Custom Views and Multitouch

Custom Views and Multitouch

- At certian points during Android programming you will be required to develop a custom view
  - As not everything that you will require in terms of widgets will be provided by the SDK
  - A lot of applications will have at least on custom view in their layouts and views
- Here we will discuss the various elements that you will require to make a fully working custom view that interacts with touch

What the SDK provides

- The reason that we require custom views is that the views provided by the SDK only cover the most common items that most if not all applications require.
  - Standard things like buttons, radiobuttons, checkboxes, edit texts etc
  - If the SDK was to provide for all cases then the toolkit would be unfeasibly large and messy
  - To combat this (like all other GUI toolkits) a base view is provided which you can extend and override to produce your own views

Methods of Making a Custom View

- There are three methods of constructing a custom view in Android
  - Method 1: Subclass an already existing view
  - Method 2: Compose a view out of already existing views
  - Method 3: Do a full custom implementation

Method 1: Subclass an Existing View

- If you only want to add a small piece of behavior to an already existing view that can't be set through attributes then this is a good method
  - A good example of this is extending the NumberPicker class to accept a minimum and a maximum value through XML (see Example030)
  - Depending on behaviour required may need very little code
  - Rare chances that you can do this but great when you can.

Method 2: Compose out of existing views

- Here you would combine multiple existing views to create a more complex view.
  - An example of this would be creating an edit text with a clear button. (users find it frustrating to clear all text by hand)
  - Here we compose a small custom layout that defines our view. Note that when subclassing you will subclass the layout not one specific view
  - Like method 1 this requires very little code but again there are rare occasions when you will use this method.

Method 3: Full Custom Implementation

- Here you require a view that is not provided by the SDK, in this case you have to take the bare view class, subclass it, and do all the drawing operations yourself.
  - The most common form of writing custom views.
  - Requires a lot of mathematics and working with primitives
  - Mainly is a trial and error process as the display views can only be verified visually.

Requirements for a Custom View

- You have a minimum set of requirements that must be satisfied if you want your custom view to work correctly
  - You must provide all of the drawing operations
  - You must provide all interpretation of user touches on your view (if your view uses touch)
  - You must provide ways and means for your view to be customisable through attributes.

Constructors

- When you first subclass a view within Android you will be required to implement three constructors. All of which are used in a specific case. Assume we have a class CustomView, here is the format of the constructors
  - CustomView(Context c)
  - CustomView(Context c, AttributeSet as)
  - CustomView(Context c, AttributeSet as, int def_style)

Constructors

- The first constructor is used in the case where your custom view is being initialised directly in java code
  - Generally not recommended that you do this
  - The Context attribute is the context of the activity that will own this custom view
  - Allows android to build an internal visual tree to manage event handling

Constructors

- The second constructor is used in the case where the custom view is being added through XML and is being constructed by the layout inflator
  - The AttributeSet holds the list of XML attributes that the caller wishes to set on this custom view
  - You may accept as many attributes that you like
  - Generally a good idea as it makes your view much more flexible

Constructors

- Finally the third constructor is used when the caller applies a style to views within his XML layout
  - Permits your view to apply that style to itself
  - To enable better integration of your custom view into an application
  - Requires the most work but great custom views are completely flexible if they are to be used elsewhere

Constructors: Recommended Practice

- It is recommended that when you go to implement these constructors that all common code between them is split off into a different method
  - Basic bit of code refactoring. Write once, not three times
  - Thus if you find a mistake in common behaviour you need only fix it once
  - For our purposes we will call this method init()

What init() should be used for?

- Should be used for all items that are shared between all three constructors
  - Things like initialising colours, shapes, and any parameters or data structures that are relevent to your custom view
  - It is a good idea to initialise and construct all the objects required by your custom view here if you can
  - For performance reasons

onDraw()

- Every custom view you create must implement the onDraw() method
  - Responsible for redrawing the entire view, whenever android requests it, or you change the data to be represented by the view
  - All drawing must be described through primitives or bitmaps
  - Recommended that you use primitives as they are much quicker to render

Do not create objects in onDraw()

- Under no circumstances should you construct any objects in the onDraw() method
  - Will degrade performance as memory allocation/deallocation is an expensive operation
  - Make every effort to move all construction to the init method if possible
  - As this method is capable of being called 60 times a second depending on the amount of refreshes that are required

The importance of Algebra

- I'm sure many of you here have often wondered what the point and use of all that algebra you did in school/college was
  - Well you will need to use a lot of it here as the only method we have of describing drawing operations to a computer is through the use of mathematics
  - You will have to come up with all sorts of simple formulas to determine how to draw your components
  - Especially if you have primitives or sets of primitives that are required to move around the view

Advantages of describing through Algebra

- If you manage to describe a view accurately through algebra you will gain the following benefits
  - You will get a view that will display properly on any screen, regardless of the pixel density of the device or the physical screen size
  - If you previously used bitmaps to implement the view then you can remove them and save space
  - The component can be reused elsewhere with little to no modification required

Where should objects be created?

- We mentioned earlier that objects should be created in the init method where possible
  - This is because the view can possibly be updated upto 60 times a second (assuming a 60Hz refresh rate)
  - That means for every object you create on each draw call it can be created and destroyed a combined 120 times a second
    - This is bad because memory allocation and deallocation are very expensive operations, especially when a garbage collector is involved

Primitives

- As you will be doing all drawing you will have access to basic primitives
  - Things like, lines/arcs/circles/rectangles/ovals/text etc.
  - As these are easy to describe mathematically. More complex shapes can be described by combining these shapes together
  - Along with the use of transformation operations and a matrix stack

The Matrix Stack

- This is a stack of 4x4 matricies that determine where the origin of drawing is in the current view
  - As hardware acceleration is provided by OpenGL ES the view uses the same model that OpenGL ES provides
  - Also used to elimiate rounding errors that appear in floating point numbers.
    - As floating point numbers are an approximation and not a precise value

The Matrix Stack

- Initially when the view is first constructed and initialised the origin of drawing is at the top left corner of the view.
  - If you were to change the position and move back by using offsets you would eventually introduce rounding errors
    - As the position is representing by floating point numbers
  - By maintaining a history of the position of the drawing origin we can prevent these errors from occuring

Save and Restore

- Thus when drawing the view canvas will provide two methods for saving and restoring drawing origin history
  - We have save() for pushing the current origin onto the stack and restore() for reverting to the previous origin that is on the stack
  - Be aware that for every save() you must have a corresponding restore()
  - Otherwise you will cause graphical glitches very soon into your program.

Translate, Scale, and Rotate

- Standard operations for manipulating the drawing origin. Each of these operations will change the current drawing matrix by multiplying it with a matrix representing the given transform.
  - translate() moves the origin by a specific offset as we are doing 2D drawing the offset is an x and y value
  - scale() allows you to draw an object larger or smaller without having to change the coordinates of your object
  - rotate() allows you to rotate an object by any angle without having to change its coordinates by hand

Invalidation

- There will be times when you will modify the state of your view internally and will need it to be redrawn, as it is now out of date
  - For this we need the invalidate() method
  - Makes a request to the android system to redraw the view
  - Always use invalidate() **never ever ever** call the onDraw() method directly

onMeasure()

- Sometimes you will want to restrict the size and shape of your view
  - For example with game boards you may want to have a square shaped view.
  - onMeasure() will be called for you when the size of the view changes. This gives you a chance to state to android to reduce your view size if you do not require the extra pixels
  - Any changes made here will be reflected in the new size of your view.

onMeasure()

- For example if you wish to make a view square in android using this method
  - You will get a width and a height provided to you by this method
  - Pick whichever one is the smallest
  - Set the new width and height to the smallest value.

onTouchEvent()

- Most custom views will require some form of interaction with the user. In this environment the main method of interaction is through touch
  - Your interaction with touch is handled through the onTouchEvent() method
  - Either through the use of single touch (simple)
  - Or combining multiple touches together (complex)

Single Touch

- Single touch is the easiest case as interaction here is similar to that of a pointer in a GUI such as Java Swing or JavaFX. There are three main actions
  - ACTION_DOWN: when the touch has been started
  - ACTION_MOVE: when the touch is being dragged along the screen in any direction
  - ACTION_UP: when the touch has been finished

Multi-touch

- Multi-touch is more complex however, because the android method of touch was originally designed only to accept single touches
  - However, it was modified to include multi touch while still retaining support for applications that only supported single touch. Meaning we have two extra actions
  - ACTION_POINTER_DOWN: for when an additional touch is added
  - ACTION_POINTER_UP: for when an additional touch is removed

Integration with Layouts

- It is possible to include your custom views in layouts by adding them into XML layout files
  - Because your view is not one provided by the SDK you must provide the fully qualified class name of your java file containing the view
  - <package-name>.<class-name>

Attributes

- It is also possible to make your view understand attributes to make them more flexible
  - This is where that AttributeSet becomes handy as it divides attributes into namespaces
  - So for example the attribute foo:bar would have a namespace of foo with an attribute name of bar
  - Attributes without namespaces have a null namespace

Event Handlers

- Finally if you are considering releasing your custom view for the use of others you may wish to implement event handlers and event listeners
  - Prevents the need for other developers to access your source code
  - Gives them the minimum required effort to integrate your view into their own application
  - Won't cover here but is easy enough to implement