HANDS ON
Create apps in Basic for Android, part 2
Darien Graham-Smith demonstrates how to create graphical Android apps with dynamic layouts

In part 1 of our introduction to Basic for Android, we saw how simple it is to create Android apps in this low-cost development environment. This month, we’ll look at some advanced features and functions you can include in your apps. If you missed last month’s instalment, download it from www.pcpro.co.uk/links/2211d1 – or you could just dive straight in. B4A is highly accessible and builds on established programming principles, so if you’ve ever written a BASIC program before you’ll find it easy to hit the ground running.

Adding graphics to your app
Last month we saw how easy it is to place Android views – buttons, labels and so forth – on the screen and plumb in code to make them all work together. However, an app built entirely out of standard system UI elements will look drab and characterless.

In B4A, it’s easy to add custom graphics to your app. As an example, let’s imagine we’re creating a simple stopwatch app with three custom buttons: start, stop and reset. We might start by positioning these as regular buttons in the Designer, then modify their properties to make them appear as custom images rather than the default appearance.

To do this, click on “Add Images” in the Designer and select the files you want to use – most popular file formats are supported. In our example, we’ve chosen three royalty-free images from Find Icons (www.findicons.com).

The images don’t immediately appear on-screen; adding them just makes them available for your interface to use. To attach the “play” image to our first button, we must change which sort of “Drawable” it is, from the default mode to one with its own set of customisable properties. We do this by selecting the button and changing its Drawable property from DefaultDrawable to StateListDrawable.

When you change this setting, you’ll see a new set of properties. Set EnabledDrawable to BitmapDrawable and a property will appear entitled “Image file”. Click in the blank space next to this for a dropdown menu showing your imported images. Choose the appropriate icon and the Designer view will update to show your image, scaled to fit the area of your button (you can change this behaviour by adjusting the image’s Gravity property). The name of the button is superimposed on the graphic, but this is just for convenience within the Designer – if you check the button’s Text property you’ll see it’s blank, and there’ll be no text overlaid when your code runs.

You can also specify additional images (or colours or gradients) for when the button is pressed and when it’s disabled.

Animation
If you’re interested in writing games, you’ll want to know how to animate your views. Let’s create an example app that moves a spaceship randomly around the screen. This involves using library functions that aren’t available in the B4A trial – the full version is required.

We’ll start by creating a panel in the designer; this will be our play area. We’ll use the GradientDrawable mode to give it a colour gradient, and we’ll set the Corner radius to ten to give it rounded corners.

Next, we’ll place our spaceship – we’ve used a charmingly retro image, once again from Find Icons – in its starting position. For this we’ll use an ImageView, a type of view that houses a graphic, and we’ll set its parent property to specify the panel rather than the activity. This means the spaceship can move only within the play area – it won’t be visible if it goes off the edge – and its dimensions and co-ordinates will be relative to the panel rather than the activity.

“An app built entirely out of standard system UI elements will look characterless”
As before, we click Add Images to load our spaceship, and adjust the Image file property of the ImageView to display it. We've used an image editor to give the graphic a transparent background, so it sits neatly on top of the playfield, and saved our layout under the name “spaceship-layout”. Now it’s time to return to the editor and create the necessary code to make the spaceship move. Here’s what we’ve come up with:

```vbnet
Sub Process_Globals 'This sub must always be present, even when it has no content
    End Sub

Sub Globals
    Dim spaceship As ImageView
    Dim playArea As Panel
    Dim animate As Animation
    Dim newX, newY As Int
    Dim moveTimer As Timer
End Sub

Sub Activity_Create(FirstTime As Boolean)
    Activity.LoadLayout("spaceship-layout")
    moveTimer = New Timer()
    Initialize("moveTimer", 1000)
    moveTimer.Enabled = True
End Sub

Sub moveShip
    MoveTimer.Tick
    newX = Rnd(0, playArea.Width)
    newY = Rnd(0, playArea.Height)
    MoveShip
End Sub

Sub Animation_AnimationEnd
    spaceship.Left = newX
    spaceship.Top = newY
End Sub
```

As this code illustrates, animations in B4A are objects. They’re configured via methods and properties, and activated using the Start method, passing the target object as a parameter. We’ve used the InitializeTranslate method to set up a lateral movement, but with other methods we could animate the alpha (opacity), scale and rotation of our spaceship. These tools are all found in B4A’s external Animation library, so if you want to compile this program for yourself, you’ll need to add a reference to it. To do this, click to activate the Libs panel at the right of the B4A window and tick “Animation”.

If you now try running this code, you’ll immediately see that in Android animated objects return to their original position once the animation is complete. In our program, the spaceship repeatedly moves away from its original location then jumps back to its original position. Happily, this problem can be easily fixed as follows.

```vbnet
Sub Animation_AnimationEnd
    spaceship.Left = newX
    spaceship.Top = newY
End Sub
```

As an animation completes, an AnimationEnd event is raised, which we can check for with a regular event-handling sub. Within this sub, we use the SetLayout method to immediately move the spaceship to the position where the animation ended. To the user, it appears that the spaceship has moved to its destination and stayed there.

Finally, this program also illustrates the use of timers to trigger events. When the program first runs – that is, when the main activity is created – we set up a timer for one second (the parameter is in milliseconds) and enable it. Since our animation is set to last for 750 milliseconds, the result is a spaceship that moves, hesitates briefly, then moves again.

The program continues to run until externally terminated, but you could easily add elements to give the user more control, such as allowing him or her to adjust or suspend the timer. Alternatively, you could create an event handler for playArea_Touch, and program the spaceship to home in on the user’s finger when the screen is tapped.

Handling multiple orientations
Our flying saucer program fits nicely on the screen in portrait mode, but if you turn your phone on its side – or press Ctrl-F11 to rotate JND into landscape mode – the orientation changes and suddenly the play area doesn’t fit on the screen any more. All the views keep their original positioning, relative to the top-left corner of the screen, but effectively, the screen itself changes shape.

The easiest way of dealing with this is to tell your app to always run in portrait mode (or if you prefer, always in landscape), and never rotate. You can do this from the B4A editor’s menus – just go to Project | Orientations Supported and make your choice.

If you’d like to handle things more gracefully, however, you can make your app

---

**Generating your personal encryption key is necessary so you can prove your identity and update your own apps on Google Play if need be**

www.pcpro.co.uk
Animation functions aren’t available until you click to include the relevant library.

Regardless of the screen size and resolution. For more on Designer scripting, see www.pcpro.co.uk/links/221id2 — and for more on working with different screen sizes, see Lucky dips, right.

All of the variants you create are saved within the same layout file, and Android automatically chooses the layout that most closely matches the user’s screen size and orientation, so once you’ve set up your variants there’s almost nothing more for you to do.

We say “almost”, since you may find your program behaves unexpectedly if the user rotates their phone or tablet while it’s running. Doing this causes Android to destroy the current activity and jump back to the Activity_Create sub to load the new layout. This might cause your app to look like it’s abruptly restarted. If there’s something you don’t want to be repeated each time the orientation changes, you can test the FirstTime argument that’s referred to in the sub declaration.

Sub Activity_Create(FirstTime As Boolean) 'Activity_LoadLayout("spaceship")
    If FirstTime = True Then
        MsgBox("Welcome to my app!",
        ")
    End If
End Sub

What if you’re not using the designer? As we saw last month, you can create and modify layouts in code, using the AddView and SetLayout methods (among others). You might choose to do this way if, for example, you want to produce a layout that changes dynamically according to what’s happening in your program.

In this case, you can easily determine the dimensions and orientations of the screen by checking the properties of the activity, and then direct your code accordingly.

If Activity.Height > Activity.Width Then
    'set up portrait layout here
Else
    'set up landscape layout here
End If

File handling

Android apps don’t automatically terminate when the user returns to the homescreen; they sit suspended in memory, ready for the user to return to. However, if the user presses the back button the app is terminated. Suspended process will also be killed if the device runs out of memory or reboots.

So you can’t assume that variables and program state will survive between user sessions. If you want data such as the high score for a game or the user’s current information to survive, you’ll need to write it out to storage.

Sample code to keep a persistent high score value might look like this:

Dim myScore As Integer
Dim hiScore As Integer
If File.Exists(File.DirDefaultExternal, "hiscore") Then
    hiScore = File.ReadInteger(File.DirDefaultExternal, "hiscore")
Else
    hiScore = 100
End If

If myScore > hiScore Then
    hiScore = myScore
    File.WriteString(File.DirDefaultExternal, "hiscore", hiScore)
End If

The general-purpose File object provides the methods and properties used for simple file handling. In the example above, we use it to check whether a file called “hiscore” is present in the default external directory (we’ll talk more about this in a moment). If it is, we load its contents into an integer variable; otherwise, we set that to a default value. Later in the game, we can overwrite the file with a new value if the player achieves a higher score. You’ll notice that B4A isn’t at all fussy about variable types, allowing us to read raw data from a file and assign its value directly to an integer variable, with no conversion required.

The default external directory, where we’ve saved our high-score file, is a folder dedicated to our particular app, buried away (but publicly accessible) on the device’s SD card. It’s created automatically when needed and is a natural place to save persistent, app-specific data. You have other options, though. The File.DirRootExternal property refers to the root of the external storage.
volume, so your program can access the user’s own files, or create files in plain sight.

For sensitive data, another option is File.DirInternal, which is inaccessible to other apps and less likely to disappear should the user swap out their SD card. However, files written here are stored on the device’s internal storage, which may be comparatively small, so use it thoughtfully.

A final location to be aware of is File.DirAssets. This is a read-only resource that contains copies of the files in your app’s Files folder (a subdirectory of the folder where you saved the source code). If you used the Designer to add images to your layout, this is where they’ll be stored. The entire contents of the folder are bundled and included with your app, so keep your Files folder as clean as possible or you’ll be wasting the user’s space. The Tools 1 Clean Files folder will automatically remove any unreferenced files from here — but use it with caution, since unused files are permanently deleted.

These methods and properties should provide everything a simple program needs, but B4A can handle more advanced tasks too. The RandomAccessFile library that accompanies the full version of B4A enables you to work with big data files without having to load them into memory in their entirety, and the SQL library makes it easy for your app to store and refer to entire databases of information.

Submitting your apps to Google Play

The topics we’ve covered these past two months may not include everything you need to create an Android masterpiece, but you should now be well equipped to continue working on your own, making use of the extensive documentation on the B4A website, and its informative development discussion forums. It may not be long before you’re ready to start distributing your creations via Google Play — and since B4A produces standard APK files, this process is fairly straightforward.

Before submitting your first app, you’ll need to sign it with a private encryption key that proves you’re the creator. A key can be generated from within the B4A editor by selecting Tools 1 Private Sign Key, entering your personal details and providing a password. Your key will be generated and saved in a keystore file (click Browse to choose a location and filename). Don’t lose this file. Once you start submitting apps to Google Play, you’ll need this key to update them. B4A will automatically use this key to sign compiled programs from now on, unless you load a new one via the same menu item.

You should also double-check that your Package Name and Application Label are set appropriately — you’ll find these options under the Project menu. Select Project 1 Choose Icon to give your app a distinctive icon, and select Project 1 Application Version to give your app a version number and an identifying string.

Once the above is complete, it’s time to generate your APK file. Set the compilation mode to “Release” — or “Release (obfuscated)” if you want to make your program harder for hackers to decompile — and hit the Run button. Now look in the source code folder and you’ll see a directory called Objects. You’ll find the APK for your program in here.

Lucky dips

As well as different screen sizes and resolutions, different Android devices have different pixel densities. Some, such as the Nexus 7, pack the pixels so densely you can barely see them with the naked eye, while older smartphones have larger pixels. As a result, if you create a button that’s 50 pixels square, for example, you might find that it’s easy to press on one device but fiddly on another.

Android gets around this by letting you specify sizes in “density independent pixels”, also known as dips or dpi for short. These are virtual pixels that scale automatically to suit the display density of the device at hand. A button measuring 160 x 160 dips should always appear on screen at around 1 in square, regardless of the screen type and resolution.

The B4A designer measures dimensions in dips. If you create an interface element with dimensions of 100 x 100, for example, it will be drawn using however many pixels are needed to obtain the correct real-world scale of around two-thirds of an inch. You can find out the absolute size of a view in pixels by reading its width and height properties, and you can convert any number of dips to pixels using the DipToCurrent() function, putting the number of dips as a parameter in brackets.

In fact, if you’re moving or creating elements within your code, you may choose to forget about pixels altogether. Dimensions and distances can be specified in dips, or as percentages of the activity’s horizontal and vertical dimensions. For example, if you want to position a 2in square panel halfway down the page, use the code:

myPanel.SetLayout(0, 500y, 320dip, 320dip);

Many programmers advise avoiding pixel measurements altogether and specifying your entire layout in dips; this helps your app cope not only with a wide range of current screens, but with unimagined future ones too.

To upload your app to Google, you’ll need to register as an Android developer, if you haven’t already. This costs $25 — around £15 — and if you have an existing Google account it can be done in less than a minute at https://play.google.com/apps/publish/signup. Once this is approved, you’ll be taken to the Google Developer Console, where you’ll see a link to upload your app.

All done? Not quite. You’ll now be required to upload at least two screenshots of your app in approved formats, and a large 512 x 512-pixel icon for your app, as well as listing

“If you want to charge for your app, you’ll need to sign up for a Google Merchant account”

details. It’s worth spending some time on these details, since they’ll represent the entirety of your “shopfront” in Google Play. If you want to charge for your app, you’ll also need a Google Merchant account, which you can sign up for from the page.

When this is all done, scroll back up to the top of the page, select the APK files tab and activate the file you’ve just uploaded. Hit Publish and — as long as there are no errors in your submission to correct — your app will go live within a few hours. Congratulations! You’re a published app author. Now all you have to do is sit back and wait for the money to roll in.

Our spaceship app shows how easy it is to create animated graphics

www.pcpro.co.uk