# Lecture ~~4~~: 5
# Transformations

## Part 1 of several

# Last Time

- Canvas pen model
- State stacks
- 2D Primitives
    - polygons

- translate

# Today: Transformations

1. Translation - the simplest transformation
   - think in terms of coordinate systems
   - thinking forwards and backwards
2. Undo
3. Composition
4. More types of transformations
   - Scale
   - Rotate
5. Combining types (maybe next time)

# Transform vs. Transformation

Transform is a verb or a noun

Transformation is a noun

# Thinking about Transformation

with demos...

```
context.fillStyle="blue";
context.fillRect(20,20,20,20);
context.translate(20,0,20,0);
context.fillStyle="red";
context.fillRect(0,0,20,20);
```

(demo 1 - same square)

- both squares in the same place
- translate effects later points
- translate changes the coordinate system
- reading programs backwards

# Forwards and Backwards

Transformations move the coordinate system

Global to local

## Another way to think about it...

Work backwards from the object

The code is not written this way (backwards)

You cannot run your code **backwards**

(the demo program animates how to think about it)

# Caveat to in-class "code"

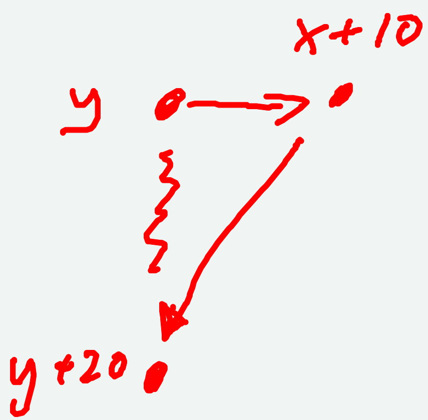
```
context.fillRect(x,y, w,h);
```

- w,h are not coordinates - how are they transformed?
- demo code actually draws rectangle with lineto
- use a triangle instead today...

# An Example Triangle

```
function tri(x,y) {
    context.beginPath();
    context.moveTo(x,y);
    context.lineTo(x+10,y);
    context.drawTo(x,y+20);
    context.closePath();
    context.fill();
    context.stroke();
}
```

- Intentionally not symmetric



# We could use a transformation!

```
function tri(x,y) {


    context.beginPath();
    context.moveTo(x,y);
    context.lineTo(x+10,y);
    context.drawTo(x,y+20);
    context.closePath();
    context.fill();
    context.stroke();

}
```

```
function tri(x,y) {
    context.save();
    context.translate(x,y);
    context.beginPath();
    context.moveTo(0,0);
    context.lineTo(10,0);
    context.drawTo(0,20);
    context.closePath();
    context.fill();
    context.stroke();
    context.restore();
}
```

# Some key ideas are there...

- save and restore
- transform all points

- Object defined **locally**
- Doesn't know where it goes
- Could put translate outside
- Can re-use the object

```
function tri(x,y) {
    context.save();
    context.translate(x,y);
    context.beginPath();
    context.moveTo(0,0);
    context.lineTo(10,0);
    context.drawTo(0,20); lineTo
    context.closePath();
    context.fill();
    context.stroke();
    context.restore();
}
```

# Three Triangles, Three Transformations

```
context.fillStyle="red"
triangle(context,0,0);
context.translate(20.0,0.0);
context.fillStyle="blue"
triangle(context,0,0);
context.translate(0.0,20.0);
context.fillStyle="green"
triangle(context,0,0);
```

- apply transformations in the **current coordinate system**

(demo)

# Undo!

- Transformation applied in the **current coordinate system**
- Coordinate system is state
- How do we get back?

1. save/restore
2. Remember the transform and apply the opposite — inverse

```
context.translate(x,y)
// draw something
context.translate(-x,-y)
```

inverse

# Instancing

Create the graphics object

Use transformations to put it into the right place

Re-use, Re-use, Re-use

# Three trangles, Three Ways

```
tri(0,0);

tri(20,0);

tri(20,20);
```

```
tri();

context.translate(20,0);
tri();

context.translate(0,20);
tri();
```

```
tri();
context.save();
context.transtlate(20,0);
tri();
context.restore();
context.save()
context.translate(20,20);
tri();
context.restore()
```

Triangle assumes 0,0

Its own coordinates

**local** coordinates

# Composition of Transforms

$X_1 + X_2 + X_3 = X_{123}$

- Transformations "add up"
- Steps combine into one "bigger"

$$\boxed{T_1} + \boxed{T_2} = T_{12}$$

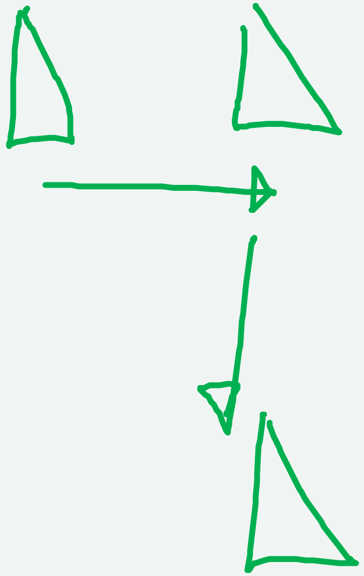
- Translations are easy (literally add)
- More generally, transformations **compose**

$$T_1 \circ T_2 = T_{12}$$

- $T_{12}$ is some new transformation (that combines $T_1$ and $T_2$)

# Why Compose?

- Use simple building blocks
  - translateX, translateY --> translate
- Build things up gradually
  - use intermediate results
  - easier to understand
- More important when combining transform types
  - when we have multiple types...



# What Types of Transformations?

Can be anything $(x', y') = \mathbf{f}(x, y)$

We'll start with a few basic types...

# What Types of Transformations?

- Basic ones
  - translate, rotate, scale
  - skew ← shear
- Why these?
- Linear transformations
  - mathematically nice (next time)
  - really useful!
- More over the course of the semester

# Why Compose?

- Use simple building blocks
  - translateX, translateY --> translate
- Build things up gradually
  - use intermediate results
  - easier to understand
- More important when combining transform types
  - when we have multiple types...

# What Types of Transformations?

- Why these?
- Linear transformations
  - mathematically nice (next time)
  - really useful!
- Basic ones
  - translate, rotate, scale
  - skew
  - projection (when we get to 3D)
- More over the course of the semester

# Scale Transformations

Make it bigger! (factor of 2)

```
function tri() {
    let s = 2;
    context.save();

    context.beginPath();
    context.moveTo(0  *s, 0  *s);
    context.lineTo(10 *s, 0  *s);
    context.drawTo(0  *s, 20 *s);
    context.closePath();
    context.fill();
    context.restore();
}
```

## Note:

- multiply all coordinates by s (=2)
- it will be twice as big
- zero stays zero (center)

center of scaling

# Scale Transformations

Make it bigger! (factor of 2)

```
function tri() {
    let s = 2;
    context.save();

    context.beginPath();
    context.moveTo(0  *s, 0  *s);
    context.lineTo(10 *s, 0  *s);
    context.drawTo(0  *s, 20 *s);
    context.closePath();
    context.fill();
    context.restore();
}
```

```
function tri() {
    let s = 2;
    context.save();
    context.scale(s,s); // note x,y
    context.beginPath();
    context.moveTo(0    , 0 );
    context.lineTo(10   , 0 );
    context.drawTo(0    , 20);
    context.closePath();
    context.fill();
    context.restore();
}
```

# Non uniform scale

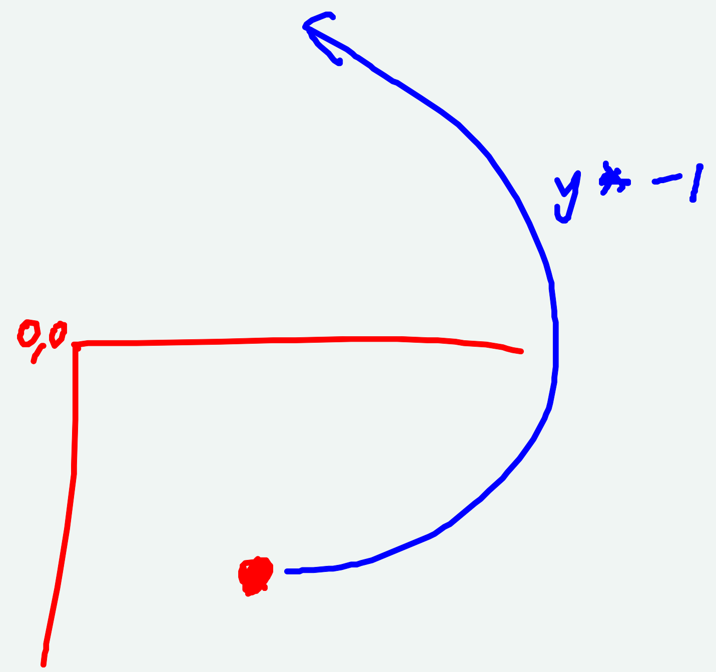
```
context.scale(sx,sy);
```

means:

```
context.moveTo(0  *sx, 0  *sy);
context.lineTo(10 *sx, 0  *sy);
context.drawTo(0  *sx, 20 *sy);
```

## Use for Stretching or Flipping

```
context.scale(1,-1);
```



# Scales compose by multiplication

```
scale(s1x,s1y)
scale(s2x,s2y)

context.moveTo(0  *s1x*s2x, 0  *s1y*s2y);
context.lineTo(10 *s1x*s2x, 0  *s1y*s2y);
context.drawTo(0  *s1x*s2x, 20 *s1y*s2y);
```

# Change of coordinates?

Yes, stretch (scale the coordinates)

Notice that **zero** stays in place

(demo)

# Scaling

- has a **center**
- everything else moves

- this is inconvenient

# Combining Scale and Translate

```
context.scale(sx,sy);
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo(X *sx, Y  *sy);
```

```
context.translate(tx,ty)
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo(X +tx, Y +ty);
```

# Scale and Translate

```
context.scale(sx,sy);
context.translate(tx,ty);
context.lineTo(X,Y);
```

$sx * (tx + X)$

means for all X,Y:

```
context.lineTo( sx* (X+tx), sy* (Y+ty) );
```

# Isn't this backwards?

# Scale and Translate

```
context.scale(sx,sy);

context.translate(tx,ty);

context.lineTo(x,y);
```

set up a **scaled** coordinate system

translate in the **scaled** coordinate system

draw in the **scaled** and **translated** csys

So...

context.lineTo( sx* (X+tx), sy* (Y+ty) );

# Order Matters!

```
context.scale(sx,sy);
context.translate(tx,ty);
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo( sx* (X+tx), sy* (Y+ty));
```

```
context.translate(tx,ty);
context.scale(sx,sy);
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo( tx+ (X*sx), ty+ (Y*sy) );
```

this needs a demo....

# Scale and Translate

```
context.scale(sx,sy);
context.translate(tx1,ty1);
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo( sx* (X+tx1), sy* (Y+ty1));
```

```
context.translate(tx2,ty2);
context.scale(sx2,sy2);
context.lineTo(X,Y);
```

means for all X,Y:

```
context.lineTo( tx2+ (X*sx), ty2+ (Y*sy) );
```

These translate a different amount

We could find `tx2` so it does the same thing as `tx1`

# Forwards and Backwards

Transformations move the coordinate system

Global to local

## Another way to think about it...

Work backwards from the object

The code is not written this way (backwards)

You cannot run your code **backwards**

(the demo program animates how to think about it)

# Center of Scaling

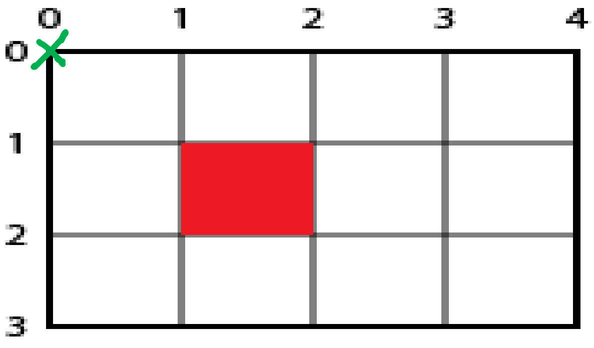
Choose what point doesn't change

Pick **some** point to be the center

[demo]

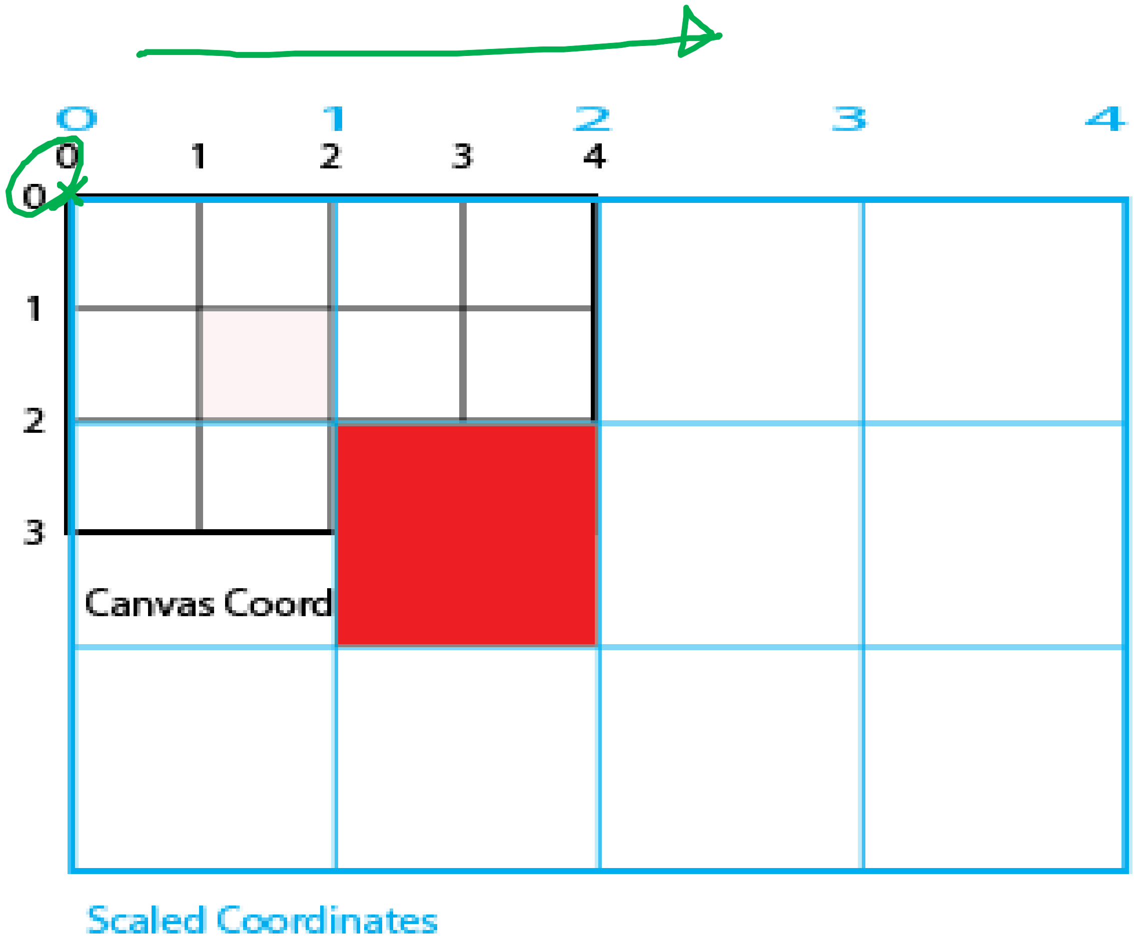
```
context.save();

context.translate(cx,cy);   1
context.scale(sx,sy);       2
context.translate(-cx,-cy); 3

context.restore();
```

# Scale about Center

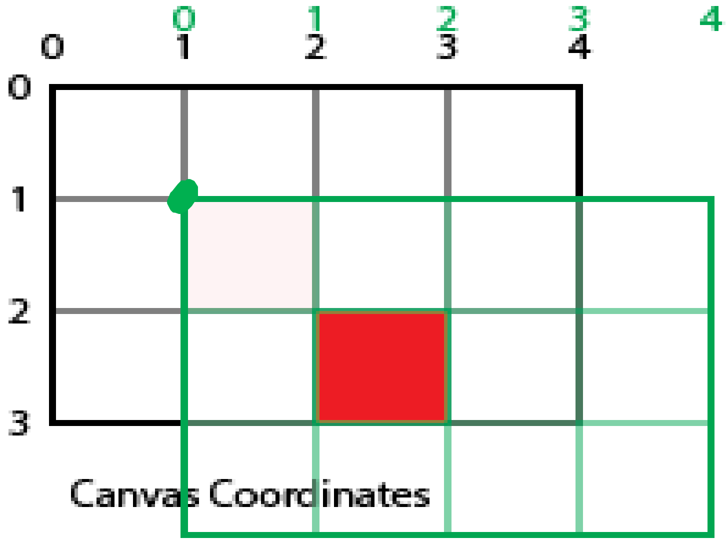



Canvas Coordinates

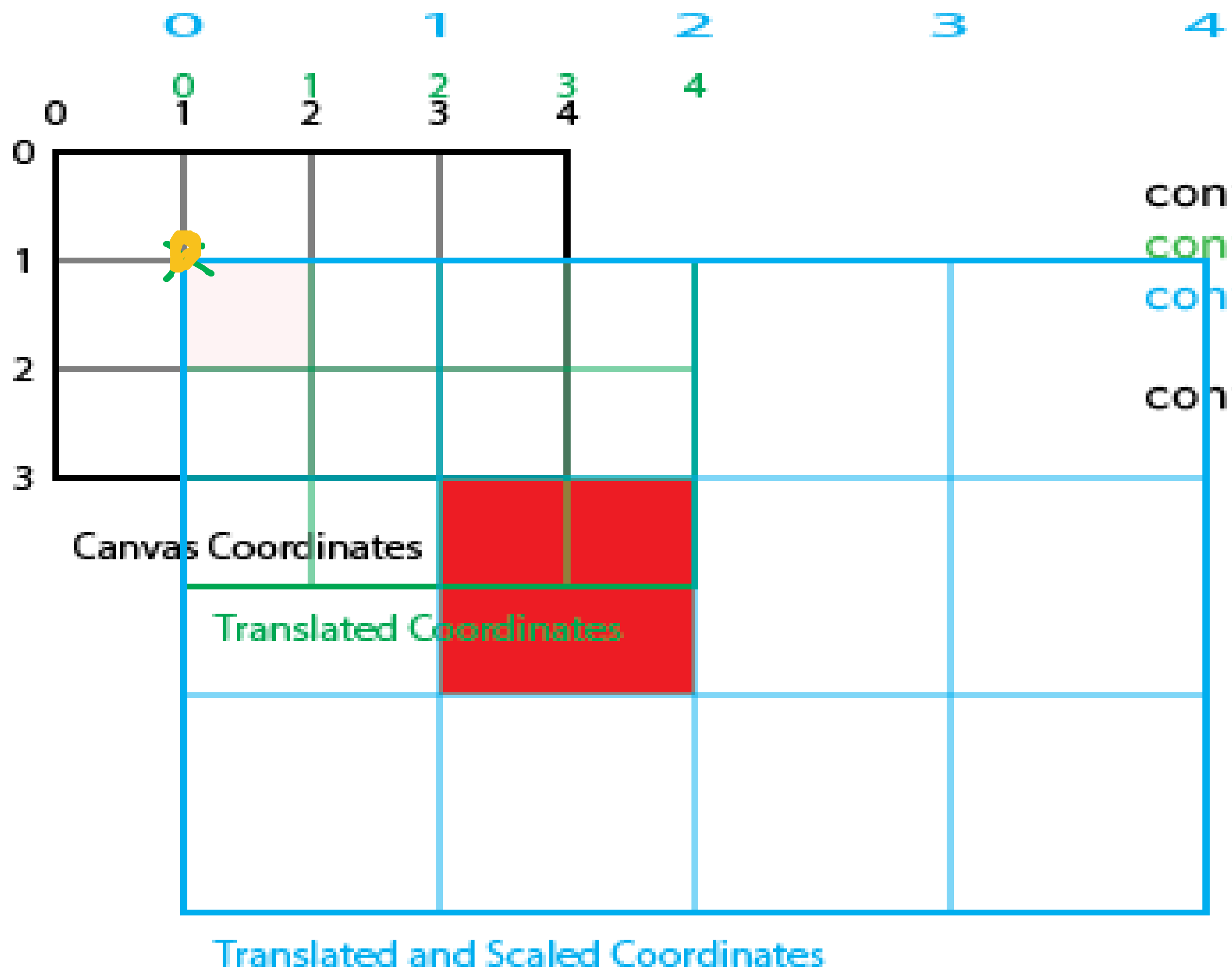
```
context.fillStyle = "red"

context.fillRect(1,1,1,1);
```



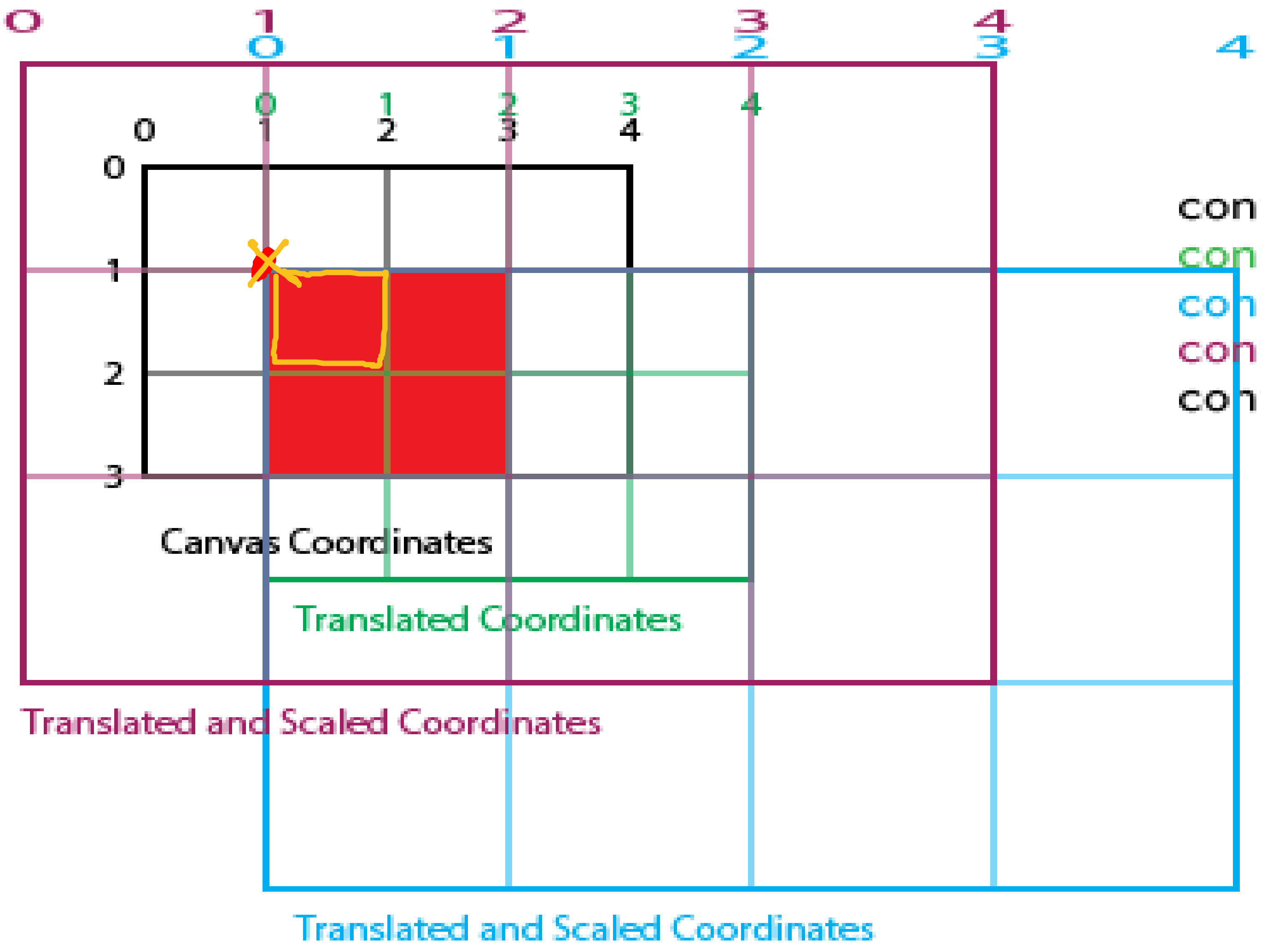

```
context.fillStyle = "red"

context.scale(2,2);

context.fillRect(1,1,1,1);
```



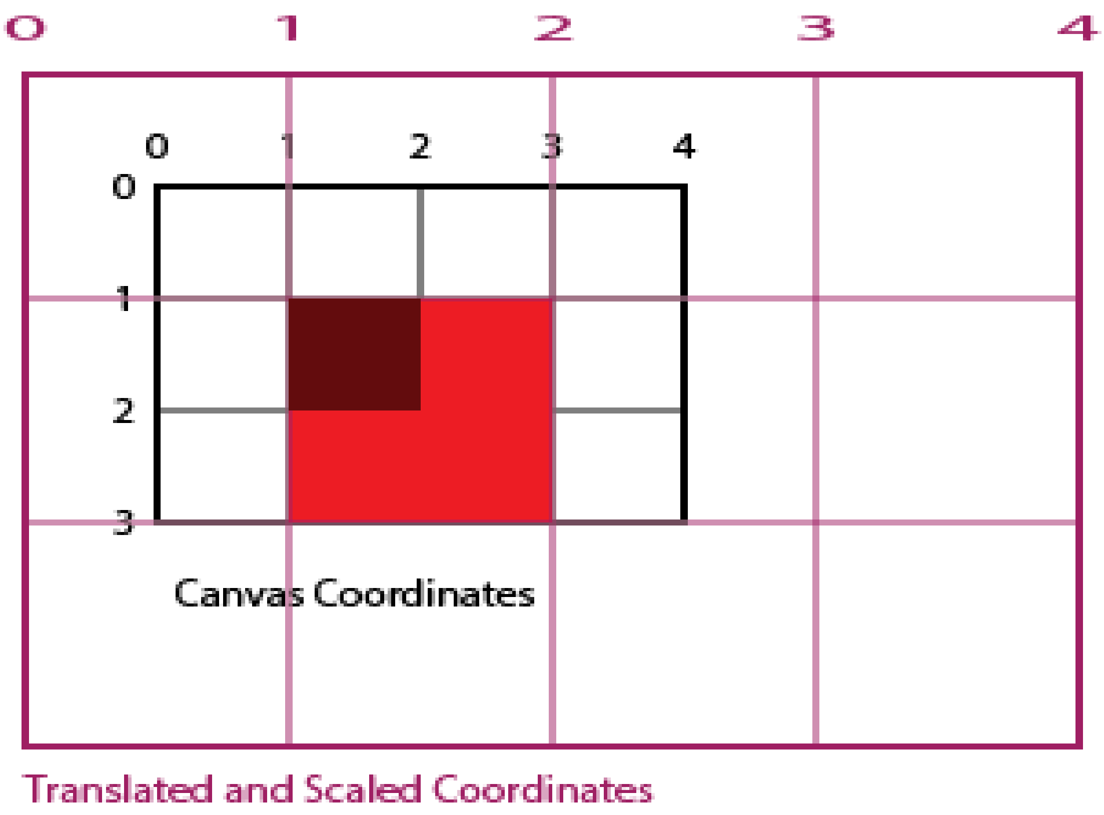

```
context.fillStyle = "red"
context.translate(1,1);

context.fillRect(1,1,1,1);
```




```
context.fillStyle = "red"
context.translate(1,1);
context.scale(2,2);

context.fillRect(1,1,1,1);
```




```
context.fillStyle = "red"
context.translate(1,1);
context.scale(2,2);
context.translate(-1,-1);
context.fillRect(1,1,1,1);
```




```
context.save();
context.fillStyle = “red”
context.translate(1,1);
context.scale(2,2);
context.translate(-1,-1);
context.fillRect(2,1,1,1);
context.restore();
```

```
context.fillStyle=“darkred”
```

```
context.fillRect(1,1,1,1);
```

# Did we have to ...

```
translate(cx,cy)
scale(s,s)
translate(-cx,-cy)
```

couldn't we just

```
scale(s,s)
translate(ty,ty)
```

for some ??,??

Yes, but... this is a general way to figure out those values

And it will work for everything else later too

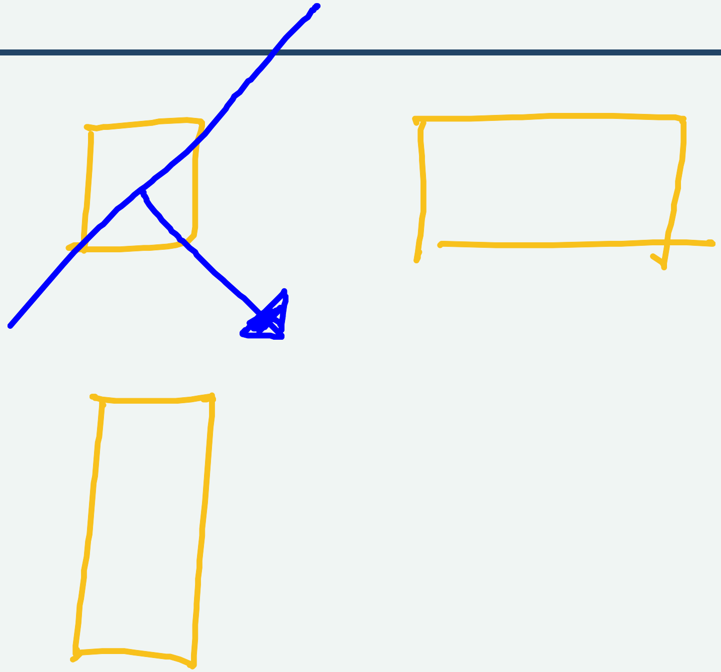
# Non-Uniform Scale

Scale differently in each direction

Scale in X, Scale in Y

Useful for "flipping" (make Y go up)

What about stretch in a different direction?



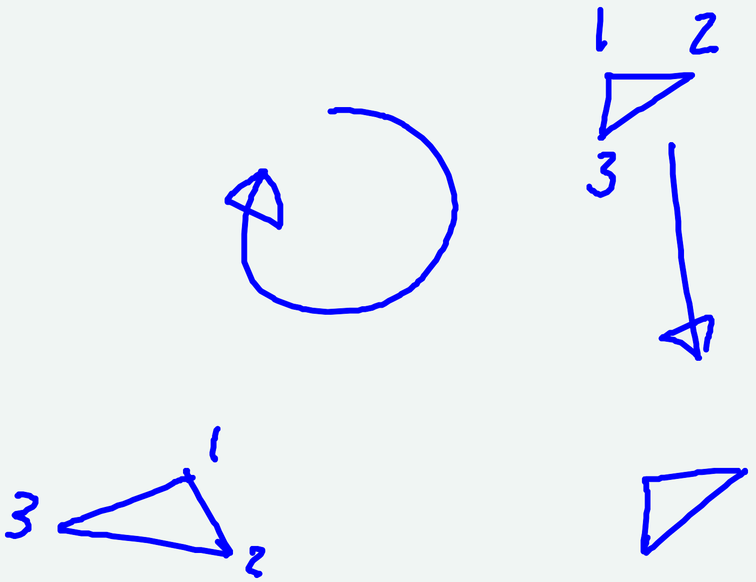
# Rigid Transformations

**Distances** between points are preserved

**Handedness** is preserved

Two types of transforms have these properties:

- translation (all points move the same)
- rotation (spin around a center point)



# Handedness

Curl your fingers from X to Y

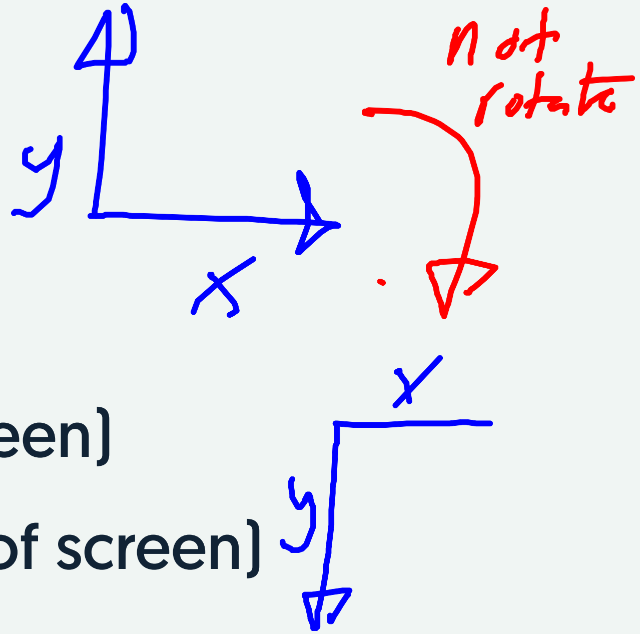
- right-handed vs. left-handed (will make sense in 3D)

Rotates around your thumb

- clockwise (Canvas coordinates, Y down, thumb into screen)
- counter-clockwise (math coordinates, Y up, thumb out of screen)

Handedness - preserves the direction of your thumb

Mirror reflection does not preserve handedness (it is not rigid)



# Rotations

(in 3D, this is more complicated)

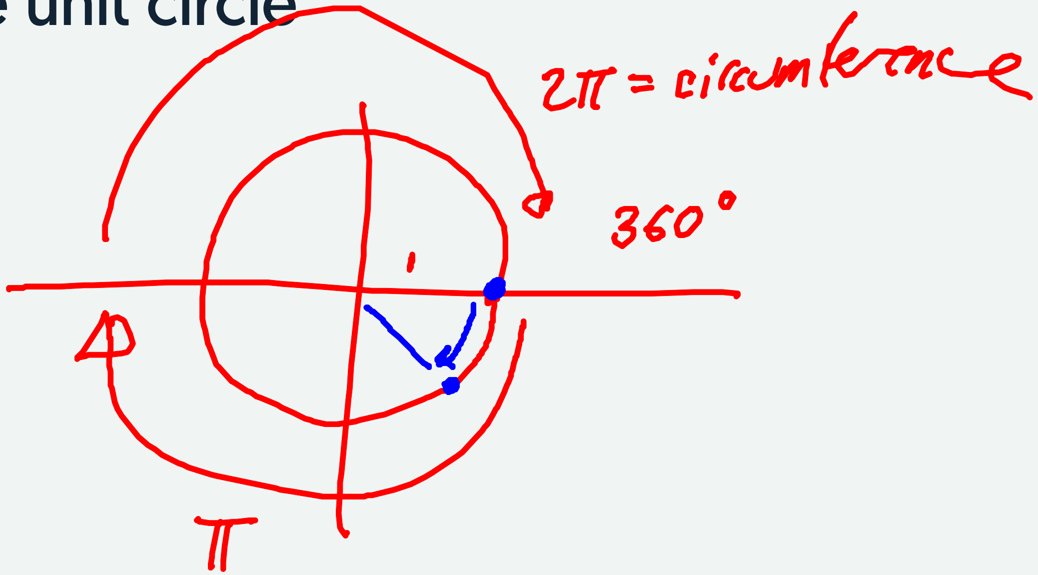
Spin the coordinate system around a point

Center - the point that doesn't move

# Measuring rotations (in 2D)

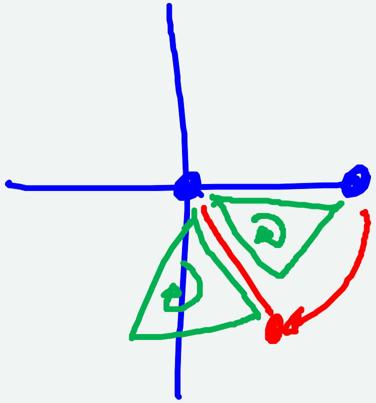
radians = amount a point moves on the unit circle

(convert to degrees - if that's easier)



# Properties of rotations

- There is a **zero** (point that doesn't move)
- Distances are preserved
- Handedness is preserved (not mirror reflection)



# Rotated Coordinate Systems

Axes point in new directions

# Convention

We rotate around the center of the coordinate system

We change change the center

- just like we did with scale

# Demos

# Translate then rotate

# Rotate then translate

Rotation is about the **center** of rotation

# Useful idiom

Rotate around a center point

- translate the origin to **some point**
- rotate about the origin
- translate the origin back from **some point**
- draw the object

# rotate around a point

```
context.translate(cx,cy);
context.rotate(angle);
context.translate(-cx,-cy);
drawThing();
```

- make the object
- move the center point to origin
- rotate around the origin
- move the center point back
- **this is reading backwards**

# Summary: Transformations

- Think of transformation as changing the **coordinate system**

- translate
- rotate
- scale

- compose transformations to do useful things
- more useful things next time
- math soon too!