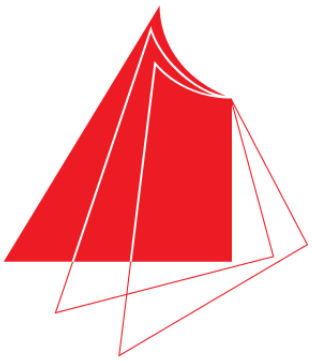


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Introduction to the Android Graphics Pipeline

Wissenschaftliche Arbeit zur Erlangung des Grades *Bachelor of Science (B.Sc.)* in Informatik an der Hochschule Karlsruhe - Technik und Wirtschaft

Mathias Garbe

Karlsruhe, im März 2014

Betreuer:	Christian Meder
Referent:	Prof. Dr. Holger Vogelsang
Korreferent:	Prof. Dr. Ulrich Bröckl

Statement of Authorship

I hereby declare that this bachelor thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne unerlaubte Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Karlsruhe, March 17, 2014.

Abstract

This bachelor thesis outlines the current state of the Android graphics pipeline. The graphics pipeline is the part of Android which displays the user interface by using the graphics processor.

This thesis begins by describing the principles of current graphics hardware. Afterwards the different hardware architectures of modern graphic processors are explained.

The main part of the thesis starts by looking at the history of the Android graphics stack. An explanation of specific optimizations employed by Android and their software implementation follows. The Android graphics pipeline is then explained by demonstrating a sample application and tracing all drawing operations down the pipeline to the actual rendering calls. Finally, current issues and problems like driver support and overdraw are addressed.

Kurzfassung

Die vorliegende Bachelorarbeit erläutert den aktuellem Stand des Android-Grafikstacks. Hierbei handelt es sich um den Teil von Android der Benutzeroberflächen mithilfe des Grafikprozessors darstellt.

Zunächst wird der Aufbau und die Funktionsweise aktueller Grafikhardware erläutert. Anschließend werden die verschiedenen Architekturen moderner Grafikprozessoren beschrieben.

Im Hauptteil wird die Historie des Android-Grafikstacks betrachtet. Darauf aufbauend werden konkrete Optimierungen und Implementierungsdetails von Teilen des Grafikstacks erklärt. Anhand eines Beispielprogrammes wird die Android-Grafikpipeline untersucht und abschließend werden die Auswirkungen aktueller Probleme und Themen wie Treibersupport und Overdraw getestet und bewertet.

About the Source Code Listings

The source code listing in this thesis do not aim to be complete. To make the listings more clear, `imports` and `includes` are omitted. In some cases, source code was converted to pseudo-code. Likewise, no source code listing is intended to be compiled or executed. All listings are solely for illustration purposes and need to be viewed in their respective context.

Acknowledgments

During the course of this thesis, several persons have contributed to whom I owe my gratitude.

Foremost, I wish to thank my supervisor, Christian Meder, for his encouragements and keeping the thesis on track and within scope.

I wish to thank Tim and Mark for their continuous interest and input on the thesis. Their inquiries have always helped me considering new approaches and generally driving this thesis forward.

I wish to thank Benjamin, Erik, Siggy, Heike, Florian and Daniel for proof-reading and correcting countless versions of this thesis and always making excellent suggestions on how to improve it.

Last but not least, I would to express my sincere gratitude to Anika, my family and all my friends, who have always motivated and supported me with encouragement and consolation.

Contents

1. Introduction

The Android operating system has a wide base of users and application developers. Android's Application Programming Interface (API) for writing applications is well documented and many tutorials are available on the web.

Unfortunately, documentation of the actual Android operating system is scarce. The inner workings of the graphics pipeline in particular are only ever mentioned by Google employees in blog posts and on conference talks. This thesis aims to provide a basic overview of the graphics pipeline.

This thesis is targeted at interested developers with prior knowledge of the Android framework. Readers can also use the official Android framework documentation[1] and the Android Open Source Project (AOSP) website[2] as a reference.

The analysis and breakdown of the Android source code is based on Android 4.4.2 r2 (`android-4.4.2_r2`[3]). All source code listings are also based on this Android version.

1.1. inovex GmbH

inovex GmbH[4] was founded in 1999 in Pforzheim, Germany. Since then, it has been managed by its sole owner and founder, Stephan Müller. inovex GmbH has about 13 employees, with offices in Pforzheim, Karlsruhe, Cologne and Munich.

The company is focused on contract work and is divided into multiple lines of business, which are Application Development, Business Intelligence, IT Engineering and the

[1] `http://developer.android.com/`
[2] `http://source.android.com/`
[3] `http://source.android.com/source/build-numbers.html`
[4] `http://www.inovex.de/`

inovex Academy. This thesis was written with the Android embedded development team, which is a part of the Application Development line of business.

Lectures at symposiums and published works in professional journals also have a high significance for its public image and are highly encouraged. The Droidcon and the Mobile Tech Conference are just two examples where inovex employees appear as regular speakers.

2. Introduction to the Architecture of Modern Graphic Processors

Today's Graphics Processing Units (GPUs) differ greatly from early ones. While these typically only supported accelerated 2D drawing and bitmap blitting, modern GPUs have a much broader field of application. These early GPUs had hardware support for basic 2D drawing and animations. In practice this is copying image data from a bitmap to the framebuffer at a specific location, scale and rotation. Quite surprisingly, these early GPUs also introduced the hardware cursor to unburden the operating system and Central Processing Unit (CPU). This meant a huge boost in performance for this time and made non-text-based games possible.

Modern GPUs still have all these basic 2D hardware features hidden somewhere on the chip itself (the die) [Gie11a], but their main focus has shifted quite heavily. Nowadays, GPUs provide accelerated 3D rendering and hardware video decoding. Recently, General-Purpose computing on Graphics Processing Units (GPGPU) also got popular. This is possible because a modern GPU is basically a huge number of parallel floating point processing units bundled together. After all, 3D rendering in its purest form is only matrix and vector calculus.

These processing units themselves have a Single Instruction Multiple Data (SIMD) like instruction set which allows vector calculation using very few instruction cycles. For example, the multiplication of $\vec{a}$ with $\vec{b}$ followed by the addition with $\vec{c}$

$$(\vec{a} \times \vec{b}) + \vec{c}$$

can be performed in one cycle with the Multiply-Add instruction (MAD) instruction.

Modern CPUs do have comparable features, with the latest addon being the Advanced Vector Extension (AVX) [Int08] instruction set. Nevertheless, they are still much slower because of the simple fact that their processor count is much lower.

To fully understand the Android rendering optimizations and pipeline a low-level understanding of GPUs graphics pipeline is necessary. Because no vendor is very specific about the internals of their GPU architecture, one has to sift through marketing presentations, blog posts and white papers to find the relevant pieces of information. Therefore, most of the information presented here is to be considered a simplification of what the hardware actually does.

2.1. GPU Vendors

Vendor	**2013**
Intel	62.0%
AMD	21.9%
Nvidia	16.1%

Table 2.1.: Market Share of desktop GPUs. [Jon14a]

The traditional desktop market is seemingly dominated by Intel, but this is due to the fact that since 2010 every new Intel CPU has an integrated GPU called Intel HD Graphics. While these GPUs can handle simple 3D rendering, more complex scenes and computations still need a discrete GPU. AMD's market share is influenced by the fact that Bitcoin miners prefer the AMD Radeon GPUs due to a much faster hash calculation compared to Nvidia's GeForce series. [Kha14]

Vendor	**2013**
Qualcomm	32.3%
Imagination Technologies	37.6%
ARM	18.4%
Vivante	9.8%
Nvidia	1.4%
DMP	0.5%

Table 2.2.: Market Share of mobile GPUs. [Jon14b]

The mobile market is more diverse and all of the vendors sell their GPUs as Intellectual Property (IP) Cores, which will be bundled with other components like the CPU and main memory on a System on a Chip (SOC). Qualcomm's Adreno, which was originally bought from AMD, and Imagination Technologie's PowerVR are the biggest players in the market to date. Imagination Technologies owes its big market share to Apple, as all their iOS devices are powered by an PowerVR GPU.

Quallcomm Adreno is currently the only GPU that is being used with Windows Phone 8 devices. Nvidias Tegra is still not very popular, with the newest devices being Nvidia Shield and Nvidia Note 7 from 2013.

2.2. Overview of the OpenGL Rendering Pipeline

In order to render objects to the screen, the Open Graphics Library (OpenGL) defines a sequence of steps. This is called the OpenGL Rendering Pipeline (Figure 2.1). This pipeline is described in a general, low level way here.

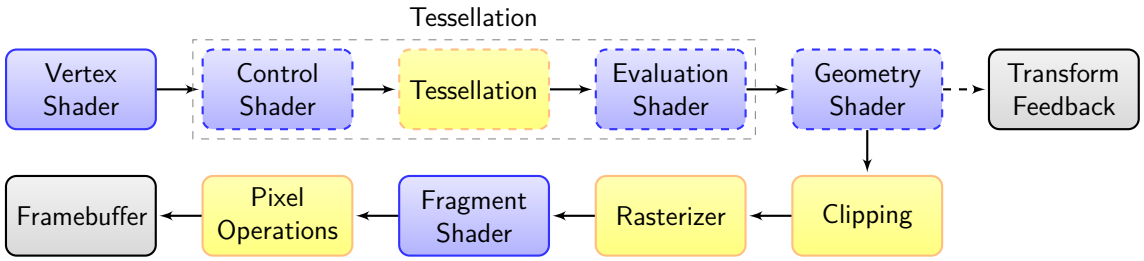


Fig. 2.1.: Simplified OpenGL Rendering Pipeline overview. Programmable stages are colored in blue and optional stages are marked with dashes. (Image based on [Ope13d])

It is important to mention that the hardware implementation can differ greatly and is not defined by the OpenGL API. Neither is the architecture of the renderer (section 2.3) defined, nor whether it implements optimizations such as an Early-Z (see glossary), Hierarchical Z-Buffer (Hi-Z, see glossary) or Hidden Surface Removal (HSR, see glossary) pass. But regardless of how the hardware is designed and what kind of optimizations are used, it has to make sure that the output is exactly the same as the OpenGL standard dictates.

2.2.1. Vertex Generation

Graphics APIs like OpenGL or DirectX store their geometry data as a collection of primitives like triangles or quads. Each of these primitives is defined by a set of vertices. To render these, the API provides a stream of Vertex Attributes for the Vertex Generation stage. These attributes typically contain the 3D coordinates (in model space), surface color and a normal vector. [FH08b, p. 52]

2.2.2. Vertex Processing

After the API submitted its geometry stream to the hardware, the vertex processing stage is executed. For each vertex, which is composed of a series of Vertex Attributes [Ope13f], the Vertex Shader (VS) is executed. The VS is the first programmable stage in the rendering pipeline and for every processed input vertex it must output an vertex.

Typically, this is where the Model-View-Projection Matrix is applied to the vertex, which transforms its position from model coordinates to screen coordinates. The color and normal information of the vertex are stored for processing in the Fragment Shading stage.

2.2.3. Primitive Generation

Since OpenGL version 4.0, the vertices are then passed to the programmable tessellation stage. The purpose of the Tessellation Control Shader (TCS) is to subdivide primitives into patches, thus increasing the detail of the surface. It is also responsible for ensuring that these new patches do not have gaps and breaks, which can occur during subdivision. The Tessellation Evaluation Shader (TES) is responsible for the calculation of new vertex attributes for the position, color and normal vector of the newly generated primitives. [Ope13e]

If no tessellation stage is defined, the output from the vertex shader is passed as a primitive to the next stage.

2.2.4. Primitive Processing

Each primitive is handed to the Geometry Shader (GS), which outputs zero or more primitives. The type of input and output primitives does not have to match - the GS could output a list of triangles generated from a list of points. A typical use-case for the GS is animating of a particle system, or creating of billboards from a list of points. [Ope13b]

If no geometry shader is defined, the primitives are directly passed to the next stage.

This is also the stage where Transform Feedback (also called Stream Out) happens: the pipeline is writes all primitives to the Transform Feedback Buffer, which can then be cached and rendered again. This is useful when a TCS or GS does heavy computations, like tessellating a model.

When animating particles, this can also be used to save the current state of all particles so that the application can read the current positions. In the next frame, this new state is passed to the GS which in turn updates all particles.

2.2.5. Primitive Assembly and Fragment Generation

All primitives need to broken down into individual points, lines or triangles. The Primitive Assembly converts Quads and Triangle Strips to triangles, which are then passed to the rasterizer. All primitives can also be discarded with the `GL_RASTERIZER_DISCARD` option, which prevents the primitives from being rasterized. This can be used to debug performance issues in the previous stages, but also to prevent rendering of primitives generated with Transform Feedback. [Ope13c]

The primitives are handed to the Fragment Generation stage, also called “Rasterizer”, which converts the primitive to a list of fragments which are passed to the Fragment Shading stage. Before modern GPUs, rendering was commonly done in software with a scanline algorithm. This rasterized the geometry line-by-line and than shaded each pixel [Hec95]. While this was a good approach on general purpose hardware like the CPU, it does not transform well to a hardware implementation.

Juan Pineda presents a hardware-friendly approach in [Pin88]. The basic idea is that the distance to a line can be computed with a 2D dot product. In order to rasterize a triangle, all candidate pixels (which can be determined with a bounding shape) are tested against all three triangle edges. If the pixel is inside all edges, the pixel is marked as inside and is handed to the fragment shading stage.

This can be further optimized and implemented in a parallel fashion. The AMD chip inside the XBox360 uses 8×8 sized tiles, which are rasterized in parallel [AHH08, p. 862]. Modern GPUs use another, more coarse tile rasterizer which precedes the main one, in order avoid wasted work on the pixel level. This rasterizer only tells the fine rasterizer if a tile is potentially covered by a triangle [Gie11d]. High-performance GPUs employ multiple of these rasterizer combinations.

2.2.6. Fragment Shading

All visible fragments are passed to the Fragment Shader (FS). The main responsibility of the programmable FS is to write an output color for the processed fragment, but it can also write output depth and even multiple colors to multiple buffers. The color can be calculated in a number of ways, incorporating textures, light, global illumination and other factors. Per-vertex attributes, which are passed from the VS, are interpolated to the fragment position. Also, the FS has access to the `discard` keyword, which can be used to discard a pixel, preventing it from proceeding further down the pipeline [Ope13a].

Each fragment is independent from another, they can be executed with the FS in parallel. Modern GPUs have a number of shader cores dedicated to this. Fragments are bundled as multiple batches and then processed by the shader cores. As these cores operate in a lockstep fashion, dynamic branches in shaders are expensive. If only one core takes a branch, all other shader cores need to execute the branch too, even though they will discard the result anyway [Gie11e].

2.2.7. Pixel Operations

After each primitive is processed by the FS, they have to be merged into a final image. This stage is also responsible for blending and pixel testing, which needs to happen in the same order as the application sends the primitives. Any hardware optimizations of the drawing order need to be reverted here. Both blending and pixel testing are relatively cheap computations, but have a high bandwidth requirement as they need to both read from and write to the render target.

Blending happens in a fixed-function block which is not yet programmable but only configurable (section 2.7). To blend two pixels together, the current color must first be read from the render target. According to the configuration, the color is blended with the process fragment and written back to memory. This is an important step when rendering translucent surfaces. Pixel tests include stencil, scissor and depth testing, which can be turned on or off by the graphics API.

2.2.8. Framebuffer

The resulting image is written to a framebuffer, which can in turn be swapped to the display or used as a new texture. Figure 2.2 shows an example of an operation in the graphics pipeline.

2.3. Pipeline Variants

Currently, there are two pipeline variants, which are both very old and date back to the same time periods of early GPUs. Vodoo, Nvidia and ATI started out with a straightforward and easy to implement Immediate Mode Renderer (IMR), and PowerVR used the more complex Tile-Based Renderer (TBR) approach. Nvidia and ATI/AMD still use the immediate-mode approach and are the main vendors for desktop GPUs. On mobile chipsets, the tile-based approach is more common, due to power, heat and space constraints. The Nvidia Tegra is the only mobile chipset that uses immediate-mode rendering.

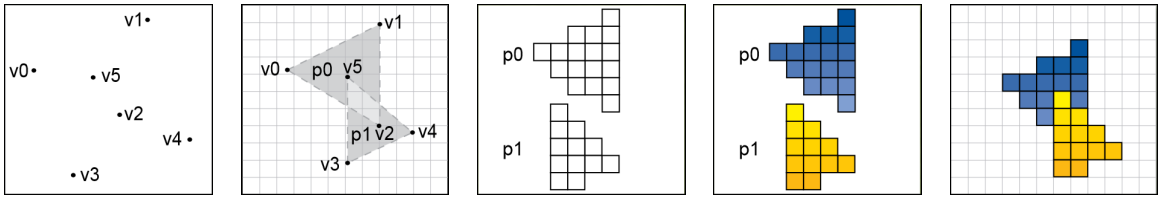


Fig. 2.2.: Example of an operation trough the Graphics Pipeline [FH08a, p. 21]

a) Six vertices (*v0*-*v5*) from the Vertex Generation stage, defining two triangles.

b) After Vertex Processing and Primitive Generation the two triangle primitives (*p0* and *p1*) are projected and transformed to screen-space coordinates.

c) Fragment Generation produces two sets of fragments from the triangles.

d) Each fragment is shaded in the Fragment Shading stage, according to the associated fragment shader.

e) Fragments are combined in the pixel operations stage and the final image is transferred to the framebuffer. Triangle *p1* occludes *p0* as it is nearer to the camera.

2.3.1. Immediate Mode Renderer

Traditionally, a IMR renders every polygon without any knowledge of the scene. This is because old graphic APIs such as OpenGL were originally designed as an immediate mode API, where every primitive was rendered in order of submission. This has caused a huge waste of processing power due to Overdraw (see glossary). Newer versions of OpenGL phased out immediate mode and the OpenGL 4 Core Profile deprecated all immediate mode functions. Optimizations such as Early-Z help with reducing Overdraw, but are only as good as the application geometry ordering. Figure 2.3 gives a simplified overview of the pipeline of a traditional IMR.

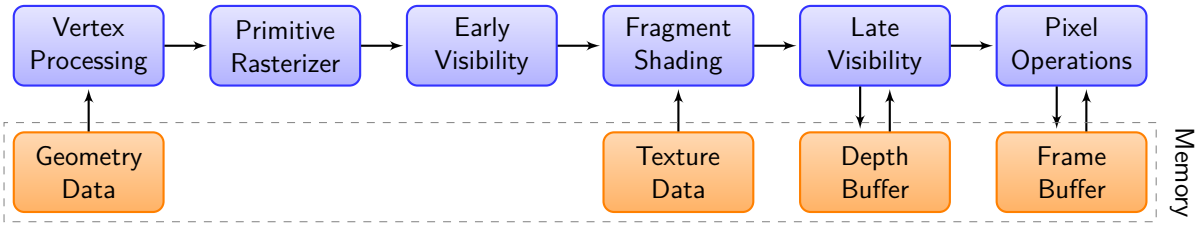


Fig. 2.3.: Simplified Immediate Mode Renderer Pipeline (Image based on [Gan13, p. 5])

Despite the name, tiling is not absent from IMRs. Modern GPUs use multiple rasterizers, where Nvidia uses up to four [Pur10, p. 8] and AMD up to two [Fow10, p. 5]. With Hi-Z whole tiles can be rejected before they arrive in the fragment shading stage of the pipeline. Finally, fragment shaders are also working in so called shading blocks, which are processed in parallel [Fat10, p. 27]. It is to be assumed that these shading blocks are orderded in a specific way to maximize memory page coherency [Gie11f].

2.3.2. Tiling-Based Renderer

Mobile GPUs must balance performance and power consumption, which is why they will never be as powerful as desktop GPUs. One major power consumer is the memory, as it is usually shared with the CPU. The memory also has a high latency as it is not located on the chip itself but needs to be addressed over a slow memory bus, which is shared with other peripherals such as the camera or wireless connectivity. Power consumption and latency also increases with the length of this memory bus. Especially in the pixel operations stage, the renderer needs to read from and write to the framebuffer. For example when using transparency, the old pixel value needs to be fetched from the framebuffer in order to be blended with the partially transparent pixel, and only then can the resulting pixel be written back to the framebuffer.

To improve performance and reduce power usage, most mobile GPUs use a tile-based rendering approach, with the famous exception of the Nvidia Tegra, which is still an IMR. These TBRs move the frame and depth buffer to high speed on-chip memory, which is much closer to the GPU and therefore much less power is needed. As the on-chip memory takes a lot of space on the chip itself, it cannot be the full size of the framebuffer and must be smaller. This tile buffer is usually only a few pixels big [Mer12].

In order to render a scene, the TBR first splits the scene into multiple equally-sized tiles and saves them to memory (called the “frame data”). The OpenGL API is designed as an immediate-mode renderer, it specifies primitives to be drawn with the current state. So for this tile-based approach to work, the GPU needs knowledge of all objects to be drawn in the current frame. This is also the reason why calling

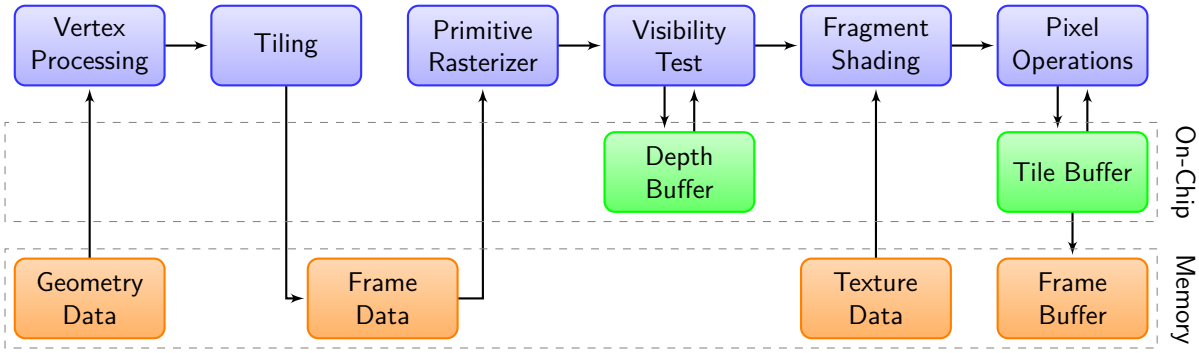


Fig. 2.4.: Simplified Tiling-Based Renderer Pipeline (Image based on [Gan13, p. 7])

`glClear()` seems to take much longer than actual draw calls, because only when the driver can be certain that a frame is complete it can issue the rendering commands to the hardware. The GPU can now render each of these tiles, one at a time, into the on-chip tile buffer. As the tiles are completely independent from each other, this operation scales almost linearly and GPUs can employ multiple renderers to improve performance. After a tile is rendered completely, the contents of the color buffer are copied back to the framebuffer. Figure 2.4 gives an overview of the pipeline of a TBR. In addition to the tile buffer, the visibility test on a TBR also employs an on-chip depth buffer [Gan13].

As seen in the effort of reverse engineering of the vendor driver and implementation of a free driver for the Adreno chipset family called freedreno, the tiling effort is usually not handled in hardware but in software in the driver itself [Fre13a].

The term "Tiling-Based **Deferred** Renderer" (TBDR) is coined by the Imagination Technologies PowerVR GPU Series. This is mostly a marketing strategy, as it is basically a regular TBR with added HSR. This is basically a per-tile Z-Buffer, which is located on very fast memory. The geometry is rasterized per tile (without shading) and written to this Z-Buffer. It is used to discard pixels that are occluded. Since the PowerVR MBX Series [Ima09, pp. 6-7], rasterization of tiles only uses one processing cycle per processed line. On a 32×32 tile buffer such a line is 32 pixels wide. This has the added benefit of free horizontal and vertical clipping, as HSR is only performed on on-screen pixels.

2.4. Driver and Command Processor

The GPU driver implements the graphics API and serves as an abstraction layer for the hardware. The driver translates the API calls into a stream of commands which is sent to the GPUs command processor. These commands are stored in a ring buffer, from which the command processor is reading. They usually set registers or trigger rendering commands. Some modern GPUs have separate command processors for 2D and 3D rendering. Changing the rendering state of the GPU is tricky. On nearly every state change, the GPU is forced to completely flush all work that is still being processed in the pipeline, which can mean a major slowdown of the rendering. For example, when changing the current texture or shader, the driver needs to wait for all current processing work to complete in order to not interfere with the rendering result.

Examples of such commands can be seen in the free driver for the Qualcomm Adreno, freedreno [Fre13b].

2.5. Memory Architecture and Limitations

Modern GPUs employ a different memory subsystem than modern desktop CPUs. The memory architecture of a GPU favors bandwidth over latency, which means really fast transfers with high waiting periods. This trade-off is a result of GPUs dealing with increasingly higher resolutions of textures and displays. To mitigate the high latency, modern GPUs employ a series of caches. The high bandwidth also comes with a big hit in power draw of the memory bus.

A first-generation Intel Core i7 has a peak bandwidth of almost 20 GB/s with a memory latency of 47 ns. Dividing this latency by the clock rate of 2.93 GHz results in a cache miss penalty of about 140 cycles. The Geforce GTX 480 on the other hand has a peak bandwidth of 180 GB/s, a shader clock rate of 1.5 GHz and a memory latency of 400 to 800 cycles [Ren11, p. 11]. This is a bit more than 4× the average

memory latency with almost half the clock rate of an Core i7. The resulting cache miss penalty is therefore many times higher [Gie11a].

Many mobile devices and some low-cost desktop and notebook devices feature so called unified or shared memory. In this scenario, the GPU has no discrete memory to work with but is allocated a part of the system memory. The rest of the system is therefore sharing the already small, available memory bandwidth with the GPU. This is more noticeable on mobile devices. The Nexus 7 (2013) has a peak bandwidth of 13 GBit/s with a resolution of 1920×1200 pixels. These pixels are stored inside the framebuffer in a RGB565 configuration (16 bit per pixel). This alone results in a 2.2 GBit/s bandwidth need to achieve 60 frames per seconds if every pixel is rendered once. Overdraw, texture access and render targets are also putting more stress on the memory bus.

2.6. Shader Architecture and Limitations

The first programmable hardware stages or shader units used a assembly-like programming language. The vertex and fragment shader used different hardware architectures and therefore different subsets of said programming language. Each hardware architecture had different performance trade-offs, with vertex and texture caching as an example.

After the introduction of high-level shader languages like the OpenGL Shading Language (GLSL) or the DirectX counterpart High-Level Shading Language (HLSL), which was developed alongside with Nvidias C for Graphics (Cg), the unified shader model was introduced. This unified the vertex and fragment shader hardware designs into one design, which is now used for both shader types.

The shader hardware consists basically of fast Arithmetic Logic Units (ALU) built around a Floating Multiply-Accumulate (FMAC) unit. There is also special hardware for common used mathematical functions (for example the square root, trigonometric and logarithmic functions). The shader units are optimized for high throughput and are running a high number of threads, which run the same shader code in parallel. Due

to the architecture the hardware is not particularly good at branching code, especially if these branches are not coherent. In some cases its even possible that a batch of threads execute unnecessary branches and discard the results, only because one thread needs to execute the branch.

With the unified shader model, vertex, fragment and other shaders have virtually no hardware differences, but the used shader language may impose artificial limitations which are enforced in software. AMD is using a SIMD architecture that is wide enough for a four component vector, which is implied by most shading languages. Nvidia uses scalar instructions [Gie11b].

Another reason to run the shader in batches is texture access. Memory access is optimized for high bandwidth with the drawback of high latency (section 2.5). If a shader asks for a texture sample, it does so in bigger blocks. Texture sample requests of 16 to 64 pixels per thread are now being used by the major vendors. While waiting for the hardware to fetch the texture samples, the shader unit will pause execution and switch to another shader batch if there is one available. Most vendors also employ a two-level texture cache to mitigate the huge memory latency and exploit the fact that most texture access is bilinearly sampled and uses four texture pixels. Every texture cache hit is a huge performance improvement because the hardware does not have to fetch a new texture block [Gie11c].

2.7. Render Output Units

The Render Output Unit (ROP), also called Raster Operations Pipeline on occasion, is the final step in the rendering process. The name is purely historical, as it originates from early 2D hardware accelerators, in which their main purpose was fast bitmap blitting. It hat three inputs: the source image, a binary image mask and the destination coordinates. The ROP first blitted the mask to the framebuffer at the destination with an AND operation. Finally, the source image would be copied to the destination with an OR operation. Nowadays, the binary image mask is replaced with an alpha value, and the bitwise operations with a configurable blending function [Gie11f].

On a TBR, blending and other pixel operations are very fast, because the required buffers are also on the on-chip memory and only need to be flushed to the framebuffer when all operations are completed.

3. Android Graphics Architecture

Traditionally, all android views were drawn to a bitmap in software using a software renderer called `Skia`. These bitmaps were uploaded to the GPU and all views were composed in hardware since before the first Android release. This allowed effects like scrolling, pop-ups and fading to be fast, as these frequently used effects are implemented in the compositing stage of the rendering pipeline and happened in hardware [Hac11].

Nevertheless, compared to an iOS device, Android felt much slower. This was due to the fact that usually the default browsers of both platforms were used to compare performance. The iOS browser uses a tiled approach, which renders the webpage into smaller tiles. This makes zooming and panning much smoother. Android rendered the page directly to the screen, which was done to eliminate artifacts when zooming. But this made rendering and zooming more complex and therefore slower. As of Android 3.0 the stock browser also uses a tiled rendering approach.

Starting with Android 3.0 Honeycomb (API Level 11), hardware accelerated UI rendering was introduced. With the help of OpenGL, Android could now renderer frequently used canvas operations with the GPU. This still needed manual enabling in the `AndroidManifest` file and not all canvas operations were supported, but the foundations for a faster rendering system were laid. Unsupported operations included the canvas methods `drawPicture`, `drawTextOnPath` and `drawVertices`. With Android 4.0 Ice Cream Sandwich (ICS, API Level 14) the opt-in hardware acceleration became opt-out, an application targeted for ICS or later now needs to specifically disable the acceleration via the manifest [Hac11]. Using a unsupported canvas operation will turn the hardware acceleration off. Even with Android 4.4 KitKat (API Level 19) not all canvas operations are supported [And13a].

Using OpenGL to render the UI is merely a trade-off between rendering speed and memory usage. Currently, only initializing an OpenGL context in a process can cost up to 8 MB memory usage. This can be a huge hit to the memory footprint of the application. Therefore, the Android engineers decided to specifically disable hardware acceleration on most system processes such as the navigation bar and status bar.

To reduce stress on the GPU and memory system only invalidated parts of the UI are redrawn. This is implemented with the use of clipping rectangles. The use of hardware overlays further reduces bandwidth use, as they do not have to composited on the screen. If the number of available overlays are not sufficient, framebuffer objects are used which are essentially textures. These have to be composited on the screen, meaning a copy of the contents to the framebuffer.

3.1. Hardware Accelerated UI Rendering

As the Android rendering pipeline has grown for several years now, the codebase itself is very big and is not explained in depth here. Nevertheless, all critical code paths in all major components are explained that will be used when Android renders a standard application. Rendering is only a small part of the applications execution and only possible in conjunction with the window and input systems, which are not explained here.

3.1.1. Asset Atlas Texture

The Android start-up process Zygote (see glossary) always keeps a number of assets preloaded which are shared with all processes. These assets are containing frequently used NinePatches (see glossary) and images for the standard framework widgets. But for every asset used by a process, there exists a GPU copy as a texture. Starting with Android 4.4 KitKat, these frequently used assets are packed into a texture atlas, uploaded to the GPU and shared between all processes at system start.

Figure 3.1 shows an asset atlas texture generated on a Nexus 7, containing all frequently used framework widget assets. It is important to note, that the NinePatches in the

asset texture do not feature the typical borders which indicate the layout and padding areas. The original asset file is still used on system start to parse these areas, but it is not used for rendering.

The `SystemServer`, started by Zygote at boot-time, initializes and starts the `AssetAtlasService`[1]. On the first boot or after a system update, this service brute-forces trough all possible atlas configurations and looks for the best one, which maximizes the number of assets packed and minimizes the size of the texture. This configuration is written to `/data/system/framework_atlas.config` and contains the chosen algorithm, dimensions, whether rotations are allowed and whether padding has been added. All used assets are first sorted by width and height. This continues until all textures have been packed into the atlas.

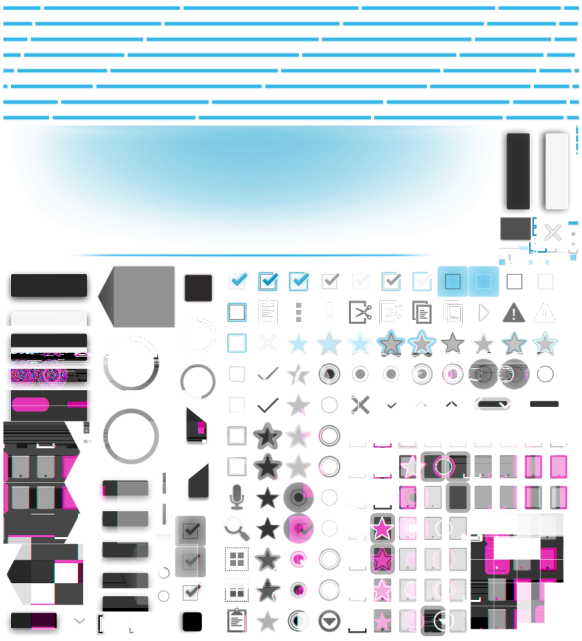


Fig. 3.1.: A sample texture atlas generated on a Nexus 7 [And13d]

The `Atlas`[2] is responsible for packing the assets. It starts with an empty atlas texture, divided into a single cell. After placing the first asset, the remaining space is divided into two new rectangular cells. Depending on the algorithm used, this split can either be horizontal or vertical. The next asset texture is added in the first cell that is large enough to fit. This now occupied cell will be split again and the next asset is processed. The `AssetAtlasService` is using multiple threads for this process. The algorithm producing the best fitting atlas will then be saved to the configuration file for future usage. The internal implementation is based on a linked list for performance reasons, but the algorithm itself is best represented as a simple binary tree.

[1] File location: `frameworks/base/services/java/com/android/server/AssetAtlasService.java`
[2] File location: `frameworks/base/graphics/java/android/graphics/Atlas.java`

When booting the system, the atlas configuration file is read from disk and the `Atlas` will recompute the asset atlas texture with the supplied parameters. A RGBA8888 graphic buffer is allocated as the asset atlas texture and all assets are rendered onto it via the use of a temporary `Skia` bitmap. This asset atlas texture is valid for the lifetime of the `AssetAtlasService`, only being deallocated when the system itself is shutting down.

When a new process is started, its `HardwareRenderer` queries the `AssetAtlasService` for this texture. Every time the renderer needs to draw a bitmap it checks the atlas first. When the atlas contains the requested bitmap, it will be used in rendering [And13d].

3.1.2. Display Lists

In the world of the android graphics pipeline, a display list is a sequence of graphics commands needed to be executed to render a specific view. These commands are a mixture of statements that can be directly mapped to OpenGL commands, such as translating and setting up clipping rectangles, and more complex commands such as `DrawText` and `DrawPatch`. These need a more complex set of OpenGL commands. Listing 3.1 shows an example of a basic display list which renders a button, located at the origin $p_{xy} = (0, 0)$. The parent of the button is responsible for translating this origin to its actual position inside the view hierarchy, and then executing the buttons display list. In the example, the first step is to save the current translation matrix to the stack, so that it can be later restored. It then proceeds to draw the buttons NinePatch, followed by another save command. This is necessary because for the text to be drawn, a clipping rectangle is set up to the region that is affected by the text, and the origin is translated to the text position. The text is drawn and the original translation matrix is restored from stack.

```
Save 3
DrawPatch
Save 3
ClipRect 20.00, 4.00, 99.00, 44.00, 1
Translate 20.00, 12.00
DrawText 9, 18, 9, 0.00, 19.00, 0x17e898
Restore
RestoreToCount 0
```

Listing 3.1: Example display list for rendering a Button [GH11]

Every view generates its own display list, recursively descending the view hierarchy. If a view gets invalidated due to user input events or animations, the affected display lists will be rebuilt and eventually redrawn. The root view is responsible for triggering this rebuilding of the display lists after an invalidation.

Replaying an display list is much more efficient than executing the view's `onDraw()` method, as the display list lives on the native C++ side of the pipeline and is basically just a huge sequence of drawing operations. The display lists are built around a canvas (`GLES20RecordingCanvas`, subsection 3.2.8), which does not execute the drawing operations but records them for later playback.

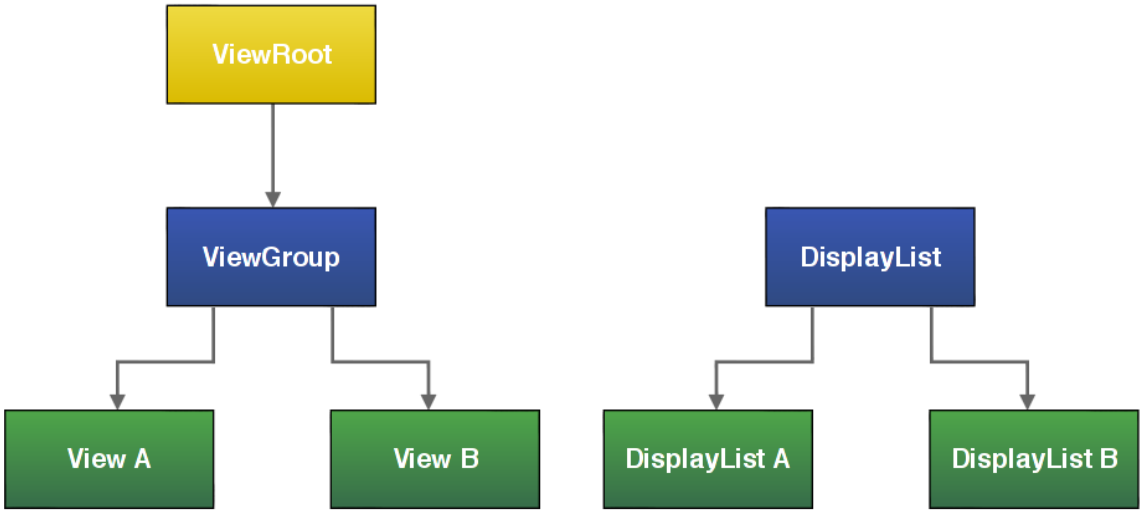


Fig. 3.2.: An example view hierarchy and the corresponding display list hierarchy [GH11]

For example, if `View B` (Figure 3.2) gets invalidated, only the `DisplayList B` needs to be rebuild. This also means that only the display region that is affected by this display list needs to be redrawn. Internally, this is done by setting up a clipping

rectangle and redrawing all display list that could possibly affect this rectangle. It is possible for the GPU to reject pixels that are not inside the clipping region.

Android 4.3 introduced Display List Properties, which are used to prevent a complete display list rebuild on specific invalidation changes. Changing a views position, rotation, scale or transparency now results in a optimal invalidation, as the display list does not need to be rebuild but only redrawn with the changed properties. The Android documentation claims that no redrawing of the targeted view is needed[3], but that only applies to execution to the views `onDraw()` method. The display list still has to be replayed.

The display list itself is implemented in `DisplayList`[4], and the `GLES20DisplayList`[5] is only a small abstraction layer build on top of the native display list and an instance of a `GLES20RecordingCanvas` (subsection 3.2.8). This native display list in turn is only a small wrapper around a `DisplayListRenderer` (subsection 3.2.11) and is responsible for managing the display list properties. A transformation matrix is calculated from these properties, which will be applied before the display list is replayed. Replaying a display list is as simple as iterating over all display list operations and calling the `replay()` method on it (Listing 3.2).

```
void DisplayList::replay(...) {
    for (unsigned int i = 0; i < displayListOps.size(); i++) {
        DisplayListOp *op = displayListOps[i];
        op->replay(...);
    }
}
```

Listing 3.2: Replaying a display list by iterating over all operations and calling `replay()`

3.1.3. Merging of Operations

Before Android 4.3, rendering operations of the UI were executed in the same order the UI elements are added to the view hierarchy and therefore added to the resulting

[3] `http://developer.android.com/guide/topics/graphics/hardware-accel.html`
[4] File location: `framework/base/libs/hwui/DisplayList.cpp`
[5] File location: `framework/base/core/java/android/view/GLES20DisplayList.java`

display list. This can result in the worst case scenario for GPUs, as they must switch state for every element. For example, when drawing two buttons, the GPU needs to draw the NinePatch and text for the first button, and then the same for the second button, resulting in at least 3 state changes (section 2.4).

With the introduction of reordering, Android can now minimize these state changes by ordering all drawing operations by their type. As seen in Figure 3.3, a naive approach to reordering is not sufficient. Drawing all text elements and then the bitmap (or the other way around) does not result in the same final image as it would without reordering.

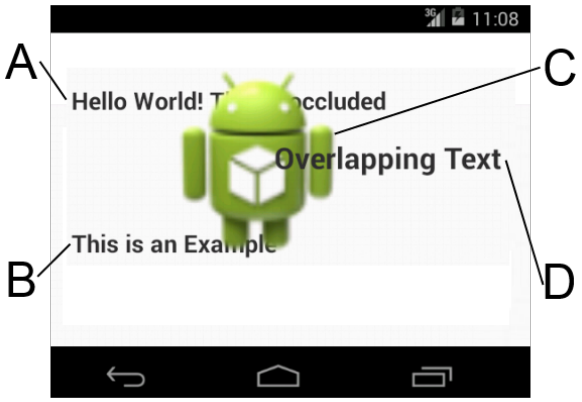


Fig. 3.3.: Example Activity with overlapping text elements and one drawable.

In order to correctly render the Activity in Figure 3.3, text elements `A` and `B` have to be drawn first, followed by the bitmap `C`, followed by the text element `D`. The first two text elements can be merged into one operation, but the text element `D` cannot, as it would be overlapped by bitmap `C`. Regardless of the merging, the `FontRenderer` will be further optimizing this case (subsection 3.2.14).

To further reduce the drawing time needed for a view, most operations can be merged after they have been reordered. Listing 3.3 shows a simplified algorithm Android uses to reorder and merge drawing operations. This happens in the so-called `DeferredDisplayList`[6], because the execution of the drawing operations does not happen in order, but is deferred until all operations have been analyzed, reordered and merged. Because every display list operation is responsible for drawing itself, an operation that supports the merging of multiple operations with the same type must be able to draw multiple, different operations in one call. Not every operation is capable of merging, so some can only be reordered.

To merge operations in an application's `DisplayList` each operation is added to the deferred display list by calling `addDrawOp(DrawOp)`. The drawing operation is

[6]File location: `frameworks/base/libs/hwui/DeferredDisplayList.cpp`

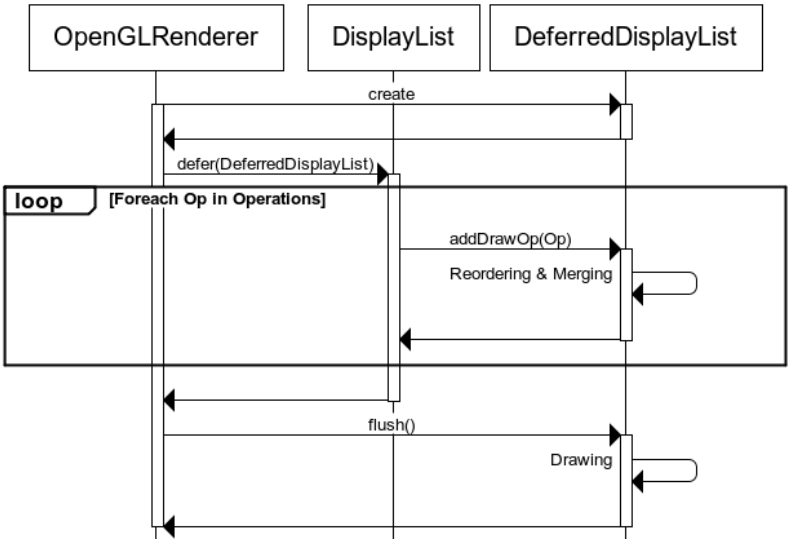


Fig. 3.4.: The `OpenGLRenderer` creates a `DeferredDisplayList` and the original `DisplayList` is adding every operation to the new deferred one. The deferred display list then being flushed.

asked to supply the `batchId`, which indicates the type of the operation, and the `mergeId` by calling `DrawOp.onDefer(...)` (subsection 3.2.13). Possible `batchId`s include `OpBatch_Patch` for a NinePatch and `OpBatch_Text` for a normal text element, which are defined as an enum (`enum OpBatchId`[7]). The `mergeId` is determined by each `DrawOp` itself, and is used to decide if two operations of the same `DrawOp` type can be merged. For a NinePatch, the `mergeId` is a pointer to the asset atlas (or bitmap), for a text element it is the paint color. Multiple drawables from the asset atlas can potentially be merged into one batch, resulting in a greatly reduced rendering time.

After the `batchId` and `mergeId` of an operation are determined, it will be added to the last batch if it is not mergeable. If no batch is already available, a new batch will be created. The more likely case is that the operation is mergeable. To keep track of all recently merged batches, a hashmap for each batchId is used which is called `MergeBatches` in the simplified algorithm. Using one hashmap for each batch avoids the need to resolve collisions with the `mergeId`.

If the current operation can be merged with another operation of the same `mergeId` and `batchId`, the operation is added to the existing batch and the next operation can be added. But if it cannot be merged due to different states, drawing flags or

[7]File location: `frameworks/base/libs/hwui/DeferredDisplayList.h`

bounding boxes, the algorithm needs to insert a new merging batch. For this to happen, the position inside the list of all batches (`Batches`) needs to be found. In the best case, it would find a batch that shares the same state with the current drawing operation. But it is also essential that the operation does not intersect with any other batches in the process of finding a correct spot. Therefore, the list of all batches is iterated over in reverse order to find a good position and to check for intersections with other elements. In case of an intersection, the operation cannot be merged and a new `DrawBatch` is created and inserted into the `MergeBatches` hashmap. The new batch is added to `Batches` at the position found earlier. In any case, the operation is added to the current batch, which can be a new or an existing batch.

The actual implementation is more complex than the simplified version presented here. There are a few optimizations worth being mentioned. The algorithm is tries to avoid Overdraw by removing occluded drawing operations, and also tries to to reorder non-mergeable operations to avoid GPU state changes. These optimizations are not shown by Listing 3.3.

```
vector<DrawBatch> batches;
Hashmap<MergeId, DrawBatch*> mergingBatches[BatchTypeCount];

void DeferredDisplayList::addDrawOp(DrawOp op):
    DeferInfo info;
    /* DrawOp fills DeferInfo with its mergeId and batchId */
    op.onDefer(info);

    if(/* op is not mergeable */):
        /* Add Op to last added Batch with same batchId, if first
           op then create a new Batch */
        return;

    DrawBatch batch = NULL;
    if(batches.isEmpty() == false):
        batch = mergingBatches[info.batchId].get(info.mergeId);
        if(batch != NULL && /* Op can merge with batch */):
            batch.add(op);
            mergingBatches[info.batchId].put(info.mergeId, batch);
            return;

        /* Op can not merge with batch due to different states,
           flags or bounds */
        int newBatchIndex = batches.size();
        for(overBatch in batches.reverse()):
            if (overBatch == batch):
                /* No intersection as we found our own batch */
                break;

            if(overBatch.batchId  == info.batchId):
                /* Save position of similar batches to insert
                   after (reordering) */
                newBatchIndex == iterationIndex;

            if(overBatch.intersects(localBounds)):
                /* We can not merge due to intersection */
                batch = NULL
                break;

    if(batch == NULL):
        /* Create new Batch and add to mergingBatches */
        batch = new DrawBatch(...);
        mergingBatches[deferInfo.batchId].put(info.mergeId, batch);
        batches.insertAt(newBatchIndex, batch);
    batch.add(op);
```

Listing 3.3: Simplified merging and reordering algorithm of the Android drawing operations

3.1.4. Software VSync

Vertical Synchronization (VSync, see glossary) was always a component of Android to prevent the screen from tearing. But in Android 4.2 and earlier versions, the hardware VSync events did not get used in Android. Android would start to render a frame when the system got around to do it, triggered by user input or view invalidation.

Starting with Android 4.3, these VSync events are now used for the drawing system as well. By starting to draw a frame right after the display contents got refreshed (the VSync), Android has the maximum amount of time to finish the frame before the next refresh. If the display is refreshing at 60 Hz, Android has 16 ms to process user input and issue draw commands.

There are two kind of VSync events in the Android system. The `HWComposer` is responsible for reporting the hardware VSync events which come from the display itself. These hardware events are converted to a software event by the `SurfaceFlinger` and distributed via the `Binder` to the `Choreographer` [And13e].

3.1.5. Triple Buffering

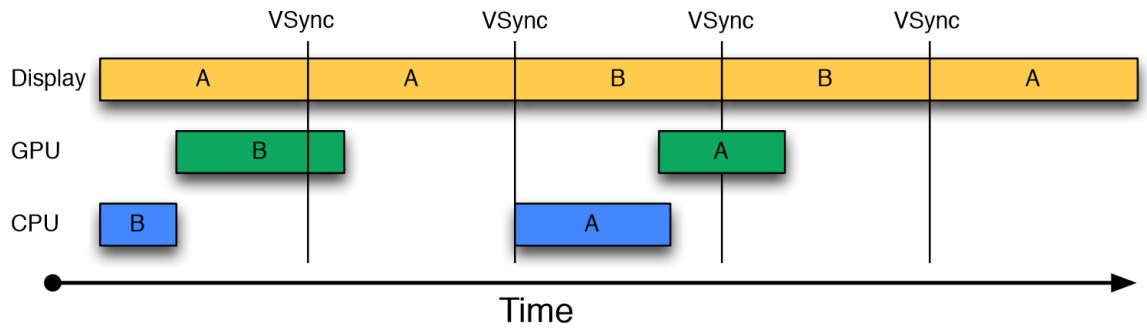


Fig. 3.5.: Rendering with VSync can result in missed frames if the next VSync event is missed [HG12]

By default, Android is a double buffered system. One front buffer containing the currently visible screen contents, and one back buffer, containing the next frame's screen contents. On the next VSync event, these two buffers get swapped, which is called page flipping.

To render a full scene to the buffer, the CPU first needs to generate the display list and replay it. The GPU renders the elements from the display list to the buffer. Depending on the complexity of the view, the generation of the display list and the rendering of the display list can take more than one frame, resulting in a dropped frame (Figure 3.5). As the system missed the VSync event after the delivering of buffer `B`, it can only start to process the next buffer at the following VSync event. Therefore every second frame is missing and the user experience is not smooth anymore.

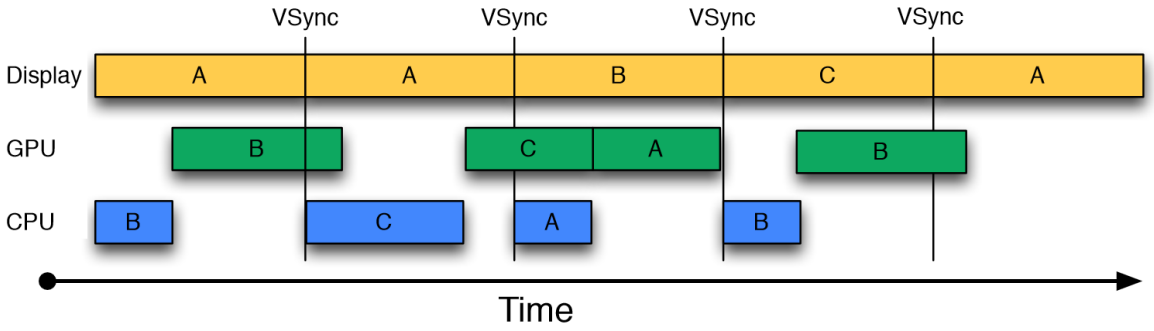


Fig. 3.6.: If a missed frame is detected, a third buffer is used for the rendering of the application [HG12]

If the Android system notices such a frame drop, it automatically sets the application to use a third buffer (Figure 3.6). While one buffer is displayed and the GPU is rendering to the second buffer, the CPU can start on preparing the next frame by working on the third buffer. Triple Buffering is therefore a trade-off between input latency and rendering latency. Android is willing to introduce an input latency of at least one additional frame to improve the rendering latency, which will result in an all-around smoother user experience.

3.2. Rendering an example application

Using an example application, the Android UI rendering pipeline is explored in depth, starting with the public Java API, going to native C++ Code and finally looking at the raw OpenGL drawing operations. The activity (Figure 3.7) consists of a simple `RelativeLayout`, an `ActionBar` with the application icon and title and a `Button` which reads “Hello world!”.

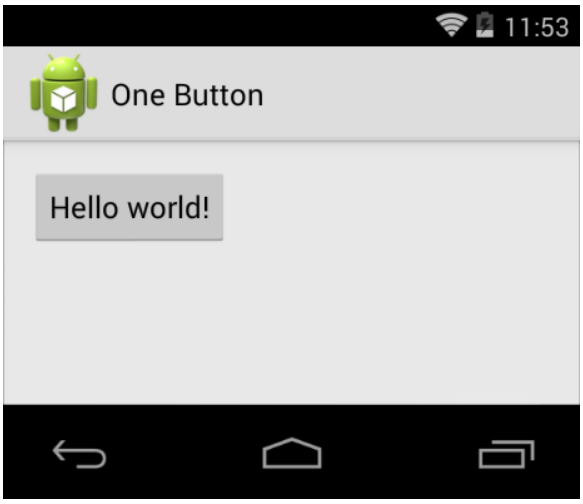


Fig. 3.7.: Example Activity used to trace through the graphics architecture

The RelativeLayout consists of a simple color-gradient background. More complex, the action bar is composed of the background, which is a gradient combined with a bitmap, the "One Button" text element and the application icon, which is also a bitmap. A NinePatch is used as the background for the button, and the string "Hello World!" is drawn on top of it. The navigation bar and status bar at the top and bottom of the screen are not part of the applications activity and therefore will not be examined.

3.2.1. Activity Start

The application is launched by clicking on the application icon in the launcher. The launcher will call `startActivity(...)` with an intent to launch the application. Eventually, the `ActivityManagerService` will process this intent and start the new application with `startViaZygote(...)`, which tells Zygote to fork a new process and returns the new process id [Kar10]. The `ActivityManagerService` is bound to the new application and app specific classes are loaded into memory. It will then launch the final applications code and calls the `onCreate()` and `onStart()` methods. The application is now fully started.

3.2.2. Root View

Every Android activity has a implicit root view at the top of the view hierarchy, containing exactly one child view. This child is the first real view of the application defined by the application developer. The root view is responsible for scheduling and executing multiple operations such as drawing, measuring, layouting and invalidating views.

Figure 3.8 shows the view hierarchy as displayed by the Android debug monitor. The first `FrameLayout` is generated by the activity to host the action bar. The `RelativeLayout` is hosting the button of the example view.

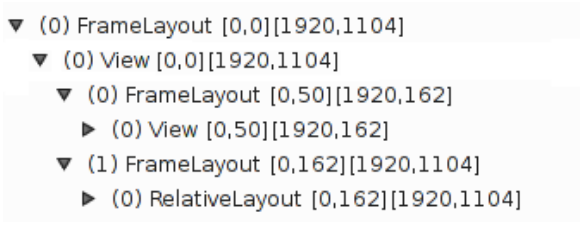


Fig. 3.8.: View hierachy of the example application

The root view of an application is implemented in `ViewRootImpl.java`[8]. Started by the `WindowManager`, it holds a reference to the first view of the application in `mView` which is set in `setView(View, ...)`. This method also tries to enable hardware acceleration, which depends on the application flags and whether the application is a persistent process, like the navigation or notification bar. Hardware acceleration is never enabled for system processes, and disabled for persistent processes on low-end devices. This is decided in `enableHardwareAcceleration(...)`. If hardware acceleration is to be used, a new `HardwareRenderer` (subsection 3.2.3) is created. In the example application, this is the case. Also, the root view creates a new `Surface` (subsection 3.2.9) to draw upon. This surface is initially not-valid, but will be made valid by the `SurfaceFlinger` once resources are available and allocated.

If the member view is invalidated, resized or animated, the view will call the root views `invalidateChildInParent(...)` method (Listing 3.4). The invalidated rectangle is united with the local `dirty` rectangle, resulting in a new rectangle which covers the new and all previously invalidated rectangles. The `scheduleTraversal()` method then attaches the root view to the software VSync events via the `Choreographer` (Figure 3.9).

Invalidation of a rectangle can happen, for example, on an input event. If the user pushes the button of the example view, the button will be invalidated, as it now needs to be redrawn in the pushed state. This will invalidate part of the parent view of the button, cascading all the way up to the root view.

[8]File location: `frameworks/base/core/java/android/view/ViewRootImpl.java`

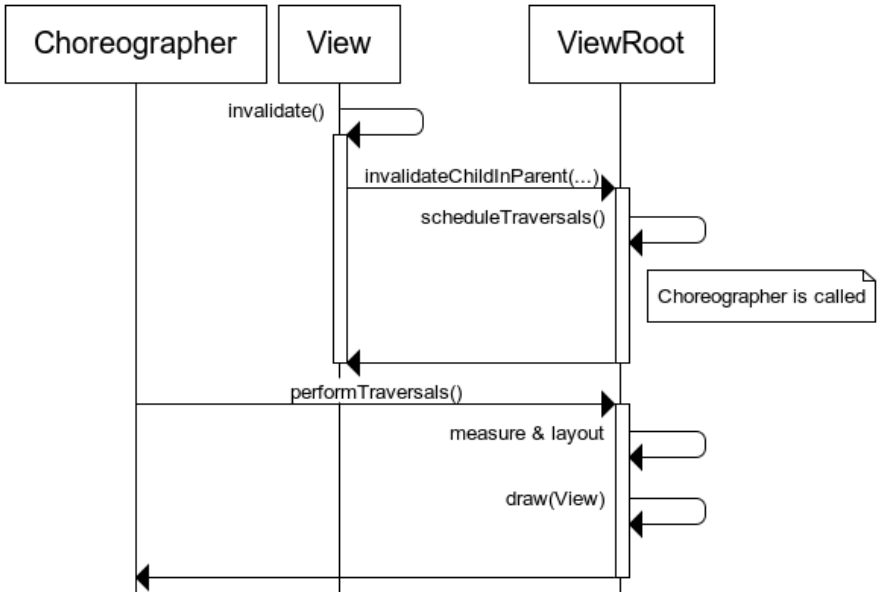


Fig. 3.9.: When the `View` gets invalidated, it will call the `ViewRoot`, which in turn will schedule a traversal using the `Choreographer`. At the next VSync event, the choreographer calls `performTraversals()` on the `ViewRoot`.

```
public ViewParent invalidateChildInParent(int[] location,
                                          Rect dirty) {
    // ...
    // Add the new dirty rect to the current one
    mDirty.union(dirty.left, dirty.top,
                 dirty.right, dirty.bottom);
    // ...
    if (!mWillDrawSoon /* ... */) {
        scheduleTraversals();
    }
    return null;
}
```

Listing 3.4: Invalidation of a view will cause its invalidation rectangle to be added to the root views dirty rectangle, and a new traversal is scheduled.

On the next VSync event happening right after the display contents got refreshed, the `Choreographer` calls the root views `performTraversals(...)` method. This method does all the heavy lifting regarding the applications view model. In the step called "measure and layout" it handles fading in and out the application when opening and closing. Resizing, for example on orientation changes, and other layout changes are also handled. The window manager is notified of this change in window dimensions

in `relayoutWindow()`, as is the `HardwareRenderer`, which needs to reinitialize the underlying canvas.

When the view needs to be refreshed, `draw(View)` gets called inside the traversal. This is not the `onDraw(...)` method which can be overloaded, but an internal method. The dirty regions which need to be redrawn are calculated and the `HardwareRenderer` gets told to draw a specific region of the member view. If hardware acceleration is disabled, the view will get redrawn using the old software rendering path.

3.2.3. HardwareRenderer

The `HardwareRenderer`[9] is an interface to render a view hierarchy to a canvas. In order to draw a view, it must create a `DisplayList` containing all drawing operations necessary to draw itself and all of its children. This display list is being handed to the underlying canvas, which will in turn draw the DisplayList itself. The hardware renderer is also responsible for managing the OpenGL context.

On Android 4.4, the hardware renderer instance created by the root view is always a `GL20Renderer` (subsection 3.2.8). On initialization in `initializeIfNeeded()`, a `GLES20Canvas` is being generated. This canvas is the main canvas for the application, on which all drawing will happen. It will later be displayed on the screen. The shared asset atlas will be fetched in `initAtlas()` and the reference to it saved for rendering.

When the `draw(View, ...)` method is being called by the root view, the view is asked to build and return the required display list via `View.getDisplayList()`, and this display list is drawn on the internal canvas via `canvas.drawDisplayList(...)` (Figure 3.10). After drawing the display list, the `eglSwapBuffers(...)` command is called. This command is implemented in the custom Android `EGL` implementation called `AGL`, which is the layer between the rendering API and the underlaying windowing system. It handles surface and buffer binding, context management and rendering synchronization. The command queues the current buffer into the buffer rotation, and dequeues a used buffer out of the rotation for the next drawing operations. The new contents will be then visible on screen with the next VSync event.

[9] File location: `frameworks/base/core/java/android/view/HardwareRenderer.java`

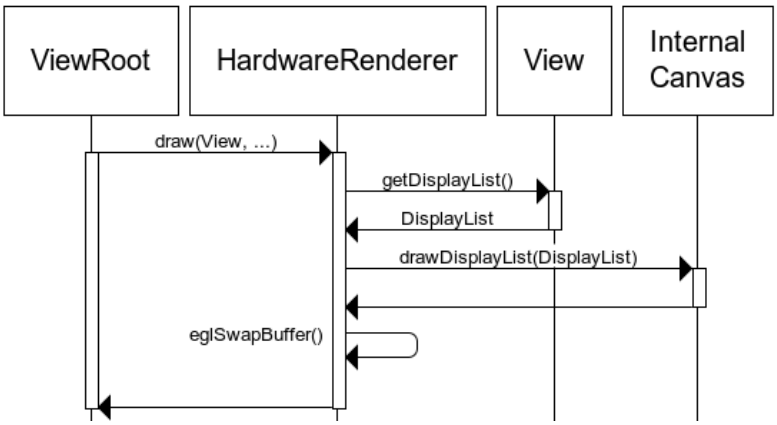


Fig. 3.10.: The `HardwareRenderer` draws a view by fetching the corresponding display list and drawing it to the internal canvas. Finally `eglSwapBuffer()` is called.

The overdraw counter is also drawn by the `HardwareRenderer` in the `debugOverdraw()` method. This counter measures the amount of overdraw per application. A new `GLES20Canvas` is created and the special debug flag `setCountOverdrawEnabled(...)` is set, which counts the ratio of overdrawn pixel. The internal display list is rendered on this invisible canvas and the number of overdraws is being written to the visible canvas, on top of the active view, in `drawOverdrawCounter(...)`. Overlaying overdrawn areas with colors is handled by the `OpenGLRenderer`.

In order to debug performance, the `HardwareRenderer` can keep record of the time per frame over the last 128 frames. These measurements can be visualized on-screen or dumped to the console. Section 3.3 is using this data for Overdraw analysis.

3.2.4. View, ViewGroup and Layouts

Every view has a reference to its parent. The first view inside the view hierarchy, which the root view references, has this root view as a parent. The `View`[10] serves as a base class for almost every visible element and widget of the Android framework. It occupies a rectangular space on the screen and does not support any children [And13f]. The subclass `ViewGroup`[11] supports multiple children and serves as a base class for all Android layouts.

[10]File location: `frameworks/base/core/java/android/view/View.java`
[11]File location: `frameworks/base/core/java/android/view/ViewGroup.java`

With currently almost 20 000 lines of code, the `View` is one of the bigger classes inside the Android framework. This comes as no surprise, as it is the building block for every widget and application. It handles the keyboard, trackball and touch events, as well as scrolling, scrollbars, layouting and measuring. Focus and visibility events are also processed [And13f].

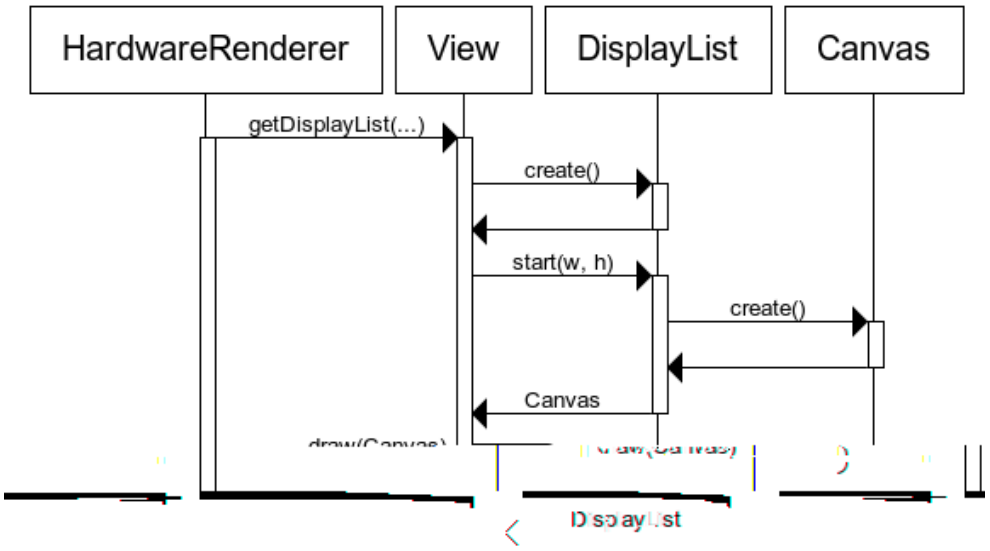


Fig. 3.11.: The `HardwareRenderer` requests a display list from a view. The view will create a new display list and calls the `start(w, h)` method to get a reference to a canvas, on which the view can draw itself. The display list is then returned.

Drawing events are also processed by the `View`. The `HardwareRenderer` will eventually call the `View.getDisplayList(...)` method. If this is the first call made to this method, the view will create a new internal display list (Figure 3.11). This display list is used for the rest of the views lifetime.

Listing 3.5 shows the critical path on how the view hierarchy is drawn by calling `getDisplayList(...)` on the first child in the view hierarchy. The display list is asked to supply a canvas with the view's size. If the view is not using the default layer type (`LAYER_TYPE_NONE`) but a hardware or a software layer, display lists are not used. These hard- and software layers can be used for special effects like blending and color filters. The hardware layer will render a View to an OpenGL texture, on which effects can be used more easily.

The example application does not use a layer, and the canvas returned by the display list is a `GLES20RecordingCanvas` and will be used by all views in the viewing hierarchy to draw upon. The canvas is translated by the the scrolling offset, and then handed to the `draw(...)` method. In the rare case that the view does not have a background, it does not have to draw itself but only the children, if any, by calling `dispatchDraw(...)`.

After drawing, the display list is cached and then returned to the hardware renderer. The cached version is used in subsequent draw calls, in case the view has not changed but only the children.

```
private DisplayList getDisplayList(DisplayList displayList,
                                   boolean isLayer) {
    // ...
    HardwareCanvas canvas = displayList.start(width, height);
    if (!isLayer && layerType != LAYER_TYPE_NONE) {
        // Layers don't get drawn via a display list
    } else {
        computeScroll();
        canvas.translate(-mScrollX, -mScrollY);

        // Fast path for layouts with no backgrounds
        if ((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) {
            dispatchDraw(canvas);
            if (mOverlay != null && !mOverlay.isEmpty()) {
                mOverlay.getOverlayView().draw(canvas);
            }
        } else {
            draw(canvas);
        }
    }
    // ...
    return displayList;
}
```

Listing 3.5: Calling `getDisplayList(...)` on a view will cause it to draw itself to the supplied canvas, which records the drawing operations as a display list. Children will be drawn by the `ViewGroups.draw(...)` method.

In case `draw(canvas)` gets called, the view will draw the background to the canvas and execute the application specific `onDraw()` code. Followed by a call to the `dispatchDraw(canvas)` method, the children of the view will be drawn. It will then

add fading edges if necessary and draws the views decorations (like the scrollbar, for instance).

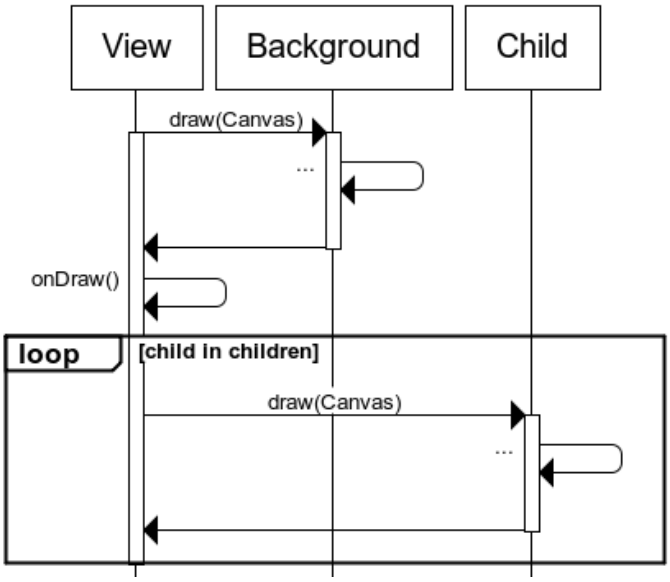


Fig. 3.12.: In order to draw itself, a View will first draw the background. It then calls the `onDraw()` method which may contains special code written by the applications developer. The view proceeds to draw all of its children.

The view base class only implements an empty `dispatchDraw(...)` method as it does not support any children. The sub-class `ViewGroup` implements this method (Listing 3.6). Among other things, it tells all of its children to draw itself to the supplied canvas. The children could be anything, from a normal button to another layout, which itself includes another set of children.

```
protected void dispatchDraw(Canvas canvas) {
        // ...
        final int count = mChildrenCount;
        final View[] children = mChildren;
        for (int i = 0; i < count; i++) {
            final View child = children[i];
            if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE
                 || child.getAnimation() != null) {
                child.draw(canvas, this, drawingTime);
            }
        }
        // ...
}
```

Listing 3.6: `dispatchDraw(Canvas)` is drawing all children of the `ViewGroup` and by iterating over them and calling the `draw(Canvas, ...)` method.

3.2.5. TextView

The `TextView` is a specialized view for basically everything that needs to draw text and a NinePatch. For example, the Android `Checkbox`, `Radiobox` and `Button` are all based on it. To correctly position its text, background and NinePatch, it uses an internal layout.

Implemented in `TextView.java`[12], this view has almost 10 000 lines of code and handles all conceivable corner cases, including support for Right-To-Left and non-latin languages, marquees and text styling. It also handles text editing, such as normal input but also the copy and paste functionality via a context menu.

In its `onDraw(Canvas)` method, it first draws the background via the `onDraw(Canvas)` method of its base class. It then proceeds to position the internal layout and draw it on the supplied canvas.

[12]File location: `frameworks/base/core/java/android/widget/TextView.java`

3.2.6. Button

The `Button`[13] is only a text view with a customized background, the Java class extends `TextView`. This background is set in the constructor of the button, and there is nothing else that distinguishes it from a normal text view apart from a different class name.

The `TextView` will draw the various states of the button via the customized background drawable, changing its appearance based on the state of the background selector drawable. This is also true for the normal background of a `TextView`, which will look differently when focused.

3.2.7. Choreographer

With the `Choreographer`[14], Android apps have a way to connect to the VSync signal from the display subsystem by posting a callback which gets called after the next VSync took place, giving the application the maximum amount of time between two frames to draw itself. For an application developer, there is generally no need to use the choreographer, unless they are using OpenGL or other drawing methods that are not part of the Android framework. Each process has its own instance of the `Choreographer`.

The callback can be registered by calling `postFrameCallbackDelayed(...)` on the choreographer. Among other uses, `ViewRootImpl` is using this callback to schedule the next drawing operation after the root view has been (partially) invalidated. On systems without native VSync support, a default frame interval of 10 ms is used.

Only the VSync events of the main display are currently used. Secondary displays are not yet supported.

3.2.8. GLES20Canvas

The `GLES20Canvas` is a small wrapper around one of multiple renderer implementations implemented in native C++ code. This canvas extends the normal Android `Canvas`,

[13]File location: `frameworks/base/core/java/android/widget/Button.java`
[14]File location: `frameworks/base/core/java/android/view/Choreographer.java`

which offers the same functionality but with a `Skia` backend, only providing software rendering. Listing 3.7 shows an example of how the layer between the native and Java code is built using the Java Native Interface (JNI). All needed renderer methods are exported to Java with the custom Android JNI wrapper.

```
static JNINativeMethod gMethods[] = {
    // ...
    { "nCreateRenderer", "()I",
      (void*) android_view_GLES20Canvas_createRenderer
    }, // ...
    { "nDrawCircle", "(IFFFI)V",
      (void*) android_view_GLES20Canvas_drawCircle
    }
};

static OpenGLRenderer* android_view_GLES20Canvas_createRenderer
                                    (JNIEnv* env, jobject clazz) {
    OpenGLRenderer* renderer = new OpenGLRenderer();
    renderer->initProperties();
    return renderer;
}

static void android_view_GLES20Canvas_drawCircle(JNIEnv* env,
            jobject clazz, OpenGLRenderer* renderer, jfloat x,
            jfloat y, jfloat radius, SkPaint* paint) {
    renderer->drawCircle(x, y, radius, paint);
}
```

Listing 3.7: `android_view_GLES20Canvas.cpp`: Layer between native C++ code and Java abstraction using JNI.

Depending on the supplied arguments in the constructor, the canvas will choose the appropriate renderer. In the normal case this is an `OpenGLRenderer` (subsection 3.2.10). It can also be a `LayerRenderer`, directly rendering to a Framebuffer Object (FBO) that is visible on screen. This is the case when using an OpenGL surface view (`GLSurfaceView`) to issue OpenGL commands directly from the application code, for instance in a 3D game. A `GLES20RecordingCanvas`, which inherits from the `GLES20Canvas`, is a special case, as it constructs its base class with a special recording flag. The canvas will then initialize itself with a `DisplayListRenderer` (subsection 3.2.11). The decision logic is quite simple and displayed in Listing 3.8.

The listing also shows how the method `drawCircle(...)` is only a wrapper for the underlying renderer.

In the example application, the canvas is using a `OpenGLRenderer`. The canvas of the `View` is using a `DisplayListRenderer` created by the display list.

```
protected GLES20Canvas(boolean record, boolean translucent) {
    // ...
    if (record) {
        mRenderer = nCreateDisplayListRenderer();
    } else {
        mRenderer = nCreateRenderer();
    }
    // ...
}

public void drawCircle(float cx, float cy,
                       float radius, Paint paint) {
    nDrawCircle(mRenderer, cx, cy, radius, paint.mNativePaint);
}
```

Listing 3.8: `GLES20Canvas.java`: Choosing a renderer as a backend in the constructor, and delegating all method calls to it using the example of `DrawCircle(...)`.

3.2.9. Surface

Like the `GLES20Canvas`, the `Surface` also is implemented in native code and uses a small JNI wrapper to make it available to Java. The surface is a handle to a raw buffer that is being managed by the screen compositor, the `SurfaceFlinger`. The communication is based on `Binder` IPC mechanisms, and the actual surface buffer handle is therefore a `Parcelable`. In order to send it, it will be flattened into a parcel and send via the `Binder`.

3.2.10. OpenGLRenderer

The `OpenGLRenderer`[15] is an implementation of the `Skia` interface. It allows to use the same method calls as with `Skia`, but it does all the drawing hardware accelerated

[15]File location: `frameworks/base/libs/hwui/OpenGLRenderer.cpp`

with OpenGL. On the way trough the pipeline, this is the first native-only class implemented in C++. The renderer is designed to be used with the `GLES20Canvas`. It was introduced with Android 3.0 and is only used in conjunction with display lists.

Listing 3.9 shows how the drawing commands get translated to actual OpenGL commands, using the example of the `DrawCircle(...)` method. The circle coordinates are translated into a `Skia` path object (`SkPath`). The renderer was created to replace the `Skia` software drawing, but `Skia` is still used internally. The created path is tessellated and the resulting polygons are written to a vertex buffer. This vertex buffer is finally drawn with the `glDrawArrays(...)` OpenGL command as a triangle strip.

```
status_t OpenGLRenderer::drawCircle(float x, float y,
                            float radius, SkPaint* p) {
    SkPath path;
    path.addCircle(x, y, radius);
    return drawConvexPath(path, p);
}

status_t OpenGLRenderer::drawConvexPath(
                    const SkPath& path, SkPaint* paint) {
    VertexBuffer vertexBuffer;
    PathTessellator::tessellatePath(path, paint,
                    mSnapshot->transform, vertexBuffer);
    return drawVertexBuffer(vertexBuffer, paint);
}

status_t OpenGLRenderer::drawVertexBuffer(
                    const VertexBuffer& vertexBuffer,
                    kPaint* paint, bool useOffset) {
    // Set up OpenGL drawing state
    // ...
    // Draw vertices using OpenGL
    glDrawArrays(GL_TRIANGLE_STRIP, 0,
                 vertexBuffer.getVertexCount());
    return DrawGlInfo::kStatusDrew;
}
```

Listing 3.9: The `OpenGLRenderer` draws a circle by converting it to a path and rendering the tessellated path via `glDrawArrays(...)`

Drawing a display list, and therefore a whole view hierarchy, is quite simple for the renderer, as the operations inside the display list do all the rendering themselves

(Listing 3.10). The display list is first converted into a deferred display list by reordering and merging the operations (subsection 3.1.3). The deferred display list is flushed and will replay all internal drawing batches.

```
status_t OpenGLRenderer::drawDisplayList(
                DisplayList* displayList, Rect& dirty,
                int32_t replayFlags) {
    // All the usual checks and setup operations
    // (quickReject, setupDraw, etc.)
    // will be performed by the display list itself
    if (displayList && displayList->isRenderable()) {
        DeferredDisplayList deferredList(*(mSnapshot->clipRect));
        DeferStateStruct deferStruct(
            deferredList, *this, replayFlags);
        displayList->defer(deferStruct, 0);
        return deferredList.flush(*this, dirty);
    }
    return DrawGlInfo::kStatusDone;
}
```

Listing 3.10: A DisplayList is drawn by first converting it into a DeferredDisplayList and then replaying it

There are several layers of abstraction used with the `OpenGLRenderer`, for example the use of `SkPaint`. This paint can be applied to almost all rendering operations, for example when rendering font or backgrounds, and can consist of a single color, a gradient or even a custom shader attached to it. Single colors and gradients are also implemented with shaders.

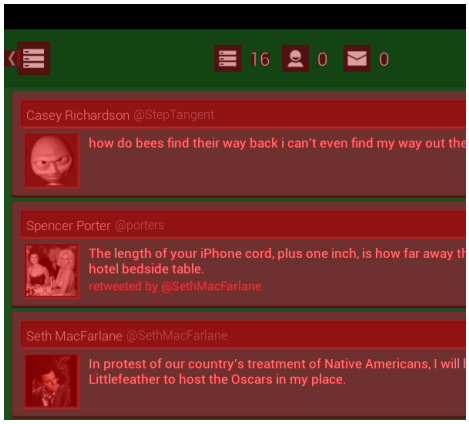


Fig. 3.13.: Visualized Overdraw on a sample applications view [Guy12a]

Drawing debug visualizations of Overdraw (Figure 3.13) is also a function of the renderer. This is done via a multi-pass algorithm. When debugging is enabled, the scene is first rendered normally, but with the stencil buffer enabled. Every pixel written will increment the stencil counter at the pixel's position, regardless of whether it is discarded by the depth test or not. After this, four different rectangles are drawn over the complete scene, each with a different color and stencil

value attached. For example, the first and green rectangle will only color pixels where the stencil value is one. This results in only the pixels being green, that are overdrawn once. This is repeated for every other debug color as well.

3.2.11. DisplayListRenderer

The `DisplayListRenderer` implements the same interface as the `OpenGLRenderer`. It is used with the `GLES20RecordingCanvas`. Instead of rendering directly to a surface, the display list renderer stores each drawing operation to the internal display list for later replay.

A method call is converted into a drawing operation and then added to the internal list of display list operations (Listing 3.11). Similarly, every `Skia` API function call is converted into a `DisplayListOp`.

```
Vector<DisplayListOp*> displayListOps;
status_t DisplayListRenderer::drawCircle(float x, float y,
                                 float radius, SkPaint* paint) {
    addDrawOp(new (alloc()) DrawCircleOp(x, y, radius, paint));
    return DrawGlInfo::kStatusDone;
}

void DisplayListRenderer::addDrawOp(DrawOp* op) {
    displayListOps.add(op);
}
```

Listing 3.11: The DisplayListRenderer converts every drawing call to a DisplayListOp and adds it to the list of operations.

When multiple views are nested, as it is in the example and in almost every other application, the display list renderer is asked to add a whole display list to the list of drawing operations. The display list is converted into a `DrawDisplayListOp` (Listing 3.12), which makes nesting of views possible.

```
status_t DisplayListRenderer::drawDisplayList(
            DisplayList* displayList,  Rect& dirty,
            int32_t flags) {
    addDrawOp(
            new (alloc()) DrawDisplayListOp(displayList, flags));
    return DrawGlInfo::kStatusDone;
}
```

Listing 3.12: A nested DisplayList is added as a DrawDisplayListOp.

3.2.12. DeferredDisplayList

The `DeferredDisplayList`[16] is responsible for reordering and merging display list operations into batches. The merging is done in the same order the operations were added to the original display list, by calling `addDrawOp(...)` for every operation (subsection 3.1.3).

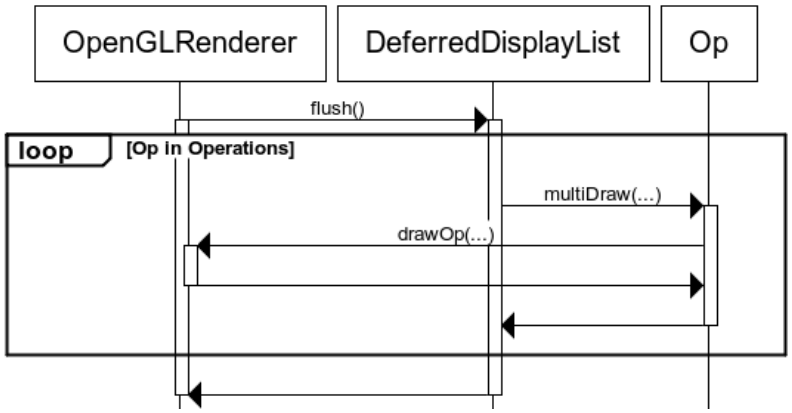


Fig. 3.14.: A `DeferredDisplayList` is flushing its operation by calling `multiDraw()` on the first operation of a batch. An `Op` will draw itself and all other operations inside the batch by calling the corresponding `OpenGLRenderer` method.

The `OpenGLRenderer` will create these deferred display lists and eventually calls the `flush(...)` method, replaying every batch (Listing 3.13). In case of a merged batch, the batch consists of a list of multiple operations. The method `multiDraw(...)` will be called on the first operation in that list, with all the other operations as an argument. The called operation is responsible for drawing all supplied operations at once and will also call the `OpenGLRenderer` to actually execute the operation itself (Figure 3.14).

[16]File location: `frameworks/base/libs/hwui/DeferredDisplayList.cpp`

```
Vector<Batch*> mBatches;
status_t DeferredDisplayList::flush(OpenGLRenderer& renderer,
                                    Rect& dirty) {
    status_t status = DrawGlInfo::kStatusDone;
    status |= replayBatchList(mBatches, renderer, dirty);
    return status;
}

static status_t replayBatchList(const Vector<Batch*>& batchList,
        OpenGLRenderer& renderer, Rect& dirty) {
    status_t status = DrawGlInfo::kStatusDone;
    for (unsigned int i = 0; i < batchList.size(); i++) {
        if (batchList[i]) {
            status |= batchList[i]->replay(renderer, dirty, i);
        }
    }
    return status;
}
```

Listing 3.13: Replaying of a DeferredDisplayList is as simple as iterating over every batch in the correct order and replaying each of them.

3.2.13. DisplayListOp

Each drawing operation to be executed on a canvas has a corresponding display list operation. All display list operations must implement the `replay()` method, which executes the wrapped drawing operation. These drawing operations call the `OpenGLRenderer` to render themselves. The reference to the renderer needs to be supplied when creating an operation. `onDefer()` must also be implemented and must return the operation's draw and `mergeId`. Non-mergable batches are setting the draw id to `kOpBatch_None`. Mergable operations must implement the `multiDraw()` method, which is used when a whole batch of merged operations need to be rendered at once. This subsection will explain a few important operations that are used inside the example application and shows examples of how the `replay()`, `onDefer()` and `multiDraw()` are implemented.

3.2.13.1. DrawTextOp

The `DrawTextOp` is the most common operation for text drawing, used by almost every standard widget that needs to display text.

The `onDefer(...)` method is called by the deferred display list. Listing 3.14 shows how the `mergeId` and `batchId` are determined. The `batchId` depends on whether the text is colored or not, and the `mergeId` depends on the color used. This makes sure that all text operations with the same color can be merged together in one batch. Text decorations will not be merged as they need to be drawn in the correct order. Also, text shadows are not drawn by this operation as they cannot be correctly merged with text.

```
virtual void onDefer(OpenGLRenderer& renderer, DeferInfo& deferInfo,
        const DeferredDisplayState& state) {

    deferInfo.batchId = mPaint->getColor() == 0xff000000 ?
            DeferredDisplayList::kOpBatch_Text :
            DeferredDisplayList::kOpBatch_ColorText;
    deferInfo.mergeId = (mergeid_t)mPaint->getColor();

    // don't merge decorated text
    deferInfo.mergeable = !(mPaint->getFlags() &
            (kUnderlineText_Flag | kStrikeThruText_Flag));
}
```

Listing 3.14: The `DrawOpText` is determining its `mergeId` and `batchId` by the color of the used paint.

Depending on whether an operation was merged with others or not, a different method is used for drawing (Listing 3.15). `applyDraw(...)` is called when only one text operation is needed to be drawn, which just renders the text with the `OpenGLRenderer`.

For drawing a whole text batch, the method `multiDraw(...)` iterates over every operation inside the batch. For each operation `OpenGLRenderer.drawText(...)` is called. The last operation inside a batch sets the `drawOpMode` is to flush, which tells the `FontRenderer` to stop caching operations and start drawing all cached text.

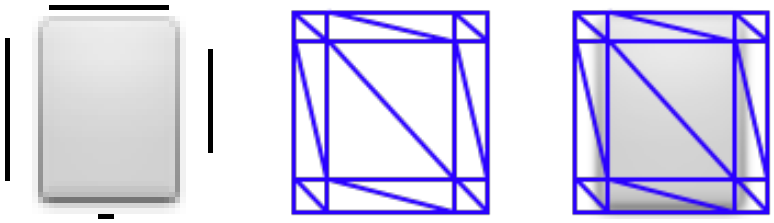
```
virtual status_t applyDraw(OpenGLRenderer& renderer, Rect& dirty) {
    return renderer.drawText(mText, mX, mY, /* ... */);
}

virtual status_t multiDraw(OpenGLRenderer& renderer, Rect& dirty,
              const Vector<OpStatePair>& ops, const Rect& bounds) {
    status_t status = DrawGlInfo::kStatusDone;
    for (unsigned int i = 0; i < ops.size(); i++) {
        DrawOpMode drawOpMode = kDrawOpMode_Defer;
        if(i == op.size() - 1)
            drawOpMode = kDrawOpMode_Flush;

        DrawTextOp& op = *((DrawTextOp*)ops[i].op);
        status |= renderer.drawText(op.mText, op.mX, op.mY,
                                    drawOpMode);
    }
    return status;
}
```

Listing 3.15: Rendering of one and multiple text elements, respectively. In order to render multiple text elements, the font renderer is asked to cache all drawing operations and flushing them once the last text element is processed.

3.2.13.2. DrawPatchOp

The `DrawPatchOp` is responsible for drawing a NinePatch. It is divided into nine different areas (Figure 3.15b), which are stretched according to the one-pixel border layout rules and according to the needs of the contents of the patch. The `batchId` of a NinePatch is always `kOpBatch_Patch`, the merging id is a pointer to the used bitmap. Therefore, all patches that use the same bitmap can be merged together. This is even more important with the use of the asset atlas, as now all heavily used NinePatches from the Android framework can potentially be merged together as the reside on the same texture.

To draw one NinePatch the `DrawPatchOp` uses a single call to the `OpenGLRenderer` (Listing 3.16). The function `getMesh(...)` returns a set of triangles, scaled to the contents of the patch. This mesh is drawn by the renderer with the appropriate NinePatch texture.



(a) (b) (c)

Fig. 3.15.: A NinePatch and the resulting mesh.

a) Original Android NinePatch for a button, including one-pixel layout description border.

b) Set of triangles which result from the NinePatch. This is the mesh that will be used to draw the button.

c) The used mesh with the applied texture.

In order to draw multiple patches at once, the `DrawPatchOp` needs to build a buffer containing all triangles from all operations. This is done by iterating over all operations and adding all vertices from their meshes to the vertex buffer. The vertex buffer is then drawn with one OpenGL call.

```
virtual status_t applyDraw(OpenGLRenderer& renderer, Rect& dirty) {
    return renderer.drawPatch(mBitmap, getMesh(renderer),
                              getAtlasEntry(), getPaint(renderer));
}

virtual status_t multiDraw(OpenGLRenderer& renderer, Rect& dirty,
        const Vector<OpStatePair>& ops, const Rect& bounds) {
    Vector<TextureVertex> vertices;

    for (unsigned int i = 0; i < ops.size(); i++) {
        DrawPatchOp* patchOp = (DrawPatchOp*) ops[i].op;
        const Patch* opMesh = patchOp->getMesh(renderer);
        TextureVertex* opVertices = opMesh->vertices;
        for (uint32_t j = 0; j < opMesh->verticesCount;
             j++, opVertices++) {
            vertices.add(TextureVertex(opVertices->position[0],
                                       opVertices->position[1],
                                       opVertices->texture[0],
                                       opVertices->texture[1]));
        }
    }
    return renderer.drawPatches(mBitmap, getAtlasEntry(),
                        &vertices[0], getPaint(renderer));
}
```

Listing 3.16: Rendering of one and multiple NinePatches, respectively. In order to render multiple patches, one vertex buffer is build which includes all meshes from all patches.

3.2.13.3. DrawDisplayListOp

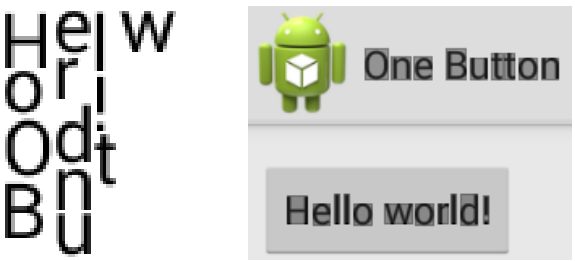
Most applications have a complex view hierarchy and therefore display lists need to be nestable. This operation in itself does not do very much, but is nevertheless essential to support this nesting behavior. The operation will only relay the `replay(...)` and `defer(...)` methods to the wrapped display list (Listing 3.17). Deferring will add all operations of the nested display list to the deferred display list. This results in multiple, nested display lists merged together, which is ideal for optimizing all operations.

```
virtual void defer(DeferStateStruct& deferStruct, int level) {
    if (mDisplayList && mDisplayList->isRenderable()) {
        mDisplayList->defer(deferStruct, level + 1);
    }
}

virtual void replay(ReplayStateStruct& replayStruct, int level) {
    if (mDisplayList && mDisplayList->isRenderable()) {
        mDisplayList->replay(replayStruct, level + 1);
    }
}
```

Listing 3.17: Replaying and deferring a `DrawDisplayListOp` is implemented by relaying the command to the nested display list.

3.2.14. FontRenderer

The `FontRenderer` is responsible to render font with software and hardware rendering. The `Font` used by the renderer are cached across the whole system and are shared between applications.



(a) (b)

Fig. 3.16.: Font texture and font geometry in the example application

a) Font texture used by the `FontRenderer`.

b) Geometry used to render the characters, display as quads.

Figure 3.16a shows the font texture that is generated by the FontRenderer and uploaded to the GPU. When comparing with the view of the example application, one can see that the font texture contains all used characters. Both the buttons text and the action bars text were merged into one drawing operation and the characters are sorted into the columns by their width.

To draw the font to the screen, the renderer needs to generate a geometry to which the texture gets bound (Figure 3.16b). The geometry is generated on the CPU and then drawn via the OpenGL command `glDrawElements()`.

If the device supports OpenGL ES 3.0, the `FontRenderer` will update and upload the font cache texture asynchronously at the start of the frame, while the GPU is mostly idle. This saves a few milliseconds per frame. The cache texture is a Pixel Buffer Object (PBO), which makes a asynchronous upload possible [Guy13].

3.2.15. SurfaceFlinger

When all drawing commands are finished processing and `eglSwapBuffers()` was called, the surface is queued again and sent as a parcel to the `SurfaceFlinger`, where it will be composed and drawn on the screen.

3.2.16. Summary

(a) (b) (c) (d) (e)

Fig. 3.17.: The example application can be drawn with only five batches on the Nexus 7 (2013):

a) The layout draws the background image, which is a linear gradient.

b) Both the ActionBar and Button background NinePatch are drawn, which are both on the AssetAtlas texture. These two operations were merged into one batch.

c) A linear gradient is drawn for the ActionBar.

d) Text for the Button and the ActionBar is drawn, using the same font texture. Again, these two operations were merged into one batch, and the `FontRenderer` also used one font texture for both text elements.

e) The application's icon is drawn.

This was only a brief overview of all critical elements in the Android UI rendering pipeline. Figure 3.17 shows the visual steps taken by the renderer to display the view to the user. The complete display list can be found in the appendix (Appendix A), as well as the full list of generated OpenGL command calls (Appendix B).

3.2.17. Code Quality

All parts that are visible to and potentially used by a third party application developer are documented in great depth via JavaDoc. This includes many Java classes like the `View` and also some native classes used by the Native Development Kit (NDK).

But this documentation effort does not extend to the whole Android source code. As soon as one looks at classes not designed to be used by an app developer, almost no documentation exists. Newer files seem to have more comments embedded in the code than older ones. But there are also a lot of comments describing missing features ("Todo") and describing potentially bugs that no one has come around fixing yet.

The Android source code is under heavy development since late 2007, with many new features added that were not originally planned. It also shows in the source code, as new features are designed around old ones in order not to disturb them. This results in a lot of indirection and multiple code paths with complex decision logic.

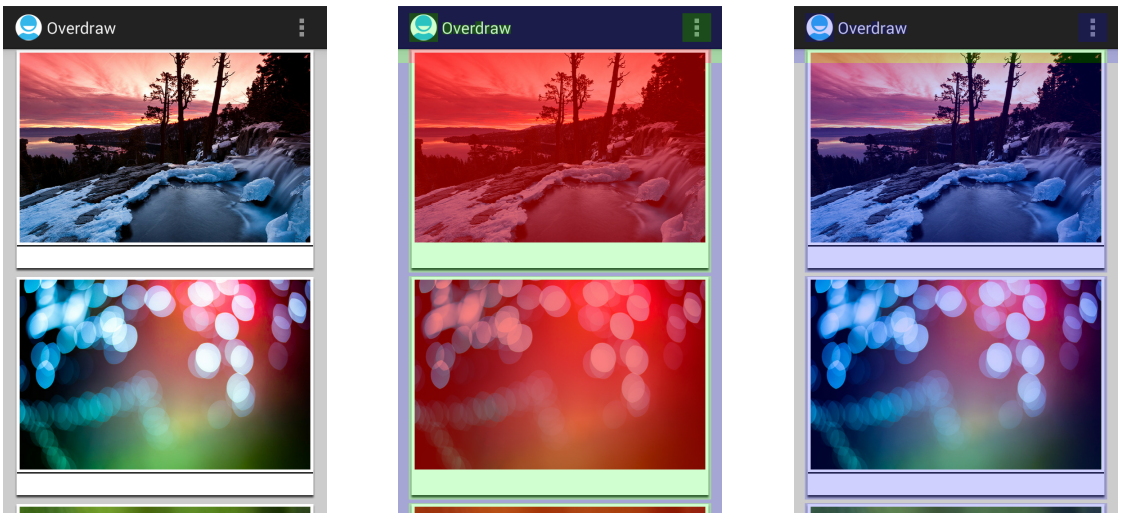
Naming of the classes can also be quite confusing. Some names are used multiple times in different components. For example, the Java `DisplayList` is actually a glorified `GL20RecordingCanvas`, but the C++ `DisplayList` is an array of drawing operations. These two classes are used in the Android rendering pipeline, but do not quite have the same meaning.

3.3. Overdraw

In December 2012 Romain Guy, at the time a core member of the Android Graphics Team, published an article about Android performance. Titled "Android Performance Case Study", this article described how to use the Android debugging tools

to optimize performance. The focus of the article is on minimizing the problem of Overdraw [Guy12a].

This article was published in 2012 when Android 4.2 was the newest release. Since then, a lot of optimizations have been incorporated into the rendering pipeline, but they are still referenced today as one of the most valuable reference on how to optimize an Android application.



(a) (b) (c)

Fig. 3.18.: Font texture and font geometry used in the example application

a) Normal view of the application.

b) Unoptimized state of application, many pixels overdrawn 3 times (red areas). Average overdraw is 4.13 times per pixel.

c) Optimized application, large areas are only overdrawn 1 time (blue areas). Average overdraw is 1.80 times per pixel.

To verify if these findings by Guy are still valid with Android 4.4, overdraw was tested with an example application. This example application was originally engineered by Guy [Guy12b], featuring intentionally high overdraw (Figure 3.18b). The application is showing multiple images in a list. A NinePatch is used as a border for the images, making it look like an image taken by a instant camera. Finally, the images are displayed with a small delay, simulating slow loading. In order to convey this to the user, a placeholder image is used in place of the real images. The view hierarchy is shown in Figure 3.19.

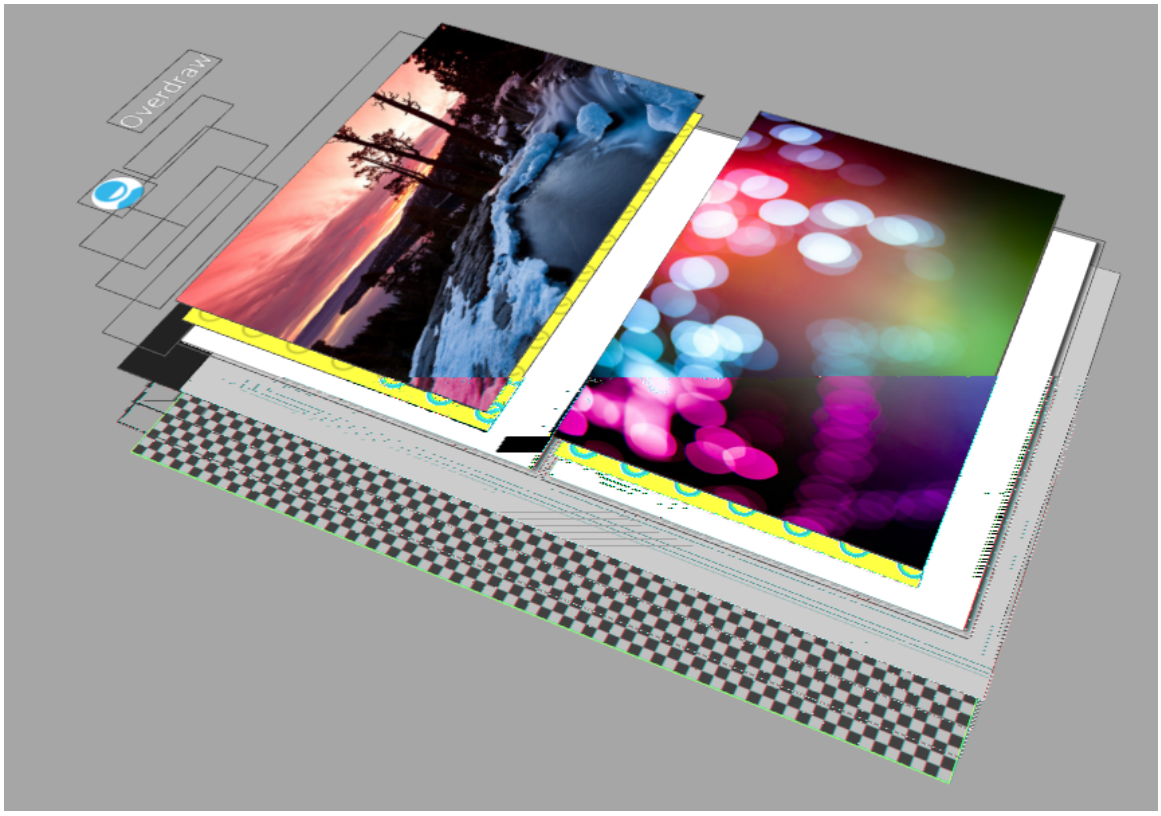


Fig. 3.19.: View hierarchy generated with DroidInspector [Sri13]

Multiple optimizations were added to reduce overdraw (Figure 3.18c). A non-visible background was removed by setting the window background to `null`. In addition, placeholder images behind the real images were removed once the real image finished loading by setting the image background to `null`. Finally, the used NinePatch was modified to be fully translucent at the position where the image is displayed, effectively cutting a hole in it. All optimizations can be toggled via the menu.

For testing, the Galaxy Nexus with an PowerVR SGX540 and Nexus 7 (2013) with a Qualcomm Adreno 320 were chosen to represent the TBRs (subsection 2.3.2). To represent IMRs (subsection 2.3.1), the Nexus 7 (2012) was chosen, which uses a Nvidia Tegra 3. All devices were running Android 4.4.2. Testing was done by scrolling down the image list revealing one new image and scrolling up again multiple times.

		Draw	Process	Execute
Galaxy Nexus	Full Overdraw	0.27 ms	0.67 ms	0.27 ms
	Minimized Overdraw	0.27 ms	0.58 ms	0.31 ms
Nexus 7 (2012)	Full Overdraw	0.29 ms	0.525 ms	0.63 ms
	Minimized Overdraw	0.29 ms	0.44 ms	0.54 ms
Nexus 7 (2013)	Full Overdraw	0.43 ms	0.98 ms	0.46 ms
	Minimized Overdraw	0.43 ms	0.79 ms	0.46 ms
Nexus 7 (2013) defering disabled	Full Overdraw	0.4 ms	1.04 ms	0.43 ms
	Minimized Overdraw	0.4 ms	0.82 ms	0.43 ms

Table 3.1.: Measurement results of the example application on different devices. The displayed values are the median of 128 sample frames.

Test results (Table 3.1) were taken with the `dumpysys` command[17] which outputs samples taken over the last 128 frames. The Application was tested twice, once with all optimizations disabled and once enabled. Additionally, deferring was disabled on the Nexus 7 (2013) for one test by setting the `debug.hwui.disable_draw_defer` property to false[18]. For complete measurement results of all devices see Appendix C.

The "Draw" time represents the time it takes the `HardwareRenderer` to call and run `view.getDisplayList()`. This includes the user-supplied `onDraw()` method. "Process" time represents the drawing of the display list to the underlying canvas. It includes reordering and merging, as well as actually dispatching OpenGL commands. Finally, the "Execute" time is how long calling `eglSwapBuffers` takes. Most TBR GPUs are only now starting to draw the view to the screen. IMR on the other hand are only waiting for all drawing commands to be finished.

Draw time was not affected by the optimizations and does not differ for any of the devices. This was expected, as the optimizations did not change any `onDraw(...)` method.

"Process" time was reduced on all devices with the optimizations enabled. The two removed backgrounds are partially responsible for this, as they do not need to be

[17] `adb shell dumpsys gfxinfo com.example.overdraw`
[18] `adb shell setprop debug.hwui.disable_draw_defer false`

added to the display list anymore, reducing the complexity of the view hierarchy. These backgrounds would be concealed by other elements and therefore TBRs would not render them. IMRs like the Nexus 7 (2012) are wasting processing time on these elements. However, Android 4.4 detects both of these fully concealed backgrounds and does not issue OpenGL calls for them, reducing the "Process" time. The optimized NinePatch affects the "Process" as well as the "Execute" time. The texture assigned to the content is empty and the geometry can be omitted so it does not have to be rendered.

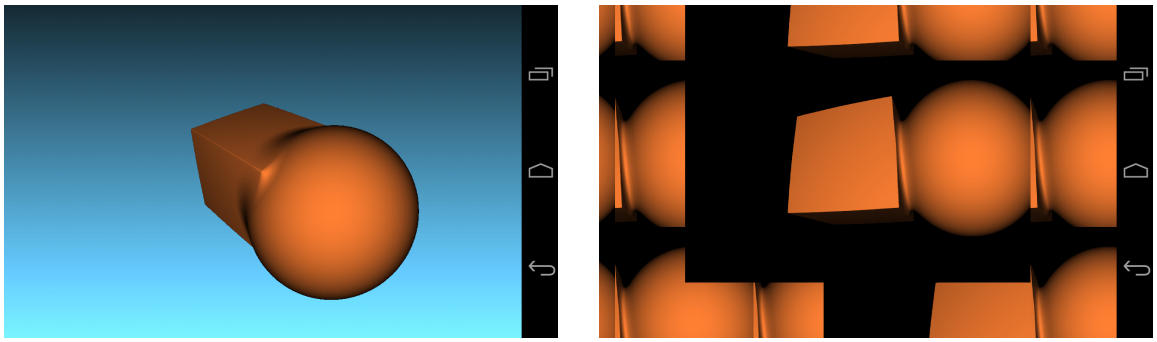
Execution time was only reduced on the Nexus 7 (2012), indicating a lower GPU load.

In conclusion, overdraw is not a major problem on modern Android devices anymore. Complex view hierarchies including a lot of drawables are still the root cause of most performance issues, which are indicated by the draw and process time. The debug visualization of the overdraw is showing a symptom of the problem, but not the problem itself. Developers who use the visualization to optimize their application usually get a small performance boost, but only because they incidentally reduce the view hierarchy complexity and number of drawables. More rewarding is using the Hierarchy Viewer included in the Android Monitor and removing superfluous Views. If the actual draw time is too high this could also indicate that too much work is done in the `onDraw(...)` methods. By using a system trace these issues can also be tracked down. Simplifying NinePatches can lead to some improvements, but are generally not worth the effort.

3.4. Mobile Drivers and Vendor Tools

In order to test the vendor supplied tools for debugging OpenGL applications on Android, a small demo application was used. This application only displays one quad to the screen and relies on a complex fragment shader to render the scene (Figure 3.20a). This is done via ray marching, which is a form of ray tracing. While the OpenGL shader code is valid according to the specifications, most of the mobile devices have trouble executing it. This results in a number of artifacts (Figure 3.20b),

device crashes and sometimes in compilation errors in the driver. The shader caused no issues on AMD's, Nvidia's desktop and Qualcomm's Adreno GPUs.



(a) (b)

Fig. 3.20.: Example OpenGL application, using ray marching in the fragment shader to render the scene.

a) Correct rendering of the scene without artifacts.

b) Artifacts caused by a faulty graphics driver.

To debug these artifacts, various vendor tools were tested under Ubuntu 12.04 and Windows 7. In almost two weeks of attempts, no tool was able to attach to any device and debug any OpenGL applications correctly. Nvidia's Perf HUD ES did not connect to the Nexus 7 (2012) running Android 4.4.2 and no support was available in their official developer forum. Downgrading and flashing the device with a special version of Android 4.2 supplied by Nvidia made the tool work. But as this Android version is already outdated, no tests were conducted.

Qualcomms Adreno SDK did also not connect to any device. Furthermore, the demo application "Neocore" refused to launch on an Android 4.4 device and crashed on startup.

The PowerVR SDK could connect via ADB to a Galaxy Nexus, but only managed to read the profiling data. No API trace could be generated. Furthermore, the offline shader compiler crashed with the valid but complex fragment shader code.

As none of the tested tools worked with the application, the shader coded had to be debugged and profiled by trial and error. Compilation failures of the shader can be debugged by printing the log to the console, but not all drivers managed to produce

a helpful error message. Commenting out various code pieces and trying again was the last resort on those devices. To test whether a specific code path was chosen, a path can output a special pixel color to the screen.

4. Conclusion and Outlook

The current state of the Android graphics pipeline was explored in this thesis, giving an overview of the technology and operating principle of the UI rendering. Not only because Android is supporting both software and hardware rendering, the rendering and UI code is very complex. The nearly absent documentation does not help with understanding the code. So this thesis offers a solid starting point to deepen one's knowledge of the Android platform and the UI rendering in particular. It can serve as a foundation to every developer who wants to start working with the Android graphics source code.

Hardware accelerated UI rendering seems to have a bright future in the Android world and is a huge performance improvement. However, dropping software rendering support seems not very likely in the near future. Most system applications still use software rendering. Not all canvas operations are supported with OpenGL, and some may never be. The available Android debugging tools, like the Android Monitor, are a good start, but need a lot more work. Vendor tools in particular need to be updated more often to support the newest Android version. The evaluation of problem areas such as overdraw and graphics driver support have been a brief analysis, but further studies are needed.

A. Display list for the example view

```
Start display list (0x5ea4f008, PhoneWindow.DecorView, render=1)
  Save 3
  ClipRect 0.00, 0.00, 720.00, 1184.00
  SetupShader, shader 0x5ea5af08
  Draw Rect    0.00    0.00  720.00 1184.00
  ResetShader
  Draw Display List 0x5ea64d30, flags 0x244053
  Start display list (0x5ea64d30, ActionBarOverlayLayout, render=1)
    Save 3
    ClipRect 0.00, 0.00, 720.00, 1184.00
    Draw Display List 0x5ea5ad78, flags 0x24053
    Start display list (0x5ea5ad78, FrameLayout, render=1)
      Save 3
      Translate (left, top) 0, 146
      ClipRect 0.00, 0.00, 720.00, 1038.00
      Draw Display List 0x5ea59bf8, flags 0x224053
      Start display list (0x5ea59bf8, RelativeLayout, render=1)
        Save 3
        ClipRect 0.00, 0.00, 720.00, 1038.00
        Save flags 3
        ClipRect   32.00   32.00  688.00 1006.00
        Draw Display List 0x5cfee368, flags 0x224073
        Start display list (0x5cfee368, Button, render=1)
          Save 3
          Translate (left, top) 32, 32
          ClipRect 0.00, 0.00, 243.00, 96.00
          Draw patch    0.00    0.00  243.00   96.00
          Save flags 3
          ClipRect   24.00    0.00  219.00   80.00
          Translate by 24.000000 23.000000
          Draw Text of count 12, bytes 24
          Restore to count 1
        Done (0x5cfee368, Button)
        Restore to count 1
```

```
35        Done (0x5ea59bf8, RelativeLayout)
36      Done (0x5ea5ad78, FrameLayout)
37      Draw Display List 0x5ea64ac8, flags 0x24053
38      Start display list (0x5ea64ac8, ActionBarContainer, render=1)
39        Save 3
40        Translate (left, top) 0, 50
41        ClipRect 0.00, 0.00, 720.00, 96.00
42        Draw patch    0.00    0.00  720.00   96.00
43        Draw Display List 0x5ea64910, flags 0x224053
44        Start display list (0x5ea64910, ActionBarView, render=1)
45          Save 3
46          ClipRect 0.00, 0.00, 720.00, 96.00
47          Draw Display List 0x5ea63790, flags 0x224053
48          Start display list (0x5ea63790, LinearLayout, render=1)
49            Save 3
50            Translate (left, top) 17, 0
51            ClipRect 0.00, 0.00, 265.00, 96.00
52            Draw Display List 0x5ea5fe80, flags 0x224053
53            Start display list (0x5ea5fe80,
54                                ActionBarView.HomeView, render=1)
55              Save 3
56              ClipRect 0.00, 0.00, 80.00, 96.00
57              Draw Display List 0x5ea5ed00, flags 0x224053
58              Start display list (0x5ea5ed00, ImageView, render=1)
59                Save 3
60                Translate (left, top) 8, 16
61                ClipRect 0.00, 0.00, 64.00, 64.00
62                Save flags 3
63                ConcatMatrix
64                  [0.67 0.00 0.00] [0.00 0.67 0.00] [0.00 0.00 1.00]
65                Draw bitmap 0x5d33ae70 at 0.000000 0.000000
66                Restore to count 1
67              Done (0x5ea5ed00, ImageView)
68            Done (0x5ea5fe80, ActionBarView.HomeView)
69            Draw Display List 0x5ea63618, flags 0x224053
70            Start display list (0x5ea63618, LinearLayout, render=1)
71              Save 3
72              Translate (left, top) 80, 23
73              ClipRect 0.00, 0.00, 185.00, 49.00
74              Save flags 3
75              ClipRect    0.00    0.00  169.00   49.00
76              Draw Display List 0x5ea634a0, flags 0x224073
77              Start display list (0x5ea634a0, TextView, render=1)
78                Save 3
```

```
79                ClipRect 0.00, 0.00, 169.00, 49.00
80                Save flags 3
81                ClipRect    0.00    0.00  169.00   49.00
82                Draw Text of count 9, bytes 18
83                Restore to count 1
84              Done (0x5ea634a0, TextView)
85              Restore to count 1
86            Done (0x5ea63618, LinearLayout)
87          Done (0x5ea63790, LinearLayout)
88        Done (0x5ea64910, ActionBarView)
89      Done (0x5ea64ac8, ActionBarContainer)
90      Draw patch    0.00  146.00  720.00  178.00
91    Done (0x5ea64d30, ActionBarOverlayLayout)
92  Done (0x5ea4f008, PhoneWindow.DecorView)
```

B. OpenGL commands for the example view

```
1  eglCreateContext(version = 1, context = 0)
2  eglMakeCurrent(context = 0)
3  glGetIntegerv(pname = GL_MAX_TEXTURE_SIZE, params = [2048])
4  glGetIntegerv(pname = GL_MAX_TEXTURE_SIZE, params = [2048])
5  glGetString(name = GL_VERSION) = OpenGL ES 2.0 14.01003
6  glGetIntegerv(pname = GL_MAX_TEXTURE_SIZE, params = [2048])
7  glGenBuffers(n = 1, buffers = [1])
8  glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 1)
9  glBufferData(target = GL_ARRAY_BUFFER, size = 64, data = [64 bytes],
10             usage = GL_STATIC_DRAW)
11 glDisable(cap = GL_SCISSOR_TEST)
12 glActiveTexture(texture = GL_TEXTURE0)
13 glGenBuffers(n = 1, buffers = [2])
14 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 2)
15 glBufferData(target = GL_ARRAY_BUFFER, size = 131072, data = 0x0,
16             usage = GL_DYNAMIC_DRAW)
17 glGetIntegerv(pname = GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
18              params = [16])
19 glGetIntegerv(pname = GL_MAX_TEXTURE_SIZE, params = [2048])
20 glGenTextures(n = 1, textures = [1])
21 glBindTexture(target = GL_TEXTURE_2D, texture = 1)
22 glEGLImageTargetTexture2DOES(target = GL_TEXTURE_2D,
23                               image = 2138532008)
24 glGetError(void) = (GLenum) GL_NO_ERROR
25 glDisable(cap = GL_DITHER)
26 glClearColor(red = 0,000000, green = 0,000000, blue = 0,000000,
27             alpha = 0,000000)
28 glEnableVertexAttribArray(index = 0)
29 glDisable(cap = GL_BLEND)
30 glGenTextures(n = 1, textures = [2])
31 glBindTexture(target = GL_TEXTURE_2D, texture = 2)
```

```
32 glPixelStorei(pname = GL_UNPACK_ALIGNMENT, param = 1)
33 glTexImage2D(target = GL_TEXTURE_2D, level = 0,
34              internalformat = GL_ALPHA, width = 1024, height = 512,
35              border = 0, format = GL_ALPHA, type = GL_UNSIGNED_BYTE,
36              pixels = [])
37 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MIN_FILTER,
38                 param = 9728)
39 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MAG_FILTER,
40                 param = 9728)
41 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_S,
42                 param = 33071)
43 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_T,
44                 param = 33071)
45 glViewport(x = 0, y = 0, width = 800, height = 1205)
46 glPixelStorei(pname = GL_UNPACK_ALIGNMENT, param = 1)
47 glTexSubImage2D(target = GL_TEXTURE_2D, level = 0,
48                 xoffset = 0, yoffset = 0,
49                 width = 1024, height = 80, format = GL_ALPHA,
50                 type = GL_UNSIGNED_BYTE, pixels = 0x697b7008)
51 glInsertEventMarkerEXT(length = 0, marker = Flush)
52 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 0)
53 glBindTexture(target = GL_TEXTURE_2D, texture = 1)
54 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_S,
55                 param = 33071)
56 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_T,
57                 param = 33071)
58 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MIN_FILTER,
59                 param = 9729)
60 glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MAG_FILTER,
61                 param = 9729)
62 glCreateShader(type = GL_VERTEX_SHADER) = (GLuint) 1
63 glShaderSource(shader = 1, count = 1, string =
64     attribute vec4 position;
65     attribute vec2 texCoords;
66     uniform mat4 projection;
67     uniform mat4 transform;
68     varying vec2 outTexCoords;
69
70     void main(void) {
71         outTexCoords = texCoords;
72         gl_Position = projection * transform * position;
73     }
74 , length = [0])
75 glCompileShader(shader = 1)
```

```
76 glGetShaderiv(shader = 1, pname = GL_COMPILE_STATUS,
77               params = [1])
78 glCreateShader(type = GL_FRAGMENT_SHADER) = (GLuint) 2
79 glShaderSource(shader = 2, count = 1, string =
80     precision mediump float;
81
82     varying vec2 outTexCoords;
83     uniform sampler2D baseSampler;
84
85     void main(void) {
86         gl_FragColor = texture2D(baseSampler, outTexCoords);
87     }
88 , length = [0])
89 glCompileShader(shader = 2)
90 glGetShaderiv(shader = 2, pname = GL_COMPILE_STATUS, params = [1])
91 glCreateProgram(void) = (GLuint) 3
92 glAttachShader(program = 3, shader = 1)
93 glAttachShader(program = 3, shader = 2)
94 glBindAttribLocation(program = 3, index = 0, name = position)
95 glBindAttribLocation(program = 3, index = 1, name = texCoords)
96 glGetProgramiv(program = 3, pname = GL_ACTIVE_ATTRIBUTES, params = [2])
97 glGetProgramiv(program = 3, pname = GL_ACTIVE_ATTRIBUTE_MAX_LENGTH,
98                params = [10])
99 glGetActiveAttrib(program = 3, index = 0, bufsize = 10, length = [0],
100                   size = [1], type = [GL_FLOAT_VEC4], name = position)
101 glGetActiveAttrib(program = 3, index = 1, bufsize = 10, length = [0],
102                   size = [1], type = [GL_FLOAT_VEC2], name = texCoords)
103 glGetProgramiv(program = 3, pname = GL_ACTIVE_UNIFORMS, params = [3])
104 glGetProgramiv(program = 3, pname = GL_ACTIVE_UNIFORM_MAX_LENGTH,
105                params = [12])
106 glGetActiveUniform(program = 3, index = 0, bufsize = 12, length = [0],
107                    size = [1], type = [GL_FLOAT_MAT4], name = projection)
108 glGetActiveUniform(program = 3, index = 1, bufsize = 12, length = [0],
109                    size = [1], type = [GL_FLOAT_MAT4], name = transform)
110 glGetActiveUniform(program = 3, index = 2, bufsize = 12, length = [0],
111                    size = [1], type = [GL_SAMPLER_2D], name = baseSampler)
112 glLinkProgram(program = 3)
113 glGetProgramiv(program = 3, pname = GL_LINK_STATUS, params = [1])
114 glGetUniformLocation(program = 3, name = transform) = (GLint) 2
115 glGetUniformLocation(program = 3, name = projection) = (GLint) 1
116 glUseProgram(program = 3)
117 glGetUniformLocation(program = 3, name = baseSampler) = (GLint) 0
118 glUniform1i(location = 0, x = 0)
119 glUniformMatrix4fv(location = 1, count = 1, transpose = false,
```

```
120                     value = [ 0.0025,  0.0,          0.0, 0.0,
121                               0.0,    -0.001659751,  0.0, 0.0,
122                               0.0,     0.0,         -1.0, 0.0,
123                              -1.0,     1.0,         -0.0, 1.0])
124 glUniformMatrix4fv(location = 2, count = 1, transpose = false,
125                     value = [800.0,    0.0, 0.0, 0.0,
126                                0.0, 1205.0, 0.0, 0.0,
127                                0.0,    0.0, 1.0, 0.0,
128                                0.0,    0.0, 0.0, 1.0])
129 glEnableVertexAttribArray(index = 1)
130 glVertexAttribPointer(indx = 0, size = 2, type = GL_FLOAT,
131                       normalized = false, stride = 16, ptr = 0x681e7af4)
132 glVertexAttribPointer(indx = 1, size = 2, type = GL_FLOAT,
133                       normalized = false, stride = 16, ptr = 0x681e7afc)
134 glVertexAttribPointerData(indx = 0, size = 2, type = GL_FLOAT,
135                           normalized = false, stride = 16, ptr = 0x??,
136                           minIndex = 0, maxIndex = 4)
137 glVertexAttribPointerData(indx = 1, size = 2, type = GL_FLOAT,
138                           normalized = false, stride = 16, ptr = 0x??,
139                           minIndex = 0, maxIndex = 4)
140 glDrawArrays(mode = GL_TRIANGLE_STRIP, first = 0, count = 4)
141 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 2)
142 glBufferSubData(target = GL_ARRAY_BUFFER, offset = 0, size = 576,
143                 data = [576 bytes])
144 glBufferSubData(target = GL_ARRAY_BUFFER, offset = 576, size = 192,
145                 data = [192 bytes])
146 glEnable(cap = GL_BLEND)
147 glBlendFunc(sfactor = GL_SYNC_FLUSH_COMMANDS_BIT,
148             dfactor = GL_ONE_MINUS_SRC_ALPHA)
149 glUniformMatrix4fv(location = 2, count = 1, transpose = false,
150                     value = [1.0, 0.0, 0.0, 0.0,
151                              0.0, 1.0, 0.0, 0.0,
152                              0.0, 0.0, 1.0, 0.0,
153                              0.0, 0.0, 0.0, 1.0])
154 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 0)
155 glGenBuffers(n = 1, buffers = [3])
156 glBindBuffer(target = GL_ELEMENT_ARRAY_BUFFER, buffer = 3)
157 glBufferData(target = GL_ELEMENT_ARRAY_BUFFER, size = 24576,
158              data = [24576 bytes], usage = GL_STATIC_DRAW)
159 glVertexAttribPointer(indx = 0, size = 2, type = GL_FLOAT,
160                       normalized = false, stride = 16, ptr = 0xbefdcf18)
161 glVertexAttribPointer(indx = 1, size = 2, type = GL_FLOAT,
162                       normalized = false, stride = 16, ptr = 0xbefdcf20)
163 glVertexAttribPointerData(indx = 0, size = 2, type = GL_FLOAT,
```

```
164                             normalized = false, stride = 16, ptr = 0x??,
165                             minIndex = 0, maxIndex = 48)
166 glVertexAttribPointerData(indx = 1, size = 2, type = GL_FLOAT,
167                             normalized = false, stride = 16, ptr = 0x??,
168                             minIndex = 0, maxIndex = 48)
169 glDrawElements(mode = GL_MAP_INVALIDATE_RANGE_BIT, count = 72,
170                 type = GL_UNSIGNED_SHORT, indices = 0x0)
171 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 2)
172 glBufferSubData(target = GL_ARRAY_BUFFER, offset = 768, size = 576,
173                  data = [576 bytes])
174 glDisable(cap = GL_BLEND)
175 glUniformMatrix4fv(location = 2, count = 1, transpose = false,
176                     value = [1.0, 0.0, 0.0, 0.0,
177                              0.0, 1.0, 0.0, 0.0,
178                              0.0, 0.0, 1.0, 0.0,
179                              0.0, 33.0, 0.0, 1.0])
180 glVertexAttribPointer(indx = 0, size = 2, type = GL_FLOAT,
181                       normalized = false, stride = 16, ptr = 0x300)
182 glVertexAttribPointer(indx = 1, size = 2, type = GL_FLOAT,
183                       normalized = false, stride = 16, ptr = 0x308)
184 glDrawElements(mode = GL_MAP_INVALIDATE_RANGE_BIT, count = 54,
185                 type = GL_UNSIGNED_SHORT, indices = 0x0)
186 glEnable(cap = GL_BLEND)
187 glUniformMatrix4fv(location = 2, count = 1, transpose = false,
188                     value = [1.0, 0.0, 0.0, 0.0,
189                              0.0, 1.0, 0.0, 0.0,
190                              0.0, 0.0, 1.0, 0.0,
191                              0.0, 0.0, 0.0, 1.0])
192 glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 0)
193 glBindTexture(target = GL_TEXTURE_2D, texture = 2)
194 glVertexAttribPointer(indx = 0, size = 2, type = GL_FLOAT,
195                       normalized = false, stride = 16, ptr = 0x696bd008)
196 glVertexAttribPointer(indx = 1, size = 2, type = GL_FLOAT,
197                       normalized = false, stride = 16, ptr = 0x696bd010)
198 glVertexAttribPointerData(indx = 0, size = 2, type = GL_FLOAT,
199                       normalized = false, stride = 16, ptr = 0x??,
200                       minIndex = 0, maxIndex = 80)
201 glVertexAttribPointerData(indx = 1, size = 2, type = GL_FLOAT,
202                       normalized = false, stride = 16, ptr = 0x??,
203                       minIndex = 0, maxIndex = 80)
204 glDrawElements(mode = GL_MAP_INVALIDATE_RANGE_BIT, count = 120,
205                 type = GL_UNSIGNED_SHORT, indices = 0x0)
206 glGenTextures(n = 1, textures = [3])
207 glBindTexture(target = GL_TEXTURE_2D, texture = 3)
```

```
glPixelStorei(pname = GL_UNPACK_ALIGNMENT, param = 4)
glTexImage2D(target = GL_TEXTURE_2D, level = 0, internalformat = GL_RGBA,
             width = 64, height = 64, border = 0, format = GL_RGBA,
             type = GL_UNSIGNED_BYTE, pixels = 0x420cd930)
glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MIN_FILTER,
                param = 9728)
glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_MAG_FILTER,
                param = 9728)
glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_S,
                param = 33071)
glTexParameteri(target = GL_TEXTURE_2D, pname = GL_TEXTURE_WRAP_T,
                param = 33071)
glUniformMatrix4fv(location = 2, count = 1, transpose = false,
                   value = [64.0,  0.0, 0.0, 0.0,
                             0.0, 64.0, 0.0, 0.0,
                             0.0,  0.0, 1.0, 0.0,
                            16.0, 38.0, 0.0, 1.0])
glBindBuffer(target = GL_ARRAY_BUFFER, buffer = 1)
glVertexAttribPointer(indx = 0, size = 2, type = GL_FLOAT,
                      normalized = false, stride = 16, ptr = 0x0)
glVertexAttribPointer(indx = 1, size = 2, type = GL_FLOAT,
                      normalized = false, stride = 16, ptr = 0x8)
glBindBuffer(target = GL_ELEMENT_ARRAY_BUFFER, buffer = 0)
glDrawArrays(mode = GL_TRIANGLE_STRIP, first = 0, count = 4)
glGetError(void) = (GLenum) GL_NO_ERROR
eglSwapBuffers()
```

C. Overdraw measurements

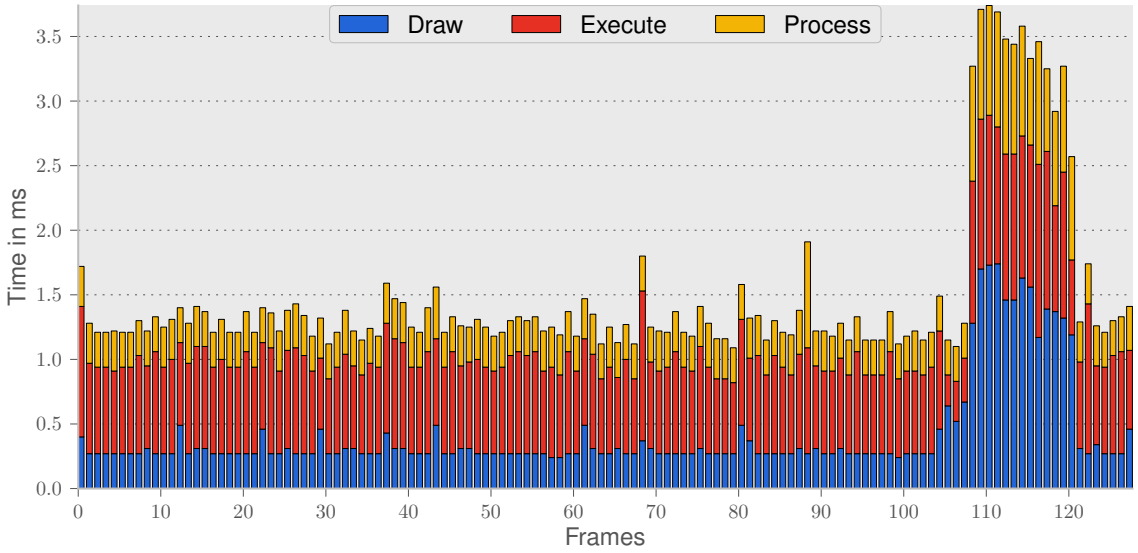


Fig. C.1.: Galaxy Nexus: Full overdraw

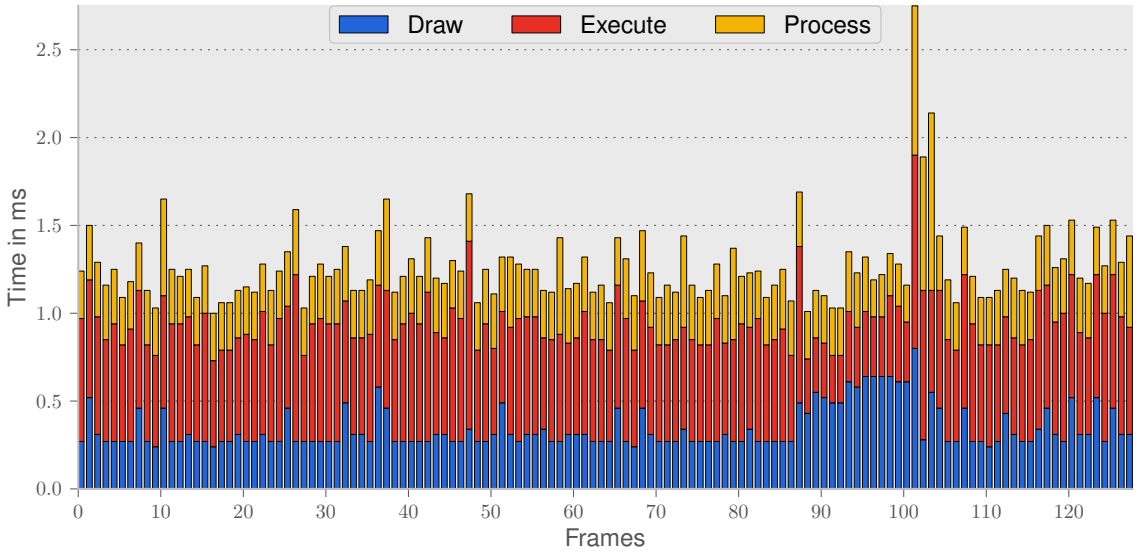


Fig. C.2.: Galaxy Nexus: Optimized overdraw

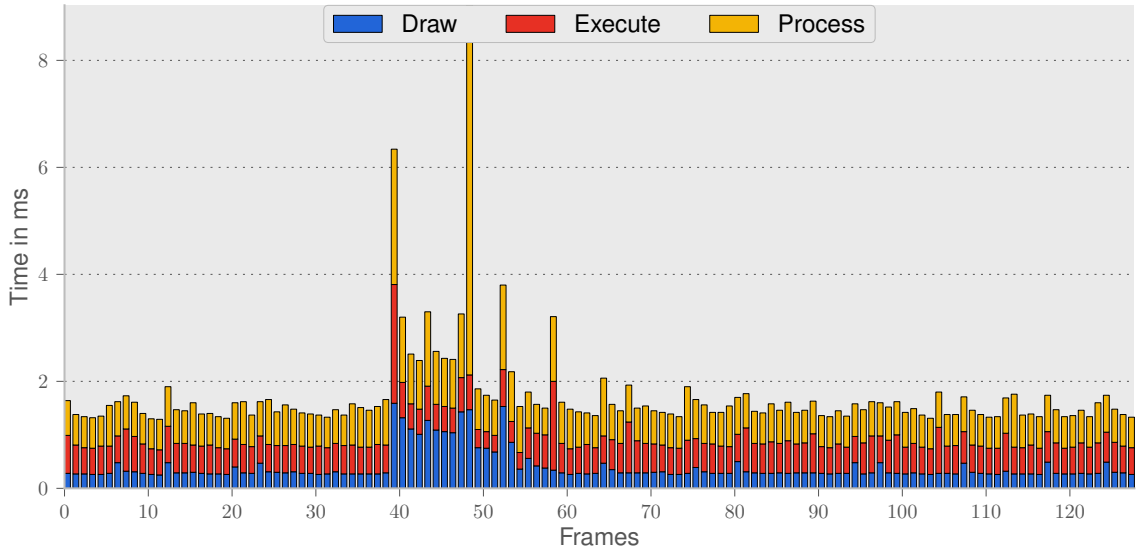


Fig. C.3.: Nexus 7 (2012): Full overdraw

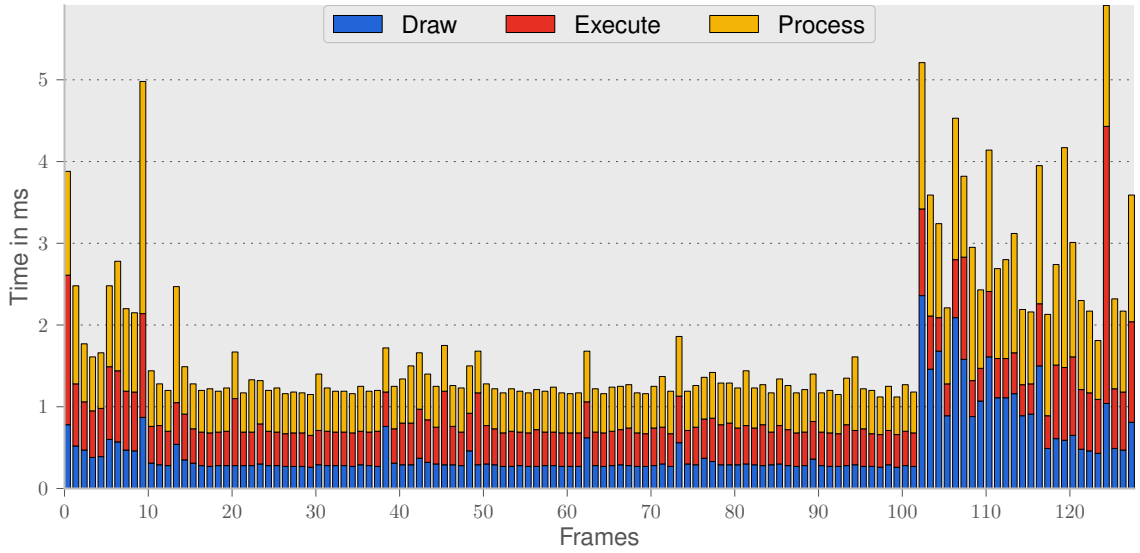


Fig. C.4.: Nexus 7 (2012): Optimized overdraw

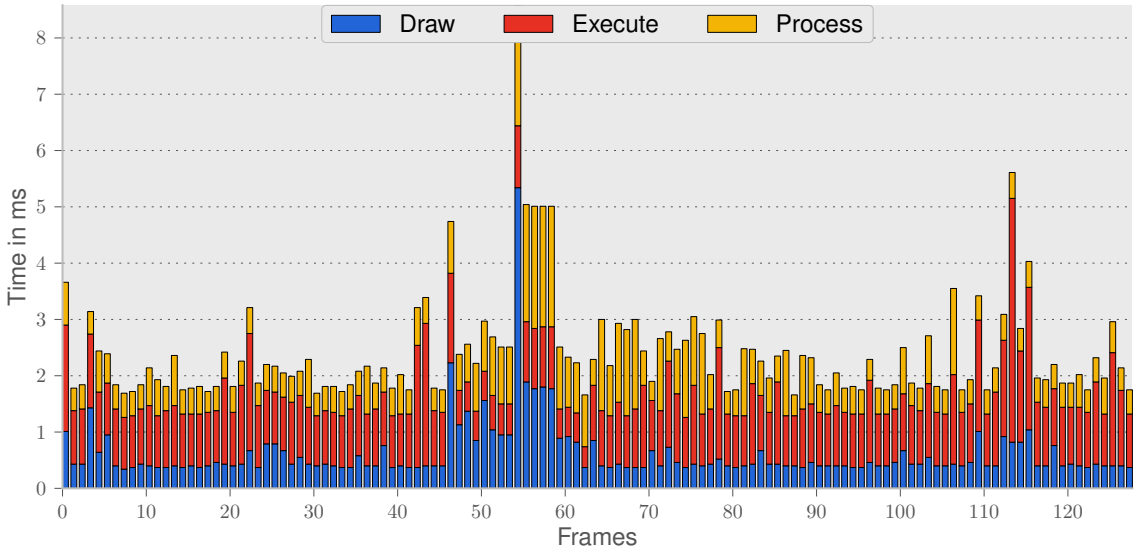


Fig. C.5.: Nexus 7 (2013): Full overdraw

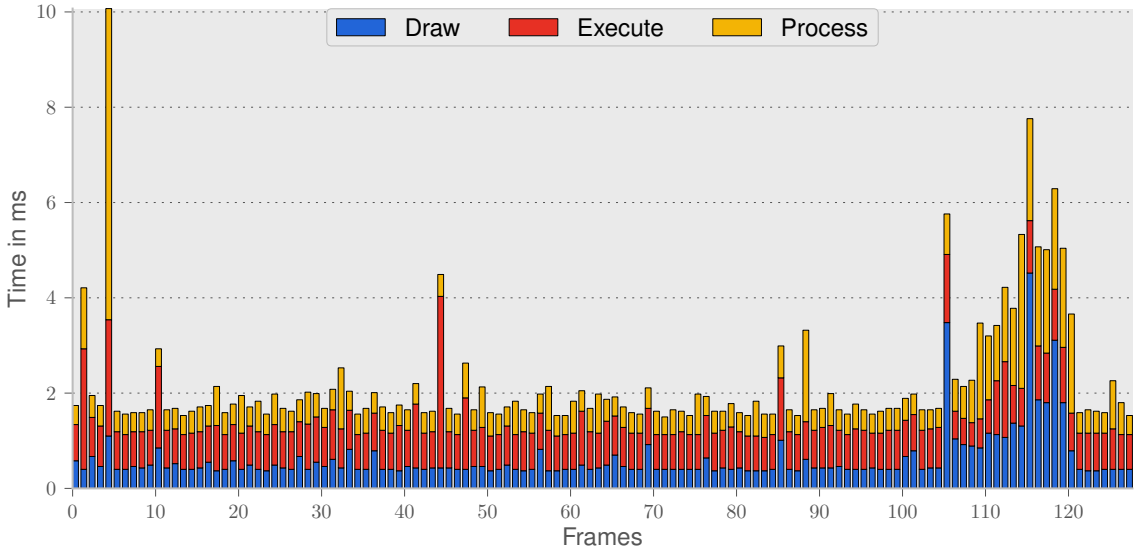


Fig. C.6.: Nexus 7 (2013): Optimized overdraw

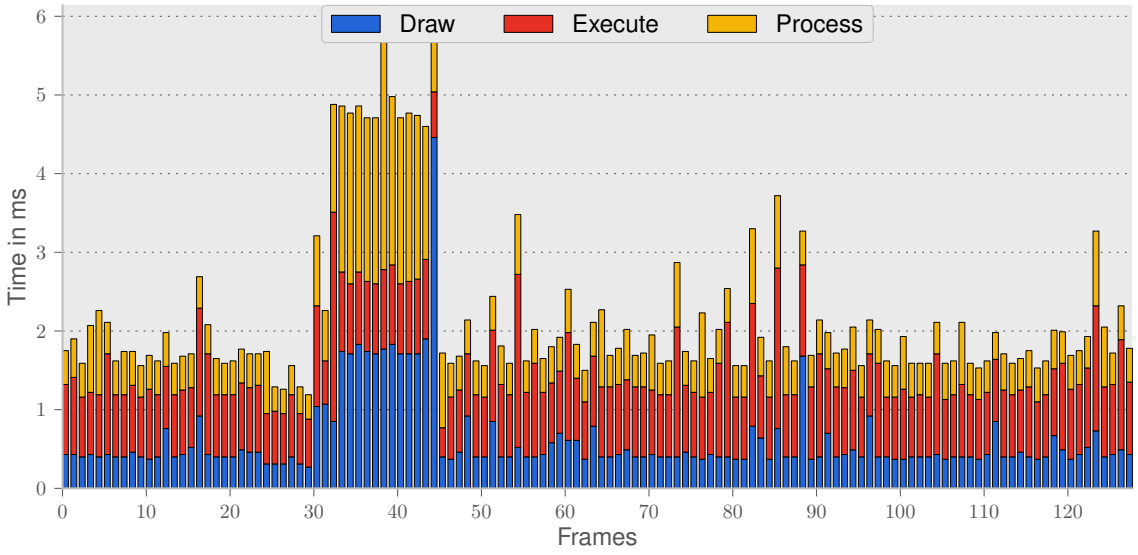


Fig. C.8.: Nexus 7 (2013): Optimized overdraw with disabled deferred rendering

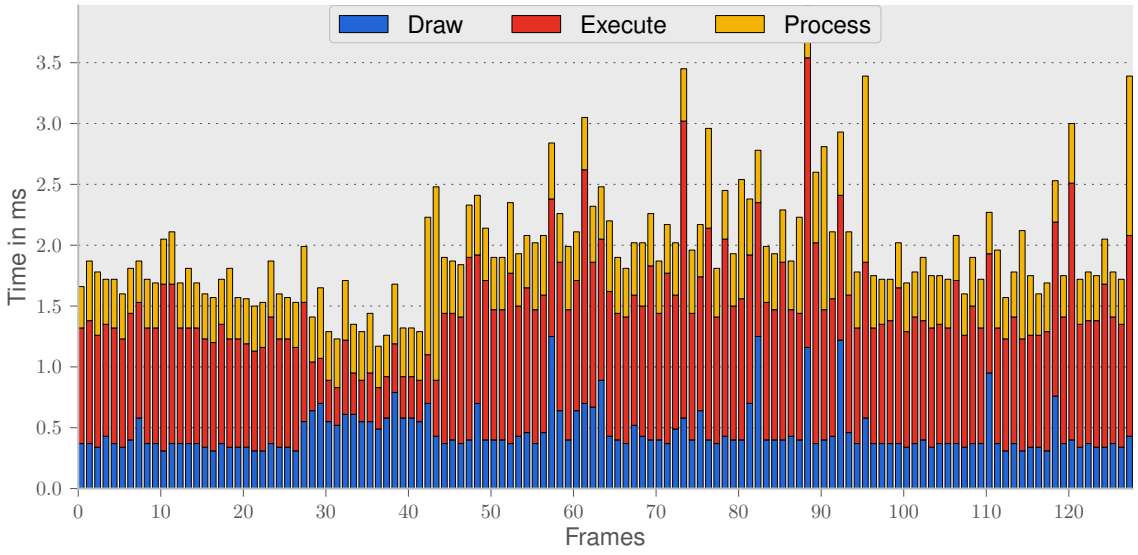


Fig. C.7.: Nexus 7 (2013): Full overdraw with disabled deferred rendering

All diagrams were generated with Matplotlib [Hun07].

Bibliography

[AHH08] Thomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. Third Edition. 2008 (Cited on page 8).

[Ana13] Anatomy of Android. *Anatomy of Android: Zygote*. 2013. URL: http://anatomyofandroid.com/2013/10/15/zygote/ (visited on 02/15/2014) (Cited on page 80).

[And13a] Android Open Source Project. *Activity*. 2013. URL: http://developer.android.com/reference/android/app/Activity.html (visited on 11/15/2013) (Cited on page 17).

[And13b] Android Open Source Project. *Canvas and Drawables: NinePatch*. 2013. URL: http://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch (visited on 01/03/2014) (Cited on page 79).

[And13c] Android Open Source Project. *NinePatch*. 2013. URL: http://developer.android.com/reference/android/graphics/NinePatch.html (visited on 01/03/2014) (Cited on page 79).

[And13d] Android Open Source Project. *Pack preloaded framework assets in a texture atlas*. Version 3b748a44c6bd2ea05fe16839caf73dbe50bd7ae9. 2013. URL: https://github.com/android/platform_frameworks_base/commit/3b748a44c6bd2ea05fe16839caf73dbe50bd7ae9 (visited on 01/11/2014) (Cited on pages 19, 20).

[And13e] Android Open Source Project. *SurfaceFlinger: SW-based vsync events*. Version faf77cce9d9ec0238d6999b3bd0d40c71ff403c5. 2013. URL: https://android.googlesource.com/platform/frameworks/native/+/

faf77cce9d9ec0238d6999b3bd0d40c71ff403c5 (visited on 02/11/2014) (Cited on page 27).

[And13f] Android Open Source Project. *View*. 2013. URL: http://developer.android.com/reference/android/view/View.html (visited on 01/03/2014) (Cited on pages 33, 34).

[Fat10] Kayvon Fatahalian. *From Shader Code to a Teraflop: How Shader Cores Work*. Stanford University. SIGGRAPH. 2010. URL: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf (visited on 11/15/2013) (Cited on page 11).

[FH08a] Kayvon Fatahalian and Mike Houston. "A Closer Look at GPUs." In: *ACM QUEUE* (Apr. 2008). URL: http://graphics.stanford.edu/~kayvonf/papers/fatahalianAcmQueue.pdf (visited on 11/18/2013) (Cited on page 10).

[FH08b] Kayvon Fatahalian and Mike Houston. "A Closer Look at GPUs." In: *Communications of the ACM* 51.10 (Oct. 2008). URL: http://graphics.stanford.edu/~kayvonf/papers/fatahalianCACM.pdf (visited on 11/18/2013) (Cited on page 6).

[Fow10] Mark Fowler. *ATI Radeon HD5000 Series: In Inside View*. AMD Inc. 2010. URL: http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_AMD.pdf (visited on 11/13/2013) (Cited on page 11).

[Fre13a] Freedreno Team. *Adreno tiling*. 2013. URL: https://github.com/freedreno/freedreno/wiki/Adreno-tiling (visited on 11/18/2013) (Cited on page 12).

[Fre13b] Freedreno Team. *Command Stream Format*. 2013. URL: https://github.com/freedreno/freedreno/wiki/Command-Stream-Format (visited on 11/18/2013) (Cited on page 13).

[Gan13] Guillem Vinals Gangolells. *It's all about triangles! Understanding the GPU in your pocket to write better code*. Imagination Technologies. Droid-

con Berlin. 2013. URL: http://de.droidcon.com/2013/node/177 (visited on 11/13/2013) (Cited on pages 10, 12, 79).

[Gie11a] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 2: GPU memory architecture and the Command Processor.* 2011. URL: http://fgiesen.wordpress.com/2011/07/02/a-trip-through-the-graphics-pipeline-2011-part-2/ (visited on 11/14/2013) (Cited on pages 3, 14).

[Gie11b] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 3: 3D pipeline overview, vertex processing.* 2011. URL: http://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/ (visited on 11/25/2013) (Cited on page 15).

[Gie11c] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 4: Texture samplers.* 2011. URL: http://fgiesen.wordpress.com/2011/07/04/a-trip-through-the-graphics-pipeline-2011-part-4/ (visited on 01/14/2014) (Cited on page 15).

[Gie11d] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 6: (Triangle) rasterization and setup.* 2011. URL: http://fgiesen.wordpress.com/2011/07/06/a-trip-through-the-graphics-pipeline-2011-part-6/ (visited on 11/18/2013) (Cited on page 8).

[Gie11e] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 8: Pixel processing – fork phase.* 2011. URL: http://fgiesen.wordpress.com/2011/07/10/a-trip-through-the-graphics-pipeline-2011-part-8/ (visited on 11/13/2013) (Cited on page 8).

[Gie11f] Fabian "ryg" Giesen. *A trip through the Graphics Pipeline 2011, Part 9: Pixel processing – join phase.* 2011. URL: http://fgiesen.wordpress.com/2011/07/12/a-trip-through-the-graphics-pipeline-2011-part-9/ (visited on 11/13/2013) (Cited on pages 11, 15).

[Guy13] Romain Guy. *Android 4.4 rendering pipeline improvements.* 2013. URL: https://plus.google.com/+RomainGuy/posts/9QSTyVCSoz3 (visited on 01/12/2014) (Cited on page 51).

[Guy12a] Romain Guy. *Android Performance Case Study*. 2012. URL: http://www.curious-creature.org/docs/android-performance-case-study-1.html (visited on 02/09/2014) (Cited on pages 42, 53).

[Guy12b] Romain Guy. *Android Performance in Practice*. 2012. URL: http://www.curious-creature.org/2012/12/06/android-performance-in-practice/ (visited on 02/09/2014) (Cited on page 53).

[GH11] Romain Guy and Chet Haase. *Accelerated Android Rendering*. Google. GoogleIO. 2011. URL: http://www.google.com/events/io/2011/sessions/accelerated-android-rendering.html (visited on 01/14/2014) (Cited on page 21).

[HG12] Chet Haase and Romain Guy. *For Butter or Worse: Smoothing Out Performance in Android UIs*. Google. GoogleIO. 2012. URL: https://developers.google.com/events/io/2012/sessions/gooio2012/109/ (visited on 01/29/2014) (Cited on pages 27, 28).

[Hac11] Dianne Hackborn. *How about some Android graphics true facts?* 2011. URL: https://plus.google.com/105051985738280261832/posts/2FXDCz8x93s/ (visited on 01/12/2014) (Cited on page 17).

[Hec95] Chris Hecker. "Perspective Texture Mapping, Part II: Rasterization." In: *Game Developer* (June 1995). URL: http://chrishecker.com/images/9/97/Gdmtex2.pdf (visited on 11/18/2013) (Cited on page 7).

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment." In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95 (Cited on page 72).

[Ima09] Imagination Technologies Ltd. *POWERVR MBX - Technology Overview*. 2009. URL: http://www.imgtec.com/factsheets/SDK/PowerVR%20Technology%20Overview.1.0.2e.External.pdf (visited on 11/13/2013) (Cited on pages 12, 79).

[Int08] Intel Software Solutions Group. *Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency*. 2008. URL: http://software.intel.com/sites/default/files/m/8/1/c/a/0/1371-Intel___

AVX_New_Frontiers_in_Performance_Improvements_and_Energy_Efficiency_WP.pdf (visited on 11/12/2013) (Cited on page 4).

[Jon14a] Jon Peddie Research. *AMD winner in Q2, Intel up, Nvidia down.* 2014. URL: http://jonpeddie.com/press-releases/details/amd-winner-in-q2-intel-up-nvidia-down/ (visited on 01/09/2014) (Cited on page 4).

[Jon14b] Jon Peddie Research. *Mobile Devices and the GPUs inside.* 2014. URL: http://jonpeddie.com/publications/mobile-devices-and-the-gpus-inside (visited on 01/09/2014) (Cited on page 4).

[Kar10] Radhika Karandikar. *Android Application Launch Part 2.* 2010. URL: http://multi-core-dump.blogspot.de/2010/04/android-application-launch-part-2.html (visited on 01/27/2014) (Cited on page 29).

[Kha14] Olga Kharif. *Bitcoin-Equipment Boom Benefits TSMC, AMD Sales, Report Says.* 2014. URL: http://www.bloomberg.com/news/2014-01-02/bitcoin-equipment-boom-benefiting-tsmc-amd-sales-report-says.html (visited on 02/12/2014) (Cited on page 4).

[Mer12] Bruce Merry. "Performance Tuning for Tile-Based Architectures." In: *OpenGL Insights.* Ed. by Patrick Cozzi and Christophe Riccio. CRC Press, July 2012, pp. 322–335. URL: http://www.openglinsights.com/ (Cited on page 11).

[Ope13a] OpenGL.org. *Fragment Shader.* 2013. URL: http://www.opengl.org/wiki/Fragment_Shader (visited on 11/18/2013) (Cited on page 8).

[Ope13b] OpenGL.org. *Geometry Shader.* 2013. URL: http://www.opengl.org/wiki/Geometry_Shader (visited on 11/15/2013) (Cited on page 7).

[Ope13c] OpenGL.org. *Primitive Assembly.* 2013. URL: http://www.opengl.org/wiki/Primitive_Assembly (visited on 11/18/2013) (Cited on page 7).

[Ope13d] OpenGL.org. *Rendering Pipeline Overview.* 2013. URL: http://www.opengl.org/wiki/Rendering_Pipeline_Overview (visited on 11/12/2013) (Cited on page 5).

[Ope13e] OpenGL.org. *Tessellation*. 2013. URL: http://www.opengl.org/wiki/Tessellation (visited on 11/15/2013) (Cited on page 6).

[Ope13f] OpenGL.org. *Vertex Processing*. 2013. URL: http://www.opengl.org/wiki/Vertex_Processing (visited on 11/14/2013) (Cited on page 6).

[Per07] Emil Persson. *Depth In-depth*. AMD Inc. 2007. URL: http://developer.amd.com/wordpress/media/2012/10/Depth_in-depth.pdf (visited on 11/13/2013) (Cited on page 79).

[Pin88] Juan Pineda. "A Parallel Algorithm for Polygon Rasterization." In: *Computer Graphics* 22.4 (Aug. 1988). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.4621&rep=rep1&type=pdf (visited on 11/18/2013) (Cited on page 8).

[Pur10] Tim Purcell. *Fast Tessellated Rendering on Fermi GF100*. Nvidia Corporation. 2010. URL: http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_NVIDIA.pdf (visited on 11/13/2013) (Cited on page 11).

[Ren11] Steve Rennich. *Fundamental Optimizations, Global Memory*. Nvidia Corporation. 2011. URL: http://www.stanford.edu/dept/ICME/docs/seminars/Rennich-2011-04-25.pdf (visited on 01/12/2014) (Cited on page 13).

[Sri13] Sriram Ramani. *Rendering Pipeline Overview*. 2013. URL: http://www.sriramramani.com/droidinspector/ (Cited on page 54).

Glossary

Early-Z is an optimization step in the render pipeline, which is traditionally executed right before the fragment shader and therefore before the Z-Test. It operates on a per-pixel level and allows the rejection of fragments which are occluded by the current content of the depth buffer. Early-Z is disabled if the fragment shader uses the `discard` instruction, writes to the depth-buffer or when alpha test is enabled [Per07, pp. 2-3]. (5, 10)

Hierarchical Z-Buffer (Hi-Z) allows the rejection of whole tiles on a hierarchical fashion. This is done by rendering a depth map from the scene into a low resolution buffer, in which non-occluded pixels (which represent tiles in the full resolution scene) can be rejected. This can be done multiple times with different resolutions to achieve the best result [Per07]. The same limitations as with Early-Z also apply. (5, 10)

Hidden Surface Removal (HSR) is, much like Early-Z, and optimization step in the render pipeline before the execution of the fragment shader. This technology is often used with Tiled-Rendering (see 2.3.2) [Gan13], where a Z-Buffer is saved in an extremely fast on-die memory and is used to reject pixels that will be occluded [Ima09, pp. 6-7]. The same limitations as with Early-Z also apply. (5, 12, 13)

NinePatch is a kind of bitmap, which uses a one-pixel border around the image to divide it into nine or more sections. Each of these sections will then be scaled to the content, with the mentioned border defining of the stretchable areas [And13c]. More information is available on the Android developer documentation [And13b]. (18–20, 23, 24, 29, 37, 47, 53, 54, 56)

Overdraw happens when an object is drawn to the framebuffer only to be completely or partially painted over by yet another object. This wasted processing time is called "overdraw". (10, 14, 25, 33, 42, 53)

Vertical Synchronization (VSync) is a way to prevent screen tearing, which happens when the display shows information from two or more frames at once. VSync prevents this by disallowing writes to the currently displayed framebuffer's memory. The driver commonly uses two framebuffers, one that is currently displayed and one that can be written to, and switches between them. This page flipping usually happens at the refresh rate of the display. (27, 28, 30–32, 38)

Zygote is an Android system service which is used to launch applications. It is one of the most important system services, as it is the parent of all launched application processes. On startup, `Zygote` starts a Java process and preloads frequently used Java classes from the Android framework. To start a new application, `Zygote` forks and the applications code is loaded in the child. No additional memory is allocated, as all needed Java classes are already in memory. Android implements a Copy-on-Write strategy [Ana13]. (18, 19, 29)