Graphics with limited resources

Ludvig Jonsson
ludjon@kth.se
Tim Johan Malmström
tjma@kth.se

Degree Project in Computer Science, First Level, DD143X
Supervisor: Mads Dam
Examiner: Mårten Björkman

Bachelor of Science Thesis
Stockholm, Sweden 2012
Abstract

This paper aims to investigate what is being done to improve the use of graphics on mobile devices with focus on the Android platform and especially Android-based smartphones. Today most people use a smartphone in their everyday life and much of the intended use of traditional personal computers are moving towards mobile devices. Because of this, making the most of the limited system resources is a highly relevant issue.

The subject is divided into three parts. First off, it investigates how graphics rendering is done on the Android platform. The second part is a comparison of the available rendering options on Android and their respective resource usage. The last part consists of a practical test of the different rendering methods and results analysis.

The conclusions that could be drawn from this was that there is no method that is optimal for all situations, all available rendering options have different uses and advantages. The choice of method should be based on what kind of graphics are being rendered and how heavy the rendering instructions are.
# Contents

## Contents

### Introduction 1
- Preface ......................................................... 1
- Purpose ......................................................... 1
- Research focus ............................................... 2
- Problem statement ............................................. 2

### Background 4
- History .......................................................... 4
- Resources ......................................................... 4
  - Platform ......................................................... 5
  - Central Processing Unit (CPU) .............................. 5
  - Graphics Processing Unit (GPU) ........................... 6
  - Random Access Memory (RAM) ............................. 6
- Android graphics APIs and rendering ....................... 7
  - Canvas API ..................................................... 7
  - OpenGL ......................................................... 7
  - Renderscript ................................................... 8
- Rendering optimization techniques ......................... 9
  - Hardware and software rendering ........................... 9
  - Android and hardware acceleration ........................ 9

### Discussion 11
- Discussion from a theoretical point of view ................ 11
  - Differences between the APIs ................................. 11
- Hardware and software rendering ............................. 12
- Rendering situations .......................................... 13

### Testing method 14
- The testing application ....................................... 14
- The testing device ............................................. 15
- Benchmarking ...................................................... 16
- Test cases ......................................................... 16
Expected results ........................................... 17

Results analysis ........................................ 18
  Implementation using the different APIs .......... 18
  Test results ........................................... 19

Conclusions ............................................. 22
  Improvements being done to the graphics rendering . 22
  Android rendering methods ........................... 23

Abbreviations ......................................... 24

Test data ............................................... 25
  Canvas with software rendering ...................... 25
  Canvas with hardware accelerated rendering ........ 25
  OpenGL ................................................. 26

Bibliography .......................................... 27
Introduction

Preface

This document was written for the course Degree Project in Computer Science, DD143X(dkand12) at the Royal Institute of Technology(KTH). It was written by Tim Malmström and Ludvig Jonsson who both at the time of writing this documents were third year students at the computer science program at the Royal Institute of Technology in Stockholm. The document was written with the purpose of investigating what is being done to improve the use of graphics within mobile devices with limited resources.

The supervisor for writing the document was Mads Dam.

The writing has been divided between the authors as follows:
Introduction - Both
Background - History - Ludvig
Background - Resources - Tim
Background - APIs and rendering - Ludvig
Background - Optimizations - Tim
Discussion - Differences - Ludvig
Discussion - Hardware and software - Tim
Discussion - rendering situations - Ludvig
Testing method - Tim
Result analysis - Tim
Conclusions - Both
Layout and Design - Both
Implementations - Tim

Purpose

With the areas of use of personal computers constantly growing the demand for those systems being more mobile has also grown. This has put the pressure on the manufacturers to create smaller and lighter products that have the same function-
INTRODUCTION

ality as the bigger ones. This developed the problem with the new systems having less resources which has put the pressure on the developers to create systems that use these limited resources as efficient as possible.

When the smartphone and the tablet was introduced to the world this was one of the manufacturers biggest problems. To get people interested in change and trying something new the new product has to have its clear advantages over the previous one. As Steve Jobs, co-founder of Apple Inc. said when introducing the iPad in 2007:

‘Do we have what it takes to establish a third category of products?
The bar is pretty high. It has to be far better at doing some key things.
We think we have the goods. Our most advanced technology in a magical and revolutionary device at an unbelievable price.’[15]

Back in 2007 the iPad’s limited resources kept it from being anything like the tablets of today in performance and usability, but it was the breakpoint in performance that made the tablet a popular product with many tablets released earlier being big flops.

The use of this kind of limited system resources is what this paper is going to focus on. In terms of graphics, what is being done and what can be done to use the resources as efficient as possible?

Research focus

The name of our subject and also the title of this document is 'Graphics with limited resources'. Since this is quite wide, we have decided to limit our focus to smartphones. We find the subject of smartphones interesting and highly relevant since most people in the modern world are using one in their everyday life.

We have also decided to focus on smartphones running the Android platform since the Android phones are dominating the smartphone market having 50.9 % of the total smartphone sales according to Gartner’s analysis of global smartphone sales in Q4 2011.[11]

Problem statement

The problem statement is as follows:

- What is currently being done to be able to use the system resources as efficient as possible?

- What is the differences between the different rendering methods used in Android and their resource usage?

- What Graphics component used in Android is best in terms of resource usage in which situation?
INTRODUCTION

With the first question the goal is to understand what techniques are used to make the resource usage more efficient and why. The second question will be more of a research and comparison between the different rendering methods. The third question is going to be answered in two different parts. First the rendering methods will be compared from a theoretical point of view and then the theoretical conclusions will be tested through testing the methods using benchmarks to compare their performance. Following this the test results will be analysed hopefully leading to conclusions that can answer the problem statement.
Background

History

If you compare a modern mobile phone to one released fifteen, ten or even five years ago, you can clearly see that the development is progressing very fast. They still serve the same basic purposes, like being able to make phone calls and send text messages when you are on the go. With the development of the hardware performance and the phones being more advanced in general, the uses has grown from just a few to what seems like an unlimited amount.

You can tell some of the differences just by comparing the looks of an older mobile phone and a modern smartphone. The older monochrome displays and physical buttons have been replaced by much larger, touch-sensitive color displays. Since the older phones was mostly used to make phone calls and send text messages, the hardware was not and did not really need to be that powerful. With the new, larger screens supported by the new more powerful hardware, the differences in functionality between the smartphone and the standard personal computer have become virtually non-existent.

Earlier, the different functions of the mobile phones have been integrated in the operating system and all the different phone manufacturers all had their own platforms. Nowadays almost all smartphones are based on an already established platform like Android or iOS. The big advantage with this is that when you release an Android-based smartphone (for example), there already exists a couple of million applications which can be installed to extend the functionality of the device.

The more powerful hardware makes it possible for these applications to be graphically heavy, but since the smartphones are limited in aspects like physical size and battery life, it’s still very important to use the hardware as efficiently as possible in the graphics rendering process to make the most of it.

Resources

The graphics rendering process is a process that requires the use of several different system resources. What resources are being used and how they are being used has varied a lot during the development, from having distinct RAM and GPU that does not affect the performance of the rest of the system whatsoever, to the more
advanced cooperations between GPUs and the CPUs.

The graphics rendering in modern personal computers are developing in a direction that increases the separation between the graphics processing and other processes. This is mostly since the space required for cooling and for the processors is not a problem. This however is not the case with mobile devices. Because of the limited space within a mobile device most of the newly developed devices contain a CPU with an integrated GPU which means that the graphics processing share the same resources as other system processes. This also makes efficient graphics rendering an even more important part of the applications.

The most important system resources used by the graphics rendering is the CPU, GPU, RAM and the platform which isn’t really a resource per say, but more of a controller for the resources. Even though the CPU and GPU use the same resources they are separated since separating CPU requests from GPU requests makes the process handling more efficient.

Platform

A platform is both a sort of hardware architecture and a chipset containing a software framework.[13] In the case of a mobile device the platform usually contains all the hardware of the device since you rarely have any extension ports or in other ways any possibility to change any hardware of the device. This is something that makes the system more stable and reliable than a personal computer since there is less that can go wrong when you don’t have the possibility to switch hardware.

The software framework put on the chipset is in Android devices parts of operating system and some basic resources like timers, boot instructions and interrupt controllers. It also contains the instructions and controllers for the hardware components.

The reason why the platform is relevant in this case is because the platform controls the hardware resource usage and thereby how much and what resources can be used by the processes. When testing the performance of different rendering methods the tests could show totally different results on different platforms since they could use the system resources with different efficiency or possibly just with different priorities.

The platform that we are going to perform our tests on is the one used in the Android phone Samsung Galaxy Nexus. The platform is called OMAP4460 and is developed by Texas Instruments.[10] We chose this platform because the Samsung Galaxy Nexus is one of the most recently released Android smartphones. The platform in the Samsung Galaxy Nexus uses the operating system Android 4.0.

Central Processing Unit (CPU)

The central processing unit is the part that executes instructions on a system, it takes instructions in assembler code and executes them, since this is the component that performs all operations the CPU is very important for the graphics rendering.
BACKGROUND

The biggest problem with the CPU within mobile devices is that it generates a lot of heat. This often forces the manufacturers to limit the processors capabilities to be able to use them in the smaller devices.

In our testing the device will use a 1.2 GHz dual core CPU, according to Texas Instruments the processor used in the OMAP4460 has the capacity to reach speeds up to 1.5 GHz although because of the problem with cooling in a mobile device Google has chosen to lower the clock frequency.

Graphics Processing Unit (GPU)

A GPU is a electronic circuit designed to be able to quickly alter memory and perform instructions in a way that accelerates processing of graphical components and building images in a frame buffer that is intended to be shown on a display. As mentioned earlier there are several different kinds of GPU types. The ideal type performance-wise is the one where the GPU runs a separate processor with the sole purpose of executing graphics instructions. This gives the graphics processor better image rendering speeds than the more general CPU. The kind of GPU that is going to be used in our testing will be an integrated GPU.

The integrated GPU does not have a processor of its own but is instead more of an optimizing unit that optimizes the instructions to calculate graphics and then relies on the CPU for executing them. Since GPUs have come a long way this provides enough performance to calculate graphics in most situations. There are many advantages with using integrated GPUs - it is more cost efficient, more space efficient and consumes less power and the only downside is the performance.

Random Access Memory (RAM)

The random access memory (RAM) is a memory which is made for being able to read and write from all memory positions quickly. RAM is therefore used as a temporary data storage for active processes as data variables and states, any data that needs to be accessed quickly. For more long term storage, devices use a storage memory that is not as quick but has room for much more data.

In mobile devices the RAM is usually divided into two parts, the part that is accessed and used by the operating system and the part reserved for other processes. This is mostly so that the more stable OS can not be affected by badly written code from applications, and if many processes are running simultaneously allocating a most of the memory, this does not affect the performance of the operating system.

The RAM is a very important resource for the graphics processing since the already existing bitmaps used for the rendering as well as the draw buffer are stored there for fast access.
BACKGROUND

Android graphics APIs and rendering

When speaking in terms of computer graphics, rendering refers to the process of generating (or simply just drawing) the graphical components of a computer program (in our case an Android application).

When coding an application for the Android platform, there is a couple of different rendering methods to choose from based on what type of graphics to render. API is short for Application Programming Interface and is a specification of functions and object classes with the purpose of being an interface between software components [16]. The purpose of this section is to go through the different rendering methods, their advantages and disadvantages.

Canvas API

The Canvas API is a very simple tool for rendering graphics on Android. To be able to understand how the Canvas API works, you first need to understand what a View object is and how it is used in the Android development process. The View object is the basic building block for user interface components when developing a standard SDK application for Android, it handles the graphics rendering and user input.

A Canvas is drawn by calling the onDraw method of the View with the Canvas object as a parameter. By calling methods of the Canvas class, such as drawText, drawCircle and drawPicture, you can decide what graphical components should be rendered and where they should be positioned on the screen.

Since Android 3.0 when the support of hardware acceleration was added, the overall performance of the Canvas API has improved. By enabling hardware acceleration, you can let the GPU handle the graphic instructions and thereby offload the CPU.[6]

The Canvas API is the standard way of rendering graphical components of a GUI application on Android. It is a great tool for simple graphics, but you have to look at other methods if you want to include more advanced graphics in your application.

OpenGL

OpenGL (short for Open Graphics Library) is a cross-platform graphics language for producing 2D and 3D graphics. It is produced by Silicon Graphics and is the most widely used graphics API of the computer industry.[17]

Android uses a subset of OpenGL called OpenGL ES (OpenGL for Embedded Systems). It is specifically designed for embedded systems like video game consoles and mobile phones, and is basically a stripped down version of OpenGL. [8]

Android supports OpenGL in two different ways - through the standard Android framework API and through something called Android NDK. NDK is short for Native Development Kit and it lets you build applications (or parts of applications) in
BACKGROUND

native code outside the Android Java VM (Virtual Machine) as opposed to building applications in Java on the SDK level.

When building applications with the NDK, you don’t have access to the usual GUI toolkit (View objects, events) of the Android platform, but with full access to OpenGL this is ideal for a few specific purposes (like porting applications written in OpenGL for another platform).

Using native code does not automatically increase the performance of the application, but it provides a more complex view of the application and lets you do exactly what you want with it. It also gives you the possibility of embedding corresponding native libraries into the application package file and thus the possibility to embed OpenGL.

The standard option is to use something called OpenGL wrapper functions. A wrapper function is basically a function whose main purpose is to call a second function in another framework or library. In this case, the wrapper functions are using something called JNI (Java Native Interface) to make calls down to the native OpenGL library. OpenGL wrapper functions makes it possible for the developer to use the OpenGL API at the SDK level. This means that you can write an application by using the SDK and thereby the usual SDK APIs and still use at least parts of the OpenGL library.

Renderscript

Renderscript is another graphics rendering API for Android, developed with the goal of bringing a lower level, high performance graphics API to the Android platform. An application using Renderscript is a standard SDK application running in the Android VM but with some external parts operating at native level. The external parts are written in Renderscript code (in the C99 language).

The application can be divided into two parts, the Android framework and the Renderscript runtime. The lower level Renderscript runtime is controlled by the higher level Android system that runs in the Android VM. The memory allocation is handled by the Android VM which binds it to the Renderscript runtime so the Renderscript code can read from and write to it.

By compiling the Renderscript native code at runtime, the application maintains the portability of the application and the benefits of native code.

With Renderscript, you can optimize the performance of your application by running graphics operations or other heavy computations on the native level (when they simply take too much time to run at the SDK level). Renderscript also has the possibility to, at runtime, determine the best way to run a particular operation to achieve the highest possible performance. It may decide to send the graphic operations to the GPU or decide that it’s better to split up the computations and send them to the multi-core CPU.

All this can be done without losing any of the functionality of the SDK APIs.
BACKGROUND

Rendering optimization techniques

After choosing a API to work with there are often several techniques that can be used to improve the rendering performance further. Often these techniques aren’t really ways of making any changes in the performance of the used resources themselves but instead more of a way to spread the resource usage between different resources and thereby improving the overall performance. The most relevant technique used in Android devices nowadays is what Google call 'Hardware Accelerated rendering' and is an option for the Canvas API.

Hardware and software rendering

The main difference between hardware and software rendering is how rendering processes are run. When rendering using software rendering all instructions are sent directly to the CPU without using any form of graphics hardware. As opposed to software rendering the hardware rendering runs all its rendering instructions through the graphics hardware. There are advantages and disadvantages of both methods.

The biggest advantage of using software rendering is that it is simple. Since the instructions are sent directly to the general purpose CPU they don’t have to be restricted by the more limited capabilities of the GPU. Those limitations in the GPU are therefore also the biggest disadvantage with using hardware rendering, they make the coding more complex. Another disadvantage of using hardware rendering is that the more complex instructions require the use of more RAM. But since the instructions are more specific in the hardware rendering the processor usually can execute them faster than in the case with software rendering.

So choosing between hardware and software rendering is a question of the balance between the usage of the RAM and the CPU. Since they both have their advantages and disadvantages the choice must be made for each case.

Android and hardware acceleration

Android has always used some kind of hardware acceleration[4] although before the 3.0(Honeycomb) release it was limited to the use of system graphics such as menus and the top scrollbar. Since the graphics in android devices has always been divided into different components called windows you could say that the handling of those windows was handled using hardware rendering while the content of those windows was handled by the software.

If content in any of the windows change the re-rendering is done using the software. However none of the other windows are redrawn, and the (if needed) recomposition of the windows is done using hardware rendering. Any movement of a window without changing the content of it as for example the sliding up or down of the menu is handled by the hardware.
BACKGROUND

This has before Android 3.0 also provided additional stability and security since no third party applications had full access to the hardware resources of the device. In Android 3.0 Google added the

\[
\text{<application android:hardwareAccelerated="true" ... >}
\]

attribute to applications which enabled the content of windows to be hardware accelerated as well as the window management. This was mostly made as a test to see if the system would run stable using the hardware acceleration and was therefore disabled as default. After seeing the improvement this made, Google decided to have the hardware acceleration enabled as default in Android 4.0 even though it is not yet usable for all of the drawing operations.

As problems regarding hardware acceleration usually manifest themselves as invisible elements, exceptions or wrongly rendered pictures\[1\] Android gives the developer the option to enable or disable hardware acceleration at four levels. Application, Activity, Window and View-level. However Google recommends a few ways to work around these problems using disabling hardware acceleration as a last resort.
Discussion

This chapter contains the theoretical discussion about the APIs from a programmers point of view and therefore contain both factual information about the APIs but also much about what factors are important to think about when using the graphical components of a Android system. There will also be some theoretical conclusions which will be tested and proved or disproved in the next chapter.

Discussion from a theoretical point of view

Differences between the APIs

In the previous chapter, four rendering methods were mentioned:

- Canvas API
- Renderscript
- OpenGL Wrappers
- NDK OpenGL

These rendering methods were of course not created with the exact same purposes in mind. The Canvas API is meant to be a simple tool for rendering the GUI components (like buttons and menus) and other simple graphics of a standard Android application. Using the more advanced options for this kind of graphics would just make things more complicated rather than improve the performance of the application.

So, the purpose of the Canvas API is to be able to render simple graphical components in a simple and straightforward way. The other options are meant to be used for more advanced graphics like 3D graphics, something that the Canvas API simply does not support. OpenGL (both with wrappers and the NDK option) and Renderscript is in the same category and were created for the same main purpose - to make it possible for the Android developers to go beyond what the Canvas API has to offer, both in terms of functionality and performance.

Since Android 3.0, when GPU acceleration was added to third party applications, the performance of the Canvas API has increased. When hardware acceleration is enabled, all rendering done at the SDK level is processed as OpenGL calls.
DISCUSSION

to the GPU. This should increase the overall performance when rendering heavy graphical components, but it won’t necessarily always increase the performance. This question is handled later in this chapter.

Both the OpenGL options come with some disadvantages. If you want to include high-performance advanced graphics in your application, the wrappers option probably isn’t the one to choose since the process of calling down from the SDK to the native level for every OpenGL call will not perform as good as native access to OpenGL. At the same time, if you choose to go with the NDK option with native access to OpenGL, you lose the portability of your application since the NDK compiles the application for the specific chip architecture you are currently working with. You also lose the GUI toolkit found in the SDK APIs, like the View objects and the events.

 Renderscript is supposed to solve the problem of having to choose between performance and portability. Renderscript offers performance similar to the performance of OpenGL with the NDK. This is accomplished by executing performance critical parts of the application on a native level as opposed to executing them in the Android VM.[3]

By compiling the Renderscript code at runtime, it solves the problem with native code and portability. This is different from the OpenGL options where you either have the whole application written in native code (with the NDK) or have the whole application running in the Android VM and just make calls to the underlying OpenGL API at native level through the OpenGL wrapper functions. You could say that Renderscript is a mix between these two in the sense that it offers both the portability of the OpenGL wrappers option (and the rest of the tools and APIs of the SDK) and performance equal to OpenGL in the NDK.

Hardware and software rendering

Since hardware rendering is considered as a optimization of the software rendering and therefore also enabled by default in the latest Android releases you could think that hardware rendering is simply faster and better than software rendering. This however is not always the case.

Since the hardware rendering uses a different model for the graphics rendering the instructions go through more stages of processing than they do in the software rendering model before actually being executed. This means that for simple rendering instructions where the graphics processing can not improve the runtime of the instructions hardware acceleration just delays the execution of the instructions.

So when the rendering instructions runs hardware accelerated they are being processed more before being executed. This also provides the effect of hardware rendering needing to allocate more memory than the software rendering would need to execute the same instructions.

To summarize this you could say that for most heavy rendering instructions the hardware rendering can improve the instruction runtime more than the actual processing takes which improves the overall performance. But if the instructions are
simple the rendering actually benefits from just running the instructions by software rendering. Hardware accelerated rendering needs to allocate more memory than the software rendering for both lighter and heavier instructions, although the amount of memory that differs between the two can often be considered to be negligible.

**Rendering situations**

The comparison of the different APIs clearly states that even if they do overlap each other in some ways, they still differ in their purposes and their advantages. Depending on what kind of graphics you want to render, you want to choose the most appropriate rendering technique for that specific situation to get the most out of the limited system resources. There is no technique that has the best performance in all situations.

When developing a standard GUI application with no additional graphical effects, the Canvas API with hardware acceleration inactivated is probably the best option. This is because the simple graphic instructions will probably not benefit from the hardware acceleration. It may even slow down the rendering process since the instructions go through more steps with hardware acceleration. As mentioned in the previous subchapter, hardware acceleration also allocates more memory.

If you need to have custom, more advanced 2D graphics in your application, Canvas API with hardware acceleration activated probably is the best option. When the graphics are heavy enough, the application performance benefits from the possibility to let the GPU handle all the graphic instructions and thereby offloading the CPU.

When you want to go beyond what the Canvas API has to offer in terms of performance and functionality, you have to go with either OpenGL or Renderscript. Since both these options give the developer the opportunity to run performance critical parts of the application on a native level rather than in the Java VM.
Testing method

As discussed in the previous chapter there are big differences between the APIs regarding performance within applications and quite specific cases for where to use which API. An important aspect in the choice is the complexity of the code used for the rendering in the different cases. To test this and to prove the thesis from the discussion chapter by seeing the resource usage of the different APIs, a simple application will be implemented to push the different APIs frame rendering to the max.

The testing application

The test case that is going to be run in the test is a simple graphics rendering application. Since the goal of the application is not to push the hardware to the maximum performance, but instead to show the basics of how rendering and hardware management is done by the different APIs.

The testing application will show ovals and squares of different colors and sizes as fast as possible. To keep the randomization of the shapes from stealing system resources from the rendering the first part of the testing is a program generating test data. It creates 1000 randomized shapes and stores them in a file read by the testing programs. This also makes sure that the APIs face the same level of difficulty in the rendering since they all render exactly the same frames during the testing.
TESTING METHOD

Figure 1. Some screenshots from the testing application

The test application will while running also log the memory usage and the number of frames rendered per second (FPS) during the testing process, to be able to make conclusions from the results of the test and for comparison.

The testing device

The testing device is as mentioned earlier a smartphone using the Android 4.0 operating system. The model chosen to work with is a Samsung Galaxy Nexus which was released in November 2011. Other than the fact that this phone contains state of the art hardware it is very relevant to the subject since the screen has a resolution of 720*1280 pixels.[5] The high resolution display makes the graphics rendering much heavier for the hardware than it was with the older displays.
Testing Method

Benchmarking

Benchmarking is the process of comparing performance metrics. The dimensions compared are often in terms of time, cost and efficiency. Since graphics rendering have a few deciding units regarding performance, benchmarking is a great way of comparing the different methods. We have decided to benchmark FPS (Frames Per Second) and memory usage in the different test cases.

FPS is probably the most interesting unit since it is the main performance unit within rendering, and user-experienced problems with graphics often concern having a low framerate. The memory usage is very interesting when looking at the improvements of the rendering techniques. As stated earlier the decision between hardware or software accelerated rendering is often a compromise between memory usage and performance.

In the testing a benchmark of FPS and memory usage will be saved once per second.

Test cases

To get results that seem relevant to the discussion presented in the previous chapter we have chosen to run the test using Canvas with both hardware accelerated rendering and software rendering. We have also chosen to run the test using OpenGL
TESTING METHOD

to get a comparison between the higher level Canvas API and one operating at a
lower level.

Expected results

The rendering used in the test cases will not be too heavy since there will only be
one object rendered at a time but it will still push the APIs to give as high FPS
as they possibly can. We are expecting OpenGL to be the API that can render
the highest amount of FPS since it runs at lower level and is made for rendering
animated graphics. Other results that are expected is that the hardware rendering
should need to allocate more memory than the software rendering and should show
a different amount of FPS. The FPS difference between the software and hardware
rendering will probably be in favor of the software rendering because of the simplicity
of the rendering instructions used.
Results analysis

Implementing and running tests using some of the APIs gives an even deeper understanding about their differences. This chapter contains information about what results the testing gave and discussion about the results.

Implementation using the different APIs

For a developer it is very important how the different APIs are implemented. If the application is not meant to be using heavy graphics rendering the developer wants simple functions for drawing graphics. Meanwhile a developer that writes a application with the heavier animations and rendering will want more freedom in customizing the graphics and therefore will not mind the more complex code.

This was something that became obvious directly when implementing the test code. As the canvas API is not made for the heavier rendering it has simple functions to draw the basic shapes. The functions such as drawRect() and drawOval() are in the API to enable the simple drawing of rectangles and ovals. These functions made the implementation of the canvas API very easy where a canvas was created, the shape was drawn, the graphics buffer was invalidated to provoke the redrawing that gave the next shape and so on.

When implementing the OpenGL test code it was directly obvious that the whole rendering process would be much more advanced, but at the same time that the possibilities of rendering more advanced graphics was there. OpenGL builds its graphics on triples of vertices that are interpreted as triangles so in the testing example when drawing a square, the square is written as 4 vertexes connected together by 5 edges.[12] The ovals was drawn by first calculating the coordinates of the edge of the oval using polar coordinates, and then rendered using triangles originating from a vertex in the center of the circle to the calculated coordinates.
RESULTS ANALYSIS

Figure 3. Example of a square rendered using 4 vertexes and 5 edges

Test results

The results from the benchmarks taken while running the tests on the device gave the following table of data:

<table>
<thead>
<tr>
<th></th>
<th>Canvas with software rendering</th>
<th>Canvas with hardware accelerated rendering</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average FPS</td>
<td>19</td>
<td>15</td>
<td>90</td>
</tr>
<tr>
<td>Maximum FPS</td>
<td>22</td>
<td>34</td>
<td>96</td>
</tr>
<tr>
<td>Minimum FPS</td>
<td>19</td>
<td>7</td>
<td>62</td>
</tr>
<tr>
<td>Memory usage(bytes)</td>
<td>3,991,872</td>
<td>4,407,840</td>
<td>4,155,632</td>
</tr>
</tbody>
</table>

As deemed in the discussion OpenGL as the API made for animations shows to be by far the fastest of the three in rendering graphics. Between the hardware accelerated rendering and the software rendering the difference is not as significant. The software rendering is a bit faster since as mentioned earlier the instructions for rendering in the test are very simple which benefits the software rendering.
RESULTS ANALYSIS

Figure 4. Graph over FPS rendering performance using the Canvas API

The really interesting fact about the FPS output from the different APIs is how stable the different APIs are. With the software rendering only having a difference of 3 FPS between the maximum and the minimum both the Canvas with hardware acceleration and the OpenGL vary much more in stability. In the graphs we can also see that the minimum FPS rendered is in the beginning of the test in the OpenGL and Canvas using hardware rendering cases. The reason for this is probably because of the GPUs drawing cache which in the beginning loads the instructions needed for future rendering. The big difference between the hardware accelerated Canvas and OpenGL is that OpenGL has clear cache instructions and therefore keeps the rendering speed after the initial cache loading while the Canvas tries to make more improvements to the rendering and by doing this has to rewrite the GPU cache which creates the frame drops that can be seen in the graph above. The software rendering also has its minimum in the beginning loading instructions to the CPU cache. After this the software rendering keeps a quite steady rate since the rendering instructions are run without any severe changes in the process.
RESULTS ANALYSIS

The memory allocation of the different APIs showed exactly as expected that the hardware accelerated canvas needed to allocate the most memory of the three during the test. OpenGL needed a bit less memory because of not trying to make unnecessary improvements for the rendering. Also as expected the software rendered Canvas needed the least memory since the instructions are run directly on the CPU and not needing to allocate any memory for the GPU improvements.

Figure 5. Graph over FPS rendering performance using OpenGL
Conclusions

In the beginning of this paper we had three questions in the problem statement:

- What is currently being done to be able to use the system resources as efficient as possible?

- What is the difference between the different rendering methods used in Android and their resource usage?

- What Graphics component used in Android is best in terms of resource usage in which situation?

The theoretical discussion combined with the test results lead to the conclusions in this chapter.

Improvements being done to the graphics rendering

The main improvement within the graphics rendering of Android devices lately has been the support for hardware acceleration. The hardware acceleration allows the code written in the higher level to be processed and improved at the lower level before actually being executed by the CPU. For advanced rendering instructions this improves the rendering performance. However since the tests run where using such simple instructions for rendering the hardware acceleration proved to be unnecessary and actually decreasing the performance, which proved the doubts that google had when introducing the option. But in general it is an improvement and since there are downsides as well Google has allowed the developers having more knowledge the choice of enabling or disabling the hardware acceleration for separate levels of the program.

In general many of the improvements being done are about getting more processing to run at a lower level which can be seen by the constant development of new lower level APIs that are being released to the Android system. However with the hardware development and the focus on graphics increasing there will always come new ways of improving the rendering performance.
CONCLUSIONS

**Android rendering methods**

Even though the Android rendering methods overlap each other in their functionality, the purposes for which they were created differ a lot. This means that there isn’t a rendering method that performs best in all situations, you have to choose a rendering method depending on what you want to accomplish.

The main difference that affects the performance of the application is what parts of the graphic instructions are being run at what level. With the Canvas API, you have the option to enable hardware acceleration. When developing standard GUI applications with no additional graphical effects, you will probably not benefit from enabling hardware acceleration. However, when the application being developed contains custom graphics and the graphical computations are heavier, hardware acceleration may be the way to go.

The other rendering options (OpenGL, Renderscript) are not meant to compete with the Canvas API, but rather be extensions where the Canvas API simply can’t do what you want. This applies to both the aspect of performance and the aspect of functionality. The main difference performance-wise is that with OpenGL, much of the computation is being done at native level. When the application contains advanced graphics with heavy computation that simply takes too much time to run in the Android VM, OpenGL will be a better choice since the application performance will benefit from running these computations at native level.
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>NDK</td>
<td>Native Development Kit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
</tbody>
</table>

Also worth mentioning is that when processing instructions are considered heavy or light this relates to the amount of processing time needed to execute them.
# Test data

## Canvas with software rendering

Benchmarked FPS at given frame number

<table>
<thead>
<tr>
<th>FPS</th>
<th>19</th>
<th>21</th>
<th>21</th>
<th>22</th>
<th>21</th>
<th>21</th>
<th>20</th>
<th>20</th>
<th>22</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>19</td>
<td>40</td>
<td>82</td>
<td>104</td>
<td>125</td>
<td>146</td>
<td>167</td>
<td>187</td>
<td>207</td>
<td>229</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>20</th>
<th>21</th>
<th>21</th>
<th>22</th>
<th>20</th>
<th>20</th>
<th>21</th>
<th>20</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>269</td>
<td>290</td>
<td>311</td>
<td>332</td>
<td>352</td>
<td>373</td>
<td>393</td>
<td>414</td>
<td>435</td>
<td>455</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>21</th>
<th>22</th>
<th>21</th>
<th>21</th>
<th>21</th>
<th>21</th>
<th>21</th>
<th>22</th>
<th>22</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>517</td>
<td>539</td>
<td>560</td>
<td>581</td>
<td>602</td>
<td>623</td>
<td>644</td>
<td>665</td>
<td>686</td>
<td>707</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>20</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>22</th>
<th>22</th>
<th>21</th>
<th>22</th>
<th>22</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>770</td>
<td>792</td>
<td>813</td>
<td>833</td>
<td>855</td>
<td>877</td>
<td>899</td>
<td>920</td>
<td>942</td>
<td>964</td>
</tr>
</tbody>
</table>

Average FPS calculated by number of frames rendered / runtime
19 FPS
Total amount of memory allocated by the application
3,991,872 bytes

## Canvas with hardware accelerated rendering

Benchmarked FPS at given frame number

<table>
<thead>
<tr>
<th>FPS</th>
<th>7</th>
<th>15</th>
<th>27</th>
<th>18</th>
<th>17</th>
<th>19</th>
<th>18</th>
<th>22</th>
<th>13</th>
<th>14</th>
<th>23</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>7</td>
<td>22</td>
<td>49</td>
<td>87</td>
<td>103</td>
<td>121</td>
<td>143</td>
<td>156</td>
<td>170</td>
<td>193</td>
<td>213</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>19</th>
<th>18</th>
<th>9</th>
<th>16</th>
<th>13</th>
<th>34</th>
<th>14</th>
<th>13</th>
<th>20</th>
<th>20</th>
<th>11</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>232</td>
<td>250</td>
<td>259</td>
<td>275</td>
<td>288</td>
<td>322</td>
<td>336</td>
<td>349</td>
<td>369</td>
<td>389</td>
<td>400</td>
<td>424</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>18</th>
<th>16</th>
<th>12</th>
<th>16</th>
<th>10</th>
<th>21</th>
<th>20</th>
<th>13</th>
<th>19</th>
<th>14</th>
<th>22</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>442</td>
<td>458</td>
<td>470</td>
<td>486</td>
<td>496</td>
<td>517</td>
<td>537</td>
<td>550</td>
<td>569</td>
<td>583</td>
<td>605</td>
<td>623</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPS</th>
<th>14</th>
<th>17</th>
<th>22</th>
<th>9</th>
<th>17</th>
<th>25</th>
<th>22</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>637</td>
<td>654</td>
<td>676</td>
<td>685</td>
<td>702</td>
<td>727</td>
<td>749</td>
<td>764</td>
<td>780</td>
<td>798</td>
<td>816</td>
<td>835</td>
</tr>
</tbody>
</table>
TEST DATA

<table>
<thead>
<tr>
<th>FPS</th>
<th>11</th>
<th>15</th>
<th>23</th>
<th>22</th>
<th>19</th>
<th>20</th>
<th>17</th>
<th>22</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>846</td>
<td>861</td>
<td>884</td>
<td>906</td>
<td>925</td>
<td>945</td>
<td>962</td>
<td>984</td>
<td>998</td>
</tr>
</tbody>
</table>

Average FPS calculated by number of frames rendered / runtime
15 FPS
Total amount of memory allocated by the application
4,407,840 bytes

OpenGL

Benchmarked FPS at given frame number

<table>
<thead>
<tr>
<th>FPS</th>
<th>62</th>
<th>82</th>
<th>87</th>
<th>81</th>
<th>86</th>
<th>95</th>
<th>96</th>
<th>96</th>
<th>95</th>
<th>96</th>
<th>96</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>62</td>
<td>144</td>
<td>231</td>
<td>312</td>
<td>398</td>
<td>493</td>
<td>589</td>
<td>685</td>
<td>780</td>
<td>876</td>
<td>972</td>
</tr>
</tbody>
</table>

Average FPS calculated by number of frames rendered / runtime
90 FPS
Total amount of memory allocated by the application
4,155,632 bytes
Bibliography


