iOS Dev Accelerator Week3 Day3

- Animation in iOS
- Custom View Controller Transitions
- Transforms

Homework Review

Animation in iOS*

*Sourced from Motion Design for iOS by Mike Rundle

Animating your app

- Thanks to iOS7's new design principles, there are really only 3 reasons to use animations in your app:
  - Transition: Smoothly animating from one visual state to the next helps the user understand what the app is doing and why.
  - Focus: Direct the user's attention to a specific aspect of the interface.
  - Delight: Make things look awesome and appealing.

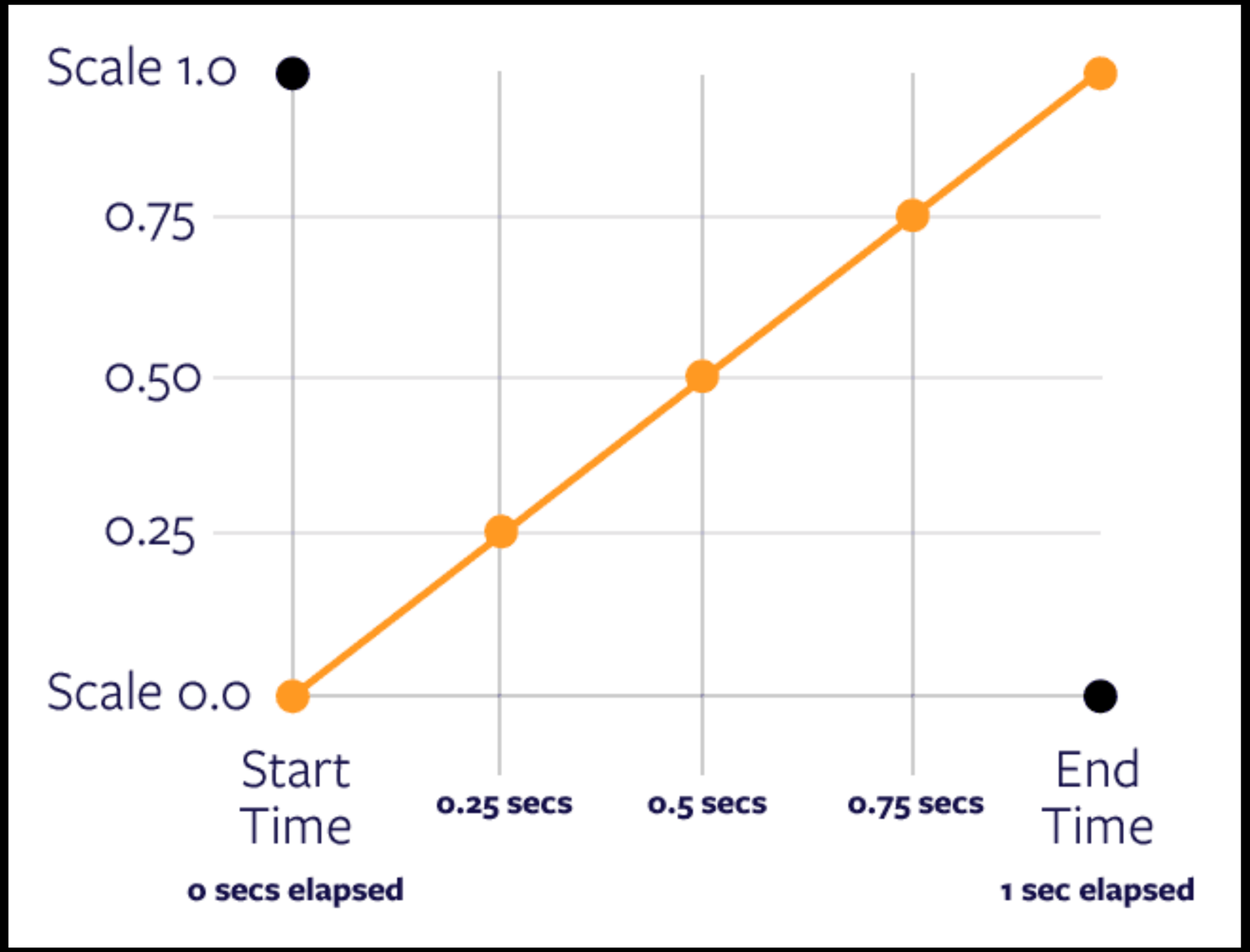
Planning out your animations

1. What are the initial properties of the item?
2. What are the final properties of the item?
3. How long should the animation take?
4. Whats happening to this item while it is animating?
5. What will happen once this item is done animating?

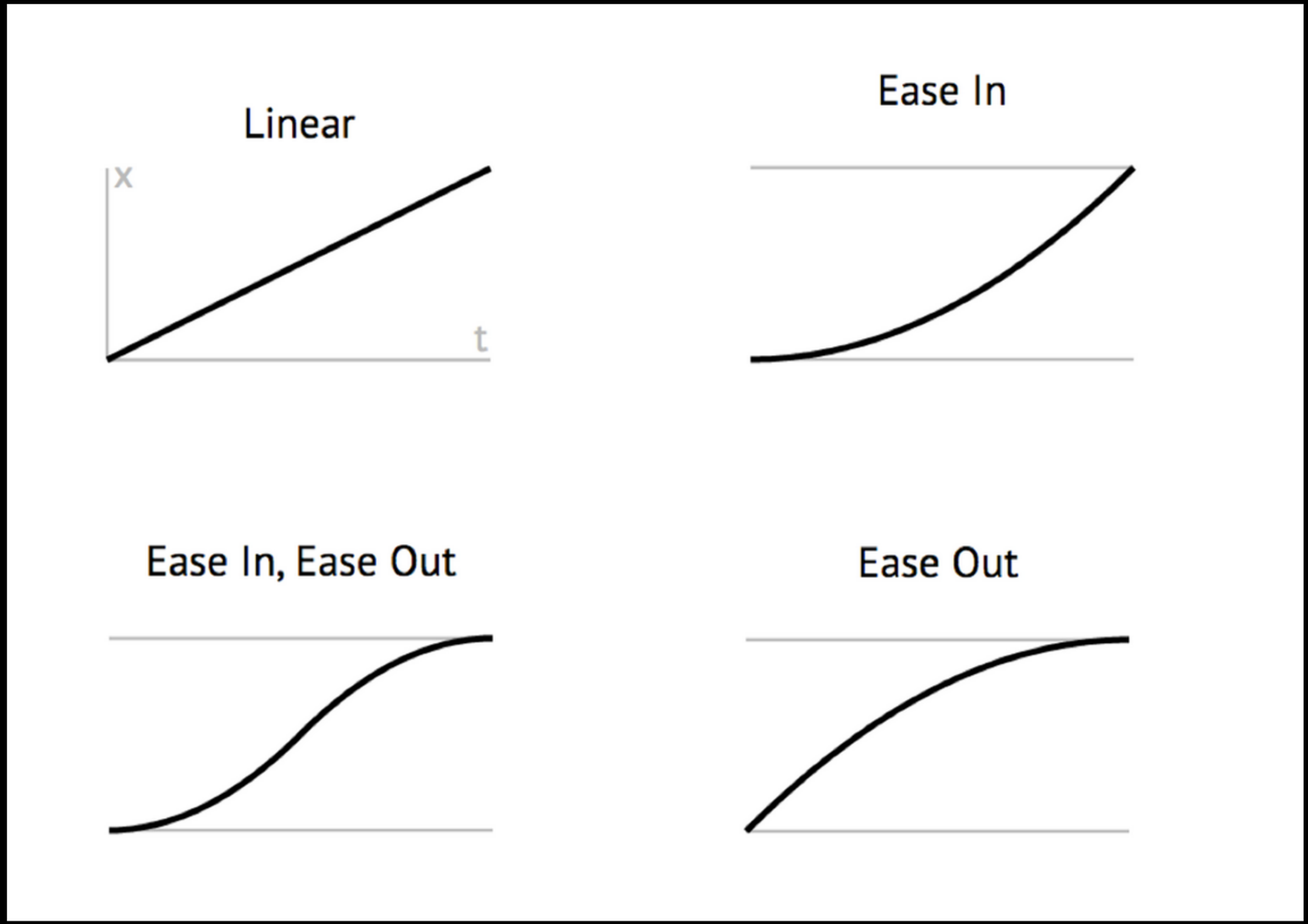
Properties that can be animated

- Position/Size/Frame: Changing the X and Y values of a views origin
- Opacity: Changing the alpha from in the range of 0 to 1
- Scale:Increase or decrease the size of a view. 1 represents its normal size.
- Color: transition one color value to another.
- Rotation: Rotate a view by radians.
- 3D Transform: Rotate the third dimension of a view
- Constraints!

Timing is everything



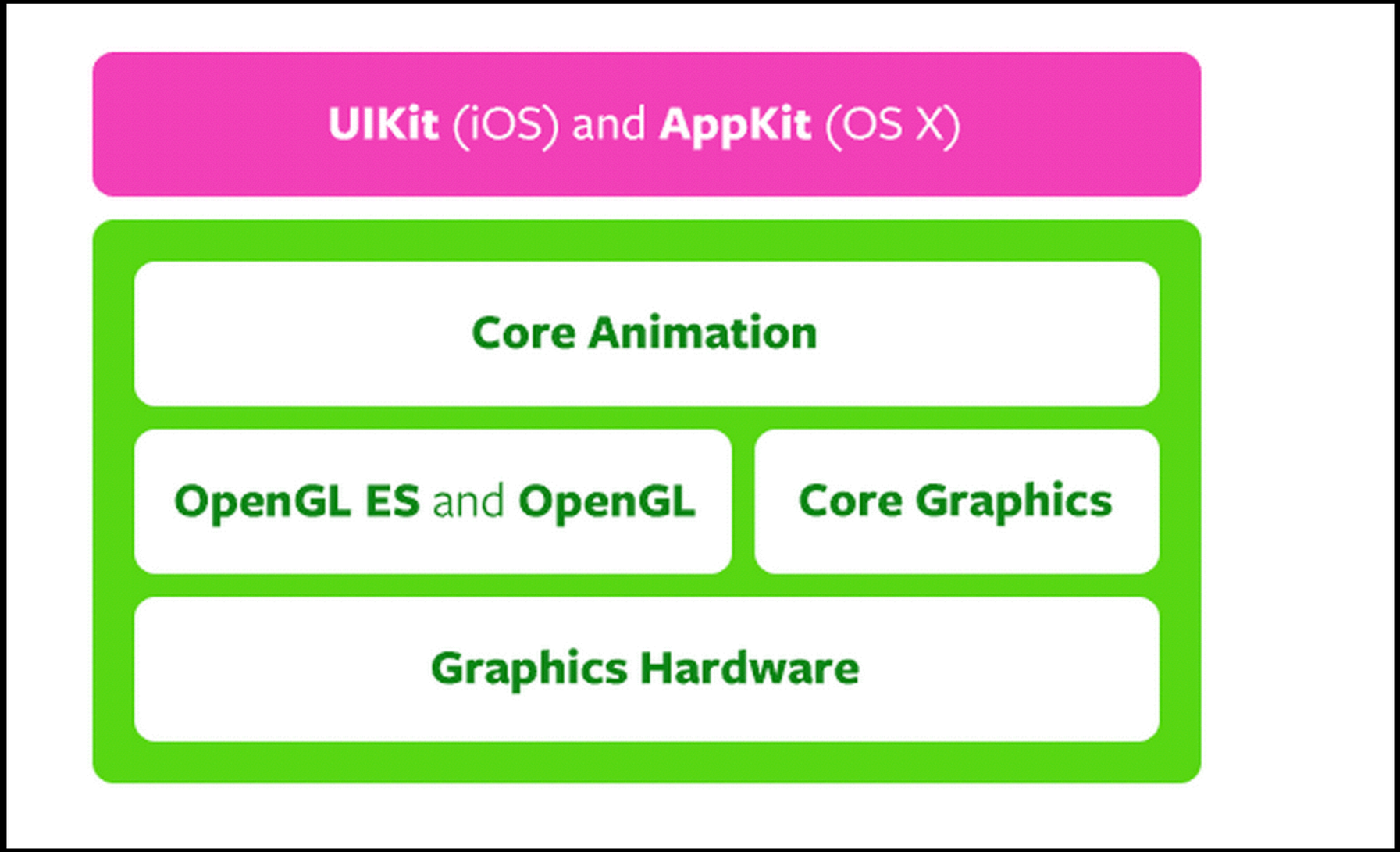
Standard animation curves



Core Animation

- “Core animation is an animation and graphics compositing framework made for speed and efficiency”
- Despite it having the word animation, Core Animation is actually in charge of all rendering on screen. Not just animations.
- Core Animation uses CALayer objects as its main unit of work. UIView objects are just thin wrappers for CALayer’s.
- Layers can be arranged in a hierarchy just like UIView’s, in fact you can build your entire interface using just layers if you wanted.

Core Animation



Built-in Animation Techniques

- Apple provides a 3 different ways to create animations:
  - Adding CAAnimation and its subclasses to a layer (Advanced)
  - Simple block/closure based system using class methods on UIView (Fairly straight-forward)
  - Key-frame animations, which is just a wrapper built around the lower level CAAAnimation method. Uses block/closure syntax as well.

Block/Closure Based

- There are a couple different class methods for the block based animation system.
- One does not have a completion block, all the others do.
- Two of them provide a parameter for a delay before the animation fires.
- One provides an options parameters where you specify the curve type to use (ease in, ease out, etc)
- One provides everything above, plus spring dampening!

Spring animations

- “The type of a motion being used to generate a nice, springy-feeling animation is typically modeled after a damped harmonic oscillation.”
- The key properties of a spring based of a spring’s motion are:
  - Mass: the weight or heft of the object
  - Stiffness: how difficult it is to stretch out the spring
  - Damping: the restrictive force or friction pushing against the object

Demo

Key Frame Animations

- Key frame animations allow to you make animations that start and stop at different times, all relative to one duration interval.
- To start a set a keyframe animation, you first create a parent block that all the key frame animations will be created in. This parent block is given a duration, for which all the key frame animations will operate by.
- Each key frame animation is setup with a relative start and stop time inside the parent block.
- The parent block can also be given a set of options for specific animation scenarios.

Key Frame Example

- The duration of the parent is 2 seconds here. The relative duration of the child animations is 0.5. So they will each take 1 second to run since 2 x 0.5 = 1!

Parent Block

Child KeyFrame Animation

Child KeyFrame Animation

```
UIView.animateKeyframesWithDuration(2.0, delay: 0.0, options:
    UIViewKeyframeAnimationOptions.AllowUserInteraction, animations:
    { () -> Void in

    UIView.addKeyframeWithRelativeStartTime(0.0, relativeDuration:
        0.5) { () -> Void in
        self.redBox.frame.origin = CGPoint(x: 100, y: 300)
    }
    UIView.addKeyframeWithRelativeStartTime(0.5, relativeDuration:
        0.5) { () -> Void in
        self.redBox.frame.origin = CGPoint(x: 300, y: 300)
    }


}) { (Finished) -> Void in
    self.redBox.backgroundColor = UIColor.orangeColor()
}
```

Demo

Transforms

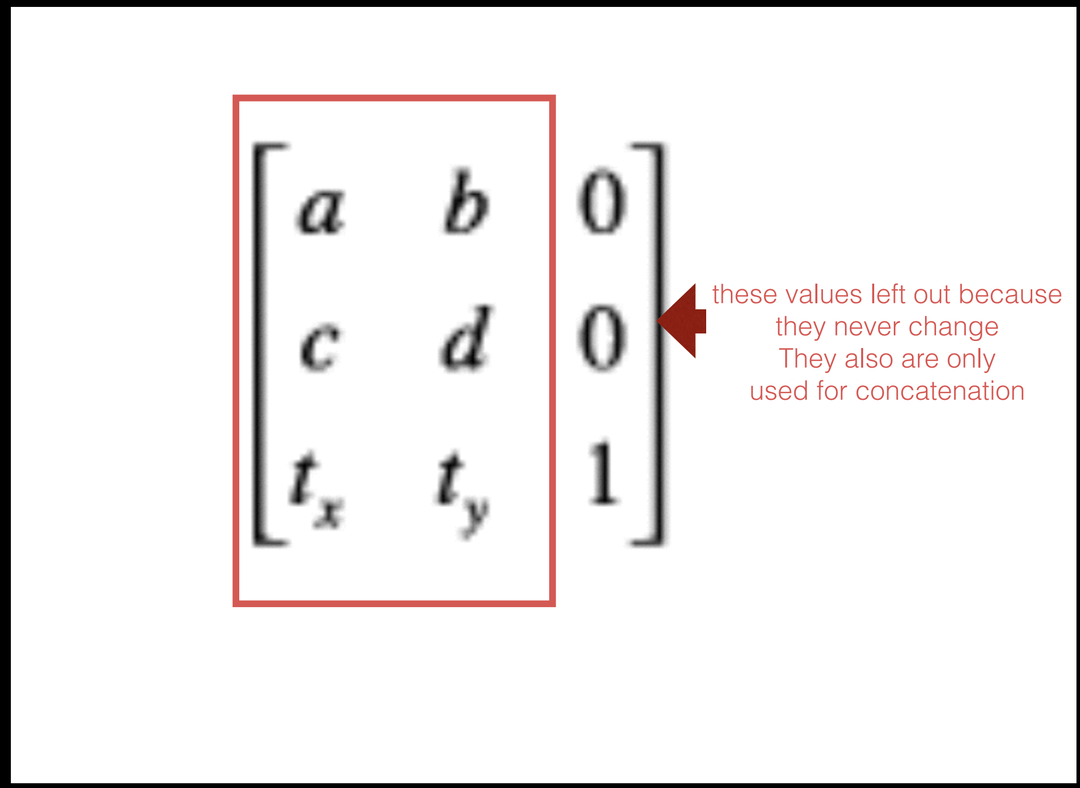
Transform

- Transforms can be applied to views and layers to translate (move), scale, rotate, and make a number of other changes to themselves.
- Every view has a .transform property which has a type of CGAffineTransform. Its a struct.
- Transforms are represented my matrices.
- You can think of them as two dimensional array of numbers.

CGAffineTransform structure:

```
struct CGAffineTransform {
    CGFloat a;
    CGFloat b;
    CGFloat c;
    CGFloat d;
    CGFloat tx;
    CGFloat ty;
};
```

The data structure
we use to represent it



An actual affine transformation matrix

Identity Matrix

- Every view (and layer) starts out with their transform set to the identity matrix:

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

- If a view's transform is set to the identity matrix, then we know no transforms have been applied to it.
- For a view that has a transform that is set to the identity matrix, the view will be drawn based only on its frame & bounds.
- Setting a view's transform back to the identity matrix will undo any transforms applied to it!

Translation

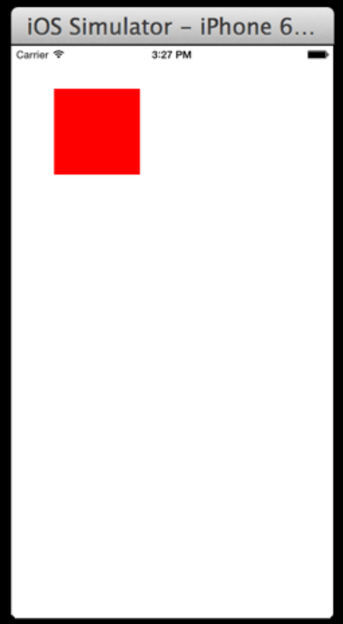
- Translation is just a fancy word for moving.
- If you a translate a view by 10,10 you just moved it 10 to the right and 10 down.
- This is the matrix that represents a translation operation:

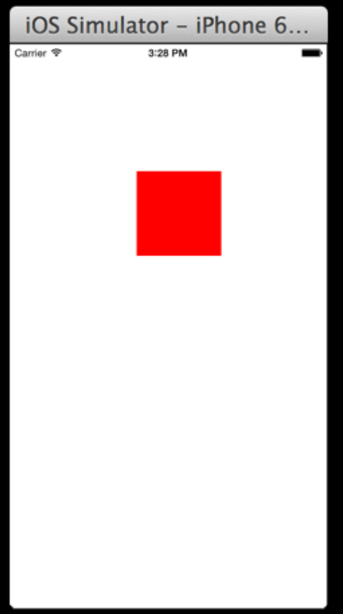
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

- So the equations that CoreGraphics uses to calculate the result are:
  - new X = old x + t_x
  - new Y = old Y + t_y

Translation Example

- CoreGraphics provides relatively simple functions for all the primary transforms.
- Here is an example of using the translation function:





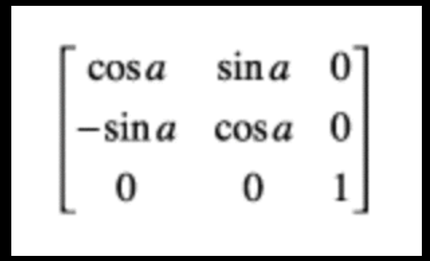

```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
```

```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformTranslate(redView.transform, 100, 100)
```

*Applying transforms is cumulative, so doing another translate just adds to the last one

Rotation

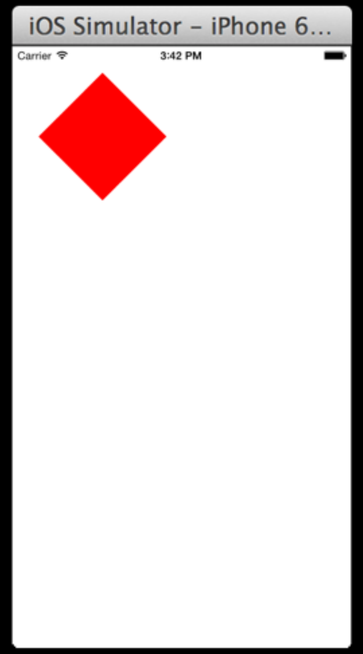
- Rotation is another of the primary transforms you can apply to views and layers.
- CoreGraphics provides the CGAffineTransformRotate() function that takes in two parameters:
  - the existing transform matrix
  - the angle or rotation in radians (Pi * degrees / 180.0)
- This is the matrix that describes rotation operations:



- This means the equations CoreGraphis uses to calculate the results are:
  - new x = old X * cos a - old Y sin a
  - new y = old X *sin a - old Y *cos a
- I hope you were paying attention in high school

Rotation Example

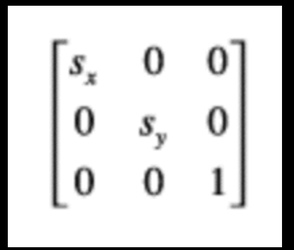
- Here is an example of rotating our red view 45 degrees clockwise




```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformRotate(redView.transform, CGFloat(M_PI * 45 / 180.0))
```

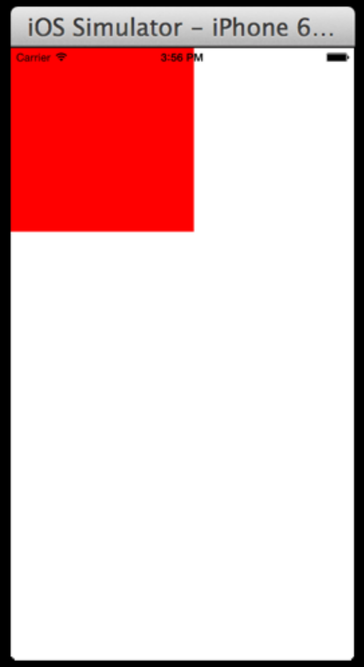
Scaling

- Scaling is another primary transform that is simply changes the size of the view.
- CoreGraphics provides the CGAffineTransformScale function that takes in 3 parameters:
  - the transform matrix we are applying the scale to
  - the factor by which to scale the x-axis
  - the factor by which to scale the y-axis
- Here is the matrix that describes the scaling operation:



Scaling Example

- Here is an example of double our red view's size:




```
var redView = UIView(frame: CGRect(x: 50, y: 50, width: 100, height: 100))
redView.backgroundColor = UIColor.redColor()
self.view.addSubview(redView)
redView.transform = CGAffineTransformScale(redView.transform, 2.0, 2.0)
```

Demo

Custom View Controller Transitions

Custom View Controller Transitions

- Debuted with iOS7
- Gives you complete control over the animation while you transition from one VC to another.
- You can even create interactive transitions that are driven by gestures.

Workflow

1. Create an animation controller - This can be any class that conforms to UIViewControllerAnimatedTransitioning protocol. This class will perform the actual animation.
2. Set the transitioning delegate - When you are about to present a view controller, you need to set its transitioning delegate. This is usually just the presenting view controller.
3. Return the animation controller when the transitioning delegate is asked for it.

Animation Controller

- Your animation controller can be any class that conforms to UIViewControllerAnimatedTransitioning protocol
- This protocol has 2 required methods:
  - transitionDuration(transitionContext : UIViewSearchControllerContextTransitioning)
  - animateTransition(transitionContext : UIViewControllerContextTransitioning)

`transitionDuration()`

- A simple method that will return how long the transition will take.
- This should always match up with the total duration of the animations you create in the next method.

animateTransition()

- This method is where you actually implement the animations.
- It has one parameter passed in, the transitionContext.
- The transitionContext gives you access to the both the presenting view controller (fromViewController), and the presented view controller (toViewController:

```
 //getting both the view controller we are presenting from (fromVC) and the one we are
presenting (toVC)
 let fromVC = transitionContext.viewControllerForKey(UITransitionContextFromViewControllerKey)
 let toVC = transitionContext.viewControllerForKey(UITransitionContextToViewControllerKey)
```

animateTransition() cont.

- The transitionContext also provides you with something called the containerView
- The containerView essentially acts as the super view for both the presenting view controller's view and the presented view controllers view.
- UIKit automatically adds the view of the presenting view controller, think of that as your starting state of the transition.
- You then add the view of the toVC to the containerView, and then animate it moving onscreen in some cool way
- Once the animation is complete, be sure to call completeTransition() on the transitionContext.
- You don't have to clean up the containerView, its all done for you once your animations are complete by calling the completeTransition()

TransitionDelegate

- The transitionDelegate is mostly just responsible for providing the animation controller at the appropriate time.
- It does this by implementing this method:

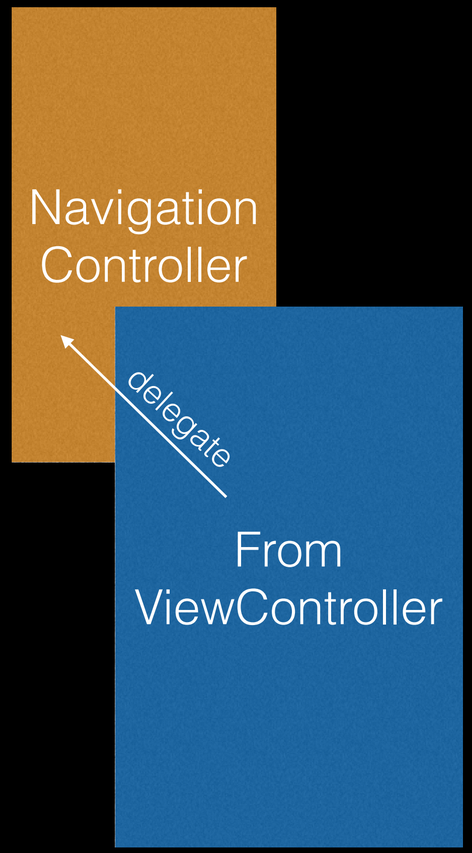
```
func animationControllerForPresentedController(presented: UIViewController, presentingController
 presenting: UIViewController, sourceController source: UIViewController) ->
 UIViewControllerAnimatedTransitioning? {
    return self.animationController
}
```

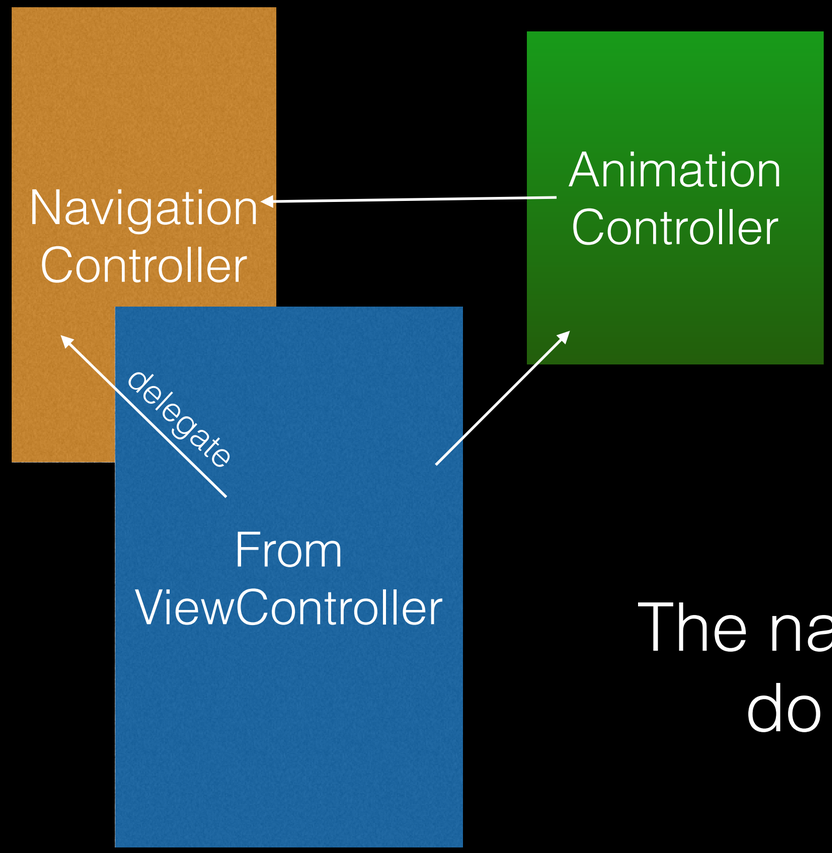
- If you have multiple transitions/segues wired up to this view controller, you would need to inspect the presented view controller to see which one is being presented on screen, and then return the appropriate animation controller

With Navigation Controller

- You can easily create custom transitions for view controllers inside of a navigation stack as well.
- In this case, the navigation controller's delegate will provide the animation controller.
- UINavigationControllerDelegate has a method you will need to implement in order to give it to the animation controller at the appropriate time:

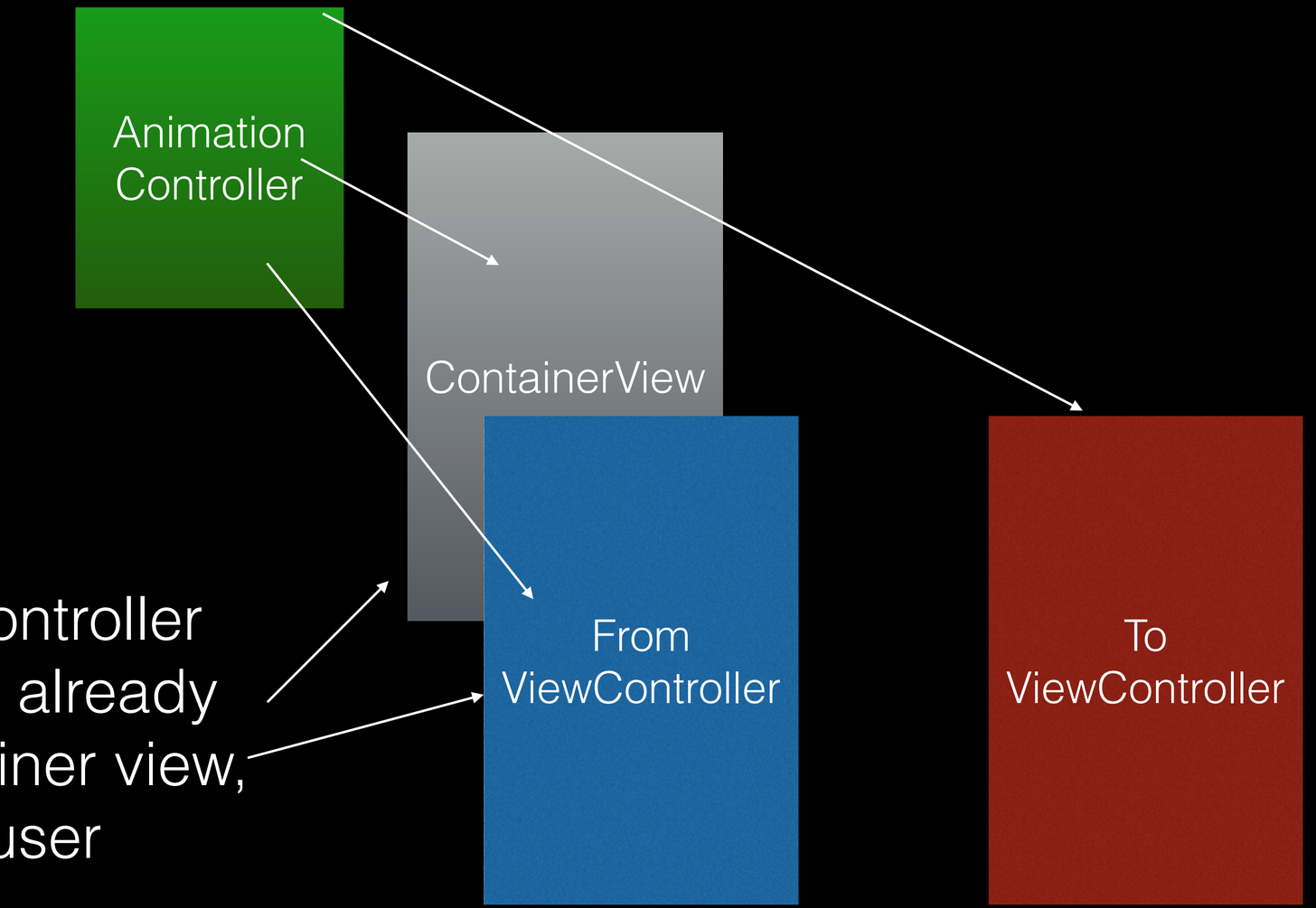
```
//MARK: UINavigationControllerDelegate

func animationControllerForPresentedController(presented: UIViewController, presentingController
    presenting: UIViewController, sourceController source: UIViewController) ->
    UIViewControllerAnimatedTransitioning? {
    return self.animationController
}
```


Navigation
Controller
delegate
From
ViewController



The navigation controller sees its about it to do a transition and asks its delegate for an animation controller to animate the transition

The animation controller takes over from here.
The transition context provided by it has
references to the To and From VC and also the container
View where the transition will take place



The From View Controller
starts with its view already
added to the container view,
visible to the user

Demo