



下载APP



09 | 如何用仿射变换对几何图形进行坐标变换？

2020-07-10 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 19:58 大小 18.30M



你好，我是月影。

前面两节课，我们学习了用向量表示的顶点，来描述曲线和多边形的方法。但是在实际绘制的时候，我们经常需要在画布上绘制许多轮廓相同的图形，难道这也需要我们重复地去计算每个图形的顶点吗？当然不需要。我们只需要创建一个基本的几何轮廓，然后通过**仿射变换**来改变几何图形的位置、形状、大小和角度。

仿射变换是拓扑学和图形学中一个非常重要的基础概念。利用它，我们才能在可视化应用中快速绘制出形态、位置、大小各异的众多几何图形。所以，这一节课，我们就来说-
仿射变换的数学基础和基本操作，它几乎会被应用到我们后面讲到的所有视觉呈现的案例中，所以你一定要掌握。



什么是仿射变换？

仿射变换简单来说就是“线性变换 + 平移”。实际上在平常的 Web 开发中，我们也经常会用到仿射变换，比如，对元素设置 CSS 的 transform 属性就是对元素应用仿射变换。

再说回到几何图形，针对它的仿射变换具有以下 2 个性质：

1. 仿射变换前是直线段的，仿射变换后依然是直线段
2. 对两条直线段 a 和 b 应用同样的仿射变换，变换前后线段长度比例保持不变

由于仿射变换具有这两个性质，因此对线性空间中的几何图形进行仿射变换，就相当于对它的每个顶点向量进行仿射变换。

那具体怎么操作呢？下面，我们就来详细说说。

向量的平移、旋转与缩放

常见的仿射变换形式包括**平移**、**旋转**、**缩放**以及它们的组合。其中，平移变换是最简单的仿射变换。如果我们想让向量 $P(x_0, y_0)$ 沿着向量 $Q(x_1, y_1)$ 平移，只要将 P 和 Q 相加就可以了。

$$\begin{cases} x = x_0 + x_1 \\ y = y_0 + y_1 \end{cases}$$

平移后的向量p的坐标

接着是旋转变换。实际上，旋转变换我们在第 5 课接触过，当时我们把向量的旋转定义成了如下的函数：

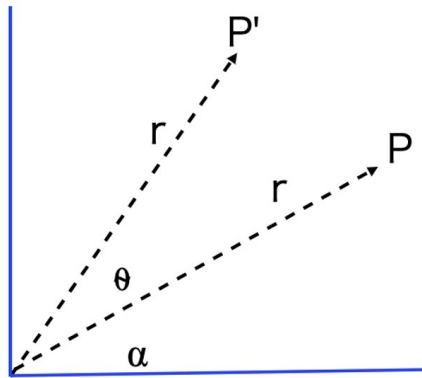
 复制代码

```
1 class Vector2D {
2     ...
3     rotate(rad) {
4         const c = Math.cos(rad),
5             s = Math.sin(rad);
6         const [x, y] = this;
7     }
8 }
```

```

8     this.x = x * c + y * -s;
9     this.y = x * s + y * c;
10
11     return this;
12 }
13 }
```

但是，我们并没有讨论这个函数是怎么来的，那在这里我们通过三角函数来详细推导一下。



假设向量 P 的长度为 r ，角度是 α ，现在我们要将它顺时针旋转 θ 角，此时新的向量 P' 的参数方程为：

$$\begin{cases} x = r \cos(\alpha + \theta) = r \cos \alpha \cos \theta - r \sin \alpha \sin \theta \\ y = r \sin(\alpha + \theta) = r \cos \alpha \sin \theta + r \sin \alpha \cos \theta \end{cases}$$

然后，因为 $r \cos \alpha$ 、 $r \sin \alpha$ 是向量 P 原始的坐标 x_0 、 y_0 ，所以，我们可以把坐标代入到上面的公式中，就会得到如下的公式：

$$\begin{cases} x = x_0 \cos \theta - y_0 \sin \theta \\ y = x_0 \sin \theta + y_0 \cos \theta \end{cases}$$

最后，我们再将它写成矩阵形式，就会得到一个旋转矩阵。至于为什么要写成矩阵形式，我后面会讲，这里你先记住这个旋转矩阵的公式就可以了。

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

然后是缩放变换。缩放变换也很简单，我们可以直接让向量与标量（标量只有大小、没有方向）相乘。

$$\begin{cases} x = s_x x_0 \\ y = s_y y_0 \end{cases}$$

对于得到的这个公式，我们也可以把它写成矩阵形式。结果如下：

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

现在，我们就得到了三个基本的仿射变换公式，其中旋转和缩放都可以写成矩阵与向量相乘的形式。这种能写成矩阵与向量相乘形式的变换，就叫做**线性变换**。线性变换除了可以满足仿射变换的 2 个性质之外，还有 2 个额外的性质：

1. 线性变换不改变坐标原点（因为如果 x_0 、 y_0 等于零，那么 x 、 y 肯定等于 0）；
2. 线性变换可以叠加，多个线性变换的叠加结果就是将线性变换的矩阵依次相乘，再与原始向量相乘。

那根据线性变换的第 2 条性质，我们就能总结出一个通用的线性变换公式，即一个原始向量 P_0 经过 M_1 、 M_2 、... M_n 次的线性变换之后得到最终的坐标 P 。线性变化的叠加是一个非常重要的性质，它是我们对图形进行变换的基础，所以你一定要牢记线性变化的叠加性质。

$$\begin{aligned}
 P &= M_1 \times M_2 \times \cdots M_n \times P_0 \\
 &= M \times P_0 (M = M_1 \times M_2 \times \cdots M_n)
 \end{aligned}$$

好了，常见的仿射变换形式我们说完了。总的来说，向量的基本仿射变换分为平移、旋转与缩放，其中旋转与缩放属于线性变换，而平移不属于线性变换。基于此，我们可以得到仿射变换的一般表达式，如下图所示：

$$\begin{array}{ccc}
 & \text{线性变换} & \text{平移} \\
 & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} \\
 P &= M \times P_0 &+ P_1
 \end{array}$$

仿射变换的公式优化

上面这个公式我们还可以改写成矩阵的形式，在改写的公式里，我们实际上是给线性空间增加了一个维度。换句话说，我们用高维度的线性变换表示了低维度的仿射变换！

$$\begin{pmatrix} P \\ 1 \end{pmatrix} = \begin{pmatrix} M & P_1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} P_0 \\ 1 \end{pmatrix}$$

这样，我们就将原本 n 维的坐标转换为了 $n+1$ 维的坐标。这种 $n+1$ 维坐标被称为**齐次坐标**，对应的矩阵就被称为**齐次矩阵**。

齐次坐标和齐次矩阵是可视化中非常常用的数学工具，它能让我们用线性变换来表示仿射变换。这样一来，我们就能利用线性变换的叠加性质，来非常方便地进行各种复杂的仿射变换了。落实到共识上，就是把这些变换的矩阵相乘得到一个新的矩阵，再把它乘以原向量。我们在绘制几何图形的时候会经常用到它，所以你要记住这个公式。

仿射变换的应用：实现粒子动画

好了，现在你已经知道了仿射变换的数学基础。那它该怎么应用呢？一个很常见的应用，就是利用它来实现粒子动画。


你可能还不熟悉粒子动画，我们先来快速认识一下它。它能在一定时间内生成许多随机运动的小图形，这类动画通常是通过给人以视觉上的震撼，来达到获取用户关注的效果。在可视化中，粒子动画可以用来表达数据信息本身（比如数量、大小等等），也可以用来修饰界面、吸引用户的关注，它是在可视化中经常会用到的一种视觉效果。

在粒子动画的实现过程中，我们通常需要在界面上快速改变一大批图形的大小、形状和位置，所以用图形的仿射变换来实现是一个很好的方法。

为了方便你理解，我们今天只讲一个简单的粒子动画。这个粒子动画的运行效果，是从一个点开始发射出许多颜色、大小、角度各异的三角形，并且通过不断变化它们的位置，产生一种撒花般的视觉效果。

1. 创建三角形

因为这个粒子动画中主要用到了三角形，所以我们第一步就要创建三角形。**创建三角形一共可以分为两步，第一步，我们定义三角形的顶点并将数据送到缓冲区。**这一步，你直接看下面创建 WebGLProgram 的步骤就能理解。如果你还不是很熟悉，我建议你复习一下第 4 节课的内容。

 复制代码

```
1  const position = new Float32Array([
2    -1, -1,
3    0, 1,
4    1, -1,
5  ]);
6  const bufferId = gl.createBuffer();
7  gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
8  gl.bufferData(gl.ARRAY_BUFFER, position, gl.STATIC_DRAW);
9
10 const vPosition = gl.getAttribLocation(program, 'position');
11 gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
12 gl.enableVertexAttribArray(vPosition);
```

第二步，我们实现一个创建随机三角形属性的函数。具体来说就是，利用 randomTriangles 随机创建一个三角形的信息，其中的参数包括颜色 u_color、初始旋转

角度 `u_rotation`、初始大小 `u_scale`、初始时间 `u_time`、动画持续时间 `u_diraction`、运动方向 `u_dir` 和创建时间 `startTime`。除了 `startTime` 之外的数据，我们都需要传给 `shader` 去处理。

[复制代码](#)

```
1 function randomTriangles() {
2   const u_color = [Math.random(), Math.random(), Math.random(), 1.0]; // 随机颜色
3   const u_rotation = Math.random() * Math.PI; // 初始旋转角度
4   const u_scale = Math.random() * 0.05 + 0.03; // 初始大小
5   const u_time = 0;
6   const u_duration = 3.0; // 持续3秒钟
7
8   const rad = Math.random() * Math.PI * 2;
9   const u_dir = [Math.cos(rad), Math.sin(rad)]; // 运动方向
10  const startTime = performance.now();
11
12  return {u_color, u_rotation, u_scale, u_time, u_duration, u_dir, startTime};
13 }
```

2. 设置 uniform 变量

通过前面的代码，我们已经将三角形顶点信息传入缓冲区。我们知道，在 WebGL 的 `shader` 中，顶点相关的变量可以用 `attribute` 声明。但是，我们现在要把 `u_color`、`u_rotation` 等一系列变量也传到 `shader` 中，这些变量与三角形具体顶点无关，它们是一些固定的值。这时候，我们就要用到 `shader` 的另一种变量声明，也就是 `uniform` 来声明。

那它们有什么区别呢？首先，`attribute` 变量是对应于顶点的。也就是说，几何图形有几个顶点就要提供几份 `attribute` 数据。并且，`attribute` 变量只能在顶点着色器中使用，如果要在片元着色器中使用，需要我们通过 `varying` 变量将它传给片元着色器才行。这样一来，片元着色器中获取的实际值，就是经过顶点线性插值的。

而 `uniform` 声明的变量不同，`uniform` 声明的变量和其他语言中的常量一样，我们赋给 `uniform` 变量的值在 `shader` 执行的过程中不可改变。而且一个变量的值是唯一的，不随顶点变化。**`uniform` 变量既可以在顶点着色器中使用，也可以在片元着色器中使用。**

在 WebGL 中，我们可以通过 `gl.uniformXXX(loc, u_color);` 的方法将数据传给 `shader` 的 `uniform` 变量。其中，`XXX` 是我们随着数据类型不同取得不同的名字。我在下面列举了一

些比较常用的，你可以看看：

`gl.uniform1f` 传入一个浮点数，对应的 uniform 变量的类型为 `float`


`gl.uniform4f` 传入四个浮点数，对应的 uniform 变量类型为 `float[4]`

`gl.uniform3fv` 传入一个三维向量，对应的 uniform 变量类型为 `vec3`

`gl.uniformMatrix4fv` 传入一个 4x4 的矩阵，对应的 uniform 变量类型为 `mat4`

今天，关于 WebGL 的 uniform 的设置，我们只需要知道这个最常用的方法就可以了，更详细的设置信息，你可以参考 [MDN 官方文档](#)。

接下来，我们实现这个函数，将随机三角形信息传给 shader 里的 uniform 变量。代码如下：

 复制代码

```
1 function setUniforms(gl, {u_color, u_rotation, u_scale, u_time, u_duration, u_
2   // gl.getUniformLocation 拿到uniform变量的指针
3   let loc = gl.getUniformLocation(program, 'u_color');
4   // 将数据传给 uniform 变量的地址
5   gl.uniform4fv(loc, u_color);
6
7   loc = gl.getUniformLocation(program, 'u_rotation');
8   gl.uniform1f(loc, u_rotation);
9
10  loc = gl.getUniformLocation(program, 'u_scale');
11  gl.uniform1f(loc, u_scale);
12
13  loc = gl.getUniformLocation(program, 'u_time');
14  gl.uniform1f(loc, u_time);
15
16  loc = gl.getUniformLocation(program, 'u_duration');
17  gl.uniform1f(loc, u_duration);
18
19  loc = gl.getUniformLocation(program, 'u_dir');
20  gl.uniform2fv(loc, u_dir);
21 }
```

3. 用 `requestAnimationFrame` 实现动画

然后，我们使用 `requestAnimationFrame` 实现动画。具体的方法就是，我们在 `update` 方法中每次新建数个随机三角形，然后依次修改所有三角形的 `u_time` 属性，通过

setUniforms 方法将修改的属性更新到 shader 变量中。这样，我们就可以在 shader 中读取变量的值进行了。代码如下：

[复制代码](#)

```
1 let triangles = [];  
2  
3 function update() {  
4   for(let i = 0; i < 5 * Math.random(); i++) {  
5     triangles.push(randomTriangles());  
6   }  
7   gl.clear(gl.COLOR_BUFFER_BIT);  
8   // 对每个三角形重新设置u_time  
9   triangles.forEach((triangle) => {  
10    triangle.u_time = (performance.now() - triangle.startTime) / 1000;  
11    setUniforms(gl, triangle);  
12    gl.drawArrays(gl.TRIANGLES, 0, position.length / 2);  
13  });  
14  // 移除已经结束动画的三角形  
15  triangles = triangles.filter((triangle) => {  
16    return triangle.u_time <= triangle.u_duration;  
17  });  
18  requestAnimationFrame(update);  
19 }  
20  
21 requestAnimationFrame(update);
```

我们再回过头来看最终要实现的效果。你会发现，所有的三角形，都是由小变大朝着特定的方向旋转。那想要实现这个效果，我们就需要用到前面讲过的仿射变换，在顶点着色器中进行矩阵运算。

在这一步中，顶点着色器中的 glsl 代码最关键，我们先来看一下这个代码是怎么写的。

[复制代码](#)

```
1 attribute vec2 position;  
2  
3 uniform float u_rotation;  
4 uniform float u_time;  
5 uniform float u_duration;  
6 uniform float u_scale;  
7 uniform vec2 u_dir;  
8  
9 varying float vP;  
10  
11 void main() {
```

```
12 float p = min(1.0, u_time / u_duration);
13 float rad = u_rotation + 3.14 * 10.0 * p;
14 float scale = u_scale * p * (2.0 - p);
15 vec2 offset = 2.0 * u_dir * p * p;
16 mat3 translateMatrix = mat3(
17     1.0, 0.0, 0.0,
18     0.0, 1.0, 0.0,
19     offset.x, offset.y, 1.0
20 );
21 mat3 rotateMatrix = mat3(
22     cos(rad), sin(rad), 0.0,
23     -sin(rad), cos(rad), 0.0,
24     0.0, 0.0, 1.0
25 );
26 mat3 scaleMatrix = mat3(
27     scale, 0.0, 0.0,
28     0.0, scale, 0.0,
29     0.0, 0.0, 1.0
30 );
31 gl_PointSize = 1.0;
32 vec3 pos = translateMatrix * rotateMatrix * scaleMatrix * vec3(position, 1.0);
33 gl_Position = vec4(pos, 1.0);
34 vP = p;
35
```

其中有几个关键参数，你可能还比较陌生，我来分别介绍一下。

首先，我们定义的 p 是当前动画进度，它的值是 $u_time / u_duration$ ，取值区间从 0 到 1。 rad 是旋转角度，它的值是初始角度 $u_rotation$ 加上 10π ，表示在动画过程中它会绕自身旋转 5 周。

其次， $scale$ 是缩放比例，它的值是初始缩放比例乘以一个系数，这个系数是 $p * (2.0 - p)$ ，在我们后面讨论动画的时候你会知道， $p * (2.0 - p)$ 是一个缓动函数，在这里我们只需要知道，它的作用是让 $scale$ 的变化量随着时间推移逐渐减小就可以了。

最后， $offset$ 是一个二维向量，它是初始值 u_dir 与 $2.0 * p * p$ 的乘积，因为 u_dir 是个单位向量，这里的 2.0 表示它的最大移动距离为 2， $p * p$ 也是一个缓动函数，作用是让位移的变化量随着时间增加而增大。

定义完这些参数以后，我们得到三个齐次矩阵： $translateMatrix$ 是偏移矩阵， $rotateMatrix$ 是旋转矩阵， $scaleMatrix$ 是缩放矩阵。我们将 pos 的值设置为这三个矩阵

与 position 的乘积，这样就完成对顶点的线性变换，呈现出来的效果也就是三角形会向着特定的方向旋转、移动和缩放。

4. 在片元着色器中着色

最后，我们在片元着色器中对这些三角形着色。我们将 p 也就是动画进度，从顶点着色器通过变量 varying vP 传给片元着色器，然后在片元着色器中让 alpha 值随着 vP 值变化，这样就能同时实现粒子的淡出效果了。

片元着色器中的代码如下：

[复制代码](#)

```
1 precision mediump float;
2 uniform vec4 u_color;
3 varying float vP;
4
5 void main()
6 {
7     gl_FragColor.xyz = u_color.xyz;
8     gl_FragColor.a = (1.0 - vP) * u_color.a;
9 }
```

到这里，我们就用仿射变换实现了一个有趣的粒子动画。

CSS 的仿射变换

既然我们讲了仿射变换，这里还是要再提一下 CSS 中我们常用的属性 transform。

[复制代码](#)

```
1 div.block {
2     transform: rotate(30deg) translate(100px,50px) scale(1.5);
3 }
```

CSS 中的 transform 是一个很强大的属性，它的作用其实也是对元素进行仿射变换。

它不仅支持 translate、rotate、scale 等值，还支持 matrix。CSS 的 matrix 是一个简写的齐次矩阵，因为它省略了 3 阶齐次矩阵第三行的 0, 0, 1 值，所以它只有 6 个值。

transform 在 CSS 中变换元素的方法，我们作为前端工程师都比较熟悉了。但你知道怎么优化它来提高性能吗？下面，我就重点来说说这一点。

结合上面介绍的齐次矩阵变换的原理，我们可以对 CSS 的 transform 属性进行压缩。举个例子，我们可以这么定义 CSS transform，代码如下：

[复制代码](#)

```
1 div.block {  
2   transform: rotate(30deg) translate(100px,50px) scale(1.5);  
3 }
```

也就是我们先旋转 30 度，然后平移 100px、50px，最后再放大 1.5 倍。实际上相当于我们做了如下变换：

$$\begin{pmatrix} 1.5 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 100 \\ 0 & 1 & 50 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

这里我就不再自己写矩阵乘法的库了，我们用一个向量矩阵运算的数学库 math，它几乎包含了所有图形学需要用到的数学方法，我们在后面课程中也会经常用到它，你可以参考 [🔗 GitHub 仓库](#) 先了解一下它。

我们简单算一下三个矩阵相乘，代码如下：

[复制代码](#)

```
1 import {multiply} from 'common/lib/math/functions/mat3fun.js';  
2  
3 const rad = Math.PI / 6;  
4 const a = [  
5   Math.cos(rad), -Math.sin(rad), 0,  
6   Math.sin(rad), Math.cos(rad), 0,  
7   0, 0, 1  
8 ];  
9  
10 const b = [  
11   1, 0, 100,  
12   0, 1, 50,
```

```
13     0, 0, 1
14 ];
15
16 const c = [
17     1.5, 0, 0,
18     0, 1.5, 0,
19     0, 0, 1
20 ];
21
22 const res = [a, b, c].reduce((a, b) => {
23     return multiply([], b, a);
24 });
25
26 console.log(res);
27 /*
28 [1.299038105676658, -0.7499999999999999, 61.60254037844388,
29  0.7499999999999999, 1.299038105676658, 93.30127018922192,
30  0, 0, 1]
31  */
```

所以呢，我们最终就可以将上面的 transform 用一个矩阵表示：

```
1 div.block {
2     transform: matrix(1.3,0.75,-0.75,1.3,61.6,93.3);
3 }
```

[复制代码](#)

这样的 transform 效果和之前 rotate、translate 和 scale 分开写的效果是一样的，但是字符数更少，所以能减小 CSS 文件的大小。

那在我们介绍完仿射变换之后，你是不是对 CSS transform 的理解也更深了呢？没错，不光是 transform，在我们之后的学习中，你也可以多想想，还有哪些内容在 CSS 中也有相似的作用，是不是也能利用在可视化中学到的知识来优化性能。

要点总结

这一节课我们介绍了用向量和矩阵运算来改变几何图形的形状、大小和位置。其中，向量的平移、旋转和缩放都属于仿射变换，而仿射变换具有 2 个性质：

1. 变换前是直线段的，变换后依然是直线段

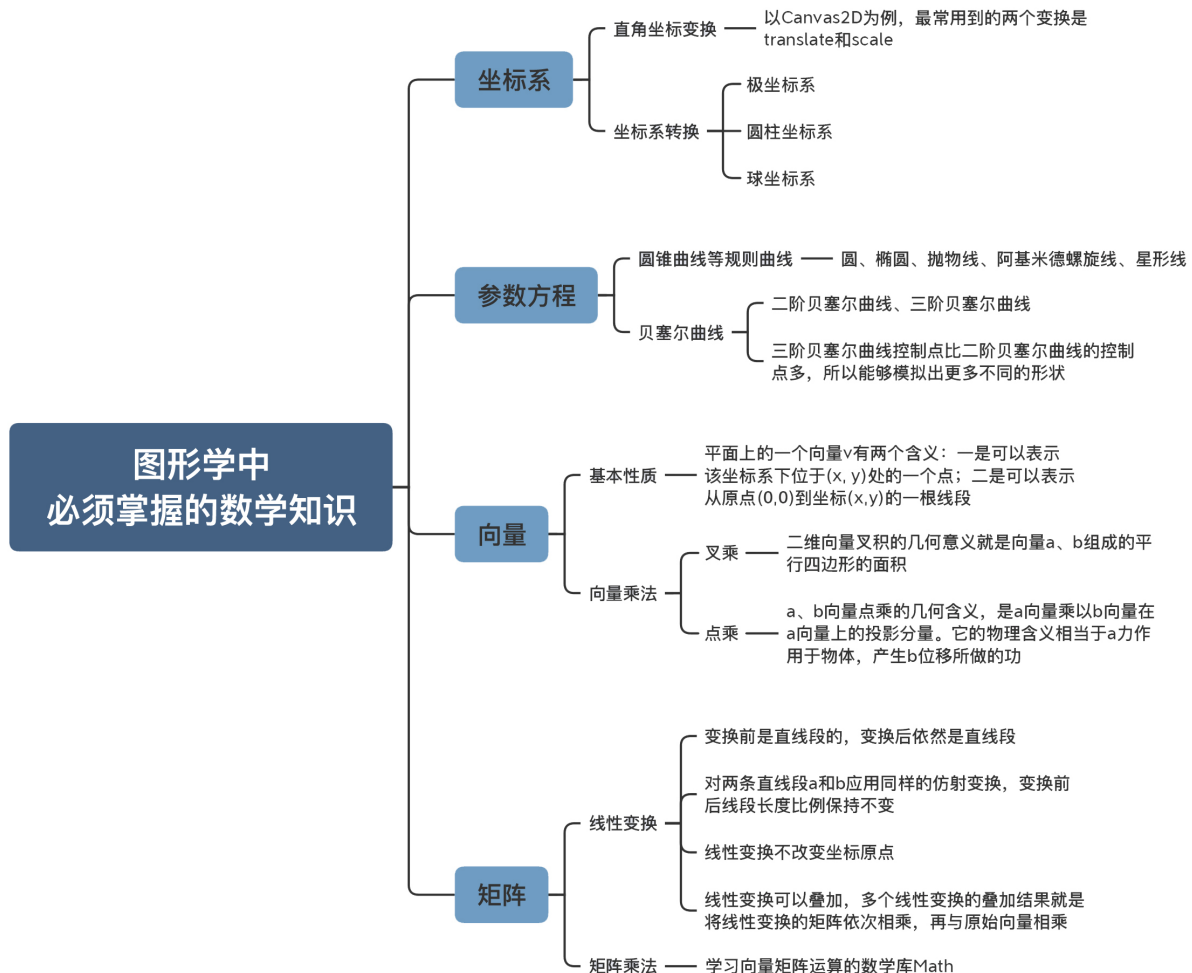
2. 对两条直线段 a 和 b 应用同样的仿射变换，变换前后线段长度比例保持不变

那仿射变换中的旋转和缩放又属于线性变换，而线性变换在仿射变换性质的基础上还有 2 个额外的性质：

1. 线性变换不改变坐标原点（因为如果 x_0 、 y_0 等于零，那么 x 、 y 肯定等于 0）
2. 线性变换可以叠加，多个线性变换的叠加结果就是将线性变换的矩阵依次相乘，再与向量相乘

通过齐次坐标和齐次矩阵，我们可以将平移这样的非线性仿射变换用更高维度的线性变换来表示。这么做的目的是让我们能够将仿射变换的组合简化为矩阵乘法运算。

到这里，数学基础篇的内容我们就学完了。在这一篇的开头，我们说了要总结出一个通用的基础数学绘图体系，这样才不至于陷入细节里。所以啊，我总结了一个简单的知识脑图，把我们在数学篇里讲过的数学知识汇总到了一起，它肯定不会是一个非常完整的数学绘图体系，但是对我们之后的学习来说，已经足够用了。



最后呢，我还想再啰嗦几句。图形学作为可视化的基础，是一门很深的学问。它牵涉的数学内容非常多，包括线性代数、几何、微积分和概率统计等等。那这门课里我们所介绍的数学知识，其实还都只是一些入门知识。

那如果你对图形学本身很感兴趣，想要深入学习它在其他领域，比如游戏、视频、AR/VR等领域的应用，这里我推荐你一些深入学习的资料。

1. [3Blue1Brown 的数学和图形学基础课程](#) 讲得深入浅出，是非常棒的入门教程。
2. [《Fundamentals of Computer Graphics》](#) 这本书是图形学入门的经典教材。

小试牛刀

1. 在实现粒子动画的时候，我们让 `translateMatrix * rotateMatrix * scaleMatrix`，这三个矩阵按这样的顺序相乘。那如果我们颠倒它们的相乘次序，把 `roateMatrix` 放到

translateMatrix 前面，或者把 scaleMatrix 放到 translateMatrix 前面，会产生什么样的结果呢？为什么呢？你可以思考一下，然后从 GitHub 上 fork 代码，动手试一试。

2. 我们知道，CSS 的 transform 除了 translate、rotate 和 scale 变换以外，还有 skew 变换。skew 变换是一种沿着轴向的扭曲变换，它也属于一种线性变换，它的变换矩阵是：

$$\begin{pmatrix} 1 & \tan \theta_x & 0 \\ \tan \theta_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

你可以使用这个矩阵，给我们的粒子动画加上随机的扭曲效果吗？

3. 因为齐次坐标和齐次矩阵的概念，可以从二维一直推广到 N 维，而且 CSS 的 transform 还支持 3D 变换。那你可以用齐次矩阵的原理对 CSS 属性的 3D 变换应用 matrix3d，实现出有趣的 3D 变换效果吗？（💡小提示：要支持 3 维的齐次坐标，需要 4 维齐次矩阵）？

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

[1]粒子动画的 [完整代码](#)

[2]矩阵运算数学库的 [完整代码](#)

推荐阅读

[1]WebGL 的 uniform 变量设置 [官方文档](#)

[2] [3Blue1Brown](#) 的数学和图形学基础课程

[3]图形学入门经典教材 [🔗](#) 《Fundamentals of Computer Graphics》

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 08 | 如何利用三角剖分和向量操作描述并处理多边形？

下一篇 10 | 图形系统如何表示颜色？


精选留言 (4)

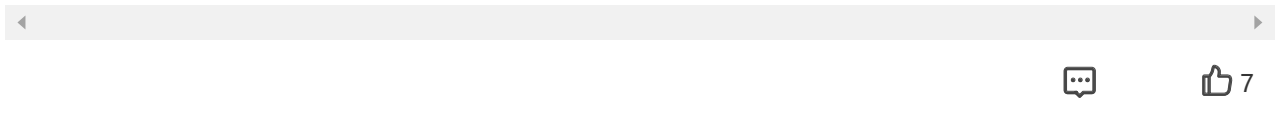
[🗨️ 写留言](#)

廖熊猫

2020-07-10

3Blue1Brown 的数学和图形学基础课程，打不开的同学，在3B1B在B站也有账号，老师发的这个是《线性代数的本质》，搜索一下就能找到了，这个视频看完以后对矩阵的理解就能深入很多了

作者回复: 



阿鑫

2020-07-19

数学知识全还给老师了

展开 



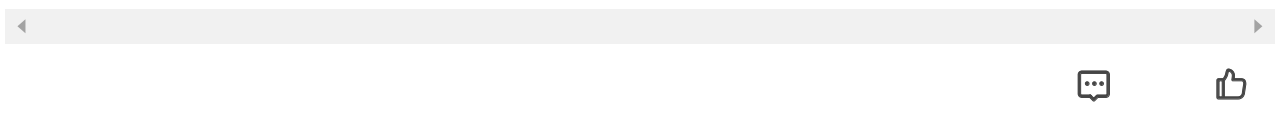
Geek_frank

2020-07-14

请问关于 webgl绘图如何代码测试？我照着代码敲一遍，没有变换效果，也不报错。把git hub的代码复制过来就可以运行了。

展开 

作者回复: 如果没报错，只是shader写的问题导致结果不对，调试起来是比较麻烦，只能有耐心地修改并测试结果了



Geek_frank

2020-07-13

。。。学了这么久的线性代数，现在终于知道一点某些概念在现实中的映射了！

作者回复: 哈哈，我觉得大学课程安排失败，大一大二基础课并不告诉你这些内容的实际用途，所以没动力学。大三大四开始应用的时候才发现基础没学好。

