Miguel Diaz

Comparison of Techniques within Android and OpenGL ES 2.0

CS 370: Undergraduate Research and Reading

Advised by Dr. Alison N. Norman

Table of Contents

Introduction

Wernher von Braun, claimed by one NASA source to be “the greatest rocket scientist in history”, once said “Research is what I’m doing when I don’t know what I’m doing”. This brings in a refreshing view of how to view curiosity and the roadblocks one must face to find answers. Computer scientists themselves can recollect memories of the struggles of compiler errors that come with learning a new software development kit, and this is the case for developers out to learn Android development and graphics programming.

However, even in exploring this curiosity, one may run into even more questions. Even though a reference manual or guide may choose to recommend one method over another, is there no such case in which the unused method is more efficient than the recommended method?

One can run tests to answer such questions after answering what ‘efficient’ means in this case. Efficiency, in the following set of experiments, will refer to the old age battle of space versus time. How much space will be saved by using one method over another? What is the time cost of using one method versus another? Additionally, code clutter will also be compared where appropriate.

Comparison of Inflating Layouts versus Programmatic Creating Layouts

INTRODUCTION

Constructing layouts for Android apps can be done using various methods. However, two of the recommended approaches for layouts involve either inflating from an XML document or by programmatically creating a layout.

A short example is as follows.

Suppose one would like to display “Hello, world” to the screen of an Android app.

One could create an XML document, named using lowercase letters and numeric characters, that would contain the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@+string/helloworld" />
</LinearLayout>
```

And, in a separate document, be it strings.xml or a custom XML document, one should include the following XML instance.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="helloworld">Hello world</string>
</resources>
```

Then, one may then use setContentView() method in an Android application’s onCreate() method to inflate this all. One will see “Hello, world” displayed at the top. One line of code will achieve this.

Alternatively, one can use several lines of code to recreate this. In the Android application’s onCreate() method, one can declare the following lines.

```
LinearLayout example = new LinearLayout(this);
TextView helloworld = new TextView(this);
helloworld.setText("Hello world");
example.addView(helloworld);

 setContentView(example);
```

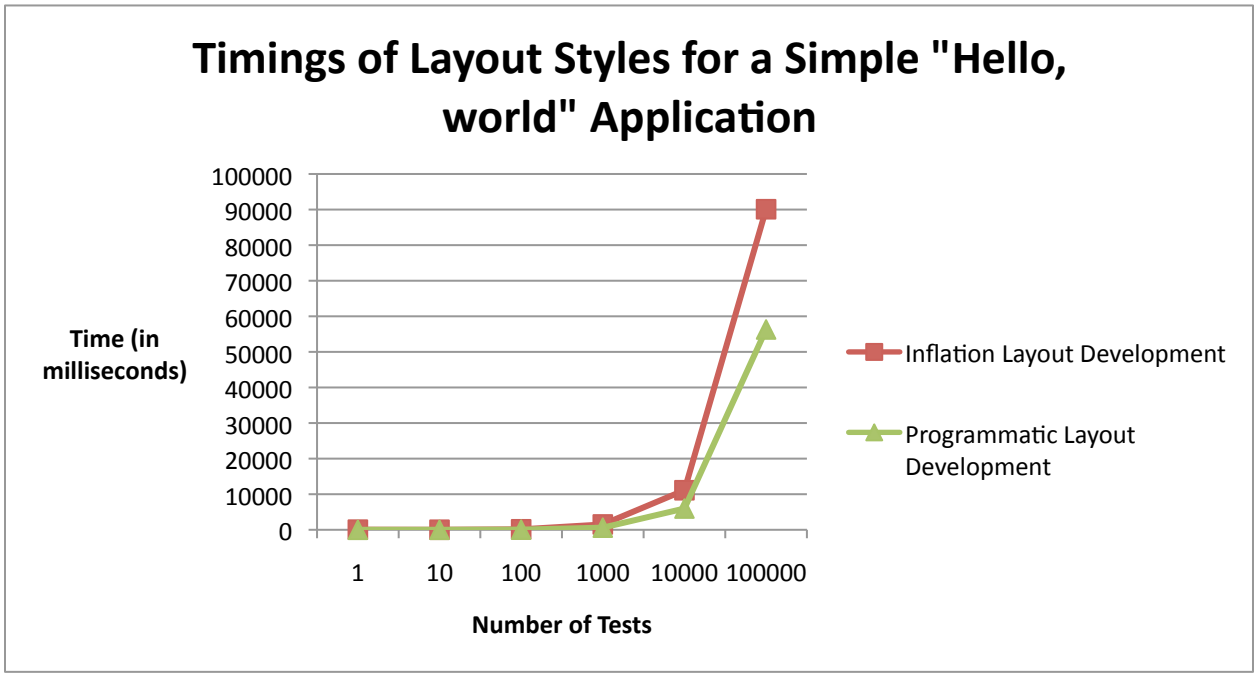
QUESTIONS

The opportunity to choose between inflation through XML and programmatically designing layouts appears to a new Android developer to be more of a developer’s choice than one that should be recommended.[1]

However, a natural conjecture is why such a recommendation is made.

To test, similar layouts will be implemented using both styles, programmatic layout development and inflation layout development. The screen will be cleared and redisplayed a set number of times and timed.

EXPERIMENT 1 – SIMPLE "Hello, world" TEST



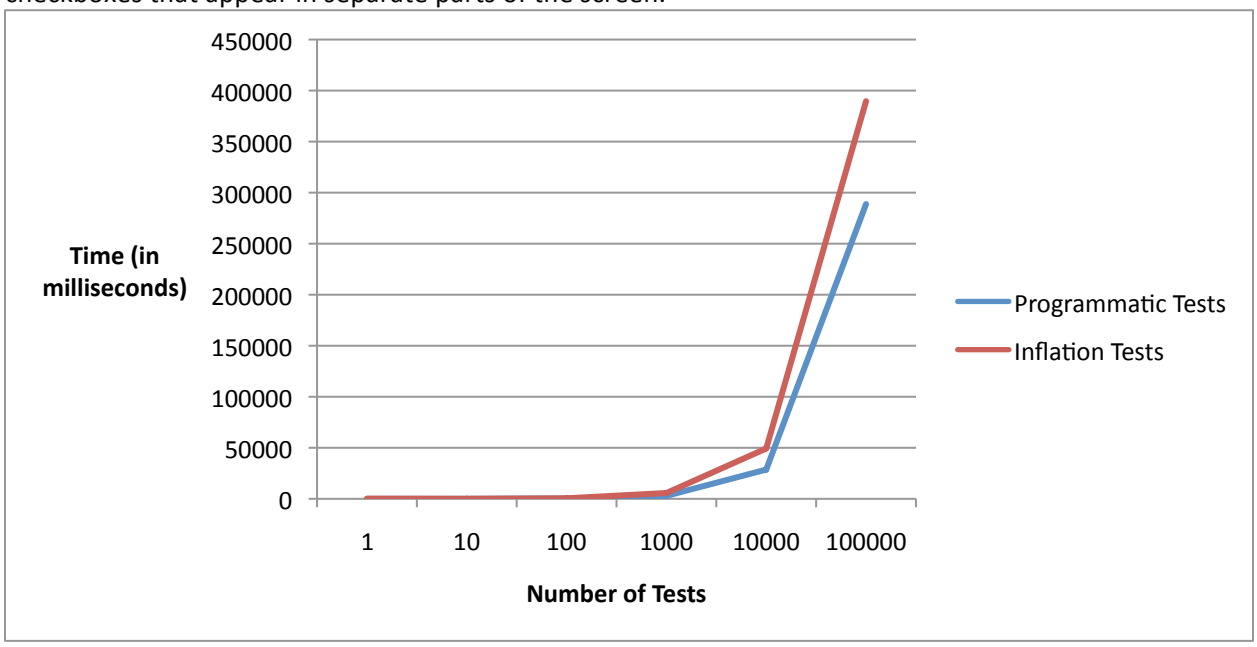
The graph shows that programmatically developing layouts, given nearly a thousand tests, begins to become a more time efficient procedure than the recommended inflation layout development style.

Number of Tests	Inflation Layout Development	Programmatic Layout Development	Time difference (in milliseconds)	Percent Difference
1	36	0	36	N/A
10	14	5	9	155.56%
100	136	48	88	154.55%
1000	1497	600	897	166.89%
10000	11113	5975	5138	216.29%
100000	90095	56299	33796	266.58%

The table shows the percent difference and, in general, the more tests generated, the more programmatic layout development becomes a more time efficient process.

EXPERIMENT 2 – A REAL LAYOUT

Code is written to test a slightly more complex example that consists of a button and two checkboxes that appear in separate parts of the screen.



Inflation tests run slower than programmatic tests do;

CODE CLUTTER

However, code clutter can easily be inspected to be 1 line of code to inflate a valid XML instance and 22 lines of code, excluding blank lines, to create the same layout programmatically.

CONCLUSION

One can predict that, for a larger project that includes many activity screens and different user interfaces for each screen, that inflating XML would be a much better idea unless one plans to create 100,000 layouts consistently, which for an app, is never the case. The average application uses well below fifteen layouts, and the time difference between each layout style is a few milliseconds.

Comparison of Loading Prerotated Images versus Calculating Rotations via Matrices

INTRODUCTION

Animating the images to present different types of scenery is a frequently done. For example, one could imagine playing a video game on an Android device such as Tetris; each block piece is rotated when a button is pressed.

QUESTIONS

A developer has two options for presenting rotating images.

The first option is to have images rotated manually via an image editor program and saved onto disk. The rotating images can then be used in the project and displayed when necessary.

The second option is to use only a non-rotated version of an image and rely on the android.graphics library to rotate the image via the use of matrix calculations.

The first option will most likely use four times as much space, and the second option is most likely to use more processor time to calculate the rotations.

The benefit of such a test can help Android developers whose applications rely heavily on images to consider tradeoffs between precalculating and calculating rotations of images.

TESTS

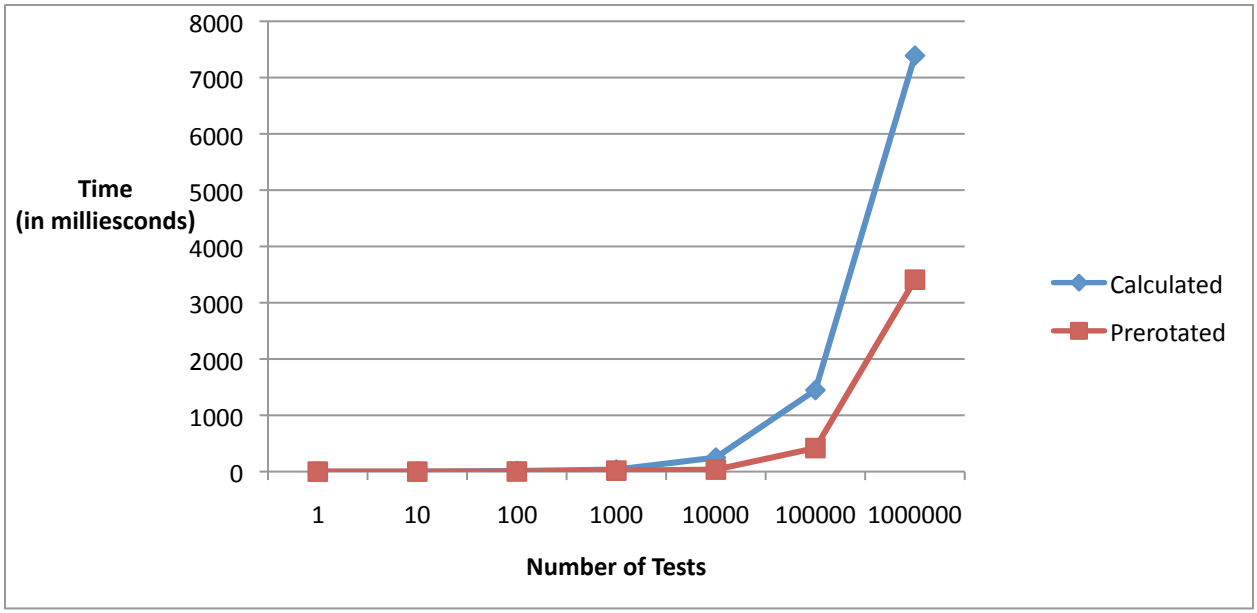
Code is set up that loads up a timer and loads the first image. Then, upon the increment of a counter, the next image that coincides with a 90 degrees clockwise rotation is loaded. This is repeated several times and timed.

Code is then set up to load an image at its 0 degrees rotation. Then, a Matrix object from the android.graphics library is used to calculate a 90 degrees clockwise rotation. The result is saved onto the instance of the image and displayed.

In the first case, the size of the files used will be the number of rotations times the image, so 4 * 40 KB = 160 KB. One can imagine that, for a smoother rotation, more intermediate rotations between the 90 degree angles would be used. Each 90 degree separation would have the same number of intermediate rotations, so one could add in four times the number of intermediate rotations to the previous result of 160 KB. The equation becomes

totalRotations = 360/90 * (1 + intermediateRotations) = 4 * (1 + intermediateRotations)
sizeOfSingleFile * totalRotations = size of all files used

RESULTS



Number of Tests	Calculated	Prerotated	Time difference (in milliseconds)	Percent Difference
1	0	0	0	N/A
10	1	0	1	N/A
100	12	2	10	120.00%
1000	37	17	20	185.00%
10000	250	36	214	116.82%
100000	1448	418	1030	140.58%
1000000	7390	3409	3981	185.63%

Rotating thousands of images begins to take a larger and larger toll on performance than would loading prerotated images would. Given that only four rotations are done, one can imagine a lengthier time difference if more rotations would be done to achieve smoother appearing animations.

CONCLUSION

For a small number of rotations, one can either calculate or precalculate images and have images be displayed in a few mere milliseconds. However, once the rotations reach well into the thousands and millions, and if enough space can be spared, a developer should begin to think about loading rotations that are frequently used.

Comparison of Using One Activity versus Multiple Activities

INTRODUCTION

Android applications tend to use more than one user interface display. A mobile application, for example, might allow a runner to track their mileage. This application might, for example, have one screen that (A) displays options to start a workout or to (B) display data for prior workouts and each option leads to a new screen that displays the appropriate information. These new screens may also have options.

Screens displayed have a one-to-one relationship with an Activity instance. Each activity instance is assumed to have a layout defined in an XML document that handles the contents of the user interface. However, if a developer so chooses, a screen can have a many-to-one relationship with an Activity instance if the developer has the Activity instance hold responsibility for the loading and destruction of user interfaces used.

To display each screens, a developer has two options: (1) create a new instance of Activity that contains handles each new screen's user interface, data variables, or (2) use the same instance of Activity and simply create the new screen by removing the old screen's View objects, saving any relevant information, and displaying the new screen's View objects.

When a new activity is created, the old activity's onPause() method is called, placed on the activity stack, and the new activity's onCreate() method is called. When the new activity is done (usually via a user hitting the a dedicated "Back" button on an Android device), the new activity's onDestroy() method is called, the last activity on the activity stack is popped, and the last activity's onResume()'s method is called.

QUESTIONS

By assuming the responsibility of managing user interfaces, a developer can avoid relying on Android's onCreate(), onPause(), onResume(), and onDestroy() methods to manage Activity() instances and thus, user interfaces.

The choice for a developer to manage user interfaces within the same Activity() instance to avoid creating multiple instances will be analyzed. In particular, the time efficiency of using the same Activity() instance will be compared. In addition, code clutter involved in being held responsible for the creation and destruction of user interfaces will be inspected.

TESTS

Experiments emulate a user executing an application and, from the application's first screen, pressing a button that would create a new screen with a different user interface.

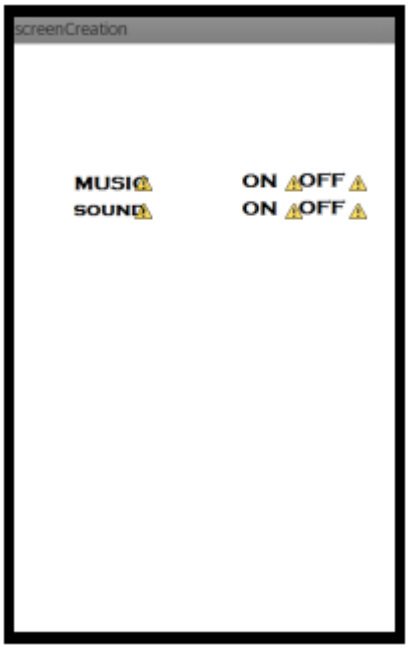
This experiment's first screen is an Android video game's typical home screen with options to start the game, see high scores, and change settings. Pressing the "SETTINGS" button would display options to prompt a user to change settings, which includes starting and stopping music.

One of the experiments is where the developer removes a screen, starts up the next screen, changes the options from on to off, and then returns back to the home screen. Each cycle of this is considered one test.

Another experiment tests for Android to change from the main screen to the next screen via an intent, which calls the main activity's onPause() method and then the setting screen's activity's onCreate() method. The settings screen's activity would have its screen loaded and then call finish(), which calls this activity's onDestroy() method followed by the main screen's onResume() method.

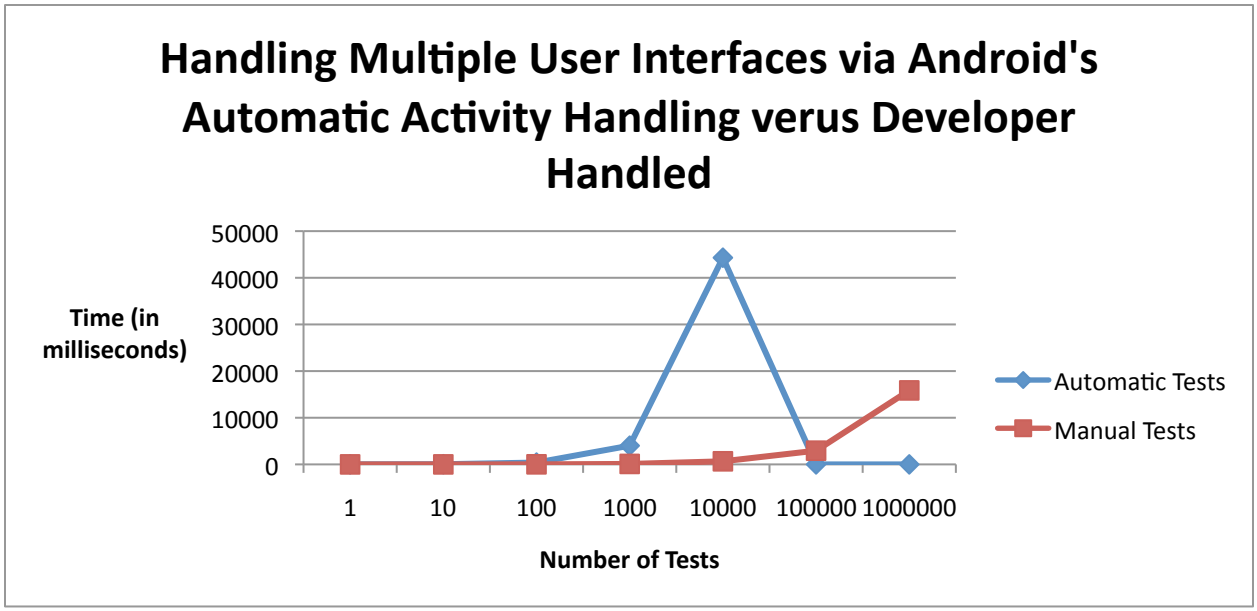
The two screens emulate a video game's typical main screen and options screen and are as follows.

Main Screen



Settings Screen

RESULTS



Tests could not be run for creating multiple activities past 10,000 test cycles because the time would begin to take upwards to 30 minutes to an hour.

RESULTS

Number of Tests	Automatic Tests	Manual Tests	Time difference (in milliseconds)	Percent Difference
1	8	0	8	N/A
10	37	0	37	N/A
100	393	16	377	2356.25%
1000	4012	100	3912	3912.00%
10000	44306	663	43643	6582.65%
100000	N/A	2939	N/A	N/A
1000000	N/A	15858	N/A	N/A

The time difference for creating switching between 100 activities can be tolerated by a user. 100+ activities can be the case for video games made for Android where there are several maps an adventurous elephant would want to travel to.

These small test cases also neglect additional effects that may help the user interface's design, such as orientation, sizing of images, or having special effects. Code clutter can easily become the case for a single Activity instance if each screen requires several features, and if a programmer choose to manually construct such a design, the code clutter can bloat easily.

CONCLUSION

For small Android apps that tend to be at most 10 screens, a developer would be best off using a separate Activity instance for each screen since the time difference is a few, unnoticeable milliseconds. Best of all, the user can take advantage of “back” and “home” buttons, and each Activity can have its own onPause() and onDestroy() methods run.

Comparison of Streaming versus Loading of Sound Clips

INTRODUCTION

Android applications can further engage a user through music and sound clips. Android has two classes devoted to these tasks. SoundPlayer and MediaPlayer are very well documented and each can play a variety of files, including MP3, WAV, and OGG. However, caution should be taken as Mario Zechner, author of “Beginning Android Games”, warns "Small sound effects fit into the limited heap memory an Android application gets from the operating system. Bigger audio files containing larger music pieces don't."

SoundPlayer has methods that are meant to take small sound clips, load them onto memory, and play them whenever the play() method is called. The sound clip can be stopped at any point using stop() and readied for reuse via the method prepare(). The maximum file size capable of being loaded is not documented.

MediaPlayer has methods that are meant to take file descriptors of an audio file, read portions of the file descriptor's sound clip, decode the portion, and pass on the decoded portion to hardware responsible for playing sounds. A developer only needs to worry about passing the file descriptor and calling methods appropriate to the management of the sound clip, such as .play(), .stop(), and additionally, .setOnCompletionListener(), which runs when the sound clip is done being laoded.

This section seeks to test this warning to understand the possibilities and limitations of sound and music in Android applications.

QUESTIONS

The time taken to prepare for a SoundClip() instance and, separately, a MediaPlayer() instance will be compared.

The time taken to play a SoundClip() instance and, separately, a MediaPlayer() instance will be compared.

The code complexity will also be compared.

Amount of memory used in Android will not be compared.

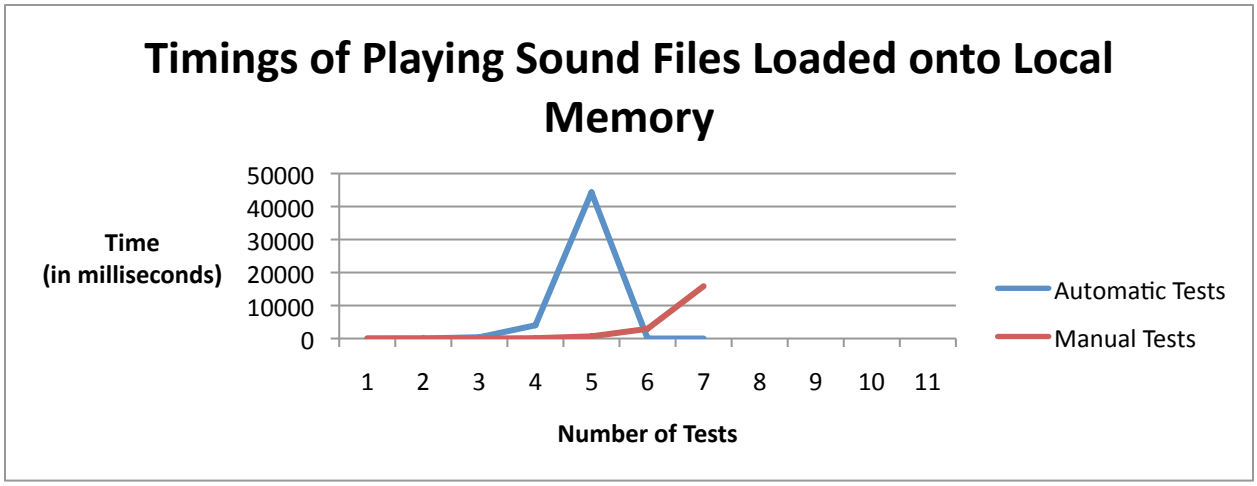
EXPERIMENTS

Two sound clips are used. One sound clip is called "Powerup.ogg" that lasts .35 seconds and is a 7 KB file. The other sound clip is "BachCmajorMinuet.ogg" that lasts 30.0 seconds and is a 462 KB file.

SoundPool() instances will be used for tests that load the sound clips into memory and play them a set number of times. Times for setup and playing are compared.

MediaPlayer() instances will be used for tests that read portions of the sound clips at a time and play them a set number of times. Times for setup and playing are compared.

RESULTS



Cache effects are not considered but may be used as one would expect each test to take as long as the length of the sound clip plus any previous tests. However, this is not the case, and in fact, the timings do not show the playing of the sound clip. Otherwise, the 462 KB file, which is a 30 second clip, would have taken minutes. Then, there are effects that are not discussed taking effect. However, these results do show that loading onto local memory can be useful. Reusing the sound clip took less time each time, thanks to effects happening behind the scenes in Android.

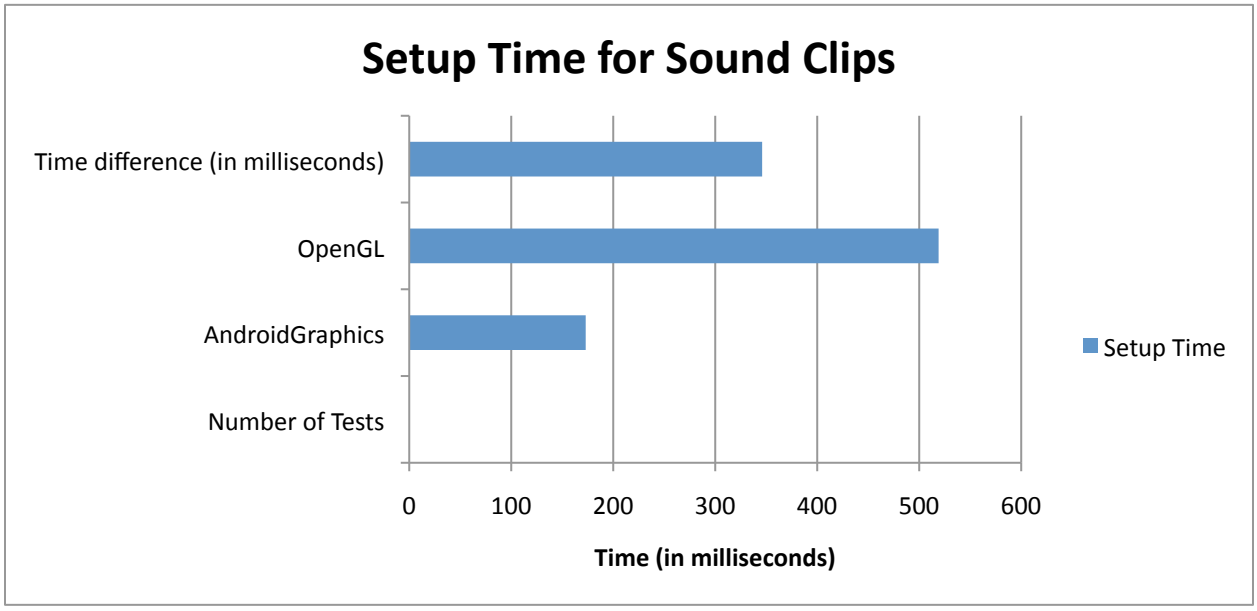
One can also see that loading clip 100 times larger than another does not take considerably more time.

Number of Tests	7 KB file	462 KB file	Time difference (in milliseconds)	Percent Difference
1	0	0	0	0%
2	6	21	15	250.00%
3	16	82	66	412.50%
4	29	158	129	444.83%
5	59	284	225	381.36%
6	91	447	356	391.21%
7	140	556	416	297.14%
8	264	679	415	157.20%
9	328	795	467	142.38%
10	377	892	515	136.60%
11	443	1108	665	150.11%

The time difference to play a sound clip 100 times larger than the other was never more than three times the smaller clip's play time.

The results for streaming sound files were inconclusive. There is no way for a developer to tell whether a streamed object has finished playing.

However, set up times can be measured for all cases and are as follows.



The results show that setting up sound clips for a stream, regardless of size, makes more use of processor times than does loading a sound clip onto memory.

	Memory	**Stream**	**Time difference (in milliseconds)**	**Percent Difference**
7 KB	3	57	54	1900.00%
462 KB	11	80	69	727.27%

Code clutter is negligible. Six lines of code what is needed for a SoundPool() instance setup whereas 4 lines of code are needed to setup a MediaPlayer() instance. Four lines of code are needed for the readying, playing, stopping, and unloading of the SoundPool() instance whereas six lines of code are needed to do the same for a MediaPlayer() instance.

CONCLUSIONS

If music clips can be kept to 30 seconds or smaller, loading the file may be worth the investment of memory. Memory is cheap and widely available and will continue to do so for the foreseeable future. The release of Android 2.1 allowed users to store Android applications on external storage, and as of August 2012, http://developer.android.com/about/dashboards/index.html shows that over 99.3% of users use Android 2.1 or higher.

Comparison of Rendering Similar Graphics using AndroidGraphics and OpenGL ES 1.1 Libraries

INTRODUCTION

OpenGL ES is known to be a subset of OpenGL, an API known for the development of hardware accelerated graphics. However, OpenGL ES is exclusively for open devices. Like OpenGL, OpenGL ES aims to be compatible across all platforms, including Apple, Android, and Microsoft products. OpenGL ES is also known to be used for graphics intensive software, such as those found in flight simulation, virtual reality, and state of the art video games such as Angry Birds and World of Warcraft. The AndroidGraphics library cannot support the mechanisms required to create the visuals found in these applications.

However, the AndroidGraphics library does have its uses and whole applications with appealing visuals have been written using only this library.

QUESTIONS

OpenGL ES is a powerhouse and can create the very same games. However, the cost of using OpenGL ES to create a similar game may be hefty.

For the purposes of a simplistic experiment, OpenGL ES 1.1 will be used.

The cost of using OpenGL ES versus the AndroidGraphics library to create a set of similar images will be analyzed. One can predict that OpenGL ES will always be more costly than AndroidGraphics since OpenGL ES constantly calculates matrices and the state of various objects whereas AndroidGraphics need only worry about a Canvas() objects and any associated Views which are often manually managed.

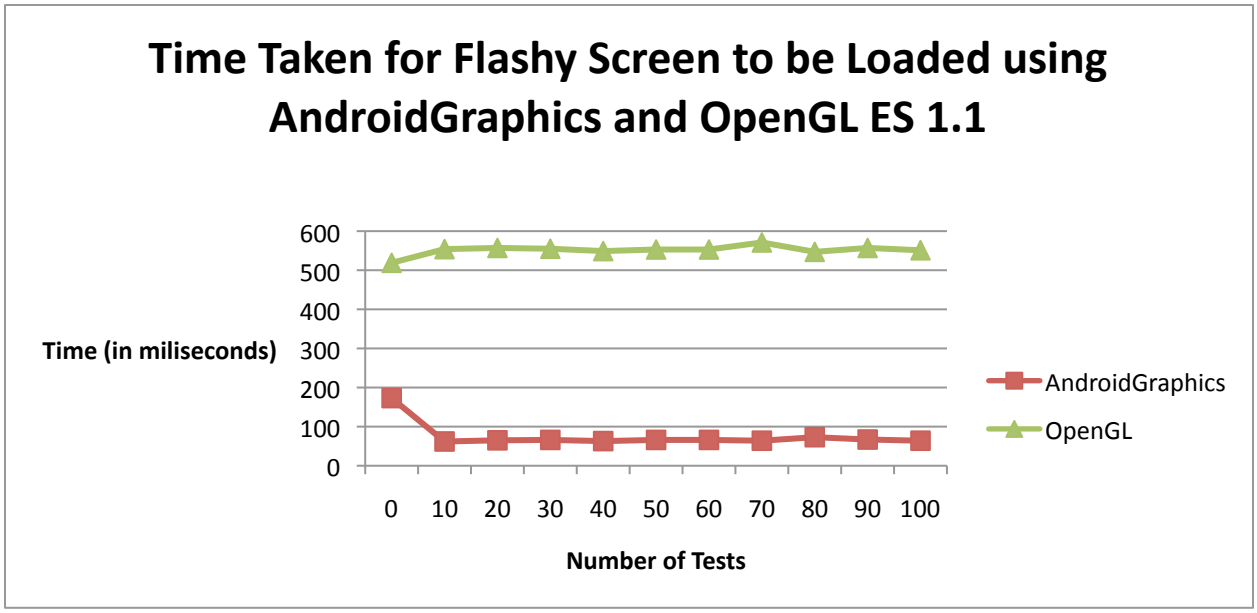
EXPERIMENT 1 - FLASHY SCREEN

Isolating the costs of using AndroidGraphics is achieved by only starting the timer on the first call to onDraw() that utilizes the Canvas() object in the AndroidGraphics library. This will ensure that Android's creation of the Activity() instance and other background events are not timed.

Isolating the costs of using OpenGL ES 1.1 is achieved by only starting the timer on the first call to onDrawFrame() in the Renderer() instance. This ensures that any overhead causes by setting up Android's Activity() or loading OpenGL ES is avoided.

A screen that picks random values for red, green, and blue values and then loops between the limits of the color values (256 for a Canvas() object, 0.0f to 1.0f for the Renderer() instance's GL10 instance that handles color values) is created. Every 30 times either onDraw() or onDrawFrame() is called, for a Canvas() instance and Renderer() instance respectively, the screen is allowed to be updated. The time interval between each refresh of the screen is written to external storage.

RESULTS



Number of Tests	AndroidGraphics	OpenGL	Time difference (in milliseconds)	Percent Difference
0	173	519	346	200.00%
10	62	554	492	793.55%
20	65	557	492	756.92%
30	66	555	489	740.91%
40	63	549	486	771.43%
50	66	553	487	737.88%
60	66	553	487	737.88%
70	64	571	507	792.19%
80	73	547	474	649.32%
90	67	557	490	731.34%
100	64	551	487	760.94%

For the same effect in all cases, OpenGL ES 1.1 takes up to 800% longer than AndroidGraphics.

CONCLUSION

For simple effects that can be created by using either AndroidGraphics or OpenGL ES 1.1, AndroidGraphics is the way to go. For effects that are slightly more complex such as drawing a simple 2D map with animated characters, one might expect AndroidGraphics to still be faster. This can be tested.

SOURCES CITED

[1] Meier, Reto. “Professional Android 2 application development”. Wiley Publications, Indianapolis, IN. c2010

[2] Zechner, Mario. “Beginning Android Games”. Apress. New York, NY. 2011.