Core Graphics and OpenGL ES

Dr. Sarah Abraham

University of Texas at Austin
CS329e
Spring 2020

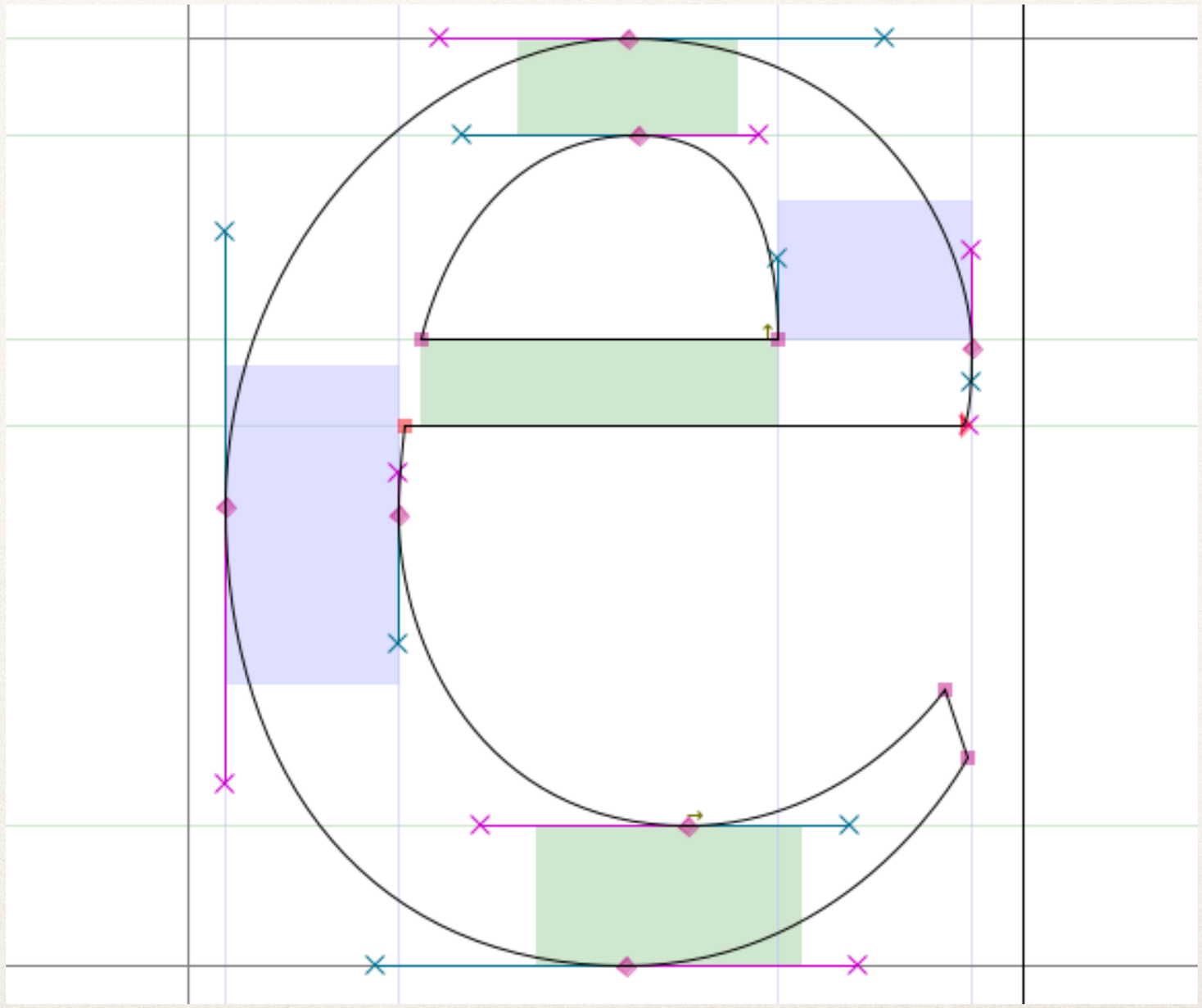
Core Graphics

- Apple's vector-drawing framework
  - Previously known as Quartz or Quartz2D
- Includes handling for:
  - Geometric data such as points, vectors, shapes etc
  - Functions for rendering pixels to screen

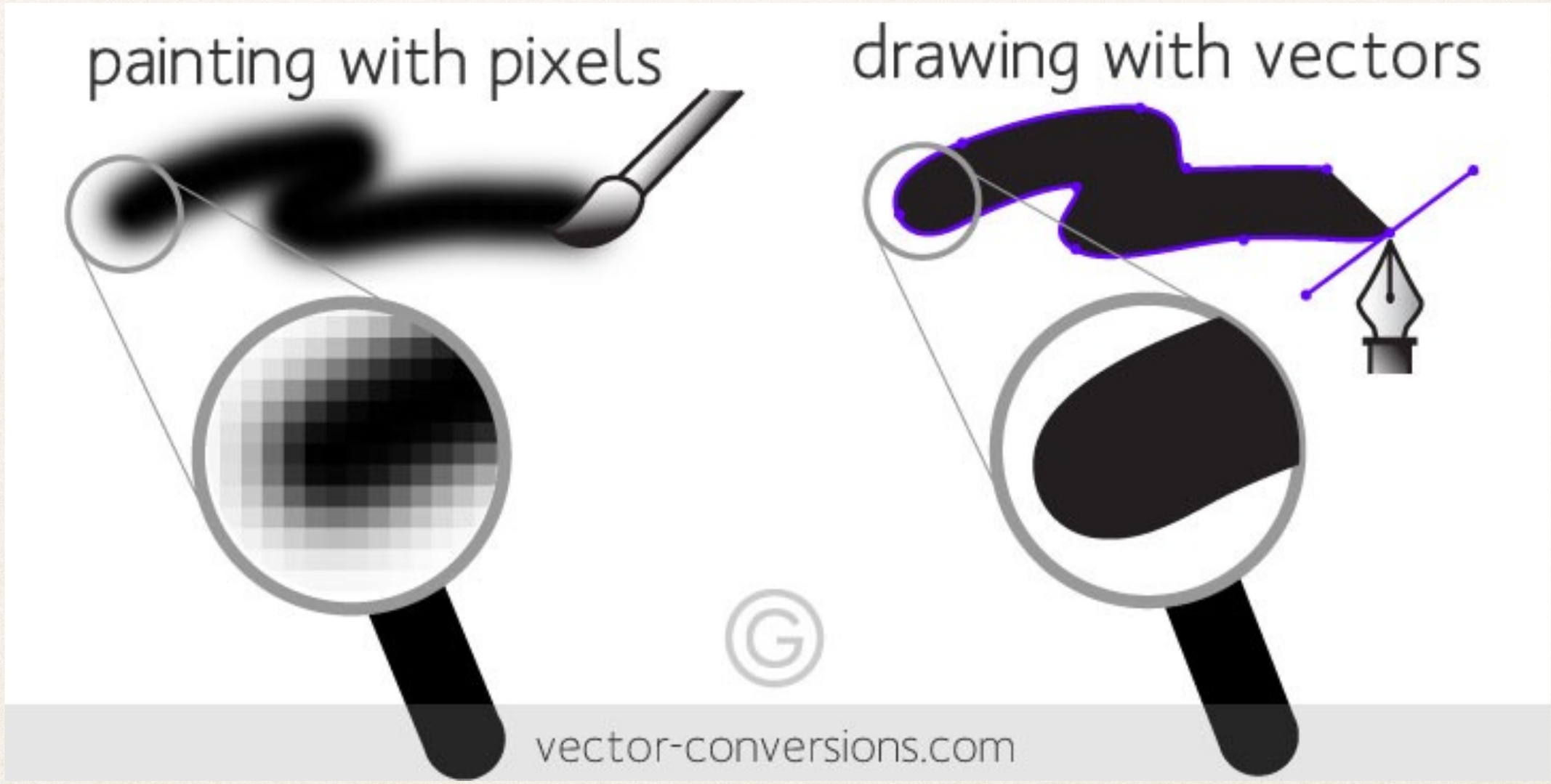
Vector Drawings

- Vectors defined mathematically (rather than by pixel)
  - Allows for continuous scaling and additional manipulation
  - More robust than bitmap (raster) graphics
- Built from geometric primitives like points, lines, curves, and shapes
  - Points define lines and curves, lines and curves define shapes etc

Vector Example



Vectorization vs Rasterization



- Vector Advantages:
  - Infinite scaling (not resolution dependent)
  - Smaller file size
  - Used most frequently for line-art with flat, uniform coloring
    - Logos, letterhead, fonts etc

- Bitmap Advantages:
  - Capture gradations and complex composition
  - A lot of detail at high resolution
  - Used most frequently for graphics used “at-resolution”
    - Photographs, scanned artwork, pixel-based art etc

Using Computer Graphics

- Graphics libraries require a graphics *context*
- Graphics context describes the "state" of world in which you are drawing

1. User tells graphics context how and where to draw
2. User specifies what to draw
3. Graphic context draws according to specification

What does the Graphics Context need to know?

- Graphic context identifies draw destination, coordinate system
  - Destination can be screen, printer, PDF file, etc
- Graphics context maintains global information about current drawing attributes
  - Transforms, line style, fill style, font style, etc
- Associated with a window and view in iOS

Example: CGRect

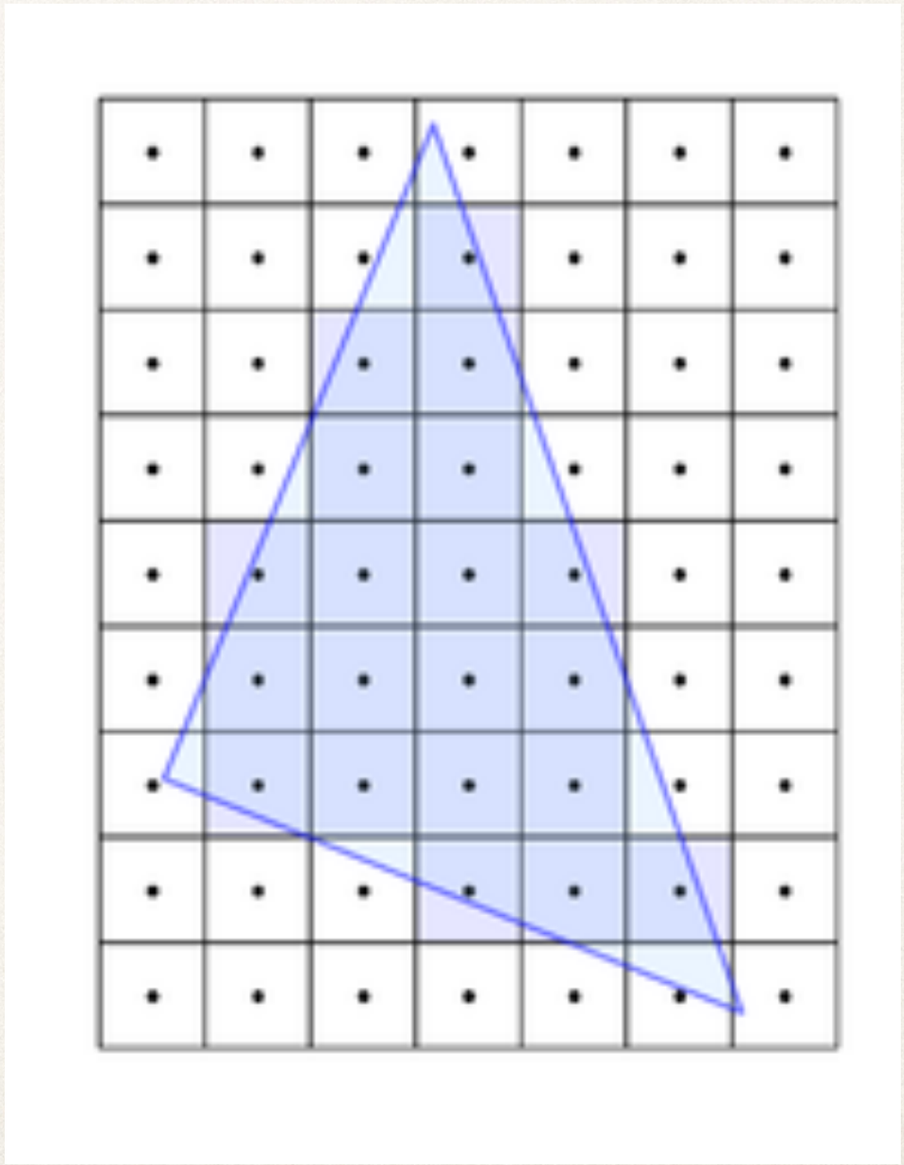
- `CGRect` objects define location and dimensions of rectangles
- `stroke()` and `fill()` functions control appearance of object's line and body properties
  - `UIColor.white.setFill()`
  - `rectangle.fill()`
- `.setFill()` defines fill color for current graphics context
- `.fill()` draws rectangle geometry with fill properties in current graphics context

Getting to the Screen

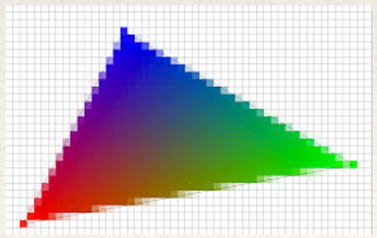
- In general, lots of things are being drawn to the screen via the graphics context
  - Multiple objects can overlap
  - Objects can have different levels of transparency
- How does the screen display all objects correctly?

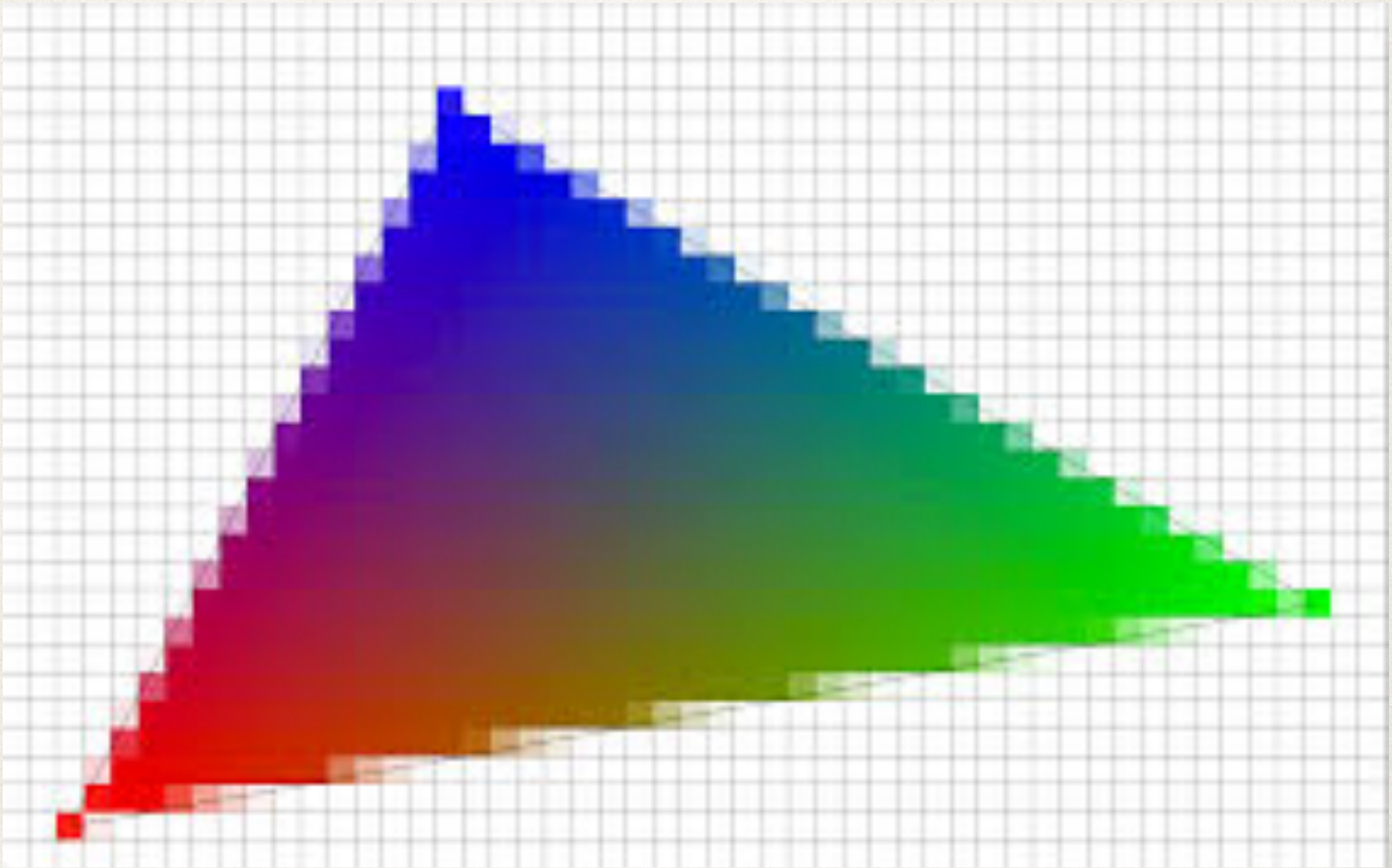
Frame Buffer

- Contains final pixel values (RGB) to be displayed on the screen



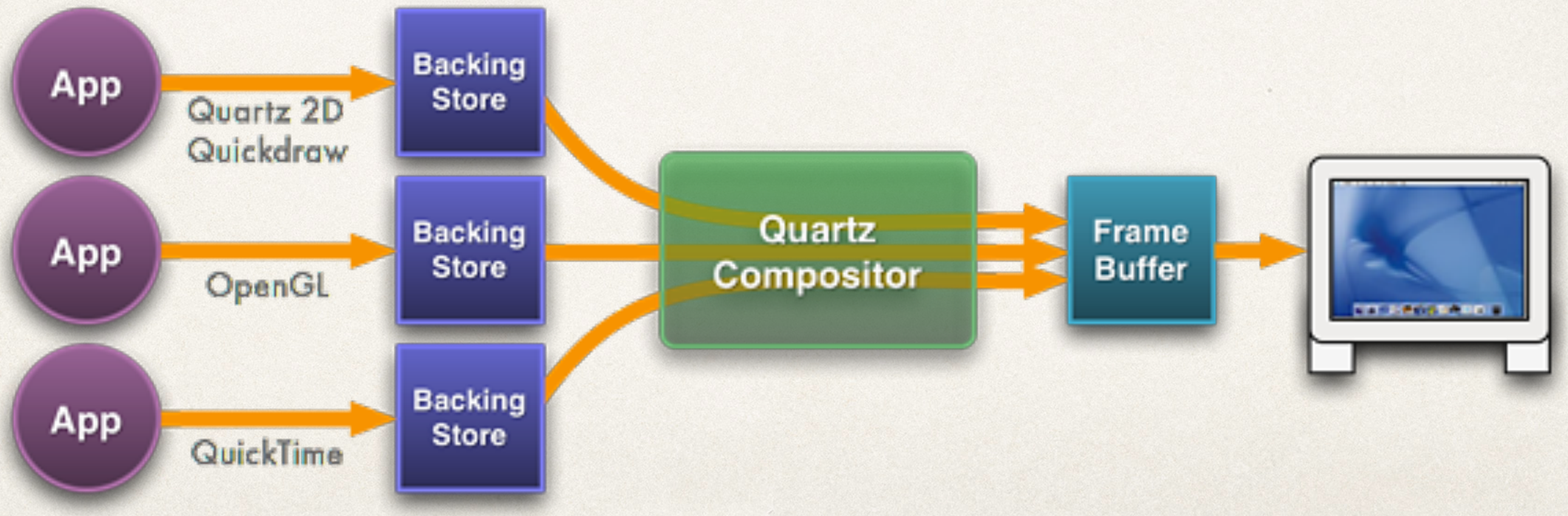
- Must account for:
  - Device aspect ratio and resolution
  - Anti-aliasing





Quartz Compositor

- Context passes 2D content to Quartz Compositor
- Quartz Compositor combines all context content into rasterized frame buffer for display



Core Graphics Framework

- Classes prefixed with `CG`
  - `CGRect`, `CGFloat`, `CGLayer` etc
- All CG calls executed in associated `CGContext`
  - Bitmap graphics, PDF graphics, OpenGL etc

Drawing Directly to Screen

- Create subclass of `UIView`
- Override `draw()` to include necessary CoreGraphics draw commands
- Associate custom `UIView` with desired ViewController
- Note: Core Graphics is part of the “V” in “MVC”
  - Should not affect your models or controllers

UIView draw()

- `draw()` is called whenever the View is updated
- **Never** call it directly
  - Use `setNeedsDisplay()` to manually request a screen refresh
- Can use a Timer from the ViewController if you want the screen to update continuously
  - Using OpenGL directly allows for continual display automatically

Drawing in Core Graphics

- Can define several types of renderers for specific tasks:
  - GraphicsRenderer, ImageRenderer, PDFRenderer
- Image and PDF Renderers optimize drawing image or PDF content to screen
- Graphics Renderers handles general draw calls
- `UIBezierPath` defines paths and curves for rendering

UIBezierPath

- Primary tool for customizing geometric paths and drawing properties within Core Graphics
- Easily reused and manipulated within the code
- Define as lines, ovals, arcs, rectangles, and arbitrary Bezier paths
- Also used for clipping and intersection tests

Defining Paths

- On UIBezierPath `init`, choose the path this object will represent
  - `init(rect: CGRect)`
  - `init(ovalIn: CGRect)`
  - `init(roundedRect: CGRect, cornerRadius: CGFloat)`
  - `init(arcCenter: CGPoint, radius: CGFloat, startAngle: CGFloat, endAngle: CGFloat, clockwise: Bool)`
  - `init(cgPath: CGPath)`

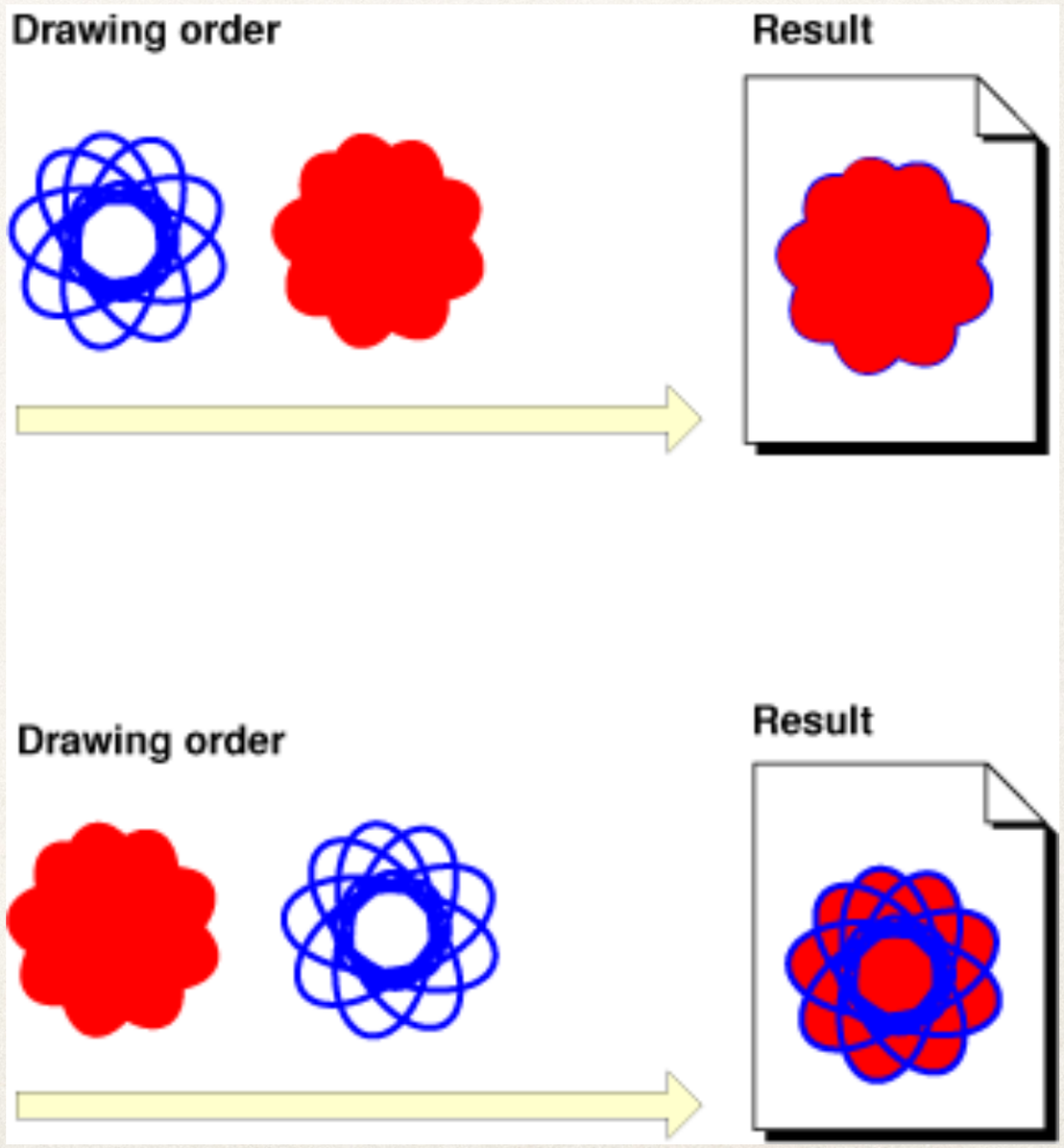
Drawing Paths

- Set drawing properties such as `lineWidth`, `lineCapStyle`, `lineJoinStyle`
- Draw enclosed region with `fill()`
- Draw stroke outline with `stroke()`
- Can also set blend modes and transparency of outline or body for more advanced graphics

UIBezierPath Demo

Order of Draw Calls

- Order matters when drawing in Core Graphics!
- Pixels cannot be changed once they're "painted"
- Must draw over existing pixels with new draw commands



Other Uses of Paths

- `addClip()` intersects object path's body with the current graphics context's clipping path to update object's path

- `contains(CGPoint)` determines if object path's body contains specified point

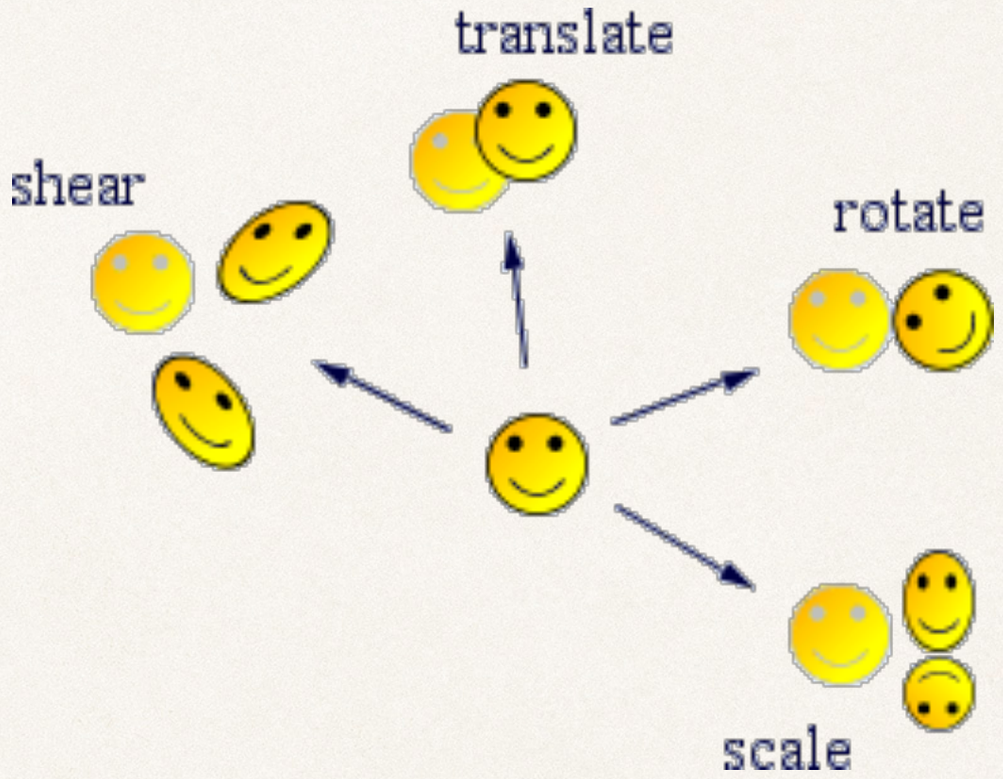
  - Allows for primitive hit detection

Instapoll Question: Overriding Views

- Which function in UIView do we override to change the view's default appearance?
  - `draw()`
  - `UIBezierPath()`
  - `stroke()`
  - `fill()`

What are Transformations?

Affine Transformations



(RichDoc Framework)

- Foundation of rendering in computer graphics
- Allows for manipulation of objects within a scene

Matrix Representation

- Represent a single vertex point p as a vector: $\begin{bmatrix} x \\ y \end{bmatrix}$
- Represent a 2-D transformation with matrix M: $\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
- Multiply p by M to apply the transformation: $\mathbf{p}' = \mathbf{M}\mathbf{p}$

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

Multiplication

- How do we multiply?

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

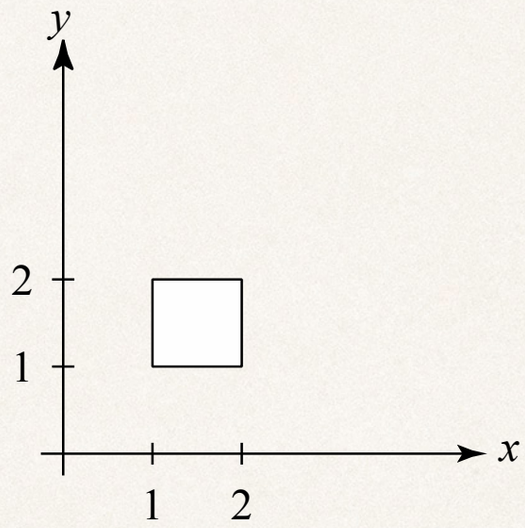
$$x' = ax + by$$
$$y' = cx + dy$$

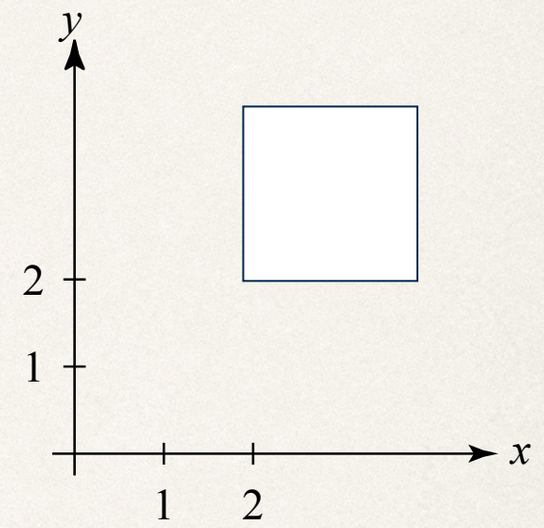
- What if we multiply by the identity matrix?

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{aligned} x' &= ax \\ y' &= dy \end{aligned}$$

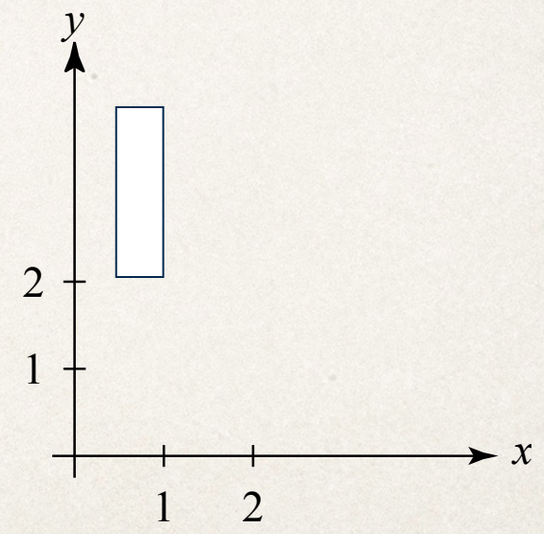
Scaling

- What happens with one of these matrices is applied to the square?



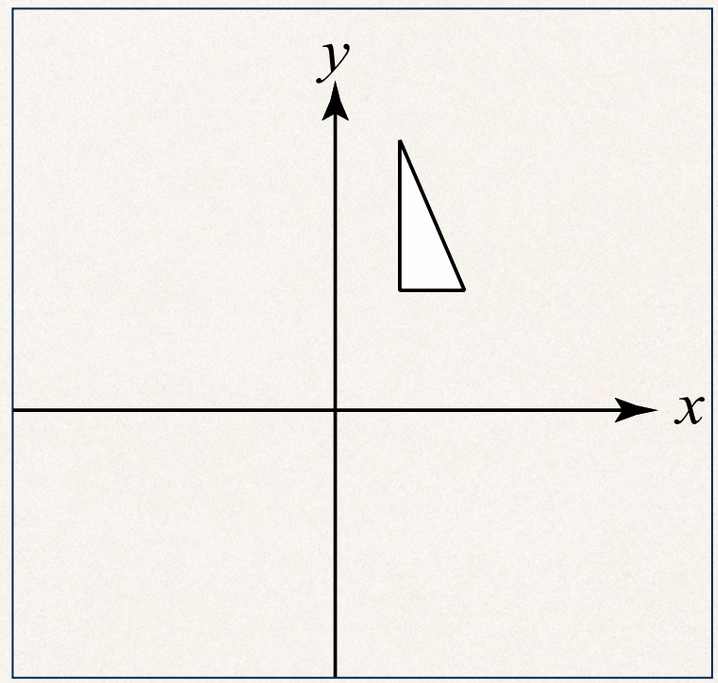


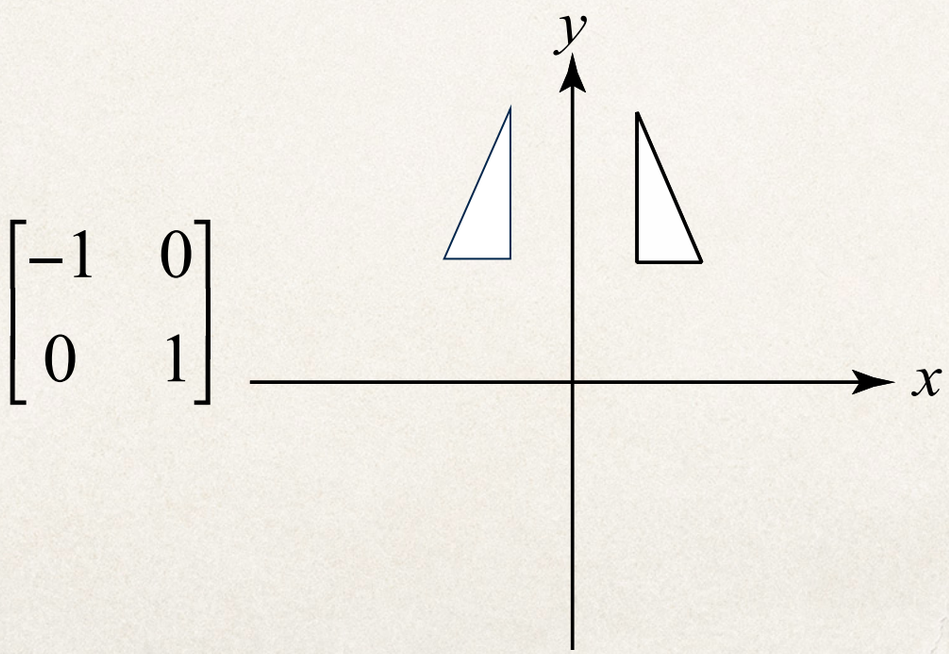
$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$



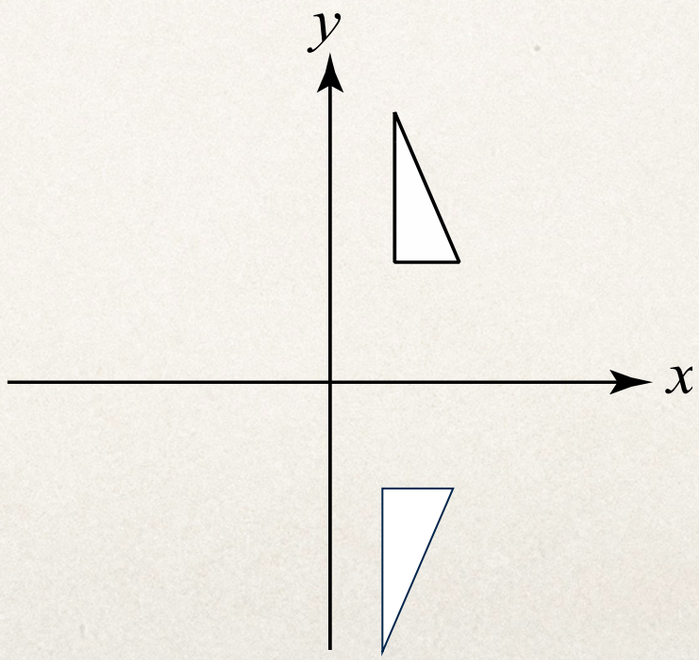
$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}$$

Reflection





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



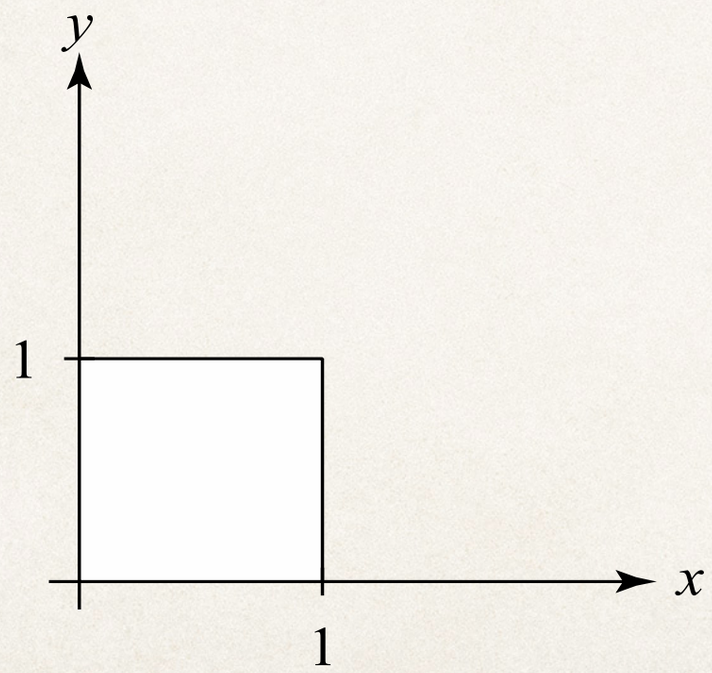
$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

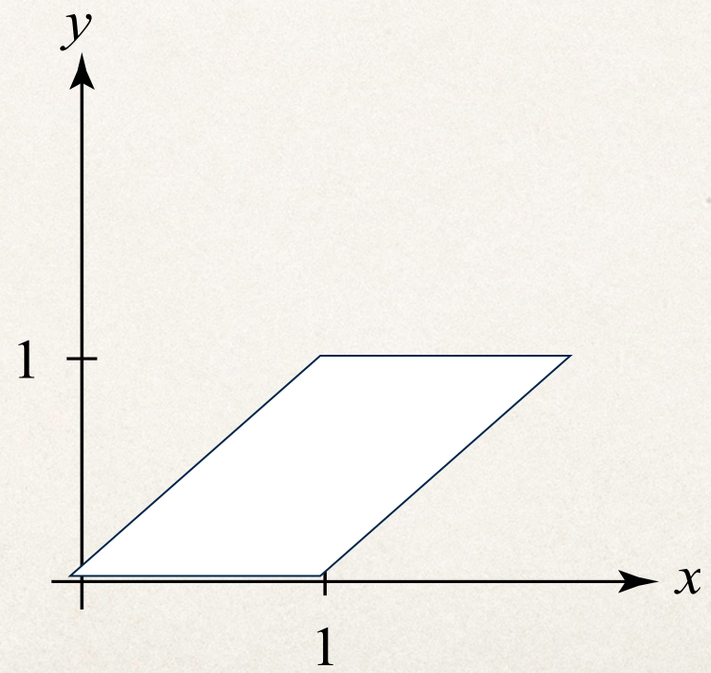
Shear

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

$$x' = x + by$$

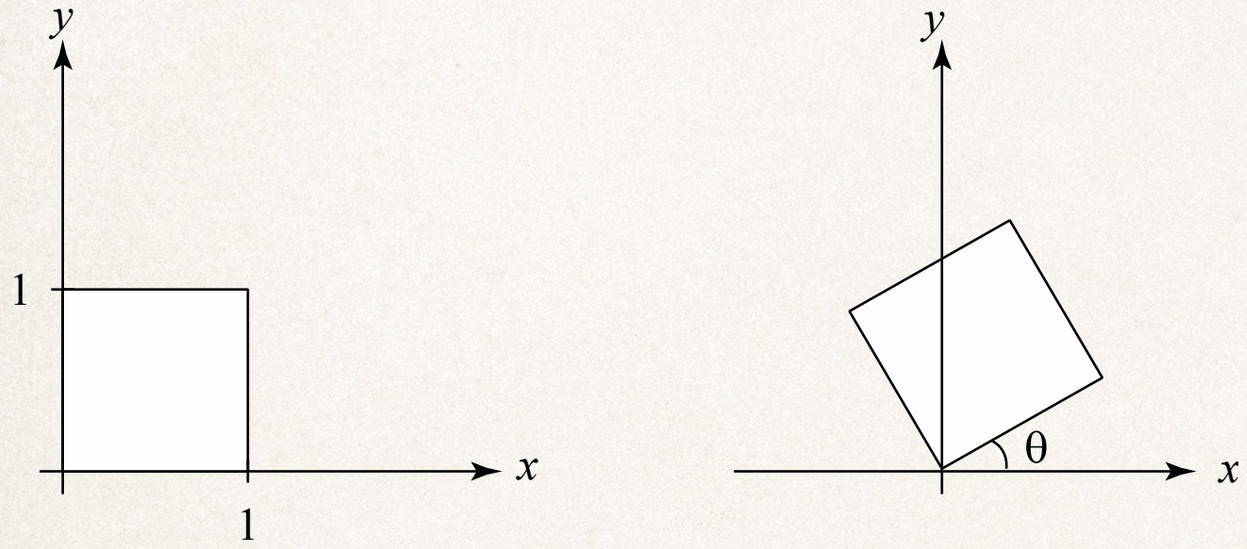
$$y' = y$$





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

Rotation



$$\begin{bmatrix}1\\0\end{bmatrix} \rightarrow \begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix}$$

$$\begin{bmatrix}0\\1\end{bmatrix} \rightarrow \begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\end{bmatrix}$$

$$M_R = R(\theta) = \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}$$

Linear Transformations

- All of these transformations are linear:
  - Scaling
  - Reflection
  - Shearing
  - Rotation
- What's missing?

Translation

- We want objects to move, or translate, through space

- Linear space (for linear transformations) has no notion of "position"

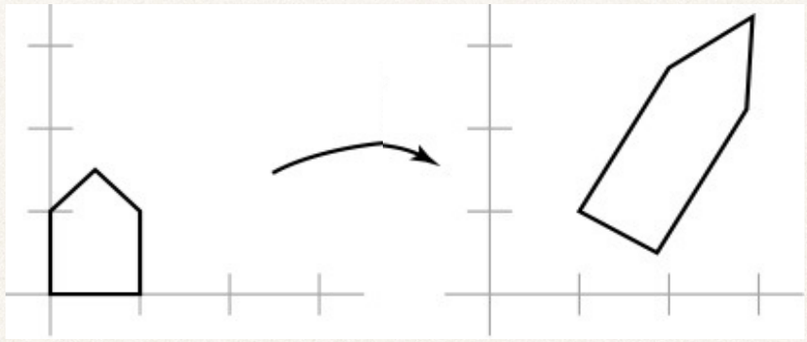
- Therefore affine space takes linear space and adds an "origin" point

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

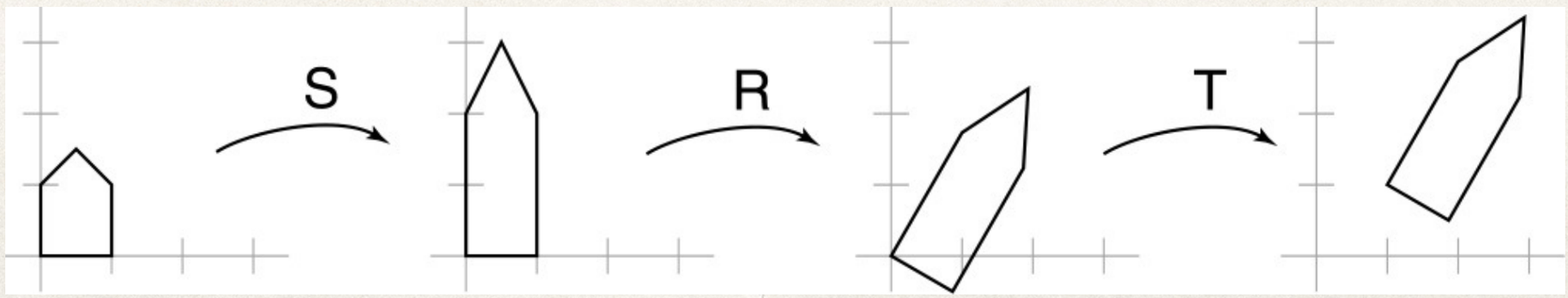
$$= \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Series of Transformations

- We can combine a sequence of transformations into one matrix to transform the geometric instance:



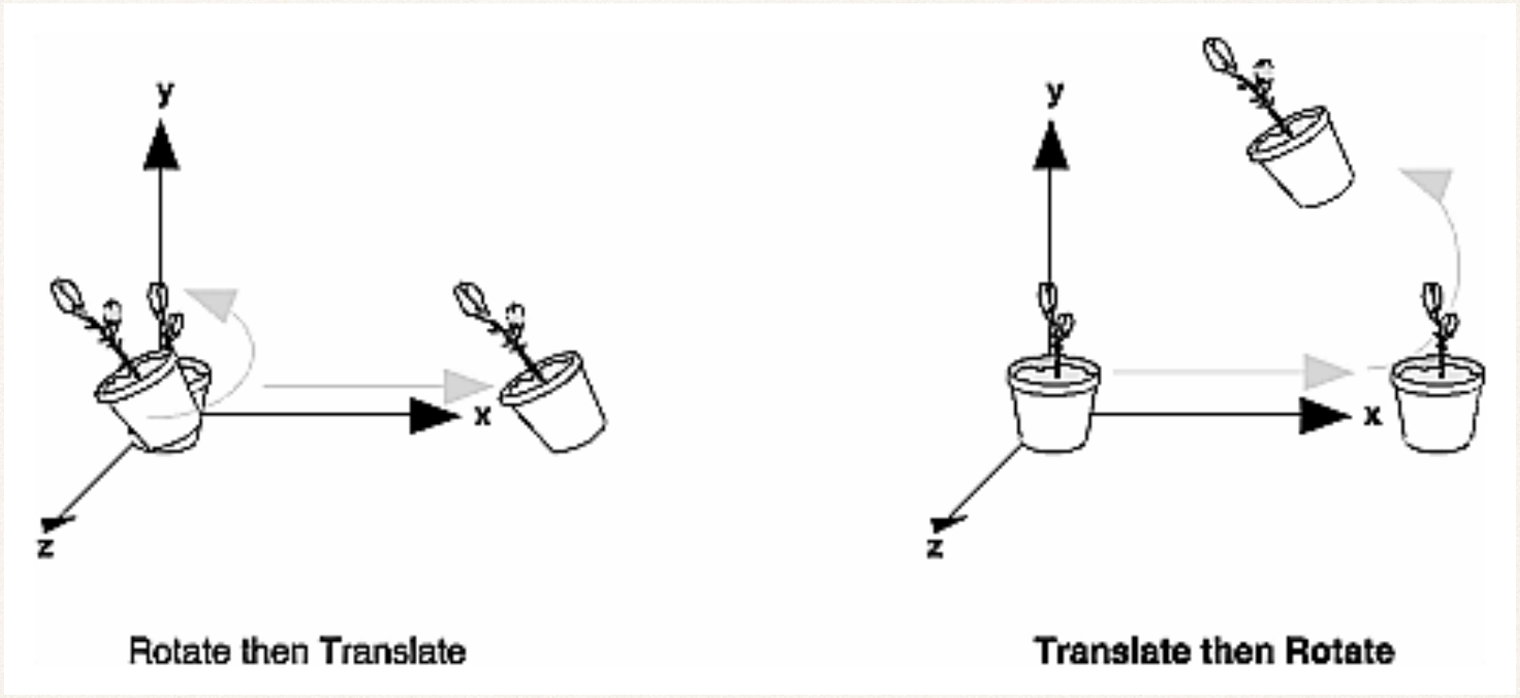
- But we can also think of this transformation as a series of simpler transformations:



Transformation Order

- Transformation order matters!
- Mathematical reason: transformation matrices do not commute under matrix multiplication (how we apply transformations)
- Intuitive reason: what happens when we rotate then translate versus translate then rotate?

Proper Transformation Order



- To rotate an object within its current position:
  - Scale -> Rotate -> Translate
- To rotate around a specific point:
  - Scale -> Translate -> Rotate

The Graphics Context

- The graphics context stores the state of the world
- State of the world includes current affine space stored as a sequence of matrix multiplications
- Necessary to move between affine spaces to perform animations

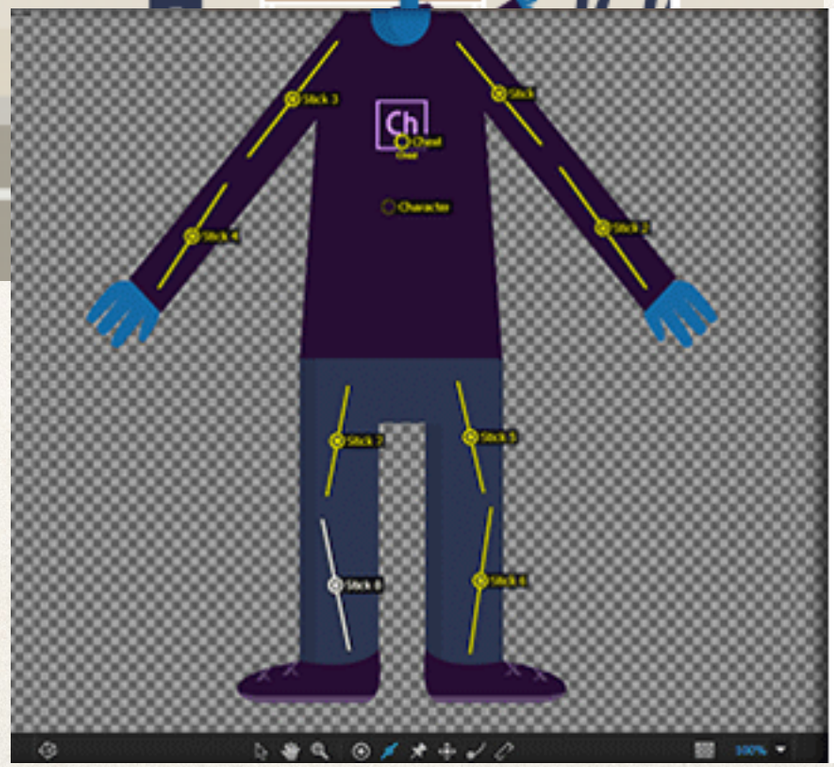
Drawing with Contexts

- Get the view's current context:

```
let context =
UIGraphicsGetCurrentContext()
```

- Contexts allow for:
  - Draw paths, rectangles, images, PDFs, gradients, etc
  - Set colors, color space, shadows, transformations, etc

Puppet Animation in 2D



Applies to 3D as well!

Transformations in Core Graphics

- Translate coordinate system origin to (tx, ty)
  - `context?.translateBy(x: CGFloat, y: CGFloat)`
- Scale coordinate system by sx and sy
  - `context?.scaleBy(x: CGFloat, y: CGFloat)`
- Rotate coordinate system by angle (radians)
  - `context?.rotate(by: CGFloat)`

Changing Contexts

- Necessary to save and restore the context's coordinate system after applying a transform
  - Transformations applied to underlying matrix that represents the coordinate space
  - After object is transformed, remove the transforms from the matrix so they don't affect additional objects
- `context?.saveGState()`
- `context?.restoreGState()`

Graphics Context Demo

Instapoll Question: Transformations

- Which of these is the standard order of transformations?

- Rotate -> Scale -> Translate
- Translate -> Rotate -> Scale
- Rotate -> Translate -> Scale
- Scale -> Rotate -> Translate

OpenGL ES

- OpenGL is standard open source graphics library for working on the GPU (graphics processing unit)
- OpenGL ES is for embedded systems (phones, tablets, consoles, etc)
  - Subset of OpenGL API
  - Optimized for simpler GPU architecture
  - Considerations for power and performance

Shaders

- Small, efficient programs that run on the GPU
  - Intended for high throughput (many simple operations in parallel)
- Designed to perform vertex transforms and apply colors to pixels (hence shaders!)
  - Works on models in 3D world space
  - Works on pixels in 2D screen space

Using OpenGL in Swift

- Under Targets -> BuildPhases -> Link Binary with Libraries add `OpenGLES.framework`
- Add `import OpenGLES` to file using OpenGL
- Use `EAGLContext` to manage the OpenGL rendering context
- Can also use `GLKit` for additional shader and math library support
- https://www.raywenderlich.com/5146-glkit-tutorial-for-ios-getting-started-with-opengl-es