第 32 章

32 定制视图与触摸事件

本章，通过开发一个名为BoxDrawingView的定制View子类，我们将学习如何处理触摸事件。响应用户的触摸与拖动，定制View将在屏幕上绘制出矩形框，如图32-1所示。

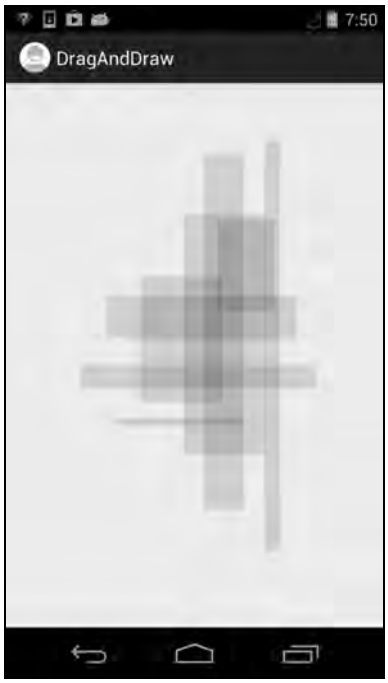

图32-1　各种形状大小的绘制框

32.1　创建 DragAndDraw 项目

BoxDrawingView类是DragAndDraw新项目的关键类。选择New → Android Application Project菜单项，弹出新建应用对话框。参照图32-2进行项目配置。然后创建一个名为DragAndDraw Activity的空白activity。

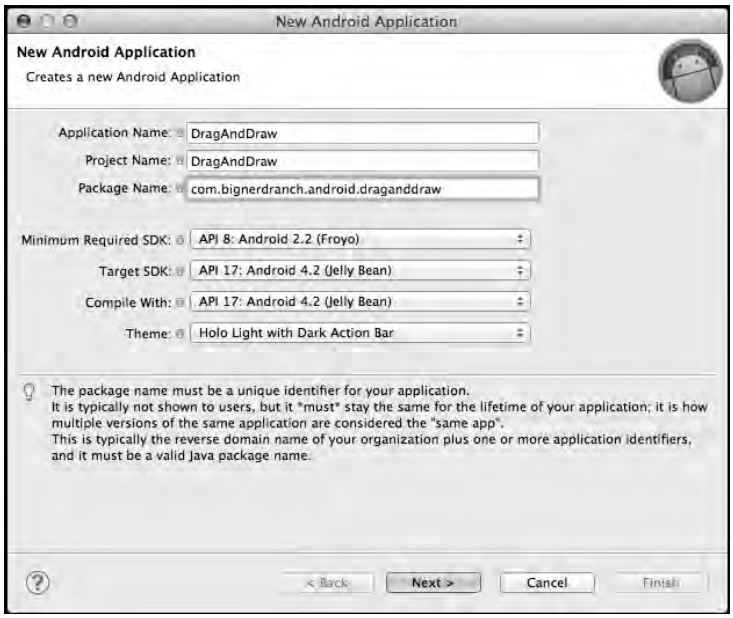

图32-2　创建DragAndDraw项目

32.1.1　创建DragAndDrawActivity

DragAndDrawActivity将设计为SingleFragmentActivity的子类，SingleFragmentActivity可实例化仅包含单个fragment的布局。在包浏览器中，将前面项目的SingleFragmentActivity.java复制到com.bignerdranch.android.draganddraw包目录中，然后再将activity_fragment.xml复制到DragAndDraw项目的res/layout目录中。

在DragAndDrawActivity.java中，调整代码改为继承SingleFragmentActivity类，并实现父类的createFragment()方法以创建返回DragAndDrawFragment对象（稍后将会创建该类），如代码清单32-1所示。

代码清单32-1　修改activity（DragAndDrawActivity.java）

```
public class DragAndDrawActivity extends Activity SingleFragmentActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_drag_and_draw);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_drag_and_draw, menu);
        return true;
    }
```

```
    @Override
    public Fragment createFragment() {
        return new DragAndDrawFragment();
    }
}
```

32.1.2 创建DragAndDrawFragment

为准备DragAndDrawFragment的布局，重命名activity_drag_and_draw.xml布局文件为fragment_drag_and_draw.xml。

DragAndDrawFragment的布局最终是由BoxDrawingView定制视图组成，稍后我们会完成该定制视图的创建。所有的图形绘制和触摸事件处理都将在BoxDrawingView类中实现。

以android.support.v4.app.Fragment为超类，创建名为DragAndDrawFragment的新类。然后覆盖onCreateView(...)方法，并在其中实例化fragment_drag_and_draw.xml布局。

代码清单32-2 创建DragAndDrawFragment（DragAndDrawFragment.java）

```
public class DragAndDrawFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_drag_and_draw, parent, false);
        return v;
    }
}
```

运行DragAndDraw应用，确认应用已正确创建，如图32-3所示。

图32-3 具有默认布局的DragAndDraw应用

32.2　创建定制视图

Android提供有众多优秀的标准视图与组件，但有时我们仍需创建定制视图，以获得专属独特的应用视觉效果。

尽管有着各式各样的定制视图，但仍可硬性将它们分为两大类别。

- 简单视图。简单视图可以有复杂的内部；之所以归为简单类别，是因为简单视图不包括子视图。而且，简单视图几乎总是会执行定制绘制。
- 聚合视图。聚合视图由一些其他视图对象组成。聚合视图通常管理着子视图，但不负责执行定制绘制。相反，图形绘制任务都委托给了各子视图。

以下为创建定制视图所需的三大步骤。

- 选择超类。对于简单定制视图而言，View是一个空白画布，因此是最常见的选择。而对于聚合定制视图，我们应选择合适的布局类
- 继承选定的超类，并至少覆盖一个超类构造方法。或者创建自己的构造方法，并在其中调用超类的构造方法。
- 覆盖其他关键方法，以定制视图行为。

创建BoxDrawingView视图

BoxDrawingView是一个简单视图，同时也是View的直接子类。

以View为超类，新建BoxDrawingView类。在BoxDrawingView.java中，添加两个构造方法。如代码清单32-3所示。

代码清单32-3　初始的BoxDrawingView视图类（BoxDrawingView.java）

```
public class BoxDrawingView extends View {
    // Used when creating the view in code
    public BoxDrawingView(Context context) {
        this(context, null);
    }

    // Used when inflating the view from XML
    public BoxDrawingView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

}
```

这里之所以添加了两个构造方法，是因为视图可从代码或者布局文件实例化。从布局文件中实例化的视图可收到一个AttributeSet实例，该实例包含了XML布局文件中指定的XML属性。即使不打算使用构造方法，按习惯做法，我们也应添加它们。

有了定制视图类，我们来更新fragment_drag_and_draw.xml布局文件以使用它，如代码清单32-4所示。

代码清单32-4 在布局中添加BoxDrawingView（fragment_drag_and_draw.xml）

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
 >

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_centerVertical="true"
    android:text="@string/hello_world" />

</RelativeLayout>

<com.bignerdranch.android.draganddraw.BoxDrawingView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    />
```

注意，我们必须使用BoxDrawingView的全路径类名，这样布局inflater才能够找到它。布局inflater解析布局XML文件，并按视图定义创建View实例。如果元素名不是全路径类名，布局inflater会转而在android.view和android.widget包中寻找目标。如果目标视图类放置在其他包中，布局inflater将无法找到目标并最终导致应用崩溃。因此，对于android.view和android.widget包以外的定制视图类，必须指定它们的全路径类名。

运行DragAndDraw应用，一切正常的话，屏幕上会出现一个空视图，如图32-4所示。

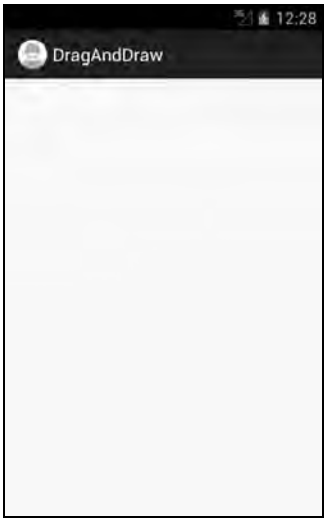

图32-4 未绘制的BoxDrawingView

接下来是让BoxDrawingView监听触摸事件，并实现在屏幕上绘制矩形框。

32.3　处理触摸事件

监听触摸事件的一种方式是使用以下View方法，设置一个触摸事件监听器:

```
public void setOnTouchListener(View.OnTouchListener l)
```

该方法的工作方式与setOnClickListener(View.OnClickListener)相同。我们实现View.OnTouchListener接口，供触摸事件发生时调用。

然而，我们的定制视图是View的子类，因此可走捷径直接覆盖以下View方法：

```
public boolean onTouchEvent(MotionEvent event)
```

该方法可以接收一个MotionEvent类实例，而MotionEvent类可用来描述包括位置和动作的触摸事件。动作则用来描述事件所处的阶段。

动作常量	动作描述
ACTION_DOWN	用户手指触摸到屏幕
ACTION_MOVE	用户在屏幕上移动手指
ACTION_UP	用户手指离开屏幕
ACTION_CANCEL	父视图拦截了触摸事件

在onTouchEvent(...)实现方法中，我们可使用以下MotionEvent方法，查看动作值：

```
public final int getAction()
```

在BoxDrawingView.java中，添加一个日志tag，然后实现onTouchEvent(...)方法记录可能发生的四个不同动作，如代码清单32-5所示。

代码清单32-5　实现BoxDrawingView视图类（BoxDrawingView.java）

```
public class BoxDrawingView extends View {
    public static final String TAG = "BoxDrawingView";

    ...

    public boolean onTouchEvent(MotionEvent event) {
        PointF curr = new PointF(event.getX(), event.getY());

        Log.i(TAG, "Received event at x=" + curr.x +
                ", y=" + curr.y + ":");
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                Log.i(TAG, "  ACTION_DOWN");
                break;
            case MotionEvent.ACTION_MOVE:
                Log.i(TAG, "  ACTION_MOVE");
                break;
            case MotionEvent.ACTION_UP:
                Log.i(TAG, "  ACTION_UP");
                break;
            case MotionEvent.ACTION_CANCEL:
                Log.i(TAG, "  ACTION_CANCEL");
```

```
                break;
            }

            return true;
        }
    }
```

注意，X和Y坐标已经封装到PointF对象中。本章的后面，我们需要同时传递二者的值。而Android提供的PointF容器类刚好满足了这一需求。

运行DragAndDraw应用并打开LogCat视图窗口。触摸屏幕并移动手指，查看BoxDrawingView接收的触摸动作的X和Y坐标记录。

跟踪运动事件

不只是记录坐标，BoxDrawingView主要用于在屏幕上绘制矩形框。要实现这一目标，有几个问题需要解决。

首先，要定义一个矩形框，需知道：

- 原始坐标点（手指的初始位置）；
- 当前坐标点（手指的当前位置）。

其次，定义一个矩形框，还需追踪记录来自多个MotionEvent的数据。这些数据将会保存在Box对象中。

新建一个Box类，用于表示一个矩形框的定义数据，如代码清单32-6所示。

代码清单32-6 添加Box类（Box.java）

```
public class Box {
    private PointF mOrigin;
    private PointF mCurrent;

    public Box(PointF origin) {
        mOrigin = mCurrent = origin;
    }

    public void setCurrent(PointF current) {
        mCurrent = current;
    }

    public Point getCurrent() {
        return mCurrent;
    }

    public PointF getOrigin() {
        return mOrigin;
    }
}
```

用户触摸BoxDrawingView视图界面时，新的Box对象将会创建并添加到现有的矩形框数组中，如图32-5所示。

回到BoxDrawingView类中，添加代码清单32-7所示代码，使用新的Box对象跟踪绘制状态。

代码清单32-7　添加拖曳生命周期方法（BoxDrawingView.java）

```
public class BoxDrawingView extends View {
    public static final String TAG = "BoxDrawingView";

    private Box mCurrentBox;
    private ArrayList<Box> mBoxes = new ArrayList<Box>();

    ...

    public boolean onTouchEvent(MotionEvent event) {
        PointF curr = new PointF(event.getX(), event.getY());

        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                Log.i(TAG, "  ACTION_DOWN");
                // Reset drawing state
                mCurrentBox = new Box(curr);
                mBoxes.add(mCurrentBox);
                break;

            case MotionEvent.ACTION_MOVE:
                Log.i(TAG, "  ACTION_MOVE");
                if (mCurrentBox != null) {
                    mCurrentBox.setCurrent(curr);
                    invalidate();
                }
                break;

            case MotionEvent.ACTION_UP:
                Log.i(TAG, "  ACTION_UP");
                mCurrentBox = null;
                break;

            case MotionEvent.ACTION_CANCEL:
                Log.i(TAG, "  ACTION_CANCEL");
                mCurrentBox = null;
                break;
        }

        return true;
    }
}
```

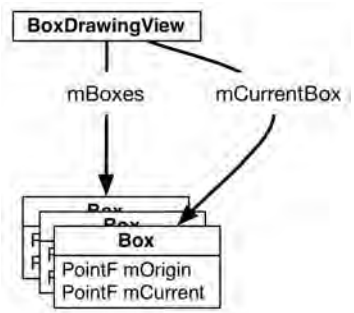

图32-5　DragAndDraw应用中的对象

只要接收到ACTION_DOWN动作事件，我们都以事件原始坐标新建Box对象并赋值给mCurrentBox，然后再添加到矩形框数组中。（下一小节实现定制绘制时，BoxDrawingView会将数组中的全部Box都绘制到屏幕上。）

用户手指在屏幕上移动时，mCurrentBox.mCurrent会得到更新。而在取消触摸事件或用户手指离开屏幕时，我们应清空mCurrentBox以结束屏幕绘制。已完成的Box会安全地存储在数组中，但它们再也不会受任何动作事件影响了。

注意ACTION_MOVE事件发生时调用的invalidate()方法。该方法会强制BoxDrawingView重新绘制自己。这样，用户在屏幕上拖曳时就能实时看到矩形框。这同时也引出我们接下来的任务：在屏幕上绘制矩形框。

32.4 onDraw(...)方法内的图形绘制

应用启动时，所有视图都处于*无效状态*。也就是说，视图还没有绘制到屏幕上。为解决这个问题，Android调用了顶级View视图的draw()方法。这将引起自上而下的链式调用反应，视图完成自我绘制，然后是子视图的自我绘制，再然后是子视图的子视图的自我绘制，如此调用下去直至继承结构的末端。当继承结构中的所有视图都完成自我绘制后，最顶级View视图也就不再无效了。

为参与这种绘制，可覆盖以下View方法：

```
protected void onDraw(Canvas canvas)
```

前面，在onTouchEvent(...)方法中响应ACTION_MOVE动作时，我们调用invalidate()方法再次让BoxDrawingView处于失效状态。这迫使它重新完成自我绘制，并再次调用onDraw(...)方法。

现在我们来看看Canvas参数。Canvas和Paint是Android系统的两大绘制类。

- Canvas类具有我们需要的所有绘制操作。其方法可决定绘制的位置及图形，例如线条、圆形、字词、矩形等。
- Paint类决定如何进行绘制操作。其方法可指定绘制图形的特征，例如是否填充图形、使用什么字体绘制、线条是什么颜色等。

返回BoxDrawingView.java中，在BoxDrawingView的XML构造方法中创建两个Paint对象，如代码清单32-8所示。

代码清单32-8 创建Paint（BoxDrawingView.java）

```
public class BoxDrawingView extends View {
    private static final String TAG = "BoxDrawingView";

    private ArrayList<Box> mBoxex = new ArrayList<Box>();
    private Box mCurrentBox;
    private Paint mBoxPaint;
    private Paint mBackgroundPaint;

    ...

    // Used when inflating the view from XML
    public BoxDrawingView(Context context, AttributeSet attrs) {
        super(context, attrs);
```

```
            // Paint the boxes a nice semitransparent red (ARGB)
            mBoxPaint = new Paint();
            mBoxPaint.setColor(0x22ff0000);

            // Paint the background off-white
            mBackgroundPaint = new Paint();
            mBackgroundPaint.setColor(0xfff8efe0);
        }
    }
```

有了Paint对象的支持，现在可将矩形框绘制到屏幕上了，如代码清单32-9所示。

代码清单32-9　覆盖onDraw（Canvas）方法（BoxDrawingView.java）

```
    @Override
    protected void onDraw(Canvas canvas) {
        // Fill the background
        canvas.drawPaint(mBackgroundPaint);

        for (Box box : mBoxes) {
            float left = Math.min(box.getOrigin().x, box.getCurrent().x);
            float right = Math.max(box.getOrigin().x, box.getCurrent().x);
            float top = Math.min(box.getOrigin().y, box.getCurrent().y);
            float bottom = Math.max(box.getOrigin().y, box.getCurrent().y);

            canvas.drawRect(left, top, right, bottom, mBoxPaint);
        }
    }
```

以上代码的第一部分简单直接：使用米白背景paint，填充canvas以衬托矩形框。

然后，针对矩形框数组中的每一个矩形框，通过其两点坐标，确定矩形框上下左右的位置。绘制时，左端和顶端的值将作为最小值，右端和底端的值作为最大值。

完成位置坐标值计算后，调用Canvas.drawRect(...)方法，在屏幕上绘制红色的矩形框。

运行DragAndDraw应用，尝试绘制一些红色的矩形框，如图32-6所示。

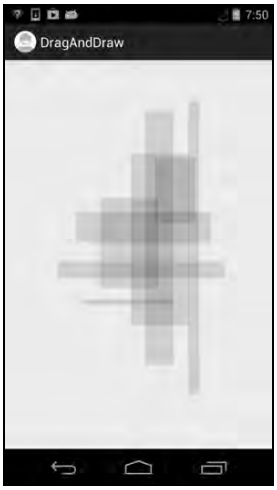

图32-6　程序员式的情绪表达

32.5 挑战练习：设备旋转问题

设备旋转后，我们绘制的矩形框会消失。要解决这个问题，可使用以下View方法：

```
protected Parcelable onSaveInstanceState()
protected void onRestoreInstanceState(Parcelable state)
```

以上方法的工作方式不同于Activity和Fragment的onSaveInstanceState(Bundle)方法。代替Bundle参数，这些方法返回并处理的是实现Parcelable接口的对象。我们推荐使用Bundle，这样就不需要亲自去实现Parcelable接口了。（Parcelable接口的实现很复杂，如有可能，应尽量避免。）

作为一个较有难度的练习，请实现以两根手指旋转矩形框。完成这项挑战，我们需在MotionEvent实现代码中处理多个触控点（pointer），并旋转canvas。

处理多点触摸时，还需了解以下概念。

- pointer index。获知当前一组触控点中，动作事件对应的触控点。
- pointer ID。给予手势中特定手指一个唯一的ID。

pointer index可能会改变，但pointer ID绝对不会。

请查阅开发者文档，学习以下MotionEvent方法的使用：

```
public final int getActionMasked()
public final int getActionIndex()
public final int getPointerId(int pointerIndex)
public final float getX(int pointerIndex)
public final float getY(int pointerIndex)
```

另外，还需查阅文档学习ACTION_POINTER_UP和ACTION_POINTER_DOWN常量的使用。