GeekBrains

Пользовательский интерфейс iOS-приложений

Создание кастомных UI-компонентов

Рисование с помощью CoreGraphics. @IBDesignable и @IBInspectable. CALayer: тени, границы, маска, градиент. Трансформация. Обработка пользовательских жестов. UIControl. UIAppearance.

Оглавление

Рисование с помощью CoreGraphics

- Рисование простых фигур
- Рисование сложных фигур
- Метод setNeedsDisplay

IBDesignable и IBInspectable

Работа со слоем CALayer

- Работа со слоем UIView
- Свойства CALayer
- Создание слоя-градиента

Трансформация UIView

- 2D-трансформация
- Объединение нескольких трансформаций
- 3D-трансформация

Обработка пользовательских жестов

- Обработка методов UIResponder
- Распознавание жестов

Создание кастомного контрола

- Класс UIControl

UIAppearance

Практика

- Создание UI-компонента с градиентом
- Создание контрола «Выбор дня недели»

Практическое задание

Пример выполненной работы

Дополнительные материалы

Используемая литература

Рисование с помощью CoreGraphics

При разработке приложения создают множество UI-компонентов. Некоторые из них — простые элементы (контейнеры, группы компонентов). Но часто приходится создавать компоненты, которых нет в стандартном наборе. Фреймворк **UIKit** предоставляет дополнительные инструменты, с помощью которых можно сделать любой UI-компонент.

Рисование простых фигур

С помощью фреймворка **CoreGraphics** можно «рисовать» на **CALayer** и **UIView**. Это происходит в методе **draw(_ rect: CGRect)** для **UIView** и **draw(in context: CGContext)** для **CALayer**. Используется экземпляр класса **CGContext**, который можно получить, вызвав метод **UIGraphicsGetCurrentContext()**.

Экземпляр **CGContext** содержит графический контекст — параметры, с которыми выполняется рисование. С его помощью можно рисовать, используя два типа методов: **fill** (выполняет заливку заданной фигуры) и **stroke** (обводит).

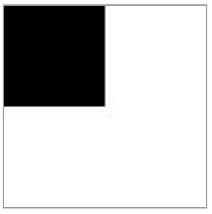
Чтобы выполнить заливку прямоугольника размером 50x50, создадим наследника **UIView** и переопределим метод **draw**:

```
class TestView: UIView {
    override func draw(_ rect: CGRect) {
        super.draw(rect)
        guard let context = UIGraphicsGetCurrentContext() else { return }
        context.fill(CGRect(x: 0, y: 0, width: 50, height: 50))
    }
}
```

После этого можно создать экземпляр класса **TestView**. Зададим ему размер 100x100 и установим белый цвет фона:

```
let testView = TestView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
testView.backgroundColor = .white
```

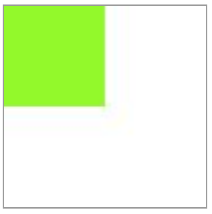
В результате компонент будет выглядеть так:



Чтобы поменять цвет заливки или обводки, необходимо установить его для нашего контекста. Сделать это можно с помощью метода **setFillColor(_ color: CGColor)** или **setStrokeColor(_ color: CGColor)**. Добавим перед методом **fill** установку цвета:

```
class TestView: UIView {
    override func draw(_ rect: CGRect) {
        super.draw(rect)
        guard let context = UIGraphicsGetCurrentContext() else { return }
        context.setFillColor(UIColor.green.cgColor)
        context.fill(CGRect(x: 0, y: 0, width: 50, height: 50))
    }
}
```

В результате компонент будет выглядеть так:



Рисование сложных фигур

С помощью методов **fill** и **stroke** можно рисовать прямоугольник и круг. Чтобы изобразить сложную фигуру — например, звезду, — можно использовать линии.

Чтобы нарисовать линию, надо:

- Установить начало линии с помощью метода **move(to: CGPoint)**;
- Установить конец линии с помощью метода **addLine(to: CGPoint)**;
- Закрыть нарисованный **path** методом **closePath()**;
- Выполнить обводку методом **strokePath()**.

Чтобы понять принцип, представьте, что **CGContext** рисует карандашом по заданным точкам. С помощью метода **move** ставится начальная точка, потом посредством метода **addLine** добавляются еще. После вызова **strokePath** карандаш рисует линии от начальной точки до последней. Метод **closePath** соединяет последнюю точку с первой (иначе это не произойдет). Важно не забывать писать метод **closePath** после того, как закончили рисовать отдельную фигуру, так как следующая будет соединена с предыдущей.

Нарисуем обычную звезду:

```
class TestView: UIView {
    override func draw(_ rect: CGRect) {
        super.draw(rect)
        guard let context = UIGraphicsGetCurrentContext() else { return }
        context.setStrokeColor(UIColor.red.cgColor)
        context.move(to: CGPoint(x: 40, y: 20))
        context.addLine(to: CGPoint(x: 45, y: 40))
        context.addLine(to: CGPoint(x: 65, y: 40))
        context.addLine(to: CGPoint(x: 50, y: 50))
        context.addLine(to: CGPoint(x: 60, y: 70))
        context.addLine(to: CGPoint(x: 40, y: 55))
        context.addLine(to: CGPoint(x: 20, y: 70))
        context.addLine(to: CGPoint(x: 30, y: 50))
        context.addLine(to: CGPoint(x: 15, y: 40))
        context.addLine(to: CGPoint(x: 35, y: 40))
        context.closePath()
        context.strokePath()

    }
}
```

В результате на **view** будет такая звезда:

Можно также добавлять прямоугольники, овалы и закругления.

Когда добавляем линии, на самом деле контекст строит объект **CGPath**, который представляет собой набор точек. Иногда необходимо изображать одни и те же фигуры в разных компонентах, и для этого лучше иметь нарисованный в них объект, который будет хранить описание фигуры. Чтобы создать его, нужно использовать класс **UIBezierPath**. Он позволяет описать фигуру, которую потом можно будет нарисовать. В случае со звездой создание объекта будет выглядеть так:

```
class TestView: UIView {
    override func draw(_ rect: CGRect) {
        super.draw(rect)
        guard let context = UIGraphicsGetCurrentContext() else { return }
        context.setStrokeColor(UIColor.red.cgColor)
        let path = UIBezierPath()
        path.move(to: CGPoint(x: 40, y: 20))
        path.addLine(to: CGPoint(x: 45, y: 40))
        path.addLine(to: CGPoint(x: 65, y: 40))
        path.addLine(to: CGPoint(x: 50, y: 50))
        path.addLine(to: CGPoint(x: 60, y: 70))
        path.addLine(to: CGPoint(x: 40, y: 55))
        path.addLine(to: CGPoint(x: 20, y: 70))
        path.addLine(to: CGPoint(x: 30, y: 50))
        path.addLine(to: CGPoint(x: 15, y: 40))
        path.addLine(to: CGPoint(x: 35, y: 40))
        path.close()
        path.stroke()
    }
}
```

Заметьте, что объект **UIBezierPath** может нарисовать себя сам с помощью метода **stroke**. Также его можно добавить в текущий контекст:

```
context.addPath(path.cgPath)
context.strokePath()
```

Результат будет аналогичный.

Метод setNeedsDisplay

Метод **draw** вызывается перед первым показом **view** на экране. Если потребуется перерисовать **view** — например, при изменении его размеров, — это не произойдет автоматически. Чтобы заново нарисовать view, необходимо вызвать метод **setNeedsDisplay**. Он в нужный момент создаст графический контекст и вызовет метод **draw**. Его нельзя вызвать вручную, так как не будет создан графический контекст и, следовательно, ничего не будет нарисовано.

Вариация метода **setNeedsDisplay** — **setNeedsDisplay(_ rect: CGRect)**. Этот метод передаст в **draw** прямоугольник под изображение. Будет нарисована только часть, которая входит в него.

IBDesignable и IBInspectable

Если в **storyboard** выбрать **view** и открыть **attributes inspector**, можно увидеть разные свойства — например, **backgroundColor**, **alpha** и **tint**. При смене этих свойств будет меняться и сама **view**.

При создании кастомной **view** удобно менять ее свойства в **storyboard**. Чтобы свойство появилось в **attributes inspector**, нужно добавить к нему атрибут **@IBInspectable**. Это сделает его видимым в **attributes inspector** и позволит изменять. Но не все типы поддерживают данный атрибут. **@IBInspectable** можно применить к следующим:

- CGFloat;

- CGPoint;
- CGSize;
- CGRect;
- UIColor;
- числовые типы;
- строки;
- Bool.

Если переопределить метод **draw**, то все, что будет в нем нарисовано, не отобразится в **storyboard**. А иногда нужно видеть, как **view** будет выглядеть.

Чтобы в **storyboard** отображалась **view** и при этом был виден результат работы метода **draw**, нужно добавить атрибут **@IBDesignable** к классу.

Таким образом, сочетание **IBDesignable** и **IBInspectable** даст возможность корректировать свойства **view** и видеть результат этих изменений.

Создадим простой компонент, на котором будет нарисован круг, и добавим возможность изменять его радиус в **storyboard**:

```
@IBDesignable class TestView: UIView {

    @IBInspectable var radius: CGFloat = 10 {
        didSet {
            setNeedsDisplay()
        }
    }

    override func draw(_ rect: CGRect) {
        super.draw(rect)
        guard let context = UIGraphicsGetCurrentContext() else { return }
        context.setFillColor(UIColor.red.cgColor)
        context.fillEllipse(in: CGRect(x: rect.midX - radius,
                                       y: rect.midY - radius,
                                       width: radius * 2,
                                       height: radius * 2))

    }
}
```

Теперь свойство **radius** можно менять в **storyboard** и видеть результат.

Но в использовании **IBDesignable** есть один минус. Чтобы отобразить **view** в **storyboard,** Xcode выполнит сборку проекта. Это будет происходить каждый раз при открытии **storyboard**. Когда проект большой и сборка занимает много времени, отображение **view** затягивается. Поэтому часто используют атрибут **IBInspectable** без **IBDesignable**, чтобы избежать лишней сборки. Свойства **view** по-прежнему можно будет менять в **storyboard**, но результат виден не будет.

Работа со слоем CALayer

Работа со слоем UIView

У каждого **view** есть слой, на котором происходит рендеринг его содержимого. Этот объект является классом **CALayer** или его наследником и задается в свойстве **layerClass**, которое находится во **view**. По умолчанию у обычного **UIView** это свойство возвращает класс **CALayer**, но его можно переопределить:

```
class TestView: UIView {
    override class var layerClass: AnyClass {
        return CAShapeLayer.self
    }
}
```

Класс **CAShapeLayer** — наследник **CALayer**, который представляет собой слой-фигуру.

Если нет необходимости, чтобы **view** имело другой слой, это свойство переопределять не нужно — оно будет возвращать базовый класс, **CALayer**.

Свойства CALayer

У слоя больше свойств, чем у **view**, и это предоставляет более обширные возможности для настройки дизайна UI-компонента.

С помощью **CALayer** можно настроить отображение границ. Для этого нужно установить свойства **borderWidth** и **borderColor**, обратившись к свойству **UIView.layer**:

```
let testView = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
testView.layer.borderWidth = 2
testView.layer.borderColor = UIColor.red.cgColor
```

В результате получим view такого вида:

Часто в дизайне используется закругление краев. Для этого в классе **CALayer** есть свойство **cornerRadius**. Попробуем закруглить края у **view**:

```
testView.layer.cornerRadius = 16
```

В результате получим такой результат:

Закруглений нет. Это произошло потому, что у слоя не включено свойство **masksToBounds**. По умолчанию оно равно **false**, но если установить **true**, оно активирует эффект закругления и не отображает то, что находится за пределами слоя — «обрезает» его. После установки данного свойства увидим закругления:

У **CALayer** есть свойства, позволяющие добавить тень. Их несколько:

- **shadowColor** — устанавливает цвет тени;
- **shadowOpacity** — устанавливает прозрачность тени;
- **shadowRadius** — устанавливает размер тени;
- **shadowOffset** — устанавливает сдвиг тени по X и Y.

Добавим тень к **view**:

```
testView.layer.shadowColor = UIColor.black.cgColor
testView.layer.shadowOpacity = 0.5
testView.layer.shadowRadius = 8
testView.layer.shadowOffset = CGSize.zero
```

Тень не появилась, так как установлено свойство **masksToBounds**, которое обрезает ее. Если убрать его, тень отобразится:

Иногда необходимо не только закруглять края, но и придавать **view** форму и делать так, чтобы контент не выходил за ее границы. Для этого у слоя есть свойство **mask**. Это тоже объект типа **CALayer**, который представляет собой фигуру, ограничивающую контент слоя, то есть маскирующую его. Обычно в качестве маски используется **CAShapeLayer**, так как его можно сконфигурировать. Чтобы придать ему вид сложной фигуры, нужно установить свойство **path**. Это тот же самый **path**, который мы делали, когда рисовали звезду. Можем использовать его, чтобы сделать слой-маску в форме звезды. Для этого нужно написать следующий код:

```
testView.backgroundColor = .red
let maskLayer = CAShapeLayer()
let starPath = UIBezierPath()
starPath.move(to: CGPoint(x: 40, y: 20))
starPath.addLine(to: CGPoint(x: 45, y: 40))
starPath.addLine(to: CGPoint(x: 65, y: 40))
starPath.addLine(to: CGPoint(x: 50, y: 50))
starPath.addLine(to: CGPoint(x: 60, y: 70))
starPath.addLine(to: CGPoint(x: 40, y: 55))
starPath.addLine(to: CGPoint(x: 20, y: 70))
starPath.addLine(to: CGPoint(x: 30, y: 50))
starPath.addLine(to: CGPoint(x: 15, y: 40))
starPath.addLine(to: CGPoint(x: 35, y: 40))
starPath.close()
starPath.stroke()
maskLayer.path = starPath.cgPath // Тот path, с помощью которого рисовали звезду
testView.layer.mask = maskLayer
```

В результате получим view:

Создание слоя-градиента

С помощью класса **CAGradientLayer** можно создать линейный градиент, состоящий из нескольких цветов. У данного класса есть свойства:

- **colors** — массив цветов, из которых состоит градиент;
- **locations** — массив точек, в которых заканчивается цвет градиента. Он должен совпадать по размеру с массивом **colors** и содержать числа от 0 до 1 в порядке возрастания;
- **startPoint** — относительная точка начала градиента, значения которой должны быть в диапазоне от 0 до 1;
- **endPoint** — относительная точка конца градиента.

Создадим простой слой с черно-белым вертикальным градиентом:

```
let gradientLayer = CAGradientLayer()
gradientLayer.colors = [UIColor.black.cgColor, UIColor.white.cgColor]
gradientLayer.locations = [0 as NSNumber, 1 as NSNumber]
gradientLayer.startPoint = CGPoint.zero
gradientLayer.endPoint = CGPoint(x: 0, y: 1)

testView.layer.addSublayer(gradientLayer)
gradientLayer.frame = testView.bounds
```

В результате получим **view**:

Трансформация UIView

Чтобы скорректировать размер или позицию **view**, можно изменить параметр **frame**. Но бывают случаи, когда надо несколько раз менять положение или размер **view**. В таком случае будет неудобно каждый раз пересчитывать **frame**. Для этого используются трансформации: они позволяют перемещать, менять размер view и поворачивать их.

2D-трансформация

Первый тип трансформации — в 2D-пространстве, по осям X и Y. За нее отвечает класс **CGAffineTransform**. У него есть конструкторы для различных трансформаций. Рассмотрим каждый из них:

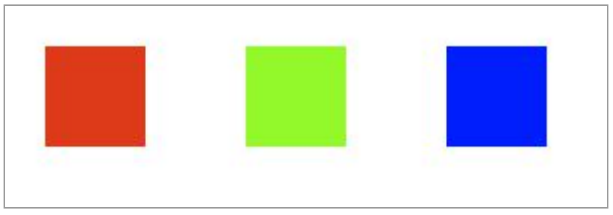
```
let translation = CGAffineTransform(translationX: 10, y: 20)
let scale = CGAffineTransform(scaleX: 1.5, y: 1.5)
let rotation = CGAffineTransform(rotationAngle: .pi / 4)
```

Первый конструктор создает трансформацию, при которой объект переместится на 10 точек вправо по оси X и на 20 — вниз по Y. Второй увеличивает объект в полтора раза в длину и в ширину. Третий поворачивает объект на половину длины окружности по часовой стрелке. В качестве угла поворота нужно передать угол в радианах, а не в градусах.

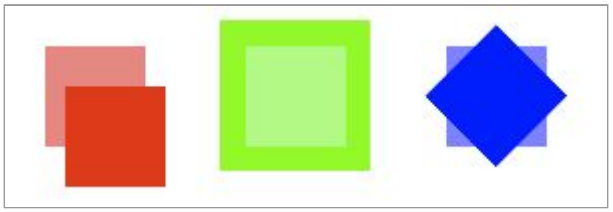
Теперь можно создать **UIView** и применить трансформации:

```
let view = UIView(frame: CGRect(x: 20, y: 20, width: 40, height: 40))
view.transform = translation
view.transform = scale
view.transform = rotation
```

Перед применением трансформаций эти **view** будут выглядеть так:



После:



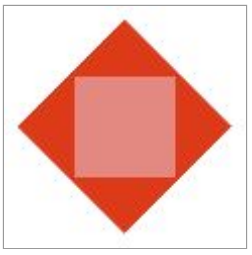
У **CGAffineTransform** есть свойство **identity** — отсутствие трансформации. Изначально свойство **UIView.transform** равно **CGAffineTransform.identity**. Если требуется применить трансформацию, а затем вернуть **view** в изначальное положение, нужно установить свойство **transform**, равное **CGAffineTransform.identity**.

Объединение нескольких трансформаций

Чтобы применить сразу несколько трансформаций — например, уменьшить и повернуть **view**, — нужно использовать метод **concatenating**, который находится в экземпляре класса **CGAffineTransform**. Объединим трансформации **scale** и **rotation**:

```
let concatenatedTransform = scale.concatenating(rotation)
```

В результате получим такую трансформацию:



3D-трансформация

В некоторых случаях необходимо применить 3D-эффект к view — например, как в стандартном компоненте **UIPickerView**. Для этого используется 3D-трансформация, которую можно создать с помощью методов **CATransform3D**. Рассмотрим основные из них:

- **CATransform3DMakeTranslation(tx: CGFloat, ty: CGFloat, tz: CGFloat)**;
- **CATransform3DMakeRotation(angle: CGFloat, x: CGFloat, y: CGFloat, z: CGFloat)**;
- **CATRansform3DMakeScale(sx: CGFloat, sy: CGFloat, sz: CGFloat)**.

Они работают идентично методам 2D-трансформации, но добавляют возможность изменять **view** по оси Z.

Метод **CATransform3DMakeTranslation** описывает перемещение в трехмерном пространстве. В качестве аргументов нужно передавать расстояние, на которое надо переместиться **view** относительно каждой оси.

CATransform3DMakeRotation описывает вращение в трехмерном пространстве. В качестве первого аргумента нужно передать угол поворота в радианах. Оставшиеся параметры определяют оси, относительно которых будет выполнен поворот. Если передано число, отличное от 0 (обычно –1), поворот выполнится, а если 0, относительно этой оси поворота не будет.

CATRansform3DMakeScale описывает изменение размера в трехмерном пространстве. В качестве параметров нужно передать относительное увеличение или уменьшение **view**. Значение по умолчанию — 1. Если необходимо уменьшить **view** по горизонтали в 2 раза, в параметр **sx** передают значение **0.5**.

Создадим каждую из этих трансформаций:

```
let translation3D = CATransform3DMakeTranslation(10, 20, 100)
let rotation3D = CATransform3DMakeRotation(.pi / 4, 0, 0, 1)
let scale3D = CATransform3DMakeScale(0.5, 0.5, 1)

view.layer.transform = translation3D
view1.layer.transform = rotation3D
view2.layer.transform = scale3D
```

Обратите внимание, что 3D-трансформация применяется не напрямую к **UIView**, а к его свойству **layer**.

В результате получатся такие view:

Обработка пользовательских жестов

Обработка методов UIResponder

Класс **UIView** реализует протокол **UIResponder**. Это означает, что данный объект может обрабатывать пользовательские жесты. По умолчанию **UIView** не обрабатывает жесты, но если переопределить некоторые методы из протокола **UIResponder**, эта возможность появится. Следующие методы можно переопределить для обработки жестов:

- **func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)**;
- **func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?)**;
- **func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?)**;
- **func touchesCancelled(_ touches: Set<UITouch>, with event: UIEvent?)**.

Структура этих методов одинаковая. Первый параметр — это объекты, которые содержат информацию о нажатии на экран, второй — событие, в рамках которого произошли нажатия.

Первый метод вызывается в тот момент, когда пользователь коснулся одним или несколькими пальцами экрана. Второй — когда переместил пальцы. Третий — когда отпустил. Четвертый — когда касания были прерваны (например, показом системного алерта).

Чтобы получить координаты нажатия, необходимо вызвать метод **location(in view: UIView)** класса **UITouch**. Он определяет координаты нажатия в определенной **view** и возвращает объект типа **CGPoint**. После получения координат можно использовать их.

Распознавание жестов

Для распознавания жестов, таких как нажатие и движение, в **UIKit** есть классы. Базовый для всех жестов — **UIGestureRecognizer**. Он содержит общую логику распознавания, которую используют дочерние классы. Список распознавателей:

- **UITapGestureRecognizer** — распознает одиночное нажатие;
- **UILongPressGestureRecognizer** — длительное одиночное нажатие;
- **UIPanGestureRecognizer** — движение пальца;
- **UISwipeGestureRecognizer** — жест «смахивания»;
- **UIRotationGestureRecognizer** — жест вращения двух пальцев по окружности;
- **UIPinchGestureRecognizer** — жест разведения или сведения двух пальцев.

Чтобы распознать любой из перечисленных жестов, необходимо создать объект распознавателя, настроить его и определить в нужный **view**. Добавим распознаватель одиночного нажатия:

```
class TestView: UIView {
    lazy var tapGestureRecognizer: UITapGestureRecognizer = {
        let recognizer = UITapGestureRecognizer(target: self,
                                                action: #selector(onTap))
        recognizer.numberOfTapsRequired = 1    // Количество нажатий,
необходимое для распознавания
        recognizer.numberOfTouchesRequired = 1 // Количество пальцев, которые
должны коснуться экрана для распознавания
        return recognizer
    }()

    @objc func onTap() {
        print("Произошло нажатие")
    }

    override init(frame: CGRect) {
        super.init(frame: frame)
        addGestureRecognizer(tapGestureRecognizer)
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

}
```

При создании распознавателя нужно указать объект и его метод, который будет вызван, когда распознается жест.

Создание кастомного контрола

Класс UIControl

Контролы — это UI-компоненты, которые реагируют на действия пользователя. Стандартные контролы: **UIButton**, **UISwitch**, **UISlider**, **UISegmentedControl**.

У каждого контрола есть несколько состояний, в которых он может находиться. Они перечислены в **UIControlState**:

- **normal** — обычное состояние, при котором контрол активен;
- **highlighted** — контрол выделен;
- **disabled** — не активен;
- **selected** — выбран;
- **focused** — находится в фокусе.

Каждый контрол посылает события — например, что на него нажали. Чтобы обработать событие, нужно подписаться на него:

```
button.addTarget(self,
                 action: #selector(onTap(_:)),
                 for: .touchUpInside)
```

В этом примере мы подписались на нажатие на кнопку, после которого будет вызван метод **onTap**. В качестве параметра **action** может быть один из следующих методов:

- **@objc func onTap()** — обычный метод без аргументов;
- **@objc func onTap(_ sender: UIButton)** — метод, в аргумент которого попадает контрол;
- **@objc func onTap(_ sender: UIButton, forEvent event: UIEvent?)** — метод, в аргументы которого попадает контрол и событие, которое произошло в момент вызова действия.

В качестве третьего параметра в метод **addTarget** мы передали событие **touchUpInside**. Всего таких событий несколько, вот основные:

- **touchDown** — произошло нажатие;
- **touchUp** — пользователь отпустил палец;
- **touchUpInside** — пользователь отпустил палец, находясь в пределах контрола;
- **touchUpOutside** — пользователь отпустил палец, находясь за пределами контрола;
- **valueChanged** — значение контрола изменилось (например, у **UISlider**).

Чтобы добавить собственный контрол, нужно создать наследника класса **UIControl** и переопределить следующие методы:

```
func beginTracking(_ touch: UITouch, with event: UIEvent?) -> Bool
func continueTracking(_ touch: UITouch, with event: UIEvent?) -> Bool
func endTracking(_ touch: UITouch?, with event: UIEvent?)
func cancelTracking(with event: UIEvent?)
```

Они похожи на методы **UIView**, отвечающие за отслеживание нажатий.

Метод **beginTracking** вызывается в момент первого нажатия пользователя на контрол. В качестве возвращаемого значения надо передать **true**, если отслеживание прикосновения должно продолжиться, и **false** в обратном случае.

Метод **continueTracking** вызывается в момент смены позиции нажатия. В качестве возвращаемого значения надо передать **true**, если прикосновение должно продолжиться, и **false** — если нет.

Метод **endTracking** вызывается, когда пользователь отпустил палец.

Метод **cancelTracking** вызывается, если отслеживание касания было отменено системой.

UIAppearance

Когда в приложении появляется много экранов и UI-компонентов, становится тяжело менять дизайн — каждый раз для всех компонентов. В помощь был создан протокол **UIAppearance**. Он позволяет задавать свойства любого UI-компонента перед тем, как он будет создан. Сделать это нужно один раз, и все компоненты будут создаваться с заданными свойствами.

Чтобы задать свойства компонента с помощью **UIAppearance**, нужно получить объект-прототип, вызвав метод **appearance()**. Установим дизайн для **UINavigationBar**:

```
let navigationBarAppearance = UINavigationBar.appearance()
navigationBarAppearance.tintColor = UIColor.white
navigationBarAppearance.barTintColor = UIColor.black
```

Теперь любой **navigation bar** в приложении будет черного цвета с белыми иконками.

Иногда бывает необходимо сделать для всех компонентов один дизайн, а для какого-то конкретного — другой. В этом случае можно использовать метод **appearance(whenContainedInInstanceOf: [UIAppearanceContainer.Type])**. Он меняет дизайн компонента, который находится в одном из переданных. Пример — установка цвета фона у **UITextField**, который находится в **UISearchBar**:

```
UITextField
    .appearance(whenContainedInInstancesOf: [UISearchBar.self])
    .backgroundColor = UIColor.black
```

Чтобы установить разный дизайн для устройств или экранов, можно использовать метод **appearance(for traitCollection: UITraitCollection)**. Например, сделать разный шрифт у **UILabel** для iPad и iPhone:

```
let iPadLabelAppearance = UILabel.appearance(for:
UITraitCollection(userInterfaceIdiom: .pad))
iPadLabelAppearance.font = UIFont.systemFont(ofSize: 24)

let iPhoneLabelAppearance = UILabel.appearance(for:
UITraitCollection(userInterfaceIdiom: .phone))
iPhoneLabelAppearance.font = UIFont.systemFont(ofSize: 17)
```

Также можно менять свойства собственных UI-компонентов. Для этого необходимо дописать в объявлении свойств ключевое слово **dynamic**. Так можно сделать единый дизайн для всех UI-компонентов в приложении.

Практика

Создание UI-компонента с градиентом

В качестве практического задания создадим **view** с градиентом. Требования к этому компоненту будут следующие:

- градиент должен быть двухцветным;
- возможность настроить любой параметр градиента;
- возможность редактировать и визуализировать **storyboard**.

Создадим новый файл **swift** и класс **GradientView**:

```
import UIKit
class GradientView: UIView {

}
```

Изменим класс слоя на **CAGradientLayer**. Для этого добавим следующий код в класс:

```
override static var layerClass: AnyClass {
    return CAGradientLayer.self
}
```

Можно создать вычисляемую переменную для удобной работы со слоем:

```
var gradientLayer: CAGradientLayer {
    return self.layer as! CAGradientLayer
}
```

Добавим свойства, отвечающие за параметры градиента:

```
var startColor: UIColor = .white // Начальный цвет градиента
```

```
var endColor: UIColor = .black // Конечный цвет градиента
var startLocation: CGFloat = 0 // Начало градиента
var endLocation: CGFloat = 1 // Конец градиента
var startPoint: CGPoint = .zero // Точка начала градиента
var endPoint: CGPoint = CGPoint(x: 0, y: 1) // Точка конца градиента
```

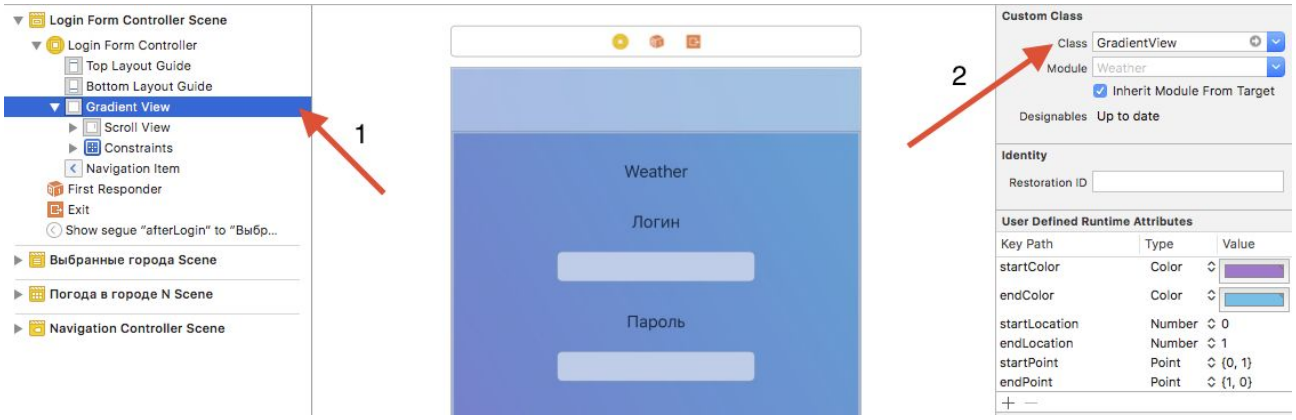
Нужны методы, которые будут обновлять параметры слоя с градиентом:

```
func updateLocations() {
    self.gradientLayer.locations = [self.startLocation as NSNumber,
self.endLocation as NSNumber]
}

func updateColors() {
    self.gradientLayer.colors = [self.startColor.cgColor, self.endColor.cgColor]
}

func updateStartPoint() {
    self.gradientLayer.startPoint = startPoint
}

func updateEndPoint() {
    self.gradientLayer.endPoint = endPoint
}
```

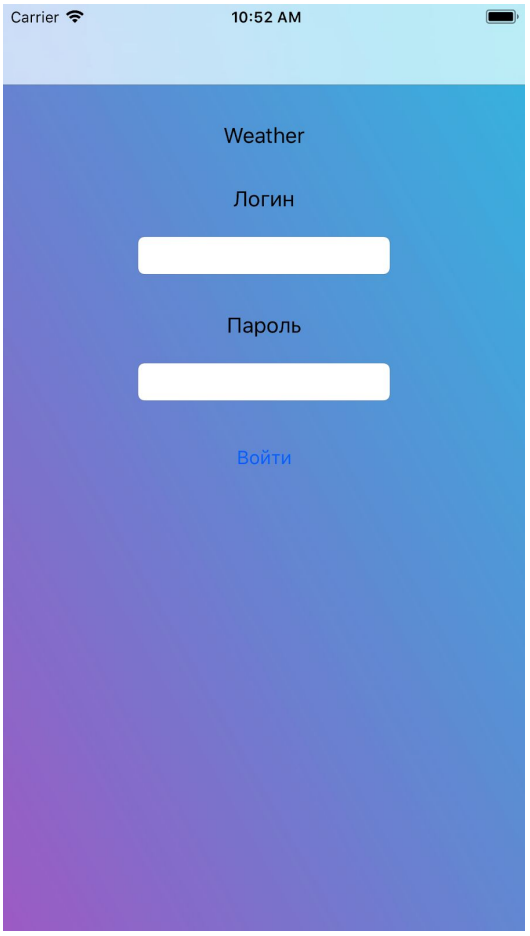
Добавим вызов этих методов после изменения свойств, а также атрибуты **@IBDesignable** и **@IBInspectable** для поддержки редактирования и визуализации **view** в **storyboard**:

```
@IBInspectable var startColor: UIColor = .white {
    didSet {
        self.updateColors()
    }
}
@IBInspectable var endColor: UIColor = .black {
    didSet {
        self.updateColors()
    }
}

@IBInspectable var startLocation: CGFloat = 0 {
    didSet {
        self.updateLocations()
    }
}
@IBInspectable var endLocation: CGFloat = 1 {
    didSet {
        self.updateLocations()
    }
}

@IBInspectable var startPoint: CGPoint = .zero {
    didSet {
        self.updateStartPoint()
    }
}
@IBInspectable var endPoint: CGPoint = CGPoint(x: 0, y: 1) {
    didSet {
        self.updateEndPoint()
    }
}
```

Компонент готов. Можно зайти в **main.storyboard**, выбрать экран авторизации и изменить класс корневого **view** на **GradientView**:



После можно изменить параметры. Результат:



Итоговый код выглядит так:

```
import UIKit

@IBDesignable class GradientView: UIView {

    @IBInspectable var startColor: UIColor = .white {
        didSet {
            self.updateColors()
        }
    }
    @IBInspectable var endColor: UIColor = .black {
        didSet {
            self.updateColors()
        }
    }

    @IBInspectable var startLocation: CGFloat = 0 {
        didSet {
            self.updateLocations()
        }
    }
    @IBInspectable var endLocation: CGFloat = 1 {
        didSet {
            self.updateLocations()
        }
```

```
    }

    @IBInspectable var startPoint: CGPoint = .zero {
        didSet {
            self.updateStartPoint()
        }
    }
    @IBInspectable var endPoint: CGPoint = CGPoint(x: 0, y: 1) {
        didSet {
            self.updateEndPoint()
        }
    }

    override static var layerClass: AnyClass {
        return CAGradientLayer.self
    }

    var gradientLayer: CAGradientLayer {
        return self.layer as! CAGradientLayer
    }

    func updateLocations() {
        self.gradientLayer.locations = [self.startLocation as NSNumber,
self.endLocation as NSNumber]
    }

    func updateColors() {
        self.gradientLayer.colors = [self.startColor.cgColor,
self.endColor.cgColor]
    }

    func updateStartPoint() {
        self.gradientLayer.startPoint = startPoint
    }

    func updateEndPoint() {
        self.gradientLayer.endPoint = endPoint
    }

}
```

Создание контрола «Выбор дня недели»

В качестве примера рассмотрим кастомный контрол для выбора дня недели. Он должен содержать сокращенные названия семи дней недели с возможностью выбора одного. Также должна быть функция «подписаться на изменения» с помощью метода **addTarget**.

Создадим перечисление, содержащее все дни недели. Сделаем его типа **Int**, тогда каждый элемент будет иметь порядковый номер в неделе. Также нужно добавить переменную **title**, которая будет возвращать сокращенное название дня. В итоге код перечисления будет выглядеть так:

```
enum Day: Int {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
    case sunday

    static let allDays: [Day] = [monday, tuesday, wednesday, thursday, friday,
saturday, sunday]

    var title: String {
        switch self {
        case .monday: return "ПН"
        case .tuesday: return "ВТ"
        case .wednesday: return "СР"
        case .thursday: return "ЧТ"
        case .friday: return "ПТ"
        case .saturday: return "СБ"
        case .sunday: return "ВС"
        }
    }
}
```

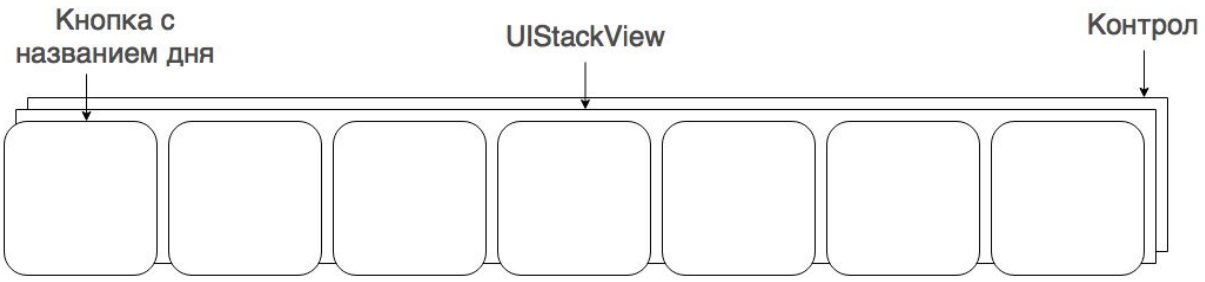
Теперь можно приступить к разработке контрола. Сначала создадим класс, который будет наследником **UIControl**:

```
@IBDesignable class WeekDayPicker: UIControl {

}
```

Чтобы сохранять выбранный день, создадим переменную **selectedDay** в классе:

```
var selectedDay: Day? = nil
```

Схематически изобразим структуру будущего компонента:



Контрол будет состоять из семи кнопок, которые находятся в **UIStackView**. Он используется для простоты расстановки кнопок — расставлять и изменять констрейнты было бы гораздо дольше. Необходимо хранить в классе **UIStackView** и массив кнопок:

```
private var buttons: [UIButton] = []
private var stackView: UIStackView!
```

Добавим метод, отвечающий за создание **UIStackView** и кнопок, а также метод, который будет обновлять интерфейс в соответствии с выбранной кнопкой:

```
private func setupView() {
    for day in Day.allDays {
        let button = UIButton(type: .system)
        button.setTitle(day.title, for: .normal)
        button.setTitleColor(.lightGray, for: .normal)
        button.setTitleColor(.white, for: .selected)
        button.addTarget(self, action: #selector(selectDay(_:)), for:
.touchUpInside)
        self.buttons.append(button)
    }

    stackView = UIStackView(arrangedSubviews: self.buttons)

    self.addSubview(stackView)

    stackView.spacing = 8
    stackView.axis = .horizontal
    stackView.alignment = .center
    stackView.distribution = .fillEqually
}

private func updateSelectedDay() {
    for (index, button) in self.buttons.enumerated() {
        guard let day = Day(rawValue: index) else { continue }
        button.isSelected = day == self.selectedDay
    }
}
```

Заметьте, что мы не установили констрейнты для **UIStackView**. Вместо них будем менять **frame** в методе **layoutSubviews**:

```
override func layoutSubviews() {
    super.layoutSubviews()
    stackView.frame = bounds
}
```

Метод **setupView** нужно вызвать в инициализаторах класса:

```
override init(frame: CGRect) {
     super.init(frame: frame)
    self.setupView()
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    self.setupView()
}
```

Метод **updateSelectedDay** вызывается каждый раз при изменении свойства **selectedDay**. Это можно сделать, добавив метод **didSet**:

```
var selectedDay: Day? = nil {
    didSet {
        self.updateSelectedDay()
    }
}
```

Чтобы обработать нажатия на кнопки, создадим метод **selectDay**, который указывали, когда добавляли таргет к кнопке:

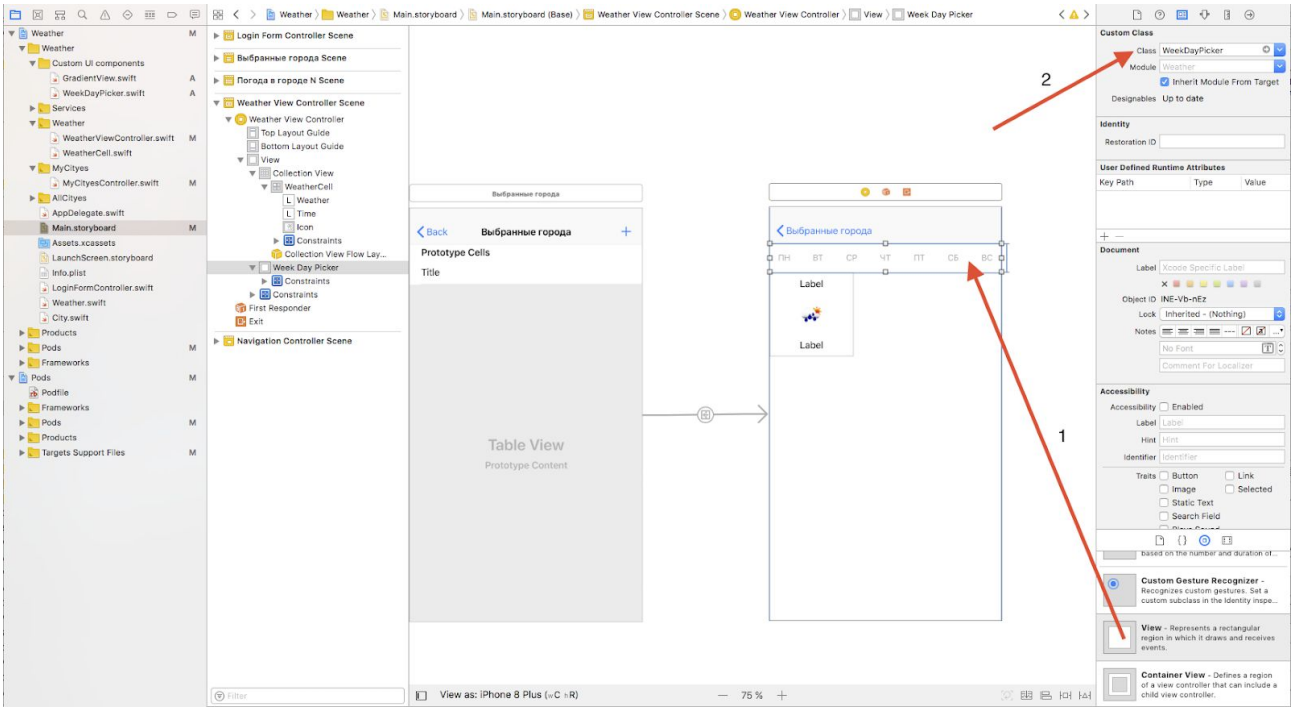
```
@objc private func selectDay(_ sender: UIButton) {
    guard let index = self.buttons.index(of: sender) else { return }
    guard let day = Day(rawValue: index) else { return }
    self.selectedDay = day
}
```

В этом методе сначала получаем индекс кнопки (который является номером дня), затем пытаемся получить день, передав в инициализатор его номер. После присваиваем новый день свойству **selectedDay**.

Осталось вызвать метод **sendActions**, который сообщит об изменении значения контрола. Это нужно сделать после изменения свойства, то есть в методе **didSet**:

```
var selectedDay: Day? = nil {
    didSet {
        self.updateSelectedDay()
        self.sendActions(for: .valueChanged)
    }
}
```

На этом разработка контрола закончена. Теперь надо встроить его в экран погоды. Но нельзя добавлять дополнительные **UIView** на **UICollectionViewController**. На этом примере, вы можете лично убедиться, что его использование приводит к проблемам. Чтобы продолжить, переделайте экран погоды в обычный **UIViewController** со встроенной **UIColletionView**, как было показано во втором уроке. Затем добавим **UIView** выше коллекции и установим класс **WeekDayPicker**:



Также нужно создать **IBOutlet** в классе **WeatherViewController** и связать его с созданным **view**. После можно запустить приложение и увидеть результат:

Код полностью:

```
enum Day: Int {
    case monday, tuesday, wednesday, thursday, friday, saturday, sunday

    static let allDays: [Day] = [monday, tuesday, wednesday, thursday, friday,
saturday, sunday]

    var title: String {
        switch self {
        case .monday: return "ПН"
        case .tuesday: return "ВТ"
        case .wednesday: return "СР"
        case .thursday: return "ЧТ"
        case .friday: return "ПТ"
        case .saturday: return "СБ"
        case .sunday: return "ВС"
        }
    }
}

@IBDesignable class WeekDayPicker: UIControl {

    var selectedDay: Day? = nil {
        didSet {
            self.updateSelectedDay()
            self.sendActions(for: .valueChanged)
        }
    }

    private var buttons: [UIButton] = []
    private var stackView: UIStackView!

    override init(frame: CGRect) {
        super.init(frame: frame)
        self.setupView()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        self.setupView()
    }

    private func setupView() {
        for day in Day.allDays {
            let button = UIButton(type: .system)
            button.setTitle(day.title, for: .normal)
            button.setTitleColor(.lightGray, for: .normal)
            button.setTitleColor(.white, for: .selected)
            button.addTarget(self, action: #selector(selectDay(_:)), for:
.touchUpInside)
            self.buttons.append(button)
        }
```

```
        stackView = UIStackView(arrangedSubviews: self.buttons)

        self.addSubview(stackView)

        stackView.spacing = 8
        stackView.axis = .horizontal
        stackView.alignment = .center
        stackView.distribution = .fillEqually
    }

    private func updateSelectedDay() {
        for (index, button) in self.buttons.enumerated() {
            guard let day = Day(rawValue: index) else { continue }
            button.isSelected = day == self.selectedDay
        }
    }

    @objc private func selectDay(_ sender: UIButton) {
        guard let index = self.buttons.index(of: sender) else { return }
        guard let day = Day(rawValue: index) else { return }
        self.selectedDay = day
    }

    override func layoutSubviews() {
        super.layoutSubviews()
        stackView.frame = bounds
    }

}
```

Практическое задание

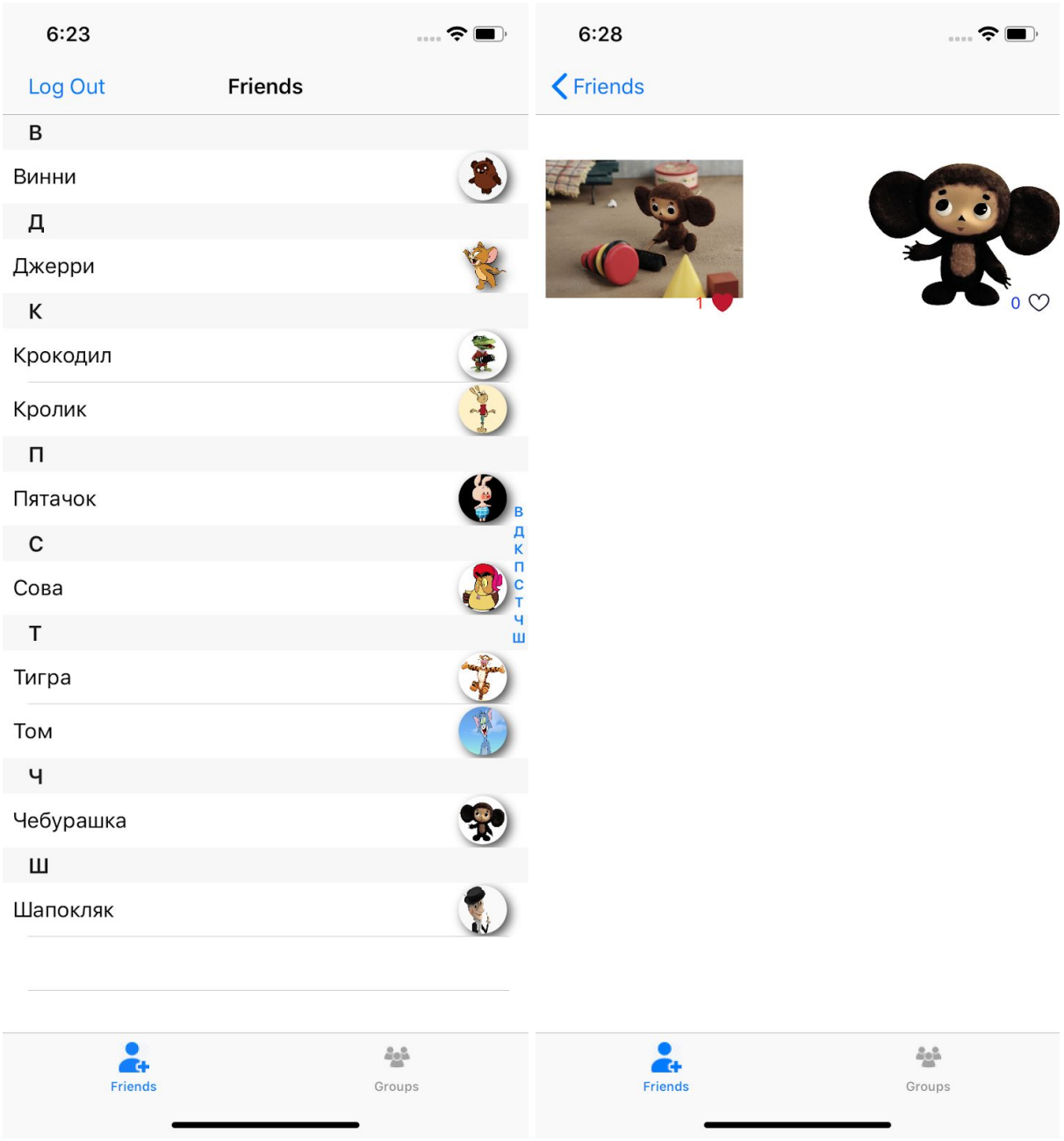
На основе предыдущего ПЗ:

1. Создать свой **View** для аватарки. Он должен состоять из двух **subview**:
    a. Должен содержать **UIImageView** с фотографией пользователя и быть круглой формы.
    b. Должен находиться ниже и давать тень по периметру фотографии. Учтите, что тень будет отображена, если **backgroudColor != .clear**.
    Предусмотреть возможность изменения ширины, цвета, прозрачности тени в **interface builder**. (Задание на самостоятельный поиск решения.)
2. Создать контрол «Мне нравится», с помощью которого можно поставить лайк под постом в ленте. Данный контрол должен объединять кнопку с иконкой сердца и количеством отметок рядом с ней. При нажатии на контрол нужно увеличить количество отметок, а при повторном нажатии — уменьшить (как это реализовано в приложении «ВКонтакте»). В состоянии, когда отметка поставлена, иконка и текст должны менять цвет.
3. * Сделать контрол, позволяющий выбрать букву алфавита. Он понадобится на экране списка друзей. Дизайн можно позаимствовать у оригинального приложения «ВКонтакте». Должна быть возможность выбрать букву нажатием или перемещением пальца по контролу. При

выборе нужно пролистывать список к первому человеку, у которого фамилия начинается на эту букву.

Желательно сделать так, чтобы в этом контроле не было букв, на которые не начинается ни одна фамилия друзей из списка. (Необязательная часть задания — для тех, у кого есть время).

Пример выполненной работы



Дополнительные материалы

1. UIAppearance Tutorial: Getting Started.
2. Core Graphics.

3. How to build a custom (and “designable”) control in Swift.
4. Core Graphics Tutorial Part 1: Getting Started.
5. Core Graphics Tutorial Part 2: Gradients and Contexts.
6. Core Graphics Tutorial Part 3: Patterns and Playgrounds.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Implement a Custom Control.
2. Core Graphics.
3. UIGestureRecognizer.
4. UIAppearance.