**Лекция 15 Графика и сенсорика**

В этой лекции обсуждаются различные способы добавления «визуального движения» (графической анимации) в приложения Android. Она охватывает [2D-рисование](https://developer.android.com/guide/topics/graphics/2d-graphics.html) с пользовательскими представлениями, [анимацию свойств](https://developer.android.com/guide/topics/graphics/prop-animation.html) (также используемую в эффектах материалов) и то, как обрабатывать [сенсорные жесты](https://developer.android.com/training/gestures/index.html) .

В этой лекции используется код, найденный по адресу <https://github.com/info448/lecture15-graphics> .

**15.1 Рисование графики**

Android предоставляет [API 2D-графики,](https://developer.android.com/guide/topics/graphics/2d-graphics.html) схожий по духу и использованию с [HTML5 Canvas API](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API) : он предоставляет *интерфейс* , с помощью которого разработчик может *программно* генерировать 2D-изображения на основе растра. Этот API можно использовать для рисования графиков, показа обработанных изображений и даже выполнения анимаций!

Как и в HTML5, в Android этот API доступен через класс [50.](https://info448.github.io/graphics.html#fn50) Подобно элементу HTML5 или классу Java SE , Android предоставляет графический «контекст», на котором разработчик может «рисовать» прямоугольники, круги и даже изображения ( ) для отображения на экране.[Canvas](https://developer.android.com/reference/android/graphics/Canvas.html)<canvas>[Graphics2D](https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html)CanvasBitmaps

**Пользовательские представления**

Чтобы рисовать картинки, нам нужно иметь View для рисования (который предоставит Canvasконтекст). Самый простой способ получить этот View — создать его самостоятельно: определить **пользовательский View** , который мы можем указать как имеющий «рисованный» вид. Мы можем сделать свой собственный специальный View View, создав подкласс (помните: Buttonsи EditTextsявляются всего лишь подклассами View!), а затем заполнив функцию обратного вызова ( onDraw()), которая определяет, как это Viewотображается на экране.

Слово *«рендерить»* в данном случае означает «вызывать к жизни», то есть создавать графическое изображение и выводить его на экран.

Настройка Viewне слишком сложна, но для экономии времени в коде лекции представлен полный пример в форме класса DrawingView. Ниже приведены заметки о реализации этого класса:

* Класс extends Viewдля создания подкласса базового Viewкласса.
  + Также обратите внимание, как мы указываем пользовательское представление как <view>элемент в XML-макете, указывая classатрибут.
* Viewимеет ряд различных конструкторов. Мы переопределяем их все, поскольку каждый из них используется другой частью системы Android (и поэтому нам нужно предоставить пользовательские реализации для каждого). Однако мы заставим каждый вызывать «последний» для того, чтобы фактически выполнить какую-либо настройку.
  + В конструкторе мы настраиваем [Paint](http://developer.android.com/reference/android/graphics/Paint.html), который представляет *то, как* мы хотим рисовать: цвет, ширину обводки, размер шрифта, параметры сглаживания и т. д. В основном мы будем использовать Paintобъекты для цвета.
* Мы переопределяем **onSizeChanged()**обратный вызов, который будет выполнен при Viewизменении размера. Это происходит при инфляции (которая происходит как часть Activity onCreate, то есть обратный вызов будет вызван при вращении). Этот обратный вызов может действовать немного как Fragment onCreateView(), в том смысле, что мы можем выполнять работу, основанную на созданном View в этой точке.
* Мы также должны переопределить **onMeasure()**обратный вызов, как [рекомендуют](https://developer.android.com/guide/topics/ui/custom-components.html#custom) руководства Android. Этот обратный вызов используется для указания того, как View должен изменяться в зависимости от того, задана ли его ширина или высота как wrap\_content. Это особенно важно для создания таких вещей, как пользовательские кнопки. Однако наш пример пропустит это из-за времени и места, поскольку DrawingViewвсегда должен занимать весь экран.
* Наконец, мы переопределяем **onDraw()**, и вот тут-то и происходит магия. Этот метод вызывается всякий раз, когда Viewнужно отобразить (как paintComponent()в фреймворке Swing). Этот обратный вызов передает Canvasобъект в качестве параметра, предоставляя контекст, на который мы можем опираться!
  + Как и все остальные обратные вызовы жизненного цикла: **мы никогда не вызываем onDraw()!!** Система Android UI вызывает его за нас!

Предоставленную информацию Canvasможно использовать несколькими способами:

* Мы можем вызывать такие методы, как drawColor()(для заполнения фона), drawCircle()или drawText()рисовать на нем определенные фигуры или сущности. Эти методы аналогичны по использованию HTML5 Canvas.
* Мы также можем нарисовать [Bitmap](https://developer.android.com/reference/android/graphics/Bitmap.html), который представляет собой графический растр (например, двумерный массив пикселей). Если у нас есть Bitmap, мы можем задать цвета отдельных пикселей (используя setPixel(x,y,color)), а затем нарисовать Bitmapна холсте (тем самым дважды буферизируя!). Это полезно для рисования на уровне пикселей или когда вы хотите создать более сложную графику или произведение искусства.
  + Если вы использовали MS Paint, то разница заключается в параметрах рисования фигур и «увеличенной» раскраске пикселей.

Обратите внимание, что мы заставляем Canvas«перерисовываться» (чтобы наш onDraw()метод был вызван) путем вызова invalidate(): Viewэто заставляет Android пересоздавать его, тем самым перерисовывая. Повторно вызывая, invalidate()мы можем сделать что-то похожее на анимацию!

* Мы можем сделать это с помощью рекурсивного цикла, вызвав invalidate()в конце onDraw(). Это позволяет нам «запросить» Android вызвать его onDraw()снова после завершения, но **мы не вызываем его** .
* В качестве демонстрации мы можем убрать Ballслайд с экрана, слегка изменив его положение:
* ball.cx += ball.dx;

ball.cy += ball.dy;

Мы также можем добавить столкновения со стенами:

**if**(ball.cx + ball.radius > viewWidth) { *//left bound*

ball.cx = viewWidth - ball.radius;

ball.dx \*= -1;

}

**else** **if**(ball.cx - ball.radius < 0) { *//right bound*

ball.cx = ball.radius;

ball.dx \*= -1;

}

**else** **if**(ball.cy + ball.radius > viewHeight) { *//bottom bound*

ball.cy = viewHeight - ball.radius;

ball.dy \*= -1;

}

**else** **if**(ball.cy - ball.radius < 0) { *//top bound*

ball.cy = ball.radius;

ball.dy \*= -1;

}

*Анимация* — это процесс «придания жизни» (от латинского ***«anima»*** ). Мы склонны иметь в виду придание чему-либо **движения** — создание видимости движения объекта с течением времени. Подумайте, что это значит для того, как люди понимают «жизнь».

Видеоанимация подразумевает показ последовательности изображений с течением времени. Если изображения сменяются достаточно быстро, то человеческий мозг интерпретирует их как часть одного и того же последовательного движения, и любые объекты на этих изображениях будут считаться «движущимися». Каждое изображение в этой последовательности называется «кадром». Фильм, как правило, имеет частоту 24 кадра в секунду ( **fps** ), видео — 29,97 кадров в секунду, а видеоигры стремятся к 60 кадрам в секунду. Любое видео, идущее со скоростью не менее 16 кадров в секунду, будет восприниматься как в основном плавное движение.

Достичь этих 16 кадров в секунду на самом деле может быть довольно сложно, так как определение *того, что* рисовать, требует больших вычислительных затрат! Если мы вычисляем каждый пиксель на дисплее 600x800, то нам нужно вычислить полмиллиона пикселей! При 60 кадрах в секунду это 28 миллионов пикселей в секунду. Для масштабирования, процессор 1 ГГц может выполнять 1 миллиард операций в секунду — так что если каждый пиксель требует 5 операций, мы используем 15% нашего процессора. Это часть того, почему большинство графических систем используют выделенный GPU (графический процессор) — он обеспечивает массивное распараллеливание для ускорения этого процесса.

**SurfaceViews**

Поскольку все эти вычисления (на уровне пикселей) могут занять некоторое время, мы хотим быть осторожными, чтобы они не блокировали поток пользовательского интерфейса! Вместо этого мы хотели бы выполнить рисование в фоновом режиме. Сама визуализация должна происходить в потоке пользовательского интерфейса, но мы хотим, чтобы вся *логика рисования* происходила в фоновом потоке, чтобы работа пользовательского интерфейса была максимально быстрой (представьте: повесить заранее напечатанный плакат вместо того, чтобы печатать его полностью).

* Однако an AsyncTaskне подходит, потому что мы хотим делать это неоднократно. Аналогично, an IntentServiceне сможет реагировать достаточно быстро, если нам нужно ждать, пока система доставит Intents.

Android предоставляет класс, специально разработанный для «рисования» в фоновом потоке: [**SurfaceView**](https://developer.android.com/reference/android/view/SurfaceView.html). В отличие от basic Views, которые являются несколько эфемерными, a SurfaceViewвключает выделенную поверхность рисования, с которой мы можем взаимодействовать в отдельном потоке. Он разработан для поддержки этой потоковой работы, не требуя *слишком* много кода синхронизации.

Настройка этих функций требует еще больше работы, поэтому полный пример ( DrawingSurfaceView) снова приведен в коде лекции:

* Этот класс extends SurfaceViewи *реализует* SurfaceHolder.Callback . A SurfaceHolder— это объект, который «держит» (содержит) базовую поверхность для рисования; несколько похоже на ViewHolderшаблон, используемый с адаптером. Мы взаимодействуем с SurfaceViewчерез держатель, чтобы убедиться, что мы *потокобезопасны* : только один поток взаимодействует с поверхностью в каждый момент времени.
  + В общем, будет два потока, обменивающихся использованием держателя: наш фоновый поток, который рисует на поверхности («печатает плакат»), а затем поток пользовательского интерфейса, который показывает поверхность пользователю («вывешивает напечатанный плакат»). Вы можете думать о держателе *как* о постере в этой метафоре!
* Мы регистрируем держателя в конструкторе с помощью предоставленного getHolder()метода и регистрируем себя для обратных вызовов при изменении держателя. Мы также создаем новый экземпляр Runnable, который будет представлять обратный вызов, выполняемый в отдельном (фоновом) потоке для выполнения рисования.
* Интерфейсу SurfaceHolder.Callbackтребуются методы, определяющие момент изменения поверхности, поэтому мы их заполняем:
  + onSurfaceCreated()запускает наш фоновый поток (потому что поверхность уже создана)
  + onSurfaceChanged()в конечном итоге действует очень похоже onSizeChanged()на базовыйDrawingView
  + onSurfaceDestroyed()останавливает фоновый поток «безопасным» способом (код адаптирован из Google)
* Если мы посмотрим на Runnable, то это по сути бесконечный цикл:
  + Возьмите Surface Canvas, «заблокируйте» его, чтобы использовать только в этом (фоновом) потоке.
  + Рисуйте на нем.
  + Затем «выталкиваем» холст обратно в остальной мир, по сути говоря: «Мы закончили рисовать, теперь вы можете показать его пользователю».

В целом, этот процесс заставит Surface «перерисовываться» как можно быстрее, и все это без блокировки потока пользовательского интерфейса! Это отлично подходит для анимации, которую можно контролировать и синхронизировать (например, в update()вспомогательном методе, обновляя переменные только с определенной скоростью). Более того, он обеспечивает рисуемую поверхность, с которой можно взаимодействовать!

И это дает нам поверхность для рисования, с которой мы можем взаимодействовать таким же образом, используя ту же логику движения/взаимодействия.

* Это демонстрирует один из способов создания низкоуровневой игровой и анимационной логики с использованием базовой работы Java; не требуется никаких специальных игровых движков (хотя они тоже существуют).

**15.2 Прикосновения и жесты**

На данный момент у нас есть простая анимация и движение, но мы хотели бы сделать их более интерактивными. Наш Viewзанимает весь экран, поэтому мы не хотим добавлять кнопки, но есть и другие доступные варианты.

В частности, мы можем добавить [жесты касания](http://developer.android.com/training/gestures/index.html) . Сенсорные экраны — это огромная часть устройств Android (и мобильных устройств в целом, особенно с момента появления первого iPhone), которые знакомы большинству пользователей. Мы уже косвенно использовали сенсорный интерфейс, когда пользователи нажимали на кнопки (теоретически с помощью сенсорного экрана). Но здесь нас интересуют не только нажатия кнопок, которые действительно могут исходить откуда угодно. Вместо этого нас интересует, как мы можем реагировать на *то, где* пользователь может коснуться экрана, и даже на различные способы, которыми пользователь может *ласкать* экран: броски, перетаскивания, мультитач и т. д.

Устройства Android автоматически распознают различные сенсорные взаимодействия (именно так работают кнопки); мы можем реагировать на эти **сенсорные события** , переопределяя onTouchEvent()обратный вызов, который выполняется всякий раз, когда происходит что-то, связанное с сенсорным экраном.

* Например, мы можем выйти из события, чтобы увидеть, какие подробности оно содержит.

Есть *много* вещей, которые могут вызвать TouchEvents, поэтому большая часть нашей работы заключается в попытках определить, какой *семантический* «жест» сделал пользователь. К счастью, Android предоставляет ряд вспомогательных методов и классов, которые помогут с этим.

Самый простой — это MotionEvent#getActionMasked(), который извлекает «тип действия» события из записанного движения:

int action = motionEvent.getActionMasked(); *//get action constant*

float x = event.getX(); *//get location of event*

float y = event.getY() - getSupportActionBar().getHeight(); *//closer to center...*

**switch**(action) {

**case** (MotionEvent.ACTION\_DOWN) : *//put finger down*

*//e.g., move ball*

view.ball.cx = x;

view.ball.cy = y;

**return** **true**;

**case** (MotionEvent.ACTION\_MOVE) : *//move finger*

*//e.g., move ball*

view.ball.cx = x;

view.ball.cy = y;

**return** **true**;

**case** (MotionEvent.ACTION\_UP) : *//lift finger up*

**case** (MotionEvent.ACTION\_CANCEL) : *//aborted gesture*

**case** (MotionEvent.ACTION\_OUTSIDE) : *//outside bounds*

**default** :

**return** **super**.onTouchEvent(event);

}

Это позволяет нам реагировать на базовые прикосновения. Например, мы можем сделать так, чтобы нажатия ( ACTION\_DOWN) «телепортировали» мяч туда, где мы щелкаем! Мы также можем использовать ACTION\_MOVEсобытия, чтобы позволить нам перетаскивать мяч.

**Расширенные жесты**

Мы также можем обнаруживать и реагировать на более сложные жесты: длинные нажатия, двойные нажатия или «броски» (щелчок или смахивание по экрану). Как и в случае с базовыми жестами, спецификация Material Design описывает некоторые [конкретные шаблоны](https://www.google.com/design/spec/patterns/gestures.html#gestures-drag-swipe-or-fling-details) , которые следует учитывать при использовании этих взаимодействий.

Android предоставляет [GestureDetector](https://developer.android.com/training/gestures/detector.html#detect)класс, который можно использовать для идентификации этих действий. Самый простой способ использовать этот класс, особенно когда нас интересует определенный жест (например, рывок), — это *расширить его* GestureDetector.SimpleOnGestureListener , чтобы создать собственный «слушатель» жестов. Затем мы можем переопределить обратные вызовы для жестов, на которые нам интересно реагировать: например, onFling().

* Обратите внимание, что в официальной документации говорится, что мы также должны переопределить onDown()метод и заставить его возвращать true, чтобы указать, что мы «потребили» (обработали) событие — аналогично тому, что мы сделали с OptionsMenus. Если мы вернем false из этого метода, то *«система предполагает, что вы хотите игнорировать остальную часть жеста, и другие методы GestureDetector.OnGestureListener никогда не будут вызваны».* Однако в моем тестировании обнаружение жестов работает в любом случае, но пока мы будем следовать спецификации.

Мы можем создать экземпляр GestureDetector, передав наш слушатель в GestureDetectorCompatконструктор:

mDetector = **new** GestureDetectorCompat(**this**, **new** MyGestureListener());

Затем в обратном вызове Activity onTouchEvent()мы можем передать событие в Gesture Detector для обработки:

boolean gesture = **this**.mDetector.onTouchEvent(event); *//check gestures first!*

**if**(gesture){

**return** **true**;

}

* Поскольку детектор onTouchEvent()возвращает значение, booleanуказывающее, был ли обнаружен жест, мы можем проверить, следует ли нам в противном случае обрабатывать жест самостоятельно.

Это дает нам возможность «бросить» мяч, взяв обнаруженную *скорость* броска и назначив ее в качестве скорости мяча. Обратите внимание, что нам нужно инвертировать скорости, поскольку они зарегистрированы как «обратные» от координат, которые использует наша система рисования (хотя это не соответствует задокументированным примерам). Уменьшение скорости до 3% дает мне разумную скорость движения мяча. Мы также можем замедлить мяч на 1% при каждом обновлении, чтобы он дрейфовал до остановки!

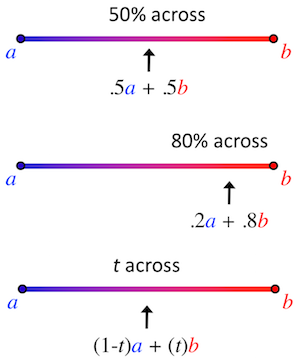
**15.3 Анимация свойств**

Мы увидели, как можно создавать анимацию, просто настраивая рисунок, который мы делаем в каждом кадре. Это отлично подходит для игр или других сложных анимаций, если мы хотим иметь *большой* контроль над нашим графическим макетом... но иногда мы хотим иметь некоторые более простые, специфичные для платформы эффекты (которые работают более плавно!) Android на самом деле включает в себя ряд различных систем анимации, которые можно использовать внутри и *между* Views:

* Ранее мы говорили о некоторых [Material Animations,](https://developer.android.com/training/material/animations.html) встроенных в библиотеку поддержки Material Design; в частности, мы обсуждали создание Activity transition. Android также включает в себя надежную структуру для [Scene Transitions](https://developer.android.com/training/transitions/index.html) даже за пределами Material; см. [Adding Animations](https://developer.android.com/training/animation/index.html) для получения более подробной информации.
* Android также поддерживает [OpenGL Animations](https://developer.android.com/guide/topics/graphics/opengl.html) для создания 3D-анимированных систем. Для этого требуется знание OpenGL API.

В этом разделе мы обсудим, как использовать [**Property Animation**](https://developer.android.com/guide/topics/graphics/prop-animation.html) . Это общая структура анимации, в которой вы указываете начальное состояние для *свойства* объекта (атрибута), конечное состояние для этого свойства и длительность анимации; затем системы Android изменяют свойство от начального состояния до конечного в течение этого периода времени — тем самым создавая анимацию!

* Изменение состояния свойства с течением времени (то есть в любом заданном «кадре») вычисляется с помощью [**интерполяции**](https://en.wikipedia.org/wiki/Interpolation) . Это по сути «взвешенное среднее» между начальным и конечным состояниями, где вес определяется тем, насколько вы близки к «началу» или «концу». Хотя мы часто используем *линейную интерполяцию* (так что 70% по горизонтали означает, что конец получает 70% веса), также можно использовать *нелинейную интерполяцию* (например, вам нужно получить 70% по горизонтали, чтобы конец имел 50% веса).



Линейная интерполяция цветов

Основным движком для выполнения такого рода интерполированной анимации в Android является класс [51.](https://info448.github.io/graphics.html#fn51) Этот класс позволяет указать начальное состояние, конечное состояние и продолжительность анимации. Затем он сможет пройти и вычислить все «промежуточные» значения на протяжении всей интерполированной анимации. Класс предоставляет ряд статических методов (например, , , ), которые создают «аниматоры» для интерполяции различных *типов значений* . Например:[ValueAnimator](https://developer.android.com/guide/topics/graphics/prop-animation.html#value-animator)ValueAnimator.ofInt().ofFloat().ofArgb()

ValueAnimator animation = ValueAnimator.ofFloat(0f, 1f); *//interpolate between 0 and 1*

animation.setDuration(1000); *//over 1000ms (1 second)*

animation.start(); *//run the animation*

Конечно, простое выполнение этой интерполяции с течением времени не дает никакого видимого результата — она изменяет числа, но эти числа ничему не соответствуют.

Мы можем получить доступ к интерполированным значениям, зарегистрировав [слушателя](https://developer.android.com/guide/topics/graphics/prop-animation.html#listeners) и переопределив обратный вызов, который мы хотим наблюдать (например, onAnimationUpdate()из ValueAnimator.AnimatorUpdateListener). Но чаще всего мы хотим, чтобы наша интерполированная анимация изменила *свойство* некоторого объекта, например, цвет View, положение Button или переменные экземпляра объекта, такого как Ball.

Мы можем легко сделать это с помощью подкласса [52.](https://info448.github.io/graphics.html#fn52) Этот подкласс запускает анимацию так же, как и , но имеет встроенную функциональность для изменения свойства объекта на каждом интерполированном шаге. Он делает это, вызывая **сеттер** для этого свойства — таким образом, объект должен иметь [метод сеттера](http://docs.oracle.com/javaee/6/tutorial/doc/gjbbp.html) для свойства, которое мы хотим анимировать:[ObjectAnimator](https://developer.android.com/guide/topics/graphics/prop-animation.html#object-animator)ValueAnimator

*//change the "alpha" property of foo: will call `foo.setAlpha(float)`*

ObjectAnimator anim = ObjectAnimator.ofFloat(foo, "alpha", 0f, 1f);

anim.setDuration(1000);

anim.start();

* В этом примере объект будет изменен путем вызова setAlpha()метода (имя метода генерируется из имени свойства в обычном стиле CamelCasing).
* Если у объекта нет такого сеттера, например, потому что мы используем класс, предоставленный кем-то другим, мы можем либо создать «обертку», которая вызовет соответствующую функцию изменения, либо просто использовать ValueAnimatorс соответствующим прослушивателем.

Например, мы можем использовать этот подход для изменения размера или положения круга с помощью интерполированной анимации (сделать его «пульсирующим»).

* Обратите внимание, что мы просто меняем свойство объекта; единственная причина, по которой мы видим измененный рисунок, заключается в том, что Android постоянно обновляет наш SurfaceView.

Методы ObjectAnimatorинтерполяции также поддерживают ряд вариаций:

* Пока у объекта есть соответствующий **геттер** , можно передать аниматору только конечное значение (указывающее, что анимация должна интерполироваться «из текущего состояния в указанное конечное состояние»).
* Мы можем использовать .setRepeatCount(ObjectAnimator.INFINITE)и , .setRepeatMode(ObjectAnimator.REVERSE)чтобы заставить его повторяться туда и обратно.

Если мы хотим включить несколько анимаций последовательно, мы можем использовать [AnimatorSet](https://developer.android.com/guide/topics/graphics/prop-animation.html#choreography), который предоставляет нам методы, используемые для указания порядка:

*//example from docs*

ObjectAnimator animX = ObjectAnimator.ofFloat(obj, "x", 50f);

ObjectAnimator animY = ObjectAnimator.ofFloat(obj, "y", 100f);

AnimatorSet animSetXY = **new** AnimatorSet();

animSetXY.playTogether(animX, animY);

animSetXY.start();

Анимации AnimatorSet могут быть сложными, и мы можем захотеть использовать их повторно. Поэтому лучше всего вместо этого определить их в XML как [ресурсы](https://developer.android.com/guide/topics/graphics/prop-animation.html#declaring-xml) . Ресурсы анимации помещаются в /res/animatorкаталог ( **а не** в папку /res/anim/, которая предназначена для [View Animations](https://developer.android.com/guide/topics/graphics/view-animation.html) ).

**<set** android:ordering="together"**>** *<!-- together is default -->*

**<objectAnimator**

android:propertyName="x"

android:duration="500"

android:valueTo="400"

android:valueType="intType"**/>**

*<!-- ... -->*

**</set>**

* Полную схему XML см. в разделе [«Ресурсы анимации свойств»](https://developer.android.com/guide/topics/resources/animation-resource.html#Property)[53 .](https://info448.github.io/graphics.html#fn53)
* Обратите внимание, что определение анимаций как ресурсов также означает, что мы можем легко использовать разные конфигурации устройств для использования разных анимаций (например, объекты могут двигаться быстрее на больших дисплеях).

Чтобы использовать XML, нам нужно будет **расширить** ресурс Animator, так же, как мы это делали с Layouts:

ObjectAnimator anim = (ObjectAnimator)AnimatorInflater.loadAnimator(context, R.anim.my\_animation);

anim.setTarget(myObject);

anim.start();

Обратите внимание, что мы также можем использовать эту же структуру для анимации изменений в Views: кнопках, изображениях и т. д. Представления являются объектами и имеют свойства (вместе с соответствующими методами *получения* и *установки* ), поэтому мы можем использовать просто ObjectAnimator! См. [Анимация представлений](https://developer.android.com/guide/topics/graphics/prop-animation.html#views) для списка свойств, которые мы можем изменить (список, который включает x, y, rotation, alpha, scaleX, scaleY, и другие)

Чтобы сделать этот процесс еще проще, Android также предоставляет ViewPropertyAnimatorкласс. Этот Animator может легко анимировать несколько свойств вместе (одновременно), и делает это гораздо более эффективно. Мы можем получить доступ к этому Animator через View#animate()метод. Затем мы вызываем соответствующие методы сочетания клавиш для этого Animator, чтобы «добавить» дополнительные анимации свойств в набор анимаций:

*//animate x (to 100) and y (to 300) together!*

myView.animate().x(100).y(300);

Это позволяет легко задавать достаточно сложную анимацию свойств для Viewобъектов.

* Но на самом деле, если вы хотите анимировать изменения макета на современном устройстве, вам следует использовать [переходы](http://developer.android.com/training/transitions/index.html) , такие как те, которые мы использовали в Material Design.

Есть [еще много способов](https://developer.android.com/guide/topics/graphics/prop-animation.html) настроить именно то, что вы хотите, чтобы ваша анимация была. Вы также можете посмотреть [официальные демо](https://android.googlesource.com/platform/development/+/master/samples/ApiDemos/src/com/example/android/apis/animation) для большего количества примеров различных стилей анимации.

1. <https://developer.android.com/reference/android/graphics/Canvas.html>[↩](https://info448.github.io/graphics.html#fnref50)
2. <http://developer.android.com/guide/topics/graphics/prop-animation.html#value-animator>[↩](https://info448.github.io/graphics.html#fnref51)
3. <http://developer.android.com/guide/topics/graphics/prop-animation.html#object-animator>[↩](https://info448.github.io/graphics.html#fnref52)
4. <http://developer.android.com/guide/topics/resources/animation-resource.html#Property>[↩](https://info448.github.io/graphics.html#fnref53)