Разработка под iOS. Начинаем

Часть 3

View

Оглавление

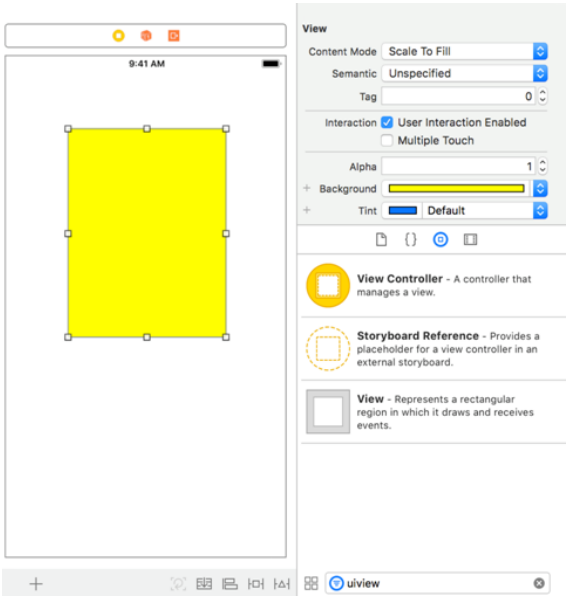
Глава 1

Разработка под iOS. Начинаем

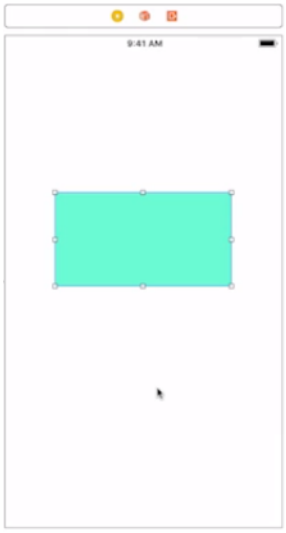
1.1. Представления = UIView

На этой неделе мы рассмотрим представления, или UIView. Мы разберём, что это такое, что он умеет, где найти подходящий системный элемент, как и зачем строится иерархия, и напишем немного кода.

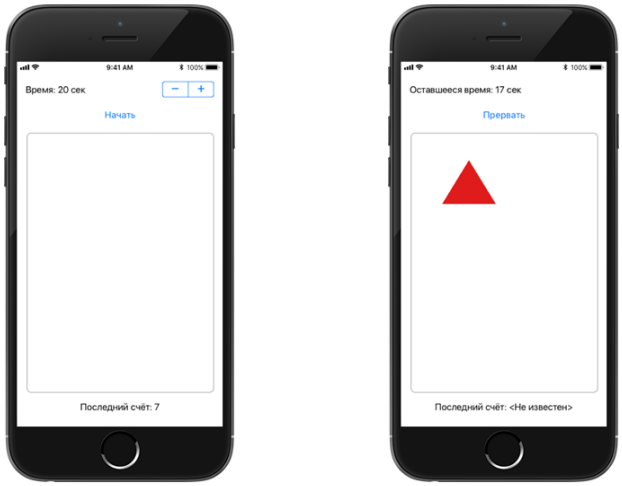
UIView – это визуальное представление модели приложения (информации). Как можно догадаться из названия, представления нужны для представления приложения, или иначе – пользовательского интерфейса. В iOS базовой единицей представления является класс UIView. В базовом виде он выглядит как прямоугольник заданного цвета.



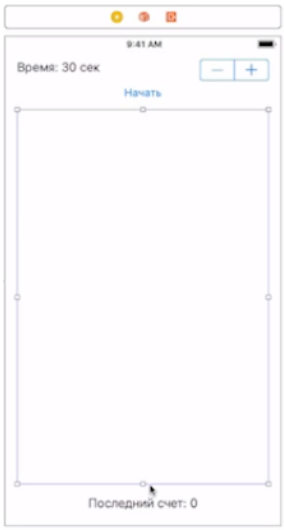
Давайте создадим проект и посмотрим на этот строительный блок в Interface Builder-е. Запускаем Xcode, создаем новый проект Views, открываем Main.storyboard. В списке объектов ищем UIView и перетаскиваем в основное окно приложения. Появился белый прямоугольник на белом фоне. Давайте изменим цвет прямоугольника. Это, как и отредактировать некоторые другие свойства, можно в панели свойств. Изменяем Background, выбираем Ocean. Отлично, наш прямоугольник стал зелёным. Разноцветные прямоугольники – это прекрасно. Но только из них будет сложно построить красивый пользовательский интерфейс.



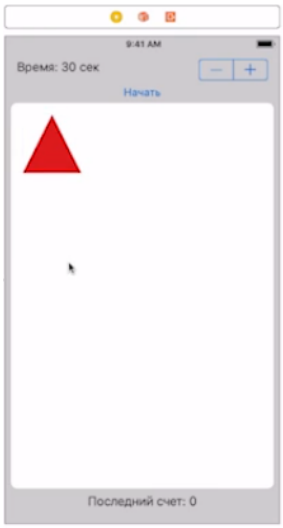
Что же ещё умеет UIView?
Во-первых, он умеет рисовать внутри своей области графические объекты. Самостоятельно рисовать внутри вьюшек мы будем чуть позже, а пока что станем использовать системные объекты-наследники UIView. Во-вторых, он может содержать в себе другие вьюшки, образуя иерархию. В-третьих, он может принимать пользовательский ввод – касания экрана. Сначала рассмотрим системные объекты-наследники UIView.
Красивое описание их состава и назначения можно найти в Apple Human Interface Guidelines. Открываем поисковую систему и ищем Apple HIG, переходим по этой ссылке. В списке слева есть группа Views. Открываем её, и видим несколько элементов с говорящими названиями, например, ImageViews, которые предназначены для показа картинок. Я рекомендую внимательно изучить каждый из элементов этой коллекции.
А сейчас мы обратим внимание на следующий пункт – Controls. UIControl – это наследник UIView, с помощью которого пользователь управляет приложением. В списке представлены такие элементы, как кнопки, надписи и другие. Также рекомендую внимательно их все изучить. Поработаем с представлением на практике – напишем небольшую игру, дизайн которой представлен вот на этом изображении:



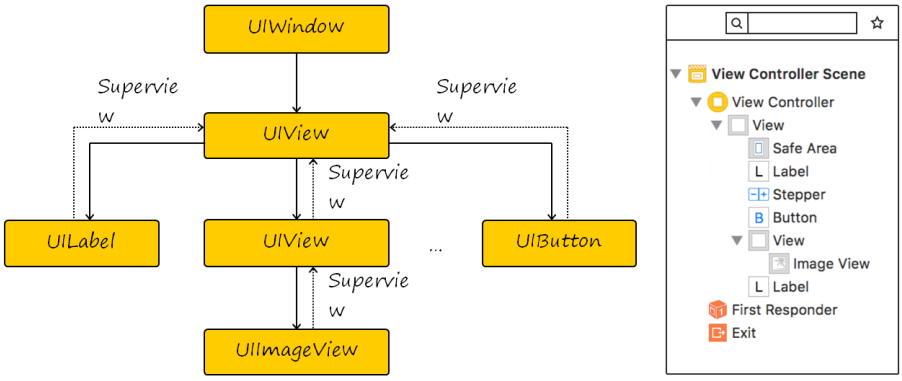
Как должна работать наша игра?
Пользователь может настроить время игры, оно отображается в поле в верхнем левом углу. При нажатии на кнопку ”Начать” на экране начинают появляться треугольники в произвольных местах поля ниже. Пользователь должен нажимать на них.
После истечения времени, или после нажатия на кнопку ”Прервать” (в которую превращается кнопка ”Начать” во время начала), в нижней части экрана должен быть отображён счёт – количество попаданий по треугольникам.
Давайте начнём с составления интерфейса.
Первым делом – текст в левом верхнем углу. Снова откроем Human Interface Guidelines и попробуем найти подходящий элемент. Это может показаться несколько странным, но он находится в разделе Controls и называется Labels. Открыв описание, легко убедиться в правильности выбора по скриншоту. Из текста становится ясно, почему этот элемент относится к управляющим – пользователи могут копировать текст, находящийся в нём.
Добавим элемент на наше рабочее поле. Теперь давайте добавим кнопку ”Начать” и этот элемент с минусом и плюсом. Для кнопки логично использовать UIButton, разместим её под текстом. Элемент с минусом и плюсом очень похож на UIStepper. Дальше разместим поле для игрового счёта. Также используем UILabel, который перетащим вниз окна и изменим текст на нужный нам. Как же сделать игровое поле? Для начала давайте для него добавим UIView.



Как же разместить треугольник внутри?
Помните, чуть раньше говорилось о том, что UIView может размещать внутри себя другие элементы представления, образуя иерархию. Давайте попробуем добавить на наше поле элемент UIImageView, который мы недавно видели в Human Interface Guidelines. Но прежде добавим картинку треугольника в наши ресурсы. Перетаскиваем картинку (pdf) в Assets, перетаскиваем UIImageView на поле в Main.storyboard, устанавливаем картинку для новой вьюшки.



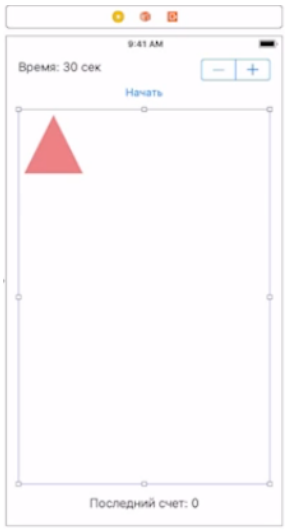
Картинка у нас получилась размещена как бы внутри UIView, привязана к нему. Схематично это можно представить следующим образом:



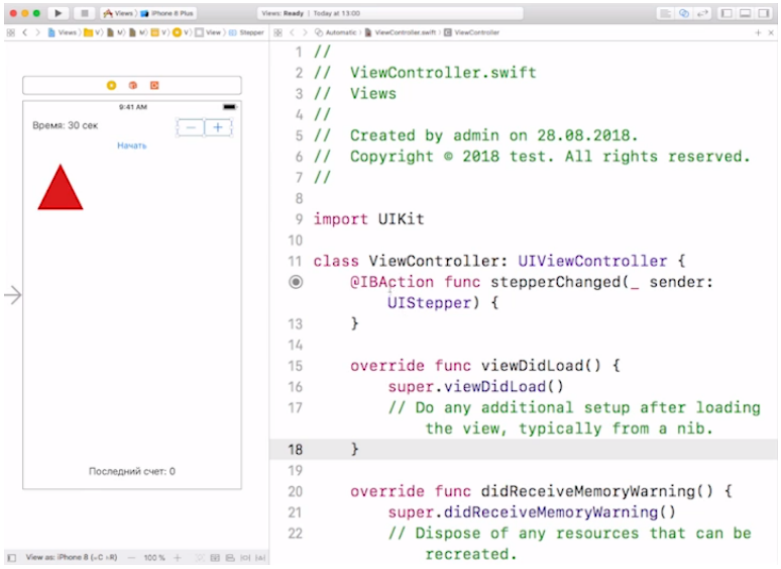
В самом верху находится UIWindow – это объект, который отвечает за отрисовку всех представлений, расположенных ниже. Далее – наше "полотно", которое есть тоже UIView. В его иерархию входят элементы, которые мы добавляли ранее. И у игрового поля также имеется UIImageView, внутри.

Давайте посмотрим на эту же иерархию в инспекторе InterfaceBuilder. Здесь вместо UIWindow у нас ViewController. Что это за зверь, вы познакомитесь в следующих частях курса. Сейчас же условимся, что это некоторый объект, который во время выполнения отдаёт свою корневую UIView для UIWindow. Остальная иерархия повторяет то, что представлена на схеме.

Что даёт нам иерархия? Грубо говоря, она позволяет группировать объекты по смыслу. Например, игровое поле. Значения свойств элементов выше могут "перекрывать" значения соответствующих свойств ниже по иерархии. Сделаем игровое поле скрытым – установим свойство Hidden в истину.



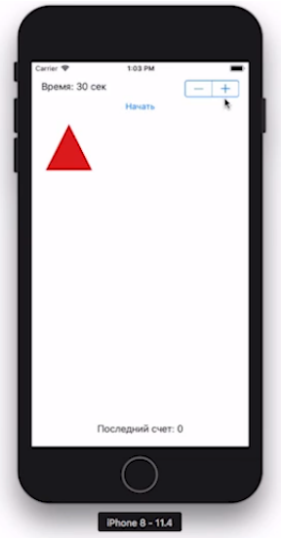
Мы видим, что наш треугольник также стал невидим. В InterfaceBuilder он, конечно, становится просто полупрозрачным, чтобы его было видно при проектировании. Также при удалении поля, ImageView исчезает. Теперь оживим степпер. Для этого откроем ассистента.
Нам нужно получать событие изменения значения степпера. Для этого существует удобный механизм – IBAction. Зажмём Ctrl и протянем линию прямо в наш код. В появившемся окне выбираем Action, проверяем Event – Value Changed, Any меняем на UIStepper, и указываем имя для нового метода-обработчика этого события. Назовём его stepperChanged. Нажимаем Connect.



Появился метод, помеченный аннотацией @IBAction. Она расшифровывается как Interface Builder Action, и помогает ему понять, что тот или иной метод должен быть использовать как обработчик события с элементом. Нам нужно изменять текст в надписи, а для этого как-то к ней обращаться из кода. Для этого есть аналогичный метод – IBOutlet. Также протянем линию от надписи в код, убедимся, что сейчас выбран Outlet, назовём новую переменную timeLabel. Connect. Появилось поле класса с нужным именем. Теперь осталось записать установку текста надписи:

```
timeLabel.text = "Время: (sender.value) сек"
```

Также можно настроить наш UIStepper, чтобы его начальное значение было 40, а изменялся он с шагом 5 (а не 1). Сделаем это в инспекторе. Теперь можно запустить приложение. Нажимаем на степпер. Если не считать нуля с точкой, то всё работает.



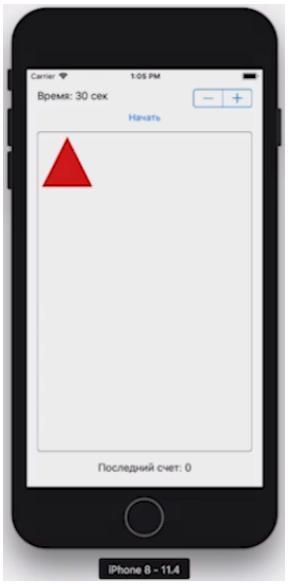
Наш интерфейс отличается от макета отсутствием рамки вокруг игрового поля. Её можно добавить, но только в коде. Создадим Outlet для игрового поля. В методе viewDidLoad, который вызывается после загрузки всех представлений в память, установим некоторые свойства для layer. Вначале, зададим толщину рамки в 1 пункт и установим серый цвет:

```
gameFieldView.layer.borderWidth = 1
gameFieldView.layer.borderColor = UIColor.grey.cgColor
```

В iOS цвет может быть описан структурами UIColor и CGColor. Цвет рамки лейера использует CGColor, при этом перечисление стандартных цветов заданы в UIColor. К счастью, у последнего есть свойство cgColor, который мы и использовали.
Последний штрих – закруглённые края. Их также устанавливаем в коде у лейера:

```
gameFieldView.layer.cornerRadius = 5
```

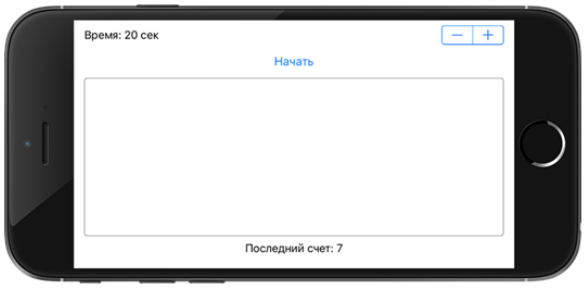
Запускаем и проверяем.

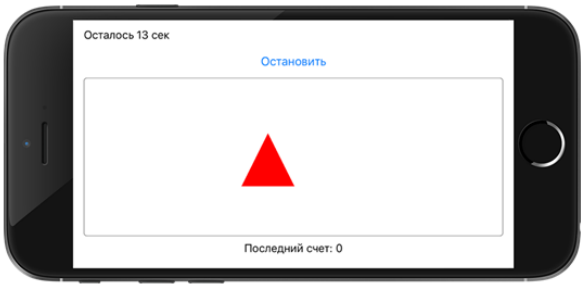


Поздравляю! Мы сформировали наш первый интерфейс по заданному дизайну. В следующий раз мы сделаем наши элементы интерактивными.

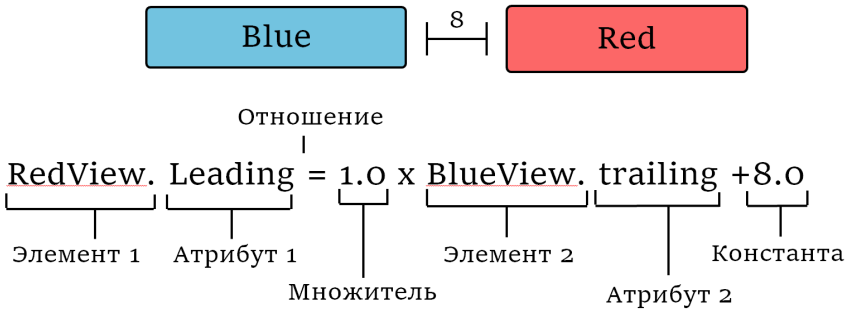
1.2. Autolayout

В предыдущий раз мы сделали первую версию приложения, где реализовали все основные компоненты интерфейса. Они ещё малофункциональны, но мы уже можем взаимодействовать с приложением — путём поворота телефона на 90 градусов. В симуляторе это можно сделать, нажав одновременно клавиши Command и стрелку влево или вправо — в какую сторону хотите повернуть телефон. Давайте сделаем это и посмотрим, что получилось.
Вряд ли мы хотели получить такой результат. На самом деле, дизайнеры предоставили нам макет и для горизонтальной ориентации. Вот он:





При повороте все элементы должны растягиваться и сжиматься, чтобы заполнить всё доступное пространство. Для реализации этого эффекта, и вообще адаптации нашего интерфейса под разные размеры экранов, существует два метода.
Первый — это механизм под названием Autolayout, который мы рассмотрим в этом видео. Второй — ручное вычисление размеров представлений, о нём мы поговорим в последующих лекциях.



При работе с Autolayout вы можете указать для элемента, что находится слева, справа, сверху и снизу от него, на каком расстоянии, а также задать его ширину и высоту. В итоге, расположение и размеры всех элементов экрана описываются системой линейных уравнений, которую iOS автоматически пересчитывает при любых изменениях лэйаута. Давайте посмотрим на практике, как с этим работать.
Начнём со степпера, который должен располагаться в правом верхнем углу экрана. Задать привязки, или констрейнты, для элемента в визуальном редакторе можно двумя способами. Первый способ — это специальная кнопка в нижней части редактора. Выделим элемент и нажмём на неё.
В появившемся окне можно задать расстояние до ближайших элементов с 4 сторон выделенного элемента, его ширину и высоту, а также ряд других привязок. Сейчас нас интересуют расстояния сверху и справа. Ближайший элемент к степперу — это край экрана, поэтому данное окно предоставляет нам весь необходимый функционал. Несколько увеличим расстояние сверху до 8, чтобы степпер не наезжал на строку состояния.
Обратите внимание, что линия от квадрата в центре до расстояния сверху стала яркой и цельной, по сравнению с соседними линиями. Это означает, что после нажатия кнопки Add Constraints снизу, мы добавим эту привязку. Привязки, помеченные бледным, добавлены не будут. Расстояние справа оставим по умолчанию, и активируем эту привязку.
Сверху и справа от элемента появились синие линии — это и есть новые привязки. Давайте посмотрим на свойства одной из них.
В верхней части указан первый элемент привязки, в данном случае это Safe Area.Trailing. Начиная с iOS 11, было введено понятие Safe Area — области, которая никогда не перекрывается конструктивными элементами телефона, такими как ”нотч” в iPhone X. Подробнее про это вы

можете прочитать в документации, а сейчас достаточно просто использовать этот механизм при расположении элементов.
Продолжим знакомиться со свойствами. Дальше указано отношение, Relation, и второй элемент — наш степпер, и его сторона — Trailing. Она всего лишь означает правую границу. Ещё ниже указана константа — то самое расстояние, которое мы задавали ранее. Потом идёт приоритет. Он позволяет системе определить, какие привязки использовать, если для элемента заданы конфликтующие правила. Он, как и остальные свойства, не понадобятся нам в крусе, поэтому оставим их на самостоятельное изучение. Дальше добавим привязки для label-а со временем. Его нужно центрировать по вертикали по отношению к степперу. Сделать это можно, потянув мышкой от одного элемента к другому, зажав Control.
В появившемся меню можно выбрать различные варианты привязки, зависящие от расположения элементов. Нам нужно отцентрировать по вертикали, выбираем этот пункт. Появилась жёлтая рамка — элемент находится не на своём месте. Нажмём на ещё одну кнопку в нижней части экрана, чтобы исправить это. Далее добавим привязку слева прежним способом. Сверху привязка не нужна, так как мы уже задали вертикальное расположение по отношению к степперу.
Кнопку Начать расположим по центру, на некотором расстоянии от степпера. Для центрирования по горизонтали используем третью кнопку в нижней части экрана.
Здесь собраны различные варианты центрации. Нам нужно разместить кнопку в центре контейнера горизонтально. Далее разместим элемент на некотором расстоянии от степпера, создав привязку через Ctrl, выбрав Vertical Spacing. Следующее на очереди — игровое поле. Зададим привязки со всех 4 сторон. Счёт разместим по центру и зададим отсуп снизу.
Мы добавили все привязки, но в левой панели осталось предупреждение — вот этот жёлтый кружок. Давайте посмотрим, что же не так.
Мы не указали привязку справа для времени. Может так случиться, что оно "наедет" на степпер, или даже за пределы экрана, и Xcode нас об этом предупреждает. Давайте выберем первый пункт — фиксированный левый край и плавающий правый.
Это не совсем то, что мы ожидали — время может растягиваться до конца экрана, а не только до степпера. Удалим эту и создадим свою привязку.
Сейчас расстояние задано жёстко. Чтобы оно было "плавающим", удалим константу и изменим отношение на больше или равно.
Отлично, сейчас предупреждений нет. Прежде чем запустить симулятор, посмотрим на наш экран в Preview.
Выглядит хорошо. Запускаем симулятор и проверяем результат нашей работы.
Интерфейс успешно адаптируется к изменениям в размерах экрана, и нам не пришлось написать ни строчки кода! Поздравляю! Вы научились адаптировать интерфейс под разные размеры контейнера, используя механизм AutoLayout. Познакомились с различными добавлениями привязок и увидели, как проверить поведение интерфейса без запуска симулятора. В следующем

видео мы всё же начнём писать код и добавим немного интерактива в нашу игру.

1.3. IBAction и установка таймера

В предыдущих видео мы подготовили интерфейс игры, который хорошо адаптируется к разным размерам экрана и его поворотам. Сейчас добавим в наше приложение немного кода. Давайте запустим приложение и определим, какую логику мы хотим реализовать.
Во-первых, добавим возможность изменять время игры при нажатии на кнопки степпера. Во-вторых, позволим запускать игру нажатием на кнопку "Начать". Что ж, приступим.
Мы уже создали IBAction для изменения значения степпера. Начнём заполнять его кодом. Напомню, нам необходимо изменять текст лейбла со временем в соответствии со значением степпера. Добавим код в stepperChanged(_:):

```
timeLabel.text = "Время: \(Int(stepper.value)) сек"
```

Для того, чтобы отображалось время без дробной части, значение степпера мы сконвертировали в Int. Также не забудем установить значение степпера на Storyboard.
Теперь можно начать и закончить игру при нажатии на соответствующую кнопку в интерфейсе. Как и для степпера, создадим IBAction для кнопки.
Для того, чтобы понять — нужно начать или закончить игру, нам потребуется индиактор текущей стадии игры. Заведём такое поле.

```
private var isGameActive = false
```

В начале игра, конечно, не активна, поэтому значение по умолчанию — false. Для запуска и остановки игры создадим функции-заглушки.

```
private func startGame() { }
private func stopGame() { }
```

Теперь можно завершить обработчик нажатия на кнопку старта:

```
if isGameActive {
    stopGame()
} else {
    startGame()
}
```

Что же должно произойти при начале игры? Во-первых, необходимо заблокировать степпер. Для этого создадим его переменную. И деактивируем его в начале игры.Добавим код в startGame():

```
stepper.isEnabled = false
```

Далее необходимо установить время до окончания игры в значение степпера. Для начала создадим переменную для хранения этого времени:

```
private var gameTimeLeft: TimeInterval = 0
```

Используем тип данных TimeInterval, который есть синоним к Double, а семантически предназначен, как можно догадаться, для представления интервалов времени. В начале игры установим его значение равным значению степпера.

```
gameTimeLeft = stepper.value
```

Теперь изменим надпись про время на более подходящее:

```
timeLabel.text = "Осталось \(Int(gameTimeLeft)) сек"
```

Текст кнопки "Начать" также стоит заменить. Создадим outlet для неё и изменим заголовок. Создадим IBOutlet для кнопки и назывем actionButton:

```
actionButton.title
```

В подсказках видно, что установить title нельзя. Но есть метод setTitle(title:for:). Почему так происходит? Элемент управления "Кнопка" может находиться в разных состояниях — обычное, нажатое, неактивное и так далее. Можно установить текст для каждого из состояний отдельно. При этом заголовок "обычного" состояния — это заголовок по умолчанию, он будет подставлен для любого другого статуса, если он отдельно не установлен. Соответственно, зададим его.

```
actionButton.setTitle("Остановить", for: .normal)
```

Также нам надо установить флаг активности игры:

```
isGameActive = true
```

Теперь заполним функцию stopGame. Установим признак активности игры.

```
isGameActive = false
```

Далее разблокируем степпер:

```
stepper.isEnabled = true
```

Вернём назад текст для времени игры:

```
timeLabel.text = "Время: \(Int(stepper.value)) сек"
```

Хм, кажется, что мы это уже писали в обработчике изменения значения степпера. Действительно. Повторение кода — не очень хорошая практика. В данном случае, при необходимости изменить текст на, например, "Ваше время", нам придётся сделать это в двух местах. Скорее всего, с первой попытки мы обязательно забудем изменить его во втором месте. Попробуем избежать дублирования и вынесем код обновления представления в отдельную функцию — updateUI.

```
private func updateUI() { }
```

Интерфейс у нас может находиться в двух состояниях. Опишем их.

```
if isGameActive {
} else {
}
```

Теперь вынесем всё обновление поля времени в новую функцию. Выделяем и вырезаем строку из stepperChanged, вставляю в ветку else updateUI(). Выделяем строку обновления timeLabel в startGame, вырезаем, вставляем в ветку isGameActive updateUI(). Удаляем строку обновления timeLabel из stopGame.
Сюда же можно перенести установку активности степпера. Добавляем строку в начало функции updateUI() :

```
stepper.isEnabled = !isGameActive
```

Удаляем установку активности степпера из stopGame. В новую функцию также можно добавить обновление заголовка кнопки начала/остановки игры.

```
if isGameActive {
    ...
    actionButton.setTitle("Остановить", for: .normal)
} else {
    ...
    actionButton.setTitle("Начать", for: .normal)
}
```

Удалим строку установки заголовка кнопки из startGame. Нам осталось добавить вызов новой функции в нужных местах. Добавим вызов updateUI в stepperChanged. Добавим вызов updateUI в startGame. Добавим вызов updateUI в stopGame.
Запустим нашу игру. Нажмём на "Начать". Заголовки изменились, при нажатии на степпер ничего не происходит. Нажимаем на "Остановить". Всё вернулось на свои места, время игры — прежнее, то есть степпер действительно был заблокирован. Сейчас он время изменяет.
Итак, поздравляю! Мы добавили функцию начала и завершения игры. Мы увидели, как работают IBAction, потренировались в изменении свойств IBOutlet, а также провели небольшой ре-

факторинг кода, убрав его дублирование. В следующем видео мы начнём работу над механикой самой игры — заставим фигуру смещаться в пределах поля через равные интервалы времени.

1.4. Таймер

В прошлый раз мы добавили возможность начинать и останавливать игру. Однако игровой логики как таковой всё ещё нет. В этом видео мы познакомимся с объектом Timer, который позволит обновлять оставшееся в игре время и перемещать фигуру.
Наша задача — обновлять оставшееся время каждую секунду после запуска игры. Для этого будем использовать объект с аналогичным названием Timer. Давайте создадим его и будем хранить в отдельном свойстве.

```
private var gameTimer: Timer?
```

В функции startGame() напишем

```
gameTimer = Timer.sched
```

Есть несколько варианитов создания таймера. В частности, можно создать таймер, выполняющий переданный ему блок кода. Эта функция доступна с iOS 10. Мы же используем более полную его версию — таймер с вызовом функции заданного класса. В списке подсказок выбираем функцию. Пройдемся по параметрам:

- timeInterval — интервал вызова функции в секундах. Он должен быть 1.
- target — объект, у которого нужно вызвать функцию. Это self, текущий объект. сейчас пропустим.
- userInfo — дополнительная информация, которая может быть передана вызываемой функции. Она нам не нужна, поэтому nil.
- repeats — необходимо ли повторять вызов функции, либо вызвать её лишь один раз через заданный интервал после создания таймера. Нам нужны повторения, поэтому устанавливаем true.

Чтобы указать функцию, которую необходимо вызвать у объекта когда-то позже, в Swift используется концепция селекторов. Мы пишем:

```
selector: #selector()
```

И в скобках указываем полную сигнатуру функции. Сейчас у нас её нет. Сделаем функцию-заглушку:

```
private func gameTimerTick() {}
```

И передадим её сигнатуру селектору:

```
selector: #selector(gameTimerTick)
```

Появилась ошибка. В чём её смысл? На самом деле, концепция селекторов относится не к Swift, а к его предшественнику Objective-C. Многие классы стандартной библиотеки до сих пор написаны на нём. А взаимодействие с классами на Swift осуществляется по определённым правилам. В частности, для того, чтобы метод класса из Swift стал доступен для Objective-C, для него нужно добавить специальный аттрибут. Конструктор селектора ожидает увидеть как раз таки метод, который можно вызывать в Objective-C, и компилятор предлагает добавить аттрибут. Примем это предложение. Нажимаем на Fix.
Отлично. Но перед созданием, и запуском, нового таймера, предыдущий необходимо остановить. Добавим строку в startGame() перед созданием таймера:

```
gameTimer?.invalidate()
```

Также не забудем остановить таймер при остановке игры. Добавим строку в конец stopGame():

```
gameTimer?.invalidate()
```

Теперь наполним функцию gameTimerTick смыслом. В прошлый раз мы создали функцию updateUI(), которая обновляет текст времени в соответствии со значением внутренней переменной gameTimeLeft. Используем её при срабатывании таймера. Уменьшим оставшееся время на 1 и вызовем updateUI().

```
gameTimeLeft -= 1
updateUI()
```

Также добавим вывод строки-уведомления о срабатывании таймера в консоль:

```
print("gameTimerTick called")
```

Запустим игру. Уменьшим время игры до 5 секунд и нажмём "Начать". Отлично, время идёт, в логах видим сообщения о срабатывании таймера. Подождите, похоже, мы ушли в минус по времени и сейчас портим свою кредитную историю. Пожалуй, при достижении 0 игру нужно автоматически останавливать.Останавливаем игру, переходим в функцию gameTimerTick(), добавляем строку после уменьшения оставшегося времени:

```
if gameTimeLeft <= 0 {
    stopGame()
} else {
```

```
}
```

Переносим updateUI() в ветку else. Снова запускаем игру, устанавливаем время на 5 секунд и дожидаемся окончания времени. Отлично! Игра остановилась, в логах новых сообщений о срабатывании таймера не видно.
В начале мы хотели также сдвигать фигуру по таймеру. Нам надо учесть, что период задержки фигуры на экране не обязательно должен равняться 1 секунде. Поэтому для этой задачи потребуется создать две дополнительные переменны: новый таймер и время задержки фигуры на одном месте. Добавляю переменные в класс

```
private var timer: Timer?
private let displayDuration: TimeInterval = 2
```

Также сразу добавим функцию-заглушку, которую будет вызывать таймер:

```
@objc private func moveImage() { }
```

И создаём таймер в начале игры, не забыв его предварительно остановить. Добавляем код в startGame():

```
timer?.invalidate()
timer = Timer.scheduledTimer(
    timeInterval: displayDuration,
    target: self,
    selector: #selector(moveImage),
    userInfo: nil,
    repeats: true
)
```

Мы хотим, чтобы функция перемещения фигуры была вызвана сразу в начале игры, а не только через сколько-то секунд. Для этого вызовем метод fire() у таймера:

```
timer?.fire()
```

Не забудем остановить таймер при остановке игры:

```
timer?.invalidate()
```

В функцию передвижения фигуры на сейчас добавим просто print, а из функции gameTimerTick() его уже можно удалить — мы проверили её работоспособность. Удаляем print из функции gameTimerTick(). Добавляем строку в функцию moveImage()

```
print("moveImage called")
```

Запустим игру и убедимся, что наша функция вызывается во время игры. И не вызывается, когда игра остановлена. Прекрасно, всё работает как задуманно!
В этом видео мы познакомились с объектом Timer, который позволяет выполнять действия с заданным интервалом времени. Мы также посмотрели на концепцию селекторов и взаимодействия с Objective-C кодом на уровне функций. В следующий раз мы вдохнём жизнь в нашу фигуру — заставим её перемещаться по полю.

1.5. Перемещение фигуры. Autolayout constant

В предыдущем видео мы добавили два таймера и заглушку функции перемещения фигры по игровому полю. В этом видео сделаем так, чтобы фигура на самом деле перемещалась.Откроем Main.storyboard.
Положение объектов относительно друг друга мы можем задавать с помощью механизма Auto Layout. Картинка расположена внутри игрового поля, для неё ограничения не заданы. Добавим их. Выделяем объект, добавляем привязки слева и сверху, устанавливаем константу 0 для обеих, ширину и высоту задаем равными 80.
Для constraints также можно создавать IBOutlet, что мы и сделаем для привязок слева и сверху. Выделяем фигуру, последовательно протягиваем аутлеты для левой и верхней привязок, называя их shapeX и shapeY.
У привязок есть свойство constant, определяющее расстояние между объектами. Сейчас константы у обеих привязок равны 0. В функции moveImage() нужно установить их случайными величинами. Давайте для начала диапазон возможных значений ограничим вручную заданными значениями. Предварительно удалим наш print. Удаляем print.

```
let maxX = 100
let maxY = 100
shapeX.constant = arc4random_uniform(maxX)
```

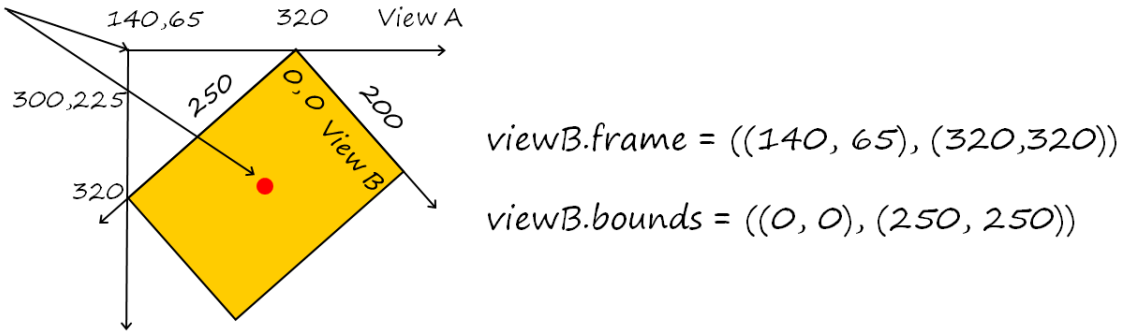
Функция arc4random_uniform генерирует случайные числа в диапазоне от 0 до верхней границы, передаваемой ей в качестве параметра. При этом параметр должен иметь тип UInt32. Функция возвращает значение с таким же типом.
Наши координаты, конечно, другого типа — не UInt32. Добавим явное преобразование типов. Изменяем строку установки константы .

```
shapeX.constant = CGFloat(arc4random_uniform(UInt32(maxX)))
```

Также установим случайной константу для привязки сверху. Добавляем строку в функцию moveImage():

```
shapeY.constant = CGFloat(arc4random_uniform(UInt32(maxY)))
```

Запустим игру. Нажимаем "Начать". Прекрасно, фигура переместилась! И продолжает перемещаться каждые 2 секунды.
Сейчас нам нужно научиться высчитывать максимальные координаты для перемещения фигуры, чтобы она ходила по всему полю при любом размере и ориентации экрана. Начнём с определения границ игрового поля. Для него уже есть переменная gameFieldView.
UIView имеет два свойства, определяющих её размер и положение. Это frame и bounds.



Размер самой UIView содержится в переменной bounds, её положение как правило соответствует координате 0, 0 в системе координат view. Frame описывает прямоугольник, в котором отображается UIView внутри своего superview. Соответственно, это свойство вложенных представлений использует superview при расчётах. Его не следует использовать внутри наследников UIView. Хотя frame и bounds часто совпадают, есть много ситуаций, когда это не так. Например, если представление повёрнуто, как на этом примере. Также эти свойства отличаются у scroll view.
В нашем случае мы хотим определить доступное пространство для перемещения вложенного представления. Поэтому используем свойство bounds у gameFieldView. У этого свойства есть поля maxX и minX, определяющие те самые границы, которые нам нужны. Заменим let maxX, let maxY на следующий код:

```
let maxX = gameField.bounds.width
let maxY = gameField.bounds.height
```

Нам осталось учесть размеры самой фигуры, иначе она будет выходить за пределы поля. Например, когда расстояние слева будет равно ширине поля. Для этого сначала создадим переменную для фигуры. Создаем IBOutlet для фигуры и называем gameObject.
Так как мы позиционируем вложенный объект, нас интересует его свойство frame для определения размеров. Вычтем соответствующие размеры из размеров поля. Изменяем код для maxX и maxY:

```
let maxX = gameFieldView.bounds.width - gameObject.frame.width
let maxY = gameFieldView.bounds.height - gameObject.frame.height
```

Запустим игру и посмотрим на результат. Фигура перемещается явно дальше заданных ранее координат. Будем надеяться, что она захватывает всё поле и не выходит за его пределы.
Поздравляю! В этом видео мы увидели, как можно перемещать объекты, изменяя константы привязок; научились генерировать случайные числа; познакомились с концепциями frame и bounds для определения размеров объектов. В следующем видео наша игра научится реагировать на нажатия пользователя по фигуре и вести подсчёт набранных очков.

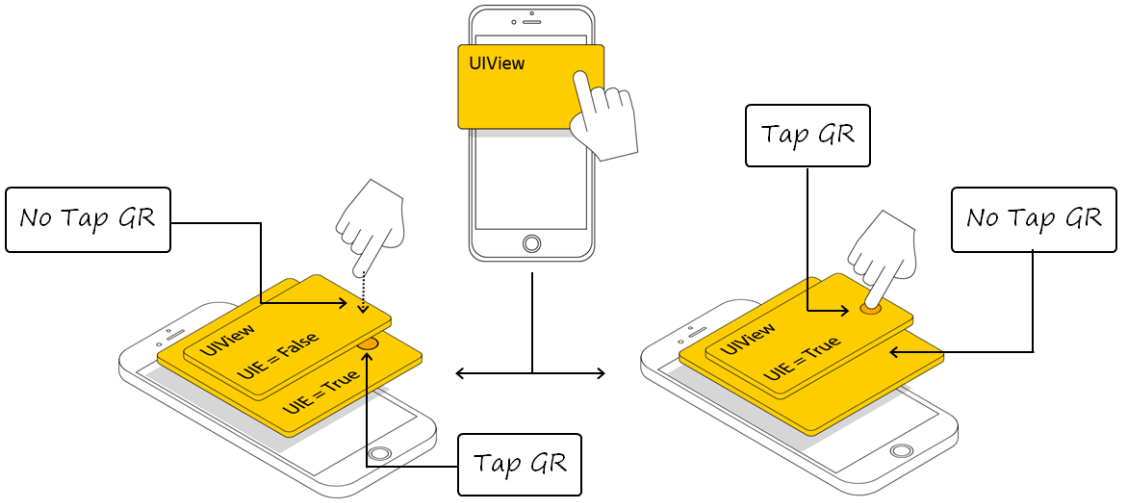
1.6. TapGestureRecognizer

В прошлый раз мы вдохнули жизнь в фигуру — она стала перемещаться по полю случайным образом. Но у пользователя нет возможности её "поймать". Игра не реагирует на нажатия по фигуре. Давайте это исправим.
Один из наиболее простых способов реагировать на взаимодействие пользователя с элементами интерфейса — использовать различные gesture recognizer-ы. Или иначе распознаватели жестов. Их существует достаточно много стандартных, а также можно создать свои собственные, унаследовав их от UIGestureRecognizer.
Рекогнайзеры можно добавить визуально с помощью Interface Builder-a. Они находятся в палитре объектов, как и все другие элементы, которые мы добавляли ранее. Открываем Main.storyboard, в строке поиска объектов пишем gesture.
Здесь перечислены стандартные рекогнайзеры. Нам нужно "ловить" жест нажатия, или иначе — тапа. Поэтому первый элемент, TapGestureRecognizer, как раз то, что нам нужно. Перетащим его на целевой объект.
Рекогнайзер появился в самом конце списка корневых объектов иерархии сцены. Чтобы понять, к какому объекту относится тот или иной рекогнайзер, нужно нажать на нём правой кнопкой и посмотреть состав раздела Referencing Outlet Collections.
Обработчик срабатывания рекогнайзера можно создать уже привычным методом — как IBAction. Ctrl-протягиваем связь от рекогнайзера перед разделом Private-функций, выбираем Action, Type — UITap..., Name — objectTapped.
Внутри функции традиционно напишем функцию print, чтобы убедиться в работоспособности нашей конструкции. Добавляю строку в метод objectTapped(_:)

```
print("objectTapped called")
```

Запустим игру и попробуем нажать на фигуру. Странно, в консоли не появляется ожидаемое сообщение. Можно предположить, что сама платформа обладает хорошим интеллектом, и не

вызывает обработчик нажатия, так как мы не начали игру. На самом деле, всё намного прозаичнее.



У UIView и всех его наследников есть свойство userInteractionsEnabled. Если оно установлено в истину, то представление может получать пользовательский ввод. Например, нажатия. Если же оно установлено в ложь, то любые взаимодействия пользователя с этим объектом будут проигнорированы, и перенаправлены объектам, расположенным под ним.
По умолчанию это свойство установлено в ложь для всех наследников UIView, кроме контролов. У нашего imageView оно также — ложь. Это легко проверить, найдя его в палитре свойств.
Выделяем треугольник, открываем палитру свойств, находим User interactions enabled. Флажок действительно не установлен. Исравим это. Запускаем снова. Нажимаем на треугольник и видим, что в консоли появилось долгожданное сообщение. Можем двигаться дальше.
При тапе на фигуру нужно перемещать фигуру и увеличивать счёт игры. Код для перемещения фигуры у нас написан в методе startGame() — там, где мы создавали таймер и вызывали fire(). Так как он нам нужен ещё в одном месте, вынесем его в отдельную функцию. Создадим функцию:

```
private func repositionImageWithTimer() {
}
```

Переносим код из startGame() про timer в новую функцию. Добавляем вызов repositionImageWith Timer() в startGame(). И вызовем эту функцию в обработчике нажатия на фигуру, удалив print. Для счёта необходимо завести новую переменную:

```
private var score = 0
```

При старте будем её обнулять. Добавим в начало startGame() :

```
score = 0
```

А при окончании игры нужно обновить информацию про последний счёт. Создадим переменную для надписи. Создаем IBOutlet для надписи со счётом под названием scoreLabel. И обновим её текст при окончании игры. Для этого добавляю код в конец stopGame():

```
scoreLabel.text = "Последний счет: \(score)"
```

Добавим увеличение счёта в функцию-обработчик нажатия на фигуру. Добавляем код в конец objectTapped(_:)

```
score += 1
```

Теперь снова запустим игру и посмотрим, что получилось. Нажимаем начать и несколько раз попадаем по объекту. Как минимум, он перемещается на новую позицию при попадании. Теперь остановим игру. Последний счёт обновился, показывает количество успешных попаданий по фигуре.
Нажмём на фигуру ещё раз. Она переместилась, хотя игра не запущена. Нужно добавить проверку на то, что игра запущена, при обработке нажатия на фигуру. Добавляем код в начало objectTapped(_:)

```
guard isGameActive else { return }
```

А вообще, было бы хорошо показывать фигуру, только когда игра активна. Обновим функцию updateUI. Если игра активна, показываем фигуру. Добавим код в начало функции updateUI():

```
gameObject.isHidden = !isGameActive
```

При запуске приложения поле выглядит так, как мы настроили его на Storyboard-е. При этом в коде зашито уже достаточно много логики, описывающей правильное отображение элементов. Это и время игры, и счёт. Сейчас ещё и видимость фигуры. Чтобы нам не приходилось следить за соответствием Storyboard тому, что делает код при дальнейшей жизни игры, вызовем обновление UI в функции viewDidLoad. Добавляем код в конец viewDidLoad() updateUI().
Запустим игру. Фигуры не видно. Начнём играть. Фигура появилась.
После завершения игры фигура исчезла. Счёт всё также обновляется.
Поздравляю! Мы сделали свою первую игру! В этом видео мы познакомились с концепцией GestureRecognizer, научились включать и выключать получение событий взаимодействия пользователя с определённым представлением, а также скрывать определённые view. Хотя игра стала полностью функциональной, есть ряд мест, которые можно сделать лучше. В следующем видео мы полностью переделаем игровое поле — выполним его как произвольную, или кастомную, view.

1.7. Рисуем поле с помощью drawRect

В прошлый раз мы увидели один из недостатков подхода с представлением нашего треугольника в виде ImageView — мы не знаем, попал ли пользователь в сам треугольник, или же в пустое место по периметру.
Одним из возможных путей исправления этого недостатка является отрисовка фигур внутри поля вручную. Для этого потребуется создать класс-наследник UIView. Назовём его GameFieldView. Нам нужен UIKit, импортируем его.
Создадим одноимённый класс-наследник UIView:

```
class GameFieldView: UIView {
}
```

Для ручной отрисовки внутри представления переопределим метод drawRect. Как обычно, добавим вызов его реализации из класса-родителя.
Нам нужно нарисовать фигуру определённого цвета. Для этого определим свойство shapeColor с типом UIColor. Пусть по умолчанию оно будет красным.

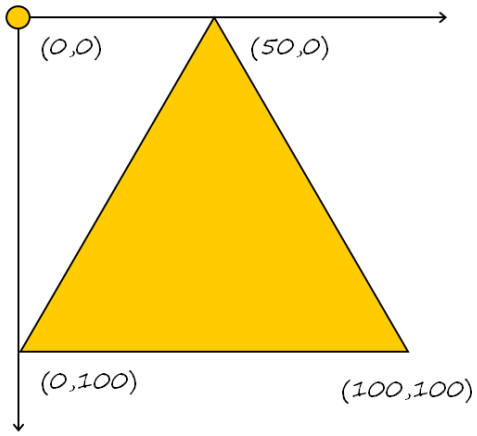
```
var shapeColor: UIColor = .red
```

Теперь можно добавить в наш метод установку цвета заливки фигур, которые мы будем рисовать.

```
shapeColor.setFill()
```

Дальше нам нужно отрисовать саму фигуру. Для этого используем UIBezierPath. Это очень мощный инструмент создания произвольных изображений, с которым вы можете ознакомиться намного глубже в документации. Здесь же рассмотрим базовые приёмы работы с ним с помощью прямых линий.

```
let path = UIBezierPath()
path.move(to: CGPoint(x: 0, y: 100))
path.addLine(to: CGPoint(x: 0, y: 100))
path.addLine(to: CGPoint(x: 0, y: 100))
path.close()
path.stroke()
path.fill()
```



В начале работы с новой кривой курсор располагается в точке с координатами (0; 0) — левый верхний угол. Мы перемещаем его в нижний левый угол нашего треугольника — (0; 100). Дальше рисуем линию до верха треугольника — (50; 0), а затем — до правого нижнего угла — (100; 100). Теперь замыкаем путь, система сама добавит линию до точки начала пути — нижнего левого угла. Теперь прорисовываем линии — stroke. И заливаем фигуру выбранным цветом — fill.
Этот код можно перенести в проект, заменив абсолютные координаты на те, что передают в качестве параметра. Я вставлю в наш класс заготовленную функцию getTrianglePath(in:).
На вход она принимает структуру CGRect, которая описывает прямоугольник, в котором рисуем фигуру. Он содержит координаты левого верхнего угла (origin) и размер. Также эта структура предоставляет удобные вычисляемые поля min-, mid- и maxX, а также их аналоги для Y.
Сейчас можно использовать новый метод для отрисовки треугольника в методе drawRect. Но сперва нужно завести ещё два поля — координату верхнего левого угла прямоугольника для размещения фигуры и его размер.

```
var shapePosition: CGPoint = .zero
var shapeSize: CGSize = CGSize(width: 40, height: 40)
```

По умолчанию размещаем его в левом верхнем углу и задаём размер 40 на 40 пунктов. Получаем кривую Безье из нашего метода, и заполняем её цветом:

```
let path = getTrianglePath(in: CGRect(origin:shapePosition, size: shapeSize))
path.fill()
```

Теперь давайте используем новый класс в InterfaceBuilder. Удалим ImageView из нашего поля, а в свойствах самого поля укажем класс. Переключаем вкладку настроек, указываем GameFieldView в качестве класса.
К сожалению, мы не видим никаких изменений. Для того, чтобы в Interface Builder выполнялся код кастомных представлений, их нужно аннотировать с помощью @IBDesignable. Для параметров, которые должны быть отображены в палитре свойств элемента, нужно добавить аннотацию @IBInspectable. Добавим их для нашего класса.
Добавляем @IBDesignable перед классом. Добавляем @IBInspectable перед (1) shapeColor, (2) shapePosition, (3) shapeSize.
Открываем инспектор, ожидаем сборки проекта, и видим, что на поле появился треугольник. Изменим его размер в палитре свойств — изменения применяются здесь же, в Interface Builder. В инспекторе свойств указываем Shape Size на 80, 80.
Интегрируем наш новые метод в существующий код. Открываем файл ViewController.swift. Мы уже удалили Image View, сейчас нужно удалить несуществующие более поля gameObject, shapeX, shapeY, а также изменим тип gameFieldView на GameFieldView.
Ранее мы изменяли видимость image view при старте и завершении игры. Сейчас этого объекта нет, и логику нужно перенести в новый класс. Добавляем свойство, также с аннотацией

IBInspectable:

```
@IBInspectable var isShapeHidden: Bool = false
```

При отрисовке, в drawRect, достаточно ничего не рисовать:

```
guard !isShapeHidden else { return }
```

Заменим обращения к свойству isHidden прежнего объекта gameObject на новое свойство поля. В методе updateUI() заменяю

```
gameObject.isHidden = !isGameActive
```

на

```
gameFieldView.isShapeHidden = !isGameActive
```

Ещё одно место использования gameObject, а также двух других удалённых переменных — метод moveImage. Перенесём этот код в класс GameFieldView — в метод randomizeShapes().
Добавляю internal метод randomizeShapes(), вставляю в него скопированный код.
Немного модифицируем его для задания нового положения фигуры в заданных пределах. bounds теперь нужно использовать у объекта self, а размер фигуры хранится в shapeSize. Первые две строки заменяем на

```
let maxX = bounds.width - shapeSize.width
let maxY = bounds.height - shapeSize.height
```

Для перемещения фигуры сейчас используем координаты, а не константы constraint-ов. Изменяю последние две строки метода randomizeShapes():

```
let x = CGFloat(arc4random_uniform(UInt32(maxX)))
let y = CGFloat(arc4random_uniform(UInt32(maxY)))
```

Теперь можем переместить фигуру по новым координатам:

```
shapePosition = CGPoint(x: x, y: y)
```

В методе moveImage() вызовем новую функцию нашего игрового поля:

```
gameFieldView.randomizeShapes()
```

Последняя часть, ради которой во многом всё затевалось — корректная обработка нажатий. Как понять, что пользователь коснулся нашего представления? Один вариант нам уже известен — использовать UIGestureRecognizer. Сейчас мы используем второй вариант — переопределим метод touchesEnded. В рамках данного курса мы не будем рассматривать сопуствующие ему другие методы: touchesBegan, Moved, Cancelled.

Метод система вызывает, когда пользователь закончил взаимодействие с нашим элементом. Проще говоря — отодвинул палец от экрана внутри элемента. Под методом draw(_:) печатаем touchesEnd, выбираем вариант. Добавляю код внутри super.touchesEnded(touches, with: event). Не забываем вызвать соответствующий метод объекта-родителя.
Первый параметр функции — это касания, которых может быть несколько, если касались несколькими пальцами одновременно. Чтобы определить, есть ли среди них касание по нашему объекту, используем функцию contains(where:):

```
let hit = touches.contains(where: { touch -> Bool in })
```

Чтобы узнать координаты касания в координатной системе нашей view (относительно её базовой точки — origin), у UITouch есть метод location(in:):

```
let touchPoint = touch.location(in: self)
```

Теперь нужно понять, входит ли данная точка внутрь нашей фигуры. У UIBezierPath есть, подобно UIView, метод contains(_), который даёт ответ на этот вопрос. Нам нужно сохранить последнюю нарисованную кривую в новую скрытую переменную. Добавляю свойство над getTrianglePath(in:):

```
private var curPath: UIBezierPath?
```

Добавляем код в draw(_:):

```
guard ... else {
curPath = nil
}
```

Добавляем код в конец draw(_:):

```
curPath = path
```

Используем эту переменную для определения попадания пользователя в фигуру. Сначала проверим, определена ли она. Добавляем код в начало draw(_:):

```
guard let curPath = curPath else { return }
```

Далее используем его в нашем методе contains:

```
return curPath.contains(touchPoint)
```

Теперь добавим свойство-обработчик тапа на фигуру:

```
var shapeHitHandler: (() -> Void)?
```

Сделаем его Optional, чтобы не пришлось устанавливать в инициализаторе.
Добавим в touchesEnded последний штрих — вызов обработчика, если пользователь попал в фигуру:

```
if hit {
    shapeHitHandler?()
}
```

Теперь в нашем прежнем коде можно убрать аннотацию IBAction и параметр из метода object Tapped. Удаляем аннотация и параметры из метода objectTapped(_:).
Добавим обработчик нажатия на фигуру на поле в функции viewDidLoad:

```
gameFieldView.shapeHitHandler = { [weak self] in
    self?.objectTapped()
}
```

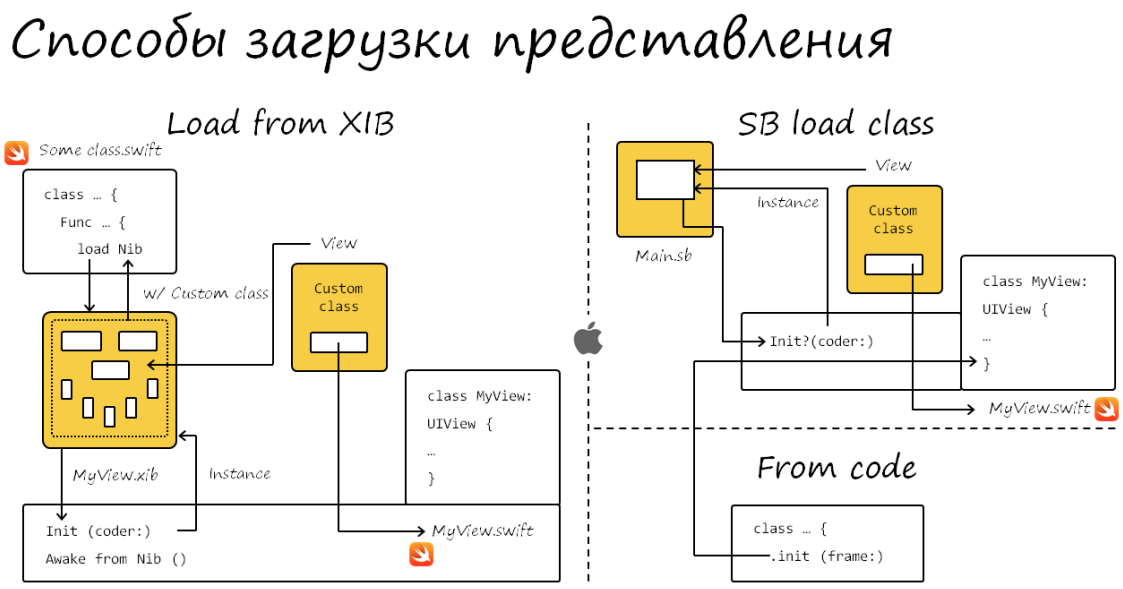
Теперь можно запустить проект. Начинаем игру. Фигура не появилась! Дело в том, что метод drawRect вызывается, только когда система посчитает это необходимым. Когда изменяются свойства view, заданные нами, она так не думает. Но можно ей об этом подсказать, вызвав метод setNeedsDisplay(). Так мы порекомендуем системе перерисовать наше представление во время следующего выполнения run loop. Вызовем этот метод при установке соответствующих свойств. Добавляем код в didSet свойства (1) isShapeVisible, (2) shapePosition, (3) shapeSize, (4) shapeColor. setNeedsDisplay().
Запустим проект снова. Начинаем игру. Ура! Фигура появилась. Нажмём на фигуру... Она сместилась. Значит, всё работает.
Поздравляю! Вы проделали большую работу. В этом видео мы научились создавать собственные представления (или кастомные вьюшки), рисовать внутри них и познакомились с функцией setNeedsDisplay(). В следующий раз мы научимся загружать представления из специальных файлов — xib-ов.

1.8. Загрузка view из Nib/Xib

До сих пор мы создавали представления на Storyboard с помощью визуального редактора, а также из кода путём рисования с помощью Bezier Path. У первого подхода есть недостаток — сложно переиспользовать полученные представления в других местах приложения. Второй подход этого недостатка лишён, но часто можно использовать готовые компоненты, а не рисовать всё вручную. Также его недостатком можно назвать невозможность построения интерфейса графически.
У нас есть ещё два метода создания представлений, которые лишены этих недостатков, впрочем

как и некоторых преимуществ рассмотренных ранее подходов.
Первый — загрузка представлений из специальных файлов, xib-ов или nib-ов. Его мы рассмотрим в этом видео. Второй — программное создание представления с использованием стандартных компонент. О нём поговорим в следующий раз.
Предположим, нам нужно переиспользовать панель управления игрой в другом месте. Панель управления — это элементы над игровым полем. Давайте вынесем их в отдельную сущность. Создадим новый файл.
Выберем View в разделе User Interface. Далее. Назовём его GameControlView. В проекте появился файл GameControlView.xib. Сейчас он представляет собой пустое поле, размеры которого напоминают экран телефона. Откроем инспектор свойств. Три верхних свойства, Size, Top Bar и Bottom Bar, определяют, как выглядит наше представление в Interface Builder. К его размерам и наличию баров в запущенном приложении они не имеют никакого отношения. Настроем их, как нам удобно. А именно, установим Size во freeform, чтобы изменять размеры вручную.
Уменьшим размеры полотна до тех, что больше похожи на реальные. Теперь можно перенести наши элементы на новое место. Открываем файл Main.storyboard. Выделяем три элемента управления. Вырезаем их через Command-X. Открываем файл GameControlView.xib. И вставляем на новое место через Command-V. У элементов существовали привязки к другим элементам на прежнем месте. Сейчас их нет, механизм Auto layout не может определить их положение на экране. Поэтому оставшиеся привязки подсвечены красным — не понятно, как их удовлетворить. Исправим это.
Начнём со степпера. Добавим привязки сверху и справа. Прижмём его к правому краю. Время прижмём влево, а по вертикали его положение уже определено. Кнопку нужно центрировать по горизонтали. А также задать расстояние до низа ноль.
Открываем окно с иерархией. Ошибок нет. Теперь нам нужно добавить элемент UIView на Storyboard. В нём должно загрузиться наше новое представление. Открываем файл Main.story board. Ищем uiview в палитре объектов, перетаскиваем его на поле.
Добавим привязки для нового элемента. Странно, у нас снова появились ошибки layout-а. Дело в том, что сама UIView не имеет своей высоты, так как в ней ничего не лежит. Поэтому механизм auto layout не может определить положение зависимых от неё представлений по вертикали. Эта проблема разрешится, когда мы изменим класс для представления.
Перейдём в Identity инспектор. В поле Class укажем созданное ранее представление — GameCont rolView. Xcode не подсказывает нам имя нашего объекта. Это не удивительно, так как он ожидает класс. Мы же создали просто представление, которое не умеет ничего, кроме позиционирования элементов на экране. Добавим для него класс.
Создаем новый файл в корне. Как и для GameFieldView выберем тип Swift. Обычно классу и xib-у дают одинаковые имена. То есть нам стоило бы назвать его как GameControlView. Но в учебных целях, чтобы было понятнее, что мы используем далее в каждом конкретном случае, xib или сам класс, назовём класс как GameControlViewClass.

Импортируем UIKit и определим класс-наследник UIView. Так как мы хотим использовать класс в Storyboard, сразу укажем для него аттрибует IBDesignable.
Теперь можно указать класс в нашем интерфейсе. Возвращаемся в Main.storyboard и указываем класс GameControlViewClass.
В представлении ничего не изменилось, красные линии не пропали. Всё потому, что класс и xib никак друг с другом не связаны. Даже если бы мы их назвали одинаково, магии бы не случилось.
Сначала небольшое отступление про конструкторы UIView. Класс некоторого UIView может быть инициализирован из Xib-а или Storyboard, если этот класс указан в Identity инспекторе в Interface Builder. В этом случае будут вызываны методы init?(coder:) и затем awakeFromNib(). Также представление может быть инициализировано из кода, тогда будет вызыван конструктор init(frame:).



При работе с Xib-ами их обычно загружают в коде специальной функцией, а они уже создают объект класса, указанного в поле Custom Class для своей view. Storyboard же работает, во многом, как Xib — он так же создаёт указанный класс, и не умеет сперва загружать какой-либо xib. Но загрузку xib-а можно прописать в конструкторе самого класса. Сейчас мы это и сделаем.
Сперва, класс нужно задать в Xib. Открываем файл GameControlView.xib и в инспекторе иерархии выбираем File's Owner. В Identity инспекторе указываем наш новый класс. Возвращаемся к самому классу.
Сперва напишем функцию загрузки View из Xib-а:

```
private func loadViewFromXib() -> UIView {
}
```

Получаем текущий Bundle. Мы не будем рассматривать, что это такое в данном видео. Кратко — это контейнер, содержащий объекты нашего проекта. Подробнее можете прочитать в документации. Добавляем код в loadViewFromXib():

```
let bundle = Bundle(for: type(of: self))
```

Далее получаем Xib по имени:

```
let nib = UINib(nibName:  "GameControlView", bundle: bundle)
```

Создаём и возвращаем сам view:

```
return nib.instantiate(withOwner: self, options: nil).first! as! UIView
```

Мы делаем force unwrap всех значений, так как в данном случае ошибка, если присутствует, будет выявлена на этапе разработки. И, таким образом, её легче найти. Но нужно иметь ввиду, что в общем случае force unwrap — достаточно опасный подход и его нужно избегать.
Как я рассказывал чуть раньше, у нас есть два метода инициализации представления. Переопределим их оба, не забываю вызвать код супер-класса:

```
override init(frame: CGRect) {
    super.init(frame: frame)
}
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
```

Оба метода должны делать одну вещь — загружать наш View из xib-а. Определим ещё одну вспомогательную функцию — setupViews:

```
private func setupViews() {
}
```

И вызовем ее из конструкторов. Добавляем вызов setupViews() в двух init-ах. Внутри функции мы получаем загруженную из Xib-а view. Добавляем код в setupViews():

```
let xibView = loadViewFromXib()
```

Сразу установим размер загруженной view равной границам текущего представления. Добавляем код в setupViews():

```
xibView.frame = self.bounds
```

Чтобы загруженное view заполняло всё доступное пространство при изменениях размеров экрана, воспользуемся механизмом autoresizing masks. Он пришёл в iOS до Autolayout и иногда ока-

зывается весьма полезен. Подробнее о нём, опять же, вы можете прочитать в документации. Сейчас же просто установим маску так, чтобы элемент всегда растягивался по горизонтали и вертикали:

```
xibView.autoresizingMask = [.flexibleWidth, .flexibleHeight]
```

И добавляем наше представление в иерархию:

```
self.addSubview(xibView)
```

Теперь можно посмотреть на результат. Открываем Main.storyboard. Наши элементы управления показались на экране, ошибки auto layout пропали. Нам осталось сделать самую малость — перенести программный интерфейс в новый класс. Открываю файл GameControlViewClass.swift. Интерфейс должен включать поля длительности игры, оставшегося времени, признак активности игры и обработчик нажатия на кнопку начала/остановки. Создадим эти поля:

```
@IBInspectable var gameDuration: Double = 20
@IBInspectable var gameTimeLeft: Double = 7
@IBInspectable var isGameActive: Bool = false
var startStopHandler: (() -> Void)?
```

Также создадим привязки для элементов формы. Открываем GameControlView, открываем ассистента. Создаем IBOutlet для всех трёх элементов: timeLabel, stepper, actionButton. Также добавим IBAction-ы. Создаю действия для степпера и кнопки: stepperChanged, actionButtonTapped. При нажатии на кнопку нужно вызвать обработчик. Добавляем код в actionButtonTapped(_:):

```
startStopHandler?()
```

При изменении свойств объекта нам необходимо обновлять представление. Поэтому скопируем функцию updateUI и адаптируем её. Открываем файл ViewController.swift и копируем метод updateUI(). Мы можем оставить в ней всё как есть, но удалить строку с gameFieldView, так как он нам недоступен:

```
gameFieldView.isShapeHidden = !isGameActive
```

Добавим вызов новой функции при изменении свойств (didSet) gameTimeLeft, isGameActive, а также при изменении значения степпера — в функцию stepperChanged.
Свойство gameDuration — особенное. Для него добавим сеттер и геттер по значению степпера:

```
get {
    return stepper.value
}
set {
```

```
    stepper.value = newValue
    updateUI()
}
```

Вернёмся в основной файл, и обновим код. Удалим отсутствующие более поля: timeLabel, stepper, actionButton. А также метод stepperChanged и поле gameTimeLeft.
Метод actionButtonTapped превратим в обычный метод без переменных. Для этого у метода actionButtonTapped(_:) удаляем @IBAction, добавляем private, удаляем параметры вызова.
Добавим новое поле для контрола. Открываем Main.storyboard, ассистент, добавляем поле gameControl. Удаляем две строки с ошибкой из startGame(). Вместо них устанавливаем время игры. Удаляю строки с ошибкой из startGame(), добавляю код на их место:

```
gameControl.gameTimeLeft = gameControl.gameDuration
```

В функции gameTimerTick используем оставшееся время игры из контрола. Изменяем два вхождения gameTimeLeft на gameControl.gameTimeLeft в методе gameTimerTick(). В updateUI оставляем только строку с gameFieldView, а всё остальное заменяем на gameControl.isGameActive = isGameActive.
Далее настраиваем контрол во viewDidLoad. Добавляем код в конец viewDidLoad():

```
gameControl.startStopHandler = { [weak self] in
    self?.actionButtonTapped()
}
```

И установим время игры по умолчанию — 20 секунд:

```
gameControl.gameDuration = 20
```

Запустим проект. Наше представление загрузилось. Уменьшим время до 4 секунд. Степпер работает. Нажмём кнопку "Начать". Игра запустилась. Понажимаем на фигуры. Время вышло, игра остановилась, контрол корректно обновился. Нажмём "Начать" снова, теперь остановим. Всё работает как задуманно.
Поздравляю, мы успешно выполнили очередной рефакторинг кода — вынесли часть представления в отдельный файл, который можно переиспользовать в других местах проекта. Или других проектах.
В этом длинном видео вы познакомились с проектирование интерфейса в Xib-файлах, концепцией загрузки наследников UIView из кода и из Storyboard, научились использовать представления из Xib-файлов в Storyboard.
В следующем финальном видео мы избавимся от xib-файла, а интерфейс построим на стандартных элементах полностью из кода. Также мы не будем использовать Auto layout, а размещать элементы станем вручную.

1.9. Manual Layout

В прошлый раз мы использовали xib-файл для создания отдельного представления элементов управления игрой. Данный метод достаточно удобен, так как позволяет создавать представления в визуальном редакторе. Когда представление загружается достаточно редко, на нём можно остановиться. Но если вы используете этот методв, например, с таблицей с большим количеством разнородных ячеек, то можете обнаружить проблемы с производительностью приложения.
Загружать xib — это достаточно дорогая операция, намного быстрее, в терминах выполнения, создать view программно. Это первое. Второе — механизм Auto layout-а при всем его удобстве, обрабатывает сложные связи намного дольше, чем ручной расчёт положения элементов. В нашем простом случае, конечно, эти проблемы будет заметить не так уж просто. Тем не менее, в этом видео мы представим, что они есть. И удалим Xib-файл и перейдём на ручное управление положением элементов. Или manual layouting.
Итак, удалим файл GameControlView.xib. Откроем файл GameControlViewClass.swift. Удалим функцию loadViewFromXib, так как загружать view больше не из чего. Также удаляем всё из метода setupViews(). Сейчас мы дополним его новым кодом. Все аутлеты мы заменяем на константы, создавая их сразу при определении. В setupViews добавим каждый элемент в под-view:

```
addSubview(timeLabel)
addSubview(stepper)
addSubview(actionButton)
```

Так как мы планируем задавать размер и положение каждого элемента вручную, нам необходимо установить свойство translatesAutoresizingMaskIntoConstraints в истину:

```
timeLabel.translatesAutoresizingMaskIntoConstraints = true
stepper.translatesAutoresizingMaskIntoConstraints = true
actionButton.translatesAutoresizingMaskIntoConstraints = true
```

В прошлый раз мы использовали autoresizing mask для того, чтобы загруженная view заполняла всю супер-view. На самом деле, во время выполнения программы эти маски автоматически преобразуются в Auto Layout constraint-ы. Маска есть всегда у любого элемента. Если она не задана нами, а также отсутствуют вручную установленные constraint-ы, то они будут созданы как фиксированная ширина и высота, равные соответствующим значениям, указанным при инициализации объекта или при обновлении фрейма. Такое поведение позволяет нам использовать объекты с ручным управлением объектами внутри представлений с AutoLayout.
Свойство translatesAutoresizingMaskIntoConstraints как раз и говорит, нужно ли создавать вспомогательные constraint-ы. Наши IBAction, конечно, тоже не работают. Изменим их сигнатуру — сделаем их простыми функциями, доступными в Objective-C. Заменяем @IBAction на @objc private и удалим параметры в stepperChanged(:), actionButtonTapped(:).
Добавим соответствующие обработчики из кода. Это делается с помощью метода addTarget:

```
stepper.addTarget(
    self,
    action: #selector(stepperChanged),
    for: .valueChanged
)
actionButton.addTarget(
    self,
    action: #selector(actionButtonTapped),
    for: .touchUpInside
)
```

Здесь снова применяем селекторы, отсюда аннотация objc. Названия событий говорят сами за себя. Установку фреймов всех элементов будем выполнять в функции layoutSubviews, которую переопределим.
Вызываем super.layoutSubviews(). Чтобы определить, сколько занимает места та или иная view без каких-либо ограничений, можно использовать свойство intrinsicContentSize. Начнём с установки frame-а степпера. Получаем его размер:

```
let stepperSize = stepper.intrinsicContentSize
```

И далее устанавливаем его frame, который имеет тип данных CGRect. Это структура, описываемая координатами левого верхнего угла, и размером — шириной и высотой.

```
stepper.frame = CGRect(origin:
```

origin — это координата левого верхнего угла. Нам необходимо прижать его к правому краю, отсюда координаты:

```
stepper.frame = CGRect(
    origin: CGPoint(
        x: bounds.maxX - stepperSize.width,
        y: bounds.minY
    ),
    size: stepperSize
)
```

Далее аналогичным образом получаем размер timeLabel:

```
let timeLabelSize = timeLabel.intrinsicContentSize
```

И прижимаем его с левому краю. При этом вертикально его необходимо разместить по центру по отношению к степперу:

```
timeLabel.frame = CGRect(
    origin: CGPoint(
        x: bounds.minX,
        y: bounds.minY + (stepperSize.height - timeLabelSize.height) / 2
    ),
    size: timeLabelSize
)
```

Кнопку нужно разместить на некотором расстоянии под степпером и отцентрировать по горизонтали. Получаем её размер:

```
let buttonSize = actionButton.intrinsicContentSize
```

Чтобы было легче изменять расстояние между элементами в будущем, опеределим расстояние между степпером и кнопкой в отдельной константе:

```
private let actionButtonTopMargin: CGFloat = 8
```

И используем её для установки frame-а кнопки:

```
actionButton.frame = CGRect(
    origin: CGPoint(
        x: bounds.minX + (bounds.width - buttonSize.width) / 2,
        y: stepper.frame.maxY + actionButtonTopMargin
    ),
    size: buttonSize
)
```

Мы написали код расстановки наших вложенных представлений. Теперь посмотрим, что получилось на Storyboard. Открываем файл Main.storyboard. Подождём, пока наш файл перекомпилируется. Представление выглядит не так, как мы планировали. Кроме того, у нас снова появились ошибки auto layout в текущем файле. Проблема заключается в том, что мы не возвращаем желаемый размер своего представления. Тот самый intrinsicContentSize. Чтобы это исправить, вернёмся к коду нашего представления. Открываем файл GameControlViewClass.swift. Переопределим intrinsicContentSize под init?(coder:).
Для его расчёта нам снова потребуются размеры вложенных представлений. Добавим соответствующие константы в intrinsicContentSize:

```
let stepperSize = stepper.intrinsicContentSize
let timeLabelSize = timeLabel.intrinsicContentSize
let buttonSize = actionButton.intrinsicContentSize
```

Ширина представления складывается из размера текста времени, степпера и желаемого расстояния между ними. Для расстояния снова заведём отдельную константу:

```
private let timeToStepperMargin: CGFloat = 8
```

Теперь рассчитаем высоту. Добавляем код в конец intrinsicContentSize:

```
let width = timeLabelSize.width + timeToStepperMargin + stepperSize.width
```

Высота представления равна высоте степпера, плюс расстояние до кнопки, плюс сама кнопка:

```
let height = stepperSize.height + actionButtonTopMargin + buttonSize.height
```

Теперь можно вернуть итоговый размер:

```
return CGSize(width: width, height: height)
```

Посмотрим, что получилось. Открываю Main.storyboard. Ждём обновления представления. Ошибка с autolayout пропала. Но текст времени и кнопка до сих пор не видны. Но ведь мы не установили никакой текст для них!
Легче всего победить данную проблему — это добавить вызов updateUI в конец setupViews.
Посмотрим на результат. Время появилось, но кнопки не видно. А давайте запустим приложение. Кнопки всё также нет. А если нажать на то место, где она должна быть... То игра волшебным образом запускается... И останавливается, если нажать на пустое место снова. На самом деле никакой магии здесь нет. Кнопка присутствует на законном месте, только мы её не видим, так как её текст — белый. Всё дело в том, что при добавлении кнопки через Interface Builder к ней автоматически применяется цвет из свойства tintColor. При создании кнопки из кода этого не происходит, нужно всё делать вручную:

```
actionButton.setTitleColor(actionButton.tintColor, for: .normal)
```

Проверим результат сразу в Storyboard. Кнопка появилась. Точнее сказать, пРоявилась.
Опять какая-то магия! Что за свойство, которое мы нигде не определяли? tintColor — это одно из свойств, которое можно установить через UIAppearance. Он, в свою очередь, предоставляет удобный механизм для быстрой стилизации приложения — задания различных глобальных свойств. Как, например, tintColor. Подробное рассмотрение и UIAppearance в целом, и tintColor в частности выходит за рамки данного видео.
Ещё раз запустим игру и убедимся, что всё работает, как запланировано. Когда текст Label и кнопк становятся длиннее изначально установленного — он обрезается. Так происходит потому, что система не вызывает layoutSubviews при изменении этих же свойств. Так же, как происходило при прорисовке треугольника. Чтобы это исправить, вызовем метод setNeedsLayout. Его мы применяем, когда нужно не перерисовать представление, а перераспределить Subviews внутри нашего View. Остановим приложение, откроем GateControlView И добавим код в updateUI():

```
setNeedsLayout()
```

Запустим приложение. Начинаем игру. Отлично. Текст больше не обрезается. Останавливаем. Повернём экран, и обратно. Всё работает.
Поздравляю! В этом видео вы познакомились с созданием собственных представлений из кода, с ручным управлением frame-ами каждого из вложенных представлений, и даже затронули tintColor из состава UIAppearance.
Это финальное видео на тему представлений. На этой неделе вы узнали, что такое представления и как их использовать для создания пользовательского интерфейса, как реагировать на взаимодействие пользователя с приложением, а также научились адаптировать интерфейс под разные экраны. И, конечно же, написали игру, которая поможет вам скоротать время. На этой неделе вам ещё предстоит решить практическое задание, в котором вы сможете закрепить полученные знания на практике. Удачи!