**100 days of Swift**

**CoreImage**

Super-fast and super-powerful framework from Apple, which applies filters to images. It makes use of a CIContext so be sure to import it and initialize it on the viewDidLoad(), you will also need a filter object, which is of type CIFilter.

Not all filters have the kCIInputImageKey key available, therefore, we need to check beforehand which keys are available. For this, we can use the contains method in the currentFilter.inputKeys property.

To save an image to the photo album, use the UIImageWriteToSavedPhotosAlbum method, which as a third parameter receives a callback, that should be a method to call when the image has been saved to the photo library, you can show an alert to confirm it was saved or if there was an error.

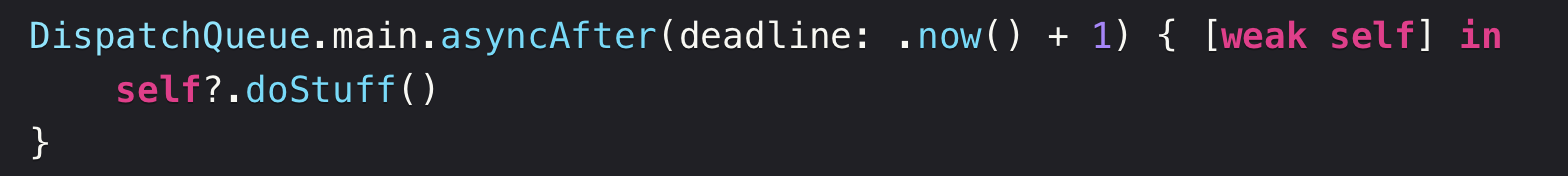
**SpriteKit**

Higher Y values in SpriteKit place nodes towards the top of the scene.

SKCropNode is a special kind of SKNode which uses an image as a cropping mask, anything in the colored part will be visible, anything in the transparent part will be invisible. In our case we used it to hide the penguins by having a crop mask shaped like the hole that makes the penguin invisible when it moves outside the mask.

We can change the image inside our penguin sprite by changing its .texture property. This takes a new class called SKTexture which takes an image as well, but it’s more efficient than creating new nodes.

To schedule some code to be executed after certain amount of time we can use Grand Central Dispatch (GCD) code asyncAfter() which is used to schedule a closure to execute after the time has been reached, it looks like this:



SKAction helps us create and manage actions, here some common methods:

* SKAction.wait(forDuration:) creates an action that waits for a period of time, measured in seconds.
* SKAction.run(block:) run any code we want, provided as a closure. "Block" is Objective-C's name for a Swift closure.
* SKAction.sequence() takes an array of actions and executes them in order. Each action won't start executing until the previous one finished.

You need to declare all the actions and then pass them to a SKSpriteNode object, calling charNode.run(sequence) where charNode is a SKSpriteNode object.

**Animation**

To animate we will be using the UIView.animate() method. We don’t need to use **[weak self]** because there’s no risk of strong reference cycles here.

CGAffineTransform represents a specific kind of transform that we can apply to any UIView object or subclass. You have to apply it to the **self**.imageView.transform property.

As a reminder, we can use **[unowned self]** in closures when we mean ”I know you want to capture **self** strongly so that it can be used later, but I want you not to have any ownership at all.”. As an alternative we could’ve written **[weak self]**, which could capture \_ in the closure as an optional, and we’d need to run **self?.**doStuff()… **self** doesn’t own **DispatchQueue.main**, so the reference will be destroyed once the closure finishes, even though it is recommended to always add **[weak self].**

**MapKit**

MapKit is Apple’s mapping framework, which handles fetching data and rendering, it handles scrolling and zooming, and it even handles placemarks, routes, and more.

You need to make a delegate of MapView the controller it belongs to, so be sure to control drag it to the controller and choose *delegate.*

Every time the map needs to show an annotation, it calls a **viewFor** method on its delegate.

**SpriteKit Part 2**

The method advanceSimulationTime() can simulate X seconds passing in the emitter, thus updating all the particles as if they were created X seconds ago. This can work for filling our screen with star particles.

Timer is responsible for running code after a period has passed, either once or repeatedly. It has five parameters: how many seconds you want the delay to be, what object should be told when the timer fires, what method should be called on that object when the timer fires, any context you want to provide, and whether the time should repeat.

**linearDamping** and **angularDamping** properties indicate if the movement and rotation will slow or not over time. Setting them up to 0 creates a frictionless space environment!

**Debugging**

Remember that print is a variadic function, which means that can receive any number of parameters. It can also receive separators as print(1, 2, 3, 4, 5, separator: "-") or terminators as print("Some message", terminator: " "), which will be added to the end.

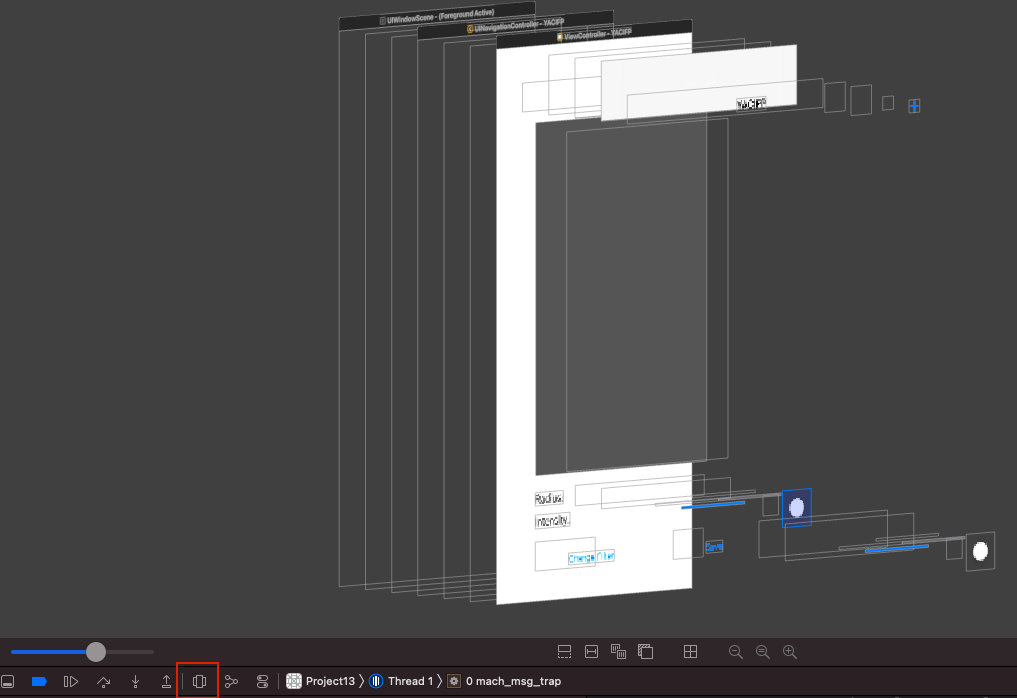
Assertions, which are debug-only checks that will force your app to crash if a specific condition isn't true. The assertions crashes only when debugging, once released, they are removed. This means you can set up an extremely strict environment while you’re developing, ensuring that all values are present and correct, without causing problems for real users assert(test\_that\_should\_pass, "Message")

To add a breakpoint, click on the number from the specific line of code you want to debug, then, when it is hit, you can step over line by line (▶) by pressing F6. There's another command called Continue (Ctrl+Cmd+Y) that means "continue executing my program until you hit another breakpoint."

The screen on the left side, which holds all the threads currently executing your app, is a back trace which can lead you to the problem.

Graphical user interface, text

Description automatically generatedYou can also print variables in the console by entering **p + name\_of\_var**. You can move the breakpoint bar to move the instruction pointer. Also, we can make breakpoints conditional, this by clicking “Edit Breakpoint”. Breakpoints can also be automatically triggered when an exception is thrown, saying “pause execution as soon as an exception is thrown” and you can examine. For this, press Cmd+8 or click the debugger menu, and then go to the bottom and add exception breakpoint.

Another debugging technique for views is to use the View Debugging, which is useful to debug views (that for some reason you cannot see for example).