Core Graphics & Animation

iOS App Development
Fall 2010 — Lecture 17

Questions?

Announcements

- Assignment #5 out later this week
  - Last of the short assignments

Today's Topics

- Custom UIViews
  - -drawRect:
- Core Graphics
  - Basics & terminology
  - Graphics contexts
  - Drawing shapes, images & text
- Core Animation
  - Layers

Notes

- I'm showing the relevant portions of the view controller interfaces and implementations in these notes
- Remember to release relevant memory in the -dealloc methods — they are not shown here
- You will also need to wire up outlets and actions in IB
- Where delegates or data sources are used, they too require wiring in IB

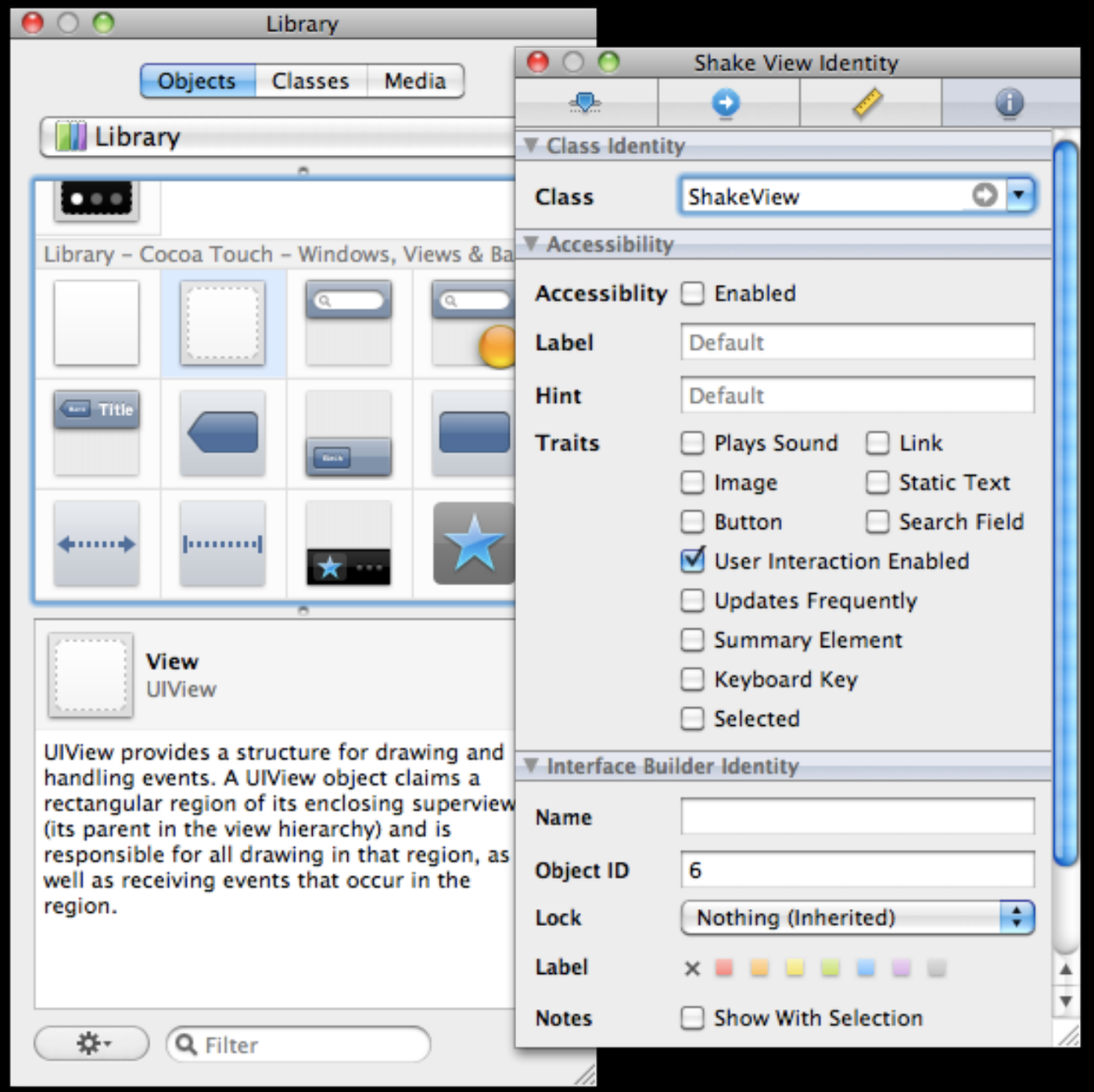
Custom UIViews

UIView Recap

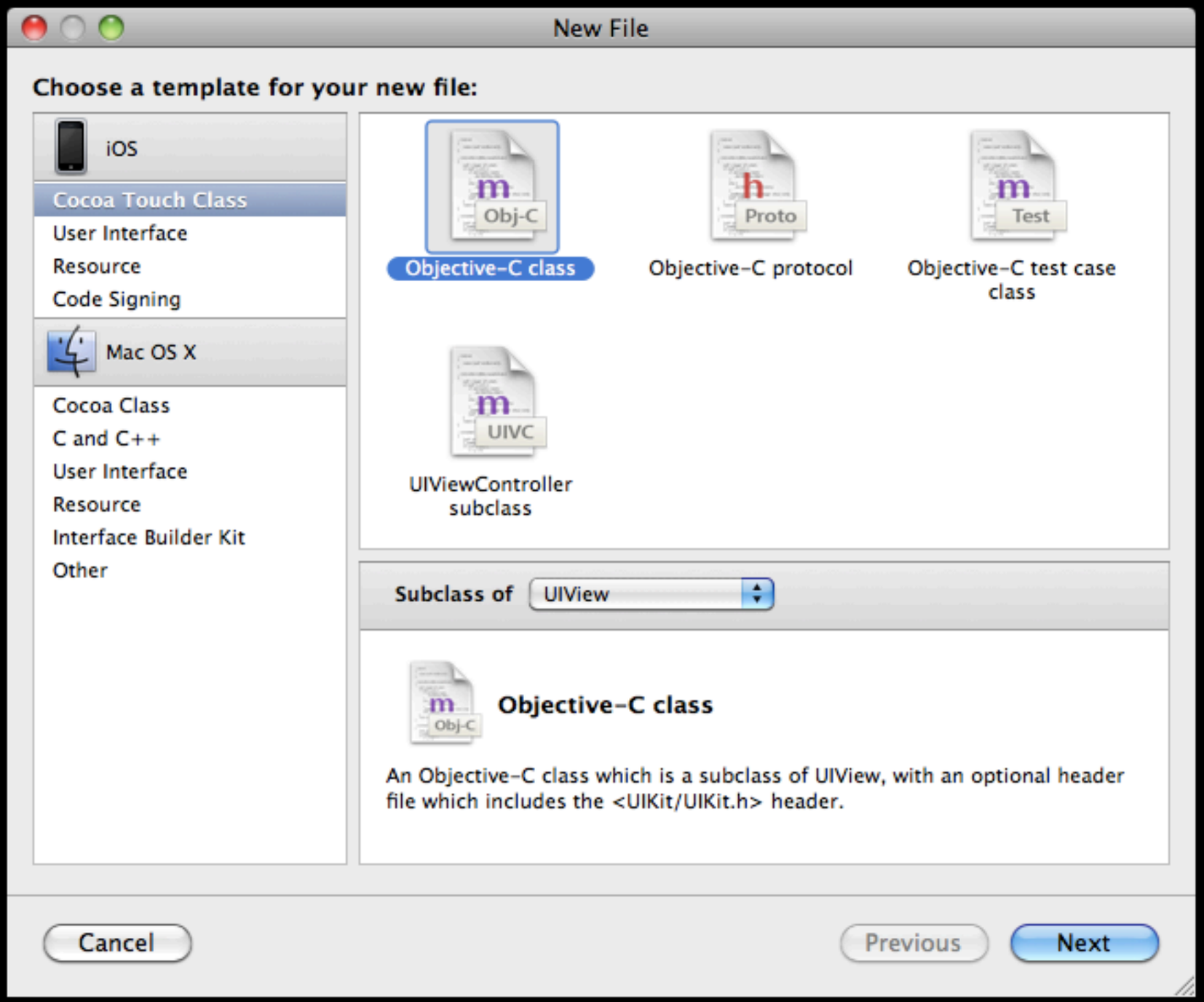
- Rectangular area on screen
- Draws content
- Handles events
  - Subclass of UIResponder
- Views are arranged hierarchically
  - Every view has exactly one superview
  - Every view has any number (zero or more) subviews

Creating Views

- Thus far we've pretty much used "off the shelf" components from IB
- Drag out existing high-level views or controls (subviews)
- We looked at using our own custom UIView subclass last class for handling events



Subclassing UIView



Subclassing UIView

```
#import "CustomView.h"

@implementation CustomView

- (id)initWithFrame:(CGRect)frame {
    if ((self = [super initWithFrame:frame])) {
        // Initialization code
    }
    return self;
}

/*
// Only override drawRect: if you perform custom drawing.
// An empty implementation adversely affects performance during animation.
- (void)drawRect:(CGRect)rect {
    // Drawing code
}
*/

- (void)dealloc {
    [super dealloc];
}

@end
```

UIView's -drawRect:

- If we want to do drawing in a custom UIView subclass, we need to override UIView's -drawRect: method

```
- (void)drawRect:(CGRect)rect;
```

- If not overridden, then backgroundColor is used to fill
- The rect argument is the area to draw
  - May be entire view
  - May be some portion of the view

UIView's -drawRect:

- The -drawRect: method is invoked automatically by UIKit
  - Do not call it directly
- If a view needs refreshing call UIView's -setNeedsDisplay method to trigger a re-paint

```
- (void)setNeedsDisplay;
```

Using Basic UIKit Methods

- UIKit offers very basic drawing functionality...

```
void UIRectFill(CGRect rect);
void UIRectFrame(CGRect rect);
```

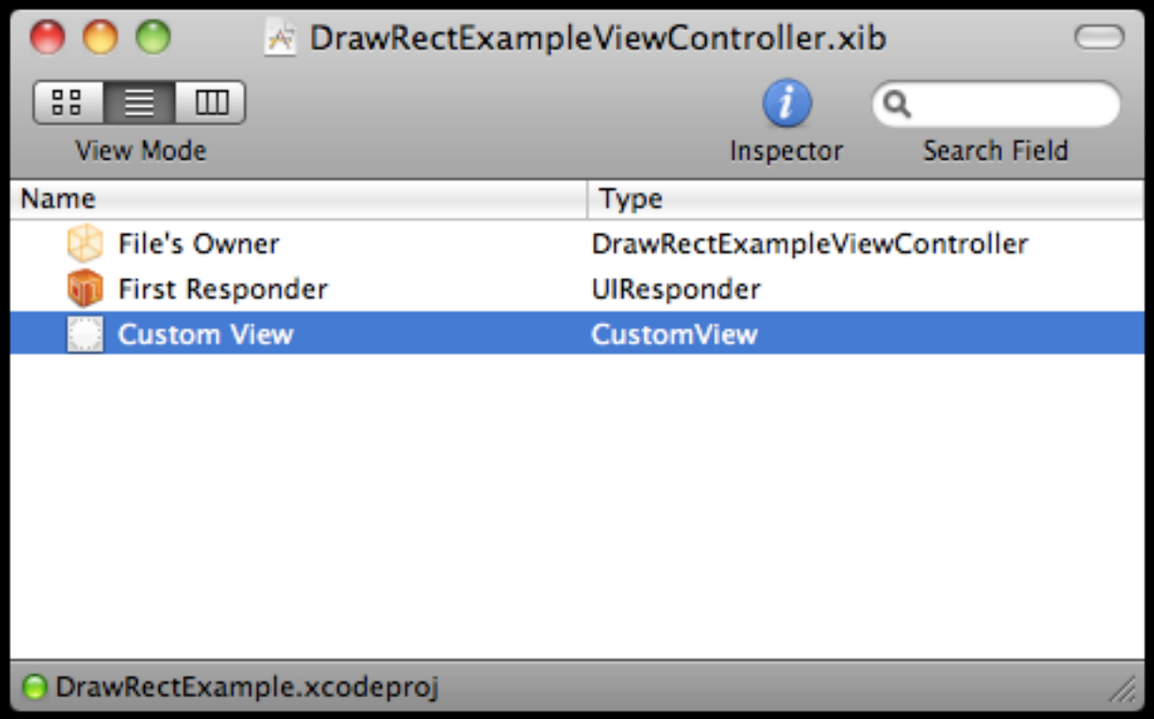
- For example, we could paint our entire view with the following -drawRect: implementation in our custom view class...

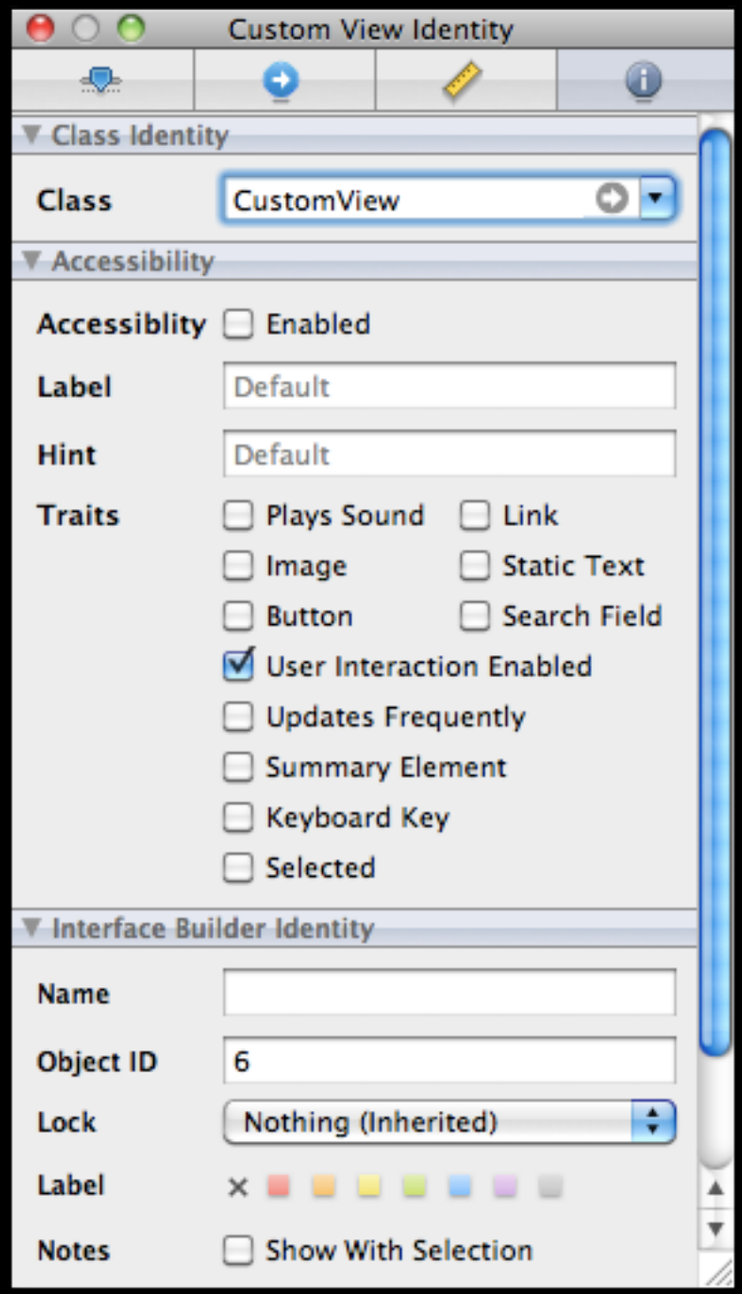
```
- (void)drawRect:(CGRect)rect {
  [[UIColor purpleColor] setFill];
  UIRectFill(rect);
}
```

Subclassing UIView

```
#import "CustomView.h"

@implementation CustomView

/* ... */


/* ... */

@end
```

Changing the View Controller's View





Our ~~Awesome~~ Lame App

- The result of our work is the rather plain app to the left
- In order to perform non-trivial drawing tasks we need something that provides us with more capability

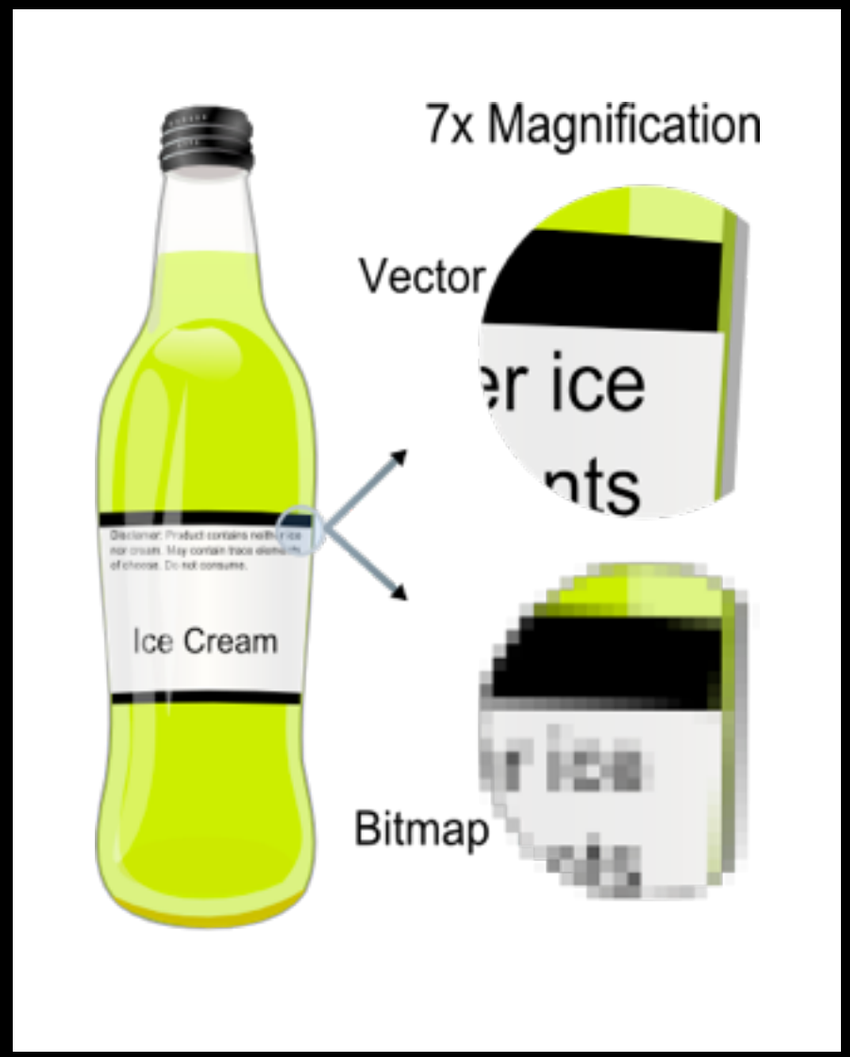
Core Graphics

Core Graphics

- The Core Graphics framework is a C-based API that is based on the Quartz advanced drawing engine
- It provides low-level, lightweight 2D rendering
- You use this framework to handle path-based drawing, transformations, color management, offscreen rendering, patterns, gradients and shadings, image data management, image creation, masking, and PDF document creation, display and parsing

Vector vs. Raster Based

- Core Graphics is a vector-based drawing library
- Vector based drawing deals with defining objects in terms of their mathematical representation
  - Not bound to screen, printer, etc.
  - Scale nicely without artifacts
- Whereas, raster based systems draw their primitives a rectangular grid (such as a pixel array)
  - Scaling up looks blocky



(Image care of Wikipedia)

Core Graphics Related Structures

- CGPoint — a location in 2D space

```
struct CGPoint {
  CGFloat x;
  CGFloat y;
};
typedef struct CGPoint CGPoint;
```

- CGSize — size in 2D space

```
struct CGSize {
  CGFloat width;
  CGFloat height;
};
typedef struct CGSize CGSize;
```

- CGRect — location and dimensions

```
struct CGRect {
  CGPoint origin;
  CGSize size;
};
typedef struct CGRect CGRect;
```

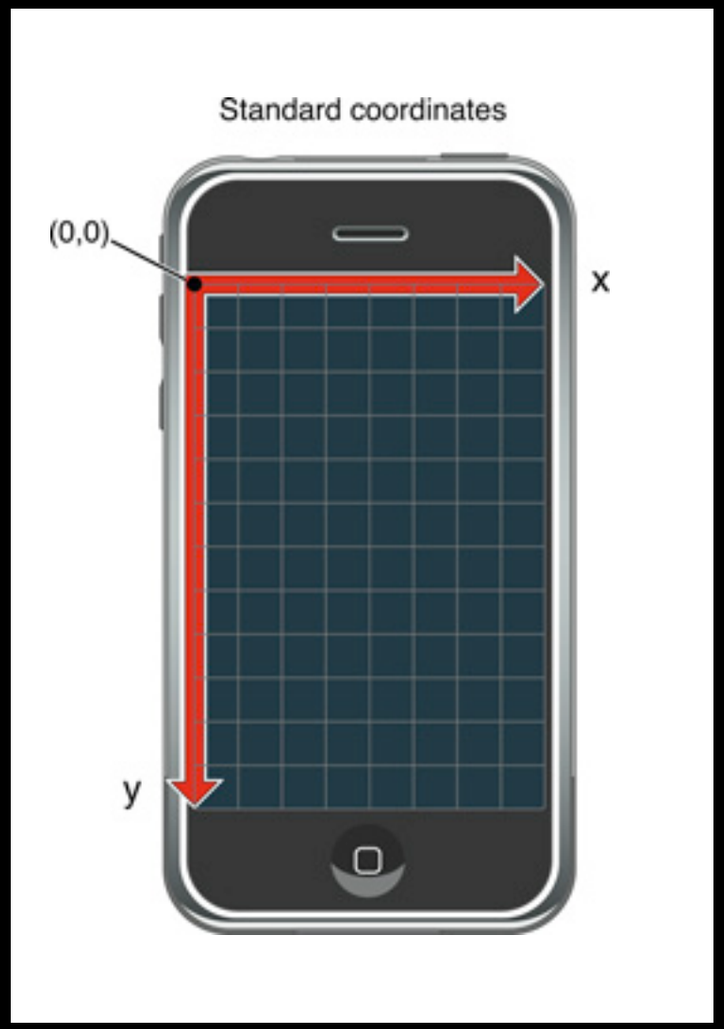
Convenience Functions

- Like many of the other structs that we've had to generate in other Frameworks, Core Graphics provides utility functions to create these objects in a concise manner...

```
CGPoint CGPointMake(CGFloat x, CGFloat y);


CGSize CGSizeMake(CGFloat width, CGFloat height);


CGRect CGRectMake(CGFloat x, CGFloat y, CGFloat width, CGFloat height);
```

UIView Coordinate System

- Origin in upper left corner
- Y axis grows downwards



- Note: this differs from Mac OS X where the origin is in the bottom left

Core Graphics Wrappers

- Some CG functionality wrapped by UIKit...
- UIColor
  - Convenience for common colors
  - Easily set the fill and/or stroke colors when drawing

```
UIColor *redColor = [UIColor redColor];
[redColor set];
```

- UIFont
  - Access system font
  - Get font by name

```
UIFont *font = [UIFont systemFontOfSize:14.0];
[myLabel setFont:font];
```

Drawing a Checkerboard UIView

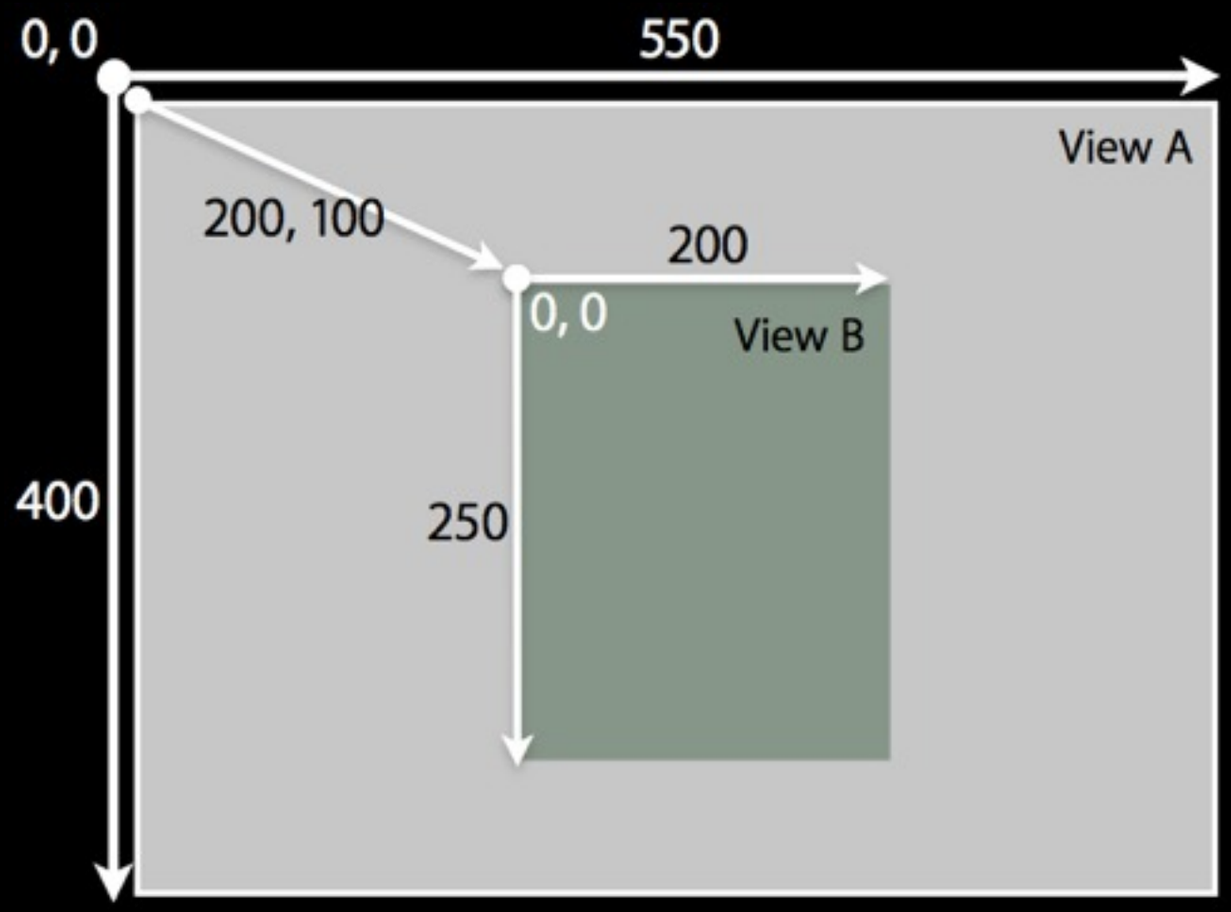
- Override -drawRect:
  - Loop over x and y
  - Draw squares alternating between red and black colors

Drawing a Checkerboard UIView

```
- (void)drawRect:(CGRect)rect {

  NSArray *colors = [NSArray arrayWithObjects:
                     [UIColor redColor],
                     [UIColor blackColor],
                      nil];

  for (int row = 0; row < rect.size.height; row += kHeight) {
    int index = row % (kHeight * 2) == 0 ? 0 : 1;
    for (int col = 0; col < rect.size.width; col += kWidth) {
      [[colors objectAtIndex:index++ % 2] set];
      UIRectFill(CGRectMake(col, row, kWidth, kHeight));
    }
  }

}
```

Frame, Bounds & Center

- View's location and size expressed in two ways
  - Frame is in superview's coordinate system
  - Bounds is in local coordinate system
- Views also have a center property that will adjust the frame accordingly



View A frame:
- origin: (0, 0)
- size: 550 x 400

View A bounds:
- origin: (0, 0)
- size: 550 x 400

View B frame:
- origin: (200, 100)
- size: 200 x 250

View B bounds:
- origin: (0, 0)
- size: 200 x 250

Frame vs. Bounds

- Which to use?
  - Usually depends on the context
- If you are using a view, typically you use frame
- If you are implementing a view, typically you use bounds
- Matter of perspective
  - From outside it's usually the frame
  - From inside it's usually the bounds
- Examples:
  - Creating or positioning a view in superview — use frame
  - Handling events, drawing a view — use bounds

Graphics Context

- A graphics context represents a drawing destination. It contains drawing parameters and all device-specific information that the drawing system needs to perform any subsequent drawing commands
- A graphics context defines basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and several others

CGContextRef

- The graphics context is represented by the CGContextRef class from Core Graphics
- Context setup automatically before invoking -drawRect:
  - Defines current path, line width, transform, etc.
  - Access the graphics context within -drawRect: by calling...

```
CGContextRef UIGraphicsGetCurrentContext(void);
```

  - Use CG calls to change settings
- Context only valid for current call to -drawRect:
  - Do not cache a CGContextRef

Drawing More Complex Shapes

- Common steps for -drawRect: are...
  - Get current graphics context
  - Define a path
  - Set a color
  - Stroke and/or fill path
  - Repeat, as needed

Paths

- A path defines one or more shapes, or subpaths
- A path can consist of straight lines, curves, or both
- It can be open or closed
- A path can be a simple shape, such as a line, circle, rectangle, or star, or a more complex shape such as the silhouette of a mountain range or an abstract doodle

Complex Shapes — Building Blocks

- Core Graphics provides us the following constructs from which to build more complex objects from...
  - Points
  - Lines
  - Arcs
  - Curves
  - Ellipses
  - Rectangles

Points

- Points consist of an x and y coordinate
- The function CGContextMoveToPoint moves to a starting location for building a path
- Quartz keeps track of the current point, which is the last location used for path construction
  - For example, if you call the CGContextMoveToPoint function to set a location at (10, 10), then you draw a horizontal line 50 units long, the last point on the line, that is, (60, 10), becomes the current point
- Lines, arcs, and curves are always drawn starting from the current point

Lines

- A line is defined by its two endpoints
- Its starting point is always assumed to be the current point
  - So, when you create a line, you specify only its endpoint
- You use the function CGContextAddLineToPoint to append a single line to a path
- You can add a series of connected lines to a path by calling the function CGContextAddLines and passing in an array of points
  - Quartz connects each point in the array with the next point in the array, using straight line segments

Arcs

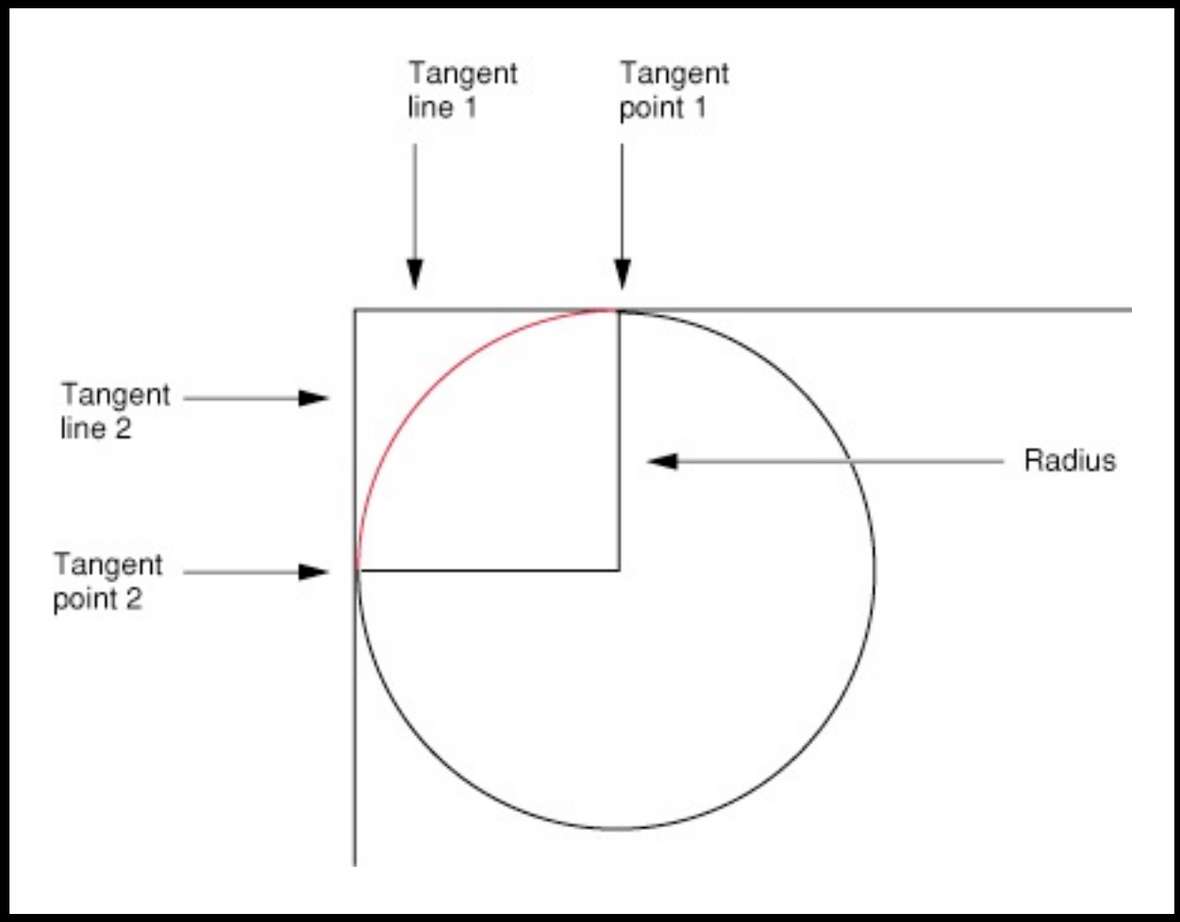
- Arcs are circle segments
- Quartz provides two functions that create arcs

Arcs — CGContextAddArc

- The function CGContextAddArc creates a curved segment from a circle, where you specify...
  - You specify the center of the circle
  - The radius of the circle
  - The radial angle (in radians)
    - Create a full circle by specifying 2 * M_PI

Arcs — CGContextAddArcToPoint

- CGContextAddArcToPoint is frequently used to round the corners of a rectangle
  - The endpoints you supply to create two tangent lines
- You also supply the radius of the circle from which Quartz constructs the arc
- The center point is the intersection of two radii

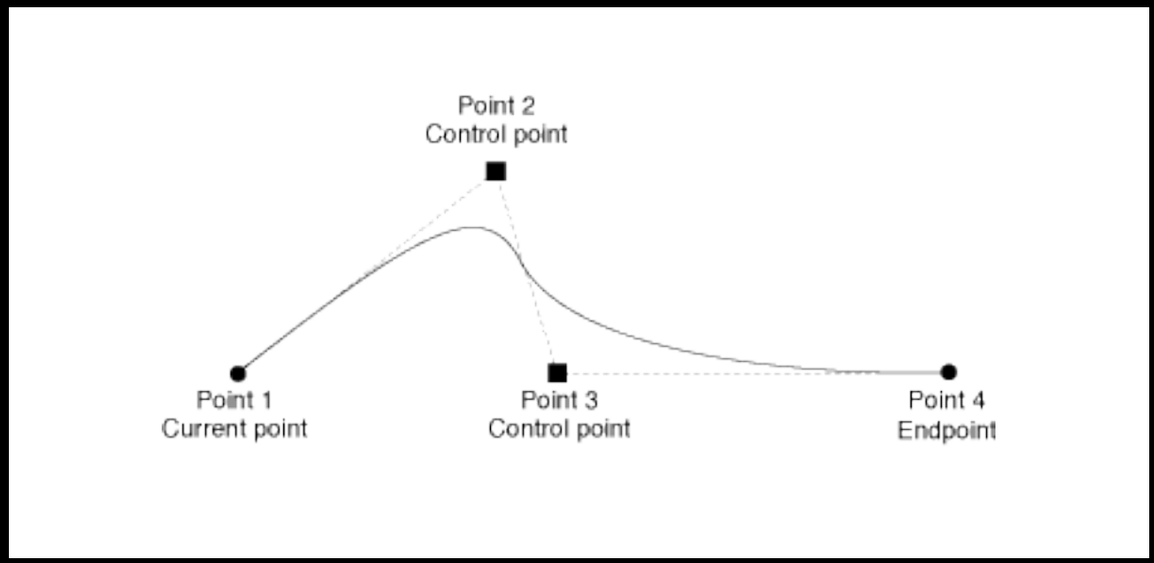


Curves

- Quadratic and cubic Bézier curves are algebraic curves that can specify any number of interesting curvilinear shapes
- Points on these curves are calculated by applying a polynomial formula to starting and ending points, and one or more control points
- Shapes defined in this way are the basis for vector graphics
- A formula is much more compact to store than an array of bits and has the advantage that the curve can be recreated at any resolution

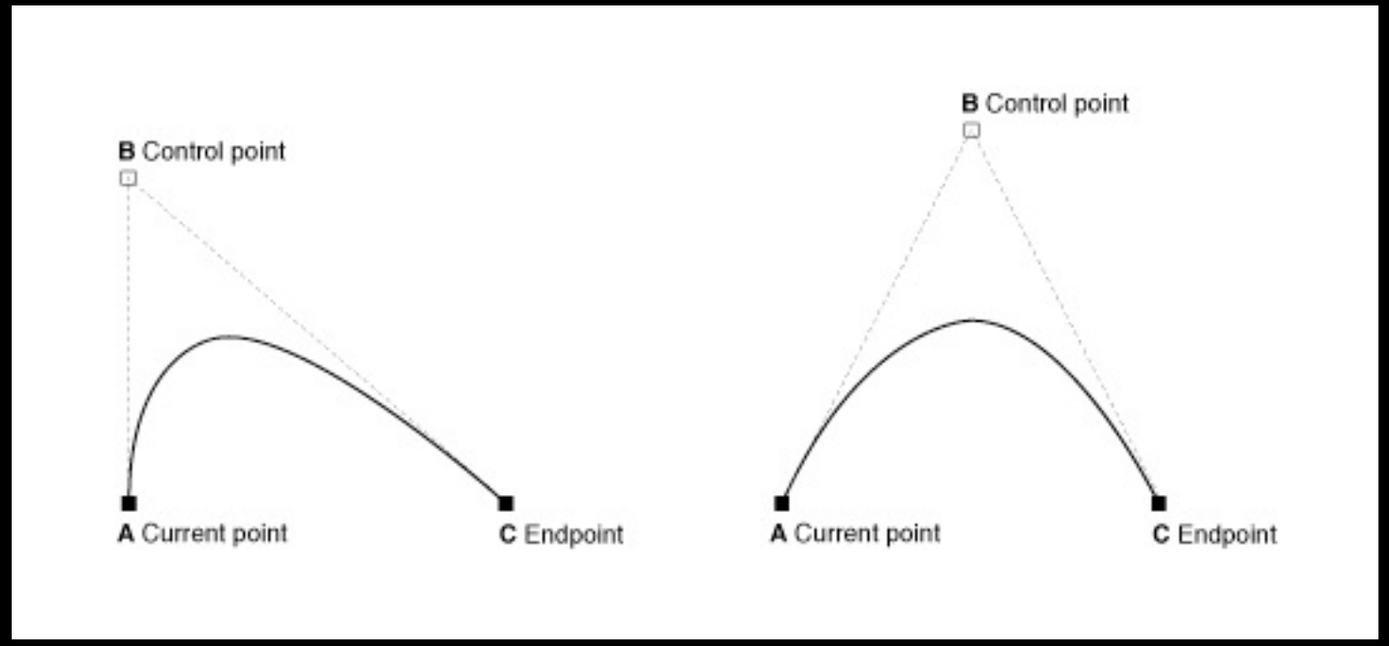
Cubic Bézier Curves

- The function CGContextAddCurveToPoint appends a cubic Bézier curve from the current point, using two control points and an endpoint



Quadratic Bézier Curves

- The function CGContextAddQuadCurveToPoint appends a quadratic Bézier curve from the current point, using a control point and an endpoint



Ellipses

- An ellipse is essentially a squashed circle
- Add an ellipse to the current path by calling the function CGContextAddEllipseInRect.
  - You specify a rectangle that defines the bounds of the ellipse
- Quartz approximates the ellipse using a sequence of Bézier curves

Rectangles

- Add a rectangle to the current path by calling the function CGContextAddRect
  - You specify a CGRect structure that contains the origin of the rectangle and its width and height
  - Quartz draws the rectangle at the origin you specify (the current point is not used in the placement)
- The CGContextAddRects function can add multiple rectangles at once
  - Takes an array of CGRect structs

Fill and Stroke

- By default, these objects do not draw anything to the screen
- They simply define a path — it is up to us to color them in
- We can apply fill and stroke colors to objects we create
- Additionally, we can define how the stroke looks
  - We can set the width of the stroke
  - How strokes are capped and joined
  - If strokes should have some pattern (i.e. dashes, dots, etc.)

Stroke Join Styles

- Lines may be joined by one of the following styles (miter join is the default)...

Miter Join Round Join Bevel Join

```
enum CGLineJoin {
    kCGLineJoinMiter,
    kCGLineJoinRound,
    kCGLineJoinBevel
};
typedef enum CGLineJoin CGLineJoin;
```

Stroke Cap Styles

- Lines may be capped with one of the following styles (butt cap is the default)...

Butt Cap

Round Cap

Projecting Square Cap

```
enum CGLineCap {
    kCGLineCapButt,
    kCGLineCapRound,
    kCGLineCapSquare
};
typedef enum CGLineCap CGLineCap;
```

An Example

An Example

- For this example, I'm creating a custom UIView class and am going to override the -drawRect: method
- Since I'm drawing multiple shapes in this scene, I'm going to split out the different drawing operations into separate methods to keep things organized
- Will eventually call all of these individual methods from the overridden -drawRect method

Drawing an Ellipse

```
- (void)drawEllipse {

  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGRect rectangle = CGRectMake(10, 100, 300, 280);
  CGContextAddEllipseInRect(ctx, rectangle);
  CGContextSetFillColorWithColor(ctx, [UIColor orangeColor].CGColor);
  CGContextFillPath(ctx);

}
```

Drawing a Triangle

```
- (void)drawTriangle {

  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGContextBeginPath(ctx);
  CGContextMoveToPoint(ctx, 160, 220);
  CGContextAddLineToPoint(ctx, 190, 260);
  CGContextAddLineToPoint(ctx, 130, 260);
  CGContextClosePath(ctx);
  CGContextSetFillColorWithColor(ctx, [UIColor blackColor].CGColor);
  CGContextFillPath(ctx);

}
```

Drawing a Rectangle

```
- (void)drawRectangle {

  CGRect rectangle = CGRectMake(100, 290, 120, 25);
  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGContextAddRect(ctx, rectangle);
  CGContextSetFillColorWithColor(ctx, [UIColor blackColor].CGColor);
  CGContextFillPath(ctx);

}
```

Drawing a Curve

```
- (void)drawCurve {

  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGContextBeginPath(ctx);
  CGContextMoveToPoint(ctx, 160, 100);
  CGContextAddQuadCurveToPoint(ctx, 160, 50, 190, 50);
  CGContextSetLineWidth(ctx, 20);
  CGContextSetStrokeColorWithColor(ctx, [UIColor brownColor].CGColor);
  CGContextStrokePath(ctx);

}
```

Drawing a Circle

```
- (void)drawCircleAtX:(float)x Y:(float)y {

  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGContextSetFillColorWithColor(ctx, [UIColor blackColor].CGColor);
  CGContextAddArc(ctx, x, y, 20, 0, 2 * M_PI, 1);
  CGContextFillPath(ctx);

}
```

The Overridden -drawRect Implementation

```
- (void)drawRect:(CGRect)rect {

  [self drawEllipse];
  [self drawTriangle];
  [self drawRectangle];
  [self drawCurve];
  [self drawCircleAtX:120 Y:170];
  [self drawCircleAtX:200 Y:170];

}
```

Any Guesses?

The Resulting App

- Hint: it's a Jack-o'-lantern

Storing Paths

Storing Paths

- In our example we had rather trivial paths that we were creating between calls to...

```
void CGContextBeginPath(CGContextRef c);
void CGContextClosePath(CGContextRef c);
```

- However, perhaps we're programmatically creating the path and it is expensive to do so
  - We don't want to have to do that over and over again
- Core Graphics allows us to create and store paths for later re-use thus saving us the overhead of re-creating them

CGPath

- Two parallel sets of functions for using paths
  - CGContext "convenience" throwaway functions
  - CGPath functions for creating reusable paths

CGContext	CGPath
CGContextMoveToPoint	CGPathMoveToPoint
CGContextLineToPoint	CGPathAddLineToPoint
CGContextAddArcToPoint	CGPathAddArcToPoint
CGContextClosePath	CGPathCloseSubPath
...	...

Example Path Creating

- Below is an example implementation of the -drawTriangle method like we just saw, but instead of directly using the context the path is constructed and stored in an ivar
- Using lazy creation to construct the path
- I've provided a getter method for the trianglePath ivar thus avoiding the compiler default
- When accessed for the first time we check to see if it the ivar has been set, if not we do so then
- After the ivar has been initialized (if needed) it is returned

Example Lazy Creation of a Path

```
- (CGMutablePathRef)trianglePath {
  if (!trianglePath) {
    trianglePath = CGPathCreateMutable();
    CGPathMoveToPoint(trianglePath, NULL, 160, 220);
    CGPathAddLineToPoint(trianglePath, NULL, 190, 260);
    CGPathAddLineToPoint(trianglePath, NULL, 130, 260);
    CGPathCloseSubpath(trianglePath);
  }
  return trianglePath;
}


- (void)drawTriangle {
  CGContextRef ctx = UIGraphicsGetCurrentContext();
  CGContextSetFillColorWithColor(ctx, [UIColor blackColor].CGColor);
  CGContextAddPath(ctx, self.trianglePath);
  CGContextFillPath(ctx);
}
```

Images and Text

Images & Text

- If we want to draw images or text from the -drawRect: of a subclassed UIView we can easily do so

```
- (void)drawRect:(CGRect)rect {
  UIImage *img = [UIImage imageNamed:@"happy"];
  [img drawAtPoint:CGPointMake(110, 20)];

  NSString *str = @"iPhone is Happy!";
  UIFont *font = [UIFont fontWithName:@"Marker Felt"
                                 size:36];
  [str drawAtPoint:CGPointMake(40, 240)
          withFont:font];
}
```

More Information

- Core Graphics is a large area, and consists of more than we can explore in a single class
- The "Quartz 2D Programming Guide" provides a lot of additional information — including documentation on:
  - Gradients
  - Color spaces
  - Transforms
  - Patterns
  - Shadows
  - Etc...

Core Animation

What is Core Animation?

- Core Animation is a collection of Objective-C classes for graphics rendering, projection, and animation
- It provides fluid animations using advanced compositing effects while retaining a hierarchical layer abstraction that is familiar to developers using the Application Kit and Cocoa Touch view architectures

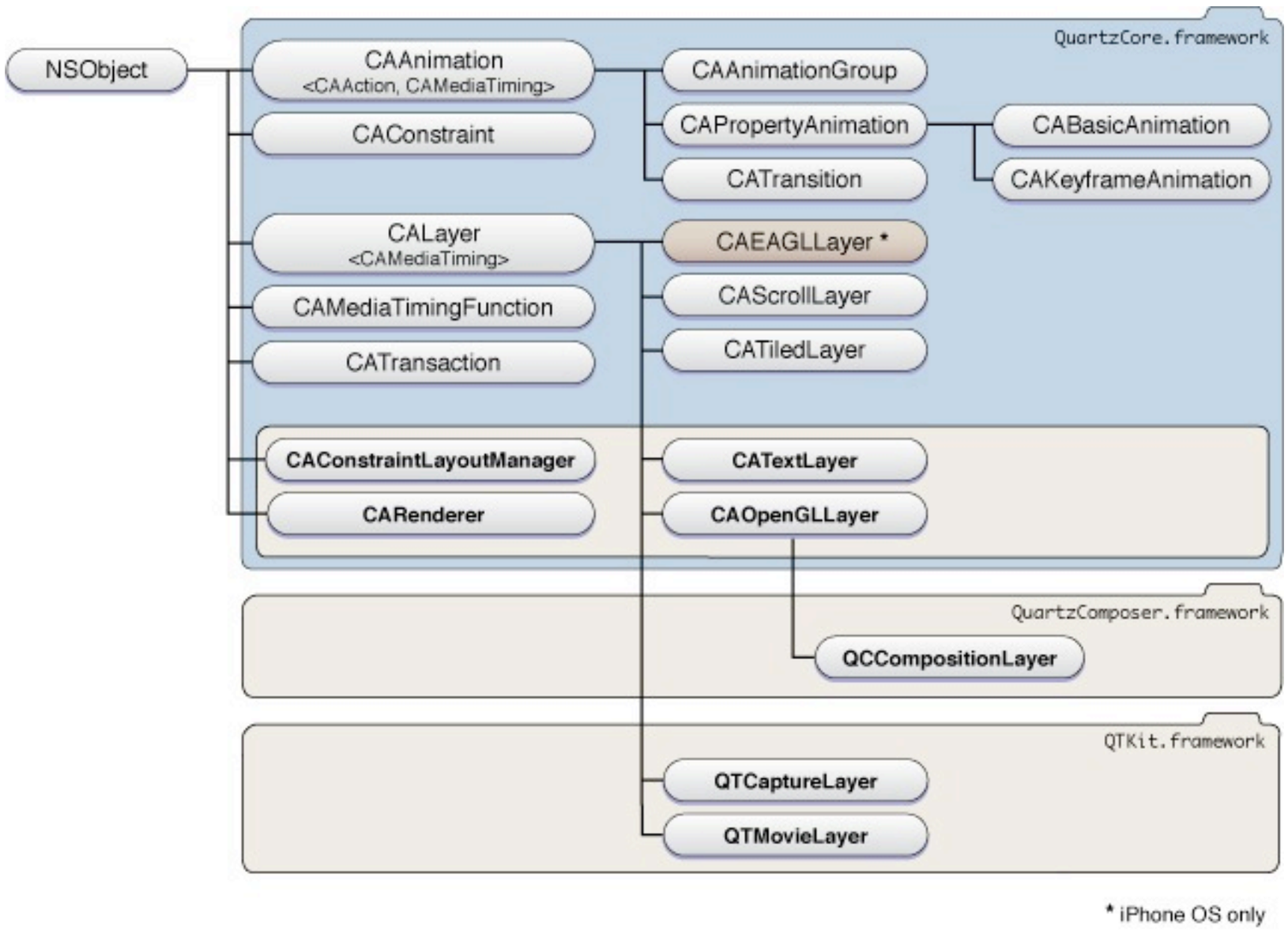
Core Animation Features

- High performance compositing with a simple approachable programming model
- A familiar view-like abstraction that allows you to create complex user interfaces using a hierarchy of layer objects
- A lightweight data structure — you can display and animate hundreds of layers simultaneously
- Manages animations and runs them at frame-rate in separated thread
- Improved application performance — applications need only redraw content when it changes
- Flexible layout manager model

Why Core Animation?

- Using Core Animation, developers can create dynamic user interfaces for their applications without having to use low-level graphics APIs such as OpenGL to get respectable animation performance

Core Animation Class Hierarchy



Layers

- Layers form the foundation of Core Animation and are similar to UIViews
  - Layers live in a hierarchy — each layer has a single parent (superlayer) and a collection of sublayers
  - You can specify geometry relative to superlayer, thus creating a local coordinate system
  - Supports transform matrices that allow you to rotate, skew, scale, and project the layer content
- UIView has a layer property which represents the view's layer used for rendering
- Layers are backed by the CALayer class

CALayer

- The CALayer class introduces the concept of a key-value coding compliant container class
  - Store arbitrary values, using key-value coding compliant methods, without having to create a subclass
- CALayer also manages the animations and actions that are associated with a layer
  - Layers receive action triggers in response to layers being inserted and removed from the layer tree, modifications being made to layer properties, or explicit developer requests

Animation and Timing

- Many visual properties of a layer are implicitly animatable...
  - Simply changing the value of an animatable property the layer will automatically animate between values (if it happens in an animation block)
  - For example, setting the backgroundColor property triggers an animation that causes the layer to gradually fade from its current color to the new color
  - Most animatable properties have an associated default animation which you can easily customize and replace

Traditional Animation “Blocks”

- UIView provides several methods for animating views — the more traditional method requires a begin/commit “block”...
- +beginAnimations:context: starts an animation block

```
+ (void)beginAnimations:(NSString *)animationID context:(void *)context;
```

- Changes can be made on view properties within the block
- When done we can +commitAnimations to start the default animation behavior for our changes

```
+ (void)commitAnimations;
```

Using ObjC 2.0 Blocks

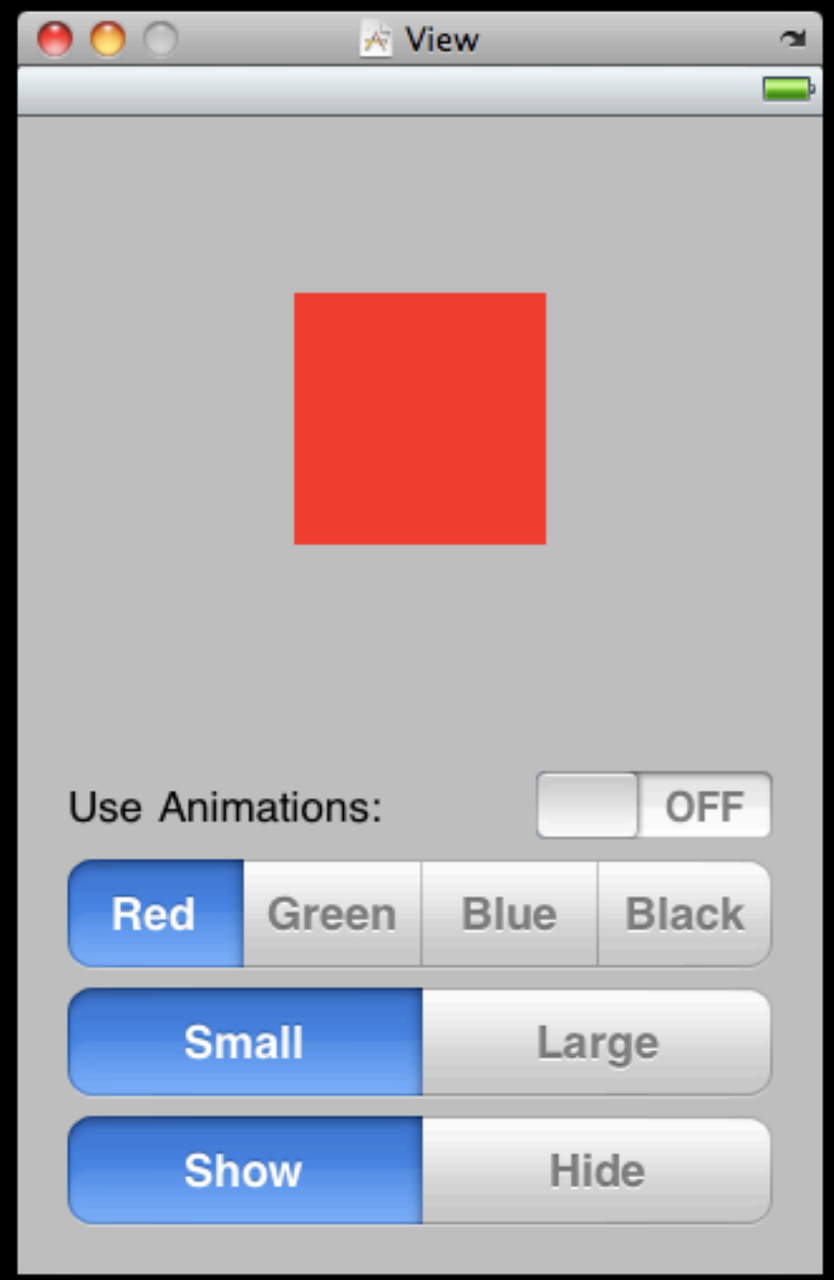
- Starting with iOS 4.0, support has been added to UIView to specify an ObjC 2.0 block instead of specifying animations between begin/commit statements

```
// animate with given duration all changes in animation block
+ (void)animateWithDuration:(NSTimeInterval)duration
                 animations:(void (^)(void))animations;

// same as previous + specify block to exec upon completion
+ (void)animateWithDuration:(NSTimeInterval)duration
                 animations:(void (^)(void))animations
                 completion:(void (^)(BOOL finished))completion;

// same as previous + specify animating options
+ (void)animateWithDuration:(NSTimeInterval)duration
                      delay:(NSTimeInterval)delay
                    options:(UIViewAnimationOptions)options
                 animations:(void (^)(void))animations
                 completion:(void (^)(BOOL finished))completion;
```

An Example

An Example

- For our example, we're going to create a simple UI with a subview (shown in red) that we'll manipulate
- Switch to toggle animations on/off
- 3 segmented controls each tied back to a corresponding action method
  - One to switch colors
  - One to change the size of the view
  - One to show/hide the view



AnimationViewController.h

```
#import <UIKit/UIKit.h>

@interface AnimationViewController : UIViewController {

}

@property (nonatomic, retain) IBOutlet UIView *square;
@property (nonatomic, retain) IBOutlet UISwitch *animation;
@property (nonatomic, retain) IBOutlet UISegmentedControl *colors;
@property (nonatomic, retain) IBOutlet UISegmentedControl *sizes;
@property (nonatomic, retain) IBOutlet UISegmentedControl *visibilities;

- (IBAction)updateColor;
- (IBAction)updateSize;
- (IBAction)updateVisibility;

@end
```

AnimationViewController.m

```
#import "AnimationViewController.h"

#define kAnimationDuration 3

@implementation AnimationViewController

@synthesize square, animation, colors, sizes, visibilities;

- (IBAction)updateColor {

  if (self.animation.isOn) {
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:kAnimationDuration];
  }

  NSArray *choices = [NSArray arrayWithObjects:[UIColor redColor], [UIColor greenColor],
                      [UIColor blueColor], [UIColor blackColor], nil];
  self.square.backgroundColor = [choices objectAtIndex:
                                 [self.colors selectedSegmentIndex]];

  if (self.animation.isOn) {
    [UIView commitAnimations];
  }
}

/* ... */
```

AnimationViewController.m

```
/* ... */

- (IBAction)updateSize {

  if (self.animation.isOn) {
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:kAnimationDuration];
  }

  self.square.bounds = [self.sizes selectedSegmentIndex] == 0
                       ? CGRectMake(0, 0, 100, 100) : CGRectMake(0, 0, 200, 200);

  if (self.animation.isOn) {
    [UIView commitAnimations];
  }

}

/* ... */
```

AnimationViewController.m

```
/* ... */

- (IBAction)updateVisibility {

  if (self.animation.isOn) {
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:kAnimationDuration];
  }

  self.square.alpha = [self.visibilities selectedSegmentIndex] == 0 ? 1.0 : 0.0;

  if (self.animation.isOn) {
    [UIView commitAnimations];
  }
}

/* ... */

@end
```

The App Thus Far

- Here's the resulting app, mid-transition between...
  - Red and Blue

Animating Multiple Properties

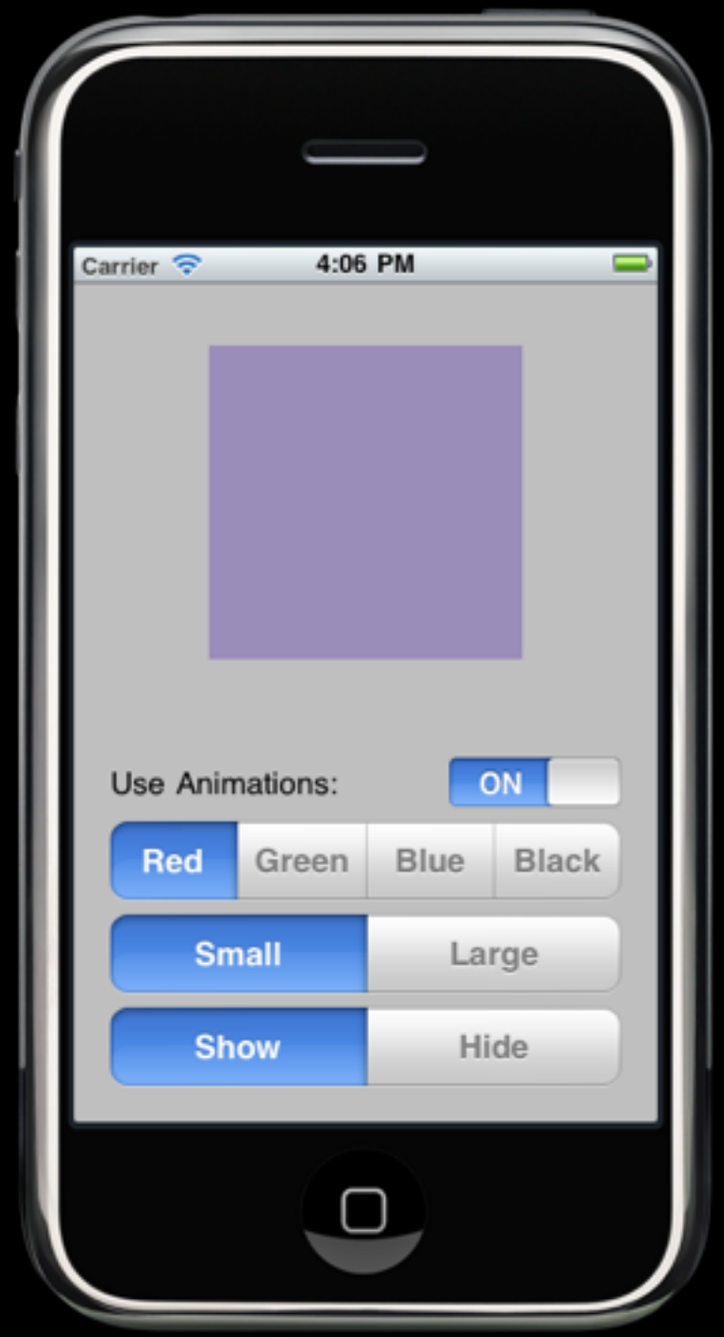
- Let's add a shake-to-reset feature to our app
- Last class we looked at subclassing UIView to respond to shake gestures
- Let's move this recognition into the view controller
- Remember, if a view is back by a controller, it has the ability to become the responder
- Need to state that the view controller is capable of becoming the first responder
- And lastly, have the view controller actually become the first responder

AnimationViewController.m

```
/* ... */

-(BOOL)canBecomeFirstResponder {
  return YES;
}

-(void)viewDidAppear:(BOOL)animated {
  [super viewDidAppear:animated];
  [self becomeFirstResponder];
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event {
  if (motion == UIEventSubtypeMotionShake) {
    [UIView animateWithDuration:kAnimationDuration
                     animations:^ {
                       self.square.bounds = CGRectMake(0, 0, 100, 100);
                       self.square.backgroundColor = [UIColor redColor];
                       self.square.alpha = 1.0;
                     }];
    self.colors.selectedSegmentIndex = 0;
    self.sizes.selectedSegmentIndex = 0;
    self.visibilities.selectedSegmentIndex = 0;
  }
}

/* ... */
```

The Final App

- The app midway through all 3 simultaneous transitions



Easing Functions

Animation and Timing

- Animatable properties can also be explicitly animated
- Create an instances of one of CA's animation classes and specify the required visual effects
- CA provides animation classes that can animate the entire contents of a layer or selected attributes using both basic animation and key-frame animation
- The animation classes also define a timing function that describes the pacing of the animation as a simple Bezier curve

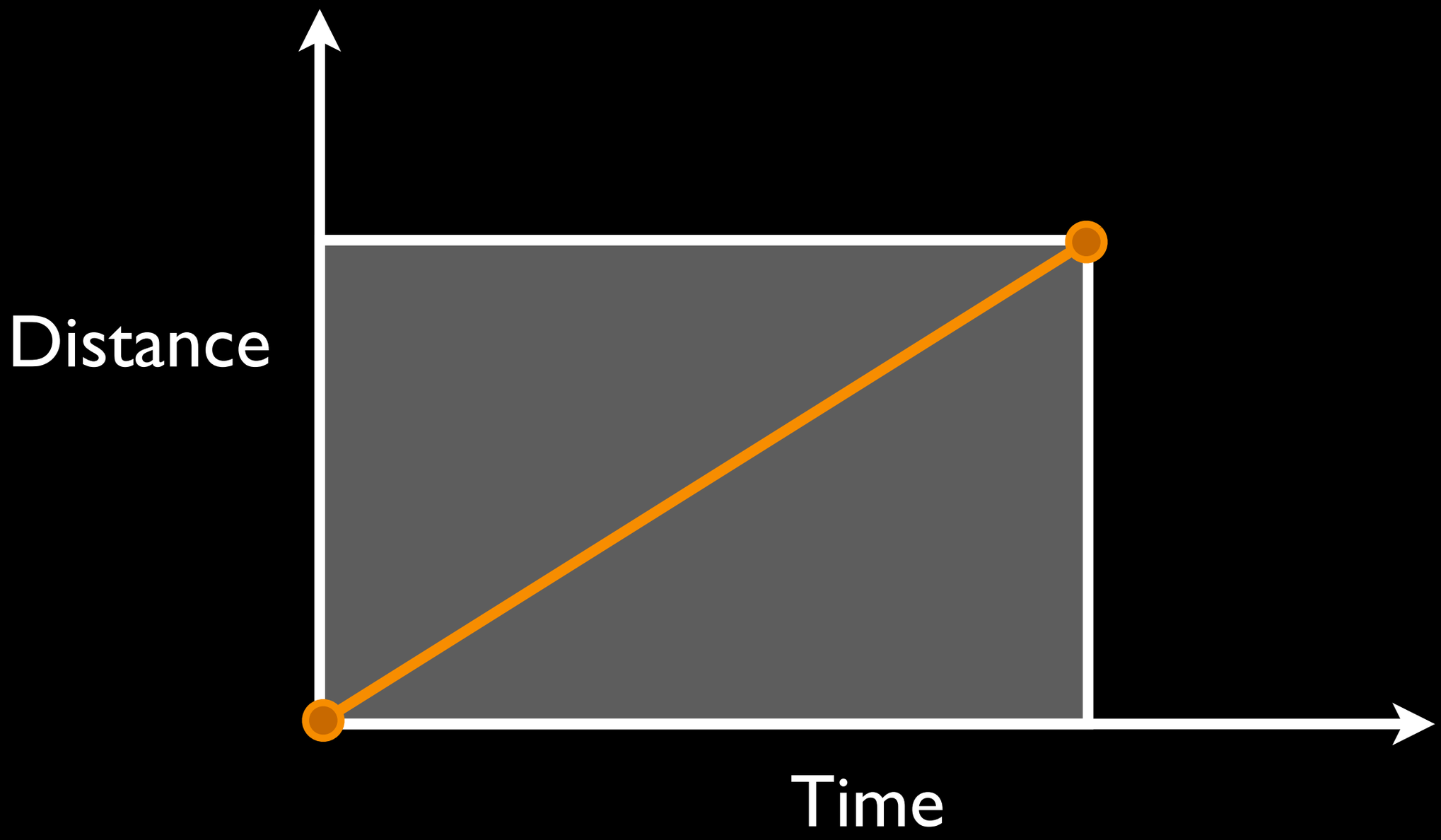
Animation Timing

- The following animation curves (or easing functions) are built into Core Animation...

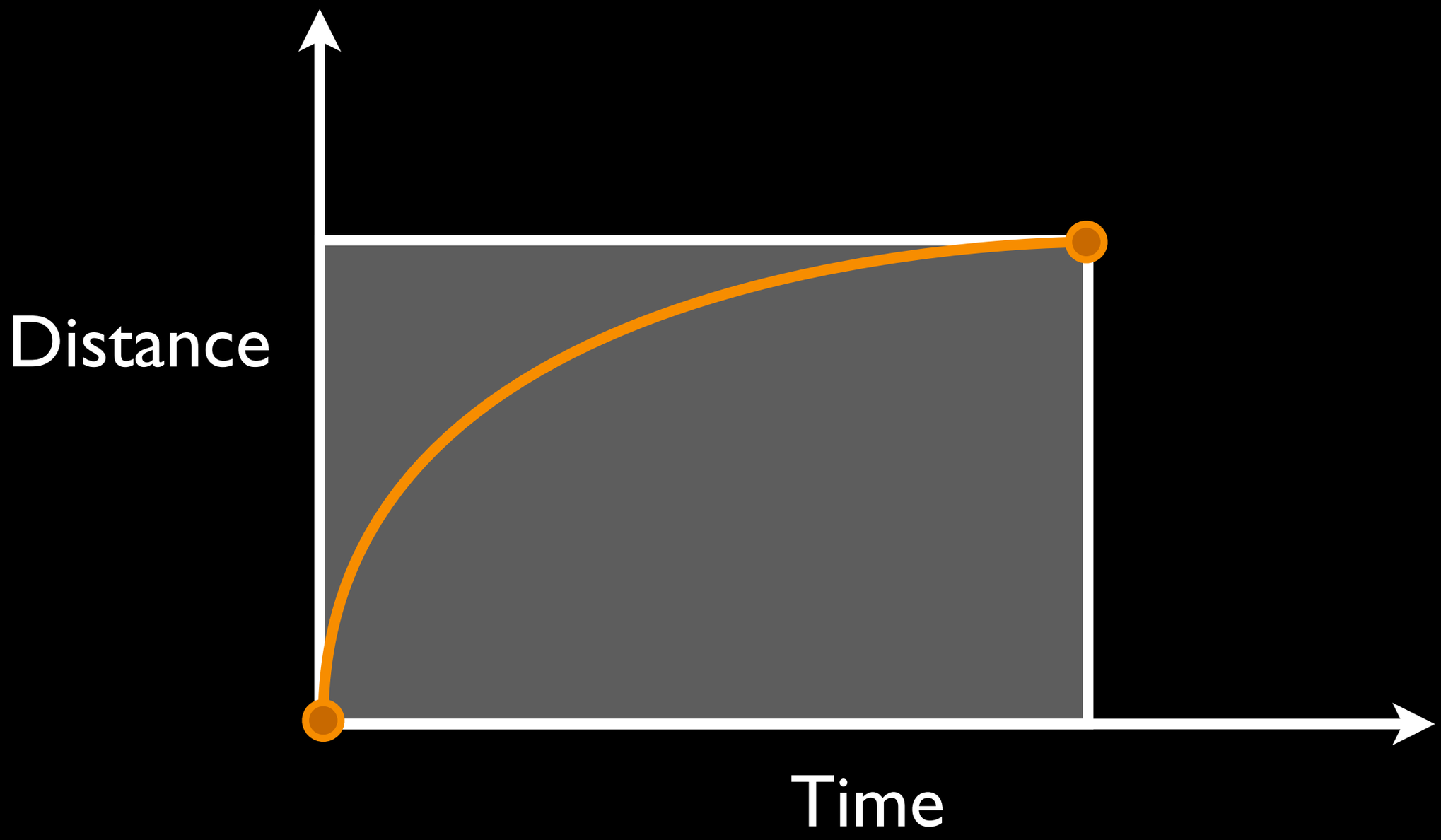
```
typedef enum {
    UIViewAnimationCurveEaseInOut,          // slow at beginning and end
    UIViewAnimationCurveEaseIn,             // slow at beginning
    UIViewAnimationCurveEaseOut,            // slow at end
    UIViewAnimationCurveLinear
} UIViewAnimationCurve;

enum {
    /* ... */
    UIViewAnimationOptionCurveEaseInOut             = 0 << 16, // default
    UIViewAnimationOptionCurveEaseIn                = 1 << 16,
    UIViewAnimationOptionCurveEaseOut               = 2 << 16,
    UIViewAnimationOptionCurveLinear                = 3 << 16,
    /* ... */
};
typedef NSUInteger UIViewAnimationOptions;
```

Different sets of constants depending upon which animation approach is being used

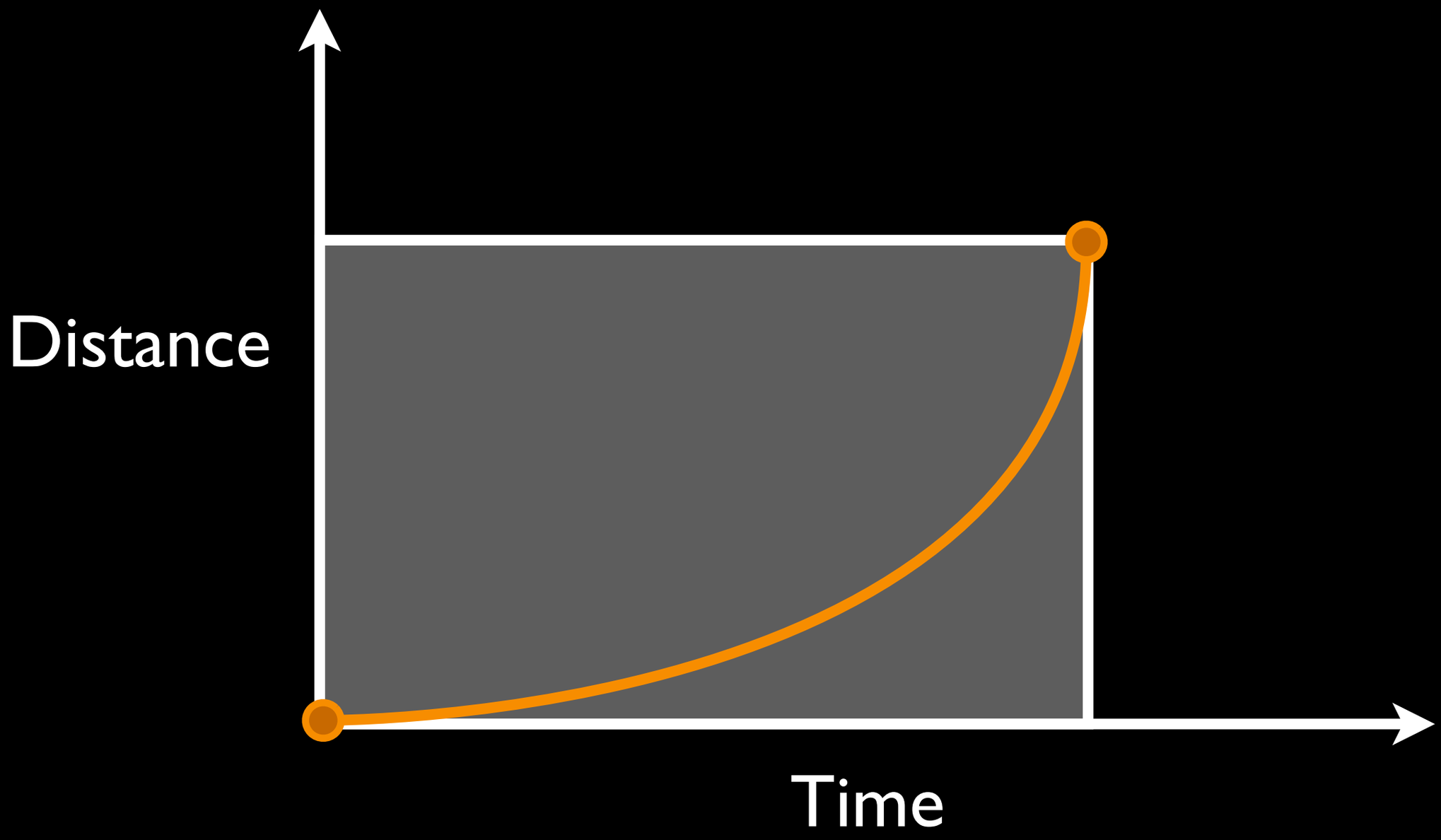
Linear



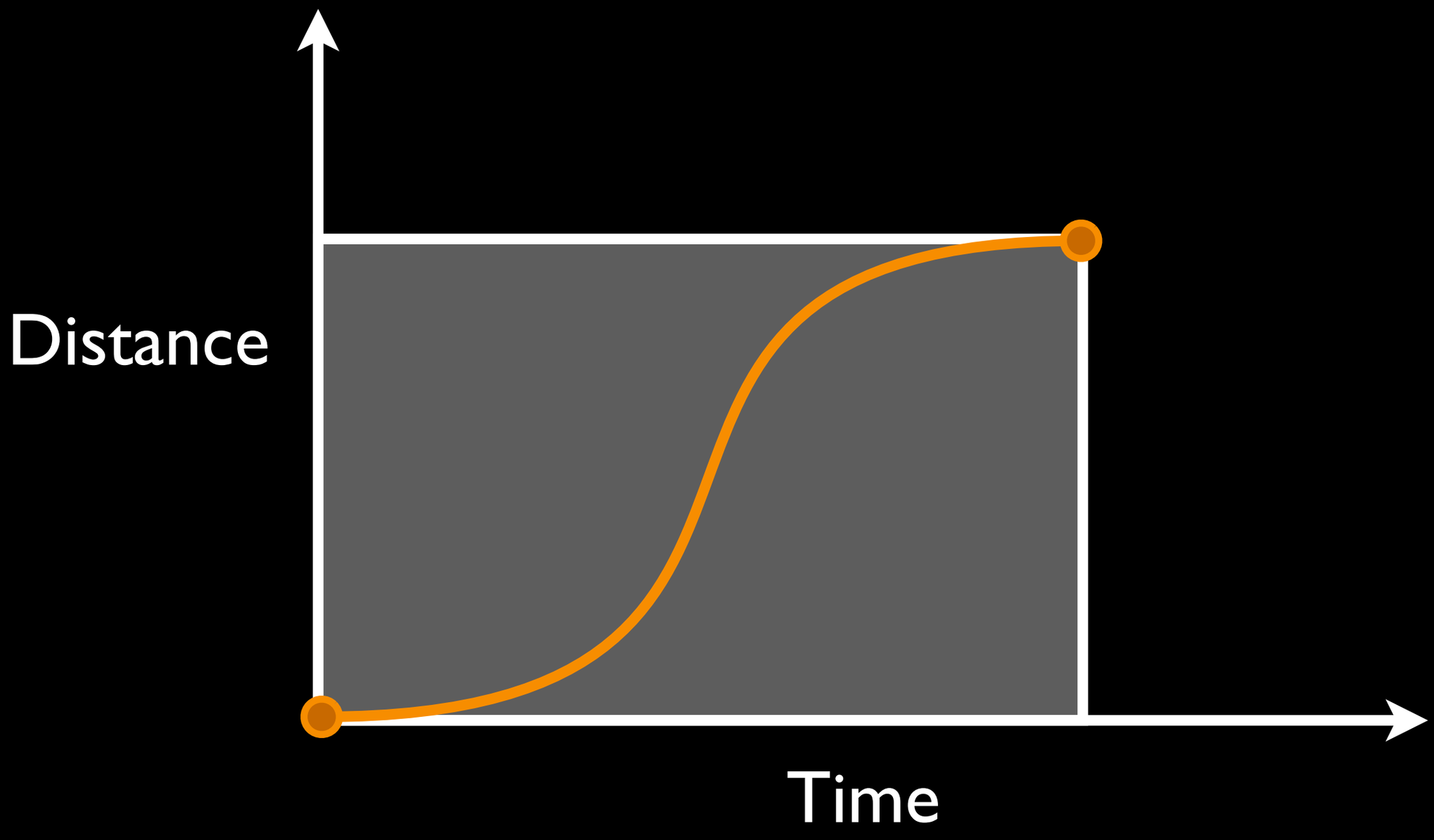
East Out



Ease In



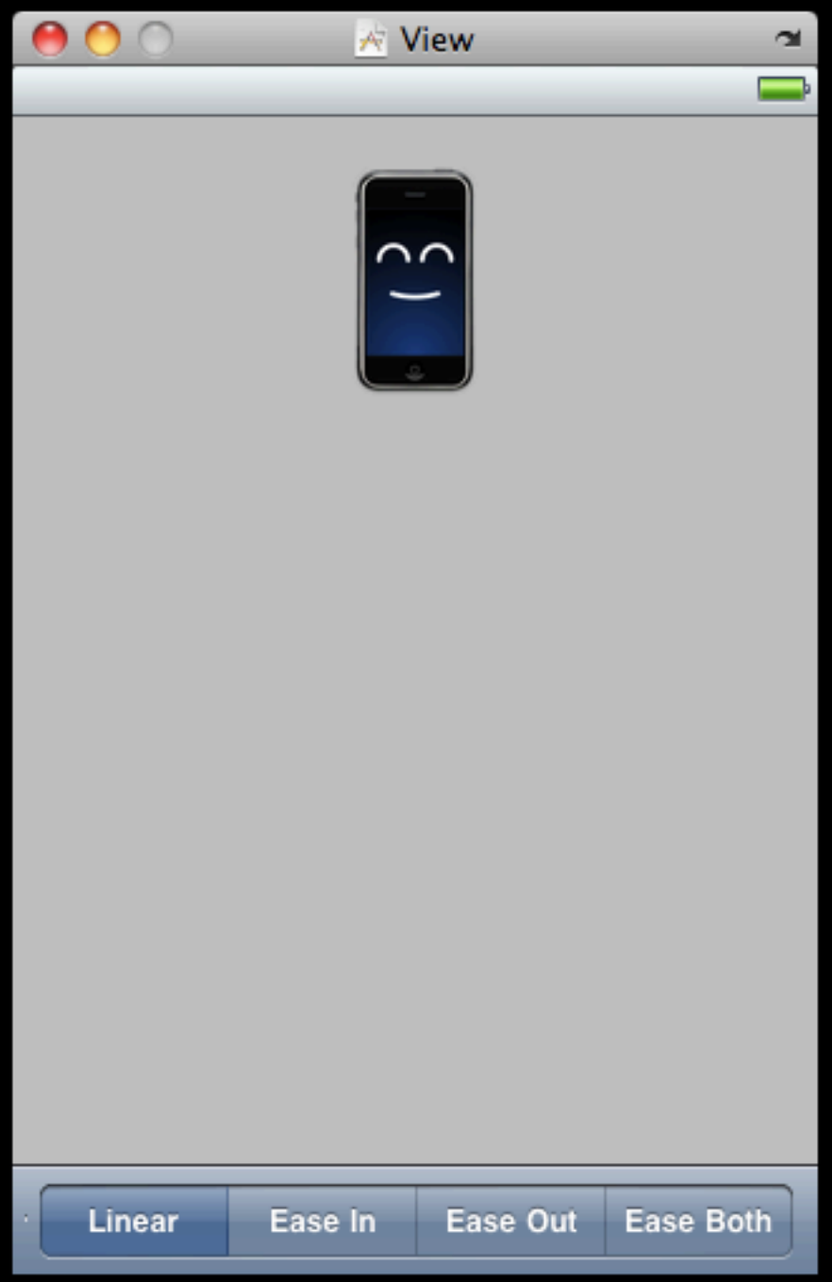
Ease In/Out



Easing Functions Example

An Example

- For our example, we're going to create a simple UI with an UIImageView which is set to be the picture of an iPhone
- One segmented control tied back to an action method which changes the saved animation curve



CoreAnimationViewController.h

```
#import <UIKit/UIKit.h>

@interface TouchMoveViewController : UIViewController {
  UIViewAnimationOptions curve;
}

@property (nonatomic, retain) IBOutlet UIImageView *image;

- (IBAction)changeAnimation:(id)sender;

@end
```

TouchMoveViewController.m

```
#import "TouchMoveViewController.h"

@implementation TouchMoveViewController

@synthesize image;

- (IBAction)changeAnimation:(id)sender {

    UIViewAnimationOptions curves[] = {
        UIViewAnimationOptionCurveLinear,
        UIViewAnimationOptionCurveEaseIn,
        UIViewAnimationOptionCurveEaseOut,
        UIViewAnimationOptionCurveEaseInOut
    };
    curve = curves[[sender selectedSegmentIndex]];

}

/* ... */
```

TouchMoveViewController.m

```
/* ... */

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

  [UIView animateWithDuration:2.0
                        delay:0
                      options:curve
                   animations:^ {
                                  self.image.center = [[touches anyObject]
                                                       locationInView:self.view];
                              }
                   completion:NULL];
}

/* ... */

@end
```

The Resulting App

- The resulting app — the iPhone image follows us around where we tap

Animation Delegates

Animation Delegates

- We also have the opportunity to hook-into different parts of the animation life-cycle using delegates
- To set the delegate we use the following methods of UIView...

```
+ (void)setAnimationDelegate:(id)delegate;
```

- To hook into the start and stop points of the animation...

```
+ (void)setAnimationWillStartSelector:(SEL)selector;
+ (void)setAnimationDidStopSelector:(SEL)selector;
```

3D Transformations

3D Transformations

- You can think of layers as being these 2D objects that live in 3D space
- As such, we can animate the layers around in 3D space
- For this example, we're going to be creating a custom CAKeyframeAnimation to store the animation states
- We have to add the Quartz Core framework to the project and include the appropriate header to use the CAKeyframeAnimation class

FlipViewController.h

```
#import <UIKit/UIKit.h>

@interface FlipViewController : UIViewController {

}

@property (nonatomic, retain) IBOutlet UIImageView *image;

@end
```

FlipViewController.m

```
#import "FlipViewController.h"
#import <QuartzCore/QuartzCore.h>

@implementation FlipViewController

@synthesize image;

/* ... */

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
  CAKeyframeAnimation *animation = [CAKeyframeAnimation
                                    animationWithKeyPath:@"transform"];
  NSValue *initial = [NSValue valueWithCATransform3D:
                      CATransform3DMakeRotation(0.0, 1.0f, -1.0f, 0.0f)];
  NSValue *flip1 = [NSValue valueWithCATransform3D:
                    CATransform3DMakeRotation(M_PI, 1.0f, -1.0f, 0.0f)];
  NSValue *flip2 = [NSValue valueWithCATransform3D:
                    CATransform3DMakeRotation(M_PI, 1.0f, 1.0f, 0.0f)];
  animation.values = [NSArray arrayWithObjects:initial, flip1, initial,
                                               flip2, initial, nil];
  animation.duration = 2.0f;
  [(id)self.image addAnimation:animation forKey:@"transform"];
}

/* ... */

@end
```

The Resulting App

Additional Resources

- Quartz 2D Programming Guide
  - http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/
- Core Animation Programming Guide
  - http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreAnimation_guide/
- Core Animation Cookbook
  - http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/CoreAnimation_Cookbook/

For Next Class

- OpenGL ES Programming Guide for iOS
  - http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/
- Khronos Group OpenGL ES pages
  - http://www.khronos.org/opengles/
- OpenGL ES 1.1 online man pages
  - http://www.khronos.org/opengles/sdk/1.1/docs/man/
- OpenGL ES 2.0 online man pages
  - http://www.khronos.org/opengles/sdk/docs/man/

For Next Class

- Jeff LaMarche's "OpenGL ES from the Ground Up" Series
  - http://iphonedevelopment.blogspot.com/2009/05/opengl-es-from-ground-up-table-of.html
- Simon Maurice's iPhone OpenGL ES Tutorial Series
  - http://web.me.com/smaurice/AppleCoder/iPhone_OpenGL/iPhone_OpenGL.html
- NeHe Production's OpenGL Lessons
  - http://nehe.gamedev.net/lesson.asp?index=01