



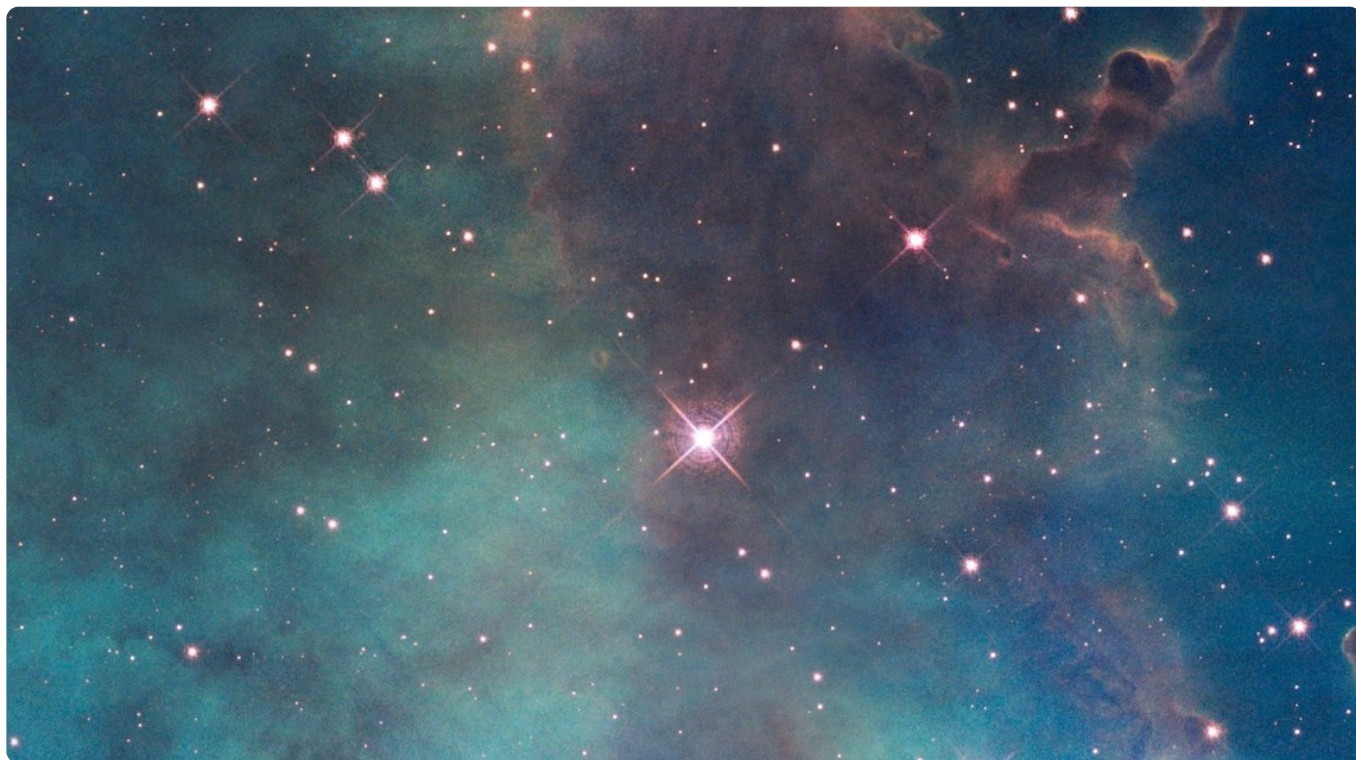
下载APP



19 | 如何用着色器实现像素动画？

2020-08-05 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 11:15 大小 10.32M



你好，我是月影。

上节课，我们以 HTML/CSS 为例，讲了三种动画的实现方法，以及标准的动画模型。我们先来回顾一下：

固定帧动画：为每一帧准备一张图片，然后把 CSS 关键帧动画的 easing-function 设为 step-end 进行循环播放。

增加增量动画：在每帧给元素的相关属性增加一定的量，比如增加一个 rotate 角度



时序动画：通过控制时间和动画函数来描述动画，首先定义初始时间和周期，然后在 update 中计算当前经过时间和进度 p，最后通过 p 来更新动画元素的属性。

标准动画模型：先定义 Animator 类，然后使用线性插值实现匀速运动的动画，以及通过缓动函数实现变速运动的动画。

而 WebGL 实现动画的方式和以上这些方式都有差别。所以这节课，我们就接着来讲怎么用着色器来实现动画。

因为实现固定帧动画最简单，所以我们还是先来说它。

如何用着色器实现固定帧动画

我们完全可以使用在片元着色器中替换纹理坐标的方式，来非常简单地实现固定帧动画。为了方便对比，我还是用上一节课实现会飞的小鸟的例子来讲，那片元着色器中的代码和最终要实现的效果如下所示。

 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 float fWidth;
8  uniform vec2 vFrames[3];
9  uniform int frameIndex;
10
11 void main() {
12     vec2 uv = vUv;
13     for (int i = 0; i < 3; i++) {
14         uv.x = mix(vFrames[i].x, vFrames[i].y, vUv.x) / fWidth;
15         if(float(i) == mod(float(frameIndex), 3.0)) break;
16     }
17     vec4 color = texture2D(tMap, uv);
18     gl_FragColor = color;
19 }
```



会飞的小鸟

利用片元着色器实现固定帧动画的关键部分，是 main 函数中的 for 循环。因为我们的动画只有 3 帧，所以最多只需要循环 3 次。

我们还需要一个重要的参数，vFrames。它是每一帧动画的图片起始 x 和结束 x 坐标，我们用这两个坐标和 vUv.x 计算插值，最后除以图片的总宽度 fWidth，就能得到对应的纹理 x 坐标。替换纹理坐标之后，我们就能实现一个会飞的小鸟了。

实现这个固定帧动画对应的 JavaScript 代码如下：

[复制代码](#)

```
1  const canvas = document.querySelector('canvas');
2  const renderer = new GLRenderer(canvas);
3  const textureURL = 'https://p.ssl.qhimg.com/t01f265b6b6479fffc4.png';
4  (async function () {
5      const texture = await renderer.loadTexture(textureURL);
6      const program = renderer.compileSync(fragment, vertex);
7      renderer.useProgram(program);
8      renderer.uniforms.tMap = texture;
9      renderer.uniforms.fWidth = 272;
10     renderer.uniforms.vFrames = [2, 88, 90, 176, 178, 264];
11     renderer.uniforms.frameIndex = 0;
12     setInterval(() => {
13         renderer.uniforms.frameIndex++;
14     }, 200);
15     const x = 43 / canvas.width;
16     const y = 30 / canvas.height;
17     renderer.setMeshData([
18         positions: [
19             [-x, -y],
20             [-x, y],
21             [x, y],
22             [x, -y],
23         ],
24         attributes: {
25             uv: [
26                 [0, 0],
27                 [0, 1],
28                 [1, 1],
29                 [1, 0],
30             ],
31         },
32         cells: [[0, 1, 2], [2, 0, 3]],
33     ]);
34     renderer.render();
35 }());
36
```

实际上 WebGL 实现固定帧动画的思路，和上一节课的思路是类似的。只不过，上一节课我们直接用 CSS 的 background-image，来切换 background-position 就可以实现动画。而在这里，我们需要将图片纹理 tMap 传进去，然后根据不同的 frameIndex 来计算出对应的纹理坐标，并且这个计算是在片元着色器中进行的。

如何用着色器实现非固定帧动画


好了，知道了怎么实现固定帧动画。接着，我们再来说增量动画和时序动画的实现。由于这两种动画都要将与时间有关的参数传给着色器，处理过程非常相似，所以我们可以将它们统称为非固定帧动画，放在一起来说。

由于这两种动画都要将与时间有关的参数传给着色器，因此它们的处理过程非常相似，我们可以将它们统称为非固定帧动画，放在一起来说。

用 Shader 实现非固定帧动画，本质上和上一节课的实现方法没有太大区别。所以，我们仍然可以使用同样的方法，以及标准动画模型来实现它。只不过，用 Shader 来实现非固定帧动画更加灵活，我们可以操作更多的属性，实现更丰富的效果。下面，我们详细来说。

1. 用顶点着色器实现非固定帧动画

我们知道，WebGL 有两种 Shader，分别是顶点着色器和片元着色器，它们都可以用来实现动画。我们先来看顶点着色器是怎么实现动画的。

 复制代码

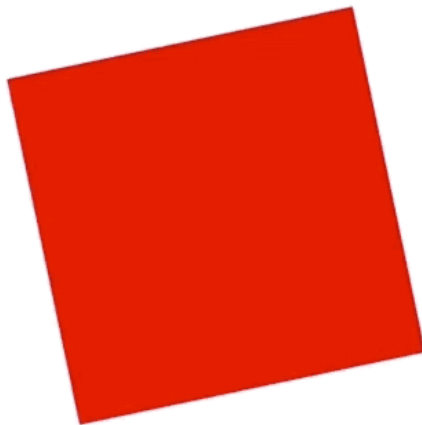
```
1 attribute vec2 a_vertexPosition;
2 attribute vec2 uv;
3
4 varying vec2 vUv;
5 uniform float rotation;
6
7 void main() {
8     gl_PointSize = 1.0;
9     vUv = uv;
10    float c = cos(rotation);
11    float s = sin(rotation);
12    mat3 transformMatrix = mat3(
13        c, s, 0,
14        -s, c, 0,
```

```
15     0, 0, 1
16 );
17 vec3 pos = transformMatrix * vec3(a_vertexPosition, 1);
18 gl_Position = vec4(pos, 1);
19`
```

在顶点着色器中，我们先绘制出一个红色的正方形，然后用三维齐次矩阵实现旋转。具体来说，就是把顶点坐标进行矩阵运算，再配合下面的 JavaScript 代码，就能让这个正方形旋转了。

[复制代码](#)

```
1 renderer.uniforms.rotation = 0.0;
2
3 requestAnimationFrame(function update() {
4     renderer.uniforms.rotation += 0.05;
5     requestAnimationFrame(update);
6 });
```



逆时针旋转的红色正方形

当然，我们也可以使用上一节课得到的标准动画模型来实现。具体来说，就是定义一个新的 Animator 对象，然后在 Animator 对象的方法中更新 rotation 属性。使用标准模型能更加精确地控制图形的旋转效果，代码如下：

[复制代码](#)

```
1 const animator = new Animator({duration: 2000, iterations: Infinity});
```

```
2 animator.animate(renderer, ({target, timing}) => {  
3   target.uniforms.rotation = timing.p * 2 * Math.PI;  
4
```

总之，WebGL 实现非固定帧动画的方法与上节课的方式基本上一样。只不过，前一节课我们直接修改 HTML 元素的属性，而这一节课我们将属性通过 uniform 变量传给着色器执行渲染。

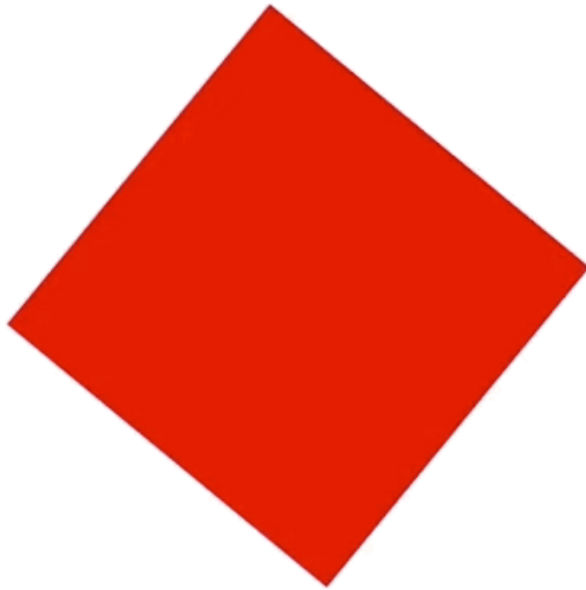
2. 用片元着色器实现非固定帧动画

除了用顶点着色器，我们也能用片元着色器实现动画。实际上，我们已经用片元着色器实现了不少动画。比如说，当我们把时间参数 uTime 通过 uniform 传给着色器的时候，就是在实现动画。

还是用上面的例子。这次，我们将旋转放到片元着色器中处理，其实也能实现类似的旋转效果。代码如下所示：

[复制代码](#)

```
1 #ifdef GL_ES  
2 precision highp float;  
3 #endif  
4  
5 varying vec2 vUv;  
6 uniform vec4 color;  
7 uniform float rotation;  
8  
9 void main() {  
10   vec2 st = 2.0 * (vUv - vec2(0.5));  
11   float c = cos(rotation);  
12   float s = sin(rotation);  
13   mat3 transformMatrix = mat3(  
14     c, s, 0,  
15     -s, c, 0,  
16     0, 0, 1  
17   );  
18   vec3 pos = transformMatrix * vec3(st, 1.0);  
19   float d1 = 1.0 - smoothstep(0.5, 0.505, abs(pos.x));  
20   float d2 = 1.0 - smoothstep(0.5, 0.505, abs(pos.y));  
21   gl_FragColor = d1 * d2 * color;  
22 }
```

顺时针旋转的红色正方形

你发现了吗，顶点着色器和片元着色器实现的旋转动画方向正好相反。为什么会出现这样的情况呢？因为在顶点着色器中，我们直接改变了顶点坐标，所以这样实现的旋转动画和 WebGL 坐标系（右手系）的方向一致，角度增大呈逆时针方向旋转。而在片元着色器中，我们的绘制原理是通过距离场着色来实现的，所以这里的旋转实际上改变的是距离场的角度而不是图形角度，最终绘制的图形也是相对于距离场的。又因为距离场逆时针旋转，所以图形就顺时针旋转了。

最后我再补充一点，一般来说，动画如果能使用顶点着色器实现，我们会尽量在顶点着色器中实现。因为在绘制一帧画面的时候，顶点着色器的运算量会大大少于片元着色器，所以使用顶点着色器消耗的性能更少。

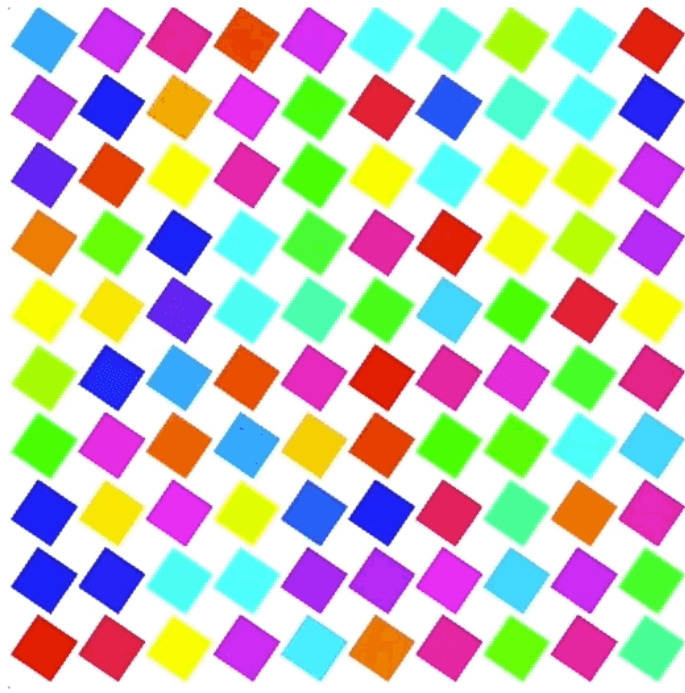
但是，在片元着色器中实现非固定帧动画也有优势。我们可以使用片元着色器的技巧，如重复、随机、噪声等等来绘制更加复杂的效果。

比如说，我们把上面的代码稍微修改一下，使用取小数和取整数的函数，再用之前网格化的思路，来利用网格实现了大量的重复动画。这个做法充分利用了 GPU 的并行效率，比用其他方式把图形一个一个地绘制出来性能要高得多。

```
1 #ifdef GL_ES
```

 复制代码

```
2 precision highp float;
3 #endif
4
5 varying vec2 vUv;
6 uniform float rotation;
7
8 float random (vec2 st) {
9     return fract(sin(dot(st.xy,
10                        vec2(12.9898,78.233))) *
11                43758.5453123);
12 }
13
14 vec3 hsb2rgb(vec3 c){
15     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)
16     rgb = rgb * rgb * (3.0 - 2.0 * rgb);
17     return c.z * mix(vec3(1.0), rgb, c.y);
18 }
19
20 void main() {
21     vec2 f_uv = fract(vUv * 10.0);
22     vec2 i_uv = floor(vUv * 10.0);
23     vec2 st = 2.0 * (f_uv - vec2(0.5));
24     float c = 0.7 * cos(rotation);
25     float s = 0.7 * sin(rotation);
26     mat3 transformMatrix = mat3(
27         c, s, 0,
28         -s, c, 0,
29         0, 0, 1
30     );
31     vec3 pos = transformMatrix * vec3(st, 1.0);
32     float d1 = 1.0 - smoothstep(0.5, 0.505, abs(pos.x));
33     float d2 = 1.0 - smoothstep(0.5, 0.505, abs(pos.y));
34     gl_FragColor = d1 * d2 * vec4(hsb2rgb(vec3(random(i_uv), 1.0, 1.0)), 1.0);
35 }
```

大量重复的旋转正方形

如何在着色器中实现缓动函数与非线性插值

在前面的例子中，我们使用 Shader 的矩阵运算实现了旋转动画。同样，轨迹动画也可以用 Shader 矩阵运算实现。

比如说，我们要在画布上绘制一个红色的方块，利用它实现轨迹动画。首先，我们要实现一个着色器，它通过设置 translation 来改变图形位置，代码如下：

[复制代码](#)

```
1 attribute vec2 a_vertexPosition;
2 attribute vec2 uv;
3
4 varying vec2 vUv;
5 uniform vec2 translation;
6
7 void main() {
8     gl_PointSize = 1.0;
9     vUv = uv;
10    mat3 transformMatrix = mat3(
11        1, 0, 0,
12        0, 1, 0,
13        translation, 1
14    );
15    vec3 pos = transformMatrix * vec3(a_vertexPosition, 1);
```

```
16   gl_Position = vec4(pos, 1);  
17 }  
18
```

然后，在 JavaScript 中，我们将 translation 依照时间变化传给上面的着色器，就可以让方块移动。那利用下面的代码，我们就让方块沿水平方向向右匀速运动一段距离。

[复制代码](#)

```
1  const canvas = document.querySelector('canvas');  
2  const renderer = new GLRenderer(canvas);  
3  const program = renderer.compileSync(fragment, vertex);  
4  renderer.useProgram(program);  
5  renderer.uniforms.color = [1, 0, 0, 1];  
6  renderer.uniforms.translation = [-0.5, 0];  
7  
8  const animator = new Animator({duration: 2000});  
9  animator.animate(renderer, ({target, timing}) => {  
10     target.uniforms.translation = [-0.5 * (1 - timing.p) + 0.5 * timing.p, 0];  
11 });  
12  
13 renderer.setMeshData([ {  
14     positions: [  
15         [-0.25, -0.25],  
16         [-0.25, 0.25],  
17         [0.25, 0.25],  
18         [0.25, -0.25],  
19     ],  
20     attributes: {  
21         uv: [  
22             [0, 0],  
23             [0, 1],  
24             [1, 1],  
25             [1, 0],  
26         ],  
27     },  
28     cells: [[0, 1, 2], [2, 0, 3]],  
29 }]);  
30 renderer.render();  
31
```



水平向右匀速运动的红色正方形

此外，我们还可以通过缓动函数来实现非匀速运动。而且我们既可以将缓动函数用 JavaScript 计算，也可以直接将缓动函数放在 Shader 中。如果将缓动函数用 JavaScript 计算，那么方法和上一节课完全一样，也就是给 Animator 传一个 easing 函数进去就可以了，这里我就不再重复了。但如果要将缓动函数写在 Shader 中，其实方法也非常简单。

我们以前面顶点着色器实现非固定帧动画的代码为例，这次，我们不使用 Animator，而是直接将时间 `uTime` 参数传入 Shader，然后在 Shader 中加入缓动函数。在这里，我们用 `smooth(0.0, 1.0, p)` 来让方块做平滑加速、减速运动。除此之外，你也可以替换缓动函数，比如 `clamp(p * p, 0.0, 1.0)` 或者 `clamp(p * (2 - p) * 0.0, 1.0)` 来实现匀加速、匀减速的运动效果。修改后的代码如下：

[复制代码](#)

```
1 attribute vec2 a_vertexPosition;
2 attribute vec2 uv;
3
4 varying vec2 vUv;
5 uniform vec4 uFromTo;
6 uniform float uTime;
7
8 float easing(in float p) {
9     return smoothstep(0.0, 1.0, p);
10    // return clamp(p * p, 0.0, 1.0);
11    // return clamp(p * (2 - p) * 0.0, 1.0);
12 }
13
14 void main() {
15     gl_PointSize = 1.0;
16     vUv = uv;
17     vec2 from = uFromTo.xy;
```

```
18     vec2 to = uFromTo.zw;
19     float p = easing(uTime / 2.0);
20     vec2 translation = mix(from, to, p);
21     mat3 transformMatrix = mat3(
22         1, 0, 0,
23         0, 1, 0,
24         translation, 1
25     );
26     vec3 pos = transformMatrix * vec3(a_vertexPosition, 1);
27     gl_Position = vec4(pos, 1);
28 }
```

总之，因为 Shader 是在 GPU 中运算的，所以所有顶点都是被并行处理的。因此，通常情况下，我们在顶点着色器中执行缓动函数会更快。

不过，直接用 JavaScript 计算和放在顶点着色器里计算，差别也不是很大，但如果把它放在片元着色器里计算，因为要把每个像素点都计算一遍，所以性能消耗反而更大一些。那我们为什么还要在着色器中计算 easing 呢？这是因为，我们不仅可以利用 easing 控制动画过程，还可以在片元着色器中用 easing 来实现非线性的插值。

那什么是非线性插值呢？我们依然通过例子来进一步理解。

我们知道，在正常情况下，顶点着色器定义的变量在片元着色器中，都会被线性插值。比如，你可以看我下面给出的顶点着色器、片元着色器，以及 JavaScript 中的代码。

[复制代码](#)


```
1 //顶点着色器
2 attribute vec2 a_vertexPosition;
3 attribute vec2 uv;
4 attribute vec4 color;
5
6 varying vec2 vUv;
7 varying vec4 vColor;
8 uniform vec4 uFromTo;
9 uniform float uTime;
10
11 void main() {
12     gl_PointSize = 1.0;
13     vUv = uv;
14     vColor = color;
15     gl_Position = vec4(a_vertexPosition, 1, 1);
16 }
17
18 //片元着色器
```

```
19
20 #ifdef GL_ES
21 precision highp float;
22 #endif
23
24 varying vec2 vUv;
25 varying vec4 vColor;
26
27 void main() {
28     gl_FragColor = vColor;
29 }
30
31 //JavaScript中的代码
32 renderer.setMeshData([
33     positions: [
34         [-0.5, -0.25],
35         [-0.5, 0.25],
36         [0.5, 0.25],
37         [0.5, -0.25],
38     ],
39     attributes: {
40         uv: [
41             [0, 0],
42             [0, 1],
43             [1, 1],
44             [1, 0],
45         ],
46         color: [
47             [1, 0, 0, 1],
48             [1, 0, 0, 1],
49             [0, 0.5, 0, 1],
50             [0, 0.5, 0, 1],
51         ],
52     },
53     cells: [[0, 1, 2], [2, 0, 3]],
54 ]]);
55 renderer.render();
```



从左往右，由红色线性过渡到绿色

通过执行上面的代码，我们可以得到一个长方形，它的颜色会从左到右，由红色线性地过渡到绿色。如果想要实现非线性的颜色过渡，我们就不能采用这种方式了，我们可以采用 uniform 的方式，通过 easing 函数来实现。

 复制代码

```
1 #ifdef GL_ES
2 precision highp float;
3 #endif
4
5 float easing(in float p) {
6     return smoothstep(0.0, 1.0, p);
7     // return clamp(p * p, 0.0, 1.0);
8     // return clamp(p * (2 - p) * 0.0, 1.0);
9 }
10
11 varying vec2 vUv;
12 uniform vec4 fromColor;
13 uniform vec4 toColor;
14
15 void main() {
16     float d = easing(vUv.x);
17     gl_FragColor = mix(fromColor, toColor, d);
18 }
```

比如，我们可以使用 smoothstep 这种 easing 函数，来实现如下的插值效果：



另外，我们还可以像利用 JavaScript 那样，在 Shader 里实现贝塞尔曲线缓动。

复制代码

```
1 // http://www.flong.com/texts/code/shapers_bez/  
2 // Helper functions:  
3 float slope_from_t (float t, float A, float B, float C){  
4     float dtdx = 1.0/(3.0*A*t*t + 2.0*B*t + C);  
5     return dtdx;  
6 }  
7  
8 float x_from_t (float t, float A, float B, float C, float D){  
9     float x = A*(t*t*t) + B*(t*t) + C*t + D;  
10    return x;  
11 }  
12  
13 float y_from_t (float t, float E, float F, float G, float H){  
14     float y = E*(t*t*t) + F*(t*t) + G*t + H;  
15     return y;  
16 }  
17  
18 float cubic_bezier (float x, float a, float b, float c, float d){  
19     float y0a = 0.00; // initial y  
20     float x0a = 0.00; // initial x  
21     float y1a = b;     // 1st influence y  
22     float x1a = a;     // 1st influence x  
23     float y2a = d;     // 2nd influence y  
24     float x2a = c;     // 2nd influence x  
25     float y3a = 1.00; // final y  
26     float x3a = 1.00; // final x  
27  
28     float A = x3a - 3.0 * x2a + 3.0 * x1a - x0a;
```



```
29 float B = 3.0 * x2a - 6.0 * x1a + 3.0 * x0a;
30 float C = 3.0 * x1a - 3.0 * x0a;
31 float D = x0a;
32
33 float E = y3a - 3.0 * y2a + 3.0 * y1a - y0a;
34 float F = 3.0 * y2a - 6.0 * y1a + 3.0 * y0a;
35 float G = 3.0 * y1a - 3.0 * y0a;
36 float H = y0a;
37
38 // Solve for t given x (using Newton-Raphelson), then solve for y given t.
39 // Assume for the first guess that t = x.
40 float currentt = x;
41 const int nRefinementIterations = 5;
42 for (int i=0; i < nRefinementIterations; i++){
43     float currentx = x_from_t(currentt, A,B,C,D);
44     float currentslope = slope_from_t(currentt, A,B,C);
45     currentt -= (currentx - x)*(currentslope);
46     currentt = clamp(currentt, 0.0, 1.0);
47 }
48
49 float y = y_from_t(currentt, E,F,G,H);
50 return y;
51
```


使用贝塞尔曲线缓动函数，我们能够实现更加丰富多彩的插值效果。



贝塞尔曲线插值色带

如何在片元着色器中实现随机粒子动画

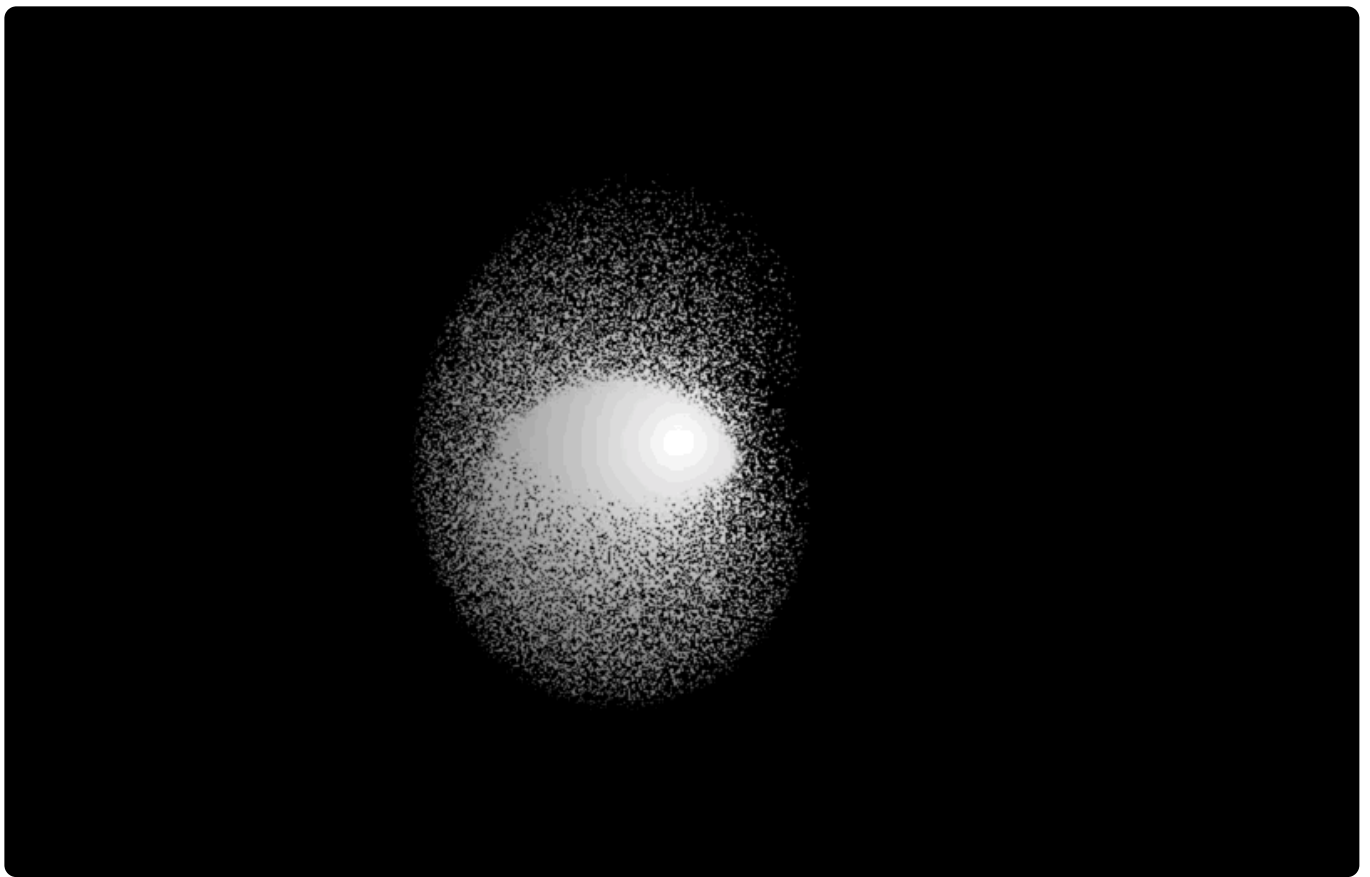
我们知道，使用片元着色器还可以实现非常复杂的图形动画，包括粒子动画、网格动画以及网格噪声动画等等。网格动画和网格噪声我们前面都详细讲过，这里我们就重点来说说怎么实现粒子动画效果。

 复制代码

```
1  #ifdef GL_ES
2  precision highp float;
3  #endif
4
5  ...
6
7  float sdf_circle(vec2 st, vec2 c, float r) {
8      return 1.0 - length(st - c) / r;
9  }
10
11  varying vec2 vUv;
12  uniform float uTime;
13
14  void main() {
15      vec2 st = vUv;
16      float rx = mix(-0.2, 0.2, noise(vec2(7881.32, 0) + random(st) + uTime));
17      float ry = mix(-0.2, 0.2, noise(vec2(0, 1433.59) + random(st) + uTime));
18      float dis = distance(st, vec2(0.5));
19      dis = pow((1.0 - dis), 2.0);
20      float d = sdf_circle(st + vec2(rx, ry), vec2(0.5), 0.2);
21      d = smoothstep(0.0, 0.1, d);
22      gl_FragColor = vec4(dis * d * vec3(1.0), 1.0);
23  }
```

如上面代码所示，我们可以使用随机 + 噪声来实现一个粒子效果。首先，我们设置随机数用来生成距离场的初始值，然后设置噪声用来形成位移，最后传入 `uTime` 变量来实现动画。

这样一来，我们就能绘制出数量非常多的点，并且让它们沿着随机轨迹运动。最终的视觉效果如下：



粒子动画效果

像这样流畅的动画效果，因为实现的过程中会涉及非常多点的运算，如果不用 shader，我们几乎是无法完成的。

要点总结

这节课我们学习了用 WebGL 实现动画的方法。

如果是实现固定帧动画，在 WebGL 中，我们可以把准备好的图片作为纹理，然后动态修改纹理坐标。

如果是实现非固定帧动画，我们可以通过 uniform，将变化的属性作为参数传给着色器处理。上节课的标准动画模型在 WebGL 中依然可以使用，我们可以利用它计算出属性，再传入着色器执行渲染。

实际上，今天讲的方法，与用 HTML/CSS、SVG、Canvas2D 实现动画的基本原理是一样的。只不过，WebGL 中的很多计算，是需要用 JavaScript 和 GLSL，也就是 Shader 来配合进行的。

这节课的实战例子比较多，我建议你好好研究一下。毕竟，使用片元着色器实现动画效果的思路，我们还会在后续课程中经常用到。

小试牛刀

1. 今天，我们在 Shader 中通过矩阵运算实现了图形的旋转和平移，你能用学到的知识完善矩阵运算，来实现缩放、旋转、平移和扭曲变换，以及它们的组合效果吗？
2. 结合今天的内容，你可以试着实现一个粒子效果：让一张图片从中心爆炸开来，炸成碎片并最终消失。

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

本节课完整示例代码见 [🔗 GitHub 仓库](#)

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 18 | 如何生成简单动画让图形动起来？

下一篇 20 | 如何用WebGL绘制3D物体？

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。