Developing Applications for iOS

Lecture 10:
Managing and Storing Data

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

- Property Lists
- Archiving Objects
- Filesystem Storing
- SQLite
- Closures (recap)
- Grand Central Dispatch
- URL Requests

Property Lists

Persistence

- How to make things stick around between launchings of your app (besides `NSUserDefaults`)

Property Lists

- A Property List is any graph of objects containing only the following classes: `NSArray`, `NSDictionary`, `NSNumber`, `NSString`, `NSDate`, `NSData`.
- Use the following methods in `NSArray` or `NSDictionary`:

```
func write(to url: URL, atomically: Bool) -> Bool
init?(contentsOf url: URL)
```

- Or `NSUserDefaults` if appropriate.
- Also `NSPropertyListSerialization` converts Property Lists to/from `NSData`.

Archiving

There is a mechanism for making **any** object graph persistent

- Not just graphs with `NSArray` or `NSDictionary` (or other Foundation classes) in them.
- For example, the view hierarchies you build in Interface Builder.

  Those are obviously graphs of very complicated objects.
- Requires all objects in the graph to implement `NSCoding` protocol:

```
func encode(with aCoder: NSCoder)
init?(coder aDecoder: NSCoder)
```

- It is extremely unlikely you will use this in this course. Certainly not during the labs.
- There are other, simpler, (or more appropriate), persistence mechanisms that we are about to discuss.

Archiving

- Object graph is saved by sending all objects `encode(with:)`.

```
func encode(with aCoder: NSCoder)
{
    super.encode(with: coder)
    aCoder.encode(2.0, forKey: "scale")
    aCoder.encode(CGPoint(x: 0, y: 0), forKey: "origin")
    aCoder.encode(object, forKey: "object")
}
```

Absolutely, must call `super`'s version or your superclass's data won't get written out!

- Object graph is read back in with `init(coder:)`.

```
required init?(coder aDecoder: NSCoder)
{
    super.init(with: aDecoder)
    scale = aDecoder.decodeFloat(forKey: "scale")
    object = aDecoder.decodeObject(forKey: "object")
    origin = aDecoder.decodeCGPoint(forKey: "origin")
    // notice that the order does not matter
}
```

Archiving

`NSKeyed{Un}Archiver` classes are used to store/retrieve graph

- Storage and retrieval is done to `NSData` objects.
- `NSKeyedArchiver` stores an object graph to an `NSData`:

```
class func archivedData(withRootObject rootObject: Any)->Data
```

- `NSKeyedUnarchiver` retrieves an object graph from an `NSData`:

```
class func unarchiveObject(with data: Data) -> Any?
```

What do you think this code does?

```
var obj = ...;
var data = NSKeyedArchiver.archivedData(withRootObject: obj)
var copy = NSKeyedUnarchiver.unarchiveObject(with: data)
```

- It makes a "deep copy" of `object`. But beware, you may get more or less than you expect. Object graphs like "view hierarchies" can be very complicated.

App Sandbox

- Your application can see the iOS file system like a normal Unix file system.
- It starts with the root directory: /.
- There are file protections, of course, like normal Unix, so you can’t see everything.

You can only write inside your “Sandbox”. Why?

- **Security** - so no one else can damage your application.
- **Privacy** - so no other applications can view your application’s data.
- **Cleanup** - when you delete an application, everything its ever written goes with it.

App Sandbox

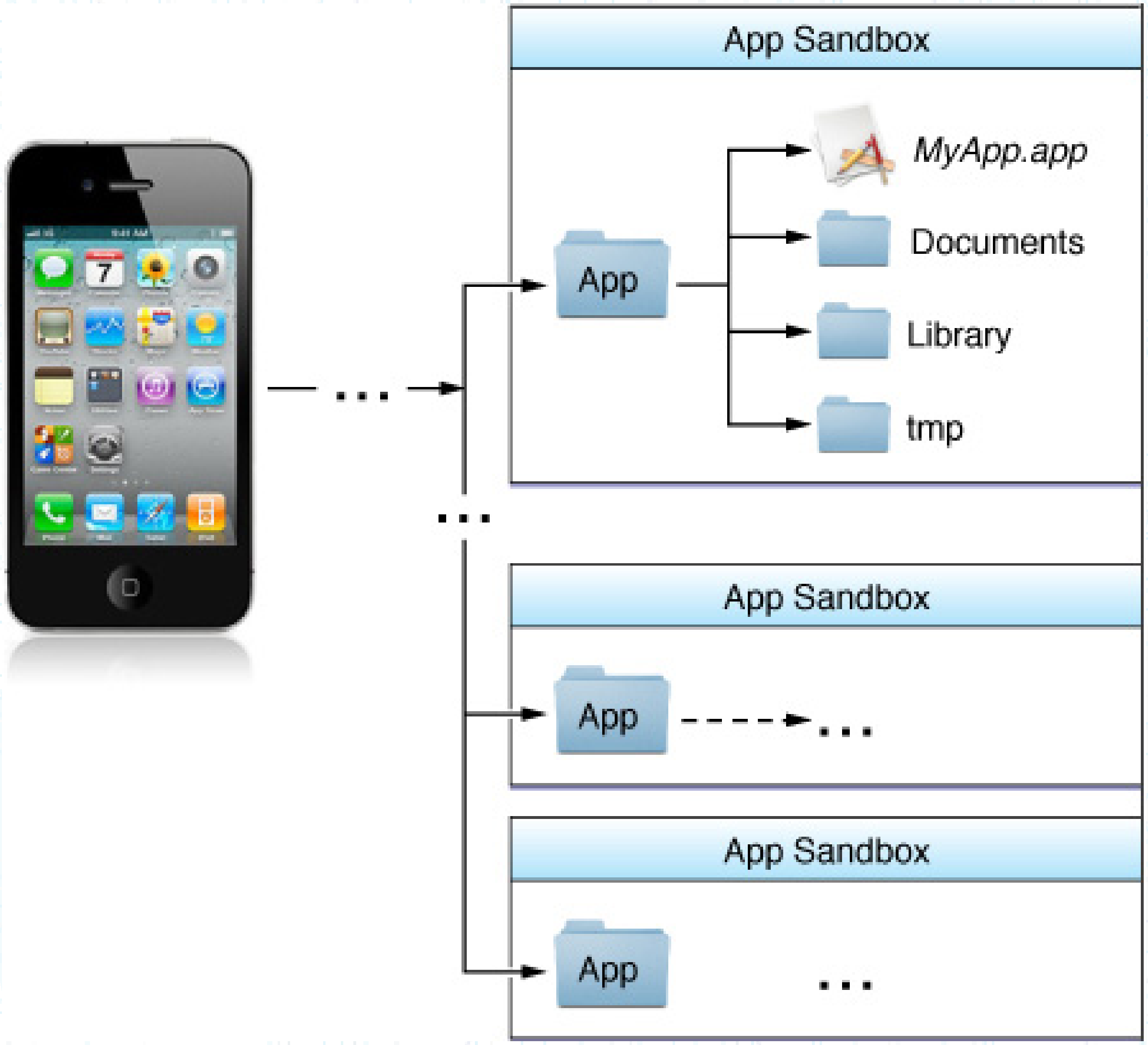
The App “Sandbox”

- As part of the sandboxing process, the system installs your app in its own sandbox directory. It acts as the home for the app and its data.

What’s in this “Sandbox”

- Application bundle directory (binary, .storyboards, .PNGs, etc.). This directory is not writeable.
- Documents directory. This is where you store permanent data created by the user.
- Caches directory. Store temporary files here (this is not backed up by iTunes).
- Other directories (check out `NSSearchPathDirectory` in the documentation).

App Sandbox


App Sandbox
App
MyApp.app
Documents
Library
tmp
App Sandbox
App
App Sandbox
App

File System

What if you want to write to a file you ship with your app?

- Copy it out of your application bundle into the documents (or other) directory to make it writeable.

How do you get the paths to these special sandbox directories?

- Use this `FileManager` method:

```
func urls(for directory: FileManager.SearchPathDirectory,
     in domainMask: FileManager.SearchPathDomainMask) -> [URL]
// domainMask is usually NSUserDomainMask
```

- Notice that it returns an `Array` of paths (not a single path).

  Since the file system is limited in scope, there is usually only one path in the array in iOS. No user home directory, no shared system directories (for the most part), etc. Thus you will almost always just use `lastObject` (for simplicity).

- Examples of `SearchPathDirectory` values: `documentDirectory`, `cachesDirectory`, `autosavedInformationDirectory`, etc.

NSFileManager

- Provides utility operations (reading and writing is done via `Data`).
- Check to see if files exist; create and enumerate directories; move, copy, delete, replace files.
- Thread safe (as long as a given instance is only ever used in one thread).
- Just `init` an instance and start performing operations. If you don't use the `delegate` you can use the `default` one:

```
var manager = FileManager.default

func createDirectory(atPath path: String,
     withIntermediateDirectories createIntermediates: Bool,
     attributes: [String : Any]? = nil) throws
func fileExists(atPath path: String) -> Bool
func isReadableFile(atPath path: String) -> Bool
func contentsOfDirectory(atPath path: String)
     throws -> [String]
```

- Has a delegate with lots of “should” methods (to do an operation or proceed after an error). Check out the documentation.

File System

NSString

- Path construction methods and reading/writing strings to files:

```
var lastPathComponent: String { get }

class  func  path(withComponents  components:  [String])  ->
String

func appendingPathComponent(_ str: String) -> String

var deletingPathExtension: String { get }

func write(toFile path: String,
           atomically useAuxiliaryFile: Bool,
           encoding enc: UInt) throws
// encoding can be ASCII, UTF-8, etc.

init(contentsOfFile path: String,
     usedEncoding enc: UnsafeMutablePointer<UInt>?) throws
```

- And plenty more. Check out the documentation.

SQLite

SQL in a single file

- Fast, low memory, reliable.
- Open Source, comes bundled in iOS.
- Not good for everything (e.g. not video or even serious sounds/images).
- Not a server-based technology (not great at concurrency, but usually not a big deal on a phone).
- Used in countless applications across many platforms, SQLite is considered a standard for lightweight embedded SQL database programming.
- SQLite is a C API, so you need a Bridging Header in order to use it from Swift (not going into details, but you should look at this).

Closures

- Closures are self-contained blocks of functionality that can be passed around and used in your code.
- A closure is a sequence of statements inside `{}`.
- Usually included in-line with the calling of method that is going to use the block of code.
- Very smart about local variables, referenced objects, etc. The closure is able to make use of variables from the same scope in which it was defined.
- Using closures in your iOS (and Mac) applications allows you to attach arbitrary code to Apple-provided methods.
- Similar in concept to delegation, but passing short in-line blocks of code to methods is often more convenient and elegant.

Closures

Functions are actually special cases of closures. Closures take one of three forms:

- Global functions are closures that have a name and do not capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Closures

- Swift's standard library provides a method called `sorted(by:)`, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. In the following example, we are sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`:

```
let names = ["Radu", "Alex", "Eva", "Dan"]
func backward(_ s1:, _ s2: String) -> Bool {
   return s1 > s2
}
var reversed = names.sorted(by: backward)
```

- `// reversed is ["Radu", "Eva", "Dan", "Alex"]`

Closures

- The example below shows a closure expression version of the `backward(_:_:)` function from the previous slide:

```
reversed = names.sorted(by: {
   (s1: String, s2: String) -> Bool in
   return s1 > s2
})
```

- Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. Because all of the types can be inferred, the return arrow (->) and the parentheses around the names of the parameters can also be omitted:

```
reversed = names.sorted(by: { s1, s2 in
   return s1 > s2
})
```

Closures

- Single-expression closures can implicitly return the result of their single expression by omitting the `return` keyword from their declaration, as in this version of the previous example:

```
reversed = names.sorted(by: { s1, s2 in s1 > s2 })
```

- Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure’s arguments by the names `$0`, `$1`, `$2`, and so on.

- If you use these shorthand argument names within your closure expression, you can omit the closure’s argument list from its definition. The `in` keyword can also be omitted:

```
reversed = names.sorted(by: { $0 > $1 })
```

Closures

- If you need to pass a closure expression to a function as the function’s final argument and the closure expression is long, it can be useful to write it as a trailing closure instead:

```
func someFunction(closure: () -> Void) {
    // function body goes here
}

// Calling without using a trailing closure:
someFunction(closure: {
    // closure's body goes here
})

// Calling with a trailing closure instead:
someFunction() {
    // trailing closure's body goes here
}
```

Closures

- The string-sorting closure from the previous slides can be written outside of the `sorted(by:)` method’s parentheses as a trailing closure:

```
reversed = names.sorted() { $0 > $1 }
```

- If a closure expression is provided as the function or method’s only argument and you provide that expression as a trailing closure, you do not need to write a pair of parentheses after the function:

```
reversed = names.sorted { $0 > $1 }
```

Closures

- A closure can capture constants and variables from the surrounding context in which it is defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists:

```swift
func makeAdder(amount: Int) -> () -> Int {
    var runningTotal = 0
    func adder() ->  Int {
        runningTotal += amount
        return runningTotal
    }
    return adder
}
let addTen = makeAdder(amount: 10)
addTen() // returns a value of 10
addTen() // returns a value of 20
```

Closures

Creating a type for a variable that can hold a closure

- Closures are kind of like “objects” with an unusual syntax for declaring variables that hold them.
- Usually if we are going to store a block in a variable, we `typealias` a type for that variable:

```
typealias PredicateHandler = ((Int) -> Bool)
```

There is an `Int` argument.

Returns a `Bool`.

- This declares a type called `PredicateHandler` for variables which can store a closure. Specifically, a closure which takes an `Int` as its only argument and returns a `Bool`.

Closures

Creating a type for a variable that can hold a block

- Then we could declare a variable of this type and give it a value:

```
var isEven: PredicateHandler = {value in
    return value % 2 == 0
}
```

- And then use the variable `isEven` like this:

```
var newRoot = cleanTree(head: root,
                        predicate: isEven)
```

- You don’t have to `typealias`, for example, the following is also a legal way to create `isEven`:

```
var isEven: ((Int) -> Bool) = {value in
    return value % 2 == 0
}
```

Memory Cycles (a bad thing)

- What if you had the following property in a class?

```
var predicate: PredicateHandler?
```

- And then tried to do the following in one of that class's methods?

```
self.predicate = {value in
    return value % self.divisor == 0
}
```

- We said that all objects referenced inside a block will stay in the heap as long as the closure does. In other words, closures keep a `strong` pointer to all objects referenced inside of them.
- In this case, `self` is an object reference in this block. Thus the block will have a `strong` pointer to `self`.
- But notice that `self` also has a strong pointer to the block (through its `predicate` property)!
- **This is a serious problem.** Neither `self` nor the block can ever escape the heap now.
- That's because there will always be a `strong` pointer to both of them (each other's pointer). This is called a memory "cycle".

Memory Cycles Solution

- You'll recall that local variables are always `strong`.
- That's okay because when they go out of scope, they disappear, so the `strong` pointer goes away.
- But there's a way to declare that a local variable is `unowned/weak`:

```
self.predicate = {[unowned self] value in
    return value % self.divisor == 0
}
```

- This solves the problem because now the closure only has an `unowned` pointer to `self`.
- Note that `self` still has a `strong` pointer to the closure, but that's okay.
- As long as someone in the universe has a `strong` pointer to this `self`, the closure's pointer is good. And since the closure will not exist if `self` does not exist (since `predicate` won't exist), all is well!

Closures

A closure is an anonymous inline collection of code that:

- Has a typed argument list just like a function.
- Has an inferred or declared return type.
- Can capture state from the lexical scope within which it is defined.
- Can modify the state of the scope.
- Can share the potential for modification with other blocks defined within the same lexical scope.

Closures

When do we use closures in iOS?

- Enumeration
- View Animations (this is really nice and easy)
- Sorting (sort this thing using a block as the comparison method)
- Notification (when something happens, execute this block)
- Completion handlers (when you are done doing this, execute this block)
- And a super-important use: Multithreading

Using the Grand Central Dispatch (GCD) API.

Grand Central Dispatch

The basic idea is that you have queues of operations

- The operations are specified using blocks.
- Most queues run their operations serially (a true “queue”). We’re only going to talk about serial queues.

The system runs operations from queues in separate threads

- Though there is no guarantee about how/when this will happen.
- All you know is that your queue’s operations will get run (in order) at some point. The good thing about this is that if your operation blocks, only that queue will block.
- Other queues (like the main queue, where UI is happening) will continue to run.

So how can we use this to our advantage?

- Get blocking activity (e.g. network) out of our user-interface (main) thread. Do time-consuming activity concurrently in another thread.

Queue Types

GCD provides three main types of queues:

- Main queue: runs on the main thread and is a serial queue.
- Global queues: concurrent queues that are shared by the whole system. There are four such queues with different priorities: high, default, low, and background. The background priority queue is I/O throttled.
- Custom queues: queues that you create which can be serial or concurrent. These actually trickle down into being handled by one of the global queues.

Synchronous versus Asynchronous

- With GCD, you can dispatch a task either synchronously or asynchronously:

  1) A synchronous function returns control to the caller after the task is completed.

  2) An asynchronous function returns immediately, ordering the task to be done but not waiting for it. Thus, an asynchronous function does not block the current thread of execution from proceeding on to the next function.

Grand Central Dispatch

Some important functions in the GCD API

- Creating and releasing serial queues:

```
init(__label label: UnsafePointer<Int8>?,
     attr: __OS_dispatch_queue_attr?)
```

- Putting blocks in the queue:

```
func async(execute workItem: DispatchWorkItem)

func sync(execute workItem: DispatchWorkItem)

func asyncAfter(deadline: DispatchTime,
                qos: DispatchQoS = default,
                flags: DispatchWorkItemFlags = default,
                execute work: @escaping () -> Void)
```

- Getting the global or main queue:

```
class func global(qos: DispatchQoS.QoSClass = default)
                  -> DispatchQueue

class var main: DispatchQueue { get }
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
    do {
        let data = try Data(contentsOf: imageURL!)
        let image = UIImage(data: data)
        self.imageView!.image = image!
        self.scrollView!.contentSize = image!.size
    }
    catch { }
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{



      do {
         let data = try Data(contentsOf: imageURL!)
         let image = UIImage(data: data)
         self.imageView!.image = image!
         self.scrollView!.contentSize = image!.size
      }
      catch { }

}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
   let queue = DispatchQueue(label: "Image Downloader")


      do {
         let data = try Data(contentsOf: imageURL!)
         let image = UIImage(data: data)
         self.imageView!.image = image!
         self.scrollView!.contentSize = image!.size
      }
      catch { }

}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
    let queue = DispatchQueue(label: "Image Downloader")
    queue.async {

        do {
            let data = try Data(contentsOf: imageURL!)
            let image = UIImage(data: data)
            self.imageView!.image = image!
            self.scrollView!.contentSize = image!.size
        }
        catch { }
    }
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
    let queue = DispatchQueue(label: "Image Downloader")
    queue.async {

        do {
            let data = try Data(contentsOf: imageURL!)
            let image = UIImage(data: data)
            self.imageView!.image = image!
            self.scrollView!.contentSize = image!.size
        }
        catch { }
    }
}
```

- **Problem:** UIKit calls can only happen in the main thread!

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
    let queue = DispatchQueue(label: "Image Downloader")
    queue.async {

        do {
            let data = try Data(contentsOf: imageURL!)
            let image = UIImage(data: data)



              self.imageView!.image = image!
              self.scrollView!.contentSize = image!.size

        }
        catch { }
    }
}
```

Grand Central Dispatch

What does it look like to call these?

- Example: fetching an image from the network (this would be slow).

```
func viewWillAppear(animated: Bool)
{
    let queue = DispatchQueue(label: "Image Downloader")
    queue.async {

        do {
            let data = try Data(contentsOf: imageURL!)
            let image = UIImage(data: data)

            DispatchQueue.main.async {

              self.imageView!.image = image!
              self.scrollView!.contentSize = image!.size
            }
        }
        catch { }
    }
}
```

Performing URL Requests

`URLRequest`

```
init(url: URL,
     cachePolicy: URLRequest.CachePolicy = default,
     timeoutInterval: TimeInterval = default)
```

`URL`

- Basically like an `String`, but enforced to be “well-formed”.
- Examples: file://... or http://...
- In fact, it is the recommended way to specify a file name in the iOS API.

```
init?(string: String)
init(fileURLWithPath path: String, isDirectory: Bool)
```

`URLRequest.CachePolicy`

- Ignore local cache; ignore caches on the internet; use expired caches; use cache only (don’t go out onto the internet); use cache only if validated.

Performing URL Requests

`URLSession`

- API for downloading content.
- Provides a rich set of methods for supporting authentication and background downloads.

```
func dataTask(with request: URLRequest,
     completionHandler: @escaping (Data?, URLResponse?,
     Error?) -> Void) -> URLSessionDataTask
```

`URLSessionTask`

- Base class for tasks in a URL session.
- Tasks are always part of a session; you create a task by calling one of the task creation methods on an `URLSession` object.

```
func resume()
func cancel()
func suspend()
var state: URLSessionTask.State { get }
var priority: Float { get set }
```

JSON Serialization

- You use the `JSONSerialization` class to convert JSON to Foundation objects and convert Foundation objects to JSON.
- An object that may be converted to JSON must have the following properties:

  1. The top level object is an `Array` or `Dictionary`.
  2. All objects are instances of `String`, `Number`, `Dictionary`, `Array`.
  3. All dictionary keys are instances of `String`.
  4. Numbers are not NaN or infinity.

- From `Data` to JSON:

```
class func jsonObject(with data: Data,
     options opt: JSONSerialization.ReadingOptions = [])
     throws -> Any
```

- From JSON to `Data`:

```
class func data(withJSONObject obj: Any,
     options opt: JSONSerialization.WritingOptions = [])
     throws -> Data
```

Performing URL Requests

```
let url = URL(string: "http://test.com/restaurants.json")
let session = URLSession.shared
var request = URLRequest(url: url!,
              cachePolicy: .reloadIgnoringLocalCacheData,
              timeoutInterval: 90.0)
request.httpMethod = "GET"
let task = session.dataTask(with: request)
{(data, response, error) in
  if let error = error { print(error) }
  else if let response = response, let data = data
  {
    let httpResponse = response as! HTTPURLResponse
    print("Status Code = \(httpResponse.statusCode)")

    do {
      let json = try JSONSerialization.jsonObject(with: data,
                 options: [.allowFragments, .mutableContainers])

      let dict = json as! [String: Any]
      print("\(dict)")
    }
    catch { print("The received JSON is not well-formatted.") }
  }
}
task.resume()
```

What should you study next?

- Core Data
- Notification Center
- Modal View Controllers
- Core Motion (gyro, magnetometer)

  Measuring the device’s movement.
- `UITextField`, `UITextView`, `UIActivityViewContoller`
- `UIView` Animation
- `UIImagePickerController`

  Getting images from the camera or photo library.
- `NSTimer`

  Perform scheduled tasks on the main thread.
- iPad and Universal Applications

  There are specific Navigation and View Controllers.
- Metal / Open GL ES