Designing Apps with Scroll Views _ Part II: Page Scrolling on Photos

Based on Apple WWDC 2010 Designing Apps with Scroll Views

Seyed Samad Gholamzadeh Jan 24, 2018

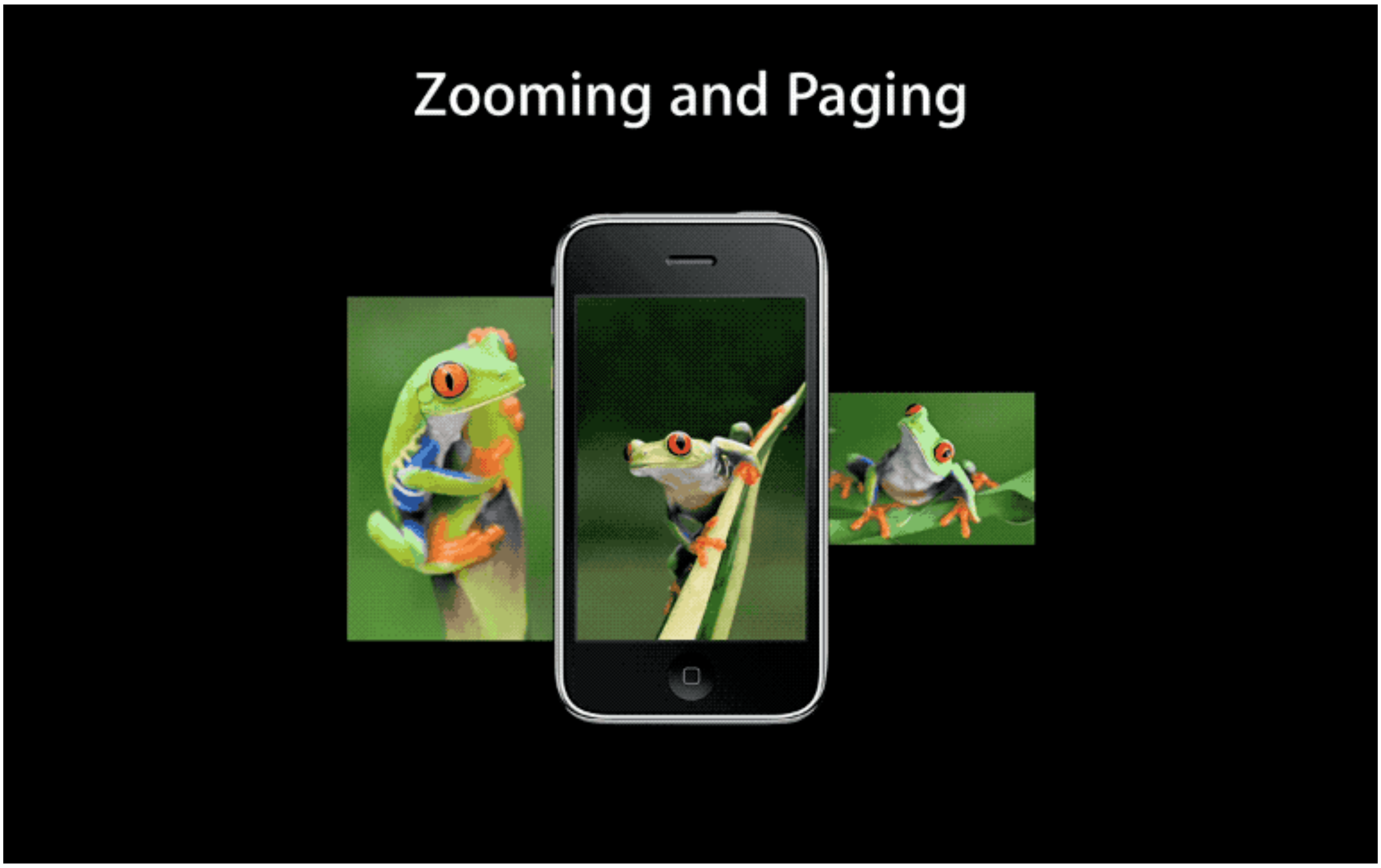
Introduction

This is the second part of a set of tutorials related to using scroll views in our app based on **Apple WWDC 2010 Designing Apps with Scroll Views** which you can watch it from here (HD|SD).
Here we want to make an app similar to iOS photos app with the help of `UIScrollView` useful features. You can study part I and part III of this tutorial from the links below:

In the previous part we learned how configure scroll view to be able to zoom and scroll on a photo. In this part we learning how to scroll between photos in order to behave like the photos application.

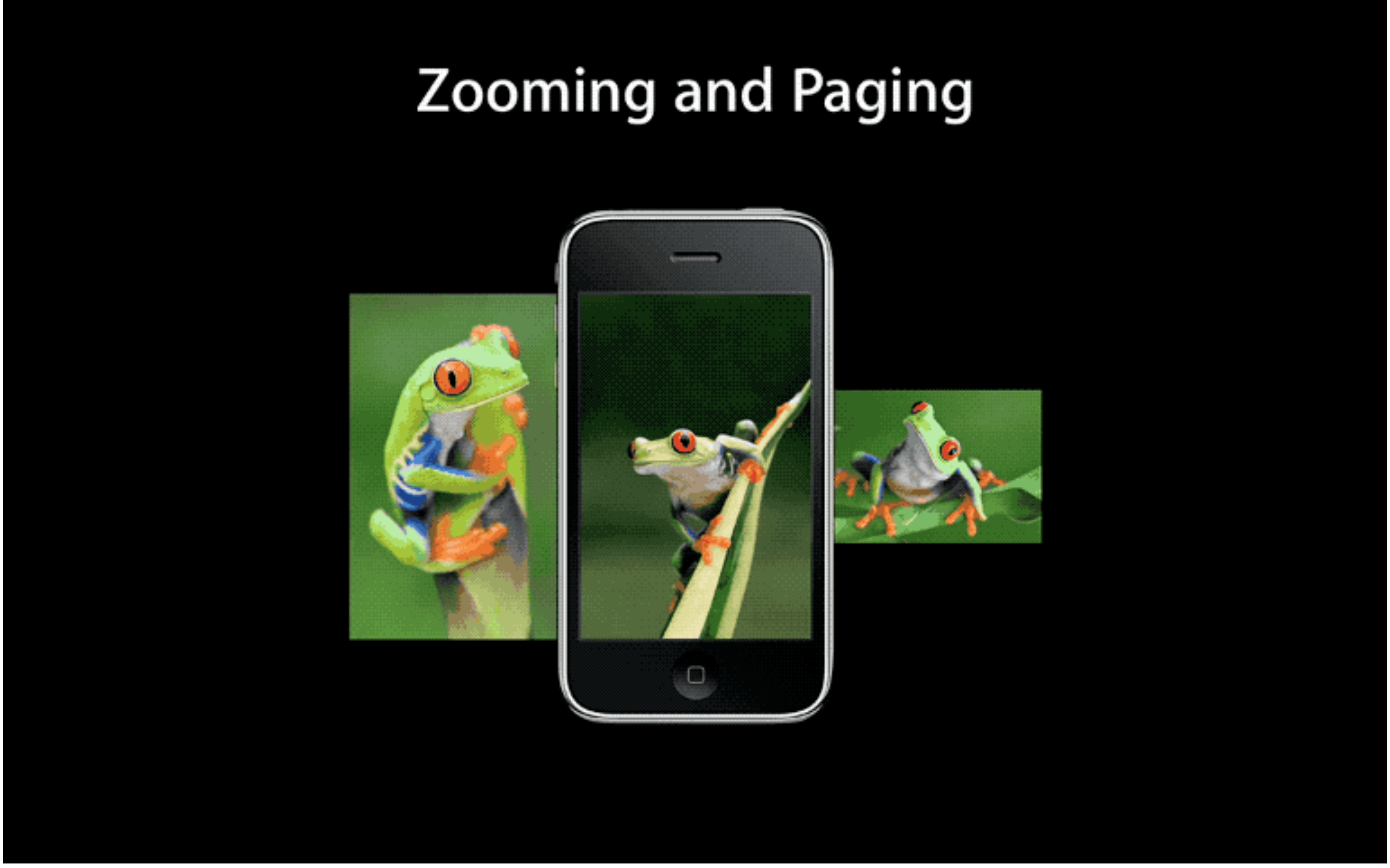
What We Want

So the question is what is that we want to get?
We want to users be able to swipe left and right to go between multiple photos, have it move over and swipe back to navigate around between their photos.



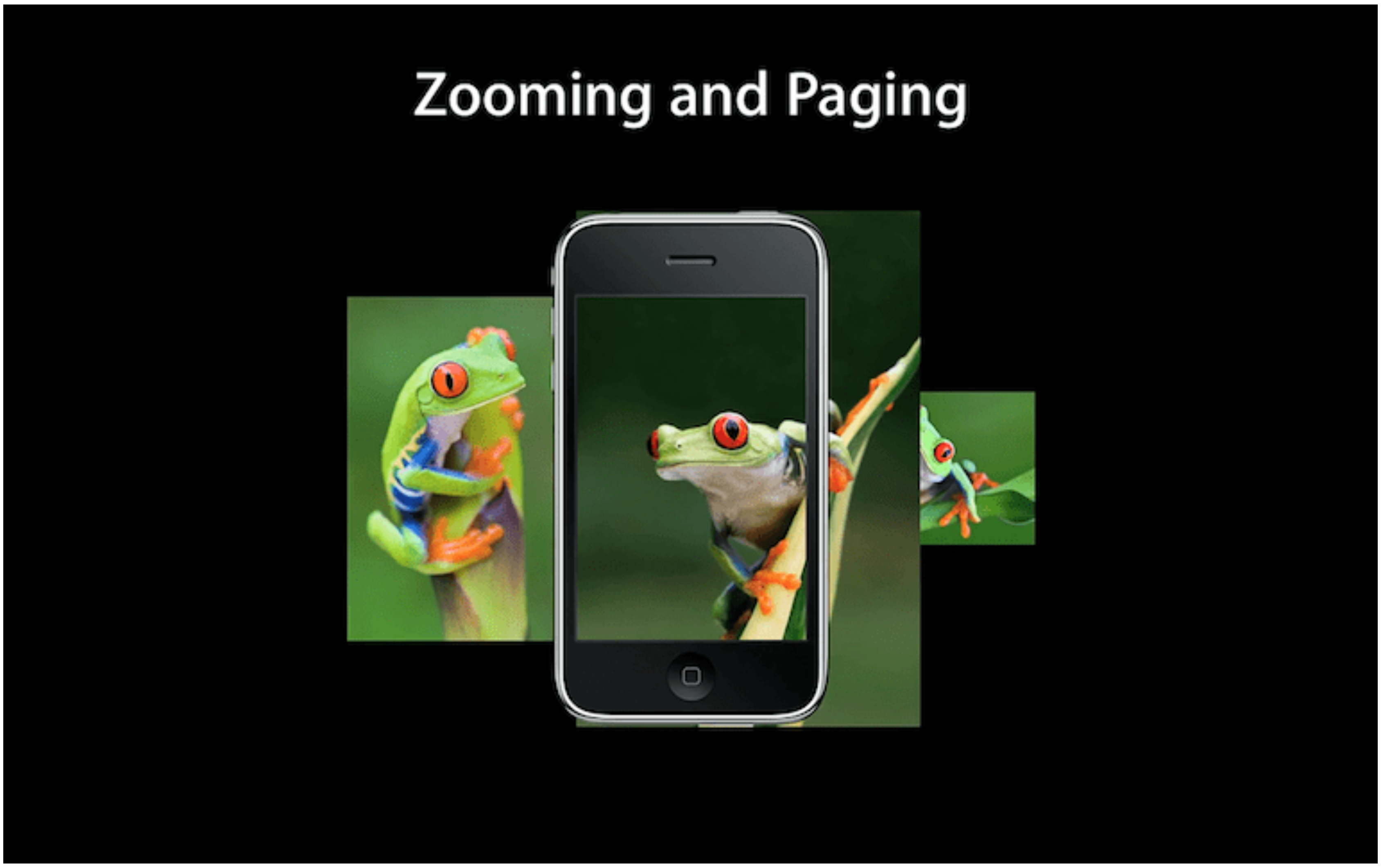
Users expect to be able to swipe left and right to go between multiple photos.

Also we want when they are at a photo, be able to zoom in on that photo by pinching or by double tapping, and once they're viewing it large, they can then swipe around on that photo to scroll around and view different parts of it, and when they get to the edge, they'll continue to pull the photo and it will swipe back to next one and shrink the one they zoomed in back down to its original size.



Users expect to be able to zoom and swipe between photos.

Now, you’ll notice something interesting happened here. When the photo zoomed in, the two photos to the left and right didn’t actually move to make room for the photo that just zoomed. They’re off screen and not visible anyway, so it doesn’t really matter where they are.



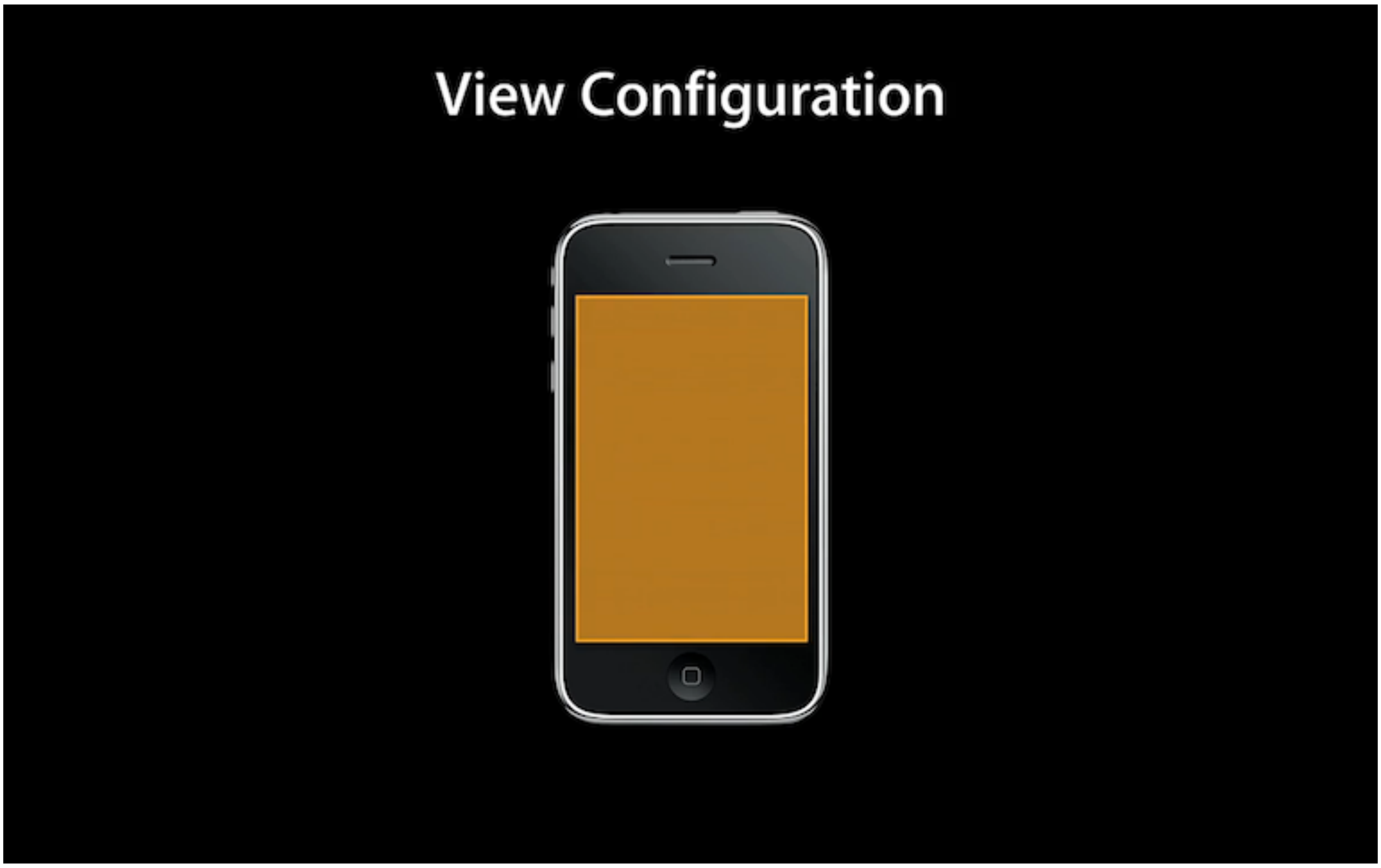
This is kind of a setup to give you an idea of how we're going to end up implementing this.

What We Need!

For implementing these features we don't have to start from scratch, we don't have to go to third party frameworks. We can use UIScrollView to get all these behaviors. Actually all is due to view hierarchy. We're going to accomplish and separate these features in two parts. We are going to look at paging, independent from zooming and consider them to be two different things.

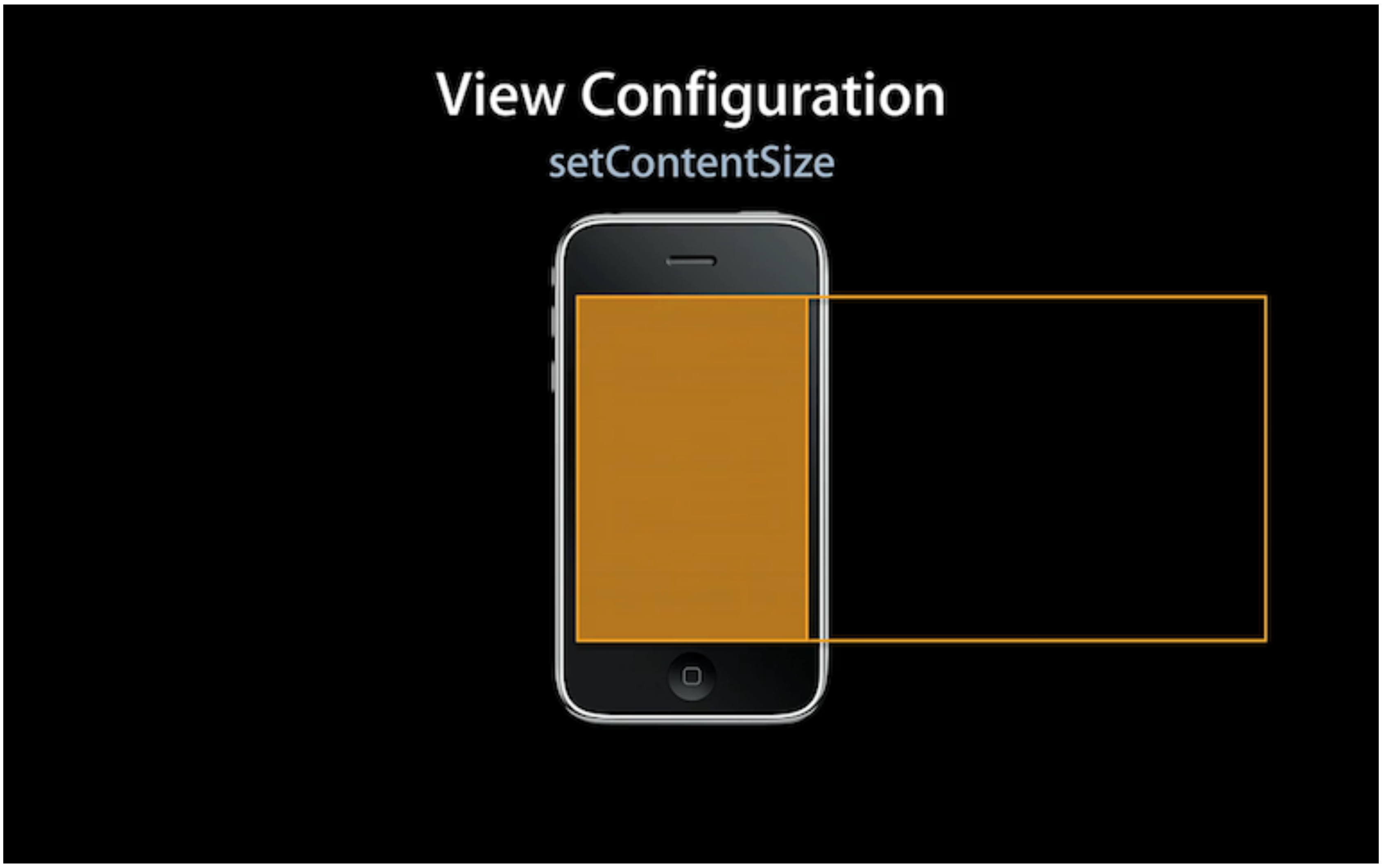
Let's start out with paging:

To implementing paging we use a paging UIScrollView. Paging UIScrollView is just a normal `UIScrollView` where you set the paging enabled bit to yes. And we'll create one that covers the entire iPhone screen. So If we assume that our iPhone screen bounds is 320 by 480 points, then `UIScrollView` is going to be 320 by 480 points too, so it fills the entire display.



The part that is orange, shows the `UIScrollView` that works as paging scrollView.

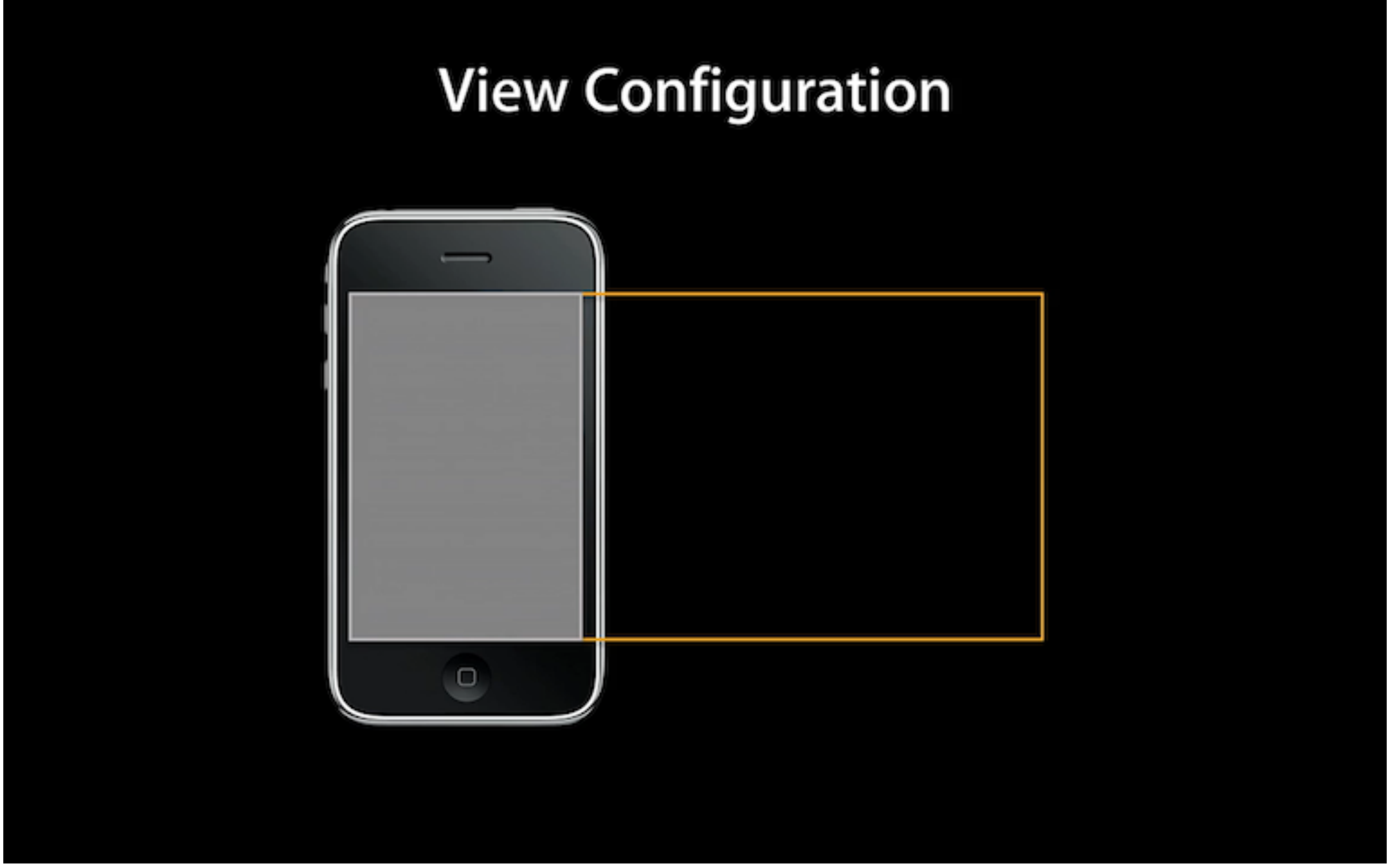
We will configure this `UIScrollView` to display three photos. In paging scroll view, the page width is determined simply by the bounds width of the scroll view. So in our case since our bounds width is 320 points, we're going to have page width of 320 points. So we'll multiply this by 3 and set our content size to 960 by 480, and it's means we can now swipe between three pages. That's all we actually have to do, to get the paging behavior that we're trying to add.



We set UIScrollView content size to 960 by 480 to be able swipe between three pages.

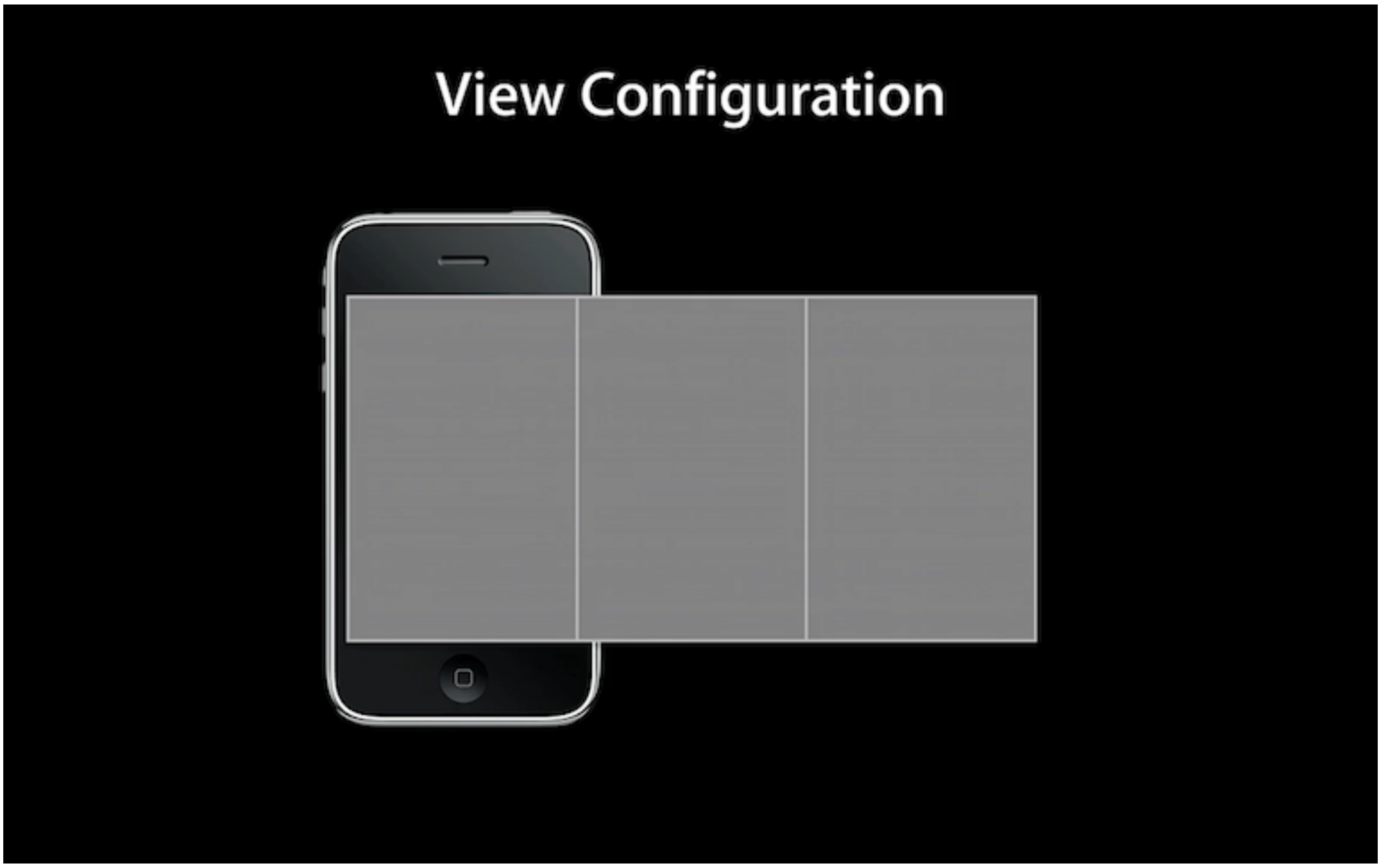
And for zooming:

For zooming, we're going to add separate UIScrollViews that handle just that zooming and panning on the zoomed images. This new `UIScrollView` is subview of our outer paging Scroll View that again covers the entire screen.



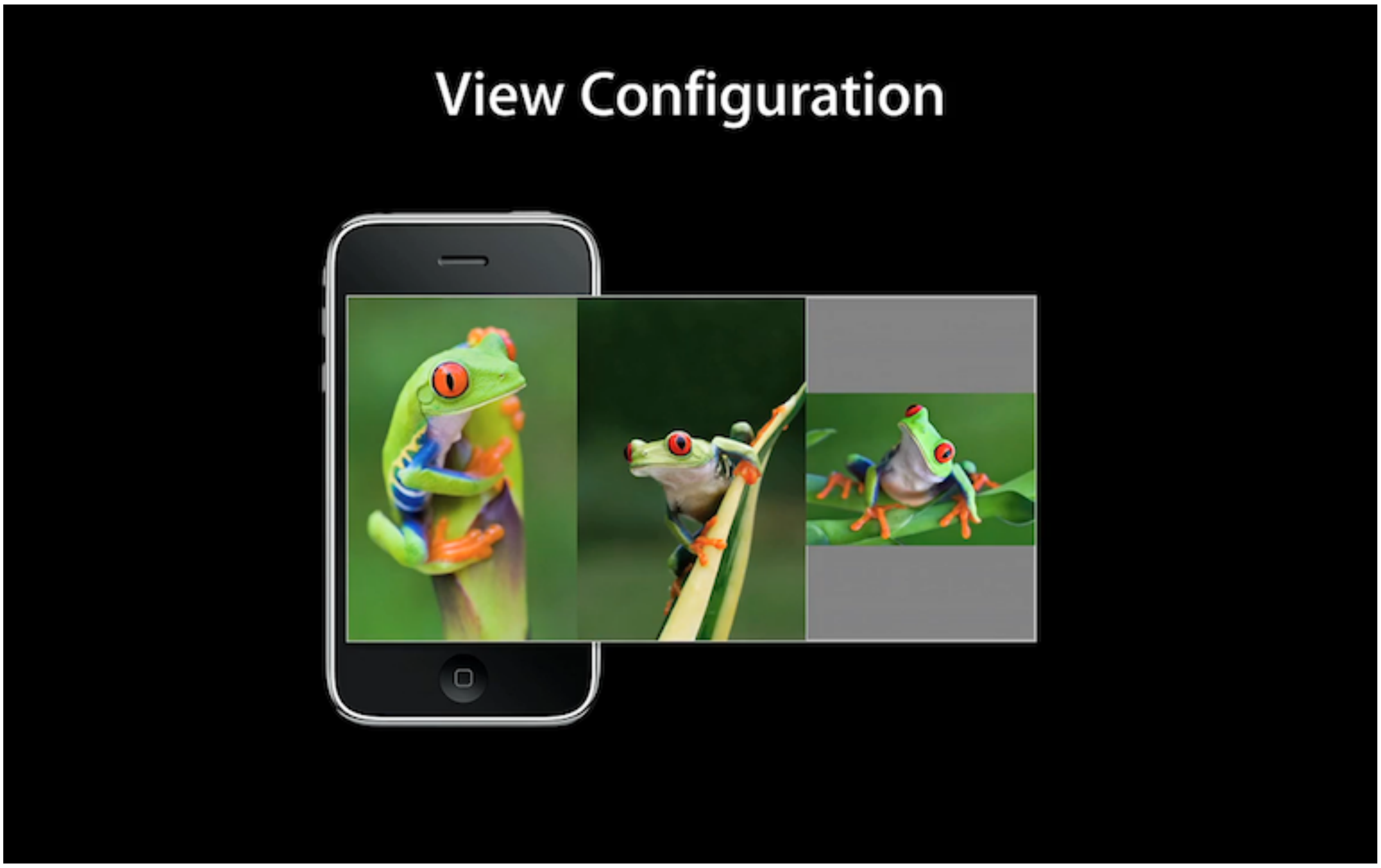
The part that is gray, shows The UIScrollView that handles just zooming and panning on the zoomed images.

Since we’ve got two other photos that we’re trying to display as well, we’ll add two more zooming scroll view to the right.



We add one zoom scroll view for each of photos.

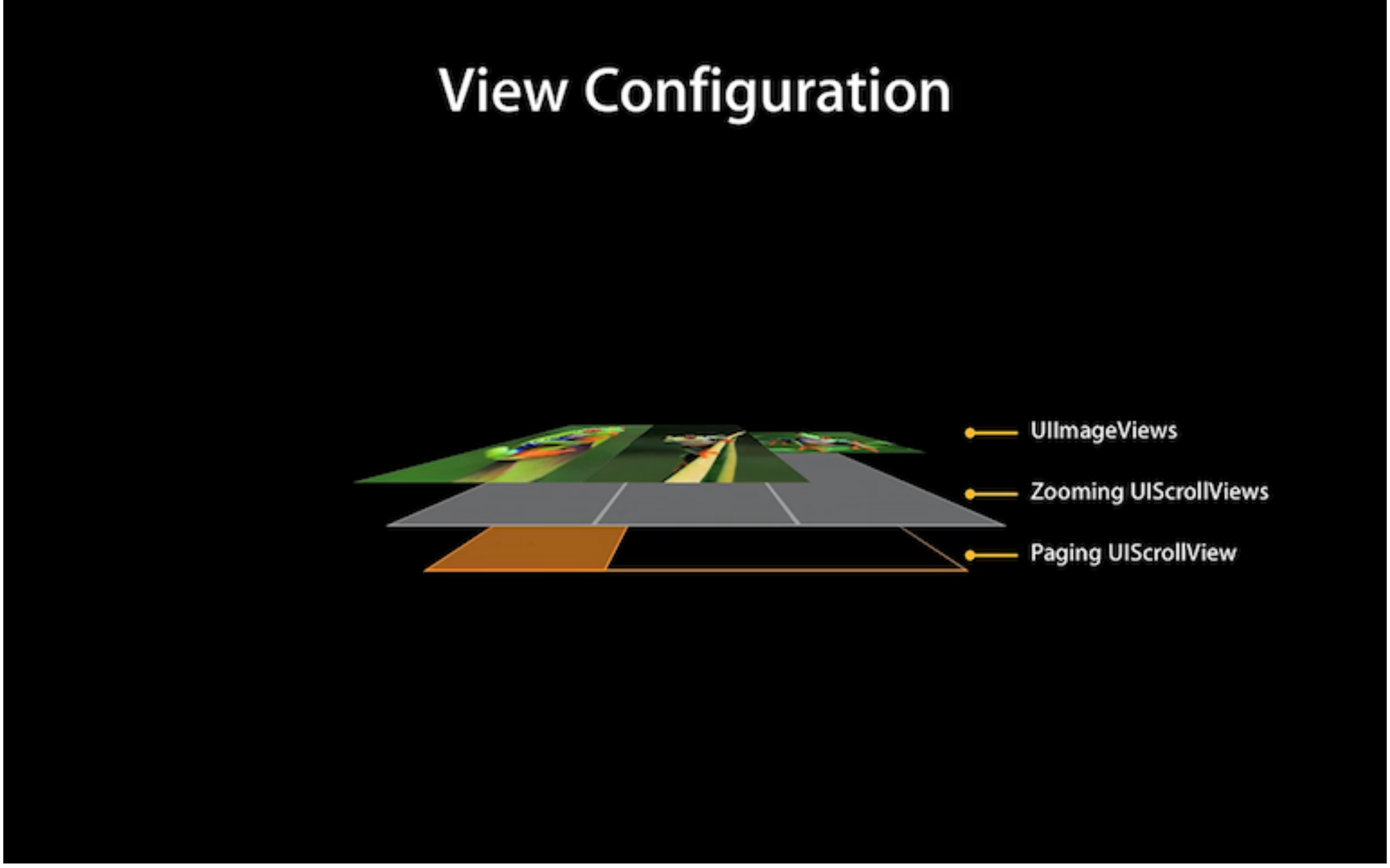
And finally, we need to actually display the photos. We add photos to zooming scroll views.



We add photos to zooming scroll views.

So in summary:

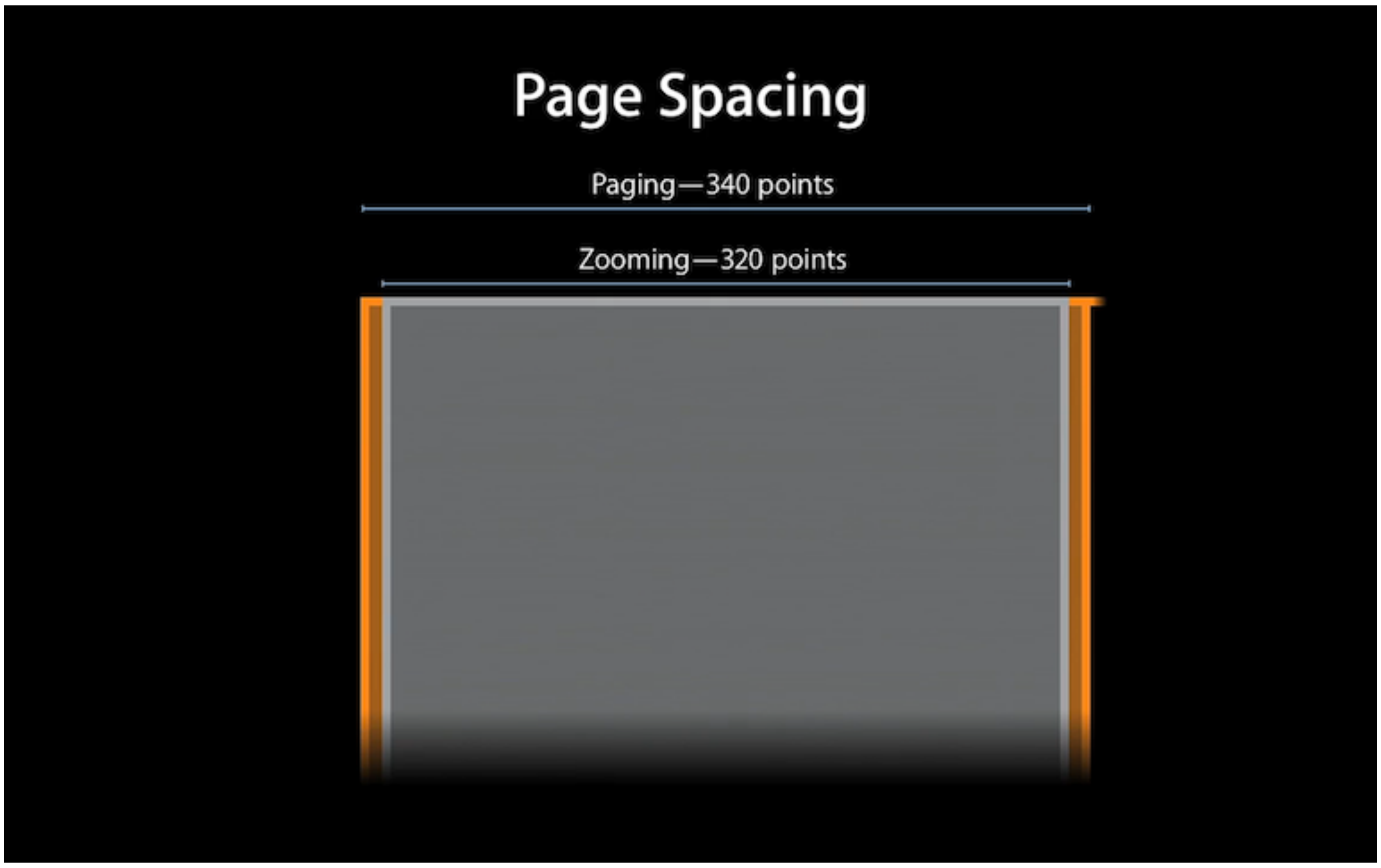
- We've got our outer paging scroll view at the bottom. That handles just the paging.
- We've got subviews to handle the zooming and those are also UIScrollViews.
- And then finally, we've got the UIImageViews subviews of the zooming scroll views that are actually displaying the photos.



Show of Layers from bottom

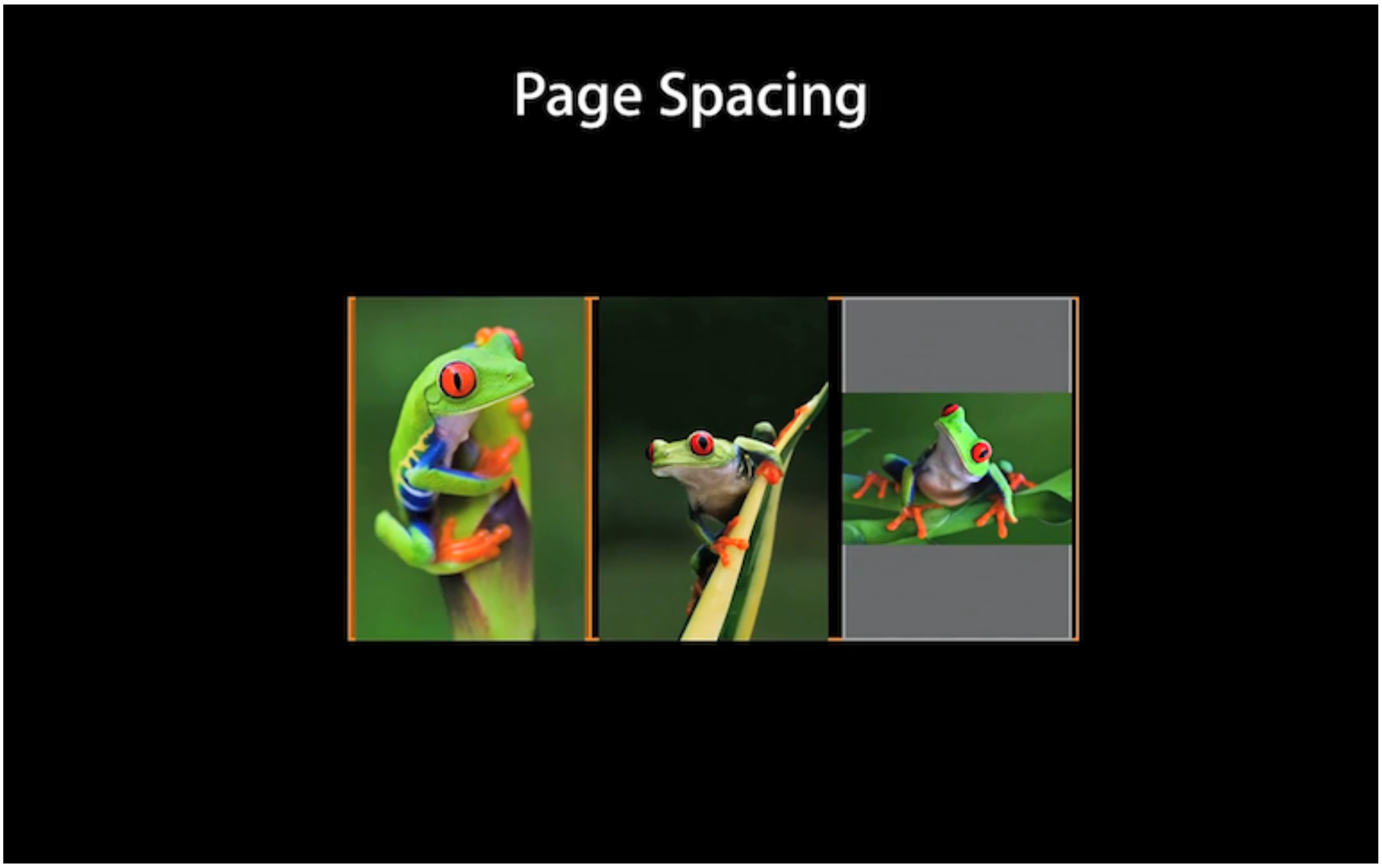
The combination of these three layers just like the above photo is all we actually have to do in order to get the exact same paging, zooming, scrolling behavior that you see in the photos app in iPhone. Just adding them as subviews takes care of everything.

One thing is missed!



Zooming scroll view is still 320 points, paging scroll view is now 340 points.

One thing that’s missing before we go and take a look at how we’ll actually implement these features in code is that the photos application, separates the photos a bit. There’s a bit of black padding between each one of the photos to make it very clear that there’s a border where one photo ends and where the next begins. In our pictures of frogs here, they’re all pretty green and as you’re scrolling between them it’s not entirely obvious where one ends and the next begins. So in order to make them stand apart a bit, we have to increase the page width so that there’s more space between each page. But as just mentioned before, the page width of a scroll view is determined by that scroll view’s bounds width. So in order to get the page width bigger, we actually have to increase the size of that outer paging scroll view. If we change the paging scroll view bounds from 320 points up to 340 points and keep it centered on screen, there’ll be 10 points hanging off on the left and 10 points on the right but they’re off the side of the screen and there’s not going to be displayed there anyway.



We add 10 points padding between photos.

It’s time to write codes!

Download starter sample code from here. Open and Take a look at it. There is no vague thing about it.

- You see an **ImageScrollView.swift** file that is exactly the same as that we created in previous part (If you didn’t read the previous part, follow this link).
- You see a **Full Images** folder that contains some photos and a property list (**ImageData.plist**) that contains some info about our photo like their name and size.
- You see **PhotoViewController.swift** file that is a simple `viewController`. There is some codes at the bottom of this file that marked as **Image Fetching tools**. These codes used for fetching photosin order of their numbers on **ImageData.plist**. They’re not complicated, you can understand them yourself.

Go to **PhotoViewController.swift**. At the top of the file after `class PhotoViewController: UIViewController {` line add these two lines of code:

In the first line, we defined a constant that is equal to `10` points. we use this for page padding length of our paging scroll view.
In the second line we defined a `UIScrollView` instance that is going to be our paging scroll view instance.

Now we want to create the Paging Scroll View that we just declared. But first we need to figure out what its frame is going to be. Add blew method after `override func viewDidLoad() {…}` method:

This method gets the frame for paging scroll view. We started with screen bounds. But as we explained, we’re going to want to have that paging scroll view hang off the sides of the screen by 10 points, so that we need leave some space on either side. For this we subtracted 10 points from `frame` **X** origin and added 20 to its width (*Note that we have used `pagePadding` constant instead of explicitly writing 10*). And the

effect of that is it will now hang off
10 points on the left and 10 points on the right.

So let's create the paging scroll view with this frame. In to the `viewDidLoad()` method after `super.viewDidLoad()` line add these lines of code:

It's time to set a few properties now. Add these lines after above lines:

We set background color of both PhotoViewController `view` and `pagingScrollView` to black. We turned off showing scroll indicators, and finally we set pagingScrollView `isPagingEnabled` property true, to enable paging for `pagingScrollView` as we explained before. One thing remains, defining `contentSize` of `pagingScrollView`. The content size is the property that determines the scrollable area. We want to make it wide enough to accommodate all of the pages that we're going to insert in `pagingScrollView` later. Add blew method after `frameForPagingScrollView() -> CGRect {…}` method:

This is how we calculate content size of `pagingScrollView`. Add this line in to the `viewDidLoad()` method and after the `self.pagingScrollView.isPagingEnabled = true` line:

The width of this content size is going to be the size of the width of a page times the number of images that we have and we got a property here, image count, which just returns the number of images that we're going to display. And the height will just be the height of the frame because we're not going to allow for scrolling in the vertical direction.

And finally we add `pagingScrollView` as subview of **PhotoViewController** `view` by adding this line after above line:

> ***Note:*** *You should not define* `pagingScrollView` *as* ***PhotoViewController*** `view` *by writing :* `self.view = self.pagingScrollView`*. Because* ***PhotoViewController*** `view`*'s frame*

> *being reset after `viewDidLoad()` method being called, and so all our configuration for `pagingScrollView`'s frame will be lost. So you must reconfigure the view frame in `viewWillAppear(_ animated: Bool)`, which it’s not good idea to repeat some codes in programming when you could avoid that.*

The next thing we need to do is add some pages. We go through all the images we have and for each image we’re going to make a page and insert it into the scroll view. Add these codes after `self.view.addSubview(self.pagingScrollView)` in to `viewDidLoad()` method:

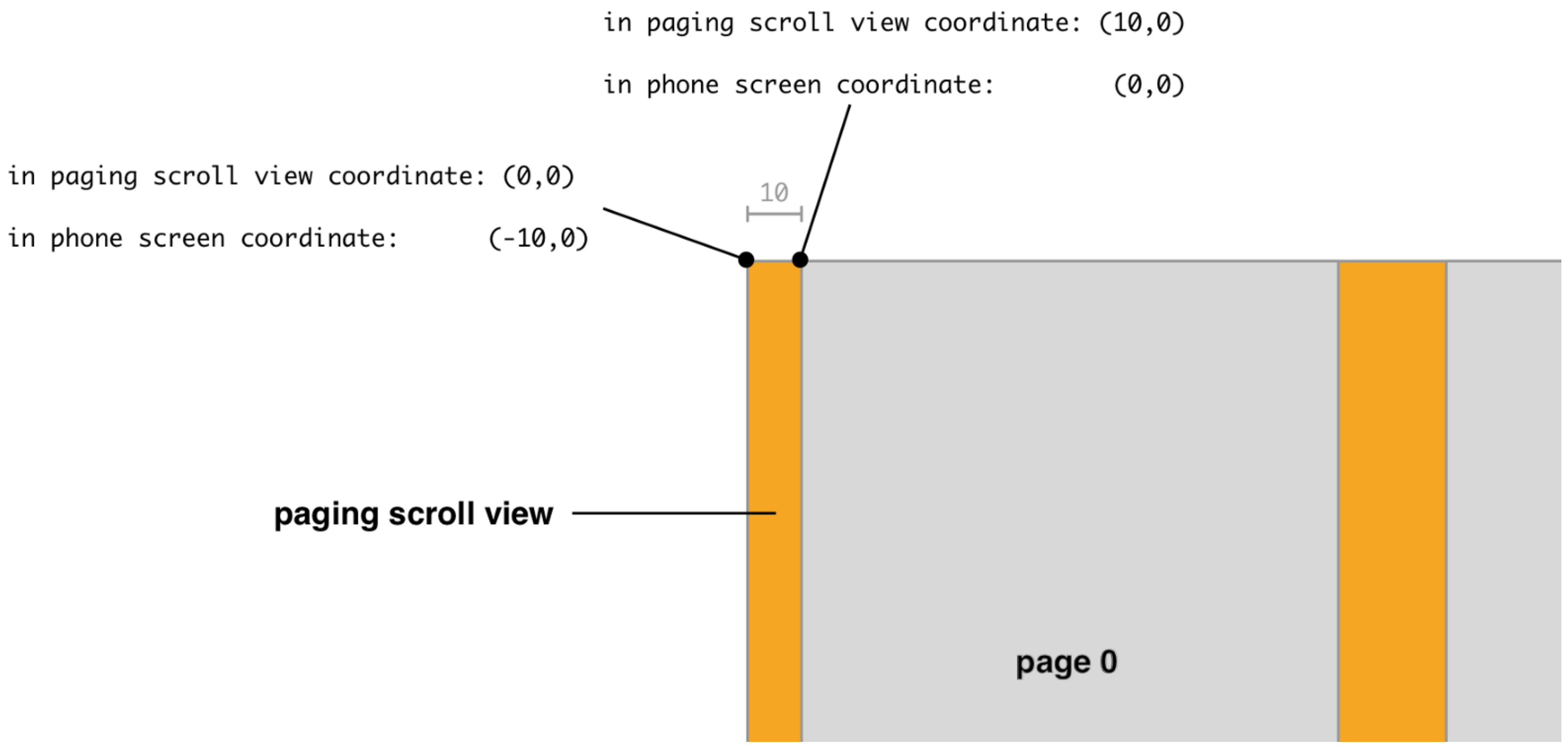
First we just iterated through our images and for each image, we created a zooming scroll view. You are familiar with `ImageScrollView` subclass of `UIScrollView` from previous part of this tutorial (and if you’re not, You can read it here), But in summary what this scroll view does is it sets up the zooming for us. Next we configured this page for the particular index that we’re at and that’s just going to set the frame of the page appropriately. Finally, we added that page as a subview of our paging scroll view.

Let’s see how is `configure(page, for: index)` method. Add this method after `viewDidLoad() {…}` method:

First we calculate and set frame of page for appropriate index and next we ask it to display image for input index.

If you are curious about how be calculated `frame` for `page` here is `frameForPage(at: index)`. Add it after `frameForPagingScrollView() -> CGRect {…}` method:

The calculation is simple, but a question that might come up to you is why we Why do we add `pagePadding` to frame of page and why not just make it equal to `(bounds.size.width*CGFloat(index))`? For example by this calculation `x` origin of image at first index which is `0` will be `10`, but we expect it to be zero? The answer is we calculate the frame of page, in coordinate of paging scroll view content which we took it `10` points back. So the zero value of x in paging scroll view content is equal to `-10` in coordinate of phone screen and therefore if we set first page `x` value to `0`, it actually would be `-10` in phone screen coordinate. So we add it `10` points and bring it to `0`.


in paging scroll view coordinate: (10,0)
in phone screen coordinate: (0,0)
in paging scroll view coordinate: (0,0)
in phone screen coordinate: (-10,0)
10
paging scroll view
page 0

There is 10 points difference between paging scroll view content origin x and phone screen origin x

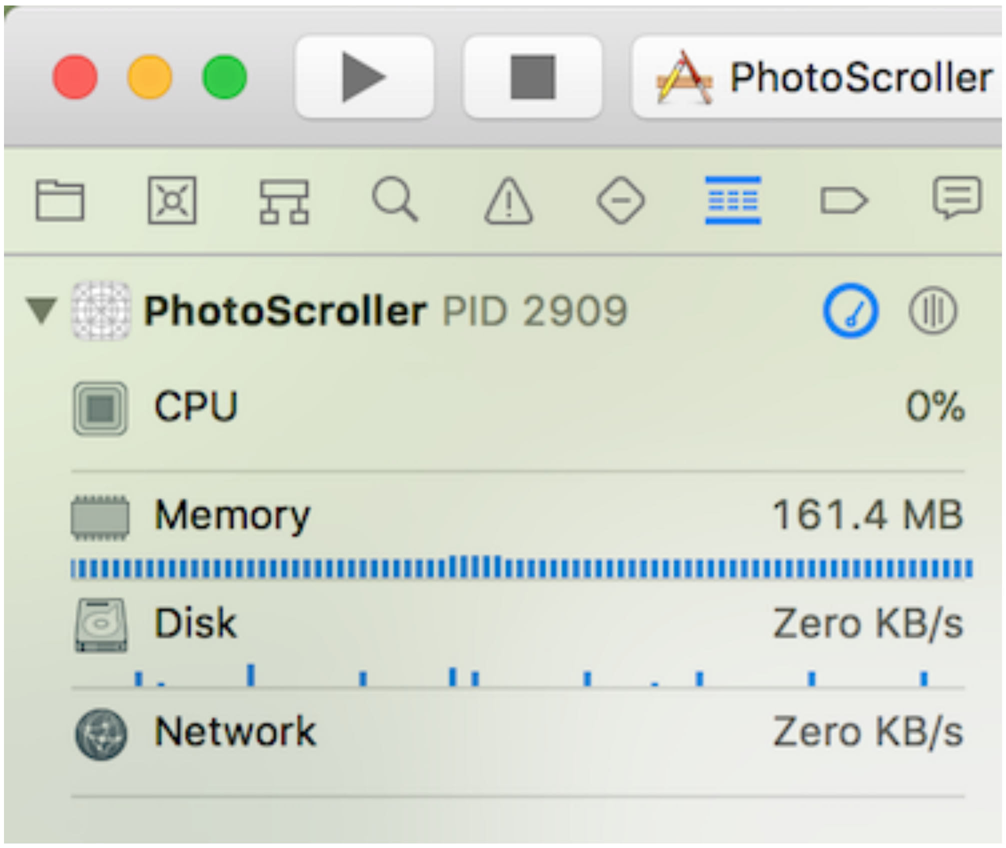
Build and run the app. You should see something like this:

You should be able to zoom in photos and swipe between them

Maybe you’ve tried that zoom in a photo and then swipe to next photo, and swipe back to the previous photo again to see if it has returned to its original size. If you have not tried this, I’ll tell you the answer: No it hasn’t!

The photo does not return to its original size after swiping back

But there’s another thing you may not have noticed. There is one big problem with this application. Run the application again with Xcode and while the application is running go to Xcode’s **Debug Navigator.**



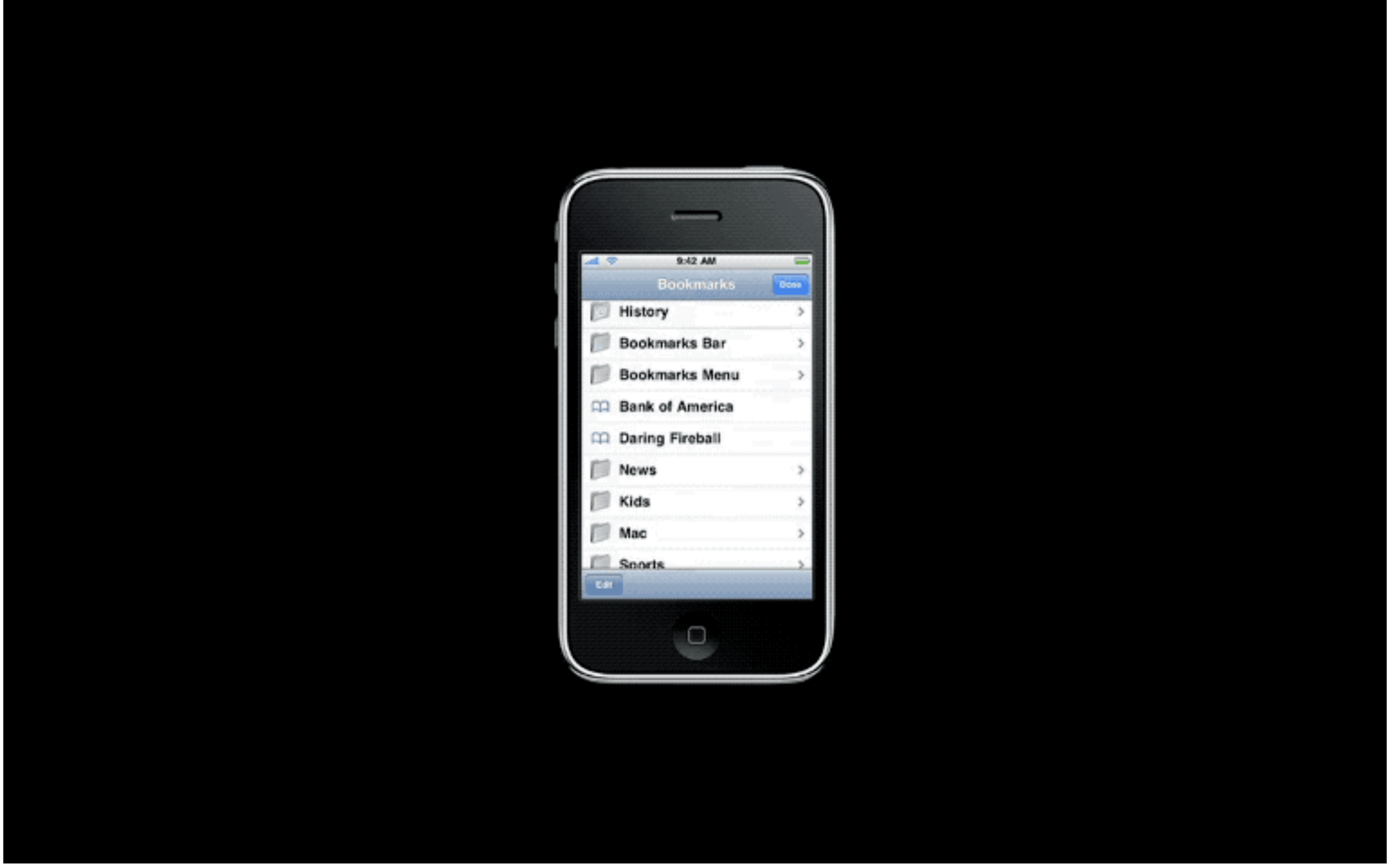
Xcode’s Debug Navigator

You should see a number like the above picture in front of **Memory**. For

the above picture this number is **161.4 MB**. This number shows that our app is using 161.4 MB of memory. But Why?
The reason of this number is that the images we used in our application are pretty large. They are 1 to 1.5 megabytes compressed and the last image is about 8 megabytes which translates to somewhere about 20 megabytes uncompressed (for the last image about 100 megabytes) and what we did here was we added every one of this images to our paging scroll view, so we have them all open in memory at once. Here we just have 4 images at all and they used 161.4 MB of memory, Imagine how much it will be for 100 or 200 or even 1000 photos? It will be a huge number and an iPhone doesn't have this much memory at all and you would crash before you even started if you were to do it this way on the actual device. Regardless of that, It is not acceptable for a simple photo app to use such volume of memory. So we have to reduce it. The solution to the problem is this, why we add 4 photo to memory while we actually show just one of them on screen? Why do not we just load the photo that we want to show? You may ask how do we do this? The answer is **Subview tiling.**

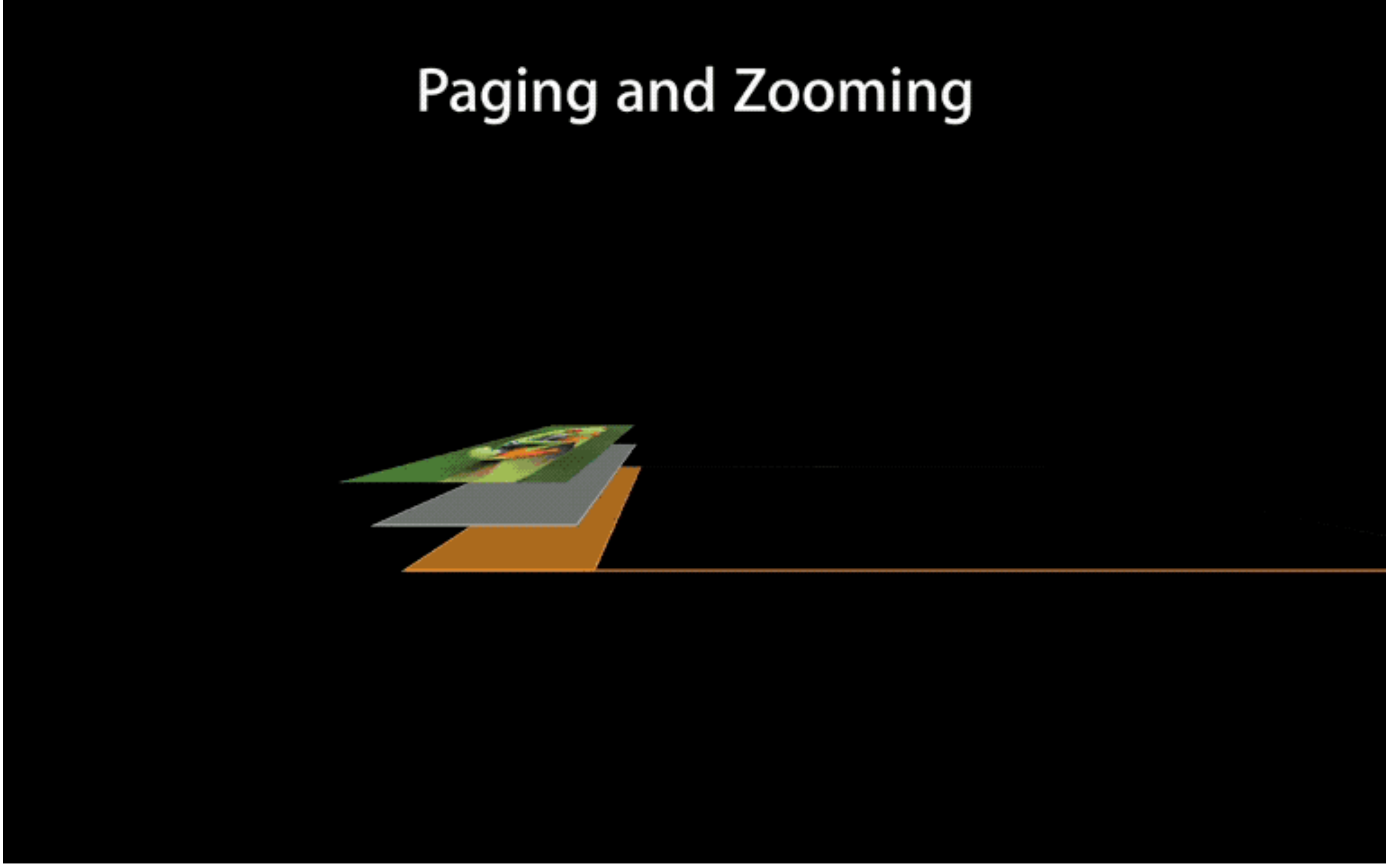
Subview Tiling

If you've used `UITableView`, you've already seen subview tiling in one way. When you implement your cell for rowAtIndexPath method in `UITableView`, the first thing that you try and do is dequeue a reusable cell with an identifier, and what's that doing is basically implementing subview tiling for you on your behalf. So, as your user scrolls through their table view, cells move off the top, you dequeue them and put new content in and they scroll in on the bottom and this happens repeatedly.



Subview tiling in tableView

We’d really like to do basically the exact same thing for our photos app. As the user is paging horizontally on a photo, and the photo moves off screen, we no longer need that scroll view to display it when it’s not visible. We can reuse it and move it in to display another photo on the right. So that’s exactly what we’re going to do. When the app runs we only have one zooming scroll view that’s visible on screen and so we only have to load one photo. As the user pages through our photos, at any point there’s only going to be a maximum of two different photos visible at any given time and there is only ever two scroll views created to show those photos. So it’s going to be much less memory and much less set up cost initially in order to even begin launching and displaying these things.



At any point there's only maximum of two different photos visible at any given time and there is only ever two scroll views created to show those photos.

Where do we want to do this?

We could implement `scrollViewDidScroll` delegate method. Basically, every time that the user scrolls any amount through the scroll view, either by dragging their finger or by flicking or having it decelerate, `scrollViewDidScroll` will be called for every frame before that frame is actually drawn on screen. So you have a chance to add subviews if you're going to need to display more content before that empty spot even becomes visible and that's exactly what we'll do. For implementing this feature to our app, we need to add a couple of instance variable to `PhotoViewController`. Go to **PhotoViewController.swift** file in Xcode Project Navigator and add these two lines at the top of `PhotoViewController` class after `var pagingScrollView: UIScrollView` line:

Our strategy is to keep track of what tiles are currently visible, and we're also going to keep track of tiles that are already used and pull out them, because they've gone off screen. So we added a **recycled pages set** and a **visible pages set** for these two purposes. Now we're going to need a method that accomplishes the tiling, but what does it look like to tile the pages? Add this method after `viewDidLoad() {...}` method :

1. The first thing that we need to do when we're tiling the pages is to calculate which pages should be visible given the current content offset of our scroll view. So for that purpose, what we're going to do is grab the visible bounds of the scroll view and once we have the **visible bounds**, you can think of that as a rectangle of the content and we're going to take a look at that rectangle as a bunch of columns of pixels. So we'll look at the first column of pixels and we'll see which page is that column of pixels associated with. Which page is that column of pixels in? And that's going to be the first page that we need to display and then we'll look at the last column of pixels and we'll find the page in which, that column is on and that's going to be our range of pages. This is the strategy and what these lines of code are doing, they're calculating the **first needed page index** and the **last needed page index**.
2. So now that we know which pages we need, let's first recycle the ones that we don't need but that we already have in our view. For that, we're going to use `visiblePages` set that we have. And we're going to look at each
   of the pages that's currently in the visible pages set and find out is it needed or not. Now, we'll take advantage of the fact that we have a custom subclass of `UIScrollView` that my pages are `ImageScrollView`. And we want teach them to know what index they are and what page index they represent. For this just add `var index: Int!` at the top of `ImageScrollView` subclass after `class`

`ImageScrollView: UIScrollView, UIScrollViewDelegate {` line. What we're going to do is use that here in `tilePages()` method. We'll just ask the page that we're on, is your index outside of our needed range? Is it less than the first or greater than the last needed page? So if it is outside of the needed range, we're going to recycle it by adding it to our `recycledPages`. We're going to remove it from the Super View and we also want to now take note of the fact that it's no longer in our visible pages set so we want to remove it from the `visiblePages` set. We could do this by just adding `self.visiblePages.remove(page)` after `page.removeFromSuperview()` , **but don't do that!** Because adding this line of code here would be mutating a set while we're enumerating it and that's a really bad idea. So we take that out and after the for loops safely, when we're finished enumerating, we take advantage of the fact that `visiblePages` and `recycledPages` are sets and we can do a set subtraction and just remove all those recycled pages from the visible pages and that's a safer way to handle that problem.

3. Now that we've recycled our pages, we need to add the ones that are needed, and aren't already in the view. So for that, we're iterate through from the first to the last needed page and ask do we actually have a page that's indexed already? And we made a convenience method here, `isDisplayingPageForIndex` that does it. What that does is it actually just looks through the visible pages and sees whether there is one at that index. So if there's not, and if we're missing this page that we need, we try to get a page by calling `dequeueRecycledPage()` method. It means we check if there is any page in `recycledPages` set, we grab it and remove it from `recycledPages` set and use it and only if we fail to get one, and if there wasn't one available, we create a new one, we make the page. This is for that we don't want to every time that we discover, we need a new page, creating one from scratch(and basically this is why we created `tilePages()`!). Then we need to configure page for the right index, we need to tell it, hey this is your new index,

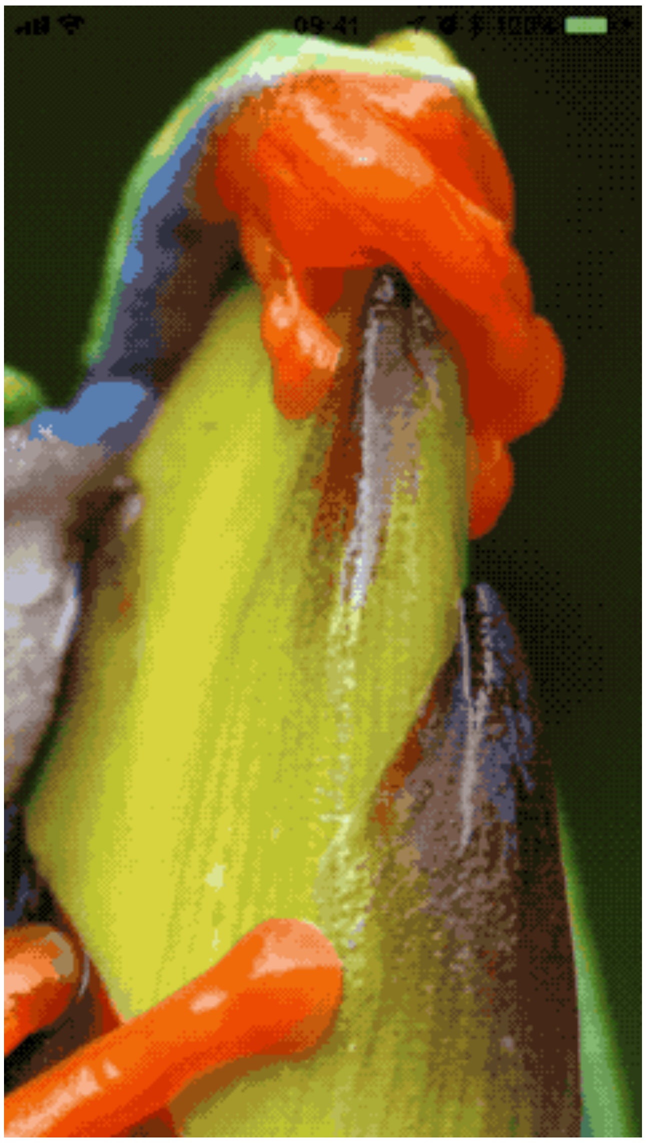
remember it! So add `page.index = index` in to `configure(_ page: ImageScrollView, for index: Int)` before `page.frame = self.frameForPage(at: index)`. Next we add page as a subview of our scroll view just like we did before, and finally, we'll note that this page is now in our visible set.

Now that we have `tilePages()` method go to `viewDidLoad()` method and delete these lines of code:

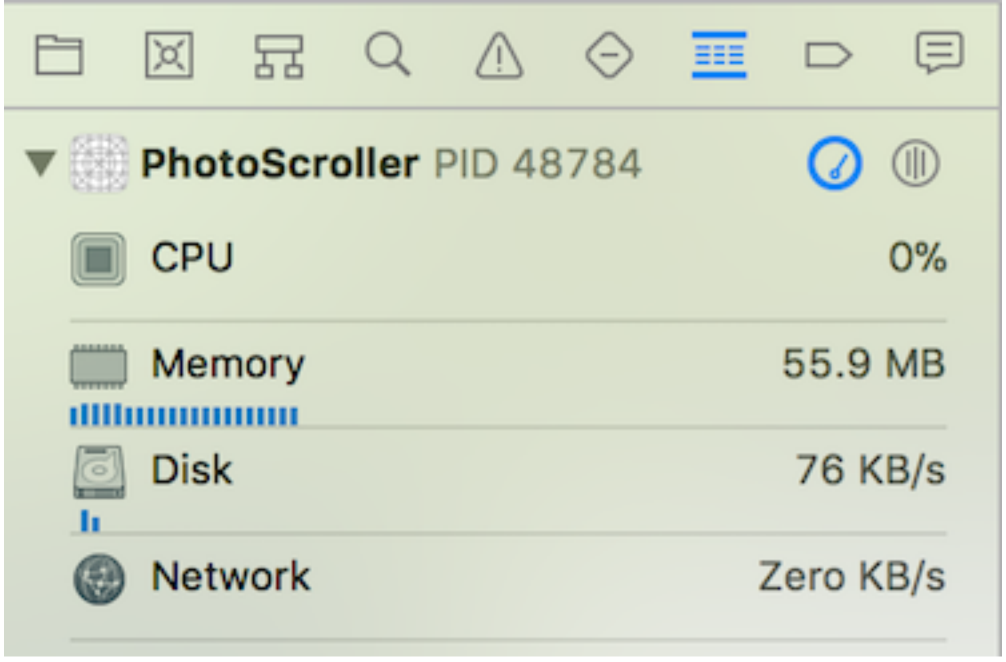
Because that's where was using up all of our memory. Instead add this line of code in their place:

We just call tile pages once to get the tiling started and that will have the effect of showing the first page since that's the page that you start on when this view is loaded. But as we pointed out it's not enough to tile the pages once. We need to tile them every time that the scroll view scrolls. So for that purpose, we going to implement `scrollViewDidScroll` delegate method. First we need to set our view controller as the delegate of the paging scroll view. Add `self.pagingScrollView.delegate = self` in to the `viewDidLoad()` method after `self.pagingScrollView.contentSize = self.contentSizeForPagingScrollView()` and before `self.view.addSubview(self.pagingScrollView)` line. Now for implementing `scrollViewDidScroll` add These lines after `configure(_ page: ImageScrollView, for index: Int) {…}` method:

All we’ll do here is call tile pages again every time the scroll view scrolls. Alright, build this version and see what happens. It looks the same, except if you zoom in a photo and swipe to next page and swipe back again you’ll see the photo is returned to its original size.


The photo does return to its original size after swiping back

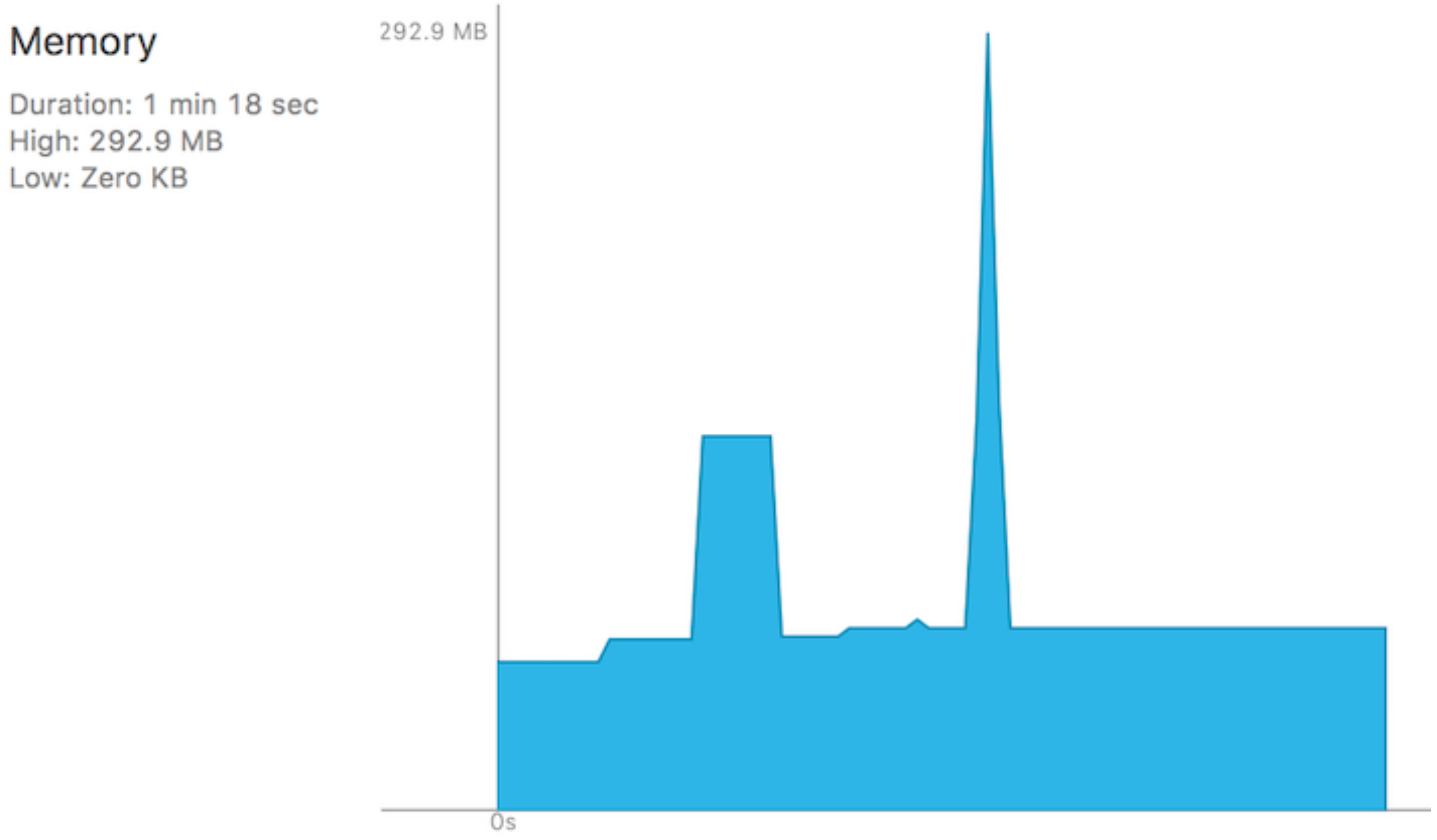
So, things seem OK. Let’s take a look at the **Debug Navigator** and see how much memory we’re using this time. While application is running go to Xcode’s Debug Navigator, and you will see something like this:


PhotoScroller PID 48784

CPU	0%
Memory	55.9 MB
Disk	76 KB/s
Network	Zero KB/s

Xcode's Debug Navigator

Now Memory is used about 56 MB. So we've cut our memory consumption
down by a significant amount. Enough that you could probably actually run
this version of this application on the device. But let me just signal a problem.
While you are in **Xcode's Debug Navigator**, click on **Memory** and try to swipe between photos. You will see something look like this in the middle of Xcode's window:



The meaning of this chart is, when you swipe between photos, the application tries to load and decompress that huge image at the beginning of the scroll, as you're getting to next page, and that causes a perceptible lag before the next page appears. So you actually get a bad performance problem that is perceptible. Could we solve this problem? I do not want to enter this topic now, but the short answer is **Yes!** If you are looking for the a long answer, we will discuss this in the next part of this tutorial. For now, we have a few things to do.

The LandScape!

We need to configure our app for landscape. First of all, add this method in to the `PhotoViewController` class before `//MARK: — Image Fetching tools` line:

we fixed edge of `pagingScrollView` to PhotoViewController `view`. Next add `self.layoutPagingScrollView()` in to the `viewDidLoad()` method, after `self.view.addSubview(self.pagingScrollView)` line to add these constraints to `pagingScrollView`. You can build and run the app now but I should tell you, it's not configured yet for landscape. Because we must reconfigure the scroll views content size at rotation time. For this, we need to save the index of page that is visible now, and reset content size of `pagingScrollView` at the rotation, also we need to recalculate `pagingScrollView` page bounds with the new size after rotation. But before adding codes for configuring rotation state, replace `frameForPagingScrollView()` whole method with blew method:

this method works like previous method, but also enabling us to calculate frame of `pagingScrollView` for a new size. The `size` input parameter in this method is an optional parameter, which at default is `nil`.

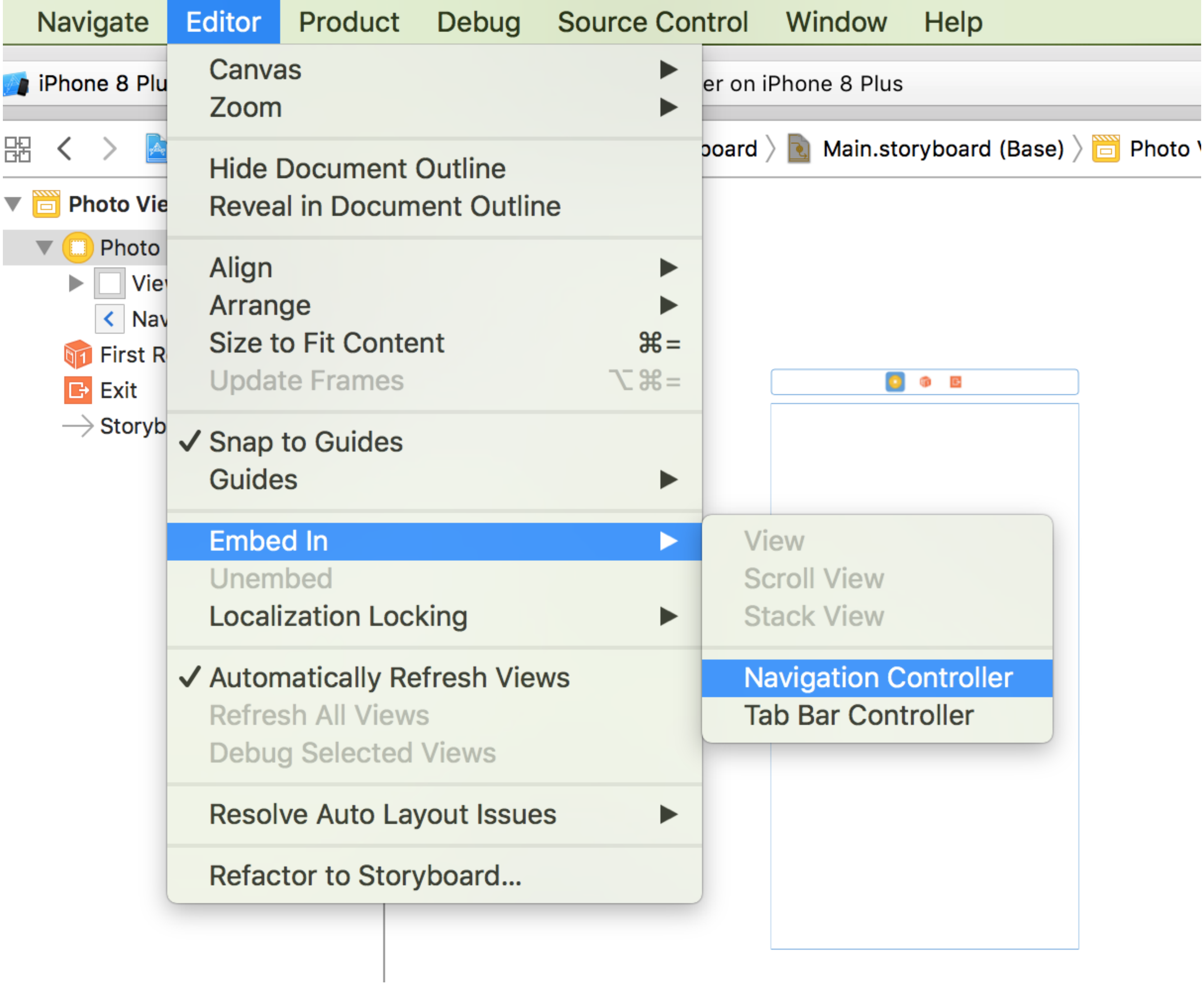
Now add `var firstVisiblePageIndexBeforeRotation: Int!` instance variable after `var visiblePages: Set<ImageScrollView>!` line, we need this variable for saving page index before rotation. Next add these lines of code after `frameForPage(at index: Int) -> CGRect {…}` method:

The methods `willTransition(…)` and `viewWillTransition(to size: CGSize, …)` methods, both are called at the rotation of view. We use first method for saving current visible page index into the saveCurrentStatesForRotation() method and the second for calling `restoreStatesForRotation(in size: CGSize)` method for recalculate pagingScrollView `bounds` and `contentOffset` with the new size, given by this method. There is nothing new here. Most of the codes are explained before. For more information, see the previous part of this tutorial. Build and run, you will see a stable application at the portrait and landscape.

Embed in Navigation Controller

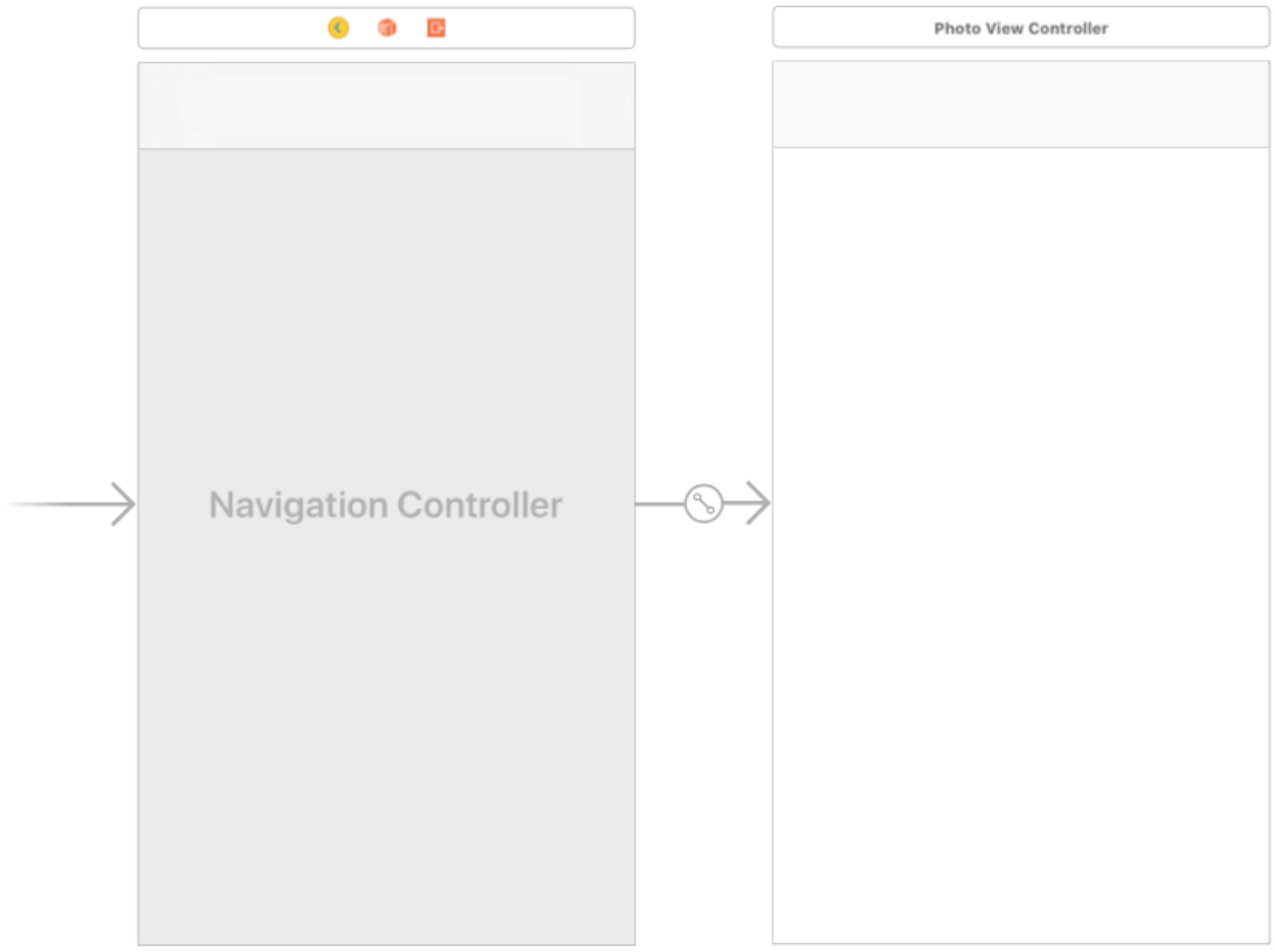
Some times your `PhotoViewController` may be embedded in to a **Navigation Controller**. You should to know how to adapt your `PhotoViewController` to these conditions.

Let's first add a Navigation Controller to our `PhotoViewController` and see what happens. Go to **Main.storyboard** file in **Xcode's Project Navigator**, and select **Photo View Controller** (Photo View Controller arounds lines being blue, when it's selected). As **Photo View Controller** is selected, in menu go to **Editor → Embed In**, and select **Navigation Controller.**



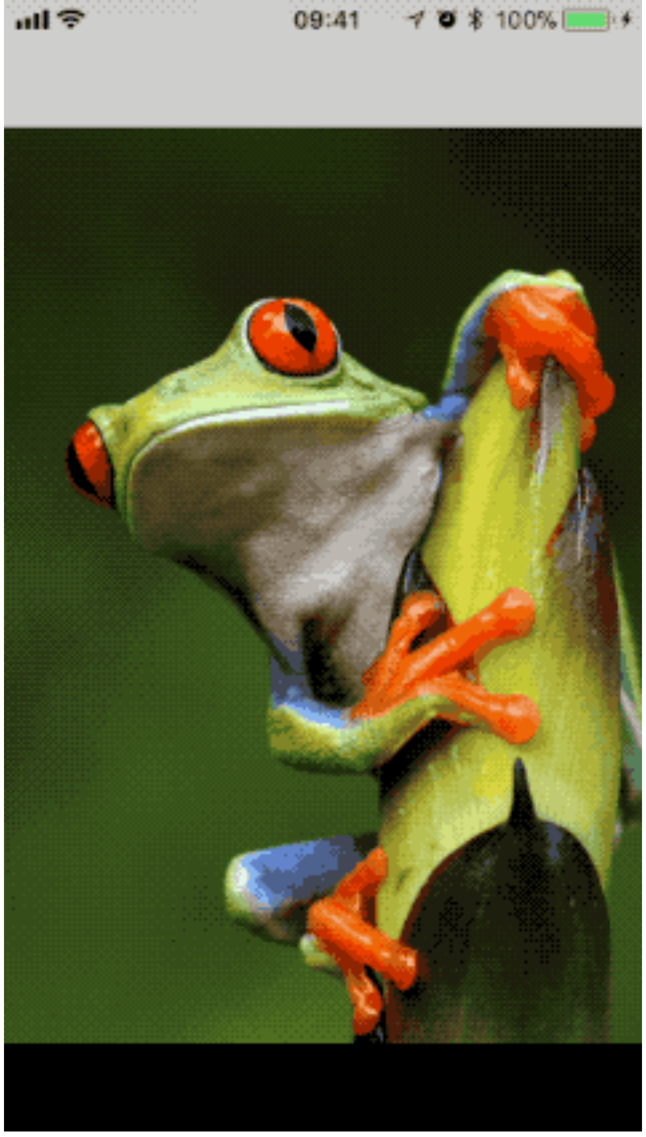
this picture shows how Embed in a **View Controller** to a **Navigation Controller**.

Now you should see photo View Controller embedded in a Navigation Controller.



Now **Photo View Controller** is Embedded in a **Navigation Controller**.

Build and run the app and try to swipe photos. You see the PagingScrollview is floating on the screen.



PagingScrollview is floating on the screen when we embed Photo View Controller in a Navigation Controller.

The reason for this behavior is a property in scroll view named `contentInsetAdjustmentBehavior`. This property specifies how the safe area insets are used to modify the content area of the scroll view. In other words, it adjusts scroll view to be scrollable in vertical when the scroll view is the content view of a view controller that is currently displayed by a navigation or tab bar controller. The default value of this property is automatic, which means it's enabled. But this is not the behavior the we want for our `pagingScrollView`, So we should disable it. For this, go to **PhotoViewController.swift** file in Xcode's Project Navigator and add `self.pagingScrollView.contentInsetAdjustmentBehavior = .never` in to `viewDidLoad()` method after `self.pagingScrollView.delegate = self` line. Build and run the app and you see It works right. But you see that navigation bar on top of screen has occupied part of our screen. It's good if you want to set some button on it, but on the other hand, it has prevented us from seeing. How about, add an effect like iOS Photos app, which in default you see navigation bar and background is white, and once you tapping on screen the navigation bar disappears and background being black. Do you agree? Let's do it.

First add this, at the top of `PhotoViewController` after `var firstVisiblePageIndexBeforeRotation: Int!` :

We define an instance of `UITapGestureRecognizer` which we use for detecting taps on screen. Next add these lines of code in to `viewDidLoad()` method after `super.viewDidLoad()` :

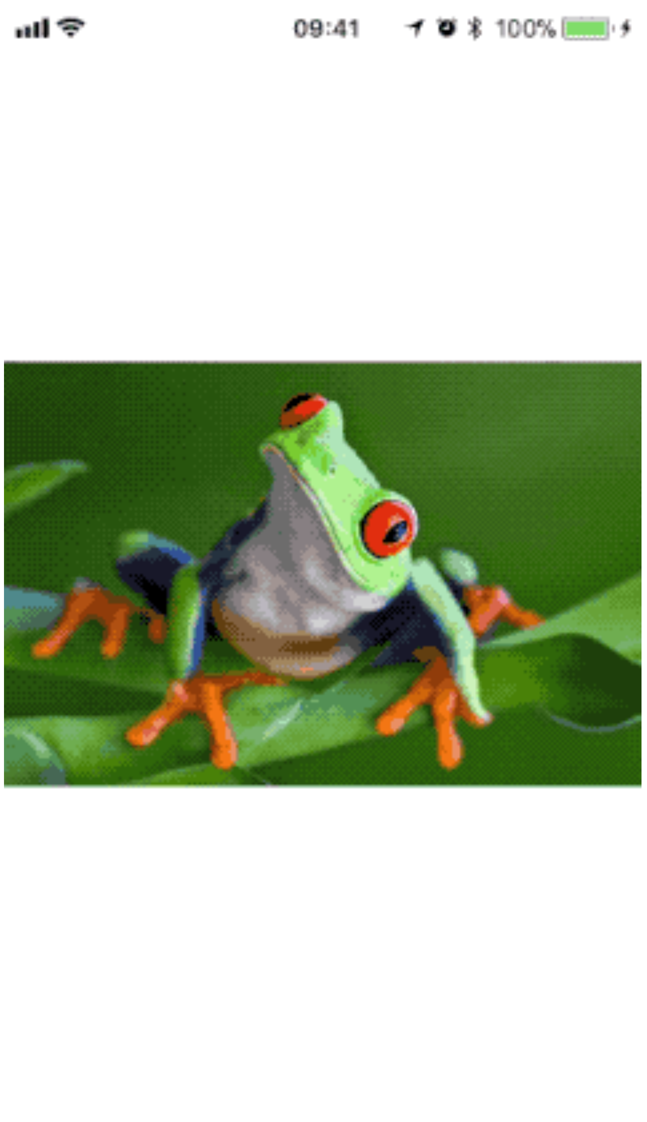
We initialized `singleTap` and we set `handleSingleTap` method as its action method (We have not created it yet). Then, we attached `singleTap` gesture recognizer to the **PhotoViewController** `view`.
Before we create `handleSingleTap` method, there is one small problem that we should consider that. We have two way to disappear navigation bar when user taps the screen. First, using navigation bar `hidden` property and second using `alpha` property. But each of them has a flaw. If we use `hidden` we can't animate disappearing of navigation bar and if we use `alpha` when user rotates the phone `alpha` value resets to its default value which is `1.0` and navigation bar appears while we ask it to be hide! So we decide to use a combination of both and also for detecting the navigation bar hidden state we define an independent variable named `navigationBarIsHidden`. So for this add `var navigationBarIsHidden: Bool = true` at the top of `PhotoViewController`, after `var singleTap: UITapGestureRecognizer!` to define this variable and to ensure that this variable is in the correct state, we check it in to `viewDidLoad()` method after `self.view.addGestureRecognizer(self.singleTap)` by these lines of code :

We check if there is `navigationController` for `PhotoViewController` and if there isn't hidden, `navigationBarIsHidden` variable be `false`. Note that we set a default value for `navigationBarIsHidden` and it's `true` so we do not need to add an `else` condition.

Now add these lines of code, after `func restoreStatesForRotation(in size: CGSize) {…}` method to create `handleSingleTap` method :

First we check, does `PhotoViewController` have a `navigationController` or not, because if there is no `navigationController`, there is no reason to handle taps. Next we check condition of navigation bar. Is it hidden or not? And as we said before, we check this with an independent variable named `navigationBarIsHidden`. So in any case we change value of `navigationBarIsHidden` to its opposite, and we change `navigationBar` alpha value and background colors animatedly. You should notice that for disappearing `navigationBar` we first fade it out with set `alpha = 0` with animation and after the end of animation we set `hidden` property of `navigationBar` to true. its because if we set `hidden` to `true` along with `alpha`, `navigationBar` suddenly disappears, and we will not have a fade out effect any more. But we do not have this problem for the appearing state. We could change `navigationBar` both `alpha` and `hidden` properties at the same time and have a fade in effect to. There is no vague point in `updateBackgroundColor()` method, we check `navigationBarIsHidden` variable and change backgrounds colors appropriately. But about `updateBackground(to color: UIColor)` method, you should notice we have some views and each view is a subview of another view. So we need change background color of all these views. So at the end all that remains is we set an appropriate background color for the time that app is launched. For this just replace `self.updateBackgroundColor()` with these lines of code in to the `viewDidLoad()` method:

Build and run the app, and check if it works fine. If everything is done right then you should see something like this:



Users can fade in/out navigation bar by tapping.

Although the app seems to work fine, but there is two issue that needs two lines of code to be fixed. First, in the above codes we just changed background color of visible pages on screen so if we scroll to other images, we'll see some of pages background color are different. Next issue, is about tapping. Tapping on screen for fading navigation works fine but, it causes `zoomingTap` of **ImageScrollView**, to be disabled. For fixing this issue, there is a magical method in `UITapGestureRecognizer` which creates a dependency relationship between the one gesture recognizer and another gesture recognizer. In other words we use this method to set priority between gesture recognizers. For example, here we ask to `singleTap` be fail if user did a `zoomingTap` which is a double tap gesture. So for fixing these two issues, add these two lines of codes into `configure(_ page: ImageScrollView, for index: Int)` method, just before `page.index = index` line :

This part ends here. Please clap this tutorial and share it to your friends if you enjoyed it.
You can download the completed sample code from the link below:

And you can study part I and part III of this tutorial from the links below: