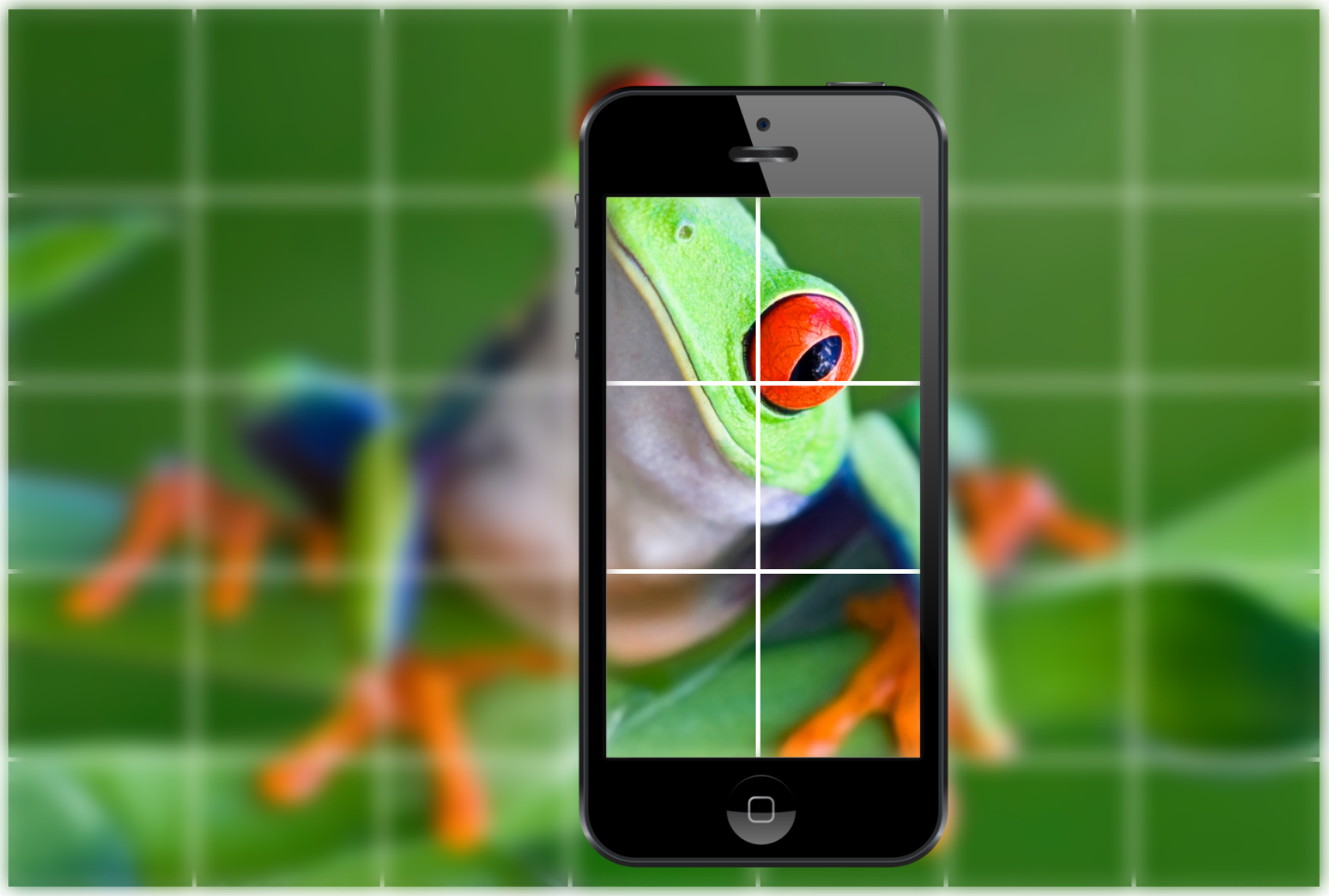
Designing Apps with Scroll Views _ Part III: Optimizing with Tiles

Based on Apple WWDC 2010 Designing Apps with Scroll Views

Seyed Samad Gholamzadeh Feb 27, 2018



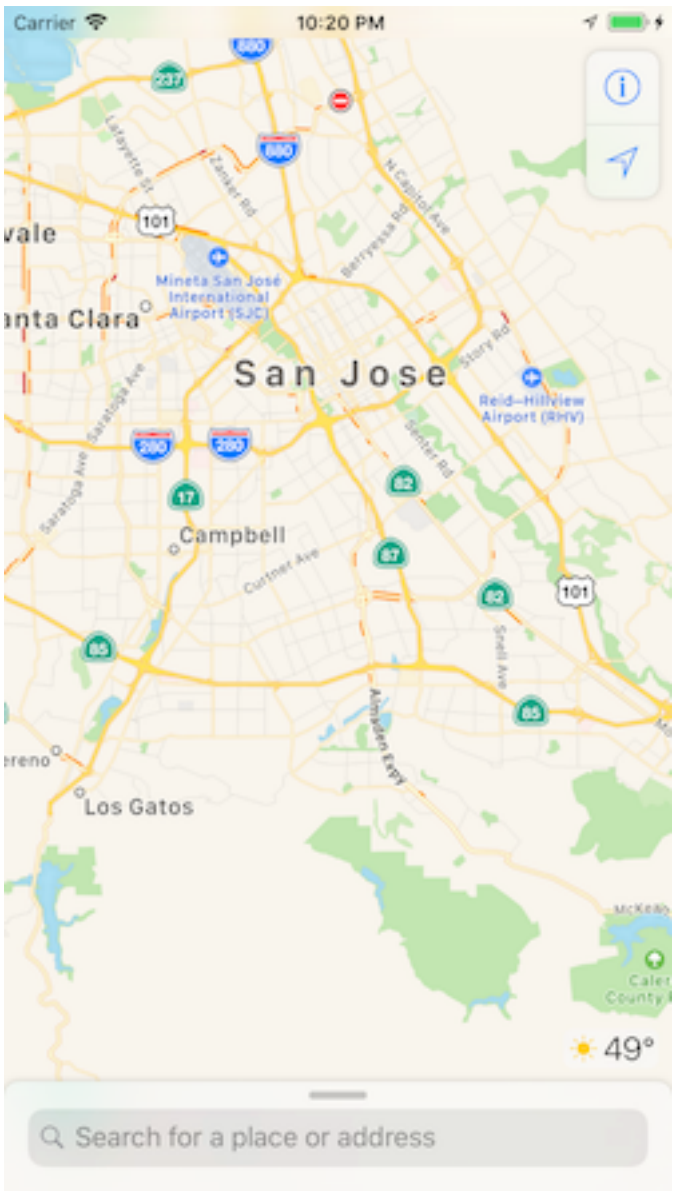
Introduction

This is the third 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 II of this tutorial from the links below :

In the previous part we learned how to create a paging scroll view and be

able to swipe and scroll on multiple photos. But if you remember, our app did not work well. The problem was that, 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. Here we talk about how to fix this issue.

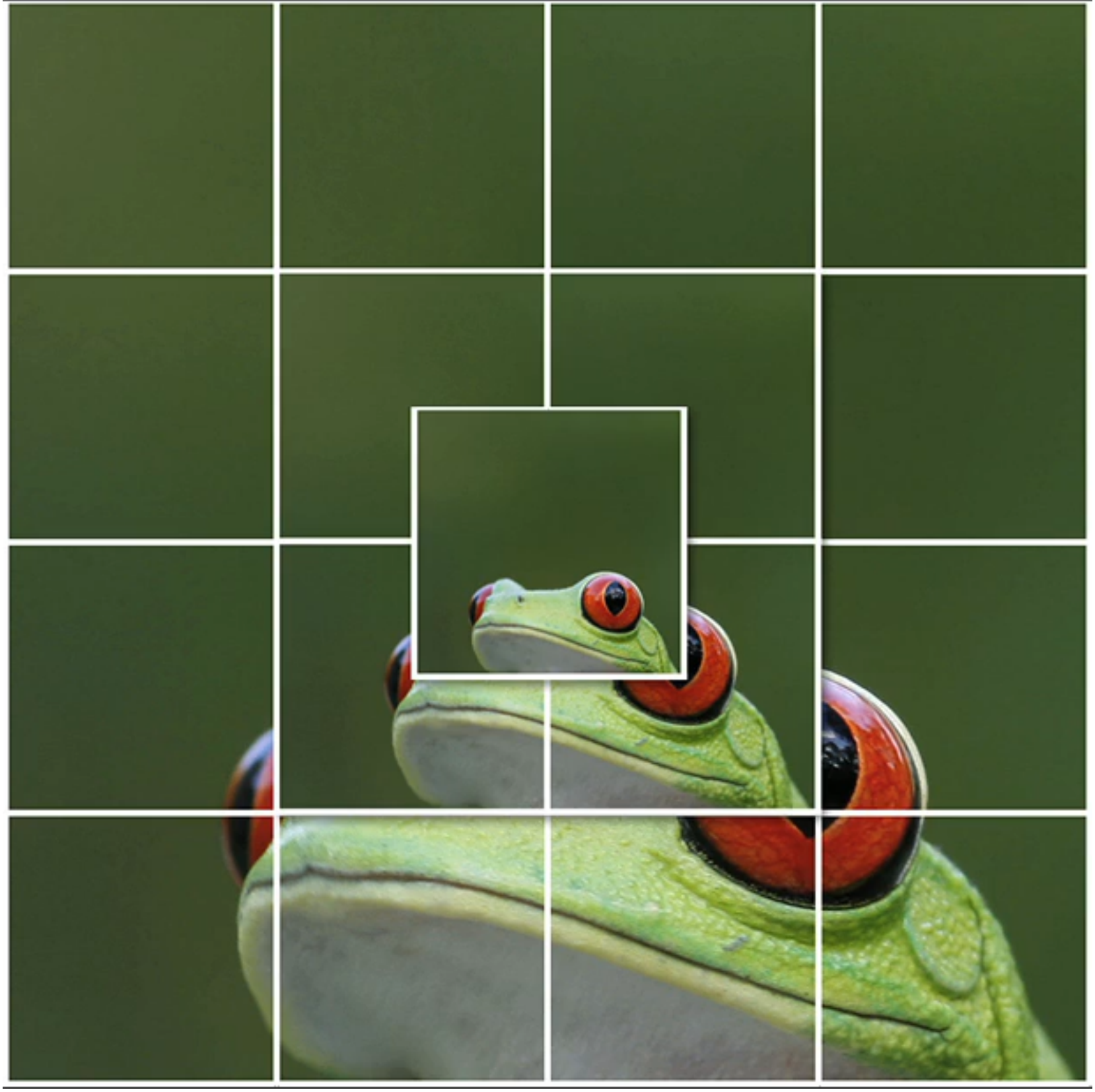
Tiling With CoreAnimation



Maps app uses `CATiledLayer to show maps images in different level of zoom`

In the picture above, you see a screenshot of iOS maps application. Most probably you know that app, it shows you exactly where you are at any given time. The way that works is by using `CATiledLayer` to lazily load all of the different tiles that make up the map that you're currently viewing and these tiles are loaded off the network on a background thread and then get drawn when they are finally available.
The good news is, a tiled layer is available for use as public API in your

own applications. So we can do the exact same thing. `CATiledLayer` easily draw your content when you're asked and handles things like caching to make sure that it's only using the amount of memory that's reasonable given what the user has looked at. It only asks you to draw things that are currently visible.
It supports multiple zoom scales. So as your user pinches to zoom in and out, it will ask you for new tiles at a different zoom scale based on how many different levels of detail you'd like to be able to display. And it manages all these for you. It's very easy to configure and you don't have to write any code to implement all that. The idea here is that we give different sets of tiles to `CoreAnimation` and as the user pinches to zoom in and out or double taps if you've set programmatic zooming, Core Animation will correctly swap between which tile is necessary to optimally display whatever zoom scale the user's currently looking at and the interesting part is that `CoreAnimation` just shows tiles that fill the screen and so what you have in memory, is just a smaller amount of whole image. it's really cool and we get all these features for free just by using `CATiledLayer`. Thanks to Programming team!



CATiledLayer uses different set of tiles to handle different levels of zoom

How to use `CATiledLayer ?`

Download the sample code from here and go to **Starter_SampleCode** folder and open the project inside it. This example is exactly what we left from previous part of this tutorial.
We are going to use a subclass of `UIView` to showing images in zooming scrollview instead of `UIImageView`. It's really easy.

In **Project navigator** in Xcode, select **ImageScrollView.swift**, and create a new file (you can do this by right clicking on ImageScrollView.swift file and select **new file** from opened menu or just

use `command+N` shortcut keys on keyboard). in opened window select **Cocoa Touch Class** and press Next. In the next window fill the textfields like below:

Class: **TilingView**
Subclass of: **UIView**
Language: **Swift**

Press Next and Create and you will see a new file named **TilingView.swift** after ImageScrollView.swift, Select it and you will see a class named `TilingView` subclass of `UIView`. This going to be the class that manages tiles for showing on screen. we need two instance variable in this class to load tiles in memory, so add this two lines in `TilingView` subclass after `class TilingView: UIView {` line:

> ***Note:*** *You will see some errors, but do not care we fix them soon.*

There are just two methods on UIView that we have to implement to begin using a `CATiledLayer`.
First off, we actually have to tell **UIKit** that we want to use `CATiledLayer`, because by default, when a `UIView` is created, that `UIView` is backed by a plain `CALayer`, not a subclass of it. So in order to change it to `CATiledLayer`, we implement a class method in this `UIView` subclass called `layerClass` and we just return `CATiledLayer` class from that method. Once we've done that, UIKit is creating a `CATiledLayer` for us to back our UIView instead of `CALayer`. So to do that, just add this lines of code in `TilingView` subclass after `var url: URL` line:

There is one more step to use the `layer` property of `TilingView` class as a `CATiledLayer` property instead of `CALayer`. Add these lines after above lines in `TilingView` class:

These lines create a get property named `tiledLayer` that just returns layer property of UIView as `CATiledLayer`. So now we can use features of `CATiledLayer` for `layer` property of `TilingView` class.

Next, add these lines of codes after above codes:

`contentScaleFactor` at default, shows rate of screen point (logical coordinate space for iPhone screen) to pixel (device coordinate space for iPhone screen). The value of this property for iPhones with retina display is more than 1 (it's typically either 2.0 or even 3.0). But to make TilingView work properly for all types of iPhone devices we need to force this value to be always 1.0 . So these lines override and force `contentScaleFactor` property of the view to be 1.0 .

Now, we need to make an initializer for our subview. add these lines of code after the above codes:

Here, `url` gives the `URL` of place that tiles placed in disk and `size` gives actual size of image we want show it in tiles. In the next two lines we initialize `url` and `imageName` that we created at the top of class, third line we implement super class `init(frame:)` initializer.
At the fourth line we tell to tiledLayer property how many levels of detail to display. What that means, is we're going to be asked in our drawRect to draw at potentially four different scales. Each one is half the previous ones. So that the maximum scale is going to be 100 percent then we're going to get asked for 50 percent, 25 percent, and 12.5 percent tiles and in fact, these images are so large that you'll see that we only are going to need the 12.5 percent tiles for quite a while as we first view them.

Now all that's left to do is actually draw our content and we do that in `draw(_ rect: CGRect)`, But what is the `draw(_ rect: CGRect)` looks like?

Uncomment `draw(_ rect: CGRect)` method at the bottom of class.

> ***Note:*** *If you don't see `draw(_ rect: CGRect)` method, just paste below lines of code in to `TilingView` class, at the bottom:*

We need two pieces of information in order to accurately draw what we're being asked. The first is the rect that comprises the tiles that class is being asked to draw, and the second is the zoom scale that `TilingView`

being asked to draw at. The rect is easy. We've got it right there.

> ***Note:*** *keep in mind, this rect is in the original bounds of that* `CATiledLayer` *and does not change while user is zooming.*

So how do we get the zoom scale though, there's no property or no parameter telling us what that is? Well, this is a little tricky. We actually have to pull the zoom scale out of the current transform matrix of the current graphics context associated with this drawRect. What does that mean? We can call `UIGraphicsGetCurrentContext` to find out which `context` we're being asked to draw into and that'll be a `CGContext` that we're actually going to draw into it using core graphics calls. Then from that, we can get the current transform matrix by calling `ctm` on the context we just got. And that's going to return the `CGAffineTransform` which is the transform that is applied to all drawings done in that context. Finally we can get horizontal and vertical scale of zoom with calling `a` and `d` on `ctm`. `ctm.a` is the `CGFloat` that represents
the horizontal scale and `ctm.d` is the `CGFloat` that represents the vertical scale that we're being asked to draw it. (It's not very important to know what they are more than this right now). So add these lines of codes into the `draw(_ rect: CGRect)` method to get these two scales:

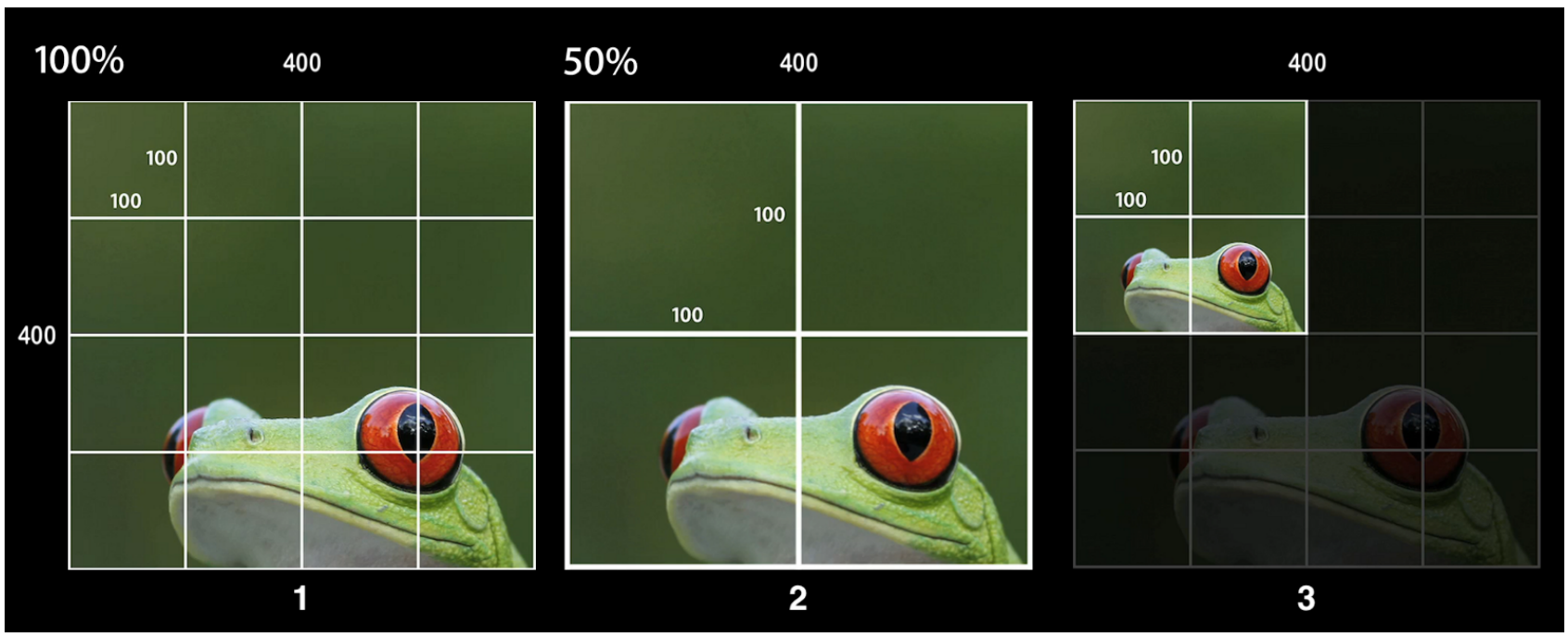
So now that we have rect and scales, in order to fill the rect that we've been passed, we're going to need to know how big the tiles are and that's a property on CATiledLayer. So all we need to do is calling `tileSize` on `tiledLayer`. Add this line into `draw(_ rect: CGRect)` method after `let scaleY: CGFloat = context.ctm.d` line:

> ***Note:*** *The default value of size is 256.0 X 256.0. It's changeable but we preferred to use the default value.*

So now comes the part that's perhaps is a little weird!
It's not going to be good enough to use this tile size as is, because if we're being asked to draw a scaled down version of our tiles, we need to

adjust the scale of tiles too. The reason for that is If we're being asked to draw at, for example, the 50 percent scale, we still need to stretch those tiles out to fill the entire region of the original image. So although our tiles have less information in them, we're going to stretch them out to be bigger than they really are to fill that full region. So we need to compensate for our scale by adjusting our tile size. And we're going to do that by dividing both the width and the height by the corresponding scale. So we're going to pretend that our tile size is bigger as we get to smaller and smaller scales.

> ***Note:*** *the `d` component of `ctm` is a negative value so, we need to multiply it by a minus (-)sign.*



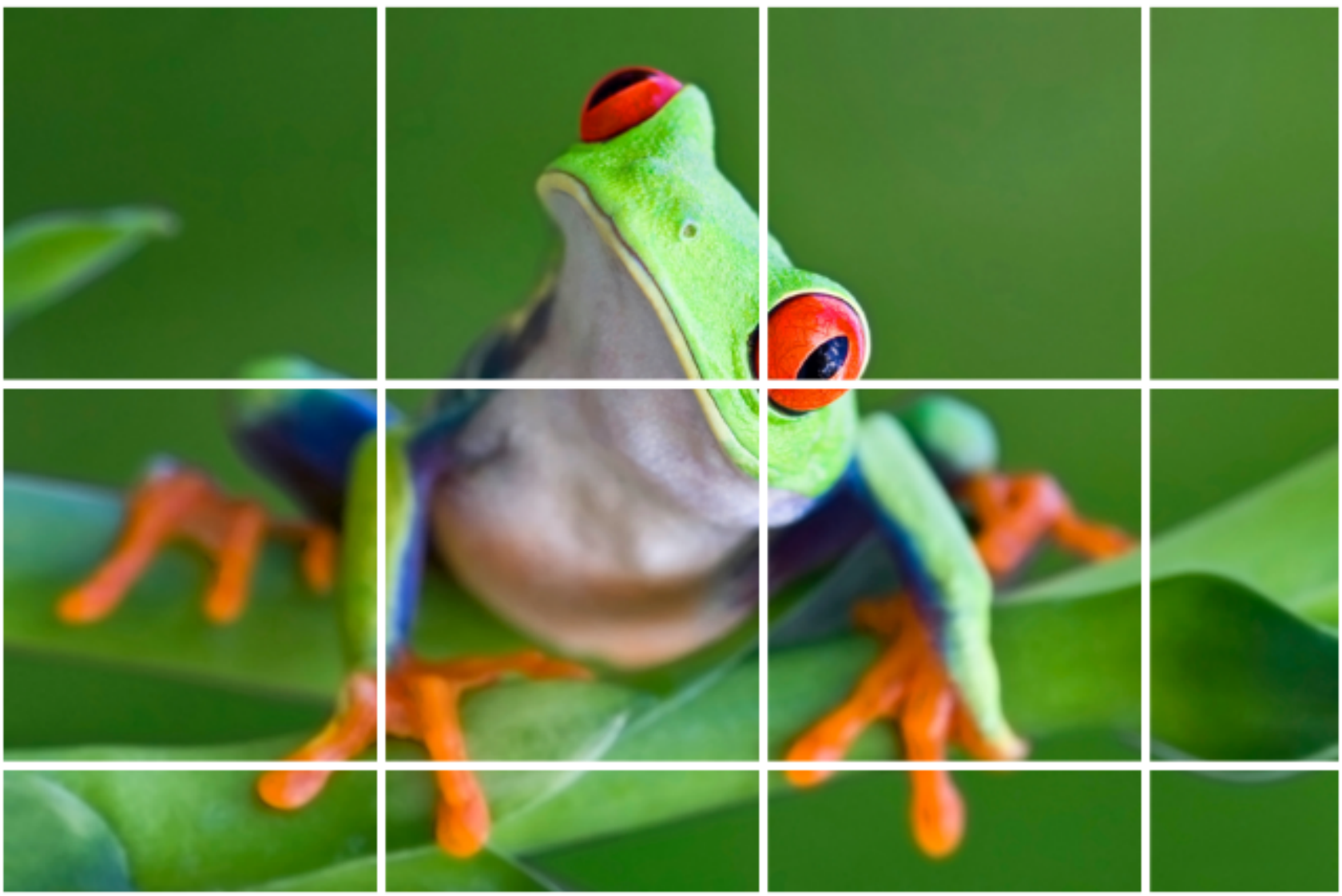
This picture shows an image with orginal size of 400 X 400. First image, shows tiles that show this image in 100% scale. Second image shows tiles that show this image in 50% scale. You see actual size of tiles in second image is 100 X 100, but they are stretched to fill original image area size. The third image shows tiles of 100% scale and 50% scale in their original size.

Now that we've got our adjusted tile size, we need to first figure out which of these tiles do we need in order to fill this rectangle that we've been passed.
Add These lines in the following of above codes:

Block of math calculating the first column that we need of tiles and the last column and the first row and the last row respect to rectangle that we need to draw tiles in it.

Next add these lines at the following of above codes:

Here we iterate for each raw and each column, and get the respective tile with a convenience method (we got it soon), calculate the rectangle that we're going to use to draw this tile into. The origin of the rectangle is the column times the width of our adjusted tile size by the row that we're on times the height of the tile size and we move over and down by the appropriate amount given what column and row that we're on. The size of our rectangle is going
to just be the size of our tiles (but adjusted once again for the scale that we're drawing at, as mentioned earlier).
There is an important point here. Our image are not exact multiple of our tile size. So we're always have at the bottom and at the right edge some partial tiles that we don't want to stretch out them. So we need to compensate for that. And we do this by checking is the rectangle that we just computed, outside of our bounds? And if it is, then we need to truncate it so that it stays within our bounds. And we do that with one line of code just by taking the rectangle intersection of our bounds and the tile rect that we're about to draw.



Most of the time we have at the bottom and at the right edge some partial tiles that we don't want to stretch out them.

So then we can just go ahead and draw our tile. For demonstration purposes we add some white lines over the tiles so that when we build the project, we'll be able to see when we change from one tile size to the next. So the last 4 lines are just a couple of codes that draw white border around the tile that we just drew.

Finally for defining `tileFor(…)` method and fixing the error, add these lines at the end of `TilingView` class:

What this method does is it just looks at the scale, the row, and the column and the name of the image that we initialized before and make the name of tile with composition of these values as string in an order that tiles are named in disk, use initialized url as place of tiles on disk and grab us the right tile.
Note that we optimize the value of scale here and that's because the value of scale that we expect to pass for 4 level of details are (0.125, 0.25, 0.5, 1), and we used this value in the name of tiles multiplied by 1000. that's mean we use (125, 250, 500, 1000) as part of the tiles names in disk memory. But sometimes the value that pass in is not exactly equal to these values. For example it may be 0.249 instead of 0.25; So we optimize the scale value to make sure we have the exact value that we expect.

This is the `TilingView` that we need.

Configure ImageScrollView

Now we should configure our `ImageScrollView` to use this `TilingView` instead of `UIImageView` for showing images.
In **Project navigator** in Xcode, select **ImageScrollView.swift** file.
In `ImageScrollView` class after `var zoomView: UIImageView!` line add this line:

```
var tilingView: TilingView?
```

We create an instance property of `TilingView` class to use it for showing image.
Next, add these lines of code, into ImageScrollView class, after `func display(_ image: UIImage) {…}` method:

We create a new method for display image with tiles. This method is very similiar to `display(_ image: UIImage) {…}`, But there are some difference.
We initialize `zoomView` property with frame. Because we have not the original image here and so we haven't size of image. Therefore we need to set the frame of zoomView manually with the `imageSize` that passed to method. Then we set a placeholder image as `image` of `zoomView`. Placeholder image is a low size image of original image and its size is fit to phone screen, and we set it as `zoomView` image to show it before tiles load on view. That's because tiles do not load instantly on the view, it takes a few milliseconds time to load and if we do not set that small placeholder, user see a whole black screen before loading of tiles and when the tiles load, user notice of them and that's not what we want. So we set a placeholder and tiles load on that, So the user will not notice the tiles anymore.
Next we initialize `tilingView` and add it as subview of `zoomView` and finally we implement `configureFor(_:)` method like `display(_ image: UIImage) {…}` method.

The last thing we need to do in `ImageScrollView` class is adding method of `placeholderImage(in:)` and get rid of that annoying Xcode's error! Add these lines at the bottom of ddd class:

This method is really straightforward. We get the name of original image from url. make name of placeholder image with compositing of original name with `"_Placeholder.jpg"` because this is the name that placeholder is saved to disk with it. Finally we make a `UIImage` with the path of

placeholder and return it.

> ***Note:*** *If you are worried about the `url` and how to get it, the rest of the tutorial is about that!*

Where are Tiles?

So far, we made `TilingView`, configured `ImageScrollView` and now we are ready to use tiles instead of whole original image in our app. But where are tiles and how we get their url address?
The bad news is that there is no Tile! We should make them manually, save them on disk and give their url address to methods to use them. The good news is, you don't need to do whole of this do by yourself. I made a simple lightweight framework that do all of these for you. It's easy to use, very high performance and optimized, so you can use it safely in your app. So let's use it in our app. Go to downloaded sample code folder. You should see a folder named **TileManager**. Open it and you will see a **TileManager.swift** file in it. Click on it, hold left mouse button and drag the file to **Project navigator** of our Xcode project and leave it after TilingView.swift file. You should see a panel window. Make sure **Copy Items if needed** is **on**. Click **Finish** and the file will be added to your project.

Now we want use this framework in our app. Go to **PhotoViewController.swift** in Xcode file and add these two lines after `var navigationBarIsHidden: Bool = true` line at top of `PhotoViewController` class:

```
var inTilingProcess: Set<String> = []

var currentImageName: String = ""
```

We will talk about these two properties later.

Now add these lines of codes into `PhotoViewController` class just after

`configure(_:,for:) {…}` method:

We created a new method to display images with tiles. In this method first we get the name of imageFile respect to its index. We set `currentImageName` property to this `imageFileName`. You will know why we do that soon. we get the url address of image in disk. Because we saved images in main bundle of app so we get the url of images in this way. And finally we made and initialized a property of our new framework `TileManager`.
Let's do some magic with this awesome framework!

Before we make tiles, we have to care about one thing. If we had some images with the sizes that are fit to screen or smaller, There is no reason to make tiles for them. Because the key idea of using tiles is we just show the part of image that is visible on the phone screen. So if the whole image is placed on the phone screen, making tiles for it is just wasting the memory of the phone.
So We should check if we need tiles for the image or not and we do this with just one simple method on `tilingManager` property:
`needsTilingImage(in:)`
So we check that and if we do not need to make tiles we just use the old method to display images in `ImageScrollView`: `display(_ : )`
and after it we say to that image please don't go further, we do not need making tiles for you with just one peace of word: `return`
So add these lines after `let tileManager = TileManager()` to do these expressions:

Now add these lines of code after above lines:

Here we've checked if the tiles are made for the image or not. And if they were made we get the size of the image and url of tiles address in disk with the respect methods on `tileManager` and then we ask page to display tiles on screen. What if we are not made the tiles, before? We need to make them.

But before it, we need to do one more thing. Add these lines of code into `else` braces:

This is why we made the `inTilingProcess` property. In this tutorial app we start tiling process for each image in the time that user swipe on that in `pagingScrollView`. So suppose user swiped on an image and its tiling process is started. It takes a few seconds to the process get done. It may user swipe back on that image again and another tiling process begins, but that's not what we want. So we should care of that. we made a `Set` property named `inTilingProcess`, and each time tiling process for an image begins we add its name to this set, and if user swipe again on that image we check this set to be sure we do not start more than one tiling process for each image. Also if we have a placeholder for our image we show that on screen. That's all that happens on above codes.
Now it's time to really make the tiles. Add these lines after above lines:

This is the method that makes tiles and placeholder for each image that we give it its url. It has two completion handles. One for placeholder and one for tiling completion. If placeholder be made we show that to user. And if tiling process be done we check if the `imageName` is returned with `tilingCompletion` is equal with the name of image user is currently on it we load tiles on screen and then we remove image name from `inTilingProcess` set. The last thing we need to do is replacing this new method for displaying images with the old one. Replace below line with the last line in `configure(_:for:)` method, which is `page.display(self.image(at: index))`:

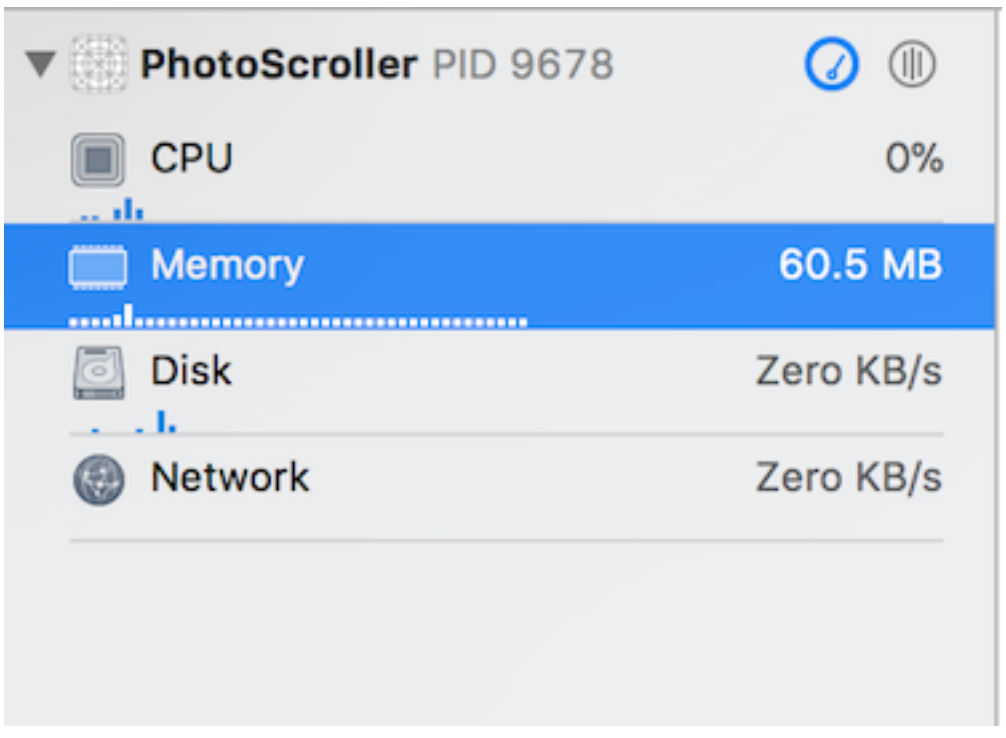
```
self.displayImage(at: index, in: page)
```

And done! The app is ready to use. Run the app and check it now and see how it works.

Analyzing

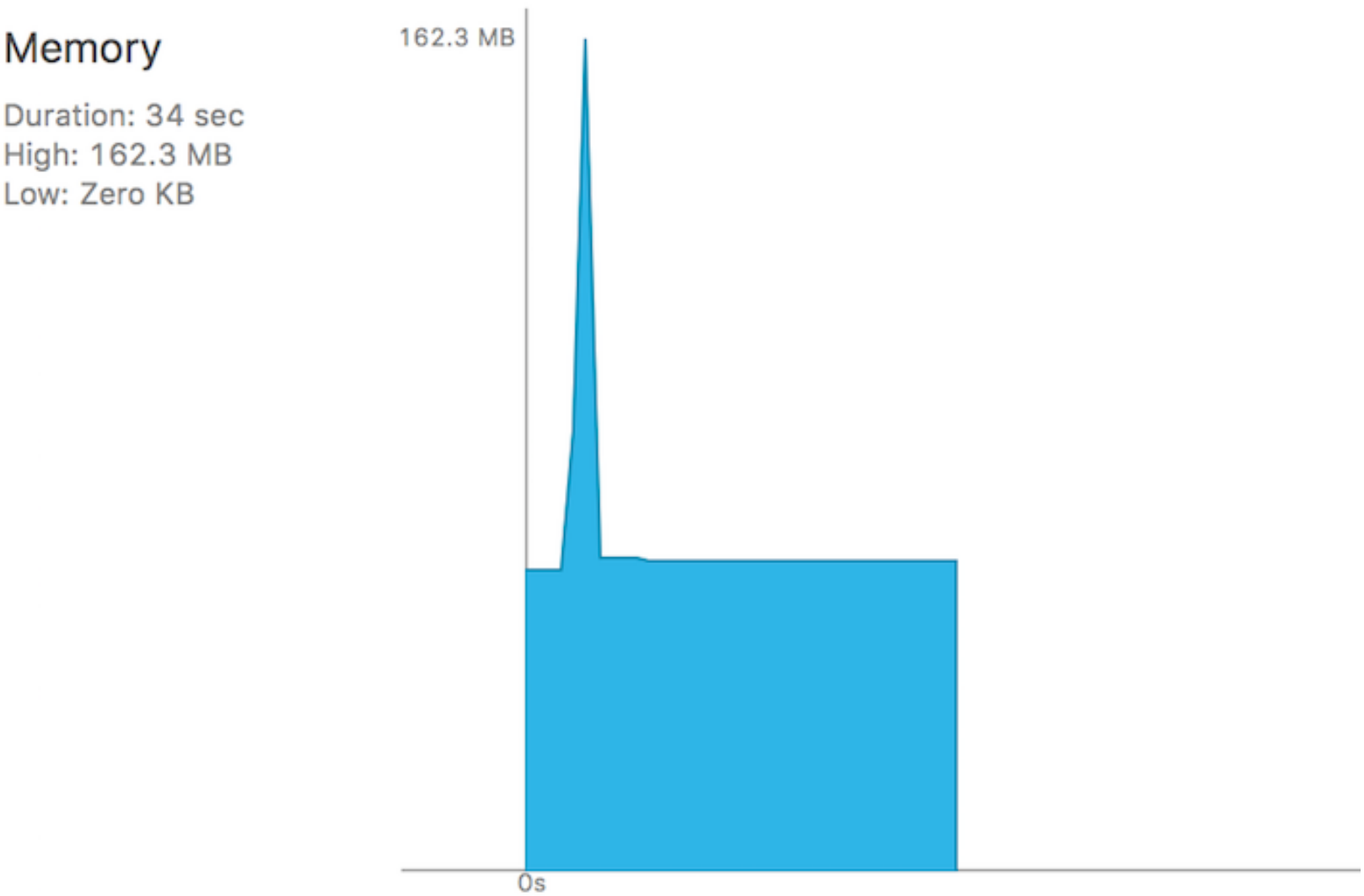
Let's take a look at the **Debug Navigator** and see how much memory

we're using this time. While you are running the application for the first time, go to Xcode's Debug Navigator, and you will see something like this:

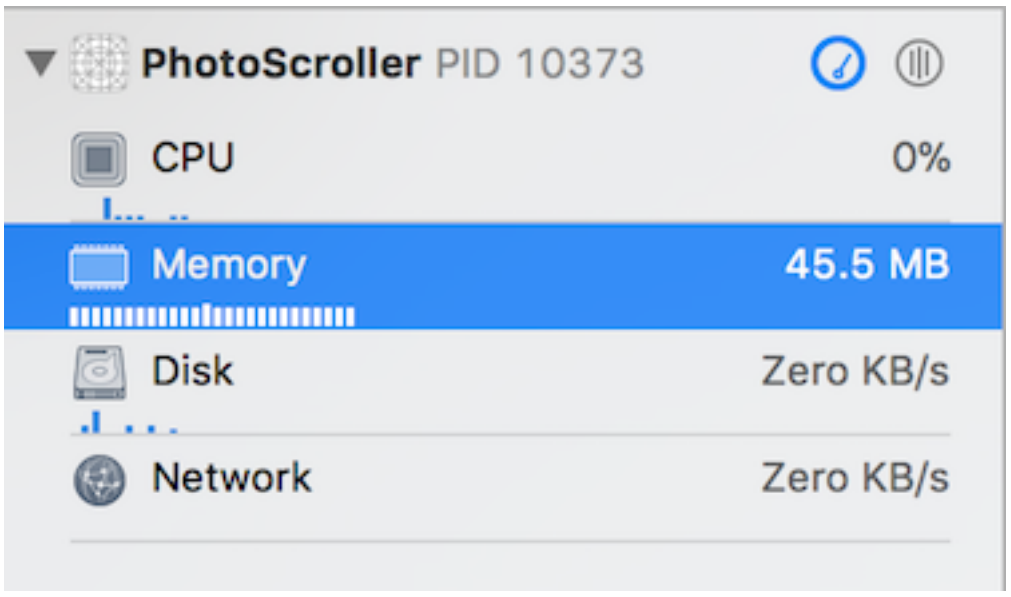


Xcode's Debug Navigator

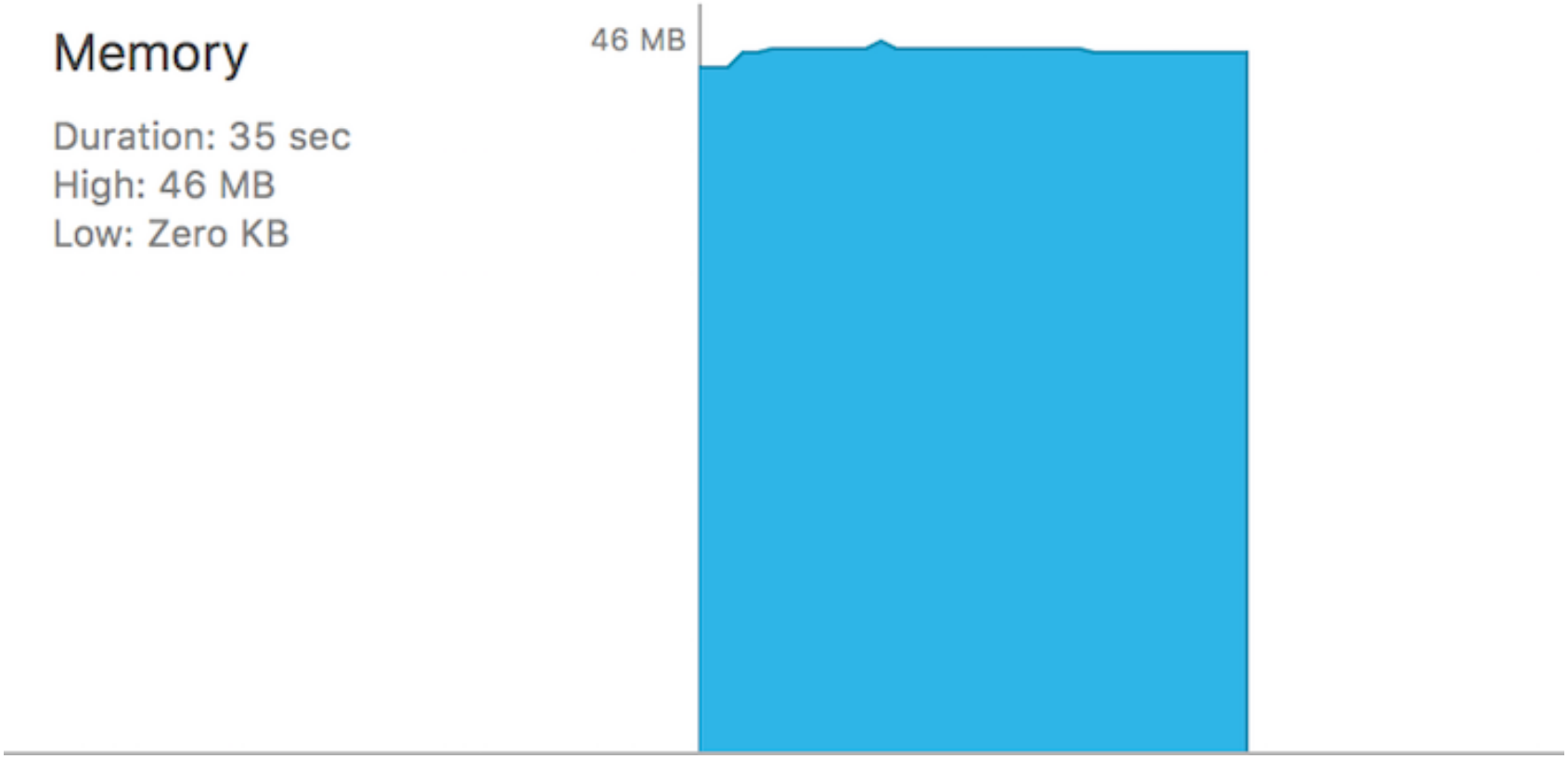
The usage of memory is about 60.5 MB. While you are in **Xcode's Debug Navigator**, click on **Memory** and you will see something look like this in the middle of Xcode's window:



This is the chart of app's memory usage. you see a high memory usage at the beginning of the chart. The details at the left side of chart show that the value of this usage was about 162.3 MB. This usage of memory is for making tiles by the app at the time user swipes on images and once the tiles were built, there is no high value usage of memory. If you felt some lags for your first swipe on images, that's because of that. Stop running of project in the Xcode.
We want run the app again, and see the difference. Because this time you run app the tiles are made and you do not need to pay any extra memory usage for making the tiles. Run the app from Xcode and go to **Debug Navigator**, you will see something like this:



Click on **Memory** and you will see something like this:



Did you get the point? It's amazing! That 46 MB are not all for scrolling photos, most of it is related to the app itself and just about 2 MB are related to our scrolling and zooming on photos. So you run the app, It shows some high quality photos, you zoom in/out on them, swipe between them and all the memory you're paying for these are about 2 megabytes. Isn't that cool?
The more interesting part here is that there is no difference between the photos quality anymore. Each photo with each quality, just uses that about 2 MB of memory. It's really like magic.

> ***Note:*** *You may ask what about the first time that the user runs the app and see that lag. The truth is that, we did not use the* `TileManager` *framework in a good place. The idea that creating tiles right when the user needs them is not a good idea, but we use it there For educational purposes. So if you want to use this framework in your app, use it where to be able creating tiles before user needs them. In this case, you get rid of that lag and you will have a really efficient paging scroll view in your app.*

Find tiles in disk

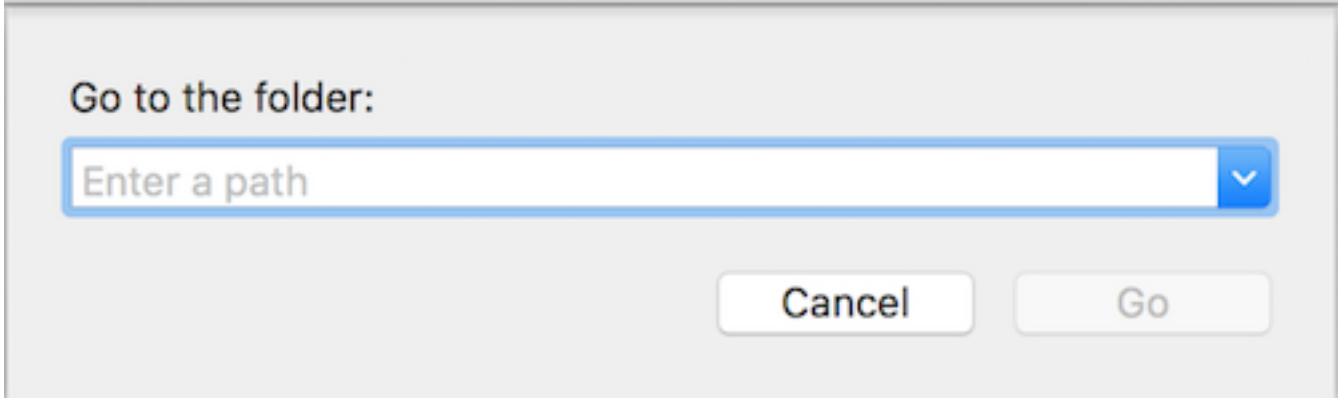
Do you want to see tiles in disk? While you running app, in the **Debug area**, in the **Console**, you should see something like this:

```
_dispatch_queue_override_invoke + 1458
12  libdispatch.dylib                   0x0000000111fc7102
_dispatch_root_queue_drain + 772
13  libdispatch.dylib                   0x0000000111fc6da0
_dispatch_worker_thread3 + 132
14  libsystem_pthread.dylib             0x000000011247f1ca
_pthread_wqthread + 1387
15  libsystem_pthread.dylib             0x000000011247ec4d
start_wqthread + 13
/Users/SeyedSamad/Library/Developer/CoreSimulator/Devices/
94BAA9F6-5A52-49A3-916E-A10FFD8E0A6A/data/Containers/Data/
Application/F08CDE73-BBE3-4885-96D3-406965A90FAE/tmp/TileManager/
CuriousFrog
/Users/SeyedSamad/Library/Developer/CoreSimulator/Devices/
94BAA9F6-5A52-49A3-916E-A10FFD8E0A6A/data/Containers/Data/
Application/F08CDE73-BBE3-4885-96D3-406965A90FAE/tmp/TileManager/
CuriousFrog
```

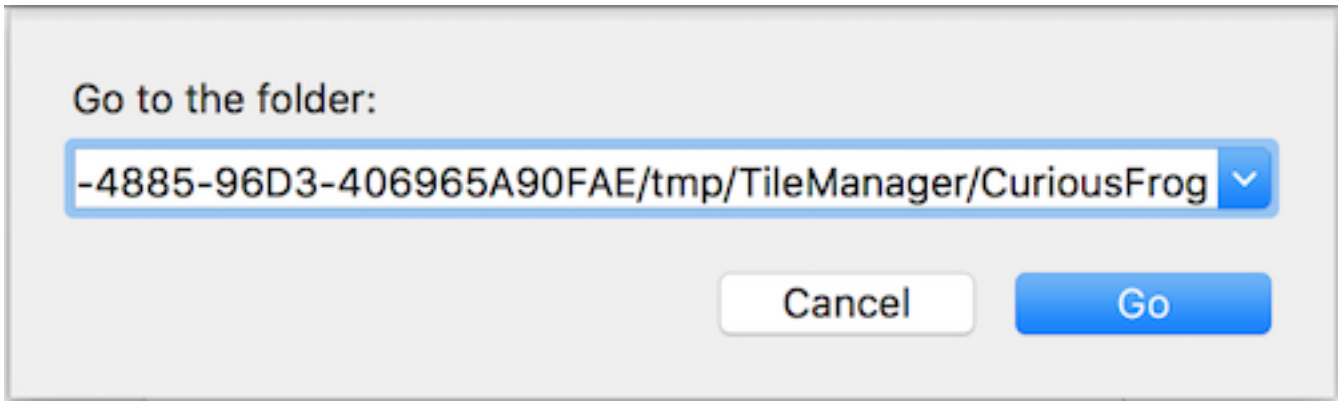
Every time you swipe on each photo, Console prints the address of the tiles are belong to that photo. In the picture above, the last two lines shows the address of tiles belong to **CuriousFrog** photo. Select on of these lines like below picture:


```
_dispatch_queue_override_invoke + 1458
12  libdispatch.dylib                   0x0000000111fc7102
_dispatch_root_queue_drain + 772
13  libdispatch.dylib                   0x0000000111fc6da0
_dispatch_worker_thread3 + 132
14  libsystem_pthread.dylib             0x000000011247f1ca
_pthread_wqthread + 1387
15  libsystem_pthread.dylib             0x000000011247ec4d
start_wqthread + 13
/Users/SeyedSamad/Library/Developer/CoreSimulator/Devices/
94BAA9F6-5A52-49A3-916E-A10FFD8E0A6A/data/Containers/Data/
Application/F08CDE73-BBE3-4885-96D3-406965A90FAE/tmp/TileManager/
CuriousFrog
/Users/SeyedSamad/Library/Developer/CoreSimulator/Devices/
94BAA9F6-5A52-49A3-916E-A10FFD8E0A6A/data/Containers/Data/
Application/F08CDE73-BBE3-4885-96D3-406965A90FAE/tmp/TileManager/
CuriousFrog
```

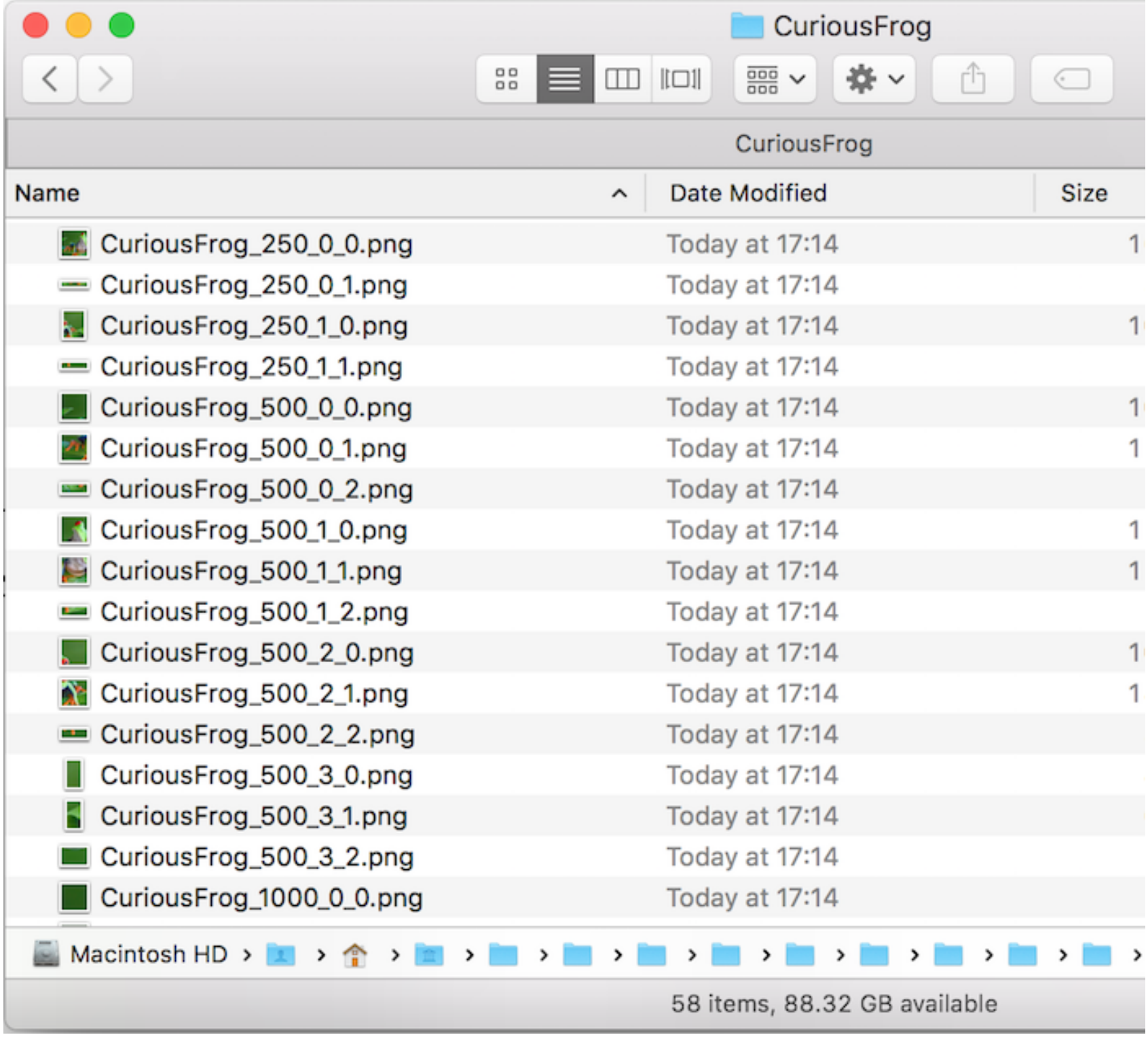

Now hold `command+C` on the keyboard to copy this line into clipboard. Click on Finder app in Dock, to go to on Finder window. Now, while you are on Finder window hold `command+shift+G` on the keyboard to see a panel on the above of Finder window, like blew picture:



Hold `command+v` on the keyboard to paste the adress that you saved in clipboard to the textfield:



Click on **Go** button and you should see a window like this:



This is the folder of CuriousFrog tiles on disk. You will see tiles are saved in different scales. If you go to Enclosing Folder (by holding `command` and pressing ↑ on keyboard), you will see the tiles folders belong to other photos.

Take a deeper look at TileManager

TileManager is a great framework If you want to take advantage of tiles in your app. It is actually a `struct` which its initializer takes three argument in which each of them have a default value. So the real initializer of this Struct is like this:

```
let tileManager = TileManager(destImageSize: Int,
sourceImageDownSizingTileSize: Int, tileSize: Int)
```

destImageSize:
When you ask of `tileManager` to make tiles for one image, it first down sizes that image to value of `destImagaeSize` and If no value is specified for this argument, it uses its default value that is **60 MB**. The reason is that the final tiles be made faster and less space be occupied on the disk. If you choose a value for this argument which be bigger than the original image size in memory, `tileManager` automatically uses that image with its original size for tiling. To know the original size for each image in memory, you can use `totalMBForImage(in: URL)` on `tileManager`.
If you do not want to reduce the size of the image, you can just set

```
tileManager.downSizeSourceImage = false
```

sourceImageDownSizingTileSize:
This argument is because of that, we do not want to down size whole of source image instantly, because that needs to load whole of source image in memory and it occupies a lot of memory. Instead we shrink the source image to some small tiles and down size these tiles in order. You should be careful about setting value of this parameter. Setting very small value causes high cpu usage and setting very large value causes high memory usage. The default value of this parameter is **20 MB**.

tileSize:
The size of each tile used for CATiledLayer. The default value is **256**.

After you initialized **tileManager**, You can use its methods to manage tiles.
Here is a short description for methods are available with this framework:

```
urlOfTiledImage(named imageName: String) -> URL
```

A method for **getting the url** of tiles for each tiled image.

```
urlOfPlaceholderOfImage(named imageName: String) -> URL?
```

A method for **getting placeholder image** of each tiled image.

```
removeTilesForImage(named imageName: String)
```

Removes directory of tiles respect to each tiled image if exist.

```
clearCache()
```

Removes directory of whole tiles that created for this app.

```
needsTilingImage(in url: URL) -> Bool
```

Checks whether it **is needed** to make tiles for the image that passed its url.

```
tilesMadeForImage(named imageName: String) -> Bool
```

Checks whether **tiles made** for the image that passed its url.

```
sizeOfTiledImage(named imageName: String) -> CGSize?
```

Returns the resolution size of image that its tiles are made.

```
resolutionForImage(in url: URL) throws -> CGSize
```

Returns the resolution size of image that passed its url.

```
destinationResolutionForImage(in url: URL) throws -> CGSize
```

This method **calculate** that how would be the **resolution of image** that passed its url if it being down sized with the parameter of initializer.

```
totalMBForImage(in url: URL) throws -> CGFloat
```

This method **calculate** that total **size (in megabyte) of image** that passed its url when it is uncompressed and loaded in memory.

```
makeTiledImage(for url: URL, placeholderCompletion: @escaping (URL?,
Error?) -> Swift.Void, tilingCompletion: @escaping (String?, CGSize?,
Error?) -> Swift.Void)
```

Down sizes, makes placeholder and Tiles for given image url.

Remove lines around tiles

Our app is now complete. We used some magical tricks and the app works now great. As a magician, we should keep our tricks secret, but those white lines around tiles reveal our secret. So we'd better get rid of them. Do this with just a small change in `TilingView` class, at the end of `draw(_ rect: CGRect)` method, where we draw white lines around tiles. By just changing `true` to `false`, the white lines disappear. The last four lines in draw `draw(_ rect: CGRect)` method, should be like this:

Build and run the app, and you will see the white lines gone.

> ***Note:*** *You may notice to some purple issues, that alert "* ***UI API called from background thread*** *". These alerts are because of that we used CATiledLayer, and in its description we read that :*
> *"A layer that provides a way to asynchronously provide tiles of the layer's content, potentially cached at multiple levels of detail.*

> *As more data is required by the renderer, the layer's draw(in:)method is called on one or more background threads to supply the drawing operations to fill in one tile of data."*
> *So, you should not pay attention to these warnings !*

conclusion

We got to the end of the third part of our tutorial and actually the final

part of **Designing Apps with Scroll Views**. We started from scratch and went forward step by step.

- In the first part we learned how to create a zoomScrollView
- In the second part we learned how to create a pagingScrollView
- And in this part we learned how to take advantage of tiling to optimize our scrollView.

Thank you for accompanying us in these tutorials.
Please clap this tutorial and share it to your friends if you enjoyed it.
You can download the completed sample code of this tutorial from here and you can access to the previous two parts of this tutorial from the links below :