Remote Android Rendering

Joel Isaacson
joel@ascender.com

March 25, 2012

Executive Summary

The material in this paper may be freely disclosed. Provisional and non-provisional patents have been filed to protect these ideas.

Android™ has been accepted as the premier operating system for mobile devices. However, there are many computer systems for which Android is not a feasible choice. What if Android apps could be used by non-Android systems? To that end, we developed an effective way of providing Android apps' capability on non-Android systems. In this novel approach, a remote server runs the Android app and communicates with a local client, which in turn, provides a graphical viewer and a user interface for the app running on the remote server.

The most demanding challenge we faced was to provide remote Android access to deliver graphics at an acceptable performance (frame rate) while using sparse network resources. Many assumptions implicit in local rendering design, such as low latency and high graphics bandwidth, were dealt with to provide remote Android graphics. We have taken an approach that exports graphics at the rendering layer (Canvas, Skia and/or OpenGL) rather than common bitmap layer approach.

In particular, some novel compression algorithms are described that provide very high compression efficiencies. These compression algorithms, which we call **procedural compression,** provide effective compression by modeling the generation of rendering streams

and then bit encoding the resulting stream. Conceptually, the compression algorithm design process is similar to that of the MPEG family of compression algorithms but the underlying data model and implementation details are totally different. The end result is similar, just as the MPEG standards enable practical video streaming, **procedural compression** enables practical remote rendering.

These techniques enable previously unattainable uses for Android technology. It is shown how ordinary Android apps that have not been specially adapted for remote execution can be run in the Cloud thereby enabling hitherto incompatible systems access to the over 400,000 apps in the Android Market.

Background

One of the most compelling trends in current computing is the rise of mobile devices. The Android operating system for mobile devices has gathered momentum since its introduction in 2007. Currently there are over 700,000 Android devices activated each day with more than 400,000 apps available in the Android Market.

Android

The enthusiasm for Android should not blind one to the fact that there are many uses for computing devices for which Android is unsuitable. The classic desktop computer (e.g. MS Windows, Linux or Mac OS) has not been made obsolete by Android. Very few "professional" computer users have completely transitioned to Android use. To quickly read an e-mail, Android is great, but to send a long and complex reply, a desktop system is preferred by most users. Creating a complex document (such as this one) can be best done on a desktop system. The Android system itself is not self-hosted and needs a large Linux or Mac OS X computer to build system images.

The massive scale of the Android Market is both a strength and a liability since the many of apps in the Market are compiled to work only on ARM processors. Thus, a x86 Android device would have a very small selection of working apps currently in the Market. Since the Market is largely dependent on hundreds of thousands of developers to develop, compile and package the apps in the market, deviating from current applicative programming interface (API's) or adding a new architecture to the Market is arduous due to the inertia involved. Since Android's introduction there have been 15 API levels, all of which must be supported in an upward compatible manner, thus forcing each Android version to include historic support for previous versions, and constraining new API's from deviating greatly from previous versions.

Thus users typically use a number of largely incompatible systems. While some of the interesting user data such as e-mail, contacts and calendar can be synchronized seamlessly, applications cannot migrate simply between systems. Desktop applications are not suitable for Android usage without being redesigned and rewritten. Some applications are very difficult to effectively use on Android due to the limitations of hardware (no keyboard or mouse) and software (multi-windows, persistent state and resource management).

Cloud Computing

Another powerful trend in current computing is “Cloud Computing” the delivery of computing as a service, whereby shared resources, software, and information are provided to computers and other devices as a service over the Internet. While a precise definition is not universally agreed upon, there is no doubt that Cloud Computing plays, and will play an essential part in the delivery of computing resources.

The principles of cloud computing are largely orthogonal to the Android system. In Android, the availability and quality of the network connection does not totally dictate the user experience. Android devices try to maintain as much usability as possible when a data network is unavailable. All programs (apps) are stored in local storage and in many cases data is either stored or cached locally to provide functionality when disconnected from the data network. For example, both the program to dial cellular phone numbers and the user contact database are locally stored, otherwise phone calls could not be completed while unconnected to a data network. Systems that are “Cloud” based rely on a high quality and reliable Internet connection. There is not much that can be done on a Cloud-based system that is disconnected from the Internet. Cloud computing decouples the computer architecture of the program running in the Cloud from the client. For example, an ARM server can provide Cloud services to x86 clients.

An example of a Cloud client is Google Chrome OS a Linux-based operating system that works exclusively with web applications. Even though both Android and Chrome OS are Google products and share similar end-user computing goals, currently there is very little common functionality between these two systems since Android apps are unavailable to users of the Chrome OS. Besides the additional expense in developing applications for two platforms, the user must learn to use two different interfaces.

Remote Access to Android Apps

Any Android app can be run on the server. It is not necessary that the remote server and the local device have the same architecture, i.e. an Intel server can provide services for an ARM device. Below are scenarios of some use cases.

Remote App Server

Consider a standard Android contact manager running on the server. As such, the contacts can then be the complete contact information of the corporation or organization that is running the server. By allowing the most current corporate contact database to be accessed without having to sync contacts with the mobile devices of personnel, a major security risk is avoided in case of lost or stolen mobile devices. One large corporate server should be able to support hundreds of concurrent Android apps.

Another possibility would be to provide data storage that is private to each client. This might be done with a private `chroot` environment for each client. In this configuration each local client would have private contact lists.

It is interesting to consider an application such as Google Maps running on the remote server. In this case, queries of the location manager originating on the remote server must be executed on the local device and returned to the remote server. Input (keys and touchscreen) must be performed locally and sent to the remote server. In addition, audio from the application (turn by turn instructions) must be sent to the local device.

App Library / Subscription Model

Currently, apps are loaded into the local device - either installed at the time of purchase or added subsequently. A significant market of post-sales installation of apps has developed. If efficient remote execution of apps can be supported, then instead of software purchases, a subscription model becomes possible. A fixed monthly fee would entitle the subscriber to access a large library of applications.

Purchase / Rental Models

In a purchase/rental model, apps can be demoed remotely, prior to purchase. If the user of the device finds the app to his/her liking, it can then be bought.

Remote Enterprise Applications

Applications may benefit from running within the enterprise's data centers, with the obvious benefits of scalability, security and maintainability.

A worker at a corporation proceeds through various computing environments during a typical day. S/he starts at home at a computing setup such as a traditional fixed (display, keyboard, mouse) device, or a semi-fixed docked mobile computer or a tablet. To prepare for the office environment, the worker can migrate his running app to a tablet, laptop, or mobile phone. Once at the office, the worker can migrate the app to the desktop device. The procedure is repeated in reverse at the end of the work day - possibly going through several migrations to different devices.

Set-top boxes

Many set-top boxes cannot practically run Android since the set-top box might not have sufficient resources or might not contain an ARM processor. A local client application that only performs remote rendering takes limited resources and needs no long term persistent state. This relatively simple generic rendering client can provide access to the vast array of Android apps.

In addition, set-top boxes typically don't have complex local installations. The challenge of enabling the previously mentioned use cases is based on the premise that the Android environment can be split between a remote execution server and a local client that will provide a graphical viewer and user interaction to the server. A set-top box that has tens of apps installed by the user would create a very difficult support challenge to the network operator, running the apps in the operator's "cloud" would greatly simplify system maintenance.

Remote Execution

The challenge of enabling the previously mentioned use cases is based on the premise that the Android environment can be split between a remote execution server and a local client that will provide a graphical viewer and user interaction to the server. The rest of this paper will provide a technical explanation of how we have accomplished this challenge.

The Android system has not been designed for remote execution. In order to run an Android application remotely, system services must be exported from or imported to the remote server. Such services include:

- Camera driver

- Audio driver
- Keypad driver
- Touchscreen driver
- Location manager
- Graphics subsystem

For example, audio output might be exported from the remote server to the local client. Audio input might be imported to the remote server from the local client. For spatially separated devices, the location manager might reside on the local client and import this service to the remote server.

Interaction with these services possibly will incur round trip latencies. Thus, for the touchscreen services, the latency between the "touch" and the graphical interaction is at least a round trip delay.

Without doubt, the service with the potential to take up the most network bandwidth is the graphics subsystem, thus optimizing the remote graphics protocol is essential to make remote access feasible.

Remote Graphics

We first consider Android graphics in releases that preceded the Ice Cream Sandwich (ICS) release. The graphic stack of Android is represented in Fig. 1 which illustrates the four layers of Android's graphics:

1. Application Layer
2. Graphic Toolkit
3. Graphic Rendering Library
4. Bitmapped Device

Graphics may be exported via any of the layers of Fig. 1. There are examples of exporting graphics at all four levels. The graphics of the bitmapped device can be exported via VNC or Chromoting. Graphics at the rendering level might be exported via X11 or Microsoft's RDP. An example of the exporting of graphics at the toolkit level is IBM's Remote Abstract Windowing Toolkit (RAWT). Exporting graphics at the application level can be done by the use of HTML5 in which the application itself is exported.

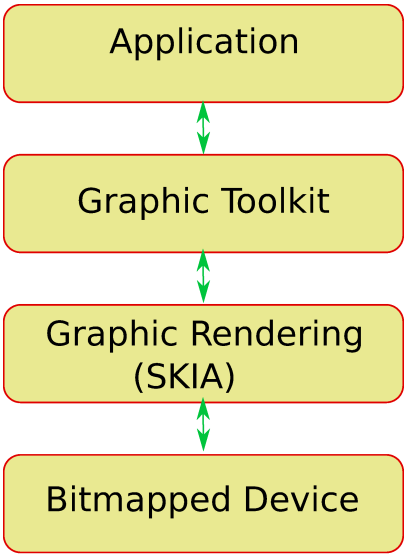

Figure 1: Graphics Stack

We want to be able to provide remote access to Android apps that have not been especially prepared to provide remote graphics. Here are some possibilities:

Bitmap level: Although exporting graphics at the bitmapped level is technically easy but it is very difficult to achieve high performance with data networks that are slow. In addition the ever increasing display sizes make the task of sending bitmaps more difficult.

Toolkit level: Alternatively, exporting graphics at the graphics toolkit level is technically difficult but can provide good performance. The graphical toolkit has many different versions that must be supported; the interface is large and complex. In addition user defined views (widgets) are frequently used which can't be exported easily.

Application level: Since the majority of Android applications are compiled by third parties, there is no access to the source code. Thus, it is impossible to provide remote access at the application level.

Rendering level: The above difficulties have prompted us to consider the rendering level for exporting remote graphics .

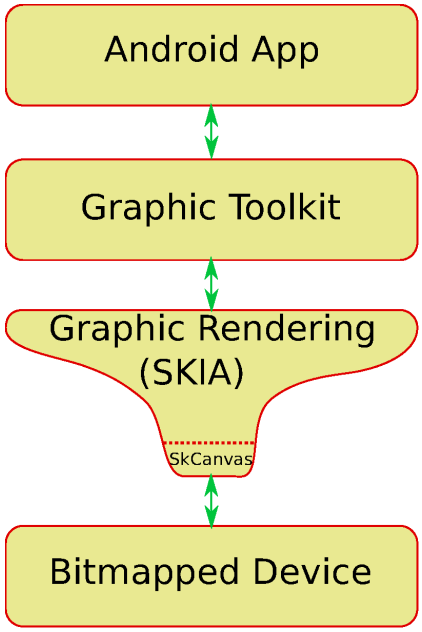

Figure 2: Graphics Stack

The graphical rendering layer is quite simple and is based on the Skia graphics engine. All graphical rendering operations are done with objects from the `SkCanvas` class (Fig. 2). This class contains all the methods that are used to render graphical operations to pixels. `SkCanvas`' draw into a `SkDevice`, which might be a simple `SkBitmap` object in memory or a framebuffer device. In Fig. 2 `SkCanvas` is drawn with this narrowing to emphasize that the graphic rendering stream is funneled at this stage through a small number of well defined API's.

Modifying the `SkCanvas` class shows that just 17 SKIA functions must be included in the remote stub to provide remote graphics access. A prototype remote-local renderer was implemented to check the feasibility of remote-local rendering. This prototype maps the Android applications into multiple X11 windows and checks that the intervention in the `SkCanvas` class is sufficient to provide remote-local rendering. In addition, traces for the remote-local rendering network stream are generated.

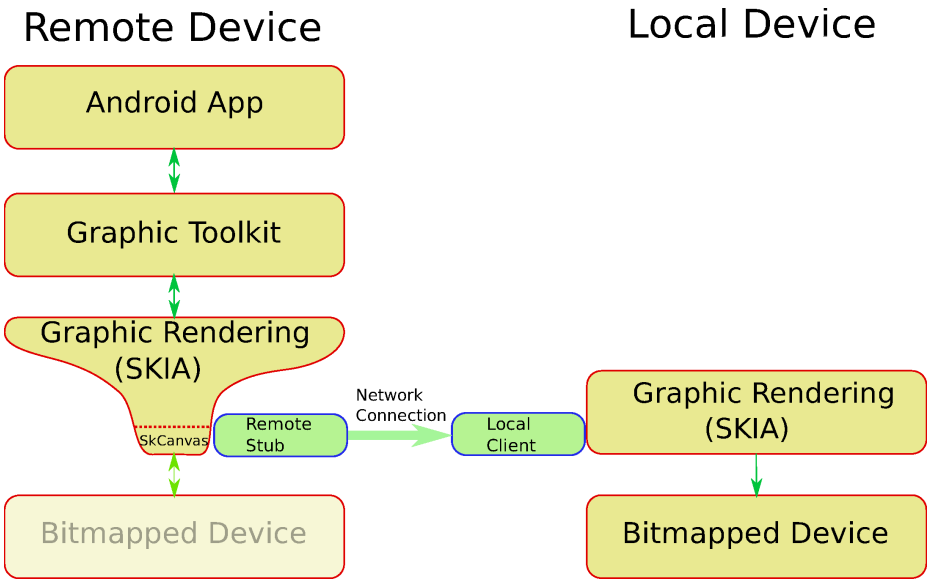

Figure 3: The Remote Graphic Stack

Remote Graphics Performance

As demonstrated, it is relatively easy to modify the Android graphics subsystem to provide remote graphics. The challenge, however, is to provide acceptable performance over networks with limited bandwidth and long haul latency.

Frame latency

Long haul networks have latencies that are attributed to both the finite speed of light and switching delays. The east-west coast round trip is approximately 70 ms. If, for example, there are 30 frames per second, and each frame causes a latency of 70 ms (one round trip), this latency alone would delay 30 frames by 2.1 seconds. The maximum number of frames in this case dictated by the round trip delay alone would be 14.2 frames a second.

Thus, the rendering stream must be a simplex (one-way data flow) link for frame rendering. Some two-way interactions can be tolerated during initialization. The problem with this scheme is that the graphic toolkit needs responses from the graphical rendering layer (Skia). One use for this back-flow of information from the rendering layer to the graphical toolkit is “measure passes”. Measure passes access

geometric information used in the creation of the layout of a frame. The measure pass might generate tens of queries to the Skia rendering software for each frame layout. If the queries are answered by the remote rendering software, many round trip delays would be incurred.

Normally, a remote procedure call (RPC) entails both sending the arguments of a procedure and receiving the results of the procedure via a reply message which is needed for the proper functioning of the software. This RPC paradigm must be broken to send messages with procedure calls without waiting for replies. This also allows messages to be batched optimally into a stream for sending via a network. The way to maintain the semantics of the renderer (Skia) is to run in parallel the renderer software on the remote machine. Both the remote and local rendering software layers are passed identical data and run the same rendering software. For this reason, the results returned by the remote renderer are identical to what would have been returned, over the data link, from the local renderer. The remote renderer can reduce greatly its CPU resources by omitting the computationally intensive task of actually rendering pixels. The rendered image is not needed on the remote side.

Fig. 3 illustrates these points. The lack of a need to render pixels on the remote side is shown by the graying out of the bitmapped device. The one-way nature of the rendering stream is shown by a unidirectional arrow between the remote and local systems.

With a remote procedure protocol guided by the design principles above, we can stream graphical frames at full bandwidth capacity without round trip latencies. Of course, interactions with devices such as a touchscreen, mouse, keyboard or keypad will suffer a round-trip delay of the graphical response.

Compression

Even without network latencies posing a serious problem, the network capacity of long haul networks presents a limiting factor on how many frames per second can be displayed. Careful optimization of the volume of data sent per frame is needed in order to achieve an acceptable frame rate. Compression is the principal tool that is used to reduce the volume of data sent over the network link.

Information Theory Analysis of Rendering Stream We can examine rendering streams in an attempt to estimate what compression rates are to be expected. The measure of information, H, is called **entropy,** and it can be used to predict what data compression can be expected. There are various data models that can be used to compute entropy.

Zero-order Model: In this model, each symbol, s_i, is statistically independent from each other and equally probable to appear. Here the entropy rate, in bits/symbol, is given by:

$$H_0 = \log_2 m \tag{1}$$

where m is the number of symbols in the stream.

First-order Model: In this model, each symbol is statistically independent from each other but has a non-uniform distribution in the data stream. Here the entropy rate, in bits/symbol, is given by:

$$H_1 = \sum_{i=1}^{m} p_i \log_2 p_i \tag{2}$$

where p_i is the probability of the symbol, s_i, in the rendering stream.

Second-order Model: In this model, each symbol, s_i, has a conditional probability $P_{j|i}$ of being found in the data stream that is dependent on, s_j, the previous symbol seen. Here the entropy rate, in bits/symbol, is given by:

$$H_2 = \sum_{i=1}^{m} p_i \sum_{j=1}^{m} P_{j|i} \log_2 P_{j|i} \tag{3}$$

General Model: In this model, $B_n$represents the first n symbols. Here the entropy rate, in bits/symbol, is given by:

$$H_\infty = \lim_{n\to\infty} \frac{1}{n} \sum P(B_n) \log_2 P(B_n) \tag{4}$$

where the sum is over all m^n possible values of B_n. Whereas it is impractical to actually calculate all the myriad conditional probabilities involved with this formula, having an accurate domain specific data model is very effective in approaching this theoretical limit, which represents the best compression possible.

As we increase the order of the data model, the entropy approaches a limit given by the general entropy model. The following condition holds true:

$$H_\infty \leq \cdots \leq H_{n+1} \leq H_n \leq \cdots \leq H_2 \leq H_1 \leq H_0 \tag{5}$$

	Procedure Calls - n	Unique Calls - m	H_{-1}	H_0	H_1^{bzip2}	H_1	H_∞^{bzip2}	$\frac{H_{-1}}{H_\infty^{\text{bzip2}}}$	$\frac{H_1^{\text{bzip2}}}{H_\infty^{\text{bzip2}}}$
Skia	13932	17	8	4.08	3.475	3.083	0.214	37.383	16.238
OpenGL	55379	45	8	5.49	4.237	3.881	0.131	61.069	32.344

Table 1: Some rendering stream compression results

Some Experiments on Rendering Streams

We have performed some simple experiments on GUI rendering streams. Two streams were analyzed:

1. A Skia software rendering stream taken from the Gingerbread contact manager.

2. An OpenGL hardware rendering stream taken from the Ice Cream Sandwich contact manager.

Here the rendering streams have been greatly simplified. Only the procedure calls have been included in the data stream. The data arguments have been deleted. This simplifies the analysis but still is indicative of the general results expected. The results of these experiments are shown in table 1.

In this table, the two rows show results for the two rendering streams, software and hardware. In general, the OpenGL hardware stream is 5-6 times more verbose than the Skia software stream. In these examples, the rendering performed in the two instances were similar, but not identical. There were only 17 (Skia) or 45 (OpenGL) different rendering routines called.

H_{-1} is the number of bits (8) used to represent each rendering call in the data stream. Simply put, each rendering call is coded as one byte in the uncompressed stream.

H_0 ($\log_2 m$) is the number of bits needed if all rendering calls have equal frequency.

H_1^{bzip2} is the average actual number of bits needed to represent each rendering call if only the frequency of calls is considered. This was not calculated from eq. 2, but rather by shuffling the procedure calls so as to destroy any conditional correlations between calls. The resulting file was compressed with bzip2, a general purpose compression program based on the Burrows-Wheeler algorithm. The Burrows-Wheeler algorithm identifies frequently-recurring character sequences. This process was performed so as to better be able to isolate the contribution of conditional probabilities to the bzip2 compression. As can be expected, it is bound from below by H_1 .

H_1 is the average number of bits needed to represent each rendering call if only the frequency of calls is considered. It was calculated from Eq. 2.

H_∞^{bzip2} is the average number of bits used by bzip2 when compressing the rendering call file. Thus, for example, compressing the OpenGL rendering stream needs 0.131 bits per rendering call.

$\frac{H_{-1}}{H_\infty^{\text{bzip2}}}$ is the compression ratio for the rendering files as compressed by bzip2. Thus, the Skia stream is compressed by a factor of 37, and the OpenGL by a factor of 61.

$\frac{H_1^{\text{bzip2}}}{H_\infty^{\text{bzip2}}}$ is the compression ratio that can be attributed to the conditional probability. To use a physical model it can be thought of as the average correlation lengths within the data stream. Thus, we can think of the OpenGL rendering stream as being made up of repeating clusters of code fragments with an average size of 32 rendering instructions.

It must be appreciated that the procedure data arguments associated with the rendering stream also have to be compressed and have not yet been addressed. The above analysis is interesting since it shows that relatively simple compression ideas are effective in greatly compressing the procedure rendering stream. We can now describe some domain specific compression algorithms.

Domain Specific Compression

To design a data compression scheme, the redundancy inherent in the data must be understood. By understanding how the data is generated, a model may be built that can exploit probable correlations within the data. Examining a somewhat similar compression domain can provide motivation to uncover techniques for compression within our domain.

MPEG Compression Video streams, whether taken photographically or photorealistically synthesized, can be compressed with MPEG standard codecs. Even though no specific technique used in MPEG compression is applicable for the compression of our problem domain (rendering streams), some of the assumptions about how the data sets are generated are similar and the compression model design is similar. Here we treat MPEG compression in a general manner, although in practice there are a number of incompatible MPEG standards.

The raw material for MPEG compression consists of a series of RGB images. Assumptions about the corpus of material that MPEG will compress are:

- The material consists of a large number of sequential images (frames).
- The target of the video images is the human visual system and consists of moving images.
- The subject matter is a product of our everyday visual world and not some random series of random images.
- The apparent smooth motion observed is an optical illusion called the **phi phenomenon** that allows our brains and eyes to perceive constant movement instead of a sequence of images. The effect is dependent on visual continuity between frames.

The assumptions above lead to the following domain specific compression techniques:

1. Color opponency theory describes the receptors in the human eye as responding to light as an RGB camera. Processing within the optic nerve converts these RGB signals into three separate channels, one for intensity and two for color differences. The conversion of RGB images to YUV is motivated by this physiological model of human vision. Since the color channels leading to the visual cortex have a lower bandwidth than the luminance channel (Y) the color channels can be sub-sampled (typically 2x2) without objectionable artifacts, thereby reducing image size by 50%.
2. Within each frame, the accuracy of spacial changes with shorter wavelengths are less important to human vision than the accuracy of longer wavelengths. This allows for wavelength dependent quantization scaling factors that drop features not noticeable to viewers of the material.
3. There are long series of frames that are highly correlated between consecutive frames, outside of cut shots and scene changes. The actual differences can generally be described by the motion of objects between the frames. For this reason, the inter-frame compression of the MPEG standard is based on finding motion vectors of regions of the current frame based on previous and subsequent frames.

The first two techniques, also used in the still image JPEG standard, exploit intra-frame (dealing with an individual frame) correlations. The third technique exploits inter-frame correlations. In addition, general compression techniques, such as run length encoding (RLE) and entropy encoding (Huffman or Arithmetic), are used to produce a minimum bit count representation of the compressed material. By their nature, MPEG video streams are amenable to lossy compression.

Rendering Compression An analysis along the lines of the MPEG example is needed to construct a compression model that efficiently compress the remote-local rendering stream shown in Fig. 3. The GUI rendering output domain is much more sensitive to degradation of image quality than the MPEG, so we will only consider lossless compression. The graphical rendering stream consists of a long sequence of rendering function calls with arguments.

```
render1(arg1, arg2, arg3); render2(arg1, arg2); ...
```

There are markers in this stream that indicate the end of a frame and the beginning of a new frame.

Here the assumptions about the corpus of material to be compressed are somewhat similar to MPEG's:

- The rendered material consists of a large number of sequential images (frames).
- The target of the GUI images is the human visual system.
- The subject matter while not being a product of our everyday visual world is modeled on the visual contexts of this world. We therefore see intensive use of continuous effects such as scrolling, swiping, cover flow, animations, etc.
- The GUI images are generated frame after frame by repeated invocation of GUI procedures by the application software.

These assumptions can lead to the following compression techniques that for lack of a better term, will be called **procedural compression.**

1. In practice, the number of unique sequences of rendering functions in execution paths taken within the code are bounded. This is because the rendering commands are generated by a fixed number of GUI functions and an application running a bounded amount of code. Conditional logic within routines can generate two different execution paths, but even here the number of paths in practice are bounded. The execution paths can be incrementally learned and entered into a **procedure dictionary** as the rendering commands are streamed. An execution path that has been previously encountered can be transmitted by dictionary reference.
2. Even if the sequences of rendering functions themselves are in the dictionary, the data arguments associated with these functions might be quite different from one another. Therefore, we keep

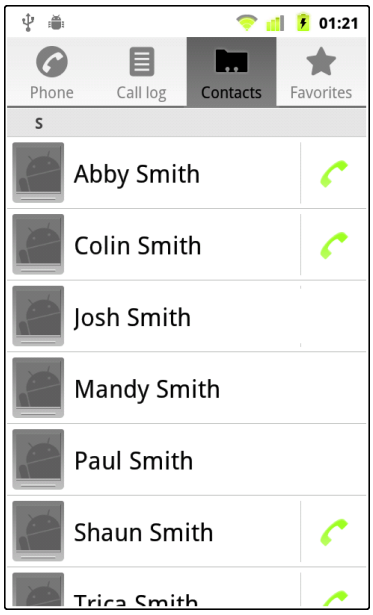

Figure 4: A contact list

a dictionary of the data arguments previously encountered. As a rendering procedural sequence with associated data is encountered, the data sequence dictionary for this procedural sequence is searched for the closest match to the current data sequence. Once the closest match is found, only the differences from this match are transmitted.

The example in Fig. 4 illustrates how these compression ideas work. Fig. 4 shows a screenshot of a contact screen with 7 entries. When we look at the rendering stream, we see that the code to compose one contact entry (as shown in Fig. 5) is run seven times in two variants. The first two, and the last two entries, have phone numbers attached to the contact and run different variants of the rendering code. As the first frame is generated, the code to render contacts is entered into the dictionary. When the second instance of this code is encountered, only a reference to the dictionary entry need be sent. The arguments differ partially for each entry. Thus, even for the first frame, intra-frame compression is effective.

For subsequent frames as the contact list is scrolled up or down, each contact that appeared in previous frames will match both the code and data dictionaries and will have very small differences (mostly in the location of the entry). Thus, references to both dictionary entries for the code and for the closest matching argument data leave very little additional information that must be sent to render a contact. Compression

Figure 5: A contact

ratios can realistically be over 50, allowing 30 frames per seconds (fps) on typical long haul networks.

Structured Procedural Compression A careful examination of the rendering stream generated by the Android GUI reveals additional structural information that can be used to improve the data model. The rendering stream has balanced **`save()`**-**`restore()`** pairs for each frame of the rendering stream. Each **`save()`** is found at the beginning of a GUI function and a **`restore()`** is found at the end of each GUI function. This information can be used to reverse engineer individual GUI and application procedures. It will also reveal the call-graph of these procedures.

Using this information, the rendering *code* dictionary becomes a rendering *procedure* dictionary. The call-graph data is best embedded in the per-procedure data dictionary. Although the reason for this might not be evident at first, it is very useful for GUI interfaces which frequently have variable procedure calls from defined procedures. This is typical of GUI software that might have container widgets with variable types of other widgets embedded.

Being able to better model the rendering stream code source will allow improved prediction of the rendering stream and increase the compression ratio. As the procedure code dictionary is constructed, a database of reverse engineered GUI procedures is built up. Since there is normally a bounded number of GUI procedures, we rapidly encounter and catalog the great majority of procedures.

Our compression algorithm was tested on the rendering trace of a 60 frame sequence for the application shown in Fig. **4**. There were 13702 rendering commands for an average of 228 rendering commands per frame. Of the 13702 rendering commands, there were 2691 functions (save/restore pairs). Of these, only 47 were unique. Only these 47 command sequences (routines) need be transmitted to the local client and entered into the rendering procedure dictionary. This gives a compression rate of 1.75% (about 1:57).

Of the 13702 rendering commands only 354 had completely unique data parameter sets and 203 had data sets which are partially different. Only these 557 data sets must be transmitted, thereby giving data compression of 4.06% (about 1:25). If the partially different data sets are differentially transmitted,, a data compression of 3.3% (about 1:30) is obtained.

For the last stage of compression, general techniques such as run length encoding (RLE) and entropy

encoding (Huffman or Arithmetic) are used to produce a minimum bit count representation of the compressed material. This additional phase should be expected to reduce the number of bits by a factor of 2-3.

Our results show that compression can be an effective means to decrease the amount of data transmitted for a rendering stream. From the raw number a compression of 1:50 to 1:100 is not unreasonable.

If these tests were run for a longer period (more frames), the compression ratios would be higher. Longer streams will have greater redundancy and thus better compression ratio. This is because the number of new procedures added to the procedure code dictionary decreases rapidly once the unique code sequences have been learned and entered into the dictionary. In the previous example of 60 frames, within the first one third (20) frames, 38 of the 47 routines (80%) are encountered. In the final one third of the frames (frames 41-60), only one new routine (2%) is added. Even if there is a "cut" to new material the previously learned routines will likely be also used in the new material. This is true to a lesser extent for the per-procedure data dictionaries.

Ice Cream Sandwich (ICS)

With the release of Android 3.0 (Honeycomb), the graphic rendering layer was redesigned. In addition to the previous Skia (software) rendering engine, an optional alternative OpenGL 2.0 (hardware) rendering engine has been added. The Android 4.0 (ICS) release has elevated the hardware option to be the default mode of operation.

With this new graphics implementation there are three alternate ways to export the rendering interface.

Hardware Rendering: The OpenGL 2.0 rendering stream can be instrumented with a remote stub that will send the rendering commands to the local client.

Software Rendering: The Skia renderer can be instrumented with a remote stub. Technically, this solution is very similar to the already described pre-Honeycomb remote rendering interface.

Abstract Renderer: This is the user visible interface to the rendering system. All rendering (drawing) operations are performed via the **`Canvas`** class. For any UI widget (a **`View`** object), one of the most important methods is **`onDraw()`**, which draws the widget onto a **`Canvas`** that is provided to

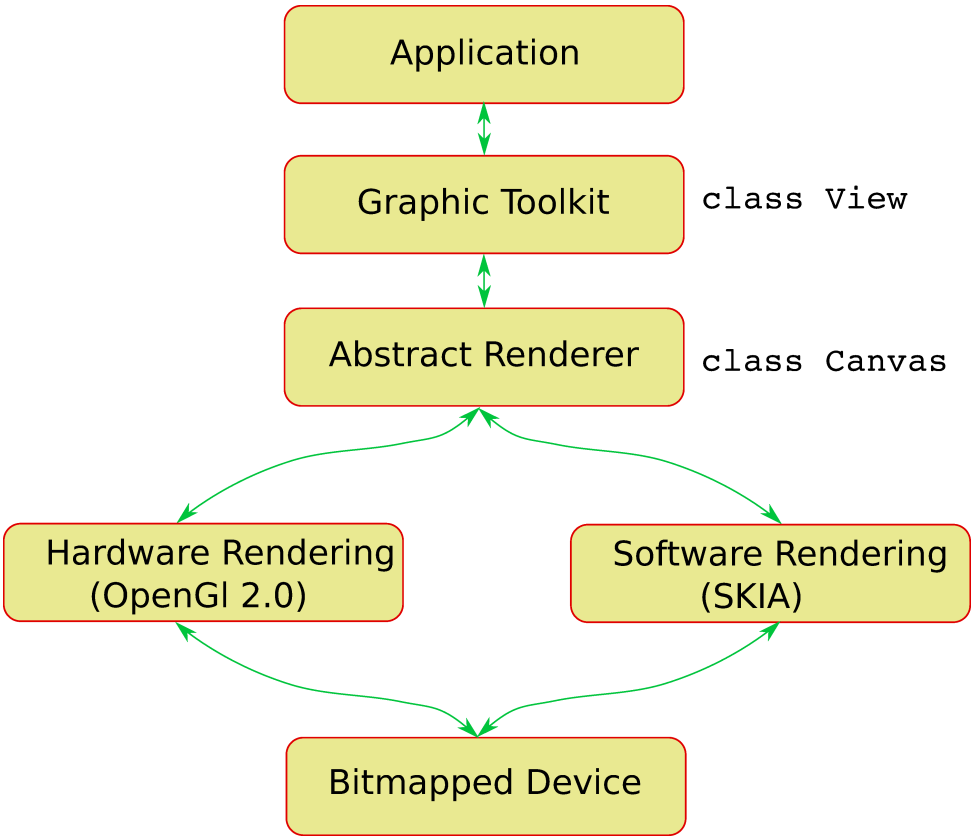

Figure 6: The ICS Graphics Stack

the method. Even a cursory look at the **`Canvas`** class shows its similarity in design and functionality to the Skia renderer. While the use of the Skia renderer is not available to user applications, **`Canvas`** objects are available and well documented. This layer has been in use in Android since version 1.0. In the newer versions of Android, API's which are more amenable to hardware rendering such as **`drawPatch(),`** have been added. Due to the similarity with the Skia renderer, this interface can be instrumented with a remote stub in a similar manner to the Skia implementation. Even if the software rendering path (Skia) is removed in the future, the **`Canvas`** class implementation alternative should remain viable and has potential to provide efficiencies for remote rendering. This would seem to be the preferred method of exporting the rendering interface.

All of above rendering streams are highly compressible by procedural compression.

Remote OpenGL Rendering

It is interesting to try to instrument the 3D OpenGL rendering for remote graphics. While it is certainly not an optimal way to export the Android GUI since the OpenGL rendering stream is typically almost an order of magnitude more verbose than the equivalent Skia stream, it is useful since remote OpenGL rendering must be dealt with if the many Android apps that use OpenGL can be supported. OpenGL rendering can be encountered in many contexts either in hardware GUI rendering, GLSurfaceView, Renderscript or Native Development Kit (NDK).

A test of instrumenting the OpenGL API shows that it is possible to export the OpenGL rendering. The OpenGL API is much more complex and verbose than Skia rendering. The effort involved with instrumenting the OpenGL API is also about an order of magnitude greater than the effort needed to instrument Skia rendering. On the other hand, the OpenGL rendering stream is more amenable to compression than the terse Skia rendering stream. It would seem possible to achieve good graphical performance even of OpenGL rendering over typical long haul networks.