Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

Tracy: A Debugger and System Analyzer for Cross-Platform Graphics Development

Sami Kyöstilä† [1] Kari J. Kangas‡ [1] Kari Pulli§ [2]

[1]Nokia
[2]Nokia Research Center

Abstract

We describe Tracy, an offline graphics debugging and system analysis toolkit for cross-platform system and application development in mobile graphics. Tracy operates by recording graphics function calls and argument data of unmodified applications into a trace file for offline playback, debugging, and performance analysis. In addition, traces can be edited and converted into platform-independent C files. We pay special attention to real-time performance; our trace compression mechanism allows interactive use of applications even when tracing long, multi-thousand-frame traces in real mobile hardware. We describe the use of the toolkit through real-world use cases such as debugging a visual error or a performance problem in an application, analyzing the application quality, and benchmarking a graphics engine.

Categories and Subject Descriptors (according to ACM CCS): I.3.4 [Computer Graphics]: Graphics Utilities, Software Support; D.2.5 [Software Engineering]: Testing and Debugging, Debugging Aids

1. Introduction

Mobile graphics is a quickly developing area of computer graphics. New 3D APIs such as OpenGL ES and M3G [PAM*07] bring the best features of desktop APIs in a more compact form to handheld devices. Also new 2D vector graphics APIs such as OpenVG and JSR 226 [Khr07,JCP06] are available for user interfaces, animations, and presentation graphics. With the help of these APIs, handheld devices are using increasingly visual user interfaces, they have become viable gaming platforms, and navigation services and maps grow in popularity.

Although tools exist for interactively debugging graphics applications on mature systems such as PCs or game consoles, many of such tools cannot be used efficiently when the target system is in an immature development phase or when the development work is distributed among different platforms. In the handheld space the development environment is usually in a constant flux and inherently cross-platform: a game developer may be developing a game, a graphics vendor develops the engine hardware and drivers, and a handset vendor develops the system software and does the system integration. All this work often happens concurrently on different hardware environments and operating systems. When the game does not work as expected, it is important to quickly pinpoint where the bug is. The source code for all the components may not be available to any of the parties, and even if it is available, digging into the source is a time consuming tedious task. Therefore, tools are needed that allow quick isolation of the bug to a minimal code sequence that can replicate the bug for any of the parties, preferably on the platform that they primarily work on.

Moreover, in handheld space resources are scarce, giving rise to various performance problems. Such problems may show up as an uneven frame rate or unnecessary use of resources, emptying batteries sooner. Tools are needed to flag out suspect graphics engine usage patterns and to suggest

† e-mail:sami.kyostila@nokia.com
‡ e-mail:kari.j.kangas@nokia.com
§ e-mail:kari.pulli@nokia.com

better ones, and to provide a detailed view into the graphics workload to give insights where the bottlenecks are. The strict resource limitations apply also to the tools that are run on the mobile hardware.

To address these issues we have created Tracy, a toolkit for tracing graphics applications to facilitate graphics engine development, application debugging, and application quality estimation in a multi-platform development environment. Our system is based on intercepting graphics commands, i.e., graphics 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. In particular, Tracy offers several key advantages over current solutions.

- Tracy shows the value of traces in the mobile graphics through real use cases. Tracing is optimized for the low performance, low bandwidth environments. While tracing, the traces are compressed in real time which allows creation of long traces of interactive applications in real mobile hardware.
- Tracy uses data-driven design to support multiple platforms and APIs. We currently support several operating systems (Windows, Linux, and Symbian) and several APIs (OpenGL ES, OpenVG, EGL, and APIs built on top of them such as M3G and JSR 226).
- Tracy allows debugging applications without requiring access or modifications to the source code. It converts the traces to platform-independent C source which can be used easily in different systems (OS, HW). Running the compiled C source yields more accurate performance characteristics for profiling and benchmarking than interpreting trace files with a player.
- Tracy allows for extracting subsequences such as single frames from longer trace files while maintaining matching rendering output. In this process, redundant graphics commands are culled through accurate state tracking, greatly reducing the resulting trace file size.

We begin with a discussion of related work (Section 2) and then describe the various components of the Tracy toolkit (Section 3). We present the key use cases in Section 4. We finish the paper with a discussion (Section 5) and conclusions including future work (Section 6).

2. Related work

The basic idea of *tracing* graphics commands used by a graphics application was introduced by Dunwoody and Linton [DL90]. They transcoded the graphics commands into an intermediate API-independent representation. Our approach is to instead save all graphics commands into an API-specific trace file without losing information, allowing the trace to match application behavior at the graphics engine level as closely as possible.

GLTrace and GLSim [Pro01] were among the first publicly available trace utilities, concentrating on tracing and analyzing OpenGL. The flexible data-driven design of our system supports several graphics APIs and provides real-time trace compression. We also highlight the use of trace utilities in various real-world use cases.

The Chromium system [HHN*02] captures and filters an OpenGL graphics command stream and passes it for example to a cluster of graphics workstations to parallelize and speed up the rendering. In addition, Chromium has been used for example to capture and modify the graphics command stream to apply stylized drawing techniques. Instead of implementing the tracer by hand as in Chromium, we use a data-driven design which helps us to easily create a tracer for any C-based API.

A concept closely related to graphics command tracing is *state tracking*. Buck et al. [BHH00] describe how they track OpenGL state in a system used to render tiles of the frame buffer correctly on a graphics workstation cluster, and reduce the communication with lazy updates of graphics state. We opted to build our own state tracking solution to easily support APIs other than just OpenGL. Like Buck et al., we use a hierarchical representation of the graphics API state. A significant difference is that we retrieve the exact function call sequence used to set up a particular graphics API state to guarantee that the meaning of the trace is not inadvertently modified through editing operations such as frame extraction. We also use state tracking to analyze the quality of graphics applications and log unnecessary state changes. Finally, state tracking enables us to serialize vertex array and texture map data given through an unbounded array in OpenGL ES and path coordinate data in OpenVG.

Several tools have been created for *interactive graphical debugging* in desktop PCs and game consoles. For example, PerfHUD [NVI07] from NVIDIA is a proprietary analysis tool for Direct3D in Windows. It shows many statistics from rendering pipeline stages and allows pausing an application and replaying the graphics commands for a frame. Another such tool is gDEBugger [gra07] from graphicREMEDY which also allows visual debugging and strives to enable quick pinpointing of errors and performance issues in OpenGL and OpenGL ES applications. It also shows content statistics obtained from the graphics hardware. A system by Duca et al. [DNB*05] allows debugging of OpenGL programs by storing information about graphics commands into a relational database. In contrast to above systems, we focus entirely on offline debugging, which is usually the most viable way to debug mobile graphics engines and applications while they are being developed, especially in immature systems. During debugging, we work with trace files rather than live applications, as a trace file contains the graphics commands causing a graphics error in a more easily usable format compared to the original application. All complex data extraction and analysis is done as a post-process.

Microsoft PIX [Mic06] is another graphics debugger for

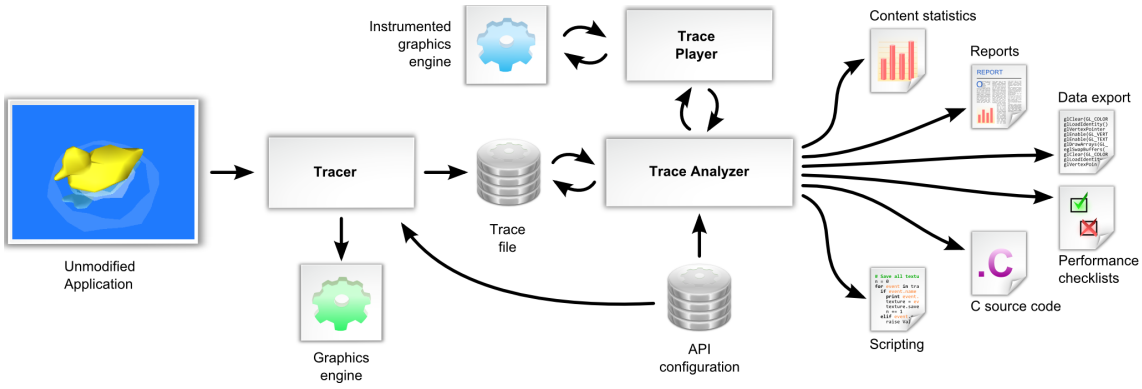

Figure 1: *Tracer intercepts graphics commands going from the application to the graphics engine and saves them into a trace file. The trace file is analyzed offline in the trace analyzer for purposes such as debugging graphics engines or applications.*

Direct3D in Windows. It allows saving graphics commands into a trace file from which they can then be analyzed offline. In comparison to PIX, our system adds support for multiple graphics APIs and engines through flexible API grammar definition and a statistics collection mechanism which is independent of the used graphics engine. We also support editing trace files to produce synthesized graphics content and converting trace files into other formats such as platform-independent C source code.

Finally, *workload characterization* collects statistics on the graphics content to allow, for example, rendering time estimation [WW03, MC99] and characterization of typical graphics content features [CL97]. The workload characteristics gathered by trace files provide an important input for the design of graphics engine architectures [SLS04, RMG*06]. Our system provides access to commonly used 3D (OpenGL ES) graphics content statistics, while also defining similar content features for 2D (OpenVG) graphics.

3. Tracy Architecture

In this section we first describe the overall architecture of Tracy toolkit, followed by the most relevant implementation details. For more details, see the M.Sc. thesis based on this work [Kyö08].

The main components of Tracy are shown in Figure 1. Tracy works by intercepting all graphics commands executed by an unmodified application using a **tracer**. The graphics commands are saved into a **trace file**, which can be replayed in a **trace player** or passed to a **trace analyzer** running in a workstation environment. The trace analyzer allows editing of trace files and extracting raw data, such as OpenGL ES textures, from the trace file. It can also extract content statistics from the trace file by running it in a trace player with an **instrumented graphics engine**. The trace analyzer provides a Python-based scripting interface which makes it easy to implement tools for specific trace processing needs. The tracer, trace player, and trace analyzer are not hard-coded to use a specific graphics API, but can be easily configured for different APIs with a data-driven design.

3.1. Tracer

The purpose of the tracer is to capture application's graphics commands into a trace file. Similar to the related work such as Chromium [HHN*02], our tracer is implemented as a dynamic link library (DLL), which provides an identical interface to the system graphics engine. This allows for tracing existing graphics applications without any source code modifications or recompilation.

3.1.1. API Structure Definition

We specify the grammar and the behavior of an API using an API structure definition. A code generator produces the tracer and the trace player from the structure definition. We favored this approach over hand-written tracer and trace player code, since it is less error-prone and allows for supporting different APIs with ease.

The most significant part of the API structure definition is the C header file defining the API functions, argument data types, and enumerants. We mark a subset of these functions as rendering, frame swapping, or API termination functions. This information is used by the tracer and the trace analyzer to choose functions which contribute to content statistics, to segment trace files into frames, and to shut down the tracer when the application terminates the API.

An essential section of the API structure definition is the set of rules for calculating the sizes of array parameters in API functions. For instance, when saving the texture data passed to the `glTexImage2D` OpenGL ES function, the amount of data to be saved must be calculated from the texture resolution and format. The API structure definition

```
glLightfv:
{
    light:      "root.light"
    pname:      "root.light.parameter"
    params:
    {
        state:    "root.light.parameter.value"
        metatype(class = "array", size = "4"):
        [
            size(condition = "pname",
                 value     = "GL_SPOT_DIRECTION",
                 result    = "3")
            size(condition = "pname",
                 value     = "GL_SPOT_EXPONENT",
                 result    = "1")
            size(condition = "pname",
                 value     = "GL_SPOT_CUTOFF",
                 result    = "1")
            ...
        ]
    }
}
```

Figure 2: *API configuration directives for the `glLightfv` OpenGL ES function. The rules specify how the function parameters affect the API state and how they should be serialized to the trace file. Most notably, the **pname** parameter is used to determine the size of the **params** array.*

provides a compact representation for specifying these array size equations. An example is shown in Figure 2. More complicated cases, such as deriving the number of path coordinates to save in the `vgAppendPathData` OpenVG function, are handled by writing custom serialization C code for the specific functions in the API structure definition. In practice, we found that hand-written serialization code was needed for only few OpenGL ES and OpenVG functions.

Graphics APIs commonly define a mechanism for extending the API. Some extensions simply define new parameter values for the existing functions in the original API. Tracing such extensions does not warrant any special consideration, unless the extension defines new parameter formats, in which case the API structure definition needs to be extended to incorporate the serialization rules or C code. However, some extensions define completely new functions. Both OpenGL ES and OpenVG use EGL to retrieve pointers to the extension functions. For the tracer to capture calls to these functions, it must intercept the function pointer queries and return a pointer to a corresponding tracer function. These extension functions are defined through the API structure definition. Our extension mechanism works also when EGL and the graphics APIs reside in different DLLs.

The API structure definition also includes a hierarchical state model and a mapping from function parameters into it. Our aim was to create a generic state modeling solution which is not limited to either OpenGL ES or OpenVG, and with enough flexibility to support foreseeable C-based graphics APIs such as OpenGL ES 2.0. Instead of explicitly specifying the complete API state, the emphasis was set on modeling the dependencies between various API functions. Our state modeling mechanism is based on a hierarchical data structure called a *state tree*. It is a directed acyclic graph, in which vertices represent the elements of a state structure and edges dependencies between them.

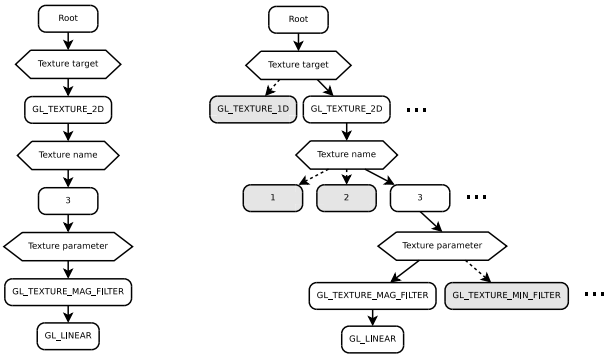

Figure 3: *A state tree for storing the filtering mode for an OpenGL ES texture object, with the type nodes drawn as rounded and the value nodes as angled rectangles. The tree on the left only shows the specific elements used to store the filtering mode, while the tree on the right also shows some alternate options for traversal.*

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

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

A corresponding state tree for this example is shown in Figure 3. There are two different kinds of elements in the tree: types and values. The types may have a set of concrete state values, one of which is marked as current.

The dependencies between commands are modeled by mapping each parameter of state-modifying API function to a type or a value in the state tree using the format shown in Figure 2. For example, the `param` parameter of the `glTexParameteri` OpenGL ES function is mapped to the following *state path*.

Root → Texture target → Texture name → Texture parameter

This state path approach can be used to describe the effects of nearly all commands in the OpenGL ES, OpenVG, and EGL APIs. The state effects of cumulative commands such as the `vgAppendPathData` OpenVG function are handled with custom code defined in the API structure definition.

The hierarchical state model is used for graphics API state tracking which is needed for two main purposes. First, we want to enable the tracer to save, for example, the ver-

tex array data in OpenGL ES. While the vertex data is defined by passing an array pointer to the graphics engine, the actual data used from the array is defined by the subsequent draw commands. State tracking allows us to determine which parts of the array are actually used by each draw command and thus to know which data to serialize into a trace file.

The second use of state tracking is to model the relative dependencies of the API functions and their parameters in the trace analyzer. This makes it possible to extract a set of frames from a longer trace and to perform in-depth analysis of the call trace.

3.1.2. Performance Considerations

Maintaining an acceptable level of performance while tracing applications is greatly dependent on the ability for the tracer to write out data crossing the API boundary to the trace file at a sufficient rate. Our initial approach of using synchronous write operations yielded unacceptable performance in most cases. Implementing write buffering in which the tracer gathered a large amount of data and wrote the whole buffer at once brought performance on average to an acceptable level. However, the synchronous buffer flushing caused a long pause whenever the buffer became full. We finally implemented *fully asynchronous write buffering*, in which a dedicated worker thread collects data into a buffer array in a round-robin fashion and flushes the filled buffers into the output file. With this, the tracer is able to sustain sufficient write performance as long as the average data bandwidth does not exceed the capabilities of the storage device.

While the actual tracing performance depends greatly on the amount of graphics data submitted by the application, we found that a triple-buffered configuration with 512 kilobytes per buffer works well for medium to complex OpenGL ES applications. Finally, to deal with crashing applications we still support fully synchronous writing, which, albeit slow, guarantees that each API call is serialized to the trace file at the time of its execution. The type of buffering and the buffer size can be defined in a run-time tracer configuration file.

In addition to improving write performance, we also *compress the trace on the fly* by detecting repeating data structures in function arguments. In our first trials, a two-minute OpenGL ES animation with roughly 30 000 rasterized triangles per frame generated a 250 megabyte trace file, which was considered too much for most embedded systems. Furthermore, due to the large amount of data being written to the trace file, the performance of the animation was reduced to less than one frame per second. However, animated graphics often exhibits a high level of frame coherence, and we found that a very high percentage of the trace file data consisted of repetitive instances of identical array data. For example in OpenGL ES, the most significant source for this duplication comes from vertex and index arrays; textures are commonly specified only once.

To reduce the trace file size, we first tried to find out whether an array had been already stored into a trace file by calculating a message digest value for the array contents and comparing that to the previous value. Unfortunately a simple CRC32 message digest algorithm was prone to collisions, in which the same digest value was assigned to different array data, and led to situations where modifications to arrays were not caught and written to the file. This resulted for example in visual artifacts in the subsequent trace playback. On the other hand, a more complex MD5 algorithm was computationally too intensive. A more complete array tracking algorithm would make internal copies of each encountered array in order to later check whether the array had been modified, although at the expense of increased memory consumption.

We implemented a compromise where we only track changes to arrays that have been encountered at least twice. During the first encounter, an array is stored into a trace file and marked as seen based on the array memory address. During the second encounter, the array is again stored in the trace file, but a copy is also kept in RAM if available memory permits. After this point the copy is used to check whether or not the array contents have changed. In practice replicating the same array a maximum of two times into the trace file yields a good compression with acceptable processing overhead. Using this approach, the 250 megabyte OpenGL ES trace file was reduced to less than 10 megabytes and the performance of the traced animation run in a modern smartphone was improved from less than one frame per second to more than 10 frames per second, compared to the 25 frames per second without tracing. The array tracker commonly uses roughly the same amount of memory as the amount of vertex, index and path data used by the application. Texture and image data is not tracked in this manner, since duplicate textures and images are usually not reissued by applications.

3.2. Trace Analyzer

The trace analyzer is used to examine and process the trace files. It provides a Python-based scripting interface with support for accessing and manipulating trace data such as individual graphics commands, vertex data, OpenGL ES textures, and OpenVG images. The interface also provides extensive trace manipulation primitives such as extracting and joining trace subsequences. It is also possible to inspect and modify the API state at the graphics command granularity and to access content statistics extracted from the trace file by running it in a trace player with an instrumented graphics engine. Finally, for report generation and for producing diagrams, the interface uses HTML, matplotlib and Python Imaging Library.

The scripting interface makes it easy to implement tools for various trace processing needs. An example of such a tool is the performance checklist which is an automated expert system that looks for known performance deficiencies

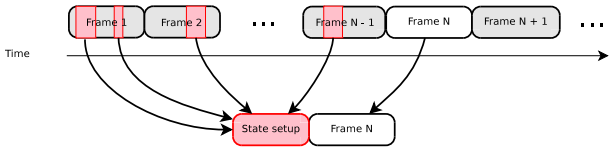

Figure 4: *A single frame is extracted from a trace file. The trace analyzer uses state tracking to determine which preceding graphics commands are needed for the extracted frame to be identical to the same frame in the full trace. These commands are assembled into a state setup sequence.*

in a trace file. Another tool converts a trace file, or a part of it, into equivalent ANSI C source code. Finally, the content statistics can be condensed into summary reports providing a quick overview of the graphics content.

3.2.1. Frame Extraction

Trace files commonly encompass all the graphics commands made by an application while it is running, and thus they typically contain tens of thousands of graphics commands and megabytes of data. Though long traces produce more reliable statistics, only a small part of this data is often necessary when tracking down a bug or preparing a benchmark, making the bulk of the trace file largely superfluous. To make it easier to focus on a particular part of a long trace file, the trace analyzer can be used to extract a subsequence of commands from a trace to form a new smaller trace.

However, simply extracting the selected graphics commands is often not enough, since the objective usually is to preserve the original rendering output of those commands. 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 set of graphics commands is played back.

Extracting a single frame along with the associated state setup sequence is illustrated in Figure 4. When a sequence of events comprising a frame is extracted from a trace file, the analyzer calculates the effective state at the start of the frame, and the graphics commands that have been used to prepare the state are prepended to the frame. This ensures that the rendering output of the extracted frame will be correct. Note that the graphics commands that influence the state may appear anywhere in the preceding trace section.

The trace subsequence extraction algorithm builds on the state modeling system described in Section 3.1.1. The algorithm is based on the observation that a graphics command is a prerequisite for a second graphics command if the state path associated with the first command is a prefix for any of the paths associated with the second call. Based on this, the algorithm can discern between commands that are necessary for setting up a required state and those that are made redundant by other commands. For example, an application might have set the current clear color multiple times before the extracted command sequence. As each color setting command completely overrides the previous one, only the last one is needed to reproduce the effective state.

Common statistics	
API calls	Time stamp, 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

Table 1: *Content statistics provided by the trace analyzer and the instrumented engines.*

3.2.2. Content Statistics

The trace analyzer provides both high-level and in-depth content statistics from a trace file. High-level statistics, such as the number of rendered frames per second, are calculated directly based on the graphics commands in the trace file, while in-depth statistics require the use of an instrumented engine. The detailed content statistics for both OpenGL ES and OpenVG provided by our implementation are listed in Table 1.

3.2.3. Performance Checklist

The performance checklist is an automated API-specific expert system in the vein of Dr. PIX in PIX [Mic06]. It verifies a trace file against a set of predefined conditions that test for known performance deficiencies and other unwanted call patterns, and automatically provides a rough quality estimate of the traced application. Some of the checklist items apply to all graphics engines, while others are specific to the characteristics of a certain implementation. For example, the

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 preloaded 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.

Table 2: *OpenGL ES performance checklist items.*

tests included in the OpenGL ES checklist are listed in Table 2. Should any of these checks fail, the analyzer provides the user with a list of the offending commands that triggered the failure along with a textual description of the detected quality problem. The performance checklist is easy to extend using the Python scripting interface and it also facilitates arbitrarily complex offline analysis.

3.3. Trace Playback

Tracy provides two different ways to execute the commands stored into a trace file: by using the trace player or by converting the trace to equivalent ANSI C code.

3.3.1. Trace Player

The trace player reads a trace file and reproduces the exact graphics command sequence stored therein. The used graphics engine or platform does not need to be the same which was used during tracing as the trace file contains a platform-independent representation of certain objects, such as windows and bitmaps, which are commonly strongly tied to the underlying platform. The trace player uses this information to create a suitable equivalent object for the targeted platform.

3.3.2. C Source Export

Although the trace player can be used to reproduce any needed graphics call sequence stored in a trace file, it does have a number of limitations. First, the player must be separately ported to every operating system it is used on. 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. Finally, 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.

The trace analyzer provides a way to overcome these limitations by converting a trace file to equivalent ANSI C source code in which each graphics operation in the trace file is converted to a corresponding function call. When compiled and executed, the generated code reproduces the original trace sequence with minimal overhead. As the code uses only standard ANSI C constructs and the relevant graphics API functions, it can be run on a wide variety of platforms with a minimal porting effort.

Part of this porting effort may relate to compiler limitations. As it is not uncommon for a trace file to contain tens of thousands of graphics commands, the code size may exceed limits allowed for a single function. We worked around this limitation by splitting up each frame of the graphics trace to a separate function, but some extremely large trace files still failed to compile even with this modification due to the large amount of array data. One solution was to provide an option to direct the array data into a separate assembly language source file, which can then be linked together with the generated C code. So far, the large amount of array data has not posed difficulty to assembly language compilers. In addition, using assembler code also speeds up the compilation of traces substantially.

4. Use Cases

In this section we cover three concrete use cases for the Tracy toolkit:

- debugging visual errors in applications,
- debugging application performance problems, and
- benchmarking graphics engine performance.

These use cases are a generalization of the actual work done in a system-level graphics integration team at Nokia. The team is responsible for delivering graphics technology to other organizational units within the company in the form

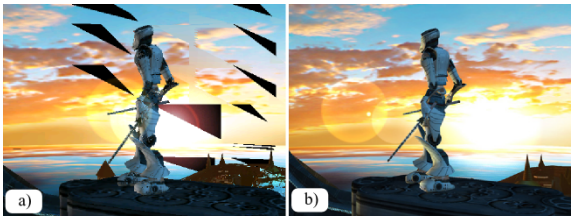

Figure 5: *a) An OpenGL ES application is showing a visual error. b) The trace produces a correct output on a reference graphics engine, indicating an error in the first engine.*

of graphics engines, performance testing, and general graphics support. The team uses Tracy actively in its daily work.

4.1. Debugging Visual Errors in Applications

The most obvious use case for the Tracy toolkit is to debug visual errors or crashes encountered in graphics applications. In this use case we demonstrate how Tracy can be used to identify and isolate the error with minimal effort. Error isolation here means reproducing the error with the minimum number of needed graphics commands. Once properly isolated, finding the root cause of the error in the application or in the graphics engine usually becomes easy.

First step in the use case is to trace the problematic application and bring the trace file into the trace analyzer. We then isolate the error into a new trace file by using the trace manipulation functionality, and play it in the original system to verify that the error still appears. If the error does not reappear, a different command sequence is selected from the original trace until it does. However, finding a correct command sequence is usually quite easy as the trace analyzer allows us, for example, to show the visual output of every drawing command in a trace file, making it easy to spot the ones with visual errors.

Once the error is isolated into a minimal set of graphics commands, we can inspect the commands directly to spot any obvious reasons for the error. Alternatively, we can run the resulting trace on a reference engine to further determine whether the bug is in the application or in the engine. If the reference engine renders the trace correctly, the original engine is the probable culprit. In this case the isolated trace or the C source code based on the trace is given to the engine team or to the engine vendor so that they can debug the hardware and drivers, and use it possibly also for regression testing. It is also possible that the application programmer is relying on unspecified graphics API behavior, leading into different results in the two engines. Such cases can be detected by inspecting the isolated commands or by using multiple reference engines to produce a more reliable estimate of the correct rendering output.

Figure 5 shows an example where an OpenGL ES application displayed invalid graphics with missing triangles and generally distorted geometry. Tracy was first used to replay the trace file from the application on a reference engine. This produced a correct visual output, indicating that the error was caused by the graphics engine. Through error isolation, the root cause of the error was soon identified as a flaw inside the primitive assembly pipeline.

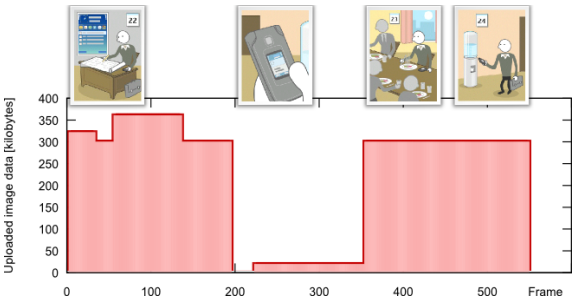

Figure 6: *An OpenVG-based SVG-Tiny player application transfers a large amount of image data to the graphics engine in most of the frames, leading to poor performance.*

In the case where the error is visible also in the reference engine(s), the error must be in the application. The graphics commands in the original and isolated trace file help us to pinpoint the incorrect use of the graphics API.

4.2. Debugging Application Performance Problems

Finding the causes for poor graphics performance is often harder than isolating and fixing obvious bugs, most likely because poor performance is usually caused by issues such as too high content complexity and the use of inefficient API features, or by a combination of such issues. In this use case we demonstrate how Tracy can be used to identify the main reasons for the poor application performance. Such identification helps us to direct the application performance optimization work into the most promising direction.

Using Tracy we can *profile* a graphics application. The trace analyzer is able to produce a number of reports and statistics concerning the graphics content from the trace file. From these, we can easily categorize the performance problems as being caused by inefficient API use patterns, too complex graphics content, insufficient engine performance, or a factor unrelated to graphics. These categories help in communicating the performance problems.

In *high-level content analysis*, the trace analyzer calculates content statistics directly from the trace file. These statistics include the amount of OpenGL ES texture or OpenVG image data uploaded per frame. An example of a performance issue we have identified with this method was in an OpenVG-based SVG-Tiny animation player (see Figure 6). Analyzing the trace showed that the application transferred a large amount of bitmap image data to the graphics

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

Table 3: *OpenVG statistics from an SVG-Tiny image loader.*

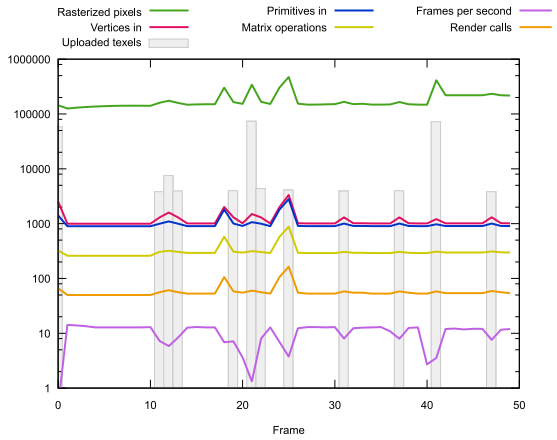

Figure 7: *Graphics content statistics from an OpenGL ES -based menu application indicate that its performance (purple line) varies greatly over time. The low performance seems to correlate with the periodic texture data uploads (gray bars) and increased rendering complexity (green line), suggesting that the workload presented to the graphics engine during these moments exceeds the capabilities of the hardware. Note the logarithmic vertical axis.*

engine in each frame. This observation lead us to a closer investigation of the SVG-Tiny player, which uncovered a flaw in image caching logic: the player kept unnecessarily redefining the same small set of static bitmap images.

In addition to the high-level profiling information presented above we can also perform *in-depth content analysis*. We have instrumented OpenGL ES and OpenVG engines to calculate and extract detailed statistics, shown in Table 1, that allow the developers to gain understanding of the content-related bottlenecks in their application. We can inspect these statistics in combination with benchmark results to see if the statistics indicate too complex content for the particular graphics engine. Such statistics of existing applications can also help in designing and balancing a future graphics system.

An example of data we have extracted with this process is shown in Table 3. Here an OpenVG-based SVG-Tiny image loader was used to load several application icons: each consecutive frame corresponds to a different SVG-Tiny file. The statistics begin with general information about the recorded trace file. This is followed by a frame-level breakdown of various content features. For each feature, the minimum, maximum and mean values of that feature are listed along with a histogram showing the value distribution.

The trace analyzer can also produce alternative visualizations of detailed content statistics. Figure 7 presents an example of such a visualization by outlining how the content complexity of an OpenGL ES -based menu application varies considerably from time to time, leading to jerky animation and reduced usability.

Other examples of performance problems that were solved with Tracy include:

- An OpenGL ES -based menu application failed to stop submitting rendering commands even when it was not being displayed, leading to reduced device usage time.
- Analysis of an OpenGL ES -based navigation software revealed that the application was using an internal proprietary software engine to render most of its output. This made the application unable to take advantage of graphics hardware acceleration.
- An OpenVG application was found to be repeatedly reinitializing the graphics engine rendering context for no practical purpose, causing superfluous processing.

Finally, we can quickly analyze the quality and API usage patterns using a simple *graphics expert system* using check lists such as presented in Table 2. Such analysis should be performed, perhaps as a part of the software quality review, even if an application is not suffering from an obvious performance problem as the analysis can highlight certain areas that can be improved to conserve battery, CPU, or memory usage. For instance, on a particular graphics engine certain invocations of the `glColorMask` command are emulated through a software work-around due to hardware limitations. By adding detection of such commands into the checklist we were able to quickly identify and fix applications which were triggering the slow software emulation code path.

4.3. Graphics Engine Benchmarking

The final use case switches the focus from applications to graphics engines: a new graphics engine needs to be bench-

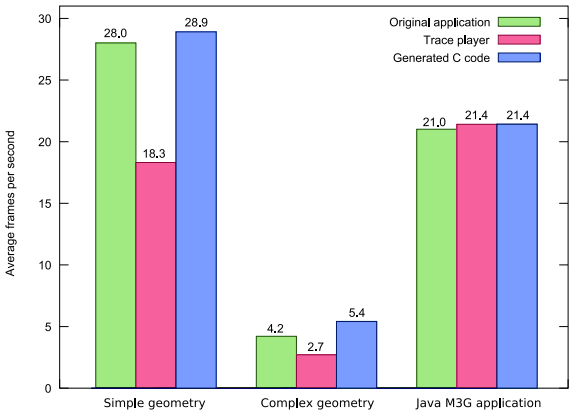

Figure 8: *Performance comparison between the original application, the trace player, and C code generated from the trace. The performance of the C code closely matches that of the original application.*

marked to estimate its fitness for rendering the graphics of a particular application. By using Tracy, to do this we do not need to port the application itself to the system with the new graphics engine. Instead, the trace file, or a selected part of it, represents the application graphics.

In graphics benchmarking we usually want to concentrate solely on the graphics performance aspects of the application. The trace player can read in the trace file and play back the graphics commands. However, the player always has a runtime overhead compared to a hard-coded, compiled application as it needs to read and decode the trace file. This overhead can be significant on platforms with a slow file system and when dealing with large trace files.

As an alternative, we can produce an ANSI C code file that directly runs the graphics commands in the trace. Figure 8 shows a performance comparison between the original application, our trace player, and generated C code. From the charts we see that running the new executable based on the generated code matches the performance of the original application much more closely than the trace player is able to do. The applications were selected so that they do little else than submit graphics commands.

Another important point in favor of code generation is that there is no need to port the trace player to different development environments. Porting of benchmark tools has proven to be a notable obstacle in cross-platform, cross-company working environments. The C files can be incorporated into any platform-specific benchmark application with minimal platform adaptation work.

The benchmarks created in this manner are executed on the graphics engine, producing an estimate on how well the new graphics engine will perform when rendering the chosen content. These benchmarks can also be used to obtain virtually zero-overhead profiling information of the time spent in each graphics engine function.

5. Discussion

Interactive graphical debugging using tools such as PIX, PerfHUD, or gDEBugger is probably the most efficient approach when working completely on a single stable platform and in a controlled environment where each developer is able to work mostly independently with his or her own source code. However, the benefits of an offline trace analysis and debugging become apparent when the final system, the graphics engine, and applications are being developed concurrently by different teams in different companies, all potentially using different hardware and software platforms. In such situations, using trace files and their platform-independent C code counterparts for representing and communicating graphics errors and benchmarks becomes a powerful tool. Especially the C source code has proven useful in practice due to minimal porting effort and close to zero performance overhead.

In addition, tracing and analyzing applications provides guidance to graphics engine developers. It also helps to improve application quality. Such analysis can be done conveniently and automatically without access to or need to study application source code.

Our approach to compress trace files is effective for typical applications that have a stable set of models which are animated using matrix transformations, but it fails to reduce the trace data volume for applications that dynamically generate new geometry 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 for each frame. Since the structure of the compound objects changes frequently, the tracer must write each encountered variation to the trace file, leading to poor performance. Fortunately, we have found very few such applications.

It is possible for an application to use multiple graphics APIs at the same time. As each graphics engine is commonly placed in a separate DLL, multiple concurrent API-specific tracers are needed in such a case. In Tracy, each recorded graphics command is annotated with a system-wide sequence identifier, which uniquely identifies the order in which the graphics commands were issued. This allows for the API-specific trace files to be merged by the trace analyzer into one coherent trace file which contains all the graphics commands executed by the application. The same trace player and C source file can support multiple APIs.

6. Conclusions and Future Work

We have presented Tracy, an offline graphics debugger and analyzer. The key design target was a toolkit that can be used on a developing environment that consists of multiple hardware platforms, operating systems, teams potentially located

into different companies with no easy access to each others' source code, and multiple graphics APIs. The toolkit facilitates both extracting bugs and optimizing the execution speed and use of system resources. The traced application can be run on a mobile phone at interactive speeds and generate very long trace sequences of thousands of frames.

A natural continuation for our work is to extend Tracy to cover OpenGL ES 2.0 and other relevant mobile graphics APIs. While basic tracing and playback and C source export of OpenGL ES 2.0 should work directly in our current system, workload characterization requires more in-depth statistics extraction from the shader programs.

The automated graphics content checklists produced by Tracy could be made more reliable by more thorough use of content statistics. For instance, if Tracy notices that a particular texture map is always rendered at the same scale, it could suggest better prefiltering of the texture and turning mipmapping off, whereas by default mipmapping is always recommended. The checklist should also be extended to support graphics engine specific profiles to account for differences in graphics engine implementations.

The content statistics provided by our system open the possibility for further research into clustering applications based on the complexity of their graphics content. Given a large enough sample set, applications could be classified into performance classes, providing a way to roughly estimate performance on new graphics architecture. Clustering could also allow for creating synthetic benchmarks based on real applications. While benchmarks created from traces are representative of a given application, they are hard to parameterize in terms of content complexity. Synthetic benchmarks derived from trace files could overcome this limitation while still remaining representative of real-world graphics content.

Acknowledgments

We would like to thank Mika Qvist for helping us throughout this work, and Tomi Aarnio, Kimmo Roimela, Jani Vaarala, and Wei-Chao Chen for reading previous versions of this paper and for suggesting many improvements. Finally, we would also like to thank the anonymous reviewers for their helpful suggestions for improving this paper.

References

[BHH00] BUCK I., HUMPHREYS G., HANRAHAN P.: Tracking graphics state for networked rendering. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Aug. 2000), pp. 87–96.

[CL97] CHIUEH T., LIN W.: Characterization of static 3D graphics workloads. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (1997), pp. 17–24.

[DL90] DUNWOODY J. C., LINTON M. A.: Tracing interactive 3d graphics programs. In *1990 Symposium on Interactive 3D Graphics* (Mar. 1990), pp. 155–163.

[DNB*05] DUCA N., NISKI K., BILODEAU J., BOLITHO M., CHEN Y., COHEN J.: A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics 24*, 3 (Aug. 2005), 453–463.

[gra07] GRAPHIC REMEDY: gDEBugger – OpenGL and OpenGL ES debugger and profiler. `http://www.gremedy.com`, 2007.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P., KLOSOWSKI J.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (July 2002), 693–702.

[JCP06] JCP: JSR 226: Scalable 2D vector API for J2ME. `http://www.jcp.org/en/jsr/detail?id=226`, June 2006.

[Khr07] KHRONOS GROUP: *OpenVG 1.0.1 Specification*, Jan. 2007.

[Kyö08] KYÖSTILÄ S.: *A mobile vector graphics quality analysis toolkit*. Master's thesis, University of Oulu, 2008.

[MC99] MITRA T., CHIUEH T.: Dynamic 3D graphics workload characterization and the architectural implications. In *ACM/IEEE Int. Symp. on Microarchitecture* (1999), pp. 62–71.

[Mic06] MICROSOFT: PIX. `http://msdn2.microsoft.com/en-us/library/bb173085(VS.85).aspx`, June 2006.

[NVI07] NVIDIA: NVIDIA PerfHUD version 5.1. `http://developer.nvidia.com/perfhud`, June 2007.

[PAM*07] PULLI K., AARNIO T., MIETTINEN V., ROIMELA K., VAARALA J.: *Mobile 3D graphics with OpenGL ES and M3G*. Morgan Kaufmann Series in Computer Graphics, 2007.

[Pro01] PROUDFOOT K.: GLTrace. `http://graphics.stanford.edu/courses/cs448a-01-fall/glsim.html`, 2001.

[RMG*06] ROCA J., MOYA V., GONZALEZ C., SOLIS C., FERNANDEZ A., ESPASA R.: Workload characterization of 3d games. *Workload Characterization, 2006 IEEE International Symposium on* (Oct. 2006), 17–26.

[SLS04] SHEAFFER J. W., LUEBKE D., SKADRON K.: A flexible simulation framework for graphics architectures. In *Graphics Hardware 2004* (Aug. 2004), pp. 85–94.

[WW03] WIMMER M., WONKA P.: Rendering time estimation for real-time rendering. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering* (June 2003), pp. 118–129.