Android 高级绘图

高级画布绘图

我们已经介绍了 Canvas，在那里，已经学习了如何创建自己的 View。在第 7 章中也使用了 Canvas 来为 MapView 标注覆盖。

画布(Canvas)是图形编程中一个很普通的概念，通常由三个基本的绘图组件组成：

Canvas 提供了绘图方法，可以向底层的位图绘制基本图形。
Paint 也称为"刷子"，Paint 可以指定如何将基本图形绘制到位图上。
Bitmap 绘图的表面。

Android 绘图 API 支持透明度、渐变填充、圆边矩形和抗锯齿。遗憾的是，由于资源限制，它还不支持矢量图形，它使用的是传统光栅样式的重新绘图。

这种光栅方法的结果是提高了效率，但是改变一个 Paint 对象不会影响已经画好的基本图形，它只会影响新的元素。

提示：
如果你拥有 Windows 开发背景，那么 Android 的 2D 绘图能力大致相当于 GDI+的能力。

1. 可以画什么？

Canvas 类封装了用作绘图表面的位图；它还提供了 draw*方法来实现设计。

下面的列表提供了对可用的基本图形的简要说明，但并没有深入地探讨每一个 draw 方法的详细内容：

drawARGB / drawRGB / drawColor 使用单一的颜色填充画布。
drawArc 在一个矩形区域的两个角之间绘制一个弧。
drawBitmap 在画布上绘制一个位图。可以通过指定目标大小或者使用一个矩阵来改变目标位图的外观。
drawBitmapMesh 使用一个 mesh(网)来绘制一个位图，它可以通过移动网中的点来操作目标的外观。
drawCircle 以给定的点为圆心，绘制一个指定半径的圆。
drawLine(s) 在两个点之间画一条(多条)直线。
drawOval 以指定的矩形为边界，画一个椭圆。
drawPaint 使用指定的 Paint 填充整个 Canvas
drawPath 绘制指定的 Path。Path 对象经常用来保存一个对象中基本图形的集合。
drawPicture 在指定的矩形中绘制一个 Picture 对象。
drawPosText 绘制指定了每一个字符的偏移量的文本字符串。
drawRect 绘制一个矩形。

drawRoundRect 绘制一个圆角矩形。

drawText 在 Canvas 上绘制一个文本串。文本的字体、大小和渲染属性都设置在用来渲染文本的 Paint 对象中。

drawTextOnPath 在一个指定的 path 上绘制文本。

drawVertices 绘制一系列三角形面片，通过一系列顶点来指定它们。

这些绘图方法中的每一个都需要指定一个 Paint 对象来渲染它。在下面的部分中，将学习如何创建和修改 Paint 对象，从而在绘图中完成大部分工作。

2. 从 Paint 中完成工作

Paint 类相当于一个笔刷和调色板。它可以选择如何使用上面描述的 draw 方法来渲染绘制在画布上的基本图形。通过修改 Paint 对象，可以在绘图的时候控制颜色、样式、字体和特殊效果。最简单地，setColor 可以让你选择一个 Paint 的颜色，而 Paint 对象的样式(使用 setStyle 控制)则可以决定是绘制绘图对象的轮廓(STROKE)，还是只填充每一部分(FILL)，或者是两者都做(STROKE_AND_FILL)

除了这些简单的控制之外，Paint 类还支持透明度，另外，它也可以通过使用各种各样的阴影、过滤器和效果进行修改，从而提供由更丰富的、复杂的画笔和颜料组成的调色板。

Android SDK 包含了一些非常好的实例，它们说明了 Paint 类中可用的大部分功能。你可以在 API demos 的 graphics 子目录中找到它们：

sdk root folder]\samples\ApiDemos\src\com\android\samples\graphics

在下面的部分中，将学习和使用其中的部分功能。这些部分只是简单地罗列了它们能实现的效果(例如渐变和边缘浮雕)，而没有详细地列出所有可能的情况。

使用透明度

Android 中的所有颜色都包含了一个不透明组件(alpha 通道)。

当创建一个颜色的时候，可以使用 argb 或者 parseColor 方法来定义它的 alpha 值，如下所示：

Java 代码：

```
1.
2.  // 使用红色，并让它 50%透明
3.  int opacity = 127;
4.  int intColor = Color.argb(opacity, 255, 0, 0);
5.  int parsedColor = Color.parseColor("#7FFF0000");
```

复制代码

或者，也可以使用 setAlpha 方法来设置已存在的 Paint 对象的透明度：

Java 代码：

```
1.
2.  // 让颜色 50%透明
3.  int opacity = 127;
4.  myPaint.setAlpha(opacity);
```

复制代码

创建一个不是 100%透明的颜色意味着，使用它绘制的任何基本图形都将是部分透明的--也就是说，在它下面绘制的所有基本图形都是部分可见的。

可以在任何使用了颜色的类或者方法中使用透明效果，包括 Paint、Shader 和 Mask Filter。

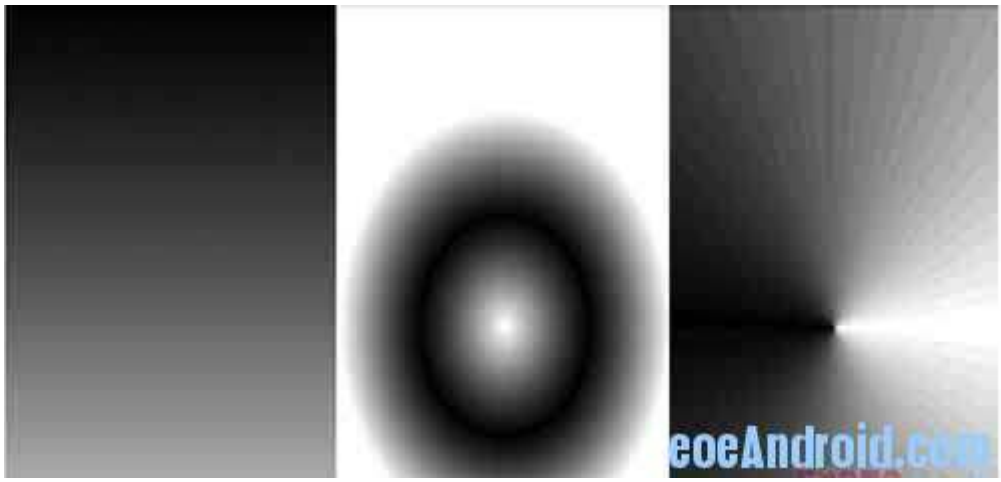
Shader 介绍

Shader 类的派生类可以创建允许使用多种固体颜色填充绘图对象的 Paint。

对 Shader 最常见的使用是定义渐变填充；渐变是在 2D 图像中添加深度和纹理的最佳方式之一。Android 包含了一个 Bitmap Shader 和一个 Compose Shader，同时，还包含了三个渐变的 Shader。

试图用语言来描述绘图的效果本来就是没有意义的，所以看一下图就应该可以知道每一种 Shader 是如何工作的。图中从左到右依次代表的是 LinearGradient、RadialGradient 和 SweepGradient.

效果图：



提示：

没有包含的是 ComposerShader，它可以创建多个 Shader 和 BitmapShader 的组合，

从而可以在一个位图图像的基础上创建一个绘图刷。

要在绘图的时候使用一个Shader，可以使用setShader方法将其应用到一个Paint中，如下面的代码所示：

```
Paint shaderPaint = new Paint();
shaderPaint.setShader(myLinearGradient);
```

你使用这个 Paint 所绘制的任何东西都将使用你指定的 Shader 进行填充，而不是使用 Paint 本身的颜色进行填充。

定义渐变 Shader

如上所示，使用渐变 Shader 可以让你使用交替改变的颜色来填充图片；你可以将颜色渐变定义为两种颜色的简单交替，如下所示：

java 代码：

```
1.
2.  int colorFrom = Color.BLACK;
3.  int colorTo = Color.WHITE;
4.  LinearGradient linearGradientShader = new LinearGradient(x1, y1, x2, y2,
    colorFrom, colorTo, TileMode.CLAMP);
```

复制代码

或者，你还可以定义更复杂的按照设定比例进行分布的颜色序列，如下面的 RadialGradientShader 例子所示：

java 代码：

```
1.
2.  int[] gradientColors = new int[3];
3.  gradientColors[0] = Color.GREEN;
4.  gradientColors[1] = Color.YELLOW;
5.  gradientColors[2] = Color.RED;
6.  float[] gradientPositions = new float[3];
7.  gradientPositions[0] = 0.0f;
8.  gradientPositions[1] = 0.5f;
9.  gradientPositions[2] = 1.0f;
10. RadialGradient radialGradientShader=new RadialGradient(centerX,centerY,
    radius, gradientColors, gradientPositions, TileMode.CLAMP);
```

复制代码

每一种渐变 Shader(线性的、辐射形的和扫描状的)都可以使用以上这两种技术来定义渐变填充。

使用 Shader TileModes

渐变 Shader 的画刷大小既可以显式地使用有边界的矩形来定义，也可以使用中心点和半径长度来定义。Bitmap Shader 可以通过它的位图大小来决定它的画刷大小。

如果 Shader 画刷所定义的区域比要填充的区域小，那么 TileMode 将会决定如何处理剩余的区域：

CLAMP 使用 Shader 的边界颜色来填充剩余的空间。

MIRROR 在水平和垂直方向上拉伸 Shader 图像，这样每一个图像就都能与上一个缝合了。

REPEAT 在水平和垂直方向上重复 Shader 图像，但不拉伸它。

使用 MaskFilter

MaskFilter 类可以为 Paint 分配边缘效果。

对 MaskFilter 的扩展可以对一个 Paint 边缘的 alpha 通道应用转换。Android 包含了下面几种 MaskFilter：

BlurMaskFilter 指定了一个模糊的样式和半径来处理 Paint 的边缘。

EmbossMaskFilter 指定了光源的方向和环境光强度来添加浮雕效果。

要应用一个 MaskFilter，可以使用 setMaskFilter 方法，并传递给它一个 MaskFilter 对象。下面的例子是对一个已经存在的 Paint 应用一个 EmbossMaskFilter：

java 代码：

```
1.
2.  // 设置光源的方向
3.  float[] direction = new float[]{ 1, 1, 1 };
4.  //设置环境光亮度
5.  float light = 0.4f;
6.  // 选择要应用的反射等级
7.  float specular = 6;
8.  // 向 mask 应用一定级别的模糊
9.  float blur = 3.5f;
10. EmbossMaskFilter emboss=new
    EmbossMaskFilter(direction,light,specular,blur);
11. // 应用 mask myPaint.setMaskFilter(emboss);
```

复制代码

SDK 中包含的 FingerPaint API demo 是说明如何使用 MaskFilter 的一个非常好的例子。它展示了这两种 filter 的效果。

使用 ColorFilter

MaskFilter 是对一个 Paint 的 alpha 通道的转换，而 ColorFilter 则是对每一个 RGB 通道应用转换。所有由 ColorFilter 所派生的类在执行它们的转换时，都会忽略 alpha 通道。

Android 包含三个 ColorFilter：

ColorMatrixColorFilter 可以指定一个 4×5 的 ColorMatrix 并将其应用到一个 Paint 中。ColorMatrixes 通常在程序中用于对图像进行处理，而且由于它们支持使用矩阵相乘的方法来执行链接转换，所以它们很有用。

LightingColorFilter 乘以第一个颜色的 RGB 通道，然后加上第二个颜色。每一次转换的结果都限制在 0 到 255 之间。

PorterDuffColorFilter 可以使用数字图像合成的 16 条 Porter-Duff 规则中的任意一条来向 Paint 应用一个指定的颜色。

使用 setColorFilter 方法应用 ColorFilter，如下所示：

```
myPaint.setColorFilter(new LightingColorFilter(Color.BLUE, Color.RED));
```

API 中的 ColorMatrixSample 是说明如何使用 ColorFilter 和 Color Matrix 的非常好的例子。

使用 PathEffect

到目前为止，所有的效应都会影响到 Paint 填充图像的方式；PathEffect 是用来控制绘制轮廓(线条)的方式。

PathEffect 对于绘制 Path 基本图形特别有用，但是它们也可以应用到任何 Paint 中从而影响线条绘制的方式。

使用 PathEffect，可以改变一个形状的边角的外观并且控制轮廓的外表。Android 包含了多个 PathEffect，包括：

CornerPathEffect 可以使用圆角来代替尖锐的角从而对基本图形的形状尖锐的边角进行平滑。

DashPathEffect 可以使用 DashPathEffect 来创建一个虚线的轮廓(短横线/小圆点)，而不是使用实线。你还可以指定任意的虚/实线段的重复模式。

DiscretePathEffect 与 DashPathEffect 相似，但是添加了随机性。当绘制它的时候，需要指定每一段的长度和与原始路径的偏离度。

PathDashPathEffect 这种效果可以定义一个新的形状(路径)并将其用作原始路径的轮廓标记。

下面的效果可以在一个 Paint 中组合使用多个 Path Effect。

SumPathEffect 顺序地在一条路径中添加两种效果，这样每一种效果都可以应用到原始路径中，而且两种结果可以结合起来。

ComposePathEffect 将两种效果组合起来应用，先使用第一种效果，然后在这种效果的基础上应用第二种效果。

对象形状的 PathEffect 的改变会影响到形状的区域。这就能够保证应用到相同形状的填充效果将会绘制到新的边界中。

使用 setPathEffect 方法可以把 PathEffect 应用到 Paint 对象中，如下所示:
borderPaint.setPathEffect(new CornerPathEffect(5));

PathEffect API 示例给出了如何应用每一种效果的指导说明。

修改 Xfermode

可以通过修改 Paint 的 Xfermode 来影响在 Canvas 已有的图像上面绘制新的颜色的方式。

在正常的情况下，在已有的图像上绘图将会在其上面添加一层新的形状。如果新的 Paint 是完全不透明的，那么它将完全遮挡住下面的 Paint；如果它是部分透明的，那么它将会被染上下面的颜色。

下面的 Xfermode 子类可以改变这种行为：

AvoidXfermode 指定了一个颜色和容差，强制 Paint 避免在它上面绘图(或者只在它上面绘图)。

PixelXorXfermode 当覆盖已有的颜色时，应用一个简单的像素 XOR 操作。

PorterDuffXfermode 这是一个非常强大的转换模式，使用它，可以使用图像合成的 16 条 Porter-Duff 规则的任意一条来控制 Paint 如何与已有的 Canvas 图像进行交互。

要应用转换模式，可以使用 setXferMode 方法，如下所示：

java 代码：

```
1.  AvoidXfermode avoid = new AvoidXfermode(Color.BLUE, 10, AvoidXfermode.Mode.
    AVOID); borderPen.setXfermode(avoid);
```

复制代码

使用抗锯齿效果提高 Paint 质量

在绘制一个新的 Paint 对象时，可以通过传递给它一些标记来影响它被渲染的方式。

ANTI_ALIAS_FLAG 是其中一种很有趣的标记，它可以保证在绘制斜线的时候使用抗锯齿效果来平滑该斜线的外观。

在绘制文本的时候，抗锯齿效果尤为重要，因为经过抗锯齿效果处理之后的文本非常容易阅读。要创建更加平滑的文本效果，可以应用 SUBPIXEL_TEXT_FLAG，它将会应用子像素抗锯齿效果。

也可以手工地使用 setSubpixelText 和 setAntiAlias 方法来设置这些标记，如下所示：

java 代码：

```
1.

2.  myPaint.setSubpixelText(true);

3.  myPaint.setAntiAlias(true);
```

复制代码

2D 图形的硬件加速

在当前这个到处都是 2D 图形爱好者的时代，Android 允许你使用硬件加速来渲染你的应用程序。

如果设备可以使用硬件加速，那么通过设置这个标记可以让活动中的每一个 View 都能使用硬件渲染。尽管减少了系统处理程序的负载，但在极大地提高了图像处理速度的同时，硬件加速也带来了相应的负面效果。

使用 requestWindowFeature 方法，可以在你的活动中应用 Window.FEATURE_OPENGL 标记来打开硬件加速，如下所示：

myActivity.requestWindowFeature(Window.FEATURE_OPENGL);

Canvas 绘图最佳实践经验

2D 自绘操作是非常耗费处理程序资源的；低效的绘图方法会阻塞 GUI 线程，并且会对应用程序的响应造成不利的影响。对于那些只有一个处理程序的资源受限的环境来说，这一点就更加现实了。

这里需要注意 onDraw 方法的资源消耗以及 CPU 周期的耗费，这样才能保证不会把一个看起来很吸引人的应用程序变得完全没有响应。

目前有很多技术可以帮助将与自绘控件相关的资源消耗最小化。我们关心的不是一般的原则，而是某些 Android 特定的注意事项，从而保证你可以创建外观时尚、而且能够保持交互的活动(注意，以下这个列表并不完整)：

考虑硬件加速 OpenGL 硬件加速对 2D 图形的支持是非常好的，所以你总是应该考虑它是否适合你的活动。另一种比较优秀的方法是只用一个单独的 View 和迅速的、耗时的更新来组成活动。一定要保证你使用的基本图形能够被硬件支持。

考虑大小和方向 当在设计 View 和布局的时候，一定要保证考虑(和测试)它们在不同的分辨率和大小下的外观。

只创建一次静态对象 在 Android 中对象的创建是相当昂贵的。因此，在可能的地方，应用只创建一次像 Paint 对象、Path 和 Shader 这样的绘图对象，而不是在 View 每次无效的时候都重新创建它们。

记住 onDraw 是很消耗资源的 执行 onDraw 方法是很消耗资源的处理，它会强制 Android 执行多个图片组合和位图构建操作。下面有几点建议可以让你修改 Canvas 的外观，而不用重新绘制它：

使用 Canvas 转换 可以使用像 rotate 和 translate 这样的转换，来简化 Canvas 中元素复杂的相关位置。例如，相比放置和旋转一个表盘周围的每一个文本元素，你可以简单地将 canvas 旋转 22.5?，然后在相同的位置绘制文本。

使用动画 可以考虑使用动画来执行 View 的预设置的转换，而不是手动地重新绘制它。在活动的 View 中可以执行缩放、旋转和转换动画，并可以提供一种能够有效利用资源的方式来提供缩放、旋转或者抖动效果。

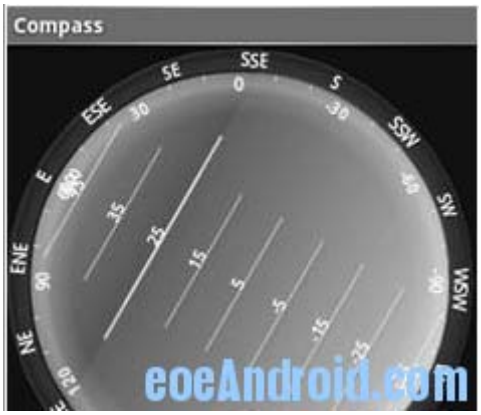
考虑使用位图和 9 Patch 如果 View 使用了静态背景，那么你应该考虑使用一个图片，如位图或者 9 patch，而不是手动地重新绘制。

高级指南针表盘的例子

已经创建了一个简单的指南针。而在上一章，你又回到了这个例子，对它进行了扩展从而使它够使用加速计硬件来显示横向和纵向方向。

那些例子中的 UI 都很简单，从而保证了那些章节中的代码都尽可能地清晰。

在下面的例子中，将对 CompassView 的 onDraw 方法做一些重要的改动，从而把它从一个简单的、平面的指南针，变成一个动态的航空地平仪(artificial horizon)，如图所示。



由于上面的图片是黑白的，所以需要实际动手创建这个控件来看到完全的效果。

(1) 首先，通过修改 colors.xml 资源文件来包含边界、表盘阴影以及天空和地面的颜色值。同时还要更新边界和盘面标记所使用的颜色。

java 代码：

```
1.
2.  <?xml version="1.0" encoding="utf-8"?>
3.  <resources>
4.  <color name="text_color">#FFFF</color>
5.  <color name="background_color">#F000</color>
6.  <color name="marker_color">#FFFF</color>
7.  <color name="shadow_color">#7AAA</color>
8.  <color name="outer_border">#FF444444</color>
9.  <color name="inner_border_one">#FF323232</color>
10. <color name="inner_border_two">#FF414141</color>
11. <color name="inner_border">#FFFFFFFF</color>
12. <color name="horizon_sky_from">#FFA52A2A</color>
13. <color name="horizon_sky_to">#FFFFC125</color>
14. <color name="horizon_ground_from">#FF5F9EA0</color>
15. <color name="horizon_ground_to">#FF00008B</color>
16. </resources>
```

复制代码

(2) 用作航空地平仪的天空和地面的 Paint 和 Shader 对象是根据当前 View 的大小创建的，所以它们不能像你在创建的 Paint 对象那样，是静态的。因此，不再创建 Paint 对象，取而代之的是构造它们所使用的渐变数组和颜色。

java 代码：

```
1.
2.  int[] borderGradientColors;
3.  float[] borderGradientPositions;
4.  int[] glassGradientColors;
5.  float[] glassGradientPositions;
6.  int skyHorizonColorFrom;
```

```
7.  int skyHorizonColorTo;
8.  int groundHorizonColorFrom;
9.  int groundHorizonColorTo;
```
复制代码

(3) 更新 CompassView 的 initCompassView 方法，来使用第(1)步中所创建的资源来初始化第(2)步中所创建的变量。现存的方法代码大部分可以保留，而只需要对 textPaint、circlePaint 和 markerPaint 变量做些许改动，如下所示：

java 代码：

```
1.
2.  protected void initCompassView() {
3.  setFocusable(true);
4.  // 获得外部资源
5.  Resources r = this.getResources();
6.  circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
7.  circlePaint.setColor(R.color.background_color);
8.  circlePaint.setStrokeWidth(1);
9.  circlePaint.setStyle(Paint.Style.STROKE);
10. northString = r.getString(R.string.cardinal_north);
11. eastString = r.getString(R.string.cardinal_east);
12. southString = r.getString(R.string.cardinal_south);
13. westString = r.getString(R.string.cardinal_west);
14. textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
15. textPaint.setColor(r.getColor(R.color.text_color));
16. textPaint.setFakeBoldText(true);
17. textPaint.setSubpixelText(true);
18. textPaint.setTextAlign(Align.LEFT);
19. textHeight = (int)textPaint.measureText("yY");
20. markerPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
21. markerPaint.setColor(r.getColor(R.color.marker_color));
22. markerPaint.setAlpha(200);
23. markerPaint.setStrokeWidth(1);
24. markerPaint.setStyle(Paint.Style.STROKE);
```

```
25. markerPaint.setShadowLayer(2, 1, 1, r.getColor(R.color.shadow_color));
```
复制代码

a. 创建径向 Shader 用来绘制外边界所使用的颜色和位置数组。

java 代码：

```
1.
2.  borderGradientColors = new int[4];
3.  borderGradientPositions = new float[4];
4.  borderGradientColors[3] = r.getColor(R.color.outer_border);
5.  borderGradientColors[2] = r.getColor(R.color.inner_border_one);
6.  borderGradientColors[1] = r.getColor(R.color.inner_border_two);
7.  borderGradientColors[0] = r.getColor(R.color.inner_border);
8.  borderGradientPositions[3] = 0.0f;
9.  borderGradientPositions[2] = 1-0.03f;
10. borderGradientPositions[1] = 1-0.06f;
```
复制代码

b. 现在创建径向渐变的颜色和位置数组，它们将用来创建半透明的"glass dome"(玻璃圆顶)，它放置在 View 的上面，从而使人产生深度的幻觉。

java 代码：

```
1.
2.  glassGradientColors = new int[5];
3.  glassGradientPositions = new float[5];
4.  int glassColor = 245;
5.  glassGradientColors[4]=Color.argb(65,glassColor,glassColor,
    glassColor);
6.  glassGradientColors[3]=Color.argb(100,glassColor,glassColor,glassColor)
    ;
7.  glassGradientColors[2]=Color.argb(50,glassColor,glassColor,
    glassColor);
8.  glassGradientColors[1]=Color.argb(0,glassColor,glassColor, glassColor);
9.  glassGradientColors[0]=Color.argb(0,glassColor,glassColor, glassColor);
10. glassGradientPositions[4] = 1-0.0f;
```

```
11. glassGradientPositions[3] = 1-0.06f;
12. glassGradientPositions[2] = 1-0.10f;
13. glassGradientPositions[1] = 1-0.20f;
14. glassGradientPositions[0] = 1-1.0f;
```
复制代码

c. 最后，获得创建线性颜色渐变所使用的颜色，它们将用来表示航空地平仪中的天空和地面。

java 代码：

```
1.
2.  skyHorizonColorFrom = r.getColor(R.color.horizon_sky_from);
3.  skyHorizonColorTo = r.getColor(R.color.horizon_sky_to);
4.  groundHorizonColorFrom = r.getColor(R.color.horizon_ground_from);
5.  groundHorizonColorTo = r.getColor(R.color.horizon_ground_to);
6.  }
```
复制代码

(4) 在开始绘制表盘之前，先创建一个新的 enum 来存储基本的方向。

java 代码：

```
1.  private enum CompassDirection { N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW,
    SW, WSW, W, WNW, NW, NNW }
```
复制代码

现在需要完全代替已有的 onDraw 方法。首先，需要计算出某些与大小有关的值，包括 View 的中心、圆形控件的半径、包围外盘元素(方向)和内盘元素(倾斜和转动)的矩形。

java 代码：

```
1.
2.  @Override
3.  protected void onDraw(Canvas canvas) {
```
复制代码

(1) 根据用来绘制方向值的字体的大小，计算出外圆(方向)的宽度。
float ringWidth = textHeight + 4;

(2) 然后计算出 View 的高度和宽度，并使用那些值来计算外刻度盘和内刻度盘的半径，同时创建每一个盘面的边界 box。

java 代码：

```
1.
2.  int height = getMeasuredHeight();
3.  int width =getMeasuredWidth();
4.  int px = width/2;
5.  int py = height/2;
6.  Point center = new Point(px, py);
7.  int radius = Math.min(px, py)-2;
8.  RectF boundingBox = new RectF(center.x - radius, center.y - radius, center.x
    + radius, center.y + radius);
9.  RectF innerBoundingBox = new RectF(center.x - radius + ringWidth, center.y
    - radius + ringWidth, center.x + radius - ringWidth, center.y + radius -
    ringWidth);
10. float innerRadius = innerBoundingBox.height()/2;
```

复制代码

(3) 在确定了 View 的外形尺寸之后，现在就要开始绘制盘面了。

从外围的底层开始，向里面和上面发展，首先是外盘面(方向)。使用之前的那个例子中所定义的颜色和位置，创建一个新的 RadialGradient Shader，把那个 Shader 分配给一个新的 Paint，然后使用它画一个圆。

java 代码：

```
1.
2.  RadialGradient borderGradient = new RadialGradient(px, py, radius,
    borderGradientColors, borderGradientPositions, TileMode.CLAMP);
3.  Paint pgb = new Paint();
4.  pgb.setShader(borderGradient);
5.  Path outerRingPath = new Path();
6.  outerRingPath.addOval(boundingBox, Direction.CW);
7.  canvas.drawPath(outerRingPath, pgb);
```

复制代码

(4) 接下来，需要绘制航空地平仪。通过把圆形表面分成两个部分来创建地平仪，其中一部分代表天空，另一部分代表地面。每一部分所占的比例与当前的高度(pitch)有关。

首先创建用来绘制天空和地面的 Shader 和 Paint 对象。

java 代码：

```
1.
2.  LinearGradient skyShader = new LinearGradient(center.x,
    innerBoundingBox.top, center.x, innerBoundingBox.bottom,
    skyHorizonColorFrom, skyHorizonColorTo, TileMode.CLAMP);
3.  Paint skyPaint = new Paint();
4.  skyPaint.setShader(skyShader);
5.  LinearGradient groundShader = new LinearGradient(center.x,
    innerBoundingBox.top, center.x, innerBoundingBox.bottom,
    groundHorizonColorFrom, groundHorizonColorTo, TileMode.CLAMP); Paint
    groundPaint = new Paint();
6.  groundPaint.setShader(groundShader);
```

复制代码

(5) 现在通过形式化俯仰角和翻转角的值来让它们分别处于±90度和±180度范围内。

java 代码：

```
1.
2.  float tiltDegree = pitch;
3.  while (tiltDegree > 90 || tiltDegree < -90) {
4.  if (tiltDegree > 90) tiltDegree = -90 + (tiltDegree - 90);
5.  if (tiltDegree < -90) tiltDegree = 90 - (tiltDegree + 90);
6.  }
7.  float rollrollDegree = roll;
8.  while (rollDegree > 180 || rollDegree < -180) {
9.  if (rollDegree > 180) rollDegree = -180 + (rollDegree - 180);
10. if (rollDegree < -180) rollDegree = 180 - (rollDegree + 180);
11. }
```

复制代码

(6) 创建用来填充圆的每个部分(地面和天空)的路径。每一部分的比例应该与形式化之后的俯仰值有关。

java 代码：

```
1.
2.  Path skyPath = new Path();
3.  skyPath.addArc(innerBoundingBox, -tiltDegree, (180 + (2 * tiltDegree)));
```
复制代码

(7) 将 Canvas 围绕圆心，按照与当前翻转角相反的方向进行旋转，并且使用在第(4)步中所创建的 Paint 来绘制天空和地面路径。

java 代码：

```
1.
2.  canvas.rotate(-rollDegree, px, py);
3.  canvas.drawOval(innerBoundingBox, groundPaint);
4.  canvas.drawPath(skyPath, skyPaint);
5.  canvas.drawPath(skyPath, markerPaint);
```
复制代码

(8) 接下来是盘面标记，首先计算水平的水平标记的起止点。

java 代码：

```
1.  int markWidth = radius / 3; int startX = center.x - markWidth; int endX =
    center.x + markWidth;
```
复制代码

(9) 要让水平值更易于读取，应该保证俯仰角刻度总是从当前值开始。下面的代码计算了天空和地面的接口在水平面上的位置：

java 代码：

```
1.  double h = innerRadius*Math.cos(Math.toRadians(90-tiltDegree)); double
    justTiltY = center.y - h;
```
复制代码

(10) 找到表示每一个倾斜角的像素的数目。

java 代码：

```
1.  float pxPerDegree = (innerBoundingBox.height()/2)/45f;
```
复制代码

(11) 现在遍历 180 度，以当前的倾斜值为中心，给出一个可能的俯仰角的滑动刻度。

java 代码：

```
1.
2.  for (int i = 90; i >= -90; i -= 10) {
3.  double ypos = justTiltY + i*pxPerDegree;
4.  // 只显示内表盘的刻度
5.  if ((ypos < (innerBoundingBox.top + textHeight)) || (ypos >
    innerBoundingBox.bottom - textHeight)) continue;
6.  // 为每一个刻度增加画一个直线和一个倾斜角
7.  canvas.drawLine(startX, (float)ypos, endX, (float)ypos, markerPaint);
8.  t displayPos = (int)(tiltDegree - i);
9.  String displayString = String.valueOf(displayPos);
10. float stringSizeWidth = textPaint.measureText(displayString);
11. canvas.drawText(displayString, (int)(center.x-stringSizeWidth/2),
    (int)(ypos)+1, textPaint);
12. }
```

复制代码

(12) 现在，在大地/天空接口处绘制一条更粗的线。在画线之前，改变 markerPaint 对象的线条粗度(然后把它设置回以前的值)。

java 代码：

```
1.
2.  markerPaint.setStrokeWidth(2);
3.  canvas.drawLine(center.x - radius / 2, (float)justTiltY, center.x + radius
    / 2, (float)justTiltY, markerPaint);
4.  markerPaint.setStrokeWidth(1);
```

复制代码

(13) 要让用户能够更容易地读取精确的翻转值，应该画一个箭头，并显示一个文本字符串来表示精确值。

创建一个新的 Path，并使用 moveTo/lineTo 方法构建一个开放的箭头，它指向直线的前方。然后绘制路径和一个文本字符串来展示当前的翻转。

java 代码：

```
1.
```

```
2.  // 绘制箭头
3.  Path rollArrow = new Path();
4.  rollArrow.moveTo(center.x - 3, (int)innerBoundingBox.top + 14);
5.  rollArrow.lineTo(center.x, (int)innerBoundingBox.top + 10);
6.  rollArrow.moveTo(center.x + 3, innerBoundingBox.top + 14);
7.  rollArrow.lineTo(center.x, innerBoundingBox.top + 10);
8.  canvas.drawPath(rollArrow, markerPaint);
9.  // 绘制字符串
10. String rollText = String.valueOf(rollDegree);
11. double rollTextWidth = textPaint.measureText(rollText);
12. canvas.drawText(rollText, (float)(center.x - rollTextWidth / 2),
    innerBoundingBox.top + textHeight + 2, textPaint);
```
复制代码

(14) 将 Canvas 旋转到正上方，这样就可以绘制其他的盘面标记了。

java 代码：

```
1.  canvas.restore();
```
复制代码

(15) 每次将 Canvas 旋转 10 度，然后画一个标记或者一个值，直到画完翻转值表盘为止。当完成表盘之后，把 Canvas 恢复为正上方的方向。

java 代码：

```
1.
2.  canvas.save();
3.  canvas.rotate(180, center.x, center.y);
4.  for (int i = -180; i < 180; i += 10) {
5.  // 每 30 度显示一个数字值
6.  if (i % 30 == 0) {
7.  String rollString = String.valueOf(i*-1);
8.  float rollStringWidth = textPaint.measureText(rollString);
9.  PointF rollStringCenter = new PointF(center.x-rollStringWidth / 2,
    innerBoundingBox.top+1+textHeight);
```

```
10. canvas.drawText(rollString, rollStringCenter.x, rollStringCenter.y,
    textPaint);
11. }
12. // 否则，绘制一个标记直线
13. else { canvas.drawLine(center.x, (int)innerBoundingBox.top, center.x,
    (int)innerBoundingBox.top + 5, markerPaint);
14. }
15. canvas.rotate(10, center.x, center.y);
16. }
17. canvas.restore();
```

复制代码

创建表盘的最后一步是在外边界绘制方向标记。

java 代码：

```
1.
2.  canvas.save();
3.  canvas.rotate(-1*(bearing), px, py);
4.  double increment = 22.5;
5.  for (double i = 0; i < 360; i += increment) {
6.  CompassDirection cd = CompassDirection.values()[(int)(i / 22.5)];
7.  String headString = cd.toString();
8.  float headStringWidth = textPaint.measureText(headString);
9.  PointF headStringCenter = new PointF(center.x - headStringWidth / 2,
    boundingBox.top + 1 + textHeight);
10. if (i % increment == 0) canvas.drawText(headString, headStringCenter.x,
    headStringCenter.y, textPaint);
11. else
12. canvas.drawLine(center.x, (int)boundingBox.top, center.x,
    (int)boundingBox.top + 3, markerPaint);
13. canvas.rotate((int)increment, center.x, center.y);
14. }
15. canvas.restore();
```

复制代码

在完成了表盘之后，还需要添加某些收尾的工作。

首先在最上面添加一个"玻璃顶"来给人一种表盘的感觉。使用你前面创建的 radial 渐变数组，创建一个新的 Shader 和 Paint 对象。使用它们在内盘面画一个圆，这样可以更像是覆盖在玻璃下。

java 代码：

```
1.
2.  RadialGradient glassShader = new RadialGradient(px, py, (int)innerRadius,
    glassGradientColors, glassGradientPositions, TileMode.CLAMP);
3.  Paint glassPaint = new Paint();
4.  glassPaint.setShader(glassShader);
5.  canvas.drawOval(innerBoundingBox, glassPaint);
```

复制代码

剩下的所有工作就是再画两个圆分别作为内表盘和外表盘范围的规则边界。然后把 Canvas 恢复为正上方向，并完成 onDraw 方法。

java 代码：

```
1.
2.  //绘制外环
3.  canvas.drawOval(boundingBox, circlePaint);
4.  // 绘制内环
5.  circlePaint.setStrokeWidth(2);
6.  canvas.drawOval(innerBoundingBox, circlePaint);
7.  canvas.restore();
8.  }
```

复制代码

7．增强地图覆盖

在以前我们已经学习了如何使用覆盖在 MapView 上添加一个注释层，用来注释 MapView 覆盖的 Canvas 和用来绘制新的 View 控件的 Canvas 是相同的类，所以，到目前为止，本章描述的所有高级功能都可以用来增强地图覆盖。

也就是说，可以使用任何 draw 方法、透明度、Shader、Color Mask 和 Filter 效果并使用 Android 图形框架来创建丰富的覆盖。

与覆盖交互

MapView 中的触摸屏交互是由 MapView 的每一个覆盖单独进行处理的。要处理覆盖

上的地图单击事件，需要重写 onTap 方法。

下面的代码段展示了一个 onTap 实现，它可以接收单击的地图坐标，以及发生单击的 MapView：

java 代码：

```
1.
2.  @Override public boolean onTap(GeoPoint point, MapView map) {
3.  // 获得和屏幕坐标相互转换的投影
4.  Projection projection = map.getProjection();
5.  // 如果我们处理了这个 onTap()，就返回 true return [ … hit test passed … ];
6.  }
```

复制代码

MapView 可以用来获取地图被单击时的投影。把它和 GeoPoint 参数一块使用，可以确定与屏幕位置相对应的真实世界的经纬度。

一个 Overlay 派生类的 onTap 方法应该在处理了单击之后返回 true(否则返回 false)。如果分配给一个 MapView 的所有 Overlay 都没有返回 true，那么单击事件将会由 MapView 本身处理，否则，活动会取消这个事件。