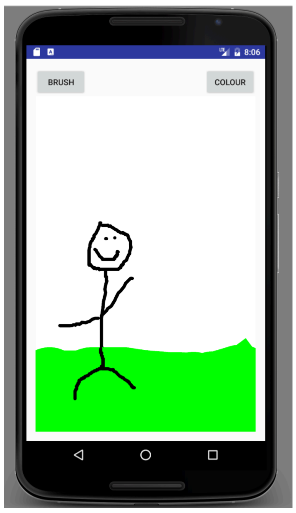
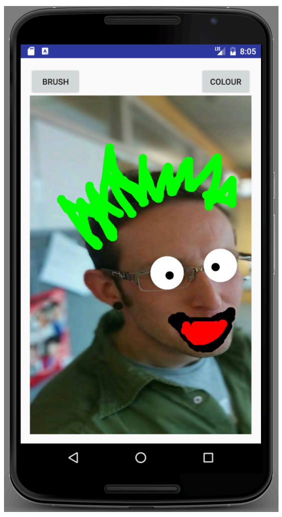
G53MDP Coursework 1-1 Fingerpainter





Summary

In this exercise you are required to build an Android painting application. This is an assessed exercise and will account for **10% of your final module mark**. This is an individual coursework, and your submission must be entirely your own work – please pay particular attention to the section of this document regarding plagiarism. This document sets out the requirements of and broad instructions for developing the application.

Your application should be submitted no later than:

- **3pm on the 5th of November 2018**

Submissions should be made electronically via Moodle. Standard penalties of 5% per working day will be applied to late submissions.

Your application should be submitted as a .zip or .tar.gz file containing all relevant source code, configuration and related files, and a compiled .apk file – i.e. the contents of the directory containing your Android Studio project. Do not submit RAR files.

Specification

You should create an application to support a simple drawing / painting task, where the user can draw onto a canvas using their finger in a variety of colours and with brushes of different sizes and shapes. The user should either begin with a blank canvas or be able to navigate from an existing image in the device's *Downloads* folder that will be loaded by the application.

Your application should consist of three Activity components:

- An *Activity* presenting an interface for the user to draw onto
- An *Activity* allowing the user to select the colour with which to draw
- An *Activity* allowing the user to select the shape and size of the drawing brush

Furthermore, the application should:

- Integrate with other applications to allow the user to manipulate existing images into the task
- Support appropriate persistence of *Activity UI state*

A custom painting *View* is provided that should form the basis of the application. It is left up to you to decide how best to design and implement layouts and other views within these activities.

Your application must be written in Java and make use of the Android SDK. There are no requirements to target a specific Android API version, however you can assume that your application will be tested on an emulated device (1080 x 1920 420dpi) running Android API version 28 (Android 9.0 Pie).

You should consider the following when implementing your application:

- Decomposition of the task into discrete Activity components
- Appropriate use of Intents, communication between Activities and appreciation of the Activity lifecycle
- Appropriate use of Widgets, Views and ViewGroups for layouts that support devices of differing screen sizes and resolutions
- Your application should have appropriate comments and variable / class names, so that a reader can easily understand how it works if necessary

Assessment Criteria

As this is a constrained exercise marks are awarded for achieving specific functionality as follows. For all elements either 0 or full marks are awarded as appropriate. There are no additional marks available for additional functionality in this exercise:

	Marks
The application has an Activity that allows the user to paint onto a blank canvas	4
A second Activity allows the user to select one of at least 5 colours	3
A third Activity allows the user to select the shape and size of the brush	4
The colour choosing Activity is passed and displays the currently chosen colour	2
The brush choosing Activity is passed and displays the current shape and size of the brush	2
The painting Activity maintains colour and brush state throughout its expected lifecycle	2
The application can be opened from and displays an image from the Downloads folder on the device for painting	3
Total	20

Plagiarism

N.B. Use of third party assets (tutorials, images, example code, libraries etc.) MUST be credited and referenced, and you MUST be able to demonstrate that they are available under a license that allows their reuse.

Making significant use of tutorial code while referencing it is poor academic practice, and will result in a lower mark that reflects the significance of your own original contribution.

Copying code from other students, from previous students, from any other source, or soliciting code from online sources and submitting it as your own is plagiarism and will be penalized as such. FAILING TO ATTRIBUTE a source will result in a mark of zero – and can potentially result in failure of coursework, module or degree.

All submissions are checked using both plagiarism detection software and manually for signs of cheating. If you have any doubts, then please ask.

Instructions

Finger Painting

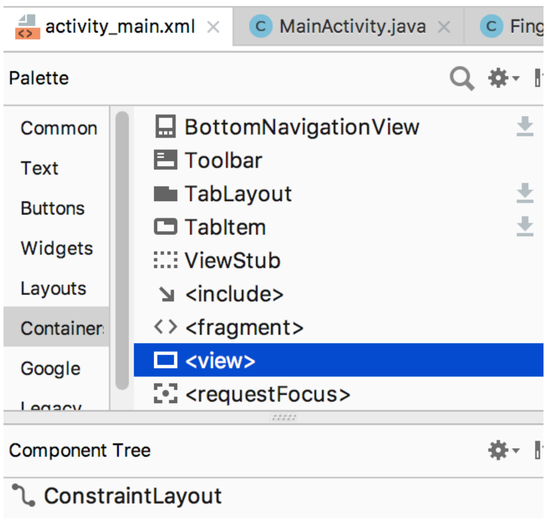
Begin by creating a new application in Android Studio as usual.

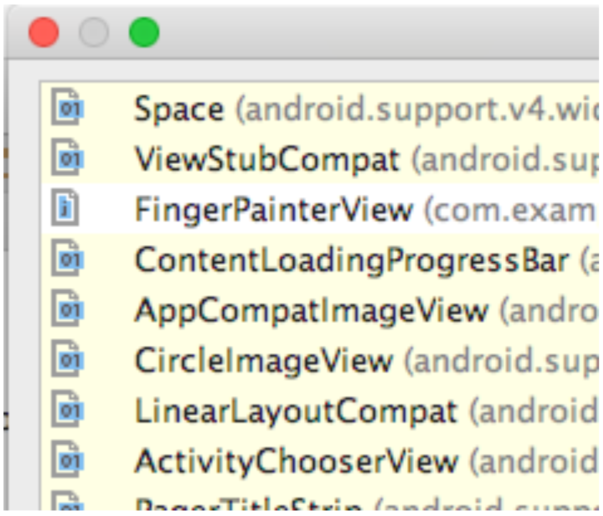
Add the custom view *FingerPainterView* to your app project. This class is available on Moodle or at the URL below:

https://github.com/mdf/g53mdp/blob/master/FingerPainter/FingerPainterView.java

You can either copy the file directly into your project's source directory (*projectname/app/src/main/java/…*) or create a new FingerPainterView Java class in your project and copy / paste the code into it, updating the package qualifier accordingly.

Add an instance of the *FingerPainterView* to the layout of your main activity. You can do this either statically through its XML layout resource (Palette->Container-><view>->FingerPainterView):




Space (android.support.v4.wi
ViewStubCompat (android.sup
FingerPainterView (com.exam
ContentLoadingProgressBar (a
AppCompatImageView (andro
CircleImageView (android.sup
LinearLayoutCompat (android
ActivityChooserView (android

Or add it to an appropriate *ViewGroup* programmatically when the activity is created:

```
        FingerPainterView myFingerPainterView = new FingerPainterView(this);
        myFingerPainterView.setId(R.id.myFingerPainterViewId);
        myFrameLayout.addView(myFingerPainterView);
```

Note in either case the FingerPainterView must have an ID, otherwise the view will not receive the cascaded save state calls (if a view does not have an ID the OS doesn't know where it should restore the view to on restore).

If adding the FingerPainterView programmatically you will need to create a static ID resource in res/values/strings.xml and assign it to the view on creation, otherwise the view will not receive the cascaded save state calls correctly:

```
<resources>
    <string name="app_name">FingerPainter</string>
    <item name="myFingerPainterViewId" type="id" />
</resources>
```

If adding the FingerPainterView via the XML layout you must also remember to manually allocate an ID to the view as using the attributes panel. You should retrieve and maintain a reference to this *View* in the same way as the other views in lab exercise two (*findViewById(…)* if you're instantiating it via the layout resource).

You may find it useful to nest the FingerPainterView inside a FrameView as in the code above.

FingerPainterView

FingerPainterView should automatically create a square paintable canvas that fills the view space that it is given, and responds to touch ("mouse") events by drawing lines onto its internal bitmap. This bitmap **is** automatically persisted to a cache file.

The class has a few public set / get methods for managing the colour and shape of the brush:

```
         public void setColour(int colour)
```

…sets the colour of the brush, where colour is of the form 0xAARRGGBB for setting alpha, red, green and blue values respectively. Alpha should always be 255, i.e. 0xFF. For example, to set the drawing colour to green:

```
        myFingerPainterView.setColour(0xFF00FF00);
```

Or:

```
import android.graphics.Color;

        myFingerPainterView.setColour(Color.GREEN);
        myFingerPainterView.setColour(Color.parseColor("#FF00FF00"));
```

To set the shape of the brush:

```
	public void setBrush(Paint.Cap brush)
```

Where brush is either round or square, defined by the *Paint.Cap enum*:

```
import android.graphics.Paint;

	Paint.Cap.ROUND
	Paint.Cap.SQUARE
```

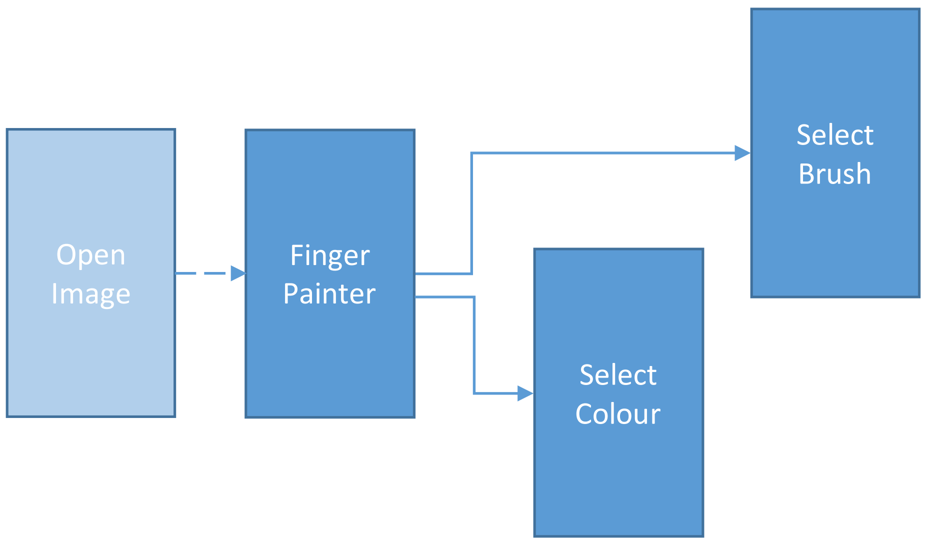
To set the width of the brush:

```
	public void setBrushWidth(int width)
```

Where width is an integer value in pixels.

The equivalent *get* methods allow you to retrieve the current state of these values. They are **not** persisted by the view between instances.

The Task



Create two new activities to allow the user to specify the colour, and size and shape of the brush respectively.

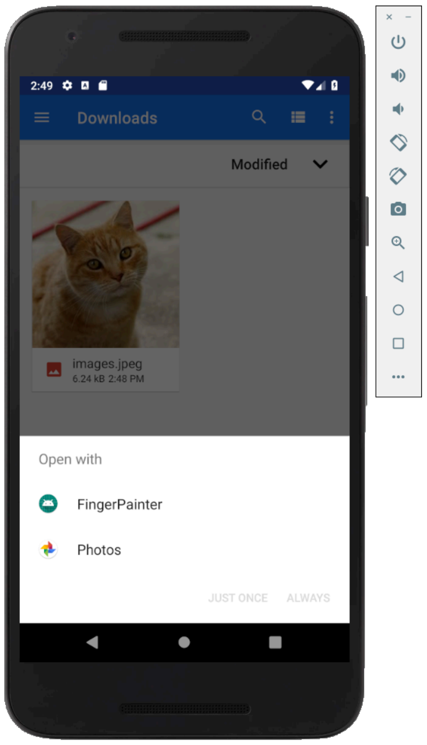
Each of these activities should be passed the current colour and brush size / shape respectively – i.e. the colour selection activity should know that the user is currently is painting in red. It is left up to you to create appropriate interfaces to allow the user to select a colour and brush (they need to input either a *SQUARE* or *ROUND* brush, and a width in pixels) and return these values to the main activity.

Persistence

FingerPainterView takes care of persisting its own internal bitmap state if it happens to be destroyed – it saves the current picture to a cache file – however the state of other components, for example the current colour and brush selection are not persisted. You should handle the relevant activity lifecycle events for your activities to ensure a coherent user experience if, for example, the device is rotated.

Implicit Intent

Finally, as in the task diagram above, the user should be able to select an image in *another application* and choose to open it with our FingerPainter app. To support this our application should respond to *implicit Intents* that conform to something that we can open – in particular the user attempting to *view* a file that happens to be an *image*.



Note that you can load files onto the emulator by downloading via the in-emulator web browser, pushing files onto the sd card via the Android Device Monitor tool from Android SDK, or on the command line using **adb**. Downloaded files are stored in /sdcard/Download/ on the device.

```
zahn:platform-tools pszmdf$ ./adb push cats.jpg /sdcard/Download/cats.jpg
cats.jpg: 1 file pushed. 21.6 MB/s (251024 bytes in 0.011s)
zahn:platform-tools pszmdf$
```

The aim here is to, when browsing through their files using the **Files** application, give the user the option of opening a file in our FingerPainter as opposed to other applications, such as the Picture Viewer. We should do this by registering our application as being able to receive the appropriate intents.

Add a second *intent-filter* to the main activity in the *AndroidManifest* – so that the application can either be started via the launcher or by an intent sent by another application.

This filter should accept intents with an *action* of *android.intent.action.VIEW*, *category* of *android.intent.category.DEFAULT*, and *data* of *mimeType* image/*

```
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="content"
        android:mimeType="image/*" />
</intent-filter>
```

The location of the image to load will be delivered to the main activity via its intent, with the URI of the image being included in the data field of the intent. FingerPainterView will attempt to load an image by URI when called from the onCreate method:

```
        myFingerPainter.load(intent.getData());
```

If successful, you can test this by opening an image from the *Downloads* app on the device, which should now give you the option of your app to open the image with, as well as other installed applications.

References

https://developer.android.com/guide/topics/ui/declaring-layout

http://developer.android.com/guide/components/activities.html

http://developer.android.com/guide/components/intents-filters.html

http://developer.android.com/reference/android/graphics/Color.html

http://developer.android.com/reference/android/graphics/Paint.html