ES3 Lecture 10

Further Android development: UI design, maps, app widgets and using OpenGLES

Menus

- Android, unlike some other mobile platforms, supports *menus* in applications
- Menus can be be either:
  - key-triggered **Options** menu (appear when the Menu key brings up a menu)
  - context menus (appear when a control is held for a long time)
- Options menu is created dynamically
  - on first menu press the **onCreateOptionsMenu**() method of the current Activity is called
  - this should populate the menu with items
  - when options are selected, the **onOptionsMenuItemSelected**() method of the current Activity is called

Context Menus

- Context menus are created in much the same way
  - **onCreateContextMenu**() is created for the first time a **View** is long-pressed
  - the **View** is passed in
  - **onContextItemSelected**() is called when an item is selected

```
public void onCreateContextMenu(ContextMenu menu, View view, ContextMenuInfo info
{
  super.onCreateContextMenu(menu, view, info);
  if(view.id == R.id.launchItem)
  {
    menu.add(0, LAUNCH_ID, 0, "Launch");
    menu.add(0, RECALL_ID, 0, "Recall");
    menu.add(0, DISABLE_ID, 0, "Disable");
   }
}

public boolean onContextItemSelected(MenuItem item)
{
   if(item.getItemId()==LAUNCH_ID) { doLaunch(item); } // etc...
}
```

Notifications

- Android supports several kinds of simple notifications
- "**Toasts**" are simple message boxes which appear for a brief time
  - They can be launched from **Services**, and will appear over the current **Activity**

```
Toast.makeText(getApplicationContext(), "Lauch failed!", Toast.LENGTH_SHORT);
```

- Notifications can also appear in the status bar
  - This is rather more complicated and requires a **Notification** to be sent to a **NotificationManager**

Styles

- Android allows user interface components to have **styles**
  - Styles are hierarchical
  - Much like the was CSS specifies styles for HTML documents
- Styles are specified in XML files
  - stored in (any) XML file in **res/values**
- Each style is an element <**style**> with a name, with a list of <**item**> subitems
  - each subitem specifies an attribute, like **layout_width**, or **textColor**
- A component definition in a layout XML file can reference the style using the notation **@style/MyStyleName** where **MyStyleName** is the name of the <**style**> element
- Styles can inherit from other styles by specifying the **parent** attribute in the <**style**> tag
  - Individual attributes in specific controls can override style parameters (e.g. specifically specify **textColor**)

Example style usage

- In **res/values/styles.xml**

```
<resources>
<style name="RedStretch">
  <item name="android:layout_width">fill_parent</item>
  <item name="android:textColor">#ff0000</item>
</style>

<style name="RedStretch.text" parent="RedStretch">
<item name="android:typeface">serif</item>
</style>
</resources>
```

- In **res/layout/main.xml**

```
<Button style="@style/RedStretch" android:text=@"Press Me!"/>

<Button style="@style/RedStretch" android:text=@"Press Me!" android:textColor="#00ff00"/>
```

  - Note that the second button overrides the font color

Tween Animations

- Android supports animations much as the iPhone does
  - Like so many other things in Android, animations are usually specified in XML files and triggered when required
- Animation definitions go in **res/anim**
  - Consist of a series of animation types
    - **rotate**, **translate**, **scale**, **alpha**, or **set**
    - **set** allows grouping of animation elements (e.g. rotate and scale at the same time)
  - Each element specifies a **duration** and an **interpolator**
    - **Sets** allow interpolators to be shared among a number of elements (for synchronization)
- Various interpolators are available, like **LinearInterpolator**, **AccelerateInterpolator**, **AnticipateOvershootInterpolator**...
  - A bit richer than the iPhone standard interpolator types (linear, with optional ease in/ease out)

Attributes

- Animation attributes specify the change in their value
  - e.g. rotation specifies a start and end angle in degrees
  - transform attributes also specify **pivots**
    - this is the centrepoint about which transforms are made
    - e.g. rotation centre
- Animations can be loaded using **AnimationUtils.loadAnimation**()
  - e.g. **Animation spinFast = AnimationUtils.loadAnimation(this, R.anim.spinFast)**
- The animation is then passed to a specific **View,** by calling **startAnimation** on the view
  - **pacmanSprite.startAnimation(spinFast)**
- Android also supports frame animations for general drawables (not for **Views**)
  - i.e. switching images rapidly
  - an <**animation-list>** tag is used to specify a list of **drawables**
  - can cycle continuously or loop once

OpenGLES in Android

- Android supports OpenGLES with a standard set of bindings

- To use OpenGLES you must explicitly use **GLSurfaceView**
  - You can use this in place of any View
- Then implement a subclass of **opengl.GLSurfaceView.Renderer**
  - and assign the renderer to the View with **setRenderer()**
  - **onSurfaceCreated()** is called when the surface is created (i.e. at init)
  - **onDrawFrame()** is called for every redraw step
    - **all drawing code goes in here**

- Each method gets passed a **GL10** context object
  - this is an object which implements OpenGL calls
  - e.g. with gl_context.glColor4f(1,1,1,0.5)
  - or gl_context.glEnable(GL10.GL_BLEND)

- Note that all constants are also class members of the **GL10** object

Starting Services

- **Services** are Android's mechanism for background computation
  - A **Service** is usually launched from an **Activity** and persists until it is shut down
  - **It does not exit when the current task ends!**
- **Services** are started with **Intents**, as with other Android components
  - **Context.startService()** takes an **Intent** which specifies the service to start up
- Services must be declared in the **AndroidManifest.xml**
  - **Intent-filters** are used to specify the Intent that the service will respond to
- **Services** extend the **Service** class
  - Usually at least override **onCreate,** which is called when the **Service** is started

Binding to services

- In order to be useful, **Activities** (and other **Services**) need to communicate with a running **Service**
- Entities communicate with a **Service** by *binding* to it
  - This opens up a communication channel
- The specification of this communication channel must be laid out beforehand
  - This specifies the method calls the entity can use to communicate with the service, and the type and direction (e.g. in only or in and out) of parameters
- Android uses a specification language called AIDL to specify the procedure calls that can be used
  - AIDL is basically like Java method prototypes

AIDL

- AIDL files define a remote interface
  - Consist of an interface definition with a series of method definitions

- Interface parameters can be primitives, Strings, CharSequences, Lists or Maps or can be types imported from other packages
- Every parameter must specify a direction
  - in, out or in out
  - primitive types can only be in (no way to write to a boolean parameter, for example)

```
package com.es3.labs.SampleApplication

// Must be specifically imported!
import com.es3.labs.SampleApplication.TargetType;

interface ILauncher {
  void setTarget(in double latitude, in double longitude);
  boolean isOnTrack();
  void getTarget(out TargetType target);
}
```

Implementing the Interface

- In order to use the AIDL file, you must provide an implementation
- This is done by providing a **stub**
  - Each interface defined by the AIDL file has an automatic **stub** variable
  - You set this to a class instance which matches the interface and returns the values
- This interface object is then returned to the binding object (e.g. the Activity that started the service) when the class is bound (with **bindService**)
  - The methods on this interface can then be called by the binding object