iOS Dev Accelerator Week 7 Day 2

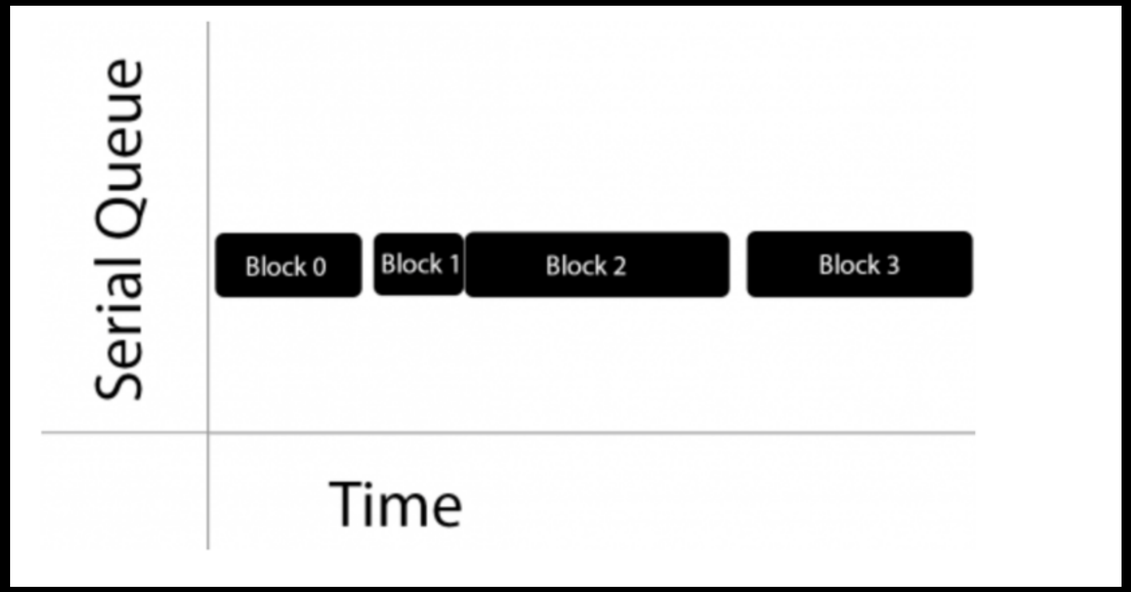
- Grand Central Dispatch
- Scroll Views
- More Objective-C things

Grand Central Dispatch

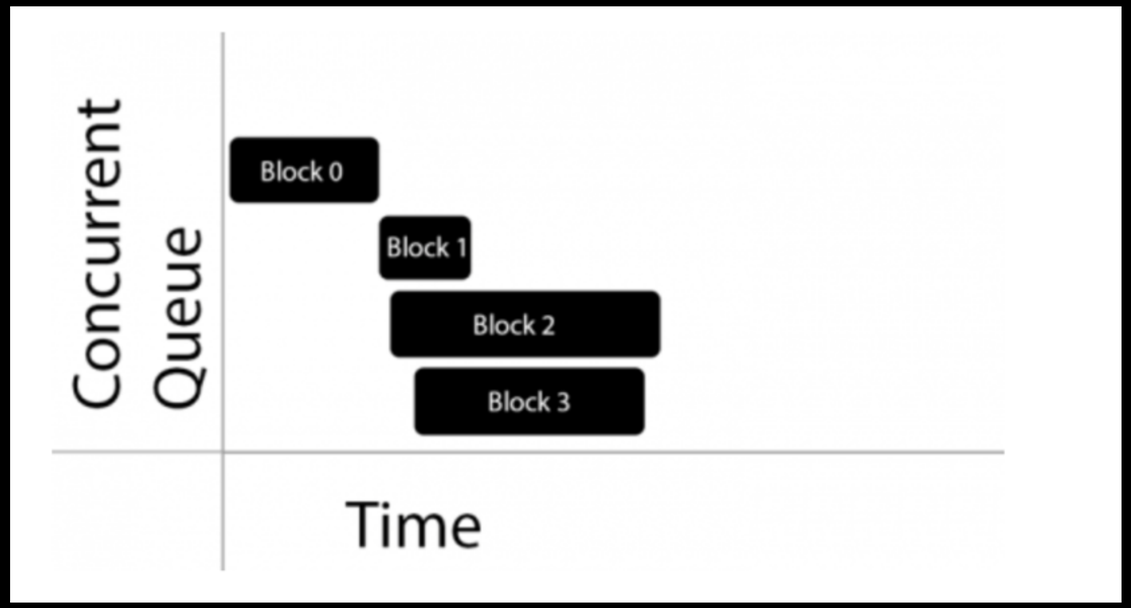
Grand Central Dispatch

- “Grand Central Dispatch (GCD) comprises language features, runtime libraries, and system enhancements that provide systemic, comprehensive improvements to the support of concurrent code execution on multicore hardware on iOS and OS X"
- GCD is a system you can use for concurrency
- GCD sits above manually creating threads and locks, and below NSOperationQueue
- GCD feels a lot like NSOperationQueue, so it’s not a drastic transition going from NSOperationQueue to GCD
- GCD is a C-style API, so you don’t call the methods on classes, they are just global functions. The functions will look_like_this

- Tasks executed serially are executed on at a time



- Tasks executed concurrently might be executed at the same time



Async vs synchronous

- A synchronous function returns only after the completion of tasks that it orders
- An asynchronous function returns immediately, ordering the task to be done but does not wait for it (does not block for it)

Concurrency and structure

- Even when using a higher level API like NSOperationQueue, you need to think about and structure your app with concurrency in mind. Specifically you need to design with these things in mind:
  - Race Conditions: The behavior of your app depends on the timing of certain events that are being executed in an uncontrolled concurrent manner
  - Dead Locks: Two or more threads are deadlocked if one thread is waiting for another thread to finish, while the other thread is waiting for the first thread to finish
  - Thread safety: Thread safe code is safe to call from multiple threads or concurrent tasks. Code that is not thread safe must be only accessed in one context at a time. An immutable array is thread safe because it cannot be changed, while a mutable array is not thread safe since one thread could be changing it while another is trying to read it.
  - Critical Sections: Any piece of code that must not be executed from more than one thread or task. For example manipulating a shared resource that isn't thread safe.

GCD and Queuing Tasks for Dispatch

- GCD provides and manage FIFO queues to which your app can submit tasks in form of blocks (or closures in swift)
- Blocks submitted on dispatch queues are executed on a pool of threads that are managed by the system
- 3 types of queues:
  - Main: tasks execute serially on your app's main thread (great for UI)
  - Concurrent: tasks are dequeued in FIFO order, but run concurrently
  - Serial: tasks execute one at a time in FIFO order

Queuing a task for the main queue

```
NSArray *results = [Question questionsFromJSON:data];
dispatch_async(dispatch_get_main_queue(), ^{
  completionHandler(results,nil);
});
```

- dispatch_async is a function that takes in two parameters:

  1. The queue you want to execute a task on

  2. a block that represents the task you want to execute

Using a concurrent global queue

- Global Queues are created by the system for your app and available to use without any setup
- To access a global queue you use the function dispatch_get_global_queue
- dispatch_get_global_queue has 2 parameters:
  - identifier: how you tell GCD which queue you want to enqueue tasks on
  - flags: ‘will be used in the future’ just use 0

global queue identifiers

These are the newest set of QOS identifiers to help you choose which global queue to use:

- QOS_CLASS_USER_INTERACTIVE: highest priority since the user experience relies on it
- QOS_CLASS_USER_INITIATED: should be used when the user is waiting on results to continue interaction
- QOS_CLASS_UTILITY: long running tasks, designed to be energy efficient
- QOS_CLASS_BACKGROUND: anything the user isn't directly aware of.

global queue identifiers

These are the old, and less confusing ones

- DISPATCH_QUEUE_PRIORITY_HIGH
- DISPATCH_QUEUE_PRIORITY_DEFAULT
- DISPATCH_QUEUE_PRIORITY_LOW
- DISPATCH_QUEUE_PRIORITY_BACKGROUND

You can use the old or new ones, just know that both sets exist

Using a concurrent global queue

```
  dispatch_queue_t imageQueue = dispatch_get_global_queue
(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(imageQueue, ^{

  NSURL *url = [NSURL URLWithString:avatarURL];
  NSData *data = [[NSData alloc] initWithContentsOfURL:url];
  UIImage *image = [UIImage imageWithData:data];

  dispatch_async(dispatch_get_main_queue(), ^{
    completionHandler(image,nil);
  });
});
```

Creating a private queue

- You can create a private queue if you want to run tasks on a serial or concurrent queue that isn't a global queue or the main queue
- The only big difference between using a private queue and a global queue is that a private queue can be a serial queue.
- To create a private queue use the function dispatch_queue_create which takes in 2 parameters:
  - label - string to attach to the queue to uniquely identity what process is launching these queues. Use reverse DNS, or pass in NULL
  - attr - specify DISPATCH_QUEUE_SERIAL or DISPATCH_QUEUE_CONCURRENT or NULL (which sets it to serial)

Queuing up your tasks

- The two primary ways of enqueuing your tasks for execution are:
  - dispatch_sync
  - dispatch_async

there is also dispatch_once which is great for singletons

dispatch_async

- Dispatches a task to to run asynchronously and returns immediately
- This is by bar the most common way to enqueue a task on a dispatch queue
- Even if adding tasks to a serial queue, still use dispatch_async

dispatch_sync

- Submits a block to execute on a dispatch queue and waits until that block completes before returning
- Rarely will use this

dispatch_async vs sync

```
dispatch_queue_t myQueue = dispatch_get_global_queue
  (QOS_CLASS_BACKGROUND, 0);

NSLog(@"1");

dispatch_async(myQueue, ^{
  NSLog(@"2");
});

NSLog(@"3");

dispatch_async(myQueue, ^{
  NSLog(@"4");
});
```

prints 1,3,2,4

```
dispatch_queue_t myQueue = dispatch_get_global_queue
  (QOS_CLASS_BACKGROUND, 0);

NSLog(@"1");

dispatch_sync(myQueue, ^{
  NSLog(@"2");
});

NSLog(@"3");

dispatch_sync(myQueue, ^{
  NSLog(@"4");
});
```

prints 1,2,3,4

dispatch_sync & deadlock

- If you ever call dispatch_sync on a queue and pass in that queue as the queue to dispatch to, you will get a deadlock.
- This happens because:
  - dispatch_sync blocks the queue you called it from
  - this now blocked queue will never be able to perform the task that was passed into dispatch_sync because the queue is already blocked

GCD vs NSOperationQueue

- Sometimes NSOperationQueue is a better choice than GCD. The advantages of NSOperationQueue:
  - Canceling operations is possible in NSOperationQueue, GCD is fire and forget
  - KVO compatibility (more on this later in the week)
  - Operation priorities make it easy to modify how operations are scheduled

UIScrollView

Sourced From www.objc.io, issue 3, Understanding Scroll Views

UIScrollView

- A UIScrollView is technically a subclass of UIView, but its really just a UIView with a few simple added features.
- So to understand how scroll views work, you must first have a great understanding of how UIViews and the view hierarchy works in iOS.

Rasterization and Composition

- Rasterization and Composition combine to make up the rendering process of your iOS app.
- Rasterization is simply just a set of drawing instructions that produces an image.
- So a UIButton draws an image with a rounded rectangle and title in the center as its rosterization process.
- But these images are not drawn onto screen during the rasterization process, they are held onto by their views, in preparation for…

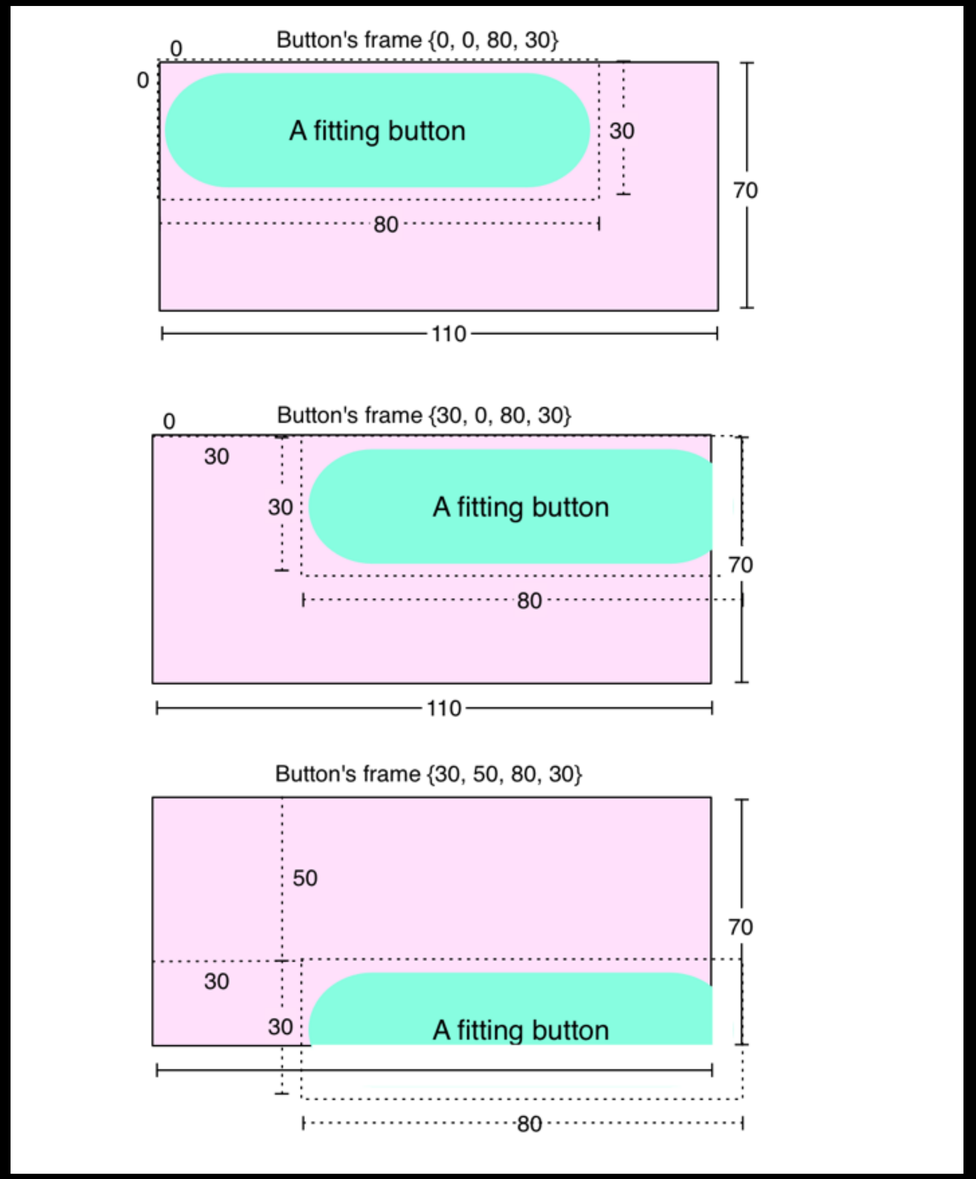
Composition

- The composition step involves all of the rosterized images being drawn on top of each other to produce one screen-sized image.
- A view's image is composited on top of its superview's image.
- Then the composited image is composited onto of the super-superview's image, and so on.
- The view at the very top of the hierarchy is the window, and its composited image is what the user sees.
- So basically, you are layering independent images on top of each other to produce a final, flat image.

Bounds vs Frame

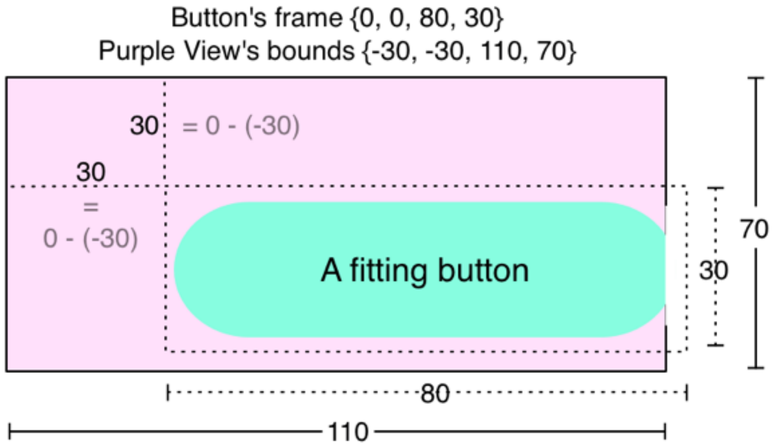
- So this is where bounds and frames come in to play.
- When the images are being laid out, the frame is used to figure out exactly where the image should be placed relative to its superview. So the frame is used in the composition step.
- During the rasterization step, a view doesn't care about its frame. The only thing it cares about is drawing its own content. Anything inside of its current bounds rect gets drawn onto the image. Anything that is outside of the bounds is discarded.
- Lets look at some pics!

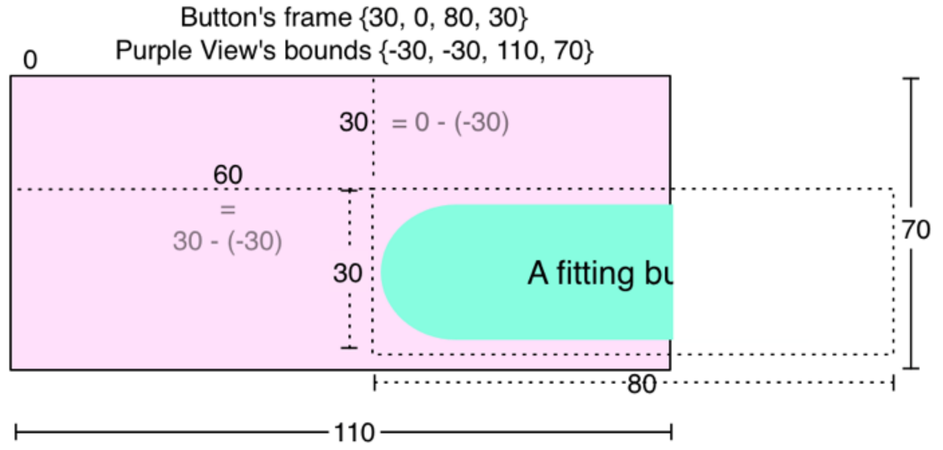
Frames

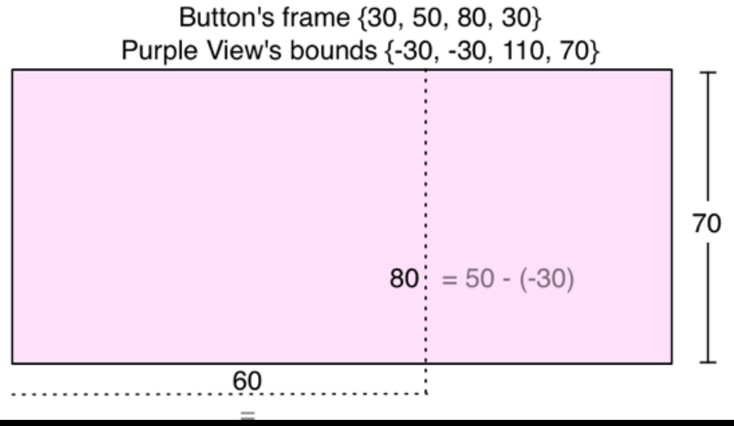

Button's frame {0, 0, 80, 30}
0
0
A fitting button
30
70
80
110
0
Button's frame {30, 0, 80, 30}
30
30
A fitting button
70
80
110
Button's frame {30, 50, 80, 30}
50
70
30
30
A fitting button
80

ScrollView

- So what does this have to do with scroll views? Everything.
- When you scroll a scroll view, the offset value changes. Under the hood, this is just changing the origin of the bounds of the scroll view.
- You could accomplish the same effect by changing the frames of all the scroll views subviews as it scrolls, but you would have to loop through every single subview every time the scroll view scrolls, which would suck.
- More Visuals!


Button's frame {0, 0, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
30
= 0 - (-30)
30
=
0 - (-30)
A fitting button
70
30
80
110


Button's frame {30, 0, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
0
30
= 0 - (-30)
60
=
30 - (-30)
30
A fitting bu
70
80
110


Button's frame {30, 50, 80, 30}
Purple View's bounds {-30, -30, 110, 70}
70
80
= 50 - (-30)
60
=

Scroll View’s contentSize

- A scroll view’s contentSize is the just the entire scrollable area.
- It doesn't effect the frame or bounds of a scroll view.
- By default it is set to 0,0
- When the content size is larger than the scroll view’s bounds, the user is allowed to scroll.
- Think of the bounds of the scroll view as a window into the scrollable area defined by the content size.

Scroll View's contentOffset

- "The point at which the origin of the content view is offset from the origin of the scroll view"
- The contentOffset is just how much the scroll view has scrolled.
- In addition to panning, you can set it programmatically with setContentOffset:animated:

Scroll View's delegate

- scroll view has an awesome delegate that is constantly notified of the state of the scroll view:
  - scrollViewDidScroll:
  - scrollViewWillBeginDragging:
  - scrollViewWillEndDragging:withVelocity:targetContentOffset:
  - scrollViewDidZoom:
  - etc

ScrollView and storyboards

- Scrollviews and storyboard are the worst of enemies
- It took a senior apple engineer 40 minutes to get a basic scrollview working with autolayout on the storyboard
- So the moral of the story is, if your scene needs a scroll view, do the scene entirely programatically.

Demo

Ternary Operator

Ternary Operators

- Ternary Operators in Objective-C allow you to do if-else conditionals in a short hand syntax
- Ternary Operators take this basic form:
  - expression_to_evaluation ? expression_if_true : expression_if_false

```
NSInteger x = 10;
NSInteger y = 9;

x == y ? NSLog(@"they match") : NSLog(@"they dont match");
```

Ternary Operators

- Its common to see ternary operators used to set variables:

```
NSString *result = x < 11 ? @"less than" : @"greater than";
```

- You can nest ternary operators

```
NSString *result = i % 2 == 0 ? @"a" : i % 3 == 0 ? @"b" : i
  % 5 == 0 ? @"c" : i % 7 == 0 ? @"d" : @"e";
```

- Don’t do this, its stupid. One ternary operator is barely readable on its own.

Demo

Using const

- Declare constants using the const keyword
- Any type can be a constant
- Value cannot change, once declared.
- Think of let keyword in Swift

```
NSString * const ReuseIdentifier = @“ReuseIdentifier";

int const limit = 100
```

Sharing const

- Expose constants in a header file to use from multiple points
- Define the actual value of the constants in an implementation file
- Useful technique to define constants and use across the app

Sharing const

- Create Constants.h and Constants.m
- In Constants.h, expose the const variables

```
extern NSString *const MyFirstConstant;
```

```
The extern keywords make this a globally available constant as long as you import constant.h
```

- Extern just lets the compiler know, and anyone reading your code that this variable is declared here, but defined somewhere else.
- In Constants.m, define the const variables value

```
NSString *const MyFirstConstant = @"FirstConstant";
```

Demo