

Рисование

Рисовать в Android можно на любом наследнике класса `View`, переопределив метод `onDraw()`. Можно создать класс-наследник, например, кнопки, и с помощью рисования придать ей нестандартный внешний вид.

Если нужно создать полностью свой графический элемент, то можно создать класс-наследник от исходного класса `View`. При этом требуется реализовать четыре конструктора, которые в разных обстоятельствах могут вызываться как средой разработки, так и операционной системой. К счастью, язык Kotlin позволяет указать, что все необходимые перегрузки конструкторов должны быть добавлены автоматически, для этого используется такая конструкция:

```
class MyView
    @JvmOverloads constructor (context: Context,
                               attrs: AttributeSet? = null,
                               defStyleAttr: Int = 0) :
        View(context, attrs, defStyleAttr) {

    // Тело класса MyView

}
```

Классы Canvas и Paint

Всё, что используется для рисования, делится на две области: класс `Canvas` занимается вопросами *что* рисовать (формы, линии, точки), а класс `Paint` – *как* это следует отрисовать (цвета, стили, шрифты и т. д).

Примеры того, что умеет класс `Canvas`:

- графические примитивы:
 - точки `drawPoint()`
 - линии `drawLine()`
 - прямоугольники `drawRect()` и `drawRoundRect()`
 - круги `drawCircle()` и овалы `drawOval()`
 - сектора `drawArc()`
- текст:
 - обычный `drawText()`
 - по заданной кривой `drawTextOnPath()`
- сложные объекты, составленные из линий, кривых и примитивов: класс `Path`
- градиенты: класс `LinearGradient`
- растровые изображения: метод `drawBitmap()`

В свою очередь класс `Paint` задаёт параметры того как это всё будет отрисовываться:

- свойство `style` определяет будет ли фигура закрашиваться (`FILL`), рисоваться только контур (`STROKE`), или то и другое вместе (`FILL_AND_STROKE`)
- свойство `color` и метод `setARGB()` задают цвет рисования фигуры или текста

- свойства `alpha` задаёт полупрозрачность (альфа-канал)
- свойства `isAntiAlias` включает или выключает сглаживание линий
- свойство `shader` включает или выключает использование шейдера, например, градиентной заливки
- свойство `textAlign` определяет выравнивание текста
- свойство `typeface` определяет шрифт текста
- ...

Метод `onDraw()` в качестве параметра получает объект класса `Canvas`, с помощью которого следует отрисовывать всё необходимое:

```
override fun onDraw(canvas: Canvas) {
    super.onDraw(canvas)
    // Код для рисования...
}
```

Поскольку метод `onDraw()` может вызываться очень часто, рекомендуется заранее загрузить все необходимые изображения, рассчитывать позиции элементов, настроить стили изображений и т. д.

Графические примитивы

Точка

Для рисования точки служит метод `drawPoint()`:

```
val paint = Paint().apply {
    color = Color.BLUE
    strokeWidth = 5f
}

canvas.drawPoint(50f, 75f, paint)
```

Размер точки (как и другие её свойства, например, цвет) определяется в объекте `Paint`, который передаётся в качестве параметра. В данном примере рисуется синяя точка размером в 5 пикселей с координатами (50, 75).

Следует обратить внимание что практически все размеры и координаты при рисовании задаются типом `Float`, поэтому числовые константы должны быть с суффиксом `f`: `50f` вместо `50` и т. д.

Метод `drawPoints()` позволяет нарисовать сразу много точек, их координаты нужно передать в виде массива `FloatArray`: `[x0, y0, x1, y1, ...]`.

Линия

Линии рисуются методом `drawLine()`, который получает четыре координаты (точку начала и точку конца линии) и объект `Paint`. Сразу много линий можно нарисовать методом `drawLines()`, которому нужно передать массив координат `FloatArray`. Каждые четыре числа в этом массиве определяют начало и конец очередной линии: 0-3 – первая линия, 4-7 – вторая линия, 8-11 – третья и т. д.

Прямоугольник

Метод `drawRect()` рисует прямоугольник (или квадрат, если все стороны имеют одинаковый размер). Координаты можно передать несколькими способами:

- четыре числа типа `Float` – координаты `left`, `top`, `right` и `bottom` соответственно
- структура `RectF` – объект, содержащий сразу четыре числа типа `Float`
- структура `Rect` – объект, содержащий четыре числа типа `Int` – это один из немногих случаев, когда допускается использование целочисленных координат

У прямоугольника можно скруглить углы, для этого используется метод `drawRoundRect()`, которому помимо обычных параметров передаются ещё радиусы `rx` и `ry` скругления углов по горизонтальной и вертикальной оси соответственно.

Круг и овал

Круг рисуется с помощью метода `drawCircle()`, которому передаются координаты центра `cx` и `cy`, а также радиус круга `radius`. Овал можно нарисовать с помощью метода `drawOval()`, которому передаётся структура `RectF` или четыре координаты: по сути это прямоугольник с максимально возможно скруглёнными углами.

Сектор овала

Метод `drawArc()` рисует сектор овала:



В качестве параметров метод принимает координаты описывающего прямоугольника, градусы линий начала и конца сектора, а также параметр `useCenter`: если он имеет значение `true`, то линии сектора будут сходиться в центре, а если `false`, то точки сектора будут соединены напрямую, по хорде.

Цвет

Задать цвет заливки или обрамления фигуры можно с помощью класса `Paint`: свойство `color` задаёт цвет, которым будет нарисован объект. Цвет задаётся в виде четырёх компонент:


- красный `r`
- зелёный `g`
- синий `b`
- альфа-канал `a`, задающий степень прозрачности объекта

Каждый из этих компонентов может принимать значения от 0 до 255, а если цвет нужно представить в виде единого числа, то компоненты входят в него со сдвигом влево:

`0xAARRGGBB`

Например, если красный компонент имеет значение 224, зелёный 64, а синий 128, то нужно перевести эти значения в шестнадцатеричный формат, а затем сформировать число

0xFFE04080

В результате получится вот такой цвет: . Особое внимание следует обратить на альфа-канал: если его не указать в числе (то есть оставить число 0xE04080), то альфа-канал будет равен нулю, и цвет будет прозрачным, фигура, фактически, нарисована не будет!

Язык Kotlin запрещает неявные преобразования типов, поэтому цвет с альфа-каналом он будет воспринимать как тип `Long`, в то время как большинство функций для работы с цветом ожидают тип `Int`. В этом случае нужно использовать явное преобразование типа:

```
canvas.drawColor(0xFFE04080.toInt())
```

В Android есть специальный класс `Color`, который содержит константы для некоторых часто используемых цветов (например, `Color.WHITE` или `Color.BLACK`), а также выполнять другие полезные операции, например, формировать числовой цвет из отдельных компонентов:

```
val c1 = Color.rgb(100, 150, 200)
val c2 = Color.argb(50, 100, 150, 200)
```

С помощью методов `drawColor()`, `drawARGB()` и `drawRGB()` можно закрасить одним цветом всё пространство, предоставляемое объектом `canvas`. Разница только в параметрах: `drawColor()` требует цвета в формате целого числа `Int`, у которого все компоненты закодированы побайтно, а два других метода позволяют задать компоненты красного `r`, зелёного `g` и синего `b`, а также, при необходимости, альфа-канал `a`, по отдельности.

Path

Класс `Path` позволяет собрать сложный контур фигуры из нескольких линий или кривых. Для добавления примитивов в контур используются следующие методы:

- `moveTo`, `rMoveTo` – перемещает точку рисования линии
- `lineTo`, `rLineTo` – добавляет линию из предыдущей точки к указанным координатам
- `quadTo`, `rQuadTo`, `cubicTo` и `rCubicTo` – добавляют квадратичную или кубическую кривую Безье
- `arcTo` – добавляет сектор
- `close` – замыкает контур фигуры
- `addRect`, `addOval`, `addCircle`, `addArc` – просто добавляет соответствующие фигуры в путь, не встраивая их в контур

Методы, которые начинаются с префикса `r`, используют относительные координаты.

Например, вызов `lineTo(50, 70)` нарисует линию из текущей точки до точки с координатами (50, 70), а `rLineTo(50, 70)` – из текущей точки до точки, которая отстоит от текущей на 50 и 70 точек по горизонтальной и вертикальной оси соответственно.

После того как фигура подготовлена, её можно в любой момент нарисовать на канве с помощью метода `drawPath()`. Если фигура была нарисована с помощью методов с относительными координатами, то её можно рисовать в разных местах, используя как штамп.

Текст

Для обычного рисования текста служит метод `drawText()`:

```
val paint = Paint().apply {  
    color = Color.rgb(0, 128, 64)  
    typeface = Typeface.create("Roboto", Typeface.BOLD)  
    textSize = 50f  
}
```

```
canvas.drawText("Привет!", 50f, 75f, paint)
```

В результате будет нарисован текст:



Если требуется нарисовать текст не горизонтально, то можно использовать метод `drawTextOnPath()`, он применяется для рисования текста вдоль подготовленного контура `Path`.

Матричные преобразования

Иногда бывает нужно повернуть или масштабировать фигуру, для этого применяются матричные преобразования. Отвечает за них объект `Matrix`, предоставляющий следующие методы:

- `setTranslate` / `postTranslate` – перемещение
- `setRotate` / `postRotate` – поворот
- `setScale` / `postScale` – масштабирование
- `values` – получение или установка значений матрицы (массив типа `FloatArray` из 9 чисел с плавающей точкой)

Пример матричного преобразования:

```
val matrix = Matrix()  
matrix.setTranslate(500f, 0f)  
matrix.postRotate(45f)  
matrix.postScale(2f, 1f)  
path.transform(matrix)
```

В этом примере объект сначала сдвигается, затем поворачивается, а после этого масштабируется отдельно по каждой оси.

Более подробно матричные преобразования изучаются в рамках компьютерной графики.

Растровые изображения

Для загрузки в программу изображения используется класс `BitmapFactory`: он позволяет загружать изображение из файла (метод `decodeFile()`), из массива в памяти (метод

`decodeByteArray()`), из ресурсов (метод `decodeResource()`), или из потока (метод `decodeStream()`). На выходе получается объект класса `Bitmap`:

```
val b = BitmapFactory.decodeResource(resources, R.drawable.logo)
canvas.drawBitmap(b, 100f, 100f, paint)
```

К изображению применимы и матричные преобразования, рассмотренные выше.

Чтобы сохранить нарисованное изображение в файл используется метод `compress`. В качестве параметров он принимает формат файла (PNG, JPEG, WEBP), степень сжатия (0-100), а также выходной поток, в который будет записано результирующее изображение.

```
val fos = FileOutputStream(file)
b.compress(Bitmap.CompressFormat.PNG, 100, fos)
fos.close()
```

Размеры канвы

Рисование – это одна из немногих областей в Android, где необходимо знать точные размеры доступной области. Хотя формально размер канвы можно узнать из размеров родительского элемента `View` (свойства `Width` и `Height`), но во время инициализации и размещения элементов управления они могут меняться, и даже принимать нулевое значение. Поэтому рекомендуется переопределить метод `onSizeChanged`, который вызывается при изменении размеров `View`, и в качестве параметров принимает новые и старые размеры элемента. В этом методе можно инициировать пересчёт позиций всех рисуемых объектов:

```
override fun onSizeChanged(w: Int, h: Int, oldw: Int, oldh: Int) {
    super.onSizeChanged(w, h, oldw, oldh)
    // Использование новых размеров
}
```

Производительное рисование

Простой способ рисования через переопределение метода `onDraw` подходит лишь для простых изображений, которые не сильно нагружают основной поток. Если изображение постоянно меняется и требуется интенсивная перерисовка (например, в играх), более правильным будет создать класс на основе `SurfaceView`: он специально предназначен для постоянной фоновой отрисовки изображения и не загружает основной поток.

В классе нужно переопределить три основные функции, которые объявлены в интерфейсе `SurfaceHolder.Callback`:

- `surfaceCreated` – вызывается при создании поверхности, на которой будет осуществляться рисование, именно тут запускается основной цикл отрисовки
- `surfaceDestroyed` – вызывается при уничтожении поверхности, нужно остановить цикл отрисовки
- `surfaceChanged` – срабатывает при изменении размеров или формата поверхности, здесь можно пересчитать различные параметры

По запросу методом `lockCanvas` Android предоставляет канву для рисования, а при вызове метода `unlockCanvasAndPost()` передаёт готовое изображение для отображения на экране.

Таким образом, максимально упрощённый класс на основе `SurfaceView` будет выглядеть следующим образом:

```
class SkyView
    @JvmOverloads constructor (context: Context,
                               attrs: AttributeSet? = null,
                               defStyleAttr: Int = 0) :
        SurfaceView(context, attrs, defStyleAttr),
        SurfaceHolder.Callback {

    init {
        // Подключение функций обратного вызова
        holder.addCallback(this)
    }

    override fun surfaceCreated(holder: SurfaceHolder) {
        // Запуск фонового потока для отрисовки
        thread {
            // Бесконечный цикл отрисовки
            while (true) {
                // Получение канвы
                val canvas = holder.lockCanvas()

                // Рисование очередного кадра
                // ...

                // Завершение рисования и передача кадра на экран
                holder.unlockCanvasAndPost(canvas)
            }
        }
    }

    override fun surfaceChanged(holder: SurfaceHolder, format: Int, w: Int, h: Int) {
        // Изменились параметры, например, размер канвы
    }

    override fun surfaceDestroyed(holder: SurfaceHolder) {
        // Работа завершена, остановка цикла рисования
    }
}
```

Задание

Придумайте и реализуйте приложение, в котором используется рисование. На экране должно быть несколько движущихся или изменяющих свои визуальные свойства объектов: например, машинки ездят по экрану, солнце перемещается по дуге (и, возможно, сменяется луной), мыльные пузыри летают по экрану и сталкиваются друг с другом, звезды на небе случайно (но плавно) мерцают, ... Удивите преподавателя (в приятном смысле!)