ASCENDER TECHNOLOGIES LTD.

Cloud Graphical Rendering:

A New Paradigm

Joel Isaacson

joel@ascender.com
www.ascender.com/remote-graphics

Version 1.7

Executive Summary

Cloud rendering of modern graphics is typically performed via remote hardware rendering and pixel-based video compression techniques for image transmission. These solutions perform poorly, profligately expending both system and network resources. In response, ASCENDER TECHNOLOGIES developed novel enabling technology where the rendering of pixels is performed only on the local client, which makes for a much more affordable solution without expensive graphical hardware in the cloud. In addition, ASCENDER'S compression techniques reduce the networking overhead, typically by over an order of magnitude.

The trend in modern graphical rendering systems is to a higher quality visual experience. Frames are generated at a very fast rate (~60 fps) and a complete re-rendering is performed for each frame. In addition, the size and density of displays have grown over the previous generation, greatly increasing the number of pixels generated for each frame. These combined changes in technology challenge the prevailing methods of providing remote graphics. In order to meet networking bandwidth constraints, remote pixel based-methods typically are forced into compromises in:

○ Latency ○ Resolution ○ Image Quality ○ Frame Rate

ASCENDER'S enabling technology does not suffer from any of the above compromises and is significantly less expensive to implement.

ASCENDER'S enabling technology is applicable to a wide-range of systems and encompasses many use cases. In this white paper some use cases are considered:

Cloud Gaming: Unlike server-side pixel renderers, such as the *Nvidia Grid,* which employs expensive hardware to render pixels in the cloud, extensive bandwidth to transmit an H264 video stream and nevertheless compromises all four performance parameters, ASCENDER'S enabling technology:

- ○ Uses less cloud resources
- ○ Does not need remote GPU hardware
- ○ Uses lower network bandwidth
- ○ Does not compromise performance

Cloud Android Apps: Android apps can be run in the cloud and the graphics displayed on the local device. Five different Android rendering API's (Canvas, OpenGLRenderer, OpenGL ES 2.0, OpenGL ES 1.x and Skia) were implemented and tested giving coverage of a large percentage of Android apps.

Other Use Cases: We also briefly consider other use cases: App Library / Subscription Model, Purchase / Rental Models, Remote Enterprise Applications, Set-Top Boxes, Automated Testing and WebGL Browser Based Implementations.

Contents

1 Introduction **4**

2 Inherent Difficulties of Modern Cloud Graphics **4**

2.1 High Frame Rate 4
2.2 Consistent Frame Rate 6
2.3 Graphical Effects 6
2.4 Latency 6
2.5 Display Resolution 6
2.6 Broadband Network Bandwidth 7

3 Difficulties By Design **8**

3.1 Heavy Use Of Cloud Resources 8
3.2 Heavy Use Of Network Resources 8
3.3 Encoding Latency 9
3.4 Image Quality 9
3.5 Frame Rate 10
3.6 Rendering Latency 10

4 The Ascender Approach **10**

4.1 No Remote Pixel Needed 12
4.2 The Four Compromises - Uncompromised 13
4.3 ASCENDER'S Remote Rendering Technology 14

5 Use Cases **20**

5.1 Use Case: Cloud Gaming 20
5.2 Use Case: Android Apps 22
5.3 Mobile Devices 26
5.4 Remote App Server 27
5.5 App Library / Subscription Model 28
5.6 Purchase / Rental Models 28
5.7 Remote Enterprise Applications 28
5.8 Set-Top Boxes 28
5.9 Automated Testing 29
5.10 WebGL Browser Implementations 29

6 In Summary - A New Cosmic Alignment **29**

1 Introduction

This whitepaper contrasts the prevailing hardware solution for cloud graphics Fig. 1.1 with ASCENDER'S software solution Fig 1.2.

The pressure to exploit cloud-based technologies is pervasive and ongoing. In parallel with the rise of cloud computing, modern graphic technologies (e.g. Android, IOS, OpenGL) have made it more difficult to implement remote graphics in the cloud. The reasons for these difficulties are manifold, some based on technology, some by design and some on economics. ASCENDER'S enabling graphical technology addresses all these difficulties and provides an economical solution.

In section 2, we explain the challenges in implementing modern cloud based graphics systems.

In section 3, we discuss the difficulties introduced by server-side pixel renderer solutions represented by the *Nvidia Grid* or AMD's *Radeon Sky*.

In section 4, we discuss ASCENDER'S elegant solution to the problems of remote rendering, otherwise entitled **Remote Rendering - Local Pixels** technology**.**

In section 5.1, we discuss the applicability of ASCENDER'S technology to cloud gaming, a high profile industry - with inherent difficulties - believed to be the wave of the future. We detail a case study of ASCENDER TECHNOLOGIES' techniques applied to a popular 3D game.

In section 5.2, we discuss how our technology enables cloud based remote Android apps. The various rendering technologies used in Android are discussed as well as how their cloud implementations lead to "coverage" of the large base of Android apps. One use of our technology can lead to the much talked about "convergence" of Chrome OS and Android. We detail some case studies.

2 Inherent Difficulties of Modern Cloud Graphics

Remote graphics has a long history, starting with legacy graphics systems such as the *X11 Window* system and *Microsoft Window's GDI*. However, trends in modern graphical systems - Android and IOS - challenge remote implementations of these systems.

2.1 High Frame Rate

Legacy graphics systems work at slow frame rates, on the order of 12 fps. This rate is sufficient to create a perception of apparent continuous movement. Even with slow frame rates, device refresh rate must be on the order of 60 cycles/sec to combat flicker.

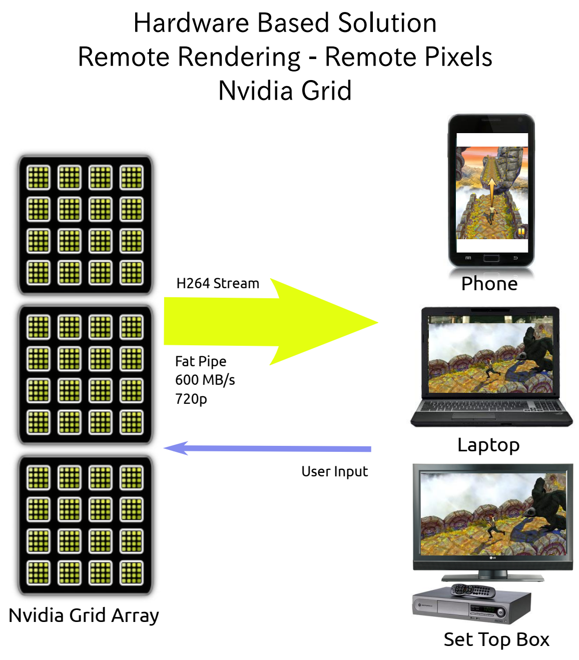

Figure 1.1: Server-side Pixel Rendering

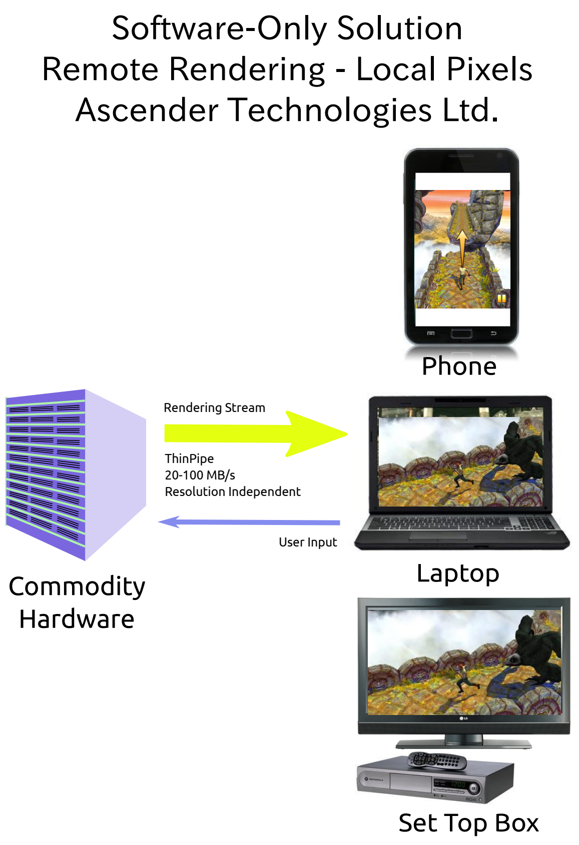

Figure 1.2: ASCENDER'S solution

Modern graphics systems peg the frame rate to the *vsync* refresh rate which is about 60 frames per second. Graphic animations are then perceived as if they are physical objects. Scrolling a list has the visual effect similar to the scrolling of a physical piece of paper.

2.2 Consistent Frame Rate

Legacy graphics systems make no attempt to deliver frames at a consistently fixed rate. The perceived visual effect is frequently “jerky”.

Modern graphics systems endeavor to supply a new frame for each display refresh. The Android “Project Butter” was launched to provide a “buttery smooth” user experience. Providing a consistent frame rate (60 fps) linked to the *vsync* timing is a foremost objective.

2.3 Graphical Effects

Legacy graphics systems have basic GUI elements: buttons, scrolling lists, canvases. Transition between application contexts (panes) are abrupt.

Modern graphics systems are based on physical models. Transition between application contexts are frequently continuous via a touch and sweep paradigm. The graphical rendering technology (i.e. OpenGL) used even in simple GUI’s has more in common with the world of 3D gaming than with the graphical rendering of legacy systems.

2.4 Latency

Inherent latencies within the cloud are associated with round-trip delays in the underlying network; they can be broken into two components:

1. Speed of signal propagation delays: normally constant, the latency is dictated by the length of the physical data path and laws of nature.
2. Switching delays: might vary over time because of current network congestion.

These latencies are difficult to remedy. In the first case only, physically lessening the distance to the cloud server will lower the latency, and in the second, only a network infrastructure upgrade will help.

2.5 Display Resolution

Display resolution inexorably increases as shown in Fig. 2.1and Table 1. The increasing number of pixels on displays has necessitated a transition from software graphical rendering (Skia on Android) to hardware rendering (OpenGL).

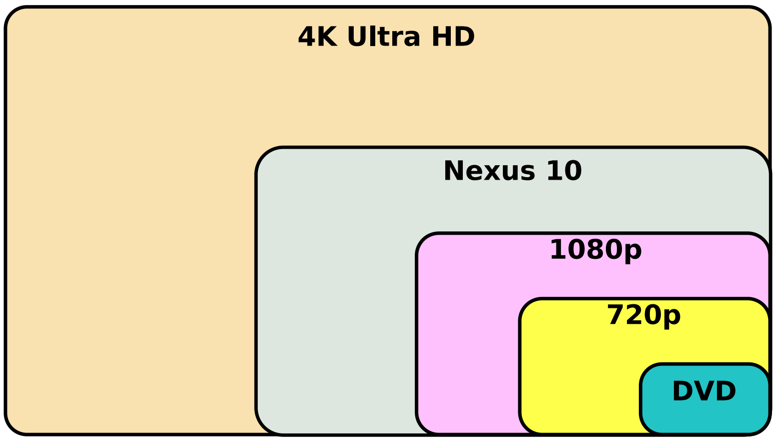

Figure 2.1: Display Resolution

Display	Commercial Availability	Rows	Columns	Pixels
DVD	1997	720	480	345,600
720p	1998	1280	720	921,600
1080p	2006	1920	1080	2,073,600
Nexus 10	2012	2560	1600	4,096,000
4K Ultra HD	2012	3840	2160	8,294,400

Table 1: Display Resolution

2.6 Broadband Network Bandwidth

Broadband network availability is variable and the service provided is not uniform. Even with H264 compression, high quality 720p video requires roughly 600 Kbytes/sec or 2.2 GB per hour which frequently strains network resources. Larger resolutions require larger network bandwidths. Even if a network delivers broadband access to a customer, the combined and simultaneous streaming by many users can overwhelm the operator's network capacity.

Today almost half (49.4%) the peak network capacity of the Internet is consumed by just two sites Netflix and YouTube. Both are streaming compressed video which is appropriate for filmed material. Computer generated content (cloud gaming, remote apps) that currently normally uses the same streaming technology as filmed material has the potential to use large amounts of network bandwidth with corresponding costs. It make sense to optimize the streaming by using more efficient techniques than can be used for filmed video. This is one of the major goals of Ascender's technology.

3 Difficulties By Design

The previous section describes difficulties intrinsic to cloud graphics, but there are also difficulties related to design decisions. In order to consider a concrete example of cloud graphics, we examine the current paradigm of cloud gaming, exemplified by the *Nvidia Grid* or AMD's *Radeon Sky*. In this design, there is an array of GPU's in the cloud. In the *Nvidia Grid* each GPU can support up to two remote cloud gamers.

Game applications are run in the cloud. The OpenGL code is executed and the graphics of each frame are rendered in the cloud into pixels. The pixels of each frame are encoded with a GPU-based H264 codec and then sent via the network to the remote cloud gamer. The H264 stream is decoded and displayed on the remote device.

3.1 Heavy Use Of Cloud Resources

The *Nvidia Grid* and the similarly designed AMD *Radeon Sky* are offerings from hardware manufacturers. Ever-more expensive hardware is being proposed to solve by brute force the difficult problems of cloud gaming. This self-serving approach by hardware manufacturers might not be in the best interests of cloud gaming providers.

It is hard to understand the great need to centralize the GPU hardware given the current trend of ever-more capable CPU's and GPU's in consumer devices. Nvidia itself is currently pushing the Tegra 4 processor with a quad-core ARM Cortex-A15 and a quite capable 72 core GPU.

3.2 Heavy Use Of Network Resources

The standard video codec, H264, is a bad choice for video streams generated by computer gaming. The reason for this relates to the principles of design of the MPEG family of video codecs of which H264 is a member and general principles of information theory. MPEG was designed to efficiently compress photographically generated video streams. The typical video streams presented to the codec are a small set of all possible video streams. Only video streams with characteristic spatial and temporal correlations that make sense to the human visual system are in the domain of MPEG compression. Essentially MPEG can handle any stream that can be cinematographically produced.

The subset of video streams which are generated by a computer game is a very restricted subset of the streams that MPEG can efficiently handle. Using knowledge of how the game's video stream is generated can enable significant efficiencies in the compression and transmission of the video streams.

A simple argument demonstrates these ideas. The published bandwidth for an *Nvidia Grid* H264 stream for a 30 fps refresh rate at 720p resolution is 600 KBytes/sec. This means that each hour of game play uses 2.1 GBytes of network bandwidth. The test case that we will examine as a typical game is the popular Android game **Temple Run 2**. This Android app is 31 MBytes. If we run this game remotely via the *Nvidia Grid*, for every 52 seconds of play we transfer as much video data as a complete download of the game app. In section 4 we describe ASCENDER'S rendering compression techniques and explain why it uses a fraction of the network bandwidth of H264.

Intuitively, there must be a better way to compress the video stream. After all, if we estimate that about half the downloaded app consists of compressed textures and vertices, and since the program is an OpenGL program, the video stream consists solely of geometric and graphical transformations of these textures. We will see in the next section that transmitting the compressed rendering stream, rather than the compressed pixel stream, leads to a much more efficient way of remotely exporting graphics.

3.3 Encoding Latency

H264 and the other members of the MPEG family were designed for an environment where the real time latency of the video image is not critical. For example: if the source is a DVD or Blu-Ray disk the compressed video is precomputed and completely available. An arbitrary number of frames can be pre-buffered to allow smooth decompression of frames. Live news feeds and sporting events can lag behind the real-time video stream by a number of seconds as long as the video stream flows smoothly. This latitude in the design of MPEG introduced the concept of group of pictures (**GOP**) which, as the name indicates, is a group of successive pictures. Decompression of a frame will typically reference frames, within the GOP, that appear both earlier and later than the current frame. This delays decoding of a frame until all the referenced frames have been received.

When an MPEG codec is used in interactive gaming many of the design assumptions are problematical and lead to unacceptable latency. Tuning the video codec to provide a low-latency stream adversely effects the compression efficiency and/or the video quality.

3.4 Image Quality

Compressed video compromises image quality for bandwidth. The compressed images are noticeably degraded.

3.5 Frame Rate

On the Android tablet (Nexus 7) used for testing, Temple Run 2 runs close to the modern standard of 60 fps. *Nvidia Grid* uses a frame rate of 30 fps, a compromise for high quality games.

3.6 Rendering Latency

There are graphics systems that use a remote rendering level protocol and local generation of pixels. X11 and its OpenGL extension GLX are well known examples. They unfortunately are not useful in environments where the round trip latency is large. If an OpenGL application is run remotely on a low latency local network using GLX, it is possible to achieve a frame rate of a few tens of frames a second. The same application run on a network with tens of milliseconds latency will yield an unacceptable frame rate. The reason for this is that each OpenGL command entails a round-trip delay while waiting for the return value of the routine. The OpenGL return value semantics must be dealt with in practical remote rendering protocols.

H264 video transport doesn't suffer from this type of rendering latency since the the protocol is simplex (transfers data in one direction only).

4 The Ascender Approach

In order to understand the approach taken by ASCENDER TECHNOLOGIES, the rendering level, Fig. 4.1, is examined. This figure corresponds to our case study of a cloud game, the popular Android game app **Temple Run 2**. Although in this case the virtual host environment is a virtual Android system, a Windows or desktop Linux game would not change any essential details of the implementation. With ASCENDER'S technology, the local device need not be Android but can be any device which supports the rendering level software, in this case OpenGL ES 2.0.

In contrast, Fig. 4.2 illustrates the system architecture of the *typical server-side pixel renderer*. Here we show remote execution of a non-mobile OpenGL game. Note that a hardware GPU is present on the remote device. Pixels are H264 encoded on the remote (cloud) device, sent via the network, decoded on the local device and displayed on the local framebuffer.

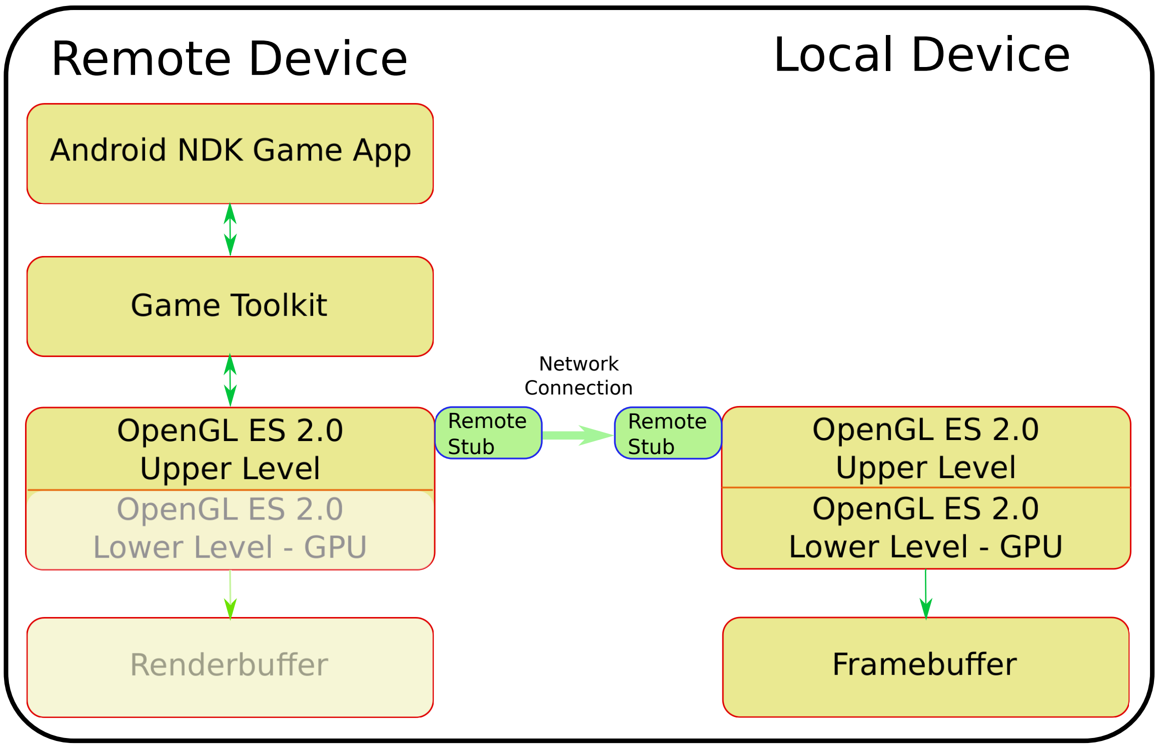

Figure 4.1: OpenGL ES 2.0 Android app

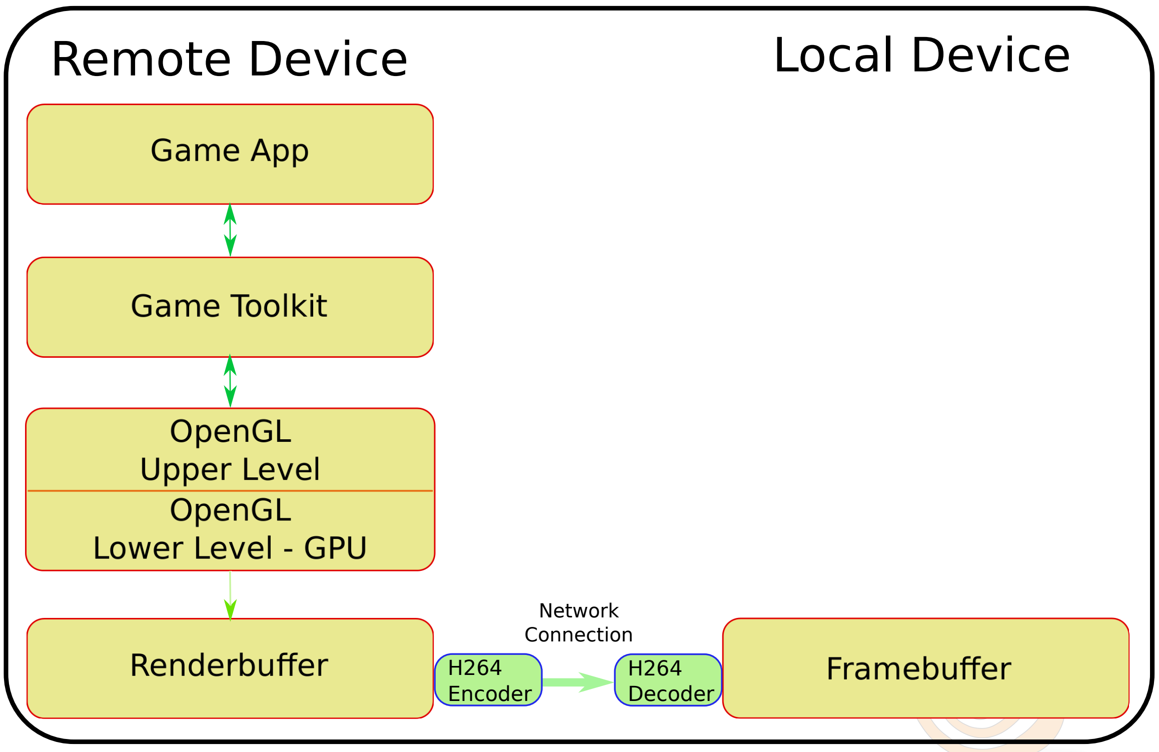

Figure 4.2: Cloud gaming - Server-side pixel rendering

4.1 No Remote Pixel Needed

Fig. 4.1 illustrates schematically the partition of the rendering layer into two levels, a general property of rendering software and not specific to OpenGL. On one hand, we don't want or need to render pixels on the remote device (cloud). On the other hand, the game toolkit might need to interact with the rendering layer. For OpenGL, shader objects must be compiled and linked, uniform and attribute variables must be queried and bound to locations, etc. This duplex interaction is indicated by the two-way arrow between the "Game Toolkit" and the upper layer of the renderer. Note that the interaction between the lower level of the renderer (OpenGL ES 2.0), and the Renderbuffer is one-way (write only). For this reason, the lower level of the renderer can be circumvented since rendered pixels are not needed at the remote location. Also note that the flow of renderer commands is one-way from the remote to the local device. Even if only one round trip (remote-local-remote) were needed per frame, a 50 msec ping will constrain the frames per second (fps) to be less than 20 fps.

There is a slight caveat to the one-way flow of renderer commands from the remote to the local device. During initialization of the connection between the remote-local devices, the local device can send information to the remote side that will define the local device OpenGL API extensions and implementation-dependent limits. In addition, it might send a list of the cached textures and vertices present on the local device. After this initial interchange the rendering stream is one-way.

This scheme can be implemented simply by taking a software-only OpenGL implementation, such as the open source Mesa software renderer and by deleting the code that actually renders pixels. The advantage of this approach is that by far the largest use of CPU time by the software renderer is in the pixel rendering. Hence, GPU hardware is needed in order to provide sufficient performance if the rendered pixels are required. However, if only the upper level API is needed, the computation load is low and GPU hardware is not needed. The upper level of the renderer must be run remotely to prevent render stream "stalls" caused by queries to the local renderer which would then entail round-trip latencies.

This general scheme applies to other renderers. Skia, the pre-Honeycomb Android renderer library renders pixels in software. The Android toolkit queries Skia when it runs measurement frames. Here too, the actual rendering of pixels can be circumvented when pixels are to be rendered remotely. By not actually rendering pixels, Skia takes only a small fraction of the CPU resources it would normally use. The upper level of Skia must be run remotely to answer measurement queries without incurring round-trip delays.

4.2 The Four Compromises - Uncompromised

ASCENDER'S technology delivers an **uncompromised solution** to remote graphic rendering: None of the four performance measures are compromised in ASCENDER'S novel "**Remote Rendering - Local Pixels**" technology. We compare server-side pixel rendering, such as the *Nvidia Grid,* with ASCENDER'S technology for the four performance measures.

4.2.1 Latency

Server-side Pixel Rendering**:** As discussed in section 2.4, some latency is unavoidable. Other latencies are caused by the choice of the encoding codec for the data stream. As described in section 3.3, the MPEG codecs are designed to have large encoding latency which contributes to efficient compression. For cloud gaming, this unintended consequence contributes to annoying latency.

Ascender: The rendering codec is "immediate": rendering commands are sent without significant delay. Since the compression algorithm works only with the knowledge of the previous rendering commands sent, the delay isw minimal. Once the last rendering command of a frame is encountered, the frame compression completes and sends the complete compressed frame.
This is in contrast with the MPEG class of codecs which contains B frames that are dependent on data from future frames, thus engendering large multi-frame latencies.

4.2.2 Resolution

Server-side Pixel Rendering**:** As the inevitable resolution of display devices increases, high quality pixel-based systems scale as the number of pixels on the device. Thus, an upgrade from 720p to 1080p will entail a bandwidth increase of two (see section 2.5). The next jump to a Nexus 10 level device will need an additional jump in bandwidth by a factor of two. An upgrade to a new 4K Ultra HD device will need an additional factor of two. This march of technology will force an increase in bandwidth by a factor of 8 in just a few years.

Ascender: As the resolution of the local display device increases, the bandwidth needed by the rendering stream remains essentially constant. The only difference might be an increase to larger pixel textures. The number of rendering commands in the render stream, which is the main determinate of bandwidth, remains constant.

4.2.3 Image Quality

***Server-side Pixel Rendering*:** With H264 compression-based systems there is always a complex balance between latency, bandwidth, resolution and image quality. A decrease in latency or bandwidth will cause a corresponding decrease in the quality and/or resolution of the video stream. The design criteria is normally “just good enough”.

Ascender: The compression of the rendering stream is lossless. Since the rendering process is performed on the local host and is identical to what would be performed on the remote host, there is no loss in image quality.

4.2.4 Frame Rate

***Server-side Pixel Rendering*:** Commercial cloud game suppliers offer 30 fps. True 60 fps, which is the modern graphics standard, would take double the current bandwidth.

Ascender: As will be shown in the next two sections (5.1 and 5.2), actual tests show that the bandwidth for an uncompromised 60 fps compressed render stream is less than an order of magnitude of the bandwidth required for a H264-720p-30fps stream, which compromises all four performance criteria. A H264-4K-60fps stream will entail a bandwidth increase of 16x. The bandwidth for an equivalent resolution and frame-rate using Ascender’s techniques are less by more than two orders of magnitude.

4.3 Ascender’s Remote Rendering Technology

Ascender’s approach to remote graphics differs from the accepted prevailing solution which encodes the pixel stream with standard lossy video codecs designed to encode photographically generated video streams. In contrast, Ascender’s lossless compression algorithms are designed for programatically rendered graphics and are therefore many times more efficient than pixel-based methods for rendered graphics.

4.3.1 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 exploits probable correlations within the data. Examining a similar compression domain provides motivation to uncover techniques for compression within our domain.

4.3.2 Modeling Photorealistic Video - MPEG

Video streams, whether photographically acquired, 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 continuous movement instead of a sequence of images. The effect is dependent on visual continuity between frames.

By their nature, MPEG video streams are amenable to lossy compression.

4.3.3 Modeling Rendering Compression

An analysis along the lines of the MPEG example can be performed to construct a compression model that efficiently compresses the remote-local rendering stream. 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 synthesized images is the human visual system.
- The subject matter, while possibly 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 images are generated frame after frame by repeated invocation of GUI procedures by the application software.

4.3.4 Procedural Compression

This data model can lead to the following compression technique that we call **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 the application is running a bounded amount of code. Conditional logic within routines can generate different execution paths, but even here the number of paths in practice is 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 a **data dictionary** of the 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.

An example from Android's Froyo contact manager illustrates these ideas. Here the rendering API of interest is the Skia software renderer. Fig. 4.3 shows a screenshot of a contact screen with seven entries. When we look at the rendering stream, we see that the code to compose one contact entry (as shown in Fig. 4.4) is run seven times in two variants. The first and the last two, 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 proceedure 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.

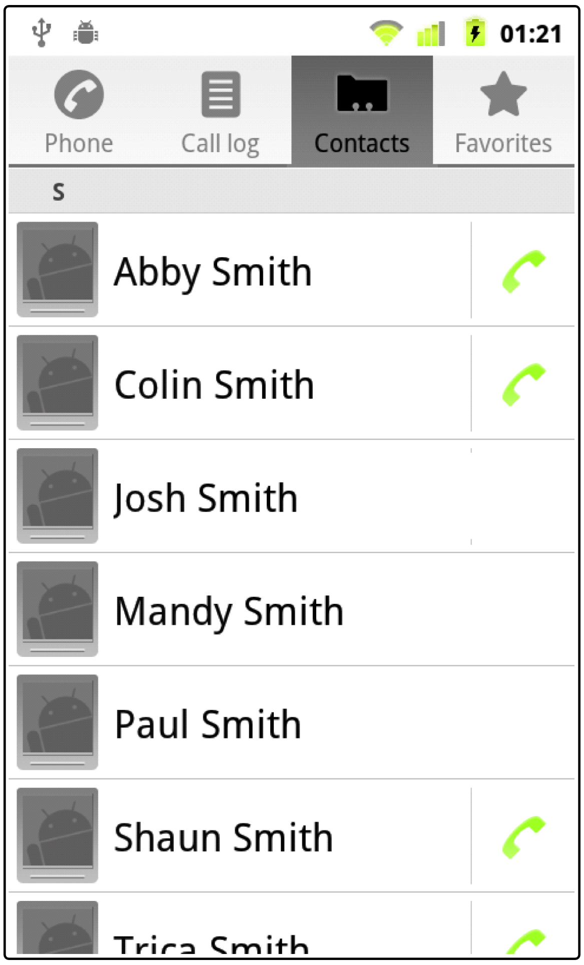

Figure 4.3: A contact list

Figure 4.4: A contact

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.

4.3.5 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. In this case the rendering API is the Skia rendering library. The rendering stream has balanced **`save()`**-**`restore()`** pairs for each frame of the rendering stream. A **`save()`** is found at the beginning of a GUI function and a **`restore()`** is found at the end of each GUI function. This bracketing information can be used to reverse engineer individual GUI and application procedures. It also reveals 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.

The ability to model better the rendering stream code source allows improved prediction of the rendering stream and increases 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 which are a mix of toolkit and application level procedures.

ASCENDER'S compression algorithm was tested on the rendering trace of a 60 frame sequence for the application shown in Fig. **4.3**. 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 to 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.

ASCENDER'S results show that compression can be an effective means to decrease the amount of data transmitted for a rendering stream. A compression of 1:100 compared to a raw render stream is typical. It should be appreciated that this compression is in addition to the approximate 1:100 compression ratio obtained from exporting graphics via the rendering stream as opposed to a raw video stream. The effective compression as compared to a MPEG (H264) stream is typically between 10 to 50.

4.3.6 Unstructured Procedural Compression

OpenGL rendering streams have a different structure than Android's other Skia-like rendering API's. In the Skia influenced API's, there are nested save/restore pairs that delineate rendering routines. The only reliable marker in a OpenGL stream is the end-of-frame (eglSwapBuffers) routine. A compression algorithm similar to the LZW (Lempel–Ziv–Welch) algorithm, which works on unstructured streams, can be used. Here too a remote-local synchronized dictionary can be dynamically constructed to enable compression in a manner very similar to the previous structured example.

4.3.7 Bitmaps, Textures and Vertices

Bitmaps, textures, and vertices are objects that are referenced by rendering API's repeatably and can be quite large. For Skia-like rendering systems, bitmaps are used. For OpenGL, textures and vertices are used. These elements are sent once and typically referenced many times. Care should be taken that these objects are sent only once and are cached for subsequent references.

If these objects are not available when referenced, there might be a delay (i.e. a stall) of the frame while the object is being loaded. It is therefore important to schedule the pre-caching of objects before they are referenced. If the cloud server repeatedly runs an application, a statistical model of object usage can be developed. This probabilistic graphical model allows highly accurate pre-caching of objects with a low probability of rendering stalls due to data dependencies.

4.3.8 Comparison between MPEG and Procedural Compression

MPEG Compression: MPEG uses lossy compression: within each frame the compression is similar to JPEG. It also uses interframe coding by encoding only the differences between nearby frames, both in the past and in the future. This forward interframe dependence can add undesired latency to the MPEG codec. The codec is asymmetrical, with compression being more computationally intensive than decompression. The most computationally intensive part of MPEG compression is the two dimensional search for motion vectors. Real-time compression is only practical for systems with specialized hardware to perform the heavy computations needed. MPEG normally requires a fairly constant bandwidth. Since the window of interframe comparisons is small, for low-latency MPEG compression, bitmaps or textures that make up the video images are effectively repeatedly sent from the remote to the local system. This is very wasteful of network bandwidth.

Procedural Compression Compression: Rendering stream compression is computationally simple. The raw rendering API stream itself is much smaller than the raw video stream, typically by two orders of magnitude. The search in the dictionary for previously encountered sequences is one-dimensional and uses only previously encountered sequences in the stream. The complexity is similar to LZW (Lempel–Ziv–Welch) compression. No special hardware is needed. Bitmaps or textures are sent once and can be used to compose hundred of frames.

5 Use Cases

ASCENDER'S remote rendering technology is applicable to many systems and platforms. It can apply to Android or a wide range of other systems. It is not necessary that the remote server and the local device have the same architecture: an Intel server can provide services for an ARM device. Below are scenarios of some use cases.

5.1 Use Case: Cloud Gaming

Cloud gaming is a high profile service whose prevailing design has been discussed in some detail in section 3. In order to test the performance of ASCENDER'S remote technology, a cloud gaming test case was examined.

	Value	Units
Frame Rate	44	frames/sec
Run Time	90	sec
Frames	3825	frames
Rendering Commands	1043507	OpenGL API commands
Compressed Textures Vertices	5	MBytes
Compressed Bytes per Frame	873	bytes
Rendering Commands per Frame	272	OpenGL API commands/frame
Bytes Per OpenGL Command	3.2	bytes
Video Bandwidth for Play	38	KBytes/sec

Table 2: Performance of Temple Run 2

5.1.1 Test Case - Temple Run 2

Temple Run 2 is an “endless running” video game based on the Unity3D game engine. For this test, the Android version was used, with an OpenGL ES 2.0 rendering API. The size of the app download is 31 MBytes.

The game ran at about 44 fps on the remote client a real, non-virtual, Nexus 7 tablet. The first “run” (until level two is reached) of the game took about 90 seconds, contained 3825 frames and had about 1 million rendering commands. The game itself pre-loaded approximately 5 MBytes of compressed textures and vertices, all textures needed for the full game during the first frame. Approximately 873 compressed bytes/frame is needed for the game play, excluding the downloaded textures and vertices. On the average each OpenGL rendering command takes 3.2 bytes compressed. The networking bandwidth needed for the remote graphics is about 38 KBytes/sec. In addition, an AAC audio stream will add something on the order of 35 KBytes/sec (which ironically is very similar to the bandwidth of the video stream). These values are summarized in Table 2.

The rationale for separating the bandwidth needed for the download of textures from the bandwidth devoted to “game play” is that the textures are downloaded once, but the game play might continue for quite a while after all the textures are downloaded. If the local device has persistent storage, the textures needed for the game can be cached so the 5 Mbyte download need not be performed for the next game invocation. Even if the textures need to be downloaded for each game invocation, the time that the Android application takes to initialize the game on a local quad core Nexus 7 is about 8 seconds - the approximate time needed to download the 5 Mbytes of textures via a nominal 600 KBytes/s network connection to the remote client. For this reason, loading the textures will not change significantly the start-up time in the cloud gaming case.

5.2 Use Case: Android Apps

5.2.1 Motivation

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. Android apps are normally installed on physical devices, a stateful model. As such, each android device is personalized and maintained, usually by the owner.

The apps of cloud based systems are normally loaded dynamically via the network on demand, and not normally persistently installed on the device. After the current session terminates, the state is reinitialized. No state is saved between sessions. The advantage is that all cloud-based devices are in a sense equivalent and non-personalized: anyone can sit at any cloud device, login and work in their personal environment. The system is maintained professionally and centrally. In addition, there are many security advantages to cloud-based systems.

These two computer paradigms, stateful and stateless, are difficult to harmonize. A good example of two systems that do not interoperate well are Google's:

- ○ Android - Mobile OS
- ○ ChromeOS - Cloud OS

Integrating ChromeOS apps on Android is easy since the Chrome browser runs under Android. However, until now, running Android apps under ChromeOS was an unsolved problem, aka the "convergence" problem of Chrome OS and Android.

ASCENDER'S remote rendering technology can effectively "converge" Chrome OS and Android. Android apps can be run in the cloud and the graphics can be efficiently "exported" using ASCENDER'S enabling technology.

5.2.2 Technical Challenges

The number of rendering API's supported by Android is large and varied. Some are now enumerated:

① **OpenGLRenderer.cpp** If hardware renderer is enabled, the Canvas.java class uses the hardware renderer as shown in the somewhat simplified view in Fig. 5.1. The C++ class largely retains the Skia-like API of Canvas.java and has a similar API to the software renderer Canvas.cpp (④) and the Skia library (③).

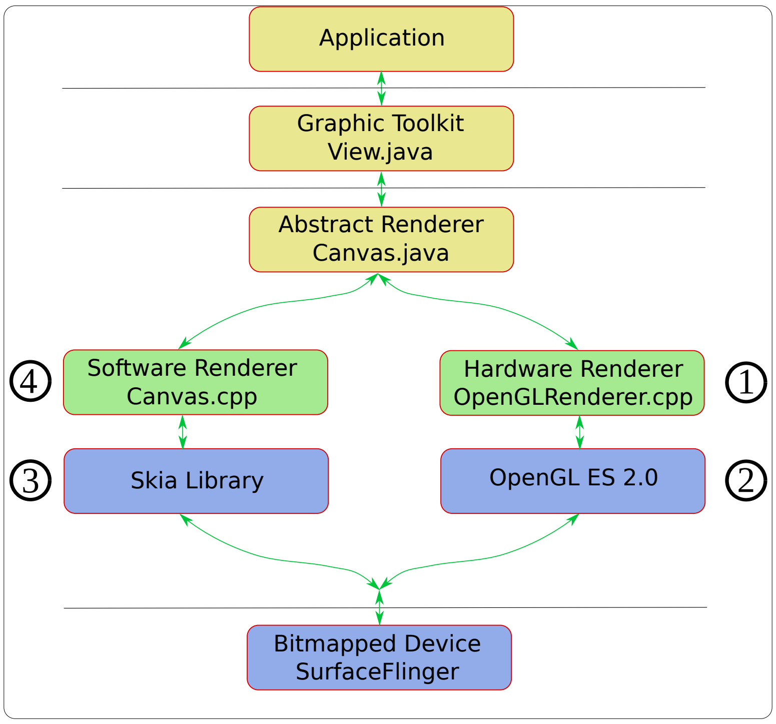
Application
Graphic Toolkit View.java
Abstract Renderer Canvas.java
④ Software Renderer Canvas.cpp
Hardware Renderer OpenGLRenderer.cpp ①
③ Skia Library
OpenGL ES 2.0 ②
Bitmapped Device SurfaceFlinger

Figure 5.1: The Android Graphics Stack

② **OpenGL ES 2.0** The lower level of the hardware rendering stack is the 3D rendering standard managed by the non-profit technology consortium Khronos Group. This rendering API is used for Android's GUI when the hardware rendering path (the right path) is taken.

③ **Skia_Rendering Library** The Skia library is used for the software rendering of pixels.

④ **Canvas.cpp** This is the software renderer that the Canvas.java class uses when the software rendering path (the left path) is taken.

⑤ **OpenGL ES 1.x** There are programs that use this earlier incompatible version of OpenGL ES.

The reason that all these rendering API's need to be considered relates to the "coverage" of the large number of apps within the apps stores. The more rendering API's that are covered, the greater the coverage of the apps within the stores.

For example: an Android 3D game might use the OpenGL ES 2.0 API as a NDK (Native Development Kit), as does Imangi's Temple Run 2, which uses the Unity3d game engine. The rendering API used is OpenGL ES 2.0.

Ascender implemented remote graphics and tested the 5 graphical rendering API's mentioned previously. Support for these API's will cover a large number of the Android apps in large repositories such as Google Play. Figure 5.2 shows the coverage of these five graphical rendering APIs mentioned previously. This figure shows the coverage of these five API's of an app repository. This figure is a Venn diagram for a finite collection of sets, ①, ②, ③, ④, ⑤ and ⑥. Some comments on this figure are appropriate.

- The ① (OpenGLRenderer.cpp) renderer API is completely covered by the ② (OpenGL ES 2.0) API. This means that support for the ① renderer is not critical. On the other hand, the bandwidth that this renderer uses is less than the bandwidth of ②, so it pays to implement it for efficiency reasons. ① should be used instead of ② if possible.
- NDK apps that directly use ② are in subset ② − ①.
- NDK apps that use OpenGL ES 1.x are in the set ⑤.
- The figure 5.2 is somewhat simplified. For example: there are applications that use both ② OpenGL ES 2.0 and ⑤ OpenGL ES 1.x. Thus ② ∩ ⑤ ≠ ∅, not as drawn in the figure.
- Apps that are in Google Play and not supported for remote rendering, via the implemented API's, are in the subset ⑥ - (① ∪ ② ∪ ③ ∪④ ∪ ⑤).

5.2.3 Standard Android UI Programs

The majority of Android apps are written in the Android UI toolkit in Java. Simple examples are the standard contact manager or the settings manager in Android. In the case of hardware rendering, the API stream generated is a number of times more compact than the OpenGL ES 2.0 stream. Each rendering command is typically compressed into 2-4 bits. Even at 60 fps, the rendering stream is typically less than 20 KBytes/sec.

The reason compression is so effective on the OpenGLRenderer.cc stream is that the objects sent in the compressed rendering stream are entries from the synchronized "routine dictionary" on the remote encoder and the local decoder. These routines map to higher level objects from the application and toolkit levels that contain many renderer commands. Effectively, dynamic analysis of the rendering stream provides a reverse engineering of the graphical routines of the application and toolkit.

In practice, even zlib compression applied to the rendering stream will provide excellent compression and consume a small amount of resources.

5.2.4 Non Graphical API's

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

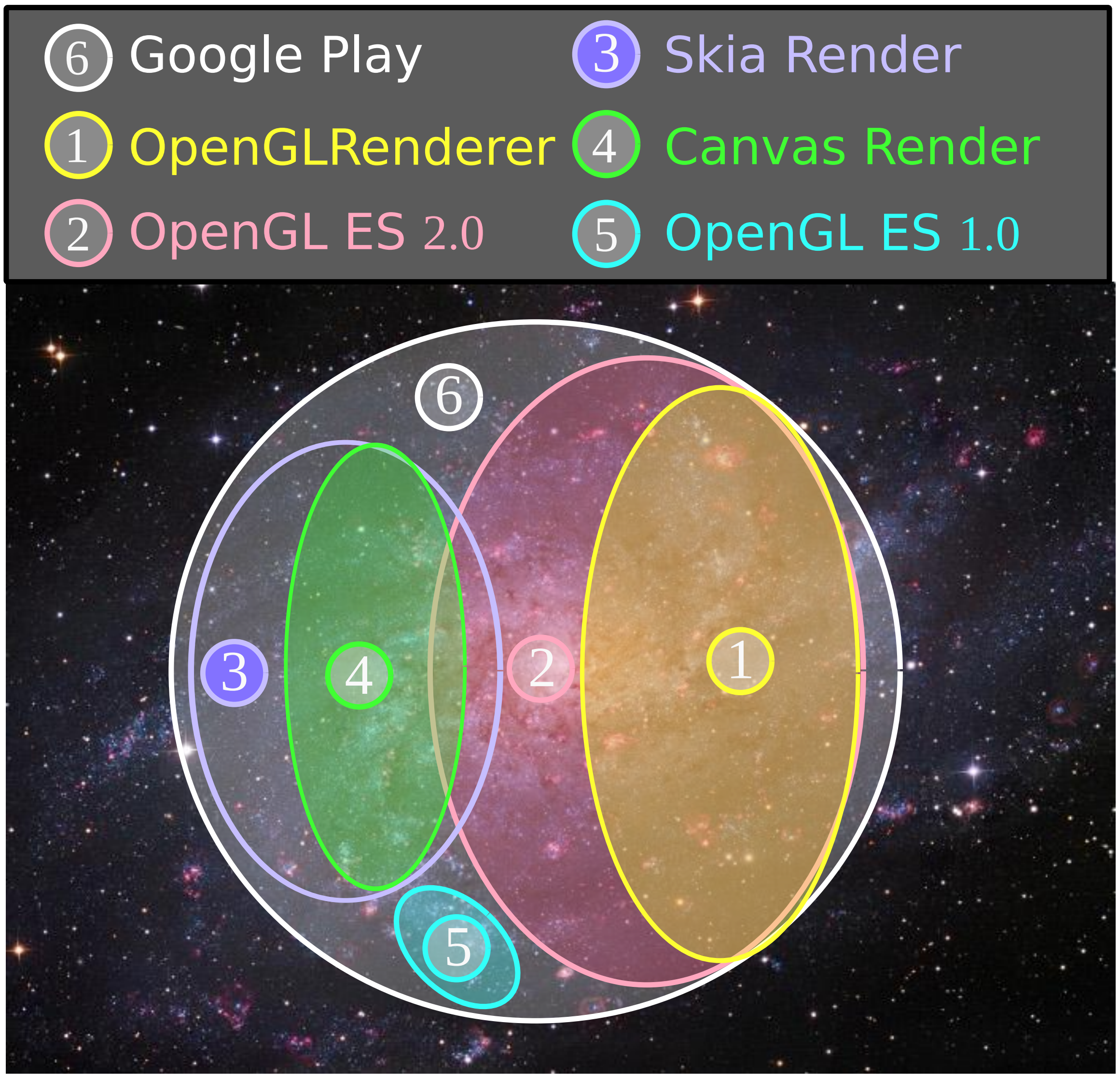

Figure 5.2: A Galaxy of a Million Stars (Google Play)

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.

5.3 Mobile Devices

The previous use case, section 5.2, can be extended to provide remote Android apps to mobile devices. There are a number of relevant use cases for mobile devices.

5.3.1 Android apps on non-Android platforms

Android's share of the mobile market is approaching 80%. IOS has an additional 13%. The remaining small market share makes it very difficult to attract developers for less popular platforms. Without developers and their apps it is difficult to grab market share. This classic "chicken-and-egg" problem of circular dependency makes it very difficult for a mobile platform to reach critical mass starting from a small market share.

Ascender's technology provides a solution to this dilemma. By enabling the 1 million Android apps that are currently available on the non-Android mobile platform, customer resistance to purchase of a less popular platform is decreased. The ability to host Android applications efficiently in the cloud using minimal resources and to transmit graphics using minimal bandwidth makes this an attractive proposition for the mobile operator.

5.3.2 BYOD in an enterprise environment

Bring your own device (BYOD) or bring your own phone (BYOP), is making significant inroads in corporate environments. One of the vexing problems of BYOD is how to secure corporate data when it has been loaded on to a device that is not part of corporate infrastructure. Another problem is that organizations might be forced to support apps for many phone platforms.

Running the app in the cloud will keep the sensitive corporate data in the corporate cloud rather than in the employees device. An app Android app running in the cloud can be accessed on a wide array of devices and platforms, mobile and fixed. This will provide the "write once, run anywhere" capability of Java fame.

Our technology allows the app to run in the cloud thereby keeping the sensitive corporate data in the corporate cloud rather than in the employee's device. Android apps running in the cloud can be accessed on a wide array of devices and platforms, both mobile and fixed, providing true "write once, run anywhere" capability.

5.4 Remote App Server

There are a number of variations on a 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 separate linux containers environments for each client. In this configuration each local client would have private contact lists.
- 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.

5.5 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.

5.6 Purchase / Rental Models

In a purchase/rental model, apps can be demoed remotely, prior to purchase, and only bought if satisfied.

Amazon's Android Appstore has a "test drive" option that allows remote testing of an app before purchase. It launches a virtual copy of Android in the Amazon EC2 cloud for up to a half an hour for a test of the app. The graphics are exported via a H264 stream. Amazon's solution suffers from all the difficulties mentioned in section 3.

5.7 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.

5.8 Set-Top Boxes

Many set-top boxes cannot practically run Android since the set-top box might not have sufficient resources or 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 apps in the operator's "cloud" would greatly simplify system maintenance.

5.9 Automated Testing

Automated testing of graphical applications is needed to ensure that applications meet specifications and perform properly. OpenGL applications typically query the viewing device as to the API extensions supported and implementation-dependent limits. The generated rendering stream is dependent on the results of the queries. The number of possible configurations is combinatorially daunting.

A simple example demonstrates: Ten OpenGL ES 2.0 extensions were reported from a query of a Nexus 7 (Tegra 3, Nvidia GPU) running 4.2.2 Android. On an Ubuntu Linux workstation with an Mesa Gallium OpenGL stack, the number of extensions reported was 30. Surprisingly, there were **no** extensions in common between these two systems! In addition, there are 32 different parameters defining the OpenGL ES 2.0 implementation-dependent limits which are different for each device.

Repeated testing for a large number of devices is difficult. However, with ASCENDER'S technology, it is possible to automatically run the application on a large number of device configurations and capture the rendering stream. This procedure can be performed in the cloud for hundreds of instances of device configurations. The resulting rendering streams can create a graphics transcription of the test execution and be tested for proper operation. The graphics can be replayed to see the visual state of the application when a test fails.

5.10 WebGL Browser Implementations

It is possible to write a WebGL client for the compressed rendering stream that gives access to a whole range of applications and games in a platform and browser agnostic fashion.

This solution will accrue the benefits of ASCENDER'S technology and should be contrasted with H264, VP8 or OTOY's ORBX.js which all suffer from the problems of rendering pixels in the cloud.

6 In Summary - A New Cosmic Alignment

ASCENDER'S novel enabling technology which provides remote access to modern graphical applications has been presented. We showed that this technology provides significantly better performance while using a fraction of the resources of the currently accepted practice.

Looking at broader trends, there is now a convergence of technological advances, ASCENDER'S technology being one component, that can effect major changes in cloud computing:

- Fast, low-power 64 bit ARM multi-processors (e.g. Cortex A50) with virtualization extensions.
- Adoption of Android apps in a broad gamut of use cases, including the enterprise.
- Ever increasing adoption of cloud based solutions.
- The introduction of WebGL browser technology enabling web-based OpenGL ES 2.0 apps.
- ASCENDER'S method of transporting modern graphic interfaces.