Урок 5

Оптимизация UITableView и UICollectionView

Ищем проблемы в работе таблиц и коллекций. Отказываемся от AutoLayout в пользу верстки на фреймах. Обрабатываем данные в параллельном потоке, кэшируем изображения.

Проблемы производительности UI

При работе с мобильными приложениями можно заметить, что некоторые из них работают плавно, а другие – рывками. Первые выигрывают в глазах пользователя, а вторые зачастую отправляются в корзину, потому что притормаживают и заставляют ждать. Всем хочется, чтобы даже на старых устройствах приложения работали идеально.

К сожалению, многие инструменты Apple не дают необходимой плавности сами по себе. Их нужно правильно настраивать или заменять на более удобные.

В этом уроке мы не будем искать способы ускорить вычисления – как правило, такие проблемы индивидуальны. Поговорим о плавности пользовательского интерфейса (UI).

В центре внимания – UITableView и UICollectionView, так как обычные экраны редко испытывают проблемы с производительностью, а коллекции – регулярно.

Перечислим основные факторы замедления UI:

1. Выполнение в главном потоке действий, не касающихся UI.
2. Большое количество элементов на экране.
3. Лишние действия при подготовке элементов.
4. Сложные правила позиционирования элементов относительно друг друга.
5. Сложные элементы: текст с переносами и атрибутами (**Attributed string**), скругления, полупрозрачность, градиент; изображения, которые приходится сжимать или увеличивать; изображения неподходящей плотности (1x на экранах 2x, 3x).

Некоторые из них можно предотвратить, с другими легко справиться, а некоторые представляют сложную проблему. Рассмотрим все по порядку.

Главный поток – легкий, как перышко

Вычисления, которые выполняются быстро, называют легкими, а те, что требуют много времени – тяжелыми. Работа с UI, которая ведется в главном потоке, достаточно тяжелая. Поэтому дополнительно нагружать его не стоит.

Начинающих разработчиков пугает необходимость многопоточных вычислений: множество вызовов GCD или операций по всему коду, необходимость распределять операции между главной и глобальными очередями. Справиться с этими проблемами поможет правильная архитектура.

Как правило, работа над приложением начинается с дизайна и описания требований. Не имея опыта и ориентируясь на эти входные данные, можно пойти ошибочным путем: сконцентрироваться именно на экранах и отображаемой информации, забыв о бизнес-логике.

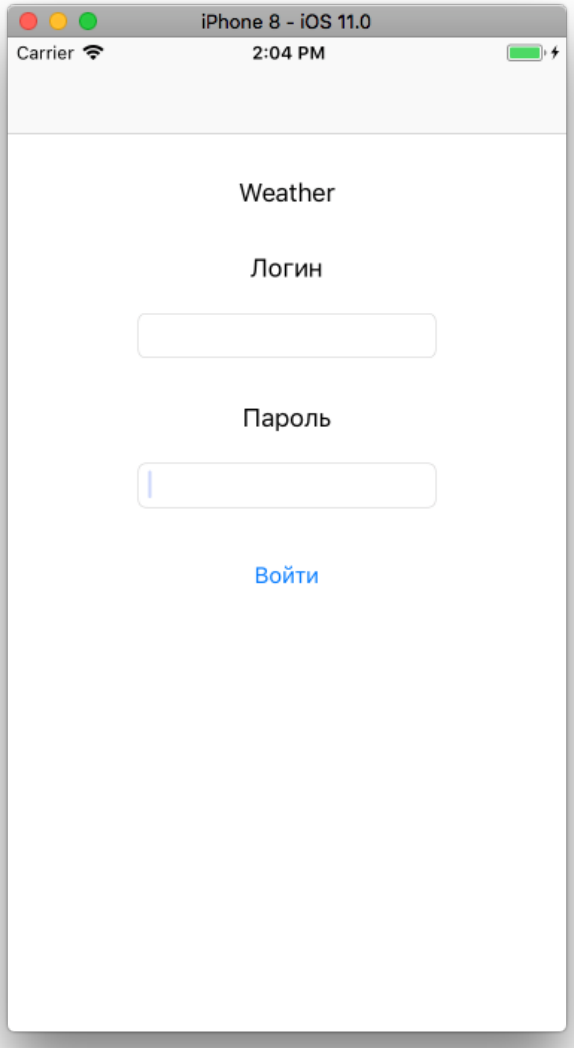
В этом случае работа начинается с создания экрана и контроллера для него, расстановки элементов. При этом используется либо Storyboard, где вносятся уточнения в коде контроллера, либо все целиком прописывается кодом в контроллере. Когда верстка готова, нужно обрабатывать действия пользователя на экране – и соответствующий код пишется в том же контроллере. Дополнительно могут прописываться блоки для получения данных из интернета, при кэшировании – для работы с базой данных, UserDefaults, файловой системой.

В итоге количество строк в классе контроллера может превышать несколько тысяч, в нем сложно разбираться и вносить изменения. А главное – весь этот код выполняется на главном потоке и выглядит как гигантский придаток к экрану.

Новички даже многопоточный запрос URLSession воспринимают как недоработку **Apple**. Именно поэтому **Alamofire** по умолчанию выполняет всю работу в главном потоке, хоть и асинхронно – но даже этого многие мечтают избежать. Это плохая практика и неправильная архитектура, получившая прозвище **massive view controller** – «огромный контроллер представления».

Чтобы избежать этого, надо четко проводить грань между пользовательским интерфейсом, бизнес-логикой и данными приложения. Экраны и все элементы, отображающиеся на них, – это только интерфейс, который необходим для предоставления информации пользователю.

Рассмотрим несколько примеров изображений и определим, что является интерфейсом, какие данные с ним связаны и какова бизнес-логика.



Это экран авторизации. К **интерфейсу** здесь относятся NavigationBar, UILabel`ы с надписями «Weather», «Логин», «Пароль», поля ввода UITextInput и кнопка «Войти». Тексты надписей в данном случае тоже являются частью UI, так как это константы, не зависящие от внешних факторов.

Данными в этом примере являются непосредственно логин и пароль. Их вводит пользователь и с ними вы работаете. Есть тонкий момент: логин и пароль как данные, и они же, введенные в поля – не одно и то же с точки зрения компонентов архитектуры. Есть элемент интерфейса – поле ввода «Пароль», и введенный в него текст, как часть этого поля, тоже принадлежит UI. А в коде можно обратиться к полю ввода и прочитать этот текст, в результате получая данные.

Бизнес-логика, связанная с этим экраном, касается проверки данных о существовании конкретного пользователя и действительности его пароля. Здесь реализуется бизнес-логика аутентификации, которая может потребовать запроса сведений из интернета, проверки базы данных, запуска определенных задач.

Еще один компонент архитектуры – **UI-логика**. На этом экране – нажатие кнопки.

Разберем **стандартную архитектуру MVC** (**Model View Controller**). Она проста и всем хороша, хоть многие и понимают ее неверно. **View (вид)** – то, что отображается на экране, и с этим очевидным понятием проблем не возникает. Пояснения требуются касательно **Model – модели**, которую каждый трактует на свой лад. Кто-то считает, что это база данных вроде Core Data или Realm. Или что это сами данные – например, список друзей, полученный из VK. На самом деле имеется в виду паттерн, доменная модель (**domain model**). Это связка бизнес-логики и данных приложения.

Например, список друзей из VK, сервисы получения его из сети и сохранения в базу данных – это часть модели.

Последний компонент – **Controller (контроллер)**, который является прослойкой между **view** и **model**. Он отображает **view**, запрашивает данные от модели для отображения, реагирует на нажатие кнопок и запускает необходимую бизнес-логику.

Рассмотрим, как эти компоненты представлены в iOS. **View (вид)** – это корневой view-экран, который содержит все остальные элементы. Именно его на прошлых уроках вы протипировали в storyboard. **Controller (контроллер)** – это подклассы UIViewController (UINavigationController, UITableController и т.д.). Он содержит в себе view, отображает его на экране, обрабатывает нажатия кнопок и взаимодействует с сервисами для работы с данными. Еще он хранит состояние модуля.

Model (модель) – компонент, касательно которого в iOS нет никаких рекомендаций или стандартных классов. Разработчик сам придумывает классы для выполнения бизнес-логики, хранения и представления данных. При этом можно использовать, например, фреймворк Core Data, для управления которым все равно понадобится класс. Один из вариантов сервисов-классов, которые отвечают за бизнес-логику и **DTO (Data Transfer Object)** – это простые классы или структуры, предназначенные только для передачи данных.

Разместим компоненты нашего примера по классам. Все, что касается интерфейса, будет описано в storyboard. Если что-то не получится сделать здесь (например, выставить цвет рамки элемента), нужно создать подкласс UIView, присвоить его корневому view в storyboard и настраивать интерфейс в нем.

UI-логика – а именно нажатие кнопки «Войти» – размещается в контроллере. При этом нажатие не должно выполнять лишней работы: в нашем примере от него требуется только получить введенные пользователем строки логина/пароля и передать сервису для аутентификации.

Логика аутентификации выносится в отдельный сервис – например, **AuthService.** Он будет решать, как именно должна работать эта бизнес-логика.

Данные у нас достаточно простые – две строки, поэтому не обязательно создавать для них специальную структуру. Хранить их не нужно, достаточно просто передать в сервис. Самое простое

решение – непосредственно в методе кнопки получить данные, вызвать метод сервиса и передать в него полученные строки.

Теперь, когда сложилось представление, что такое архитектура и где должны находиться части нашей логики, отметим, что ***весь код, который находится в сервисах, должен выполняться в фоновом потоке, все остальное – в главном.***

Из любого правила есть исключения: иногда даже бизнес-логика выполняется настолько быстро, что нет смысла выносить ее в отдельный поток. А бывают сложные UI-компоненты, которые полезно рассчитывать в фоне.

Конкретный пример: при создании погодного приложения наша авторизация – это просто один оператор **if**. Его не обязательно выносить в фоновый поток. А в уроке по GCD мы делали размытие изображения в фоне – хотя, казалось бы, это UI-задача.

Сложные экраны

Бывают экраны (как правило, с таблицами и коллекциями), где количество элементов превышает 100. Каждый из них необходимо создать, выделить для него память, рассчитать внешний вид, позицию на экране, получить данные и отобразить. Это требует очень много ресурсов.

В такой ситуации, даже если вы освободили главный поток от всей логики, его ресурсов может не хватить на UI. Это особенно заметно в таблицах: при быстрой прокрутке вниз главный поток не справляется с обработкой, и интерфейс начинает «лагать». Падает количество кадров, таблица визуально становится неравномерной, и работать с ней некомфортно.

Такая же проблема может иногда проявляться и на одном экране без таблицы. При этом будет вздрагивать анимация появления этого экрана. Чтобы этого избежать, можно попросить дизайнера уменьшить сложность экрана. А лучше все же взяться за оптимизацию: декомпозировать задачу построения UI на более мелкие и постараться большинство из них выполнить в фоновом потоке. Как

правило, это задачи расчета размеров и позиций элементов, которые за вас решает autoLayout. Но есть и другие задачи, которые можно выполнять «в фоне».

Лишние действия при создании объектов

Часто при создании элемента интерфейса требуется преобразовать данные, ввести вспомогательные объекты. Простой пример из приложения «Погода» – ячейка коллекции, отображающая дату.

```
override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

        <...>

        let dateFormatter = DateFormatter()
        dateFormatter.dateFormat = "dd.MM.yyyy HH.mm"
        let date = Date(timeIntervalSince1970: weather.date)
        let stringDate = dateFormatter.string(from: date)
        cell.time.text = stringDate

        <...>

        return cell
}
```

Здесь дата и время представлены в формате timestamp, а на экране они должны отображаться в привычном пользователю виде «день/месяц/год часы/минуты». Для преобразования даты нам необходимо:

1. Создать объект класса **DateFormatter**, который может преобразовывать строку в дату и обратно, при этом меняя формат вывода.
2. Установить формат строки с датой.
3. Преобразовать дату из формата **timestamp** в объект класса **Date,** который представляет дату в **Foundation** и с которым может работать **DateFormatter**.
4. Использовать **DateFormatter,** чтобы получить строку из **Date**.
5. Присвоить эту строку надписи в ячейке.

Эти действия выполняются многократно для каждой ячейки. Ведь при прокрутке таблицы ячейка, появляющаяся на экране, снова будет подготавливаться с расчетом даты. Как оптимизировать эту ресурсозатратную операцию?

Обратим внимание, что шаги 1 и 2 абсолютно одинаковы для всех ячеек, то есть объект **DateFormatter** и его формат не меняются. Значит, достаточно одного объекта, просто надо создать его не при формировании ячейки, а заранее в контроллере.

```
let dateFormatter: DateFormatter = {
    let df = DateFormatter()
    df.dateFormat = "dd.MM.yyyy HH.mm"
    return df
}()
```

В таком варианте можно сделать **dateFormatter** свойством контроллера, создать его через замыкание и сразу выставить формат. Это делается один раз при открытии контроллера, и в дальнейшем все ячейки используют это свойство. Теперь таблица оптимизирована: сокращено количество ресурсов, необходимых для построения UI.

Можно пойти дальше и кэшировать дату, чтобы не рассчитывать ее каждый раз заново. Создадим свойство для хранения кэша.

```
var dateTextCache: [IndexPath: String] = [:]
```

И перепишем метод формирования ячейки.

```
if let stringDate = dateTextCache[indexPath] {
    cell.time.text = stringDate
} else {
    let date = Date(timeIntervalSince1970: weather.date)
    let stringDate = dateFormatter.string(from: date)
    dateTextCache[indexPath]  = stringDate
    cell.time.text = stringDate
}
```

Теперь проверяем, была ли дата рассчитана ранее. Если была – просто «пришиваем» ее надписи, если нет – рассчитываем и сохраняем в кэш.

Метод формирования стал перегруженным. Вынесем работу с кэшем в отдельный метод.

```
func getCellDateText(forIndexPath indexPath: IndexPath, andTimestamp
timestamp: Double) -> String {
        if let stringDate = dateTextCache[indexPath] {
            return stringDate
        } else {
            let date = Date(timeIntervalSince1970: timestamp)
            let stringDate = dateFormatter.string(from: date)
            dateTextCache[indexPath]  = stringDate
            return stringDate
        }
    }
```

Метод ячейки теперь выглядит гораздо лучше.

```
override func collectionView(_ collectionView: UICollectionView,
cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {

        <...>
        cell.time.text = getCellDateText(forIndexPath: indexPath,
andTimestamp: weather.date)
        <...>
        return cell
}
```

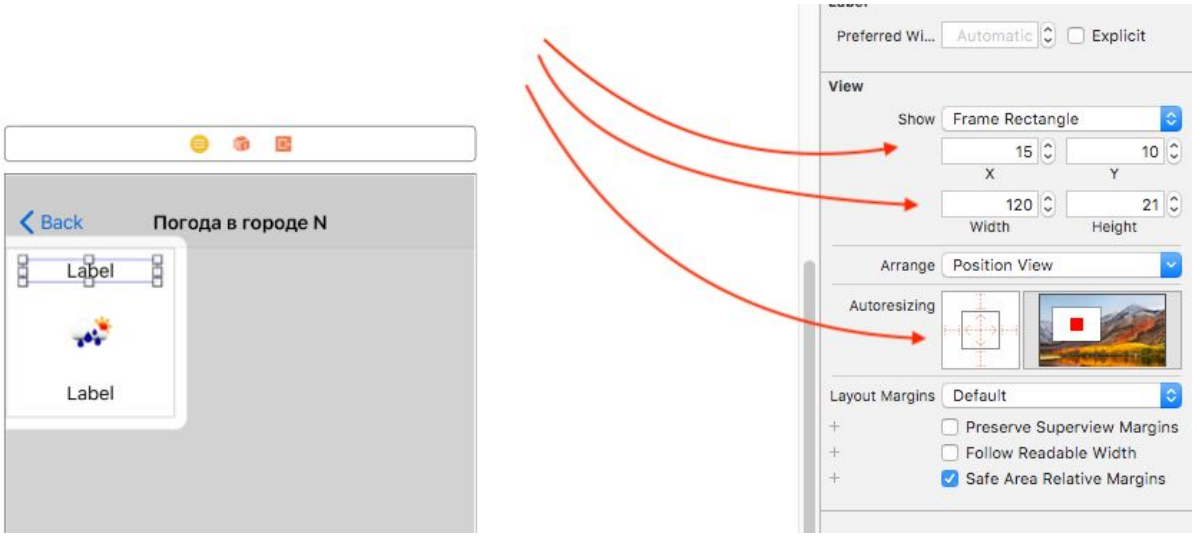
Вывод: ***всегда проверяйте код подготовки элементов***. Выносите все лишнее, что не выносится – кэшируйте, а задачи расчета выполняйте в фоновых потоках. Например, в нашем коде можно еще вынести в параллельный поток конвертирование даты, чтобы дополнительно разгрузить главный.

Расчет размеров и позиций элементов интерфейса

Чтобы указать размер и позицию элемента на экране, используется **AutoLayout**. Это удобно, но медленно. Особенно если элементы могут менять размер в зависимости от содержимого, например, как надписи. В таблице это зачастую приводит к уменьшению плавности – если элементов немного и правила простые. А когда количество элементов возрастает, скорость падает очень заметно. Это наиболее ощутимо на старых устройствах: iPhone 5 и ниже.

Решение проблемы – отказаться от **AutoLayout** и вести расчеты вручную. Этот способ называют **layout на фреймах** (от слова **frame** – «рамка»). Он требует очень большого количества кода, но если вам важно качество приложения, придется решиться на этот шаг.

В качестве примера снова возьмем ячейку погоды. Удалим все **constraints** у ее элементов. Расставим их размеры на соответствующей вкладке правой панели. Можно еще установить позицию от верхнего левого угла, высоту и ширину. Но эти размеры не динамичные: если изменить размер ячейки, элементы останутся на своих местах. Кроме того, лейблы не подстроятся под возможное изменение шрифта. Еще один неочевидный момент: на панели размеров есть раздел **Autoresaizing**, где предлагается настроить правила изменения размера элемента. Раньше это был основной способ верстки в iOS, но сейчас он оставлен только для совместимости. Он не делает ничего, кроме создания своего **constraints**, и пользоваться им не рекомендуется.



Перенесем эту верстку в код. Откроем класс **WeatherCell**. Прежде чем верстать, **отключим создание автоматических constraints для всех элементов.**

```
@IBOutlet weak var weather: UILabel! {
        didSet {
            weather.translatesAutoresizingMaskIntoConstraints = false
        }
    }
@IBOutlet weak var time: UILabel! {
        didSet {
            time.translatesAutoresizingMaskIntoConstraints = false
        }
    }
@IBOutlet weak var icon: UIImageView! {
        didSet {
            icon.translatesAutoresizingMaskIntoConstraints = false
        }
    }
```

Далее переопределяем метод **layoutSubviews**. В нем происходит верстка всех дочерних для **UIView** элементов, в том числе надписей и иконок. Суть ручной верстки заключается в том, чтобы самостоятельно определить размеры элемента, его координаты и задать их элементу в свойстве **frame**.

Добавим свойство с величиной отступов: их можно сделать как разными для каждой стороны, так и одинаковыми – например, по 10 пунктов.

```
let instets: CGFloat = 10.0
```

Рассчитываем размера текста в **UILabel**:

```
func getLabelSize(text: String, font: UIFont) -> CGSize {
        // определяем максимальную ширину текста – это ширина ячейки минус
отступы слева и справа
        let maxWidth = bounds.width - instets * 2
        // получаем размеры блока под надпись
        // используем максимальную ширину и максимально возможную высоту
        let textBlock = CGSize(width: maxWidth, height:
CGFloat.greatestFiniteMagnitude)
        // получаем прямоугольник под текст в этом блоке и уточняем шрифт
     let rect = text.boundingRect(with: textBlock, options:
.usesLineFragmentOrigin, attributes: [NSAttributedStringKey.font: font],
context: nil)
        // получаем ширину блока, переводим ее в Double
        let width = Double(rect.size.width)
        // получаем высоту блока, переводим ее в Double
        let height = Double(rect.size.height)
        // получаем размер, при этом округляем значения до большего целого
числа
        let size = CGSize(width: ceil(width), height: ceil(height))
        return size
}
```

В этом методе мы передаем размеры прямоугольника, в котором будет строиться текст. При этом ограничена ширина, но не высота. Затем используем метод **boundingRect**, чтобы получить размеры строки в этом блоке. Если текст не войдет по ширине, он сделает перенос на следующую строку. При этом уточняем шрифт, от которого тоже зависит размер. Округляем ширину и высоту полученного прямоугольника до ближайшего целого числа, чтобы iOS легко преобразовал их в пиксели.

В итоге получим размер прямоугольника, в который войдет текст, если писать его указанным шрифтом. Если сделать лейбл по этому размера, текст в нем поместится.

Следующий шаг – верстка UILabel, который отображает градусы.

```
func weaterLabelFrame() {
        // получаем размер текста, передавая сам текст и шрифт
        let weaterLabelSize = getLabelSize(text: weather.text!, font:
weather.font)
        // рассчитываем координату по оси X
        let weaterLabelX = (bounds.width - weaterLabelSize.width) / 2
        // получаем точку верхнего левого угла надписи
        let weaterLabelOrigin =  CGPoint(x: weaterLabelX, y: instets)
        // получаем фрейм и устанавливаем его UILabel
        weather.frame = CGRect(origin: weaterLabelOrigin, size:
weaterLabelSize)
}
```

Используя метод расчета размеров, получаем величину надписи. Затем рассчитываем координату по оси X, чтобы надпись находилась посередине (по Y координата постоянна и равна отступу). Получаем точку верхнего угла надписи: значения в ней – это отступ от верхнего левого угла ячейки. Используя

эту точку и размер, получаем **CGRect** – рамку с координатами. Передаем ее в свойство frame нашего UILabel, и он займет позицию согласно рамке.

Верстка UILabel с данными времени выглядит похоже, только надо рассчитать и координату по оси Y.

```
func timeLabelFrame() {
        // получаем размер текста, передавая сам текст и шрифт
        let timeLabelSize = getLabelSize(text: time.text!, font:
time.font)
        // рассчитываем координату по оси X
        let timeLabelX = (bounds.width - timeLabelSize.width) / 2
        // рассчитываем координату по оси Y
        let timeLabelY = bounds.height - timeLabelSize.height - instets
        // получаем точку верхнего левого угла надписи
        let timeLabelOrigin =  CGPoint(x: timeLabelX, y: timeLabelY)
        // получаем фрейм и устанавливаем UILabel
        time.frame = CGRect(origin: timeLabelOrigin, size: timeLabelSize)
}
```

Последней выполняем верстку иконки. У нее постоянный размер, который задаем мы – например, 50 пунктов.

```
func iconFrame() {
        let iconSideLinght: CGFloat = 50
        let iconSize = CGSize(width: iconSideLinght, height:
iconSideLinght)
        let iconOrigin = CGPoint(x: bounds.midX - iconSideLinght / 2, y:
bounds.midY - iconSideLinght / 2)
        icon.frame = CGRect(origin: iconOrigin, size: iconSize)
    }
```

Переопределим метод расчета позиции элементов.

```
override func layoutSubviews() {
        super.layoutSubviews()

        weaterLabelFrame()
        timeLabelFrame()
        iconFrame()
}
```

Так как метод расчета позиции вызывается только по необходимости или по нашему требованию, размеры надписей не будут подстраиваться под изменения текста. Добавим два метода установки текста в надписи.

```
func setWeather(text: String) {
        weather.text = text
        weaterLabelFrame()
}

func setTime(text: String) {
```

```
        time.text = text
        timeLabelFrame()
}
```

В них получаем текст и устанавливаем его нужной надписи, после чего снова рассчитываем рамку.

Полный листинг класса выглядит так:

```
class WeatherCell: UICollectionViewCell {
    @IBOutlet weak var weather: UILabel! {
        didSet {
            weather.translatesAutoresizingMaskIntoConstraints = false
        }
    }
    @IBOutlet weak var time: UILabel! {
        didSet {
            time.translatesAutoresizingMaskIntoConstraints = false
        }
    }
    @IBOutlet weak var icon: UIImageView! {
        didSet {
            icon.translatesAutoresizingMaskIntoConstraints = false
        }
    }

    let instets: CGFloat = 10.0

    func setWeather(text: String) {
        weather.text = text
        weaterLabelFrame()
    }

    func setTime(text: String) {
        time.text = text
        timeLabelFrame()
    }

    override func layoutSubviews() {
        super.layoutSubviews()

        weaterLabelFrame()
        timeLabelFrame()
        iconFrame()
    }

    func getLabelSize(text: String, font: UIFont) -> CGSize {
        //определяем максимальную ширину, которую может занимать наш текст
        //это ширина ячейки минус отступы слева и справа
        let maxWidth = bounds.width - instets * 2
        //получаем размеры блока, в который надо вписать надпись
```

```
        //используем максимальную ширину и максимально возможную высоту
        let textBlock = CGSize(width: maxWidth, height:
CGFloat.greatestFiniteMagnitude)
        //получим прямоугольник который займет наш текст в этом блоке,
уточняем, каким шрифтом он будет написан
        let rect = text.boundingRect(with: textBlock, options:
.usesLineFragmentOrigin, attributes: [NSAttributedStringKey.font: font],
context: nil)
        //получаем ширину блока, переводим ее в Double
        let width = Double(rect.size.width)
        //получаем высоту блока, переводим ее в Double
        let height = Double(rect.size.height)
        //получаем размер, при этом округляем значения до большего целого
числа
        let size = CGSize(width: ceil(width), height: ceil(height))
        return size
    }

    func weaterLabelFrame() {
        //получаем размер текста, передавая сам текст и шрифт.
        let weaterLabelSize = getLabelSize(text: weather.text!, font:
weather.font)
        //рассчитывает координату по оси X
        let weaterLabelX = (bounds.width - weaterLabelSize.width) / 2
        //получим точку верхнего левого угла надписи
        let weaterLabelOrigin =  CGPoint(x: weaterLabelX, y: instets)
        //получаем фрейм и устанавливаем UILabel
        weather.frame = CGRect(origin: weaterLabelOrigin, size:
weaterLabelSize)
    }

    func timeLabelFrame() {
        //получаем размер текста, передавая сам текст и шрифт.
        let timeLabelSize = getLabelSize(text: time.text!, font:
time.font)
        //рассчитывает координату по оси X
        let timeLabelX = (bounds.width - timeLabelSize.width) / 2
        //рассчитывает координату по оси Y
        let timeLabelY = bounds.height - timeLabelSize.height - instets
        //получим точку верхнего левого угла надписи
        let timeLabelOrigin =  CGPoint(x: timeLabelX, y: timeLabelY)
        //получаем фрейм и устанавливаем UILabel
        time.frame = CGRect(origin: timeLabelOrigin, size: timeLabelSize)
    }

    func iconFrame() {
        let iconSideLinght: CGFloat = 50
        let iconSize = CGSize(width: iconSideLinght, height:
iconSideLinght)
        let iconOrigin = CGPoint(x: bounds.midX - iconSideLinght / 2, y:
```

```
bounds.midY - iconSideLinght / 2)
        icon.frame = CGRect(origin: iconOrigin, size: iconSize)
    }

}
```

Устанавливать текст в надписи теперь надо с помощью методов:

```
cell.setTime(text: time)
cell.setWeather(text: String(weather.temp))
```

В результате коллекция будет работать без использования **AutoLayout**, а значит, ее скорость повысится.

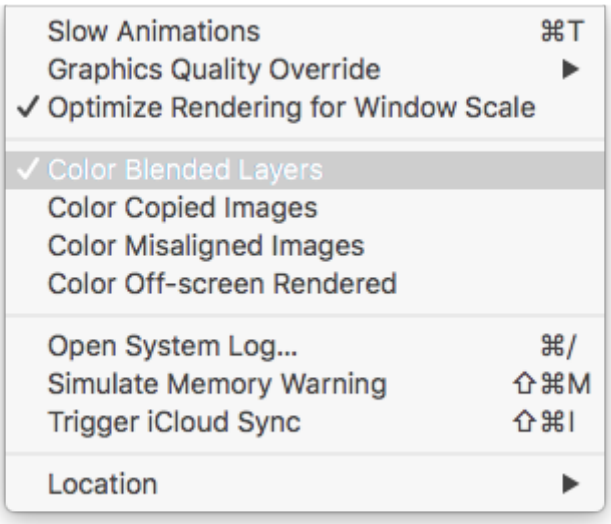
Сложные для отображения элементы

Последняя проблема, влияющая на производительность, – сложность отображения некоторых элементов. Речь идет о наложении цветов, сложных расчетах, в том числе, расчета параметров для отображения букв в строке с атрибутами. Проблему могут представлять и изображения с меняющимся размером и полупрозрачностью, а также пиксели, для которых надо вести расчеты на основе нескольких элементов, расположенных одни под другими.

Текст с атрибутами оптимизировать невозможно, поэтому стоит сначала применить остальные правила оптимизации, и если результат все еще неудовлетворительный, отказаться от такого текста. Есть еще вариант перевода всего интерфейса на слои (**layers**), но это сложная задача, за которую пока браться не будем.

Разберем **смешивание цветов для разных слоев** элементов. Например, поверх белого экрана лежит красный прямоугольник с прозрачностью 50%. В результате пользователь увидит смешение красного и белого цветов – элемент будет розовым. Для разработчика нет разницы – делать прямоугольник розовым или красным с полупрозрачностью. Это может быть важно только в редких случаях, когда фон под элементом может меняться.

Разница очевидна с точки зрения системы отображения интерфейса: приходится проделывать дополнительную работу по сложению цветов и расчету итогового значения. Чем больше полупрозрачных слоев находятся один над другим, тем сложнее вычисления. Проблема возникает, даже если складывать полностью прозрачный фон с другим – схемы расчета останутся те же, так как для компьютера «прозрачный цвет» не отличается от остальных. В симуляторе есть специальный режим для обнаружения проблем наложения цветов – «**Color Blended Layers**».

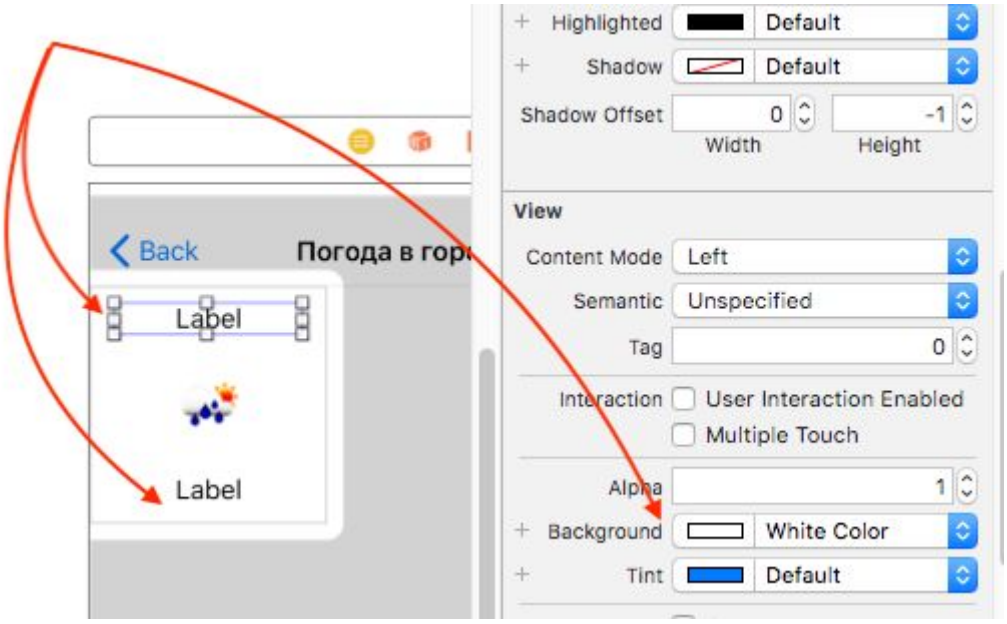


Включаем этот режим, и симулятор подсвечивает красным слои, для которых пришлось смешивать краски. Участки без цветовых наложений светятся зеленым.

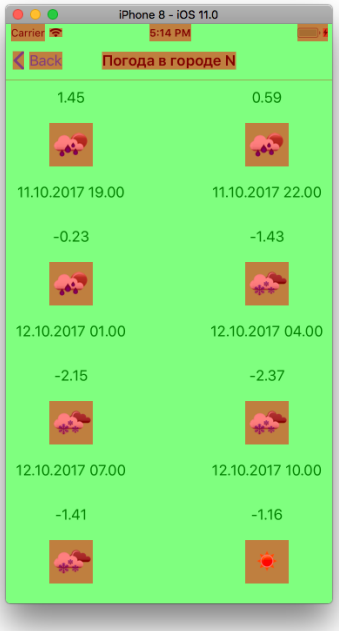
Оказывается, смешение затронуло все элементы, даже системные. Попробуем уйти от этого, ведь для большинства элементов нам не надо выполнять расчет.

Разберемся с панелью навигации: откроем ее атрибуты, деактивируем галочку **translucent** – это свойство делает панель полупрозрачной. Смело отказываемся от него, так как в приложении это не нужно.

Далее меняем прозрачный фон надписей на белый. С точки зрения внешнего вида ничего не изменится, а работа ускорится.



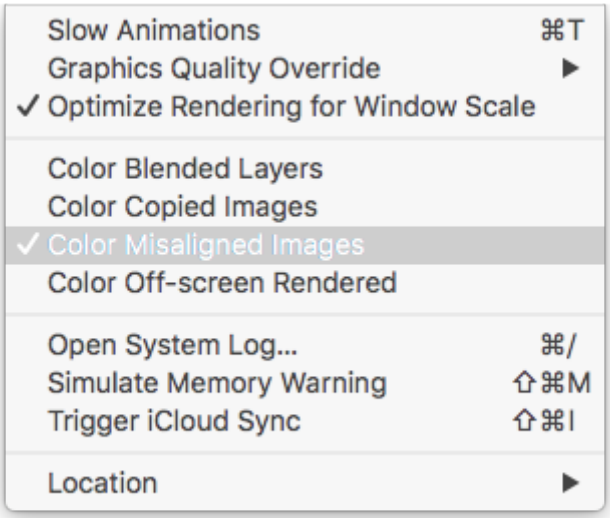
Изображение тоже полупрозрачное, но на него не повлиять – ведь оно скачано с сервера. Но и в таком виде смешения красок стало куда меньше, а значит приложение ускорено.



Оставшиеся две проблемы – с изображениями и строками с атрибутами – мы не решили, так как это требует перевода верстки на слои (**layers**). Это задача за пределами данного курса, да и в целом редкая практика. Плавности интерфейса можно достичь, не впадая в такие крайности.

Еще одна возможная проблема – **несоответствие элементов пикселям**. Так бывает, если размеры элементов дробные – 10,1, например. На экране все состоит из пикселей, и в приложении мы

оперируем пунктами, равными одному, двум или трем пикселям в зависимости от плотности дисплея. И «цифры после запятой» в размерах очень осложняют размещение элементов. В симуляторе существует режим для поиска таких случаев – «**Color Misaligned Images**».



Рассмотрим измененный пример с ручной верстkой ячейки таблицы.

```
func getLabelSize(text: String, font: UIFont) -> CGSize {
        // определяем максимальную ширину текста: ширина ячейки минус отступы
слева и справа
        let maxWidth = bounds.width - instets * 2
        // получаем размеры блока под надпись
        // используем максимальную ширину и максимально возможную высоту
        let textBlock = CGSize(width: maxWidth, height:
CGFloat.greatestFiniteMagnitude)
        // получаем прямоугольник под текст в этом блоке и уточняем шрифт
        let rect = text.boundingRect(with: textBlock, options:
.usesLineFragmentOrigin, attributes: [NSAttributedStringKey.font: font],
context: nil)
        // получаем ширину блока, переводим ее в Double
        let width = Double(rect.size.width)
        // получаем высоту блока, переводим ее в Double
        let height = Double(rect.size.height)
        // получаем размер
        let size = CGSize(width: width, height: height)
        return size
}
```

При получении размера в предпоследней строке исчезли функции округления. Код сразу потерял в производительности.

В этом режиме симулятор подсвечивает цветом проблемные блоки. Фиолетовые – как раз те, у которых дробные значения размера. Это приводит к проблемам отрисовки, качество изображения портится, падает скорость обработки этих элементов. Исправляем: возвращаем округление размеров.

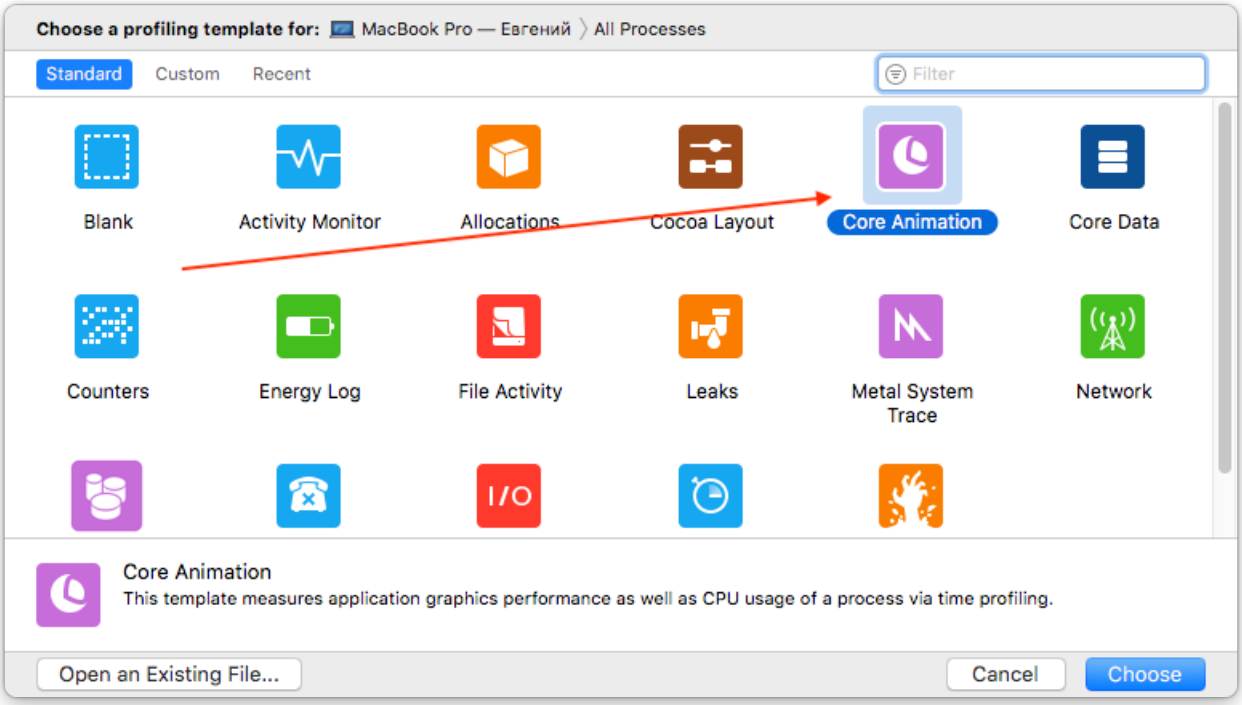
```
let size = CGSize(width: ceil(width), height: ceil(height))
```

Последняя проблема с изображениями связана с их неподходящим размером. Есть правило: если на экране изображение занимает квадрат 50 на 50 пунктов, его разрешение должно быть 50x50, 100x100 или 150x150 пикселей – в зависимости от плотности дисплея. В каталоге ресурсов есть три ячейки для изображений – как раз для этих случаев.

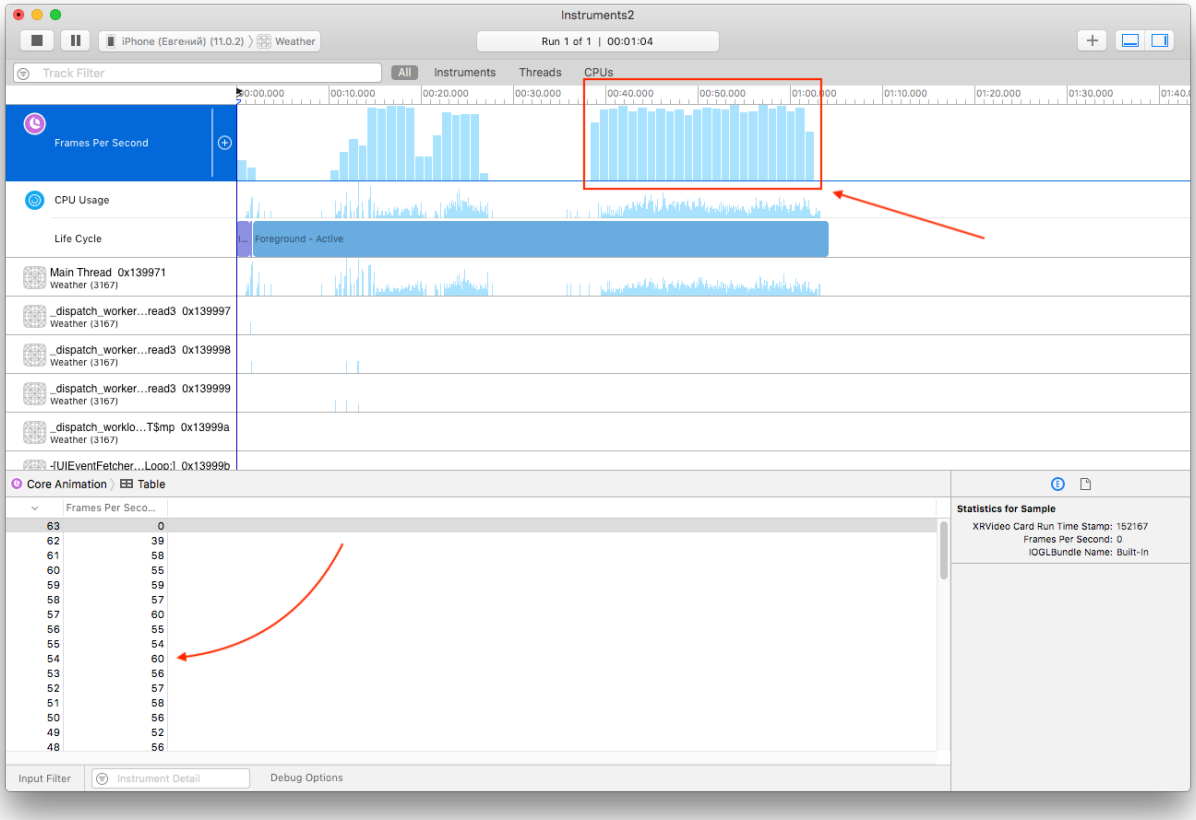
С картинкой, загруженной из интернета, нельзя ничего сделать. Но если бы это были локальные изображения, стоило попросить дизайнера подготовить три варианта каждого.

На этом мы закончим тему ускорения пользовательского интерфейса в целом и таблиц в частности. Выводы: старайтесь не допускать простых ошибок вроде ненужной полупрозрачности; избегайте лишней работы при создании элементов, как в примере с датой. Бизнес-логику держите в параллельных потоках. Остальные приемы используйте по необходимости: не всегда AutoLayout тормозит приложения, и даже если это происходит, можно попробовать обойтись небольшими расчетами без переписывания всей верстки в коде.

Чтобы определить количество кадров в секунду, используйте специальный инструмент из обоймы xcode.



Он поможет определить, требуется ли оптимизация. Использовать этот инструмент можно только при запуске на реальном устройстве, возраст которого также имеет значение. Тесты на iPhone 8 могут быть идеальными, а при запуске на iPhone 5 проблемы видно и без тестов.



Внизу отображается частота кадров в определенный момент времени, то же самое можно увидеть вверху в графическом виде. Когда на экране ничего не меняется, частота кадров падает. При

интенсивной анимации, как при прокручивании таблицы, частота кадров должна стремится к 60. Если показатель ниже 40 – интерфейс будет заметно тормозить.

Практическое задание

1. Выполнить оптимизацию приложения по рекомендациям из этого урока.

Дополнительные материалы

1. Доменная модель

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/documentation