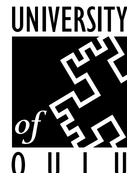

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING
DEGREE PROGRAMME IN INFORMATION ENGINEERING

A MOBILE VECTOR GRAPHICS QUALITY ANALYSIS TOOLKIT

Author ______________________
Sami Kyöstilä

Supervisor ______________________
Juha Röning

Accepted _____ /_____ 2008

Grade ______________________

Kyöstilä S. (2008) A Mobile Vector Graphics Quality Analysis Toolkit. Department of Electrical and Information Engineering, University of Oulu, Oulu, Finland. Master's thesis, 100 p.

ABSTRACT

High resolution displays, fluid user interfaces and impressive graphics have become essential features of modern mobile devices. This trend has prompted a move from traditional pixel-based graphics to more flexible and efficient vector graphics. The relationship between graphics content and performance, however, is not always straightforward with vector graphics. This discrepancy has resulted in applications suffering from a number of quality issues, such as poor performance, high memory usage or extraneous power consumption.

This thesis examines the process of analyzing these quality problems. We focus on mobile applications that use three-dimensional OpenGL ES or two-dimensional OpenVG graphics. To help solve a given quality issue, we classify it as a distinct class, based on the dominant cause behind the issue. We note that this classification process requires a great deal of information about the graphics of the examined application. Obtaining this quantity of information is not practical with the tools currently available.

To obtain this data, we will design and implement a set of tools. The key idea is to capture or trace the graphics drawing commands executed by an application into a file for further offline processing. These commands are then analyzed, processed and transformed to gain the necessary level of insight into the examined quality issue.

We then demonstrate the usage of the toolkit in examining poor application performance, a visual error in an application, the quality of a seemingly well-performing application, the performance profiles of a number of graphics implementations and the detailed graphics content features of an application.

Having completed these use cases successfully, we conclude that the tracer paradigm is a viable approach for analyzing quality issues in mobile vector graphics applications.

Keywords: graphical debugging, tracing, content features, OpenGL ES, OpenVG

Kyöstilä S. (2008) Mobiilivektorigrafiikkasovellusten laadun analysointityökalu. Oulun yliopisto, sähkö- ja tietotekniikan osasto. Diplomityö, 100 s.

TIIVISTELMÄ

Tarkoista näytöistä, virtaviivaisista käyttöliittymistä ja näyttävästä grafiikasta on tullut nykyaikaisten mobiililaitteiden perusedellytyksiä. Tämä suuntaus on johtanut siirtymiseen perinteisestä pikselipohjaisesta grafiikasta monipuolisempaan ja tehokkaampaan vektorigrafiikkaan. Graafisen sisällön ja suorituskyvyn suhde ei kuitenkaan ole aina suoraviivainen vektorigrafiikkaa käytettäessä. Tämä on johtanut laadullisesti puutteellisiin sovelluksiin, jotka kärsivät huonosta suorituskyvystä, korkeasta muistinkäytöstä tai liiallisesta tehonkulutuksesta.

Tässä diplomityössä käsitellään näiden laatuongelmien analysointia. Työssä tutkitaan mobiilisovelluksia, jotka hyödyntävät kolmiuloitteista OpenGL ES -grafiikka tai kaksiuloitteista OpenVG-grafiikkaa. Sovelluksissa esiintyviä laatuongelmia pyritään ratkaisemaan luokittelemalla ongelmat niitä aiheuttavan tekijän perusteella. Tämän luokittelupäätöksen tekeminen edellyttää tarkkaa tietoa sovelluksen graafisesta sisällöstä. Tällaisen tiedon hankinta ei kuitenkaan ole käytännöllistä nykyisin saatavissa olevilla työkaluilla.

Työssä esitellään työkaluohjelmisto laatuongelmien luokitteluun vaadittavan tiedon hankintaan. Työkalun keskeinen lähtökohta on tallentaa sovelluksen tekemät graafiset piirtokäskyt tiedostoon myöhempää analysointia varten. Näitä piirtokäskyjä analysoidaan ja käsitellään tarvittavan ymmärryksen saavuttamiseksi tutkittavasta laatuongelmasta.

Työkalun toimintaa esitellään tutkimalla riittämättömän suorituskykyistä sovellusta, visuaalista virhettä sovelluksen grafiikassa, näennäisesti laadukkaan sovelluksen toteutusta, useiden eri grafiikkatoteutusten suorituskykyä sekä yksityiskohtaisia tilastoja sovelluksen graafisesta sisällöstä. Näiden käyttötapausten onnistuneella toteutuksella osoitetaan, että grafiikkakäskyjen tallentaminen ja jatkokäsittely on käytännöllinen menetelmä laatuongelmien analysointiin vektorigrafiikkaa hyödyntävissä mobiilisovelluksissa.

Avainsanat: graafinen vianetsintä, käskyjen tallennus, sisällön ominaisuudet, OpenGL ES, OpenVG

CONTENTS

ABSTRACT

TIIVISTELMÄ

PREFACE

LIST OF SYMBOLS AND ABBREVIATIONS

1. INTRODUCTION 8

2. MOBILE COMPUTER GRAPHICS 10

2.1. Vector and Bitmap Graphics . . . 10
2.1.1. Hardware Acceleration and Standardization . . . 11
2.2. Mobile Application Platform . . . 12
2.3. Challenges in Mobile Vector Graphics . . . 13
2.4. Vector Graphics and the Perception of Quality . . . 15
2.5. Detecting Quality Problems . . . 16
2.6. Classifying Quality Problems . . . 17
2.6.1. Graphics API Usage . . . 17
2.6.2. Graphics Content Complexity . . . 18
2.6.3. Graphics Engine Performance . . . 19
2.6.4. Quality Problems Unrelated to Graphics . . . 19
2.7. The Graphics Quality Analysis Toolkit . . . 19
2.8. Previous Work . . . 20
2.8.1. Graphics Command Tracing . . . 21
2.8.2. Graphics State Tracking . . . 22
2.8.3. Graphical Debugging . . . 23
2.8.4. Graphics Workload Characterization . . . 24

3. VECTOR GRAPHICS APIS 25

3.1. Evolution of 3D Graphics APIs . . . 25
3.2. Origins of 2D Vector Graphics APIs . . . 27
3.3. Graphics Primitives . . . 27
3.4. OpenGL ES 1.x Rendering Pipeline . . . 28
3.5. OpenVG 1.x Rendering Pipeline . . . 30
3.6. Native Windowing System Integration . . . 32
3.7. Performance Factors . . . 33

4. GRAPHICS QUALITY ANALYSIS TOOLKIT REQUIREMENTS 37

4.1. Functional Requirements . . . 37
4.1.1. Tracer . . . 37
4.1.2. Trace Player . . . 38
4.1.3. Trace Analyzer . . . 38
4.2. Non-functional Requirements . . . 39
4.3. Use Cases . . . 39

4.3.1. Unsatisfactory Application Performance 40
4.3.2. Visual Error in Application 41
4.3.3. Application Quality Analysis 42
4.3.4. Graphics Engine Benchmarking 43
4.3.5. Graphics Content Analysis 44

5. GRAPHICS QUALITY ANALYSIS TOOLKIT ARCHITECTURE 46
5.1. Tracer and Trace Player Generator 46
5.1.1. API Configuration 46
5.1.2. Working with Generated Code 48
5.2. Tracer 49
5.2.1. Platform Security 52
5.2.2. Performance Considerations 52
5.2.3. Portability 54
5.2.4. Trace Files 54
5.2.5. State Tracking 55
5.2.6. Tracing OpenGL ES 58
5.2.7. Tracing OpenVG 58
5.2.8. Tracing EGL 60
5.3. Trace Player 63
5.3.1. Trace Normalization 63
5.3.2. Trace Portability 64
5.3.3. Performance Considerations 65
5.4. Trace Analyzer 66
5.4.1. User Interface 66
5.4.2. Trace Manipulation 68
5.4.3. Content Statistics and Graphs 70
5.4.4. State and Frame Extraction 70
5.4.5. Scripting Interface 73
5.4.6. Trace Query Language 74
5.4.7. Exporting Traces as C Code 74

6. USE CASE DEMONSTRATION 78
6.1. Unsatisfactory Application Performance 78
6.2. Visual Error in Application 81
6.3. Application Quality Analysis 83
6.4. Graphics Engine Benchmarking 85
6.5. Graphics Content Analysis 87

7. DISCUSSION 92
7.1. Future Work 93

8. CONCLUSION 95

9. REFERENCES 97

PREFACE

This work was carried out for the Display & Graphics Software group of the Technology Platforms unit of Nokia Corporation, R&D Oulu. A number of illustrative elements used in this thesis were derived from the Oxygen Icon project under the Creative Commons Attribution-Share Alike 3.0 License.

The initial design for the Graphics Quality Analysis Toolkit was created together with Kari J. Kangas, Mika Qvist and other members of the Nokia Display & Graphics Software group. This project builds on earlier work in graphics performance measurement done by Kari J. Kangas and Kari Pulli since 2004.

I would like to thank my supervisors Juha Röning, Dr. Tech., and Kari Pulli, Ph.D., for their valuable input and discussions during the course of this work. I also wish to extend my warm thanks to those who have helped me in completing this work, especially Kari J. Kangas, Mika Qvist, Sila Kayo, Marianne Yrjänä, Jani Vaarala and others from the Display & Graphics Software team.

Oulu, Finland, February 16, 2008

Sami Kyöstilä

LIST OF SYMBOLS AND ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
API	Application programming interface
CPU	Central processing unit
DLL	Dynamic link library
GDI	Graphics device interface
GPU	Graphics processing unit
GPS	Global positioning system
M3G	Java Mobile 3D graphics
OS	Operating system
SVG	Scalable vector graphics
SQL	Structured query language
VGA	Video graphics array

1. INTRODUCTION

The use of vector graphics in smartphones and other mobile devices is increasing rapidly. Mobile applications such as games, map navigation software, and system user interfaces are making the transition from traditional bitmap-based graphics to vector graphics. Free scalability, reduced power consumption through hardware acceleration, flexible animation support and compact representation are among the key advantages of the technology. These benefits make vector graphics an attractive solution with which to meet the demand for increasing display resolutions and fluid user interfaces in modern mobile devices.

The adoption of the technology, however, has not been without challenges. Many mobile vector graphics applications suffer from various quality issues from jerky animation to high processor, memory and power consumption. These problems are caused by factors such as:

- the quick transition to vector graphics technology in embedded devices, which have traditionally only used bitmap graphics
- the complex relationship between vector graphics content and the resulting performance
- developer experience with much more powerful platforms such as PCs, where software inefficiencies can be mitigated with ever-increasing hardware speeds and
- not designing applications with performance in mind from the start, but instead tackling performance issues as they arise

Such quality issues have an immediate impact on device usability and on perceived product quality. Solving them is therefore a way to enhance user experience. Although graphics analysis tools such as debuggers exist, they are generally designed for more powerful devices such as PCs and commonly limited to a single platform or graphics API. The development environment for embedded devices is usually cross-platform and more limited in terms of performance. Furthermore, embedded software is often written at a stage when the target hardware is only available in an immature prototype form, making it difficult to use debugging software built for production hardware.

This M.Sc thesis focuses on the process of analyzing and solving these quality issues; our aim is to provide practical and efficient means to study the quality of mobile vector graphics applications. Our system is based on intercepting graphics commands, i.e., graphics API function calls and associated argument data, to a trace file, which is then analyzed with a dedicated tool in a workstation environment, surpassing the limitations of embedded hardware. Our work focuses on applications employing either OpenGL ES 1.1 or OpenVG 1.0 -based graphics. We mainly examine applications running on Nokia smartphones, which use the Symbian operating system.

This text is written from the perspective of a system level graphics integration team within Nokia. As our work revolves around developing, testing, and integrating graphics technologies for various platforms, the low quality of vector graphics applications

is an issue with which we are very familiar. Although our organizational responsibilities are undoubtedly reflected in the design of the toolkit and the quality analysis process, it should be stressed that the utility of these tools is not limited to a system integration level.

2. MOBILE COMPUTER GRAPHICS

Computer graphics is the practice of using computational constructs to produce and manipulate synthetic visual images. A broad term, computer graphics encompasses everything from image processing to digital animation, video games, graphical user interfaces and more. For the purposes of this thesis, we focus on two major sub-fields of computer graphics: two-dimensional (2D) and three-dimensional (3D) graphics.

In this chapter, we will first introduce the concepts of computer and vector graphics and discuss their applications in mobile devices. We then move on to examine vector graphics quality, its measures and the methods used in analyzing quality issues. Finally, we present the design of the Graphics Quality Analysis Toolkit for enabling practical and efficient analysis of vector graphics quality issues.

2.1. Vector and Bitmap Graphics

The fundamental principle of vector graphics is the use of mathematical descriptions of geometric primitives such as lines, triangles, and curves to produce pictures. This is in contrast with more traditional bitmap graphics, which employ two-dimensional arrays of picture elements or pixels; each pixel represents the color at a single point in the corresponding picture.

The benefits of vector graphics, as opposed to bitmap graphics, include unlimited scaling and transforming, as illustrated in Figure 1, resolution independence and a compact memory-efficient representation. Due to these inherent advantages, vector graphics has been widely adopted in fields such as computer graphics and the printing industry.

In computer graphics, two major variants of vector graphics have emerged: three-dimensional graphics, which are constructed in three-dimensional space, and two-dimensional graphics that reside on a two-dimensional plane. The former is most often used in virtual reality systems and games, while the latter is usually employed in user interfaces, graphic illustrations, and drawing programs.

In order to display vector graphics, a process called rasterization must be performed. In rasterization, vector graphics are converted from their mathematical representation to an array of pixels, that is, an image that can be displayed on the screen.

In terms of algorithmic complexity, drawing traditional bitmap graphics is quite straightforward: the computational and storage cost of bitmap images is usually linearly-dependent on the number of pixels involved. From this relation it quickly becomes apparent why ensuring high graphics performance becomes more difficult with increasing screen resolutions: if the screen resolution doubles, the cost of using bitmap images is effectively multiplied by four. More importantly, the storage space requirements of bitmap images also increase by a factor of four, leading to more memory consumption.

In contrast, the algorithmic complexity of vector graphics rasterization is far from obvious: vector images can consist of arbitrarily simple or detailed shapes, regardless of the resolution of the display device. Similarly the storage space requirements of vector images are independent of the display size and depend mainly on the level of detail in the image.

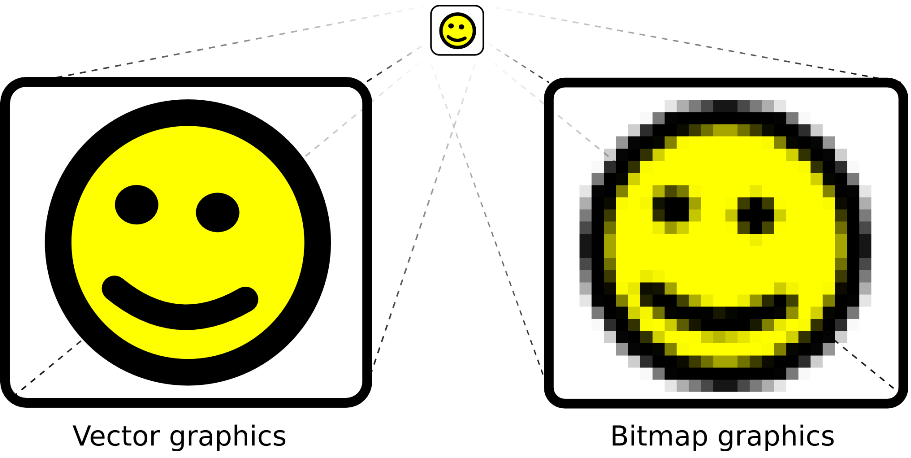

Figure 1. Vector graphics can be scaled freely, while bitmap graphics become pixelated in the same process.

A similar division between vector and bitmap graphics arises in animated graphics. Animation using bitmap images is commonly created by preparing multiple images or frames, each of which show a particular stage of the complete animation. These images are then shown in rapid succession, creating the illusion of continuous motion. Due to the high number of images required for non-trivial animations, the memory cost of this method quickly becomes prohibitive. One solution is to reduce the data set with video compression methods that exploit the redundancies in successive frames. This method has the downside that due to the high computational cost of most of the compression algorithms, the animation content is essentially fixed and cannot be generated on demand as a response to, for instance, user input. Again, vector graphics offers a viable alternative approach for producing animation: the animation frames can be drawn with vector shapes by changing the mathematical properties, such as the size, form or position, of shapes based on the progression of time. With this method, each animation frame is generated on demand by rasterizing the vector graphics scene at that point in time. This method is also much more efficient in terms of storage space, since an animation file only needs to contain the basic properties of the animated shapes instead of their full pixel representation as with bitmap graphics.

2.1.1. Hardware Acceleration and Standardization

The designer of a vector graphics system has two choices: vector graphics may be rasterized using the main application processor of the system, or a specialized hardware accelerator can be built to perform the rasterization task. These graphics processors units, GPUs, can typically perform rasterization many orders of magnitude faster than an implementation using the generic application processor, a so-called "software implementation". Another advantage of hardware accelerators is that they can operate in parallel with the application processor, enabling other processing while vector graphics are being rasterized.

As the process of rasterizing vector graphics images essentially results in a bitmap image, it is apparent that as with all bitmap images, the complexity of this operation is also very much dependent on the number of pixels produced. In other words, the resolution of the display device is an important factor in determining the performance of a vector graphics system. For a software vector graphics implementation, this represents a major stumbling block: even though the vector graphics images themselves are resolution-independent and compactly represented with geometric primitives, the final rasterization step becomes prohibitively expensive as screen sizes increase. A dedicated hardware accelerator, however, is not limited in this fashion: depending on the architecture, it may produce multiple output pixels in parallel, which translates into a much improved rasterization speed. Therefore, as the display resolutions in mobile devices continue to increase, the need for hardware accelerated vector graphics becomes apparent.

The first step in enabling robust hardware acceleration of a technology is to create a standard abstraction layer that provides a clean separation between the application and the actual implementation of the technology. In software development terms, this layer is called the application programming interface or API. In computer graphics terms, the software or piece of hardware that implements the actual API functionality is called the graphics engine. A common configuration of these components is illustrated in Figure 2. The use of a standard API effectively decouples the application from the graphics engine implementation, enabling application portability between different engine implementations.

Thanks to the long heritage of the technology, many APIs have been created for programming vector graphics. Today, the dominant 3D graphics APIs are Microsoft's Direct3D [1] and Khronos Group's OpenGL [2], both of which are widely used in desktop PCs, game consoles, engineering tools, and scientific applications.

The mobile and embedded electronics vendors, however, saw that these APIs had accumulated such a vast array of features during their lifetime that implementing them on a mobile device would not be feasible. Direct3D also had the added constraint of being a closed standard controlled by Microsoft and at the time limited to devices running Windows. The solution chosen by the industry was to take OpenGL and strip out all the non-essential components forming an embedded subset [3 p. 68], OpenGL ES [4]. A more recent addition to the field is OpenVG [5], which is aimed at 2D vector graphics.

To oversee the standardization of these and other new embedded media APIs, a consortium called The Khronos Group was formed. Their mission is to create open standard, royalty-free APIs that enable the authoring and accelerated playback of dynamic media on a wide variety of platforms and devices [6].

2.2. Mobile Application Platform

Symbian is the leading smartphone operating system (OS) with a 72% market share in the first quarter of 2007 [7]. It is used in a number of mobile device models by Nokia, Motorola, Sony Ericsson, Siemens, and others. Symbian provides a comprehensive third-party software development environment open to third-party developers.

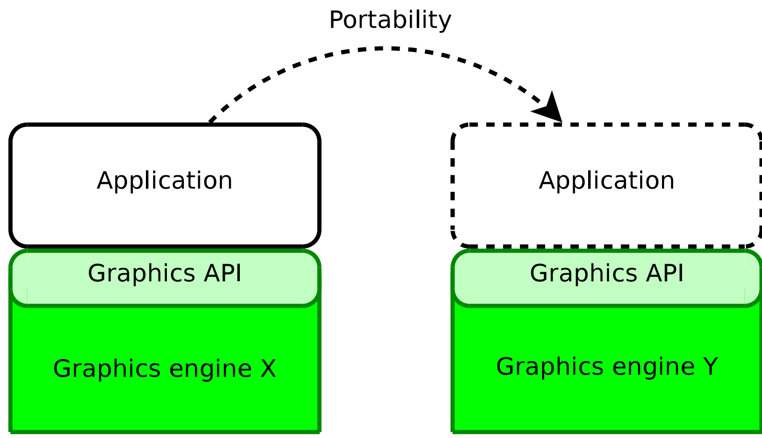

Figure 2. Separating the application from the graphics engine with a standardized API enables application portability from one engine to another.

Symbian OS is based on a real-time nanokernel architecture that provides process scheduling and isolation, memory management, device drivers, file system services, windowing, and all other features one would expect from a fully fledged modern embedded operating system. It has a rich set of APIs that offer, for example, imaging, audio, positioning and internet services. In the context of vector graphics, both OpenGL ES and OpenVG are supported [8] [9].

In addition to Symbian, a major component in Nokia smartphones is an application middleware called the S60 Platform [10]. The S60 Platform provides the device's graphical user interface and the built-in applications. From the developer's perspective, the S60 Platform offers additional services, the most relevant of which is the graphical user interface toolkit Avkon [11 p. 21]. From a vector graphics standpoint, an interesting recent addition to the platform is the support for Scalable UI [12], which decouples applications from the physical screen resolution. Technically this is achieved by drawing the whole graphical interface using scalable vector graphics (SVG) images.

Due to the relatively rapid development of the industry, the range of available Symbian devices on the market is very wide; in the six years following the introduction of Nokia's first Symbian-based smartphone, central processing unit (CPU) speeds have more than tripled to over 330 MHz, display sizes have reached quarter-VGA (video graphics array) resolution (320 by 240 pixels) and memory capacity has surpassed 64 megabytes [13]. A mobile application developer must acknowledge that these numbers only represent the very high end of devices. Taking the whole class of smartphones into account, these performance figures will vary greatly. This presents a considerable challenge for mobile application development: how to ensure that the application will perform adequately on all intended devices.

2.3. Challenges in Mobile Vector Graphics

Vector graphics and vector graphics accelerators have been introduced in mobile devices in a fraction of the time it took for them to appear in commodity workstations.

This rapid development has placed a great deal of pressure on legacy system software and bitmap-based application middleware layers, which have not been designed to benefit from graphics hardware acceleration.

Some aspects of traditional bitmap-based graphics interfaces are particularly poorly suited for hardware acceleration. An example of such a feature is the action of allowing applications to directly modify the pixel contents of graphics images. With graphics accelerators, image data is often stored in dedicated graphics memory, which is inaccessible from the application processor. Allowing applications to modify graphics images directly in such systems results in heavy memory bus traffic, as the contents of the graphics images must be transported back and forth between the dedicated graphics memory and the main memory. Another major stumbling block is the use of synchronous graphics operations, which are performed completely before returning control to the application. For example, if an application tries to acquire a copy of the rendered image, it is forced to wait until the accelerator has finished drawing it. Since high performance vector graphics acceleration often relies on efficient parallelism, synchronous graphics operations can have a major negative performance impact.

In an ideal world, the system software would simply be modified to work better with hardware graphics acceleration. In most cases, this cannot be done due to the large implementation effort and interest in retaining the binary compatibility of the platform, that is, allowing applications written for an older revision of the platform to work on newer versions without modifications. This is less of a problem for proprietary platforms, where the software vendor has full control over all the applications, but more so on open platforms such as Symbian. A common approach is that backward compatibility in new systems is retained through special emulation code, which allows older unmodified applications to work with the latest graphics accelerators. This emulation, however, may come at a performance cost, and it may not always be entirely obvious which graphics functionality is implemented through the emulation. This poses the application developer with the challenge of how to steer clear of these performance pitfalls.

In addition to the system software, the mobile application development tools also pose a set of problems for the programmer. At the time of writing, performance validation tools for Symbian applications are limited to source level debuggers and sampling profilers, and therefore reliable performance data is hard to obtain. Furthermore, since a profiling tool only measures activity on the application processor, comprehensive data for dedicated graphics accelerators is not available.

In graphics system integration work, a common situation is that a poorly performing graphics application is not written by the same person who is assigned to debug the problem. Especially in such cases, trying to analyze the graphics operations of an unknown application solely by looking at its source code in a debugger is not a sustainable way of working. Firstly, there are no guarantees for the quality of the source code, and discerning the internal logic of the application may be time consuming; secondly, the parameters of single function calls do not convey enough information about the effective graphical content of the application; and finally, it is not uncommon that the application source code is simply not conveniently available, for instance, if the application was written by a third party developer outside Nokia.

Experience is also a major issue; a developer with a PC background is unlikely to be able to appreciate the special limitations of a mobile device. While the available

set of features will make most desktop developers feel at home, Symbian devices often have substantial performance limitations that should be factored into application design from the beginning. In the scope of vector graphics, the most limiting factors are memory and bus bandwidth, the lack of a dedicated graphics processing unit and the fact that at the time of writing, floating point calculations are commonly performed with slow software emulation [3].

The vector graphics APIs are also somewhat problematic in terms of performance. One of the virtues of an API is to hide implementation details from the application developer. Conversely, the performance response to different graphics content of implementation is often also concealed. With APIs such as OpenGL ES and OpenVG, this causes a disconnection from the graphics content to the final performance seen by the developer. This makes performance estimation of unknown graphics hardware very difficult without extensive benchmarking work.

2.4. Vector Graphics and the Perception of Quality

Quality is a term generally associated with the superiority, or at least non-inferiority, or the usefulness of something [14 p. 311]. For the purposes of this thesis, we can define a number of characteristics of vector graphics and user interaction that contribute to the quality of an application:

- **Fidelity**—the graphics should be free of visually distracting artifacts and other flaws.
- **Fluidness**—animated graphics should be performed at a sufficient rate to produce the illusion of continuous motion.
- **Responsiveness**—the application should respond to user interaction with a tolerable delay.
- **Usability**—the graphics should not hinder usage of the application.
- **Low resource utilization**—the application should not consume excessive resources, such as processing capability or memory space while producing the graphics.

This enumeration is by no means exhaustive, nor can the listed items be said to be fully independent; the intent here is to convey a general idea of the type of factors we consider when establishing the quality of a vector graphics application.

Vector graphics are mainly used in mobile devices to construct a graphical user interface for the person using the device. Since the interface is, by definition, very visible for the user, any major quality deficiencies in it will have an effect on his or her perception of the overall device quality. This kind of direct exposure emphasizes the importance of quality vector graphics in mobile devices.

While jitter and synchronization skew in non-interactive media has been shown to be tolerable to a certain degree [15] and the information content of video material is understood even at a low frame rate [16], latency and lag in a graphical user interfaces has a direct impact on the system response time and ultimately on the ease of use

of the system [17]. Therefore, minimizing graphics latency and jitter improves user experience.

Traditionally, mobile graphical user interfaces have been mostly static displays of information due to limited processing power and display hardware constraints. In such systems, the display was usually updated quite rarely as a direct response to user actions, such as menu navigation. While the latency of continuously animated graphics was not a major quality factor in such a system, the input response latency of the system has and always will be of paramount importance to a good perception of quality. Additionally, research has shown that animation helps the user gain a more thorough understanding of presented data, especially if the data is spatial in nature, such as a hierarchy of menu options [18]. In this light, it is not surprising that that mobile user interfaces are quickly moving from the traditional static constructs to animated transition effects [19] and other dynamic elements. The general trend is that non-animated interfaces are seen as old fashioned and boring, while flashy animated interfaces gather much more attention [20]. It is therefore important that a mobile device has proper support for animated graphics from both the usability and marketing perspective.

2.5. Detecting Quality Problems

Due to the visible role of graphics in user interfaces, quality problems can generally be detected simply by using the device in question; common symptoms such as sluggish responsiveness, long waiting times, poor battery life and graphical glitches are hard to miss. The difficulty lies in turning these subjective measures into concrete metrics and using them to systematically identify and solve vector graphics quality problems. An equally important part of the issue is determining whether the problem has in fact anything to do with graphics.

For animated graphics, a universally understood measure of quality is the frame rate, which indicates how many times the device display is updated per second. A commonly considered lower bound for real time animation is 10 frame updates per second [21]. When the animation is controlled or triggered by the user, a similar measure of system latency can be used. It indicates the time from user input to a visible result on the screen. In addition to graphics latency, system latency also includes all other processing done in the system. These two types of time measures are illustrated in Figure 3.

In the context of mobile application platforms, we must also consider the power consumption of graphical processing. Power consumption, measured in amperes, indicates how much current is drawn from the device battery at a given time. Since battery charge is a finite resource, conserving it should be a top priority. As graphics are often drawn several times per second, minimizing their power usage is worthwhile.

In the following sections, we will examine what aspects of a vector graphics system have an impact on these measures and how those aspects can be studied.

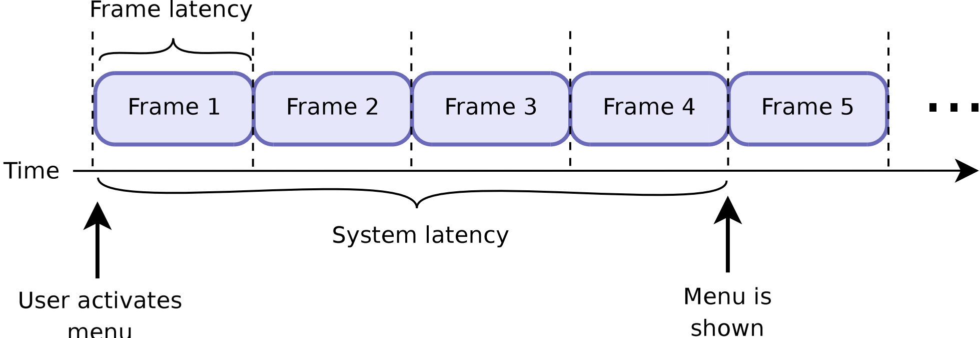

Figure 3. Frame latency is defined by the graphics display update rate, while system latency is the time taken by the system to respond to user input.

2.6. Classifying Quality Problems

Once a quality problem is detected, the next step is to investigate what factors contribute to it. Quality issues are usually brought on by a combination of various problems throughout the system, but often one cause can be seen as the dominating one. After the most significant quality issue has been identified, it can be used to plan further optimization work. Therefore it is advantageous to make this classification process as efficient as possible.

In this work, we employ a quality problem classification model based on the dominating contributing factor:

1. Dominant cause in API usage patterns
2. Dominant cause in graphics content complexity
3. Dominant cause in graphics engine
4. Dominant cause unrelated to graphics

Next we examine each problem class in detail and discuss what kind of information is needed to determine whether a problem belongs in that particular class. It should be noted that we do not strive for a fully automated classification system, but rather to provide a toolset for making the process practical for a graphics engineer.

2.6.1. Graphics API Usage

The way an application uses a graphics API has great consequences in terms of quality. Quality problems in this area often stem from the fact that in both OpenGL ES and OpenVG, and in vector graphics APIs in general, there are many ways to accomplish a given goal. For instance, drawing a forest scene could be done by drawing each tree separately or by drawing the whole forest with a single draw call; the graphical end result is the same for both methods, but the performance is most likely superior in the second case. Another example is the usage of certain synchronous API functions,

which may have significant performance costs on hardware graphics accelerators while having a much lower overhead on software renderers.

Analyzing the API usage pattern of an application begins by looking at the sequence of API function calls executed by the application. The traditional way of doing this is by manually reviewing the source code of the application. Experience with embedded graphics software integration has shown that this is an ineffective way of working. In general, it is anything but straightforward to discern reliably how an application will behave merely by looking at its source code. Furthermore, in some cases the source code may not even be conveniently available. While an interactive debugger may be used work around this limitation, it is an impractical way of working if the number of API calls is large. Therefore, a more effective way to obtain the API call sequence is needed. As embedded devices are unsuitable for extensive analysis work due to processing power and user interface constraints, using a more powerful workstation for analyzing the API call sequence should also be possible.

Graphics API usage analysis is also needed to determine the cause of visual rendering errors. The error could be caused by erroneous API usage or a fault in the graphics engine. In either case, the API call trace is needed in order to isolate the error.

Obtaining the API call sequence is only the first step, however. The resulting sequence will also need to be analyzed and processed to identify any possible quality problems. As a common graphics application can execute thousands of graphics commands per second, manual analysis of the often enormous call trace is not always feasible. Practical tools for examining API call sequences are therefore also necessary.

2.6.2. Graphics Content Complexity

The graphics content of an application comprises the graphical primitives drawn during its operation. The graphical primitives in this context can be elementary geometric shapes such as points and triangles or aggregations of multiple shapes, such as a complete three-dimensional object.

The key thing to consider when evaluating graphics content quality is whether or not it presents an appropriate workload for the graphics engine. A common mistake is to model graphical objects with too many details without regard to how large they appear on the device screen. Another often-made oversight is to first draw a complex object, only to later cover it completely with another object.

Estimating the workload produced by graphics content can be based on characterizing various empirically measurable aspects of the content. A canonical example of such a metric is the number of triangles used to build the rendered graphical objects. Once the metrics have been calculated, they can be used to judge how well the content matches the capabilities of the underlying platform. Additionally, the same content features can be used to develop synthetic benchmarks with matching content features. The advantage of synthetic benchmarks in comparison to using regular applications for benchmarking is that their content features can be more easily parameterized to obtain content variants with extrapolated complexity.

Calculating detailed content statistics is traditionally done with the help of a specially instrumented graphics engine. As of today, there is no such comprehensive tool

for obtaining such measurements from mobile graphics applications. For the purposes of graphics quality analysis, one is needed.

2.6.3. Graphics Engine Performance

In order to make meaningful judgments about graphics content complexity, one also needs to establish the performance profile of the graphics engine that will do the actual rendering work. This is often done by running special benchmarks programs that draw some synthetic graphics and measure the performance of the graphics engine. The term synthetic is used here to differentiate the graphics content of a real application versus the graphics content created specifically for benchmarking purposes. The problem with this approach is that the content characteristics of synthetic benchmarks may not necessarily match those of real-world applications. That is, the benchmarks could essentially be measuring the wrong thing. These synthetic benchmarks can be made better by basing their design on measured content features as explained in the previous section.

Actual graphics applications can also be used for benchmarking purposes, but that will usually require programming special measurement routines into each used application. Furthermore, the tested graphics engine may only work on prototype hardware that is incapable of running regular applications. Due to these limitations, using real application content for benchmarking mobile graphics engines is usually impractical.

Accurate and reliable engine performance profiling benefits from the use of both real application graphics content and content feature-based synthetic benchmarks.

2.6.4. Quality Problems Unrelated to Graphics

The final class of quality problems contains the problems that fall outside the scope of vector graphics. If the graphics API usage patterns and content analysis indicate no adverse issues, a quality problem can be considered to be caused by some other component in the system or by the application itself. An example of such a problem is an application that is performing so many internal calculations, that it is unable to submit graphics commands to the graphics engine at a high enough rate. Problems that fall under this class should be dealt with traditional software development tools such as debuggers and profilers.

2.7. The Graphics Quality Analysis Toolkit

From the previous discussion, it is apparent that there is great room for improvement in the process of vector graphics quality analysis. For this reason, we set forth to build a special set of tools for assisting the solving of vector graphics application quality issues and estimating graphics engine performance. The main objective of the toolkit is to make it as easy as possible to examine the graphics produced by a mobile vector graphics application in great detail. The toolkit also offers support for benchmarking graphics engines using real and synthetic graphics content. While toolkit is mainly tar-

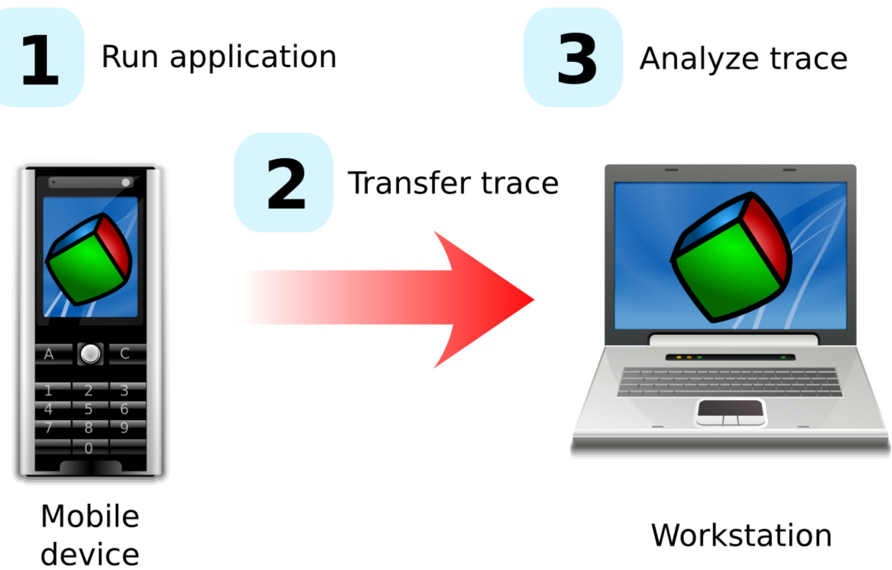

Figure 4. The general workflow for the Graphics Quality Analysis Toolkit is to trace a vector graphics application on a mobile device and to analyze the captured trace in a workstation environment.

geted at the OpenGL ES and OpenVG APIs and applications running on the Symbian platform, it is also designed to be portable to other APIs and platforms. An important requirement is offline analysis, since extensive development work on a mobile terminal is impractical due to processing constraints and user interface limitations. The desired general workflow of the toolkit is illustrated in Figure 4.

Based on these high level requirements, The Graphics Quality Analysis Toolkit is designed to comprise the following components:

1. A **Tracer** for capturing all OpenGL ES and OpenVG graphics commands executed by a Symbian application. The commands will be saved to a trace file, which can be copied to a workstation for further analysis.

2. A **Trace Player** for repeating the captured graphics commands. The player can be run on a variety of platforms with any compatible graphics engine.

3. A **Trace Analyzer** for examining and manipulating the graphics command trace files. The analyzer is used on a Windows workstation.

4. **Instrumented OpenGL ES and OpenVG graphics engines** for extracting detailed content statistics from trace files.

The overall configuration and interaction of these components is shown in Figure 5.

2.8. Previous Work

The following sections present a survey of graphics research work that is related to the domain of vector graphics quality analysis.

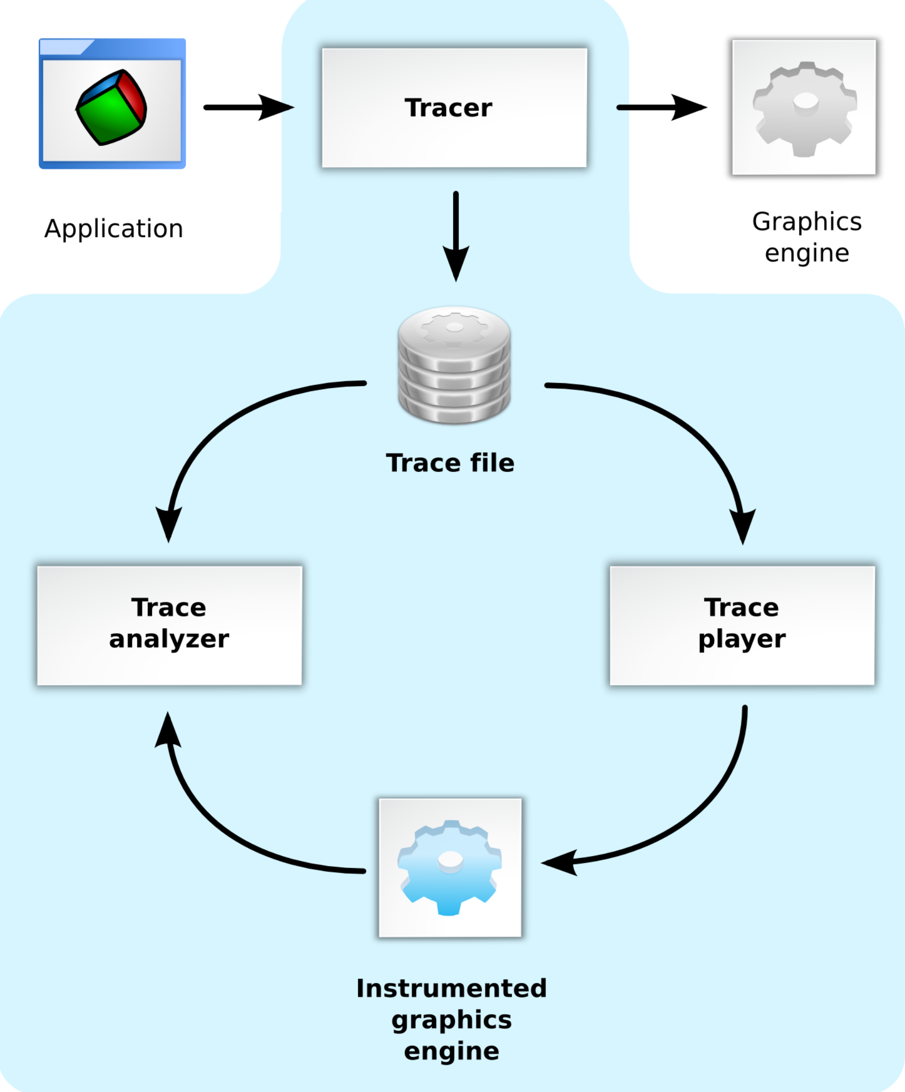

Figure 5. The main components of the Graphics Quality Analysis Toolkit are the **Tracer**, which captures graphics commands to a **trace file**; the **Trace Player**, which replays captured graphics commands and the **Trace Analyzer**, which uses an **instrumented graphics engine** to extract information from a trace file.

2.8.1. Graphics Command Tracing

The method of unintrusively tracing graphics calls from programs was formally introduced by Dunwoody and Linton [22]. In their design, the OpenGL commands of a graphics application are captured in a binary trace file in a graphics API neutral language for later analysis. Their tracer masquerades as the system graphics engine by providing an identical binary interface to the application. The inherent advantage of not requiring any instrumentation of the application code or the graphics engine was an important reason for choosing the same method of graphics call tracing as the fundamental basis of our work. Our approach, however, differs in that we keep the graphics command abstraction level at the level of individual API calls instead of specifying

a higher level intermediate language. This ensures that we could capture the exact behavior of the traced application as closely as possible.

The same basic technique of tracing has also been utilized by Humphreys et al., in a system called Chromium [23]. Chromium is an extensible OpenGL command stream processing library, which provides a virtual OpenGL graphics engine implementation. The captured OpenGL API calls are routed through a network of SPUs or Stream Processing Units. Each SPU can perform arbitrary processing of the command stream. Example uses include diverting OpenGL graphics rendering to a cluster of computers and compositing the resulting image on a local display. Chromium has also been used in performance estimation [24], where an OpenGL trace is used to drive a simulated graphics processing unit model.

Chromium has a very similar architecture to the tracer in our system, and thus could have served as a basis for our implementation. Instead, we chose to implement a custom tracer through code generation. This enables us to support nearly any C-based API, such as OpenGL ES and OpenVG, while Chromium would have directly offered support only for OpenGL. Additionally, our platform portability and performance requirements on embedded systems also necessitate a custom, more focused solution.

The OpenGL API defines a built-in mechanism for tracing and playing back graphics command sequences or display lists [25]. Applications can request that several commands are compiled into a display list, which can be later played back with a single command. Display lists can be used to improve performance, as they enable applications to redraw commonly used graphical objects with minimal overhead. As display lists are immutable, the graphics driver can preprocess their contents to improve efficiency. From the application's point of view, however, display lists are opaque and no information about the contained commands can be retrieved or stored outside the application. Our approach differs in that we make the contents of trace files explicitly available for detailed analysis and processing.

2.8.2. Graphics State Tracking

OpenGL ES, OpenVG and other similar graphics APIs are essentially state machines, the state being the active rendering settings enabled in the graphics engine. This graphics state has a very direct relation to graphics performance, since different rendering settings produce different workloads for the graphics engine. From a vector graphics quality analysis point of view it is therefore imperative to know the graphics state used to draw primitives.

Graphics state tracking is commonly used in distributed or remote graphics rendering, where it is beneficial to minimize the number of rendering commands sent over the network. The graphics libraries themselves also often employ some form of state tracking in order to avoid needless expensive round trips to the graphics hardware. State tracking is also needed for sharing the same GPU between multiple applications, where the graphics state of an application needs to be saved and later restored to allow other applications to use the GPU in between.

A state tracking application by Buck, Humphreys, and Hanrahan [26] employs a hierarchical state tracking model that can be used to split the rendering work of one

application to multiple rendering nodes. The Chromium system also offers a state tracker for satisfying application state queries locally in addition to other uses.

State tracking also plays a very central role in our work. It enables many of the key use cases of the Tracer and the Trace Analyzer, such as extracting frame sequences from longer traces and pinpointing quality problems in traces. Since a requirement for the toolkit is to be graphics API-neutral, the same principle also applies to the state tracker. This lead us to develop a generic tree data structure suitable for describing the state of EGL, OpenGL ES, OpenVG, and other similar APIs. This method greatly reduces the amount of work for adding new graphics APIs to our system in comparison to hand-written state tracking code as used by Buck et al., Chromium and others. Our method also ensures that the traced graphics calls are not modified in any substantial way in comparison to the original application. This is important, since our system is designed for reproducible application debugging and any unintended modification of the graphics command stream might change the behavior of the graphics engine substantially.

2.8.3. Graphical Debugging

NVIDIA PerfHUD [27] is comprehensive interactive Direct3D analysis tool by NVIDIA. It allows the user to suspend a running graphics application and examine detailed statistics from different units of the graphics pipeline. Additionally, the user can modify the rendering settings, shader source code, textures and other attributes of the inspected application in real-time. NVIDIA PerfHUD is available for Microsoft Windows and works only with NVIDIA graphics hardware.

GDEBugger [28], a product by Graphics Remedy, is an interactive graphical debugger for examining the execution of OpenGL and OpenGL ES programs. It provides a very broad range of statistics on the running program, such as the executed graphics calls, graphics state variables, performance graphs and counters and the relative load of the different graphics pipeline stages. It is a developer friendly tool indented for quick pinpointing of errors and performance issues in graphical applications. GDEBugger runs on the Microsoft Windows and Linux operating systems.

GLIntercept [29] is also a similar, albeit less sophisticated tool. Rather than a full-fledged debugger, it is more of a graphical event logger. It can save all the OpenGL commands made by an application to a file, along with extra data such as framebuffer snapshots and texture images. It also has a mechanism for introducing interactive analysis features, such as free camera movement around the 3D graphics drawn by the application.

A relational graphics debugger introduced by Duca et al. [30] focuses more on the data extraction stage of the debugging process. Their system is built on a powerful structured query language (SQL) -based language that can be used to extract data from all stages of the graphics pipeline. While the debugger can answer simple queries directly based on submitted graphics commands, more complex queries are answered by replaying a portion of the graphics commands through automatically instrumented shader programs.

These and other commonly used graphical debugging solutions are based on live interaction with the debugged application. Our approach is to instead focus completely

on offline trace analysis. In the context of embedded applications, offline analysis is essential since the target device often lacks the necessary processing power and usability for interactive graphical debugging. Working with trace files rather than live applications also has the added benefit of providing completely repeatable graphics sequences; in effect, it separates the examined quality issue from the application. Offline analysis also facilitates remote debugging, in which the execution environment and the application can be physically separate from the analysis environment. Our system also adds the possibility of transforming the trace file into different formats, such as plain text, portable C code and numerical data for statistical analysis.

2.8.4. Graphics Workload Characterization

Graphics workload characterization is the process of deriving concrete numerical feature points from a sequence of graphics operations. The intention is that these statistics can be used to gauge the performance characteristics and the overall complexity of the examined graphics sequence.

Workload characterization is a useful technique for hardware vendors, since they can use data acquired through it to evaluate new graphics hardware designs. In effect, the vendors can estimate how a given hardware implementation would perform when given content similar to that which is used for workload characterization.

Workload statistics are useful also beyond hardware design; in an application by Wimmer and Wonka [31], a rendering time equation is derived from various content measures and used to estimate the rendering latency of a graphics scene before it is drawn. They demonstrate the use of the equation for maintaining a minimum frame rate in interactive graphics applications by scaling the level of graphics detail.

The common problem in all workload characterization is deciding what to measure. This is difficult because the actual workload of particular graphics content depends heavily on the underlying graphics renderer architecture. Chiueh and Lin [32] introduced a set of 3D workload characterization statistics based on an analysis of the 3D graphics pipeline. To obtain the measurements, they used a manually-instrumented version of the open-source Mesa OpenGL graphics library. Some of the measurements they used were the percentage of culled triangles, the average triangle span width and the depth complexity or overdraw amount of the scene. This idea was extended by Mitra et al. [33] by also taking into account low level hardware characteristics such as texture memory traffic and chunked rasterization.

The previous research into graphics workload estimation and characterization is used as a basis for the trace file content features that can be calculated with our system. Our objective was to build a system that provides the features most commonly used for workload estimation. While previous research only covers three-dimensional graphics, our toolkit also provides content features for two-dimensional vector graphics.

3. VECTOR GRAPHICS APIS

An application programming interface or an API is a programming abstraction that enables clean and systematic code reuse. An API is essentially a contract between a library of code and a program that uses the code. The API specifies all messages and data flowing from the program to the library and vice versa. The power of the API concept comes from the fact that the implementation of the library is essentially undefined from the application's point of view, as long as it abides by the interface specified by the API.

An API is usually targeted at a specific programming language. The C language is a popular choice, since it is supported nearly universally and C-based libraries are usable in many other higher-level languages through wrappers and other extension mechanisms.

The purpose of a vector graphics API is to allow programs to draw images using vector graphics. As shown in Figure 2 on page 13, a chief virtue of a such an API is to insulate the application from the physical graphics hardware. With a standardized API in place, applications can be moved from one graphics hardware to another with little or no changes. The lack of such an API leads to each vendor defining proprietary ways of controlling their graphics hardware, which has the effect of coupling applications tightly with a particular piece of hardware.

Most vector graphics APIs also strive to hide the functional differences between different graphics hardware by mandating a common feature set that must be present. If a particular piece of hardware does not directly support a particular feature of the API, it is usually required that the missing feature should be emulated via software. This makes it easier to write portable applications, but almost impossible to know if a particular graphics operation sequence triggers slow software emulation in some platforms.

In this chapter, we will look at the present state and evolution of a number of two- and three-dimensional vector graphics application programming interfaces. As the main subjects of this thesis, OpenGL ES and OpenVG are then presented in more detail. In addition, a binding API called EGL is introduced. The chapter closes with a discussion on the performance aspects of the presented APIs.

3.1. Evolution of 3D Graphics APIs

In the earlier days of vector graphics, a standard vector graphics API was nowhere to be found, and unsurprisingly, the need for common APIs soon surfaced. By the late 1980s and early 90s, a hardware independent API called PHIGS was gaining momentum. The name stood for "programmers' hierarchical interactive graphics system". PHIGS was a relatively high level API, where graphics were produced from hierarchical data structures of 3D objects [34]. A company called SGI, however, saw PHIGS as a threat to its line of graphics workstations. At the time, SGI's machines were using a proprietary graphics API called IRIS GL, which was an abbreviation for "integrated raster imaging system graphics library". To ensure its relevance in the market, SGI started licensing an open version of IRIS GL to its competitors. This new graphics library, introduced in 1992, was called OpenGL [35].

In comparison to PHIGS, OpenGL's approach to graphics was different in a fundamental way. Vector graphics APIs can be roughly categorized into two distinct groups based on the way the graphics are constructed. PHIGS is an example of a retained mode API, where the complete description of the graphics scene is first given and then drawn with one command. Retained mode graphics APIs deal with high level constructs such as objects and their hierarchies. A retained mode graphics library is usually built around the concept of a scene graph, which describes the drawn graphical objects and their relations. The original OpenGL, on the other hand, was an immediate mode graphics API. It differed in the sense that the available graphics commands were very fine grained; a triangle, for instance, was drawn by specifying the coordinates, colors and other attributes of its vertices, each with its own API call. This provides applications with a very high level of control over the produced graphics, although many more steps are required to draw them.

OpenGL's low level approach enabled applications to attain better performance by optimizing the graphics commands sent to the graphics library. It also enabled the drawing of graphics that could not conveniently be described using a hierarchical scene graph. These advantages, combined with the APIs relative simplicity, lead to OpenGL's quick dominance of the 3D graphics APIs. [36 p. 138]

Microsoft, one of the early adopters of OpenGL, opted instead to create their own proprietary, even lower level vector graphics API called Direct3D. Originally, Direct3D featured both a retained mode and an immediate mode, but the former was soon abandoned due to lack of developer interest. The initial immediate mode of Direct3D was similar to OpenGL, but at a lower level and with a much more restricted feature set; the first version of Direct3D, released in 1996, had developers essentially filling command queues for graphics chips. This cumbersome interface was later upgraded with higher level convenience functions. [37]

Direct3D has since become the dominating graphics API for the computer games industry. Its wide adoption can be explained in part with the focus on high performance from the start and the fact that the reduced feature set made it easier for hardware vendors to produce Direct3D-compatible video cards. However, as video games graphics have become increasingly demanding, the feature sets of both Direct3D and OpenGL have evolved to quite an equal standard [38, 39].

More recent developments in graphics hardware have brought the deprecation of the traditional fixed function pipeline in favor of programmable shader units. Modern versions of both OpenGL and Direct3D allow the user to execute special programs on the graphics processor to dynamically control the shape, material, and composition of objects [38, 39]. As neither OpenGL ES 1.1 nor OpenVG support programmable shaders, they are not discussed here in any more detail.

Both OpenGL and Direct3D have seen embedded variants, namely OpenGL ES and Direct3D Mobile [40]. In 2004, Symbian became the first embedded platform to offer an OpenGL ES 1.0 graphics engine accessible to third party developers with the introduction of the Nokia 6630 smartphone [41].

3.2. Origins of 2D Vector Graphics APIs

A dominant force in the development of 2D vector graphics is the printing press, where the high resolution of the medium is a strong justification for using a vector-based representation. The industry standard for printing 2D vector images is Adobe's PostScript. Introduced in 1984, PostScript is not merely an API, but rather an executable, Turing-complete language with well-integrated graphics support. [42]

Drawing in PostScript is based on the concept of paths. A path is a collection of line segments and curves. To produce graphics, the area inside a path is filled or the outline of a path is traced or stroked. This same concept is also the basis for most other 2D vector graphics APIs.

As computer display resolutions and processing capabilities have increased, 2D vector graphics also started seeing interactive applications, ultimately leading to the development of standard 2D graphics APIs. One of the earliest of such APIs was an evolution of PostScript called Display PostScript. It extends the PostScript model with better support for the relative low resolution of computer displays and includes optimizations for better performance. [42]

As interpreted domain languages, PostScript and its derivatives are difficult to integrate to a C-based application. Outside dedicated printing equipment, their robust hardware acceleration is not common. To overcome these issues, a number of more traditional declarative APIs have also been developed for producing 2D vector graphics. Graphics user interface systems such as Windows GDI [43] (graphics device interface), Xlib [44] and Symbian GDI [45 p.487–509] commonly implement a limited subset of the PostScript imaging model, although until recently they have been more geared toward pixel-oriented graphics for performance reasons. Common limitations in such toolkits are the lack of free graphics transformation and resolution dependence. However, as modern applications are more demanding in terms of graphics operations, UI toolkits are quickly transitioning to using more robust vector graphics techniques [46, 47].

Of the more recent APIs, Khronos Group's OpenVG is the most notable one from the perspective of this thesis. OpenVG is a low level, OpenGL-style C API, which is based on the same concept of filling and stroking paths as PostScript. It was introduced in 2005 and designed with embedded systems and hardware acceleration in mind. One of its main selling points is the support for popular media formats such as Adobe's Flash and W3C's SVG. [48]

3.3. Graphics Primitives

A key concept in any graphics API is the fundamental graphics primitive that is used to construct everything seen on the screen. Some basic geometric primitives for 3D graphics are points, lines, triangles, and polygons. Of these, triangles are most commonly used, because they can be used to approximate almost any kind of natural geometry by being grouped into meshes. Triangles also have the important property of always lying in a plane. This makes rasterizing them into pixels a straightforward operation and easily accelerated in hardware. In contrast, polygons can be shaped arbi-

trarily, even in ways that cause them to self-intersect. This is why support for polygons was not included in OpenGL ES when compared to OpenGL. [36 p. 61]

The 3D objects constructed from geometric primitives can be made to look more realistic by simulating different materials and lighting effects. A common technique is texture mapping, where a two-dimensional image is stretched over a 3D object. OpenGL ES provides support for modeling the effects of these and some additional physical phenomena. [49]

Primitives for two-dimensional rendering include points, lines, circles, rectangles and curves. A special type of curve called the Bézier curve is typically used to construct round shapes that will appear smooth regardless of their scale. OpenVG supports all of these primitives.

Since 2D vector graphics are not primarily about simulating the appearance of real objects, OpenVG does not directly try to simulate different lighting conditions or materials. Instead, it applies the concept of a paint, which can be used to color a drawn shape with a single color, a color gradient or an arbitrary image pattern. [48]

The process of composing images with these fundamental graphics primitives and operations in both OpenGL ES and OpenVG is illustrated in Figure 6.

In OpenGL ES, a number of 3D coordinates called vertices are first connected with triangles to produce meshes. In Figure 6, vertices are drawn as filled circles. These meshes are then drawn as seen from a specified camera position with optional lighting, texturing, and other effects.

In OpenVG, a path is first defined using concatenated line segments or curves. The boxes shown in Figure 6 are the control points for the path. The path is then stroked and filled using specified paint styles. OpenVG also allows the use of image filters, such as blurring, to post-process the rendered image.

In vector graphics APIs, the drawing commands and other data submitted by the application flow through a network of components called the graphics pipeline. The term pipeline is used, because these constructs are usually very serial in nature and most of the data flows through the same path. Although the components or stages of a pipeline are arranged serially, the stages themselves are very parallel in nature and can often work on different parts of the submitted data simultaneously.

In the following two sections, we will examine the pipeline structures of both OpenGL ES and OpenVG in more detail.

3.4. OpenGL ES 1.x Rendering Pipeline

The high level structure of the OpenGL ES 1.x rendering pipeline is illustrated in Figure 7 [49 p. 11]. The main components are:

1. **Per-Vertex Operations and Primitive Assembly:** The rendering operation begins here with the application submitting a number of vertices and the set of graphics primitives to be assembled using these vertices. The OpenGL ES library then transforms these vertices, based on where the virtual camera is placed and calculates which produced primitives are visible. The effects of lighting and fog are also calculated for each vertex here. [49 p. 11]

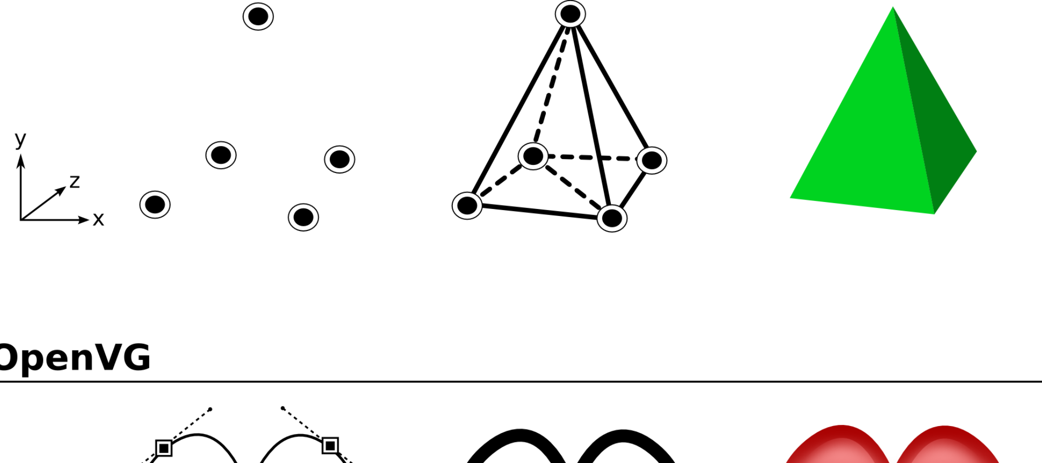

Figure 6. OpenGL ES and OpenVG images are composed of respective fundamental graphics primitives: the triangle and the path.

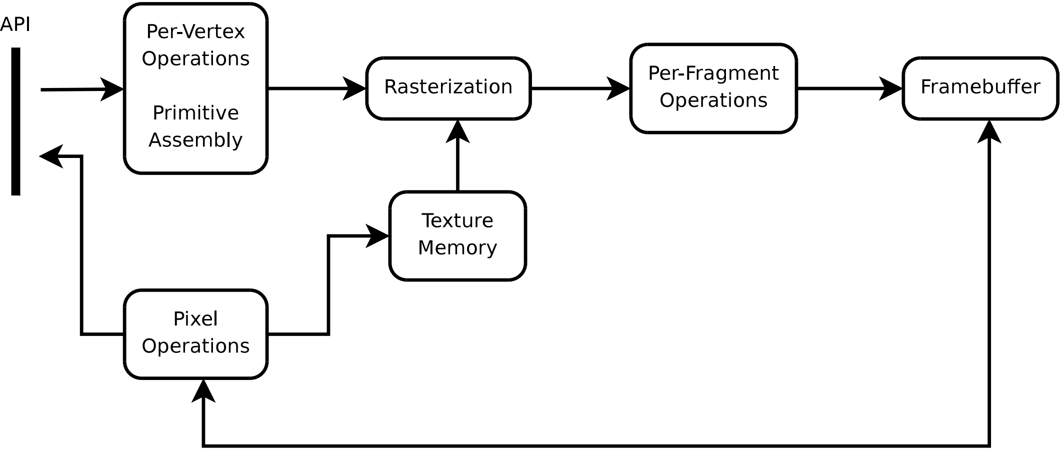

Figure 7. The stages of the OpenGL ES rendering pipeline.

2. **Rasterization:** This stage converts the visible primitives into a number of framebuffer elements called fragments. Each fragment corresponds with a single pixel in the framebuffer. The collection of fragments for a primitive is essentially the image of the primitive on the framebuffer. The color of a fragment can be defined through the vertices making up the primitive or texture assigned for the primitive. A technique called antialiasing can be used to smooth the edges of rasterized primitives by taking into account that a particular pixel may be only partially covered by the primitive. [49 p. 11]

3. **Per-Fragment Operations:** Finally, the produced fragments are written to the framebuffer to produce a visible image. Each fragment is not necessarily written as is; semitransparent fragments can be blended or mixed with fragments that are already in the framebuffer. A depth test can also be used to determine that an existing fragment in the framebuffer is closer to the viewer, leading to the new fragment being discarded. The visibility of individual fragments can also be controlled based on alpha color values with the alpha test, rectangular regions with the scissor test, and arbitrarily shaped areas with the stencil test. [49 p. 99]

4. **Framebuffer:** The framebuffer is a two-dimensional pixel store that will contain the final rendered image. In addition to visible color information, the framebuffer can store additional attributes for each pixel, such as a transparency factor or alpha component, the depth or distance from the viewer, or a stencil value that can be used to limit drawing to a particular area. [49 p. 99]

5. **Texture Memory:** Texture memory is used to store two-dimensional images called textures, which can be applied to the surface of drawn objects. Texture memory is highlighted as a distinct component, because in hardware accelerators it is often implemented with dedicated memory embedded in the accelerator itself. A higher rasterizing performance can be attained using this approach instead of accessing texture data in the main system memory. Software engines can also benefit from this design, since the data in the texture memory can be specially formatted for optimal read performance. Using dedicated texture memory requires a separate texture-uploading step, in which texture data is transferred from application memory to the texture memory. [49 p. 72]

6. **Pixel Operations:** In some cases, more direct access to the framebuffer pixels is needed. One example involves clearing the whole framebuffer to a particular color before starting to draw new primitives. Some applications might also need to read back a portion of the framebuffer for further processing. This pipeline component enables such pixel-level operations. [49 p. 109]

3.5. OpenVG 1.x Rendering Pipeline

The overall structure of the OpenVG 1.x rendering pipeline is illustrated in Figure 8 [48 p. 4]. The different stages of the pipeline are:

1. **Path, Transformation, Stroke, and Paint Setup:** To begin drawing, an application constructs one or more paths using OpenVG commands. Each path

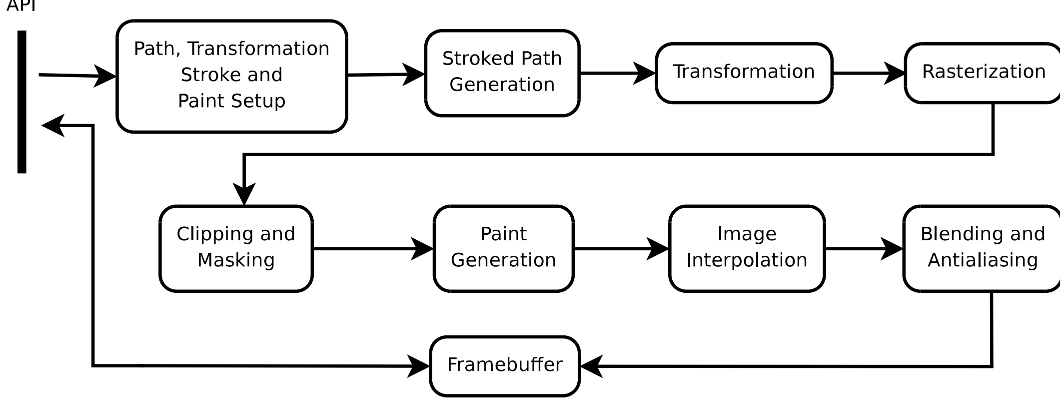

Figure 8. The stages of the OpenVG rendering pipeline.

consists of one or more segments. These paths may be oriented by setting a transformation matrix, which specifies how the viewer is positioned above the drawing plane. The stroke and paint attributes can be set to control whether the drawn shape is filled, or only the outline of the shape is drawn, or both. Stroking also controls the thickness of the shape's outline, while painting defines the color or pattern used to draw the shape. [48 p. 5]

2. **Stroked Path Generation:** If the application specified that the drawn shape is to be stroked, this stage generates a new virtual path that specifies the area inside the shape's outline. This way, the rest of the pipeline only needs to deal with filling the area inside a path. [48 p. 5]

3. **Transformation:** This stage moves and transforms the path to correspond with the transformation matrix set by the application. [48 p. 5]

4. **Rasterization:** Similar to the rasterization step in OpenGL ES, this stage calculates which framebuffer pixels are inside the drawn shape. OpenVG also has provisions to calculate partial pixel coverage values to perform high-quality antialiasing. [48 p. 5]

5. **Clipping and Masking:** The drawn shape can be restricted to a particular area of the framebuffer using a mask or a rectangular clipping region. This stage removes the pixels that do not coincide with the mask or the clipping region. [48 p. 6]

6. **Paint Generation:** This stage calculates the color of each of the shape's pixels, based on what type of paint was used. The different types of paints are solid colors, linear gradients, radial gradients and image patterns. This step is analogous to the texturing and lighting functionality of the OpenGL ES. [48 p. 6]

7. **Image Interpolation:** If the drawn shape was an image, this stage performs filtering to minimize the blockiness of the rasterized image. [48 p. 6]

8. **Blending and Antialiasing:** Finally, the produced pixels are mixed together with those already in the framebuffer. Blending and antialiasing can be used

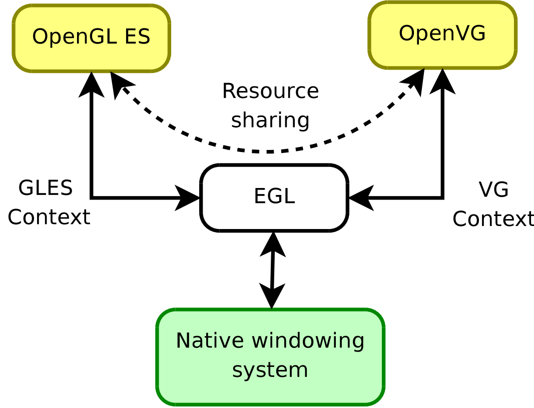

Figure 9. EGL provides a way to initialize resources for both OpenGL ES and OpenVG and coordinate their drawing with the platform's native windowing system. It also facilitates sharing of rendering resources across APIs.

to draw semitransparent shapes and to ensure that the edges of the shape are smooth. [48 p. 6]

9. **Framebuffer:** Similar to OpenGL ES, the OpenVG framebuffer houses the final rendered image. It also supports most of the equivalent pixel operations. In addition, it supports a number of filtering operations such as blurring and convolution. [48 p. 116, 121]

3.6. Native Windowing System Integration

OpenGL ES and OpenVG provide ways to draw vector graphics, but they lack the ability to determine where the graphics should actually end up. Even if the application is only interested in drawing on the physical display of the device, it still needs to use an additional API to make that choice. This API is called EGL. The role of EGL in relation to the graphics engines can be seen in Figure 9. Essentially, EGL is the glue that binds the graphics engine to the platform's native windowing system. For this reason, EGL and other similar APIs are commonly called binding APIs.

EGL is used to prepare a data structure called the graphics context to APIs such as OpenGL ES and OpenVG. The graphics context defines, for instance, the output surface and its configuration for the graphics engine. The configuration of a surface can be used to adjust the quality of the produced graphics. Among other things, it specifies the number of bits of color fidelity to use when rendering and whether or not use antialiasing to smooth the edges of geometric primitives.

EGL provides three different types of surfaces for application use. The most commonly used one is the window surface, which directly maps the graphics to a window in the underlying platform's native windowing system. Window surfaces also use back buffering, which is a way of ensuring that graphics do not appear on the screen until the application indicates that the current frame is ready. This helps to eliminate flickering graphics that would normally be seen if the screen is updated while the graphics

are still being drawn. When displaying graphics on the screen, a window surface is usually the most straightforward choice. [50 p. 25]

The second surface type is a pixel buffer or pbuffer surface. A pbuffer is an off-screen surface, and therefore graphics rendered to a pbuffer do not appear directly on the screen. Pbuffers are typically used when a native windowing system is not available, or when mixing rendered graphics with other graphics APIs that are not explicitly compatible with OpenGL ES or OpenVG. [50 p. 24]

The final available surface type is the pixmap surface. Most windowing systems provide a pixmap object, which is simply a picture that can be drawn among the user interface controls in a window. A pixmap surface is a special type of surface that can direct the rendering output of a graphics engine to such a native pixmap. This pixmap can then be used as a part of a graphical user interface. [50 p. 30]

The different EGL surfaces and configurations have important implications for application portability. Of the three surface types, only pbuffers are universally supported. Certain configuration features such as antialiasing may not be available on every platform. Applications written with portability in mind should therefore strive to adapt in the absence of such features. Surfaces and configurations also have an effect on application performance. In general, window surfaces have significantly better performance than other types of surfaces due to the fact that they are double-buffered and therefore allow the graphics engine to asynchronously work on multiple frames at once. Double-buffering ´ can also eliminate the overhead of copying the surface pixels on the screen, as the display controller can simply display one of the buffers while the graphics engine is rendering to the other one. [36 p. 261]

EGL can also be used to share certain types of resources among graphics engines. For instance, a picture rendered with OpenVG could be used as a texture in OpenGL ES. While this type of sharing could also be accomplished by explicitly copying the OpenVG framebuffer pixels to an OpenGL ES texture, such an operation could incur a significant synchronization penalty and pixel data transmission overhead between the graphics engines. Therefore, applications should strive to use the functionality provided by EGL for these purposes. [50]

3.7. Performance Factors

The relatively low level of both the OpenGL ES and OpenVG APIs provides applications with a great deal of power in the form of control over the graphics engine; the application can decide what it should draw and when. This power, however, comes with the added responsibility of making sure the rendering process is efficient. In contrast to higher level scene graph APIs, in OpenGL ES and OpenVG it is the application's responsibility to make sure not to, for instance, waste processing cycles by needlessly drawing complex geometry only to cover it later with other objects. This fine grained level of control is the source of many performance and quality problems in OpenGL ES and OpenVG applications.

The mapping from graphics content to performance is not a straightforward matter. This is because the graphics engine is in effect a black box for the application; neither OpenGL ES or OpenVG specifies the inner workings of the engine, but rather only the inputs and outputs. The obvious benefit of this approach is that graphics engine ven-

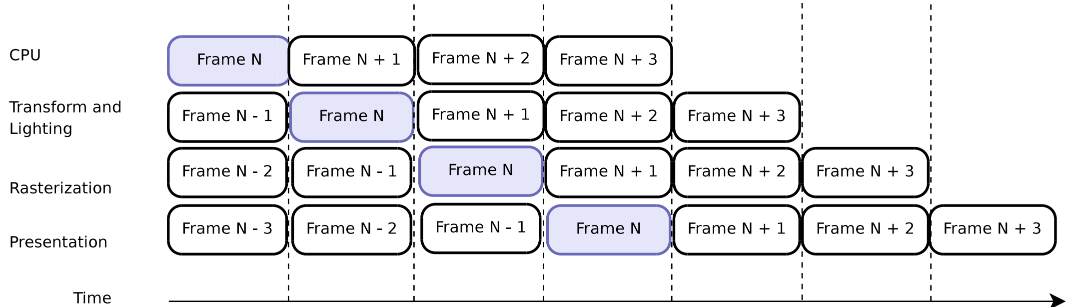

Figure 10. On hardware rendering architectures, independent pipeline stages can work on different graphics frames concurrently.

dors are free to experiment with novel rendering algorithms to improve performance. From the application developers perspective, a more undesirable effect is that particular graphics content will have a radically different performance profile depending on the graphics engine used. Even with this degree of freedom, useful general guidelines for better performance can be still defined.

Probably the most dramatic difference in performance can be seen between a software-based engine and a hardware graphics engine. Software engines are often limited by the speed at which they can rasterize pixels, especially at higher display resolutions. On such architectures, it is advantageous to limit the graphics content to its bare minimum. The old computer graphics adage that the fastest way to draw something is to not draw it at all is as valid as ever.

Graphics hardware, on the other hand, may be able to draw primitives many orders of magnitude faster, but their performance limitations might also be much more complex. It is very important to realize the extensively asynchronous nature of GPUs. In addition to the different pipeline stages working in parallel, the main application CPU is free to do other processing while the GPU is busy rendering. This level of parallelism is illustrated in Figure 10 [36 p. 106]. Notice how the level of parallelism extends to cover a full frame instead of just one rendering operation. If the application performs graphics operations that interfere with data dependence of the different pipeline units, it may lead to the whole pipeline performing synchronously. If the pipeline in the previous example was synchronous, its timeline would look similar to that of Figure 11 [36 p. 106]. The rendering throughput has decreased by a factor of four when compared to the parallel scenario.

Possible ways to induce a data dependency to the pipeline include reading back framebuffer pixels to the CPU memory, combining accelerated vector graphics with native window system rendering, or modifying existing graphics resources such as textures or paints while they are being used. What makes these operations so treacherous is that on software renderers, they do not usually cause any unexpected drops in performance, and thus the performance problems will only manifest themselves when the application is moved to a platform with a graphics accelerator. However, due to the high performance of GPUs relative to software implementations, an application using synchronous commands might still perform better on such a platform, but the full potential of the GPU would still not be utilized.

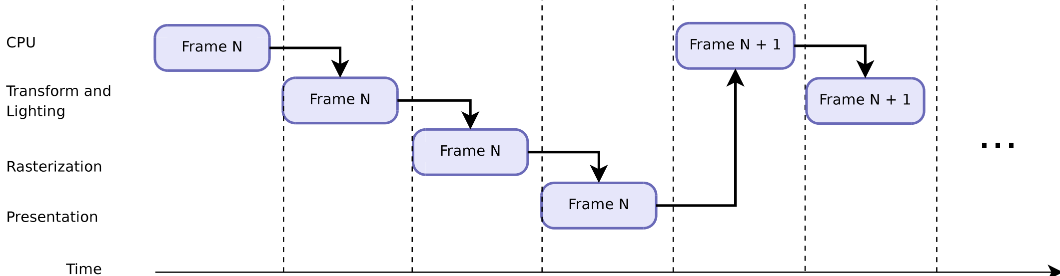

Figure 11. Introducing a synchronous dependency between pipeline stages causes a major performance degradation as the units must wait for their predecessor to finish.

API design also has a big impact on performance. As APIs evolve, their designers learn more about the performance impact of various architectural decisions and often try to amend the design to improve the situation. The opposing force for this process is the often quite justified desire to maintain backwards compatibility with existing applications using the API. The end effect is that over time, APIs accumulate various methods of doing things, some of which are less efficient than others. An article by Henning [51] highlights the delicate balancing act that API designers must do between backwards compatibility versus API usability and performance. The cost of rewriting existing applications and re-educating developers inhibits the ways a backward-compatible API can evolve.

A representative illustration of this issue comes from the number of different ways the vertices of an object can be specified in OpenGL. Traditionally, the vertices were defined one at a time with a separate API call for each vertex:

```
1 glBegin(GL_TRIANGLES);
2 glVertex3f(0.0f, 0.0f, 0.0f);
3 glVertex3f(0.0f, 1.0f, 0.0f);
4 glVertex3f(1.0f, 1.0f, 0.0f);
5 glEnd();
```

It was soon realized that this method incurred a great overhead, especially when the number of vertices was large. The issue was rectified with the introduction of vertex arrays, which can be used to specify the attributes of many vertices at once:

```
1 const GLfloat vertices[] = {
2     0.0f, 0.0f, 0.0f,
3     0.0f, 1.0f, 0.0f,
4     1.0f, 1.0f, 0.0f
5 };
6 glVertexPointer(3, GL_FLOAT, 0, vertices);
7 glDrawArrays(GL_TRIANGLES, 0, 3);
```

Here the vertex coordinates are first defined as a normal C array. The `glVertexPointer` call is used to tell OpenGL where to find the array, and finally the geometry is commonly drawn with `glDrawArrays`. With vertex arrays, the number of API

calls remains constant regardless of the number of vertices. However, when considered in conjunction with a hardware accelerator, this solution is not ideal either. The problem is that each time the draw call is made, all the vertices have to be transferred from CPU memory to the graphics accelerator. If the vertices specify geometry that does not change from one frame to another, the copying operation is unneeded overhead. This problem, in turn, was solved by adding the vertex buffer object mechanism:

```
1  const GLfloat vertices[] = {
2      0.0f, 0.0f, 0.0f,
3      0.0f, 1.0f, 0.0f,
4      1.0f, 1.0f, 0.0f
5  };
6  GLuint buffer;
7  glGenBuffers(1, &buffer);
8  glBindBuffer(GL_ARRAY_BUFFER, buffer);
9  glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat),
10                 vertices, GL_STATIC_DRAW);
11 glVertexPointer(3, GL_FLOAT, 0, 0);
12 glDrawArrays(GL_TRIANGLES, 0, 3);
```

The vertex buffer object procedure also begins with a declaration of a regular C array containing the vertex coordinates. On lines 7 and 8, a vertex buffer object is created to house the array and activated. A vertex buffer object is essentially equivalent to a C array, except that it can be stored in dedicated graphics memory for better performance. The `glBufferData` call on line 9 is used to transfer the vertex data into the vertex buffer object. Finally, the vertex buffer object is indicated as a source of vertex data with `glVertexPointer` and the geometry is drawn with `glDrawArrays`.

From these examples it is apparent that the more efficient a method of drawing geometry is, the more cumbersome and counterintuitive it usually is from the developer's perspective. Especially in the case of disruptive features, such as vertex buffer objects, which have been retrofitted to an existing API without changing the underlying design, the end result can be quite complicated to grasp. It is no wonder that some desktop OpenGL applications are still written using the first approach listed above, because it is simply the easiest way of getting things done.

When the OpenGL ES was being crafted, the inefficient nature of the vertex specification API calls was understood, and therefore they were not included. Both vertex arrays and vertex buffer objects are, however, available in OpenGL ES and the programmer should be aware of their differences.

We have now introduced the main vector graphics APIs for this thesis: OpenGL ES and OpenVG. The reader should now be familiar with the way graphics are constructed in both APIs as well as some general factors that affect the performance of vector graphics applications. Next, we discuss the set of tools we have designed to assist in solving vector graphics quality issues.

4. GRAPHICS QUALITY ANALYSIS TOOLKIT REQUIREMENTS

This chapter introduces the detailed requirements for each component of the Graphics Quality Analysis Toolkit. We begin by examining the software environment of each component. This is followed by the introduction of the core use cases, which are used to derive the detailed design of the software.

4.1. Functional Requirements

As illustrated in Figure 5 on page 21, the central components of the Graphics Quality Analysis Toolkit are:

1. a **Tracer** for capturing all OpenGL ES and OpenVG graphics commands executed by a Symbian application
2. a **Trace Player** for repeating the captured graphics commands
3. a **Trace Analyzer** for examining and manipulating the graphics command trace files
4. **Instrumented OpenGL ES and OpenVG graphics engines** for extracting detailed content statistics from trace files

As the Tracer, the Trace Player and the Trace Analyzer are all used in different environments under different circumstances, we discuss the functional requirements of each component separately.

4.1.1. Tracer

The tracer is used to extract API call traces from applications. The main targeted platform is the most recent iteration of the Nokia smartphone system software, the S60 3.x series, which is based on Symbian OS 9.x. Some special aspects of Symbian OS 9.x must be taken into account when designing the tracer. These are discussed in more detail in the next chapter, which focuses on the low level design of the toolkit components.

The tracer must be able to capture all API commands and the associated data executed by OpenGL ES 1.1 and OpenVG 1.0 applications. However, it must also be generic enough to be extensible to other similar C APIs with moderate work. The design should acknowledge that multiple graphics applications may be running simultaneously and any of the applications may be multithreaded.

An important requirement for the tracer is that it must not require access to or modification of the source code of the traced application. The source code of a debugged application may not always be conveniently available due to licensing constraints or other limitations. Furthermore, changes to the application source code imply that the application must be recompiled. This is not always practical, as some system software

might have unique build environment requirements. If the tracer was based on the modification of the application source code, the design might easily preclude tracing applications written in other languages, such as Java. Therefore, the tracer must be designed to work with completely unmodified applications.

Tracing should be a non-intrusive operation for the target application. While some performance degradation is expected, the functionality of the application must not be compromised. A necessary compromise is that if the application is exhibiting a quality issue that is very critical to timing, the tracer may not be able to capture a reproducible representation of it due to the introduced overhead.

The constraints of the mobile runtime environment imply that the tracer should strive to minimize its memory and CPU footprint. Tracing an application should not considerably hinder its performance, so that interactive applications remain usable. The memory usage of the tracer should also be bounded when required. Due to the possible high volume of trace data, it should also be possible to specify whether the trace should be saved to a file or relayed directly to a remote storage device. These and other aspects of the tracer must be configurable without having to rebuild the system software.

4.1.2. Trace Player

The Trace Player has a dual role in the toolkit configuration. Firstly, it is used directly to play back recorded trace files, possibly on a different graphics engine than the one used to record the trace. The second use is to play back recorded trace files using an instrumented graphics engine to obtain detailed content statistics. Due to these circumstances, the Trace Player must be able to run on both Symbian OS 9.x and Microsoft Windows XP.

The Trace Player must reproduce the exact API calls made by the original application. While some minor variations in memory addresses and other details are allowed, the intention of this requirement is that the sequence of graphics operations of the original application must be reproduced with enough accuracy to produce identical output. To enable reliable benchmarking, the performance of the player must be well understood.

Matching the requirements of the tracer, the Trace Player must support both OpenGL ES 1.1 and OpenVG 1.0, with similar provisions for adding support for additional C-based APIs.

4.1.3. Trace Analyzer

The Trace Analyzer is used to examine and edit recorded traces in a workstation environment. The tool is primarily run in a Microsoft Windows XP environment.

The main purpose of the Trace Analyzer is to assist in pinpointing application quality issues in recorded trace files. The aim is not to make the analysis process completely automatic, as that would limit the tool’s utility in cases that were not considered during the design phase. Instead, the analyzer should strive to be a general purpose utility for extracting as much information as possible from trace files. This information should

also be transferable to other programs in various formats from low level raw data to high level reports. The Trace Analyzer also allows for the editing of trace files and extracting logical parts of traces to form new ones.

The user interface of the first iteration of the Trace Analyzer tool is command line-based. This decision was made on the grounds that it is first more important to concentrate on the low level functionality of the analyzer. Designing a graphical user interface is not feasible until the major usage patterns of the tool are explored in practice. A graphical user interface can be later implemented to complement the command line mode once the design of the tool has stabilized.

Pervasive automation support is a major requirement of the Trace Analyzer. While this is in part provided by the command line interface, a more powerful programmatic scripting interface must also exist.

The Trace Analyzer employs the Trace Player to extract low level content statistics from a recorded trace file. The design must define a way to orchestrate this operation in coordination with various OpenGL ES and OpenVG engines. Adding new custom content features should also be possible with relative ease.

4.2. Non-functional Requirements

The general functional requirements for the toolkit can be summarized with the following principles:

- **Completeness**—each component of the analysis suite should aim for complete coverage of the respective problem field. For the tracer, this means that every API call and associated parameter is properly saved to the trace file. Similarly, the Trace Player should reproduce the original application behavior as exactly as possible.

- **Invariance**—to ensure consistent behavior, all parts of the suite should minimize the side effects of their operation. In the context of tracing, this means that the original application behavior is not altered with the introduction of the tracer, other than with possibly reduced performance. In the Trace Analyzer, editing a trace file should not inadvertently alter the ordering or function of the API calls.

- **Portability**—in the interest of portability, a common requirement for all components is that the amount of platform dependant code must be minimized. Portability also concerns trace files in the sense that they must be transferable from one system and graphics engine to another when applicable.

4.3. Use Cases

The Graphics Quality Analysis Toolkit design is derived from a number of concrete use cases:

1. Unsatisfactory application performance

2. Visual error in application

3. Application quality analysis
4. Graphics engine benchmarking
5. Graphics content analysis

These cases are a generalization of actual tasks and support requests that have been assigned to the Nokia Display & Graphics Software team. The team is responsible for delivering graphics technology to other organizational units in the form of graphics engines, performance testing, and support. This responsibility places the team in a very central role when it comes to the graphical quality and performance of applications. A common situation is that a given application is suffering from a quality problem, which is assigned to the team as a defect in the graphics engine. The graphics team must then investigate and classify the error, leading to a great workload with traditional methods. In the following use case definitions, we demonstrate how this workload can be reduced with the aid of the Graphics Quality Analysis Toolkit.

4.3.1. Unsatisfactory Application Performance

The first use case focuses on an application suffering from poor performance. The reason for the poor performance is unknown. The application in question is using either OpenGL ES or OpenVG.

The goal of this use case is to identify the reason for the poor performance as quickly and easily as possible using tools that have been made for exactly that purpose. Using these tools, the engineer examining the issue should be able to clearly communicate the cause for poor performance to the application's owner.

Experience has shown that trying to assess the reason for poor graphics performance of an application using a debugger is time consuming and frustrating. Not only must the engineer discern the internal behavior of the application from the source code, he or she will also have to make judgments about the executed graphics API calls one by one. Debugging interactive applications is also difficult because the application is halted whenever the debugger is triggered. Often the source code might not be readily available, introducing the additional challenge of deciphering application behavior from assembly code.

Using a profiler to approach the issue may not always be helpful, as profilers only work at the level of function calls. Individual function calls convey little information about the graphics content itself. The profiler only indicates how much time is spent inside a particular function, but the actual reason for the processing load might be a completely different function call executed earlier. The time spent executing a particular graphics function therefore depends on both the parameters passed to that function and the state of graphics engine at that point in time.

The first step in solving the quality problem is to classify it into one of the four categories described in Section 2.6 on page 17. This classification is the base for guiding further optimization work. If the error is found to reside in the application, the process will also provide valuable information for solving the error.

The work flow of this use case is illustrated in Figure 12. The process begins by running the problematic application on a mobile device or a PC-based emulator. The

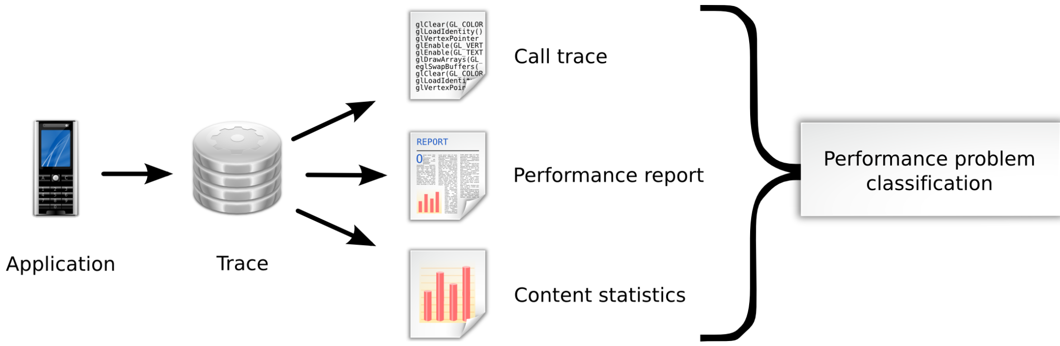

Figure 12. In the first use case, an application suffering from a performance problem is traced and the problem is categorized based on various types of trace analysis.

graphics commands of the application are captured using the tracer. Depending on the application, one or multiple traces may be taken. Then, the generated trace files are transferred onto a workstation for analysis. The Trace Analyzer is used to produce a number of reports and statistics concerning the graphics content. These figures are finally used to categorize the quality problem.

4.3.2. Visual Error in Application

In the second use case, an OpenGL ES or OpenVG application suffers from a visual error or a crash in the graphics engine. A failure in the application code is the responsibility of the application developer and solving it is outside the scope of this thesis; our objective is merely to indicate which component is causing the error.

The goal in this case is to quickly identify and isolate the error. By error isolation, we mean reproducing the error with the bare minimum of required API calls. If an error is simplified in this manner, finding the cause for the error in the graphics engine is much easier.

Traditionally errors of this kind are solved using a debugger or source code inspection. While a debugger is useful in many cases, it is easily defeated by errors that are hard to reproduce by using the application. Such errors easily lead to much work trying to narrow down the error by repeatedly triggering it. Source code inspection is also an ineffective tool for large or complicated applications.

The workflow for this use case is illustrated in Figure 13. At first, the problematic application is traced on a mobile device. The trace is then brought into the Trace Analyzer, which is used to isolate the error. The error is isolated by first extracting the API calls for the frame exhibiting the error into a new trace file. This new file is then played back on the original device to verify that the error still appears. If the error does not reappear, a different call sequence is selected from the original trace until the error reappears.

Once the error is isolated to a minimal API call trace, it is used for three different purposes: firstly, it is used to verify with a reference engine that the error is indeed caused by the graphics engine; secondly, it is used to debug the error in the graphics engine; finally, the trace is converted into a C source file that replicates the behavior of

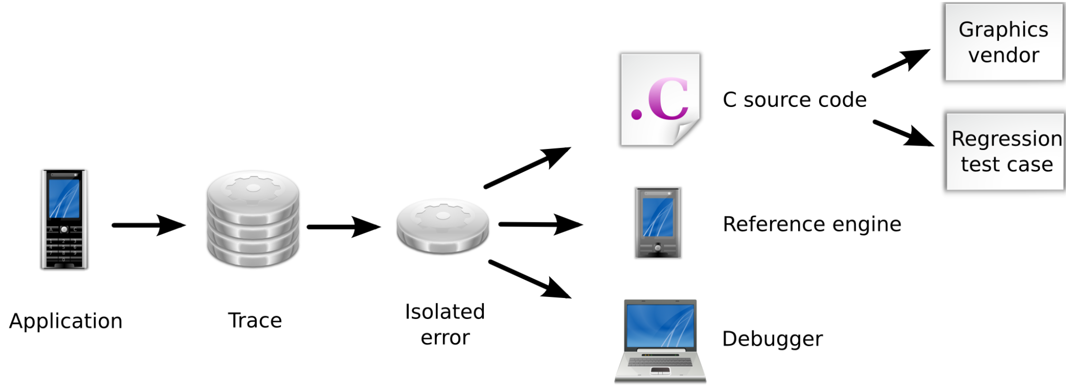

Figure 13. The second use case focuses on an application experiencing a rendering error. The application is traced, the error is isolated from the trace and analyzed further with a reference graphics engine, a debugger and as a standalone C source file.

the trace. As the C file is completely independent of the rest of the analysis toolkit as well as the original application, it can be given to the graphics engine vendor for further analysis. Additionally, a regression test case is created using the C code to guarantee that the same error does not reappear in a future version of the graphics engine.

4.3.3. Application Quality Analysis

The third use case is about assessing the quality of a vector graphics application. Such analysis should be performed even if the application in question is not suffering from an obvious performance problem. The reason for this is that the analysis can highlight certain areas that can be improved to conserve battery, CPU, or memory usage.

The technique outlined in this use case is not meant to displace traditional methods of quality analysis such as application profiling or code reviewing; the intention is to complement such methods by providing more information to guide further optimization work.

The workflow for this use case, as illustrated in Figure 14, begins with recording one or multiple graphics traces from the examined application. In general, multiple traces should be created if there are great variations in the application's graphics under different circumstances. These traces are then brought into the Trace Analyzer, which is used to produce detailed content statistics on the graphics content.

An engineer analyzing the application can be interested in a wide variety of statistics about the graphics content, but in general the focus is on figures that can be used to judge the complexity of the graphics content against the capabilities of the platform and graphics engine. Some examples of such measures are the level of overdraw versus the platform fill rate capacity, the number of transformed vertices versus the transformation capacity, and the texture upload traffic versus the system memory bus capacity. The system should make it relatively easy to extract other similar statistics.

In addition to statistics, the generated reports also include information about the executed API call patterns. Features such as platform dependant operations, inefficient procedures, redundant call sequences, and other inefficiencies should be highlighted.

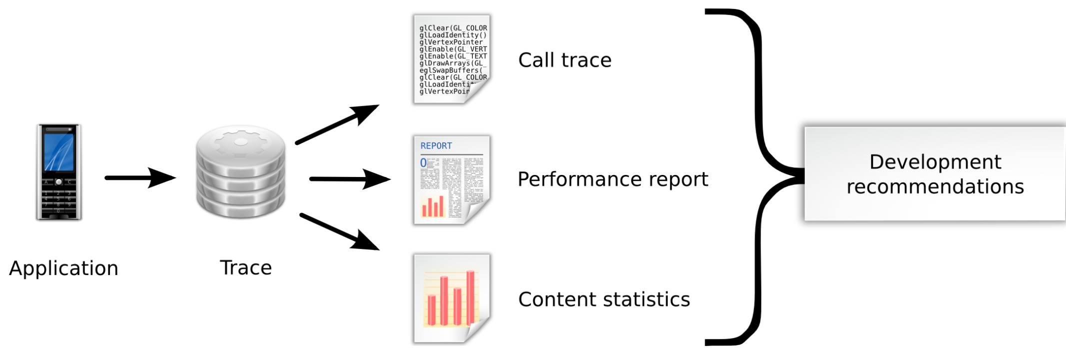

Figure 14. In the third use case, an otherwise well-behaved application is traced and analyzed in terms of vector graphics quality. A set of development recommendations is derived from the analysis.

Based on the extracted information, the engineer is able to give clear development recommendations for the application developer.

4.3.4. Graphics Engine Benchmarking

The fourth use case switches the focus from applications to graphics engines. A new graphics engine needs to be benchmarked to estimate its fitness for rendering the graphics of a particular application.

Graphics engines are commonly benchmarked with a combination of dedicated synthetic benchmarking programs and real applications. Synthetic benchmarks are problematic, because they are generally written by graphics experts and therefore do not share the same inefficiencies that real applications sometimes do. On the other hand, using real applications for benchmarking is troublesome, because the application code usually needs to be modified to implement automated reliable benchmarking runs. While such modifications can be done for a single test run, it quickly becomes a chore if the number of applications is increased. Furthermore, in early stages of development, new graphics engines commonly run on prototype hardware, which might not support the running of regular system applications.

In this case, the workflow begins creating a trace from the application to be used for benchmarking. The analyzer provides two ways of creating benchmarks. Firstly, the Trace Player can be used to directly play back a specially-created benchmark trace file. This method is straightforward and flexible, since the Trace Player can readily play back any valid trace file. A possible drawback with this method is that the process of reading and interpreting the trace file during the benchmark may skew the results. The second way is to convert the benchmark trace file into C source code, which is then compiled and executed to obtain the benchmark results. This method is more involved, since the benchmark must be recompiled each time the trace is changed. The results obtained with this method, however, are likely to be more accurate, since the compiled benchmark has less overhead during runtime than the Trace Player. Furthermore, C-based benchmarks have fewer system dependencies in comparison to the Trace Player. In this use case, we will explore both ways of creating benchmarks.

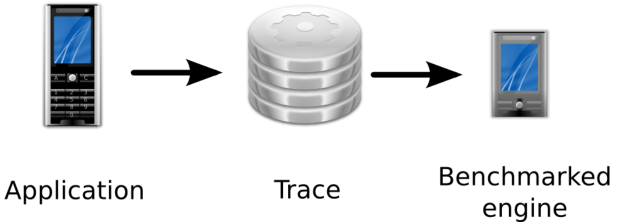

Figure 15. The fourth use case consists of benchmarking a new graphics engine using existing application content extracted via the tracer.

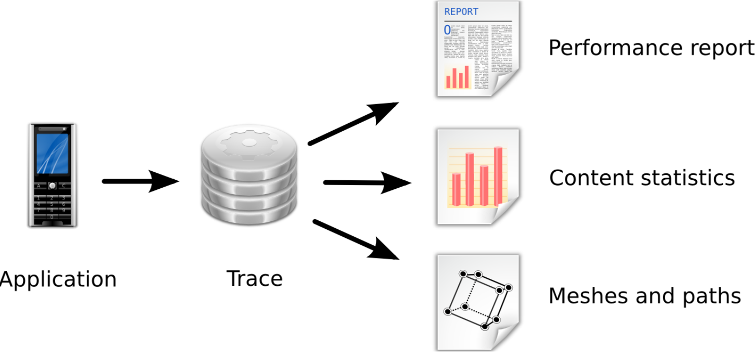

Figure 16. In the fifth use case, the graphical content of an application is analyzed in detail.

The general process for this use case is illustrated in Figure 15. The output of this use case is an estimate of how well the new graphics engine will perform when rendering the application graphics content.

4.3.5. Graphics Content Analysis

The final fifth use case demonstrates the in-depth content analysis functionality of the toolkit. This functionality is presented with the hope that it will be useful to more detailed content analysis, clustering, and performance estimation work in the future.

An overview of this use case is shown in Figure 16. As before, a selected application is traced to produce a trace file of graphics operations. The trace file is then analyzed to extract high and low level content features in the form of compiled reports and raw data. Instrumented engines for OpenGL ES and OpenVG will be used to calculate some of the statistics.

The objective of this use case is to demonstrate a systematic approach to gaining an in-depth understanding of typical graphics content. This knowledge could be used to:

- Guide the development and optimization of graphics engines. Low level statistics on real application graphics content are valuable input to this work.
- Cluster different applications into representative performance classes based on graphics content complexity. This can be used to quickly estimate the performance level of an application on a piece of graphics hardware.

- Create synthetic benchmarks for graphics engine performance estimation. The objective of these benchmarks is to overcome the limitations of using traced application content directly for benchmarking as presented in the previous use case. Synthetic benchmarks can be more easily parameterized to produce different kinds of graphics processing loads.

Previously, this kind of data was based on a limited number of application studies and anecdotal information from content developers. The process outlined in this use case defines a practical means of obtaining this information.

We have now presented the requirements and essential use cases for the Graphics Quality Analysis Toolkit. The following chapter provides insight into how these requirements are translated into a software implementation.

5. GRAPHICS QUALITY ANALYSIS TOOLKIT ARCHITECTURE

Having defined the requirements and the core use cases for the Graphics Quality Analysis Toolkit, we are now ready to proceed to the detailed design and architecture of the software components. This chapter focuses on the design choices, justifications, and limitations of each component.

5.1. Tracer and Trace Player Generator

Early in the design phase of the Tracer and Trace Player it became clear that both components would consist of large amounts of repetitive code. The OpenGL ES 1.1 and OpenVG 1.0 APIs have 187 and 84 functions respectively, and each function requires a corresponding entry in the tracer and Trace Player. Writing these functions by hand would be tedious and error-prone. To minimize the implementation effort, we decided to employ a code generator to create most of the Tracer and Trace Player code.

A code generator generally works by reading a compact domain-specific representation of the desired system and creating a corresponding program in source code form. This source code can then be fed into a normal compiler to produce an executable program. Code generators are extensively used for creating parsers, state machines, and other applications that are too complex to be written by hand.

The operation of the code generator in our system is illustrated in Figure 17. First, a set of API configuration directives combined with platform-specific information is used to create a tracer project file. This project file is then fed into the code generator to produce both the Tracer and the Trace Player for the targeted API and platform. The tracer project is also used to pass the API configuration to the Trace Analyzer.

In addition to source code, the generator also produces platform-specific build system files that are used to compile each generated component. These files are created for all the respective build systems of Symbian OS, Microsoft Windows, and Unix derivatives. Build file generation was especially useful for Symbian OS, since compiling applications targeting that platform requires writing a large number of resource files in addition to the source code itself.

5.1.1. API Configuration

All components of the toolkit require in-depth knowledge about the targeted API. This information is derived from a number of sources to make up the tracer project file. The most essential bit of information is the list of API functions including their parameters and types. It is specified in the form of a standard C header file. Additionally, a separate configuration file is used to define attributes of the API, such as:

- Which functions trigger rendering, frame swapping or API termination.
- The state model of the API and how function parameters are mapped to it. See Section 5.2.5 on page 55 for further information.

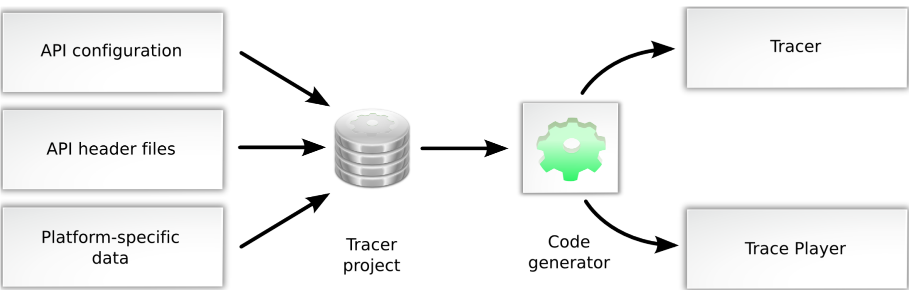

Figure 17. API- and platform-specific configuration data is combined to produce a project file, which is in turn used to generate the Tracer and the Trace Player.

- Which functions are extension functions and require special processing. See Section 5.2.8 on page 60 for further information.
- What kind of platform-specific objects the API employs.
- Rules for calculating the sizes of array parameters.

An excerpt from the OpenGL ES configuration file for the `glLightfv` function is shown in Figure 18. The `glLightfv` function is used to control different parameters of OpenGL ES lights, such as their color or position. The particular example shown in Figure 18 specifies how each of the three parameters of the function, *light*, *pname*, and *params*, should be processed by the Tracer and the Trace Player. As the first two parameters are simple integers, the configuration only indicates where they should be mapped in the state model. The quoted strings specify state model paths. Our state model system is described in more detail in Section 5.2.5 on page 55. The final parameter, however, is more interesting, as it is an array. The size of that array depends on which light attribute is being changed, as different attributes require a different number of numeric values. As the changed attribute is chosen via the *pname* parameter, the number of elements in the *params* array can also be determined through that parameter. The configuration shown in Figure 18 therefore specifies, that by default the *params* array will contain 4 elements by default, except if the *pname* parameter equals the numeric constant `GL_SPOT_DIRECTION`. In this case, the array will have 3 elements. Similar conditions are repeated for each possible variant of the *pname* parameter to cover all possibilities.

As the pattern of using different parameters to specify array sizes is very common in EGL, OpenGL ES, and OpenVG, we designed the configuration syntax to provide a compact representation for such cases. While this system covers most array parameters in these APIs, some special cases, such as EGL attribute lists, require hand-written code for calculating the array size. In practice, we found these cases to be rare.

Several configuration directives are also required for compiling the generated components. On Symbian OS, the symbols in dynamically linked libraries are not found by names but by ordinal numbers [8 p. 388]. This means that the tracer must know the ordinal number of each API function in order to find the corresponding function in the system graphics engine. The same information is also needed for compiling the tracer, because binary compatibility requires that the ordinals of its exported functions must

```
 1  glLightfv:
 2  {
 3    light:       "ctx.light"
 4    pname:       "ctx.light.parameter"
 5    params:
 6    {
 7      state:     "ctx.light.parameter.value"
 8      metatype(class = "array", size = "4"):
 9      [
10        size(condition = "pname", value = "GL_SPOT_DIRECTION",
11             result = "3")
12        size(condition = "pname", value = "GL_SPOT_EXPONENT",
13             result = "1")
14        size(condition = "pname", value = "GL_SPOT_CUTOFF",
15             result = "1")
16        size(condition = "pname", value = "GL_CONSTANT_ATTENUATION",
17             result = "1")
18        size(condition = "pname", value = "GL_LINEAR_ATTENUATION",
19             result = "1")
20        size(condition = "pname", value = "GL_QUADRATIC_ATTENUATION",
21             result = "1")
22      ]
23    }
24  }
```

Figure 18. Example configuration directives for the `glLightfv` OpenGL ES function. The settings shown here specify how the three parameters for the function are processed by the Tracer and the Trace Player; most notably, the *pname* parameter is used to determine the size of the *params* array.

match those of the original graphics engine. In our system, these ordinal numbers are parsed from the same DEF files used by the Symbian OS build system.

5.1.2. Working with Generated Code

The use of code generation greatly simplified the task of implementing both the Tracer and the Trace Player. Both are complex pieces of software, which required numerous refactoring steps to reach their present state. Without code generation, such iterative improvements and wide-reaching modifications would have been very tedious to manually apply into the code base. We routinely had to make changes that involved modifications to Tracer and Trace Player code for every API function. The most significant advantage brought by the code generator is the ease of adding support for new C-style APIs; a minimal Tracer and Trace Player can be generated almost directly from the header file of the API.

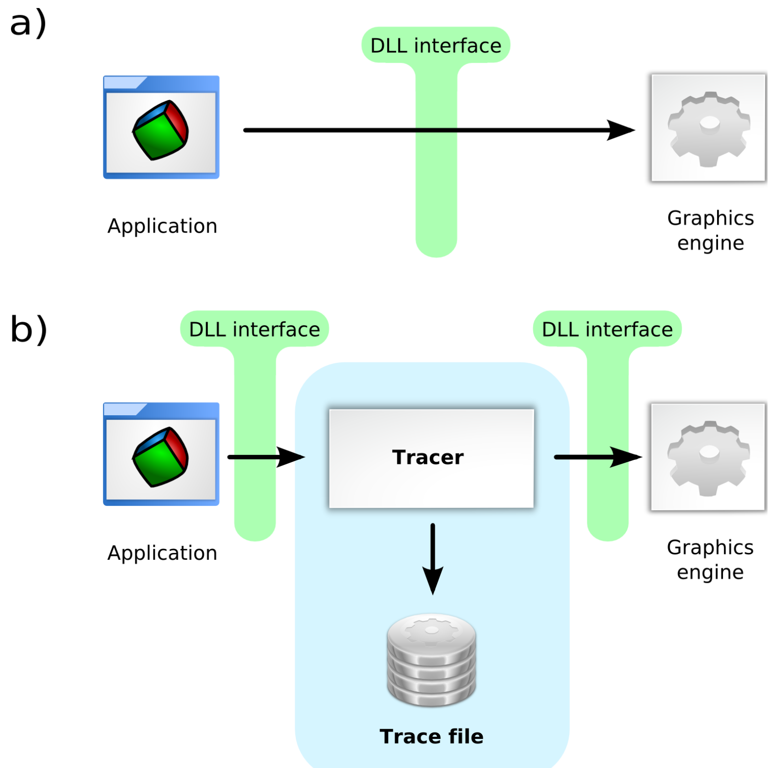

Figure 19. a) A regular Symbian graphics application is dynamically linked to the system graphics engine. b) The tracer masquerades as the system graphics engine by providing a matching DLL interface to the client application. This arrangement enables the tracer to copy all executed rendering commands to the trace file.

The downside of code generation is the introduction of a new level of abstraction into the system. Debugging was somewhat hindered by the fact that the mapping from the API configuration to the final generated code was not always straightforward. An important design principle is that the generated code must not be modified by hand. If the code is changed after generation, the changes must be redone each time the code is generated again. The need for very specific improvements to the generator output was handled through the addition of code hooks or placeholders. The API configuration can use these hooks to replace specific parts of the generated code with customizable code. This functionality is used to inject hand-written code into the Tracer or the Trace Player to, for instance, save the contents of enabled vertex arrays in the `glDrawElements` OpenGL ES function.

5.2. Tracer

The tracer is responsible for saving each executed graphics API call and the associated data to a trace file. This means that it must find a way to position itself between the data flow from the application to the graphics API. To do this, the tracer exploits the fact that on Symbian and other modern operating systems, applications are dynamically linked to the graphics engine, as seen in the top half of Figure 19. The graphics engine DLL,

dynamic link library, is renamed, and the tracer takes its place. The tracer provides an identical DLL interface to the application in order to make itself indistinguishable from the original graphics engine. This setup, as illustrated in the bottom half of Figure 19, guarantees that every graphics API call executed by the application ends up in the tracer entry point for that particular function.

Due to the security limitations of Symbian devices, the tracer is not be a user-installable program, since the graphics engines and other system software on a mobile phone cannot be overridden. For this reason, the tracer must be integrated directly into the system software image of the phone.

Function-level Tracing

Each API call entry point in the tracer is responsible for calling the respective function in the graphics engine and saving all data related to the function, including its return value, to the trace file. An example trace function is shown in Figure 20. In the code, a new event is first opened for the current function call. Each function call made while the tracer is active maps to an event in the trace file. Then the underlying graphics engine function is called with the original parameters. The function call is also surrounded by timing code that measures how much time is spent in the graphics engine. Finally, the original function parameters are added to the event and the event is closed.

The function parameters are saved only after calling the original function because some of the parameters might get modified during the engine function call, and the trace file should contain the final values of such parameters. The function might also have a return value that needs to be saved, and return values are only available after the engine function has been called.

Data Types

In the example case shown in Figure 20, all the function parameters were simple floating point values. This is not the case, however, with the full spectrum of functions found in OpenGL ES and OpenVG. The parameter types fall under three distinct classes: basic types, arrays, and opaque objects. Each of these classes requires special consideration to guarantee that the proper information is written to the trace file.

The first class of basic types is the simplest. It encompasses all atomic types, such as integers and floating point numbers of varying precision. The majority of function parameters fall under this class. Their serialization only requires that a standard encoding is used when saving and loading the values.

The second class – array parameters – contain sequential data instead of a single value. In C, the value of an array parameter is a pointer to the memory block containing the array data. The tracer must acknowledge this fact by serializing the data contained in the array rather than the memory address, because the address would be meaningless when read back from the trace file. Serializing the array data is a simple matter of copying it to the trace file, as long as the length of the array is known. In C, the length of an arbitrary array cannot generally be deduced without additional information. In

```
1  void glColor4f(GLfloat red,  GLfloat green,
2                 GLfloat blue, GLfloat alpha)
3  {
4      /* Establish a new tracer event */
5      TREvent* event = trBeginEventByIndex(0, 33);
6
7      /* Call the underlying graphics engine function */
8      trBeginCall(event);
9      TR_CALL4(void, event->function,
10              GLfloat, red,  GLfloat, green,
11              GLfloat, blue, GLfloat, alpha);
12     trEndCall(event);
13
14     /* Save the function parameters */
15     trFloatValue(event, "red",   red);
16     trFloatValue(event, "green", green);
17     trFloatValue(event, "blue",  blue);
18     trFloatValue(event, "alpha", alpha);
19
20     /* Signal the end of the event */
21     trEndEvent(event);
22 }
```

Figure 20. Example tracer code for the `glColor4f` OpenGL ES function. The tracer first calls the corresponding function in the system graphics engine and then saves the parameters of that function to the trace file.

some cases, this information is not readily available and the tracer must resort to more advanced techniques. These circumstances are discussed in more detail in Section 5.2.5.

The tracer only supports homogeneous arrays, in which each array element is of the same type. This is sufficient because most arrays in the targeted APIs are homogeneous, and those that are not, can be simply saved as arrays of raw bytes.

The third class – opaque parameters – are values that carry more meaning than just their numerical value. An example of such a value is the reference to a native window, which is passed to the `eglCreateWindowSurface` function. The reference is used to assign the newly created window surface to a particular window created by the application. To save this information to the trace file, the tracer does not write out the reference itself. Instead, every relevant detail about the window, such as its width and height, is written. This information can be later used to construct an equivalent window when playing back the trace. This approach also makes it possible to transfer traces from one system to another, where the implementation of native resources such as windows may be completely different.

5.2.1. Platform Security

To lower the probability of viruses, Trojan horses, and other malware infecting mobile devices, version 9 of Symbian OS introduced a mandatory access control mechanism called Platform Security. From an application's point of view, Platform Security restricts access to certain system files and services based on the assigned capabilities of the application. Every application binary and shared library running on the device is digitally signed, and the set of capabilities assigned to an application cannot be changed after the application is signed. Several different signing certificates are defined, each granting a different level of capabilities. [8 p. 315]

Platform Security also controls how application code can be linked together during runtime. The governing rule is that an application with a certain set of capabilities cannot use a DLL with fewer capabilities. This is done to prevent a privilege escalation, where a DLL could get more capabilities than it was originally assigned. [8 p. 321]

The implications of Platform Security for the tracer are two-fold. Firstly, the trace files may not be written into restricted system directories. For this reason, the tracer reads a runtime configuration file, which can be used to override the trace file location. Secondly, to guarantee that all vector graphics applications on the device remain working even after the tracer is installed, the tracer DLL must be given the highest level of capabilities. Furthermore, due to Platform Security, system software cannot be overridden by user installable applications. These special requirements imply that the tracer must be installed directly into the mobile device's system software image. This makes installing the tracer somewhat difficult, because the user must have access to a system software development environment. This factor is mitigated by the runtime configurability of the tracer, which guarantees that generally there is no need to alter the tracer after installation. For instance, a configuration file can be used to limit the tracer to only certain applications or graphics APIs. Furthermore, if needed, the tracer can be built as an unprivileged DLL under a name different to that of the system graphics engine. This tracer DLL can then be used by linking the examined program directly to it.

5.2.2. Performance Considerations

In the following sections, we will detail the optimizations used by the tracer in order to meet the required level of performance.

Write Buffering

Maintaining an acceptable level of performance while tracing applications is greatly dependant on the tracer's ability to write out data to the trace file at a sufficient rate. Our initial approach was to use a synchronous write operation. As Symbian only performs write caching for the file allocation table of the file system and not the actual file data [8 p. 377], the performance of this method was found to be unacceptable. We then implemented a write buffer mechanism, which gathers a large amount of data into a memory buffer and writes the whole buffer into the output file at once. This brought

performance to an acceptable level, but the synchronous buffer flushing caused a long pause whenever the buffer became full. This prompted the implementation of a fully asynchronous buffering mechanism, in which a dedicated worker thread collects data into an array of buffers in a round-robin fashion and flushes the filled buffers into the output file. With this solution in place, the tracer is able to sustain sufficient write performance, as long as the average data bandwidth does not exceed the capabilities of the output device. This solution does have a memory usage trade-off due to the buffer allocations, but the size and number of buffers can be adjusted as needed. The tracer can also be run in a completely synchronous manner to guarantee a complete trace in the case of a graphics engine crash.

Reducing Redundancy

In addition to improving write performance, it can also be beneficial to minimize the amount of data being written. In our first trials, a two minute OpenGL ES animation with roughly 30 000 rasterized triangles per frame generated a 250 megabyte trace file. This was clearly not practical, given the limited storage space of a mobile phone. A commonly observed property of animated graphics is a high level of frame coherence, that is, a frame tends to resemble the one preceding it. On the API call level, this means that nearly identical sequences of API calls are repeated from frame to frame. We found that the biggest contributors to the trace file size were the various arrays used to describe vertex coordinates, texture images, triangle indices, and other similar structures. We found that a very high percentage of the trace file data consisted of repetitive array data. Based on these findings, it made sense to implement a compression method to reduce the amount of redundant trace file data.

Array data compression in the tracer is based on the observation that graphics applications tend to keep constant array data in the same physical array; if the data does not change, then there is little point in moving it to a different array. For example, if an application draws a mesh using vertices stored at memory address 4000, a future draw call referencing vertices at memory address 4000 is likely to refer to the very same vertices. The compression scheme is based on the idea that instead of saving the vertex data again, we simply state that the data is the same as in the previous draw call.

Obviously, by only comparing memory addresses, the tracer would fail to notice if the application had in fact modified the array contents after it was saved to the trace file. This is why the tracer must, upon encountering a previously seen array, verify that the array contents have not been modified before marking it as a duplicate. If the array has changed, the tracer must write out its new contents to the trace file. Ideally, the tracer would mark the memory region occupied by the array in a way that it could automatically detect any modifications to it. Unfortunately, Symbian OS does not provide such functionality, and therefore the tracer must track array modifications explicitly.

Our initial attempt at detecting array modifications was to calculate a message digest value for the array contents and compare that to the previous value. The problem with this approach was that a simple message digest algorithm was prone to collisions, in which the same digest value was assigned to different array data, while a more complex algorithm was computationally too intensive.

A more complete array tracking algorithm would need to make internal copies of each encountered array in order to later check whether the array was modified. The algorithm implemented in the tracer is a compromise in the sense that it only tracks changes to arrays that have been encountered at least twice. In practice, this is a workable solution, since replicating the same array a maximum of two times into the trace file does not constitute a major overhead. The array tracker also allows for limiting the array cache size to ensure that system resources are not exhausted by the algorithm.

While the array tracking method described here is effective for most applications, it fails to reduce the trace data volume for applications that dynamically generate geometry that is different for each frame. An example of such an application is one that renders graphics using dynamic objects, which only contain the visible set of geometry at each point in time. Since the structure of the compound objects changes very frequently, the tracer must write each encountered variation to the trace file. This leads to poor performance. Fortunately, we have found very few such applications, so the impact of this limitation is relatively minor.

5.2.3. Portability

The C language was chosen for implementing the Tracer and the Trace Player. This was a natural choice, since the targeted APIs were also C-based. Using C also made it easier to port the software to new operating systems.

The tracer is an unusual software component in the sense that it does not have a single entry point or an initialization sequence. In contrast, normal executable programs have one main function, which is invoked by the operating system when the program is started. Similarly, dynamic libraries commonly define a single initialization function, which must be called prior to any other function in the library. Since the binary API of the tracer must match that of the traced DLL, any special initialization sequences cannot be relied on.

To overcome this limit, the tracer simply checks a global initialization flag at the start of every API call invocation; if the flag is not set, the tracer has not yet been initialized for the particular thread. The problem with this approach is that Symbian does not directly support writable static data in libraries [45 p. 38]. This limitation was worked around by using a global data API provided by Symbian, although at the minor cost of an extra kernel mode switch upon each traced API call.

In addition to global data, other services such as file system access, memory allocation and library symbol lookup are very dependant on the underlying operating system. For this reason, such services were encapsulated in a small utility library inside the tracer to keep the rest of the code platform neutral.

5.2.4. Trace Files

Trace files are essentially serialized streams of function calls and their associated parameters. The tracer can write trace files in both text and binary format. The text format is provided as a quick debugging aid, since it does not require a separate conversion step for obtaining a human readable call trace. To simplify the design of the Trace

Player and the Trace Analyzer, importing traces in plain text format is not supported by our system.

The binary trace file format is a low-overhead tokenized stream encoding, which can be read and written by all components of the toolkit. The format supports a simple phrase book compression scheme, where repeating data sequences can be assigned a shorter identifier. The encoding is designed to be self-contained in the way that it is not tightly coupled with any specific API configuration. While such coupling could have helped to somewhat reduce the trace file size, it would have also meant that changing the API configuration would have invalidated all traces taken with the old configuration. Explicitly maintaining backwards compatibility with the old format or converting existing trace files to the new format was not seen as a viable alternative, as API configuration changes were very frequent, especially early in the development.

The trace files can be saved to different output devices as needed. On a mobile device, the user can choose between a RAM disk, the built in flash memory of the device, a memory card or a high throughput proprietary debug interface to a host computer.

5.2.5. State Tracking

State tracking is the process of keeping track of the graphics engine state throughout the execution of an application. State tracking is needed for two purposes in our system. The first is to enable the tracer to save all application data passed to the graphics engine. The second use is to model the relative dependencies of the API calls, making it possible to extract a set of frames from a longer trace and perform in-depth analysis of the call trace.

Our aim was to create a generic state tracking solution that is not limited to either OpenGL ES or OpenVG. Instead of explicitly modeling the complete API state, the emphasis was set on the dependencies between various API calls due to the special requirements of the analysis tool. An important property was that the state tracker can be used to manipulate a trace with the Trace Analyzer in such a way that the order and parameters of the original API calls are preserved. As state queries or render calls do not generally modify the state, we do not need to model their behavior.

The state tracking done in the tracer is a very small subset of the full state tracking mechanism used in the Trace Analyzer. To distinguish these two cases, the simpler tracer state tracking is referred to as runtime state tracking.

Runtime state tracking is essentially about maintaining references to API call parameters that cannot be serialized to the trace file at the time of the API call itself. A classic example of such API calls is the vertex array functionality of OpenGL ES. The problem can be illustrated with the simple procedure of drawing a triangle.

```
1 const GLfloat vertices[] = {0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f};
2 glVertexPointer(2, GL_FLOAT, 0, vertices);
3 glDrawArrays(GL_TRIANGLES, 0, 3);
```

The code first defines a set of three two-dimensional coordinates for the triangle corners or vertices. The triangle corners are located at $(0, 0)$, $(0, 1)$ and $(1, 0)$. The second line tells OpenGL ES where in the memory to find these coordinates and how

they are formatted. Finally, the third line of code instructs the graphics engine to draw one triangle with the three coordinates.

All parameters of the API calls on lines 2 & 3 can be saved trivially by the tracer, except for the pointer to the coordinate data on line 2. The problem is that in the C language, it is generally impossible to deduce the length of an array merely from a pointer to it. Therefore, we cannot save the vertex data into the trace file when the `glVertexPointer` call is made, since we do not know how many vertices to expect. The number of vertices is only known when the triangle is drawn at line three.

The solution is to save the vertex pointer into the state tracker at the `glVertexPointer` call. Later, at the `glDrawArrays` call, the number of vertices can be calculated directly from the function parameters and the actual vertex data can be retrieved through the stored pointer.

Similarly, full state tracking also deals with the API call parameters. Instead of saving the parameters to a data stream, the goal here is to determine which preceding API calls the current API call depends on. This information can be used to decide which API calls are redundant and can be discarded without affecting the program outcome.

As an example, let us examine the task of choosing the filtering mode of a texture in OpenGL ES.

```
1 glBindTexture(GL_TEXTURE_2D, 3);
2 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Here, the first API call assigns texture number three to the active two-dimensional texture unit. The second call assigns the magnification filter for the bound texture. It is clear that these API calls need to be made in this order and both are needed to achieve the expected outcome. Therefore, a dependency exists between them.

To model this dependency between API calls, we designed a hierarchical data structure called the *state tree*. The state tree is a directed acyclic graph, in which vertices represent the elements of a state structure and edges the dependencies between them. The state tree also holds the concrete state values.

One possible state tree for the texture filtering example is shown in Figure 21. As seen in the figure, there are two different kinds of elements in the tree: groups and nodes. The groups can contain a list of child nodes and concrete state values, while the nodes contain a set of subgroups, one of which is marked as current. The groups are used to store state values at various levels in the tree while the nodes are kinds of switches that control traversal through the tree.

As the state tree has a single root, it is possible to refer to all other elements of the tree through unambiguous state paths. For example, the texture filtering mode GL_LINEAR can be found through the state path:

[Root] → [Texture target] → [GL_TEXTURE_2D] → [Texture name] →
[3] → [Texture parameter] → [GL_TEXTURE_MAG_FILTER]

Taking into account that the nodes in the tree implicitly define the successive group, the path can be simplified to

[Root] → [Texture target] → [Texture name] → [Texture parameter]

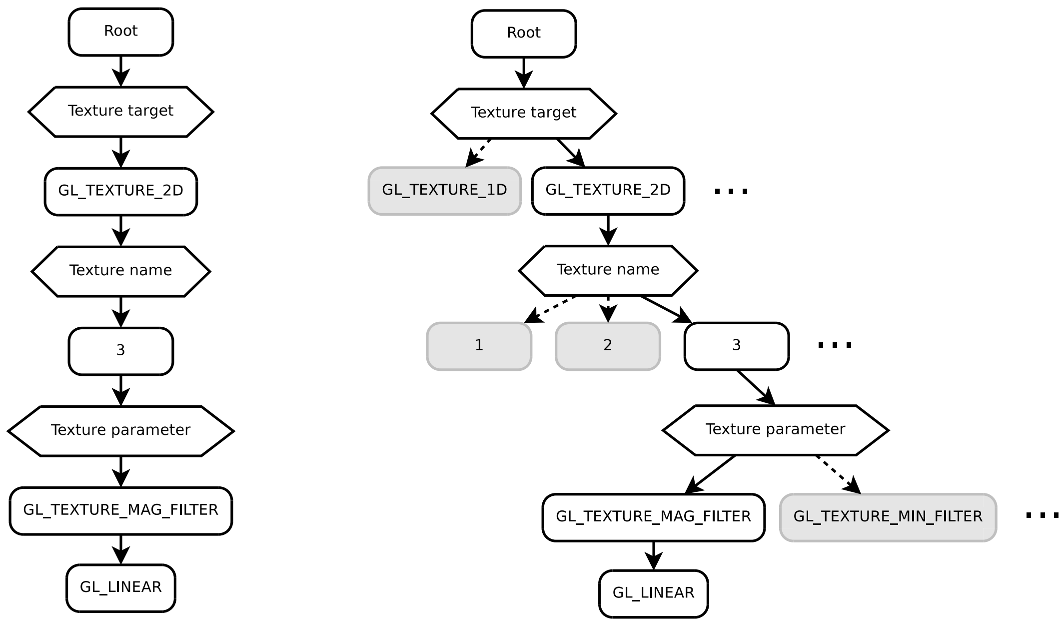

Figure 21. A possible state tree for storing the filtering mode for a texture object. The tree on the left only shows the specific elements used to store the filtering mode, while the tree on the right also shows some alternate options for traversal, highlighting the tree-like nature of the structure. The groups in the tree are drawn as rounded rectangles and the nodes are illustrated as angled rectangles.

Table 1. Mapping API call parameters to state tree elements using state paths.

API call	**Parameter**	**State path**
`glBindTexture`	target	Root→Target
	texture	Root→Target→Name
`glTexParameteri`	target	Root→Target
	pname	Root→Target→Name→Parameter
	param	Root→Target→Name→Parameter→Value

Finally, the API calls and their parameters are mapped to elements of the state tree using state paths. An example of such mapping for the API calls used in this particular case can be seen in Table 1. Each state path points to either a group or a node in the state tree. If the state path points at a group, the value of the respective parameter is stored at that group. Conversely, a state path ending in a node sets the currently active subgroup of that node.

Given a function call and the associated parameters, calculating the updated state is simply achieved by processing the state paths for the parameters ordered from the shortest to the longest. Finding the prerequisite API calls for a given API call can be done by collecting all the calls that have contributed to any part of the state paths associated with the given call. Extracting prerequisite API calls from a trace is discussed in more depth in Section 5.4.4 on page 70.

The state tracking model presented here covers most functions in OpenGL ES 1.1 and OpenVG without modifications. However, some functionality, such as matrix

stacks and object deletion, requires special code for correct dependency information tracking. Our system enables this by allowing the user to define custom state processing code for the required API calls. In practice, we found the state model's greatest benefit to be the compact representation; the 1,500 lines of OpenGL ES state configuration in our system is tiny compared to the amount of hand written code that would have been needed to implement it manually.

5.2.6. Tracing OpenGL ES

Many of the OpenGL ES state setting functions follow a common pattern with regard to the size of array parameters. For example, the `glMaterialfv` function, which is used to modify surface material properties, takes three parameters: the affected surface side, the name of the material property to set and an array containing a new value for the property. Since the final parameter is an array, the tracer must know its size in order to save it to the trace file. Here, the array size depends on which material parameter is being set. The rule is that by default the array contains four elements, unless the material shininess is being set, in which case the array contains only one element. Since similar instances of array sizes depending on the value of another parameter are abundant in the API, a special compact representation for them was implemented in the tracer.

The OpenGL ES tracer has a number of special optimizations for improving performance. The rendering state is tracked in order to limit the number of vertex arrays the tracer has to observe. The array tracker is also prevented from making internal copies of arrays such as texture data, which are unlikely to be reused in the future.

5.2.7. Tracing OpenVG

OpenVG was designed to be conceptually similar to OpenGL ES, although with a more object-oriented approach. Instead of a special set of functions for each object class, OpenVG has a set of generic attribute access functions that can be used to modify and query objects of any type. This reduces the number of different function variants from the API and generally makes the tracer design simpler.

Path Coordinate Arrays

As with OpenGL ES, the difficulties encountered while implementing the OpenVG tracer revolved around array parameters. A central object class in OpenVG is the path, which is the main geometric primitive of the library. Paths consist of a list of commands that make up the path and a list of coordinates for the commands. A command is an instruction to OpenVG, indicating how to draw the next segment of the path, i.e., should it be a line, a curve or some other shape. The list of coordinates specifies where this next path segment should be drawn. [48 p. 49]

The length of the path coordinate array depends on the commands that make up the path, since different commands use a different number of coordinates. When a path is normally extended, a list of path commands and the associated coordinates are ap-

pended to it. In this case, the tracer can simply calculate the length of the coordinate data by looking at the given path commands. However, OpenVG also allows the application to modify any contiguous subset of the stored path coordinates. Since the function for modifying path coordinates does not have a parameter indicating the size of the coordinate array, the tracer must keep a copy of the path commands in memory. This command list is then used to calculate the expected number of path coordinates upon demand. The command list is also updated to reflect the changes to the internal command list of the path object.

Negative Image Strides

Another difficulty in tracing OpenVG arises from the way that image data is uploaded to the graphics engine. Images in OpenVG are similar to textures in OpenGL ES: they can be used to apply regular patterns to shapes and to import graphics from external sources such as photographs. When images are uploaded to OpenVG, the library allows the application to specify how consecutive pixel rows or scanlines are laid out in memory. This is done with a property called stride, which indicates how many bytes there are between the beginning of one scanline and the beginning of the next one. The stride value may be distinct from the scanline length or the image width due to the fact that a single pixel may occupy more than one byte of storage. Additionally, some operating systems use bitmaps in which scanlines are not tightly packed together, but instead have a number of unused padding pixels in between. The padding is used to guarantee that each scanline starts at an aligned memory address regardless of the image dimensions. This is usually done to improve performance, since access to aligned memory can be much faster than access to unaligned memory on some systems. The relation between the image width, the number of padding pixels, and the image stride is illustrated in Figure 22.

The use of padding pixels in images has no consequences for the tracer as such, since we can assume that the padding pixels reside in readable memory and the whole image data can be accessed as a contiguous block. However, OpenVG also allows the application to specify a negative stride value [48 p. 109]. A negative stride value signifies that the image scanlines are stored in reverse order, that is, each scanline is stored at a smaller memory address than its predecessor. This can effectively be used to flip the image vertically while transferring to the graphics library. Since the OpenVG graphics coordinate system [48 p. 40] is vertically inverted in comparison to Symbian OS's coordinate system [45 p. 321], negative stride values are commonly used to achieve consistent transfer of bitmap data between the two systems. This method is illustrated in Figure 23. In the example, the image sent to the graphics engine is either transferred as is or inverted vertically depending on the sign of the stride value.

For the tracer, a significant consequence of negative stride values is that the image data pointer points to the start of the last scanline of the image. This is in contrast to the general practice in C, where an array pointer points to the start of the array data. In order to save the image data to the trace file, the tracer must first calculate where the scanlines of the image reside in memory. Since the Trace Player also needs to be able to reconstruct the image data in memory, we opted for the approach of normalizing the scanline order in memory prior to saving the data. This made the Tracer and Trace Player design significantly simpler, although at the cost of introducing

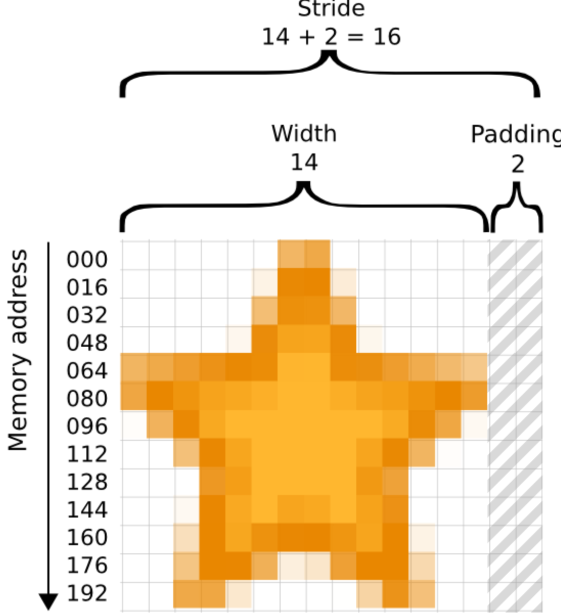

Figure 22. A stride value indicates the memory offset between the start of one bitmap scanline to the next. Here, the width of the image is 14 units, but due to the two extra padding pixels on each scanline, the resulting stride value is 16 units.

a small non-linearity to the tracer: all traced function calls that used negative stride values are converted to ones using positive values in the trace file. We feel that this design choice did not compromise the reliability of the system in a significant manner, as the transformation done by the tracer is very straightforward.

Binding APIs

One of the OpenVG engines used in this work did not support the standard EGL binding API. Instead, it exports a proprietary C++ interface that allows drawing OpenVG graphics directly into Symbian OS bitmaps. This required us to create a special variant of the OpenVG tracer in order to trace applications using this engine. This special binding API also affected the design of the Trace Player with regard to trace portability; this matter is detailed in Section 5.3.2 on page 64.

5.2.8. Tracing EGL

In comparison to OpenGL ES and OpenVG, EGL is a simple API, which was straightforward to implement into the tracer. Historically, EGL has been bundled in the same DLL as OpenGL ES on Symbian [52]. With the introduction of OpenVG, however, it became apparent that EGL should also be usable independently of OpenGL ES. This created a motive for having a separate DLL for EGL. We support this arrangement in our system by generating up to three separate tracers for EGL, OpenGL ES,

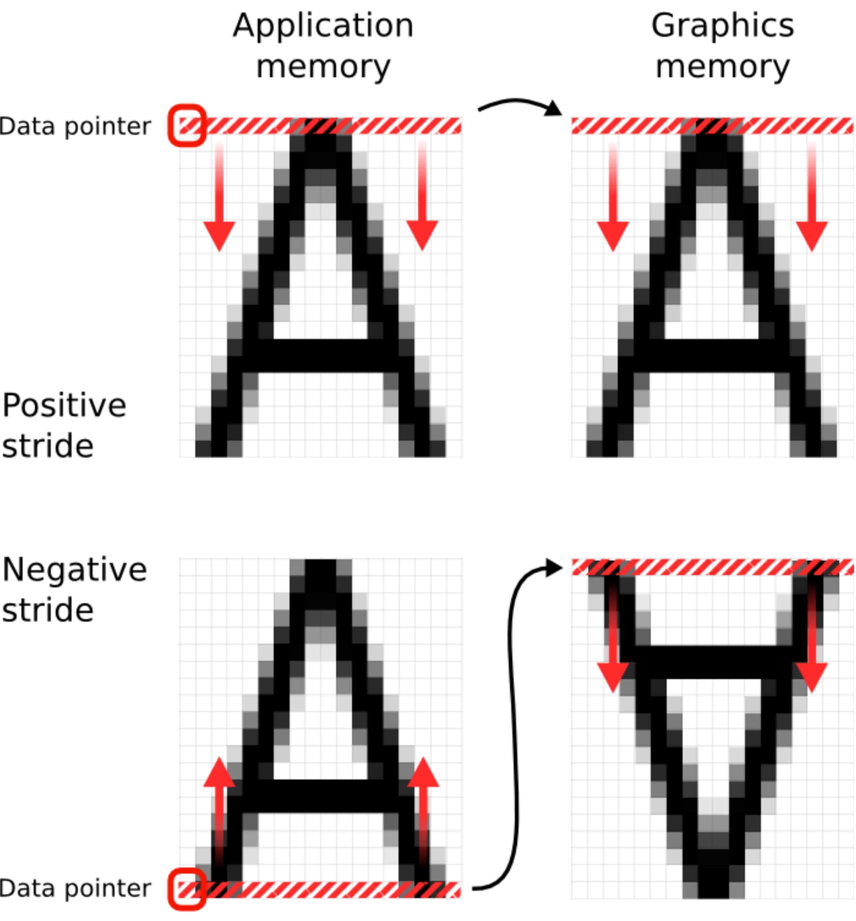

Figure 23. A negative scanline offset or stride is used to vertically mirror image data passed to a graphics library. Normally, the image stride is positive and the data pointer points to the top left corner of the image. When a negative stride value is used, the data pointer points to the first pixel of the bottom scanline of the image.

and OpenVG. Each traced API call is then annotated with a globally unique sequence number, which can be later used to combine the different API traces in a temporally coherent manner.

One notable challenge in tracing EGL was posed by its extension mechanism. While the OpenGL ES and OpenVG specifications define a set of core features supported by the respective graphics API, they also allow graphics engine vendors to export their own set of additions to the core feature set. An example of such an addition is a more efficient texture data format for OpenGL ES or an improved blending operation for OpenVG. These additions are called extensions, and the API for using them is provided by EGL.

If an extension reuses the functions of the original API, it can be supported simply by defining the new parameter values introduced by it in the tracer configuration. However, some extensions also include a set of new API functions. To use an extension function, an application must first query EGL for a pointer to that function. The

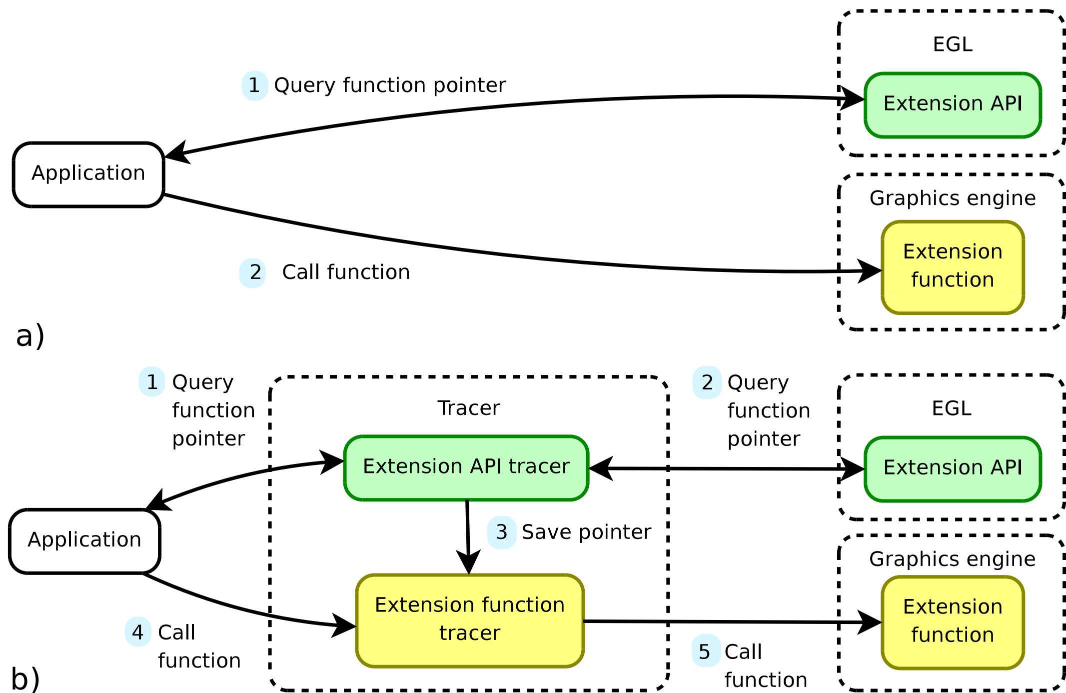

Figure 24. a) An application queries EGL for a pointer to an extension function and then calls that function. b) The tracer intercepts the application's extension function query and substitutes the return value with a pointer to its own extension function.

pointer may then be used to call the function. The problem is that the function pointer returned by EGL points directly to the extension function inside the graphics library; when the function is called, the tracer is bypassed completely and the function call does not appear in the trace file. This dilemma was solved by modifying the extension API inside the tracer to return a pointer to the tracer's own version of the extension function. The pointer returned by EGL is saved and subsequently used to call the real extension function. This procedure, as outlined in Figure 24, also works when EGL resides in a DLL different to the graphics engine.

EGL configuration objects pose a problem for trace portability, as an application can refer to EGL configurations using opaque identifier numbers without explicitly constructing them through, for instance, `eglChooseConfig`. If only the configuration identifier was saved, it would not be possible to determine the effective configuration parameters only by looking at the trace file. These attributes are essential for the Trace Player, since it must be able to find a compatible configuration when used with a different graphics engine. The solution to this problem was to treat the configuration objects as platform-dependant objects similar to windows and bitmaps. Just as the window object contains the essential attributes of a window, such as its width and height, the configuration object stores all the effective attributes of the EGL configuration in the trace file.

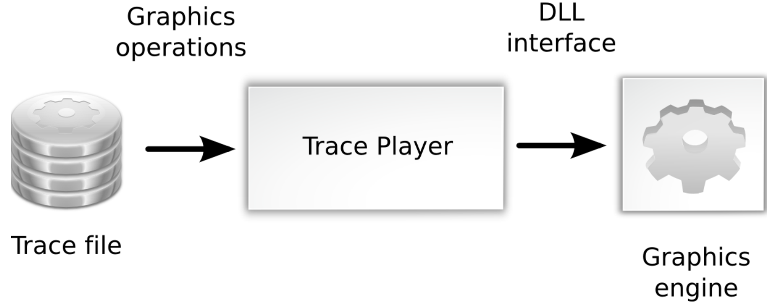

Figure 25. The Trace Player reads a trace file and executes the corresponding API calls to reproduce the original graphics operation sequence.

```
1  void glColor4f_event(TREvent* event)
2  {
3      /* Decode the arguments for the function call */
4      GLfloat red   = trGetFloatValue(event, "red");
5      GLfloat green = trGetFloatValue(event, "green");
6      GLfloat blue  = trGetFloatValue(event, "blue");
7      GLfloat alpha = trGetFloatValue(event, "alpha");
8
9      /* Call the graphics engine function */
10     glColor4f(red, green, blue, alpha);
11 }
```

Figure 26. Trace Player code for the `glColor4f` OpenGL ES function. The player first loads the parameters for the function from the trace file and then calls the corresponding function in the graphics engine.

5.3. Trace Player

The Trace Player is used to reproduce a call sequence recorded in a trace file. The basic operation of the tracer is to decode a function and its associated arguments from the trace file and then call the corresponding function in the graphics engine. As seen in Figure 25, the player is dynamically linked to the graphics engine like a regular graphics application.

Example code for replaying the `glColor4f` function can be seen in Figure 26. The program first extracts the four color components stored in the trace file and then calls the graphics engine function with the proper arguments. This code does essentially the opposite of the corresponding tracer function in Figure 20 on page 51.

5.3.1. Trace Normalization

The dynamic nature of array data in trace files creates some difficulty for the Trace Player. As the trace is read and written in a chronologically linear order, the player has

identical knowledge of an array associated with a given function call as the tracer did when originally saving the call. Because of this the OpenGL ES vertex array problem discussed in Section 5.2.5 on page 55 also affects the Trace Player. When a vertex array pointer is first set, the array contents are undefined. This is because the length of the array is not known until geometry is drawn using it. Regardless of this, the player must be able to supply a valid pointer for OpenGL ES when executing the vertex array function call. Furthermore, the memory region referenced by the pointer must be large enough to accommodate all data that is assigned to it during the remainder of the trace file.

Solving this problem while maintaining the chronological linearity of the Tracer and Trace Player required us to add a separate offline preprocessing step for trace files. This normalization operation amends the trace file in a way that the stored array objects are always fully defined before their first reference. An alternative approach would be to make the tracer seek back and change parts of the already written trace when an array becomes defined. Given the elaborate output buffering mechanism in the tracer, this option was not feasible. Similarly, having the Trace Player scan the full trace file to establish its contents prior to playing it would have reduced performance and complicated the design of the player.

5.3.2. Trace Portability

Trace portability is the ability to replay a trace with a device and graphics engine other than which was originally used to record the trace. To make this possible, the Trace Player must anticipate and work around the differences between the environments used for trace playback. In the following section we discuss some aspects of trace files that may compromise their portability.

Platform-dependent Objects

The most obvious issue inhibiting trace portability is the use of native operating system resources, which are almost guaranteed to be incompatible in different systems. Windows and bitmaps are examples of strongly platform-dependent objects commonly used by graphics applications. The Trace Player manages such resources by abstracting them into a portable representation of their essential attributes. These attributes are then used to construct equivalent resources on the playback platform.

A common pattern in APIs is to first call one function to create an object and then pass a reference of that object to a number of other functions. The reference to an object is usually a pointer or another type of opaque numerical handle. In general, the concrete values of these object references are arbitrary, since they may be based on memory addresses and other non-deterministic quantities. From this follows that object references stored in trace files are unlikely to correspond to live objects in memory when trace is played back. For example, consider the following procedure of constructing an image and copying pixel data into it in OpenVG:

```
1 VGImage image = vgCreateImage(...);
2 vgImageSubData(image, data, ...);
```

Here the return value of the first function call is the reference to the image object. This reference is then passed to the image data transfer function to designate the target image. When this application is executed and traced, the image object might get assigned the identifier number 27. On a different platform, graphics engine, or even invocation, however, the image might get a completely different identifier. For this reason, the Trace Player keeps track of object creations and references while executing traces. In the above example, the image creation command causes the tracer to translate all future references to the `VGImage` object number 27 to the new identifier returned by the graphics engine. The object type must also be considered while doing this remapping, since objects of different type may well be assigned the same identifier. The same approach is applied to all object references in the trace file. This also includes objects that are not explicitly constructed by replayed function calls, such as windows and EGL configurations, since their identifiers are similarly not fixed.

EGL Configurations

The EGL configuration mechanism requires special consideration in order to guarantee trace portability. As mentioned in Section 5.2.8 on page 60, the full set of properties of EGL configurations are saved into the trace file. When a configuration object is referenced, the Trace Player uses this information to find a suitably close match for the original configuration from the current graphics engine. The player will optionally relax some requirements of the configuration until a compatible match is found. The trace file can also be modified to force the use of a particular known configuration if a fuzzy match is not desired.

As discussed in Section 5.2.7 on page 58, one of the OpenVG engines used in this work used a proprietary binding API instead of EGL. To ensure that a trace using one binding API is playable on a platform using the other API, the tracer includes code that emulates the essential features of one API using the other API. This approach was preferred over modifying the function calls in the trace file itself, since the emulation code is quite straightforward.

5.3.3. Performance Considerations

As the Trace Player is designed to be used in benchmarking, performance is an important aspect of the implementation. The internal processing requirements of the player are quite modest, since it does not need to perform any extensive state processing or tracking. For this reason, the playback speed is essentially limited by the trace file decoding rate. The design of the asynchronous tracer output buffer was adapted into a similar input buffer mechanism for the Trace Player. This buffer is filled by a dedicated reader thread, which strives to keep the player supplied with function call data. If memory permits, the buffer size can be increased to encompass the entire trace file to maximize performance.

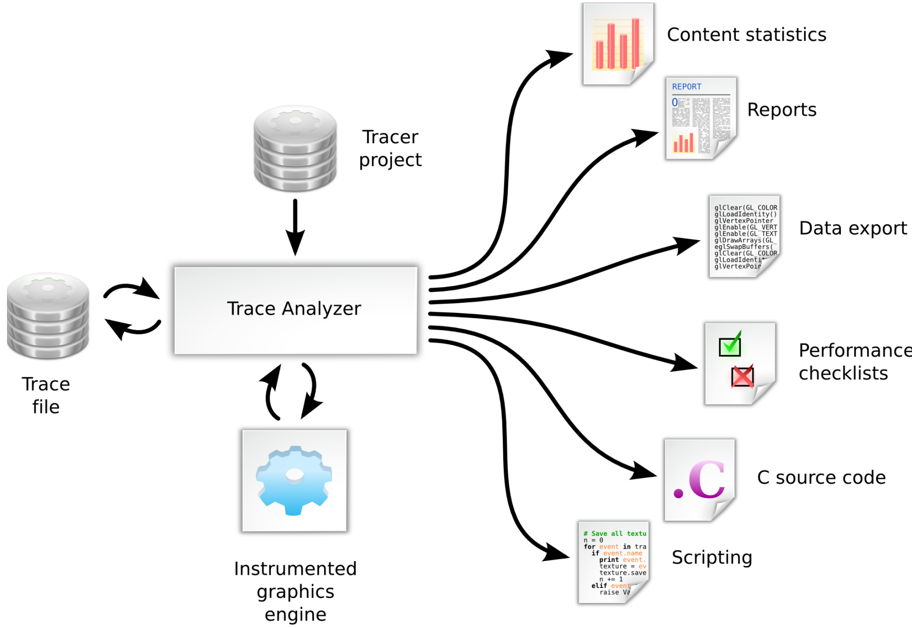

Figure 27. The Trace Analyzer is a versatile tool for editing trace files and extracting data from them.

5.4. Trace Analyzer

The Trace Analyzer is used to examine and process the binary trace files produced by the Tracer. The main purpose of the tool is to enable practical data extraction. An average trace file contains tens of thousands of API calls and megabytes of additional data. The analyzer provides access to data from the level of a single event to statistics that span the entire trace.

An overview of the Trace Analyzer and the data flow through it is shown in Figure 27. Although the main focus of the analyzer is on the extraction of data from the trace files, it also has functionality for editing trace files as well as synthesizing completely new traces.

In the following, we discuss the major features of the Trace Analyzer tool.

5.4.1. User Interface

The primary user interface of the analyzer is a command prompt in the spirit of interactive debuggers such as the GNU GDB [53]. The interface allows the user to manipulate one or multiple trace files at once using a number of low- and high-level commands. This interface can be operated both interactively and via automated scripts.

The analyzer interface is based on a simple procedural command language. It offers a set of generic commands, which work with all traces regardless of the used API, as well as a number of special commands to deal with OpenGL ES and OpenVG-specific entities. The user can also extend the language with custom commands. The built-in commands available for the user are show in Table 2.

Table 2. Commands for controlling the Trace Analyzer.

`calc-frame-stats`	For each frame marker event, calculate the sum of the instrumentation data for the intermediate events.
`calc-stats`	Calculate some derived OpenGL ES statistics based on instrumentation measurements.
`call-histogram`	Show the function call frequency histogram for a trace.
`checklist`	Run a GLES trace through a checklist of common performance issues.
`close`	Unload a trace file.
`export`	Export a trace to a file in a special format.
`extract`	Extract a portion of a trace to form a new trace.
`extract-state`	Extract a portion of a trace including the preceding state to form a new trace.
`grep`	Search for events matching a regular expression.
`help`	Show available commands or help on a specific command.
`info`	Show information about a trace.
`join`	Join two traces together to produce a new, third trace.
`list`	List events contained in a trace.
`load`	Open a trace file.
`load-inst`	Load previously computed instrumentation data for a trace file.
`merge`	Merge two traces to produce a temporally coherent third trace.
`new`	Create a new empty trace file.
`play`	Play back a trace file using a generated trace player.
`profiling`	Enable or disable call profiling for all executed commands.
`python`	Run an interactive Python interpreter that can be used to manipulate the loaded traces.
`quit`	Exit the analyzer.
`reload`	Reload the analyzer modules while keeping all loaded data intact.
`renumber`	Renumber a trace so that the events contained in it start at zero.
`report`	Generate a performance report of a trace.
`save`	Save a trace to a file.
`save-inst`	Save the current instrumentation data for a trace file.
`select`	Choose events from one or many traces based on a condition.
`set-egl-config`	Force the usage of a particular EGL config in a trace.
`show-plugins`	List the loaded analyzer plug-in modules.
`show-state`	Print out the computed API state at a particular trace event.
`simplify`	Remove redundant OpenGL ES commands from a trace.

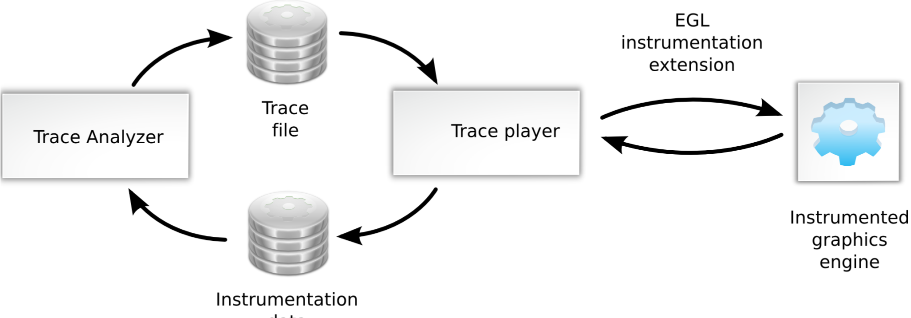

Figure 28. The Trace Analyzer uses specially instrumented OpenGL ES and OpenVG engines to extract detailed content statistics from a trace file. The analyzer plays back the examined trace using the Trace Player, which in turn queries the instrumentation sensor readings from the engine via the EGL instrumentation extension.

5.4.2. Trace Manipulation

An essential feature of the analyzer is the conversion of binary trace file data into other formats. The range of output formats spans from simple textual call sequences into more complex representation such as C source code. The analyzer also allows the extraction of a specific data subset, such as the collection of texture images from an OpenGL ES trace. Saving a trace in the binary format normalizes the trace data as required by the Trace Player.

The trace editing capabilities of the analyzer are based on two fundamental editing operations: extracting and joining sequences of serialized function calls or events. In the extraction operation, a new trace is formed from a contiguous subset of an existing trace. Conversely, the joining operation takes two existing traces and merges them to form a new trace. These basic low level operations form the basis for higher level trace manipulation functions, which can operate on larger entities such as graphics frames and non-contiguous event sequences.

Instrumented Engines

In addition to retrieving data directly stored in the trace file, the analyzer can be used to calculate more detailed measurements by playing back the trace file using a custom graphics engine with integrated statistical sensors. These statistics can be used to derive high level content features for the graphics calls stored in a trace file.

To obtain these statistics, the analyzer employs the Trace Player in conjunction with an instrumented graphics engine. This arrangement can be seen in Figure 28. First, the analyzer instructs the player to start executing the examined trace file. While replaying each function in the trace file, the Trace Player queries the latest instrumentation sensor values from the graphics engine and saves them to a data file. When the playback is complete, the Trace Analyzer reads the instrumentation data file, making the data available for the user.

We designed an EGL instrumentation extension for providing the Trace Player with a consistent and portable way of extracting the instrumentation data from a variety

Table 3. Content features and statistics provided by the Trace Analyzer and the instrumented engines.

Common content features	
API calls	Timestamp, duration, call histogram, array data traffic, frame duration, EGL configuration attributes
Buffer snapshots	Color buffer, depth buffer, stencil buffer
OpenGL ES	
General	Matrix operations, render calls, texture uploads
Primitives	Submitted, degenerate, frustum culled, backface culled, clipped, discarded, rasterized
Vertices	Submitted, transformed, viewport transformed, lit, cache accesses, cache hits
Rasterization	Fragment count, texture fetches, average triangle size, discarded fragments, estimated overdraw
OpenVG	
General	Matrix operations, render calls, image uploads, property reads/writes
Objects	Creations, attribute reads/writes
Paths	Segment count, coordinate count, tessellated polygon edges, accepted polygon edges
Rasterization	Fragment count, estimated overdraw

of graphics engines. The extension allows a graphics engine to expose a number of discrete instrumentation sensors. Each sensor is associated with a name, a textual description, and an integral or floating point number indicating the value of the sensor. The extension defines a number of functions that the Trace Player uses to query the various types of sensor data during runtime. Additionally, the extension allows the player to access a number of buffers inside the graphics engine, such as a color buffer, depth buffer or a stencil buffer. The player can optionally save the contents of these buffers into files. In this way, each event in the trace file can be associated with an image of the frame buffer at that point in time.

The analyzer can augment the collected data by calculating frame-level averages and converting the frame buffer snapshots to a standard image file format. It can also calculate some additional API-specific metrics, such the amount of uploaded texture data for OpenGL ES, without the aid of an instrumented engine.

The set of content features provided by the instrumented engines is designed to be easily extendable upon demand. The implemented features are listed in Table 3. As shown, some of the higher level features are API neutral, while the more specific statistics do not translate directly between APIs.

5.4.3. Content Statistics and Graphs

The main purpose of the Trace Analyzer tool is data extraction; as an average trace file contains tens of thousands of API calls and megabytes of additional data, practical means of finding the essential features of that data are needed. The analyzer provides access to data from the level of a single event to statistics that span the entire trace.

A simple but convenient method of examining binary trace files is to convert them to an equivalent plain text file. The textual trace format includes all the same events, parameters, return values, and timing information as the original binary file in an easy-to-read format. Additionally, the text format exporter can print content feature values next to each listed event and truncate large data arrays such as textures to make the output more manageable.

While the text format is practical for casual use, more sophisticated ways of exchanging data with other programs are also needed. The nearly universally-supported comma separated value or CSV file format can be used to transfer the instrumentation sensor readings and other statistics to other dedicated tools such as a spreadsheet.

The detailed event-level view of the analyzer is complemented by a number of higher-level abstractions. The performance checklist is an API-specific utility, which verifies a trace file against a set of predefined conditions. These conditions test for known performance deficiencies and other unwanted call patterns. Some of the checklist items apply to all graphics engines, while others are specific to the characteristics of a certain implementation. The tests included in the OpenGL ES checklist are listed in Table 4. Some of the checklist items are based on an OpenGL ES graphics optimization guide published by Nokia [54].

Should any of the checks fail, the analyzer provides the user with a list of the offending events that triggered the failure along with a textual description of the detected quality problem. In effect, the performance checklist is an expert system in the domain of vector graphics that can automatically provide a rough quality estimate of the traced application.

Since a trace file analysis commonly revolves around graphical issues, the analyzer also provides visual methods of browsing its contents. The tool can generate a report document, which contains both an overview of the whole trace in terms of event timings, instrumentation sensor readings, rendering targets, loaded textures and other statistics, as well as a detailed breakdown of selected graphics frames. To cut down on the output size, the report generator can automatically choose a number of representative frames. This selection process is done by comparing frame durations and color histograms and discarding sequential frames that are too similar. We chose to use the Bhattacharyya histogram distance measure [55] due to its low calculation overhead and sufficient classification capability.

5.4.4. State and Frame Extraction

Trace files commonly encompass all the graphics calls made by an application while it is running. Only a small part of this data is often necessary, making the bulk of the trace file largely superfluous. To make it easier to focus on a particular part of a long

Table 4. Items included in the OpenGL ES performance checklist.

Mipmap usage	Mipmap filtering reduces memory accesses and improves image quality, bilinear filtering is a cheap way to improve image quality on hardware engines.
Synchronous functions	Functions that cause the CPU to wait for the GPU may have a dramatic negative effect on performance.
Depth buffer clearing	Failing to clear the depth buffer may have a significant performance penalty on some architectures.
Vertex buffer object usage	Using vertex buffer objects reduces memory bus bandwidth utilization on some architectures.
Renderer version string differentiation	Test whether the OpenGL ES renderer version and extension strings are being examined for the presence of extensions providing better performance. For a software renderer, the complexity of graphics content should be scaled down.
Existing texture data modification	Modifying existing texture data is an expensive operation on most renderers.
Loading texture data during frame rendering	Generally texture data should be pre-loaded during the startup phase and only if needed during runtime.
Texture data compression	When supported by the hardware, texture data compression decreases memory usage and improves rendering performance.
Triangle strip geometry	Using triangle strips reduces the need to process the same vertices more than once and improve rendering performance for complex meshes.
Multisample usage	On hardware engines, multisampling improves image quality with only a small performance cost.

trace file, the analyzer can be used to extract a sequence of calls from a trace to form a new smaller trace.

Simply extracting the selected calls is often not enough, since the objective is usually to also preserve the rendering output, that is, the effect of the calls. For instance, the application might have loaded a number of textures during its initialization phase, and that texture data will also need to be resident when the extracted frame is played back. If the extracted frame, however, does not need these textures, they can be automatically discarded to reduce the size of the trace file.

Both the OpenGL ES and OpenVG design is based on the concept of a state machine; the graphics engine has an internal state that is modified using API calls. As the state is implicitly stored inside the graphics engine, the behavior of each API call depends

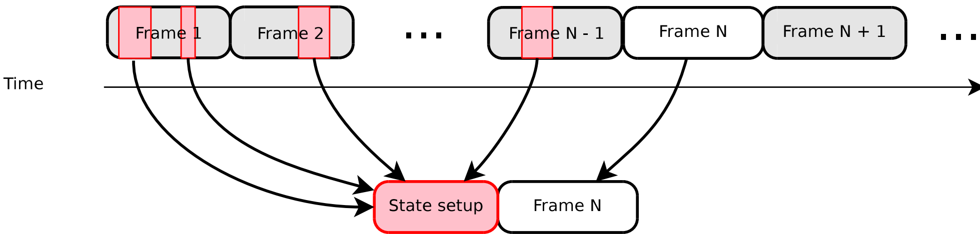

Figure 29. A single frame is extracted from a trace file. The Trace Analyzer uses state tracking to determine which preceding graphics commands are also needed to ensure that the rendering output of the extracted frame will be identical to the corresponding frame in the full trace. These extra commands are then assembled into a state setup sequence.

on the currently active state. Thus, it follows that the behavior of API calls depends on the API calls made prior to them.

The process of extracting a single frame along with the associated state setup API calls is illustrated in Figure 29. When a sequence of events is extracted from a trace file, the analyzer calculates the effective state at the extraction point. The API calls that have been used to prepare the state are then prepended to the extracted frame. This ensures that the rendering output of the extracted frames will be correct. Note that the state setup API calls may appear anywhere in the preceding trace section, and that only the required subset of the preceding calls are included. In other words, the analyzer strives to find the minimum number of calls needed that will result in the same effective state at the beginning of the extraction point.

The state computation and extraction algorithm builds on the state tracking system described in Section 5.2.5 in page 55. Recall, that the state tracker describes how API call parameters map to the branches of the internal API state tree. The state paths describing these relations are used by the extraction algorithm to collect the set of previous API calls that have affected the state. The algorithm is based on the observation that an API call is a prerequisite for a second API call if a state path associated with the first call is a prefix for any of the paths associated with the second call. For instance, as seen in Table 1 on page 57, the `glBindTexture` function is clearly a prerequisite for the `glTexParameteri` function, since the state paths of the former call appear at the start of the paths of the latter call.

Furthermore, an API call is said to be eclipsed or invalidated if all of its state paths can be matched to an identical path of a successive event. This happens, for example, when the rendering color is set twice in a row. As the second color definition overrides the first one, the former API call is eclipsed by the latter one and can be discarded.

The state extraction algorithm works by collecting every trace event prior to the extraction point to a list of effective state setting events. When an event is added to the list, all the events eclipsed by it are removed from the list. Similarly, when an event is removed from the list, all its prerequisite events are removed, provided that they are not prerequisites for any other events. This algorithm produces a minimal list of events

```
1  # Choose a reduction factor of 1/8ths
2  factor = 8
3  # Iterate over each event in the trace
4  for event in trace.events:
5    # Choose only texture uploading events,
6    # i.e., glTexImage2D, glTexSubImage2D, etc.
7    if event.name.endswith("Image2D"):
8      width   = event.values["width"]
9      height  = event.values["height"]
10     newSize = (max(1, width / factor), max(1, height / factor)),
11     # Replace the texture data with a scaled version
12     event.values["pixels"] = resample(
13         pixels  = event.values["pixels"],
14         size    = (width, height),
15         newSize = newSize,
16         type    = event.values["type"],
17         format  = event.values["format"])
18     event.values["width"]  = newSize[0]
19     event.values["height"] = newSize[1]
```

Figure 30. A simple script that automatically reduces the size of OpenGL ES textures to one eighth of the original. This is done by routing the pixel data of each texture upload function through a `resample` -function (not shown).

that, when executed, will reproduce the same state that was effective at the extraction point.

5.4.5. Scripting Interface

Due to the component's high level nature and open-ended requirements, the trace analyzer was written in the Python programming language. This choice allowed us to implement an extensible plug-in architecture as well as integrated scripting support in the form of user-supplied Python scripts. The scripting interface facilitates quick ad hoc trace analysis and processing without having to resort to creating a dedicated program for the task.

An example of an analyzer script is shown in Figure 30. The program reduces the size of each texture in an OpenGL ES trace file to 1/8th of the original. This is done by modifying the *width* and *height* parameters of each texture upload function found in the trace. The script could be used to test how varying the amount of texture traffic affects a program's performance.

```
1 > SELECT event.seq, event.name, event.duration FROM trace
2   WHERE event.duration > 10000;
3 event.seq | event.name     | event.duration
4 -------------------------------------------
5 985       | eglSwapBuffers | 15000
6 1255      | eglSwapBuffers | 11250
7 ...
```

Figure 31. A query that returns all graphics operations that ran longer than 10 milliseconds.

5.4.6. Trace Query Language

In the spirit of the relation debugger work by Duca et al. [30], the Trace Analyzer also offers a similar, albeit somewhat simpler, relational query language for extracting data from trace files. The syntax of the language is based on the de facto database standard SQL. A limitation with our implementation is that only the SELECT-statement is supported; existing trace file elements cannot be modified through the query language. If needed, this can be accomplished with the Python scripting system described in Section 5.4.5.

The workflow with our query language is the same as with SQL: the user submits a textual query expression to the analyzer, which in turn responds with a tabular list of rows of data matching that expression. An example query is shown in Figure 31. The query instructs the analyzer to look through all graphics operations in a trace file and return the ones that match the conditions specified in the WHERE-clause. In this case, the returned events will be those with a duration longer than 10 milliseconds.

A slightly more complex query is shown in Figure 32. When applied to an OpenVG trace file, this expression indicates the `vgDrawPath` commands that resulted in fewer than 16 pixels being drawn to the screen while the drawn path contained more than 10 segments. In other words, this query attempts to highlight complex OpenVG geometry being drawn at a small scale, where the highly detailed path coordinates are superfluous and may result in poor performance.

5.4.7. Exporting Traces as C Code

Although the Trace Player can be used to reproduce any required graphics call sequence stored in a trace file, it does have a number of limitations. Firstly, the player must be separately ported to every operating system on which it is used. Furthermore, it may have higher memory and storage space requirements than the original application, as the graphics commands need to be read and decoded from the trace file. Also, a trace file is of no use to a third party, unless the player is also delivered. These issues limit the utility of the Trace Player, especially in special environments such as prototype hardware and as a part of automated testing systems.

```
1 > SELECT event.seq, event.name FROM trace
2   WHERE event.sensors.average_path_size < 16 AND
3         event.state.path.segment_count > 10;
4 event.seq | event.name
5 ----------------------
6 34        | vgDrawPath
7 71        | vgDrawPath
8 ...
```

Figure 32. A query that highlights complex OpenVG paths drawn at a small scale. This is done by looking for paths that consist of more than 10 segments and produce fewer than 16 pixels when rasterized.

The Trace Analyzer provides a way to overcome these limitations by converting a trace file to an equivalent C source code file. Each graphics operation in the trace file is converted to a corresponding function call in the source code. When compiled and executed, the generated code reproduces the original trace sequence. As the code uses only standard C language constructs and the relevant graphics API functions, it can be run on a wide variety of platforms with relative ease. Since the code has a limited number of external dependencies, it may be given to a third party who is unable to run the original application due to licensing, platform, or other constraints.

Some special considerations need to be taken into account when generating C source code from a trace file. As it is not uncommon for a trace file to contain tens of thousands of graphics commands, the resulting C source code quickly reaches the limits of most compilers. In particular, the amount of code contained in a single function is often quite limited. We worked around this limitation by splitting up each frame of the graphics trace to a separate function and generating two utility functions for playing back one or all of the frames. However, some extremely large trace files failed to compile even with this modification due to the large amount of array data contained in them. The solution was to direct the array data into a separate assembly language source file, which was then linked together with the generated C code. As assembly language compilers are much simpler than C compilers, the large amount of array data did not pose any difficulty for them. Our system supports both the RVCT and GNU assemblers.

In addition to calling the correct functions, the generated C code also must ensure that proper data is stored at all times in the various data arrays appearing in the trace file. This is done by generating code that updates the contents of each array whenever the trace file indicates that the underlying data was changed. As the array data is stored directly in the source code, this method has very little performance overhead. An additional constraint for the generator to consider is that array data is polymorphic in the sense that a given array may store objects of any type during its lifetime. For example, the same memory block might be reused by an application to hold floating point vertex data at one point and byte-form texture data at another. The code generator manages this by creating a separate virtual array for each concrete type of an array.

The static typing of C places some requirements for the generated code. This is most evident in the fact that the types of function parameters must match those defined in the function prototype. To ensure this, the code generator replaces numeric constants with their symbolic equivalents. For added convenience, it also reconstructs bit fields with the original symbolic flags. For instance, instead of the numeric constant 16440, the user will see the expression `GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT` passed to the `glClear` function.

As discussed previously, trace files may also contain a number of platform-specific objects such as windows and bitmaps. In the file, these objects are described by their essential attributes, such as the on-screen coordinates in the case of a window. If such objects are encountered in the trace, the C code generator creates a function call to a user-supplied function. This function is responsible for creating an object instance of the appropriate type when given the attributes of that object. This scheme facilitates practical portability of trace files that contain platform-specific elements, since the user only needs to implement these object construction functions once for a given platform.

Figure 33 shows an example of C code generated from an OpenVG trace. The file begins with the declaration of the objects used in the trace. In this case, `CFbsBitmap` is a Symbian bitmap object, which will hold the final rendered image. Note that on a platform other than Symbian, this object can be implemented using the native bitmap equivalent of that platform; the only requirement is that the object is compatible with the used EGL implementation. The creation of the bitmap object is done through a user-supplied function on line 17. The `init` function constructs all platform-specific objects used in the trace.

The object definitions are followed by the data arrays containing the actual array data. Arrays and the data contained in them are declared separately, as a single array may hold multiple sets of data during its lifetime.

Finally, the `frame0` function holds the actual OpenVG code for the first frame of the trace file. The code uses the objects and arrays defined earlier in the file. Note how the identifier constants have been given human-readable names to make their meaning apparent.

```
1  /* Objects */
2  static CFbsBitmap* cfbsbitmap_35fe3f48;
3  static VGPaint vgpaint_2;
4  ...
5  /* 15 arrays */
6  static VGfloat vgfloat_array0[5];
7  static VGfloat vgfloat_array1[15];
8  ...
9  /* Array data */
10 static const VGfloat arrayData0[9] = {
11     5.324997f, 0.000000f, 0.000000f, 0.000000f, -5.324997f,
12     0.000000f, 0.249924f, 445.000000f, 1.000000f
13 };
14 ...
15 static void init(void* context)
16 {
17     /* CFbsBitmap attributes: width, height, mode */
18     cfbsbitmap_35fe3f48 = createCFbsBitmap(context, 320, 445, 0);
19     ...
20 }
21 ...
22 static void frame0(void* context)
23 {
24     vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_BETTER);
25     vgpaint_2 = vgCreatePaint();
26     LOAD_ARRAY(vgfloat_array14, arrayData10, 4);
27     vgSetfv(VG_CLEAR_COLOR, 4, vgfloat_array14);
28     vgSetParameterfv(vgpaint_2, VG_PAINT_LINEAR_GRADIENT,
29                      4, vgfloat_array10);
30     vgClear(0, 0, 320, 445);
31     vgAppendPathData(vgpath_5, 343, vgubyte_array4, int_array3);
32     vgDrawPath(vgpath_5, 2);
33     ...
34 }
```

Figure 33. An abbreviated C source file generated from an OpenVG trace file.

6. USE CASE DEMONSTRATION

The purpose of this chapter is to demonstrate how each of the core use cases originally presented in Chapter 4.3 on page 39 is performed using the Graphics Quality Analysis Toolkit implementation.

6.1. Unsatisfactory Application Performance

The first use case illustrates how the Graphics Quality Analysis Toolkit is used to look for causes behind the poor performance of a graphics application. The applications used in this example are an OpenGL ES-based image gallery application and an OpenVG-based SVG animation player. The gallery application allows the user to browse digital photographs, while the SVG animation player reproduces animated SVG files. Both applications are run on a Nokia N95 smartphone with S60 3rd Edition Feature Pack 1 system software. This device features a 24 bit, 240 by 320 pixel display with a hardware accelerator for OpenGL ES 1.1 and a software implementation of OpenVG 1.0.

The process begins with the generation of suitable OpenGL ES and OpenVG tracers for the targeted Symbian platform. These tracers are then compiled and integrated into a regular system software image for the N95. Installing this image on the device yields an otherwise normal smartphone, except that selected programs using OpenGL ES or OpenVG have their graphics commands silently saved to a trace file. The image gallery application and the SVG player are then installed and run on the device, resulting in one trace file for each application.

Both of the trace files are then loaded into the Trace Analyzer, which is used to produce the initial high-level statistics shown in Table 5. Both traces are also converted into a plain text format for a quick reference as to which graphics operation sequences each application was using.

It was immediately apparent that the performance of the image gallery application is not at a sufficient level: the application barely renders one frame per second, which is far below the interactive limit of 10 frames per second. Further analysis of the application's graphics trace reveals a number of possible causes for this deficiency. A large portion of the graphics calls made by the application are state setup calls that modify the rendering library parameters. This is not uncommon in itself, as the majority of the available OpenGL ES API calls are for state manipulation. In this case, however, most of the state setup commands are redundant. That is, they are used to set the same state over and over again and thus have no effect. A common call pattern seen in the application is shown in Figure 34. The redundant call setup sequence on lines 1 through 7 often repeats hundreds of times before any actual rendering takes place. With the Trace Analyzer's state tracking functionality it is determined that over 70% of the image gallery application's graphics calls were redundant. This is in sharp contrast with other examined graphics applications, where the same figure is usually below 5%. Although graphics drivers are often optimized in a way that most of such redundant state toggling is culled from the stream of commands sent to the graphics accelerator, this high amount of essentially useless graphics commands is bound to have an impact on performance.

Table 5. An overview of the captured trace files. The bar charts indicate the number of frames drawn per second during the application's execution.

	Image gallery	SVG player
Trace file size	9.2 MB	13.2 MB
Duration	28 s	39 s
Graphics operations	19 482	234 114
Frames	32	548
Frames / second	1.14	14.05

A second observation made from the image gallery trace is the large quantity of texture data uploaded by the application. As shown in Figure 35, a significant percentage of rendered frames involve a sizable amount of texture data transfer up to a total amount of 12.5 megabytes. The texture transfer rate is dependent on the graphical content of the animation at each point in time. While this is somewhat expected behavior for an application dealing with image-based animation, a closer examination reveals that a number of textures used by the application could be eliminated without changing the graphical end result. On a number of occasions the application uploads a texture only to moments later replace the contents of that same texture with a new one without using the original texture image at all. This indicates a possible flaw in the application's texture management logic. In addition to these unused textures, the Trace Analyzer highlights a number of cases where the application modifies the contents of a texture immediately after using it. Such action may inhibit rendering parallelism, since the graphics library must wait until the graphics accelerator is finished with the texture or make a new copy before changing its contents. Uploading textures is often an expensive operation, and especially so in the case of the Nokia N95, since the texture data needs to be converted into a specific format used by the graphics accelerator. Due to this it is often beneficial to limit the number of textures used by an application.

A third issue uncovered from the image gallery trace is the fact that on two occasions, the application destroys its main rendering surface and recreates an identical surface immediately afterwards. Apparently this is done as a response to the application window moving to a different region on the screen. Such explicit processing is unnecessary, since the EGL windowing system interface manages the interaction between the rendering surface and the underlying native window automatically.

While the performance of the SVG player application is considerably better than that of the image gallery application, it also suffers from a number of inefficiencies in its implementation. As shown in Figure 36, this application also utilizes a considerable amount of pixel image data. While SVG files generally do not employ pixel images, the animation in question uses them as an optimization: complex vector shapes in the animation are replaced with static image imposters. This is often a good way to improve performance, especially on software rasterizers, since they typically draw image bitmaps faster than very complex vector shapes. In this case, however, this optimization uncovers a rather large inefficiency in the SVG player: instead of uploading the imposter images once and then reusing them repeatedly, the player uploads the same image data over and over again. This is quickly confirmed by extracting all the used OpenVG images into regular image files with the analyzer. As with textures in

```
1  glLoadIdentity()
2  glTranslatex(x=0, y=0, z=-524288)
3  glScissor(x=0, y=0, width=240, height=320)
4  glScissor(x=2, y=264, width=40, height=30)
5  glTranslatex(x=1280, y=10496, z=0)
6  glScalex(x=15805, y=15805, z=0)
7  glColor4x(red=65536, green=65536, blue=65536, alpha=0)
8  glLoadIdentity()
9  glTranslatex(x=0, y=0, z=-524288)
10 ...
11 glDrawElements(mode=GL_TRIANGLES, count=6, ...)
```

Figure 34. Redundant state setup in the image gallery application consisted of hundreds of repetitions of lines 1 through 7. Each `glLoadIdentity` call discards the work done by the previous matrix manipulation commands. Similarly, calling `glScissor` multiple times without rendering anything in between serves no practical purpose.

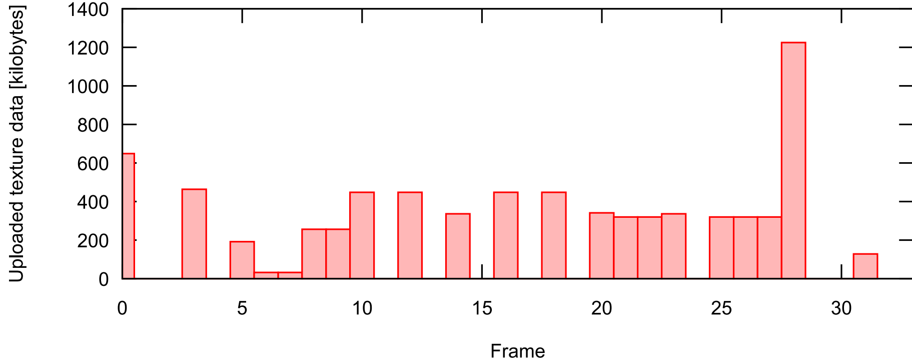

Figure 35. Texture data uploaded by the image gallery application as a function of time.

OpenGL ES, this image traffic can be a very large burden depending on the underlying OpenVG engine.

The trace file also reveals that in addition to images, the SVG player also destroys and recreates every other used OpenVG object at each drawing operation. The entire animated scene is built from the ground up for each frame, even if a small part had changed in comparison to the previous frame. This strategy is very efficient in terms of memory usage, since each OpenVG objects exists in memory only while it is being used. A major downside is that the OpenVG engine cannot apply any extensive optimization strategies, since its knowledge of the graphics scene is effectively reset after each drawing operation.

In both cases, the major contributors to the observed performance problems were found to reside in application code. The performance observations discussed above offered valuable input to the application developers in further performance optimization work. Undoubtedly the same performance issues could have also been detected with

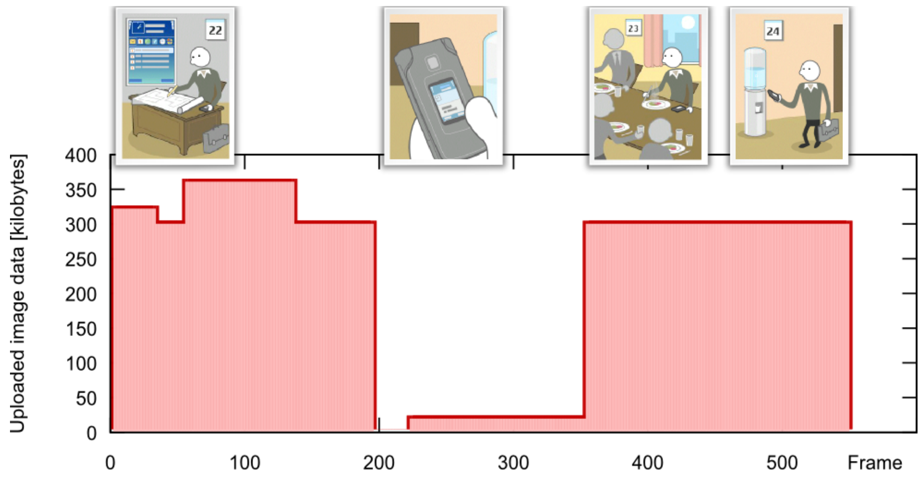

Figure 36. Image data uploaded by the SVG player application as a function of time. Frame captures from the animation are shown above the graph.

a careful review of the respective source code of both applications, but the key benefit delivered by the Graphics Quality Analysis Toolkit here is the greatly reduced effort, higher level of automation and easier data extraction when compared to the source code review.

6.2. Visual Error in Application

The second use case concerns an application with a clearly visible visual rendering error. Figure 37 shows an example where an OpenGL ES application displays invalid graphics with missing triangles and generally distorted geometry. Here the Trace Player is first used to replay the trace file from the application on a reference engine. This produces a correct visual output, which indicates that the error is caused by the graphics engine.

In this use case demonstration we use two applications: an OpenGL ES performance benchmark and an OpenVG SVG image viewer. The used hardware configuration is identical to that of the first use case. The process of tracing the affected applications is also performed as described before.

The OpenGL ES benchmark application works by rendering a number of different animated scenes and measuring how quickly each scene is drawn by the graphics library. The visual error in this case is a very apparent one: one whole scene from the benchmark is missing and nothing but a blank screen is displayed. The application functioned properly on other devices, leading to the conclusion that the error is caused by a defect in the graphics engine of this particular device.

The benchmark graphics trace is loaded into the Trace Analyzer and the Trace Player is used to locate the time range of the missing scene. Subsequently one graphics frame is extracted from the missing scene. The trace player is then used to verify that the extracted frame still reproduces the error on the hardware device. This confirmation allows the analysis to focus on the relatively simple single frame containing only 420 graphics commands instead of the full trace of 39 994 commands.

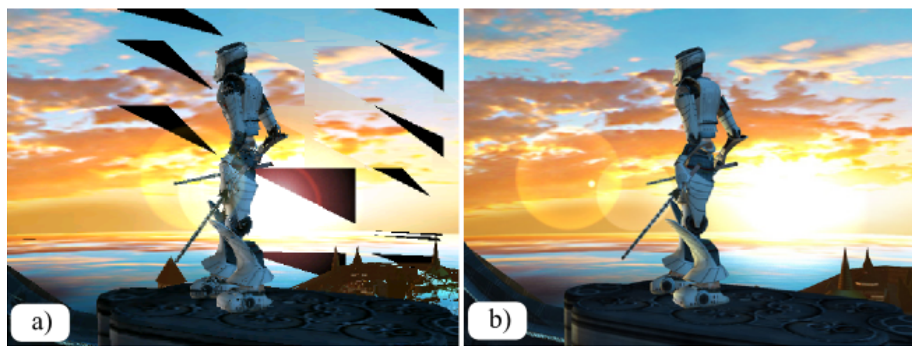

Figure 37. a) An OpenGL ES application is exhibiting a visual error. b) The application trace produces a correct visual output on a reference graphics engine, indicating an error in the first engine.

As described in Section 5.4.3 on page 70, the analyzer includes an automated checklist utility that looks for predefined call patterns in trace files. In this case, the checklist indicates that the benchmark application is using the `glColorMask` function. This function is used to restrict rendering output to a subset of the red, green, blue, and alpha color channels. Normally this would not have been a problem, but in this case it was known that the OpenGL ES hardware of the Nokia N95 may not perform color masked rendering correctly under all circumstances. The suspicion is quickly verified by disabling the color mask command in the trace file and replaying the trace on the device. With the modified trace, the missing scene is restored, albeit with slightly incorrect rendering due to the removed color mask setting.

With the problem isolated, the next step is to prepare an isolated test case for reproducing the issue. This is done by using the Trace Analyzer to automatically convert the extracted graphics frame into equivalent platform-independent C source code. This source code is then sent to the graphics hardware vendor, who is able to use it to fix the underlying defect in the graphics driver. The graphics vendor would have been unable to use the original benchmark application directly, since it is not compatible with their development platform.

The visual error exhibited by the SVG image viewer application is more subtle: a complex SVG drawing was otherwise fully rendered, except for a small missing geometric shape. As the graphics trace taken from the application indicates no obvious faults, the next step is to test whether the absent shape is caused by an error in the OpenVG renderer or the SVG viewer application itself. The trace file is replayed on a number of other OpenVG renderers, including the OpenVG reference engine. In each case, the output is essentially identical to that of the original rendering: the missing shape is also missing with the other engines. Based on this finding, the defect can be classified as being caused by the SVG viewer application. This conclusion is later confirmed in further analysis by the application developers.

In both cases, the Graphics Quality Analysis Toolkit enables efficient means for pinpointing the root cause behind a visual error. An indispensable advantage provided by the toolkit here is the ability to effortlessly transfer graphics content from one platform to another; the development platforms used by graphics hardware vendors and

Table 6. An overview of the captured trace files. The bar charts indicate the number of frames drawn per second during the application's execution.

	Application menu	GPS map navigator
Trace file size	22.0 MB	6.1 MB
Duration	45 s	1 min 4 s
Graphics operations	449 453	40 870
Frames	277	446
Frames / second	14.58	13.46

reference engines are seldom compatible with actual production systems used by the applications.

6.3. Application Quality Analysis

The third use case aims to assess the implementation quality of a vector graphics application. This situation differs from the first use case in that there is nothing immediately wrong with the graphical performance or functionality of the application. The objective is to perform a preemptive quality analysis in order to uncover issues that may not be evident through only casual use of the application.

The hardware used in this demonstration is the same Nokia N95 as in the earlier cases. Both of the traced applications, an application launcher menu and a global positioning system (GPS) map navigator, employ OpenGL ES graphics. The application launcher presents the user with a browsable menu of application icons, while the GPS map navigator displays the current position of the device on a digital street map. An overview of the captured traces is shown in Table 6.

The Trace Analyzer was first used to provide an overview of both traces, shown in Table 6. The average frames per second figures for both traces were generally within the interactive range, and the applications appeared responsive to the user.

The Trace Analyzer is then used to generate a more detailed view of a number of graphics content statistics from the menu application. These statistics are illustrated in Figure 38. The frames per second figure is particularly telling: although on average, the application maintains an interactive display refresh rate, some intermittent processing causes severe drops in the graphics refresh rate. This is perceived by the user as visible discontinuities in otherwise smooth animation. The diagram also reveals that the graphics content complexity generally tends to increase around these performance drops, suggesting that the reduced performance is related to something the application is rendering at those particular moments. Especially the amount of uploaded texture data correlates strongly with the low refresh rate. The number of graphics primitives rendered by the application is well within the capabilities of the device at all times.

These findings warrant further investigation into the moments of low performance of the menu application. The Trace Analyzer is used to extract a number of frames around these points in time. The traces for these frame sequences indicate several deficiencies: firstly, graphics are being rendered to a texture, but not through the standard OpenGL ES render to texture mechanism, but instead via explicitly reading back pixel and sub-

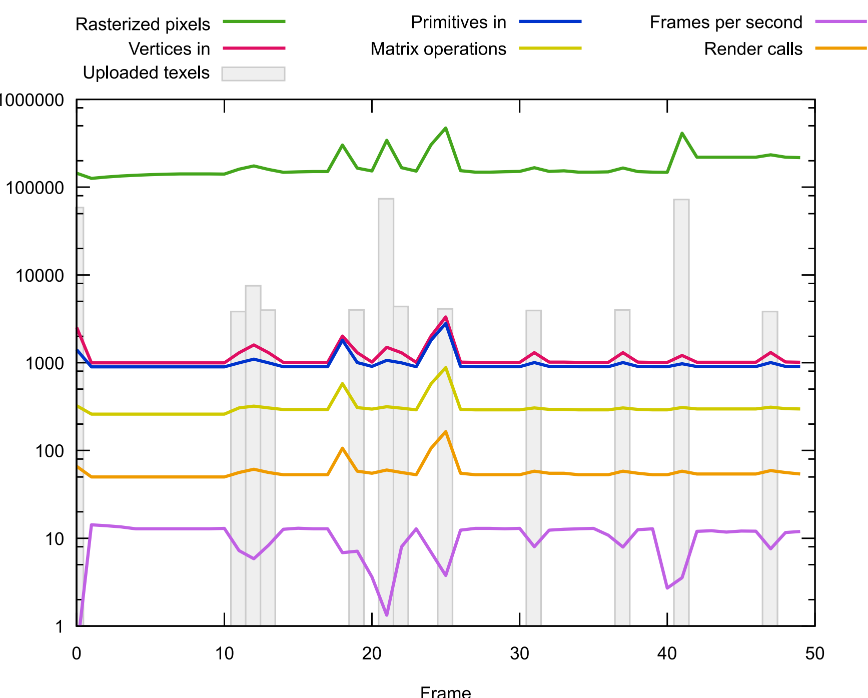

Figure 38. Graphics content statistics from the OpenGL ES-based menu application indicate that its performance (purple line) varies greatly over time. The low performance seems to correlate with the periodic texture data uploads (gray bars) and increased rendering complexity (green line), suggesting that the workload presented to the graphics engine during these moments exceeds the capabilities of the hardware. Note the logarithmic vertical axis.

sequently uploading that data into the targeted texture. Secondly, the `glColorMask` is being used, triggering an expensive emulation operation in the graphics driver due to hardware limitations. Finally, the application is modifying subregions of existing textures, inhibiting rendering parallelism. Furthermore, the application issues a large number of redundant graphics state modification commands during each frame.

In addition to these issues, the trace analysis also uncovers other problems relevant to the mobile application platform. A major issue is that the menu application does not pause its display refresh cycle at any point during its operation. The graphics are updated continuously even if there are no animated elements on the screen. Similarly, the application does not cease rendering when another application is launched from the menu and the menu itself becomes a hidden background task. This kind of behavior is especially detrimental in terms of battery life, since the graphics accelerator cannot be powered down while an application is issuing commands to it.

The GPS map navigator trace does not at first indicate any major deficiencies in the application. One exception is the fact that the graphics are being rendered into a pbuffer surface instead of a more efficient window surface. Since a pbuffer surface is not double buffered, the graphics accelerator cannot work on two sequential frames in parallel, degrading rendering performance by a factor of two or more.

More detailed analysis of the trace, however, shows that the application has two completely different modes of working in terms of graphics rendering. The first mode is used when showing an orbital view of the planet Earth, and the second scheme is activated when the camera zooms into a street level view. While dedicated graphics rendering techniques are often beneficial with large data sets, the implementation in this case is troublesome: the street view is being rendered with an entirely custom software graphics engine embedded inside the application. This meant that the hardware accelerated OpenGL ES engine is being completely eschewed in favor of the proprietary engine whenever a street level map is displayed. Since the street level view is quite probably the most common mode of operation for the application, this design choice is responsible for much increased processing power and battery charge consumption when compared to a hardware engine. This solution is also likely to cause significant performance problems on a device with a higher resolution display. As the software engine is completely encapsulated inside the navigator application, tracing its operation is not possible.

The Graphics Quality Analysis Toolkit was used here to conduct a precautionary analysis of applications that did not appear to have major performance or visual problems. The findings encountered in this case, especially with regard to battery usage, warranted for a more complete review into the design of the applications in question.

6.4. Graphics Engine Benchmarking

In the fourth use case, the objective was to evaluate a new graphics engine with graphics content extracted from existing applications. The output of this process is an estimate of how well a particular software or hardware graphics platform performs with real-world applications.

The process begins with the selection of three applications to be used as sources for graphics content: an OpenGL ES-based application launcher menu, an OpenVG system icon loader and a Java Mobile 3D Graphics (M3G) benchmark. The fact that M3G is commonly implemented on top of OpenGL ES allows us to utilize the tracer also in this case. Each application is traced and one frame is extracted from each trace. The frames are chosen manually to represent typical graphics content of each respective application as closely as possible. The frames are then compiled into a number of benchmarks by repeating the frame multiple times. When run, these benchmarks effectively indicate the steady state performance of each frame.

In the interests of reliable benchmarking, several measurements are conducted to test the relative performance of the Trace Player and C code generated from the traces. Three different animated graphics applications are chosen for the experiment: two C-based OpenGL ES applications with simple and complex geometry respectively and an application with medium geometric complexity implemented on top of Java M3G. An overview of the applications is shown in Table 7. The complexity estimates of the applications are based on the average number of triangles drawn by the application during each frame. The simple and complex geometry benchmarks are conducted on a Nokia N800 [13] Internet Tablet running Linux, while the M3G benchmark is performed on a Nokia N95 smartphone.

Table 7. Properties of the trace files used to measure trace playback performance.

	Simple geom.	Complex geom.	M3G app.
Trace file size	1.9 MB	29.9 MB	1.6 MB
Graphics operations	23 264	124 139	31 236
Frames	52	51	206
Triangles / frame	2 239	36 934	19 018

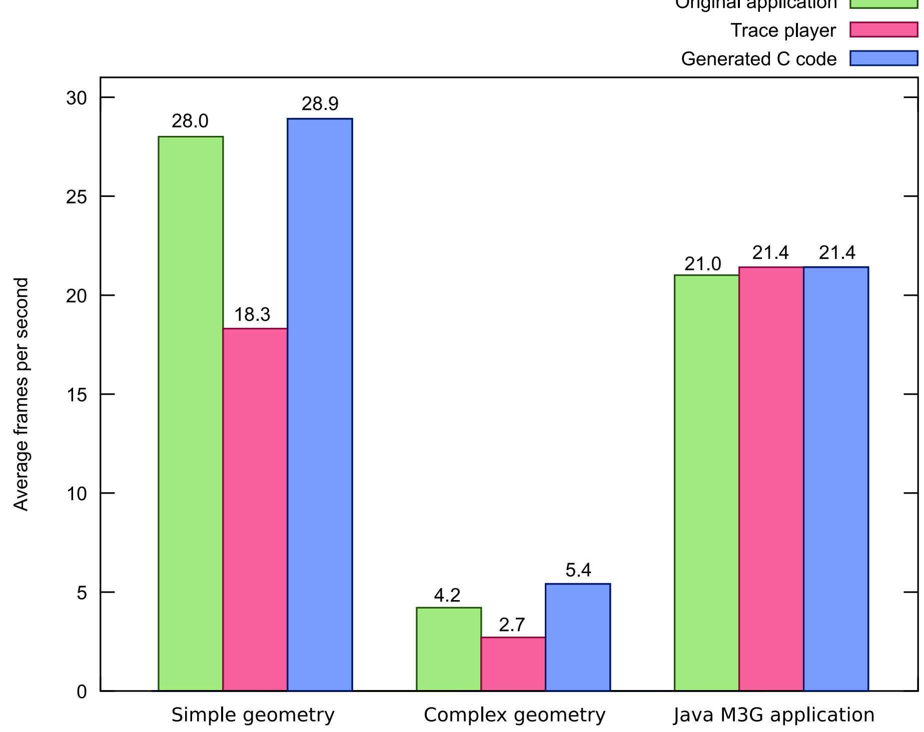

Figure 39. Performance comparison between the original application, the Trace Player and C code generated from the trace. The performance of the C code matches that of the original application.

The results of the experiment are shown in Figure 39. According to the hypothesis, the Trace Player may have some overhead compared to the original application, while the generated C code will have less overhead. The results appear to confirm these assumptions: the Trace Player does not achieve the performance of the original application, while the generated C code even surpasses the performance of the original application. The penalty of the Trace Player is especially apparent in the case of complex geometry, where large amounts of array data need to be decoded from the trace file. An exception is the M3G case, where all three benchmark variants scored nearly equally. While the application in question does have moderately complex geometry, the rendering loop does little more than submit that geometry to be rasterized. Because of this, the differences between the different benchmark programs are muted, as the majority of the work is done inside the graphics engine instead of the application logic.

In this use case, we have demonstrated how the Graphics Quality Analysis Toolkit was used to evaluate the performance of given graphics content in a wide variety of graphics engines and platforms. With the Trace Player and the C code generator, benchmarking can be performed with a minimum effort on any suitable platform irrespective of which platform the original application was using. Furthermore, no application source code changes or access was required, enabling the use of practically any graphics application for benchmarking purposes.

6.5. Graphics Content Analysis

The fifth and final use case aims to demonstrate a practical process for obtaining accurate in-depth content features from OpenGL ES and OpenVG applications. Previously this kind of data had to be acquired manually through debugging or modifications to the application or the graphics engine.

Three applications are chosen for this use case demonstration: an OpenGL ES-based application launcher menu, an SVG-Tiny image loader built on OpenVG, and an OpenGL ES graphics hardware marketing demonstration. The application launcher and the SVG image loader represent regular production software, while the marketing demonstration was created by graphics experts for the express purpose of exercising the capabilities of the underlying graphics engine. All applications are executed on a Nokia N95, except for the graphics demonstration software, which is run on a Nokia N800.

As before, the applications are traced individually and the traces are loaded into the Trace Analyzer. Each trace is then replayed with the Trace Player, which calculates the content statistics using the instrumented OpenGL ES and OpenVG engines. Some of the obtained statistics for the OpenGL ES applications are shown in Table 8 and for the OpenVG application in Table 9.

The first section of each table lists aggregate content features that span the entire respective trace. The remaining rows show a statistical breakdown of measurements from each rendered frame. For instance, the *render calls* section of the first table indicates that the application launcher submits an average of 90 drawing commands for each frame. In the case of the SVG image loader, each consecutive frame corresponds to a different displayed SVG file. The inline bar charts beside the figures represent the distribution histogram for the respective value.

While some of the reported statistics are common for both APIs, most of them only apply to a particular API. We first discuss the shared attributes, followed by the OpenGL ES and OpenVG specific statistics. The number of *graphics operations* indicates how many API calls are made in total during a frame. The number of *render calls* is a subset of this figure: it only includes the API commands that produce visible pixels; the remaining API calls are state manipulation commands. *Matrix operations* refer to API calls that modify the various matrices defined in both APIs. The figure *rasterized pixels* indicates how many pixels the application draws in relation to the size of the screen. Finally, the *texel and pixel uploads* indicate the amount of OpenGL ES texture data and OpenVG image data uploaded by the application during each frame.

The OpenGL ES-specific figures begin with the number of *primitives* or triangles submitted to the graphics engine. This figure is a very commonly used estimate of

Table 8. Calculated OpenGL ES content statistics.

		Application launcher	Marketing demonstration
Trace file size		13.1 MB	29.9 MB
Total graphics operations		297 294	124 139
Surface size		240 by 320	640 by 320
Frames		165	51
Total render calls		41 257	5087
Texture data		1.5 MB	12.9 MB
Graphics operations	min	1 421	115
	max	4 664	4 249
	mean	2 169	2 433
Render calls	min	88	55
	max	192	175
	mean	90	109
Matrix operations	min	23	279
	max	115	880
	mean	50	550
Texel uploads	min	0	0
	max	107 633	3 294 893
	mean	2 864	63 363
Primitives (triangles)	min	132	9 125
	max	444	98 771
	mean	192	36 934
Backface culled primitives	min	0	1 795
	max	0	18 177
	mean	0	6 221
Frustum culled primitives	min	0	3 142
	max	30	66 735
	mean	0.9	22 969
Vertices	min	264	27 375
	max	1 144	296 313
	mean	422	110 801
Lit vertices	min	0	22 76
	max	0	25 451
	mean	0	9 887
Transformed vertices	min	264	10 937
	max	888	168 051
	mean	383	59 789
Rasterized pixels (% of surface size)	min	31 %	105 %
	max	315 %	292 %
	mean	140 %	147 %
Triangle size (pixels)	min	588.5	3.0
	max	2 001.8	110.2
	mean	1 046.0	17.7

Table 9. Calculated OpenVG content statistics.

			SVG image loader
Trace file size			1.7 MB
Total graphics operations			28 276
Frames			162
Total render calls			1 581
Graphics operations	min		19
	max		830
	mean		173
Render calls	min		1
	max		51
	mean		9.7
Matrix operations	min		0
	max		68
	mean		16
Created objects	min		0
	max		15
	mean		3.0
Rasterized pixels (% of surface size)	min		0 %
	max		800 %
	mean		136 %
Path size (pixels)	min		0.5
	max		38 903
	mean		1 476
Path segments	min		0
	max		335
	mean		74
Tessellated edges per path	min		0
	max		111
	mean		23.9
Pixel uploads	min		0
	max		0
	mean		0
Gradient stops	min		0
	max		44
	mean		9.1

content complexity, since the number of triangles drawn has strong implications to the workload of the graphics engine. Since it is not uncommon for triangles to be backfacing or outside the graphics viewport, that is, invisible, engines commonly are able to discard such primitives at an early phase in the graphics pipeline. The number of *backface-culled primitives* and *frustum-culled primitives* describe the amount of triangles rejected in this fashion. The *vertex count* shows the number of three-dimensional coordinates used to define the drawn triangles. Although a normal triangle is made of three vertices, it is possible to define multiple triangles with fewer than three vertices per triangle with constructs such as triangle strips. This is why the number of vertices listed in the table is not exactly three times the number of drawn primitives. This technique helps to limit the number of *lit and transformed vertices*, which refer to the subset of vertices which underwent lighting and coordinate transformation calculations. Finally, the *triangle size* describes the number of pixels produced by each rasterized triangle.

The OpenVG-specific statistics begin with the number of *created objects*, such as paths, paints, and images. This is followed by the *path size*, which tells the number of pixels produced when a path was rasterized, not unlike the triangle size for OpenGL ES. The number of *path segments* shows how many segments were used to define each drawn path. The number of *tessellated edges per path* is a similar figure, but with a more direct relation to the graphics engine workload. It tells how many polygon edges the graphics engine needs to represent an infinitely smooth drawn path as discrete pixels. The more edges a path generates, the more complex the rasterization process is for the graphics engine. Finally, the *gradient stops* figure indicates how many different color values are used to define gradient paints.

The measurements listed in the tables warrant some observations of the traced applications. The application launcher and the marketing demonstration submit comparable levels of graphics operations, but the latter uses significantly more complex geometry. In other words, the application launcher employs a relatively large amount of API calls to render comparatively few triangles. An optimization possibility would be to render multiple similar objects at once.

The differences in content complexity are also apparent in the average size of the rasterized primitives: the triangles drawn by the marketing demonstration are tiny compared to the application launcher. Based on the large number of culled primitives, the marketing demonstration seems to rely on the graphics engine to efficiently discard invisible geometry. The application launcher represents the opposite extreme, mainly submitting only completely visible geometry. The number of rasterized pixels was often below 100 %, suggesting that the application goes further by only drawing the subregion of the screen that needed to be updated. Lighting shows a similar divide: the marketing demonstration makes use of OpenGL ES lighting, while the application launcher employs no lighting effects at all.

While both applications apply textures to the drawn triangles, their methods of defining the texture data are different. The marketing demonstration uploads all texture data with a single step at the start of the animation. This ensures a smooth frame rate at the expense of memory usage, since all required texture images are kept in graphics memory throughout the whole animation. In contrast, the application launcher uploads textures on demand, striving to minimize the size of the texture memory working

set. Both approaches are valid, but may result in wildly different performance figures depending on the underlying graphics engine.

The SVG image loader uses a relatively low number of OpenVG objects to render to the loaded images. The trace indicated that it reuses the same objects by replacing their previous contents with new data on demand. This usage pattern may inhibit full rendering parallelism on hardware engines. The rasterized paths, however, generate a large amount of pixels on average, indicating that the vector images do not have extraneously fine details. On average, the number of rasterized pixels is also moderate in relation to the output surface size, a sign that the images do not have many overlapping hidden regions. No pixel images are also used, meaning that the source images are fully vectorized.

With this use case, we have demonstrated a process for obtaining detailed content features from graphics applications with the Graphics Quality Analysis Toolkit. This method is both practical and straightforward, since it requires no modification to the investigated applications or system graphics engines and can be extended to provide additional statistics with moderate work. The data obtained in this manner may serve as a guide to further application or graphics engine optimization work and research into graphics workload estimation.

This final use case concludes the use case demonstration chapter. We now move to discuss the findings presented in this work.

7. DISCUSSION

The Graphics Quality Analysis Toolkit was born out of necessity for practical and efficient means to manage an ever-increasing amount of quality issues in mobile vector graphics applications. We based the design of the toolkit on well-established previous research in the field. The fundamental idea of capturing graphics commands into a trace file was directly inspired by Dunwoody & Linton [22] and other pioneers. The inherent advantage of not requiring any instrumentation of the application code or the graphics engine was an important reason for choosing their method as the basis for our work. Our approach, however, differed in that we kept the graphics command abstraction level at the level of individual API calls instead of specifying a higher level intermediate language. This ensured that we could capture the exact behavior of the traced application as closely as possible.

Chromium [23], the extensible OpenGL stream processing system, has a very similar architecture to the tracer in our system, and thus could have served as a basis for our implementation. Instead, we chose to implement a custom tracer through code generation. This enabled us to support nearly any C-based API, such as OpenGL ES and OpenVG, while Chromium would have directly offered support only for OpenGL. Additionally, our platform portability and performance requirements on embedded systems also necessitated a customized, more focused solution.

State tracking also plays a very central role in our work. It enables many of the key use cases of the Tracer and the Trace Analyzer, such as extracting frame sequences from longer traces and pinpointing quality problems in traces. Since a requirement for the toolkit is to be graphics API-neutral, the same principle also applied to the state tracker. This lead us to develop a generic tree data structure suitable for describing the state of EGL, OpenGL ES, OpenVG, and other similar APIs. In our experience, this method greatly reduced the amount of work for adding new graphics APIs to our system in comparison to hand-written state tracking code as used by Buck et al. [26], Chromium, and others. Our method also ensures that the traced graphics calls are not modified in any substantial way when played back in comparison to the original application. This was important, since our system was designed for reproducible application debugging and any unintended modification of the graphics command stream might change the graphics engine behavior substantially.

The design of the Trace Analyzer was driven by the core use cases for the system. Previous research into graphics workload estimation and characterization was used as a basis for the content features calculated from graphics traces. We met our objective of building a system that provided the features most commonly used for workload estimation. While previous research only covered 3D graphics, our toolkit also provides content features for 2D vector content.

In the domain of graphical debugging, our toolkit was inspired by related work such as NVIDIA PerfHUD [27], GDEBugger [28], GLIntercept [29], and the relational graphics debugger by Duca et al. [30]. All of these solutions, however, were based on live interaction with the debugged application. Our approach was to instead focus completely on offline trace analysis. In the context of embedded applications, offline analysis was essential since the target device often lacked the necessary processing power and usability for interactive graphical debugging.

Working with the trace file rather than the live application also had the added benefit of providing completely repeatable graphics sequences; in effect, it separated the examined quality issue from the application. Offline analysis also facilitated remote debugging, in which the execution environment and the application can be physically separate from the analysis environment. Our system also added the possibility of transforming the trace file into different formats. In our experience, the ability to generate equivalent C code from a trace file was a powerful tool in debugging and regression testing.

As with all software, some unanticipated challenges were met during the development of the Graphics Quality Analysis Toolkit. We found that even simple graphics applications could submit thousands of graphics operations in very short time periods. This placed strong scalability requirements to the Tracer and the Trace Analyzer to ensure applicability to production software. Another issue was that while the binary trace file format remained stable, a number of incompatible modifications had to be made to the format at logical level as API coverage was refined. The effect of this was that old trace files had to be manually translated into the new system to remain compatible. Our design also called for the use of C header files to define the graphics API functions. In retrospect, a simpler syntax would have been more viable due to the complexity in implementing a sufficiently robust C parser.

We began this work with an overview of mobile vector graphics and the effects of quality issues in graphics applications. We then defined a classification system for these quality problems based on the dominant cause of the issue. It quickly became apparent, that while the classification system was an effective tool, it required a comprehensive supporting toolchain for practical application. This was the driving motivation for creating The Graphics Quality Analysis Toolkit: it enabled in-depth examination of any OpenGL ES and OpenVG applications, providing the necessary information for reliable quality problem classification.

To date, the Graphics Quality Analysis Toolkit has been successfully applied to solving numerous quality issues in production software in the Display & Graphics Software group. The toolkit has helped to significantly decrease the amount of time spent in debugging common graphical errors and performance problems such as incorrect rendering and application crashes. It has also enabled new forms of graphics engine benchmarking, such as using the traced content of existing system applications for benchmarking, which previously required an impractical work effort. The toolkit has also offered a new level of insight into the graphics content of mobile applications, providing for valuable input into system design and graphics engine optimization work. We are therefore confident, that the graphics trace architecture and the offline analysis approach is a realistic and powerful approach to improving mobile vector graphics quality.

7.1. Future Work

A natural continuation to the work presented in this thesis is to follow the development of the graphics APIs themselves. The introduction of programmable shaders to embedded systems by OpenGL ES 2.0 [56] will undoubtedly bring forth new challenges in vector graphics quality. While basic tracing and replaying of OpenGL ES 2.0 graph-

ics content is possible with minor additions to our system, the workload presented to a shader programmable graphics engine is closely tied to the complexity of the used shader program. This calls for radically enhanced complexity and workload estimation methods than what are valid for the fixed function pipelines of OpenGL ES 1.x and OpenVG 1.x. We feel that the architecture of our toolkit would offer a practical environment for OpenGL ES 2.0 software analysis with the addition of such advanced content features.

Our implementation of the Trace Analyzer was based on a command line user interface. It would be advantageous to explore graphical alternatives, as the data contained in trace files is very graphical in nature.

The existing content features provided by the toolkit open the possibility for further research into clustering applications based on graphics complexity and creating synthetic benchmarks based on real application content. Given a big enough sample set, application clustering could enable classification into performance classes, providing a way to roughly estimate application performance on a new graphics architecture. The benefit of synthetic benchmarks would be to overcome the limitations of using static traced application content for benchmarking. While benchmarks created from traces are representative of the application in question, they are very hard to parameterize in terms of content complexity. It could be advantageous to explore the performance response of particular graphics content while, for instance, varying the complexity and other properties of the drawn meshes. Currently, such benchmarks are being authored by hand, creating a possible disconnection between real applications and the benchmark content.

8. CONCLUSION

The aim of this thesis was to define a practical process and supporting toolset for recognizing, analyzing and solving quality issues encountered in mobile vector graphics applications. Our main focus was on OpenGL ES 1.1 and OpenVG 1.0 applications running on Symbian OS smartphones. We began with an overview of the types of quality problems common in this domain and devised a classification system based on the dominant cause behind the issue:

1. Dominant cause in API usage patterns
2. Dominant cause in graphics content complexity
3. Dominant cause in graphics engine
4. Dominant cause unrelated to graphics

We then examined each class of problems in detail, focusing on what information was required to recognize issues belonging to that class. Based on this analysis, we concluded that the classification process required in-depth knowledge of the graphical behavior of the examined application. Acquiring such information was deemed impractical without a comprehensive supporting toolset. Existing graphics debugging solutions were found to be targeted to powerful desktop workstations or single rendering APIs; our aspiration was to build a system that was suitable for embedded systems and multiple graphics APIs.

This lack of a suitable set of tools led us to design the Graphics Quality Analysis Toolkit. The toolkit comprised three main components: a Tracer for capturing the graphics commands executed by an application; a Trace Player for repeating the captured graphics commands, and a Trace Analyzer for extracting content features and other data from the trace file with the aid of custom instrumented graphics engines. This design enabled the workflow of capturing the graphics of a mobile application to a trace file and extracting the needed content features for classification in a more powerful workstation environment.

Our approach differed from previous work in that we focused exclusively on offline analysis instead of live debugging. This was in part due to the limitations of mobile application platforms, but a stronger motivation was the repeatability of static trace files; once a trace is captured, it can be freely processed independently of the original application. This enables a number of key operations for graphics engineering work, such as seamless transferring of graphics content from one system to another regardless of the involved operating systems, graphics engines and applications. We also demonstrated the possibility to apply arbitrary transformations to the stored graphics commands, such as the conversion into C source code that reproduces the graphics of the original application. Our system also supports extracting subsequences of trace files, which enables more focused analysis of complex graphics applications.

Finally, we demonstrated the usage of our toolkit implementation through a number of facilitated use cases. The use cases were modeled after actual quality issue incidents encountered in previous graphics system integration work. We therefore believe that they accurately represent the process of analyzing real-world quality issues. In

the use cases, the toolkit was used to find the reason behind poor application performance, inspect a visual error in an application, perform application quality analysis, benchmark a number of graphics engines with traced application content and calculate detailed content features the graphics content of an application. The Graphics Quality Analysis Toolkit enabled a practical and efficient process for performing each of these use cases, leading us to conclude that the tracer paradigm is a viable approach for analyzing quality issues in mobile vector graphics applications.

9. REFERENCES

[1] Microsoft (2007), DirectX Resource Center. http://msdn.microsoft.com/directx/.

[2] Gold Standard Group (2007), OpenGL — The Industry Standard for High Performance Graphics. http://www.opengl.org.

[3] Pulli K., Roimela K., Aarnio T. & Vaarala J. (Nov.-Dec. 2005) Designing graphics programming interfaces for mobile devices. Computer Graphics and Applications, IEEE 25, pp. 66–75.

[4] Khronos Group (2007), OpenGL ES — The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/.

[5] Khronos Group (2007), OpenVG — The Standard for Vector Graphics Acceleration. http://www.khronos.org/opengvg/.

[6] Khronos Group (2007), The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies. http://www.khronos.org.

[7] Symbian Limited (2007), Fast Facts. http://www.symbian.com/about/fastfacts/fastfacts.html.

[8] Sales J. (2005) Symbian OS Internals. John Wiley & Songs Ltd, West Sussex, England, 918 p.

[9] Symbian Limited (2007), Symbian OS v9.5 Product Sheet. http://www.symbian.com/symbianos/releases/v95/productsheet.html.

[10] Nokia (2007), S60 Platform. http://www.forum.nokia.com/main/platforms/s60/.

[11] Digia (2003) Programming for the Series 60 Platform and Symbian OS. John Wiley & Songs Ltd, West Sussex, England, 521 p.

[12] Nokia (2004), Introduction to the S60 Scalable UI. http://www.forum.nokia.com.

[13] Nokia (2007), Device specifications. http://forum.nokia.com/devices.

[14] Atlantic Book Publishing (1987) Webster's Dictionary. Book Essentials Publications, Larchmont, New York, 10538.

[15] Steinmetz R. & Engler C. (2001) Human perception of media synchronization , pp. 737–750.

[16] Ghinea G. & Thomas J.P. (1998) QoS impact on user perception and understanding of multimedia video clips. In: MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia, ACM Press, New York, NY, USA, pp. 49–54.

[17] MacKenzie I.S. & Ware C. (1993) Lag as a determinant of human performance in interactive systems. In: CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press, New York, NY, USA, pp. 488–493.

[18] Bederson B.B. & Boltman A. (1999) Does animation help users build mental maps of spatial information? In: INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization, IEEE Computer Society, Washington, DC, USA, p. 28.

[19] Symbian Limited (2005), TAT targets Symbian OS smartphone market. http://www.symbian.com/news/cn/2005/cn20052591.html.

[20] Gilbertson S. (2007), Kiss boring interfaces goodbye with Apple's new animated OS. http://www.wired.com/software/coolapps/news/2007/06/core_anim.

[21] Bryson S.T. (1993) Effects of lag and frame rate on various tracking tasks. In: J.O. Merritt & S.S. Fisher (eds.) Proc. SPIE Vol. 1915, p. 155-166, Stereoscopic Displays and Applications IV, John O. Merritt; Scott S. Fisher; Eds., Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, vol. 1915, pp. 155–166.

[22] Dunwoody J.C. & Linton M.A. (1990) Tracing interactive 3D graphics programs. In: I3D '90: Proceedings of the 1990 Symposium on Interactive 3D graphics, ACM Press, New York, NY, USA, pp. 155–163.

[23] Humphreys G., Houston M., Ng Y., Frank R., Ahern S., Kirchner P. & Klosowski J. (2002), Chromium: A stream processing framework for interactive graphics on clusters. URL: http://citeseer.ist.psu.edu/humphreys02chromium.html.

[24] Sheaffer J.W., Luebke D. & Skadron K. (2004) A flexible simulation framework for graphics architectures. In: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press, New York, NY, USA, pp. 85–94.

[25] Shreiner D., Woo M., Neider J. & Davis T. (2004) OpenGL Programming Guide. Addison-Wesley, fourth ed.

[26] Buck I., Humphreys G. & Hanrahan P. (2000) Tracking graphics state for networked rendering. In: HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM Press, New York, NY, USA, pp. 87–95.

[27] NVIDIA (2007), NVIDIA PerfHUD version 5.1. http://developer.nvidia.com/perfhud.

[28] Graphic Remedy (2007), GDEBugger - OpenGL and OpenGL ES Debugger and Profiler. http://www.gremedy.com.

[29] Trebilco D. (2005), GLIntercept. http://glintercept.nutty.org.

[30] Duca N., Niski K., Bilodeau J., Bolitho M., Chen Y. & Cohen J. (2005) A relational debugging engine for the graphics pipeline. In: Proceedings of ACM SIGGRAPH 2005, ACM Press, New York, NY, USA, pp. 453–463.

[31] Wimmer M. & Wonka P. (2003) Rendering time estimation for real-time rendering. In: EGRW ’03: Proceedings of the 14th Eurographics workshop on Rendering, Eurographics Association, Aire-la-Ville, Switzerland, pp. 118–129.

[32] Chiueh T. & Lin W. (1997) Characterization of static 3D graphics workloads. In: S. Molnar & B.O. Schneider (eds.) 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware, ACM Press, New York City, NY, pp. 17–24. URL: citeseer.ist.psu.edu/707566.html.

[33] Mitra T. & Chiueh T. (1999) Dynamic 3D graphics workload characterization and the architectural implications. In: MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, Washington, DC, USA, pp. 62–71.

[34] Smith T. (1995) An introduction to PHIGS. Tech. rep., University of Arkansas.

[35] Khronos Group (2007), OpenGL Overview. http://www.opengl.org/about/overview.

[36] Pulli K., Aarnio T., Miettinen V., Roimela K. & Vaarala J. (2007) Mobile 3D Graphics: with OpenGL ES and M3G (The Morgan Kaufmann Series in Computer Graphics). Morgan Kaufmann, 500 p.

[37] Perez A. (2000), Computer Graphics I, Course Notes, Direct3D and Multi Texturing. http://www.cs.cmu.edu/afs/cs/academic/class/15462/web.00s/notes/direct3d.pdf.

[38] Blythe D. (2006) The Direct3D 10 system. ACM Trans. Graph. 25, pp. 724–734.

[39] Astle D. (2006), Advanced Visual Effects with OpenGL. http://www.gamedev.net/columns/events/gdc2006/article.asp?id=233.

[40] Microsoft Corporation (2007), Introduction to Direct3D Mobile. http://msdn2.microsoft.com/en-us/library/ms172504.aspx.

[41] Nokia (2004), OpenGL ES API And 3D Graphics On Symbian OS. http://forum.nokia.com.

[42] Adobe Systems Incorporated (1999) PostScript Language Reference. Addison-Wesley Publishing Company, third ed., 912 p.

[43] Microsoft Corporation (2007), Windows GDI. http://msdn2.microsoft.com/en-us/library/ms536795.aspx.

[44] Tronche C. (2005), The Xlib Manual. http://www.tronche.com/gui/x/xlib.

[45] Harrison R. (2003) Symbian OS C++ for Mobile Phones. John Wiley & Songs Ltd, West Sussex, England.

[46] Microsoft Corporation (2007), WPF Graphics, Animation and Media Overview. http://msdn2.microsoft.com/en-us/library/ms742562.aspx.

[47] Worth C. & Packard K. (2003) Cairo: Cross-device Rendering for Vector Graphics. In: Proceedings of the 2003 Ottawa Linux Symposium.

[48] Khronos Group (2007) OpenVG ES 1.0.1 Specification.

[49] Khronos Group (2007) OpenGL ES 1.1.10 Full Specification.

[50] Khronos Group (2006) EGL 1.3 Specification.

[51] Henning M. (2007) API design matters. ACM Queue 5, pp. 25–36.

[52] Symbian Limited (2006), OpenGL ES porting guide for Symbian OS. http://developer.symbian.com/main/downloads/papers/OpenGL.pdf.

[53] Free Software Foundation (2007), The GNU Project Debugger. http://www.gnu.org/software/gdb.

[54] Nokia (2007), Best Practices for HW-Accelerated Graphics Optimization. http://www.forum.nokia.com/info/sw.nokia.com/id/34a99a06-1d7c-4cdb-bd3b-be8cc6a28c17/Best_Practices_for_HW_Accelerated_Graphics_Optimization.html.

[55] Aherne F., Thacker N. & Rockett P. (1997) The Bhattacharyya Metric as an Absolute Similarity Measure for Frequency Coded Data. Kybernetica 32, pp. 1–7.

[56] Khronos Group (2007) OpenGL ES 2.0 Specification.