Hi, I am Neha. I am an iOS engineer at Plenty of Fish. Today, I am excited to talk to you about advanced colors in iOS.

Color is a fundamental and important aspect of app development. It identifies brand, evokes emotion, and teaches us things like whether a button can be touched. An important observation is that most of the pixels you see on screen drawn by most applications, don't come from images, even though they tend to get top billing in this kind of talk. Most pixels on screen are actually solid colors drawn by your code in your application. In the last few years alone, Apple has introduced multiple technologies making it worth revisiting how we approach the use of color in our apps.

Agenda

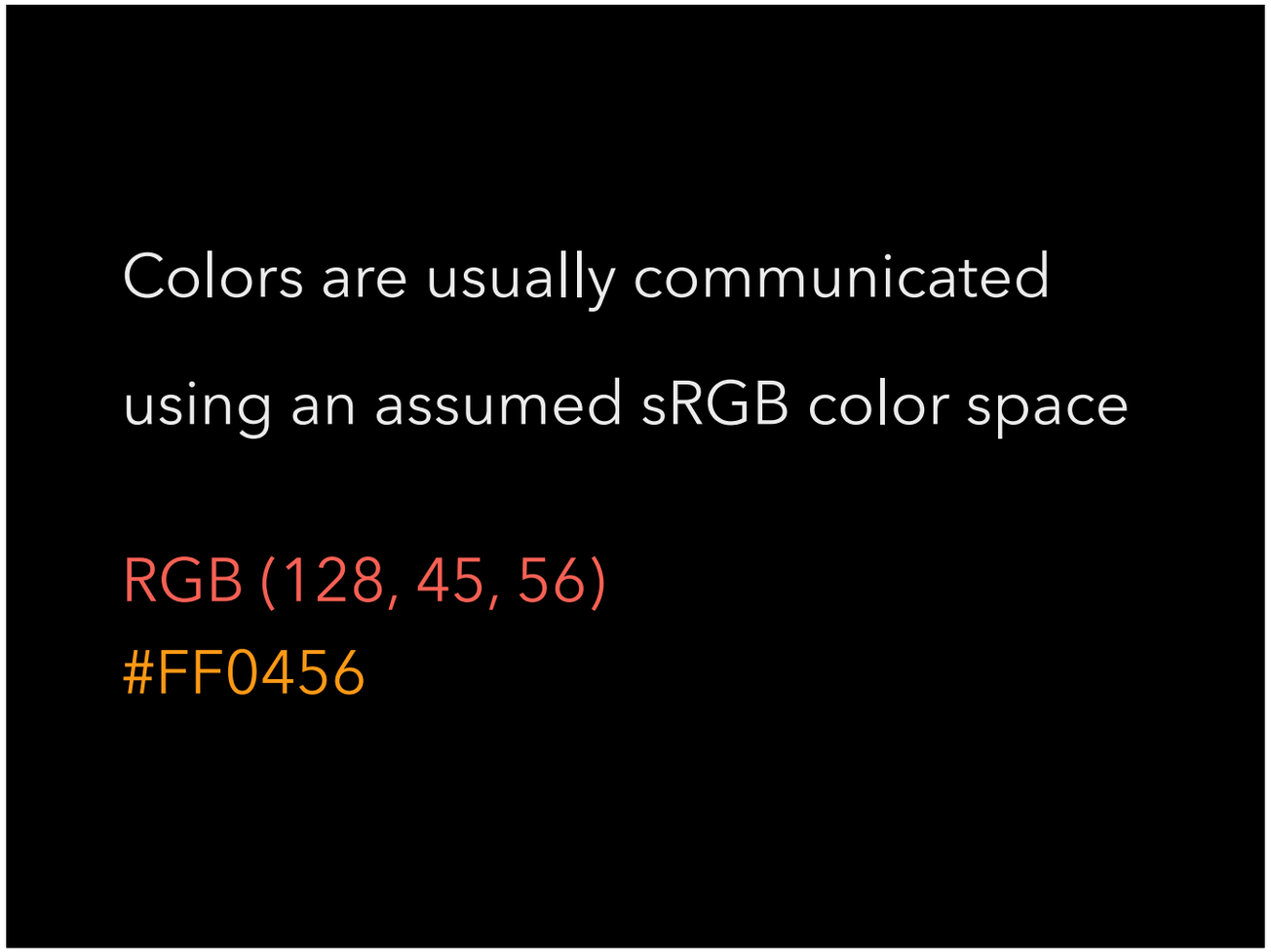
- Semantic colors
- Debugging named colors
- Name-spacing colors
- Dynamic colors
- Designers vs engineers

1. First, we will talk about semantic colors and if implemented well how they can improve the time spent on updating colors
2. Then we will talk about common problems we face debugging named colors
3. How we can name space the named colors
4. How to deal with dynamic colors
5. And lastly some of the collaboration techniques working with designers

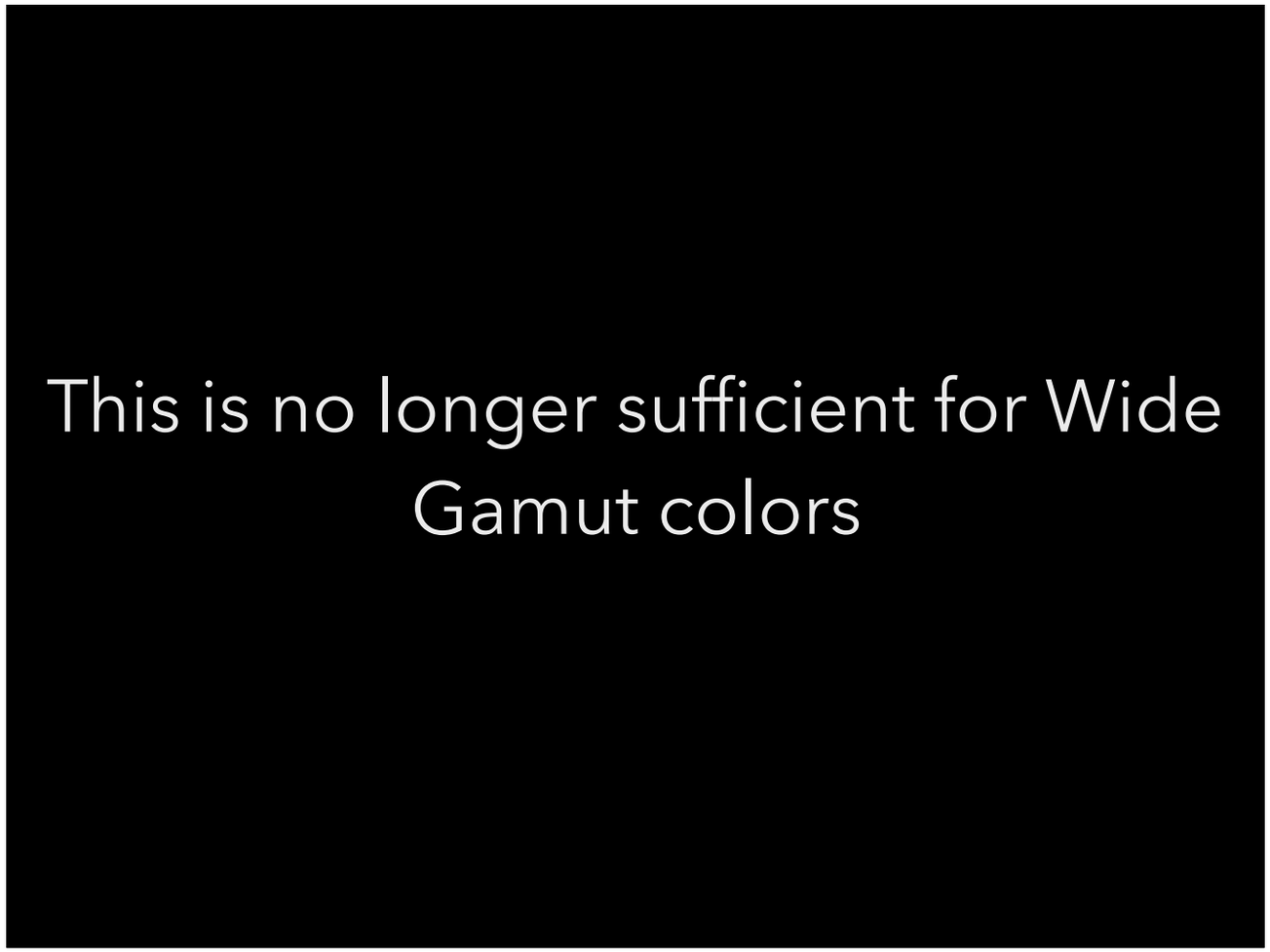
COLOR SPECIFICATION

But first… let’s talk about the color specification

So, first I want to talk about this under-appreciated problem of specifying colors. Usually when designers and engineers communicate in written form, or perhaps even verbal form, or visual form code is usually communicated with an assumed sRGB color space. That means you're probably used to seeing colors written down as something like this



RGB 128, 45, 56. This notation doesn’t specify what color space the color is in. And it's just assumed everybody knows what color that is, because everybody's using sRGB, aren't they?



Well, not anymore. This is no longer sufficient for working with wide gamut colors. So what do you do? All iPhones prior to iPhone 7 only represent colors within the sRGB color space, whereas, newer Apple devices, such as the iPhone X, use a bigger color space that covers a wider gamut of colors.

- Be specific about color space!
- Use Display P3 instead of sRGB when working with wide gamut designs

  P3 (255, 128, 191)

The most important step you can take is be specific about what color space you're working in when you communicate.
Use Display P3, instead of sRGB when you're working on wide gamut designs. And indicate that as such.

PICKING COLOR

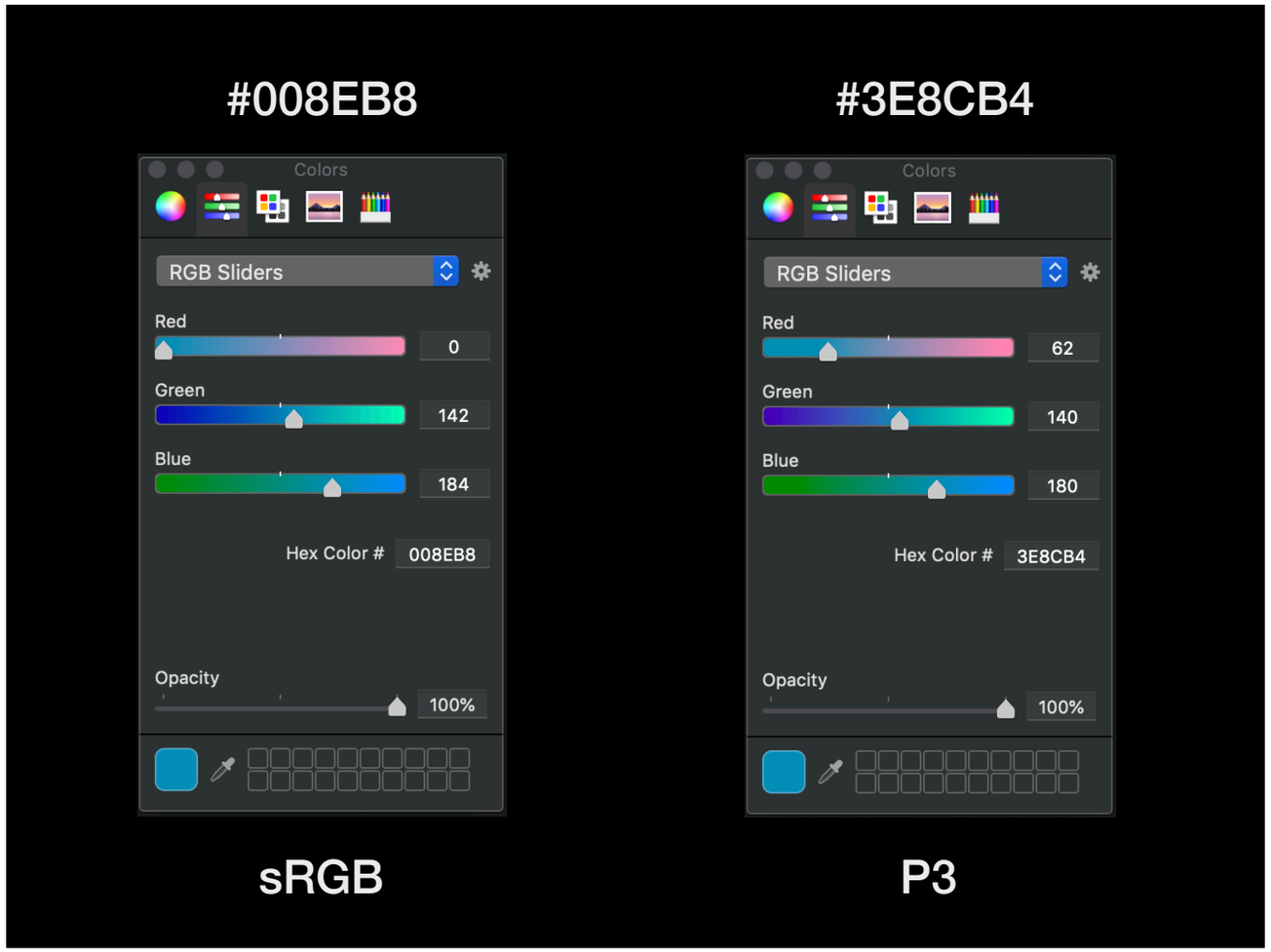
Let’s talk a bit about picking a color

We just saw how to communicate a color, but where did that color come from in the first place. You probably picked it using a color panel.

This is the standard color panel that gets shipped with the Mac, known as NSColorPanel. This of course is a very familiar UI, but it also suffers from some of the limitations we just talked about. Typically, you pick red, green and blue values, 0 to 255 numbers. Though selecting in different color spaces hasn't always been a very obvious or easy user experience. Xcode doesn't readily show you the selected color space. It's off in a side menu, because the Xcode color picker treats the color space like a viewing preference rather than component of the color.

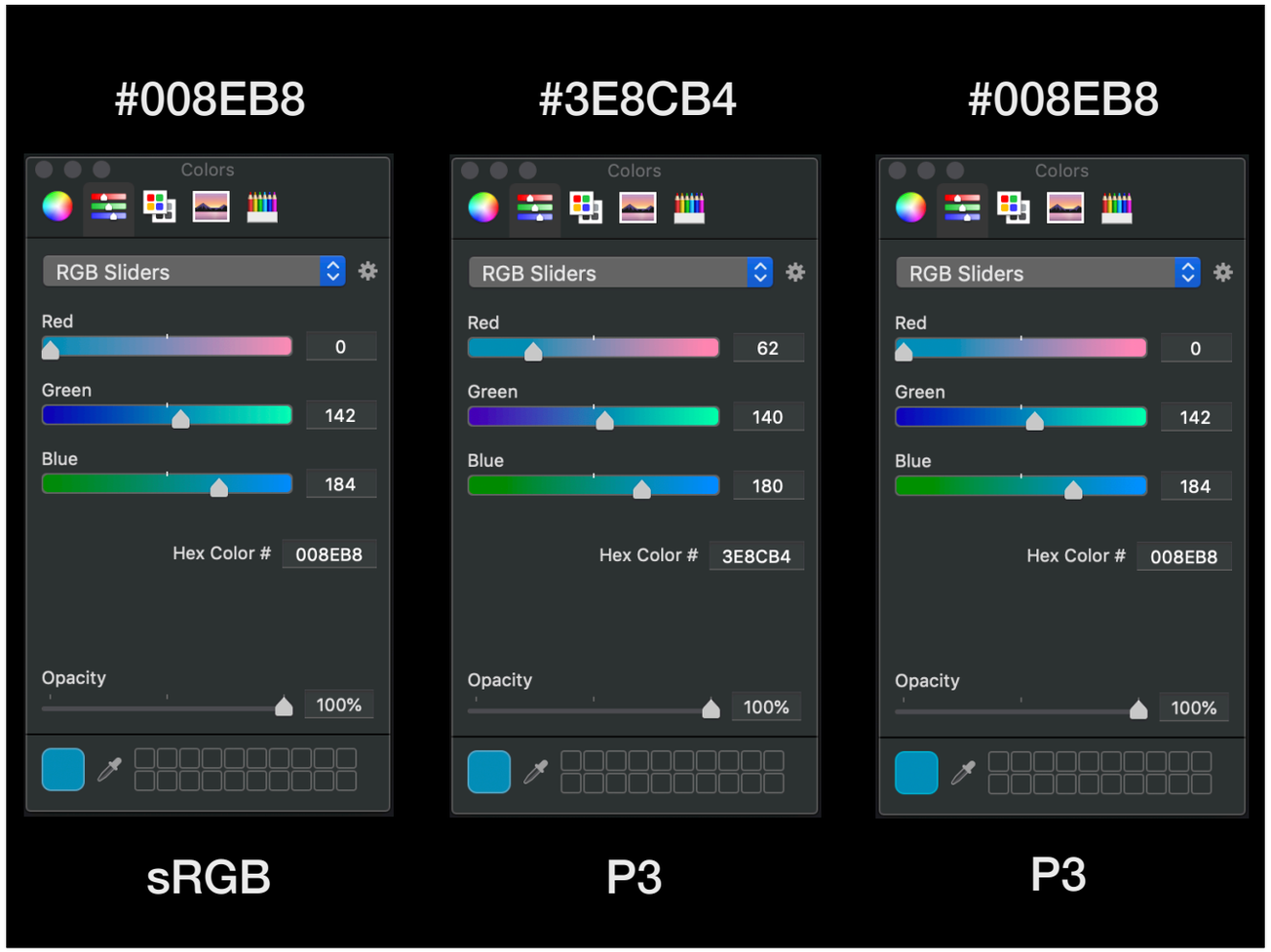
This can be annoying because, not only is it not obvious what color space is currently selected, but also because changing the color space transforms your RGB values to keep your selected color the same. CHOOSE YOUR COLOR SPACE BEFORE ENTERING IN YOUR VALUES!

Okay, moving on… Dark mode is hot! And nowadays everybody is opting for dark mode…



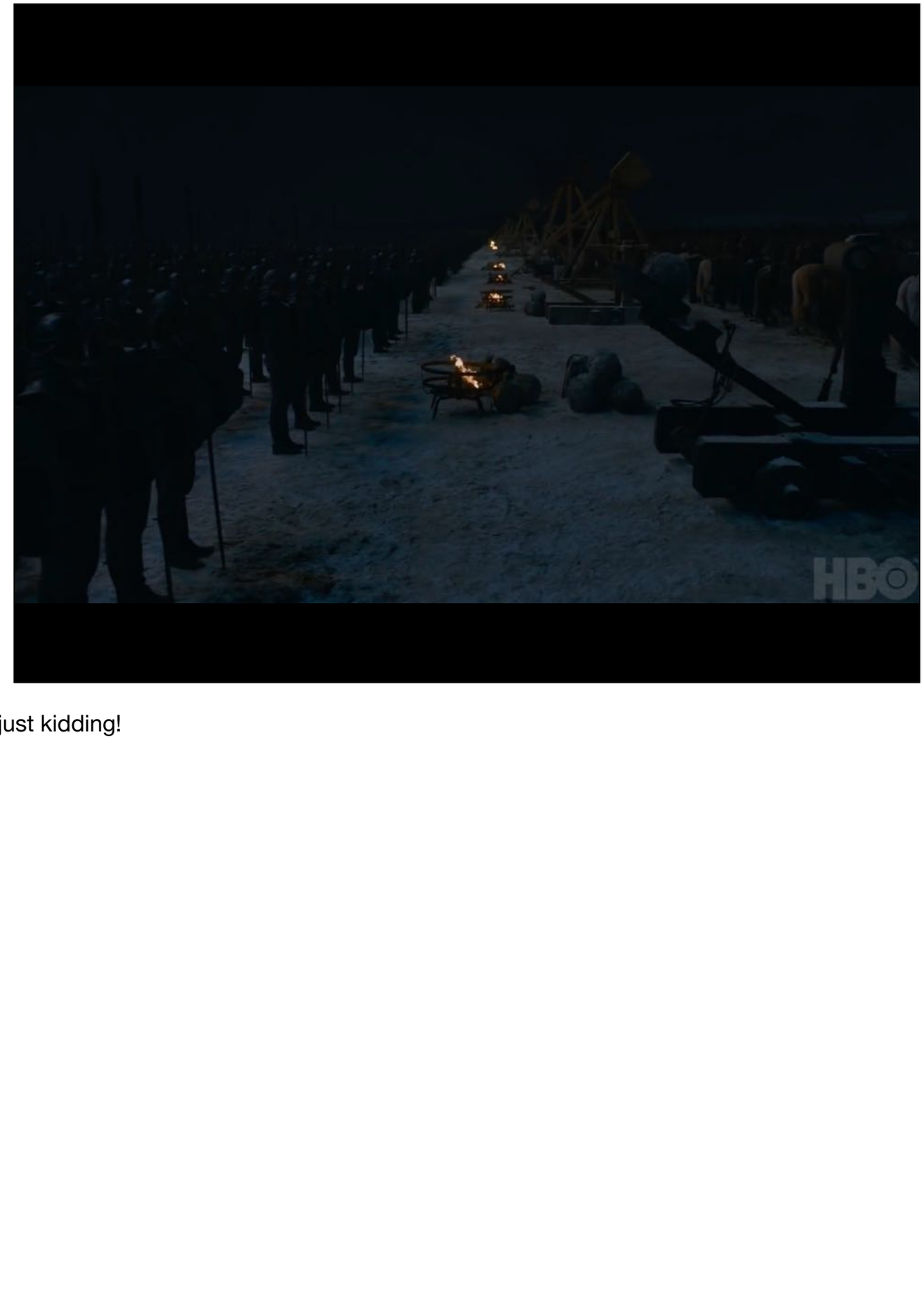
This can be annoying because, not only is it not obvious what color space is currently selected, but also because changing the color space transforms your RGB values to keep your selected color the same. CHOOSE YOUR COLOR SPACE BEFORE ENTERING IN YOUR VALUES!

Okay, moving on… Dark mode is hot! And nowadays everybody is opting for dark mode…



This can be annoying because, not only is it not obvious what color space is currently selected, but also because changing the color space transforms your RGB values to keep your selected color the same. CHOOSE YOUR COLOR SPACE BEFORE ENTERING IN YOUR VALUES!

Okay, moving on… Dark mode is hot! And nowadays everybody is opting for dark mode…



That’s Game of Thrones in dark mode, just kidding!

What's coming for iOS 13:

- Codenamed "Yukon," the refreshed operating system for iPhones and iPads includes tweaks and new features across several apps, including features originally planned for last year. The company is also already working on iOS 14, codenamed "Azul," for 2020. That release is expected to support 5G wireless network speeds and new AR functionality for next year's iPhone.
- The software, for the second year in a row, will speed up the devices and reduce bugs. There will be user interface tweaks, including a new animation when launching multitasking and closing apps. The widgets that appear to the left of the home screen will also have a cleaner look.
- A Dark Mode, a black and grey-heavy interface optimized for viewing at night, that can be enabled in Control Center, the panel for quickly accessing settings.
- The company is testing a new keyboard option that allows users to swipe across letters on the keyboard in one motion to type out words (Apple could choose to keep this feature internal). This is similar to options on Android handsets and it would compete with third-party iPhone apps such SwiftKey.
- A revamped Health app with a new homepage that better outlines your daily activity from the day. There will be a section for "hearing health," like how loud you play music on your headphones or the loudness of the external environment. It also includes more comprehensive menstrual cycle tracking, vying with period-tracking apps such as Clue, Flo and Ovia.

Bloomberg published an article stating that iOS 13 will have a dark mode that can be enabled via control panel of your iPhone

And since the iOS dark mode is a big deal, it makes semantic colors all the more important. Let’s talk about how we can leverage semantic colors to support the dark mode in our app.

What are semantic colors? Semantic colors are colors with a meaning. Semantic in the color names help us visualize the meaning or value state of the color. Let’s take a look at some examples…

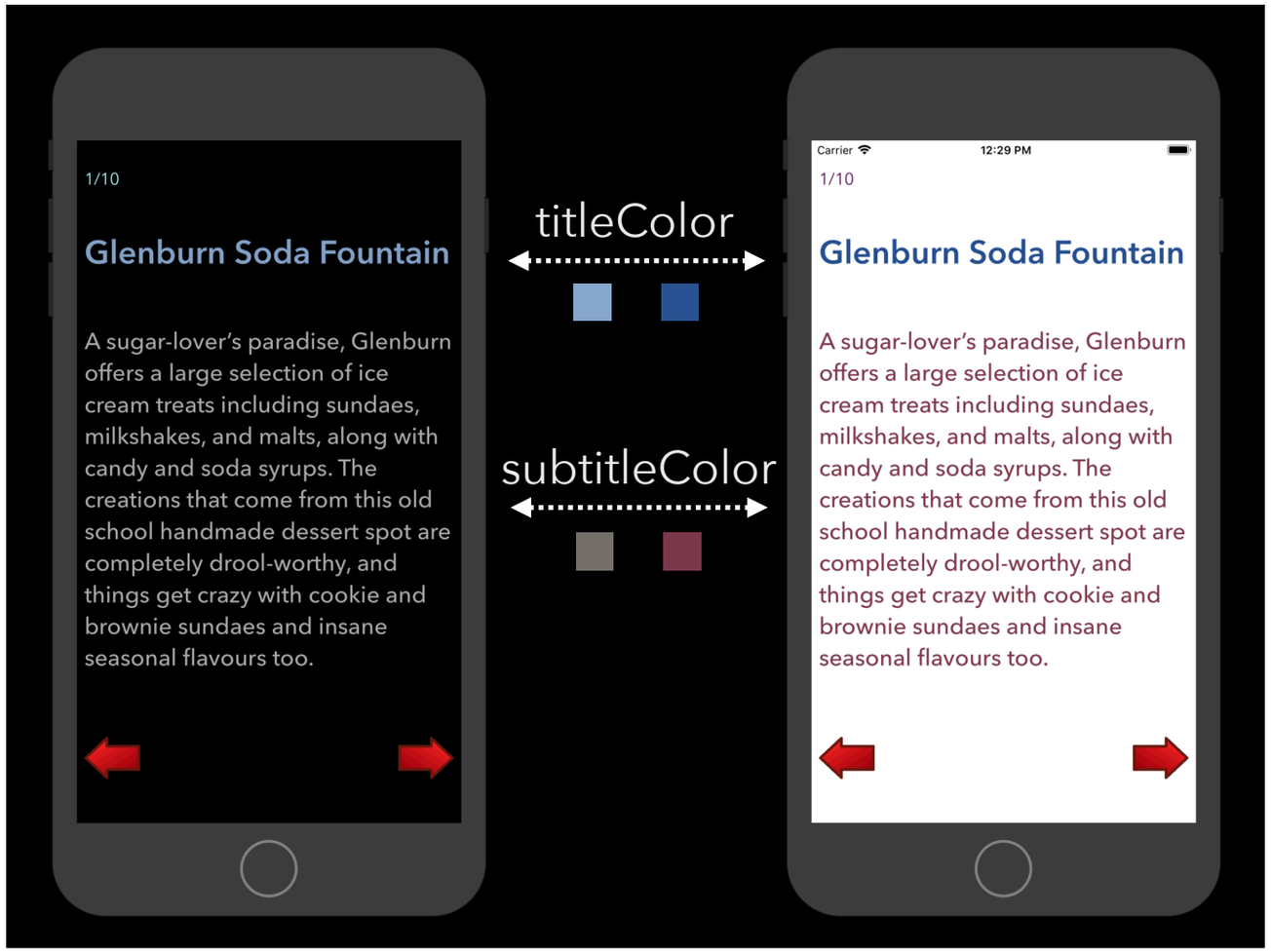
Do these colors look like semantic colors? No! These are the palette colors which are names indicative of the color’s hue. This helps in distinguishing between colors in our palette at a glance. These are also called as declarative colors!

- blue, gray, darkGray
- primaryText, destructiveButtonText

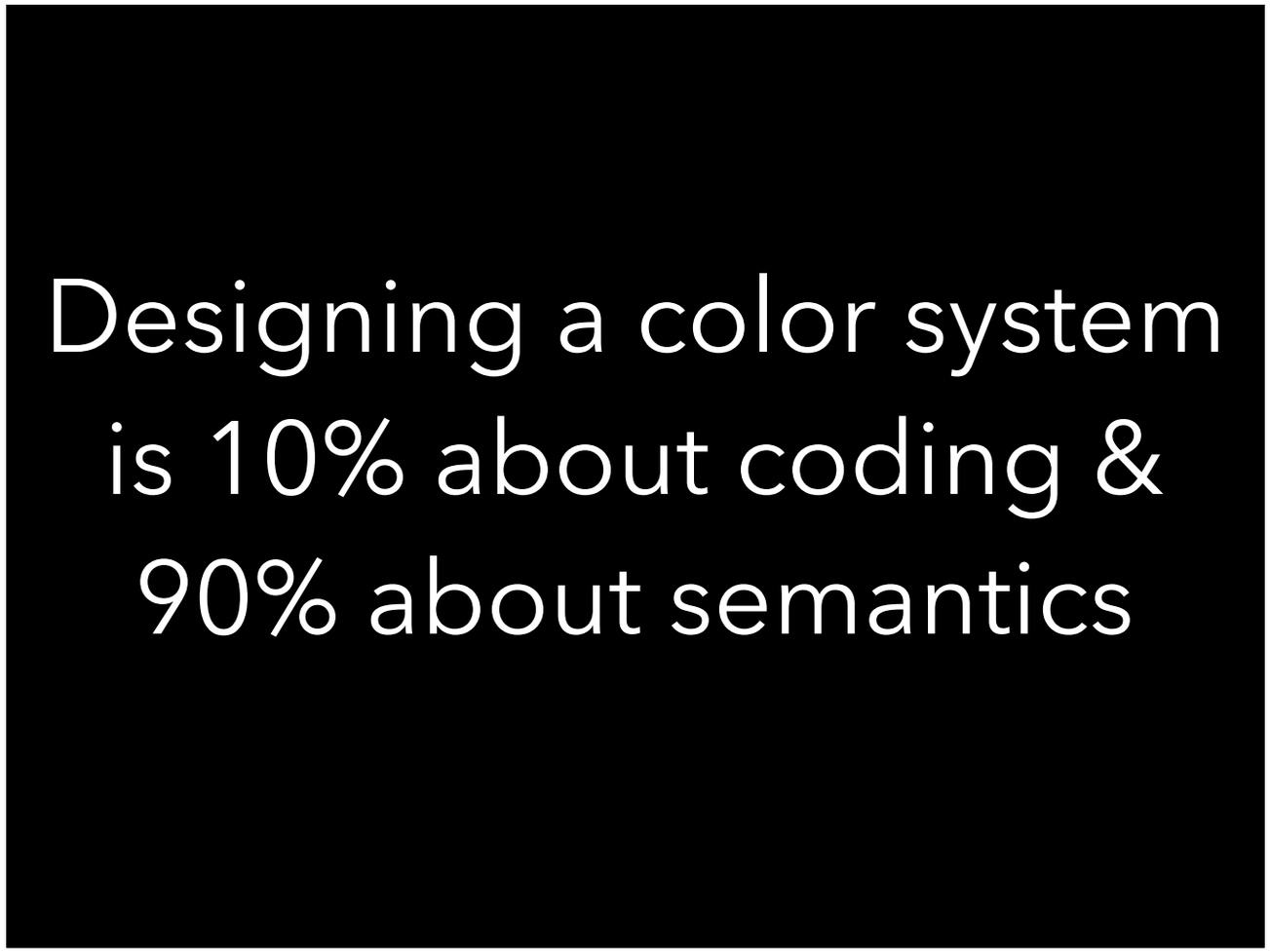
These colors do add a meaning to the underlying colors. In this case, they indicate the primary text color and destructive button text color respectively

- blue, gray, darkGray
- primaryText, destructiveButtonTitle
- dashboardButtonTextColor, tokenScreenBackgroundColor

This is an another example of how semantic color could look like, although this one looks too specific.
Depending on how granular you want to get with naming your colors, you'll end up with a number of semantics that resolve to the same underlying color. That doesn't necessarily mean that such "duplicate" semantics need to be consolidated, because you want the flexibility to change the unrelated things independently. In general I recommend being as generic as possible unless required to be specific.



As you can see from the above diagram, although my colors for the dark & light modes have changed, the semantics for the colors are intact. Which essentially means I have to change/introduce almost no code, when changing the themes. This can be extremely useful when you are supporting different themes or A/B testing colors. Semantic colors really shine when combined with named colors to support these kind of theme changes.



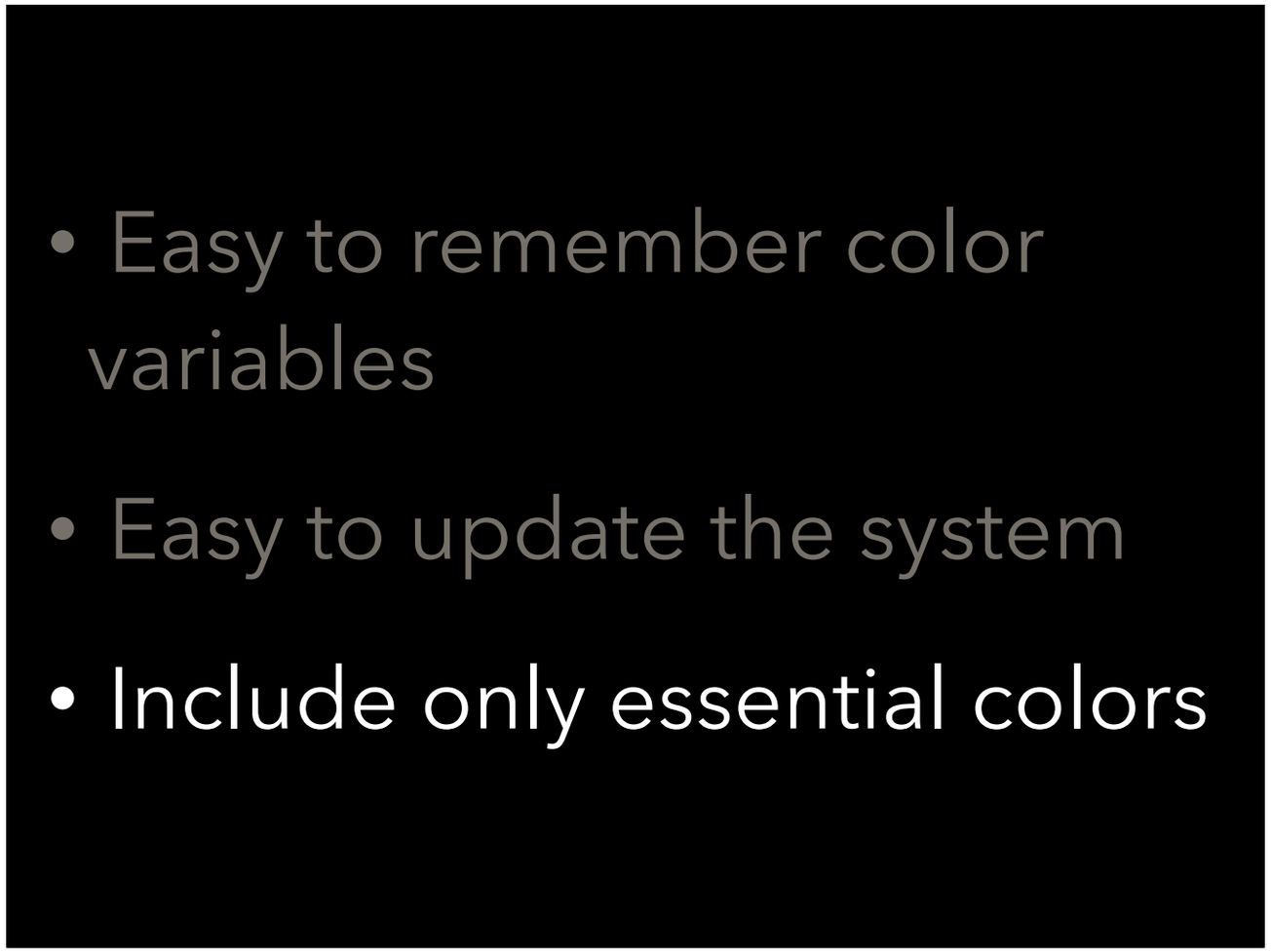
By creating a semantic color system, you are creating an abstraction layer which makes it easy to experiment with color palettes without worrying about specific code changes. It makes drastic re-designs less painful and easier to plan. Creating semantic colors does not mean incrementing the number of colors, but distributing the colors using semantic references.

- Easy to remember color variables
- Easy to update the system
- Include only essential colors

So things to keep in mind while designing your color system-
You don’t want to check the global file anytime you have to pick a color

- Easy to remember color variables
- Easy to update the system
- Include only essential colors

You will add, remove, and rename colors. Make sure doing so is not complicated.



We've heard this one so many times…yet we always end up with more colors than we need! The real success key of any system is removing all that is not necessary (colors included).

Traditionally in our iOS app at Plenty of Fish, we do a lot of experiments. The average time spent on color experiments until we implemented the semantic color system recently ranged from half an hour up to a week every sprint.
After we redesigned our system to include and use the semantic colors, it got drastically reduced as we simply had to make our code point to a specific palette of colors.

DEBUGGING NAMED COLORS

Named colors were introduced as one of the minor but intriguing features of iOS 11 SDK which received almost no on-stage attention. The immediate benefit of named colors is quite simple…

- Changing colors quickly & comprehensively
- Synchronizing color definitions across code & IB

I learnt some things that I would like to share while using the named colors

ACROSS BUNDLES

When I started using named colors, I defined the named colors in a asset bundle and the consumer of those colors was a component library that we had made for reusable UI components, when I tried using the colors in my consumer library, I couldn’t access it. Why did it happen? Just how UIImage(named:) works In iOS apps, the main bundle takes precedence by default and could be the only place where your app looks for resources, unless told otherwise. I am sure most of us have seen this problem happening.

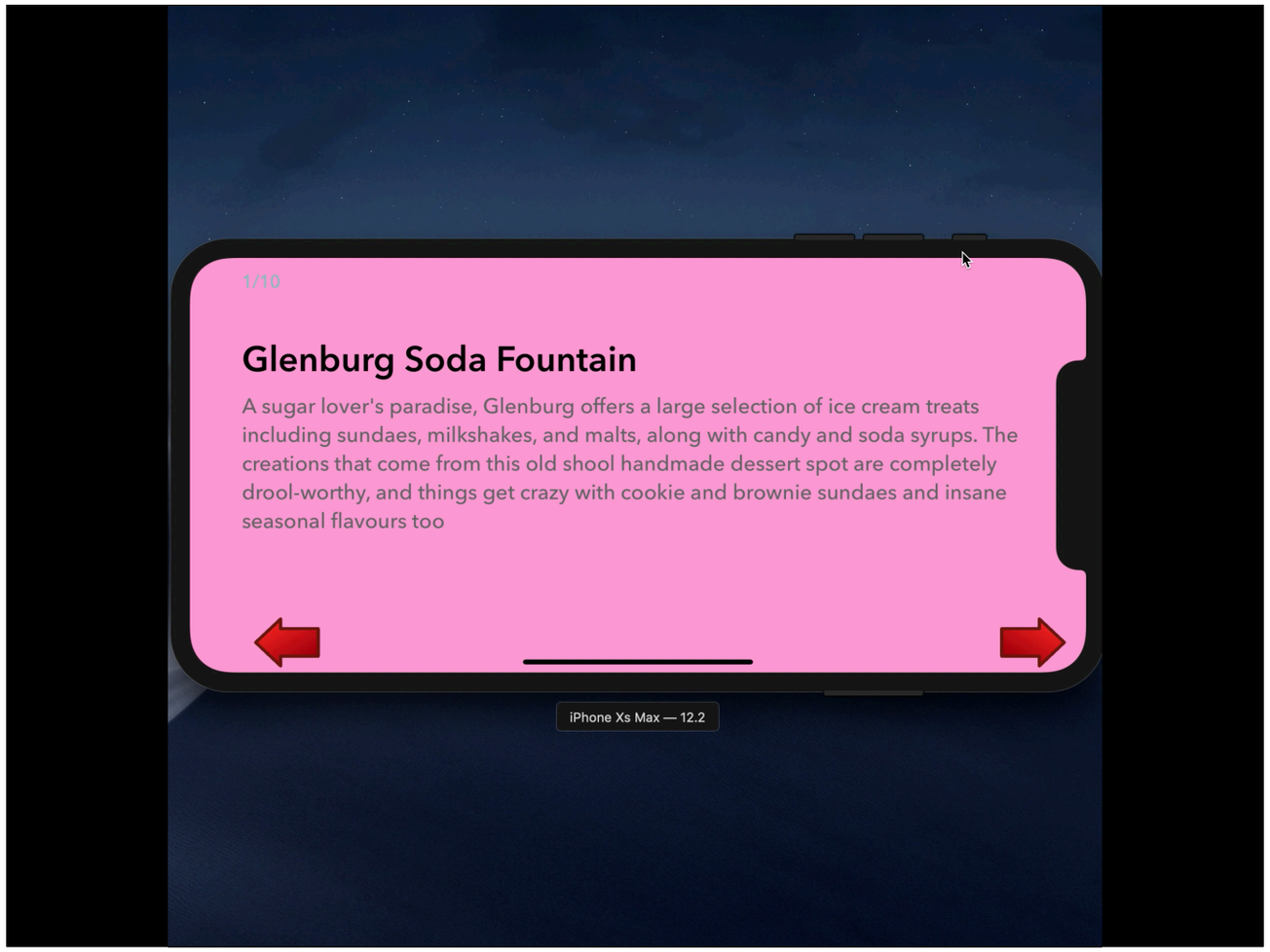
```
let colorFrameworkBundle = Bundle(identifier: "com.colorBundle.UIColors")!
label?.textColor = UIColor(named: "primaryText", in: colorFrameworkBundle,
    compatibleWith: nil)
```

On the code side of things, there’s an easy fix. Whenever you call UIColor(named:), simply ensure to set the optional in bundle: parameter. If you are looking for an IB level solution, you can create a separate bundle for assets and copy it into your targets’s compile sources and make sure they are compiled before you app.

The way Apple has gone about implementing named colors in Interface Builder is that once they are applied to an element, they get “baked into” that element’s trait collection attributes.

```swift
override func viewDidLoad() {
    super.viewDidLoad()
    updateColor()
}

func updateColor() {
    if someCondition {
        someCondition = false
        donateLabel.textColor = UIColor(named: "Dashboard/primaryText")
    } else {
        someCondition = true
        donateLabel.textColor = UIColor(named: "Dashboard/secondaryText")
    }
}
```

If you have an app, and some of the views have their background colors changed dynamically after loading based on user settings, and if you are using named colors, your code changes will be surely overridden if you are making them in the viewDidLoad() method



At runtime, the system will apply that stored value in its corresponding trait collection environment, overriding any code changes to that element’s color. Which means that once they are set in IB, the code changes won’t take effect unless you change the color one run loop after viewWillAppear

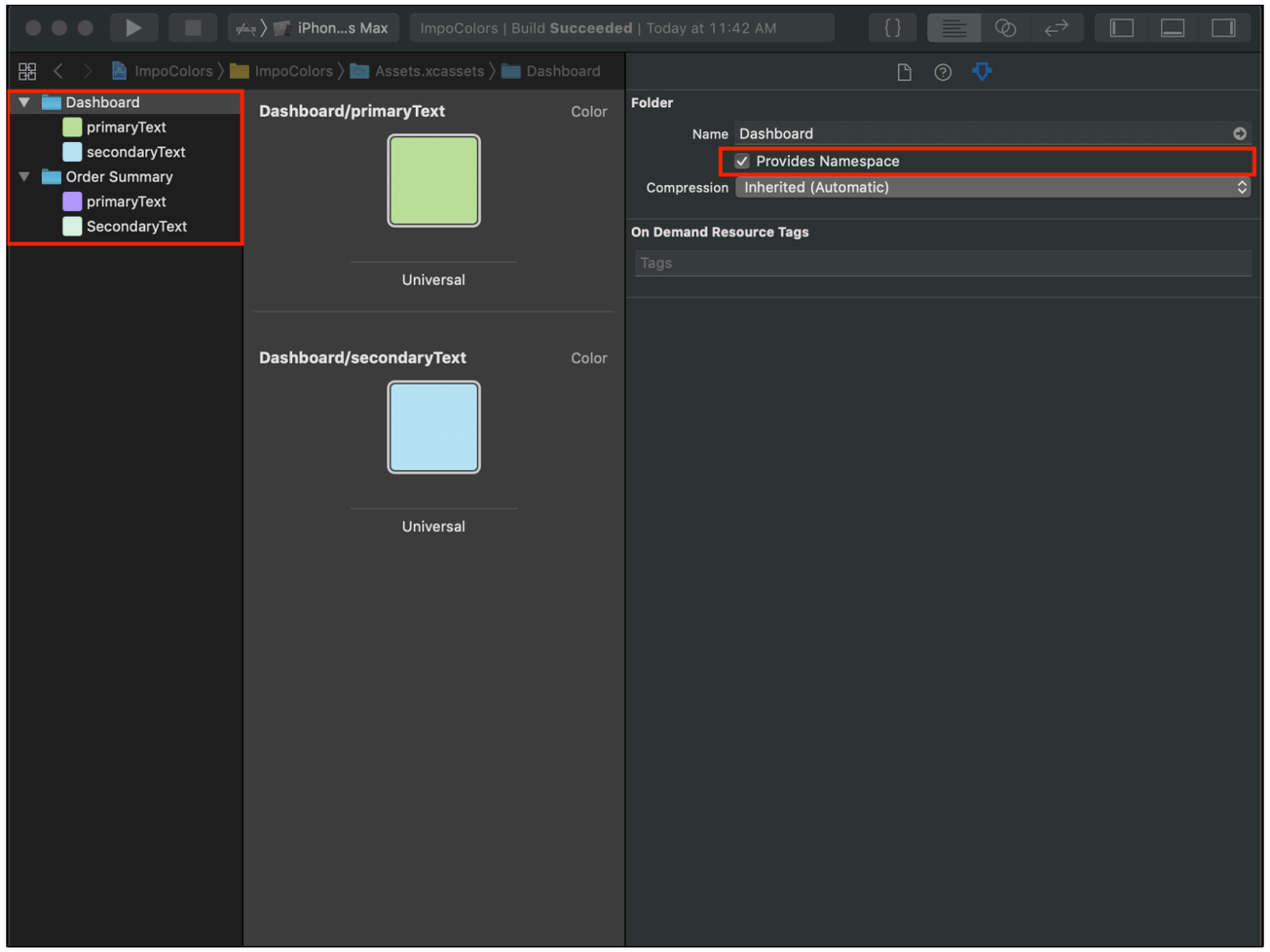
```swift
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    if let color = UIColor(named: "primaryText") {
        label.textColor = color
    }
}
```

The solution at this moment is to use viewDidAppear() method to override the colors

Some of these problems may cause some friction but aren’t insolvable, in fact named colors have evolved a lot since its inception. I would encourage everyone to leverage this nice feature. Just like we don’t like magic numbers hard coded in our apps, moving colors into our assets means we can be responsive to the designers’ needs without having to break a sweat. We can look like heroes!

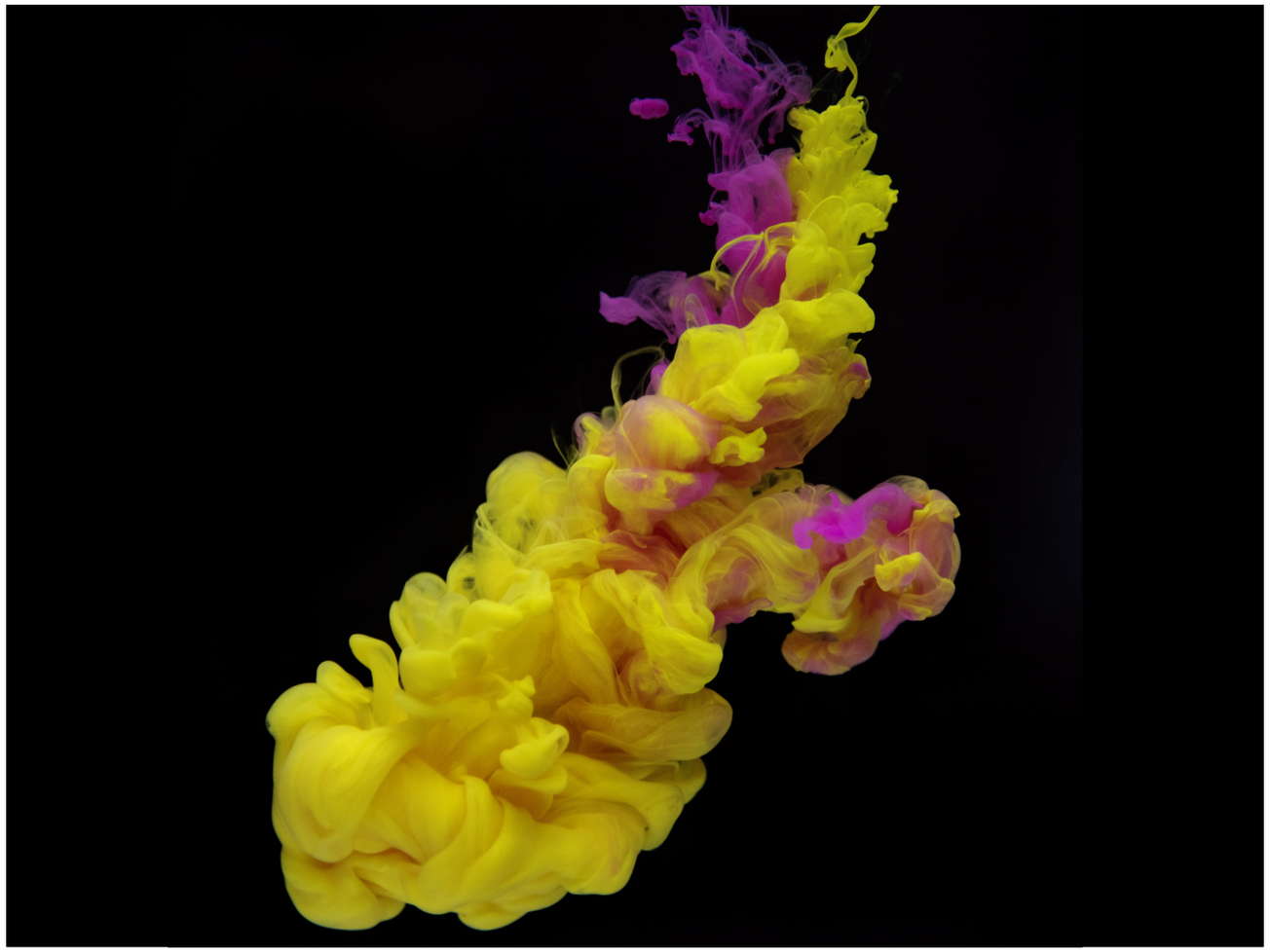
NAME-SPACING COLORS

When we talked about semantic colors in earlier slides, we talked about striving for more generic color names. We also talked that having multiple semantics that point to the same underlying color is normal. But what about the reverse? What to do when multiple colors have the same semantic name but in different context? Let's take an example- You need to support 3 different colors with a single buttonText color semantic? Will you create 3 variants like buttonText1, buttonText2 etc? Or will you be more specific like dashboardButtonText, orderSummaryButtonText etc? How about calling all those with just buttonText semantic? We can achieve that using name-spacing! In general, name-spacing is a technique employed to avoid collisions with other objects or variables in the global namespace. By applying the same terminology to colors, you can organize colors in your application into easily manageable groups that can be uniquely identified.



Luckily we can avoid the naming wars (hopefully!) with this technique. You can provide namespaces for assets, but by default everything has global scope. Enabling provides namespace will let you control asset scope via the group name. Using named colors with name spacing gives us much richer semantics, avoids name collisions and provides intuitive code-completions from your editor!

DYNAMIC COLORS



When your fixed color asset becomes RESTful, it’s called as dynamic color. Imagine a situation when you want to update the color of an animation in the app for upcoming holiday season based on a certain value you would receive from the backend, that’s when you are using a dynamic color.

- Colors delivered over the network
- Involving the design team
- Auto updating the colors

Dynamic colors can be of 2 types

- Colors delivered over the network
- Involving the design team
- Auto updating the colors

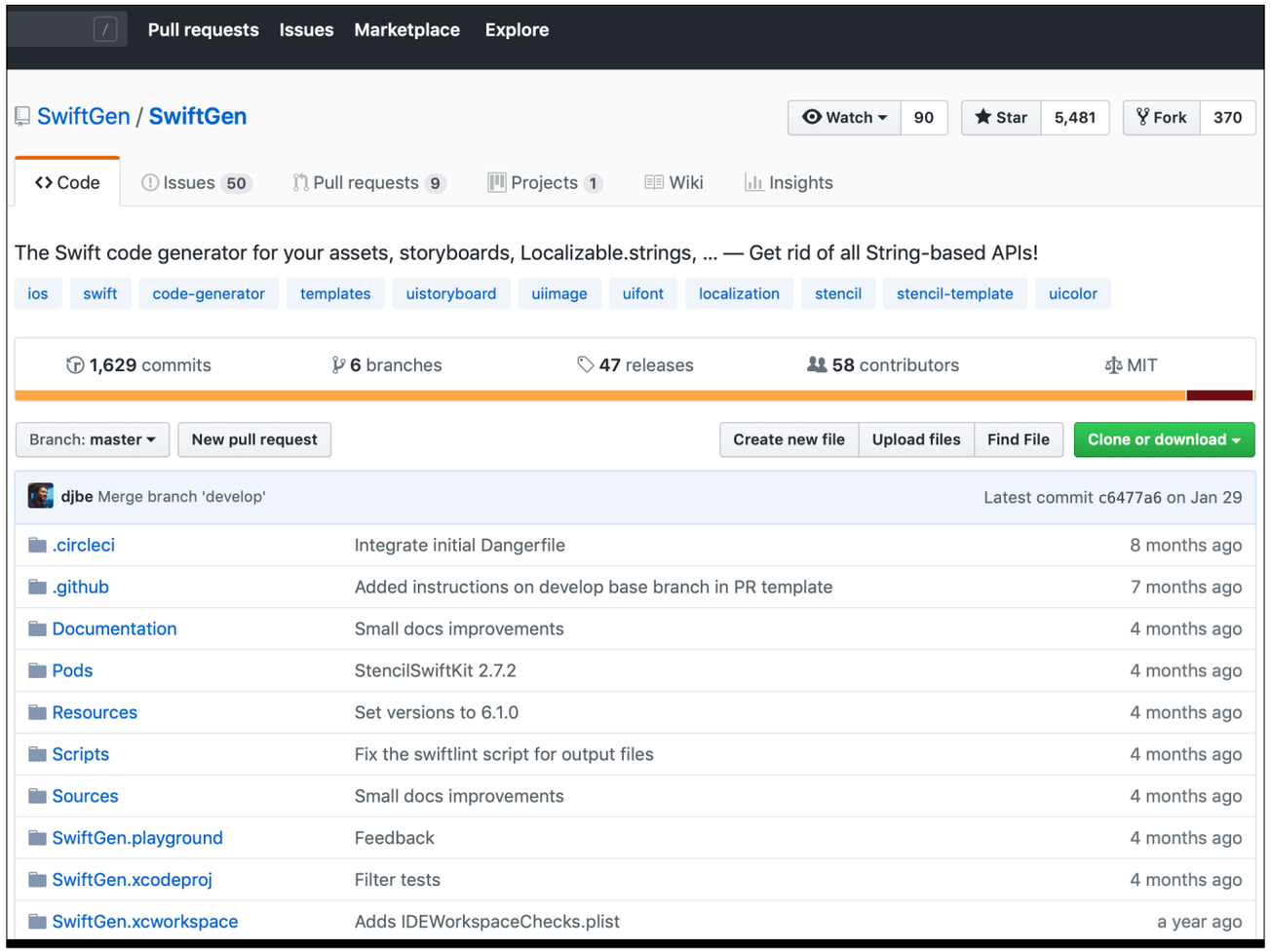
Let’s take a simple example of being update to animation on a view when the colors are delivered dynamically via json.

```
struct DynamicColor: Decodable {

    weak var animationDelegate: Animatable?
    private enum CodingKeys: String, CodingKey { case name, color }

    var name: String
    var color : UIColor {
        didSet {
            animationDelegate?.didSet(color: color)
        }
    }

    init(name: String, color : UIColor) {
        self.name = name
        self.color = color
    }

    init(from decoder: Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        name = try container.decode(String.self, forKey: .name)
        color = try container.decode(Color.self, forKey: .color).uiColor
    }
}
```

UIColor does not conform to Codable protocol, but its individual components like R G B values do. You would be able to take advantage of Decodable protocol combined with property observers to update the animation color. One area in which property observers in Swift really shine is when we want to configure a view based on a given value. Here we're using a didSet property observer to propagate changes to a color variable received from a son down to an underlying animation delegate, that's used to render the animation

- Colors delivered over the network
- Involving the design team
- Auto updating the colors

An .xcassets “file” is nothing but a collection of folders with JSON manifest documents that describe the assets contained within. In the case of named colors, there are no resource files in there either. Designers can easily take advantage of that and be able to create pull requests to add/remove/update the colors in your app.



SwiftGen is provided as a single command-line tool which helps you generate input resources like strings files, IB files, Font files, JSON files etc into constants for each types of those input files.

- Colors delivered over the network
- Involving the design team
- Auto updating the colors

Matt from NSHipster has an excellent article about dataset synchronization technique to pre-load the data from a json file and make it a part of our release process.

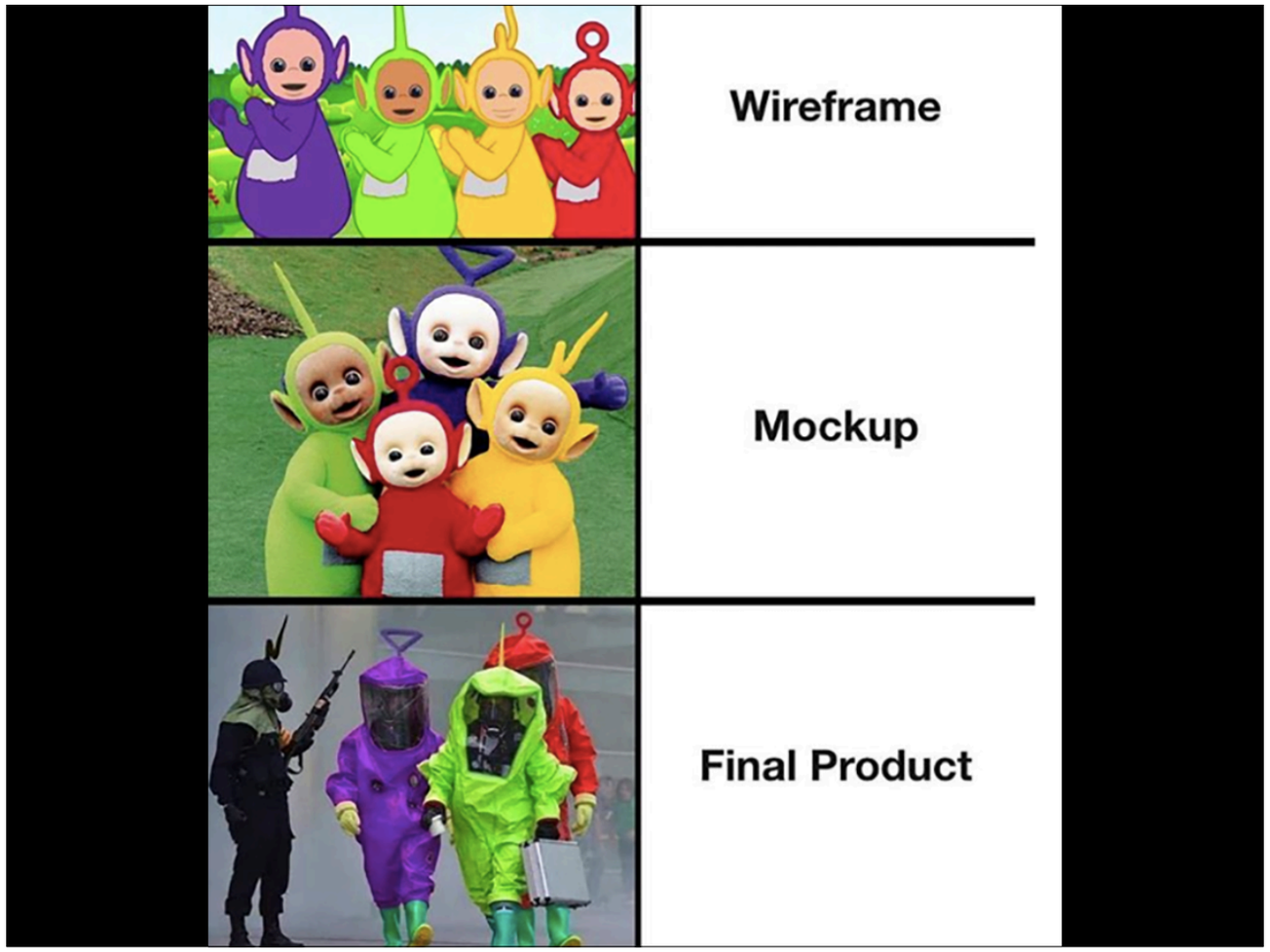
DESIGNERS VS ENGINEERS

Finally, let’s talk about designers vs engineers collaboration techniques

App designers and developers around the world work side by side every day to build some of the most incredible and innovative applications. Yet our potential for synergy is so often overlooked.

- Communicate early & often
- Follow naming conventions
- Follow clear specifications
- Single source of truth

- Communicate early & often
- Follow naming conventions
- Follow clear specifications
- Single source of truth



As you’re implementing a design, make sure to consistently show the designer your progress. Designers love seeing their work come to life, so it’s really a fun thing for everyone. Keeping the designer up to date with your progress will help ensure that your implementation is up to expectations, and there aren’t any surprises down the road. It will also help avoid situations like this

- Communicate early & often
- Follow naming conventions
- Follow clear specifications
- Single source of truth

Design handoff remains one of the biggest challenges that developers deal with. In order to ensure a smooth handoff both designers as well as engineers must agree upon a consistent asset naming convention and follow that. When using named colors, developers can detect any discrepancies in the naming early on while reviewing the design pull requests

- Communicate early & often
- Follow naming conventions
- Follow clear specifications
- Single source of truth

Proper design specifications allow a developer to see not only the colors, fonts, and measurements of selected elements, but also the flow of the app as intended by the designer. This saves the designers time by reducing the tedious step of marking up the designs. Instead, a designer can spend more time considering the user experience, while a developer focuses on how to implement it.

- Communicate early & often
- Follow naming conventions
- Follow clear specifications
- Single source of truth

Engineering & designing teams could agree before hand a single place where all of the approved deliverables can be accessed. This would ensure that everyone knows where to go to check out the necessary assets.

Drink a beer with your designer!

Never underestimate the power of socializing with team members! Get to know them, and allow them to get to know you. You can accelerate trust and communication if someone feels you care about them as a person —and not just a set of skills that you rely on to realize a design vision.

INCLUSIVE COLORS

This talk about colors would be incomplete without talking about how we can design for inclusivity. Color deficiency has a far deeper reach than most realize. It affects approximately 8% of men and .5% of women in the world. Apple introduced an incredible color filter accessibility feature in iOS 10. These filters do a remarkable job in increasing the contrast of all colors throughout the entire iOS system.

- Use both colors and symbols
- Keep it minimal
- Avoid bad color combinations
- Use clearly contrasting colors & hues

Let’s talk about simple changes we can make in our apps to help the ones afflicted by colorblindness

- Use both colors and symbols
- Keep it minimal
- Avoid bad color combinations
- Use clearly contrasting colors & hues

First, strive not to rely on color alone to provide feedback for your users. Certain types of users afflicted with the red-green limitations might miss the message when other visual cues are lacking.

- Use both colors and symbols
- Keep it minimal
- Avoid bad color combinations
- Use clearly contrasting colors & hues

Complex multilevel interface may be a real issue for disabled people, especially for those with vision disorders. That’s why it’s important to keep it simple and consistent. To scale a font with no design damage try using accessibility feature Larger Text.

- Use both colors and symbols
- Keep it minimal
- Avoid bad color combinations
- Use clearly contrasting colors & hues

You need to be smart when picking out your color combinations. Since color blindness affects people in different ways, it’s difficult to determine which colors are ‘safe’ to use in our app design.

- Green & Red
- Green & Brown
- Blue & Purple
- Green & Blue
- Light Green & Yellow
- Blue & Gray
- Green & Gray
- Green & Black

That being said, here’s a few color combinations to avoid because they’re a potential nightmare to color blind users.

- Use both colors and symbols
- Keep it minimal
- Avoid bad color combinations
- Use clearly contrasting colors & hues

The contrast ratio between the text and the background should be no less than 4.5:1 (a black and white image has a maximum contrast ratio of 21:1).

- Snook.ca – online tool (https://snook.ca)
- Stark – Sketch plugin (http://www.getstark.co)

You can use tools like … to check the app conformity with the requirement.

To recap

- Let's hope iOS 13 dark mode rumors are true
- Semantic colors + named colors = 
- Custom pipelines to generate and synchronize color palettes

2. Semantic colors when used with named colors help in designing a robust color system

3. The fact that named colors are a feature of Xcode Asset Catalogs means that custom pipelines to generate and synchronize color palettes are easy to build, which can accommodate a wide variety of team workflows.

As a community and as an ecosystem we have always strived to make beautiful apps and will continue to do that. Settle for nothing less than the whole package. Make your apps beautiful from interface to implementation.

References

- https://developer.apple.com/videos/play/wwdc2016/712/ Dynamic colors
- https://devblog.xero.com/managing-ui-colours-with-ios-11-asset-catalogs-16500ba48205 Named colors
- http://www.vsanthanam.com/writing/2017/7/6/colors-in-ios-ensuring-consistency-between-designs-interface-builder-and-uicolor


Thank you!