Lecture 4:
More 2D Graphics

Reminders / Review

Last time:

- Graphics 101: Draw and Redraw!
- Canvas 101: An HTML element we draw to

Some Questions That Come Up

- Isn't there another way to do (insert something)?
  - yes

Agenda

- ~~more about canvas basics~~
- how are graphics **ideas** in Canvas
- coordinate systems
- transformations (maybe not)

Things to notice about Canvas

Canvas is the **element**

Context is the **API**

Need to clear frame

Coordinate System

Measurement Units

Stateful Drawing

```
let canvas = document.getElementById("myc");
let context = canvas.getContext("2d");



context.clearRect(0,0, canvas.width, canvas.height);



context.fillRect(20,20, 40, 80);
context.fillStyle = "red";
context.fillRect (40,60,40,80);
```

The Three Big Questions...

- When do I draw?
  - when it's your turn
- Where do I draw?
  - in the current coordinate system
- What do I draw?
  - primitives, with styles

What do I draw: Rectangles

```
context.fillStyle = "red";
context.fillRect (40,60,40,80);
```

Rectangles are: x,y,w,h - w,h are sizes, not positions

Separate commands for **stroke** and **fill**

Styles as **state**

- fill color, stroke color
- fill patterns
- stroke patterns (dashed), line width, ...

Save and Restore

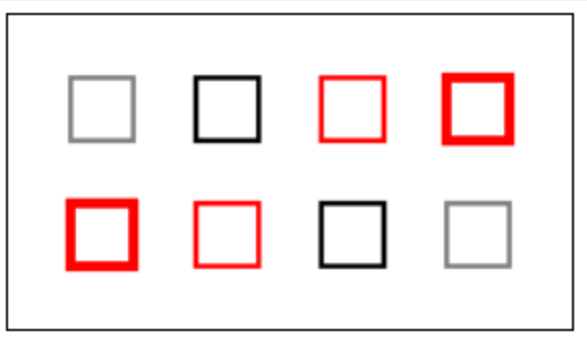
```
context.save();
context.fillStyle="red";
context.fillRect(40,40,20,20);
context.restore();
context.fillRect(50,50,20,20);
```

`save` and `restore` capture most (all?) context information

Save and restore is a stack

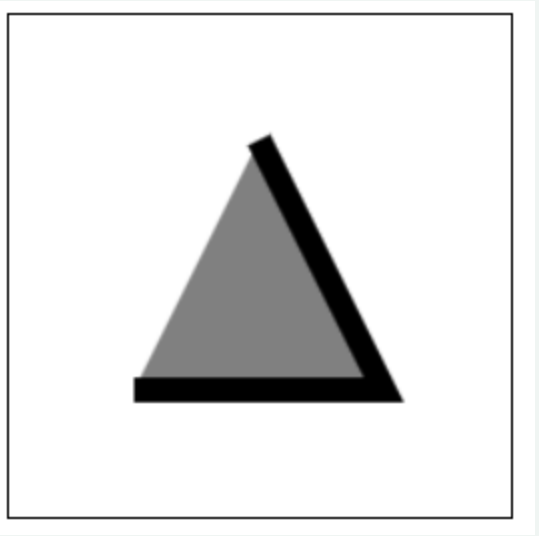
```
context.strokeStyle = "black";
context.strokeRect(25,25,25,25);
context.save();
context.lineWidth = 2;
context.strokeRect(75,25,25,25);
context.save();
context.strokeStyle = "red";
context.strokeRect(125,25,25,25);
context.save();
context.lineWidth = 4;
context.strokeRect(175,25,25,25);


context.strokeRect(25,75,25,25);
context.restore();
context.strokeRect(75,75,25,25);
context.restore();
context.strokeRect(125,75,25,25);
context.restore();
context.strokeRect(175,75,25,25);
```



Beyond Rectangles: Paths

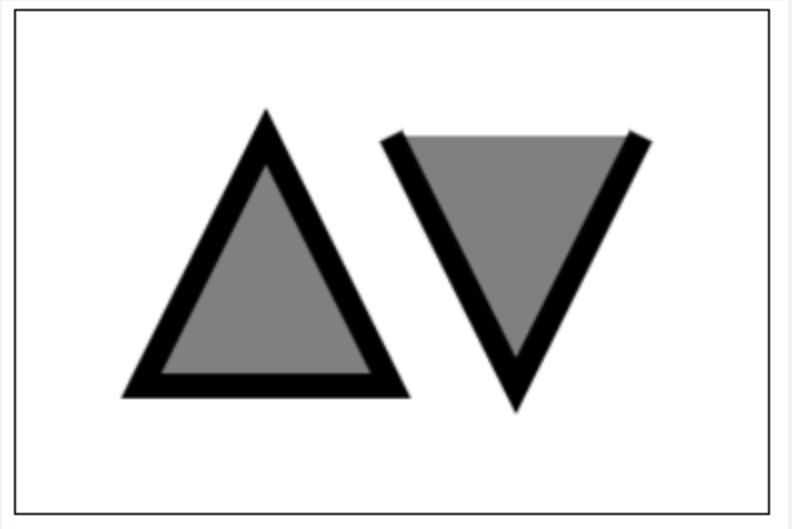
```
context.beginPath();
context.moveTo(x,y);
context.lineTo(x2,y2);
context.lineTo(x3,y3);
context.fill();
context.stroke();
```



- beginPath
- moveTo
- lineTo
- fill
- stroke

Open, Closed, Disconnected ...

```
context.beginPath();
context.moveTo(100,100);
context.lineTo(110,120);
context.lineTo(120,100);
context.closePath();
context.moveTo(150,100);
context.lineTo(160,120);
context.lineTo(170,100);
context.fill();
context.stroke();
```



The Pen Model

Methods use the **current pen position**

Methods add to the **current path**

- `moveTo`
- `lineTo`
- `closepath`

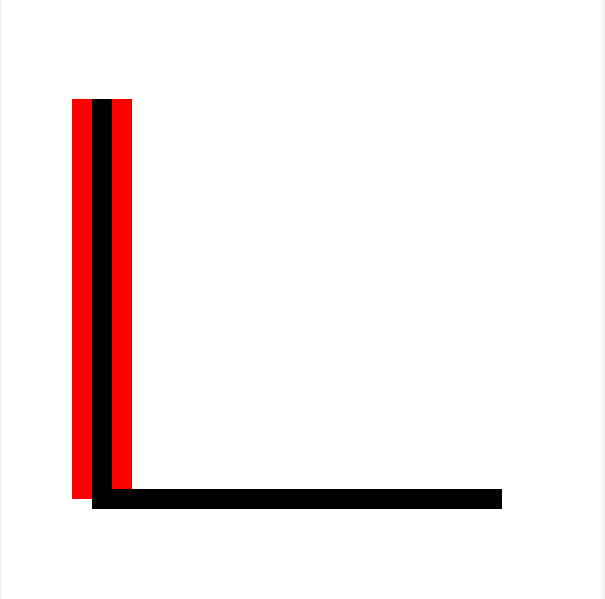
- `arc`, `arcTo`, `curveTo`, …

Stroke/Fill the entire path!

The entire path is redrawn with the current pen!

```
context.beginPath();
context.strokeStyle = "red";
context.lineWidth = 12;
context.moveTo(20,20);
context.lineTo(20,100);
context.stroke();

context.strokeStyle = "black";
context.lineWidth = 4;
context.lineTo(100,100);
context.stroke();
```



Other Shapes

More Path Operators

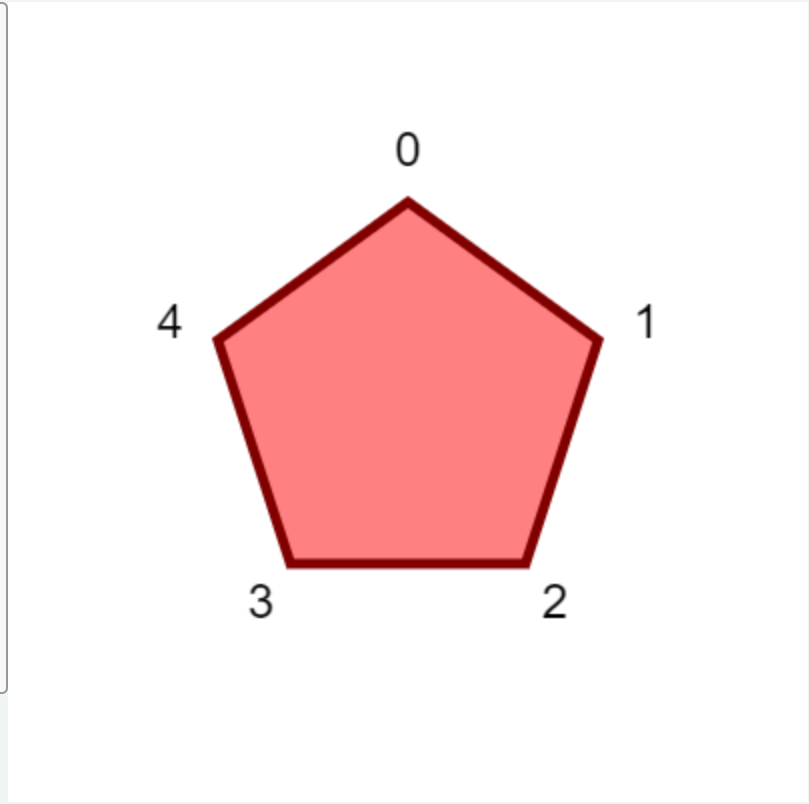
- arcs (circles) `arc` vs. `arcTo`
- curves (Bézier - wait a few weeks)

Polygon filling rules

- non-convex shapes
- non-simple (crossings)
- disconnected (holes)

Convex

```
context.beginPath();
context.closePath();
context.moveTo(...pent[0]);
context.lineTo(...pent[1]);
context.lineTo(...pent[2]);
context.lineTo(...pent[3]);
context.lineTo(...pent[4]);
context.closePath();
context.fill();
context.stroke();
```



JavaScript Tip of the Day:
Spread Syntax

`pent[0]` is an array `[200,100]`

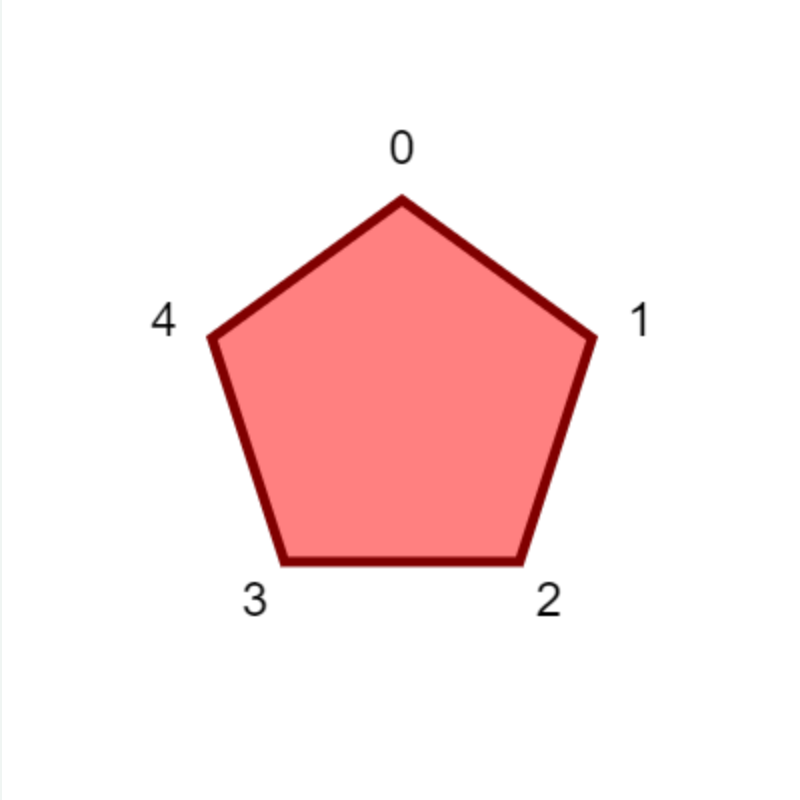
`context.moveTo()` takes 2 parameters x, and y

`context.moveTo(pent[0][0],pent[0][1])` is clunky

`context.moveTo(...pent[0])` uses the **spread operator**

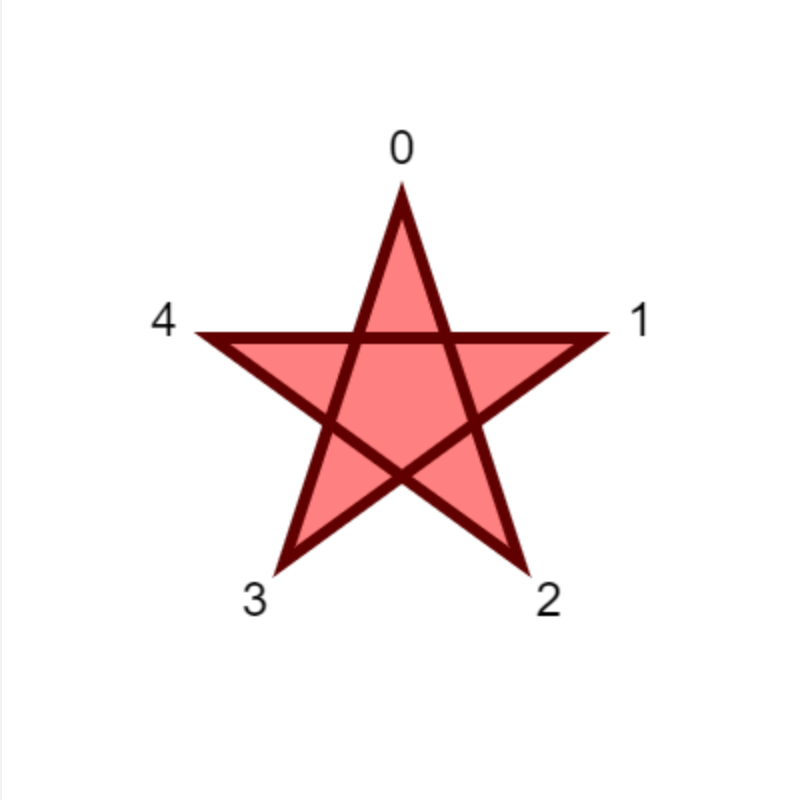
Convex

```
context.beginPath();
context.closePath();
context.moveTo(...pent[0]);
context.lineTo(...pent[1]);
context.lineTo(...pent[2]);
context.lineTo(...pent[3]);
context.lineTo(...pent[4]);
context.closePath();
context.fill();
context.stroke();
```

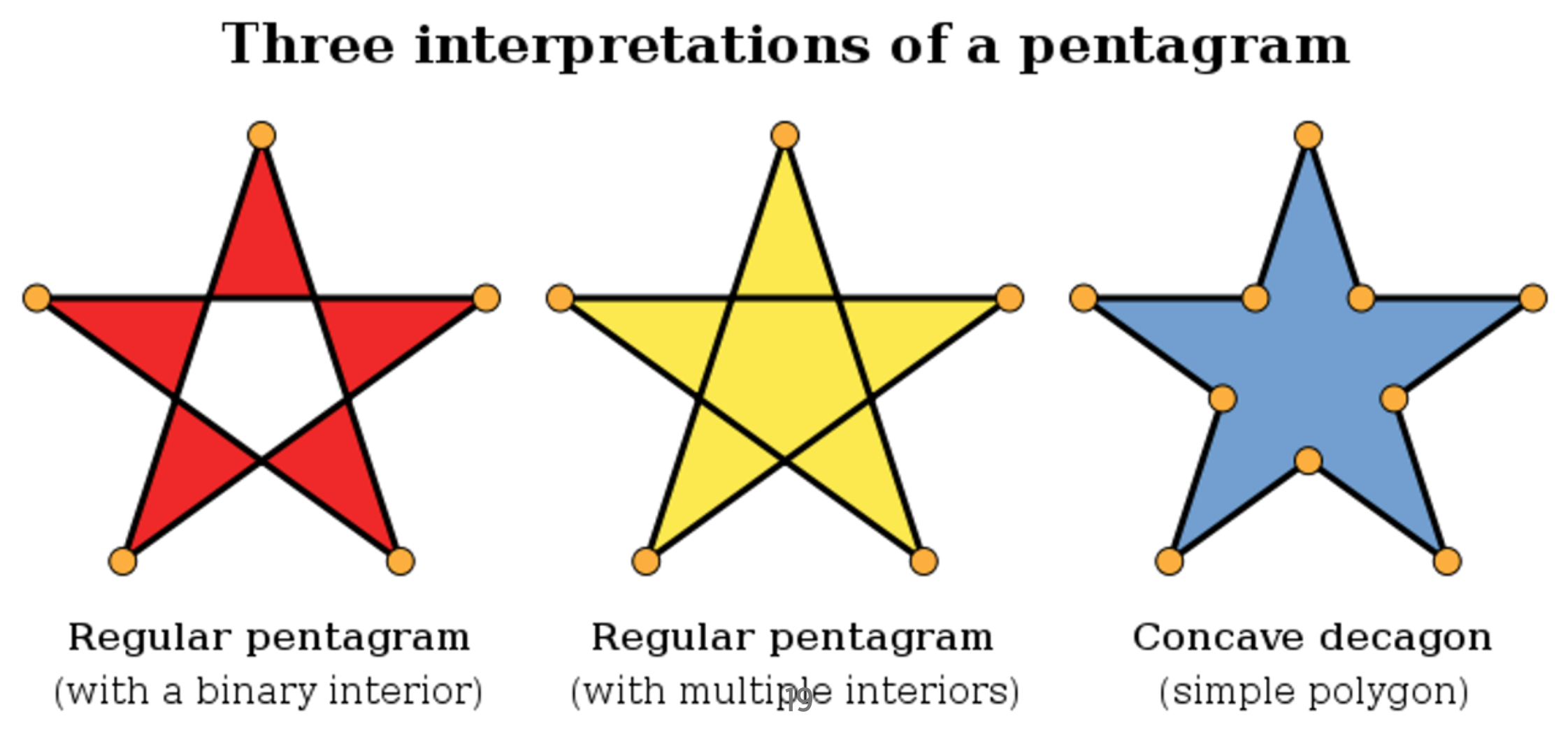


Re-order vertices (lines cross)

```
context.beginPath();
context.closePath();
context.moveTo(...pent[0]);
context.lineTo(...pent[2]);
context.lineTo(...pent[4]);
context.lineTo(...pent[1]);
context.lineTo(...pent[3]);
context.closePath();
context.fill();
context.stroke();
```



5 sides vs. 10 sides?



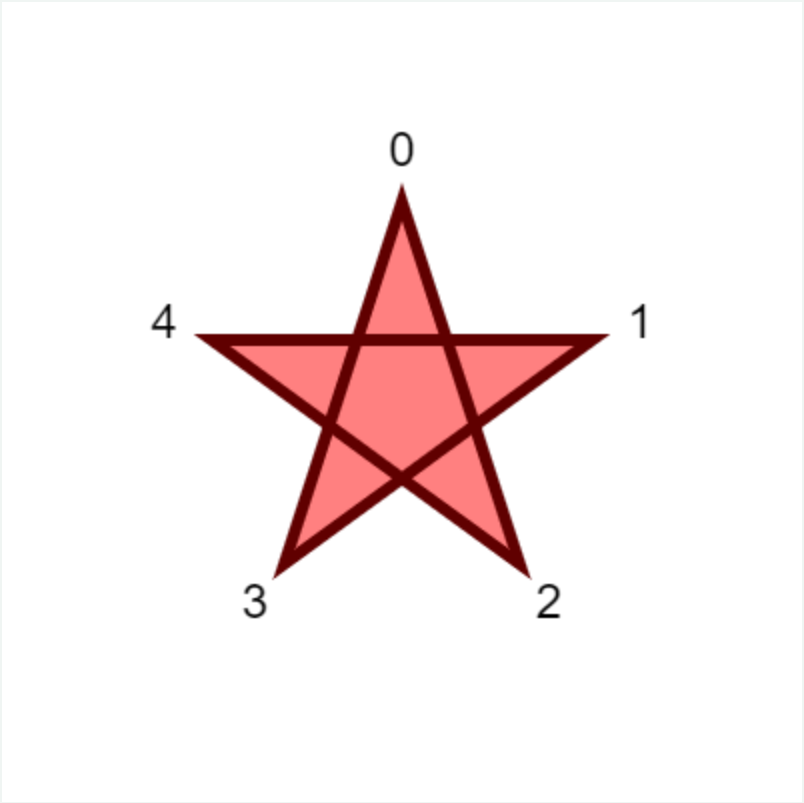
Non-Simple Polygons

- Edges Cross
- Edges are disconnected (multiple loops)
- Not simple to define inside and outside
- We'll use different rules

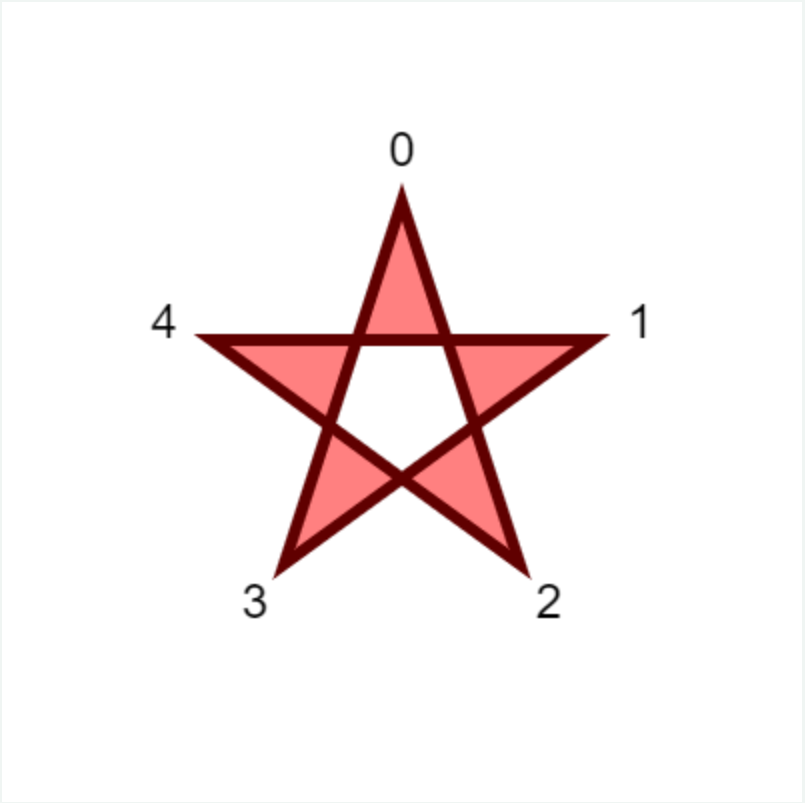
- Canvas lets you make non-simple polygons
- Canvas gives you different rules to interpret them

Two Different Rules

Non-Zero Winding



Even-Odd



Even / Odd

```
context.fill("evenodd");
```

Pick any point

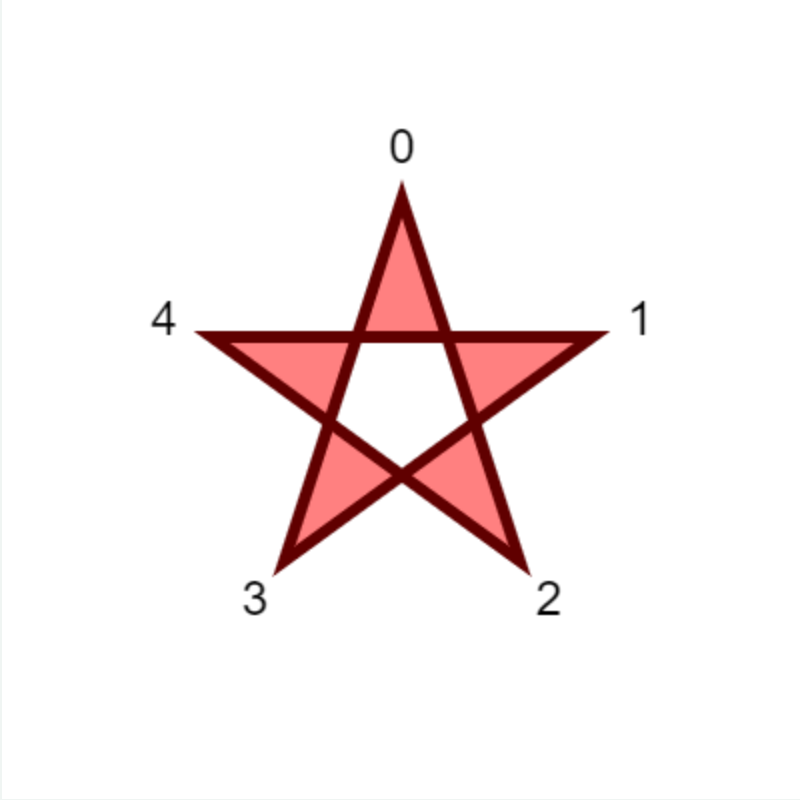
Go to infinity in any direction

Count the number of crossings

Even (includes 0) = outside

Odd = inside

Even-Odd



Winding (non-zero)

```
context.fill();
```

Count the "loops" around a point

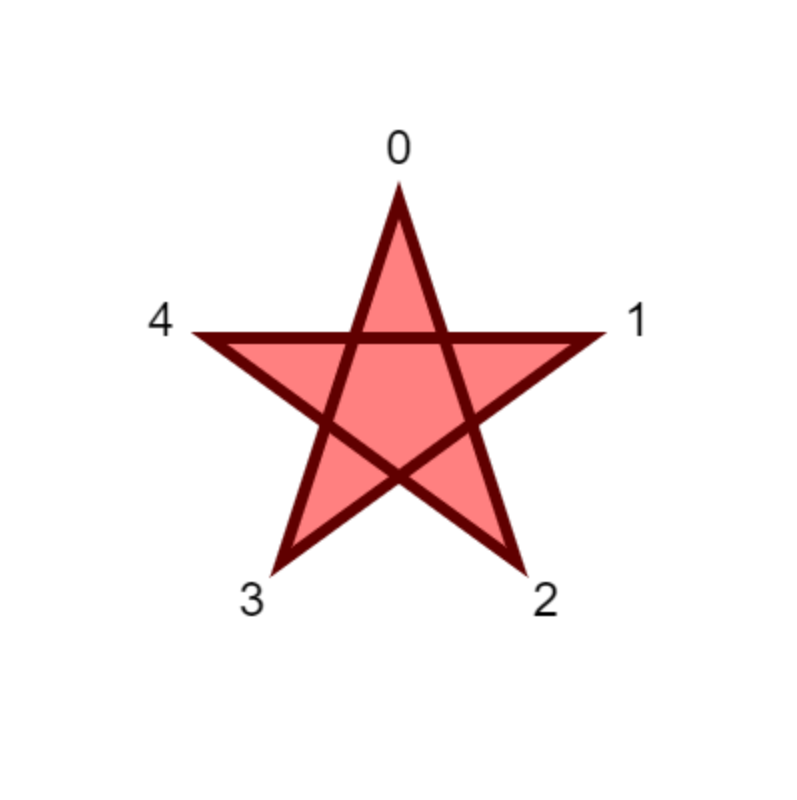
+1 for clockwise

-1 for counter-clockwise

order matters

inside if total is not zero

(inside if odd - Adobe, not Canvas)



Why use winding rules?

```
context.beginPath();     // clockwise
context.moveTo(100,100);
context.lineTo(300,100);
context.lineTo(200,275);
context.closePath();

context.moveTo(150,130); // counter
context.lineTo(200,225);
context.lineTo(250,130);
context.closePath();

context.fill();
context.stroke();
```

Use direction to control insides

```
context.beginPath();
context.moveTo(100,100); // clockwise
context.lineTo(300,100);
context.lineTo(200,275);
context.closePath();

context.moveTo(150,130); // clockwise
context.lineTo(250,130);
context.lineTo(200,225);
context.closePath();

context.fill();
context.stroke();
```

Even Odd is Easier (?)

```
context.beginPath();
context.moveTo(100,100); // clockwise
context.lineTo(300,100);
context.lineTo(200,275);
context.closePath();

context.moveTo(150,130); // clockwise
context.lineTo(250,130);
context.lineTo(200,225);
context.closePath();

context.fill("evenodd");
context.stroke();
```

In Practice...

Non-Simple Polygons are rare

Most APIs only give you simple polygons

OpenGL only gives you **triangles**

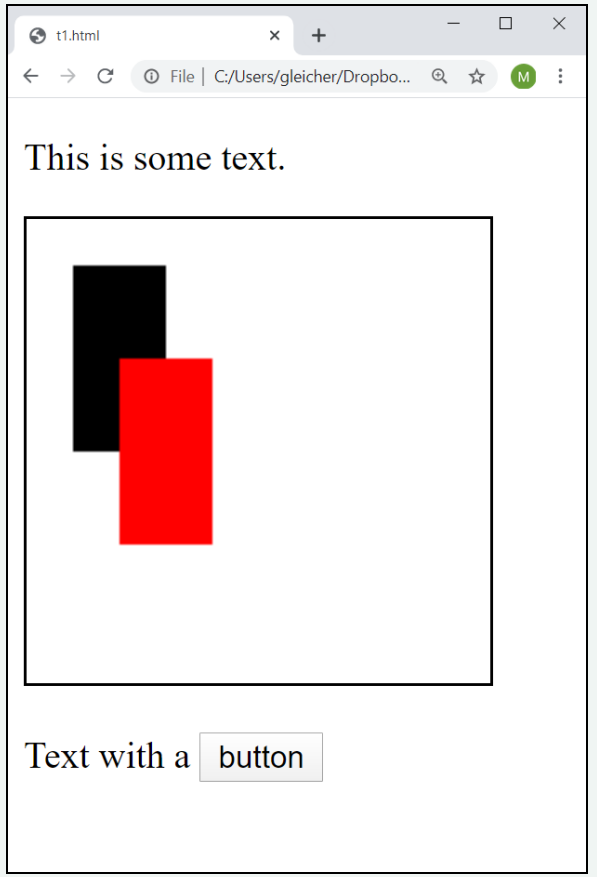
A less esoteric point...

What do the vertex positions mean?

Where do I draw?

Points (x,y) in the **current coordinate system**

```
context.fillRect(20,20, 40, 80);
context.fillStyle = "red";
context.fillRect (40,60,40,80);
```



Default coordinates:

- origin at top left (of canvas)
- x to the right in "html pixels"
- y down in "html pixels"

Convenient (for the Canvas)

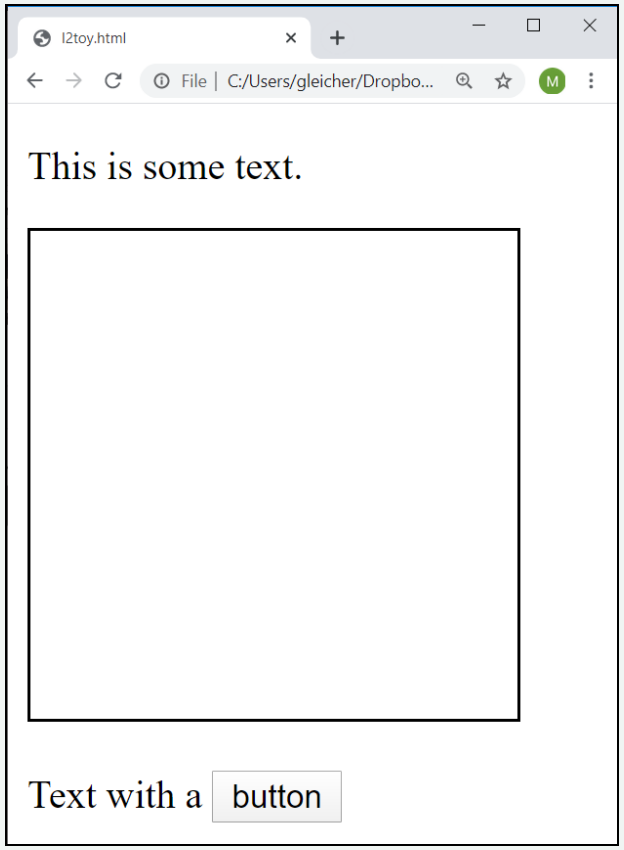
Handling Events

The **canvas** is the HTML element

The **canvas** receives events

- mouse enter / leave
- mouse move (inside)
- click

Other Coodinates?



Mouse position is in "client" coordinates

```
let box = event.target.getBoundingClientRect();
let x = event.clientX - box.left;
let y = event.clientY - box.top;
```

Need to convert from window to Canvas

It is **convenient** to draw in Canvas Coordinates

Where is the mouse?

```
window.onload = function() {
    let canvas = document.getElementById("myc");
    let context = canvas.getContext("2d");

    canvas.onmousemove = function(event) {
        let box = event.target.getBoundingClientRect();
        let x = event.clientX - box.left;
        let y = event.clientY - box.top;
        context.fillStyle = "#80800080";
        context.fillRect(x-5, y-5,10,10);
    }
    canvas.onclick = function() {
        context.clearRect(0,0,canvas.height,canvas.width);
    }
}
```

Canvas "Events"

Only the “canvas” is an HTML element

Only the “canvas” gets events

The graphics are represented in code

There is no object to get an event

Immediate mode: primtives "immediately" turned to pixels

Click in a rectangle

```
canvas.fillRect(20,20, 60,60);

canvas.onclick = function(event) {
    let mouseX = getXposition(event);
    let mouseY = getYposition(event);
    // check if event is inside of rectangle
    if ( (x>=20) and (x<=(20+60) ) and (y>=20) and (y<=(20+60))) {
        console.log("rectangle was clicked")
    }
}
```

Warning: the event must be converted to canvas coordinates!

Remember the rectangle?

```
rects = [];

canvas.fillRect(20,20, 60,60);
rects.push( { x:20, y:20, w:60, h:60} );
```

In immediate mode, the shapes are in the code - not data structures.

If you want to remember them, you need to make your own data structures.

Coordinate System

You need to know how to interpret coordinates!

- Where is the origin?
- How do I interpret the X Axis?
- How do I interpret the Y Axis
- (in 3D, we will have a 3rd axis)

We'll come back to this

Drawing More Interesting Things

Motivating Coordinate Systems

One thing inside another

```
context.fillStyle="goldenrod";
context.fillRect(10,10,50,30);
context.fillStyle="red";
context.fillRect(20,20,10,10);
context.fillRect(40,20,10,10);
```

change where this "object" is?

```
context.fillStyle="goldenrod";
context.fillRect(60,10,50,30); // changed this
context.fillStyle="red";
context.fillRect(20,20,10,10);
context.fillRect(40,20,10,10);
```

Oops!

move everything

```
    context.fillStyle="goldenrod";
    context.fillRect(50+10,10,50,30);
    context.fillStyle="red";
    context.fillRect(50+20,20,10,10);
    context.fillRect(50+40,20,10,10);
```

rect is weird since `width,height` is relative

better with a variable

```
let x=50;
context.fillStyle="goldenrod";
context.fillRect(x+10,10,50,30);
context.fillStyle="red";
context.fillRect(x+20,20,10,10);
context.fillRect(x+40,20,10,10);
```

make the variables mean something

```
let x=60;
let y=10;
context.fillStyle="goldenrod";
context.fillRect(x,y,50,30);
context.fillStyle="red";
context.fillRect(x+10,y+10,10,10);
context.fillRect(x+30,y+10,10,10);
```

The new piece

```
context.translate(x,y)
```

1. Move all future drawing points by x,y
2. Move the **coordinate system** by x,y

For translation, there isn't much difference

Demo

move the coordinate system!

```
let x=60;
let y=10;
context.translate(x,y);

context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,30);
context.fillStyle="red";
context.fillRect(10,10,10,10);
context.fillRect(30,10,10,10);
```

don't forget to put things back

```
let x=60;
let y=10;
context.save();
context.translate(x,y);
context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,30);
context.fillStyle="red";
context.fillRect(10,10,10,10);
context.fillRect(30,10,10,10);
context.restore();
```

move objects, or coordinates?

```
context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,50);
context.fillStyle="red";
context.save();
    context.translate(10,10);
    context.fillRect(0,0,10,10);
    context.translate(20,0);
    context.fillRect(0,0,10,10);
context.restore();
context.translate(0,20);
context.save();
    context.translate(10,10);
    context.fillRect(0,0,10,10);
    context.translate(20,0);
    context.fillRect(0,0,10,10);
context.restore();
```

Instancing

```
    context.fillRect(0,0,10,10);
```

Same thing, used over and over...

make it once and put it into place

Key Ideas

- transformations apply to all points
- view transformations as: moving objects
- view transformations as: moving coordinate systems
- transformations **compose**
- use transformations to get convenient coordinates
- use transformations to build hierarchy

Program forward or backwards?

Forwards: Move the Coordinate System

- move the coordinate system
- draw the object in the current coordinate system

Backwards: Move the Object

- create the object (not really)
- move the object into place

For Translation, order doesn't matter much...

Transformations!

Transform objects or change coordinate systems

Use different transformations to get convenient coordinates

Combine transformations to get more complex behvaiors

Implement transformations with linear algebra

(yes - there will be math next week)