[Как сделать custom View для Android Ч3.: анимация, сглаживание](http://codeandlife.ru/android/3-creating-custom-android-views-part-3.html" \o "Как сделать custom View для Android. Ч3.: анимация, сглаживание )

Введение

Во второй части этой серии постов мы начали делать [view для отображения графика](http://codeandlife.ru/android/2-creating-custom-android-views-part-2.html), предполагаю, что вы прочитали эту статью. Если это не так, я рекомендую вам сделать это. Итак, продолжим работу над нашим view компонентом.

У меня есть идея сделать в верхней части графика три кнопки, с помощью которых пользователь сможет переключать данные и отображать их. В данном случае это будут различные категории тренировок: ходьба, бег и езда на велосипеде.

Если мы реализуем кнопки и установку новых данных, а потом запустим приложение, то мы не увидим никаких изменений в графике. Метод setChartData() устанавливает новые данные, но view не отображает их. Почему? Потому что мы забыли сообщить view компоненту о том, что это надо сделать. Перерисовка view выполняется с помощью метода invalidate(). Если мы добавим в метод setChartData() вызов метода invalidate() - это решит проблему, график будет обновляться при смене данных. Несмотря на полученный результат, можно сделать view компонент еще лучше, добавив анимацию, возникающую при смене данных.

Для того чтобы реализовать анимацию, нужно выполнить два условия. Во-первых, нужен алгоритм, который будет изменять данные графика шаг за шагом, от старых значений к новым. Во-вторых, нужен какой-то механизм, позволяющий обновлять view с определенной периодичностью.

Давайте начнем с данных. Существует несколько вариантов, как это можно реализовать. Самый очевидный - это простая линейная интерполяция.

Dynamics класс

Для решения этой задачи, мы будем использовать класс, который я назвал Dynamics. Объект Dynamics — это, по сути точка, которая имеет позицию и скорость. У нее также есть конечная позиция, куда она в итоге должна придти, и метод update(), который обновляет позицию и скорость . В сокращенном виде класс выглядит так.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class Dynamics {      private static final float TOLERANCE = 0.01f;        /\*\* Конечная позиция точки\*/      private float targetPosition;        /\*\* Текущая позиция точки\*/      private float position;        /\*\* Текущая скорость точки \*/      private float velocity;        /\*\* Время последнего обновления \*/      private long lastTime;        /\*\* Коэффициент упругости \*/      private float springiness;        /\*\* Коэффициент затухания \*/      private float damping;    ....       public void update(long now) {          long dt = Math.min(now - lastTime, 50);          velocity += (targetPosition - position) \* springiness;          velocity \*= (1 - damping);          position += velocity \* dt / 1000;          lastTime = now;     }    .....    } |

Наибольший интерес здесь представляет метод update().  В первой строке метода выполняется вычисление временного интервала, прошедшего с момента последнего обновления. Чтобы избежать возможных сбоев в анимации, максимально возможное время ограничено 50 мс.

В следующей строке выполняется обновление скорости движения точки, величина которой зависит от удаления до конечной позиции и коэффициента упругости (springiness).  Далее обновленное значение скорости умножается на коэффициент затухания, который принимает значения от 0 до 1. Если не использовать коэффициент затухания, точка будет продолжать двигаться всегда. Можно воспринимать эту часть как амортизатор, который гасит колебания в подвеске.

В конце метода мы обновляем позицию точки, используя простое интегрирование Эйлера, и сохраняем время обновления, для того чтобы вычислить временной интервал в следующий раз.

Такой алгоритм примерно описывает поведение точки "прикрепленной" к пружине. При высоком значении коэффициента упругости точка будет ускоряться к конечной позиции быстрее и будет колебаться вокруг нее. При увеличении коэффициента затухания, ускорение замедлится и, если коэффициент достаточно высок, точка не будет колебаться.

Для того чтобы знать, когда нужно остановить анимацию, мы добавим следующий метод.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public boolean isAtRest() {      final boolean standingStill = Math.abs(velocity) < TOLERANCE;      final boolean isAtTarget = (targetPosition - position) < TOLERANCE;      return standingStill && isAtTarget;  } |

Метод возвращает true, если точка находится в конечной позиции и если скорость равна нулю. Поскольку проверять равенство двух float значений не очень хорошая идея, мы проверяем насколько эти значения близки к друг другу. В данной случае TOLERANCE равна 0,01 и для нас такой точности достаточно.

Использование dynamics

Чтобы использовать класc dynamics, нужно обновить view компонент, который мы создавали в предыдущей части. Для этого нужно поменять тип datapoints массива в тех методах, где он используется. Это несложно. Например, разница между старым и новым методами drawLineChart() будет выглядеть так.

Старый код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...  Path path = new Path();  path.moveTo(getXPos(0), getYPos(datapoints[0]), maxValue);  for (int i = 1; i < datapoints.length; i++) {      path.lineTo(getXPos(i), getYPos(datapoints[i]), maxValue);  }  ... |

Новый код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...  Path path = new Path();  path.moveTo(getXPos(0), getYPos(datapoints[0].getPosition(), maxValue));  for (int i = 1; i < datapoints.length; i++) {      path.lineTo(getXPos(i), getYPos(datapoints[i].getPosition(), maxValue));  }  ... |

В последнем случае для получения новой точки мы используем функцию getPosition().

Остальные изменения лучше посмотреть в коде проекта.

Запуск анимации

Теперь нам нужно решить, как обновлять позиции точек и перерисовывать экран. Я буду использовать для этого объект типа Runnable. Runnable – это интерфейс с методом run(). Любой объект, реализующий этот интерфейс, может быть запущен в отдельном потоке с помощью класса Thread. В  данном случае, мы будет запускать задачу в потоке пользовательского интерфейса (UI-thread), поскольку только из него можно обновлять view компоненты.

Код будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | private Runnable animator = new Runnable() {      @Override      public void run() {          boolean scheduleNewFrame = false;          long now = AnimationUtils.currentAnimationTimeMillis();            for (Dynamics datapoint: datapoints) {              datapoint.update(now);              if (!datapoint.isAtRest()) {                  scheduleNewFrame = true;              }          }            if (scheduleNewFrame) {              postDelayed(this, 15);          }          invalidate();      }  }; |

Я создал объект типа Runnable и реализовал метод run(), в теле которого выполняется обновление позиций точек. Если хотя бы одна из точек продолжает движение, метод run() будет перезапускаться с небольшой задержкой. Перезапуск выполняется с помощью метода postDelayed(..). В конце метода run() вызывается метод invalidate(), который сообщает системе о необходимости  перерисовки view компонента.

Что случится, если следующее обновление позиций точек выполнится до того как произойдет предыдущая перерисовка экрана? Теоретически это возможно, поскольку мы не можем контролировать этот процесс. По сути, ничего плохого не произойдет, потому что runnable объект "заворачивается" в сообщение и добавляется к очереди объекта Looper, обрабатываемого в UI потоке. То же самое происходит и при вызове метода ivalidate(). Looper обрабатывает сообщения в порядке поступления, поэтому перерисовка компонента будет выполняться корректно.

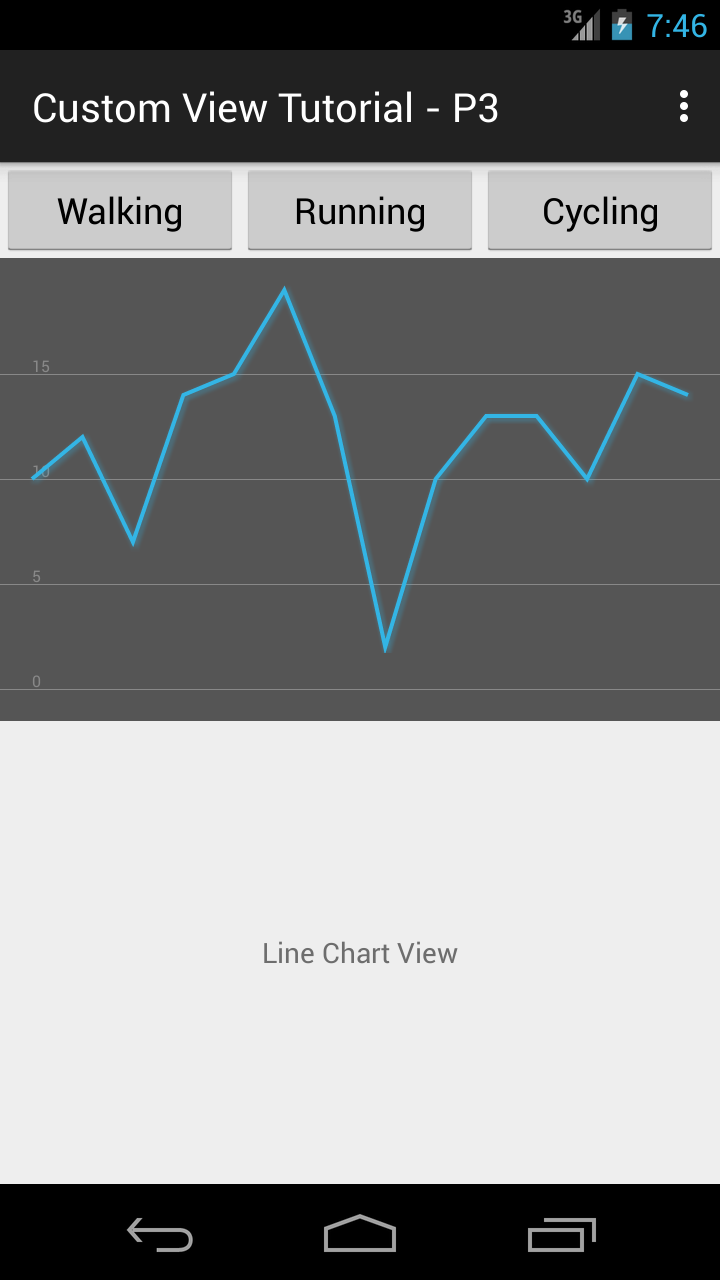
Комбинация dynamiсs и runnable объектов - удобный способ реализации анимации, который легко настраивать и легко расширять. Я регулярно использую этот паттерн с View и ViewGroup объектами. Обычно я реализую базовых функционал view компонента, а после того как все отлажу, добавляю анимацию, используя описанный выше подход.

Для запуска анимации я изменил метод setChartData(). Давайте посмотрим на него.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public void setChartData(float[] newDatapoints) {      if (datapoints == null || datapoints.length != newDatapoints.length) {          datapoints = new Dynamics[newDatapoints.length];          long now = AnimationUtils.currentAnimationTimeMillis();            for (int i = 0; i < newDatapoints.length; i++) {              datapoints[i] = new Dynamics(70f, 0.5f);              datapoints[i].setState(newDatapoints[i], 0, now);              datapoints[i].setTargetPosition(newDatapoints[i]);          }            invalidate();      } else {            for (int i = 0; i < newDatapoints.length; i++) {              datapoints[i].setTargetPosition(newDatapoints[i]);          }            removeCallbacks(animator);          post(animator);      }  } |

Метод обрабатывает две ситуации. Если у нас не было данных или новые данные отличаются по числу точек, мы создаем новый массив объектов типа Dynamics и инициализируем их. Текущую и конечную позиции точки мы устанавливаем равной Y значению, а скорость - 0. После инициализации всех точек, view компонент перерисовывается с помощью метода invalidate(). Анимация в этом случае не выполняется, поскольку изменения количества точек требует более сложного алгоритма.

С другой стороны, если у нас уже есть набор данных, нам нужно только поменять конечные позиции точек и запустить анимацию.  Поскольку анимация уже может быть запущенной, мы сначала удаляем объект animator из очереди сообщений с помощью метода removeCallbacks(). Я бы рекомендовал вам всегда делать это перед запуском новой анимации. Ну а далее мы вызываем метод post(), передав ему Runnable объект, в данном случае это наш animator. Метод invalidate() здесь вызывать не нужно, потому что текущие позиции точек не изменились.

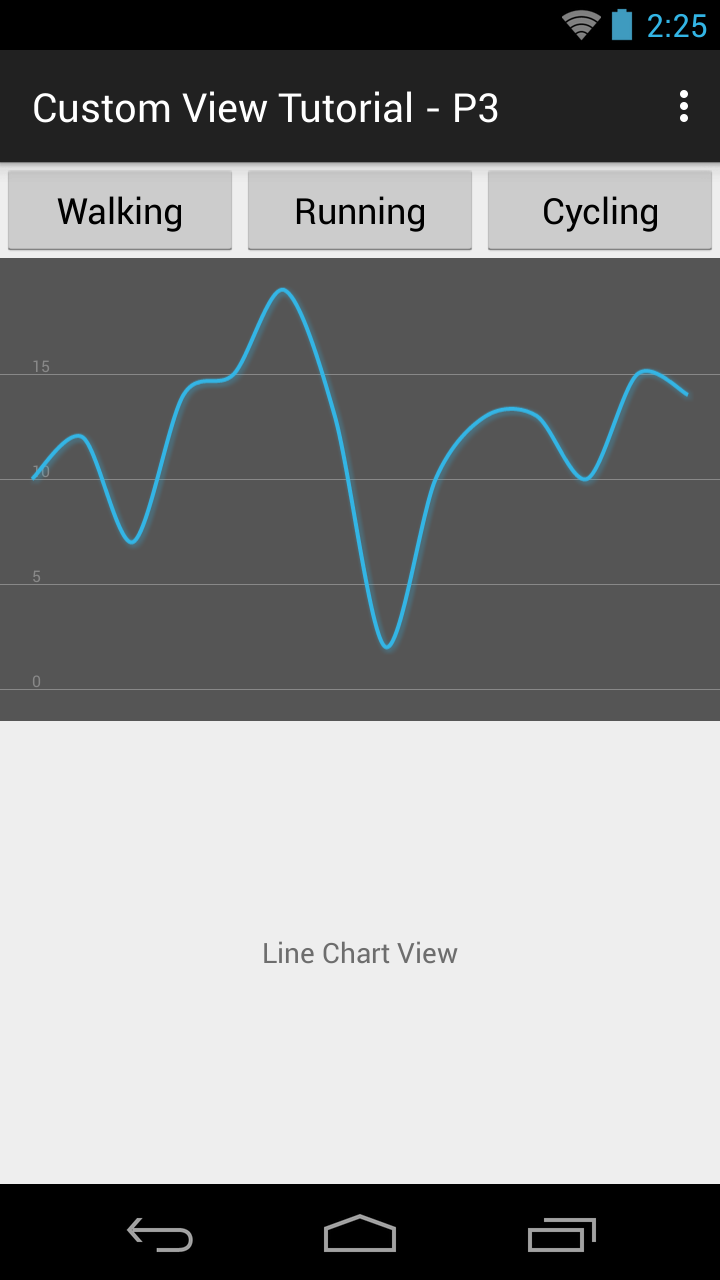


Сглаживание

Последний штрих. Вам не кажется, что график выглядит немного грубовато? Давайте поменяем методику создания объекта Path. Если использовать вместо метода lineTo() метод cubicTo(), можно нарисовать график кривыми Безье, правда это требует вычисления двух дополнительных контрольных точек.

Хороший способ получить эти точки — расположить первую контрольную точку на прямой проходящей через начальную точку i и имеющую наклон как прямая между точками i-1 и i+1. Вторая контрольная точку должна быть на прямой, проходящей через точку i+1 и имеющая такой же наклон как прямая между точками i и i+2. Добавив таким образом контрольных точек, мы отобразим наш график в виде непрерывной кривой.

На рисунке ниже можно видеть конечный результат.



У многих типов анимации, включая стандартную анимацию Android, возникают проблемы, если прерывать ход выполнения анимации. Давайте я объясню это на примере.

Например у вас есть простое приложение, которой содержит рисунок и кнопку. При запуске приложения рисунок скрыт. Нажатие на кнопку делает рисунок видимым, если он скрыт или скрывает его, если он был видимым. Изменение состояния рисунка сопровождается анимацией с использованием AlphaAnimation. Если мы нажимаем на кнопку только когда анимация завершена, все работает хорошо. Но если мы нажмем кнопку, чтобы сделать рисунок видимым, а затем быстро нажмем опять, то состояние рисунка скачкообразно изменится от половинной прозрачности до полной видимости и затем затухнет. Этот тип ошибок анимации можно встретить достаточно часто и выглядит это как мерцание.

Если вы запустите данный пример на эмуляторе, то увидите, что здесь такой проблемы нет. Изменение конечной позиции анимации не меняет текущую позицию или скорость напрямую. Анимация будет менять направление со временем, плавно двигаясь к новой конечной позиции.

Тестовый проект для этой статьи можно скачать с [GitHub](https://github.com/CodeAndLife/Custom-view-3" \o "android custom view с анимацией" \t "_blank).  
По материалам сайта[jayway](http://www.jayway.com/2012/08/29/creating-custom-android-views-part-3-animating-your-custom-views-smoothly/).  
Вольный перевод - Pavel Bobkov