Android Drawing

Sisoft Technologies Pvt Ltd

Drawing Building Blocks

1. Canvas
2. Paint class
3. Bitmap
4. View
5. Surface View

❑Canvas

- The Canvas API allows to create complex graphical effects.
- You paint on a Bitmap surface. The Canvas class provides the drawing methods to draw on a bitmap and the Paint class specifies how you draw on the bitmap.

❑Paint

- The Paint class holds the style and color information about how to draw geometries, text and bitmaps.

❑onDraw()

- The most important step in drawing a custom view is to override the onDraw() method. The parameter to onDraw() is a Canvas object that the view can use to draw itself. The Canvas class defines methods for drawing text, lines, bitmaps, and many other graphics primitives. You can use these methods in onDraw() to create your custom user interface (UI).
- Before you can call any drawing methods, though, it's necessary to create a Paint object. The next section discusses Paint in more detail

Two-dimensional Drawing

- Two-dimensional graphics in Android are implemented differently than dynamic, interactive, or three-dimensional graphics.
- Android has 2D drawing APIs for creating this kind of graphics. Using these APIs, there are two ways of implementing graphics:

1. Drawing with Drawables
   - Bitmap Drawables
   - XML Drawables
   - Nine-patch
   - Custom Drawables
2. Drawing with a Canvas
   - On a View
   - On a Surface View

1.Drawing with Drawables

Overview of Drawables

- A Drawable is a general abstraction for "something that can be drawn.
- Every Drawable is stored as individual files in one of the res/drawable folders.
- There are two type of drawables:
  - Bitmap drawables
  - XML Drawables

(a) Bitmap drawables : are images files stored in res/drawable folder. Typically these are stored for different resolutions in the -mdpi, -hdpi, -xhdpi, -xxhdpi subfolders of res/drawable

- Android supports bitmap files in a three formats: .png (preferred), .jpg (acceptable), .gif (discouraged).

(b) XML drawables: are resource definition which points to a bitmap file and can also specify the additional properties.

1.(a) Bitmap Drawables

- Android allows you to use the Bitmap class for working with bitmaps. This section explain how to create Bitmap objects via Java code and how to convert Bitmap into Drawable objects and vice versa.
- If required you can load any accessible bitmap file in your code and convert them into Drawables objects.
- The following example code shows how to create an Bitmap object for the assets folder and assign it to anImageView.

1.(a) Bitmap Drawables

```
// Get the AssetManager
AssetManager manager = getAssets();
// read a Bitmap from Assets InputStream open = null;
 try
      {
       open = manager.open("logo.png");
       Bitmap bitmap = BitmapFactory.decodeStream(open);
      // Assign the bitmap to an ImageView in this layout
      ImageView view = (ImageView) findViewById(R.id.imageView1);
      view.setImageBitmap(bitmap);
  }
 catch (IOException e)
            { e.printStackTrace(); }
 finally { if (open != null)
     {
            try { open.close(); }
             catch (IOException e){ e.printStackTrace();
} } }
```

1.(a) Bitmap Drawables

You can also access the Drawables from your *res/drawable* folder as Bitmap objects in your source code. The following code demonstrates that.

```
Bitmap b = BitmapFactory.decodeResource(getResources(), R.drawable.ic_action_search);
```

You can create a scale bitmap based on a new weight and height definition in pixel.

```
Bitmap originalBitmap = <initial setup>;

Bitmap resizedBitmap =
   Bitmap.createScaledBitmap(originalBitmap, newWidth, newHeight, false);
```

To convert a `Bitmap` object into a `Drawable` you can use the following code.

```
# Convert Bitmap to Drawable
Drawable d = new BitmapDrawable(getResources(),bitmap);
```

1(b) XML Drawables

Any Drawable subclass that supports the **inflate()** method can be defined in XML and instantiated by your application. Each Drawable that supports XML inflation utilizes specific XML attributes that help define the object properties :

i) Shape Drawables

ii) State Drawables

iii) Transition Drawables

iv) Animation Drawables

(i)- Shape Drawables

- Shape Drawables are XML files which allow to define a geometric object with colors, borders and gradients which can get assigned to Views. The advantage of using XML Shape Drawables is that they automatically adjust to the correct size.
- The following listing shows an example of a Shape Drawable.

```
<?xml version="1.0" encoding="UTF-8"?>
 <shape
    xmlns:android=http://schemas.android.com/apk/res/android
    android:shape="rectangle">

   <stroke
        android:width="2dp"
        android:color="#FFFFFFFF" />

        <gradient
          android:endColor="#DDBBBBBB"
          android:startColor="#DD777777"
          android:angle="90" />

        <corners
        android:bottomRightRadius="7dp"
         android:bottomLeftRadius="7dp"
        android:topLeftRadius="7dp"
        android:topRightRadius="7dp" />
</shape>
```

(ii)- State Drawables

- State drawables allow to define states. For each state a different drawable can get assigned to the View. For example the following defines different drawables for a button depending on its state.

```
<?xml version="1.0" encoding="utf-8"?>
 <selector
        xmlns:android="http://schemas.android.com/apk/res/android">
       <item
       android:drawable="@drawable/button_pressed"
       android:state_pressed="true”/>
       <item
       android:drawable="@drawable/button_checked"
       android:state_checked="true”/>
        <item
       android:drawable="@drawable/button_default" />
       </selector>
```

(iii)- Transition Drawables

- Transition Drawables allow to define transitions which can be triggered in the coding.

```
<?xml version="1.0" encoding="utf-8"?>
<transition xmlns:android="http://schemas.android.com/apk/res/android">
 <item android:drawable="@drawable/first_image" /> <item
android:drawable="@drawable/second_image" />
 </transition>
```

```
final ImageView image = (ImageView) findViewById(R.id.image);
final ToggleButton button = (ToggleButton) findViewById(R.id.button);
button.setOnClickListener(new OnClickListener()
{
@Override
public void onClick(final View v)
{
        TransitionDrawable drawable = (TransitionDrawable) image.getDrawable();
        if (button.isChecked())
        { drawable.startTransition(500); }
        else
        { drawable.reverseTransition(500); }
 }});
```

(iv) - Animation Drawables

- You can also define an animation drawables and assign it to a View via the setBackgroundResource() method

```
<!-- Animation frames are phase*.png files inside the res/drawable/ folder --> <animation-
list android:id="@+id/selected" android:oneshot="false">
        <item android:drawable="@drawable/phase1" android:duration="400" />
       <item android:drawable="@drawable/phase2" android:duration="400" />
        <item android:drawable="@drawable/phase3" android:duration="400" />
       </animation-list>
```

```
ImageView img = (ImageView)findViewById(R.id.yourid);
img.setBackgroundResource(R.drawable.your_animation_file);
//Get the AnimationDrawable object.
AnimationDrawable frameAnimation = (AnimationDrawable) img.getBackground();
 // Start the animation (looped playback by default).
frameAnimation.start();
```

1.3 - 9Patch Drawables

- A NinePatchDrawable graphic is a stretchable bitmap image, which Android will automatically resize to accommodate the contents of the View in which you have placed it as the background.
- *9 Patch drawables* is standard PNG image that includes an extra one pixel additional border.
- It must be saved with the extension .9.png, and saved into the res/drawable/ directory of the project
- On the top and left you define the area which should be scaled if the *Drawable* is to small for the view. This is the stretch area. On the right and bottom side you define the area where a text should be placed if this Drawable is used on a view which can write text on it, e.g. a Button.
- The ADT supplies the *draw9patch* program in the android-sdk/tools installation folder, which makes it easy to create *9 Patch drawables*.

2. Drawing with a Canvas

On a View

On a SurfaceView

1. On a View

On a View

- To draw a simple graphic, one which does not require dynamic changes in the application, extend the View class and define an onDraw() callback method.

- The onDraw() method is passed with the Canvas, which can be used to draw a simple graphics using Canvas methods.

onDraw(android.graphics.Canvas);

Displaying an Image and Text in a Normal View :

To display an image and text in a normal View, create a new Project with the activity named ViewGraphicsActivity

```
public class ViewGraphicsActivity extends Activity
{
 @Override
 public void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(new MyView(this));
        }}
```

The request Window Feature(Window.FEATURE_NO_TITLE) method is added to show a window without a title. Next, create an inner class named MyView which extends from the View class and overrides the method onDraw(Canvas)

```
class MyView extends View
{
public MyView(Context context)
                { super(context); }
@Override
 public void onDraw(Canvas canvas) { } }
```

On a View

- The next step is to implement an onDraw(Canvas) callback method. For drawing you need a Paint object through which you can set Text properties like color, size, font, and style:

```
Paint paint = new Paint();
paint.setColor(Color.WHITE);
paint.setTextSize(20);
paint.setAntiAlias(true);
```

On a View

- Now make canvas background color blue using the drawColor(Color) method.

```
canvas.drawColor(Color.BLUE);
```

- Then using the paint object, we draw the Circle as follows:

```
canvas.drawCircle(90, 60, 40, paint);
```

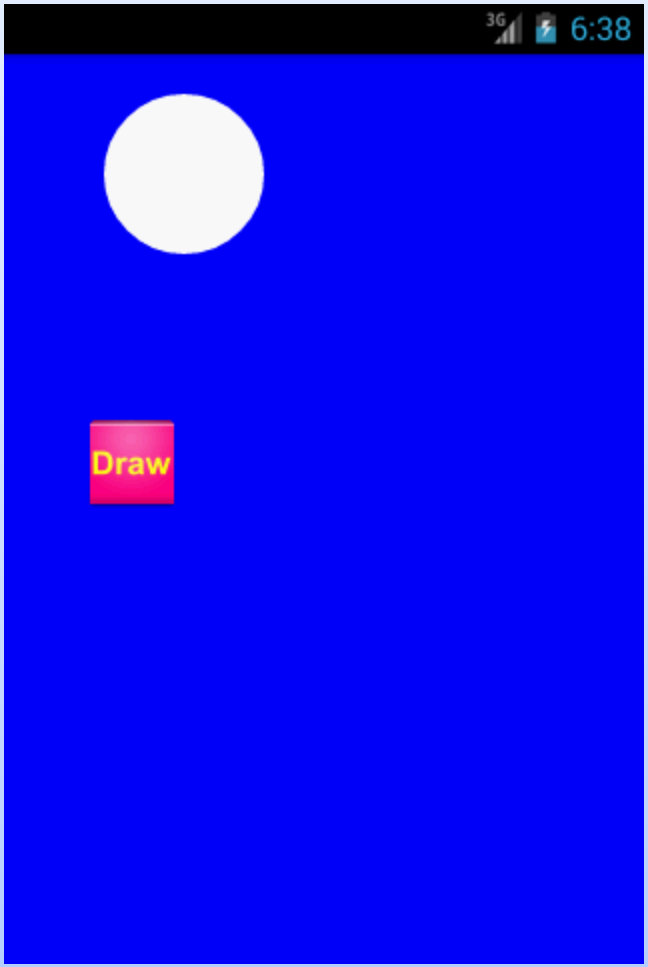
- To draw a bitmap image on the canvas, get the bitmap image from the resource:

```
Bitmap image = BitmapFactory.decodeResource(getResources(),
R.drawable.ic_launcher); canvas.drawBitmap(image, 40, 80, null);
```

Examples

```
public class ViewGraphicsActivity  extends Activity
{
@Override
public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
requestWindowFeature(Window.FEATURE_NO_TITLE);
setContentView(new MyView(this));
}
class MyView extends View
{
public MyView(Context context)
{ super(context); }

@Override
public void onDraw(Canvas canvas)
{
        Paint paint = new  Paint();
        paint.setColor(Color.WHITE);
        paint.setTextSize(20);
        paint.setAntiAlias(true);
        canvas.drawColor(Color.BLUE);
        canvas.drawCircle(10, 60, 16, paint);
        Bitmap image = BitmapFactory.decodeResource(getResources(), R.drawable.ic_launcher);
        canvas.drawBitmap(image, 40, 80, null);
}} }
```



Drawing with a Canvas
On a SurfaceView

Drawing with a Canvas on SurfaceView

- To draw dynamic 2D graphics where in your application, draw on the Canvas by extending SurfaceView.

- SurfaceView is a subclass of View where a dedicated drawing surface is offered within the View hierarchy. Here drawing is done in the application's secondary thread instead of waiting to be drawn by the system's view hierarchy.

- Your class should also implement a SurfaceHolder.Callback interface to get notification of the underlying surface events such as surface created, surface changed, and surface destroyed. These events help your application to determine when to start and stop drawing.

Displaying an Image and Text on a Surface View :

- Create a new Project with the activity named MySurfaceViewActivity. This project displays an image and text on the Canvas. The example shows how an image can be moved dynamically in the canvas. The sample code listens for the touch events and renders an image on the touched coordinates of the screen.

Displaying an Image and Text on a Surface View :

```
public class MySurfaceViewActivity extends Activity
{
 @Override public void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
requestWindowFeature(Window.FEATURE_NO_TITLE); setContentView(new MySurface(this));
} }
```

Next, create an inner class that extends SurfaceView and implements SurfaceHolder.callback and implement the three methods as shown below:

```
class MySurface extends SurfaceView implements SurfaceHolder.Callback
{
      @Override
       public void surfaceChanged(SurfaceHolder holder, int format, int width, int height){}
       @Override
        public void surfaceCreated(SurfaceHolder holder) {}
       @
       Override
       public void surfaceDestroyed(SurfaceHolder holder) {}
 }
```

On a Surface View

- Next, implement the callback methods and call the application's secondary thread inside a constructor

```
class MySurface extends SurfaceView implements SurfaceHolder.Callback
{ private SecondThread thread; //Initial position of the image
        private int x = 100;
        private int y = 200;
        public MySurface(Context context)
                { super(context); getHolder().addCallback(this);
                thread = new SecondThread(getHolder(), this); }
        @Override
         public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) { }
        @Override
         public void surfaceCreated(SurfaceHolder holder) { thread.setRunning(true);
        thread.start(); }
        @Override
         public void surfaceDestroyed(SurfaceHolder holder)
        {
        boolean retry = true; thread.setRunning(false);
        while (retry) { try { thread.join(); retry = false; } catch (InterruptedException e) {} } } }
```

On a Surface View

- Create the secondary thread in your application where the constructor requires parameters like mySurface and the SurfaceHolder.

```
class SecondThread extends Thread
 { private SurfaceHolder surfaceHolder;
private MySurface mySurface;
private boolean _run = false;
public SecondThread(SurfaceHolder surfaceHolder, MySurface mySurface)
{
    this.surfaceHolder = surfaceHolder; this.mySurface = mySurface;
}
public void setRunning(boolean run)
{ _run = run; }
 @Override
public void run()
      {       Canvas c;
               while (_run)
                {
                c = null;
               try {
                 c = surfaceHolder.lockCanvas(null);
               synchronized (surfaceHolder) { mySurface.onDraw(c); } }
               finally {
                if (c != null) { surfaceHolder.unlockCanvasAndPost(c);
      } } } } }
```

On a Surface View

- Now draw the graphics using the Canvas object:

```
public void onDraw(Canvas canvas)
{
        Paint paint = new Paint();
         paint.setColor(Color.WHITE);
        paint.setTextSize(20);
        paint.setAntiAlias(true); canvas.drawColor(Color.BLUE);
        canvas.drawText("Hello Android", 10, 20, paint);
        Bitmap image = BitmapFactory.decodeResource(getResources(),
        R.drawable.ic_launcher); //renders image using x and y parameter x and y value is filled
        by the touch //event canvas.drawBitmap(image, x, y, null);
}
```

On a Surface View

- The code renders the image using x, y parameters. The x, y parameters values are filled using touch events listener as shown in the following code:

```
@Override
public boolean onTouchEvent(MotionEvent event)
 {
      if (event.getAction() == MotionEvent.ACTION_MOVE) { x = (int) event.getX(); y = (int)
      event.getY(); } return true;
}
```

Android Touch Drawing

MotionEvent

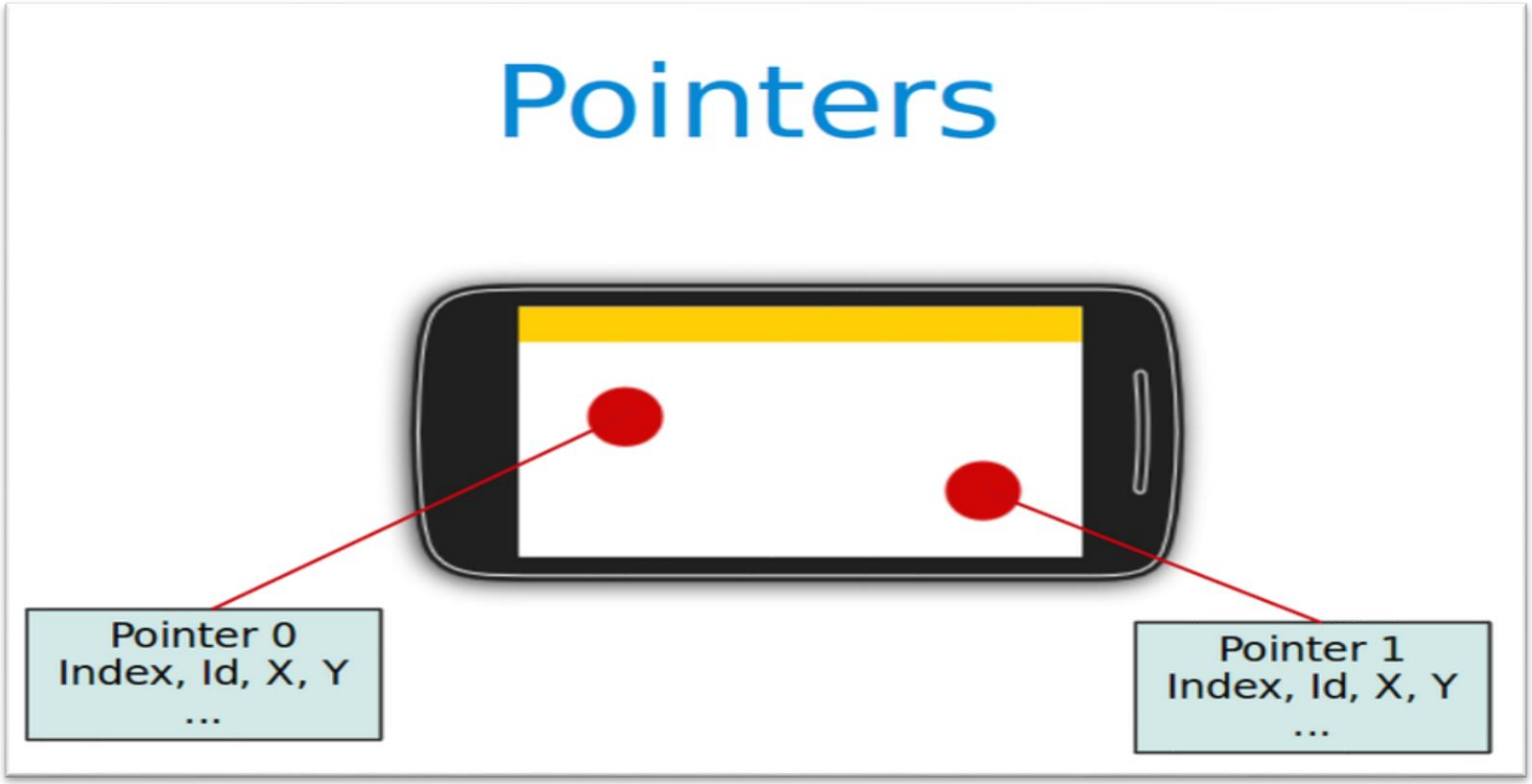
Single touch

MotionEvent

GestureDetectors

Android Touch

The Android standard View class support touch events. You can react to touch events in your custom views and your activities. Android supports multiple pointers, e.g. fingers which are interacting with the screen.



MotionEvent

- The base class for touch support is the MotionEvent class which is passed to Views via the onTouchEvent()method. To react to touch events you override the onTouchEvent() method.

- The MotionEvent class contains the touch related information.e.g. the number of pointers, the X/Y coordinates and size and pressure of each pointer.

Single touch

- If single input is used you can use the getX() and getY() methods to get the current position of the first finger.
- Via the getAction() method you receive the action which was performed. The MotionEvent class provides the following constants to determine the action which was performed.

Table 1. Touch Events

Event	Description
MotionEvent.ACTION_DOWN	New touch started
MotionEvent.ACTION_MOVE	Finger is moving
MotionEvent.ACTION_UP	Finger went up
MotionEvent.ACTION_CANCEL	Current event has been canceled, something else took control of the touch event
MotionEvent.ACTION_POINTER_DOWN	Pointer down (multi-touch)
MotionEvent.ACTION_POINTER_UP	Pointer up (multi-touch)

Multi touch

- Multi-touch is available since Android 2.0 and has been improved in the version 2.2. This description uses the API as of version 2.2.

- The MotionEvent.ACTION_POINTER_DOWN and MotionEvent.ACTION_POINTER_UP are send starting with the second finger. For the first finger MotionEvent.ACTION_DOWN and MotionEvent.ACTION_UP are used.

- The getPointerCount() method on MotionEvent allows you to determine the number of pointers on the device. All events and the position of the pointers are included in the instance of MotionEvent which you receive in the onTouch() method.

- To track the touch events from multiple pointers you have to use the MotionEvent.getActionIndex() and theMotionEvent.getActionMasked() methods to identify the index of the pointer and the touch event which happened for this pointer.

- This pointer index can change over time, e.g. if one finger is lifted from the device. The stable version of a pointer is the *pointer id*, which can be determined with the getPointerId(pointerIndex) method from theMotionEvent object.

```
@Override
public boolean onTouchEvent(MotionEvent event) {

// get pointer index from the event object
int pointerIndex = event.getActionIndex();

// get pointer ID
int pointerId = event.getPointerId(pointerIndex);

// get masked (not specific to a pointer) action
 int maskedAction = event.getActionMasked();

switch (maskedAction)
{       case MotionEvent.ACTION_DOWN:
        case MotionEvent.ACTION_POINTER_DOWN:
          { // TODO use data
                break;
           }
        case MotionEvent.ACTION_MOVE:
         { // a pointer was moved // TODO use data
             break;
         }
         case MotionEvent.ACTION_UP:
        case MotionEvent.ACTION_POINTER_UP:
        case MotionEvent.ACTION_CANCEL:
          { // TODO use data break;
         } }
         invalidate();
return true; }
```

Exercise: Singletouch Example

Draw via touch

- We will demonstrate Singletouch with an custom View.
- Create an Android project called *com.sisoft.android.touch.single* withthe *activity* called *SingleTouchActivity*.
- Create the following SingleTouchEventView class which implements a View which supports single touch.

❑Create the following SingleTouchEventView class which implements a View which supports single touch.

```
public class SingleTouchEventView extends View {
  private Paint paint = new Paint();
  private Path path = new Path();

  public SingleTouchEventView(Context context, AttributeSet attrs) {
    super(context, attrs);
    paint.setAntiAlias(true);paint.setStrokeWidth(6f);paint.setColor(Color.BLACK);
    paint.setStyle(Paint.Style.STROKE);paint.setStrokeJoin(Paint.Join.ROUND);
  }
  @Override
  protected void onDraw(Canvas canvas)
 {
    canvas.drawPath(path, paint);
  }
```

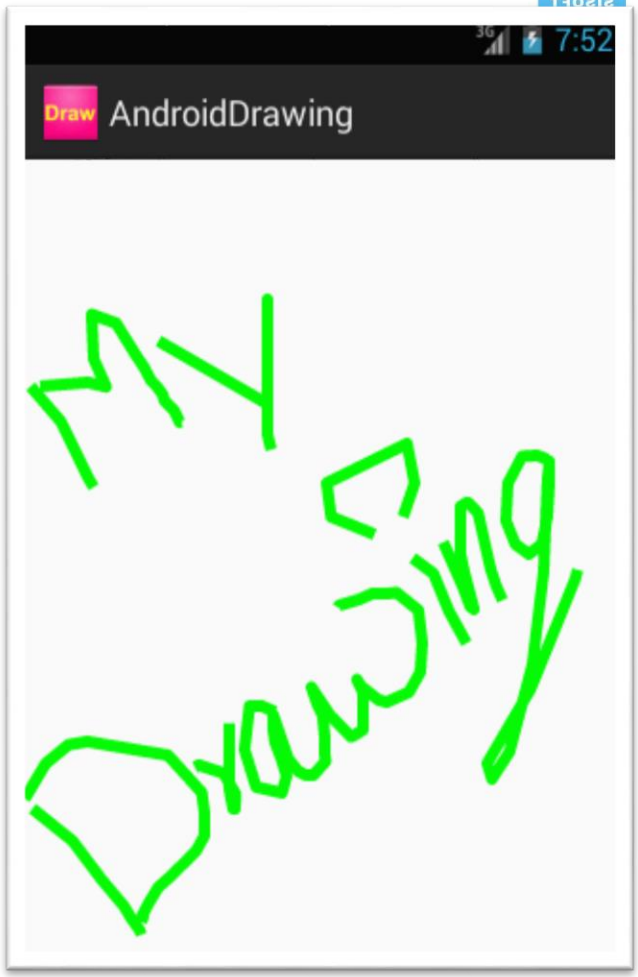
❑Create the following SingleTouchEventView class which implements a View which supports single touch.

```
@Override
 public boolean onTouchEvent(MotionEvent event) {
   float eventX = event.getX();
   float eventY = event.getY();

   switch (event.getAction()) {
   case MotionEvent.ACTION_DOWN:
     path.moveTo(eventX, eventY);
     return true;
   case MotionEvent.ACTION_MOVE:
     path.lineTo(eventX, eventY);
     break;
   case MotionEvent.ACTION_UP:
     // nothing to do
     break;
   default:
     return false;
   }
   // Schedules a repaint.
   invalidate();
   return true;
 }}
```

Now call view on Activity page :

```
import android.app.Activity;
import android.os.Bundle;
public class SingleTouchActivity extends Activity {
        @Override
   public void onCreate(Bundle savedInstanceState)
    {
          super.onCreate(savedInstanceState);
          setContentView(new SingleTouchEventView(this, null));
        }
}
```



- If you run your application you will be able to draw on the screen with your finger (or with the mouse in the emulator).
- Change your coding so that you use a layout definition based on XML. Hint: to use your own view in an XML layout definition you have to use the full-qualified class name (class including package information).

Exercise: Multitouch

- In this exercise you create a view which support multitouch and allows you to track several fingers on your device. On the Android emulator you can only simulate singletouch with the mouse.
- Create an Android project called *com.vogella.android.multitouch* with the >activity called *MainActivity*.
- Create the following MultitouchView class.

Create the following MultitouchView class :

```
public class MultitouchView extends View {

  private static final int SIZE = 60;
  private SparseArray<PointF> mActivePointers;
  private Paint mPaint;
  private int[] colors = { Color.BLUE, Color.GREEN, Color.MAGENTA,
     Color.BLACK, Color.CYAN, Color.GRAY, Color.RED, Color.DKGRAY,
     Color.LTGRAY, Color.YELLOW };

  private Paint textPaint;

  public MultitouchView(Context context, AttributeSet attrs) {
   super(context, attrs);
   initView();
  }

  private void initView() {
   mActivePointers = new SparseArray<PointF>();
   mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   // set painter color to a color you like
   mPaint.setColor(Color.BLUE);
   mPaint.setStyle(Paint.Style.FILL_AND_STROKE);
   textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   textPaint.setTextSize(20);
  }
```

```
@Override
 public boolean onTouchEvent(MotionEvent event) {

  // get pointer index from the event object
  int pointerIndex = event.getActionIndex();

  // get pointer ID
  int pointerId = event.getPointerId(pointerIndex);

  // get masked (not specific to a pointer) action
  int maskedAction = event.getActionMasked();

  switch (maskedAction) {
  case MotionEvent.ACTION_DOWN:
  case MotionEvent.ACTION_POINTER_DOWN: {
   // We have a new pointer. Lets add it to the list of pointers
   PointF f = new PointF();
   f.x = event.getX(pointerIndex);
   f.y = event.getY(pointerIndex);
   mActivePointers.put(pointerId, f);
   break;
  }
  case MotionEvent.ACTION_MOVE: { // a pointer was moved
   for (int size = event.getPointerCount(), i = 0; i < size; i++) {
    PointF point = mActivePointers.get(event.getPointerId(i));
    if (point != null) {
     point.x = event.getX(i);
     point.y = event.getY(i);
    }  }
   break;
  }
```

Exercise: Multitouch

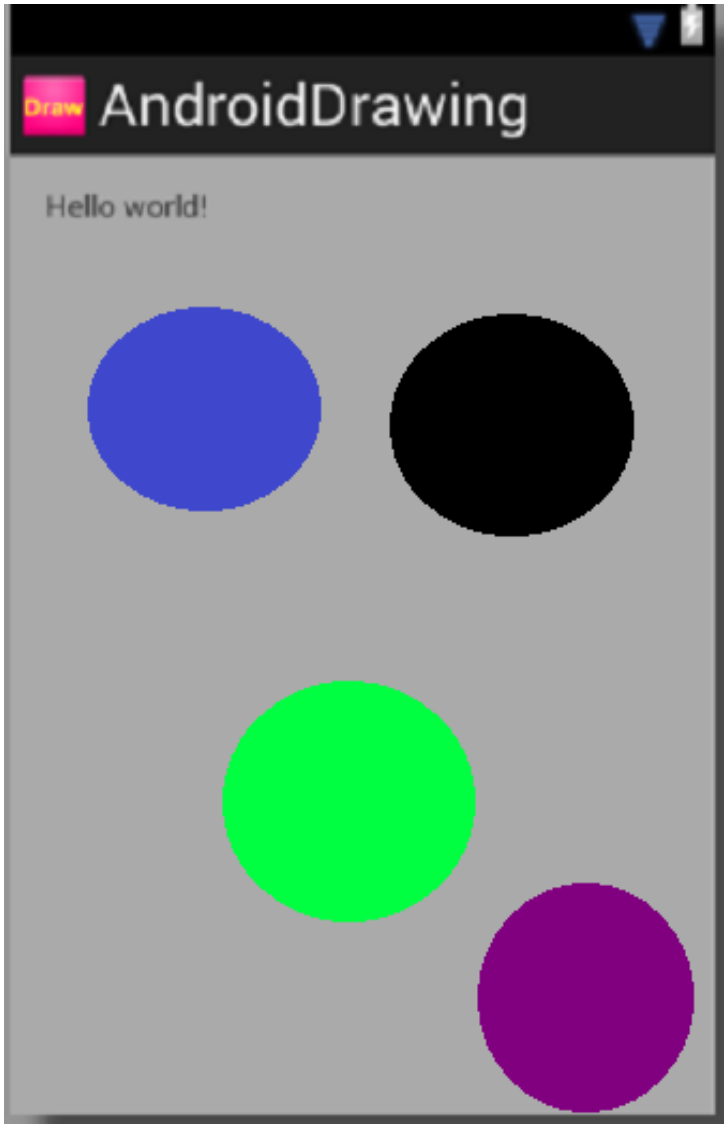
```
 case MotionEvent.ACTION_UP:
   case MotionEvent.ACTION_POINTER_UP:
   case MotionEvent.ACTION_CANCEL: {
     mActivePointers.remove(pointerId);
     break;
   }
   }
   invalidate();

   return true;
 }
@Override
 protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // draw all pointers
   for (int size = mActivePointers.size(), i = 0; i < size; i++) {
     PointF point = mActivePointers.valueAt(i);
     if (point != null)
       mPaint.setColor(colors[i % 9]);
     canvas.drawCircle(point.x, point.y, SIZE, mPaint);
   }
   canvas.drawText("Total pointers: " + mActivePointers.size(), 10, 40 , textPaint);
 }}
```

Exercise: Multitouch

- If you run your application you will be able to draw on the screen with your fingers. Every device has an upper limit how many pointers are supported, test out how many simultaneous pointers your device supports. This application should look similar to the following screenshot.



Gesture Detector

Gesture Detector

- Android provide the GestureDetector class which allow to consume MotionEvents and to create higher level gesture events to listeners.
- For example the Scale GestureDetector class allows to determine the predefined gesture of increasing and decreasing the size of the object via two fingers.
- Create the Android project called *de.sisoft.android.touch.scaledetector* with an Activity called *Scale DetectorTestActivity*.

Gesture Detector

```
public class ImageViewWithZoom extends View {
  private Drawable image;
  private float scaleFactor = 1.0f;
  private ScaleGestureDetector scaleGestureDetector;

  public ImageViewWithZoom(Context context) {
    super(context);
    image = context.getResources().getDrawable(R.drawable.ic_launcher);
    setFocusable(true);
    image.setBounds(0, 0, image.getIntrinsicWidth(),
        image.getIntrinsicHeight());
    scaleGestureDetector = new ScaleGestureDetector(context,
        new ScaleListener());
  }

  @Override
  protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Set the image bounderies
    canvas.save();
    canvas.scale(scaleFactor, scaleFactor);
    image.draw(canvas);
    canvas.restore();
  }
```

Gesture Detector

```
@Override
 public boolean onTouchEvent(MotionEvent event) {
  scaleGestureDetector.onTouchEvent(event);
  invalidate();
  return true;
 }

 private class ScaleListener extends
    ScaleGestureDetector.SimpleOnScaleGestureListener {
  @Override
  public boolean onScale(ScaleGestureDetector detector) {
   scaleFactor *= detector.getScaleFactor();

   // don't let the object get too small or too large.
   scaleFactor = Math.max(0.1f, Math.min(scaleFactor, 5.0f));

   invalidate();
   return true;
  }
 }
}
```

Gesture Detector

Add this View to your *activity :*

```
public class ScaleDetectorTestActivity extends Activity
 {
 /** Called when the activity is first created. */
@Override public void onCreate(Bundle savedInstanceState)
 { super.onCreate(savedInstanceState);
    setContentView(new ImageViewWithZoom(this));
 } }
```

If you run your application you should be able to shrink and enlarge the image via a multi-touch gesture (pitch zoom)