

# Vector Graphics

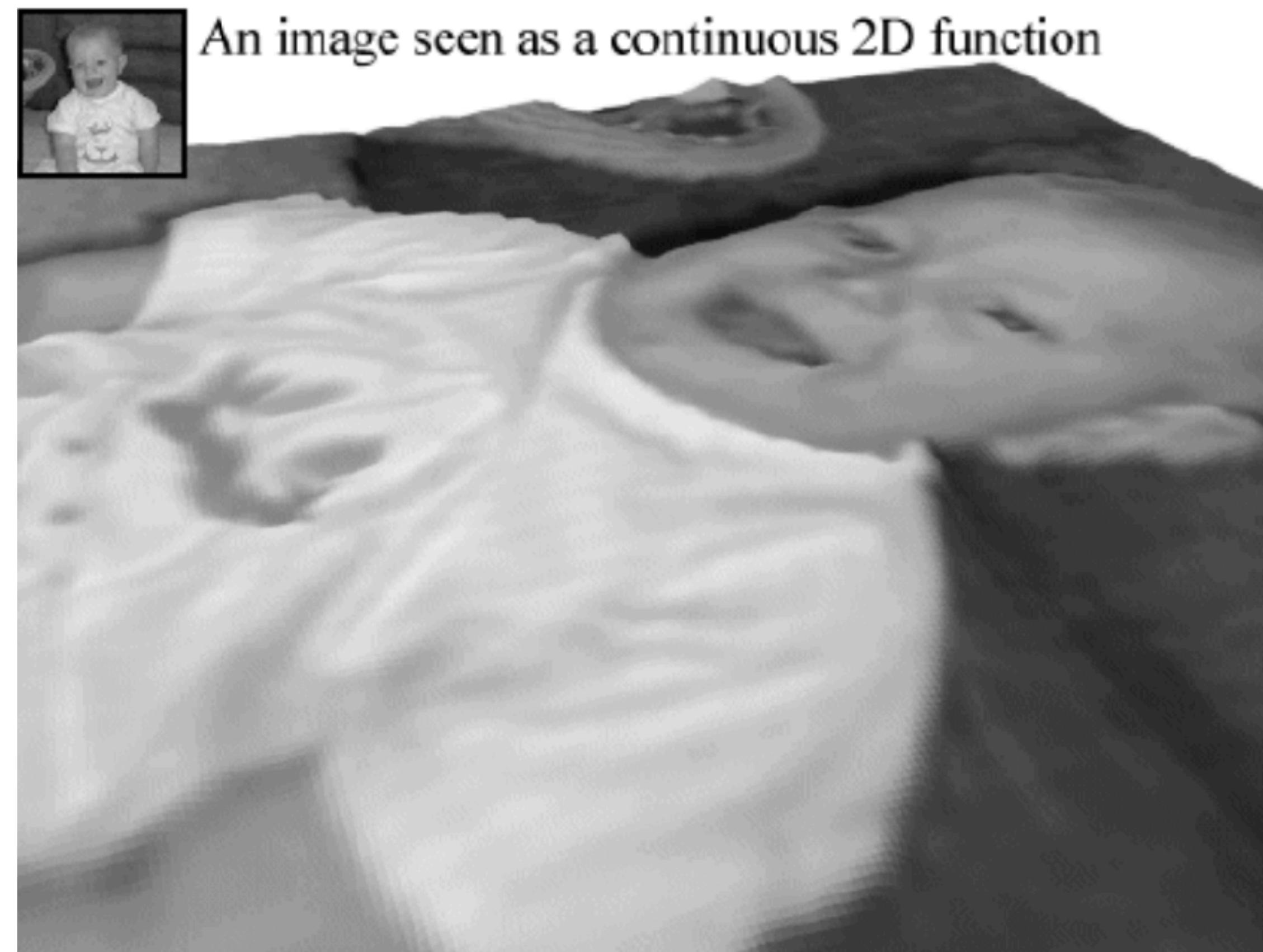
UCSD CSE 167

Tzu-Mao Li

Q: What are images?

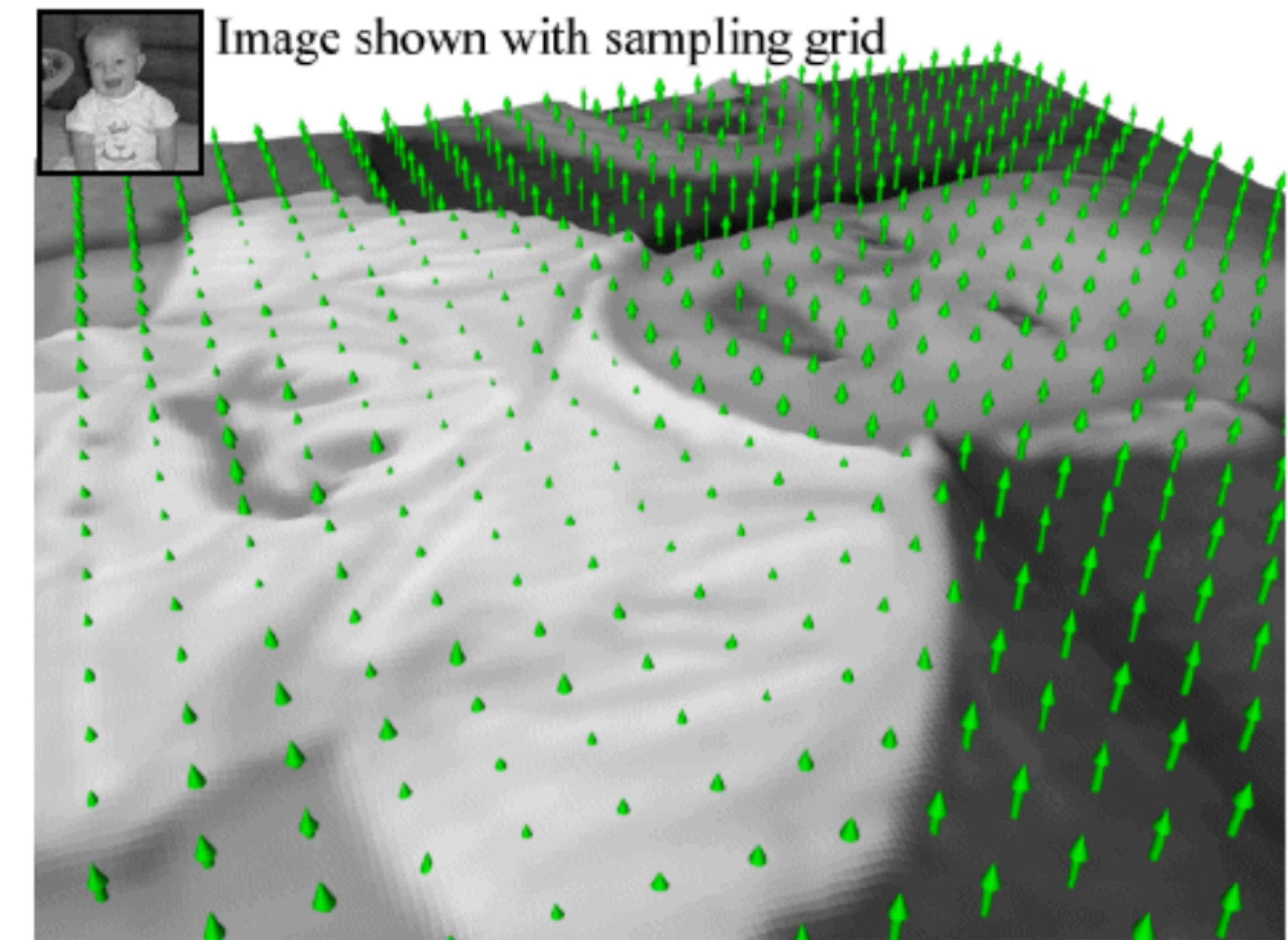
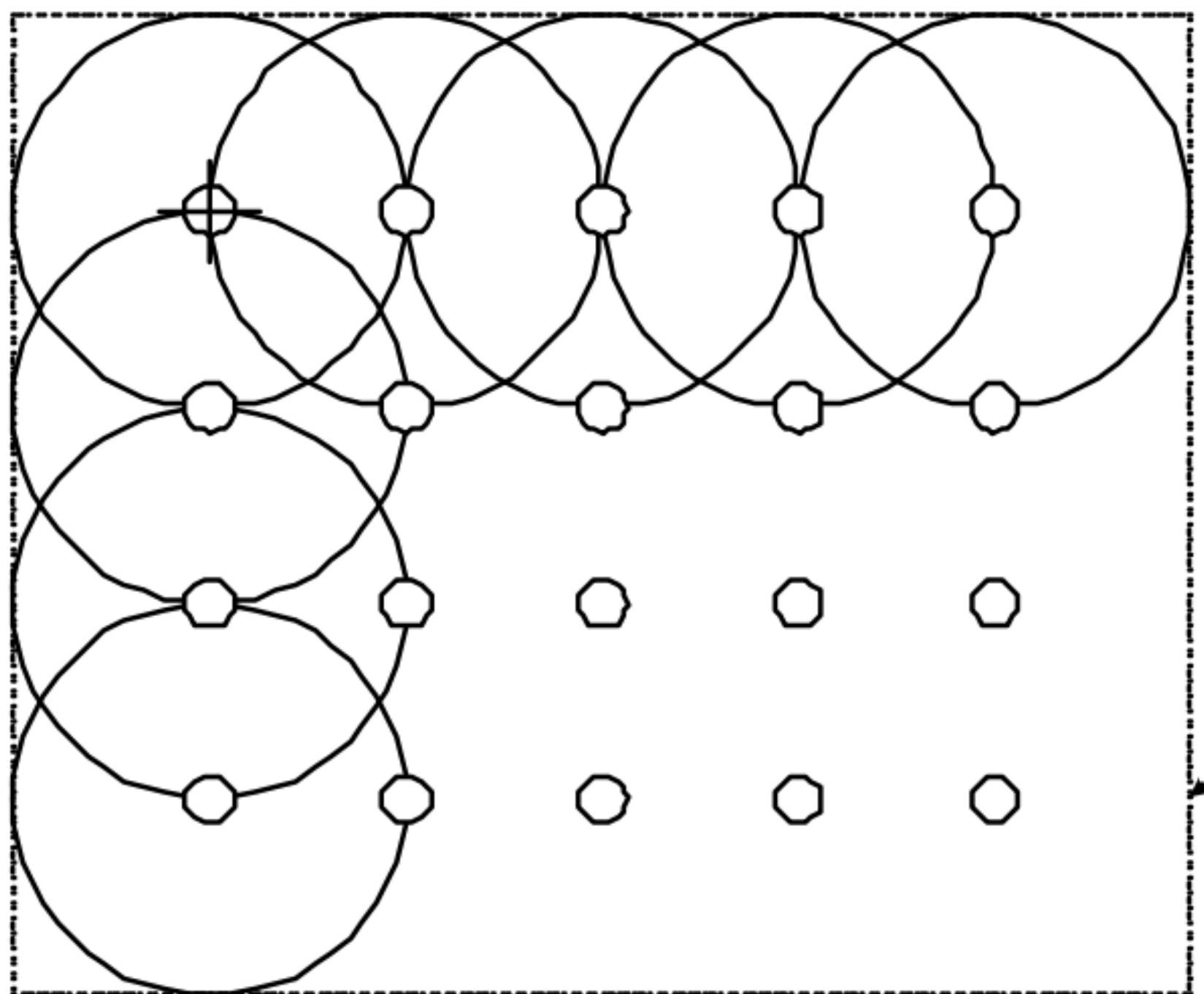
# An image is a 2D function

- an *image* is a function  $I(x, y)$  of colors with  $x, y \in \mathbb{R}$

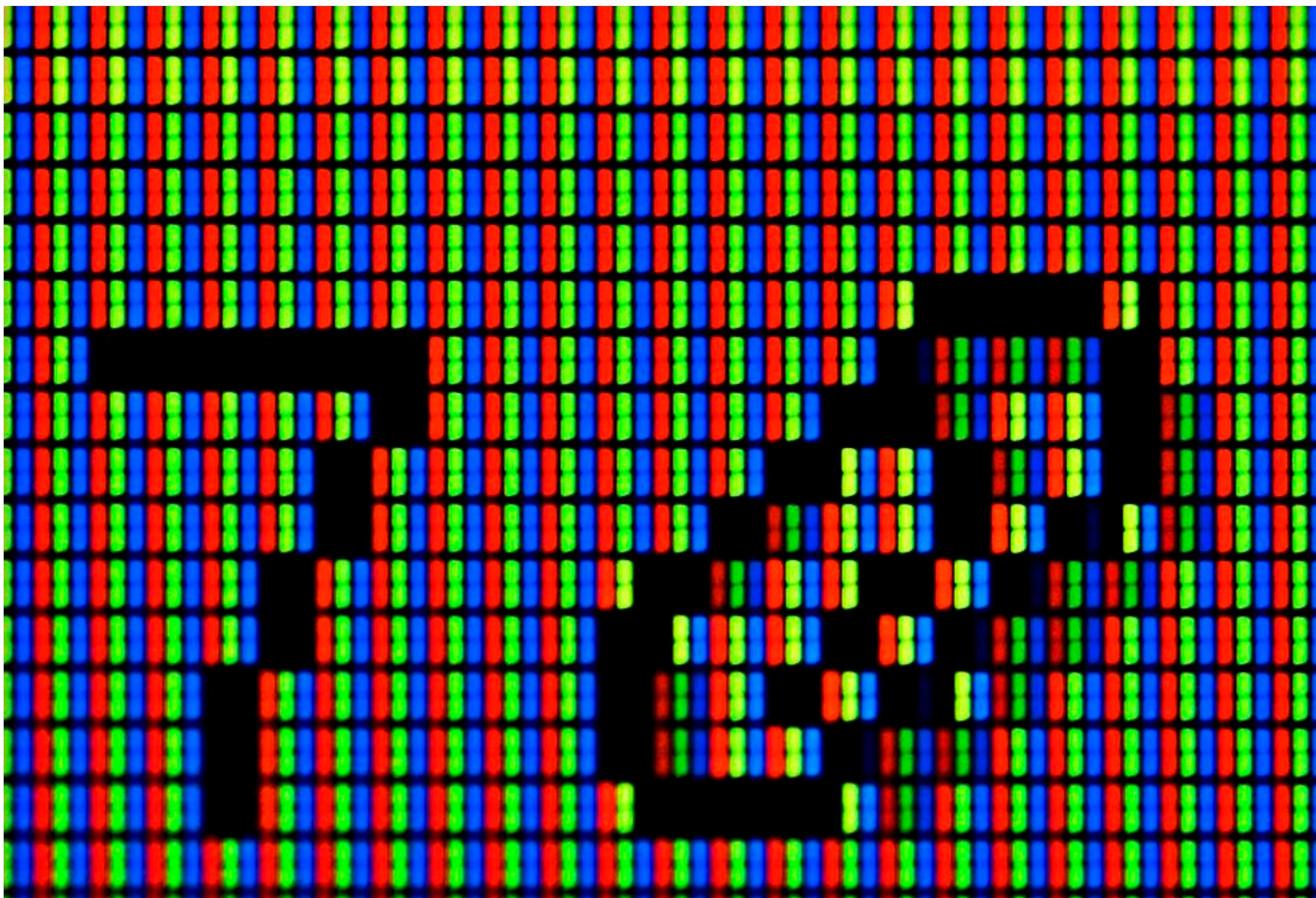


# A popular representation of image: raster images

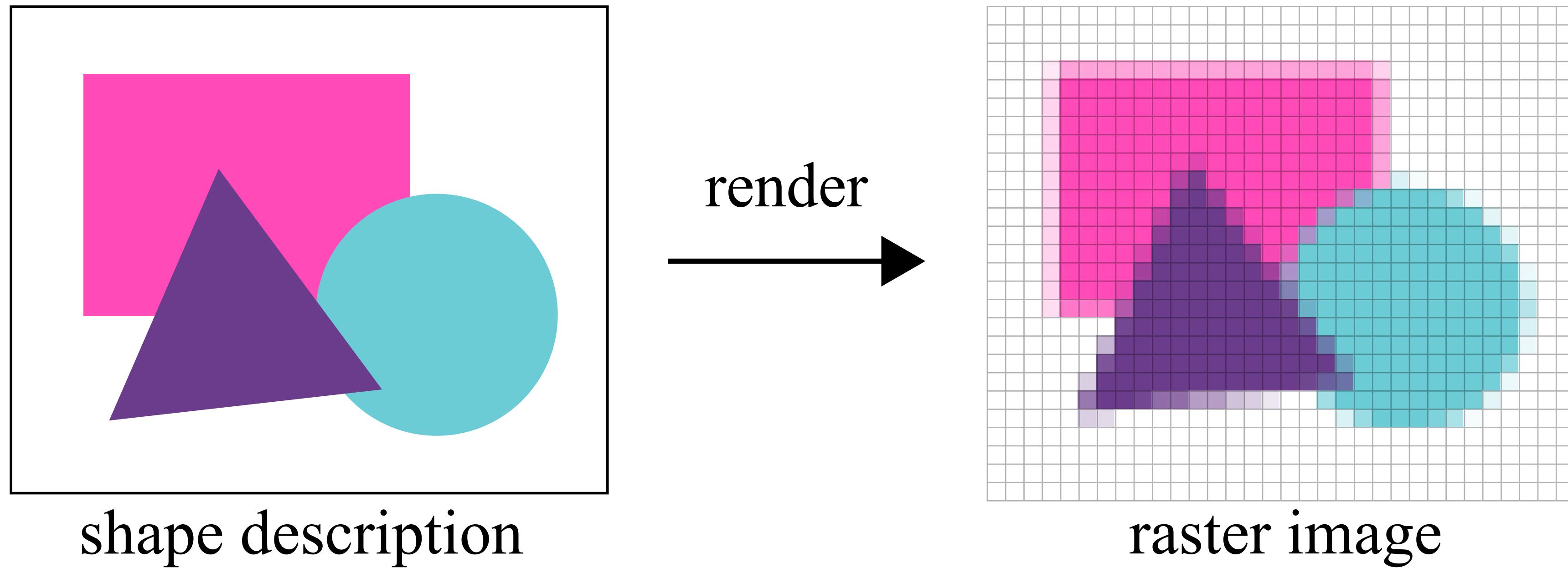
a “pixel” stores the average color around the pixel center



# A close-up look of an LCD screen

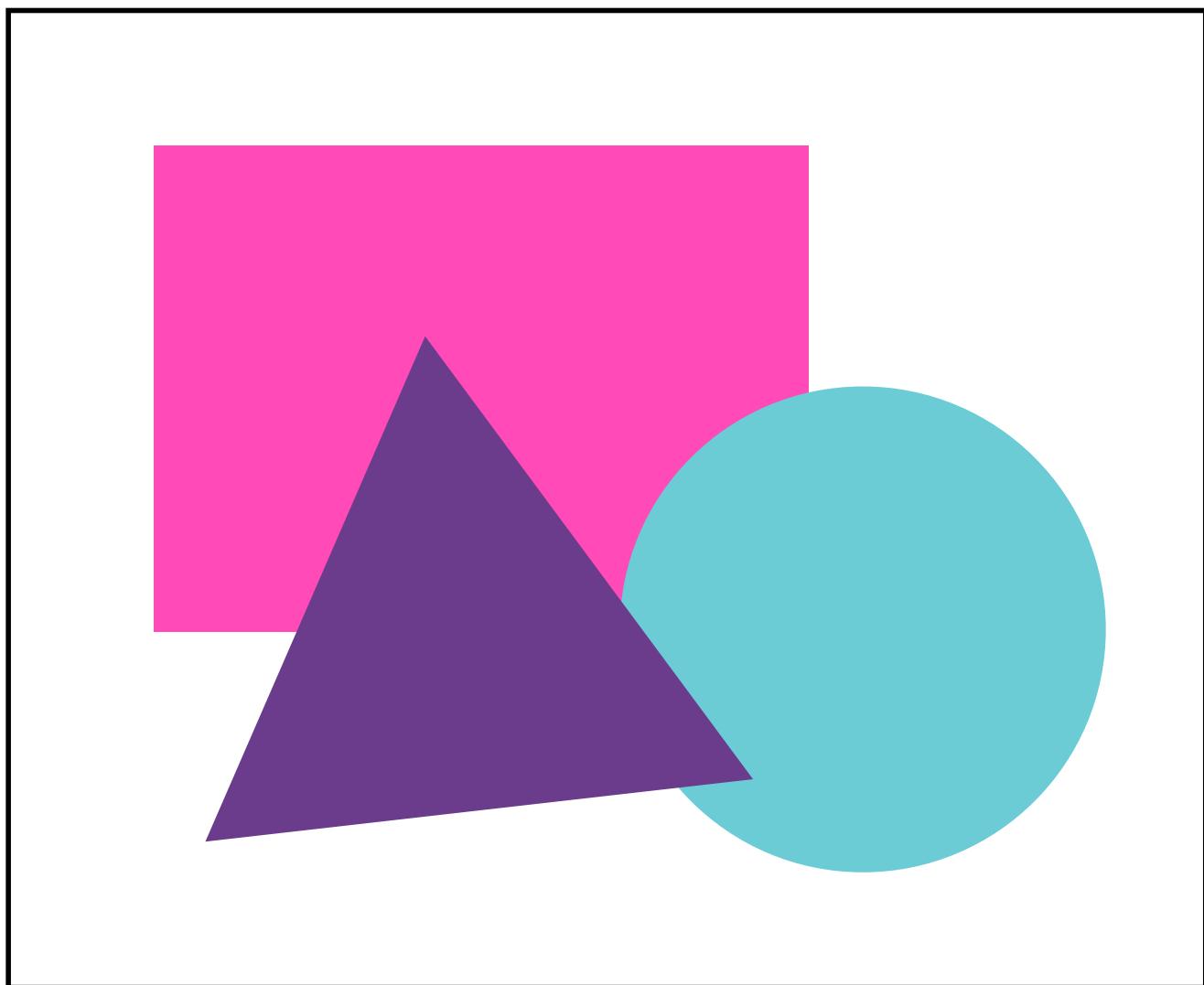


Rendering =  
painting on the raster image



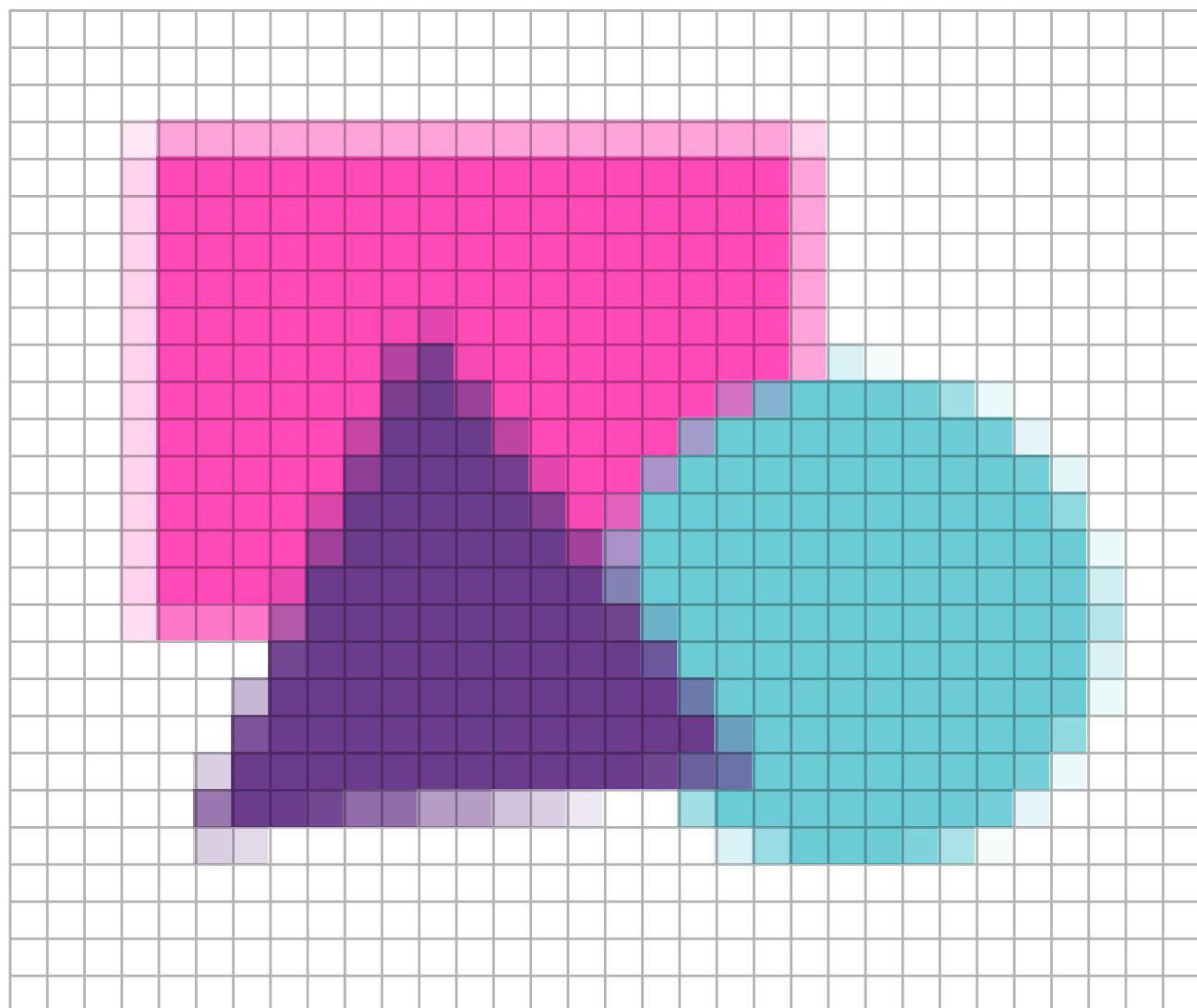
Rendering =  
painting on the raster image

in 2D, this is also called a “vector image”



shape description

render  
→



raster image

Rendering =  
painting on the raster image

in 3D, this is called a 3D “scene”

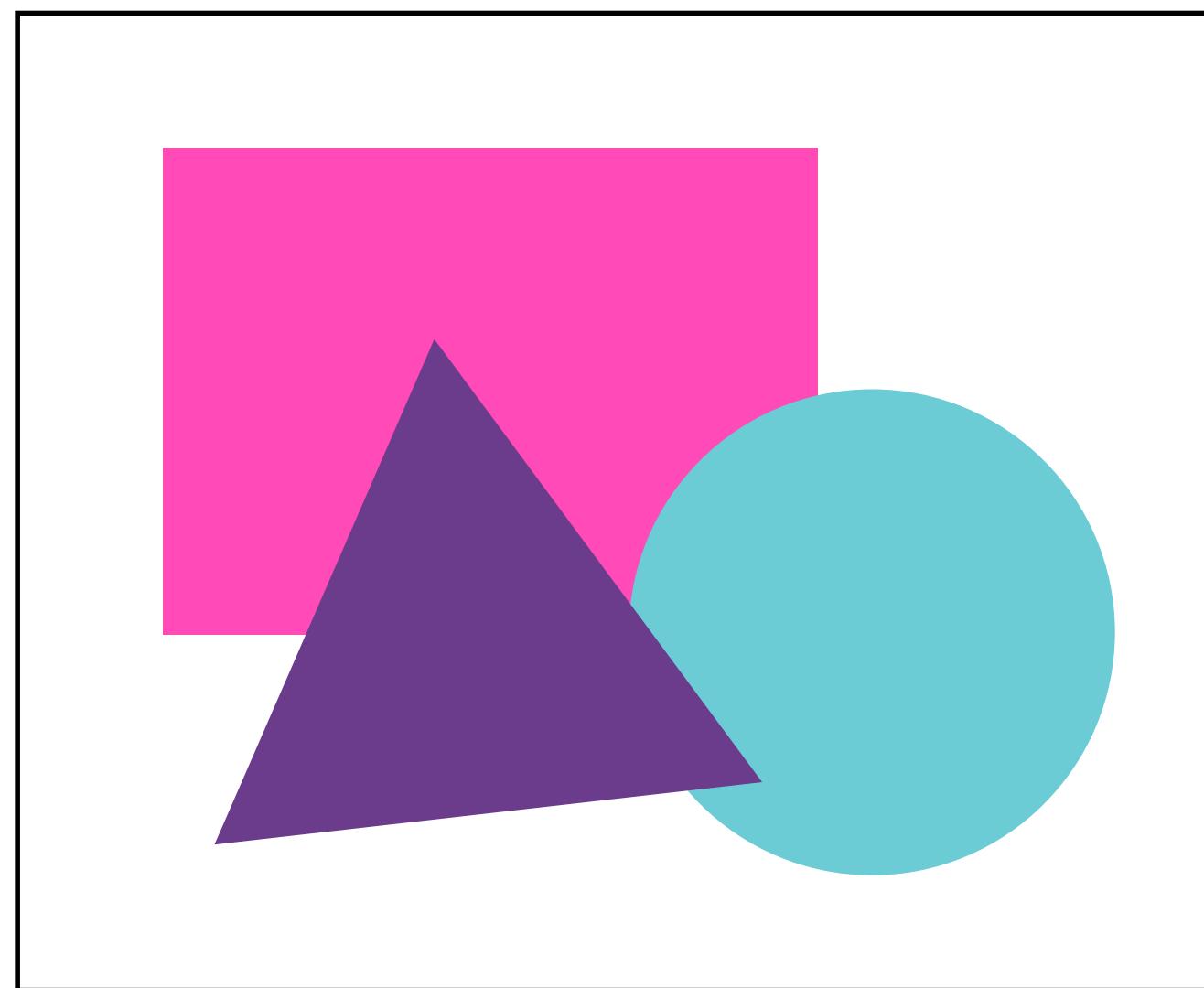


render  
→



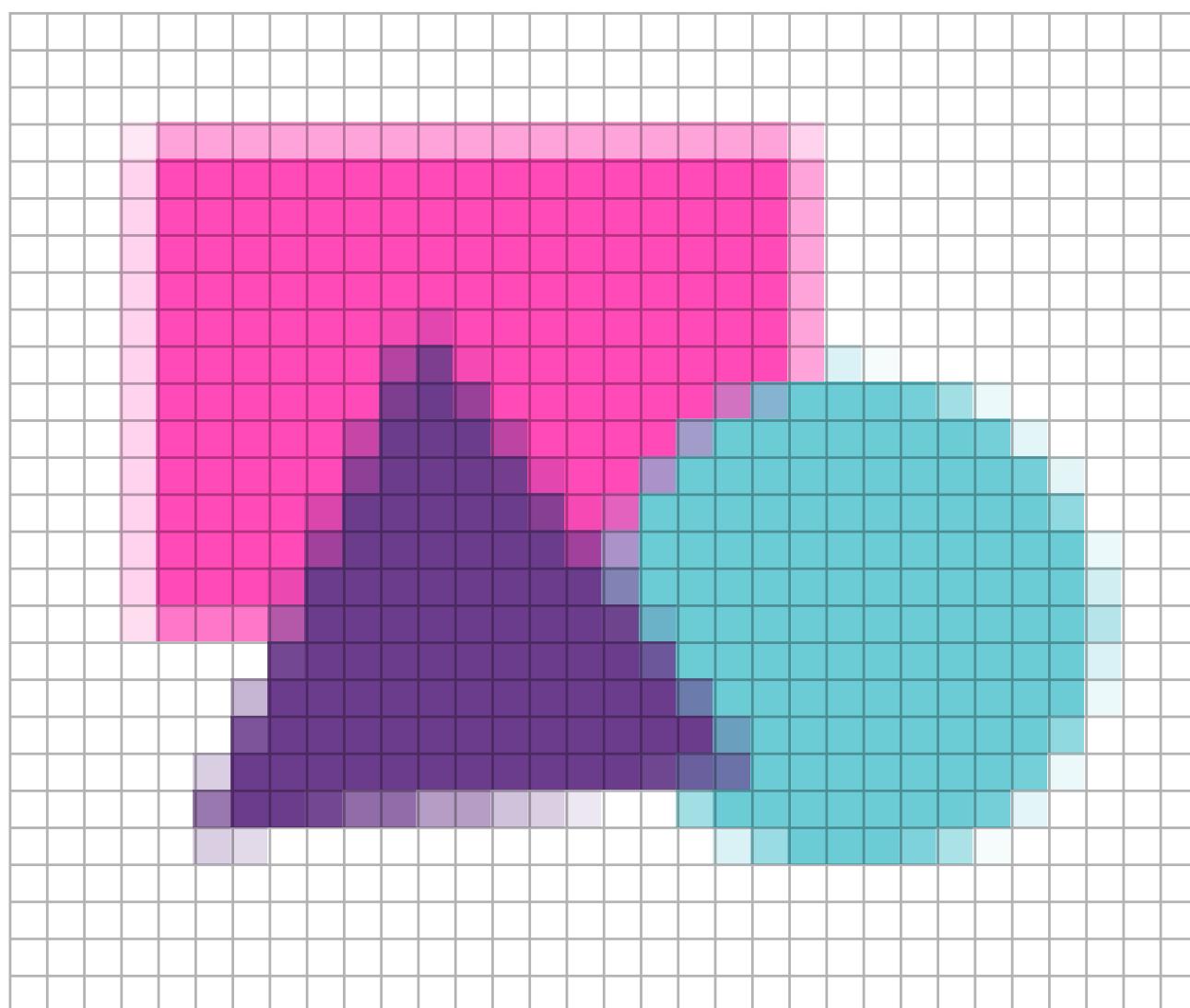
# Rendering = painting on the raster image

we will focus on 2D for now



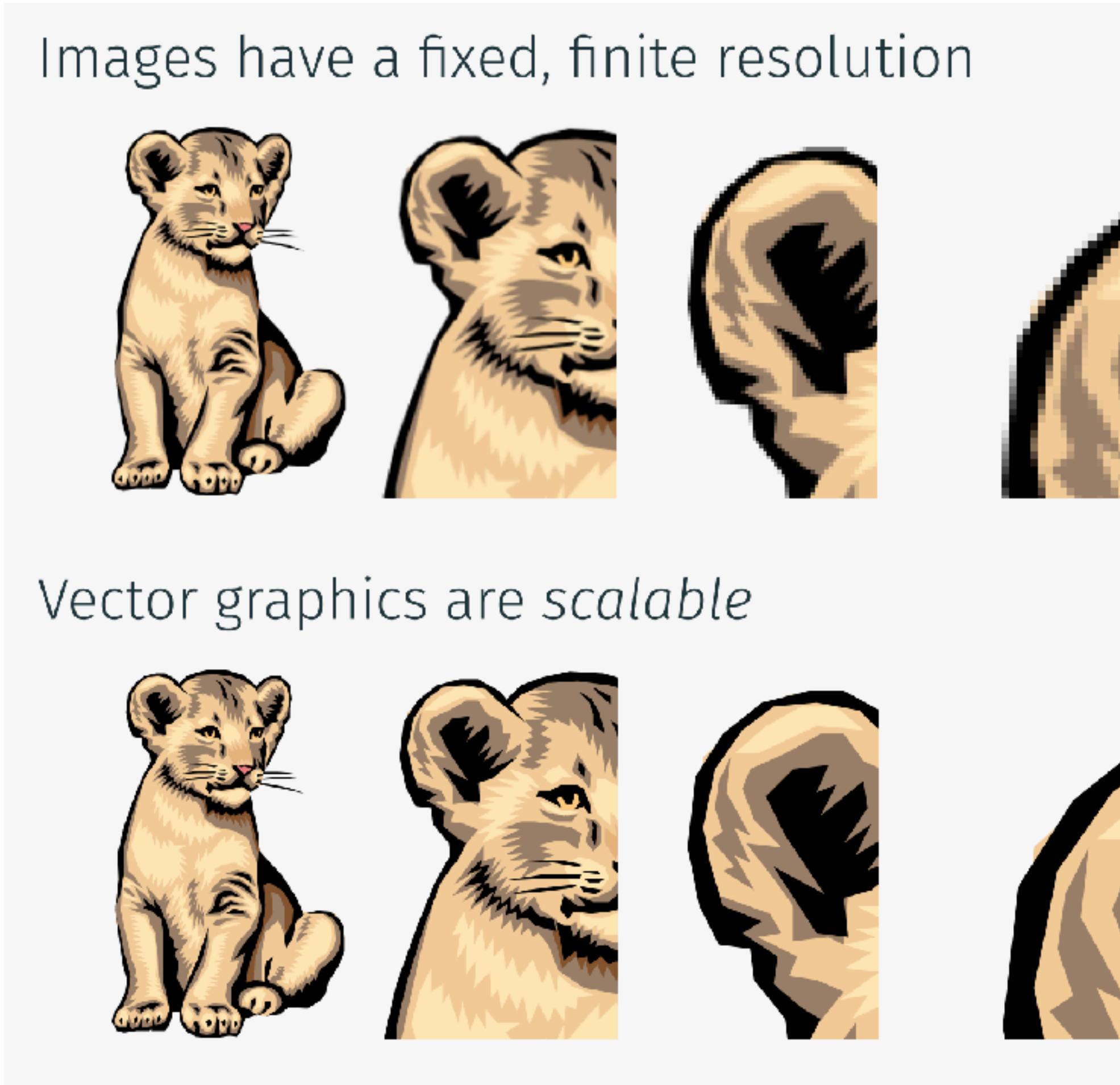
shape description

render  
→



raster image

# It is often more convenient to store vector images than raster images



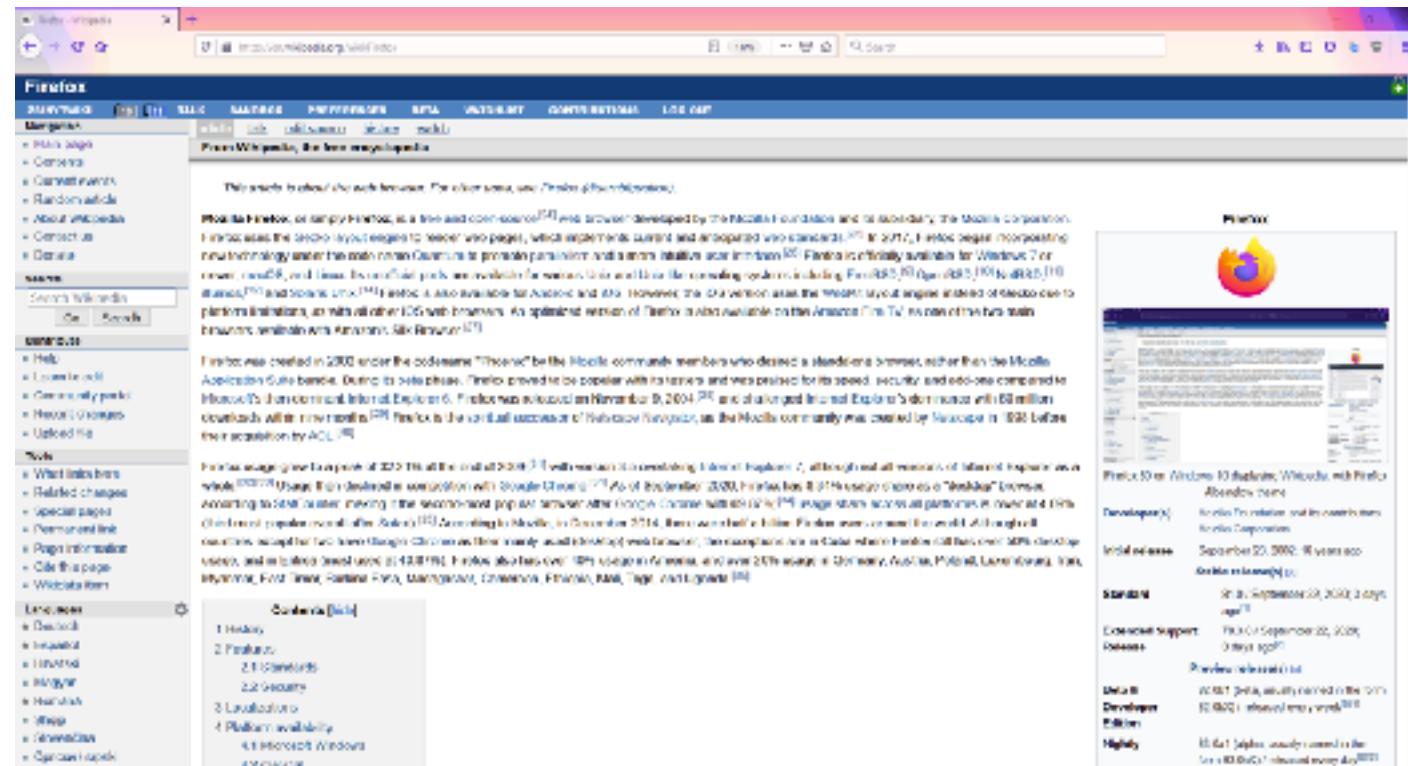
from Diego Nehab

<https://w3.impa.br/~diego/teaching/2021.0/slides-1.pdf>

# Vector graphics are everywhere



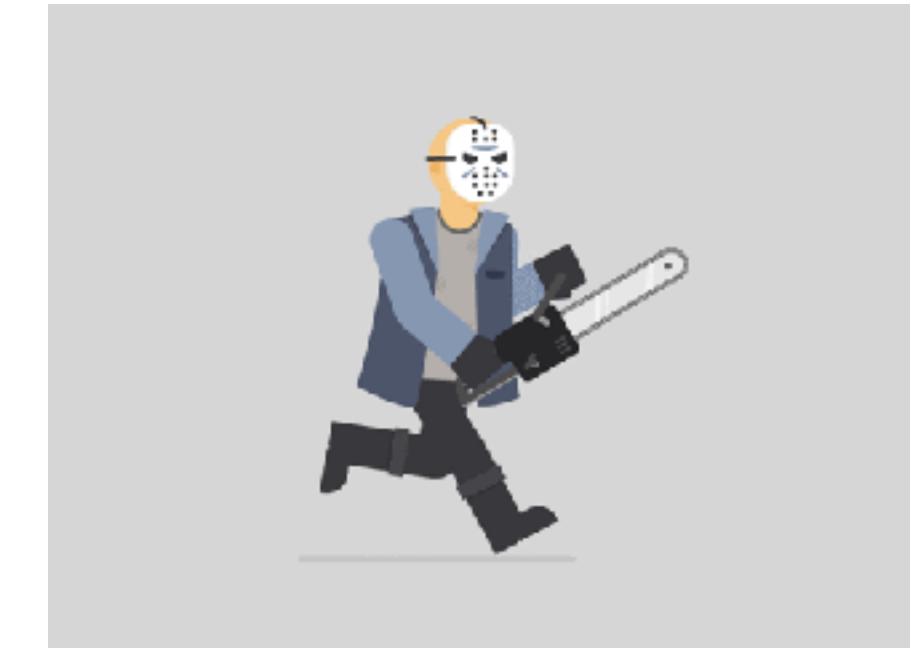
GUI



web design



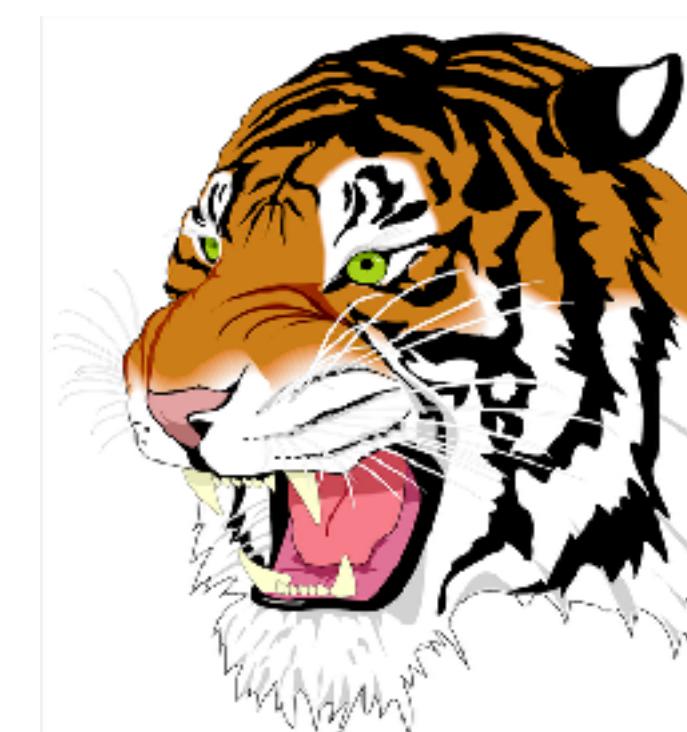
logo



animation

The quick brown fox jumps over the lazy dog  
The quick brown fox jumps  
over the lazy dog  
The quick brown fox jumps over the lazy dog

font



vector art



map

Differentiable Vector Graphics Rasterization for Editing and Learning

TZU-MAO LI, MIT CSAIL  
MICHAEL LUKÁČ, Adobe Research  
MICHAEL GHARBI, Adobe Research  
JONATHAN RAGAN-KELLEY, MIT CSAIL

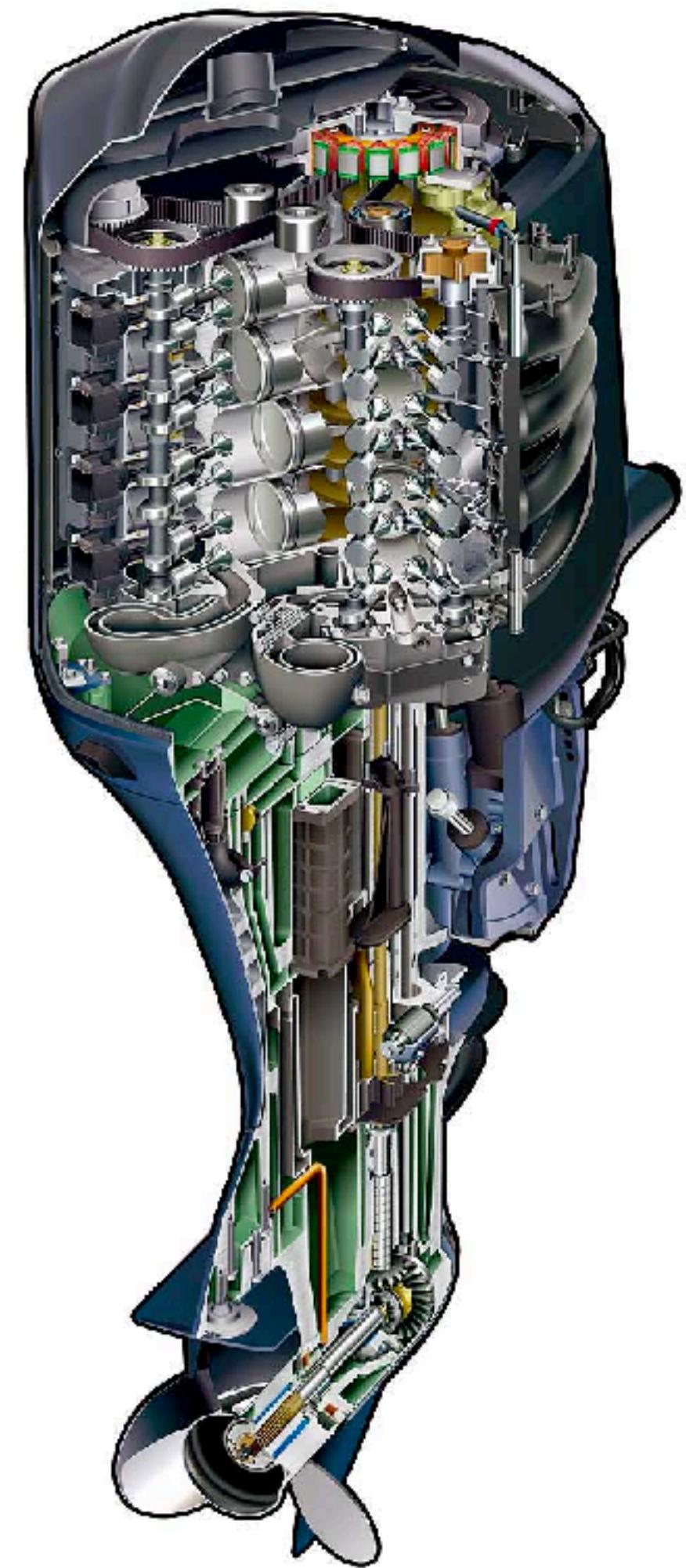
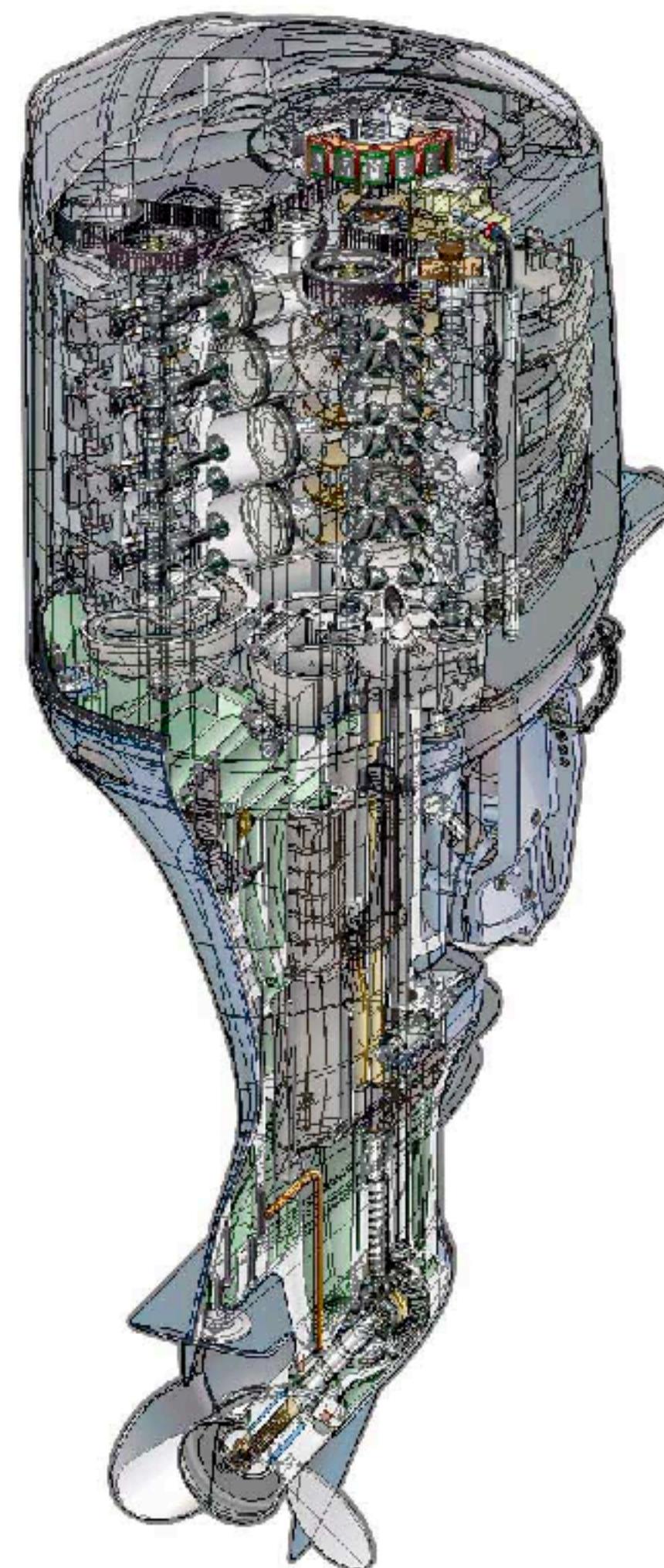


Fig. 1. We introduce differentiable rasterizers for vector graphics that bridge the vector and raster domains through backprojection. Differentiable rasterizers enable many real-world graphics applications. (a) Interactive editing that locally rasterizes thin vector strokes such as opacity under geometric constraints. (b) Cross-silhouette rendering that aligns two 3D models under fisheye cameras to a target image. (c) Training of an image generation model. (d) Editing vector graphics using generatively controlled stroke length increasing operations, vector sculpturing [Li et al., 2018] and [Li et al., 2019]. (e) Image refetching, (f) training a neural encoder-decoder [Sigma and Seung, 2014], to generate vector WINE logo [LeCun et al., 1995] and linking styles similar to photomosaics. Image courtesy of wikipedia User:Odest and User:Kather, and Freyberg.org User:ChenQian.

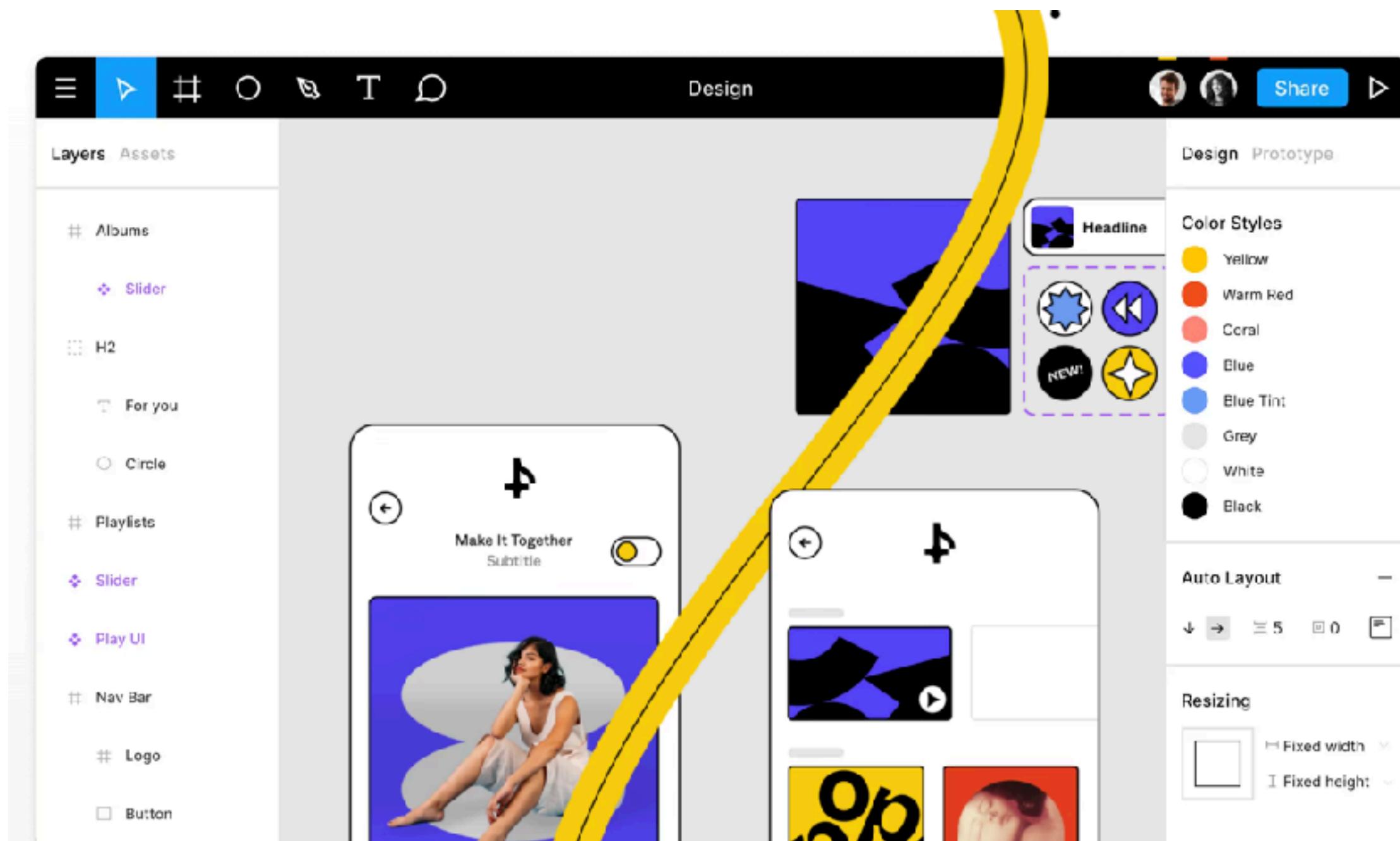
academic paper

# Amazing vector art

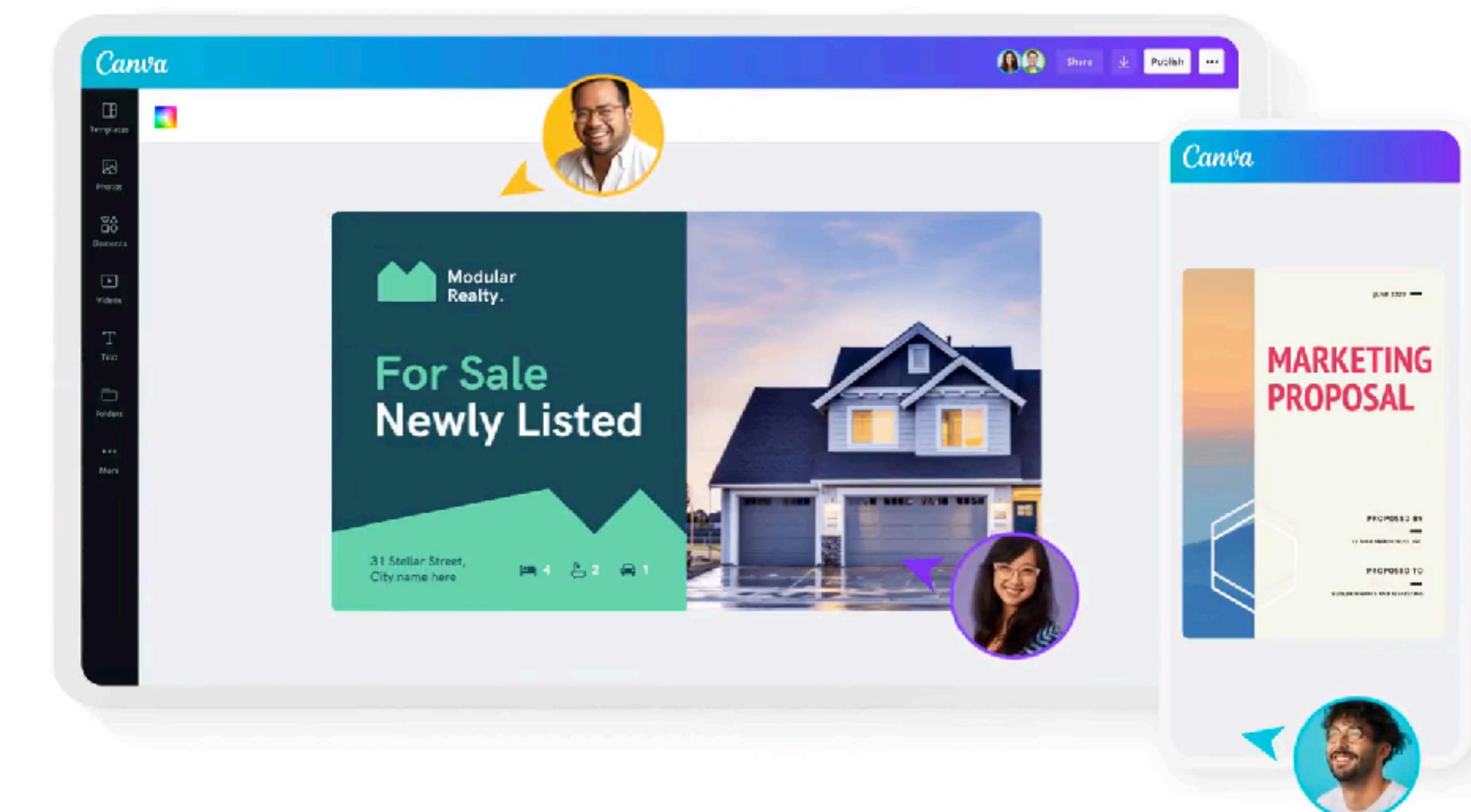
from Yukio Miyamoto



# Vector graphics startups



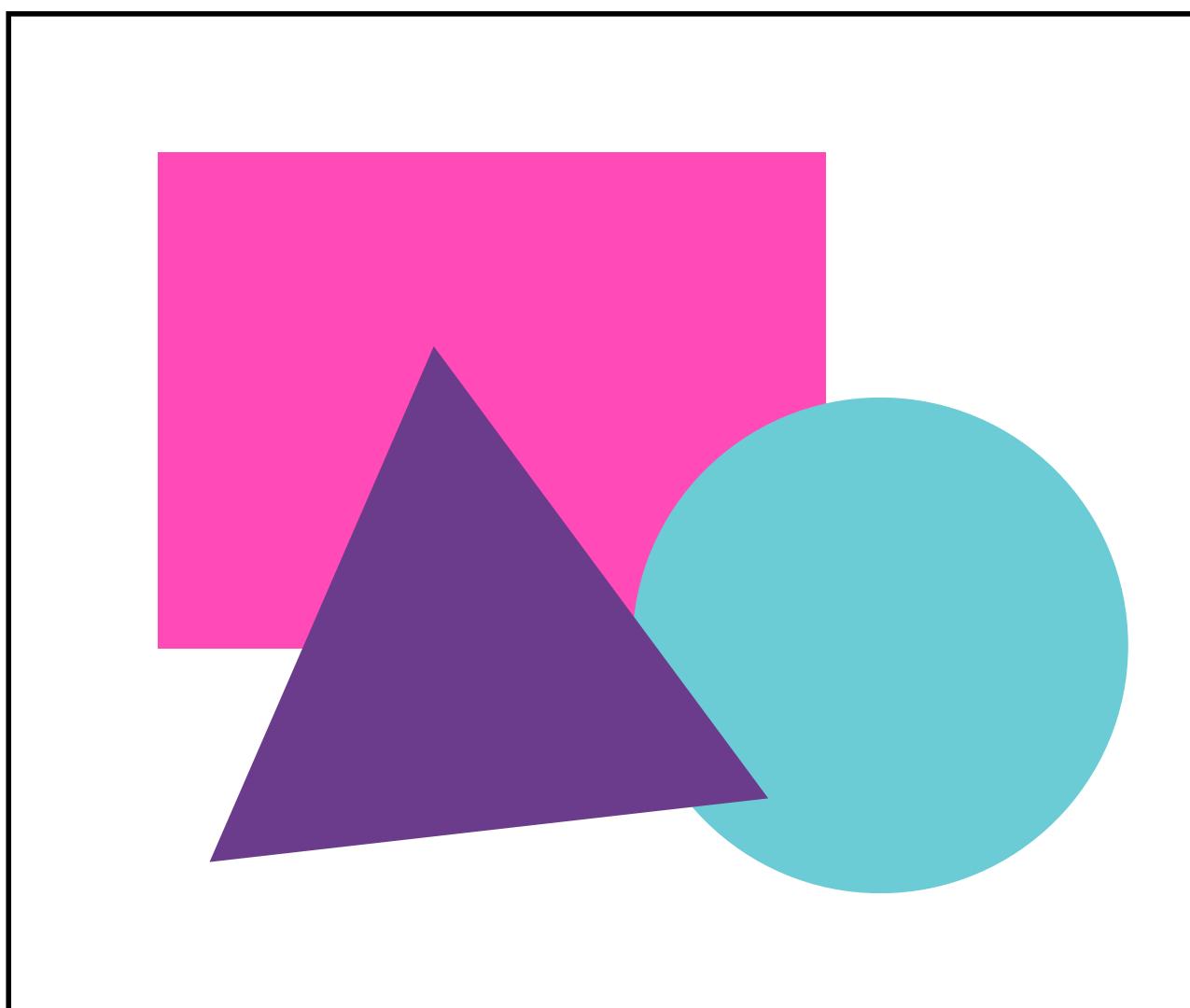
Figma



Canva

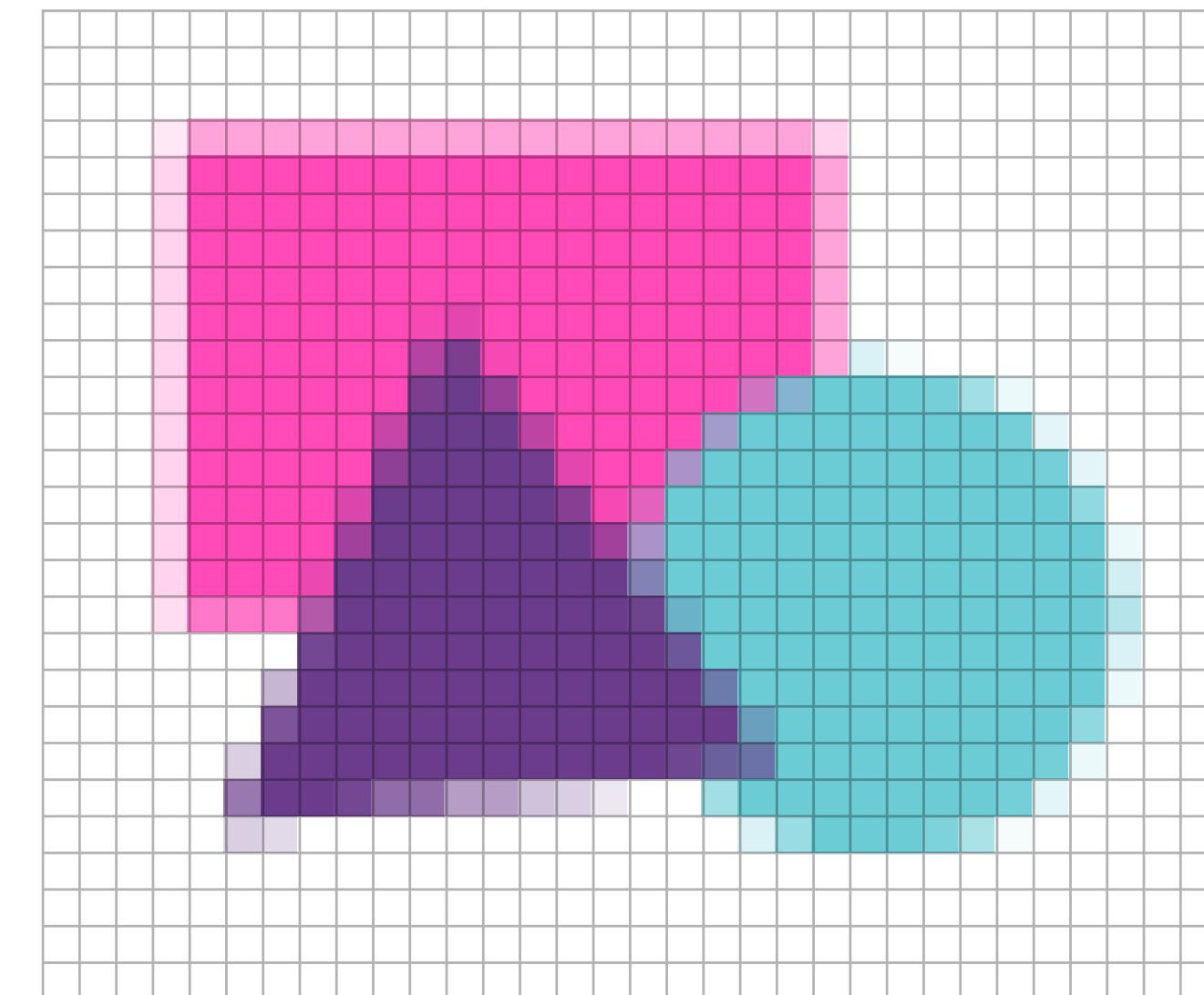
# Today: rasterizing 2D primitives

(what you need to do in Homework 1!)



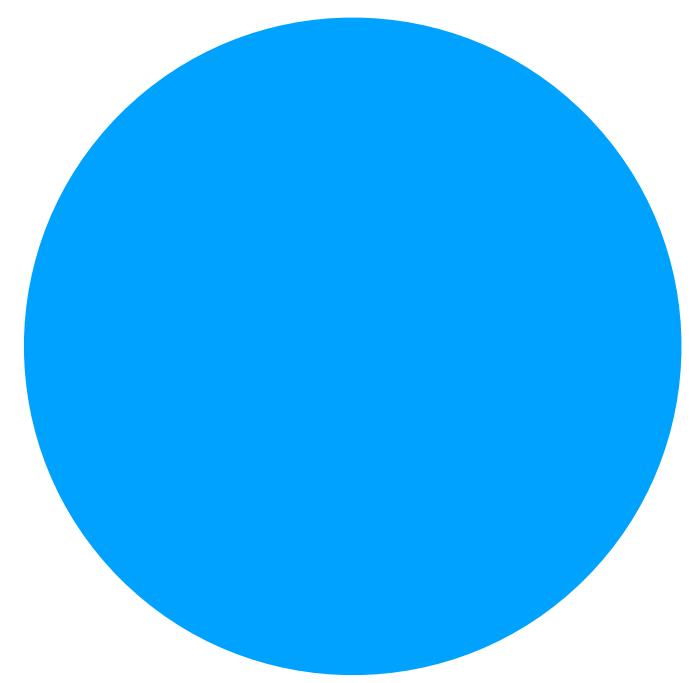
shape description

render  
→



raster image

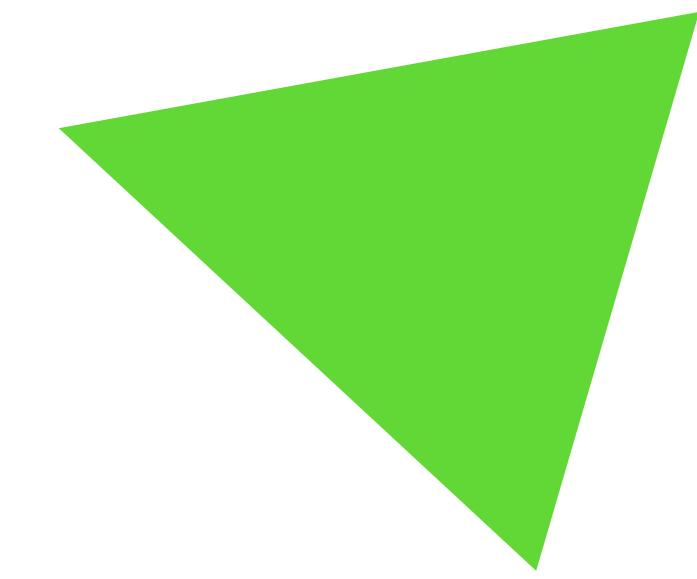
# Some common primitives



circles



rectangles



triangles

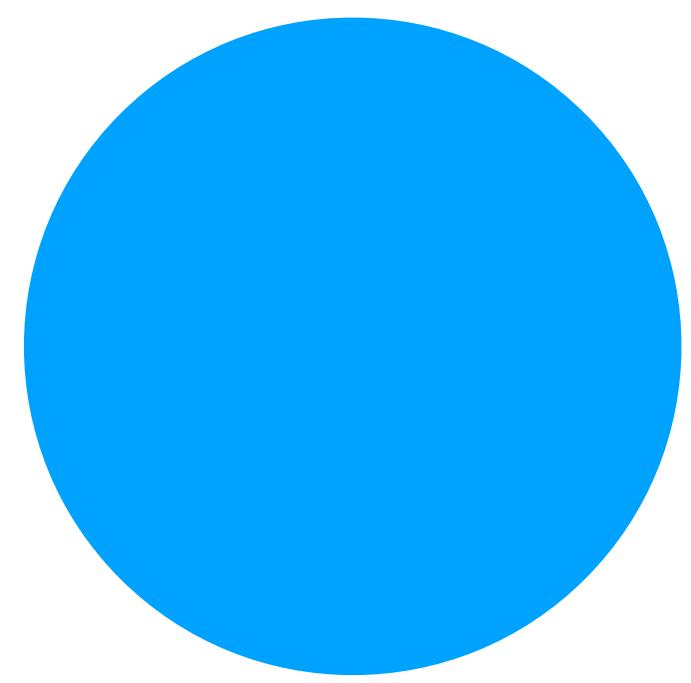


lines

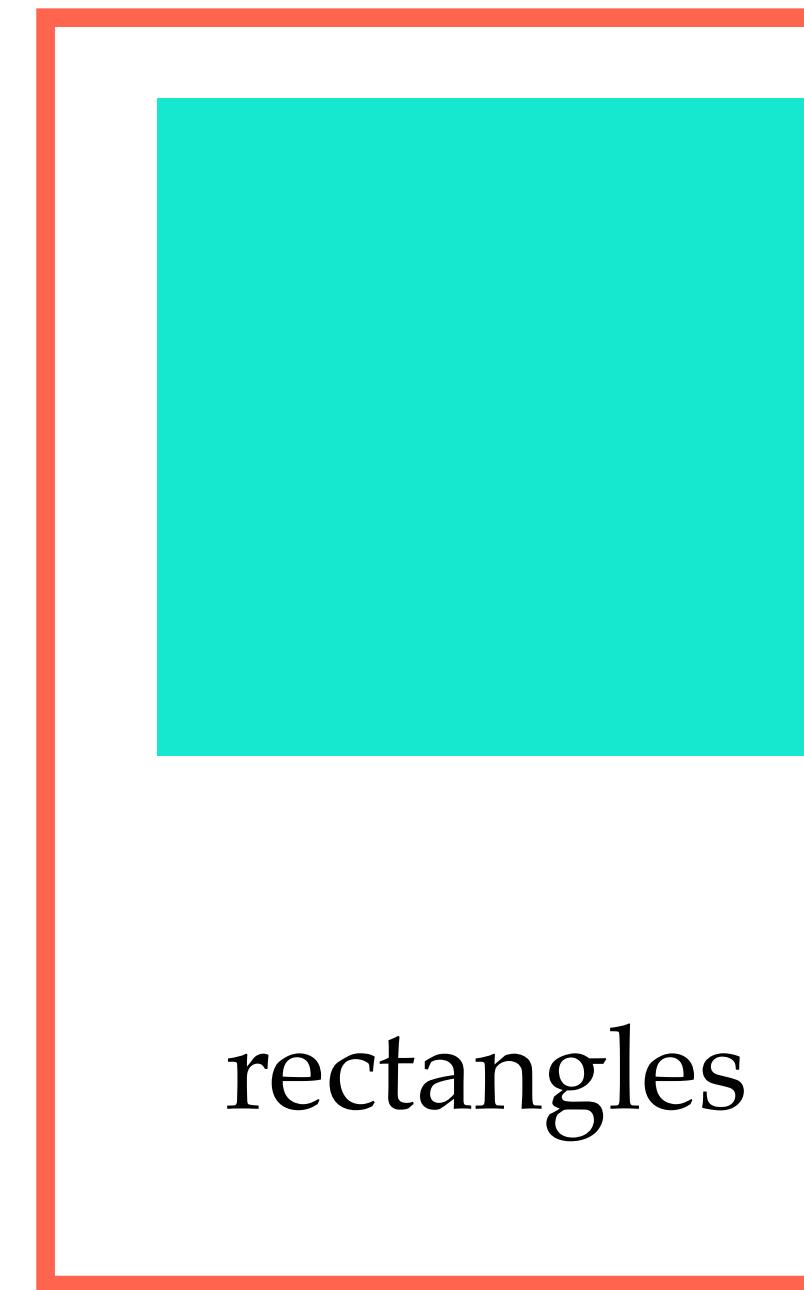


curves

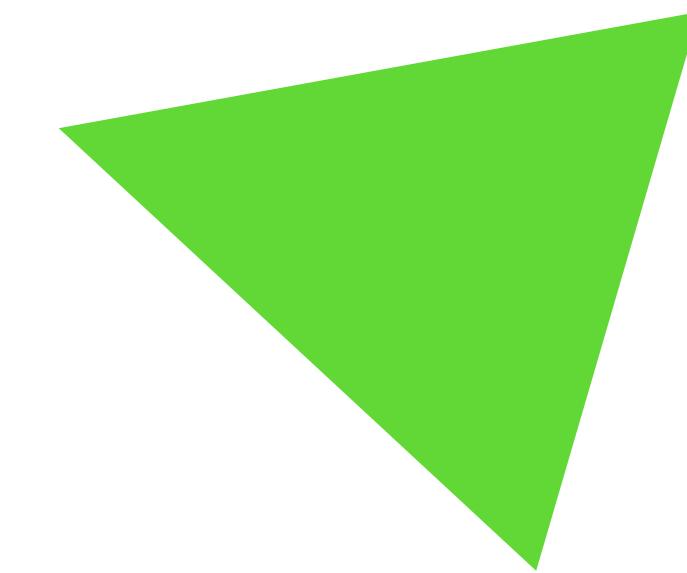
# Some common primitives



circles



rectangles



triangles

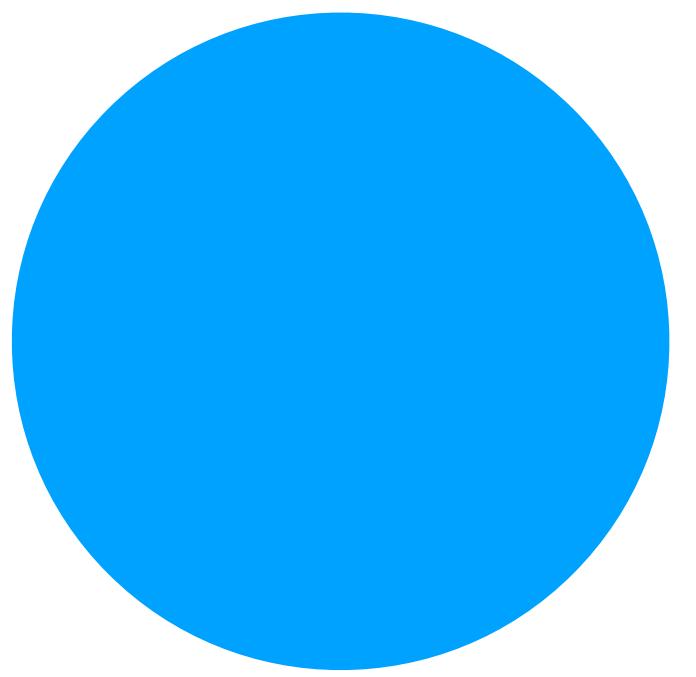


lines



curves

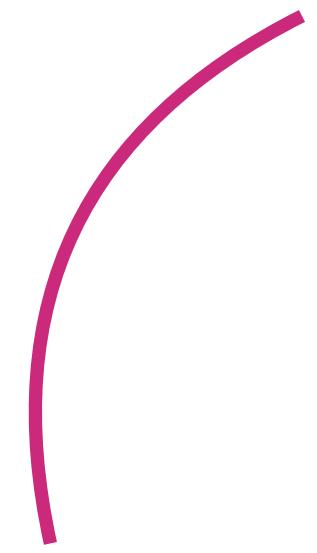
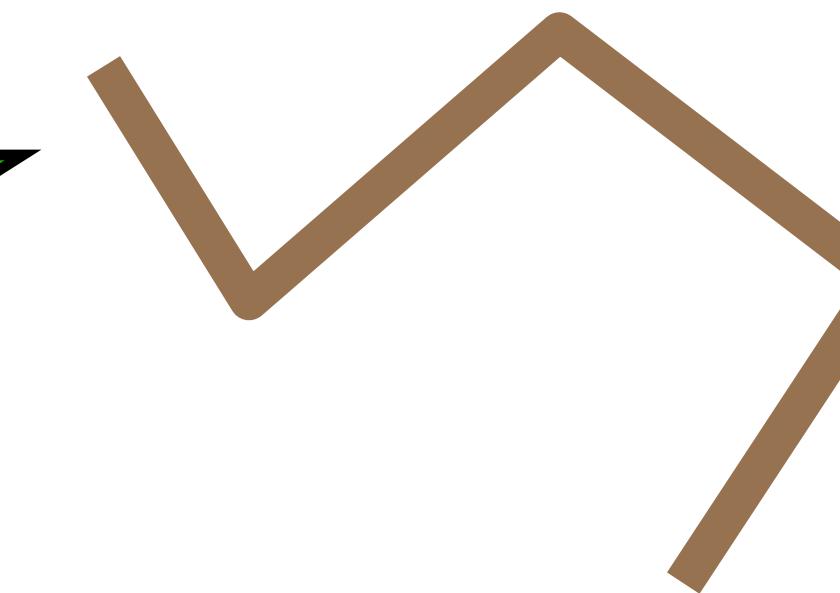
# Some common primitives



circles

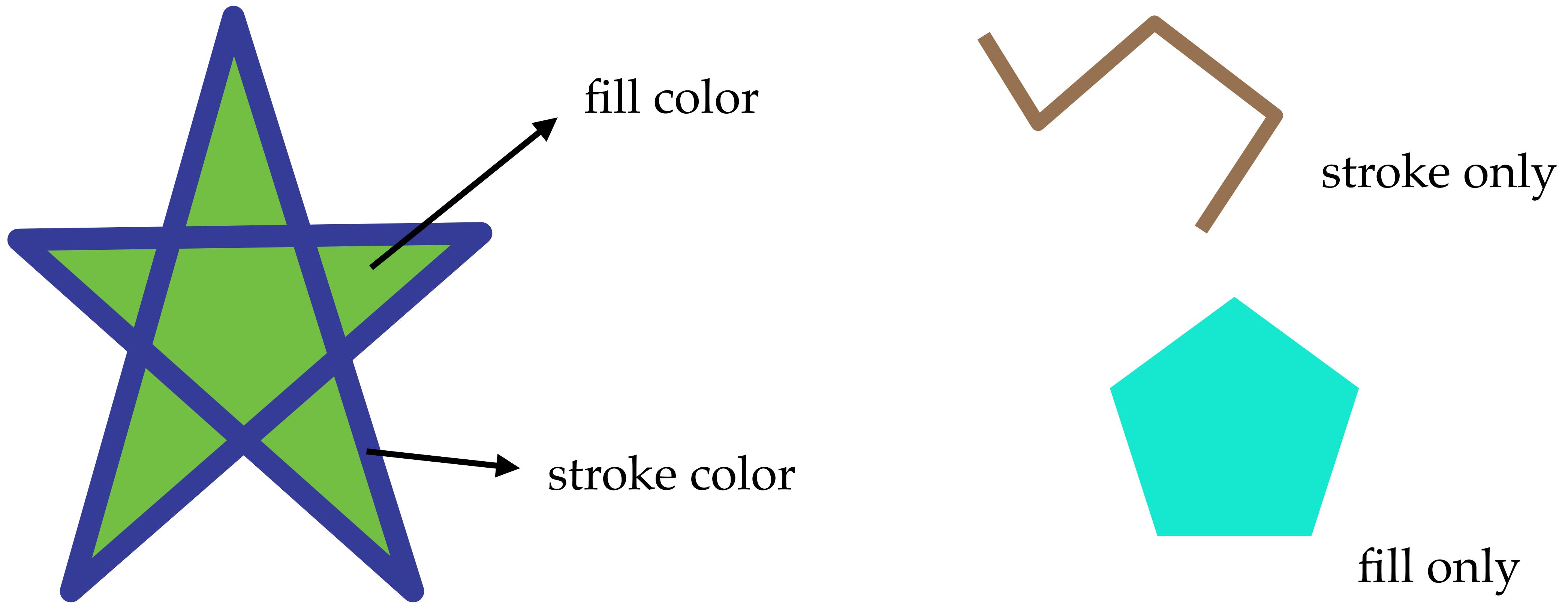


“polylines”



curves

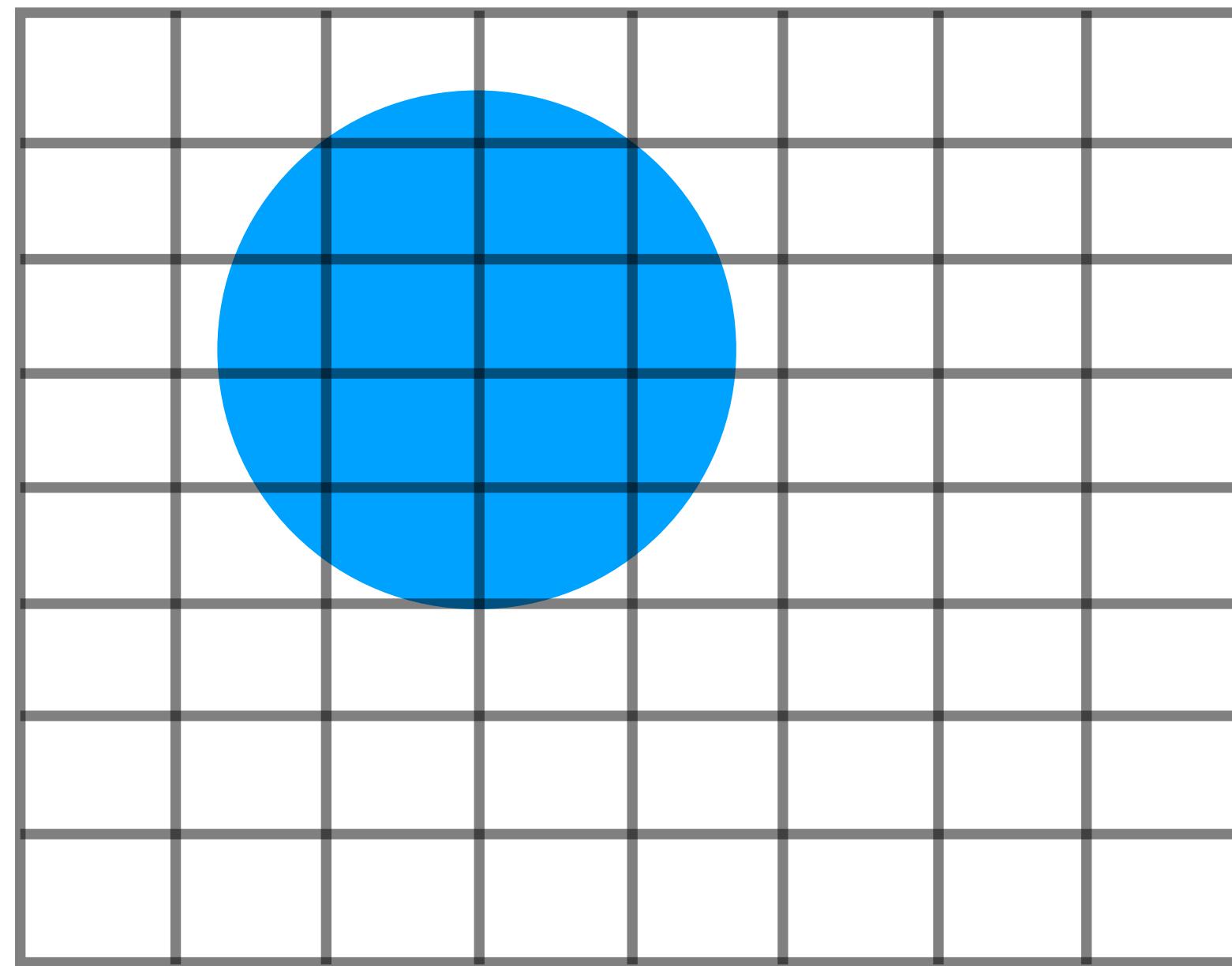
A shape can have “fill color” and  
“stroke color”



from the Scalable Vector Graphics standard

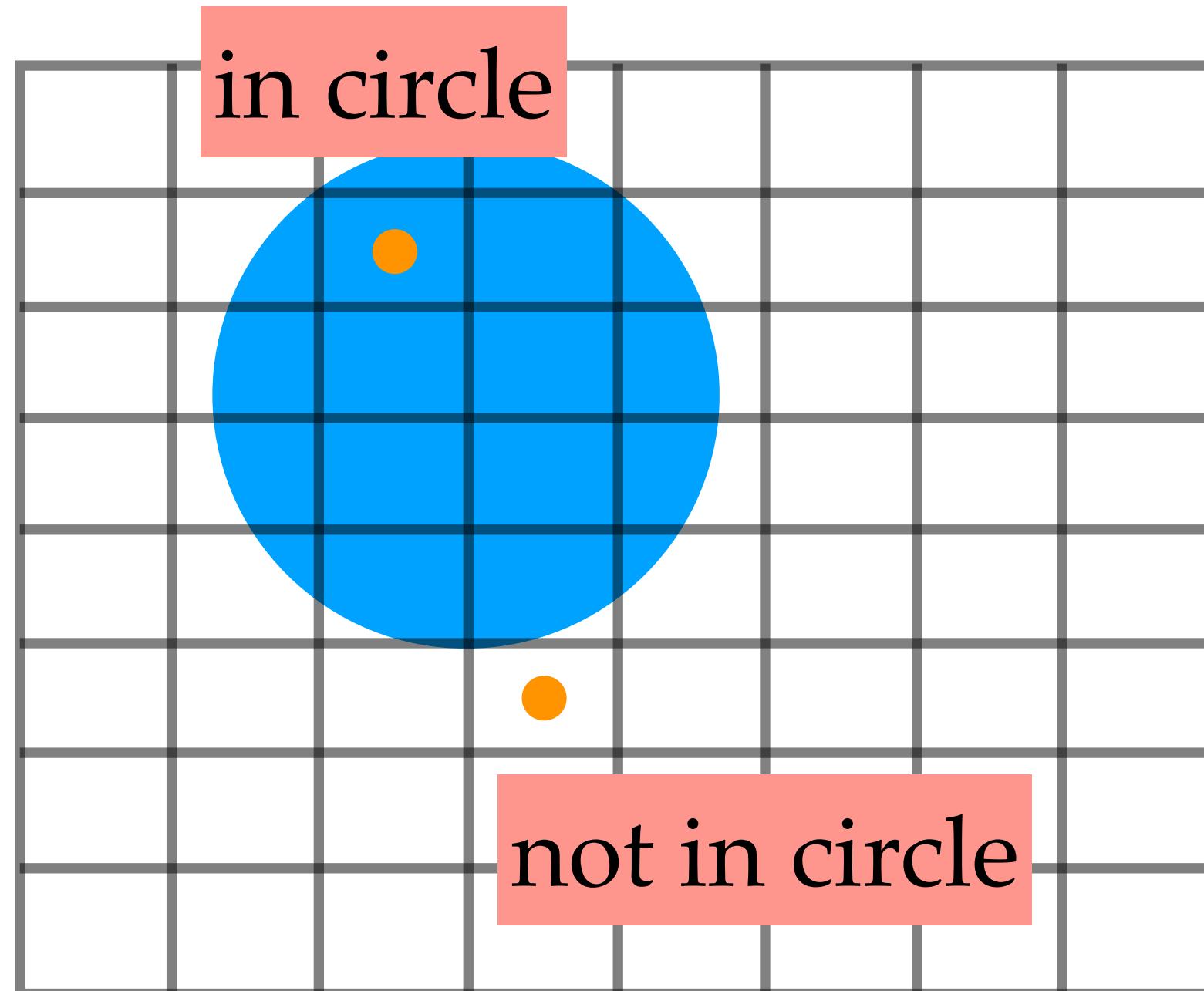
[https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorials/SVG\\_from\\_scratch/Fills\\_and\\_strokes](https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorials/SVG_from_scratch/Fills_and_strokes)

Let's first assume we only have one  
fill-only circle



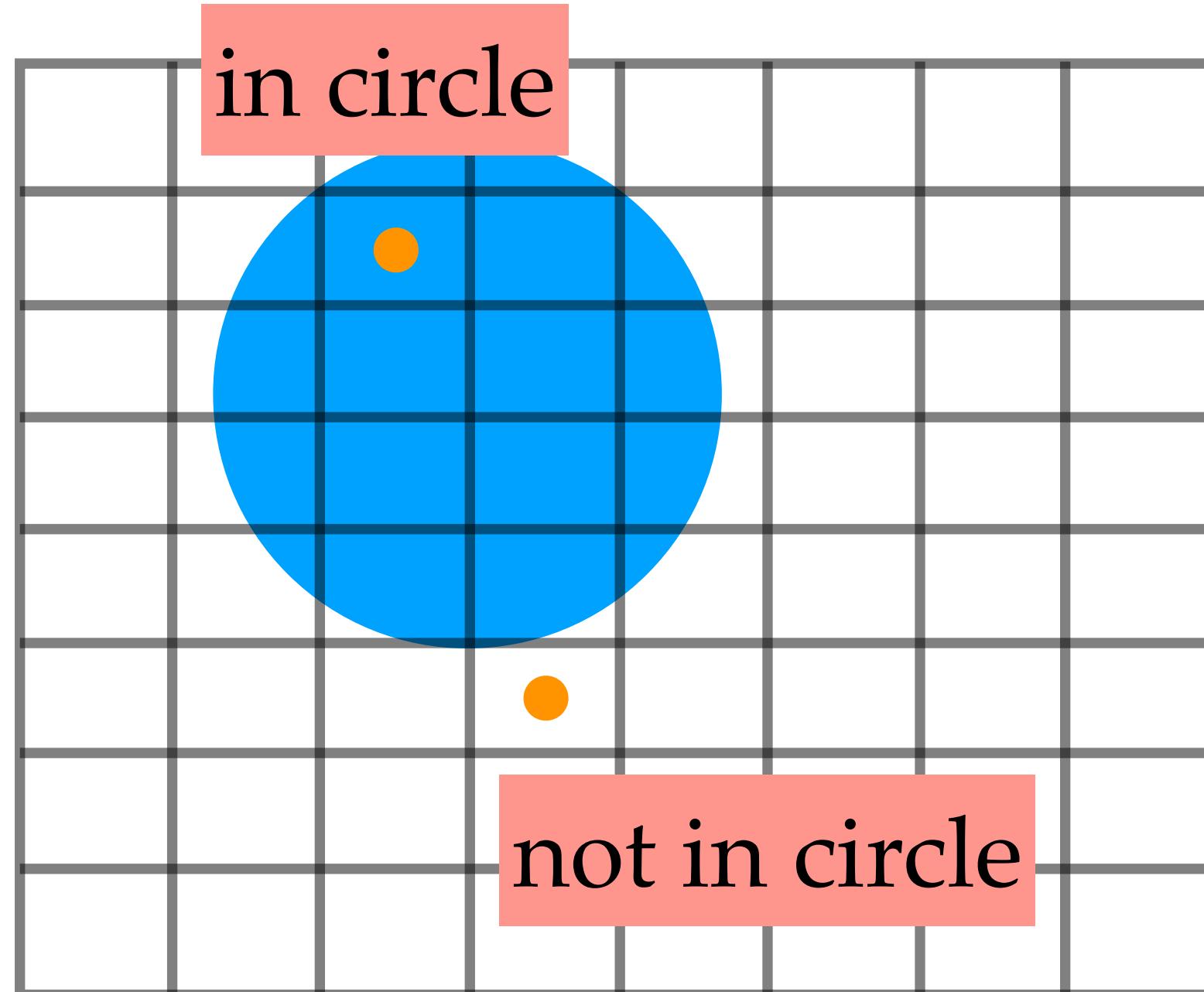
Q: how would you render it?

# Let's first assume we only have one fill-only circle



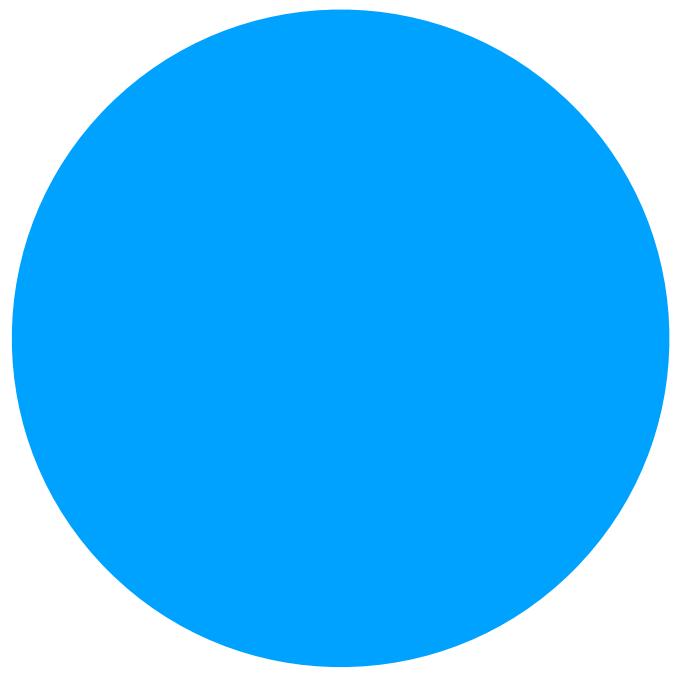
```
for each pixel
    if pixel center hits the primitive:
        color = primitive.fill_color
    else:
        color = background
```

Let's first assume we only have one  
fill-only circle



```
for each pixel
  if pixel center hits the primitive:
    color = primitive.fill_color
  else:
    color = background
```

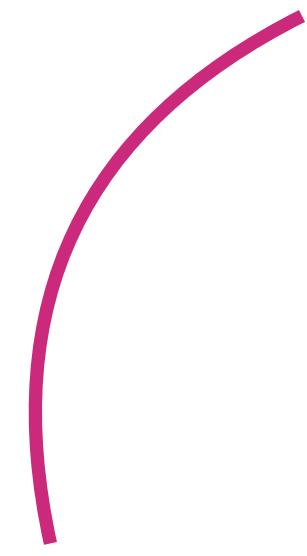
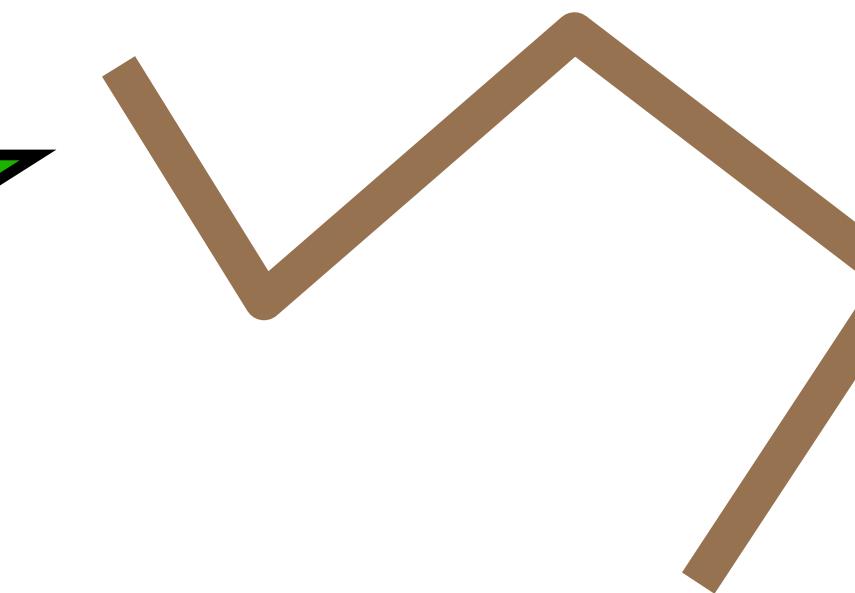
# Point in primitive test



circles

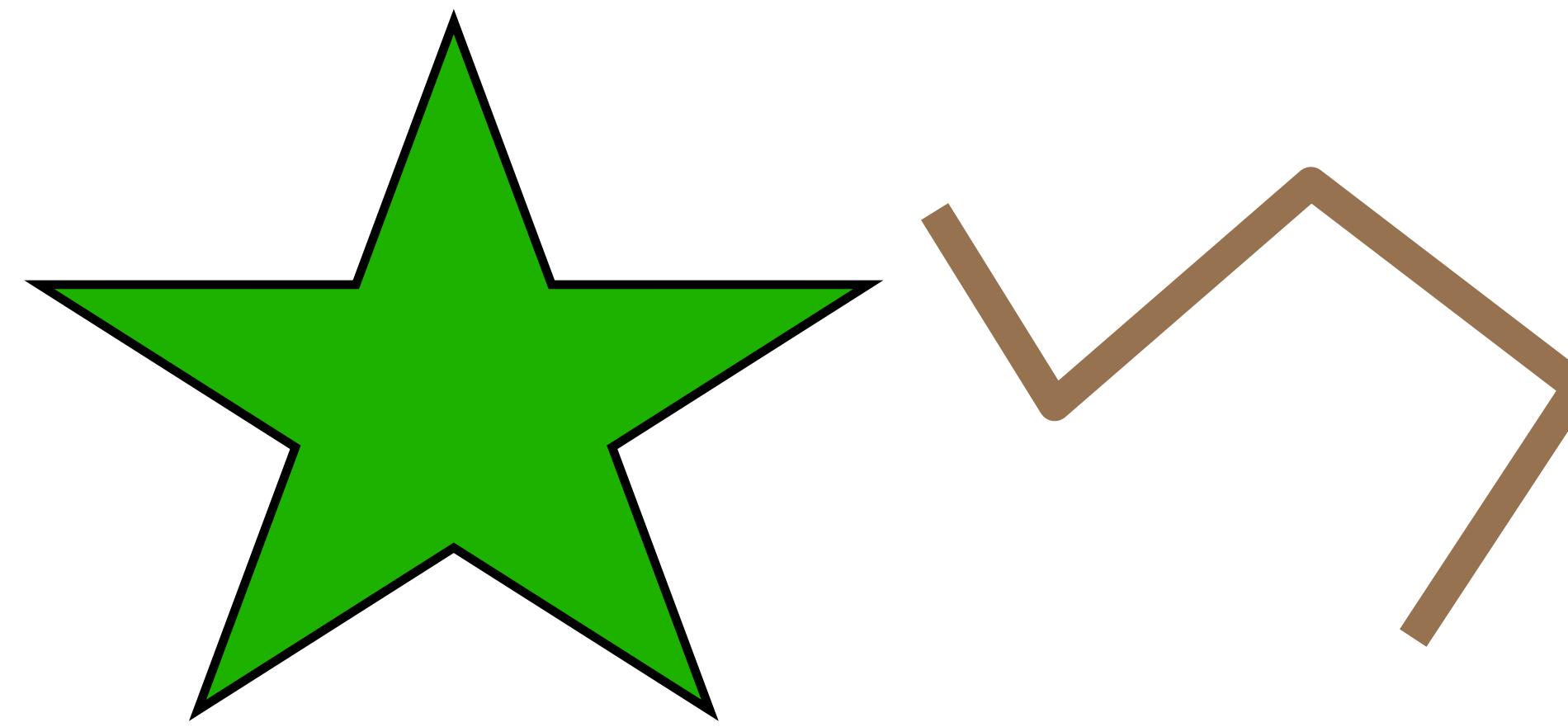


“polylines”



curves

# Point in primitive test

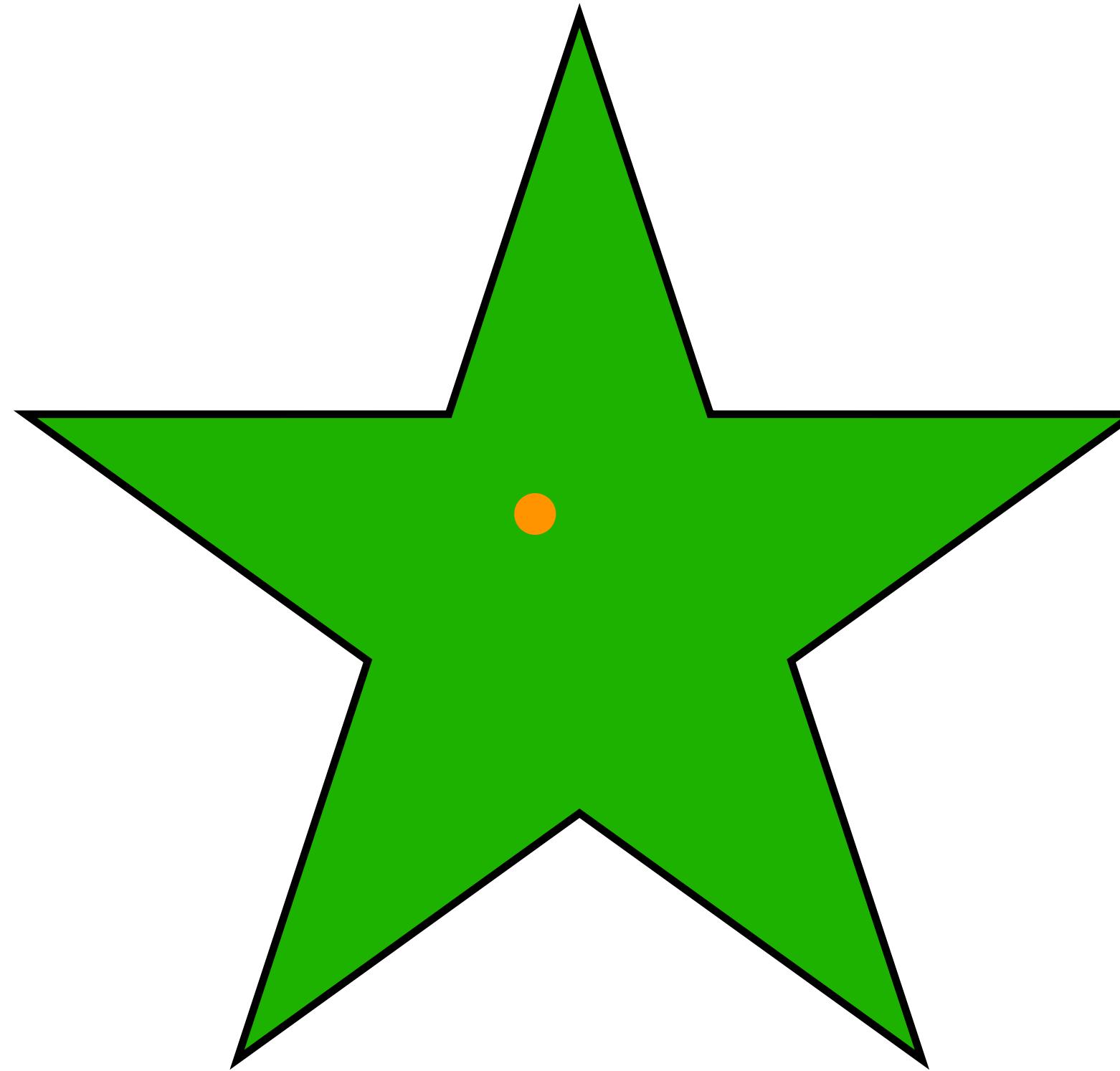


“polylines”

You figure it out!

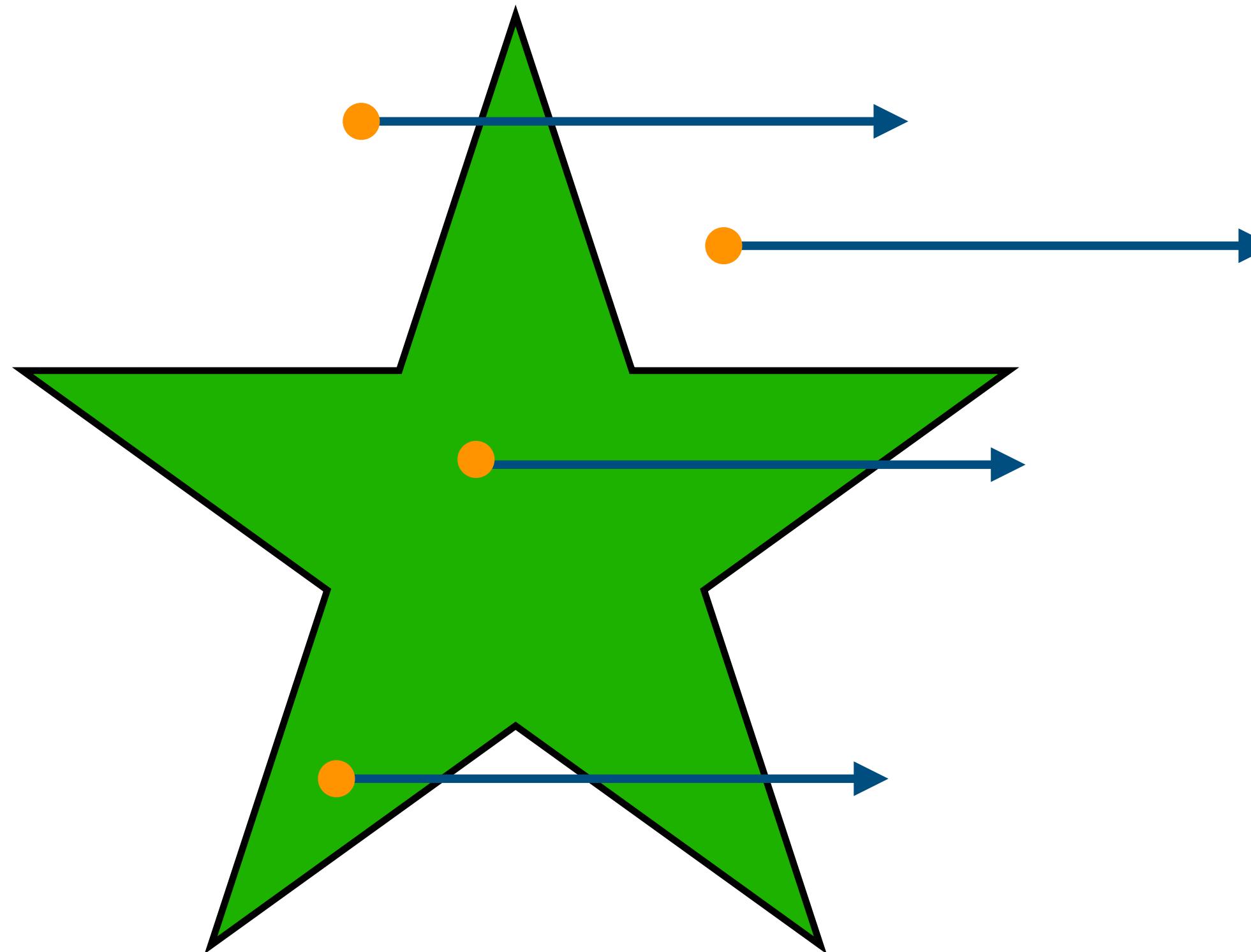
More about them later

# Point in polyline test



How do we know if a point is inside a closed polyline or not?

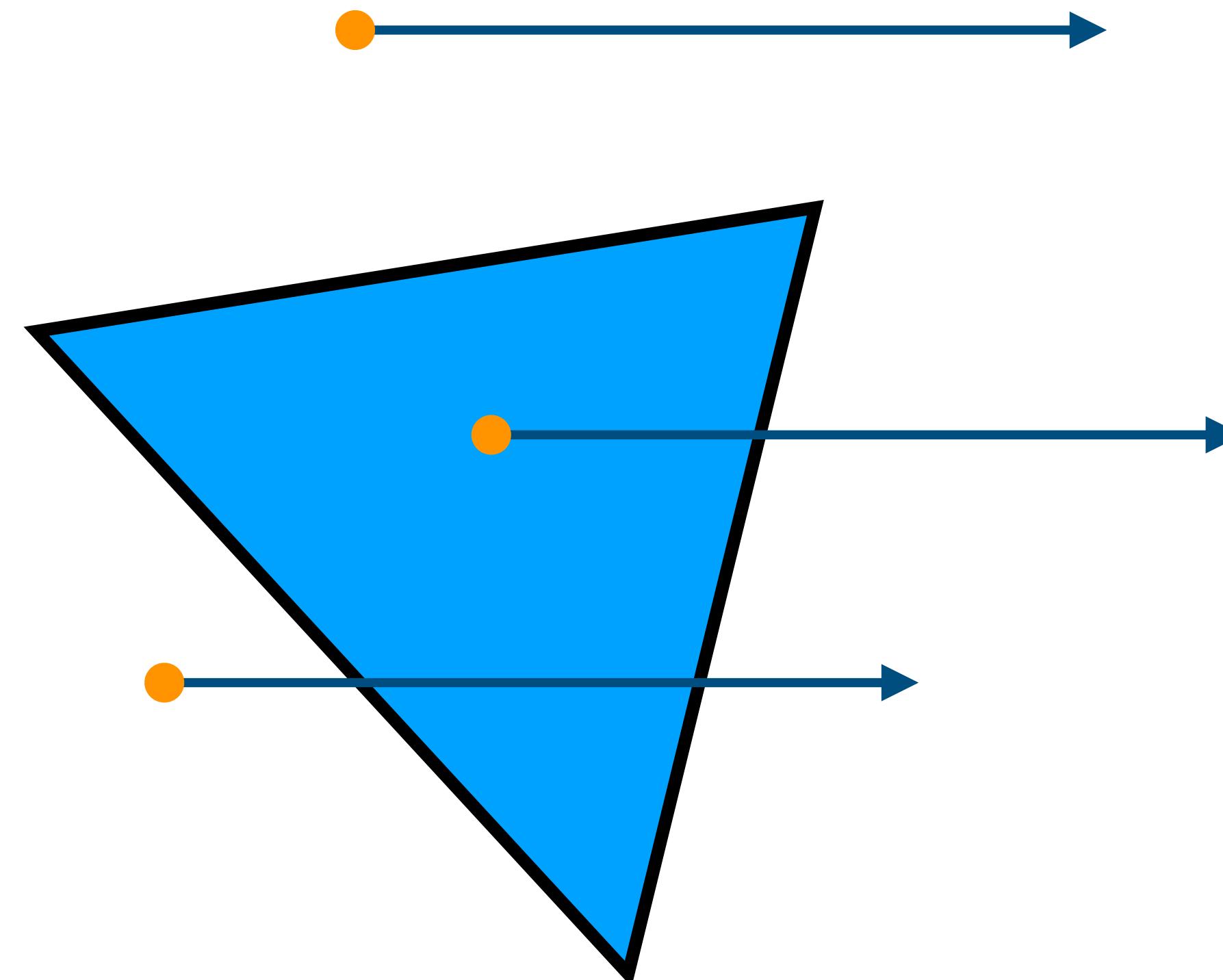
# Point in polyline test



Idea: cast a ray from the point, and count the intersections with the shape

Q: what's the criteria?

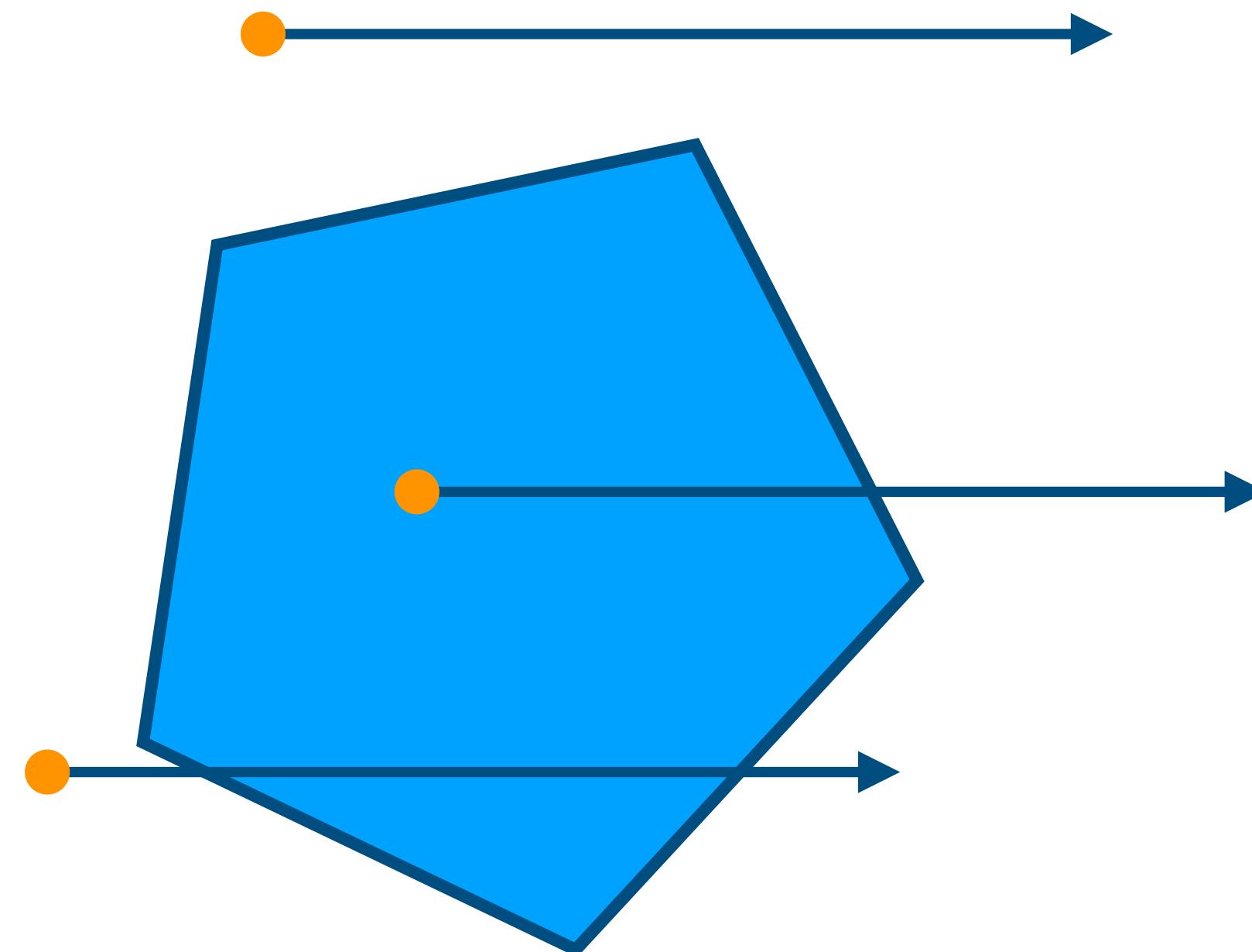
# For a triangle...



rays from points outside intersect with the triangle edges  
either zero times or two times

rays from points inside intersect with the triangle edges  
exactly one time

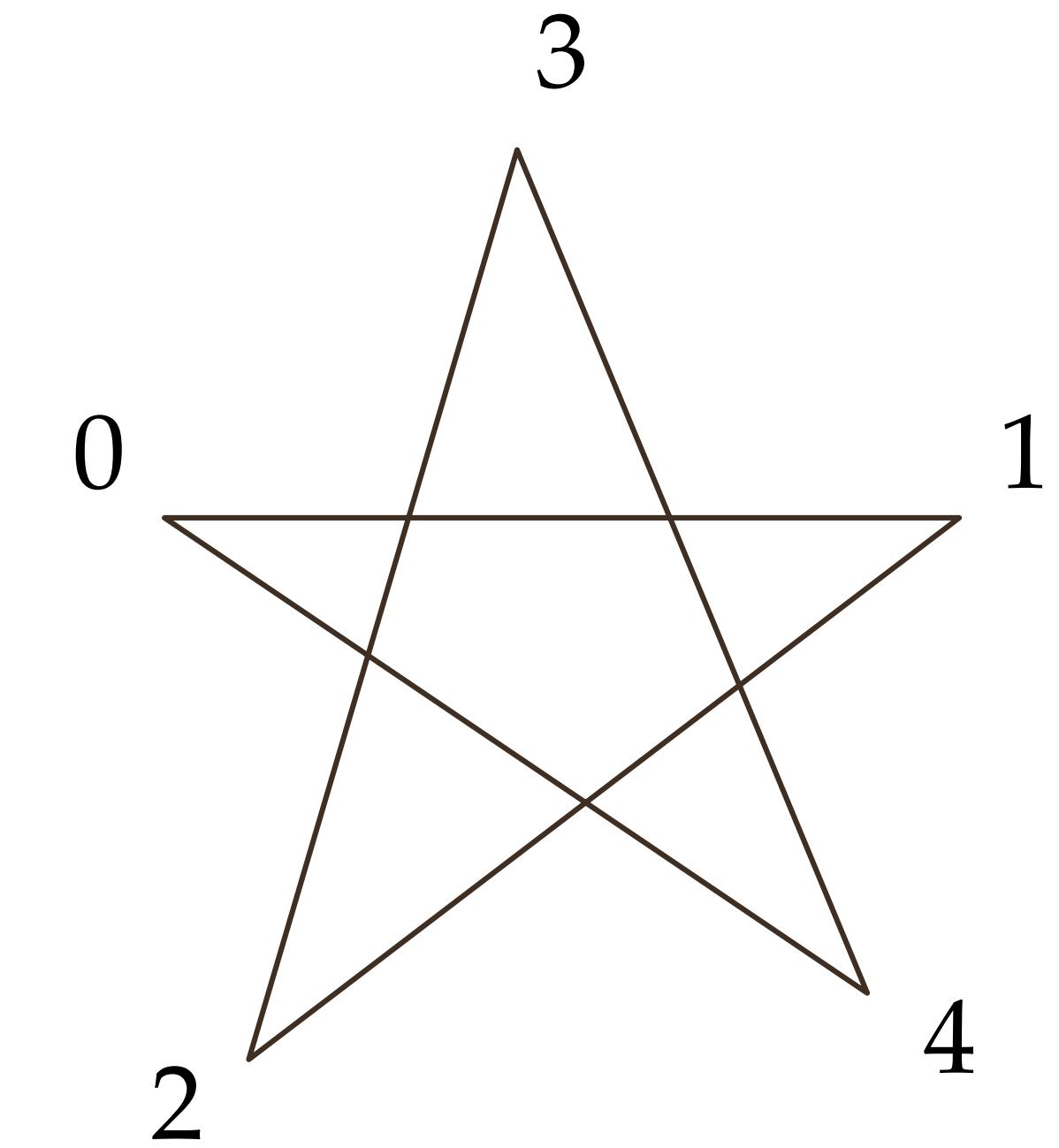
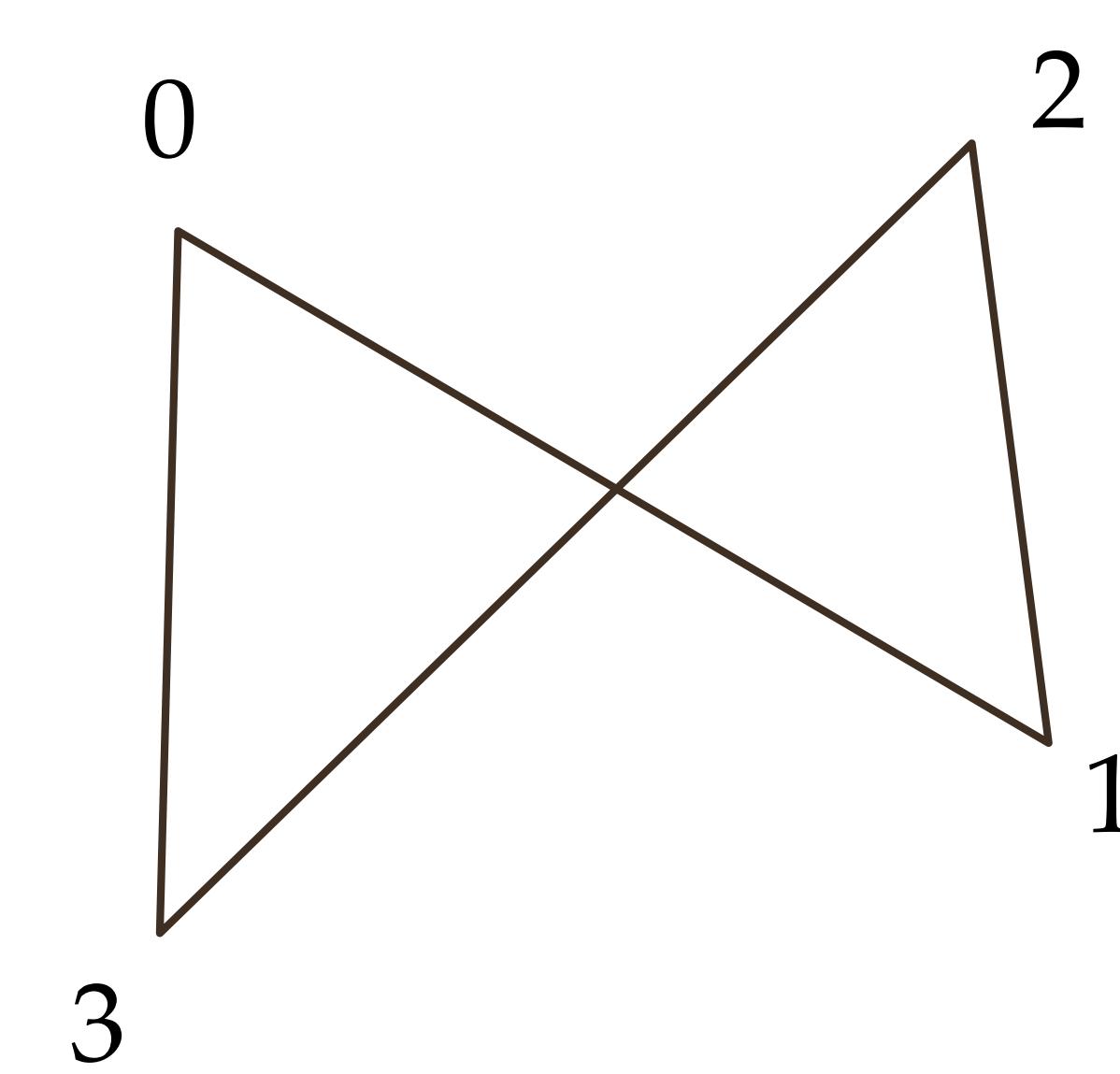
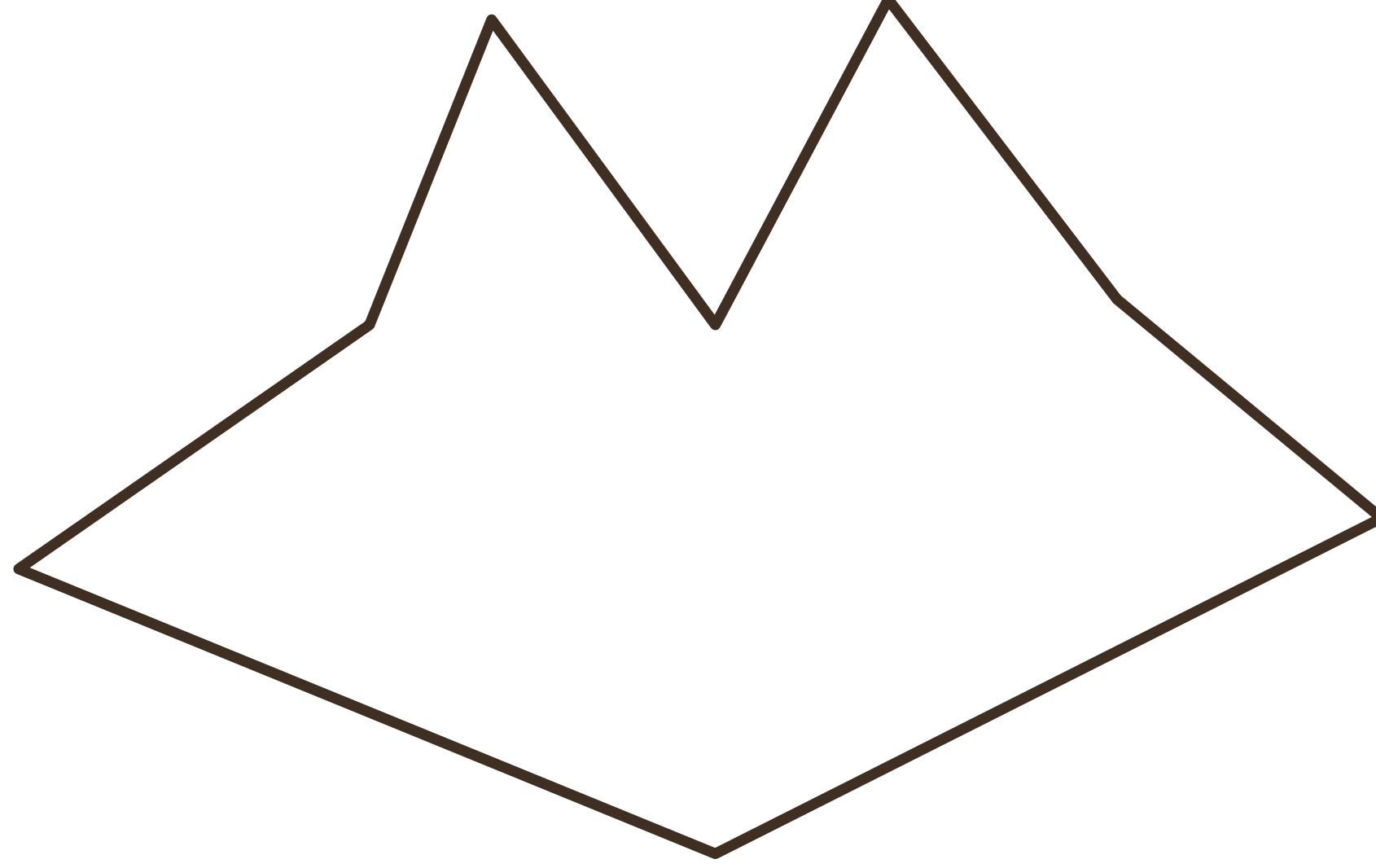
# For convex polygons...



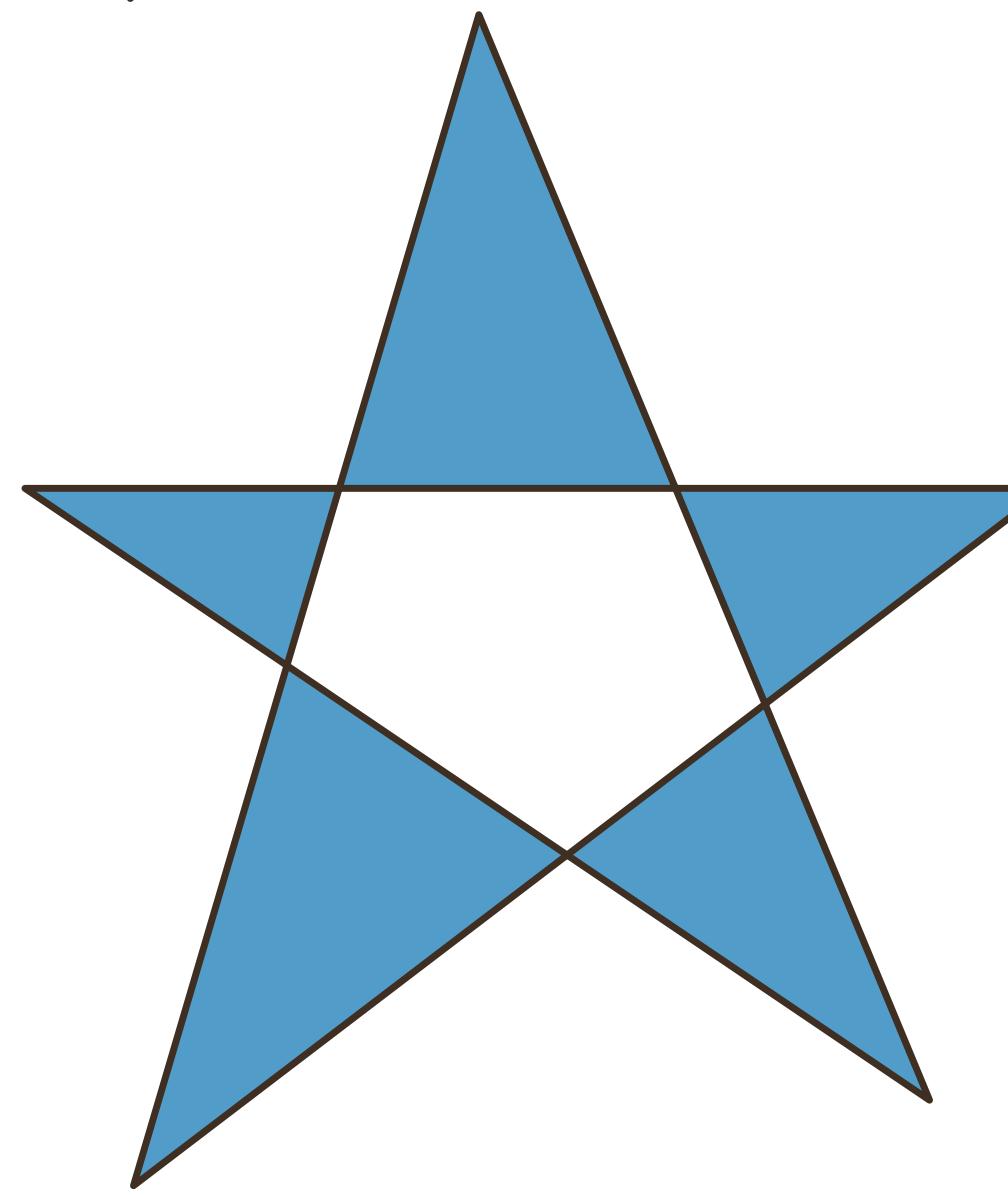
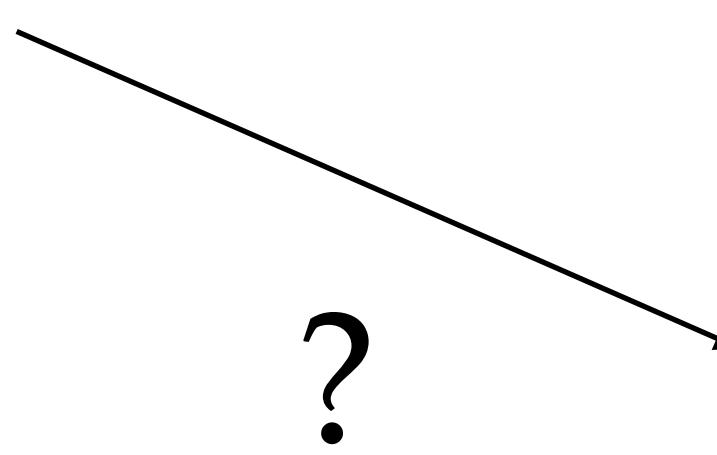
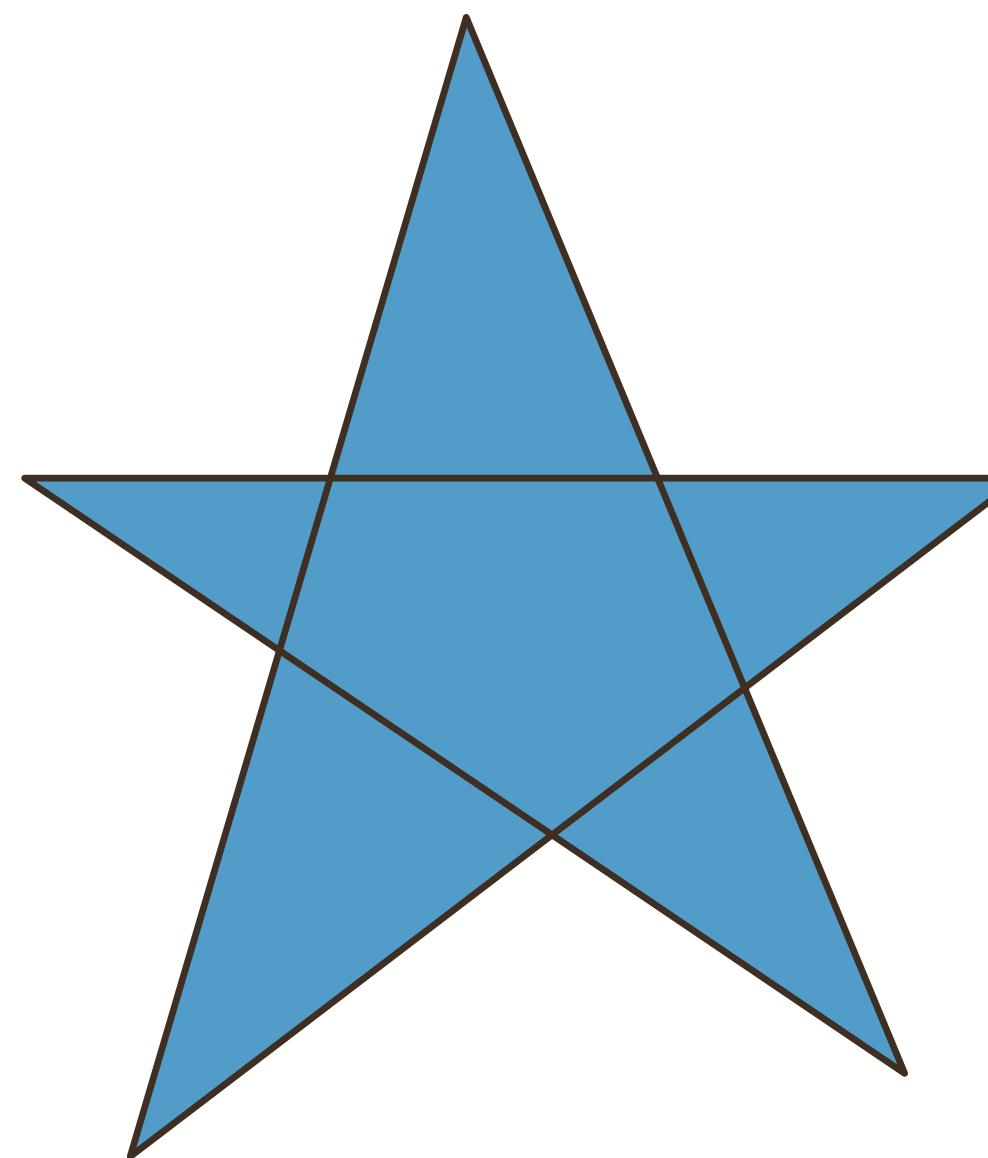
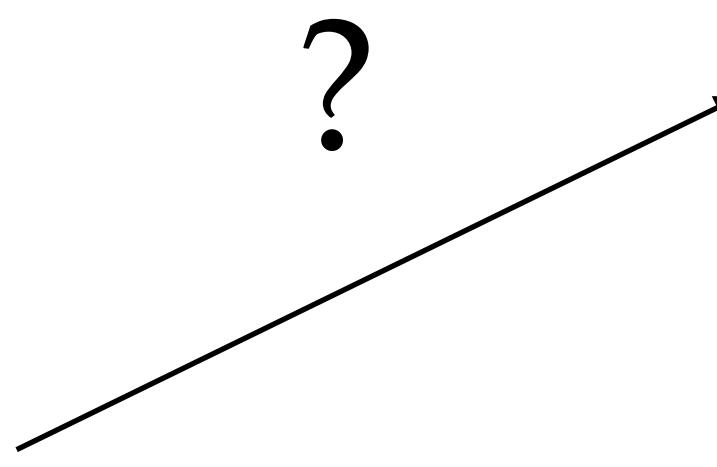
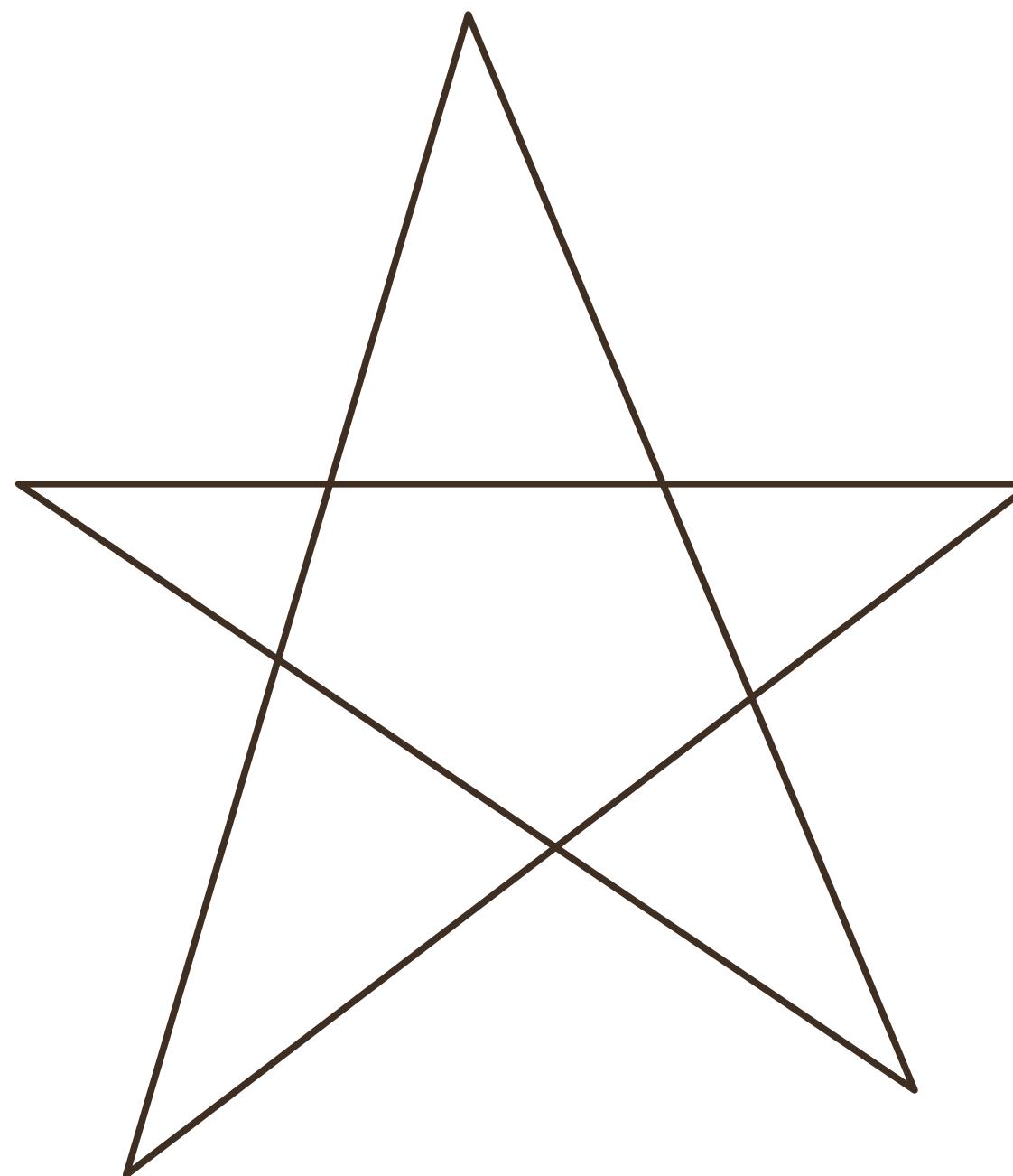
rays from points outside intersect with the polygon edges  
either zero times or two times

rays from points inside intersect with the polygon edges  
exactly one time

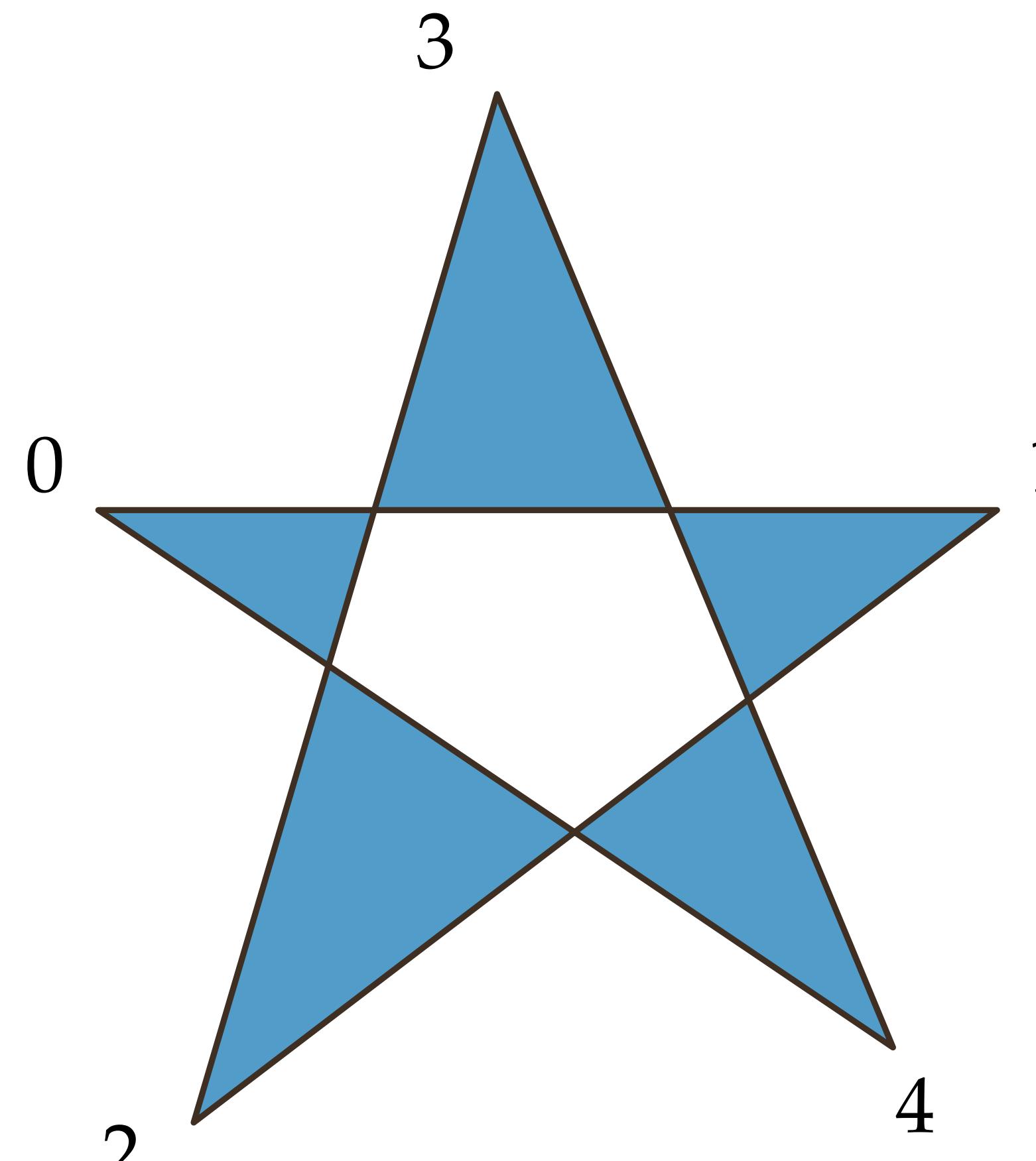
For general closed polylines, the rules are  
more complicated



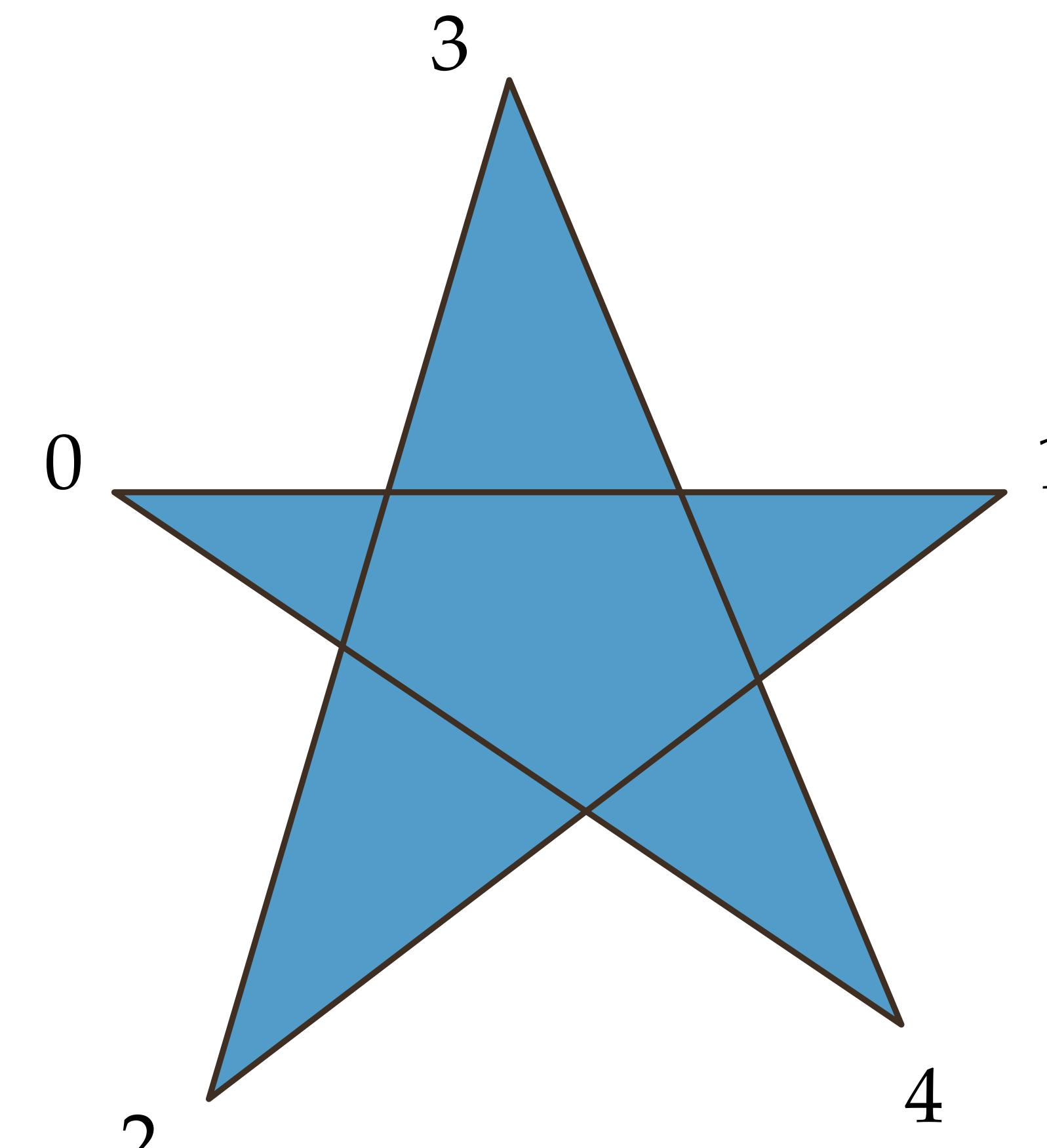
# Should we fill the center of the star?



# Two common different rules for point-in-polyline test

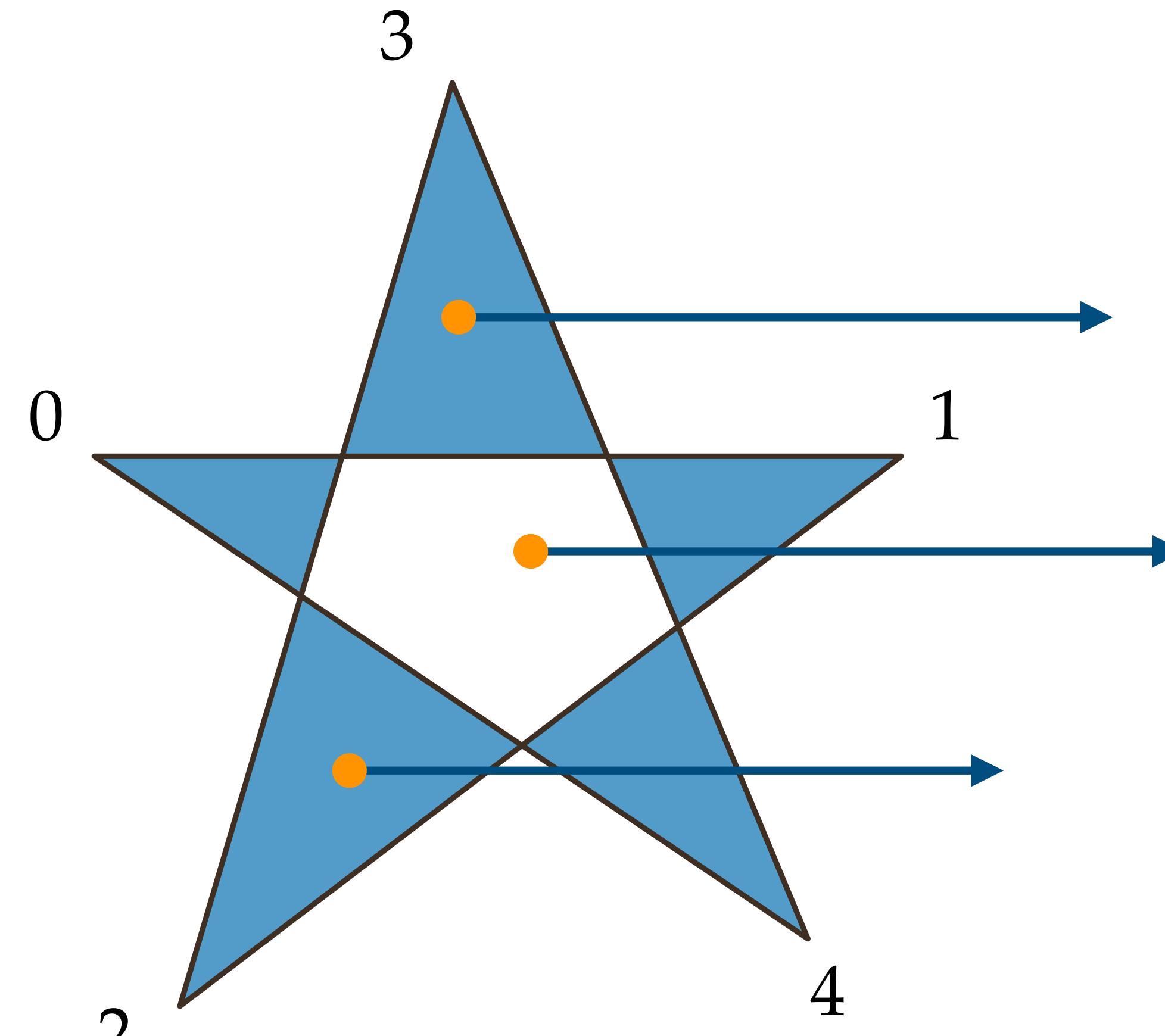


even-odd rule



nonzero rule

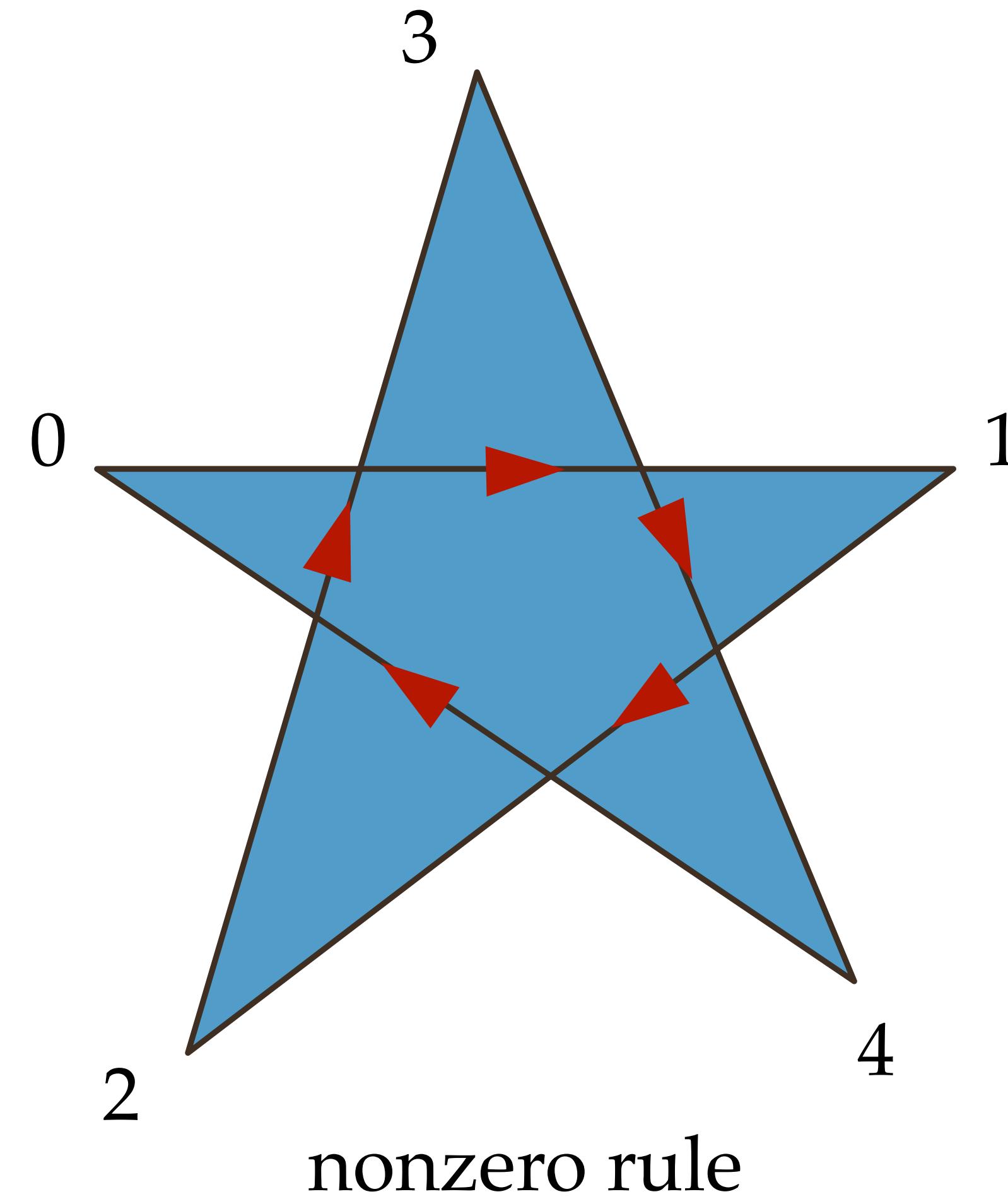
# Even-odd rule



if a ray intersect the shape **even** times,  
the origin is **outside** the shape

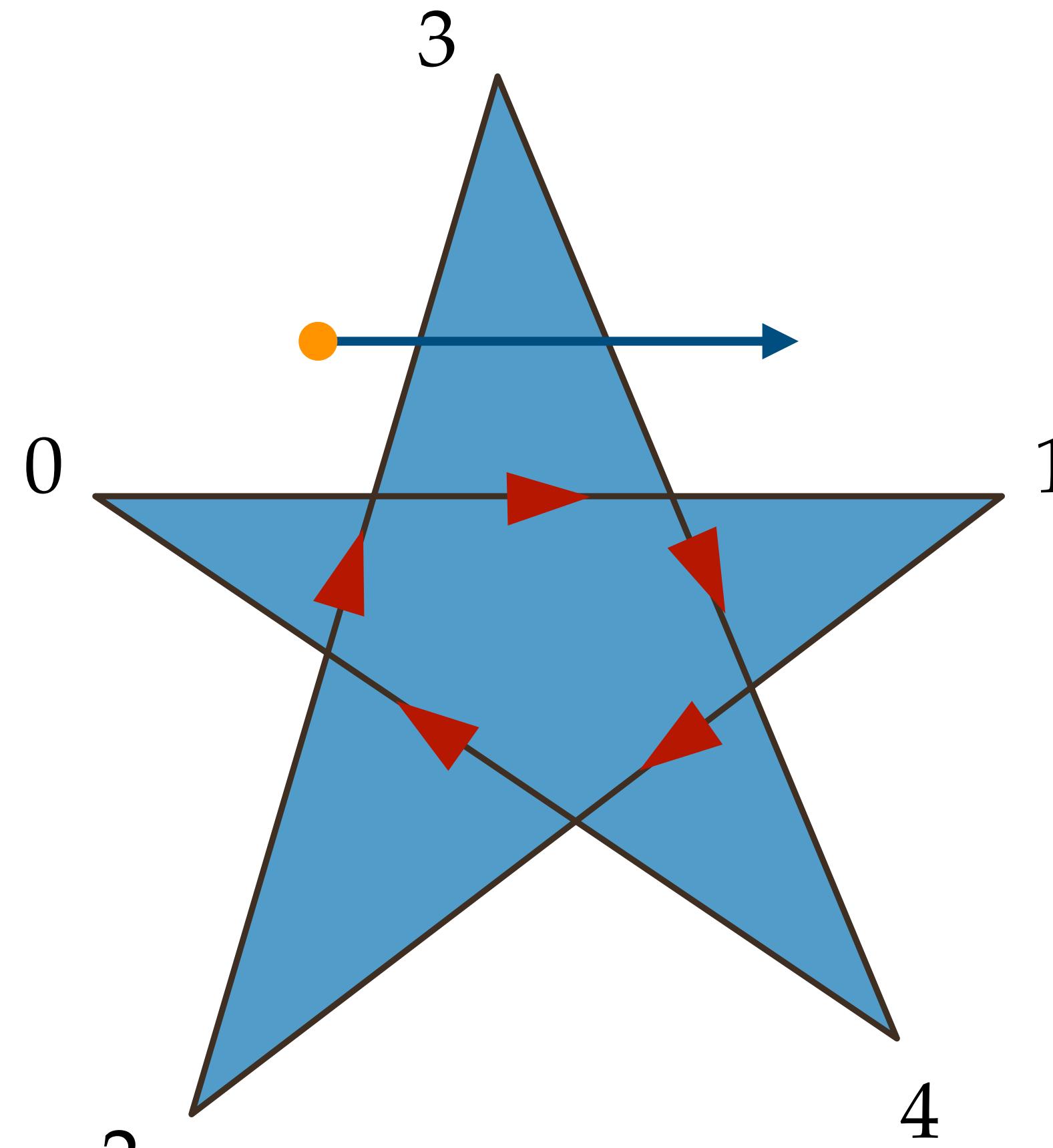
if a ray intersect the shape **odd** times,  
the origin is **inside** the shape

# Non-zero rule



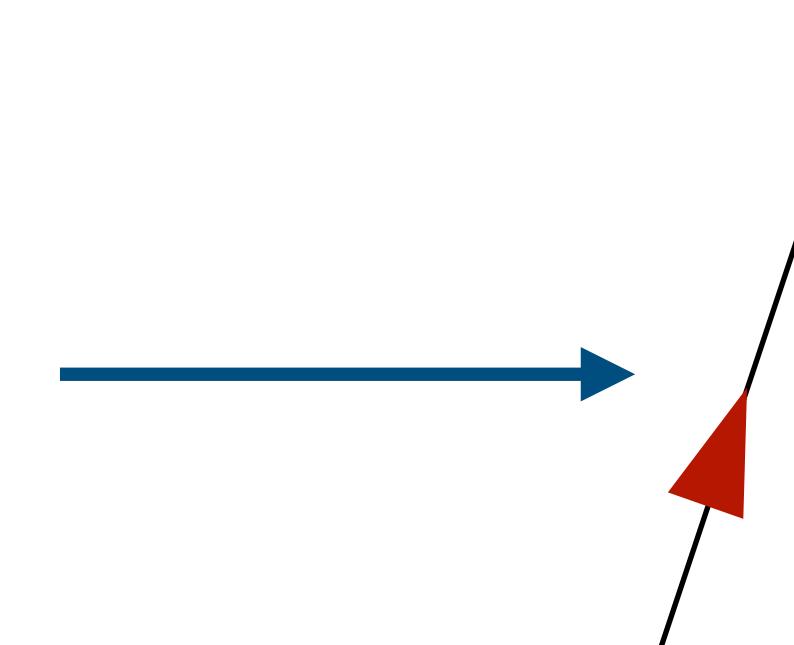
need to account for the orientation of the lines

# Non-zero rule



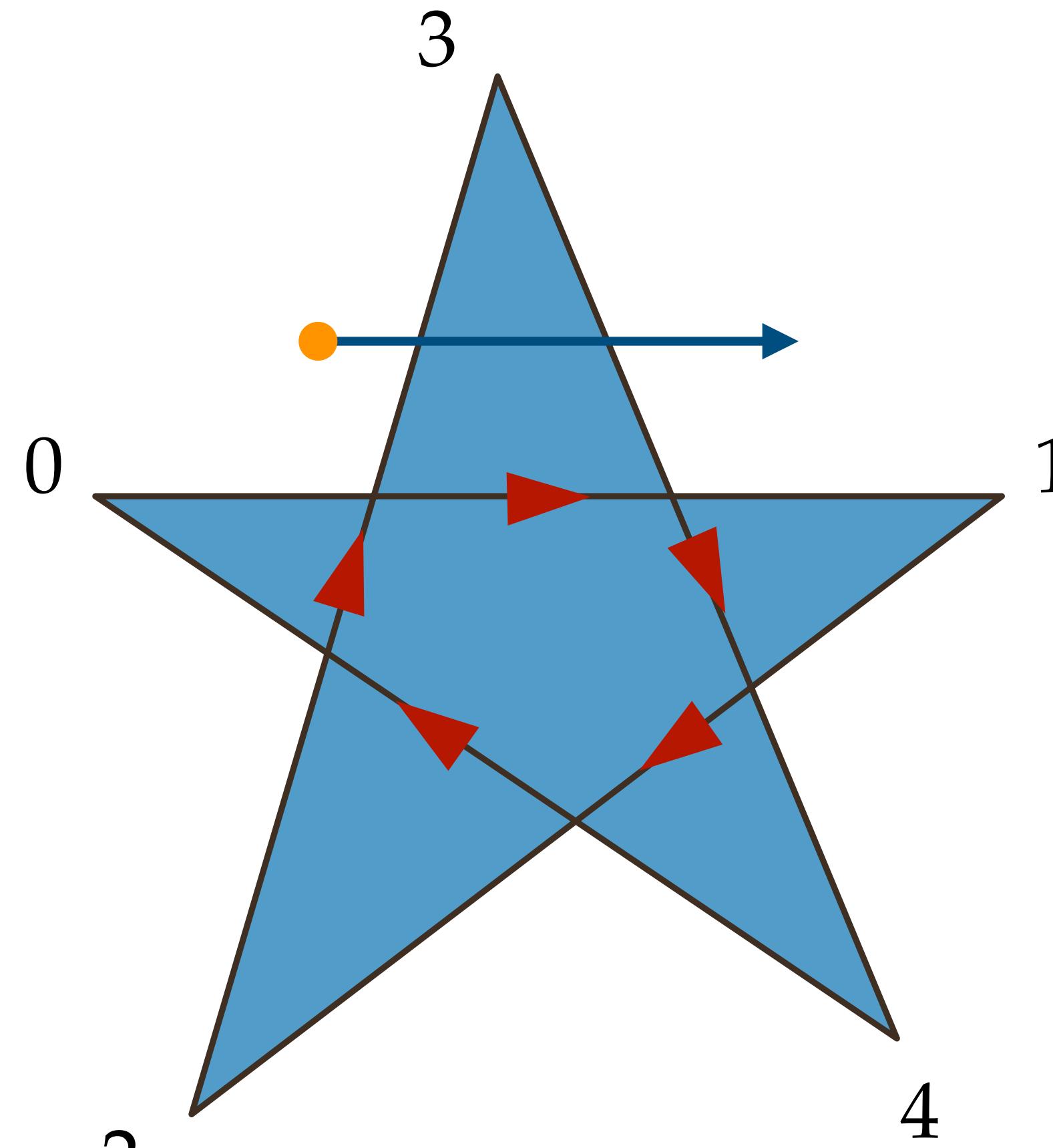
need to account for the orientation of the lines

every time a ray hits a line...



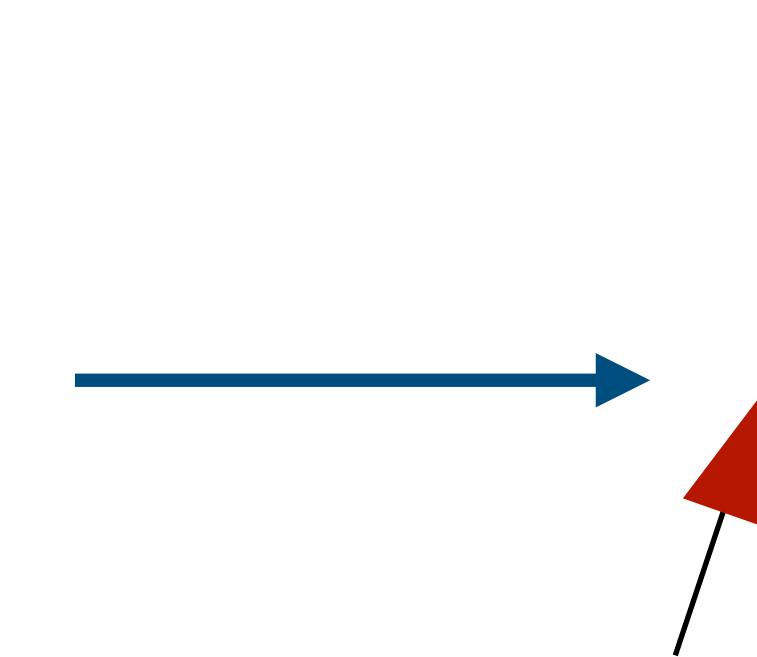
if the line is going up,  
count += 1

# Non-zero rule

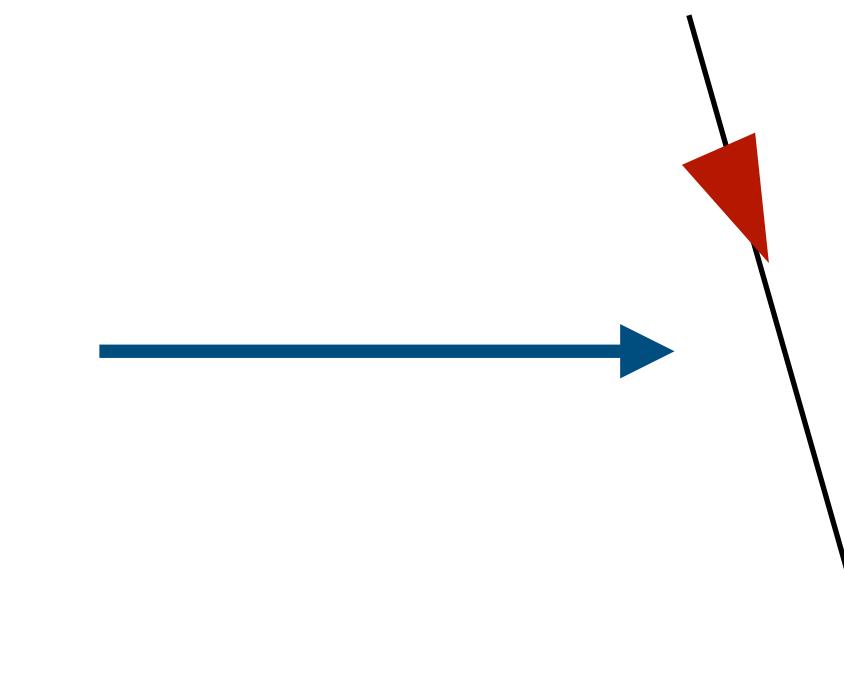


need to account for the orientation of the lines

every time a ray hits a line...

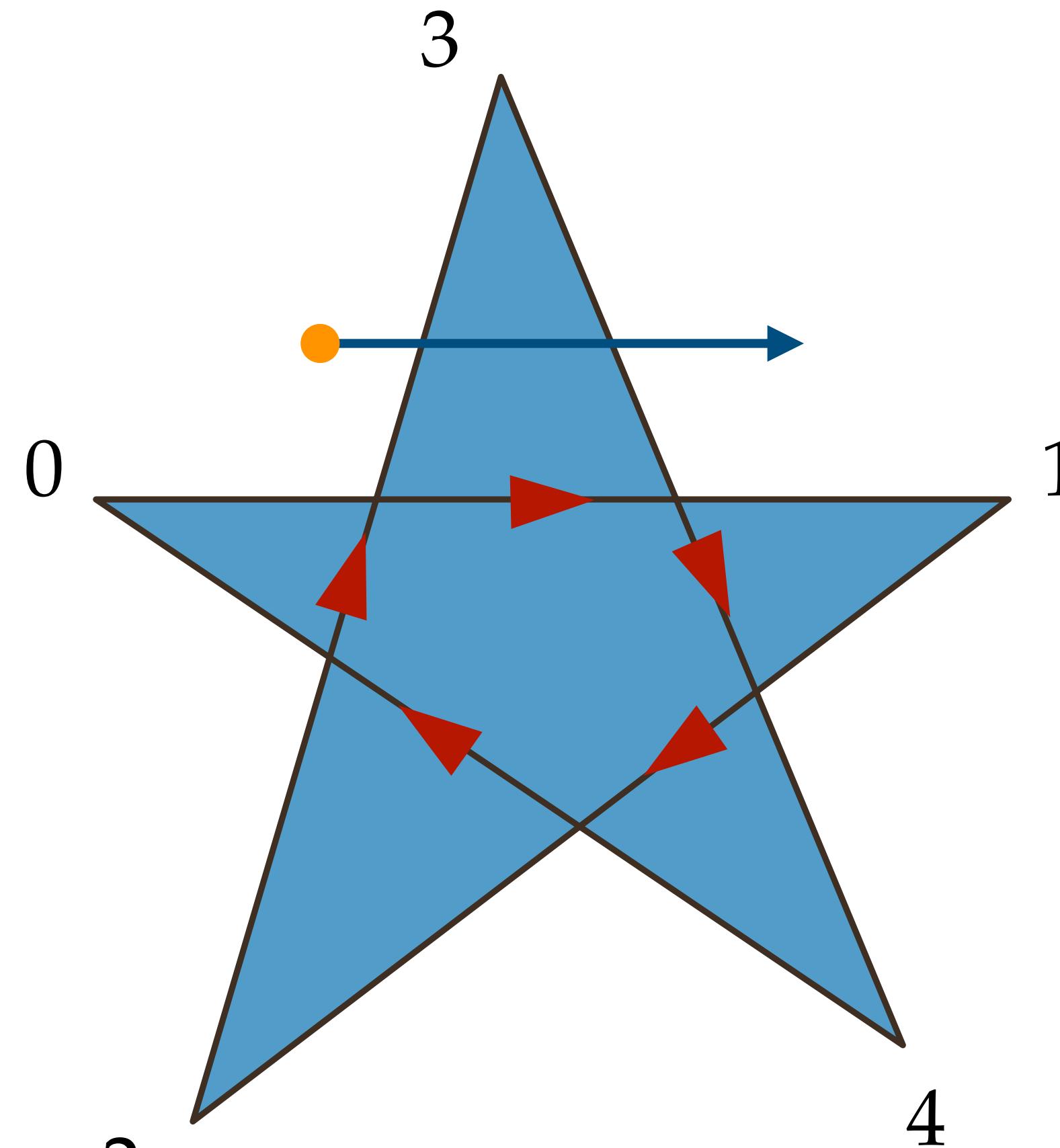


if the line is going up,  
count  $+= 1$



if the line is going down,  
count  $-= 1$

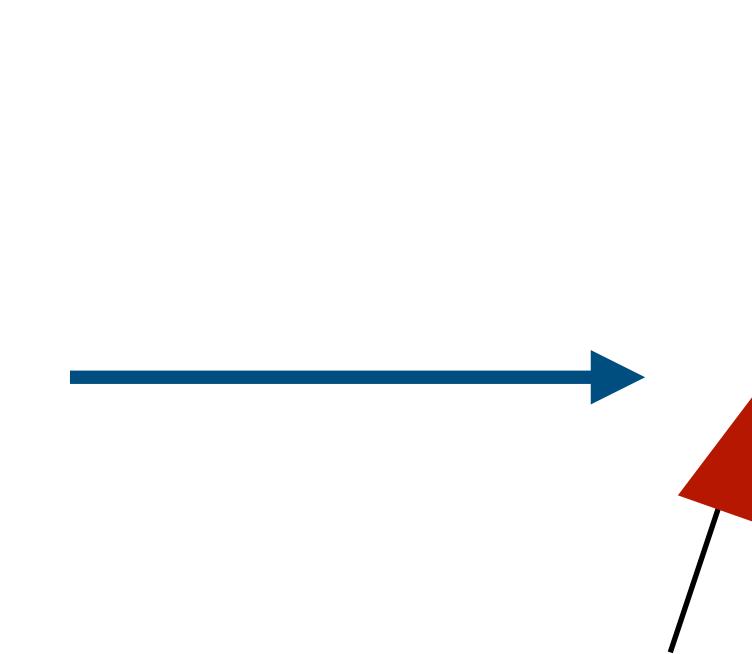
# Non-zero rule



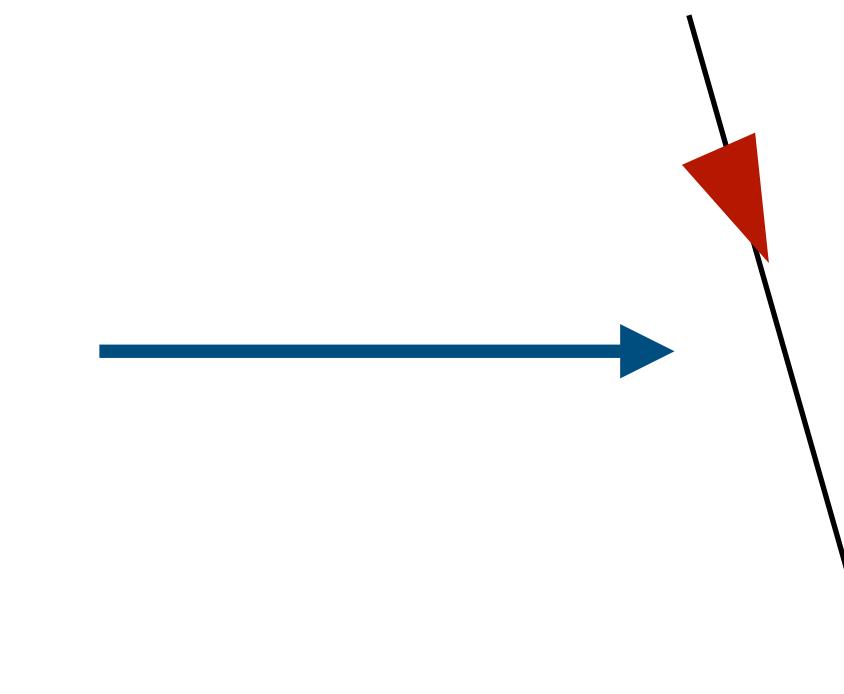
nonzero rule

need to account for the orientation of the lines

every time a ray hits a line...



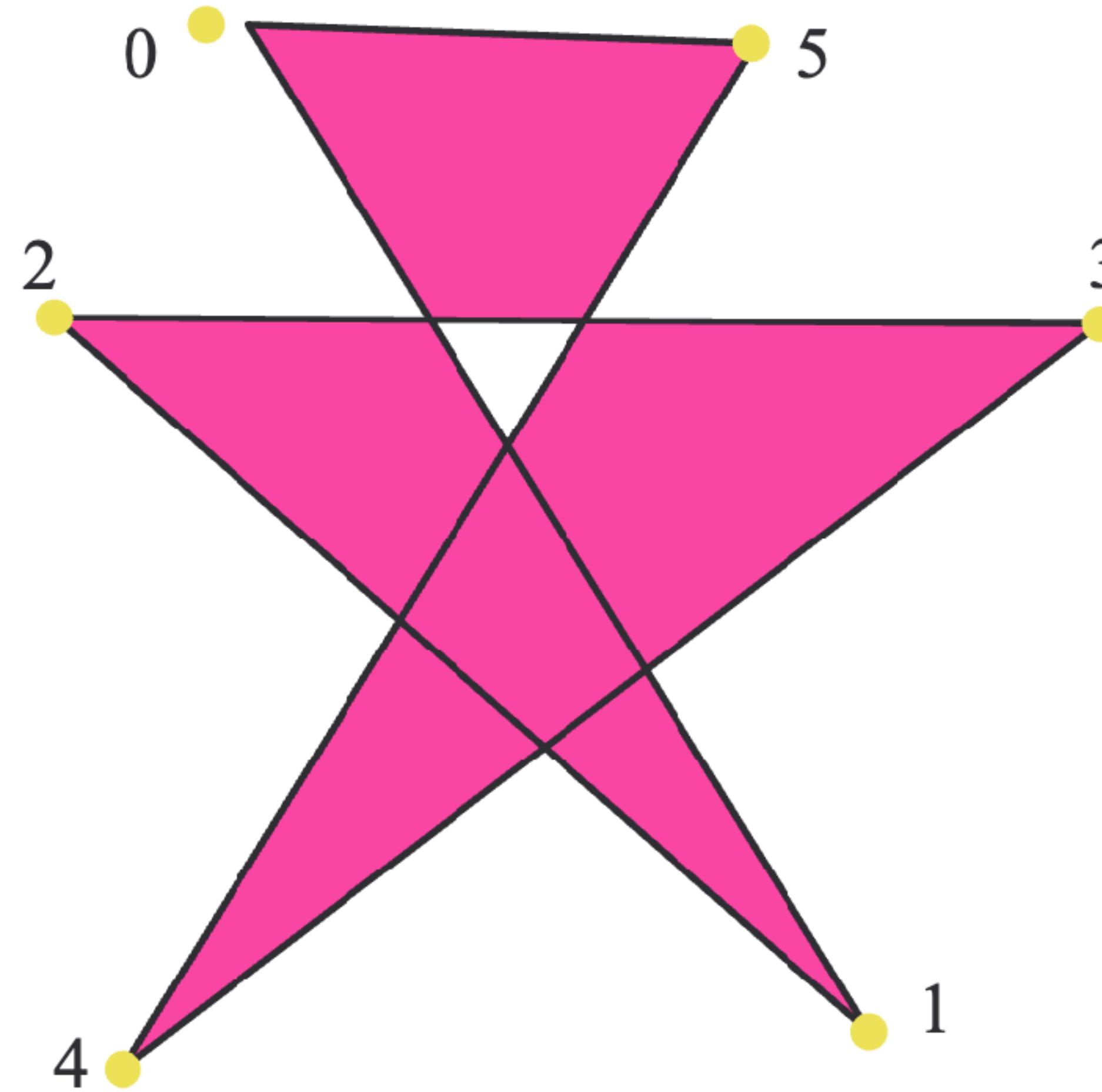
if the line is going up,  
count += 1



if the line is going down,  
count -= 1

the point is inside if count != 0  
(this count is called “winding number” in math)

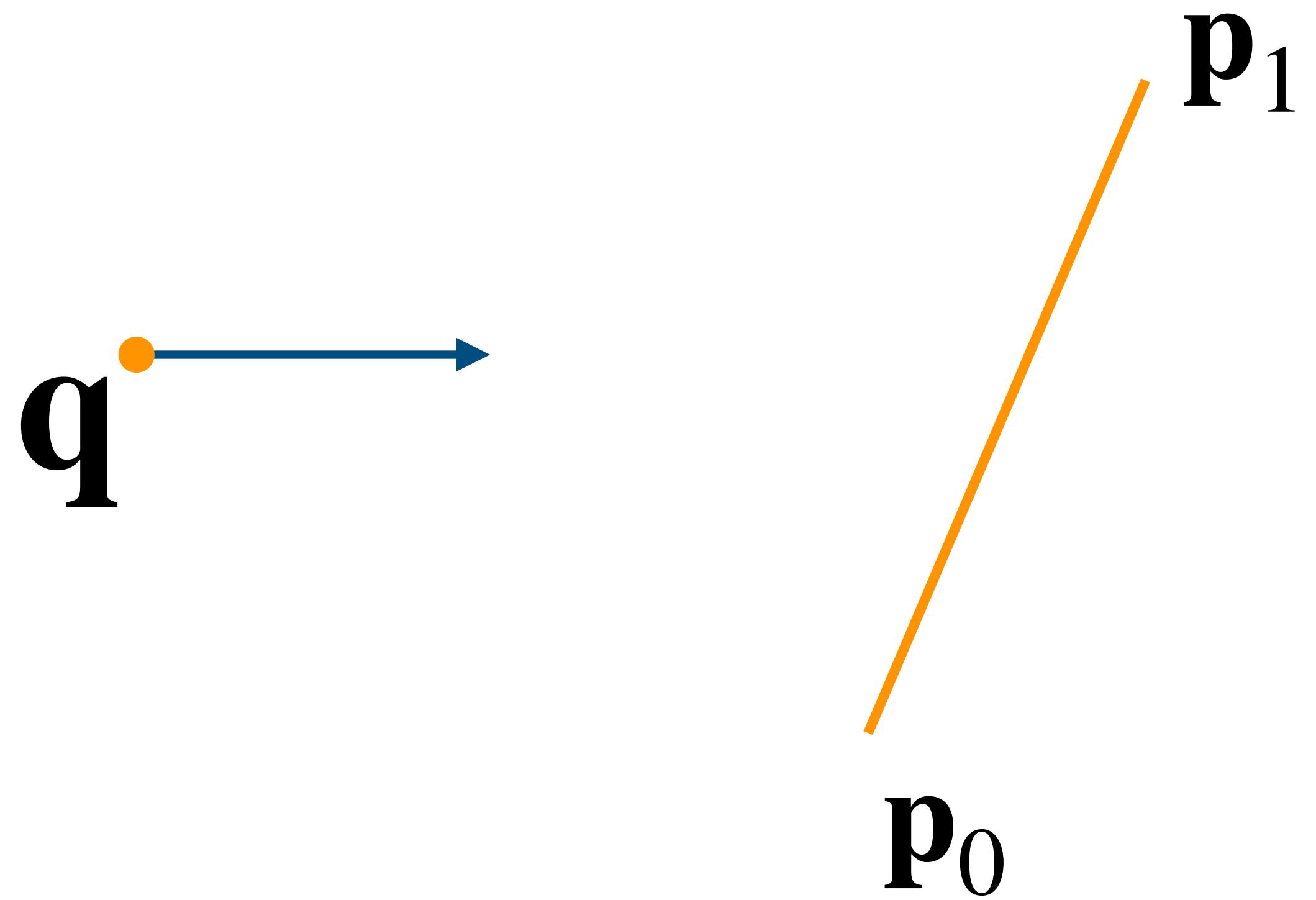
In homework, you need to guess  
which rule we used!



# More resources

- [https://en.wikipedia.org/wiki/Point\\_in\\_polygon](https://en.wikipedia.org/wiki/Point_in_polygon)
- [https://en.wikipedia.org/wiki/Even%20odd\\_rule](https://en.wikipedia.org/wiki/Even%20odd_rule)
- <https://en.wikipedia.org/wiki/Nonzero-rule>

# Ray-line intersection



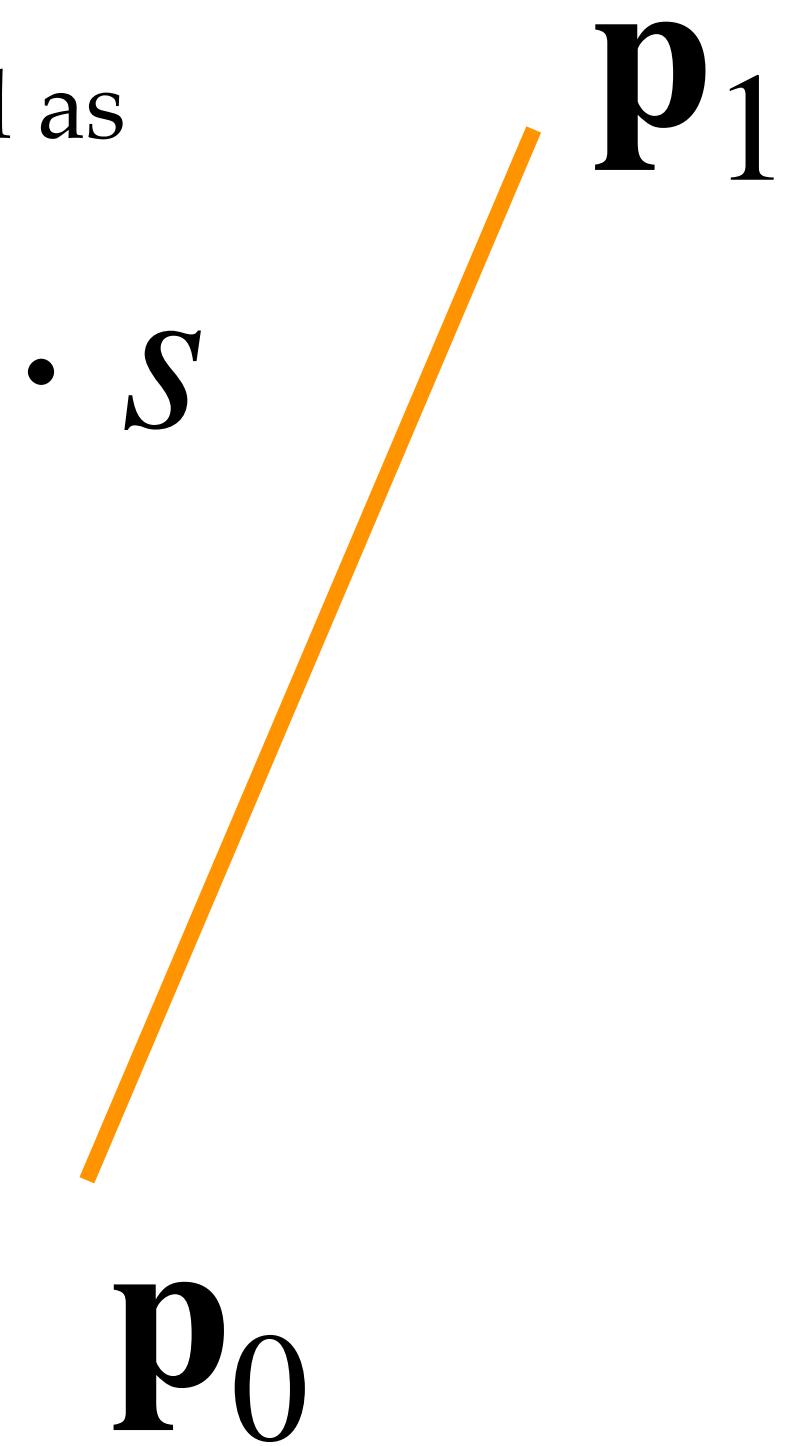
# Ray-line intersection

every point on the ray can be represented as

$$\mathbf{q} + (1,0) \cdot s$$



for  $s \geq 0$



# Ray-line intersection

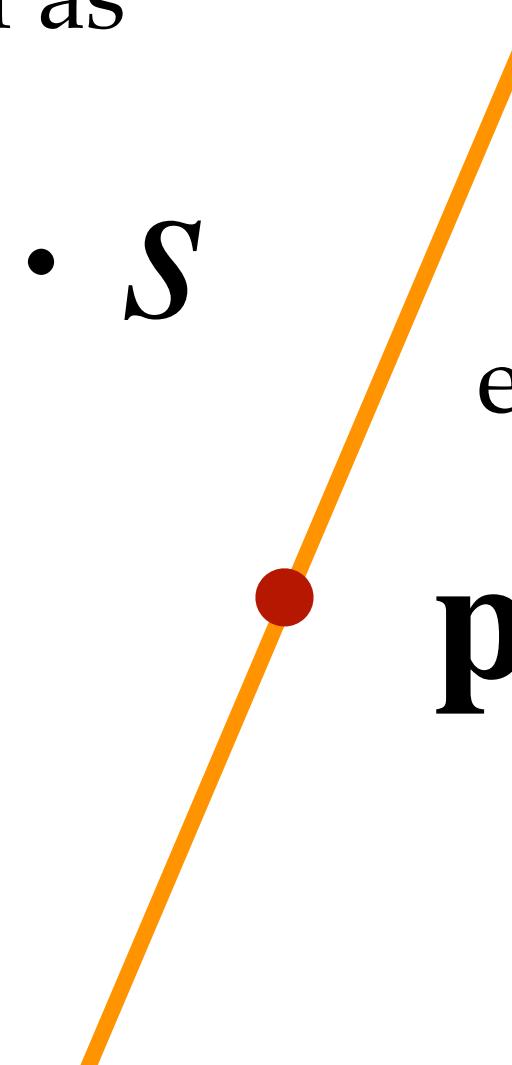
every point on the ray can be represented as

$$\mathbf{q} + (1,0) \cdot s$$

  
for  $s \geq 0$

every point on the line can be represented as

$$\mathbf{p}_0 + t \cdot (\mathbf{p}_1 - \mathbf{p}_0)$$

  
 $\mathbf{p}_0$        $\mathbf{p}_1$   
for  $0 \leq t \leq 1$

# Ray-line intersection

$$s \geq 0$$

$$0 \leq t \leq 1$$

$$\mathbf{q} + (1,0) \cdot s = \mathbf{p}_0 + t \cdot (\mathbf{p}_1 - \mathbf{p}_0)$$

# Ray-line intersection

$$s \geq 0$$

$$0 \leq t \leq 1$$

$$\mathbf{q}_x + (1,0) \cdot s = \mathbf{p}_{0x} + t \cdot \left( \mathbf{p}_{1x} - \mathbf{p}_{0x} \right)$$

$$\mathbf{q}_y + (1,0) \cdot s = \mathbf{p}_{0y} + t \cdot \left( \mathbf{p}_{1y} - \mathbf{p}_{0y} \right)$$

# Ray-line intersection

$$s \geq 0$$

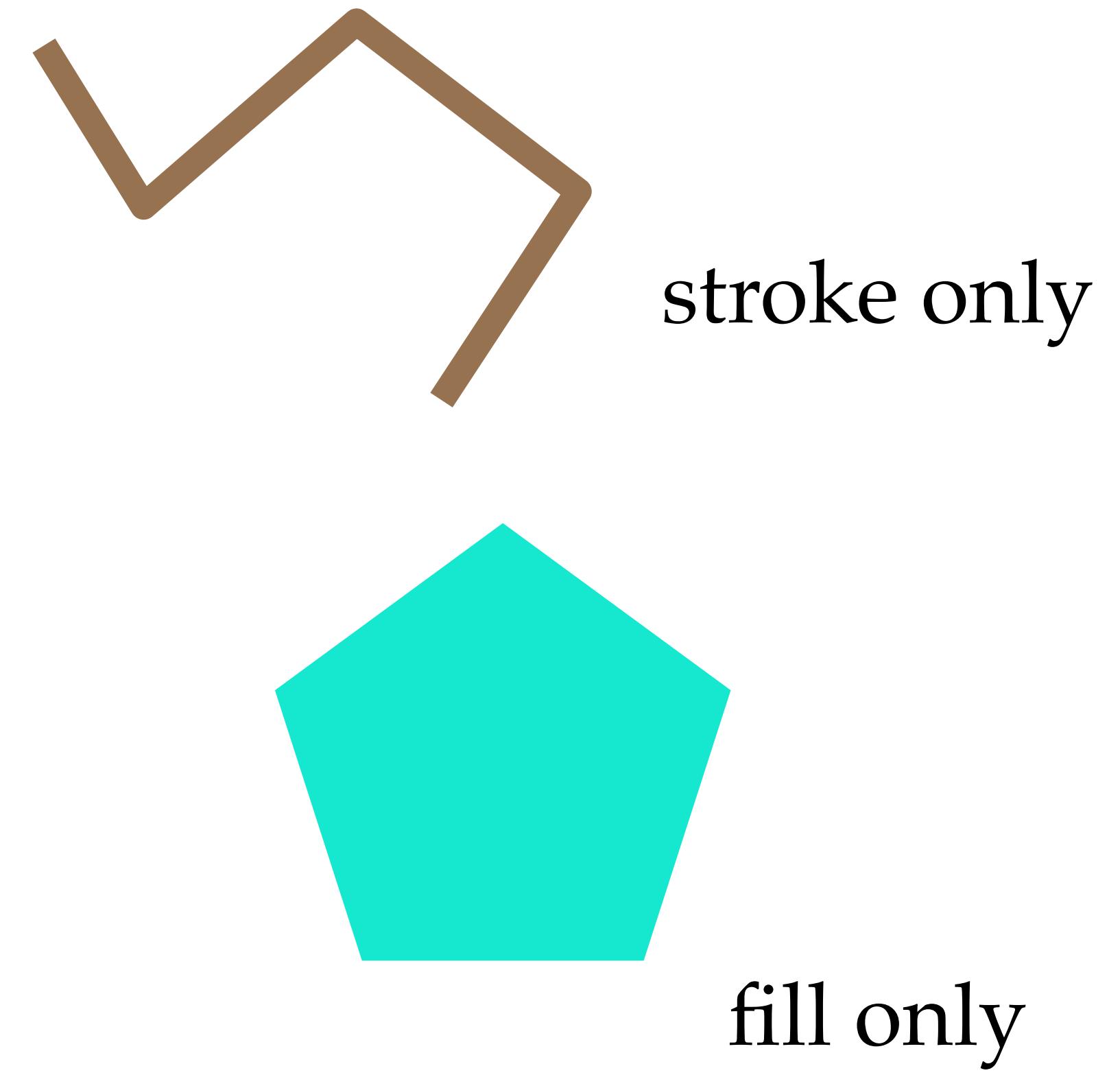
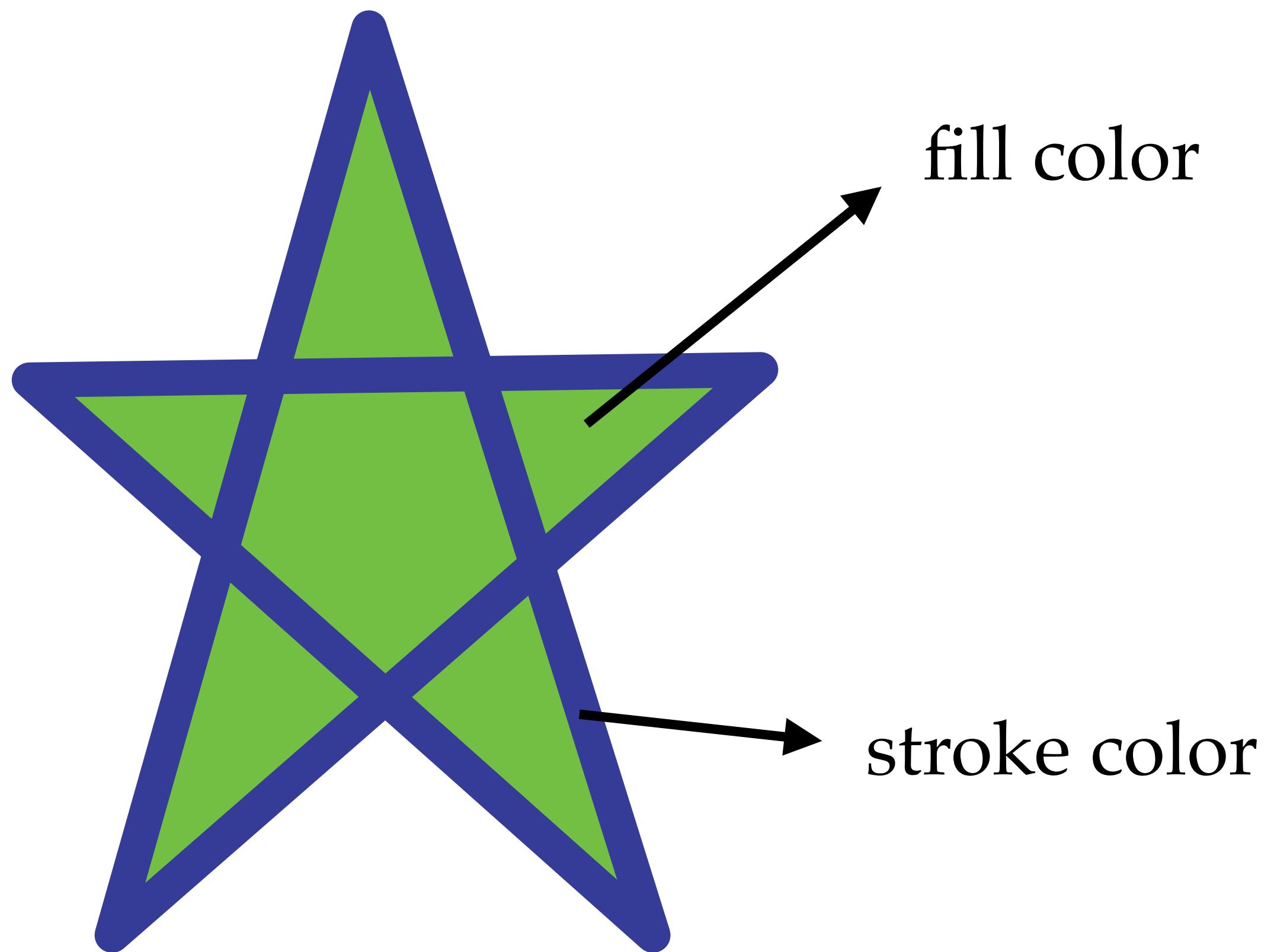
$$0 \leq t \leq 1$$

$$\mathbf{q}_x + (1,0) \cdot s = \mathbf{p}_{0x} + t \cdot (\mathbf{p}_{1x} - \mathbf{p}_{0x})$$

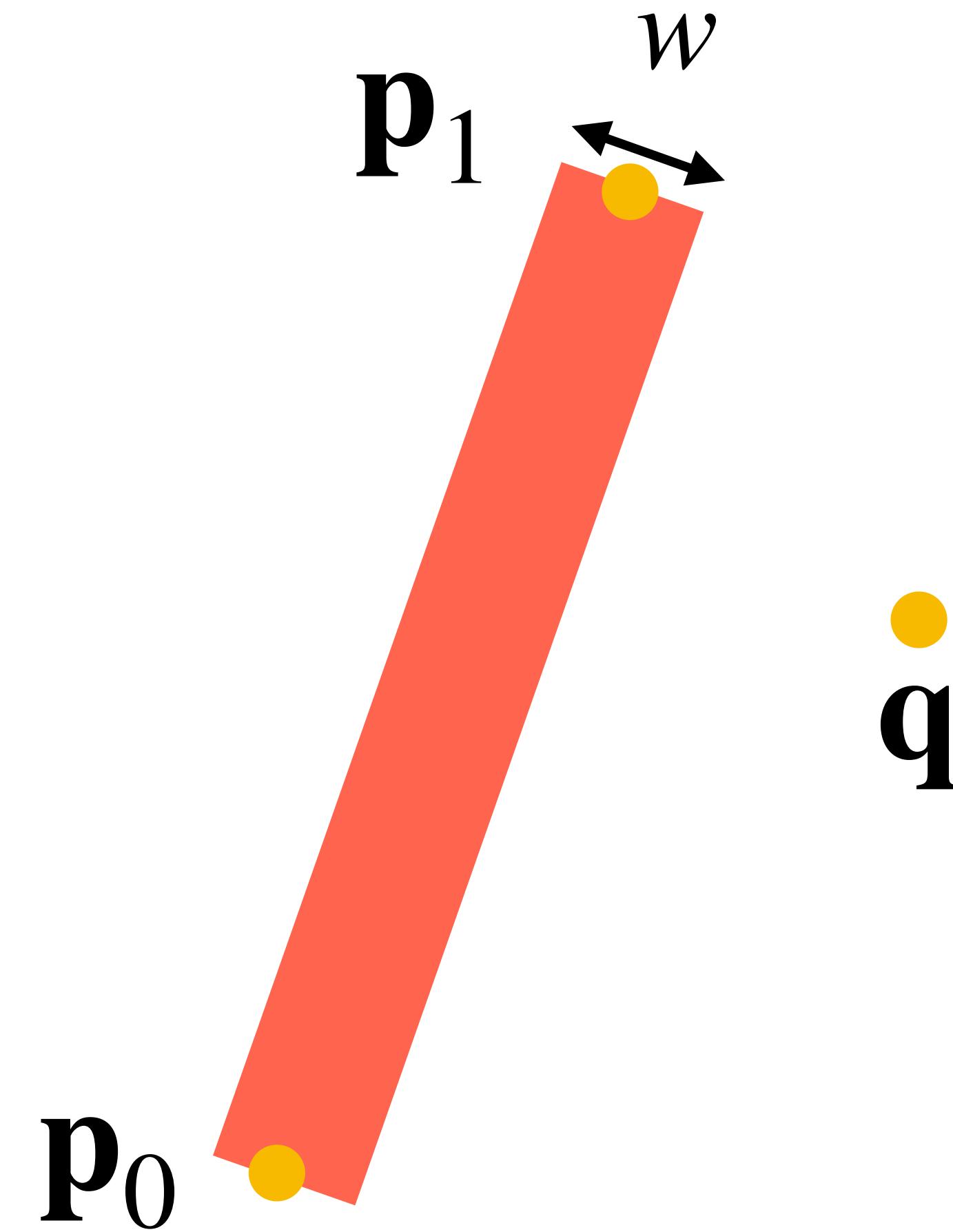
$$\mathbf{q}_y + (1,0) \cdot s = \mathbf{p}_{0y} + t \cdot (\mathbf{p}_{1y} - \mathbf{p}_{0y})$$

solve for s and t, if  $s \geq 0$  and  $0 \leq t \leq 1$ , then the ray intersects with the line

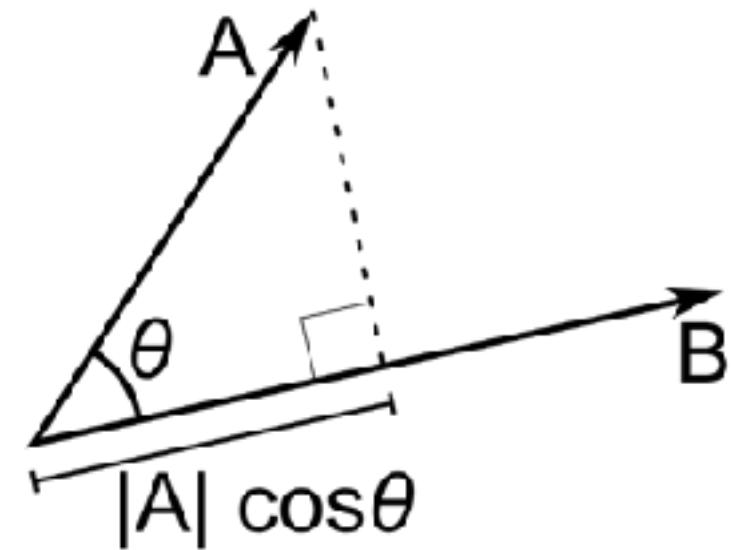
# What about “stroke color”?



Goal: determine whether the point  
is within a width of the line



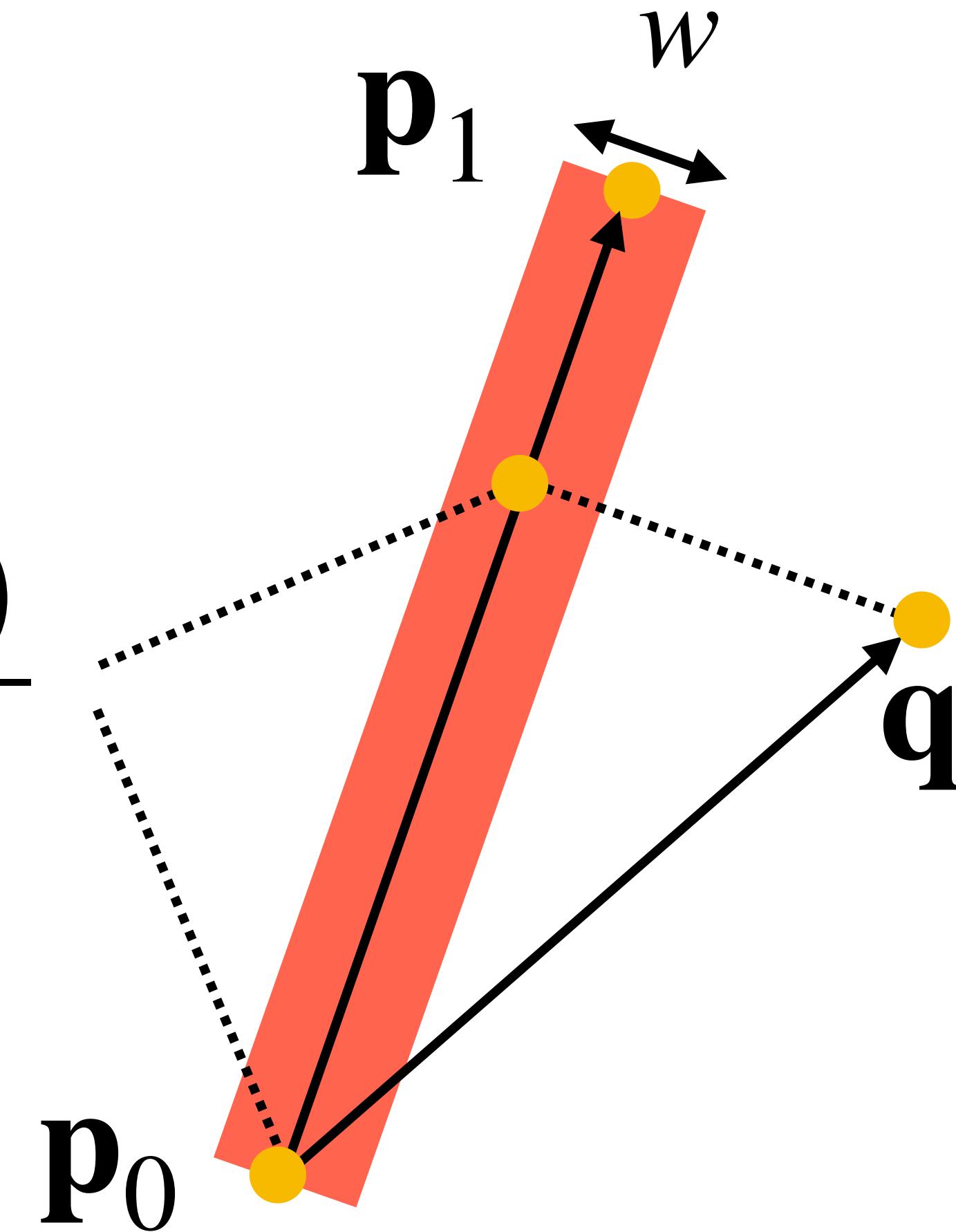
# Point-in-line test



dot product = projected length \* length of the projected vector

$$A \cdot B = \| A \| \| B \| \cos \theta$$

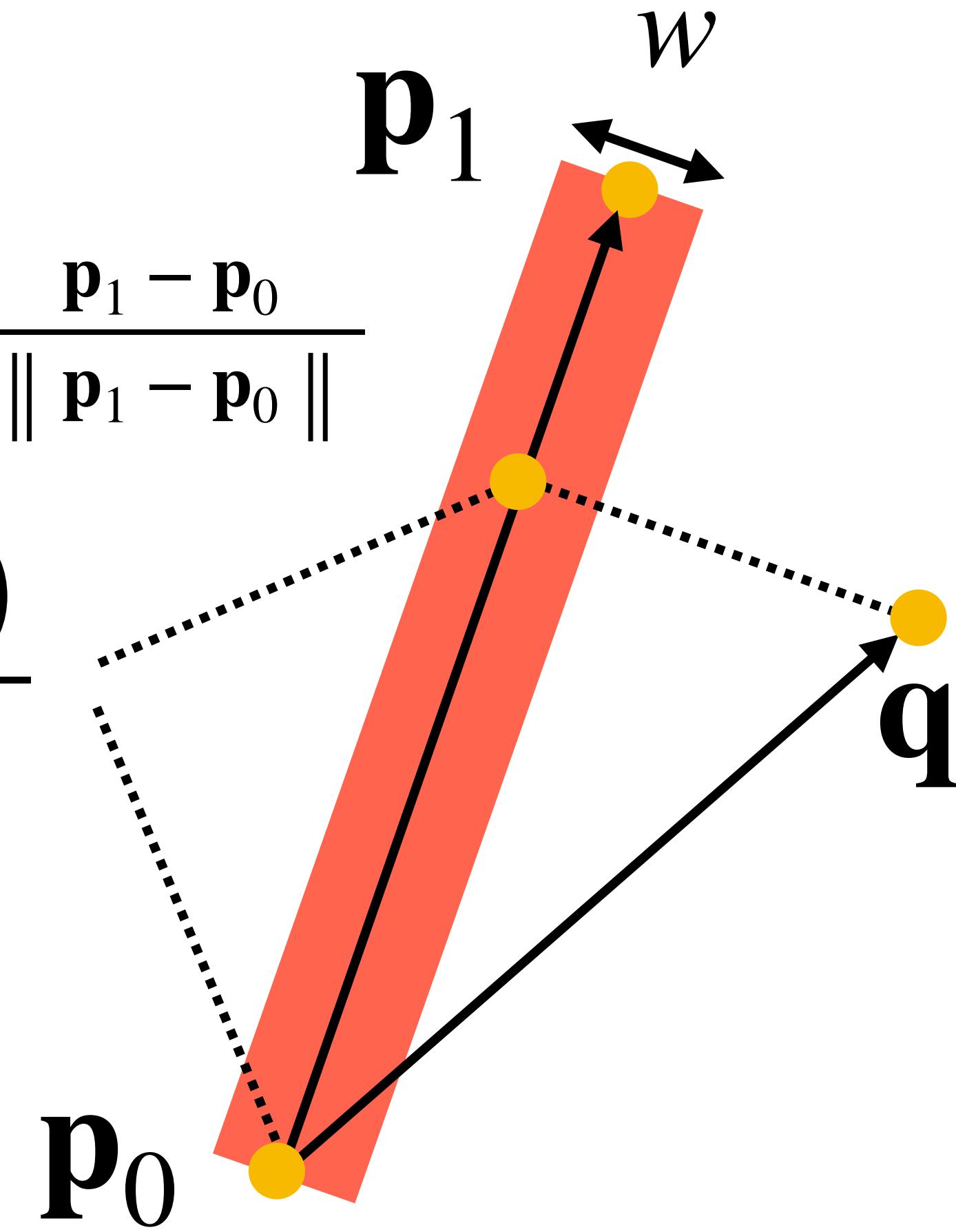
$$l = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \cdot (\mathbf{q} - \mathbf{p}_0)}{\| \mathbf{p}_1 - \mathbf{p}_0 \|}$$



# Point-in-line test

$$\mathbf{q}' = \mathbf{p}_0 + l \cdot \frac{\mathbf{p}_1 - \mathbf{p}_0}{\| \mathbf{p}_1 - \mathbf{p}_0 \|}$$

$$l = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \cdot (\mathbf{q} - \mathbf{p}_0)}{\| \mathbf{p}_1 - \mathbf{p}_0 \|}$$

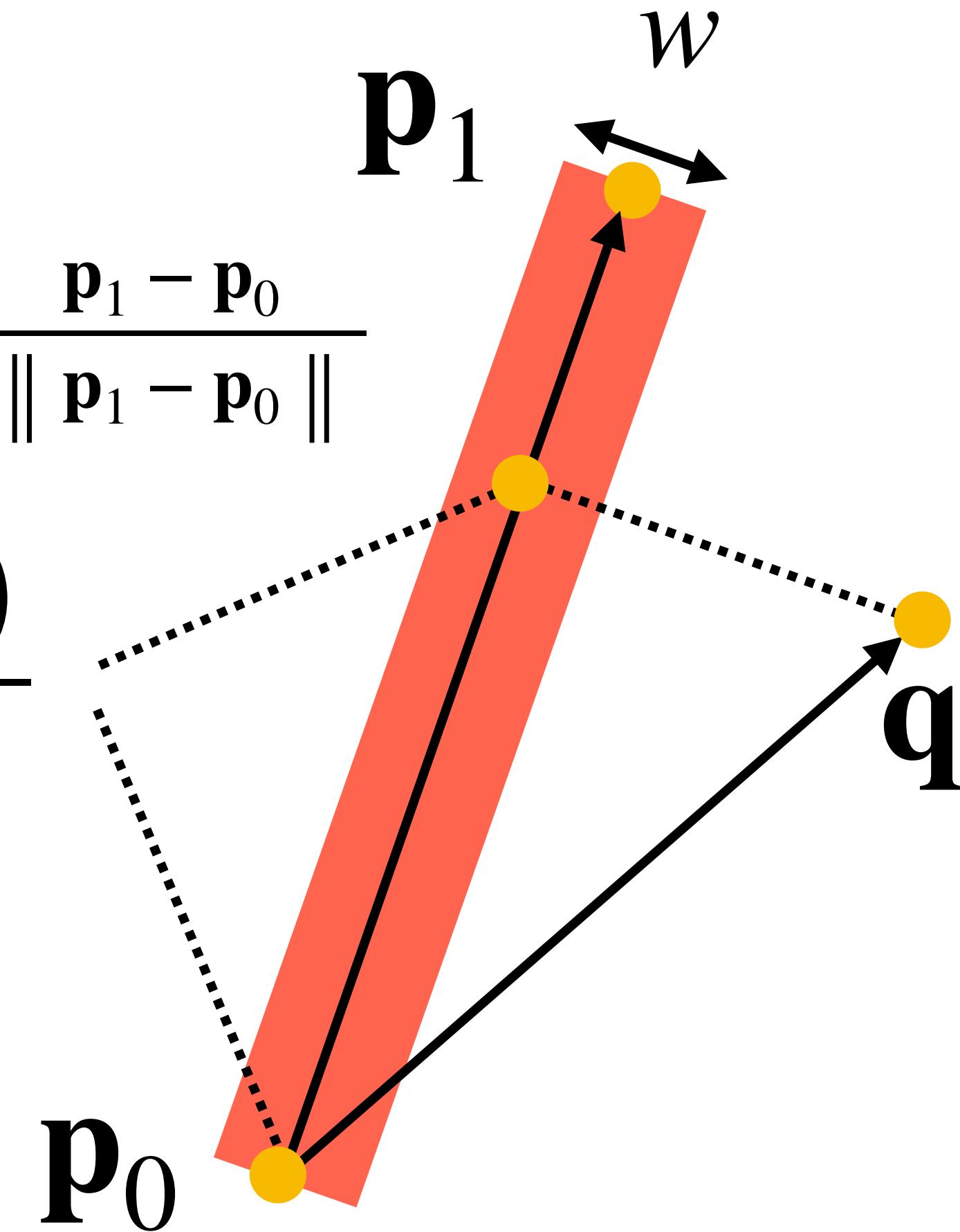


# Point-in-line test

$$\mathbf{q}' = \mathbf{p}_0 + l \cdot \frac{\mathbf{p}_1 - \mathbf{p}_0}{\| \mathbf{p}_1 - \mathbf{p}_0 \|}$$

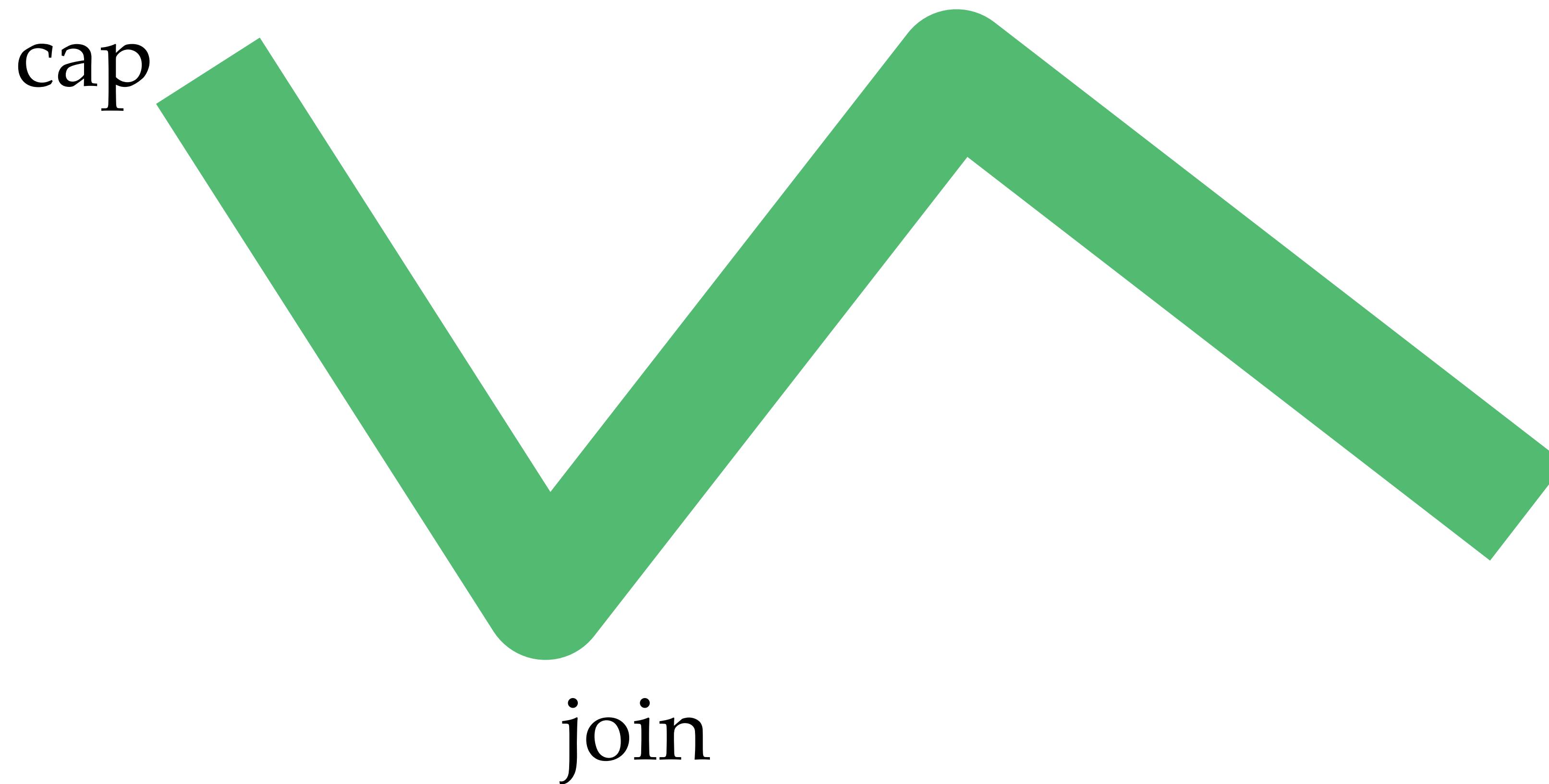
$$l = \frac{(\mathbf{p}_1 - \mathbf{p}_0) \cdot (\mathbf{q} - \mathbf{p}_0)}{\| \mathbf{p}_1 - \mathbf{p}_0 \|}$$

if  $l > 0$ ,  $l < \| \mathbf{p}_1 - \mathbf{p}_0 \|$ , and  
 $\| \mathbf{q} - \mathbf{q}' \| < \frac{w}{2}$ , then  $\mathbf{q}$  is in the line

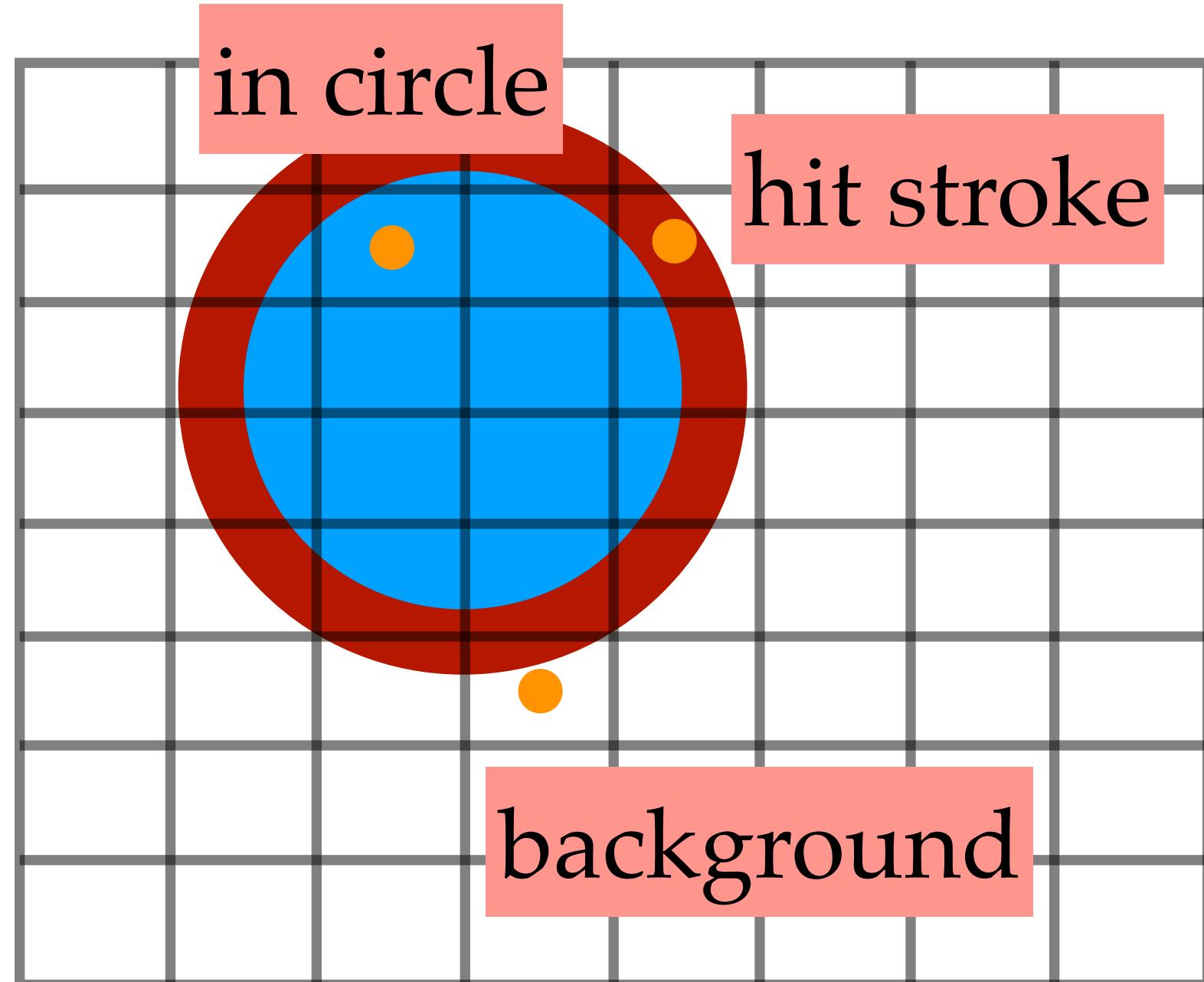


# “Joins” and “caps” of lines

in homework, we ask you to have flat caps, and round joins  
— try to figure out how to do it!

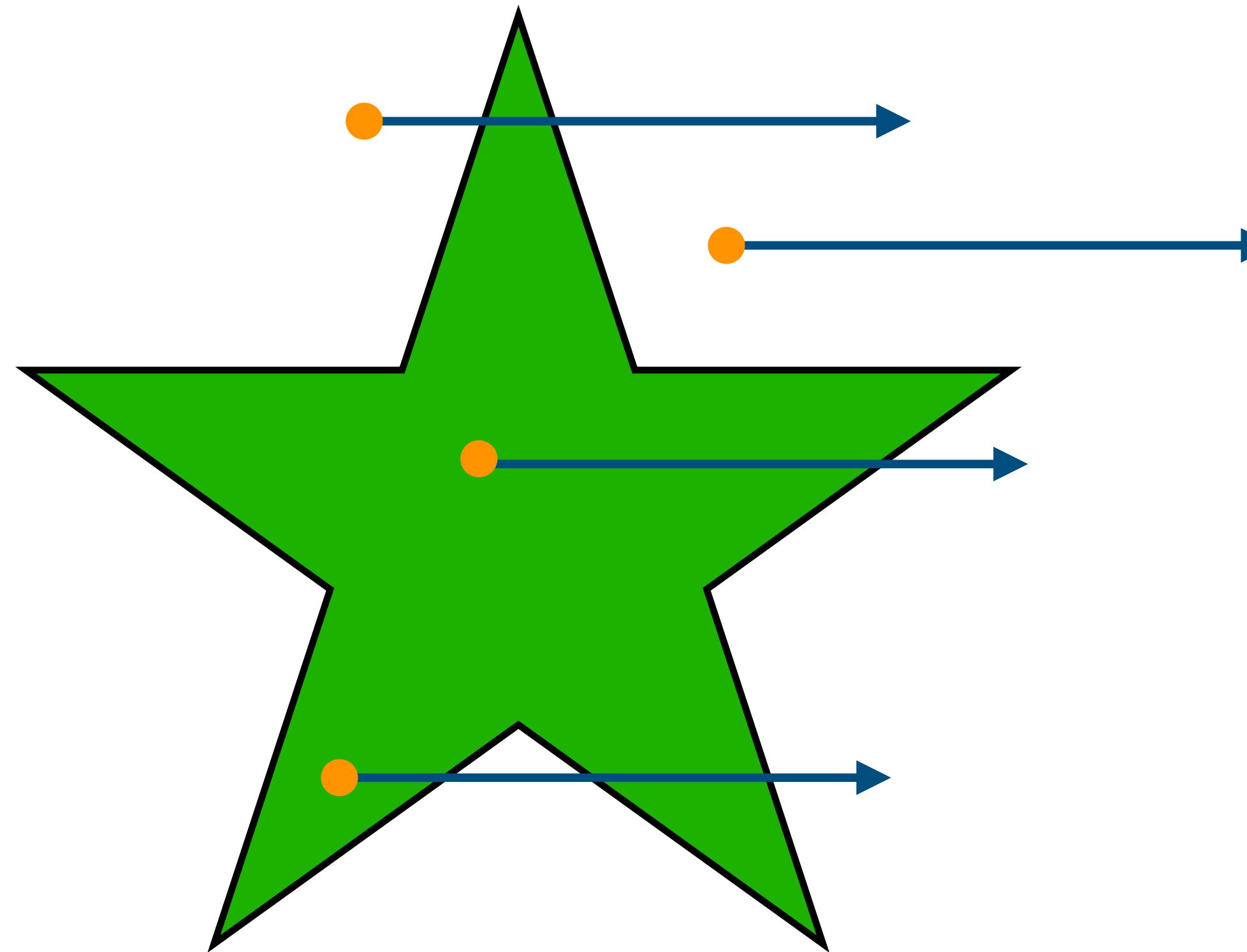


# Recap



```
for each pixel
    if pixel center hits the stroke:
        color = primitive.stroke_color
    else if pixel center is inside primitive:
        color = primitive.fill_color
    else:
        color = background
```

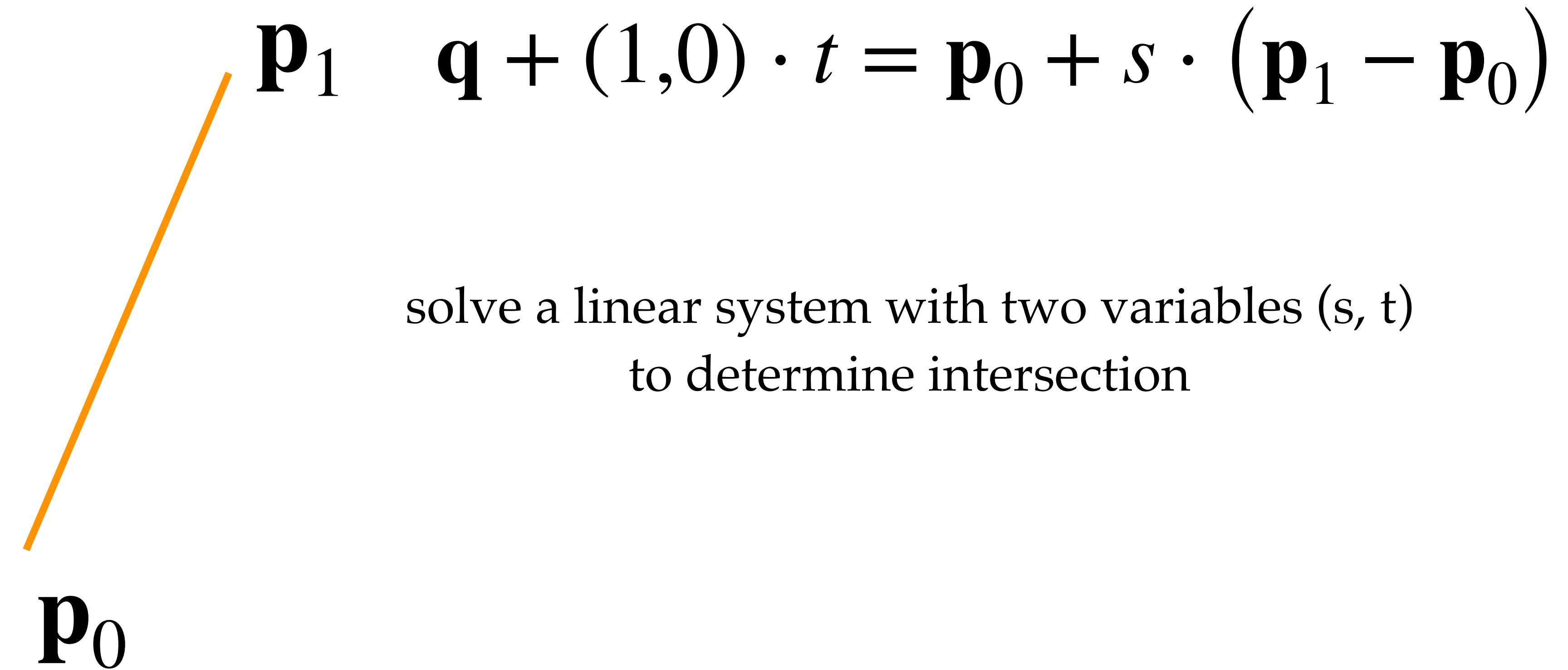
# Recap



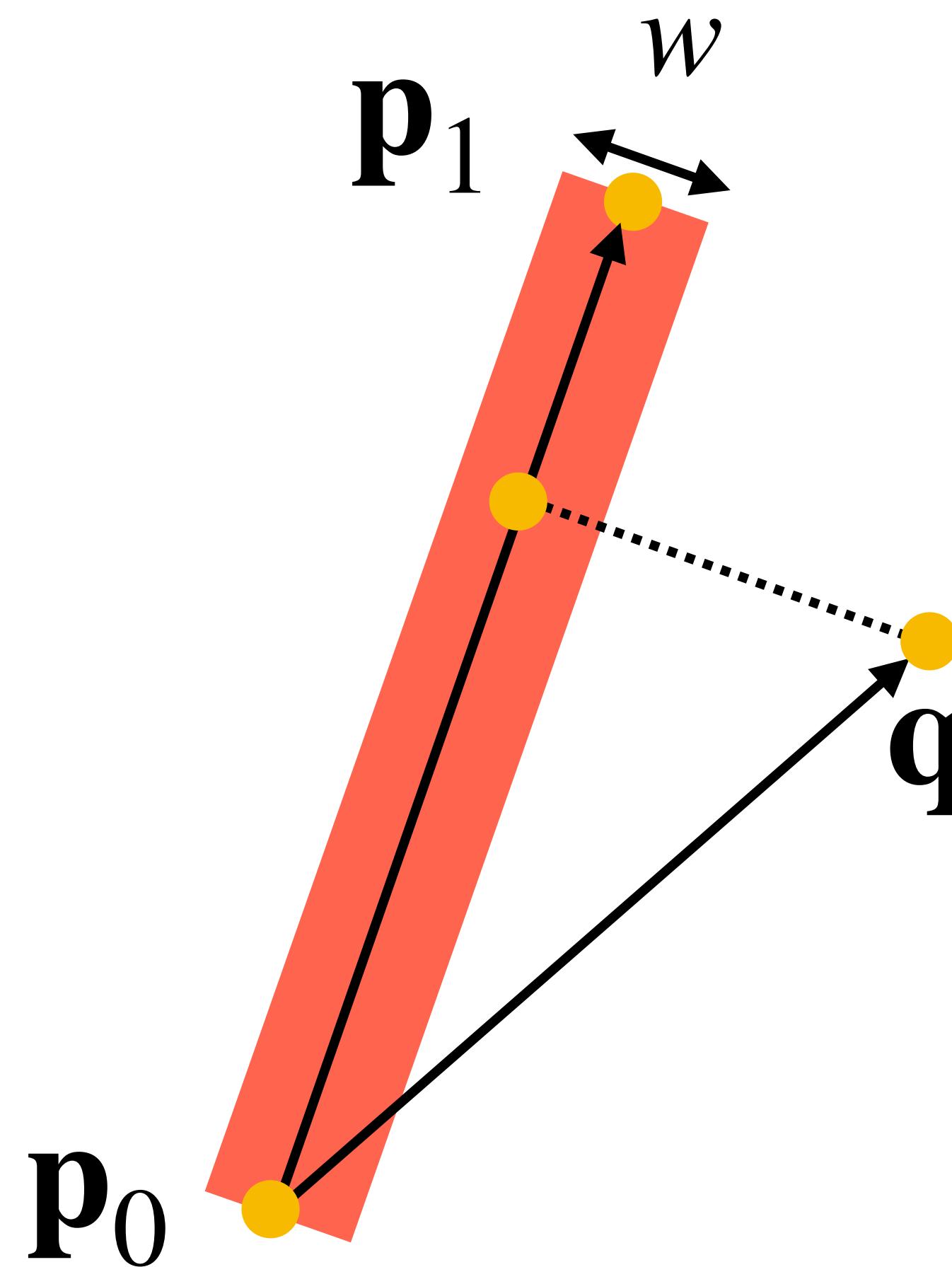
for polylines fill color  
cast ray to the right to determine inside-outside

# Recap

**q**

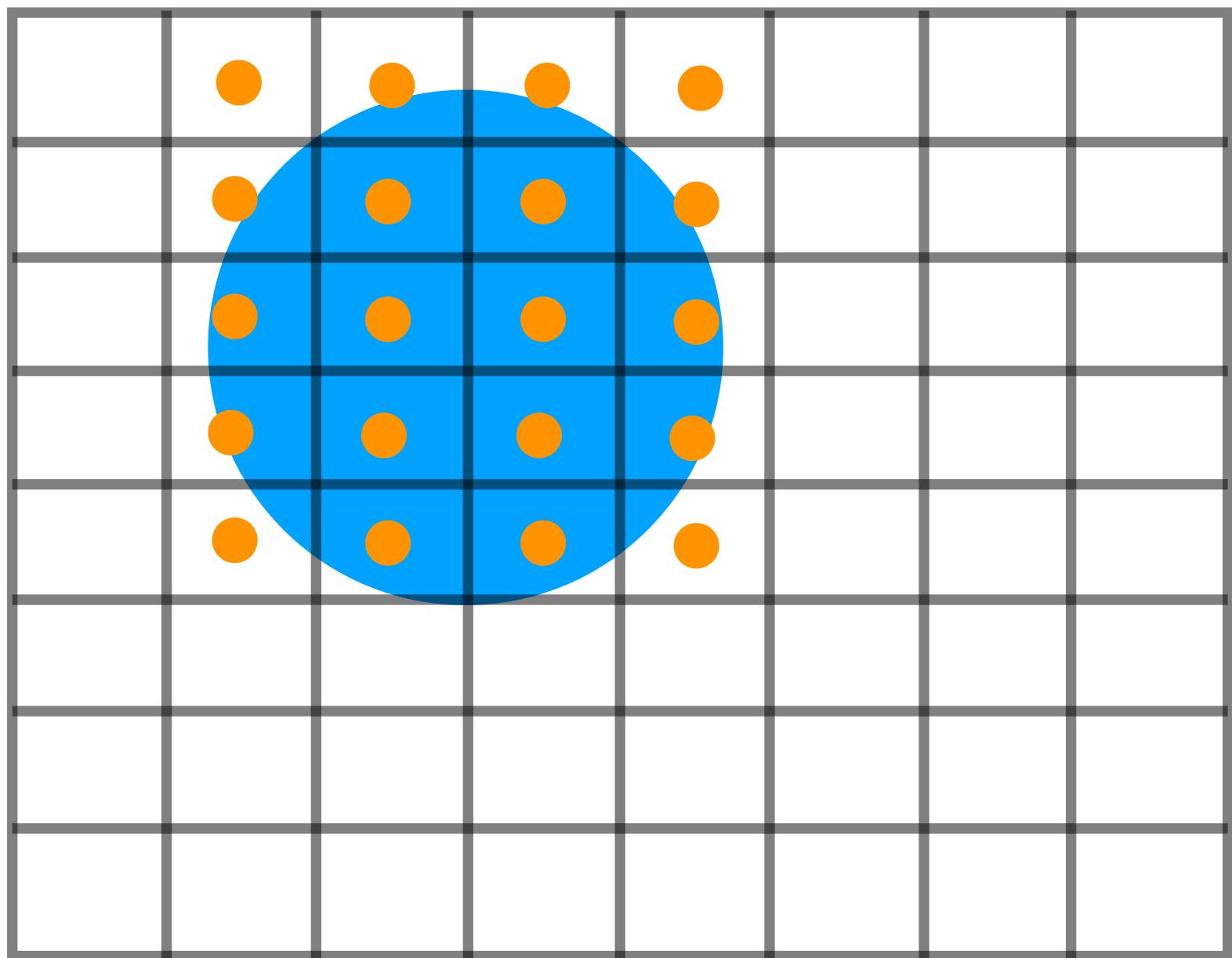


# Recap

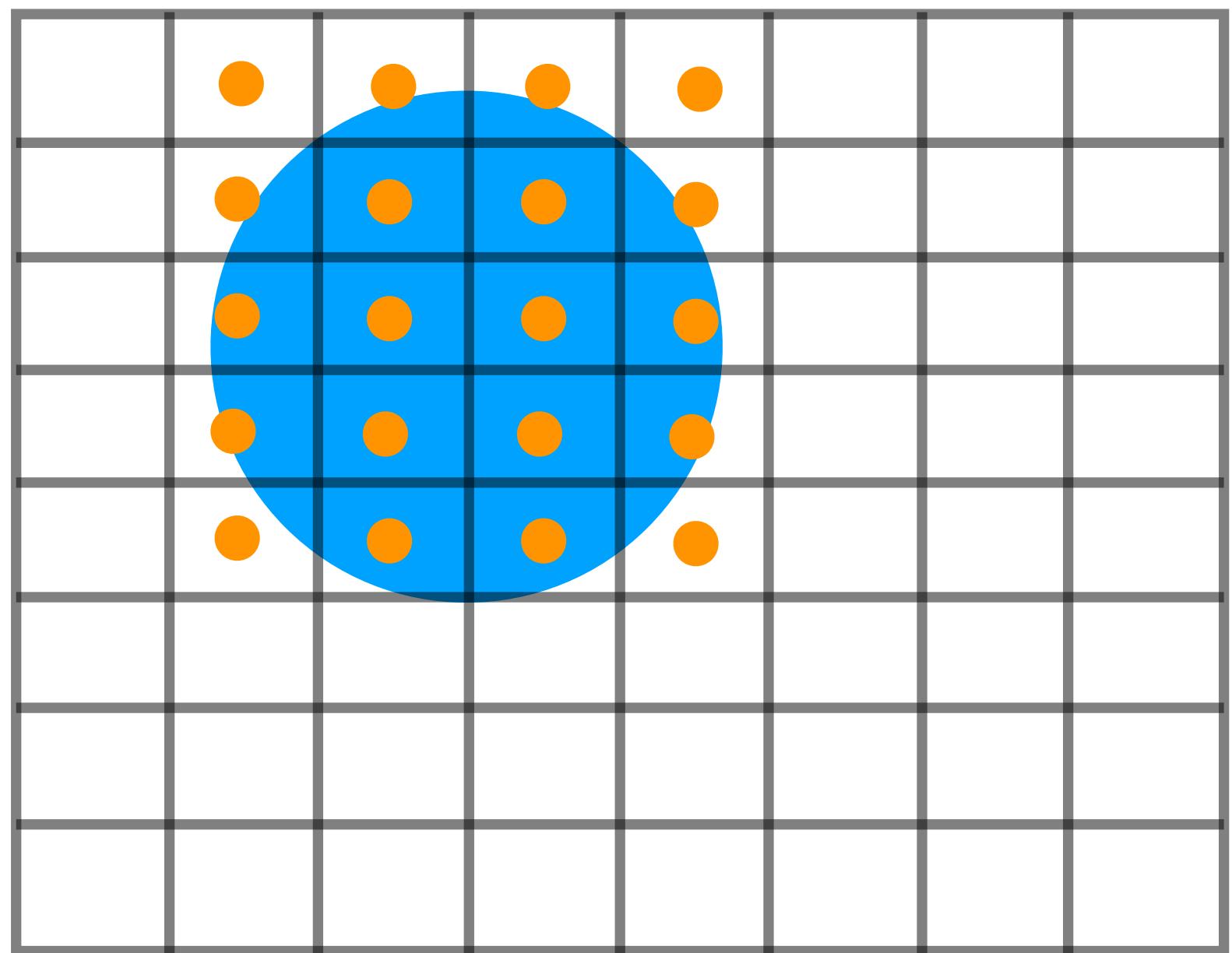


for polylines stroke color, measure the distance  
between the pixel center and the line

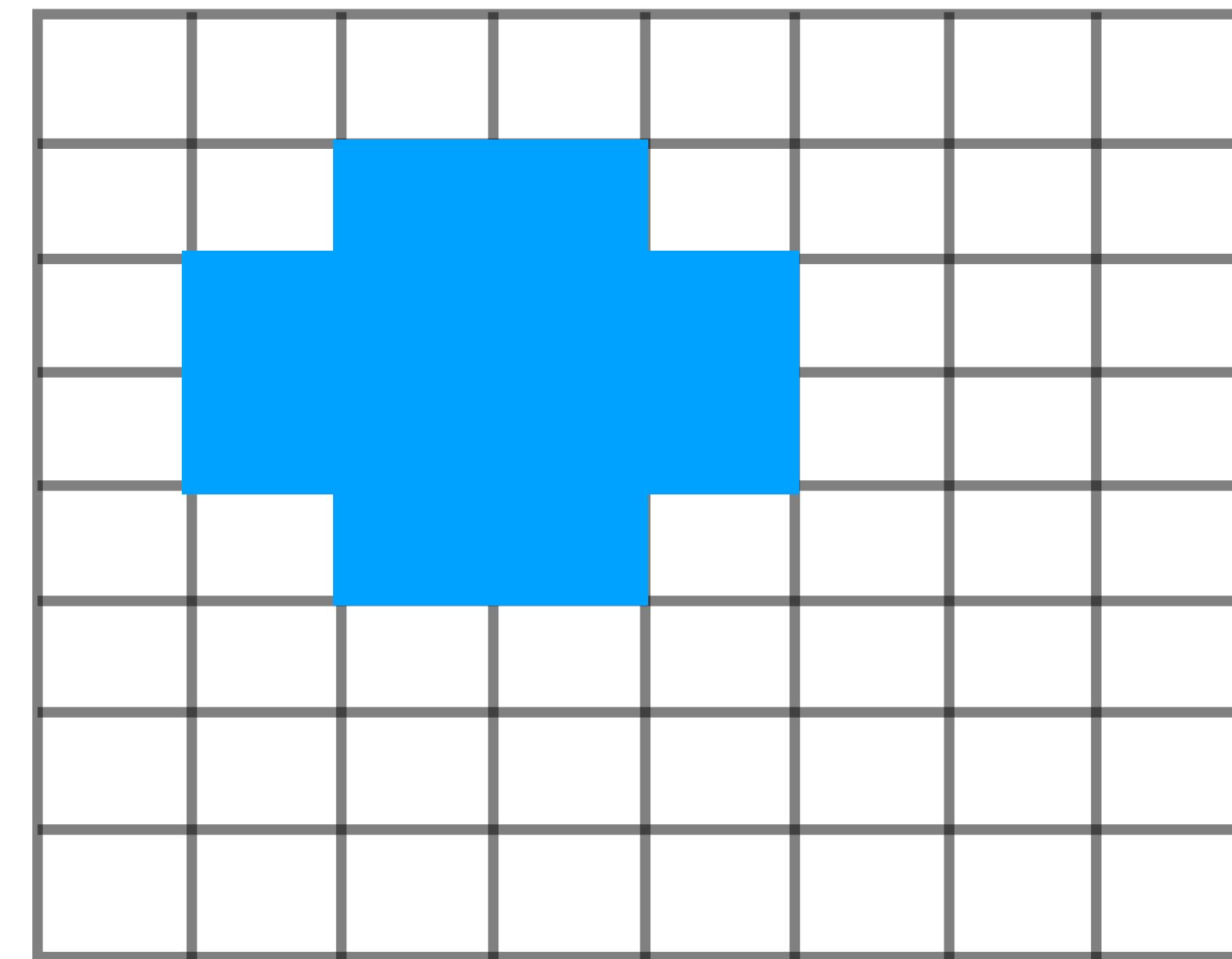
# Is testing pixel center alone sufficient?



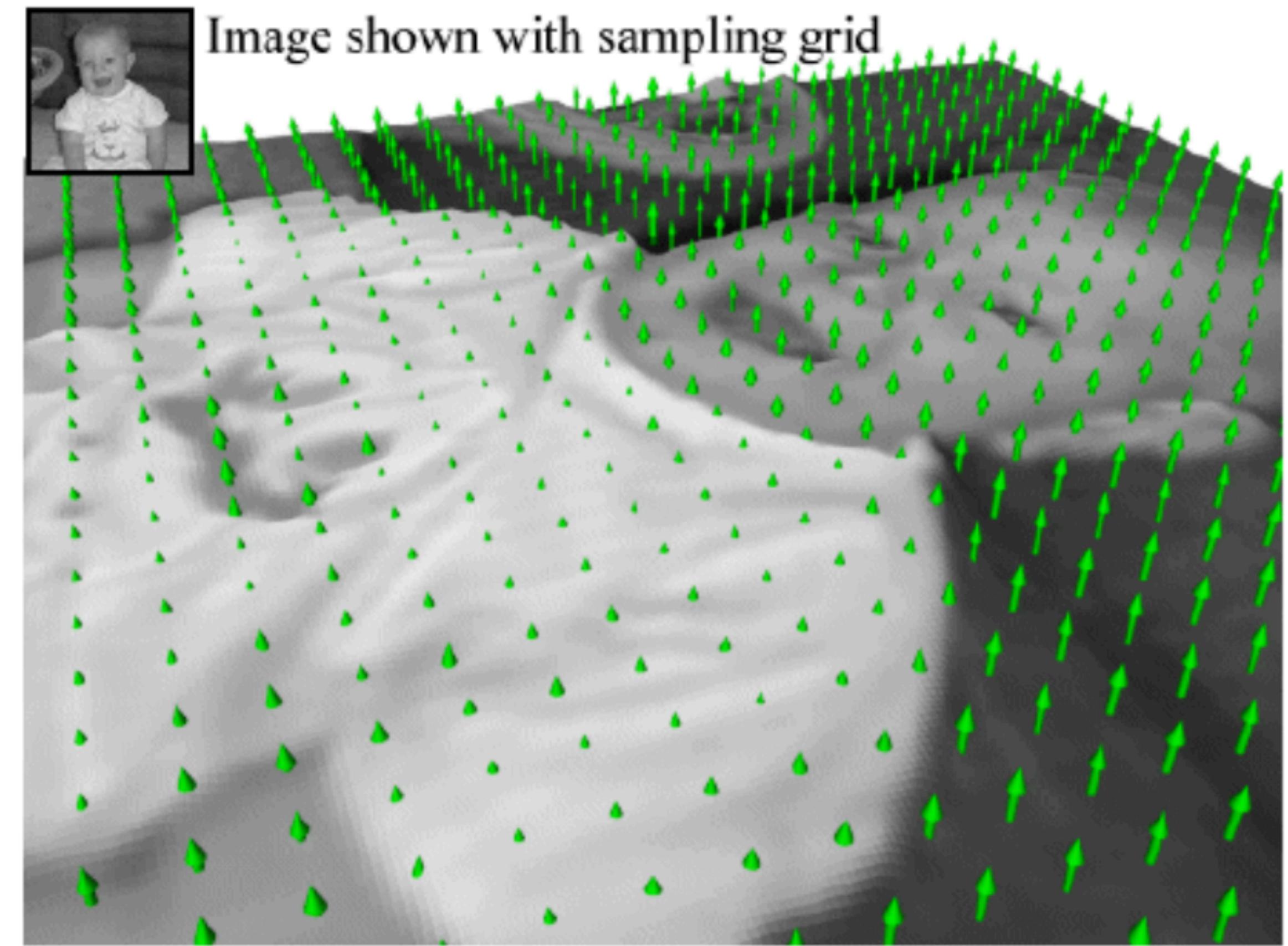
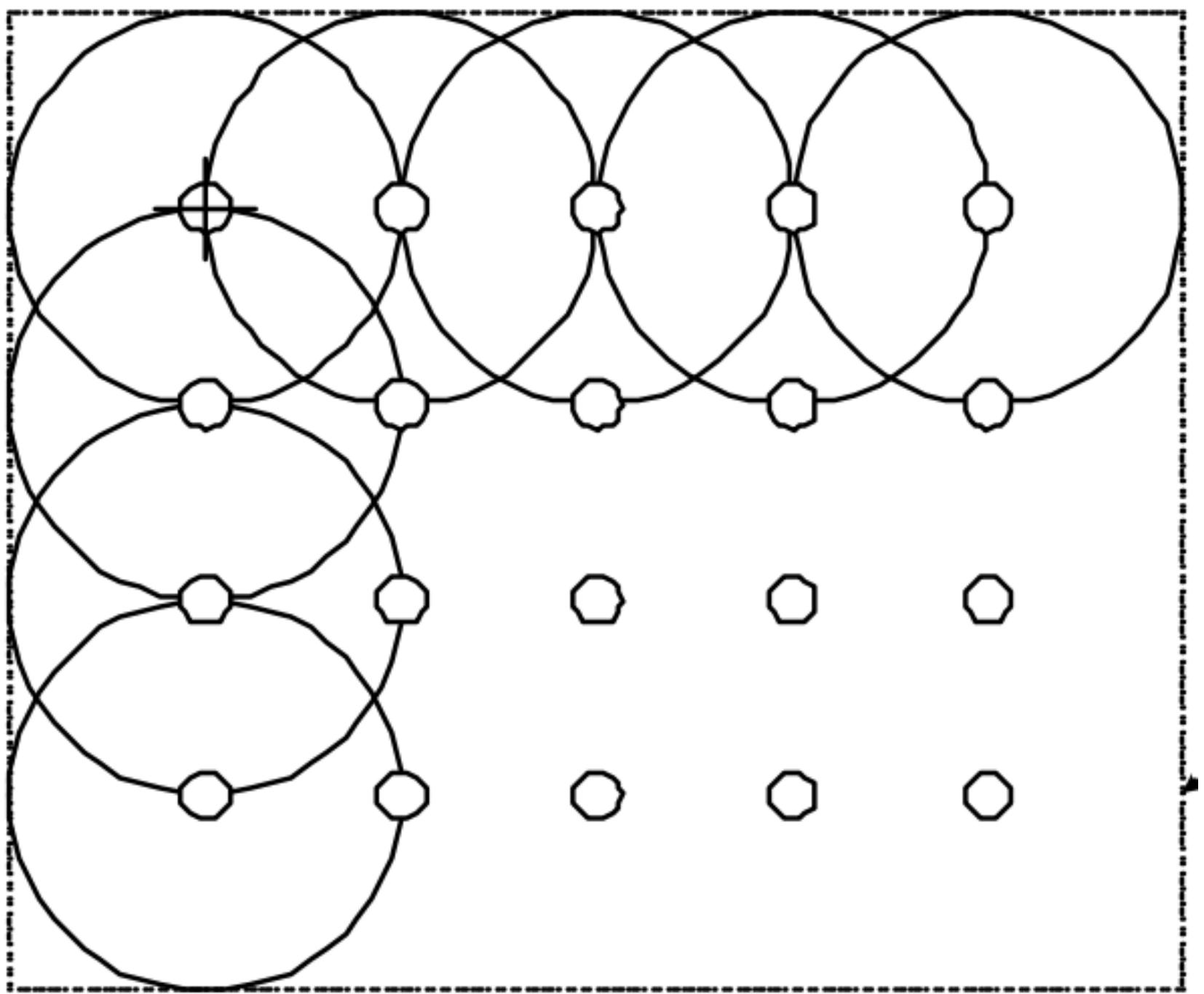
# Is testing pixel center alone sufficient?



“aliasing”

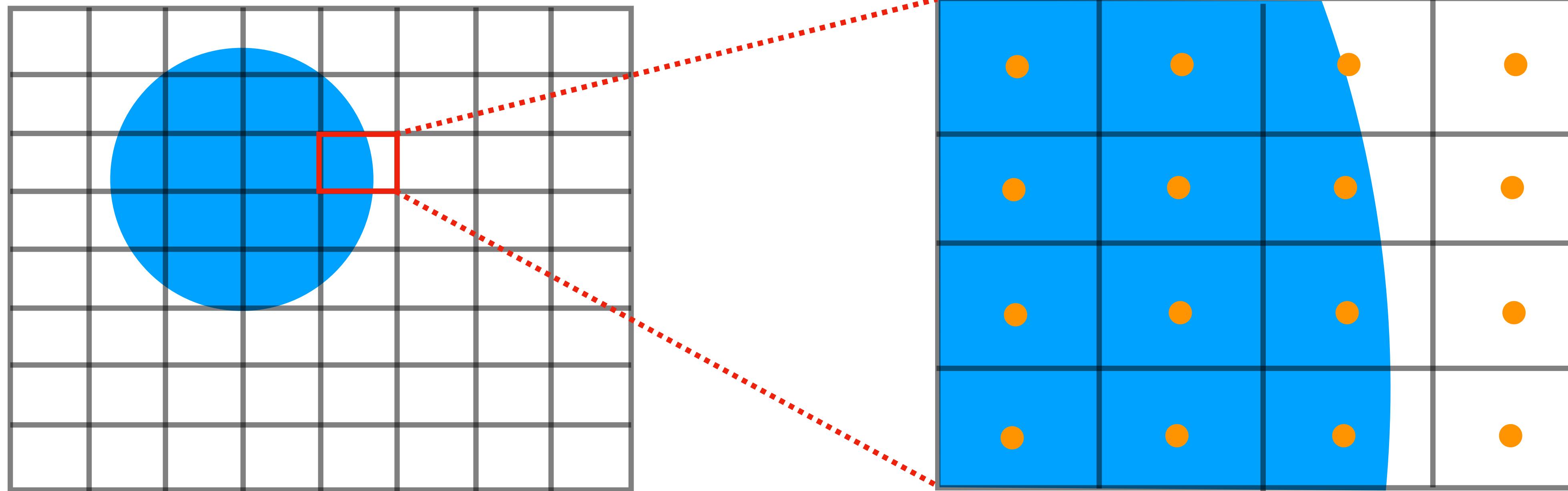


Recall: a pixel should store the average color over a region



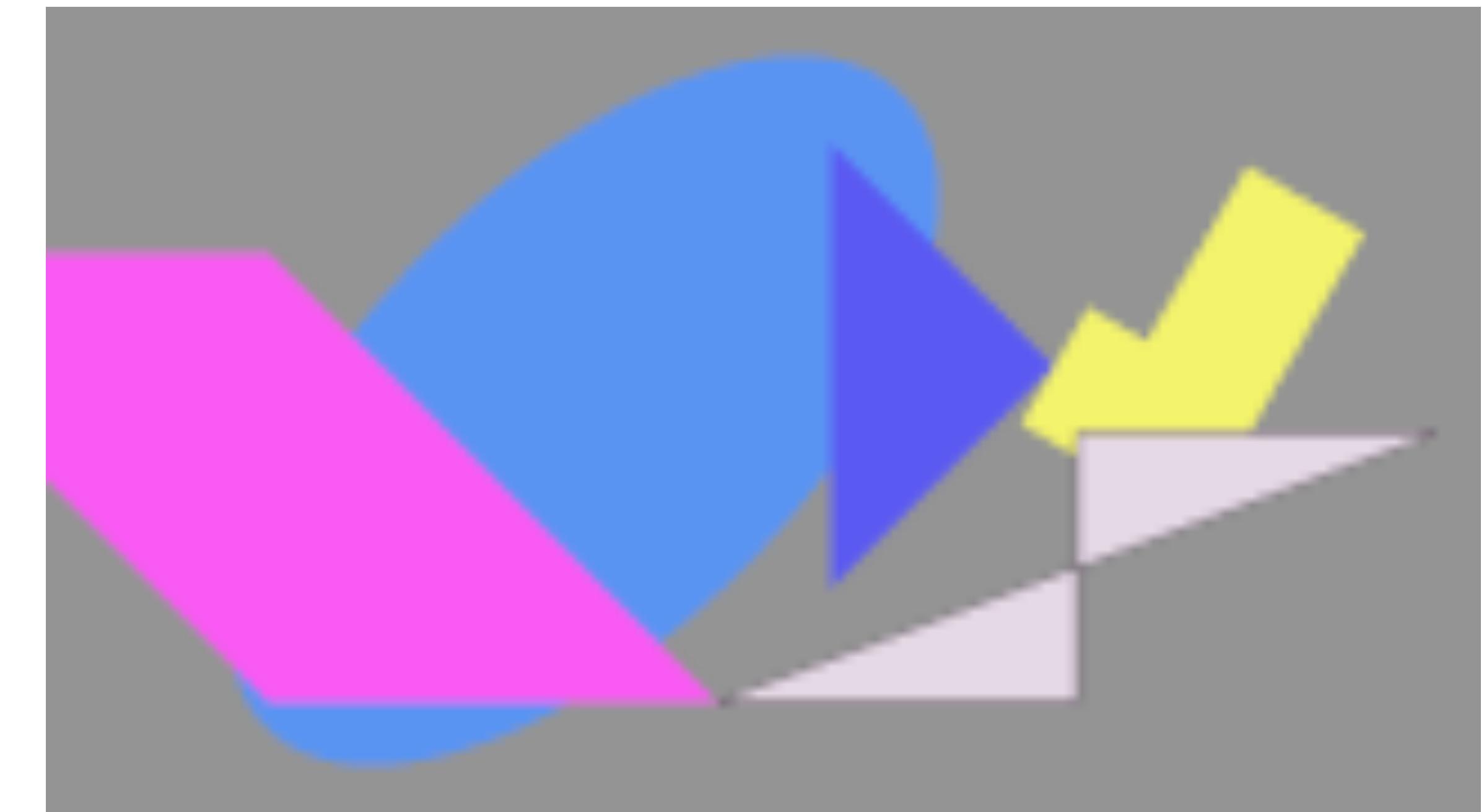
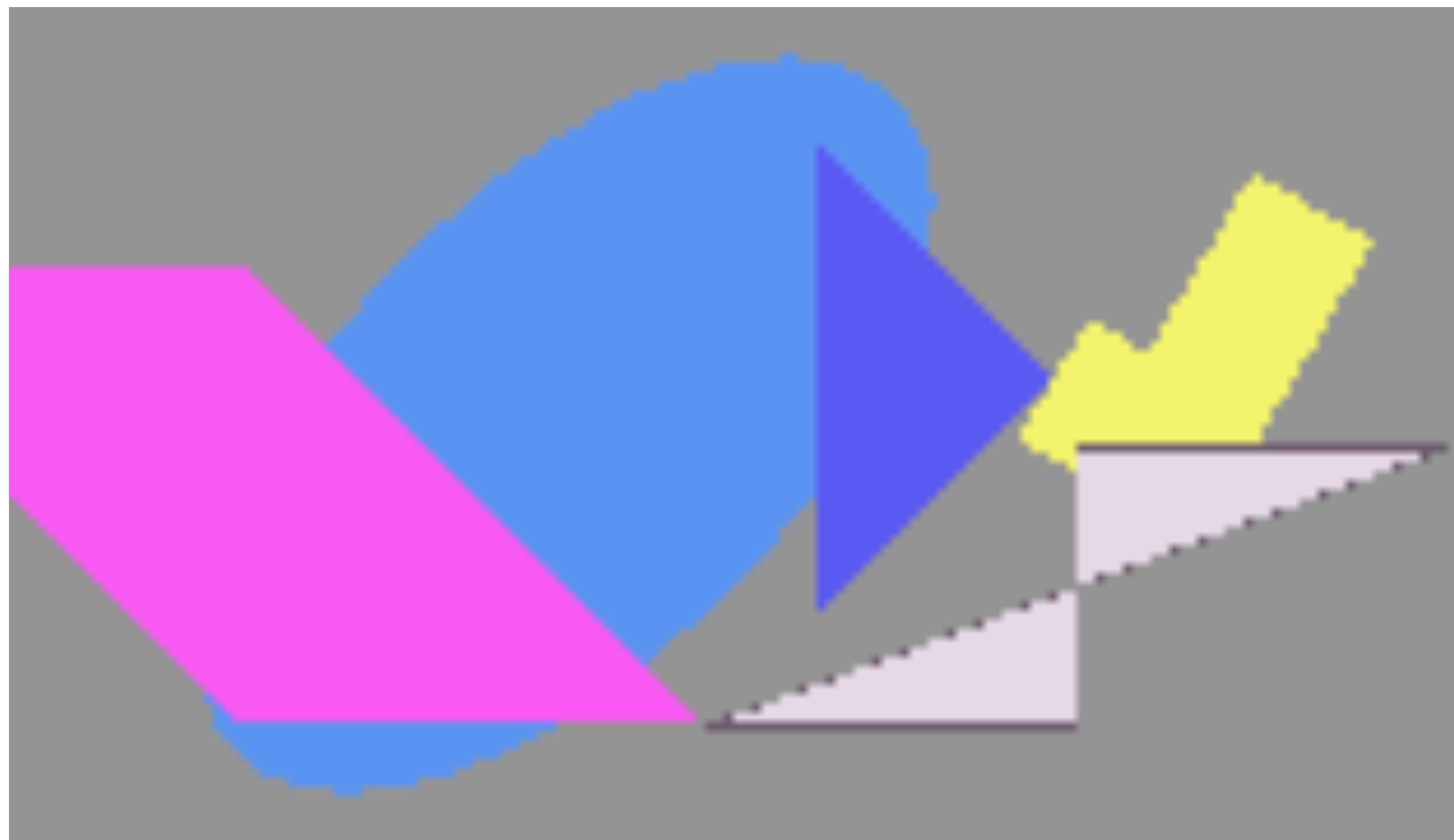
# Super-sampling Antialiasing

combat aliasing artifacts by taking average of color inside a pixel

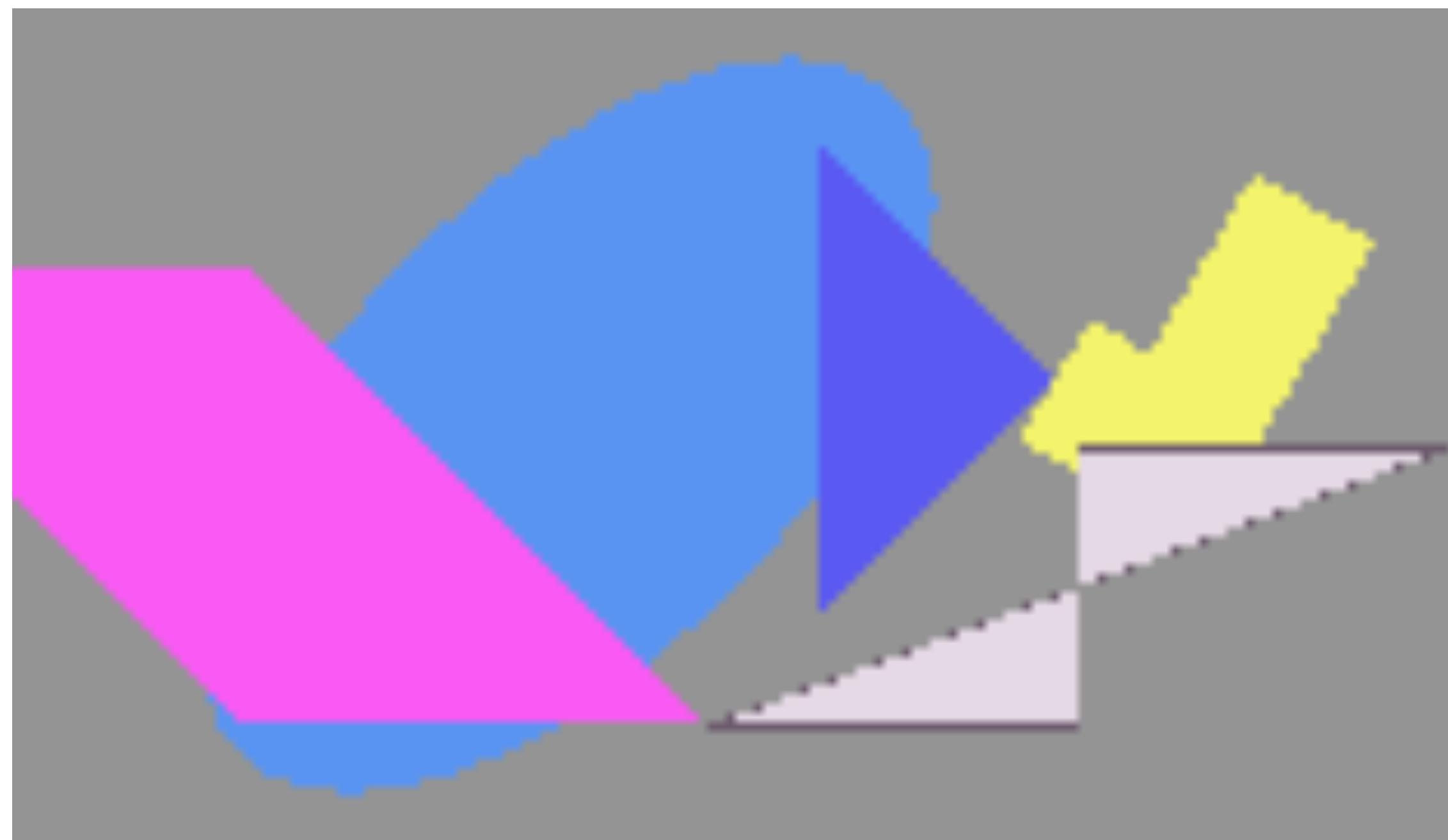


# Aliased vs antialiased

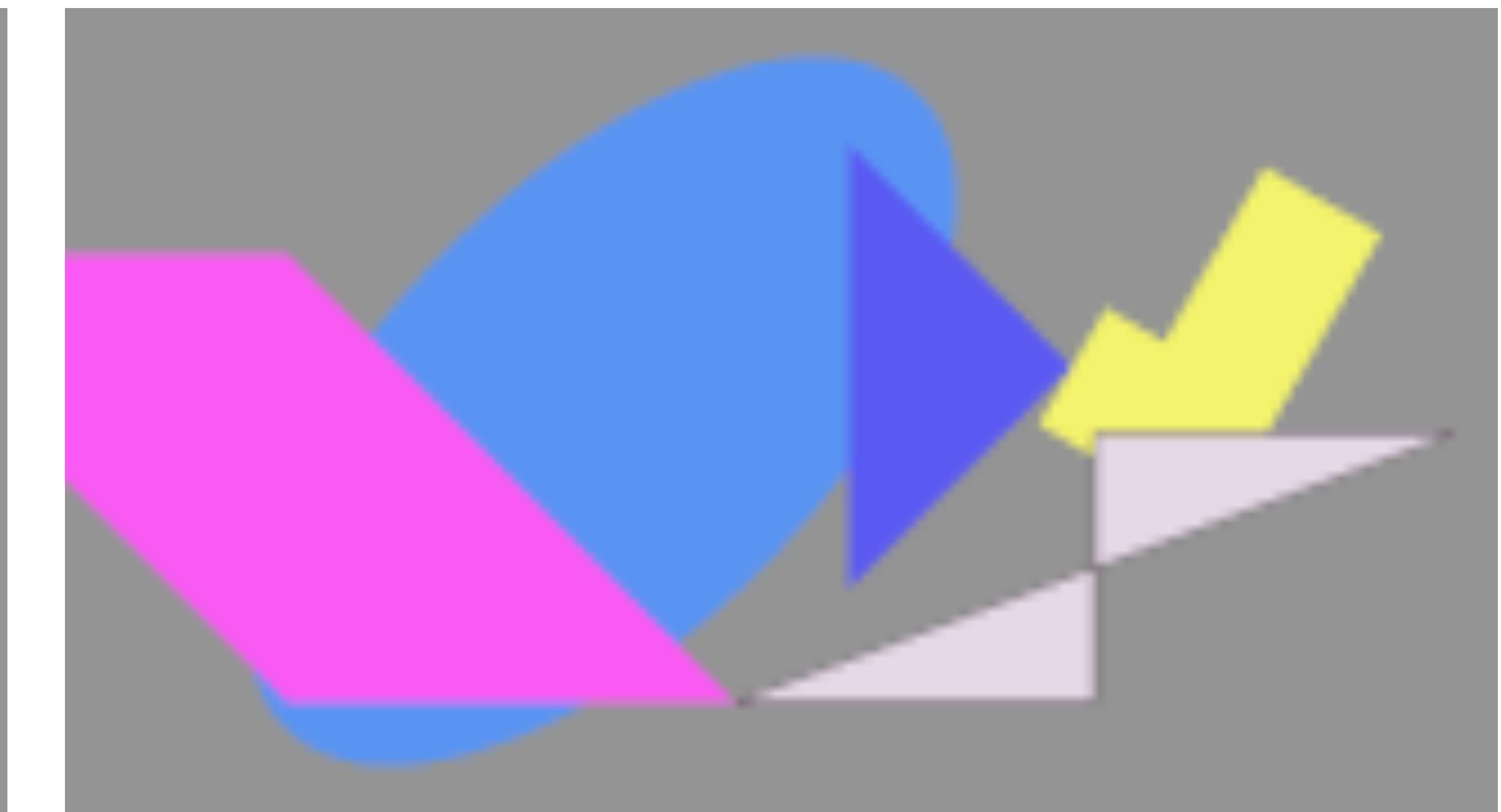
which one is antialiased?



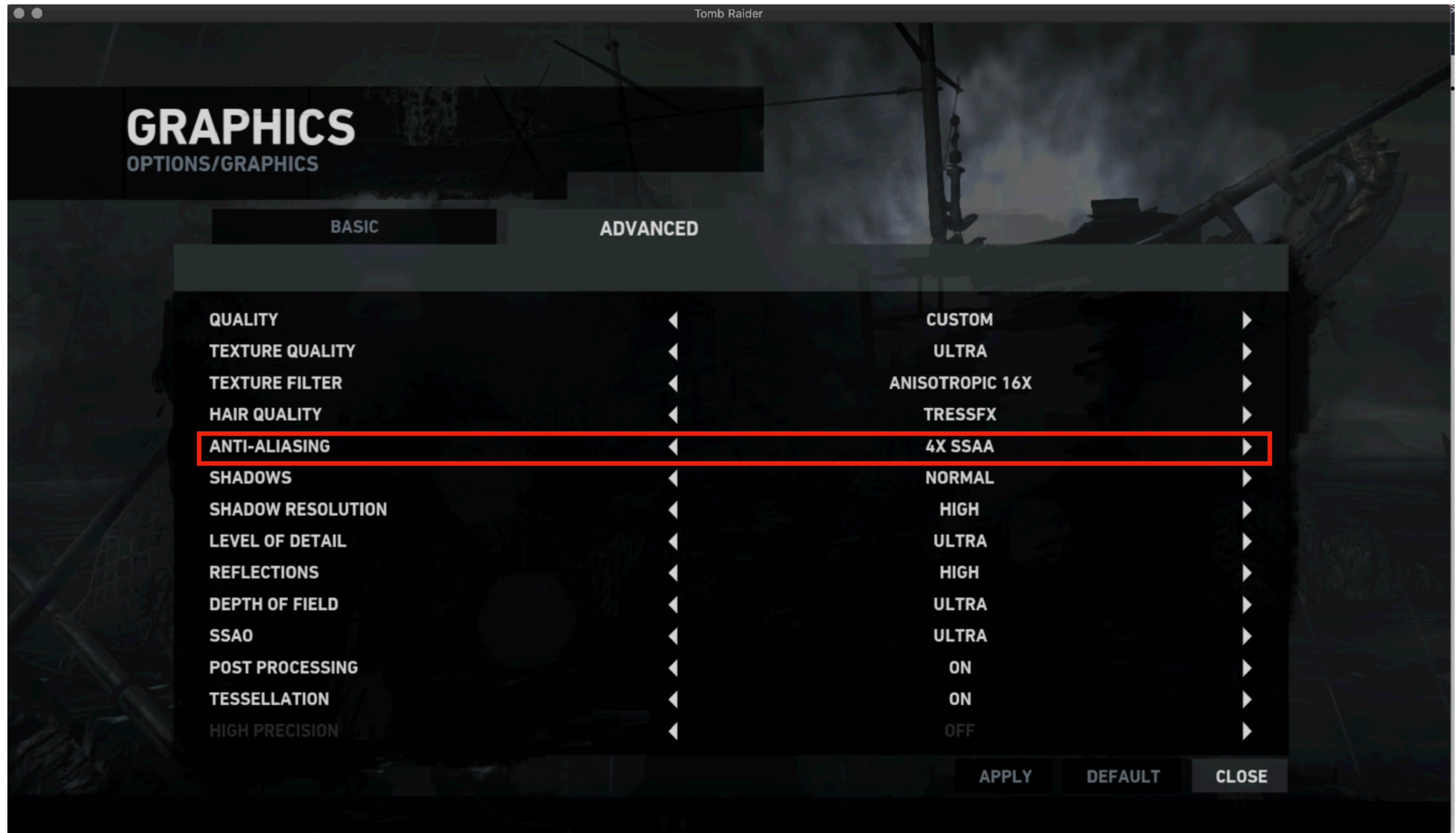
# Aliased vs antialiased



aliased



antialiased



“SSAA” = supersampling anti-aliasing

# Transparency

How do we render transparent primitives?



# Rendering a single transparent object

Q: how do we compute the color here?

color  $C$



100% opacity

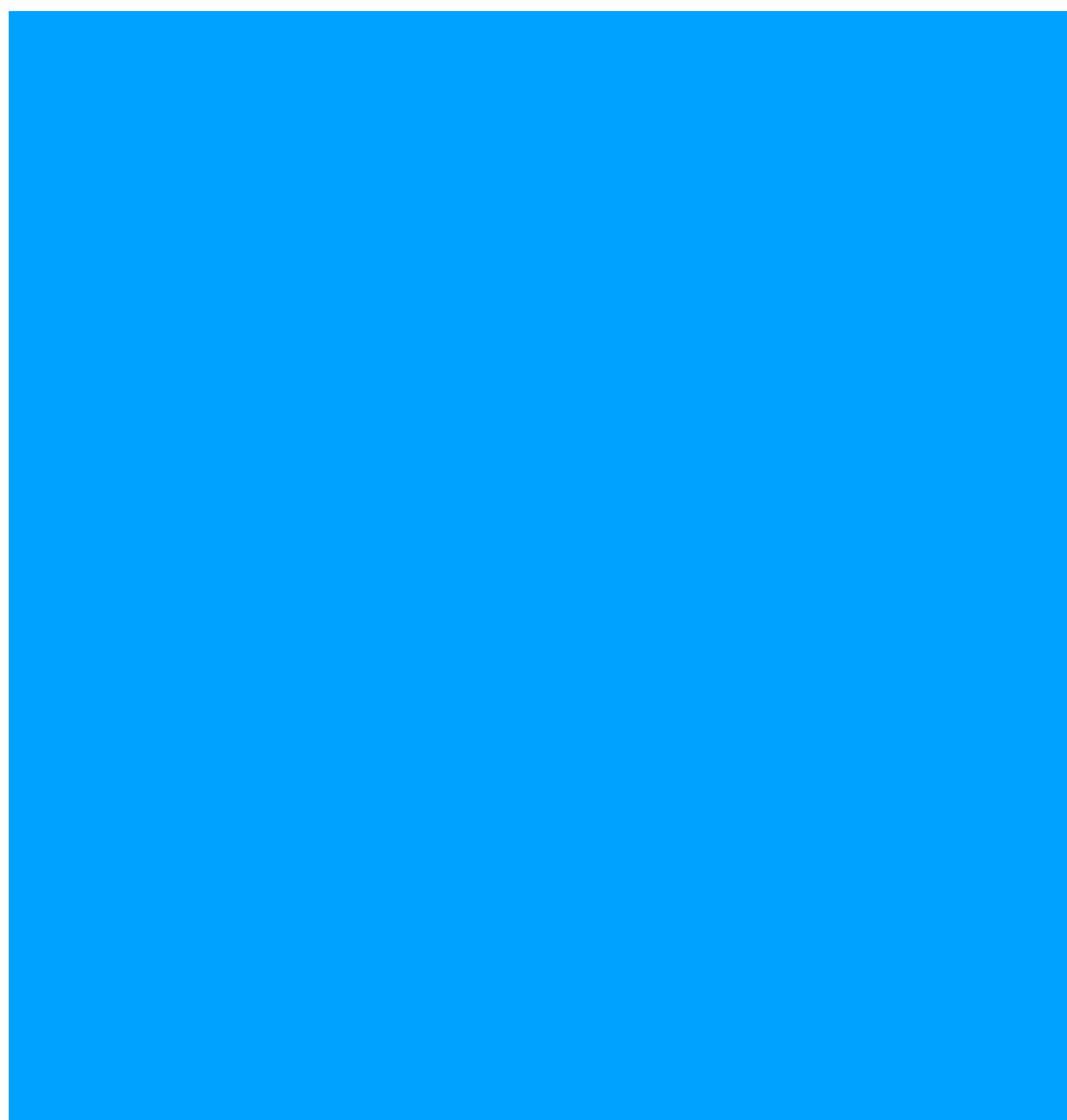


60% opacity

# Rendering a single transparent object

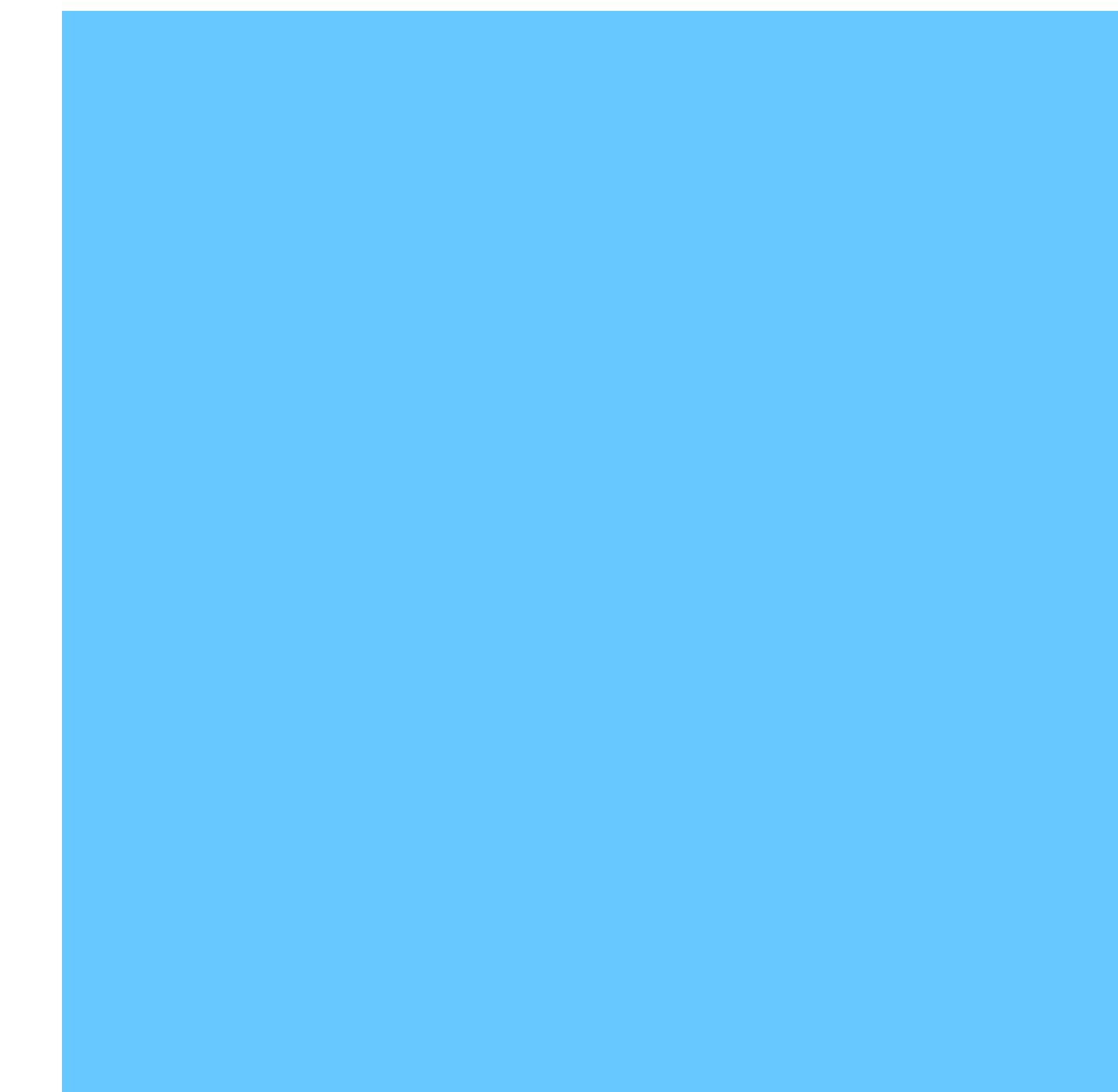
Q: how do we compute the color here?

color  $C$



100% opacity

$$0.6 * C + 0.4 * \text{background}$$



60% opacity

# Rendering two transparent objects

**quiz 1:** which one is at the front?

color  $C_0$ , opacity  $\alpha_0$



color  $C_1$ , opacity  $\alpha_1$

# Rendering two transparent objects

**quiz 1:** which one is at the front?

**quiz 2:** how do we compute the color  
in the middle?

color  $C_0$ , opacity  $\alpha_0$



color  $C_1$ , opacity  $\alpha_1$

# Rendering two transparent objects

**quiz 1:** which one is at the front?

**quiz 2:** how do we compute the color  
in the middle?

$$C_0 * \alpha_0 + C_1 * (1 - \alpha_0)\alpha_1 + \text{background} * (1 - \alpha_0) * (1 - \alpha_1)$$

color  $C_0$ , opacity  $\alpha_0$



color  $C_1$ , opacity  $\alpha_1$

# Alpha blending

$$\begin{aligned} & C_0 * \alpha_0 + \\ & C_1 * (1 - \alpha_0) \alpha_1 + \\ & C_2 * (1 - \alpha_0)(1 - \alpha_1) \alpha_2 + \\ & \dots + \\ & \text{background} * (1 - \alpha_0)(1 - \alpha_1)\dots \end{aligned}$$



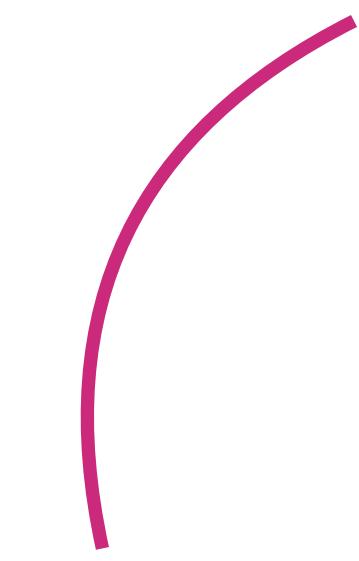
# Alpha blending

$$\begin{aligned} & C_0 * \alpha_0 + \\ & C_1 * (1 - \alpha_0) \alpha_1 + \\ & C_2 * (1 - \alpha_0)(1 - \alpha_1) \alpha_2 + \\ & \dots + \\ & \text{background} * (1 - \alpha_0)(1 - \alpha_1)\dots \end{aligned}$$



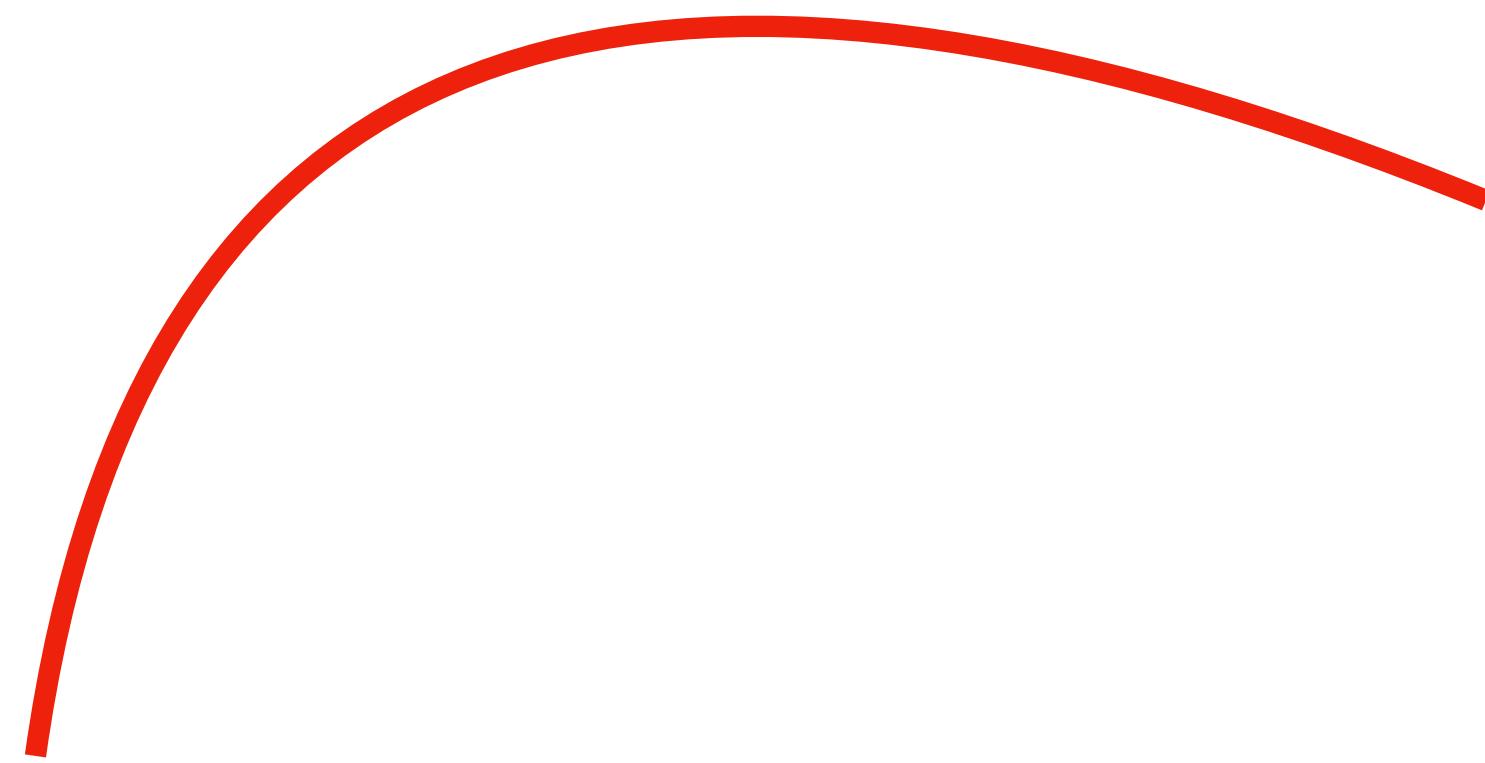
$C_i * a_i$  is often called the “premultiplied alpha”

# What about curves?



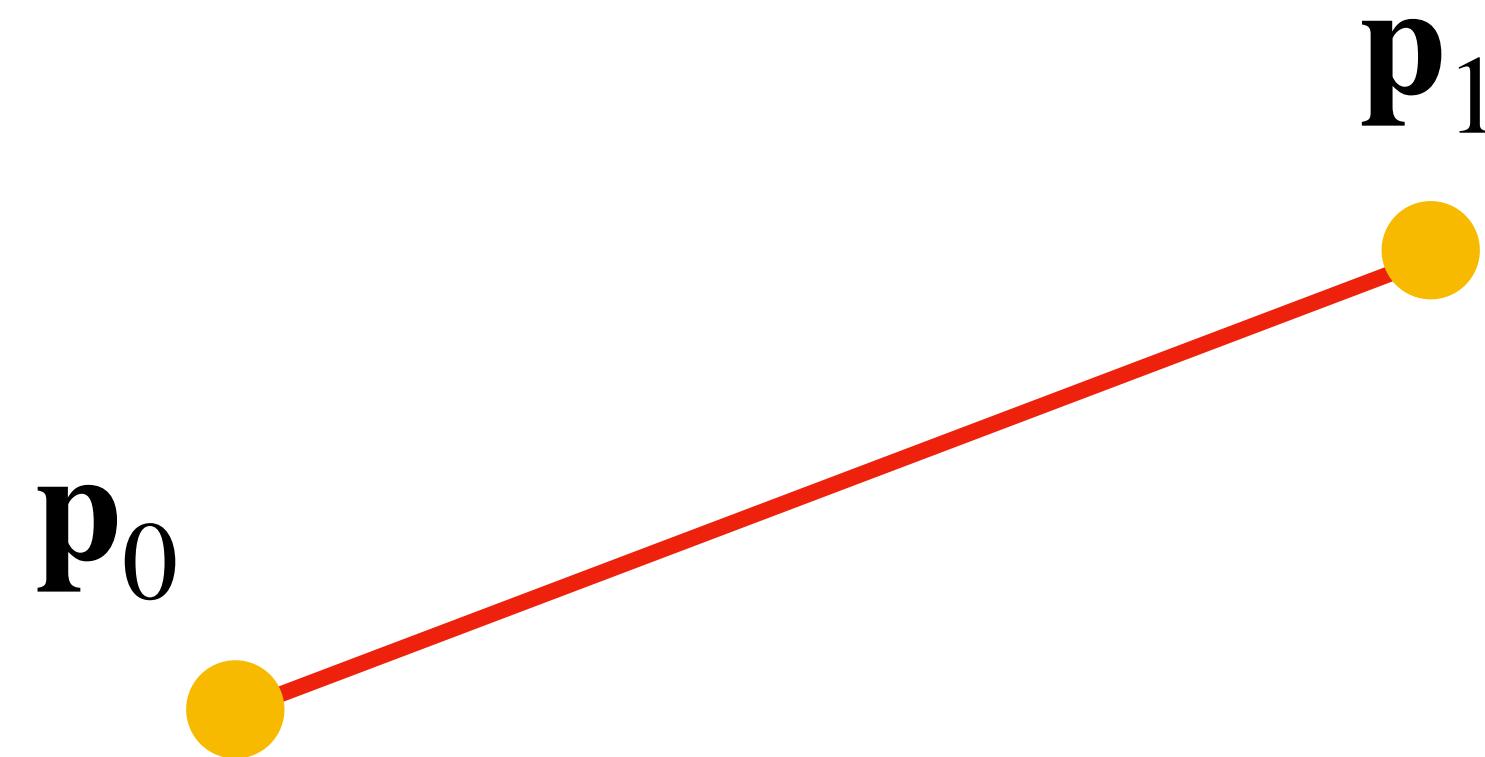
curves

# How do we represent curves?



# How do we represent curves?

let's start from a straight line



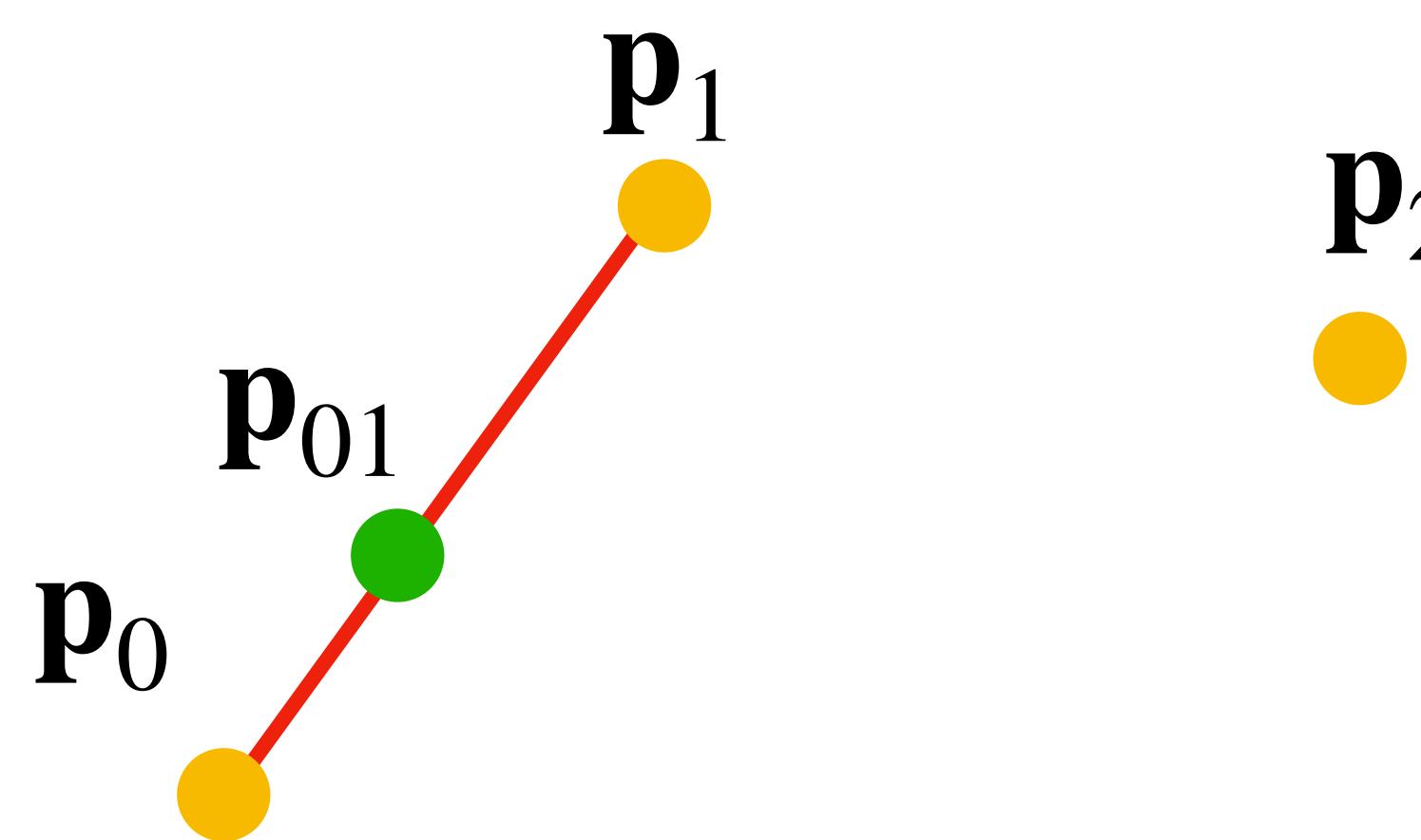
points on the line can be represented as

$$\mathbf{p}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$$

$$t \in [0, 1]$$

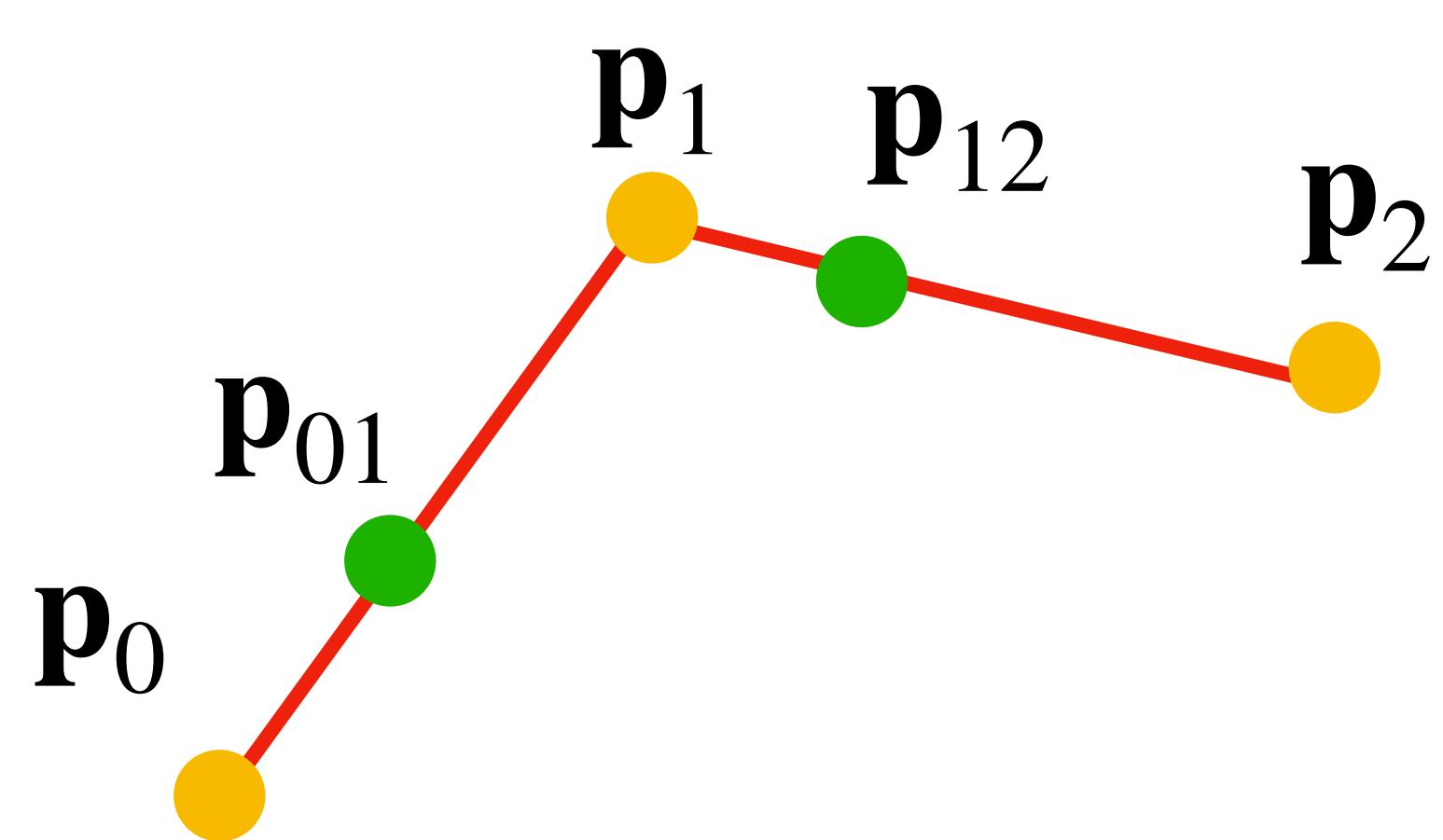
$\mathbf{p}(t)$  is a linear interpolation between  $\mathbf{p}_0$  and  $\mathbf{p}_1$

# Bézier curve [de Casteljau 1959]



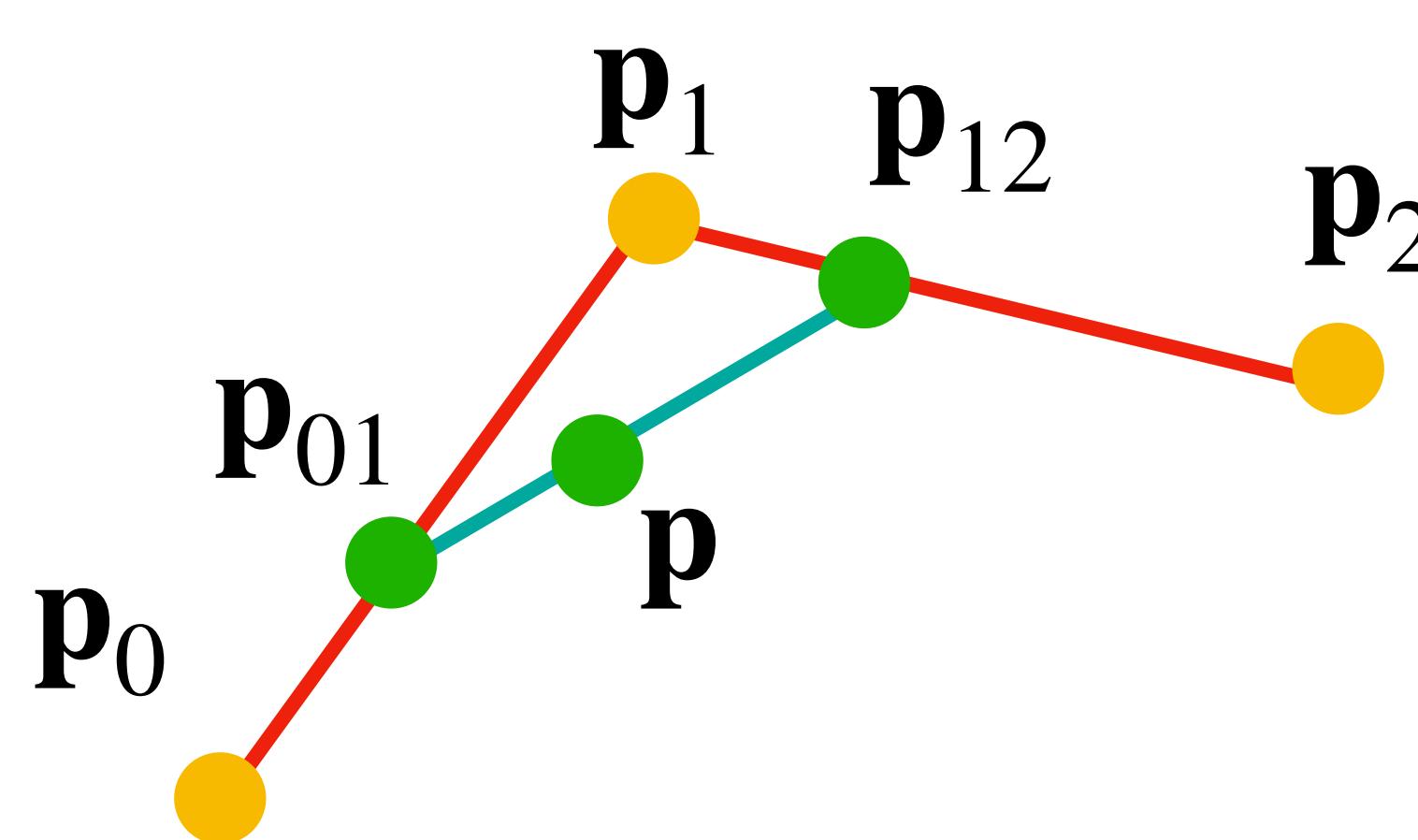
$$1. \text{ let } \mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$$

# Bézier curve [de Casteljau 1959]



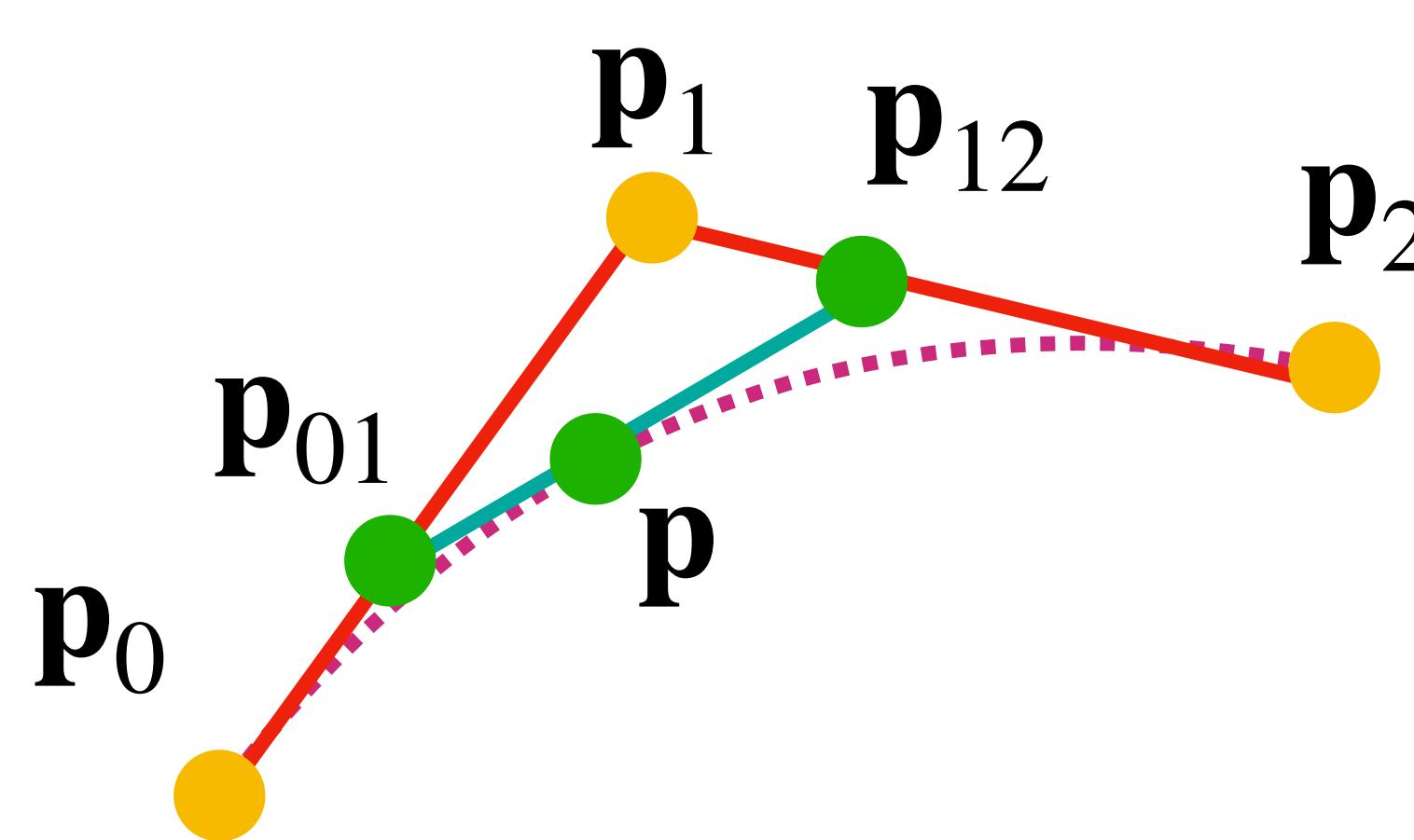
1. let  $\mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$
2. let  $\mathbf{p}_{12}(t) = \mathbf{p}_1(1 - t) + \mathbf{p}_2 t$

# Bézier curve [de Casteljau 1959]



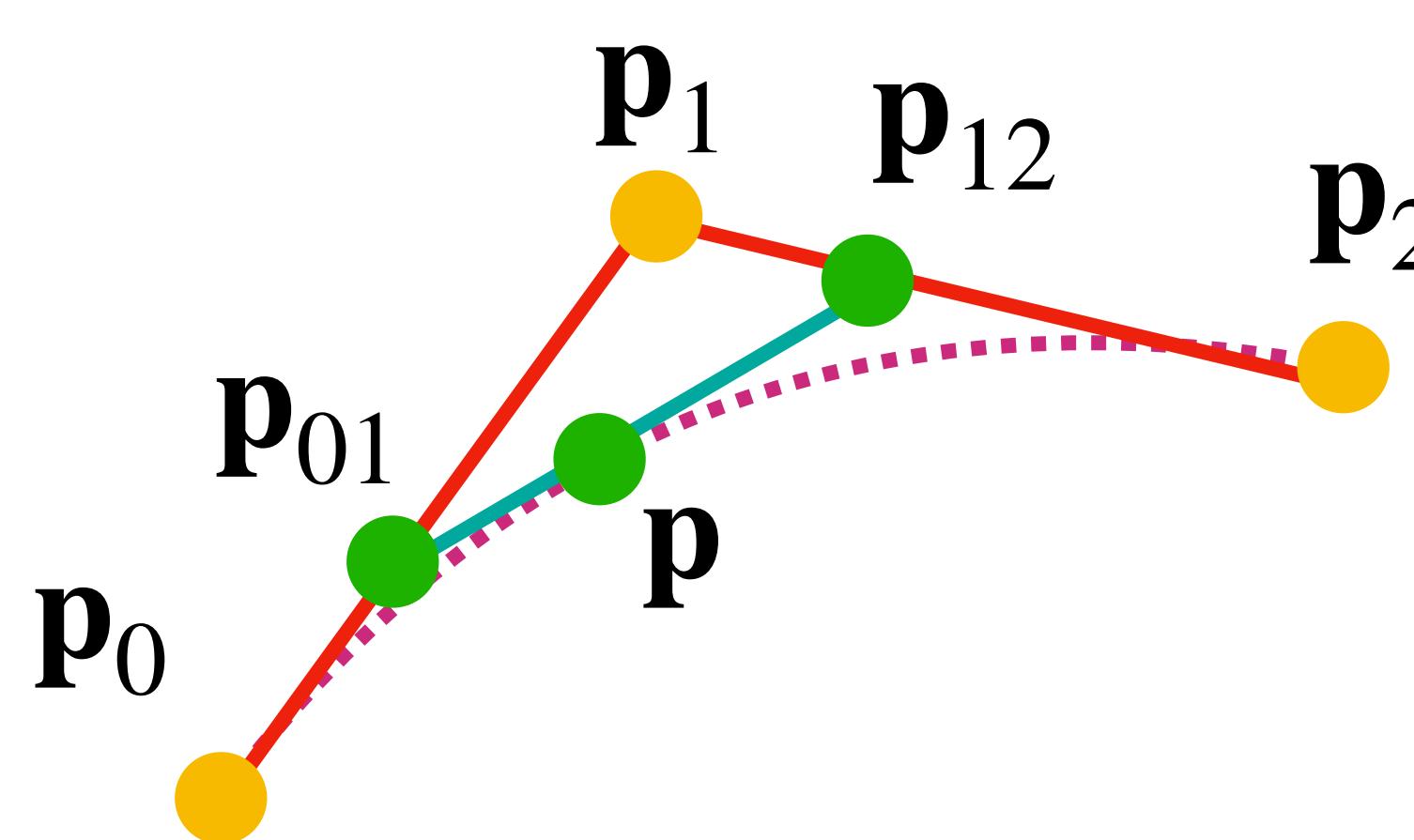
1. let  $\mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$
2. let  $\mathbf{p}_{12}(t) = \mathbf{p}_1(1 - t) + \mathbf{p}_2 t$
3. let  $\mathbf{p}(t) = \mathbf{p}_{01}(1 - t) + \mathbf{p}_{12} t$

# Bézier curve [de Casteljau 1959]



1. let  $\mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$
2. let  $\mathbf{p}_{12}(t) = \mathbf{p}_1(1 - t) + \mathbf{p}_2 t$
3. let  $\mathbf{p}(t) = \mathbf{p}_{01}(1 - t) + \mathbf{p}_{12} t$

# Bézier curve [de Casteljau 1959]

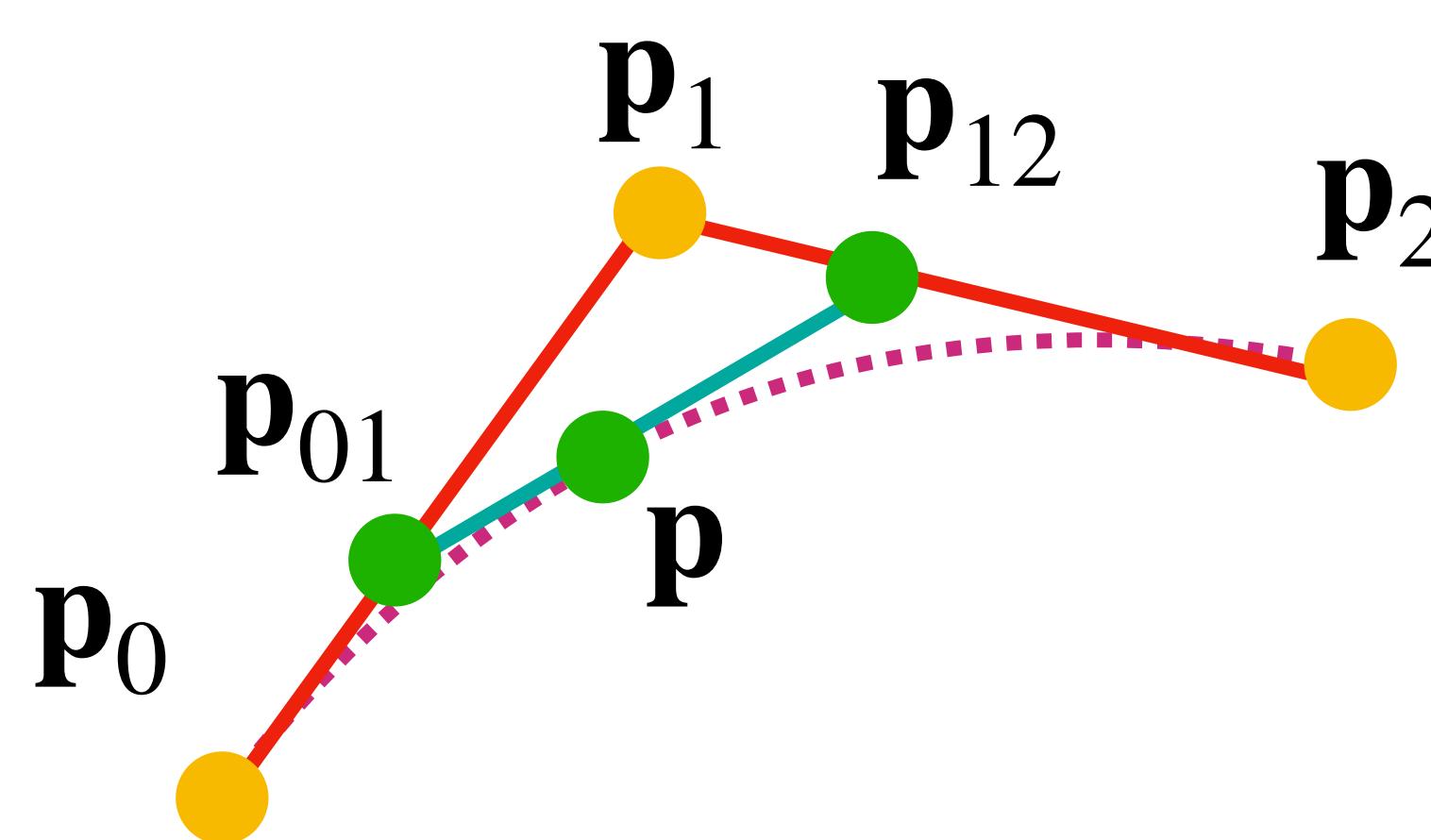


1. let  $\mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$
2. let  $\mathbf{p}_{12}(t) = \mathbf{p}_1(1 - t) + \mathbf{p}_2 t$
3. let  $\mathbf{p}(t) = \mathbf{p}_{01}(1 - t) + \mathbf{p}_{12} t$

cool animation at

<https://ciechanow.ski/curves-and-surfaces/>

# Bézier curve [de Casteljau 1959]



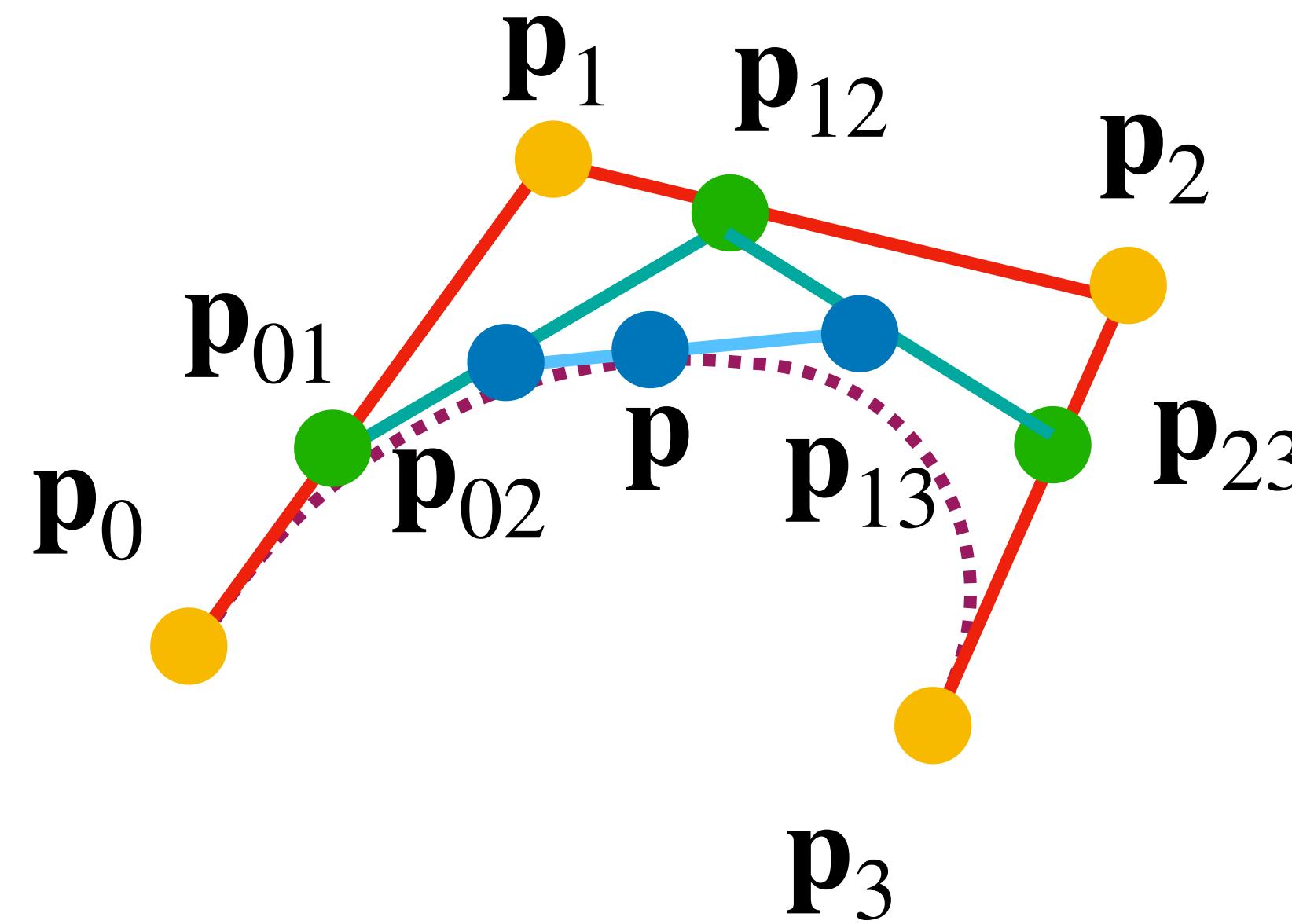
1. let  $\mathbf{p}_{01}(t) = \mathbf{p}_0(1 - t) + \mathbf{p}_1 t$
2. let  $\mathbf{p}_{12}(t) = \mathbf{p}_1(1 - t) + \mathbf{p}_2 t$
3. let  $\mathbf{p}(t) = \mathbf{p}_{01}(1 - t) + \mathbf{p}_{12} t$

expanding, we get

$$\mathbf{p}(t) = (1 - t)^2 \mathbf{p}_0 + 2(1 - t)t \mathbf{p}_1 + t^2 \mathbf{p}_2$$

this is called a “quadratic Bézier curve”

# Bézier curve [de Casteljau 1959]



we can add more control points, 4 control points form a  
“cubic Bézier curve”

$$\mathbf{p}(t) = (1 - t)^3 \mathbf{p}_0 + 3(1 - t)^2 t \mathbf{p}_1 + 3(1 - t)t^2 \mathbf{p}_2 + t^3 \mathbf{p}_3$$

in practice, usually quadratic and cubic Bézier curves are used

# Bézier curve [de Casteljau 1959]

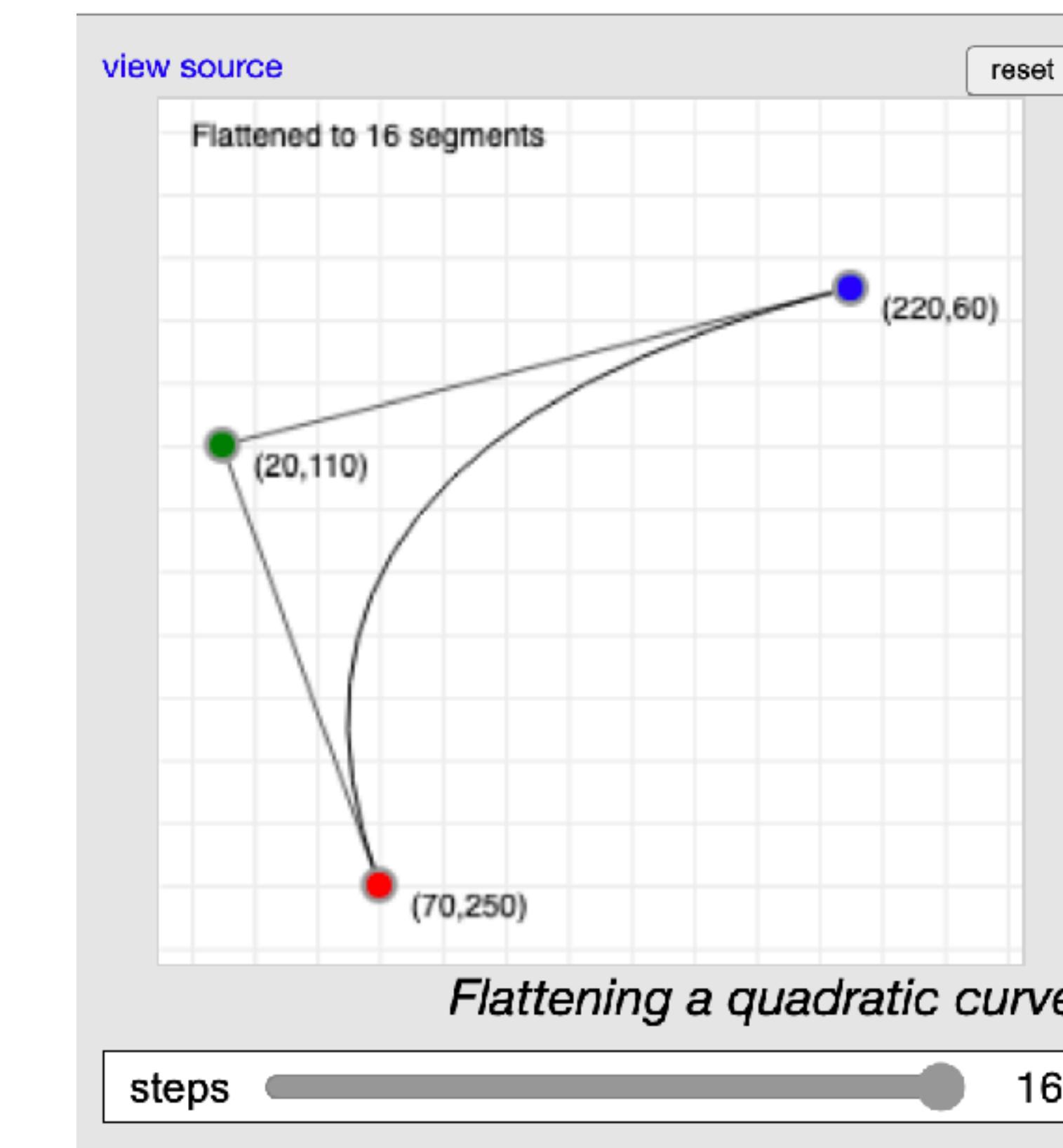
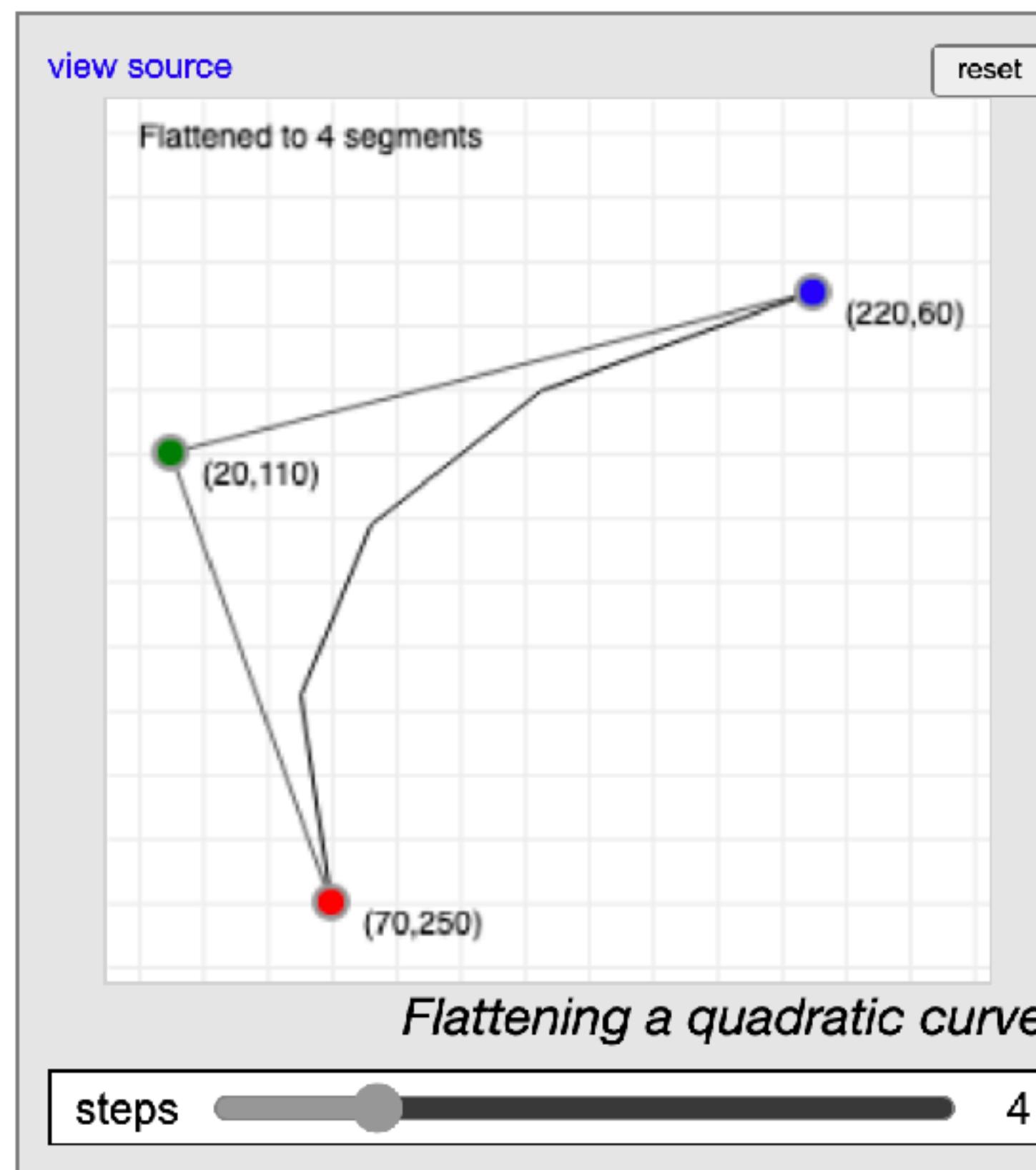
Excellent resources about Bezier curves:

- Bartosz Ciechanowski — Curves and Surfaces (<https://ciechanow.ski/curves-and-surfaces/>)
- Pomax — A Primer on Bézier Curves (<https://pomax.github.io/bezierinfo/>)

we will talk more about curves and surfaces later in the class

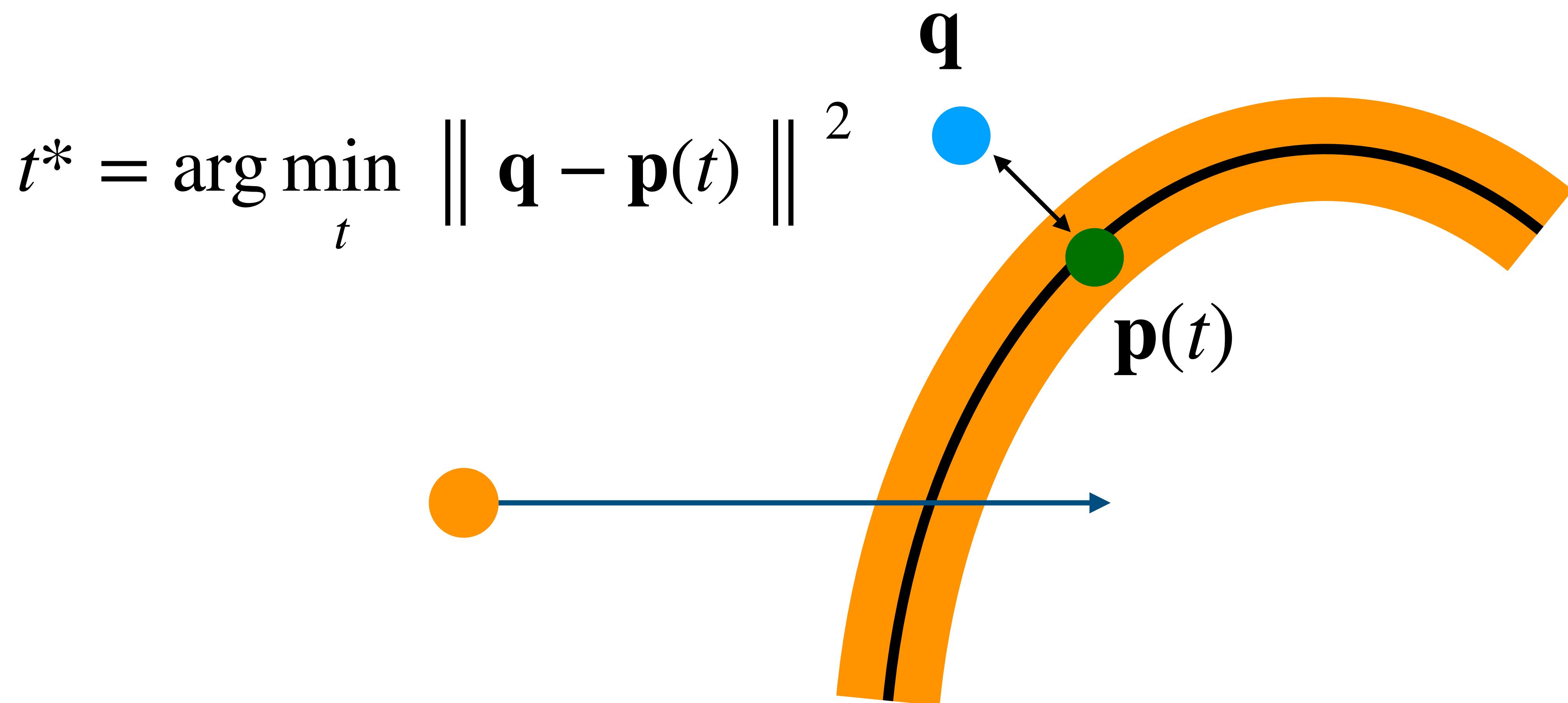
# Rendering Bézier curve

Method 1: flatten the Bézier curve into line segments



# Rendering Bézier curve

Method 2: directly compute ray-curve intersection & distance between point and curve



# SVG (Scalable Vector Graphics)

[https://www.w3schools.com/graphics/svg\\_intro.asp](https://www.w3schools.com/graphics/svg_intro.asp)

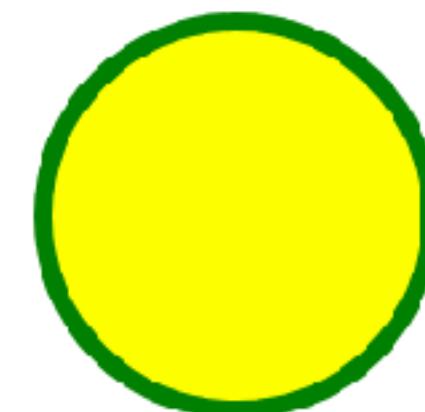
```
<html>
<body>

<h1>My first SVG</h1>

<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>

</body>
</html>
```

**My first SVG**



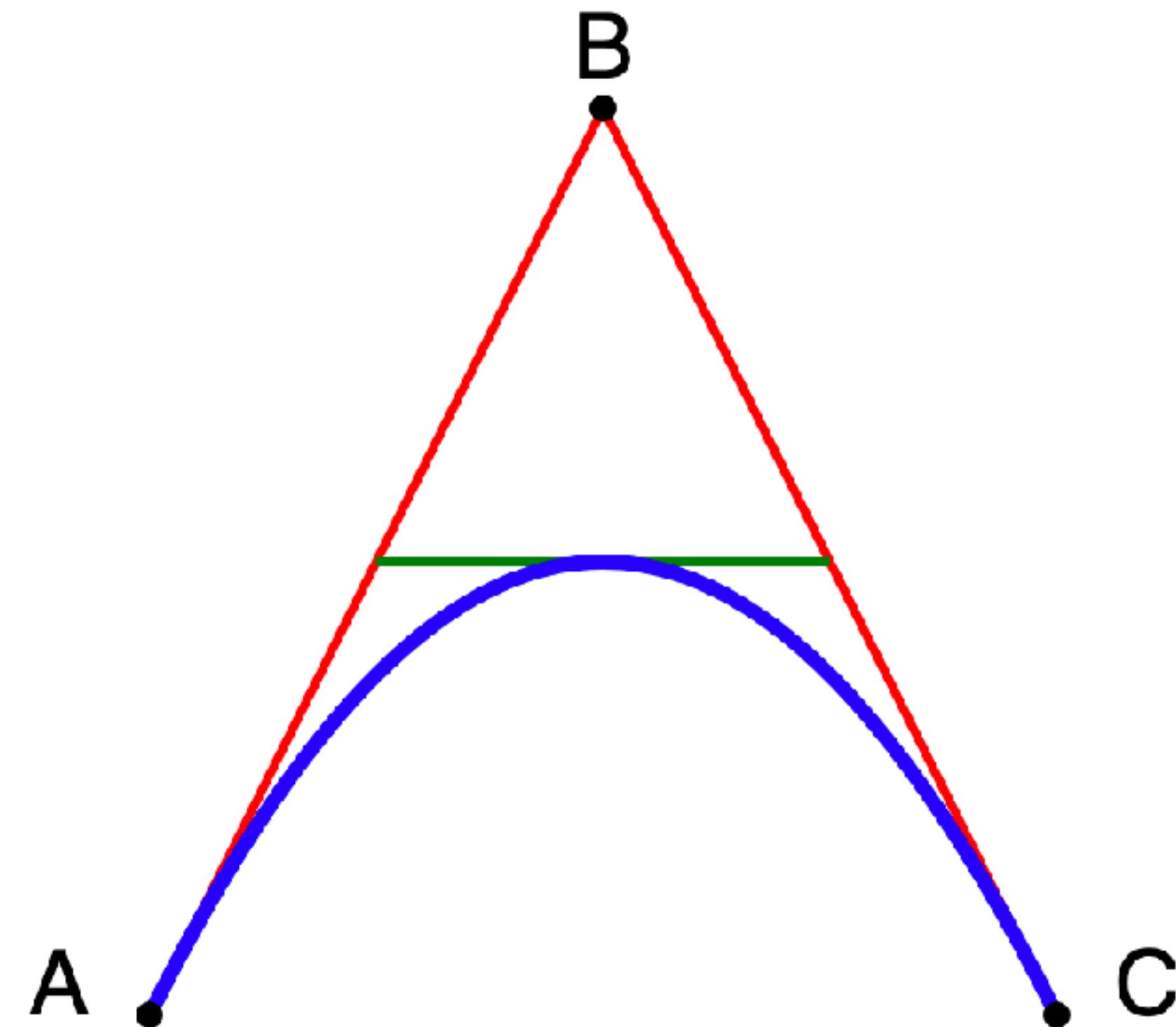
# SVG (Scalable Vector Graphics)

```
<svg width="400" height="110">
  <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)" />
</svg>
```

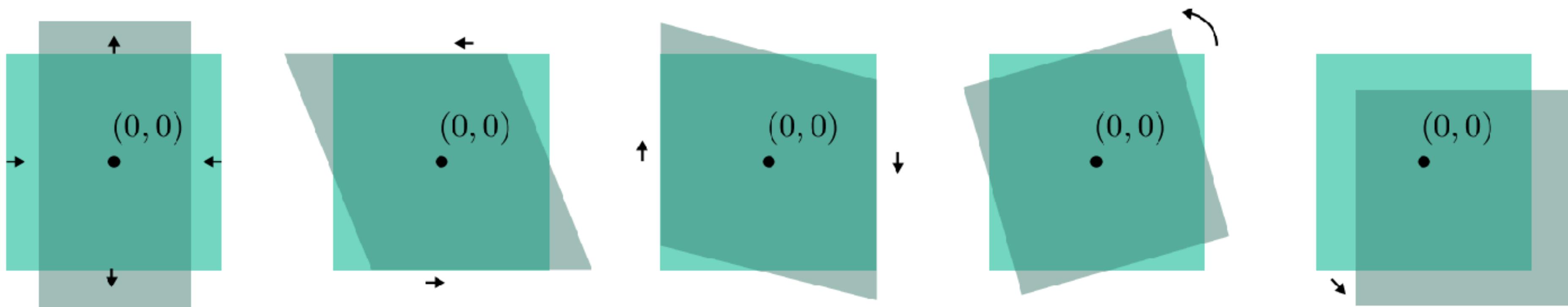


# SVG (Scalable Vector Graphics)

```
<svg height="400" width="450">
  <path id="lineAB" d="M 100 350 l 150 -300" stroke="red"
    stroke-width="3" fill="none" />
  <path id="lineBC" d="M 250 50 l 150 300" stroke="red"
    stroke-width="3" fill="none" />
  <path d="M 175 200 l 150 0" stroke="green" stroke-width="3"
    fill="none" />
  <path d="M 100 350 q 150 -300 300 0" stroke="blue"
    stroke-width="5" fill="none" />
  <!-- Mark relevant points -->
  <g stroke="black" stroke-width="3" fill="black">
    <circle id="pointA" cx="100" cy="350" r="3" />
    <circle id="pointB" cx="250" cy="50" r="3" />
    <circle id="pointC" cx="400" cy="350" r="3" />
  </g>
  <!-- Label the points -->
  <g font-size="30" font-family="sans-serif" fill="black" stroke="none"
    text-anchor="middle">
    <text x="100" y="350" dx="-30">A</text>
    <text x="250" y="50" dy="-10">B</text>
    <text x="400" y="350" dx="30">C</text>
  </g>
</svg>
```



# Next: (2D) linear transformations



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \lambda_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \lambda_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

(a) scale

(b) shear x

(c) shear y

(d) rotate

(e) translate