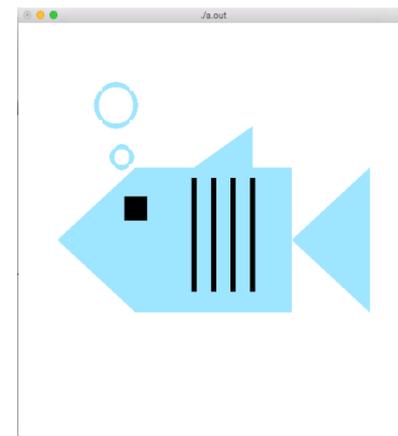
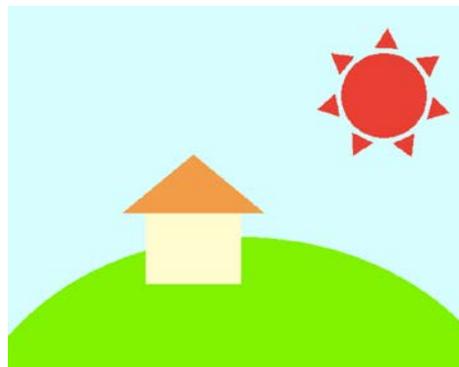
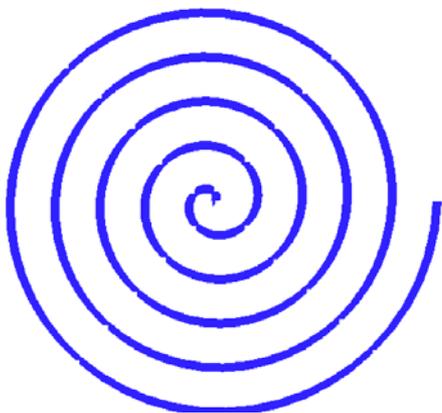
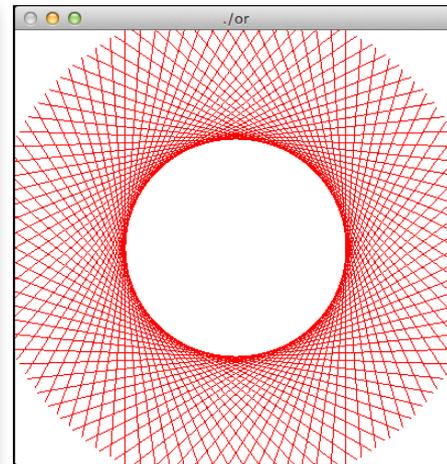
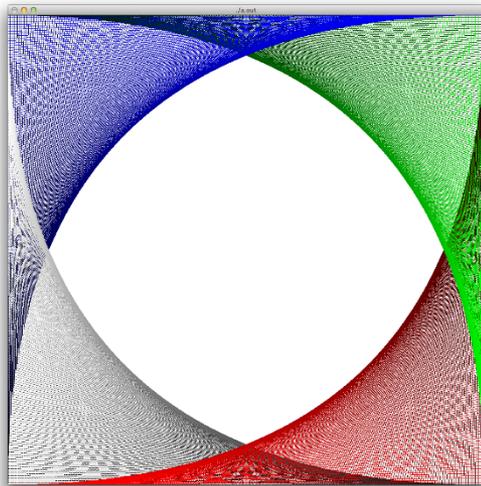
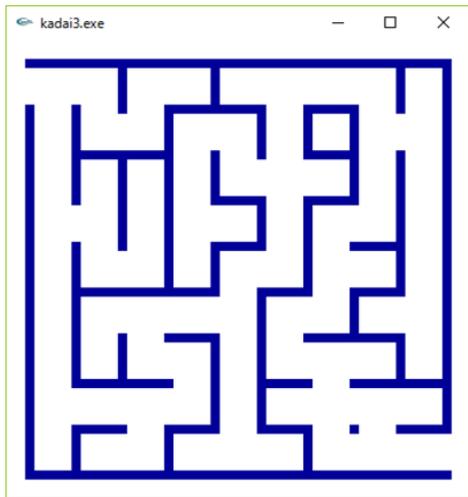
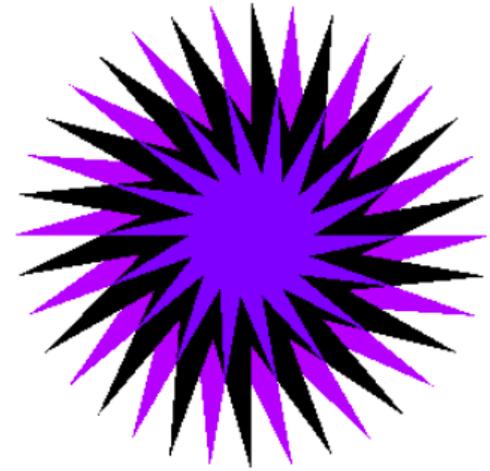
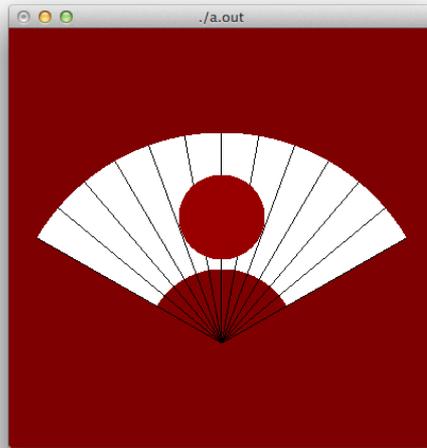
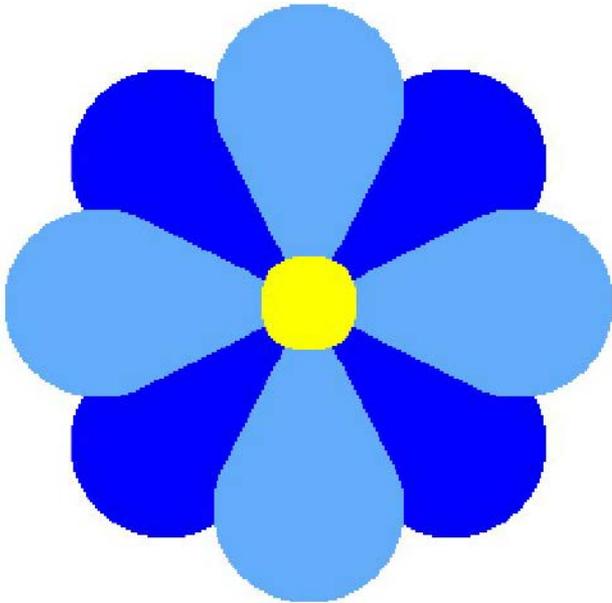
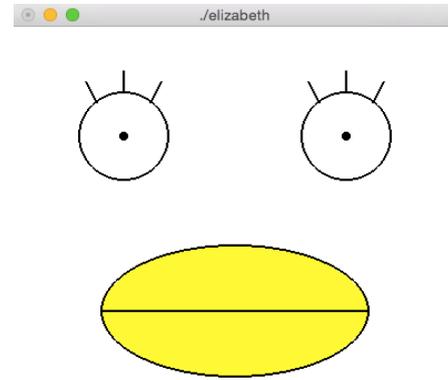
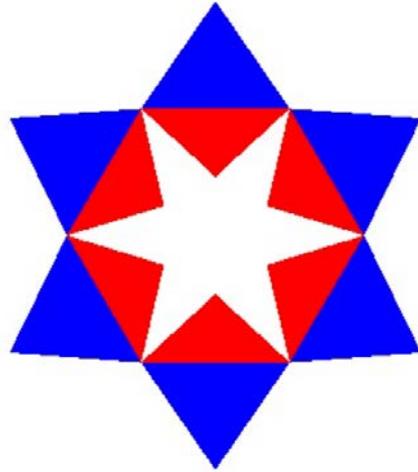
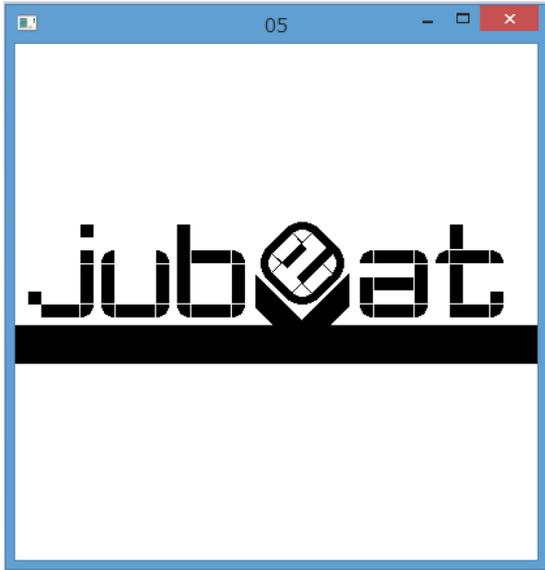


# コンピュータグラフィックス 基礎

## 第2回 2次元の座標変換

金森 由博



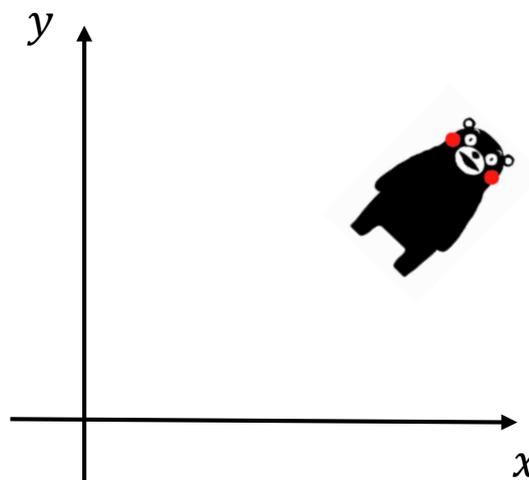
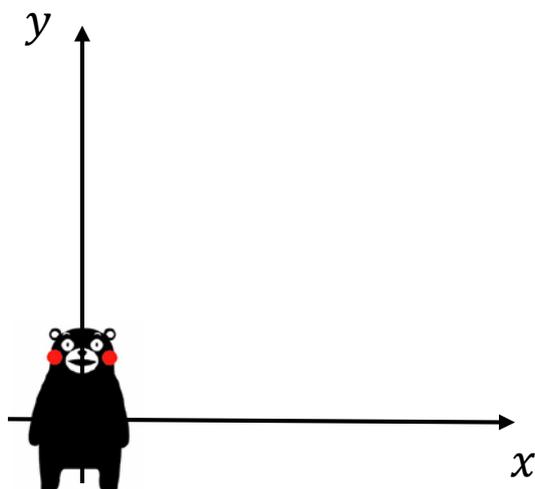


# 学習の目標

- 行列を用いた2次元での座標変換について理解する
- 2次元の座標変換を用いたプログラムの作成を行えるようになる

# 座標変換とは

- ある物体の位置や向き、形を変えること
  - 形を変える？（大きさの変化、せん断変形など）



- 位置や向きは相対的なものであることに注意
  - 物体が移動した？
  - カメラが移動した（座標系が変わった）？

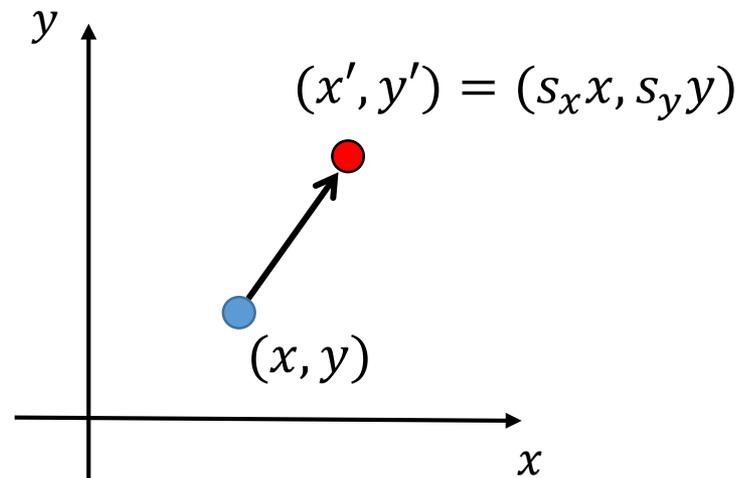
# 座標変換の式 (1/2)

- 拡大縮小

$$x' = s_x x$$

$$y' = s_y y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

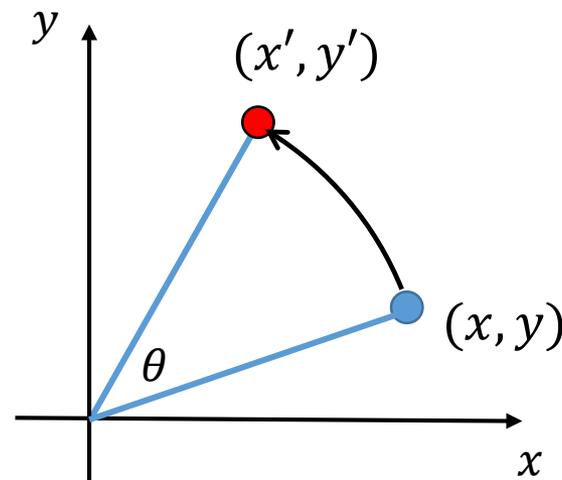


- 回転移動 (原点中心)

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

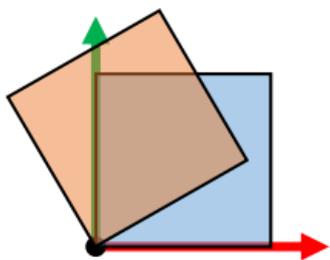
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



# 線形変換

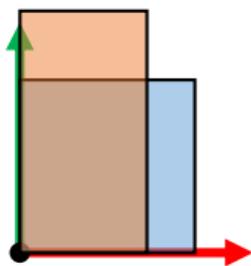
直線を直線に移す変換

回転



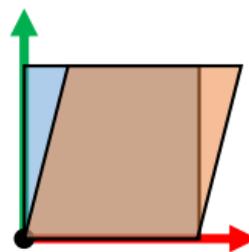
$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

スケーリング



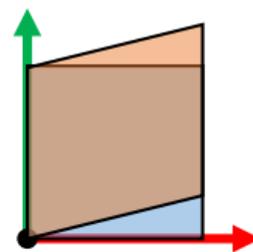
$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

せん断 (X方向)



$$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

せん断 (Y方向)



$$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

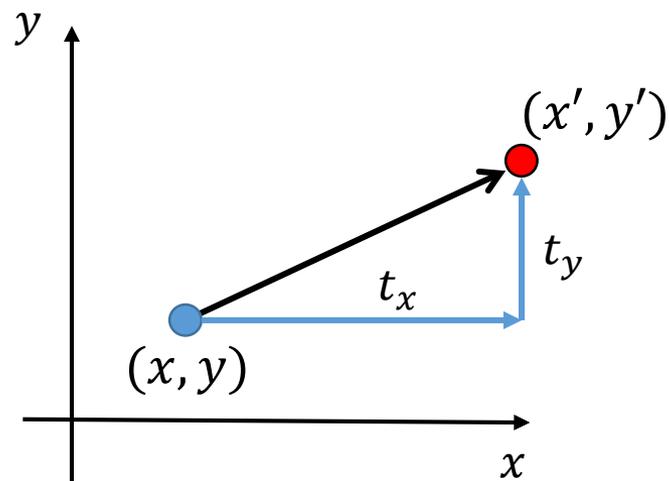
# 座標変換の式 (2/2)

- 平行移動

$$x' = x + t_x$$

$$y' = y + t_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



# アフィン変換（線形変換＋平行移動）

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

座標変換を表す操作に  
「積」と「和」の演算が含まれる。  
何かと不便なことが多い。



便宜上、1次元を上げたベクトル  
(3次元ベクトル) を用いる (同次座標)。

変換行列

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

平行移動を含む座標変換操作が  
単純な「積」の演算1つで表現された。  
(3次元ベクトルと3x3行列の演算)  
これは便利！

複数回の座標変換操作が  
単純な行列の積の繰り返しだけで済む

# 3x3行列を用いた座標変換

拡大縮小

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

回転移動

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

平行移動

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# アフィン変換の合成（変換行列の掛け合わせ）

- 異なるアフィン変換を組み合わせる座標変換



- 変換行列の掛け合わせ

注：行列は掛け合わせる順番によって結果が異なる！

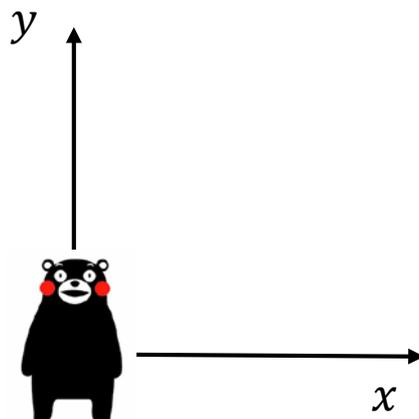
# 掛けあわせ順による結果の違い

60°の回転移動（原点中心）

$$R = \begin{bmatrix} \cos 60^\circ & -\sin 60^\circ & 0 \\ \sin 60^\circ & \cos 60^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(5,0) の平行移動

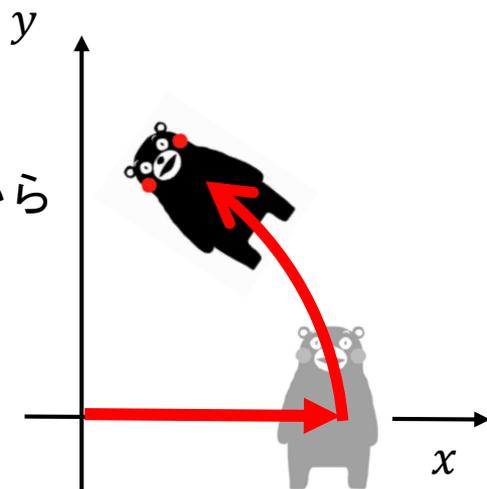
$$T = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



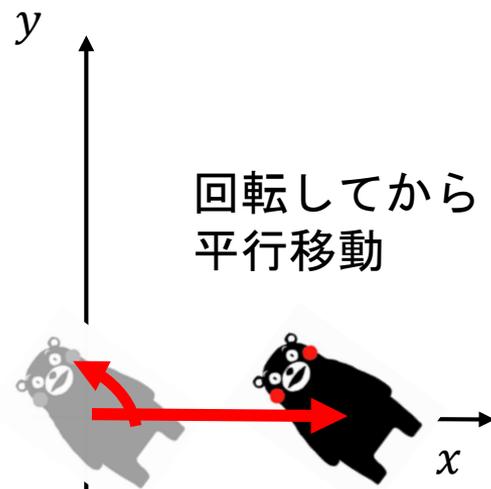
$$\mathbf{x}' = R\mathbf{T}\mathbf{x}$$

$$\mathbf{x}' = T\mathbf{R}\mathbf{x}$$

平行移動してから  
回転



回転してから  
平行移動

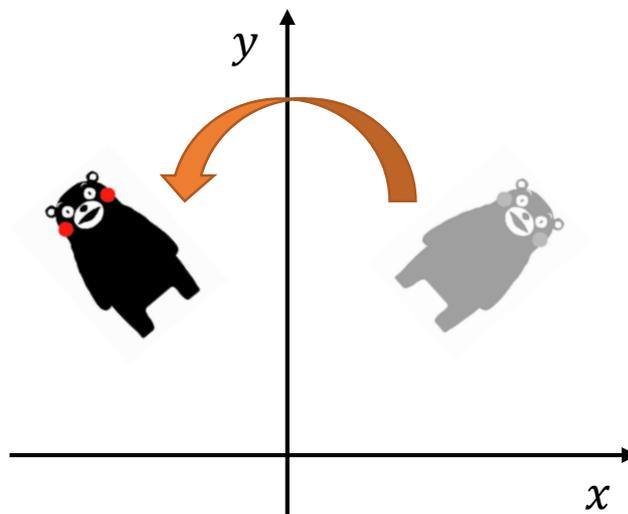


# 様々な座標変換

鏡映変換

(y 軸に関して)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

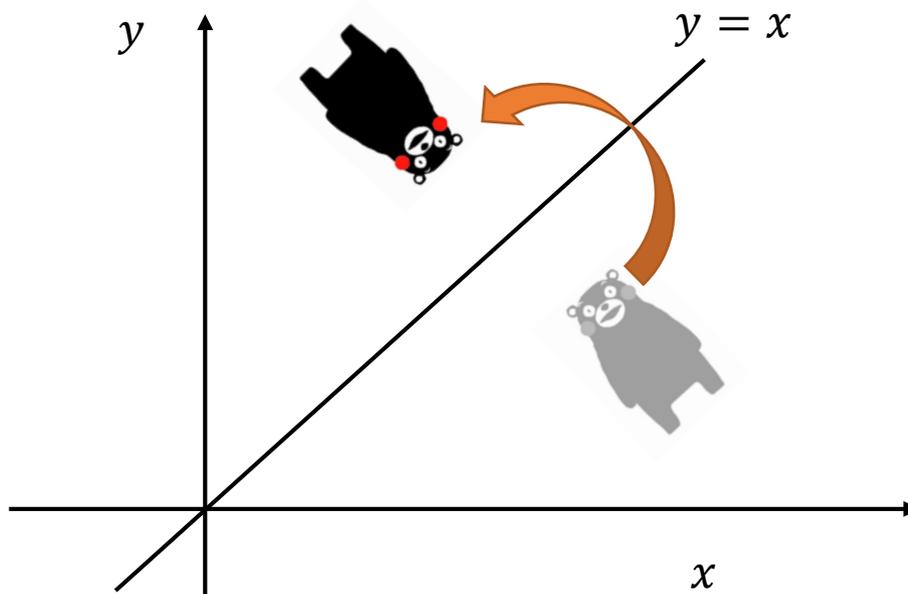


# 様々な座標変換

鏡映変換

( $x=y$  に関して)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

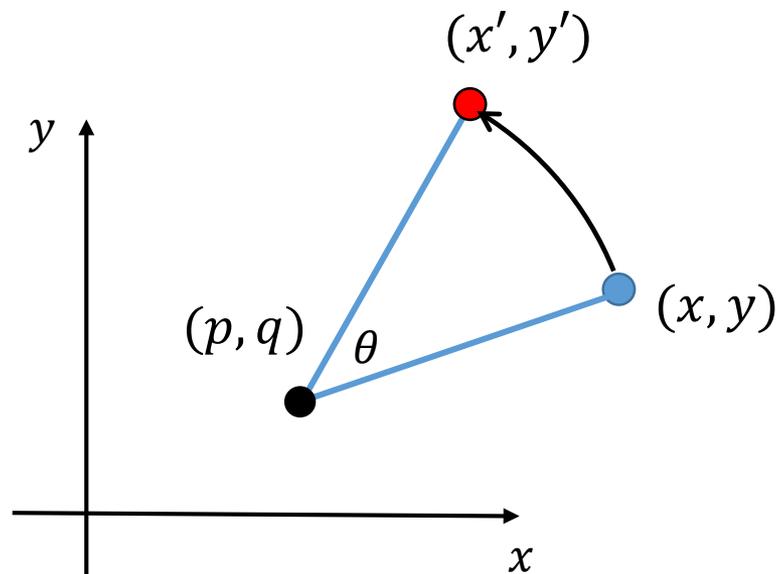


# 様々な座標変換

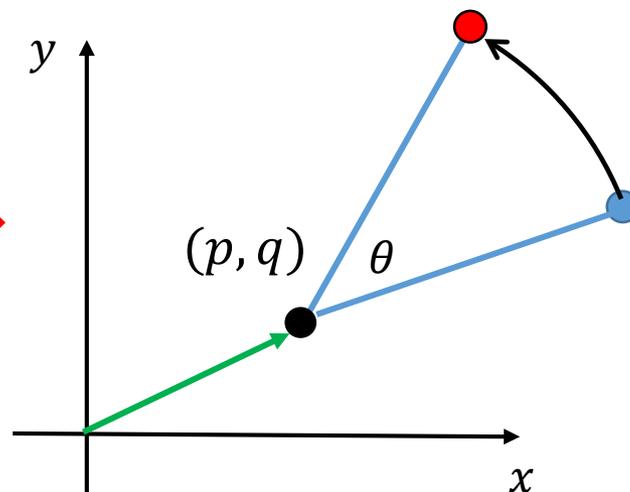
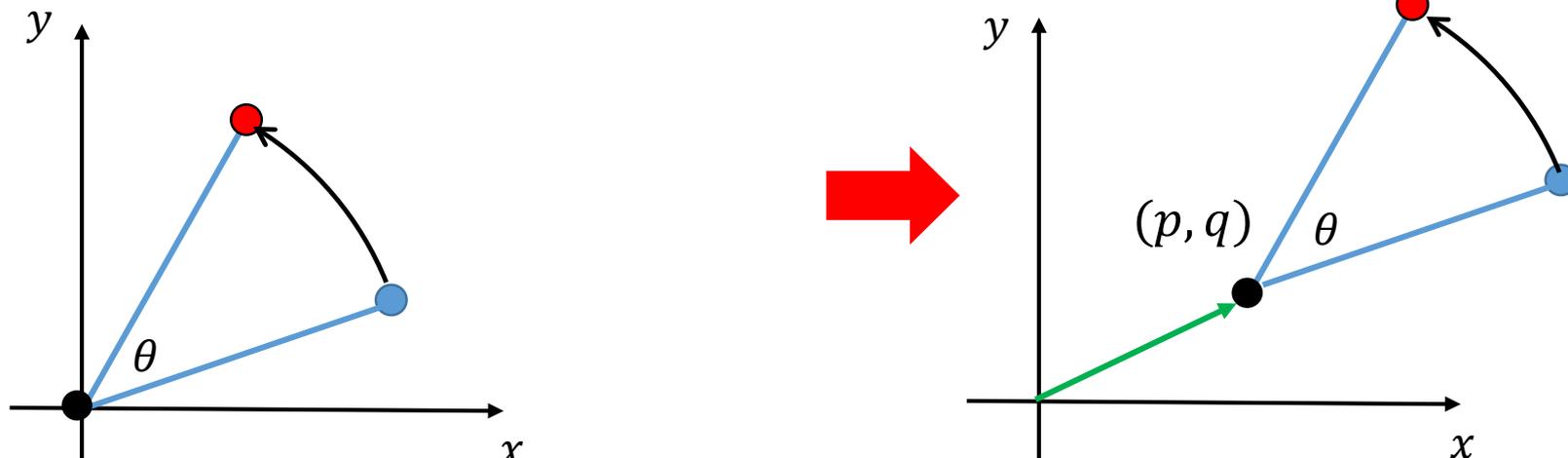
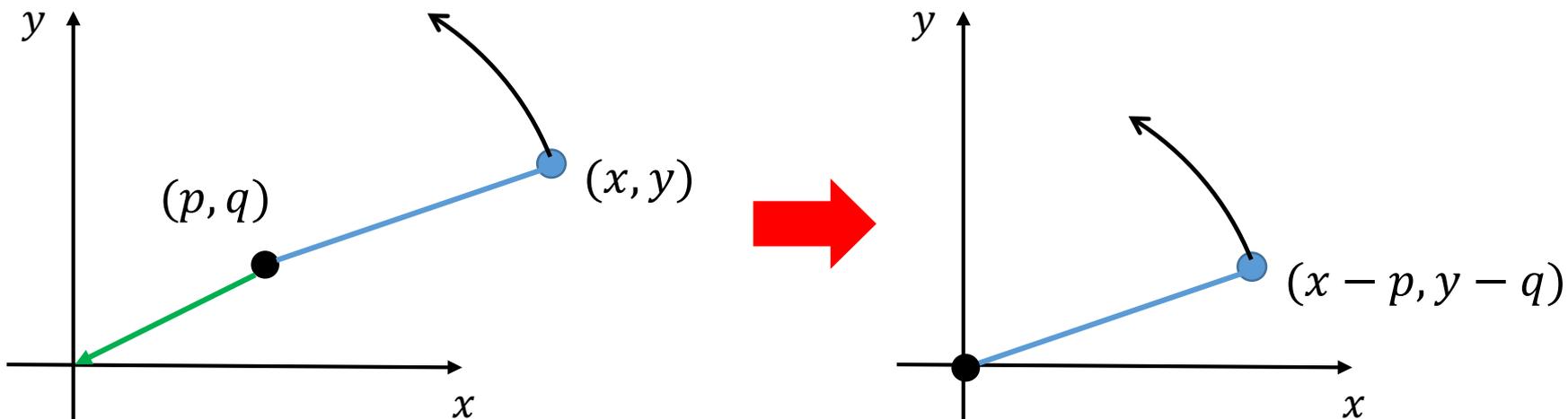
原点以外の点を中心とした回転

1. 回転の中心が原点となるような座標変換（平行移動）
2. 回転操作
3. Step.1で行った移動を元に戻すような平行移動

例：  $(p, q)$  を中心とした角度  $\theta$  の回転



# 点(p, q) 周りの回転



# 点(p, q) 周りの回転

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & p \\ 0 & 1 & q \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -p \\ 0 & 1 & -q \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{x}' = T^{-1} R T \mathbf{x}$$

$R$  : 回転行列

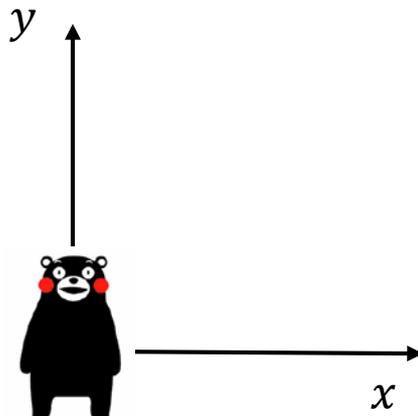
$T$  : 回転中心を原点に移す平行移動行列

$T^{-1}$  :  $T$  の逆行列

# OpenGL での座標変換



記述する順番で結果が異なる



```
glRotated( 60, 0, 0, 1);  
glTranslated( 5, 0, 0);
```

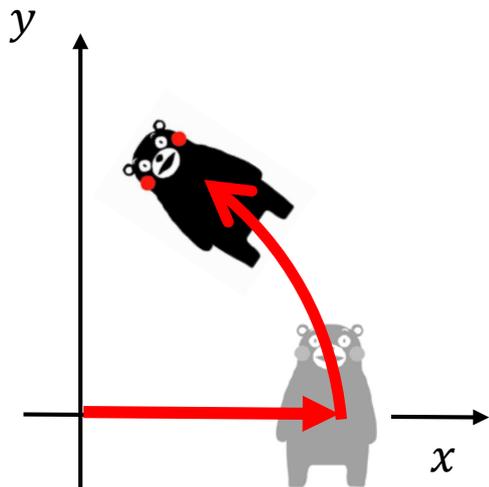
```
glTranslated( 5, 0, 0);  
glRotated( 60, 0, 0, 1);
```



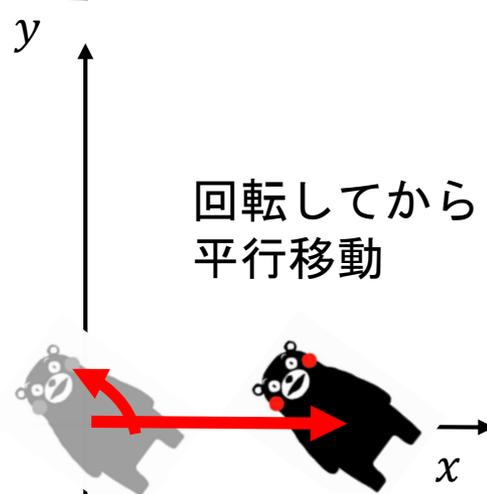
$$\mathbf{x}' = R\mathbf{T}\mathbf{x}$$



$$\mathbf{x}' = T\mathbf{R}\mathbf{x}$$



平行移動してから  
回転



回転してから  
平行移動

変換行列の演算式とプログラムコードでの  
記述する順番は同じ

記述した後ろのものから順番に適用される

{gl Scaled, gl Rotated, gl Translated} は、  
{拡大縮小、回転、平行移動} 行列を  
現在の変換行列に、**右から掛ける**。

# 変換行列の操作

## `glLoadIdentity()`

変換行列を初期化する。描画更新をするたびに初期化が必要。  
(そうしないと、前回描画した時の変換行列が残ったままになっている。)

## `glMultMatrixd(GLdouble *m)`

現在の変換行列に掛け合わせる行列を直接指定する。  
引数に、16個の要素を持つ配列を指定する (3次元を基本とするので、  
4x4行列の各要素) (授業の中では使用しない)

## `glLoadMatrixd(GLdouble *m)`

現在の変換行列を取得する。引数に、16個の要素を持つ配列を指定する。  
(3次元を基本とするので、4x4行列の各要素)  
(授業の中では使用しない)

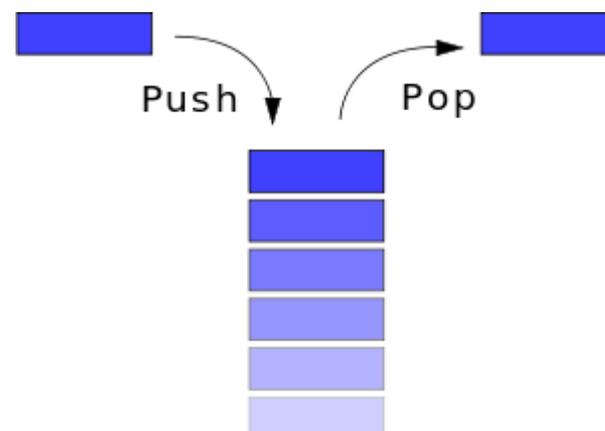
# 座標変換行列の Push と Pop

# 変換行列の格納（Push）と取り出し（Pop）

グラフィックスプログラムでは、ある時点の座標変換行列を保存しておき、後でまた、その行列を取り出したくなるのがよくある。

これを実現するために、行列スタックを使用する。

行列スタックに座標変換行列を格納し（Push）、必要な時に取り出す（Pop）。好きな順番には取りだせない。最後に入れたものが最初に取り出される。



スタック

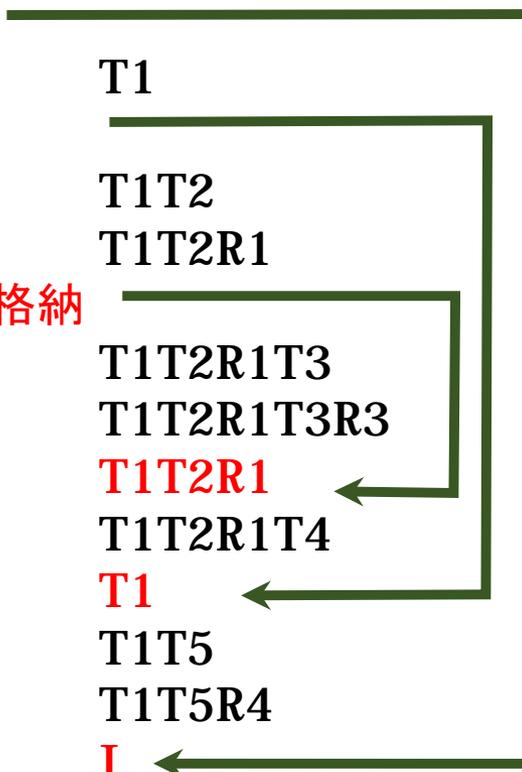
データの保存と取り出しを行う方法の1つ。後から入れたものが先に取りだされる（後入れ先出し）

# 変換行列の格納 `glPushMatrix` と 取り出し `glPopMatrix`

座標変換行列の状態

```
glLoadIdentity(); // I (単位行列)
glPushMatrix(); // スタックに[I]を格納
glTranslated(##, ##, 0); // T1
glPushMatrix(); // スタックに[T1]を格納
glTranslated(##, ##, 0); // T2
glRotated(##, 0, 0, 1); // R1
glPushMatrix(); // スタックに[T1T2R1]を格納
glTranslated(##, ##, 0); // T3
glRotated(##, 0, 0, 1); // R3
glPopMatrix(); // スタックから取出
glTranslated(##, ##, 0); // T4
glPopMatrix(); // スタックから取出
glTranslated(##, ##, 0); // T5
glRotated(##, 0, 0, 1); // R4
glPopMatrix(); // スタックから取出
```

I  
T1  
T1T2  
T1T2R1  
T1T2R1T3  
T1T2R1T3R3  
T1T2R1  
T1T2R1T4  
T1  
T1T5  
T1T5R4  
I



# 変換行列の階層化

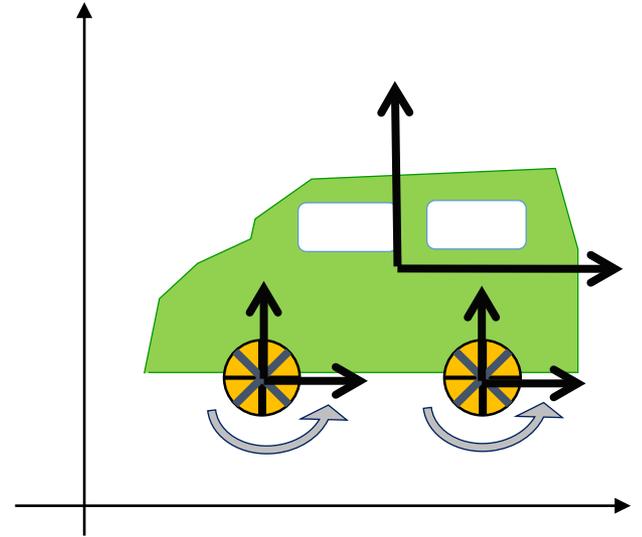
例：

車が移動している（平行移動）。

車には車輪が付いている。

車輪はそれぞれの軸を中心に回転している。

（平行移動と回転）



```
glLoadIdentity(); // I
glTranslated( #, #, 0); // 車体の座標系の移動
/* 車体の描画 */
glPushMatrix(); // 現在の变换行列をスタックに格納
glTranslated( #, #, 0); // 車体に対する前輪タイヤの座標系移動
glRotated( #, 0, 0, 1); // 前輪タイヤ座標での回転
/* 前輪の描画 */
glPopMatrix(); // スタックから取出
glTranslated( #, #, 0); // 車体に対する後輪タイヤの座標系移動
glRotated( #, 0, 0, 1); // 後輪タイヤ座標での回転
/* 後輪の描画 */
```

# ディスプレイリスト

- 同じ形を複数描画するときには？

1. 特定の形を描画する関数を作成して、その関数を呼び出す。

2. ディスプレイリストを作成しておいて、それを用いる。

描画命令の集合に ID 番号を割り振る。  
次回からは、この ID で描画を指示する。

# ディスプレイリストの使用

ディスプレイリストの作成

```
gl NewLi st ( 1 , GL_COMPILE );
```

好きな整数を割り当てる

```
/* 例えば、「車輪」を描画する命令の記述 */
```

```
gl EndLi st ();
```

ディスプレイリストの呼び出し

```
gl Cal l Li st ( 1 );
```

このような記述で現在の座標変換行列を使って「車輪」が描かれる

例 :

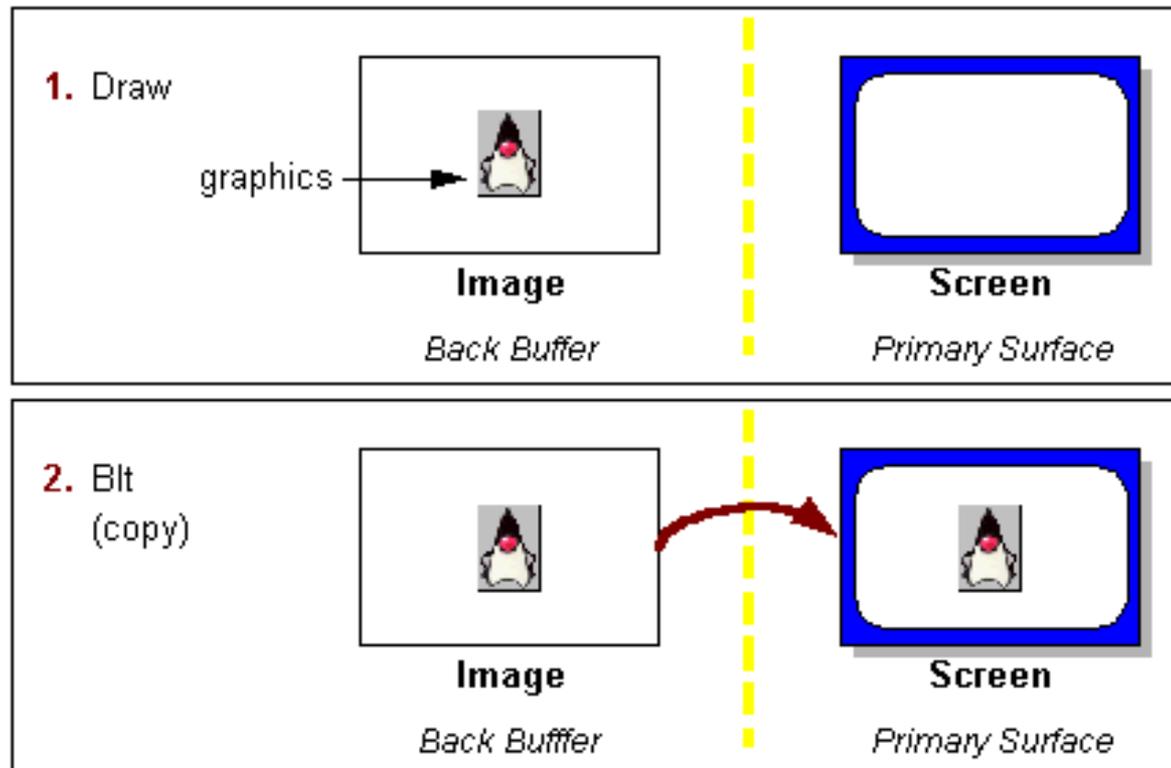
```
gl PushMatrix();  
gl Translated( #, #, 0 );  
gl Cal l Li st ( 1 );  
gl PopMatrix();  
gl Translated( #, #, 0 );  
gl Cal l Li st ( 1 );
```

# アニメーション表示のための OpenGL コマンド

# ダブルバッファリング

- 二つのフレームバッファ（描画用メモリ）を使用する
- バックバッファに描画が終わったら、画面に結果を転送する

## Double Buffering



# OpenGL でのダブルバッファリング

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
```

main関数で行う初期設定

```
glutSwapBuffers();
```

バッファの転送。

display関数での描画処理が終わったあとに

「glFlush();」の代わりに記述する

# 一定時間ごとの処理実行

`glutTimerFunc(ミリ秒, 呼び出す関数, 関数に渡す値);`

// 100ミリ秒ごとに実行される

```
void timer(int value) {
```

```
    何か処理する
```

```
    glutPostRedisplay(); // 再描画命令
```

```
    glutTimerFunc(100, timer, 0); // 100ミリ秒後に自身を実行する
```

```
}
```

```
int main(int argc, char ** argv) {
```

```
    省略
```

```
    glutTimerFunc(100, timer, 0);
```

```
    glutMainLoop();
```

```
    return 0;
```

```
}
```

2次元



これだけで、自由な図形

アニメーションができる！！！！

# 課題

- サンプルコードをコンパイルして実行してみる
- コード中の数字を変えて、結果がどのように変化するか確認する
- 星が回転するアニメーション

