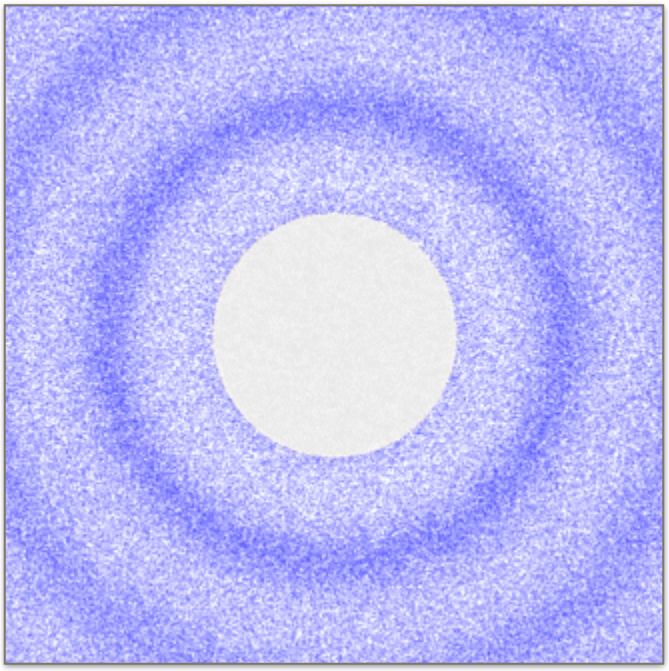
SonicAtomic Code Manual



Thomas Watts

tww55@cornell.edu

Table of Contents

Design Philosophy

The objective of this application is to create an intuitive analytic tool for aspiring scientists with visual impairments. Therefore, further development of the app must adhere to a few constraints. The user of the app is assumed to be blind and won't be able to respond to a traditional iOS app user interface that relies heavily on the ability to see the device's screen. As such, the app's response to touch input must be entirely conveyed through sound. In later sections of this text, I will describe in more detail how I address these constraints and suggestions as to how you may approach further development of the app.

Prerequisite Code Libraries

The current prototype of the app relies on a few external code libraries that do not come prepackaged with your installation of Xcode. These external libraries greatly extend the functionality of this app and offer a wide range of features that will be invaluable to you as you develop the app further. These external code libraries are AudioKit and OpenCV. AudioKit allows you to produce sound within the app and is therefore at the heart of this app's functionality. OpenCV allows you to access and process image data. Apple does not make it easy to access per pixel grayscale intensity values. OpenCV allows you to do this easily as well as offers further potential for image data processing that may become useful as the app is developed.

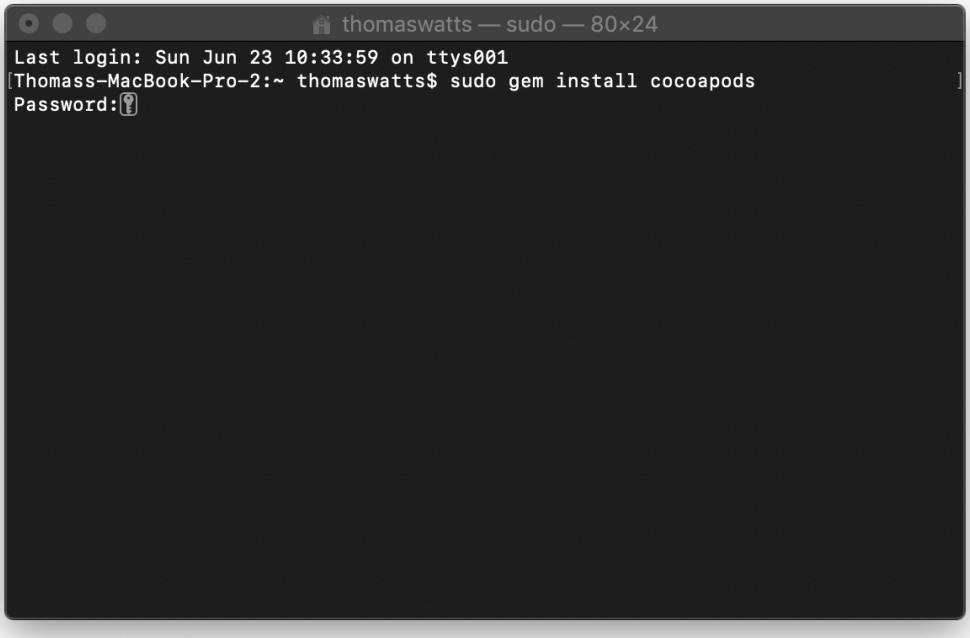
The next two sections give a step by step explanation of how you might go about installing these third party libraries in a new Xcode project. If, however, you choose to continue to develop on top of my original Xcode project, then feel free to skip to the later sections that explain how my code works: Guide to the Code Behind SonicAtomic.

Installing AudioKit

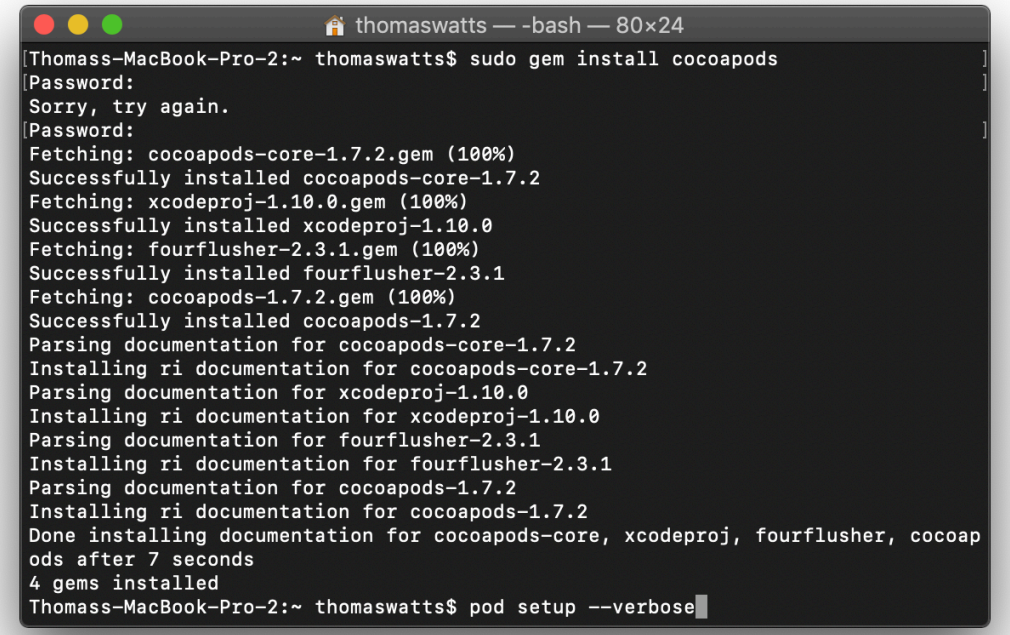
In order to use AudioKit, you must install CocoaPods on your computer. Installing CocoaPods allows you to incorporate third party code libraries into an Xcode project with relative ease. If you already have CocoaPods on your computer, then jump ahead to Adding AudioKit Pod to an Xcode Project.

Setting Up CocoaPods on your System

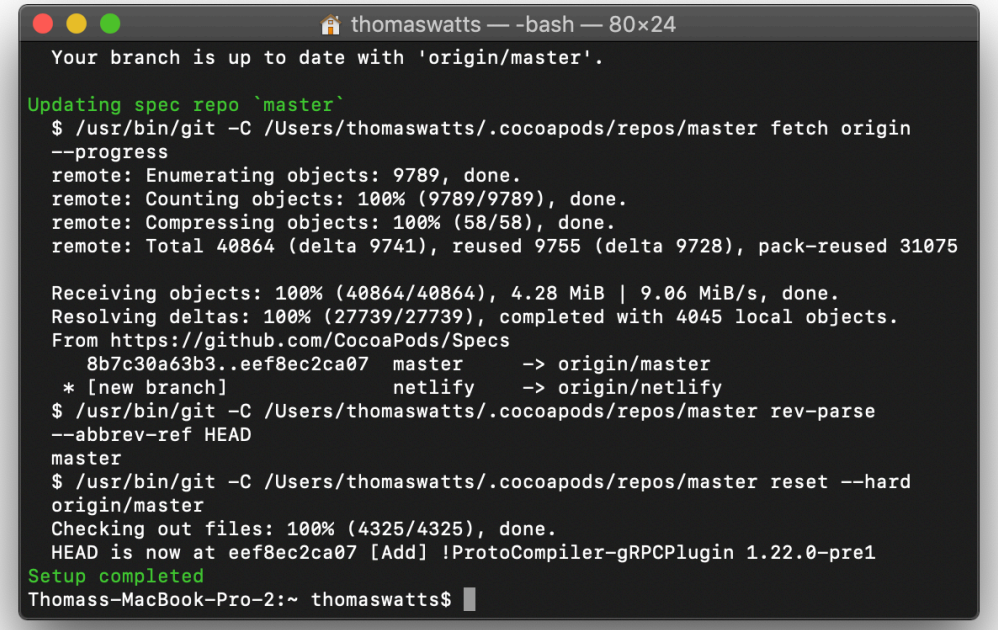
Step 1: Open **Terminal**, type `sudo gem install cocoapods` and hit **return**.



Step 2: Enter your computer password and hit **return**. Once you see `your_username$`, type `pod setup --verbose` and hit **return**.



Step 3: Wait until you see **`Setup completed`** and **`your_username$`**. You have now installed CocoaPods!

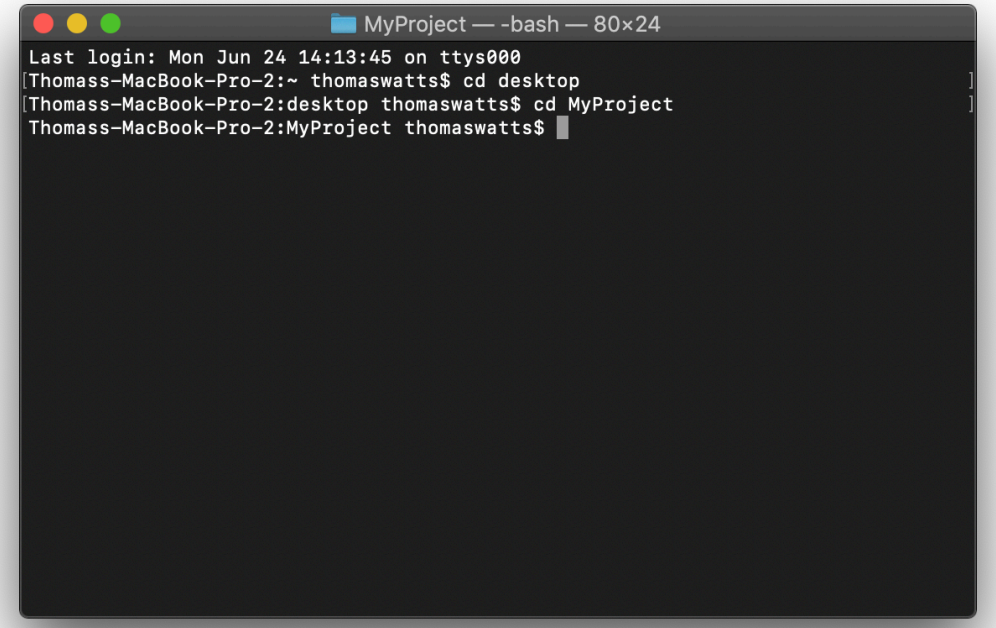


Adding AudioKit Pod to an Xcode Project

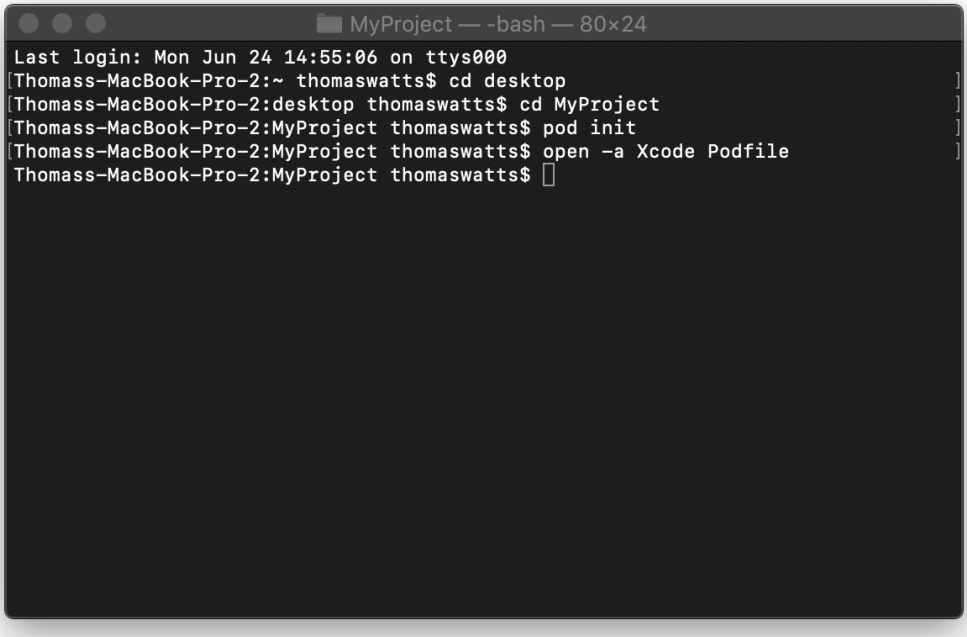
Step 0: Close all Xcode Projects before doing anything!

Step 1: Open **Terminal** and navigate to file directory where your Xcode project folder is saved. Let's suppose your project folder is called 'MyProject' and it is saved to your desktop.

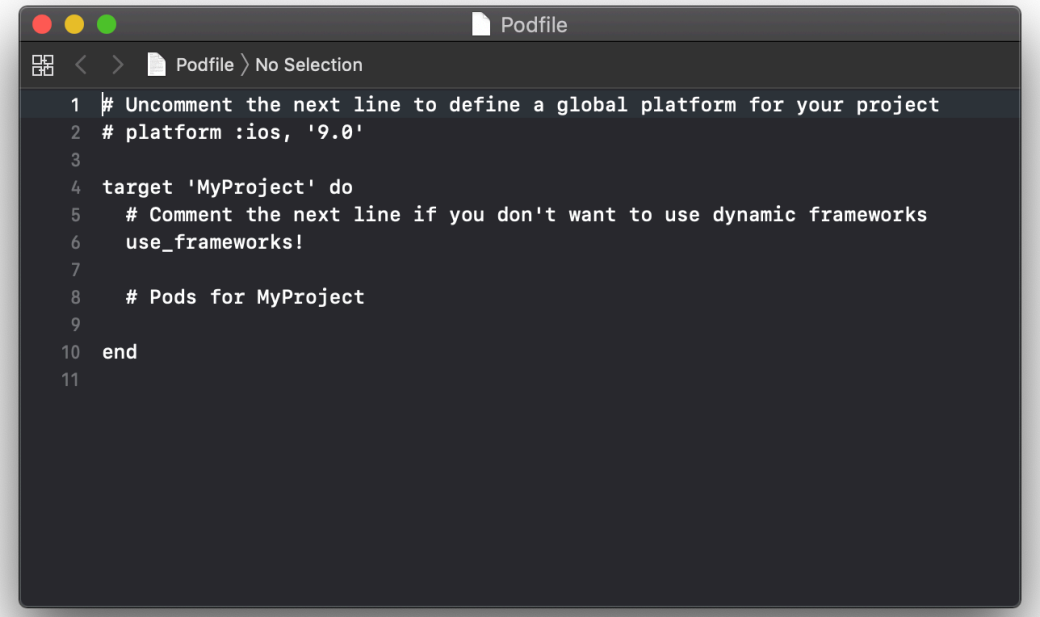
First type **`cd desktop`** and hit **return**. Then type **`cd MyProject`** and hit **return**.



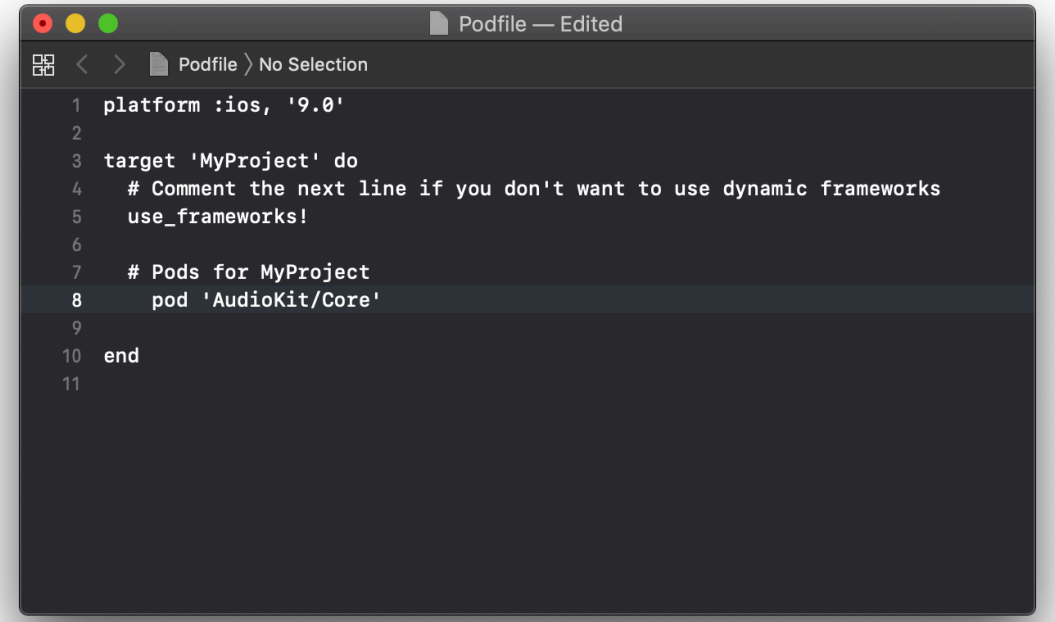
Step 2: Type **`pod init`** and hit **return**. Then type **`open -a Xcode Podfile`** and hit **return**.



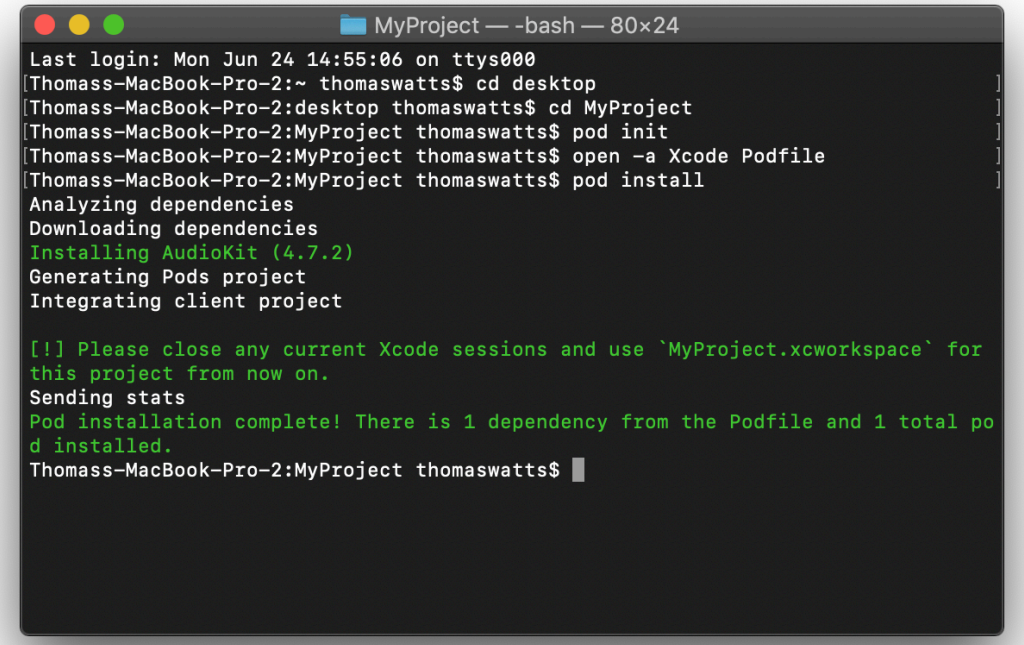
You should see the following window pop up:



Step 3: Delete **`# Uncomment the next line to define a global platform for your project`** and the delete the hashtag **`#`** that comes before **`platform :ios, '9.0'`**. Then add **`pod 'AudioKit/Core'`** below **`# Pods for [your_project_name]`**.



Step 4: Back in **Terminal**, type `pod install` and hit **return**. Wait until you see **`your_username$`**. Now, inside your Xcode project file, you should see as **.xcworkspace** file.



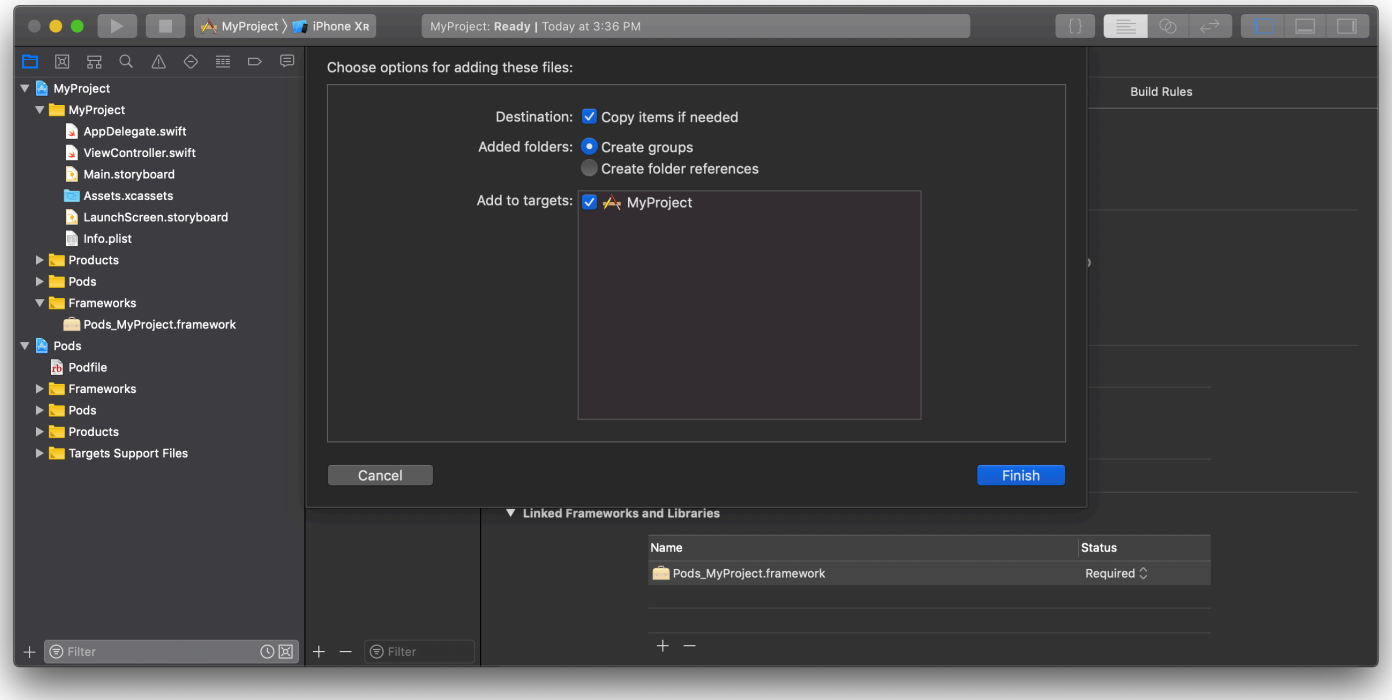
Important Note: **Only use the .xcworkspace file to edit your Xcode project!**

Installing OpenCV

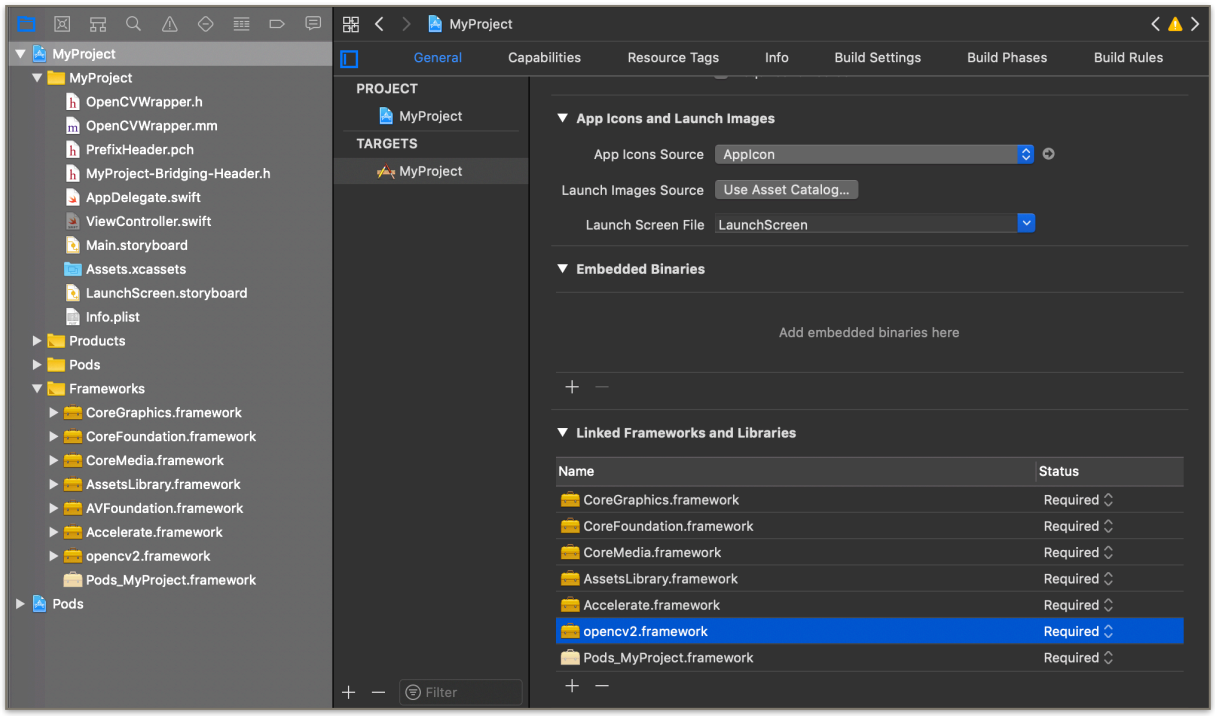
The process for installing OpenCV is more involved than the installation of AudioKit, therefore, it is important to check that you've done each step correctly!

Step 0: Go to https://opencv.org/releases/ and download the latest version of the **iOS pack** of OpenCV. You should end up with a file called **opencv2.framwork**.

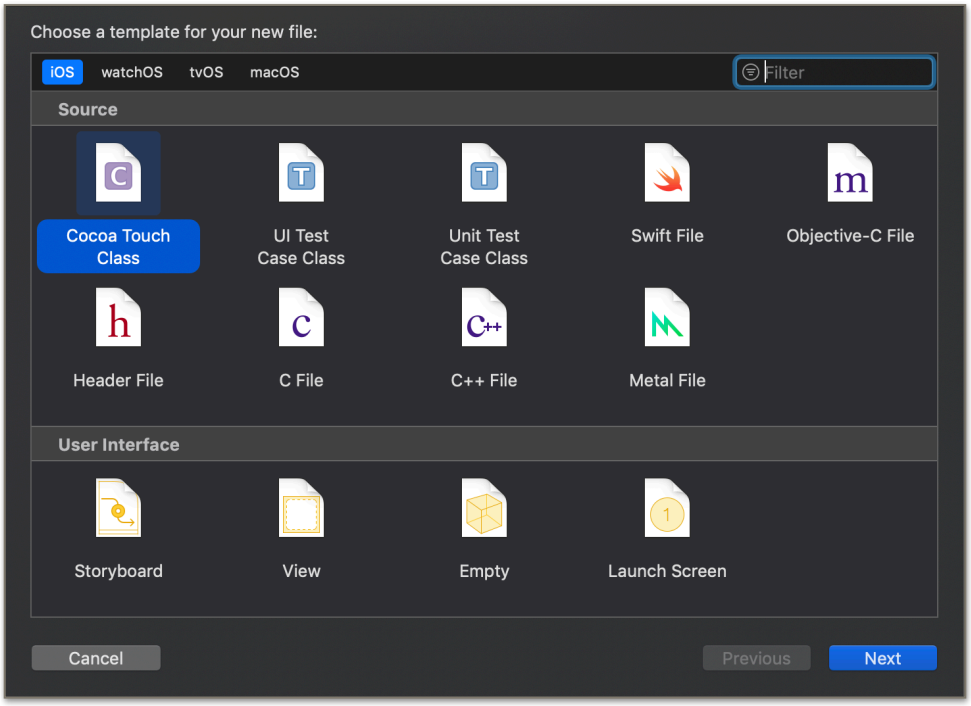
Step 1: Drag the **opencv2.framwork** file into the **Frameworks** folder in your Xcode project. When prompted, check the box next to **Copy items if needed** and click **Finish**.



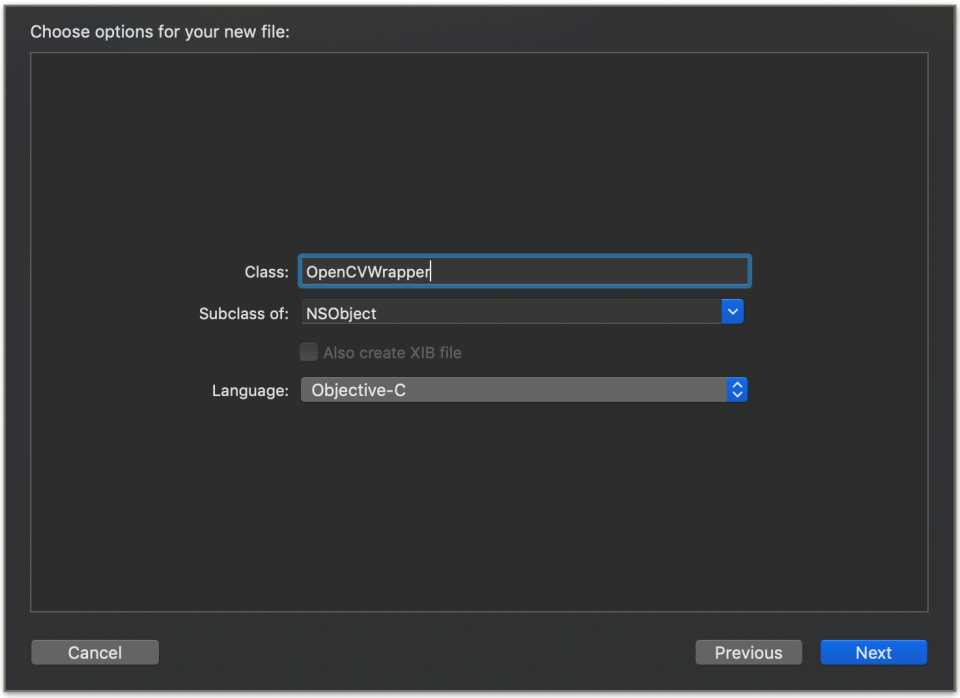
Step 2: Click **MyProject** next to the blue page icon and navigate to the **General** tab. There you will see a section called **Linked Frameworks and Libraries** under which you should see **opencv2.framwork**. Click on the plus icon **+** and add the following frameworks: **CoreGraphics**, **AssetLibrary**, **Accelerate**, **CoreMedia**, and **CoreFoundation**.



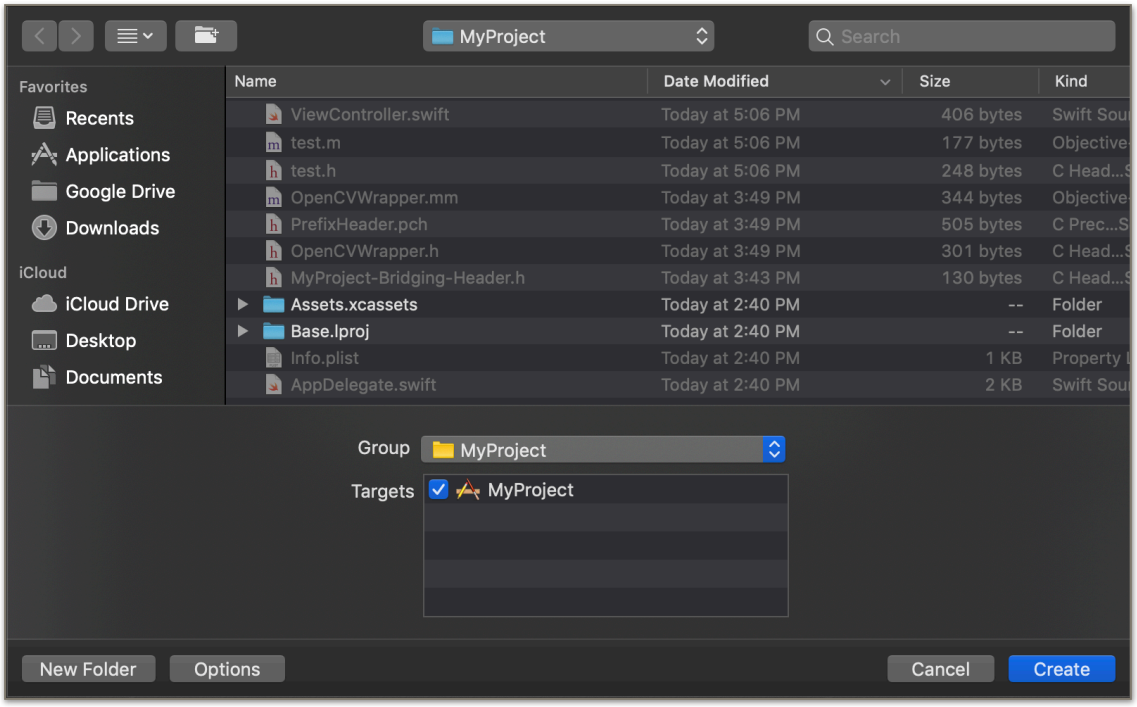
Step 3: Go to **File -> New -> File…** and select **Cocoa Touch Class**, then press **next**.



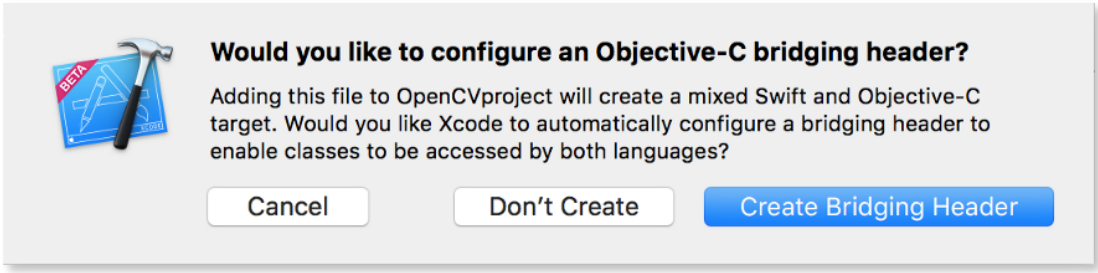
Step 4: Name this Cocoa Touch Class **"OpenCVWrapper"**. Make it is a subclass of **NSObject** and that **Objective-C** is selected as the language, then press **next**.



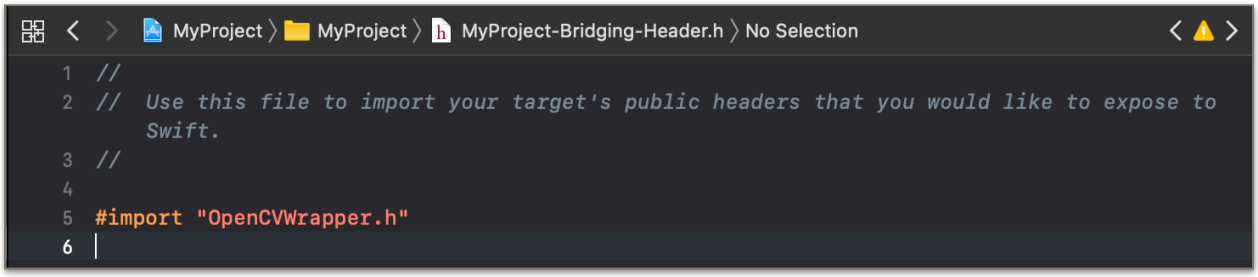
Step 5: When prompted, make sure the **MyProject** folder is selected as the **Group** and that the box next to **MyProject** is checked under **Targets**, then click **Create**.



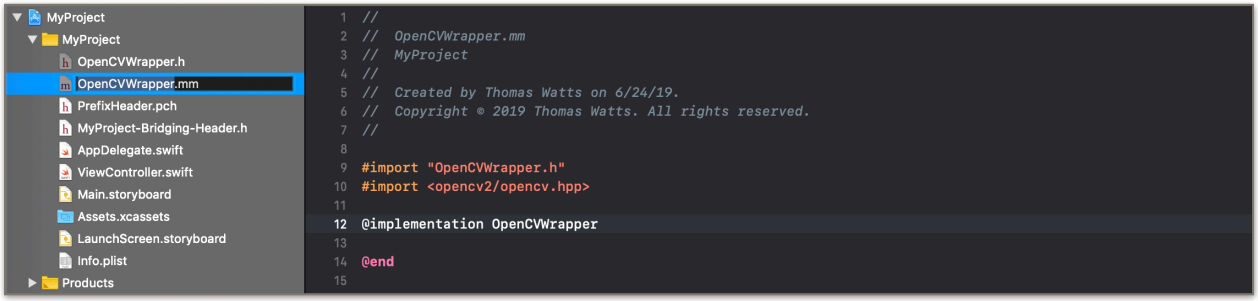
Step 6: When prompted, click **Create Bridging Header**.



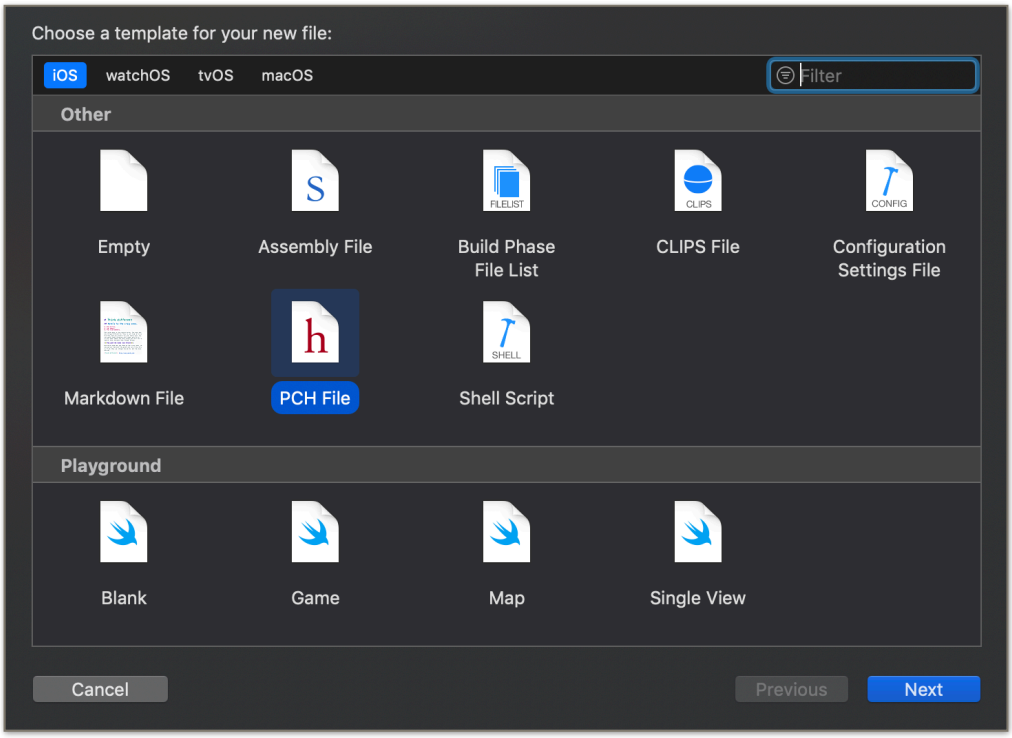
Step 7: You should now have 3 new files in your project folder: **OpenCVWrapper.h, OpenCVWrapper.m** and **MyProject-Bridging-Header.h**. Click on **MyProject-Bridging-Header.h** and add `#import "OpenCVWrapper.h"`.



Step 8: Rename **OpenCVWrapper.m** to **OpenCVWrapper.mm** and add `#import <opencv2/opencv.hpp>` at the top of the file.



Step 9: Go to **File -> New -> File…** and select **PCH file** under the **Other** tab, then press **next**. Simply leave the file named **PrefixHeader.pch** and press **Create**.

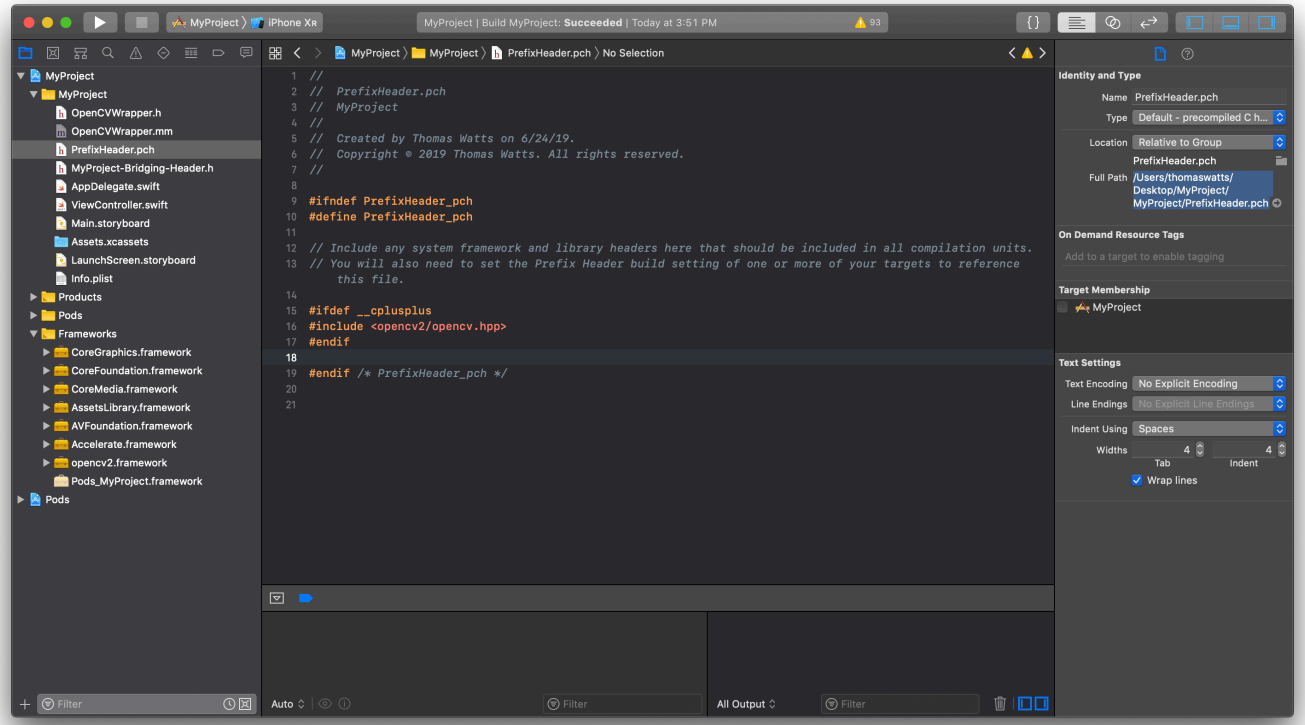


Step 10: Add the following lines of code to the top of **PrefixHeader.pch**:

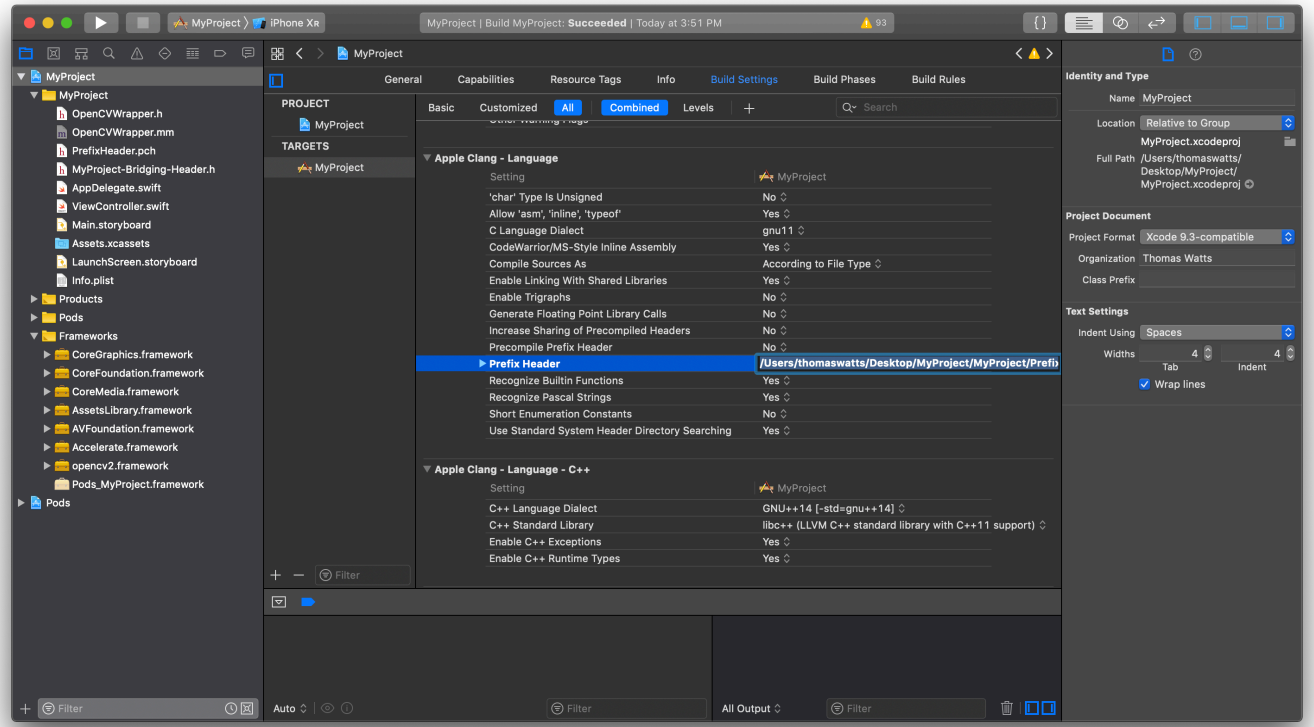
```
#ifdef __cplusplus

#include <opencv2/opencv.hpp>
#endif
```

```
1  //
2  //  PrefixHeader.pch
3  //  MyProject
4  //
5  //  Created by Thomas Watts on 6/24/19.
6  //  Copyright © 2019 Thomas Watts. All rights reserved.
7  //
8
9  #ifndef PrefixHeader_pch
10 #define PrefixHeader_pch
11
12 // Include any system framework and library headers here that should be included in all compilation units.
13 // You will also need to set the Prefix Header build setting of one or more of your targets to reference
       this file.
14
15 #ifdef __cplusplus
16 #include <opencv2/opencv.hpp>
17 #endif
18
19 #endif /* PrefixHeader_pch */
20
21
```

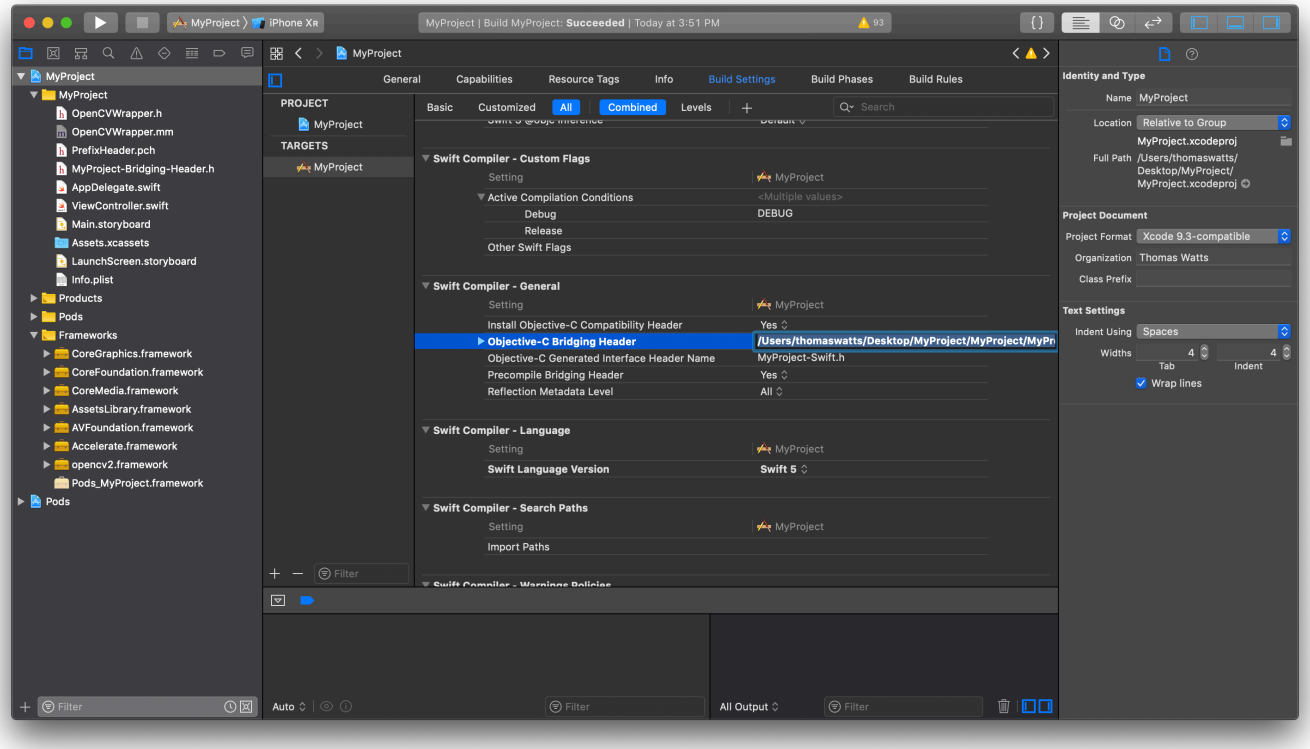
Step 10: We need to make sure Xcode can access **PrefixHeader.pch** and **MyProject-Bridging-Header.h** correctly. First copy the **Full Path** of the **PrefixHeader.pch** from the right side panel under **Identity and Type**.



Step 11: Paste copied **Full Path** into the **Prefix Header** section of your project's **Build Settings**.



Step 12: Repeat Step 10 for **MyProject-Bridging-Header.h** and copy the file's **Full Path**. Then paste the **Full Path** into the **Objective-C Briding Header** section of your project's **Build Settings**.

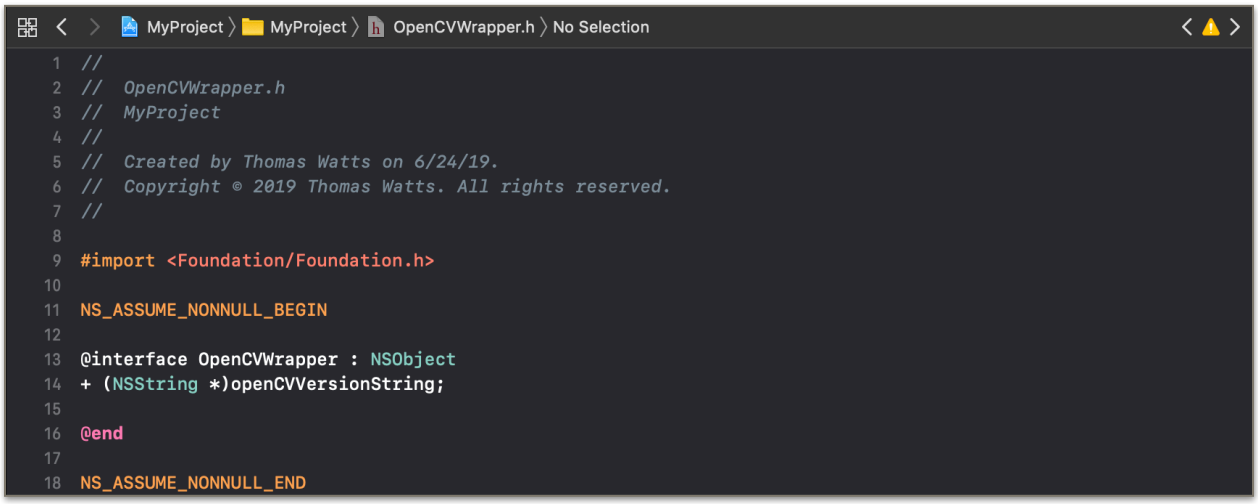


Step 13: In the **OpenCVWrapper.mm** file, paste the following code below `@implementation OpenCVWrapper`:

```
+(NSString *)openCVVersionString {
return [NSString stringWithFormat:@"OpenCV Version %s",  CV_VERSION];
}
```

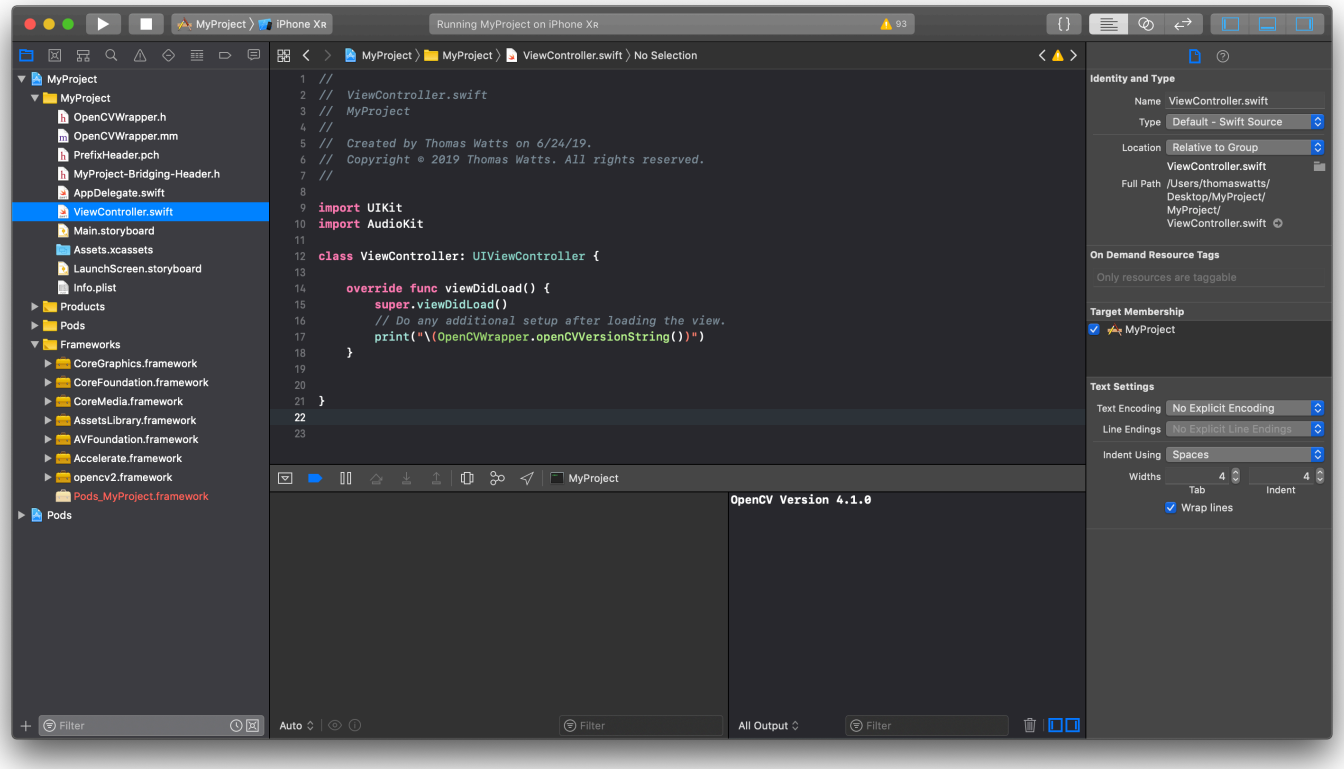
Step 14: In the **OpenCVWrapper.h** file, paste the following code below **`@Interface OpenCVWrapper:NSObject`**:

```
+ (NSString *)openCVVersionString;
```



Step 15: Now we are fully installed and only need to check that we've done it correctly. Navigate to your project's **View Controller**. In the body of the **viewDidLoad( )** function, paste the following code:

```
print("\(OpenCVWrapper.openCVVersionString())")
```



Now run the Xcode project (Press ▶ play button).
You should see "`OpenCV Version [your version number]`" pop up in the terminal.
Congratulations, you have successfully installed OpenCV!

Guide to the Code Behind SonicAtomic

Anatomy of Sonic Atomic

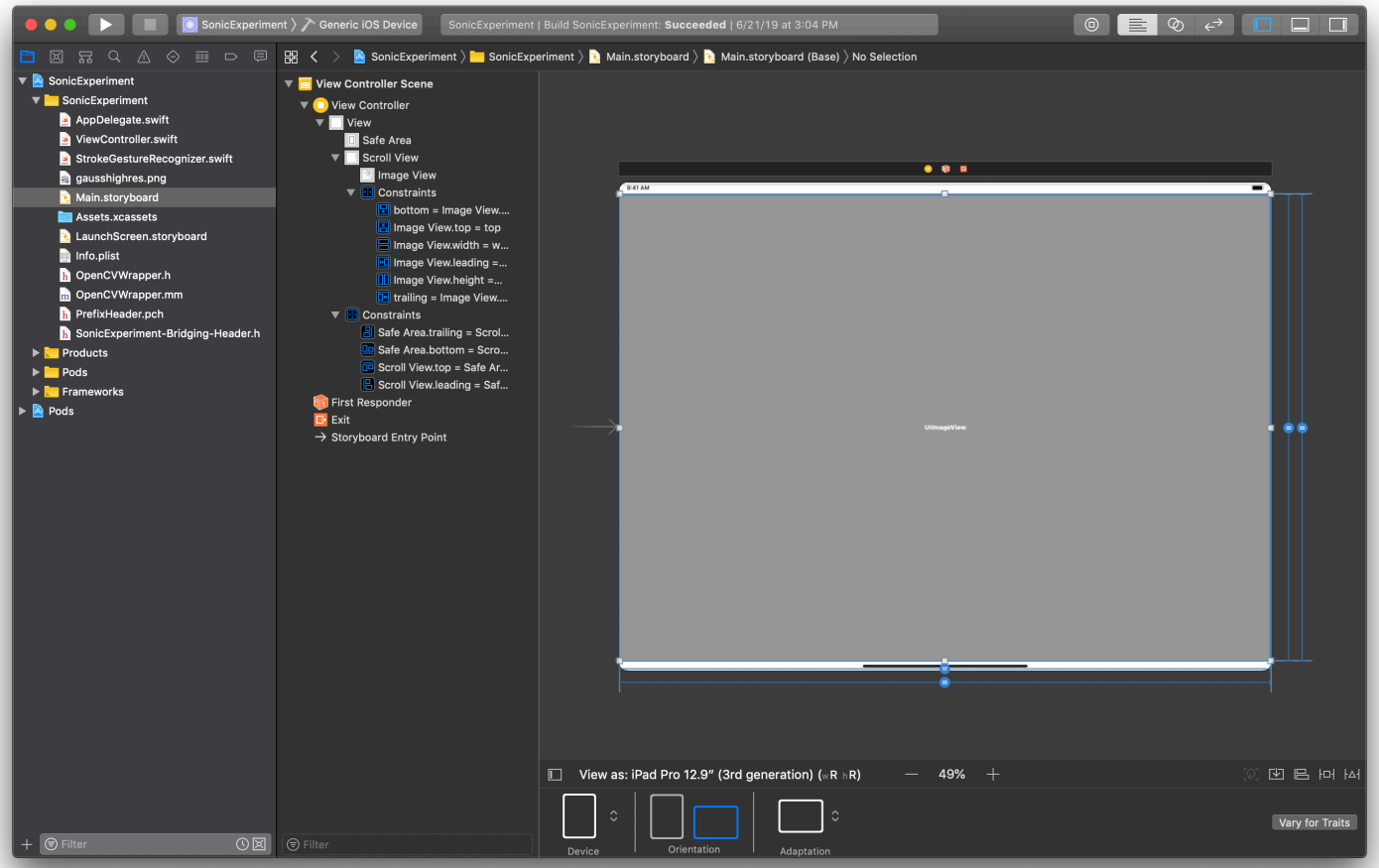
The SonicAtomic Xcode Project has 4 main sections: The Storyboard, The View Controller, The Apple Pencil Gesture Recognizer, The OpenCV Files. The Storyboard information is kept in the **Main.storyboard** file. In this file, you may configure how much of the iPad's screen are allocated to display images and icons as well as set up the app's response to touch input.

The View Controller is where most of the magic happens and is contained in the **ViewController.swift** file. In this file, you are able to dictate what the app does when it is loaded initially as well as set up how the app responds to user input. This file contains the information about how the app generates the sound that the user hears.

The Apple Pencil Gesture Recognizer is contained within the **StrokeGestureRecognizer.swift** file. This file is a custom gesture recognizer that allows the app to receive input from the Apple Pencil.

The OpenCV files allow you to access the capabilities of OpenCV. The files that allow you to access OpenCV are **OpenCVWrapper.h** and **OpenCVWrapper.mm**; this is where you write the code to which OpenCV responds. OpenCV is written in **Objective-C** hence the necessity for these additional files. Don't worry if you don't have any experience with **Objective-C**, anything you will need to do with OpenCV is online and the OpenCV function that I wrote for SonicAtomic should give you a good idea of the type of coding you may do when developing the project further. When in doubt, go to StackOverflow.

The Storyboard



There are 3 layers to the Storyboard: the **View**, the **Scroll View**, and the **Image View**. The **View** represents what is on the app's screen, everything that happens on the iPad's screen, happens inside the **View**. The **Scroll View** is inside the **View** and allows the user to pinch to zoom in on the image displayed on the screen and pan around the image (e.g. drag their finger across the screen to move the image around). The **Image View** is inside the **Scroll View** and allows us to display an image on the iPad's screen such that the user may interact with it through touch input (e.g. pan and pinch). Both the **Scroll View** and **Image View** have constraints on them which are setting that allow you to make sure the contents of the screen, like an image, stay in a specified location and don't move around based on screen size or orientation. The blue and white **=** equals icons that appear below and to the right of the storyboard indicate that the **Scroll View** and **Image View** are constrained to always have equal dimensions. Furthermore, the **Scroll View** and **Image View** are constrained to always be centered on the iPad's screen.

The View Controller

The storyboard is meant to be an outline for how the app interacts with the iPad's screen. There is no code in the storyboard itself, but rather objects from the storyboard are linked to code in the View Controller. The View Controller is where the majority of the app's functionality is programmed and is therefore more complicated in terms of its anatomy. I will break down the View Controller into the various functions and variables it contains.

The Stroke Gesture Recognizer

The Stroke Gesture Recognizer is a custom gesture recognizer that allows the iPad to recognize Apple Pencil input and do something in response to that input. The Stroke Gesture Recognizer is a blueprint for detection of and response to Apple Pencil input, it doesn't perform these functions unless we initialize it (or create an instance of it) in the **View Controller**.

```swift
//
//  StrokeGestureRecognizer.swift
//  SonicExperiment
//
//  Created by Thomas Watts on 6/11/19.
//  Copyright © 2019 Thomas Watts. All rights reserved.
//

import UIKit
import UIKit.UIGestureRecognizerSubclass

/// A custom gesture recognizer that receives Apple Pencil touch input
class StrokeGestureRecognizer: UIGestureRecognizer {
    //Variable to determine whether gesture recognizer should recieve Apple Pencil touch input or finger touch input
    //A feature intended to allow for finger touch input support to be used in future versions
    var isForPencil: Bool = true {
        didSet {
            if isForPencil {
                allowedTouchTypes = [UITouch.TouchType.pencil.rawValue as NSNumber]
            } else {
                allowedTouchTypes = [UITouch.TouchType.direct.rawValue as NSNumber]
            }
        }
    }

    //Class Variables
    var coordinateSpaceView: UIView?
    var size : CGSize!
    var origin : CGPoint!
    var imageSize : CGSize!
    var touchLocation : CGPoint!
    var lastTouch : CGPoint!
    var touchInImage : Bool = false

```

Line 9 & 10: In order to gain access to certain data types, such as **CGPoint**, as well as UI specific functions and register our custom gesture recognizer as a subclass of **UIGestureRecognizer** we must import **UIKit** and **UIKit.UIGestureRecognizerSubclass**.

```swift
//
//  StrokeGestureRecognizer.swift
//  SonicExperiment
//
//  Created by Thomas Watts on 6/11/19.
//  Copyright © 2019 Thomas Watts. All rights reserved.
//

import UIKit
import UIKit.UIGestureRecognizerSubclass

/// A custom gesture recognizer that receives Apple Pencil touch input
class StrokeGestureRecognizer: UIGestureRecognizer {
    //Variable to determine whether gesture recognizer should recieve Apple Pencil touch input or finger touch input
    //A feature intended to allow for finger touch input support to be used in future versions
    var isForPencil: Bool = true {
        didSet {
            if isForPencil {
                allowedTouchTypes = [UITouch.TouchType.pencil.rawValue as NSNumber]
            } else {
                allowedTouchTypes = [UITouch.TouchType.direct.rawValue as NSNumber]
            }
        }
    }

    //Class Variables
    var coordinateSpaceView: UIView?
    var size : CGSize!
    var origin : CGPoint!
    var imageSize : CGSize!
    var touchLocation : CGPoint!
    var lastTouch : CGPoint!
    var touchInImage : Bool = false

```

Line 13: On this line we specify the name of our custom class and specify the class as a subclass of **UIGestureRecognizer.**

Line 16-24: This **Boolean** value allows us to specify whether our custom gesture recognizer reacts to input from the Apple Pencil or finger touch input. While our custom gesture recognizer is only set up to receive Apple Pencil input, I left the option open to add support for finger touch input in future versions of app.

Line 27-33: These are the **Class Variables** that are filled with the appropriate data from within the **View Controller**. These variables are used by the **Touch Tracking Functions** that I will address next.

- `coordinateSpaceView` is a **UIView** object, this variable holds information about the **Image View** that is addressed in the Storyboard and View Controller sections. This variables allows us to get the location of the Apple Pencil touch input within the **Image View**.
- `size` is a **CGSize** object and holds information about the size of the image **after** being scaled to fit the iPad's screen.

- `origin` is a **CGPoint** object and holds information about the where the top left corner of the scaled image is within the iPad's screen. **(0,0)** would be the top left-most corner of the iPad's screen.
- `imageSize` is a **CGSize** object and holds information about the size of the image **before** being scaled to fit the iPad's screen.
- `touchLocation` is a **CGPoint** object and holds the location of the Apple Pencil touch input **within the image** being displayed on the screen once a touch input has been registered. It is important to note that the is **not** the location of the Apple Pencil touch input within the screen, this location is associated to pixels of the image. **(0,0)** would be the top left-most pixel of the image once displayed on the screen. This variable is filled by the **Touch Tracking Functions**.
- `lastTouch` is a **CGPoint** object and holds the location of the Apple Pencil touch input **within the screen**. Unlike `touchLocation`, this location is with reference to the iPad's screen, for reference, **(0,0)** would be the top left-most corner of the iPad's screen. This variable is also filled by the **Touch Tracking Functions**.
- `touchInImage` is a **boolean** variable that tracks whether or not the Apple Pencil input was registered within the boundary of the image once displayed on the screen. The purpose of this variable is to communicate with the **View Controller** and make sure that sound is only produced when the user is touching the Apple Pencil within the borders of the scaled image.

```
35
36      //Touch Tracking Functions
37      override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
38          guard let touch = touches.first else {
39              return
40          }
41          //Find point where pencil is touching with respect to the input image
42          pencilTouchHandler(touch: touch)
43          //Run Action Method in View Controller now that user has began stroke so to start producing sound
44          state = .began
45
46      }
47
48      override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
49          guard let touch = touches.first else {
50              return
51          }
52          //Find point where pencil is touching with respect to the input image
53          pencilTouchHandler(touch: touch)
54
55          if state == .began {
56              //Run Action Method in View Controller if stroke has began so to update sound produced
57              state = .changed
58          }
59      }
60
61      override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
62          //Reset touchInImage boolean
63          touchInImage = false
64          //Run Action Method in View Controller if stroke has ended in order to silence oscillator
65          state = .ended
66      }
```

`override func touchesBegin:`

This function first checks that the touch input that triggered the calling of this function is the first of the touches recorded in the array **`touches`**, this is generic code that is found in most implementations of the **`touchesBegin`** function. The function then calls the helper function **`pencilTouchHandler`**. I address this helper function below. Finally, the function changes the **state** of the gesture recognizer to **began**. The state of the gesture recognizer allows us to keep track of the stages of an Apple Pencil touch input. **Began** is the state representing the first instance of an Apple Pencil input.

`override func touchesMoved:`

This function is called when an initial Apple Pencil touch input has already been detected and the Apple Pencil has been moved along the screen without being lifted away. This function mirrors the **`touchesBegin`** function with the only difference being that **`touchesMoved`** changes the state of the gesture recognizer to **changed** in order to keep track of the fact that the Apple Pencil has been dragged across the screen but has yet to be lifted off the screen.

`override func touchesEnded:`

This function is called once the Apple Pencil has been lifted off the screen. The function then resets the **boolean** variable **`touchInImage`** to **false** and changes the state of the gesture recognizer to **ended**.

```
67
68      //Calculate point at which pencil is touching with respect to the input image
69      func pencilTouchHandler(touch: UITouch) {
70          //Get point at which user is touching
71          let currentPoint = touch.location(in: self.coordinateSpaceView)
72
73          //Excute if touch input is within the image
74          if currentPoint.x >= origin.x && currentPoint.y >= origin.y
75              && currentPoint.x < (origin.x + size.width-1)
76              && currentPoint.y < (origin.y + size.height-1){
77
78              //Track whether or not touch was within the boundaries of the image
79              touchInImage = true
80
81              //Calculate (x,y) coordinates with respect to input image
82              //Use ratio of original image dimensions
83              let xVal = (imageSize.width / size.width)*(currentPoint.x - origin.x)
84              let yVal = (imageSize.height / size.height)*(currentPoint.y - origin.y)
85
86              //Record Touch Location
87              touchLocation = CGPoint(x: xVal, y: yVal)
88              lastTouch = currentPoint
89          }
90      }
91
92
93
94  }
95
```

`func pencilTouchHandler:`

This function is called whenever the Apple Pencil is first touched to the screen as well as whenever the Apple Pencil is touched to the screen then dragged along the screen. The functions **`touchesBegin`** and **`touchesMoved`** record the **UITouch** object associated with the Apple Pencil's touch input and pass it through to this helper function. Then the location of that touch input with reference to the **Image View** (stored in the **`coordinateSpaceView`** variable) is recorded and stored in the **`currentPoint`** variable.

`pencilTouchHandler` then checks whether or not the touch input was within the bounds of the image displayed on the screen and only runs the rest of the code if the touch was registered within the bounds of the displayed image (Line 74-76). If the touch input was within the bounds of the displayed image, then the **boolean** variable **`touchInImage`** is set to **true**. Then the function initializes the variables **`xVal`** and **`yVal`** which hold the location of the Apple Pencil touch input with reference to the image being displayed. In order to obtain these values, we must take the ratio of the size of original image and the size of the scaled image. We then multiply this ratio to the location of touch input minus the location of the top left-most corner of the scaled image. This has the effect of translating the origin **(0,0)** so that it lies at the top left-most pixel of the scaled image.

`pencilTouchHandler` then records the location of the touch input with reference to the scaled image, **`touchLocation`**, and the location of the touch input with reference to the screen, **`lastTouch`**.

The OpenCV Files

The iPad encodes grayscale images as matrices whose entries correspond to each pixel's grayscale intensity value. The image that the first version of SonicAtomic was developed around is a 16-bit grayscale image meaning that for a given pixel of the image, the intensity of that pixel will be a number between **65535** and **0**. The intensity value **65535** corresponds to a **white** pixel and the intensity value **0** corresponds to a **black** pixel. All intensity values between **65535** and **0** correspond to some shade of gray. The current version of SonicAtomic only uses one OpenCV based function named **getGrayVal**. This function takes in two **Integer** values **i** and **j** that correspond to pixels of the image (e.g. entries of the associated matrix) and returns the grayscale intensity value of that particular pixel.

OpenCVWrapper.mm

```
1  //
2  //  OpenCVWrapper.mm
3  //  SonicExperiment
4  //
5  //  Created by Thomas Watts on 6/11/19.
6  //  Copyright © 2019 Thomas Watts. All rights reserved.
7  //
8
9  #import "OpenCVWrapper.h"
10 #import <opencv2/opencv.hpp>
11 @implementation OpenCVWrapper
12
13 + (UInt16) getGrayVal:(int)i :(int)j{
14     cv::Mat grayMat;
15     NSString *path = [[NSBundle mainBundle] pathForResource:@"gausshighres" ofType:@"png"];
16     const char * cpath = [path cStringUsingEncoding:NSUTF8StringEncoding];
17     grayMat = cv::imread(cpath,cv::IMREAD_ANYDEPTH);
18     return grayMat.at<UInt16>(i, j);
19 }
20 @end
```

Line 9-11: These lines of code are part of the OpenCV installation process (see Installing OpenCV) and allow OpenCV to respond to functions that we write in the **OpenCVWrapper.mm** file.

```
1  //
2  //  OpenCVWrapper.mm
3  //  SonicExperiment
4  //
5  //  Created by Thomas Watts on 6/11/19.
6  //  Copyright © 2019 Thomas Watts. All rights reserved.
7  //
8
9  #import "OpenCVWrapper.h"
10 #import <opencv2/opencv.hpp>
11 @implementation OpenCVWrapper
12
13 + (UInt16) getGrayVal:(int)i :(int)j{
14     cv::Mat grayMat;
15     NSString *path = [[NSBundle mainBundle] pathForResource:@"gausshighres" ofType:@"png"];
16     const char * cpath = [path cStringUsingEncoding:NSUTF8StringEncoding];
17     grayMat = cv::imread(cpath,cv::IMREAD_ANYDEPTH);
18     return grayMat.at<UInt16>(i, j);
19 }
20 @end
```

Line 13: Every function in **OpenCVWrapper.mm** starts with a **+** which indicates that we are writing a function.

- `(UInt16)` means that the **getGrayVal** function will return a value of type **UInt16**; the grayscale intensity values of pixels of the image are of this type.
- `getGrayVal` specifies the name of the function.
- `:(int)i :(int)j` means that the **getGrayVal** function takes in two **Integer** values **i** and **j** as inputs.

Line 14: Declares the variable **grayMat** which is an instance of OpenCV's matrix data type **cv::Mat**. Documentation on this data type can be found at https://docs.opencv.org/4.1.0/d3/d63/classcv_1_1Mat.html#a55ced2c8d844d683ea9a725c60037ad0.

Line 15: Once the app is running this line of code tells OpenCV to find the pathway to the image file called **gausshighres.png** contained in the Xcode project folder.

Line 16: This line of code converts the file pathway retrieved by line 15 and converts it into the proper data type for the function called on line 17.

When accessing data within the Xcode Project folder for OpenCV to process, use the same code I used in line 15 and line 16

```
1  //
2  //  OpenCVWrapper.mm
3  //  SonicExperiment
4  //
5  //  Created by Thomas Watts on 6/11/19.
6  //  Copyright © 2019 Thomas Watts. All rights reserved.
7  //
8
9  #import "OpenCVWrapper.h"
10 #import <opencv2/opencv.hpp>
11 @implementation OpenCVWrapper
12
13 + (UInt16) getGrayVal:(int)i :(int)j{
14     cv::Mat grayMat;
15     NSString *path = [[NSBundle mainBundle] pathForResource:@"gausshighres" ofType:@"png"];
16     const char * cpath = [path cStringUsingEncoding:NSUTF8StringEncoding];
17     grayMat = cv::imread(cpath,cv::IMREAD_ANYDEPTH);
18     return grayMat.at<UInt16>(i, j);
19 }
20 @end
```

Line 17: This line of code converts the image **gausshighres.png** into a matrix of type **cv::Mat** named **grayMat** using the method **`cv::imread`** that takes the file pathway found in line 15. The other parameter of **`cv::imread`**, **`cv::IMREAD_ANYDEPTH`** tells OpenCV to maintain the type of the grayscale intensity values as **UInt16**. Documentation on this function can be found at https://docs.opencv.org/4.1.0/d4/da8/group__imgcodecs.html#gga61d9b0126a3e57d9277ac48327799c80a0b486c93c25e8a0b0712681bb7254c18

Line 18: This line of code returns the grayscale intensity value of the desired pixel by accessing the **(i,j)**-th entry of the **grayMat** matrix using the **`.at<UInt16>(i,j)`** function. The relevant documentation can be found at https://docs.opencv.org/4.1.0/d3/d63/classcv_1_1Mat.html#a55ced2c8d844d683ea9a725c60037ad0.

OpenCVWrapper.h

```
1  //
2  //  OpenCVWrapper.h
3  //  SonicExperiment
4  //
5  //  Created by Thomas Watts on 6/11/19.
6  //  Copyright © 2019 Thomas Watts. All rights reserved.
7  //
8
9  #import <Foundation/Foundation.h>
10
11 NS_ASSUME_NONNULL_BEGIN
12
13 @interface OpenCVWrapper : NSObject
14 + (UInt16) getGrayVal:(int)i :(int)j;
15 @end
16
17 NS_ASSUME_NONNULL_END
```

Once you have written a function in **OpenCVWrapper.mm**, you must add the code written on Line 13 of **OpenCVWrapper.mm** in order to gain access to function in other parts of your Xcode project. Suppose you just wrote the function `+ (date_type) myFunction:(data_type) x { //Function Code }` in **OpenCVWrapper.mm**, then you must add `+ (data_type) myFunction:(data_type) x;` below `@interface OpenCVWrapper : NSObject` but before `@end`.