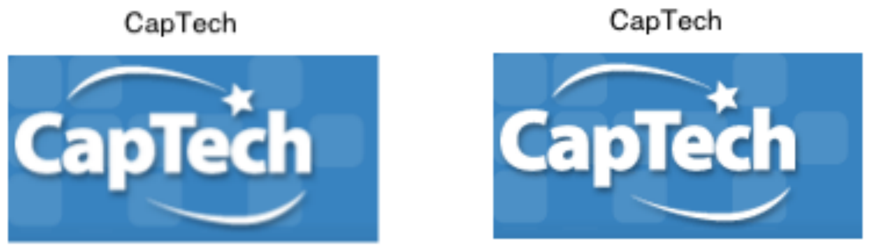
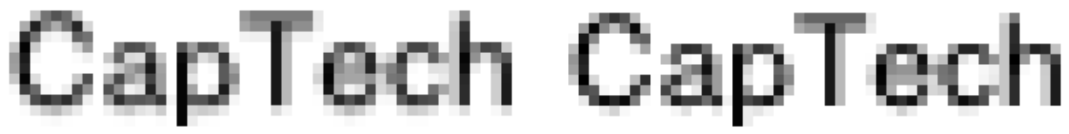
Using float values with UIView: Identifying Problems and Creating Solutions

If you have experience designing views in XCode you likely have run into an instance where a UIView or one of its subclasses is rendered blurry. This is caused by anti-aliasing, an attempt to avoid jagged looking edges by blending with the colors of the layer below it. This blog discusses both simple and complex issues with rendering UIViews with anti-aliasing.

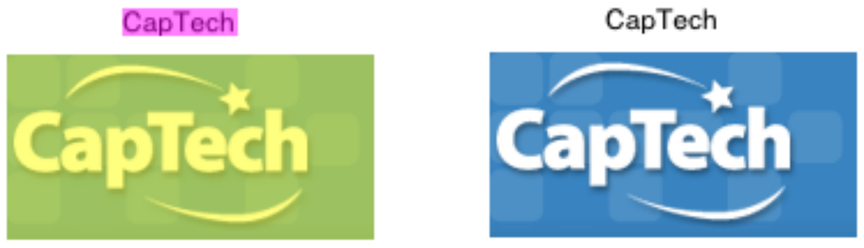
In the image below, the left text and image have been anti-aliased.



The difference can sometimes be subtle. The image of left below is anti-aliased.



You can see there are more gray pixels around the edges in the image on the left. The problem can snowball if a view is anti-aliased all of its subviews may be blurred as well. This can become more apparent on a retina device. Since it is not always easy to see where subtle blurs have occurred, the iOS Simulator has a nice feature, Debug->Color Misaligned Images. This will highlight the UIViews that have been misaligned so you can easily spot them.



The cause can be as simple as using float values for sizing views. Of course the simple solution is to always use an int type.

If only it were that simple. One of my favorite mathematical proofs is the proof of 2=1. Using only simple Algebra of doing the same operation to both sides of the equation repeatedly is needed. There is a fundamental flaw in the proof but it typically goes overlooked until you get to the final step where you show 2=1.

Without paying specific attention to view sizes, it is easy to overlook where float sizes might occur especially when you begin to size dynamically. For example you may want to center a view inside of another view by:

```
UIView *insideView = [UIView alloc]
initWithFrame:CGFrameMake(100,100,outsideView.frame.size.width/
2,outsideView.frame.size.height/2)];
```

If either the width or the height is an odd number of the outsideView you will get a misaligned view.

UIPagingControl

Other related issues to float sized views become more interesting, or infuriating depending on your view. Recently I created a view using something similar to a UIPagingControl. I had seven views evenly spaced out inside of each page. To attain the exact page size in the specification, I used a float value for each view and the overall size of each page was a float as well.

The app looked great on the first page; the animations within each view worked as expected. The Color Misaligned Images did not identify any issues. I then scrolled to the second page. It still looked great, but nothing animated. Scrolling to the third page threw the contentOffset out of whack. The debugger identified the contentOffset as an Absurd Value. We changed the specification by a couple of pixels to divide evenly by seven and all was well again.

The key here, like anything in the software development life cycle, is to start thinking about your design early in a sense to make sure that your views will come out to integral values.

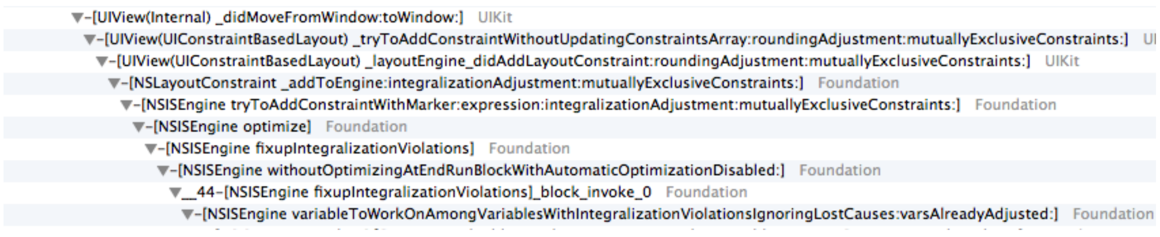
Constraints

Another way that views may be created dynamically is with constraints. When a view is related to another view with float sizes other hidden issues can surface. For example, lets say that you want to create three equally spaced views inside of another view using Visual Format Language.

```
NSString *format = @"|[view1][view2(==view1)][view3(==view1)]|";
```

In this case if the parent view is not evenly divisible by three, another type of issue may arise. In a recent project with similar constraint code to the one above there was a delay of over two seconds when pushing a new view controller on top of the one with these constraints, but only on retina devices. After digging into

Instruments, the didMoveFromWindow:toWindow was having difficulty optimizing constraints using NSISEngine as you can see in the screen shot below.



```
▼-[UIView(Internal) _didMoveFromWindow:toWindow:]  UIKit
 ▼-[UIView(UIConstraintBasedLayout) _tryToAddConstraintWithoutUpdatingConstraintsArray:roundingAdjustment:mutuallyExclusiveConstraints:]  UI
  ▼-[UIView(UIConstraintBasedLayout) _layoutEngine_didAddLayoutConstraint:roundingAdjustment:mutuallyExclusiveConstraints:]  UIKit
   ▼-[NSLayoutConstraint _addToEngine:integralizationAdjustment:mutuallyExclusiveConstraints:]  Foundation
    ▼-[NSISEngine tryToAddConstraintWithMarker:expression:integralizationAdjustment:mutuallyExclusiveConstraints:]  Foundation
     ▼-[NSISEngine optimize]  Foundation
      ▼-[NSISEngine fixupIntegralizationViolations]  Foundation
       ▼-[NSISEngine withoutOptimizingAtEndRunBlockWithAutomaticOptimizationDisabled:]  Foundation
        ▼__44-[NSISEngine fixupIntegralizationViolations]_block_invoke_0  Foundation
         ▼-[NSISEngine variableToWorkOnAmongVariablesWithIntegralizationViolationsIgnoringLostCauses:varsAlreadyAdjusted:]  Foundation
```

Once again, forethought in your design specification will save you headaches in the future.

Closing

This posts presents problems with non-integral UIView sizing. It shows how to identify and solve those issues. By thinking about sizing early in the software development lifecycle you can avoid absurd values such as 2=1. If you have questions you may contact me at strohtennis @ gmail.