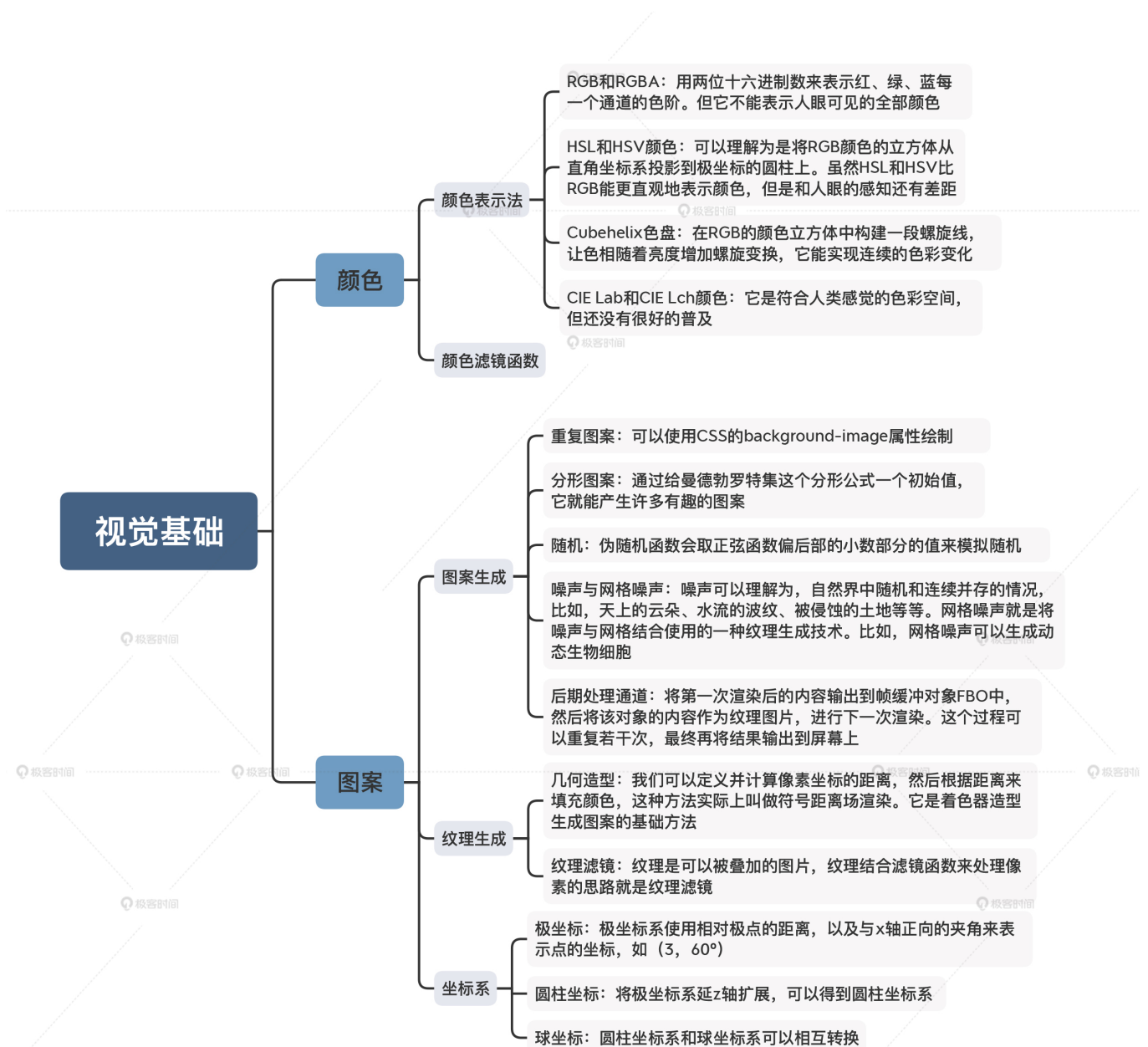


## 要点总结

今天，我们学习了怎么在 WebGL 中，使用后期处理通道来增强图像的视觉效果。我们讲了两个比较常用的效果，一个是 Blur 滤镜以及基于它实现辉光效果，另一个是烟雾效果。

那后期处理通道实现这些效果的核心原理其实都是一样的，都是把第一次渲染后的内容输出到帧缓冲对象 FBO 中，然后把这个对象的内容作为纹理图片，再进行下一次渲染，这个渲染的过程可以重复若干次。最后，我们再把结果输出到屏幕上，就可以了。

到这里，视觉基础篇的内容，我们就全部讲完了。在这个模块里，我们围绕处理图像的细节，系统地学习了怎么表示颜色，怎么生成重复图案，怎么构造和使用滤镜处理图像，怎么进行纹理造型，还有怎么使用不同的坐标系绘图，怎么使用噪声和网格噪声生成复杂纹理、以及今天学习的怎么使用后期处理通道增强图像。我把核心的内容总结成了一张脑图，你可以借助它，来复习巩固这一模块的内容，查缺补漏。





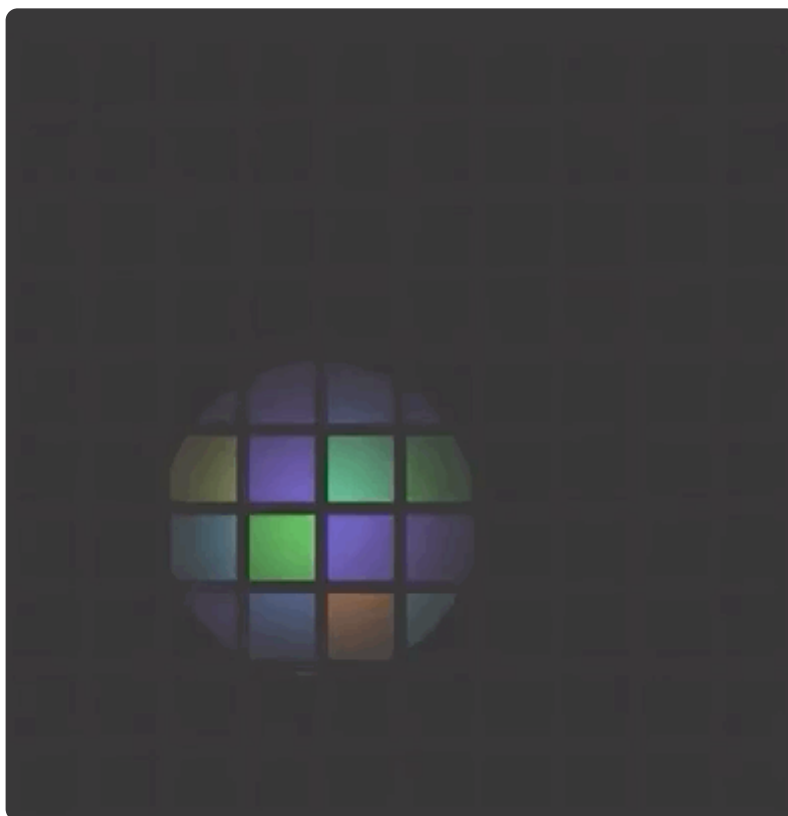
第一题，今天，我们实现的烟雾扩散效果还不够完善，也不够有趣，你能试着改进它，让它变得更有趣吗？你可以参考我给出的两个建议，也可以试试其他的效果。

1. 给定一个向量，表示风向和风速，让烟雾随着这个风扩散，这个风可以随着时间慢慢变化（你可以使用前面学过的噪声来实现风的变化），看看能够做出什么效果。
2. 尝试让烟雾跟随着鼠标移动轨迹扩散，达到类似下面的效果。



如果你觉得难以实现，这里有 [🔗 一篇文章](#) 详细讲了烟雾生成的方法，你可以仔细研究一下。

第二题，实际上，后期处理通道可不止是实现 Blur 滤镜、辉光或者烟雾效果那么简单，它还可以实现很多不同的功能。比如，SpriteJS 官网上那种类似于 [🔦 探照灯](#) 的效果，就是用后期处理通道实现的。你可以用 gl-renderer 来实现这类效果吗？



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，我们下节课再见！

---

## 源码

完整的示例代码见 [🔗 GitHub 仓库](#)

## 推荐阅读

[1] [🔗 原始 FBO 对象的创建和使用方法](#)

[2] [🔗 How to Write a Smoke Shader](#)

提建议

# 跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 16 | 如何使用噪声生成复杂的纹理？

下一篇 18 | 如何生成简单动画让图形动起来？

## 精选留言 (1)

 写留言



胡浩

2020-08-08

好秀啊

展开 ▾

