

学习ejoy2d——matrix

gaccob

2014 年 1 月 25 日

1. 矩阵理论基础

在学习这一段代码之前，先引入一些矩阵的基础理论。

- 2d图形的变换，可以用一个6元组 $\{a, b, c, d, t_x, t_y\}$ 来表示，其中， a 和 d 表示缩放， b 和 c 表示旋转， t_x 和 t_y 表示位移，这样就能得到一个基础的2d矩阵公式：

$$x = ax + cy + t_x \quad (1)$$

$$y = bx + dy + t_y \quad (2)$$

- 为了把二维图形的变化统一在一个坐标系里，引入齐次坐标的概念，即把一个图形用一个三维矩阵表示，其中第三列总是 $(0, 0, 1)$ ，用来作为坐标系的标准，这样，变换6元组就可以用一个3*3的矩阵来表示：

$$M = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix} \quad (3)$$

这样，我们就可以得到一个矩阵变换公式：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * M = \begin{bmatrix} ax + cy + t_x & bx + dy + t_y & 1 \end{bmatrix} \quad (4)$$

- 平移：当 $a = d = 1$ ， $c = b = 0$ 时，则(4)退化为：

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * M = \begin{bmatrix} x + t_x & y + t_y & 1 \end{bmatrix} \quad (5)$$

- **缩放**: 当 $b = c = t_x = t_y = 0$ 时, 则(4)退化为:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * M = \begin{bmatrix} ax & dy & 1 \end{bmatrix} \quad (6)$$

- **旋转**: 当 $t_x = t_y = 0$, $a = \cos\theta$, $b = \sin\theta$, $c = -\sin\theta$, $d = \cos\theta$ 时, 则(4)退化为:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} * M = \begin{bmatrix} x * \cos\theta - y * \sin\theta & x * \sin\theta + y * \cos\theta & 1 \end{bmatrix} \quad (7)$$

- **归一化**: 当 $a = d = 1$, $c = d = t_x = t_y = 0$ 时, 则M退化为归一化矩阵:

$$E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (8)$$

- **逆矩阵**: 所谓逆矩阵, 就是对于矩阵A, 存在矩阵B, $A*B = B*A = E$, 其中, E是归一化矩阵, 那么A和B就互为各自的逆矩阵. 对于M而言, 它的逆矩阵可以计算得到:

$$N = \begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} & 0 \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} & 0 \\ \frac{ct_y-dt_x}{ad-bc} & \frac{bt_x-at_y}{ad-bc} & 1 \end{bmatrix} \quad (9)$$

$$M * N = N * M = E \quad (10)$$

2. 源码分析

简单回顾一下理论, 下面就可以开始ejoy2d的矩阵源码分析, lib/matrix.c和lib/matrix.h封装了矩阵的操作, lib/lmatrix.c和lib/lmatrix.h是lua调用c的接口.

1. 六元组表示的矩阵(matrix):

```
1 struct matrix {
2     int m[6];
3 };
```

2. 矩阵归一化(matrix identity).

```

1 matrix_identity(struct matrix *mm) {
2     int *mat = mm->m;
3     mat[0] = 1024;
4     mat[1] = 0;
5     mat[2] = 0;
6     mat[3] = 1024;
7     mat[4] = 0;
8     mat[5] = 0;
9 }

```

与(3)中不一样的, 为了避免浮点运算, 这里以1024为基处理, 来保持整数运算, 实际上, 现在的变换矩阵(3)演变为:

$$M_{ejoy} = M * \begin{bmatrix} 1024 & 0 & 0 \\ 0 & 1024 & 0 \\ 0 & 0 & 1024 \end{bmatrix} = \begin{bmatrix} 1024 * a & b & 0 \\ c & 1024 * d & 0 \\ t_x & t_y & 1024 \end{bmatrix} \quad (11)$$

3. 矩阵的乘法(matrix multiply).

```

1 static inline void
2 matrix_mul(struct matrix *mm, const struct matrix *mm1,
3             const struct matrix *mm2) {
4     int *m = mm->m;
5     const int *m1 = mm1->m;
6     const int *m2 = mm2->m;
7     m[0] = (m1[0] * m2[0] + m1[1] * m2[2]) / 1024;
8     m[1] = (m1[0] * m2[1] + m1[1] * m2[3]) / 1024;
9     m[2] = (m1[2] * m2[0] + m1[3] * m2[2]) / 1024;
10    m[3] = (m1[2] * m2[1] + m1[3] * m2[3]) / 1024;
11    m[4] = (m1[4] * m2[0] + m1[5] * m2[2]) / 1024 + m2[4];
12    m[5] = (m1[4] * m2[1] + m1[5] * m2[3]) / 1024 + m2[5];
13 }

```

这里就是3*3的矩阵乘法, 唯一做了变化的就是在乘完之后除1024来保持scale.

4. 矩阵的逆运算(matrix inverse).

```

1 int
2 matrix_inverse(const struct matrix *mm, struct matrix *
3                mo) {
4     const int *m = mm->m;
5 }

```

```

4 | int *o = mo->m;
5 | if (m[1] == 0 && m[2] == 0) {
6 |     return _inverse_scale(m, o);
7 | }
8 | if (m[0] == 0 && m[3] == 0) {
9 |     return _inverse_rot(m, o);
10 | }
11 | int t = m[0] * m[3] - m[1] * m[2] ;
12 | if (t == 0)
13 |     return 1;
14 | o[0] = (int32_t)((int64_t)m[3] * (1024 * 1024) / t);
15 | o[1] = (int32_t)(- (int64_t)m[1] * (1024 * 1024) / t);
16 | o[2] = (int32_t)(- (int64_t)m[2] * (1024 * 1024) / t);
17 | o[3] = (int32_t)((int64_t)m[0] * (1024 * 1024) / t);
18 | o[4] = - (m[4] * o[0] + m[5] * o[2]) / 1024;
19 | o[5] = - (m[4] * o[1] + m[5] * o[3]) / 1024;
20 | return 0;
21 | }

```

这里有几点需要注意的:

- 因为归一化时做了1024的线性变换, 所以这里有1024的参数.
- 当 $b = c = 0$ 时, 退化为线性变换_inverse_scale, 这里做了简化处理提高效率.
- 当 $a = d = 0$ 时, 退化为旋转变换_inverse_rot, 这里同样也做了简化处理提高效率.

5. 矩阵的缩放(matrix scale).

```

1 | static inline void
2 | scale_mat(int *m, int sx, int sy) {
3 |     if (sx != 1024) {
4 |         m[0] = m[0] * sx / 1024;
5 |         m[2] = m[2] * sx / 1024;
6 |         m[4] = m[4] * sx / 1024;
7 |     }
8 |     if (sy != 1024) {
9 |         m[1] = m[1] * sy / 1024;
10 |        m[3] = m[3] * sy / 1024;
11 |        m[5] = m[5] * sy / 1024;
12 |     }
13 | }

```

我们在(6)中说到, 当 M 退化为只有 a 和 d 时, 可以看做一个缩放(scale). 这里的 a 对应就是代码中的 sx , b 对应的就是代码中的 sy . 此时

的(4)演进成如下的形式:

$$M * \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx * a & sy * b & 0 \\ sx * c & sy * d & 0 \\ sx * t_x & sy * t_t & 1 \end{bmatrix} \quad (12)$$

6. 矩阵的旋转(matrix rotate)

```

1 static inline void
2 rot_mat(int *m, int d) {
3     if (d==0)
4         return;
5     int cosd = icosd(d);
6     int sind = isind(d);
7
8     int m0_cosd = m[0] * cosd;
9     int m0_sind = m[0] * sind;
10    int m1_cosd = m[1] * cosd;
11    int m1_sind = m[1] * sind;
12    int m2_cosd = m[2] * cosd;
13    int m2_sind = m[2] * sind;
14    int m3_cosd = m[3] * cosd;
15    int m3_sind = m[3] * sind;
16    int m4_cosd = m[4] * cosd;
17    int m4_sind = m[4] * sind;
18    int m5_cosd = m[5] * cosd;
19    int m5_sind = m[5] * sind;
20
21    m[0] = (m0_cosd - m1_sind) /1024;
22    m[1] = (m0_sind + m1_cosd) /1024;
23    m[2] = (m2_cosd - m3_sind) /1024;
24    m[3] = (m2_sind + m3_cosd) /1024;
25    m[4] = (m4_cosd - m5_sind) /1024;
26    m[5] = (m4_sind + m5_cosd) /1024;
27 }

```

在(7)这种情况下, M 退化为旋转矩阵, 此处代码的原理也很容易理解:

$$M * \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & -\cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta * a - \sin\theta * b & \sin\theta * a + \cos\theta * b & 0 \\ \cos\theta * c - \sin\theta * d & \sin\theta * c + \cos\theta * d & 0 \\ \cos\theta * t_x - \sin\theta * t_y & \sin\theta * t_x + \cos\theta * t_y & 1 \end{bmatrix} \quad (13)$$

代码中的 d 是经过变换的角度, 这个在lua的接口中有体现, 源码截取如下:

```

1 double r = luaL_checknumber(L, 2);
2 matrix_rot(m, r * (1024.0 / 360.0));

```

代码中对角度的计算做了一个 \cos 表, \cos 和 \sin 的计算都是通过查表来得到的, 代码中做 \sin 计算时的64, 是 $\sin\theta = \cos(90 - \theta)$ 经过1024换算后得到的.

```

1 static inline int
2 icost(int dd) {
3     static int t[256] = {
4         ...
5     };
6     if (dd < 0) {
7         dd = 256 - (-dd % 256);
8     } else {
9         dd %= 256;
10    }
11
12    return t[dd];
13 }
14
15 static inline int
16 icosd(int d) {
17     int dd = d/4;
18     return icost(dd);
19 }
20
21 static inline int
22 isind(int d) {
23     int dd = 64 - d/4;
24     return icost(dd);
25 }

```

7. 矩阵的SRT变换

ejoy2d.matrix中用一个srt(scale, rotate, translate)结构体来封装了变换矩阵, 对矩阵的srt操作依次就是:scale_mat, rot_mat以及translate(直接增加 M 中的 t_x 和 t_y).

```

1 struct srt {
2     int offx;
3     int offy;
4     int scalex;
5     int scaley;
6     int rot;
7 };

```

```

8
9 void
10 matrix_srt(struct matrix *mm, const struct srt *srt) {
11     scale_mat(mm->m, srt->scalex, srt->scaley);
12     rot_mat(mm->m, srt->rot);
13     mm->m[4] += srt->offx;
14     mm->m[5] += srt->offy;
15 }
16
17
18 void
19 matrix_sr(struct matrix *mat, int sx, int sy, int d) {
20     int *m = mat->m;
21     int cosd = icosd(d);
22     int sind = isind(d);
23
24     int m0_cosd = sx * cosd;
25     int m0_sind = sx * sind;
26     int m3_cosd = sy * cosd;
27     int m3_sind = sy * sind;
28
29     m[0] = m0_cosd / 1024;
30     m[1] = m0_sind / 1024;
31     m[2] = -m3_sind / 1024;
32     m[3] = m3_cosd / 1024;
33 }

```

其中函数matrix_sr()是 $b = c = 0$ 后的简化演进，下面的公式省略了ejoy2d中1024的scale:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx * \cos\theta & sx * \sin\theta & 0 \\ -sy * \sin\theta & sy * \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (14)$$

8. 矩阵的lua接口

```

1 int
2 ejoy2d_matrix(lua_State *L) {
3     luaL_Reg l[] = {
4         { "new", lnew },
5         { "scale", lscale },
6         { "trans", ltrans },
7         { "rot", lrot },
8         { "inverse", linverse },
9         { "mul", lmul },

```

```
10         { "tostring", ltostring },
11         { "identity", lidentity},
12         { NULL, NULL },
13     };
14     luaL_newlib(L, l);
15     return 1;
16 }
```

这其中要注意的一点是，如果`lnew()`没有输入参数的话，则返回归一矩阵(scale 1024).