GeekBrains

Пользовательский интерфейс iOS-приложений

Анимация переходов между экранами

Анимация переходов между UIViewController. Интерактивные переходы между view controllers. Создание собственного segue.

Оглавление

Анимация переходов между экранами

Мы уже рассмотрели большинство техник, с помощью которых можно делать анимации на iOS. Разберемся, как сделать анимированные переходы между экранами. Самый узнаваемый способ — анимация перехода в **UINavigationController**, которая называется **push**. Но чтобы создать приложение с самобытным стилем, нужно добавлять не только собственные UI-компоненты, но и разрабатывать переходы между экранами.

Анимация простого перехода

Разберемся, как работает анимация перехода между экранами и как создать ее. Сначала вызываем метод **func present(_ viewControllerToPresent: UIViewController, animated flag: Bool, completion: (() -> Void)? = nil)**, который презентует новый **UIViewController**. При каждом вызове он проверяет наличие у **UIViewController**, из которого происходит презентация, кастомного **transitioningDelegate**. Это свойство **UIViewController**, которое имеет тип **UIViewControllerTransitioningDelegate**. Ему нужно присвоить свой объект-аниматор, чтобы анимации работали. С помощью этого делегата можно переопределить анимацию. Рассмотрим два его метода:

```
func animationController(
    forPresented presented: UIViewController,
    presenting: UIViewController,
    source: UIViewController) -> UIViewControllerAnimatedTransitioning? {}

func animationController(
    forDismissed dismissed: UIViewController) ->
UIViewControllerAnimatedTransitioning? {}
```

Первый метод возвращает объект, который будет анимировать презентацию нового экрана. Параметр № 1 — это презентуемый **UIViewController**, № 2 — тот **UIViewController**, который будет презентовать новый контроллер. Третий параметр — это тот **UIViewController**, у которого был вызван метод **present**.

Второй метод возвращает объект, который будет анимировать закрытие экрана. В качестве параметра он содержит **UIViewController**, который будет скрыт.

Чтобы создать объект, который будет анимировать переход, нужно добавить класс-аниматор и реализовать протокол **UIViewControllerAnimatedTransitioning**. У этого протокола два обязательных метода и два опциональных (один из которых мы рассматривать не будем):

```
func transitionDuration(using transitionContext:
UIViewControllerContextTransitioning?) -> TimeInterval {
    return 0.5
}

func animateTransition(using transitionContext:
UIViewControllerContextTransitioning) {}

func animationEnded(_ transitionCompleted: Bool) {}
```

Первый метод — обязательный, он возвращает длительность анимации.

Второй — тоже обязательный, он анимирует переход. В качестве параметра в него попадает контекст перехода. Это объект, который содержит информацию о переходе, а также помогает сообщить статус перехода.

Третий метод — опциональный: выполняется, когда закончился переход. Параметр **transitionCompleted** сообщает, выполнился ли переход.

Разберемся, как работать со вторым методом. Сначала нужно получить объекты **UIViewController**, между которыми происходит переход. Сделать это можно с помощью **transitionContext** и его метода **viewController(forKey: UITransitionContextViewControllerKey)**:

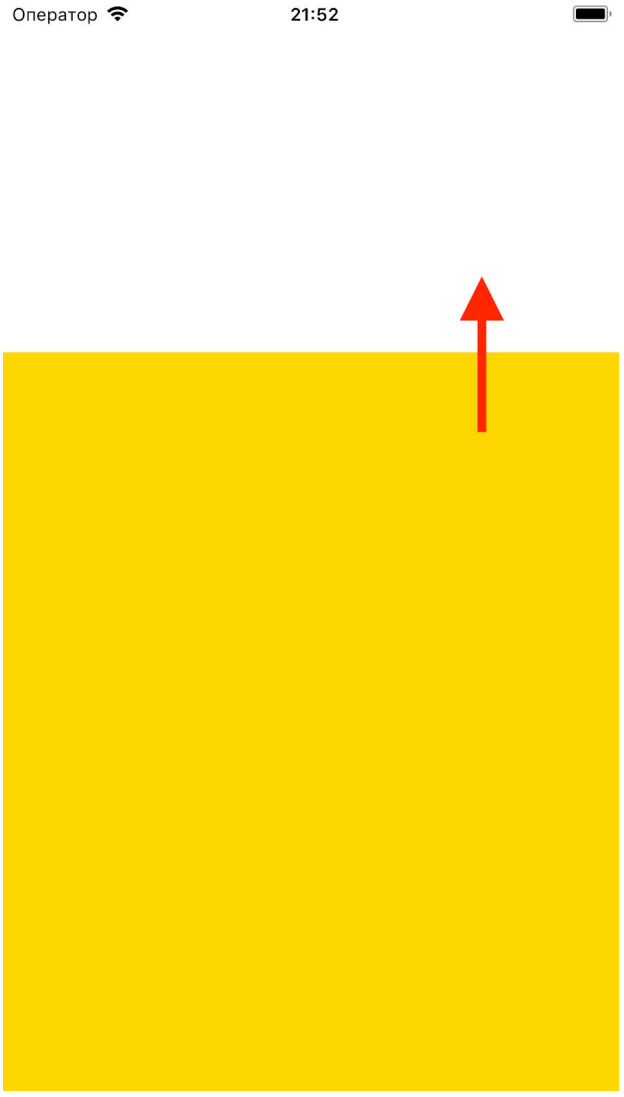
```
guard let source = transitionContext.viewController(forKey: .from)
else { return }
guard let destination = transitionContext.viewController(forKey: .to)
else { return }
```

После этого можно воспользоваться обычными методами **UIView.animate** для анимации перехода. В качестве примера рассмотрим переход, когда первый экран «уезжает» вверх, а второй перемещается снизу на его место:

```
let containerViewFrame = transitionContext.containerView.frame
let sourceViewTargetFrame = CGRect(x: 0,
                                   y: -containerViewFrame.height,
                                   width: source.view.frame.width,
                                   height: source.view.frame.height)
let destinationViewTargetFrame = source.view.frame

transitionContext.containerView.addSubview(destination.view)

destination.view.frame = CGRect(x: 0,
                                y: containerViewFrame.height,
                                width: source.view.frame.width,
                                height: source.view.frame.height)


UIView
    .animate(withDuration: self.transitionDuration(using: transitionContext),
             animations: {
                source.view.frame = sourceViewTargetFrame
                destination.view.frame = destinationViewTargetFrame
}) { finished in
    source.removeFromParent()
    transitionContext.completeTransition(finished)
}
```

Сначала получаем **frame** контейнера — это **view**, в котором происходит переход. Потом задаем конечный **frame** для обоих экранов. После этого добавляем **view** второго экрана в контейнер и задаем начальный **frame**. Создаем обычную анимацию, в которой меняем **frame** обоих **view**. В блоке завершения анимации удаляем старый экран и завершаем переход с помощью метода **completeTransition(didComplete: Bool)**.

В результате получится такой переход:



Анимация перехода между экранами в UINavigationController

У **UINavigationController** два метода для реализации переходов: **push** и **pop**. У них есть анимация по умолчанию, но ее можно изменить — реализовать протокол **UINavigationControllerDelegate** и переопределить следующий метод:

```
func navigationController(
    _ navigationController: UINavigationController,
    animationControllerFor operation: UINavigationController.Operation,
    from fromVC: UIViewController,
    to toVC: UIViewController) -> UIViewControllerAnimatedTransitioning? {}
```

Этот метод вызывается в момент перехода от одного экрана к другому. В качестве возвращаемого нужно передать такой же объект, который передавали для анимации обычного перехода в предыдущем разделе.

Разберем параметры:

- **navigationController** — это **UINavigationController**, который осуществляет переход;

- **animationControllerFor** — это перечисление, обозначающее тип перехода и содержащее два значения: **push** и **pop**;
- два последних параметра — это экраны, между которыми происходит переход.

Реализация этого метода выглядит так:

```
func navigationController(_ navigationController: UINavigationController,
                                                  animationControllerFor  operation:
UINavigationControllerOperation,
                          from fromVC: UIViewController,
                          to toVC: UIViewController)
                          -> UIViewControllerAnimatedTransitioning? {

    if operation == .push {
        return PushAnimator()
    } else if operation == .pop {
        return PopAnimator()
    }

    return nil
}
```

В зависимости от типа перехода возвращается тот или иной аниматор. Чтобы анимация заработала, нужно присвоить свойству **UINavigationController.delegate** нужный делегат:

```
self.navigationController?.delegate      =      //      Объект,      реализующий
UINavigationControllerDelegate
```

Интерактивная анимация переходов между экранами

Часто в приложениях, где есть просмотр фото, можно скрыть экран с фотографией жестом смахивания. Причем он закрывается, только если пользователь смахнул его «на достаточное расстояние». Научимся делать такие красивые и удобные переходы.

Для интерактивного перехода нужно создать класс, который будет наследником **UIPercentDrivenInteractiveTransition**, и реализовывать протокол **UIViewControllerAnimatedTransitioning**. Класс **UIPercentDrivenInteractiveTransition** управляет интерактивным переходом. Он содержит методы **update**, **start** и **finish**. Экземпляр этого объекта нужно вернуть в одном из методов протокола **UIViewControllerTransitioningDelegate**:

```
func interactionControllerForPresentation(
    using animator: UIViewControllerAnimatedTransitioning) ->
UIViewControllerInteractiveTransitioning? {}

func interactionControllerForDismissal(
    using animator: UIViewControllerAnimatedTransitioning) ->
UIViewControllerInteractiveTransitioning? {}
```

Первый метод нужен для реализации интерактивного показа экрана, а второй — для закрытия. Если реализован какой-то из этих методов, необходимо использовать один из следующих:

```
func animationController(
    forPresented presented: UIViewController,
    presenting: UIViewController,
    source: UIViewController) -> UIViewControllerAnimatedTransitioning? {}

func animationController(
    forDismissed dismissed: UIViewController) ->
UIViewControllerAnimatedTransitioning? {}
```

Чтобы интерактивно показывать экраны, нужно реализовать методы **interactionControllerForPresentation** и **animationController(forPresented:presenting:source)**. А чтобы интерактивно их закрывать — **interactionControllerForDismissal** и **animationController(forDismissed)**.

В основе любого интерактивного перехода лежит обработка **UIPanGestureRecognizer**, которая выглядит примерно так:

```
switch recognizer.state {
case .began:
    self.hasStarted = true
case .changed:
    let progress = // Вычисление прогресса на основе перемещения пальца
    self.shouldFinish = progress > threshold
    self.update(progress)
case .cancelled:
    self.hasStarted = false
    self.cancel()
case .ended:
    self.hasStarted = false
    self.shouldFinish
        ? self.finish()
        : self.cancel()
default:
    break
}
```

HasStarted и **shouldFinish** — это переменные для индикации того, начался ли переход и должен ли он закончиться, когда пользователь поднимет палец. Основная часть интерактивного перехода — расчет прогресса, который выполняется на основе перемещения пальца. Например, так:

```
let translation = recognizer.translation(in: recognizer.view)
let estimatedProgress = abs(translation.y / 200)
let progress = min(max(estimatedProgress, 0.01), 0.99)
```

Подробнее реализацию интерактивного перехода рассмотрим в практической части урока.

Создание кастомной segue

Второй способ перехода между экранами — с помощью **segue**. По умолчанию есть несколько типов **segue** со своей анимацией. Но иногда требуется создавать собственные.

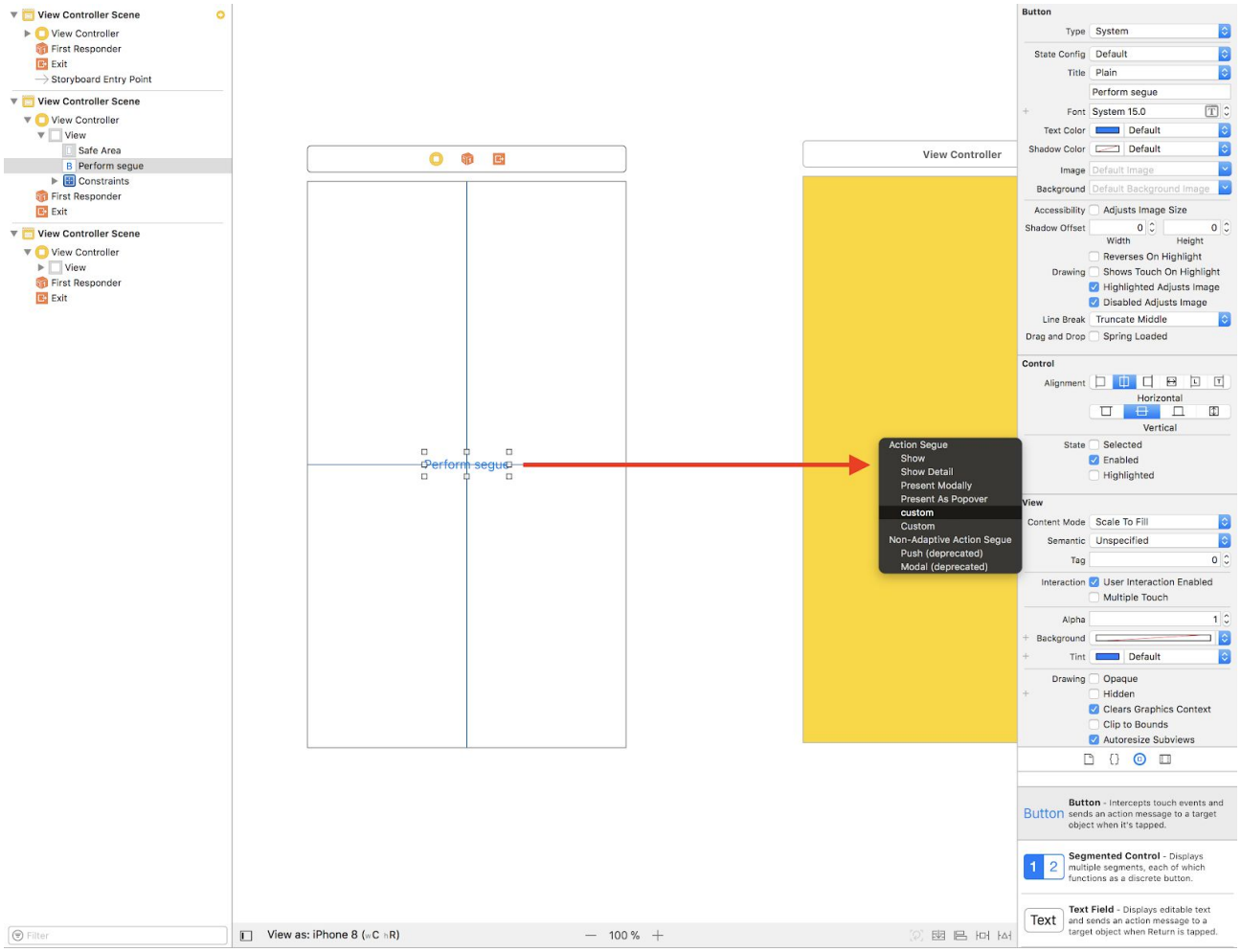
Чтобы сделать свой **segue**, нужно добавить класс-наследник **UIStoryboardSegue** и переопределить метод **perform**. Его реализация похожа на ту, что мы делали в методе **animateTransition(using transitionContext: UIViewControllerContextTransitioning)**, но здесь нет контекста, а все необходимые данные находятся в самом классе.

Рассмотрим создание **segue** на примере из первого раздела:

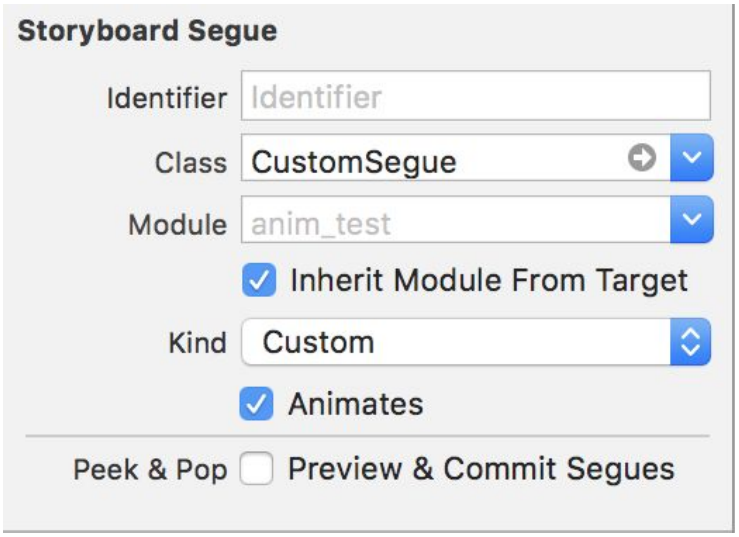
```
class CustomSegue: UIStoryboardSegue {

    override func perform() {
        guard let containerView = source.view.superview else { return }

        let containerViewFrame = containerView.frame
        let sourceViewTargetFrame = CGRect(x: 0,
                                           y: -containerViewFrame.height,
                                           width: source.view.frame.width,
                                           height: source.view.frame.height)
        let destinationViewTargetFrame = source.view.frame

        containerView.addSubview(destination.view)

        destination.view.frame = CGRect(x: 0,
                                        y: containerViewFrame.height,
                                        width: source.view.frame.width,
                                        height: source.view.frame.height)


        UIView
            .animate(withDuration: 0.5,
                     animations: {
                        self.source.view.frame = sourceViewTargetFrame
                        self.destination.view.frame = destinationViewTargetFrame
            }) { finished in
                self.source.present(self.destination,
                                    animated: false,
                                    completion: nil)
        }
    }

}
```

Похоже на код анимации перехода, но есть отличия. Во-первых, в классе **UIStoryboardSegue** нет свойства **containerView** — поэтому в качестве контейнера использовали **superview** того экрана, с которого происходит переход. Во-вторых, по окончании анимации нужно вызвать метод **present**, но без анимации. Это позволит отобразить новый экран.

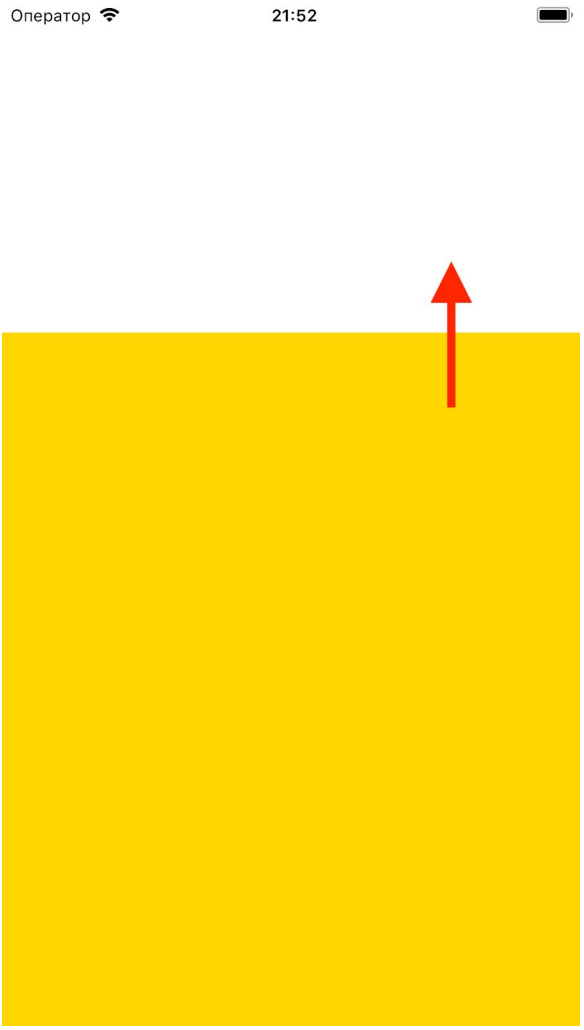
Теперь можно проверить, как работает **segue**. Для этого в **storyboard** создадим **custom segue** между двумя любыми **UIViewController's**:



После этого для **segue** надо указать класс **CustomSegue**:



После этого анимация перехода будет выглядеть так:



Чтобы сделать **segue**, которая будет скрывать экран, нужно заменить метод **present** на **dismiss** в блоке окончания анимации.

Практика

Анимация переходов UINavigationController

В качестве примера создадим анимации переходов между экранами в **UINavigationViewController**. Добавим такой переход, при котором текущий экран отдаляется и смещается в левую сторону, а следующий появляется справа и становится на его место.

Сначала создадим класс аниматора:

```
final class CustomPushAnimator: NSObject, UIViewControllerAnimatedTransitioning
{}
```

Реализуем методы протокола **UIViewControllerAnimatedTransitioning**:

```
func transitionDuration(using transitionContext:
UIViewControllerContextTransitioning?) -> TimeInterval {
    return 0.6
}
```

Задаем длительность анимации — 0,6 секунды.

Создание метода анимации разберем по частям. Сначала получаем текущий и следующий **view controller**:

```
func animateTransition(using transitionContext: UIViewControllerContextTransitioning) {
     guard let source = transitionContext.viewController(forKey: .from) else { return }
    guard let destination = transitionContext.viewController(forKey: .to) else { return }

    // Продолжение
}
```

Затем добавляем следующий **view controller** в контейнер и задаем начальные **frame** и **transform**:

```
transitionContext.containerView.addSubview(destination.view)
destination.view.frame = source.view.frame
destination.view.transform = CGAffineTransform(translationX: source.view.frame.width, y: 0)
```

Изначально следующий **view controller** находится справа за пределами экрана.

Далее необходимо создать keyframe-анимацию, которая будет состоять из трех кадров:

- на первом будет удаляться текущий **view controller**;
- на втором следующий **view controller** будет немного увеличиваться и накрывать текущий;
- на последнем кадре следующий **view controller** полностью накроет текущий.

Создадим каждый **keyframe** отдельно. Первый:

```
UIView.addKeyframe(withRelativeStartTime: 0,
                   relativeDuration: 0.75,
                   animations: {
                       let translation = CGAffineTransform(translationX: -200, y: 0)
                       let scale = CGAffineTransform(scaleX: 0.8, y: 0.8)
                       source.view.transform = translation.concatenating(scale)
})
```

Второй:

```
UIView.addKeyframe(withRelativeStartTime: 0.2,
                   relativeDuration: 0.4,
                   animations: {
                               let translation = CGAffineTransform(translationX:
source.view.frame.width / 2, y: 0)
                       let scale = CGAffineTransform(scaleX: 1.2, y: 1.2)
                                                    destination.view.transform  =
translation.concatenating(scale)
})
```

Третий **keyframe**:

```
UIView.addKeyframe(withRelativeStartTime: 0.6,
                   relativeDuration: 0.4,
                   animations: {
                       destination.view.transform = .identity
})
```

Осталось вставить эти **keyframe** в анимацию и правильно обработать completion-блок:

```
UIView.animateKeyframes(withDuration: self.transitionDuration(using:
transitionContext),
                        delay: 0,
                        options: .calculationModePaced,
                        animations: {
                            UIView.addKeyframe(withRelativeStartTime: 0,
                                               relativeDuration: 0.75,
                                               animations: {
                                                   let translation =
CGAffineTransform(translationX: -200, y: 0)
                                                   let scale =
CGAffineTransform(scaleX: 0.8, y: 0.8)
                                                   source.view.transform =
translation.concatenating(scale)
                            })
                            UIView.addKeyframe(withRelativeStartTime: 0.2,
                                               relativeDuration: 0.4,
                                               animations: {
                                                   let translation =
CGAffineTransform(translationX: source.view.frame.width / 2, y: 0)
                                                   let scale =
CGAffineTransform(scaleX: 1.2, y: 1.2)
                                                   destination.view.transform =
translation.concatenating(scale)
                            })
                            UIView.addKeyframe(withRelativeStartTime: 0.6,
                                               relativeDuration: 0.4,
                                               animations: {
                                                   destination.view.transform =
.identity
                            })
}) { finished in
    if finished && !transitionContext.transitionWasCancelled {
        source.view.transform = .identity
    }
    transitionContext.completeTransition(finished &&
!transitionContext.transitionWasCancelled)
}
```

Заметьте, что в completion-блоке мы возвращаем текущему **view** прежнюю трансформацию, если анимация закончилась и переход не был отменен. Это нужно, чтобы трансформация всегда была правильной, если будем переходить с одного экрана на другой и возвращаться несколько раз.

Далее надо сделать идентичную анимацию для перехода назад. Для этого создадим новый класс-аниматор **CustomPopAnimator**. В методе **animateTransition** изменятся только начальные **frame** и **transform**, а также порядок ключевых кадров. Рассмотрим код анимации целиком:

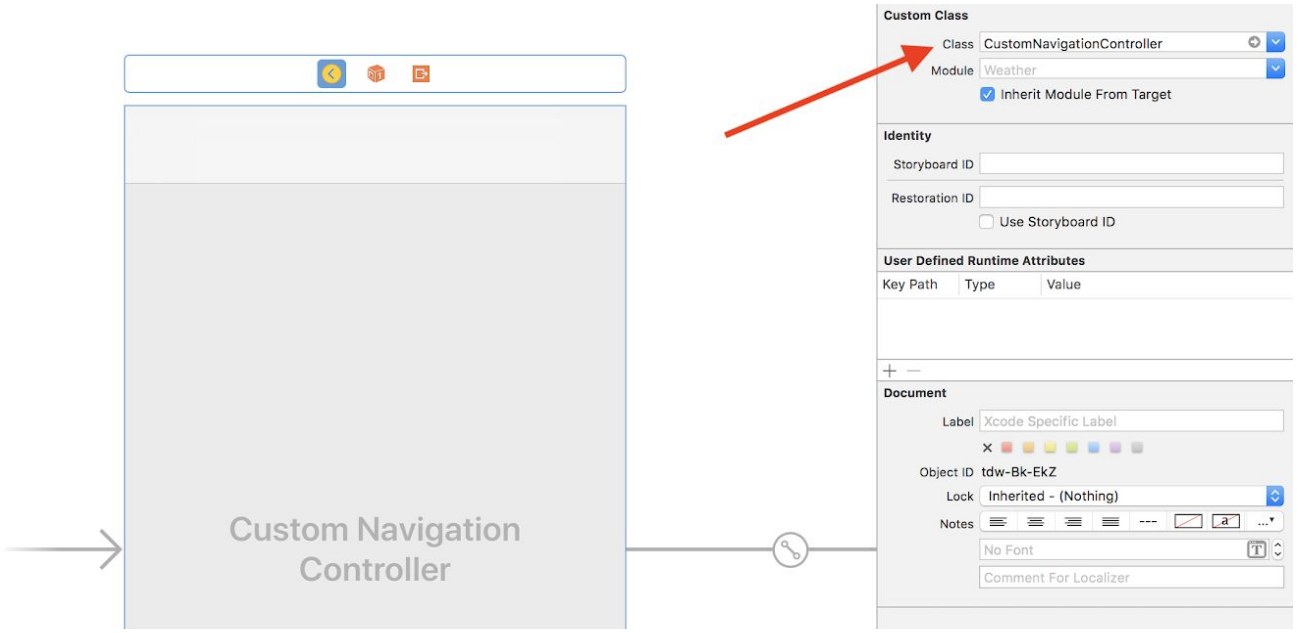
```
func animateTransition(using transitionContext:
UIViewControllerContextTransitioning) {
    guard let source = transitionContext.viewController(forKey: .from) else {
```

```
return }
    guard let destination = transitionContext.viewController(forKey: .to) else {
return }

    transitionContext.containerView.addSubview(destination.view)
    transitionContext.containerView.sendSubviewToBack(destination.view)

    destination.view.frame = source.view.frame

    let translation = CGAffineTransform(translationX: -200, y: 0)
    let scale = CGAffineTransform(scaleX: 0.8, y: 0.8)
    destination.view.transform = translation.concatenating(scale)

    UIView.animateKeyframes(withDuration: self.transitionDuration(using:
transitionContext),
                            delay: 0,
                            options: .calculationModePaced,
                            animations: {
                                UIView.addKeyframe(withRelativeStartTime: 0,
                                                   relativeDuration: 0.4,
                                                   animations: {
                                                       let translation =
CGAffineTransform(translationX: source.view.frame.width / 2, y: 0)
                                                       let scale =
CGAffineTransform(scaleX: 1.2, y: 1.2)
                                                       source.view.transform =
translation.concatenating(scale)
                                })
                                UIView.addKeyframe(withRelativeStartTime: 0.4,
                                                   relativeDuration: 0.4,
                                                   animations: {
                                                       source.view.transform =
CGAffineTransform(translationX: source.view.frame.width, y: 0)
                                })
                                UIView.addKeyframe(withRelativeStartTime: 0.25,
                                                   relativeDuration: 0.75,
                                                   animations: {

destination.view.transform = .identity
                                })


    }) { finished in
        if finished && !transitionContext.transitionWasCancelled {
            source.removeFromParent()
        } else if transitionContext.transitionWasCancelled {
            destination.view.transform = .identity
        }
        transitionContext.completeTransition(finished &&
!transitionContext.transitionWasCancelled)
    }
}
```

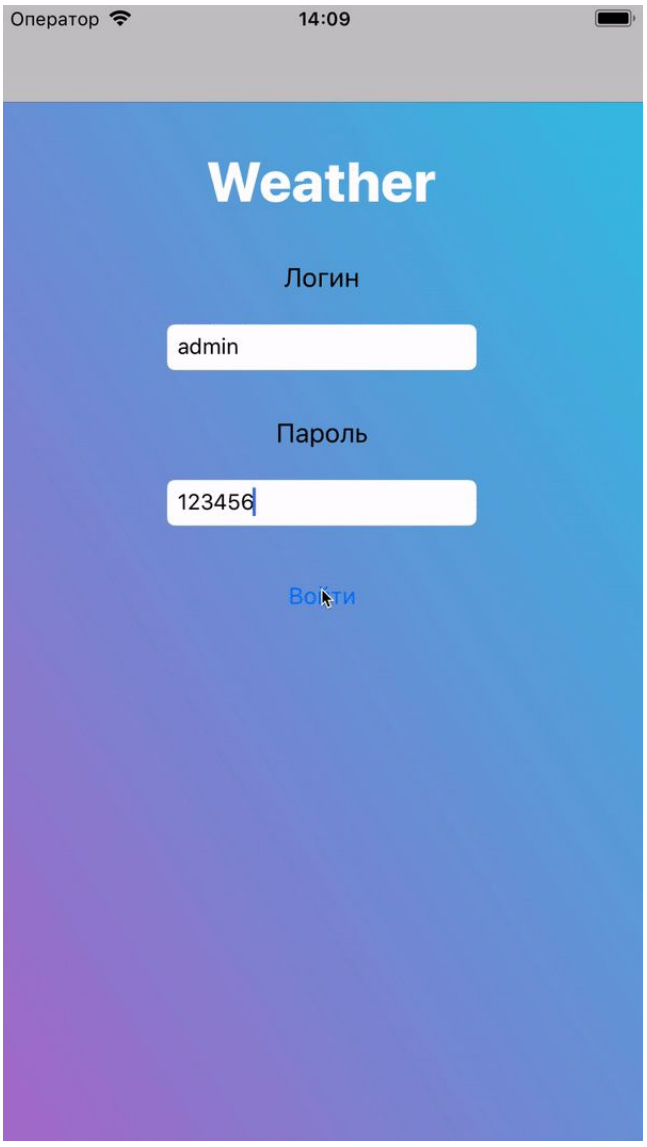
Когда аниматоры готовы, нужно создать **UINavigationControllerDelegate** и добавить их в метод анимации. В качестве делегата будет выступать сам **UINavigationController**, для которого создадим новый класс:

```
class CustomNavigationController: UINavigationController,
UINavigationControllerDelegate {

    override func viewDidLoad() {
        super.viewDidLoad()
        delegate = self
    }

    func navigationController(_ navigationController: UINavigationController,
animationControllerFor operation: UINavigationController.Operation, from fromVC:
UIViewController, to toVC: UIViewController) ->
UIViewControllerAnimatedTransitioning? {
        if operation == .push {
            return CustomPushAnimator()
        } else if operation == .pop {
            return CustomPopAnimator()
        }
        return nil
    }

}
```

Затем установим в **storyboard** класс для **UINavigationController**:



Можно посмотреть результат:



Интерактивное закрытие экрана

Интерактивное закрытие экрана должно работать с перемещением пальца от левой границы экрана, используя анимацию.

Создадим класс, который будет управлять интерактивным закрытием — обрабатывать движение пальца и делать соответствующие изменения в анимации. Он будет наследником **UIPercentDrivenInteractiveTransition**. Напишем код этого класса:

```
class CustomInteractiveTransition: UIPercentDrivenInteractiveTransition {

}
```

Добавим свойство, которое будет хранить **UIViewController** — экран, для которого будет выполняться интерактивный переход:

```
var viewController: UIViewController? {
    didSet {
        let recognizer = UIScreenEdgePanGestureRecognizer(target: self,

action: #selector(handleScreenEdgeGesture(_:)))
        recognizer.edges = [.left]
        viewController?.view.addGestureRecognizer(recognizer)
    }
}
```

Будем добавлять во **view** этого свойства распознаватель жестов. Заметьте, что мы использовали в качестве распознавателя жестов **UIScreenEdgePanGestureRecognizer**, а не **UIPanGestureRecognizer**. Это нужно, чтобы точно повторить механизм стандартного интерактивного перехода и не нарушить работу экранов, в которых уже есть **UIPanGestureRecognizer**.

Осталось реализовать обработку жеста:

```
var hasStarted: Bool = false
var shouldFinish: Bool = false

@objc func handleScreenEdgeGesture(_ recognizer:
UIScreenEdgePanGestureRecognizer) {
    switch recognizer.state {
    case .began:
        self.hasStarted = true
        self.viewController?.navigationController?.popViewController(animated:
true)
    case .changed:
        let translation = recognizer.translation(in: recognizer.view)
        let relativeTranslation = translation.x / (recognizer.view?.bounds.width
?? 1)
        let progress = max(0, min(1, relativeTranslation))

        self.shouldFinish = progress > 0.33

        self.update(progress)
    case .ended:
        self.hasStarted = false
        self.shouldFinish ? self.finish() : self.cancel()
    case .cancelled:
        self.hasStarted = false
        self.cancel()
    default: return
    }
}
```

Рассмотрим реализацию обработки состояний распознавателя.

Когда распознавание началось, меняем свойство **hasStarted** на **true**, обозначая, что интерактивный переход начался. Также в этот момент вызываем метод **popViewController** у **navigationController** экрана, чтобы начать переход.

Когда распознаватель перешел в состояние **changed**, рассчитываем процент, на который изменился переход. Для этого делим координаты текущего положения пальца на ширину **view**, на котором происходит жест. Добавляем ограничения, чтобы это число было не больше 1 и не меньше 0. После этого присваиваем свойству **shouldFinish** значение, которое зависит от прогресса. Если он больше 0,33 — **true**. В завершение вызываем метод **update** с текущим прогрессом.

Когда жест закончился, меняем значение свойства **hasStarted** на **false**. Затем вызываем метод **finish**, если свойство **shouldFinish** равно **true**, или **cancel**, если **false**.

Если переход был отменен, меняем значение свойства **hasStarted** на **false** и вызываем метод **cancel**.

Теперь, когда есть готовый обработчик, можем использовать его в кастомном **navigation controller**. Для этого добавим в него код:

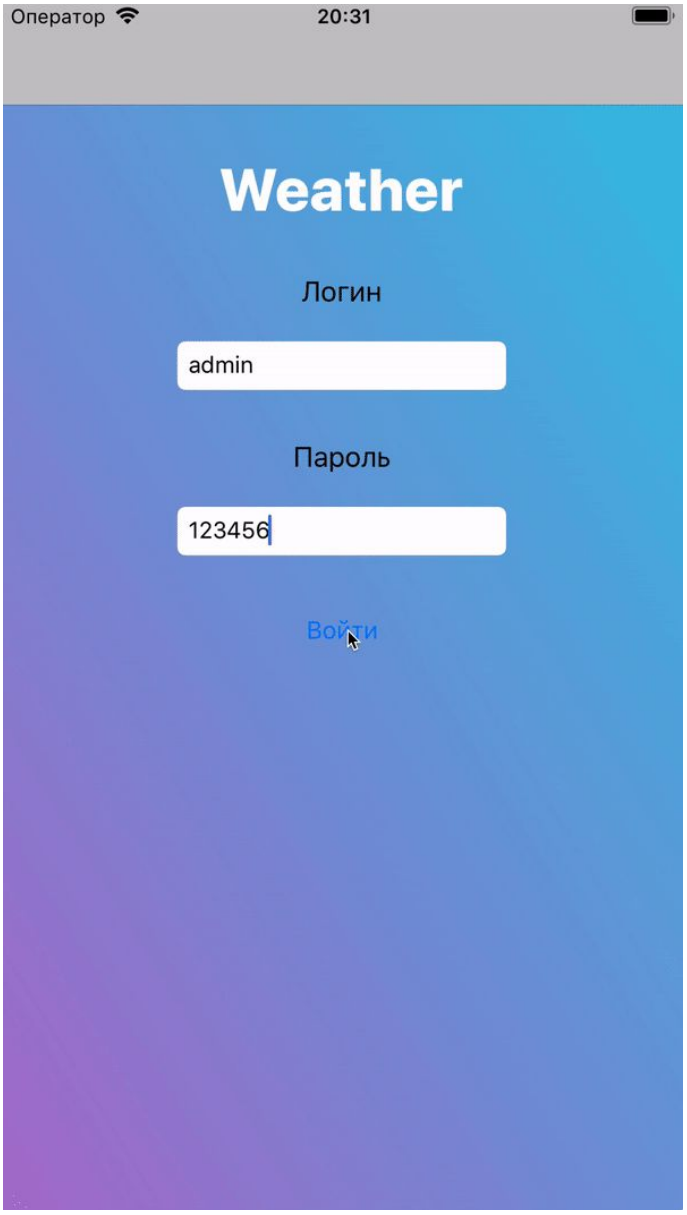
```
let interactiveTransition = CustomInteractiveTransition()

func navigationController(_ navigationController: UINavigationController,
                                    interactionControllerFor animationController:
UIViewControllerAnimatedTransitioning)
                          -> UIViewControllerInteractiveTransitioning? {
    return interactiveTransition.hasStarted ? interactiveTransition : nil
}
```

Создали объект, который будет управлять переходом. В методе делегата проверим свойство **hasStarted**: если **true** — возвращаем объект, если **false** — возвращаем **nil**.

Осталось присвоить свойству объекта тот **viewController**, для которого хотим сделать интерактивный переход. Этот код добавим в метод делегата, где он запрашивает аниматор:

```
func navigationController(_ navigationController: UINavigationController,
                                                 animationControllerFor  operation:
UINavigationController.Operation,
                          from fromVC: UIViewController,
                          to toVC: UIViewController)
                          -> UIViewControllerAnimatedTransitioning? {
    if operation == .push {
        self.interactiveTransition.viewController = toVC
        return CustomPushAnimator()
    } else if operation == .pop {
        if navigationController.viewControllers.first != toVC {
            self.interactiveTransition.viewController = toVC
        }
        return CustomPopAnimator()
    }
    return nil
}
```

Разработка окончена, можно смотреть результат:



Практическое задание

На основе предыдущего ПЗ:

1. Сделать анимацию переходов между экранами в **UINavigationController**. Появляющийся экран сначала находится за пределами видимости и повернут на 90 градусов, при этом его верхний правый угол прижат к такому же углу текущего экрана. В момент перехода появляющийся экран разворачивается относительно верхнего правого угла и становится на место текущего. Анимация закрытия должна выглядеть точно наоборот.
2. Сделать интерактивную анимацию закрытия экрана в **UINavigationController**. В качестве распознавателя жестов использовать **UIScreenEdgePanGestureRecognizer**.
3. * Сделать анимацию показа и закрытия экрана просмотра фотографии. При нажатии картинка увеличивается на весь экран, а при закрытии — уменьшается до исходного размера.
4. * Добавить возможность закрыть экран просмотра фотографии с помощью жеста смахивания вниз.

Примеры выполненных работ

1. Анимация вращающегося перехода с контроллера на контроллер.

Дополнительные материалы

1. Custom UIViewController Transitions: Getting Started.
2. iOS Animation Tutorial: Custom View Controller Presentation Transitions.
3. Introduction to Custom View Controller Transitions and Animations.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. iOS Animation Tutorial: Custom View Controller Presentation Transitions.
2. Customizing the Transition Animations.
3. Creating Custom Presentations.