

Пользовательский интерфейс iOS-приложений

Анимации. Часть I

Простые анимации UIView. Анимации переходов между UIView. Анимации слоя. Пружинные анимации.

Оглавление

[Создание первой анимации](#)

[Простые анимации](#)

[Создание анимации с задержкой](#)

[Анимирование констрейнтов](#)

[Дополнительные параметры анимации](#)

[Создание переходов между view](#)

[Методы для переходов](#)

[Опции анимаций перехода](#)

[Создание анимаций CALayer](#)

[Как работают анимации CALayer](#)

[Создание простой анимации](#)

[Отложенные анимации и свойство fillMode](#)

[Особенности работы анимации слоя](#)

[Пружинные анимации](#)

[Создание пружинной анимации](#)

[Пружинные анимации слоя](#)

[Практика](#)

[Добавление анимаций на экране авторизации](#)

[Практическое задание](#)

[Примеры выполненных работ](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Создание первой анимации

Анимации — неотъемлемая часть качественного приложения. Это способ не только украсить интерфейс, но и сделать его более отзывчивым и понятным для пользователя.

С помощью средств фреймворка **UIKit** можно создавать как простые перемещения визуальных компонентов, так и сложные цепочки анимаций.

Простые анимации

Начнем с самых распространенных анимаций — в **UIView**. Проще всего использовать метод **UIView.animate(duration:animations:)**. Чтобы переместить лейбл на экране на 100 единиц вверх, достаточно написать следующий код:

```
UIView.animate(withDuration: 0.5, animations: {  
    self.label.frame.origin.y -= 100  
})
```

Такая анимация будет длиться половину секунды и уменьшит **origin** лейбла на 100 единиц по оси **y**. В результате получится такое перемещение:



Анимировать можно большинство свойств класса **UIView**:

- **backgroundColor;**
- **bounds;**
- **frame;**
- **center;**
- **transform;**
- **zPosition;**
- **alpha.**

Ниже представлены примеры анимаций некоторых свойств:

- **alpha:**



- **bounds:**



- **center:**



- **backgroundColor:**



Создание анимации с задержкой

Чтобы анимация выполнялась не сразу, а спустя время — например, после предыдущей анимации, — в метод **UIView.animate** можно передать дополнительный параметр **delay**, который отложит анимацию на заданное количество секунд. Добавим еще одну анимацию после той, которая была сделана ранее. Например, изменение свойства **alpha**:

```
UIView.animate(withDuration: 0.5, animations: {
    self.label.frame.origin.y -= 100
})
UIView.animate(withDuration: 0.5, delay: 0.5, animations: {
    self.label.alpha = 0.5
})
```

В этом случае анимации будут выполнены одна за другой, и итоговый вид лейбла будет таким:



Второй способ запустить анимации друг за другом — передать в метод **UIView.animate** параметр **completion**. Это замыкание, которое вызовется по окончании анимации. Туда можно добавить следующую:

```
UIView.animate(withDuration: 0.5, animations: {
    self.label.frame.origin.y -= 100
}, completion: { _ in
    UIView.animate(withDuration: 0.5, animations: {
        self.label.alpha = 0.5
    })
})
```

Результат выполнения этого кода аналогичен предыдущему.

Анимирование констрейнтов

Часто требуется анимировать констрейнты вместо **frame**, **center** или **bounds**. Например, когда нужно переместить лейбл так, чтобы вместе с ним изменились UI-компоненты, привязанные к нему с помощью констрейнтов. Для этого нужно анимировать констрейнты. Это делается иначе, чем со свойствами **view**.

Рассмотрим, что будет, если изменение констрейнта просто добавить в блок анимации:

```
UIView.animate(withDuration: 1, animations: {
    self.constraint.constant = 100
})
```

Этот код не заработает, так как вычисление расположения и размера UI-компонентов происходит в методе **layoutSubviews**. Изменение констрейнта произойдет без анимации.

Чтобы анимировать констрейнт, нужно использовать такую конструкцию:

```
self.view.layoutIfNeeded()
UIView.animate(withDuration: 1, animations: {
    self.constraint.constant = 100
    self.view.layoutIfNeeded()
})
```

Сначала вызывается метод **layoutIfNeeded** у **view**, в котором находятся UI-компоненты, чьи констрейнты изменяются. Это необходимо, чтобы закончить операции **autolayout**, если они есть. Далее вызывается метод анимации, и в нем изменяются констрейнты. Снова запускается **layoutIfNeeded**, чтобы анимировать эти изменения.

Дополнительные параметры анимации

В метод **UIView.animate** можно передать еще один параметр — **options**. Он определяет дополнительные параметры анимации. В него нужно передать параметр типа **UIViewAnimationOptions**, который является перечислением. Рассмотрим его значения:

- **repeat**;
- **autoreverse**;
- **linear**;

- `curveEaseIn`;
- `curveEaseOut`;
- `curveEaseInOut`.

Значение **repeat** позволяет зациклить анимацию. **Autoreverse** — воспроизвести анимацию в прямом, а затем обратном направлении. Это значение работает только в сочетании с **repeat**. Сделаем так, чтобы лейбл поднимался вверх, а затем опускался обратно:

```
UIView.animate(withDuration: 0.2, delay: 0, options: [.repeat, .autoreverse],
  animations: {
    self.label.frame.origin.y -= 100
  })
```

В результате получим бесконечную анимацию, при которой **origin** будет сначала уменьшаться 100 по оси **y**, а затем увеличиваться до исходного значения.

Оставшиеся значения управляют динамикой анимации — скоростью ее течения в разных промежутках времени. Значение **linear** используется по умолчанию и не добавляет динамики. Значения **curveEaseIn** и **curveEaseOut** позволяют замедлить анимацию в начале и в конце соответственно. А **curveEaseInOut** является их комбинацией.

Можно сделать движение лейбла более реалистичным — чтобы он сначала медленно поднимался, а потом плавно останавливался. Для этого напишем следующий код:

```
UIView.animate(withDuration: 2, delay: 0, options: [.curveEaseInOut],
  animations: {
    self.label.frame.origin.y -= 100
  })
```

Создание переходов между view

Чтобы анимировать добавление или удаление **view**, используются переходы (**transitions**).

Методы для переходов

В фреймворке **UIKit** есть два метода для выполнения переходов:

```
UIView.transition(with: UIView,
  duration: TimeInterval,
  options: UIView.AnimationOptions,
  animation: () -> Void,
  completion: ((Bool) -> Void)? = nil)

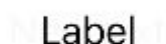
UIView.transition(from: UIView,
  to: UIView,
  duration: TimeInterval,
  options: UIView.AnimationOptions,
  completion: ((Bool) -> Void)? = nil)
```

Они похожи по сигнатуре на метод **UIView.animate**. Первый осуществляет анимацию добавления, удаления, показа или скрытия **view** в контейнере, который передается первым параметром. Второй метод выполняет переход между двумя **view**, которые передаются в первых двух параметрах. По умолчанию он удаляет **from view** и добавляет **to view**.

Рассмотрим пример применения первого метода. Задача — текст в лейбле должен измениться на новый с анимацией. Свойство **text** не анимируется, поэтому можно использовать метод **UIView.transition(with:duration:options:animations:completion):**

```
UIView.transition(with: label,
                  duration: 0.25,
                  options: .transitionCrossDissolve,
                  animations: {
    self.label.text = "New text"
  })
```

В результате анимация будет выглядеть так:



Такой тип анимации определен в параметре **options** с помощью значения **transitionCrossDissolve**. Подробнее об опциях перехода поговорим в следующем разделе.

Рассмотрим второй метод. Допустим, нужно сделать анимированный переход между двумя разными **view** — **UILabel** и **UIImageView**. **UILabel** показывается на экране, а **UIImageView** нужно отобразить на месте **UILabel**. Для этого делаем следующее:

```
UIView.transition(from: label,
                  to: imageView,
                  duration: 0.25,
                  options: .transitionCrossDissolve)
```

Опции анимаций перехода

В качестве опций в методы **UIView.transition** можно передавать следующие значения:

- **transitionFlipFromLeft;**
- **transitionFlipFromRight;**
- **transitionFlipFromTop;**
- **transitionFlipFromBottom;**
- **transitionCurlUp;**
- **transitionCurlDown;**
- **transitionCrossDissolve.**

Первые четыре значения создают анимацию переворота с указанной стороны: например, **transitionFlipFromLeft** — слева направо. Следующие два значения отвечают за эффект переворота книжной страницы, а последнее — за анимацию исчезновения и появления.

Дополнительно можно передавать рассмотренные ранее опции для анимаций. Например, чтобы замедлить анимацию в начале, использовать значение **curveEaseIn**.

Создание анимаций CALayer

Как работают анимации CALayer

Это анимации на более низком уровне. Их преимущество в том, что они выполняются напрямую на **layer**, а не на **view**, и позволяют анимировать больше свойств, чем анимации **UIView**.

Анимации **CALayer** работают так же, как в **UIView** — изменяют значение свойства за определенный промежуток времени. Но так как доступных свойств для анимации больше, расширяются возможности. Вот список свойств, которые можно анимировать у **CALayer**:

- **backgroundColor**;
- **bounds**;
- **frame**;
- **opacity**;
- **center**;
- **transform**;
- **zPosition**;
- **anchorPoint**;
- **backgroundFilters**;
- **compositingFilter**;
- **filters**;
- **borderColor**;
- **borderWidth**;
- **contents**;
- **contentsRect**;
- **cornerRadius**;
- **doubleSided**;
- **mask**;
- **masksToBounds**;

- `opacity;`
- `shadowColor;`
- `shadowRadius;`
- `shadowPath;`
- `shadowOffset;`
- `shadowOpacity;`
- `sublayers;`
- `sublayerTransform.`

Создание простой анимации

Основным классом для создания анимаций **CALayer** является **CABasicAnimation**. Это модель, описывающая, как должна выглядеть анимация. Данный объект можно переиспользовать для разных слоев, сокращая повторяющийся код в приложении. Чтобы запустить анимацию, ее надо добавить на слой с помощью метода **CALayer.add(_ anim: CABasicAnimation, forKey key: String?)**. Параметр **key** является идентификатором анимации. Он нужен, чтобы ее можно было найти среди других.

Создадим анимацию, которую мы добавляли с помощью метода **UIView.animate**, — перемещение **label** на 100 единиц вверх. Для этого напишем такой код:

```
let animation = CABasicAnimation(keyPath: "position.y")
animation.fromValue = layer.frame.origin.y
animation.toValue = layer.frame.origin.y - 100
animation.duration = 0.5
layer.add(animation, forKey: nil)
```

В итоге получим анимацию, идентичную созданной с помощью метода **UIView.animate**.

Отложенные анимации и свойство `fillMode`

Чтобы создать отложенную анимацию, нужно установить свойство **beginTime**. По умолчанию его значение равно **CACurrentMediaTime()**. Этот метод возвращает текущее время — анимация будет запущена сразу после добавления ее на слой. Чтобы отложить анимацию, нужно прибавить необходимое количество секунд к **CACurrentMediumTime()**. Создадим отложенную анимацию исчезновения:

```
let animation = CABasicAnimation(keyPath: #keyPath(CALayer.opacity))
animation.beginTime = CACurrentMediaTime() + 0.5
animation.fromValue = 0.5
animation.toValue = 0
animation.duration = 0.5
layer.add(animation, forKey: nil)
```

Получим анимацию, которая запустится через полсекунды. Обратите внимание, что в этот раз анимируемое свойство **opacity** мы задали при помощи добавленного в Swift 4.2 **#keyPath**. К

сожалению, этот синтаксис не применим к некоторым анимируемым параметрам, как например, **position.y** из прошлой анимации.

Вернемся к **opacity**. Это свойство по умолчанию равно 1, а анимация начинается со значения 0.5. Получим эффект моментальной смены прозрачности, после которой начнется анимация. Чтобы сразу после добавления анимации установить начальное значение **opacity**, равное 0.5, можно использовать свойство **fillMode**. У него четыре значения:

- **kCAFillModeRemoved;**
- **kCAFillModeBackwards;**
- **kCAFillModeForwards;**
- **kCAFillModeBoth.**

Первое используется по умолчанию и означает, что до начала анимации (до **beginTime**) и после нее слой будет выглядеть неизменно.

kCAFillModeBackwards означает, что в момент, когда анимация будет добавлена на слой, и до ее начала будет отображаться ее первый кадр. В нашем случае слой будет выглядеть так, будто его **opacity** равен 0.5.

kCAFillModeForwards — после окончания анимации будет отображаться последний ее кадр. В нашем случае слой будет выглядеть так, будто его **opacity** равен 0.5.

Значение **kCAFillModeBoth** объединяет эффекты свойств **kCAFillModeBackwards** и **kCAFillModeForwards**.

Особенности работы анимации слоя

По окончании анимации слой будет выглядеть как прежде. В случае с перемещением — без измененного **origin**. Это происходит потому, что в момент, когда анимация начинается, создается копия слоя (**snapshot**), которую мы видим на экране, а реальный слой скрывается и появляется после окончания анимации.

По завершении анимации она удаляется из слоя автоматически. Можно изменить это, установив свойство анимации **isRemovedOnCompletion** равным **false** и свойство **fillMode** в значение **kCAFillModeForwards** или **kCAFillModeBoth**. Теперь после анимации увидим слой в конечном состоянии, но это будет не исходник, а слой-копия. Чтобы они имели одинаковое состояние, после добавления анимации нужно установить анимируемые свойства слоя на те же значения, которые они будут иметь по завершении анимации, а свойство **isRemovedOnCompletion** сделать равным **true**.

Старайтесь не оставлять анимации после их завершения, так как это создает лишнюю нагрузку на прорисовку экрана.

Пружинные анимации

Не все анимации ограничиваются простым перемещением, как в предыдущих примерах. Чтобы добавить реалистичности — например, симулируя физические явления, — фреймворк **UIKit** предоставляет дополнительные возможности. В частности, сделать симуляцию пружинных анимаций.

Схематически пружинная анимация выглядит так:



При изменении параметра — например, **center**, — когда объект доходит до конечной точки, он останавливается не сразу, а постепенно, как маятник или пружина.

Создание пружинной анимации

Создать пружинную анимацию можно с помощью следующего метода:

```
UIView.animate(withDuration: TimeInterval,
               delay: TimeInterval,
               usingSpringWithDamping: CGFloat,
               initialSpringVelocity: CGFloat,
               options: UIView.AnimationOptions,
               animations: () -> Void,
               completion: ((Bool) -> Void)? = nil)
```

Параметр **usingSpringWithDamping** определяет «жесткость пружины». Чем выше его значение (может быть в диапазоне от 0 до 1), тем быстрее «успокоится» анимация.

Параметр **initialSpringVelocity** определяет начальную скорость пружины. Значение 1 устанавливает такую скорость анимации, при которой все ее расстояние будет пройдено за одну секунду. Если общее расстояние, на которое перемещается **view**, равно 100 точкам, то при значении данного параметра 0.7 начальная скорость составит 70 точек в секунду.

Добавим эффект пружины в пример с перемещением **label**. Для этого напишем такой код:

```
UIView.animate(withDuration: 0.5,
               delay: 0,
               usingSpringWithDamping: 0.5,
               initialSpringVelocity: 0,
               options: [],
               animations: {
                   self.label.frame.origin.y -= 100
               })
```

```
})
```

Пружинные анимации слоя

Для слоя тоже можно создать пружинную анимацию — для этого есть класс **CASpringAnimation**.

Он содержит два основных свойства — **damping** и **initialVelocity**, а также два дополнительных — **mass** и **stiffness**. Свойство **mass** отражает массу слоя, а **stiffness** — жесткость. По умолчанию у **mass** значение 1, при увеличении объект будет останавливаться быстрее. Свойство **stiffness** обозначает жесткость пружины и по умолчанию равно 100. Возрастая, уменьшает количество колебаний и их продолжительность.

Пример создания простой пружинной анимации слоя:

```
let animation = CASpringAnimation(keyPath: "position.x")
animation.fromValue = 100
animation.toValue = 200
animation.stiffness = 200
animation.mass = 0.5
animation.duration = 2
```

Практика

Добавление анимаций на экране авторизации

Экран авторизации встречает пользователя при входе в приложение и должен заинтересовать его. Создадим анимации для всех элементов на этом экране. Начнем с надписей над полями ввода логина и пароля — сделаем простой вылет из краев.

Лейблы должны переместиться из точки вне экрана туда, где они будут отображаться постоянно. Для этого будем использовать трансформации — это удобно тем, что изначально нужно создать трансформацию для «сдвинутого» состояния, а в качестве конечной можно использовать **CGAffineTransform.identity**.

Анимация будет длиться 1 секунду и начинаться с задержкой в 1 секунду, чтобы пользователь успел ее увидеть.

Напишем код анимации:

```
func animateTitlesAppearing() {
    let offset = view.bounds.width
    loginTitleView.transform = CGAffineTransform(translationX: -offset, y: 0)
    passwordTitleView.transform = CGAffineTransform(translationX: offset, y: 0)

    UIView.animate(withDuration: 1,
                    delay: 1,
                    options: .curveEaseOut,
                    animations: {
                        self.loginTitleView.transform = .identity
                        self.passwordTitleView.transform = .identity
                    },
                    completion: nil)
}
```

Для заголовка **Weather** создадим пружинную анимацию, будто он опускается немного ниже своего конечного местоположения, а затем возвращается к нему.

Для этого установим его начальную трансформацию, а в блоке анимации — значение **transform**, равное **.identity**. Значение **damping** установим в 0.5, чтобы получить средний «эффект маятника». В **InitialSpringVelocity** пропишем 0, так как не будет изначальной скорости. Теперь код:

```
func animateTitleAppearing() {
    self.titleView.transform = CGAffineTransform(translationX: 0,
                                                    y: -self.view.bounds.height/2)

    UIView.animate(withDuration: 1,
                    delay: 1,
                    usingSpringWithDamping: 0.5,
                    initialSpringVelocity: 0,
                    options: .curveEaseOut,
                    animations: {
                        self.titleView.transform = .identity
                    },
                    completion: nil)
}
```

Для полей ввода применим анимацию постепенного появления и будем использовать анимации слоя.

Будем изменять свойство **opacity** у **CALayer** с 0 до 1. Начальные значения выставлять не станем, а воспользуемся свойством **fillMode** и установим его равным **kCAFillModeBackwards** — чтобы до момента начала анимации отображался первый ее кадр, то есть прозрачный **layer**. Это даст тот же эффект, что и установка «нулевого» **opacity** до начала анимации.

Напишем код этой анимации и применим ее к слоям полей ввода:

```
func animateFieldsAppearing() {
    let fadeInAnimation = CABasicAnimation(keyPath: "opacity")
    fadeInAnimation.fromValue = 0
    fadeInAnimation.toValue = 1
    fadeInAnimation.duration = 1
    fadeInAnimation.beginTime = CACurrentMediaTime() + 1
    fadeInAnimation.timingFunction = CAMediaTimingFunction(name:
CAMediaTimingFunctionName.easeOut)
    fadeInAnimation.fillMode = CAMediaTimingFillMode.backwards

    self.loginView.layer.add(fadeInAnimation, forKey: nil)
    self.passwordView.layer.add(fadeInAnimation, forKey: nil)
}
```

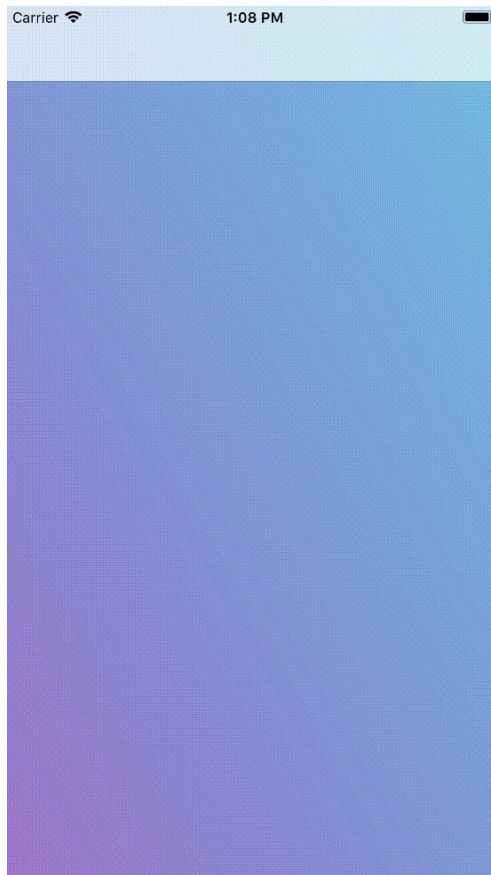
К кнопке «Войти» применим пружинную анимацию увеличения и тоже будем использовать анимации слоя.

Эффекта увеличения можно достичь, изменяя свойство по ключу **transform.scale** от 0 до 1. Увеличим вдвое от стандарта свойства **stiffness** и **mass**: первому установим значение 200, второму — 2. Так эффект маятника будет длиться немного дольше. Напишем код этой анимации:

```
func animateAuthButton() {
    let animation = CASpringAnimation(keyPath: "transform.scale")
    animation.fromValue = 0
    animation.toValue = 1
    animation.stiffness = 200
    animation.mass = 2
    animation.duration = 2
    animation.beginTime = CACurrentMediaTime() + 1
    animation.fillMode = CAMediaTimingFillMode.backwards

    self.authButton.layer.add(animation, forKey: nil)
}
```

Все анимации готовы — можно запустить проект и посмотреть результат:



Практическое задание

На основе предыдущего ПЗ:

1. Создать индикатор загрузки, который будет состоять из трех точек, меняющих прозрачность по очереди.
2. Добавить анимацию нажатия на аватарку пользователя/группы в соответствующих таблицах. По нажатию фотография должна немного сжиматься, а после — возвращаться к исходному размеру с эффектом пружины. Нужно подобрать оптимальное время анимации, чтобы получить максимально реалистичный эффект.
3. Сделать анимацию изменения количества отметок «Мне нравится». Это может быть любая анимация: переворот из стороны в сторону, плавная смена или перелистывание.
4. * Сделать анимацию появления и исчезновения ячеек с фотографиями. Перед показом ячейки она должна увеличиваться и становиться непрозрачной, а перед исчезновением — уменьшаться и становиться прозрачной. (Необязательное задание — для тех, у кого есть время.)
5. * Сделать кастомную строку поиска (как **UISearchBar**). Посередине должна находиться иконка лупы. Когда строку поиска активируют, лупа перемещается в сторону и останавливается с эффектом пружины. Также в этот момент строка поиска укорачивается с правой стороны и на пустом месте появляется кнопка отмены. Все это происходит анимированно. Когда поиск отменяется или с его строки снимается фокус, она должна вернуться в исходное состояние. (Необязательное задание — для тех, у кого есть время.)

Примеры выполненных работ

1. [Анимация трех точек, индикатор загрузки.](#)
2. [Нажатие на аватар ячейки.](#)
3. [Разбивка на секции и анимация появления ячеек.](#)
4. [Анимация строки поиска.](#)

Дополнительные материалы

1. [Анимации в iOS для начинающих. Модели, классы от Core Animation, блоки.](#)
2. [Creating Simple View Animations in Swift.](#)
3. [UI Animations with Swift.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Animations.](#)
2. [Creating Simple View Animations in Swift.](#)
3. [animateWithDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:](#)