CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Katedra měření

Master's Thesis

Collaborative GPU rendering

for lower-class mobile devices

Bc. Max Hollmann
Počítačové inženýrství

Leden 2024
TODO http://petr.olsak.net/ctustyle.html
Supervisor: Ing. Michal Sojka, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hollmann Max** Personal ID number: **483587**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Computer Engineering**

II. Master’s thesis details

Master’s thesis title in English:

Collaborative GPU rendering for lower-class mobile devices

Master’s thesis title in Czech:

Kolaborativní vykreslování pro pomalá GPU v mobilních zařízeních

Guidelines:

Computing power of GPUs on low-end mobile phones is not sufficient for running many modern games and other rendering-heavy applications. The goal of this thesis is to investigate possibilities of using more powerful, but idling devices reachable via Wi-Fi to accelerate graphics rendering or improve image quality of the mobile application and provide better experience for phone users.
1. Review existing approaches to collaborative rendering and GPU off-loading. Furthermore, review popular mobile game engines and selected games and study their use of graphics APIs on Android system. Select three games of different graphics complexity to use in testing of implemented rendering stack.
2. Modify Android OpenGL ES rendering stack to support full offloading of render operations to remote devices over Wi-Fi. Choose the most convenient device to offload to (e.g. PC with specific GPU).
3. Propose and at least partially implement one or more methods of collaborative rendering, where the mobile device will render the scene in low quality and use data from remote device to provide higher-quality result.
4. Test and compare the developed approaches on selected games. Propose several metrics for evaluation, such as latency, game or device compatibility and evaluate your solution with those metrics.
5. Carefully document the results and propose direction for further development and improvements.

Bibliography / sources:

[1] E. Cuervo et al., “Kahawai: High-Quality Mobile Gaming Using GPU Offload,” in Proceedings of the 13th Annual International Conference
on Mobile Systems, Applications, and Services, New York, NY, USA, May 2015, pp. 121–135. doi: 10.1145/2742647.2742657.
[2] C. Wu, B. Yang, W. Zhu, and Y. Zhang, “Toward High Mobile GPU Performance Through Collaborative Workload Offloading,” IEEE
Transactions on Parallel and Distributed Systems, vol. 29, no. 2, pp.435–449, Feb. 2018, doi: 10.1109/TPDS.2017.2754482.

Name and workplace of master’s thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **13.02.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until:
by the end of winter semester 2024/2025

Ing. Michal Sojka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

Acknowledgement / Declaration

I would like to express my gratitude to the following individuals, without whom this thesis would likely never see the light of day.

I would like to thank Sára Krinerová for providing loving support whenever I felt demotivated to continue working.

I would like to thank my family for pushing me forward and supporting me financially.

I would also like to thank Jakub Jíra for sharing his tea while we were working on our theses in the lab.

And I would like to thank my supervisor for his mentorship, guidance and constructive criticism, and for always pointing me in the right direction to focus my energy.

Lastly I would like to mention ChatGPT, which hasn't been as useful as I expected it to be, but it still turned to be a useful tool for overcoming writer's block and for translating the abstract (I did editorialize the output to its current form).

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

V Praze dne 9. 1. 2024

.....

Abstrakt / Abstract

Mobilní telefony se v posledních letech staly velmi populární platformou pro hraní her a jsou největším trhem pro počítačové hry. Jejich hlavní výhody, malé rozměry a provoz na baterii, jsou zároveň jejich největší slabinou. Mobilní hry jsou omezeny v grafické věrnosti nižší výkonností mobilních čipsetů a výrazně zkracují výdrž na baterii.

Tato diplomová práce představuje řešení pro kolaborativní vykreslování úpravou vykreslovacího stacku OpenGL ES.

Klíčová slova: OpenGL; vzdálené vyhodnocování kódu; Android; počítačové hry; Linux; mobilní zařízení; počítačová grafika;

Překlad titulu: Kolaborativní vykreslování (pro pomalá GPU v mobilních zařízeních)

Mobile phones have become a very popular platform for gaming in recent years and are the largest market for games. Their main advantages, their small size and battery operation, are also their greatest weakness. Mobile games are limited in graphic fidelity by the lower performance characteristics of mobile chipsets and significantly reduce battery life of the handsets.

This thesis introduces a solution for collaboratively offloading rendering by modifying the OpenGL ES rendering stack.

Keywords: OpenGL; code offload; Android; computer games; Linux; mobile devices; computer graphics;

Contents /

/ Figures

Chapter 1
Introduction

Smartphones are the most popular platform for gaming. [1] In today's world, many people have a smartphone with them at all times. In emerging markets, smartphones are often the only computing device someone might own, replacing the functions of both desktop and portable computers, but also gaming consoles, cameras and more. Because we always carry our smartphones with us, they are the device we turn to when we are looking for a distraction, such as when sitting on the toilet. However, smartphones are also an inherently limited platform for gaming. Games render complex virtual worlds with the requirement of rendering 60 times per second for fluid and interactive game-play. This requires a lot of computational power and as a result, energy.

While smartphones have been steadily increasing in performance, they are behind desktop computers or home gaming consoles in terms of computational power. Furthermore, unlike the aforementioned devices, smartphones are usually not plugged-in when in use and have to operate on battery. As a result, mobile games provide a limited experience when compared to traditional gaming devices and negatively impact battery life, making the smartphone a worse phone. Furthermore, high-performance flagship smartphones are expensive, resulting instead in lower-performance low-end or mid-range devices having the widest adoption. One advantage smartphones have over other portable gaming devices is that they are connected to the mobile network. Many smartphone games take advantage of this feature and are inherently multi-player in nature.

The goal of this project is to try out various methods of alleviating the downsides of mobile gaming when in the proximity of a different device. By offloading graphics computation to a different device, we hope to reduce power consumption of the mobile device, lower the SoC temperature, increase battery life, increase performance in GPU limited scenarios and increase graphical fidelity.

This project focuses mainly on OpenGLES games on the Android operating system. Android is the majority operating system on smartphones, with over 70% market share. It is developed as an open source project by Google, however most companies license and modify the operating system for their own devices. OpenGLES is a version of OpenGL, the popular graphics API, modified for embedded systems. OpenGLES closely resembles standard OpenGL. It uses the same client-server design as regular OpenGL and also uses the GLSL language for programmable shaders. However, some functionality from OpenGL is missing and as such, it is not possible to run OpenGL applications on devices which only support OpenGLES. The intention of this project is to leverage the natural client-server architecture of OpenGLES.

This thesis implements a collaborative rendering solution for rendering OpenGL ES applications, such as those used on the Android OS. A main goal of this thesis is to

enable collaborative rendering for all OpenGL ES applications by modifying Android's rendering stack, rather than modifying each application individually. The following chapters will discuss what technologies are used in Android's 3D rendering, how different games make use of those technologies, the design of our solution and its implementation.

Chapter 2
Background

In this chapter we make an overview of existing technologies which are relevant for this thesis. We will look at both technologies used when running games, as well as useful tools for tracing of running games.

2.1 OpenGL and OpenGL ES

OpenGL [2] is a cross-platform, open graphics API that allows developers to interact with a computer's graphics hardware. It provides a set of functions for rendering 2D and 3D graphics, making it a powerful tool for graphics programming and game development. First released in 1992, OpenGL has become an industry standard and is widely supported on various platforms. OpenGL does not include specification for creating and managing windows and rendering contexts. This functionality is left to specific platforms to decide how it should be implemented.

OpenGL ES [3] is a well-defined subset of desktop OpenGL suitable for low-power devices. Aside form embedded systems, OpenGL ES is also available on Unix and Unix-like desktop operating systems. Most importantly for this thesis, OpenGL ES is the graphics API of choice of the Android operating system.

Both OpenGL and OpenGL ES APIs are developed by Khronos Group, an open consortium of industry vendors. Currently the development of both APIs is suspended in favour of the newer Vulkan API.

2.2 EGL

In the previous section we mentioned that OpenGL and OpenGL ES cover the functionality of 3D and 2D rendering, but that they do not cover creating system windows, surfaces and rendering contexts. EGL [4] is an API which manages interaction between the native platform window system and various Khronos Group rendering APIs, including OpenGL ES.

2.3 Android

Android is an open-source operating systems primarily aimed at smartphones. It is currently the most popular OS on mobile devices [1]. Android is built using many pre-existing open-source projects and open standards, including EGL, OpenGL ES and the Linux kernel.

Unlike destop Linux distributions, Android does not use the X Window System or its newer replacement, Wayland [5]. Android instead uses a custom window and display systems called SurfaceFlinger and WindowManager [6]. Helpfully this difference is abstracted away by EGL, which also works on desktop Linux envirnments.

2.4 apitrace

apitrace [7] is a set of tools to trace, replay and inspect OpenGL and Direct3D calls. apitrace also allows for saving of recorded traces, which means it also includes a listing of all OpenGL and OpenGL ES functions including all information required for serialization of the function calls.

apitrace performs its tracing by injecting a tracing layer into games, this layer intercepts all graphics API calls, records them, and forwards them to the real graphics drivers. The approach employed by this thesis is similar to apitrace's tracing mechanism.

2.5 Software Libraries and ABI

As described in the section above, OpenGL is only a specification and implementations are provided by various vendors in their driver packages. The mechanism of how this is implemented and the limits thereof are important to understand the implementation of this thesis, and some of the problems I encountered when implementing the solution.

Software libraries are collections of pre-written code and routines that developers can use to perform common tasks or functions without having to write the code from scratch. Some libraries are available in source code form and it is the responsibility of the programmer to integrate the library with their code and compile it themselves. Most system libraries are however provided in precompiled binary form. In precompiled libraries functions are already compiled for the target architecture. While this has the benefit that the library doesn't require the lengthy compilation process, it also means that a lot of information about the functions is lost. The compiled code is enough to be executed by a CPU and location of each function is recorded in a provided symbol table, but information about how many arguments the function expects, what types they have, and where the arguments should be located is lost.

In order to use library functions, the compiler which compiled the library function, and the compiler used for the rest of the application have to both know the types of arguments, and they have to agree on where and how they should be placed before the function is called. In source libraries, this is simple because the compilers are the same and thus they will always agree. In precompiled libraries this doesn't work. We don't know how and with what compiler the library was compiled with. Instead the location and representation of data types and function arguments is specified in an ABI (Application Binary Interface). If a library and an application using it is compiled with the same ABI, even with different compilers, they are compatible and can be used together. For system libraries a stable ABI is advantageus as any compiler can the implement the ABI and give its user access to all system libraries compiled with the same ABI. The C programming language has a stable ABI on most platforms, which is why the C ABI is used for system libraries on Linux, Android and many more operating systems. Developers are provided with a C header file which contains all declarations needed to use the library, following the C ABI.

Chapter 3
Related Work

In the first chapter, we looked at the problems of mobile gaming and why we would like to take advantage of dedicated hardware. In this chapter we will take a look at some existing approaches in this area, which parts of the application they offload, how the offload is performed and what trade-offs these approaches have.

3.1 Full Offload

This section focuses on approaches where the game is fully offloaded. The application is fully run on a remote device and the local device is used only as an input device and a display. A great advantage of this approach is that it has very small performance requirements on the client device, especially if acceleration of video decoding can be accelerated by dedicated hardware. Smart TVs, for example, are great devices for this approach Mobile phones are currently a large platform for gaming, but the traditional platforms of gaming consoles and personal computers have large libraries of existing games and are the target for new big-budget experiences. As a result, a class of programs has emerged which brings these games from traditional gaming platforms to low power devices, including smartphones.

These programs run the game entirely on a gaming PC or a console and transmit only game inputs, and fully rendered frames.

3.1.1 Local game streaming

Following is a list of programs which all require a powerful gaming PC within the local network.

- Nvidia GameStream
- AMD Link
- Valve SteamLink
- Sunshine + Moonlight

The main advantage of local game streaming is its lower latency in comparison to approaches where data is transmitted over the internet. The main disadvantage is that it requires preexisting ownership of a powerful gaming device. Local game streaming is a good solution for enjoying the same gaming experience in multiple rooms within a household, it does not however enable access to games which are harder to run, than what the user's hardware already supports.

3.1.2 Cloud Gaming

Cloud gaming is similar to local game streaming, but removes the requirement of owning a gaming computer by moving the gaming device to a data center. Unlike local gaming, cloud gaming is a paid service, where users pay for access to a device in a data center. This reduces the cost of entry, but increases latency due to the gaming device being

further away from the player. By moving the computation device to a data centre, cloud gaming also allows users to enjoy games while moving around without carrying a bulky gaming device. Notable examples of cloud gaming services are:

- GeForce Now
- Xbox GameCloud
- PlayStation Now

3.1.3 Remote play on own hardware

An interesting combination of the above two technologies is offered by both Sony and Microsoft on their consoles. The Remote Play feature on Xbox and PlayStation relays information through a server in the cloud, allowing for the convenience of cloud gaming, but the user's own gaming console at home is used for rendering the game, which is why this approach does not require a monthly payment.

3.2 Remote Desktop

A rather different problem which converged to a similar solution is the problem of accessing graphical applications on remote computers. The tools in this section are popular for remotely managing many devices in corporate settings, or for allowing professionals to utilize software which runs on a centralized server. The motivation for the latter is usually due to licensing costs, performance constrains, security of data, or a combination thereof. The tools in this section were developed to allow remote access to graphical computer systems. They were designed for IT departments to manage software installations in corporate settings, or to give employees access to licensed software running on a central server. As a result, most of these tools provide inferior experience for gaming applications when compared to the tools from the previous section. However, there are two tools which stand out in their technical approach and I will take a deeper look into them in the rest of this section.

3.2.1 Remote X and GLX

The X Window System is a popular choice among Unix and Unix-like operating systems to enable graphic windows. An important design aspect of the X System is that it follows a client-server architecture, where each application is a client which connects to a server in order to draw its window. More importantly the X protocol was designed to communicate over a POSIX socket [8], including network sockets. To this day tunneling an X connection over an SSH session is the way to use graphic Linux applications remotely.

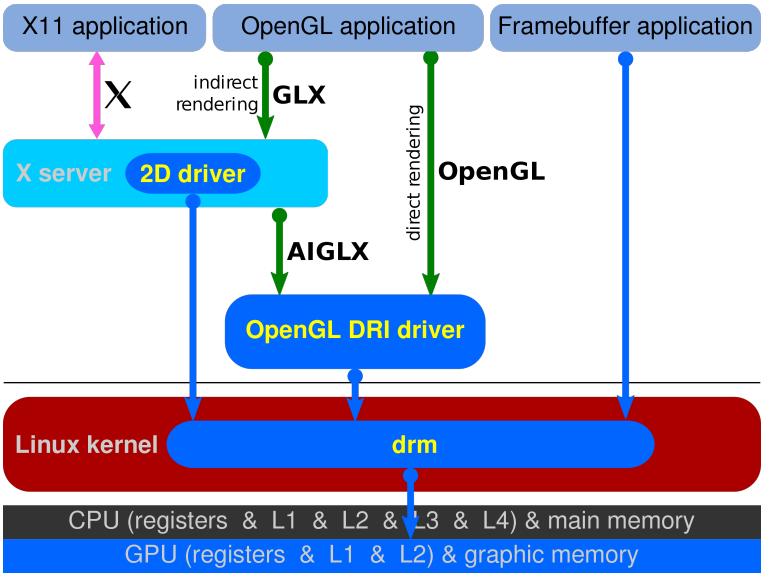

Figure 3.1. Current rendering structure of OpenGL using X11 on Linux [1]

When OpenGL was being added to the X Window System, the GLX extension to the X protocol was developed for managing OpenGL surfaces and configurations, a precursor to modern EGL. Because the X protocol communicated over BSD sockets and GLX was an extension to the X protocol, GLX also operated over BSD sockets. As a consequence, rendering to a remote X server using a network socket also forwarded GLX and OpenGL to be rendered on that X server. As we will learn later in this thesis, serializing regular C library calls to a socket incurs a substantial overhead and a Direct Rendering Infrastructure (DRI) was developed for rendering and displaying OpenGL graphics locally. As a result, the lastest supported version of OpenGL over remote X is 1.3 and needs to be explicitly enabled first in the graphics driver of the X server.

3.3 Kahawai

The current state-of-the-art and main inspiration for this thesis is Kahawai [9], a system for high-quality gaming on mobile devices. Kahawai introduces two approaches for partial offloading of GPU computation.

3.3.1 Delta Encoding

In this approach, the scene is rendered in low quality on the mobile device and in both low and high quality on a remote computer. As the low and high quality images are highly correlated, the image of their differences has very low entropy and can be highly compressed for transmission. The high quality image is reconstructed on the low-power device by decompressing the difference and applying it to the locally-rendered low quality image. A bonus advantage of this approach is that if the connection with the remote server is interrupted, the device can seamlessly fallback to displaying the locally-rendered low quality frames.

The main downside of this approach is that rendering a scene in two different qualities requires modifying the game or its underlying engine.

[1] By Shmuel Csaba Otto Traian, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=29417694`

3.3.2 I-frame rendering

The second approach is built around a typical idea in video compression. Assuming subsequent frames are likely to be similar, only the first frame can be encoded fully and following frames are encoded only as a difference from the previous frame. The independently encoded frames are called I-frames, and the frames which are encoded using a prediction from the previous frame are called P-frames.

I-frame rendering renders only I-frames on the low-power device, while rendering all frames on the high-power device. A typical video encoding is then used to transmit the P-frames to the low-power device, except that unlike a regular video encoder, the I-frames are completely omitted from the compressed stream and instead the locally-rendered P-frames are used for decoding.

While the paper demonstrated the theoretical advantages of this approach without modifying the game, it did so by rendering all frames on both devices and discarding P-frames on the low power device. Additionally, this approach is incompatible with temporal effects [10], which have become popular since the release of Kahawai.

Chapter 4
Requirements

In the previous chapter, we looked at related work. In this chapter, we will summarize the requirements we expect from the offloading system.

Given that this work is more exploratory, instead of a whole technical specification, we will draw inspiration from dividing requirements into functional and non-functional requirements.

The work is expected to be split into two components. One component running on a client device and one running on a rendering server.

4.1 Functional requirements

Functional requirements describe specific behavior and capabilities that the system must have to fulfill its intended purpose. These requirements outline what the system should do, what features it should have, and how it should respond to various inputs - detailing all actions, activities, and operations.

Here are the listed functional requirements:

- The system intercepts OpenGL ES rendering calls from a client application
- The intercepted rendering calls are sent over a network to a server component of the system
- The server evaluates the rendering requests
- The server sends the final picture back to the client part of the system
- The final picture is presented to the user as-if it were rendered regularly without any intervention of the system
- The client may also evaluate some calls locally to improve the performance and user experience of the system

4.2 Non-functional requirements

Non-functional requirements state the demands on an application or system that are not directly related to functional features but are essential for ensuring overall effectiveness, usability, system performance, or security. They focus more on its characteristics, technologies, quality, or constraints rather than specific behavior.

Here are the listed non-functional requirements:

- The client must be compatible with the Android OpenGL ES rendering stack
- Communication between the client and server must be capable of operating over Wi-Fi
- The server must be capable of evaluating OpenGL ES calls correctly
- The entire system must be transparent to the end-user

Chapter 5
Feasibility study

In the prior chapters, we have introduced the technologies used in Android gaming and prior work in the field of remote gaming and collaborative rendering. In this chapter we will review how existing games make use of the Android rendering stack, what problems we might encounter when moving GPU rendering to a remote device and if these problems occur in the games examined.

5.1 Motivation of the study

OpenGL (and by extension, OpenGL ES) is a one-directional API. Objects are prepared on the CPU, then sent off to the GPU, rendered and displayed on a monitor. In a typical application, when a scene is loaded, all objects contained in that scene are uploaded to the GPU. When rendering the scene, the objects are referred to only by their IDs, which were generated during the uploading process. When changing scenes, unused objects are deallocated using their IDs from the GPU and new objects are loaded instead. Furthermore, because displays are usually connected to the GPU directly, presenting the newly rendered frame is done by sending a command to the GPU containing the information to swap which buffer is currently being rendered-into, and which is used for display scan-out. In this architecture, moving the GPU to a remote device and sending the commands over the network should increase the rendering latency only by the network latency and the performance is not affected. However OpenGL also includes additional commands which provide the CPU with some information about the current frame. When such command is submitted, the GPU has to complete all work up until that command, compute the information the CPU requested and send the result back. The CPU is waiting for the result and does not do any useful work. On traditional systems, the GPU is connected to the CPU using PCIe or even directly when both are part of the same SoC. The latency on such systems is low enough that sparse usage of these commands does not result in degraded performance. In our case the latency between the CPU and GPU is significant.

In order to analyze how severe this issue is for current games, the following process was proposed. Trace OpenGL ES calls of several Android games. If these features are not used in practice, their performance is not of important concern. Analyze the functionality of those commands, which were encountered. If possible, replace their implementation with a conservative approximation. If a usable conservative approximation does not exist, the scene will have to be rendered locally on the mobile device. However this does not have to mean rendering the frame completely. Occlusion queries for example can be resolved completely by rendering the depth buffer only, while most complexity of modern rendering lies in advanced pixel shaders.

5.2 Tracing

In order to evaluate which API calls were used by the games tested, several tracers and debuggers were evaluated for capturing the calls.

5.2.1 apitrace

We previously looked at apitrace in section 2.4 for its definitions of OpenGL APIs. Unfortunately while apitrace is a great tool for capturing rendering traces on desktop devices, it does not support Android.

5.2.2 Android GPU Inspector

Android GPU Inspector (AGI) [11] in an official Google tool to help profile graphics on Android. Unfortunately it works mainly on Google's Pixel devices, which the author does own. Furthermore, OpenGL ES applications are run using ANGLE [12], an emulation layer implementing OpenGL ES over Vulkan, resulting in graphical glitches.

Lastly, I have found AGI less ergonomic to work with than RenderDoc.

5.2.3 RenderDoc

RenderDoc [13] is an MIT licensed graphics debugger with broad support for both platforms and APIs. Due to its wide platform support, it will allow us to compare traces of local and remote rendered frames.

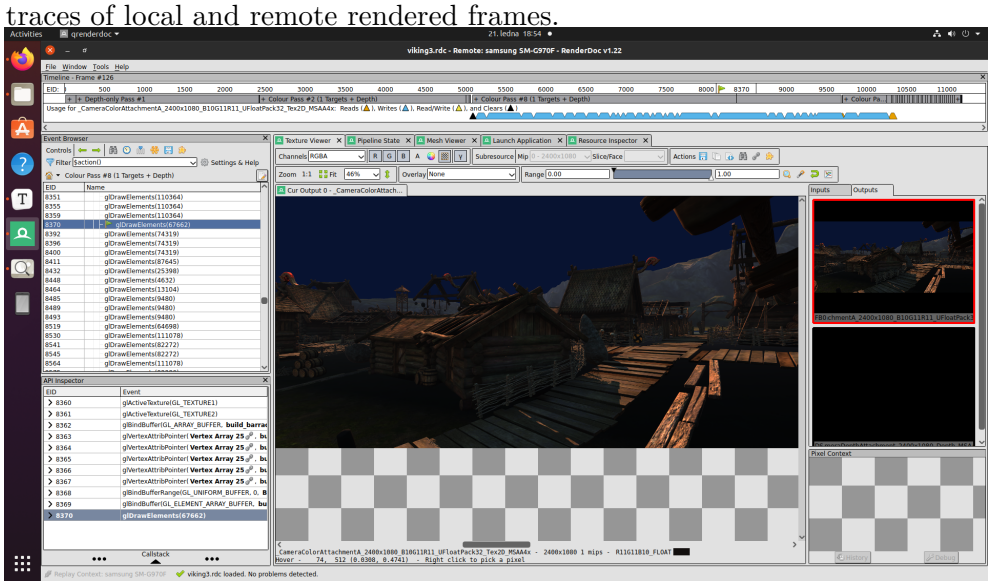

Figure 5.1. RenderDoc displaying a trace of the Viking Unity Village Demo

5.3 Games

To find out which APIs games use and validate remote rendering once implemented, the following games were selected:

5.3.1 GDTLancer

GDTLancer[1] is an open-source game in Godot, an open-source game engine. Selected to have a game with full source code available for easier debugging and development.

[1] `https://github.com/roalyr/GDTLancer`

Figure 5.2. A screenshot of the game GDTlancer

5.3.2 Viking Village Unity Demo

Viking Village Unity Demo[2] is a graphically intensive demo of the Unity game engine. Intended to run on desktops and gaming consoles, but can also be compiled to run on Android phones, resulting in very poor performance.

Figure 5.3. A screenshot of the Viking Village Unity Demo

5.3.3 Solar Smash

Solar Smash[3] is a single-player game built using the Unity game engine. It is graphically significantly simpler than the Viking Village Demo and is more representative of the graphic fidelity of mobile games.

[2] https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-urp-29140
[3] https://play.google.com/store/apps/details?id=com.paradyme.solarsmash

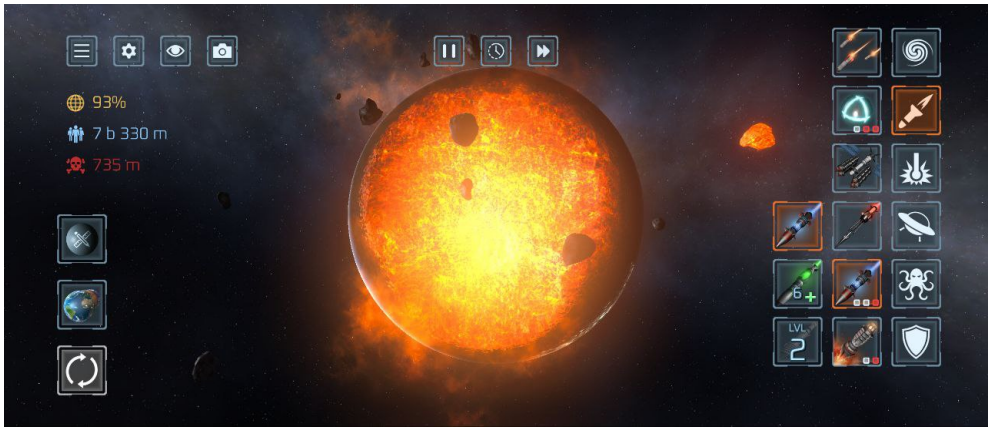

Figure 5.4. A screenshot of the game Solar Smash

5.3.4 SimCity BuildIt

SimCity BuildIt[4] is a mobile version of the popular EA city-building game.

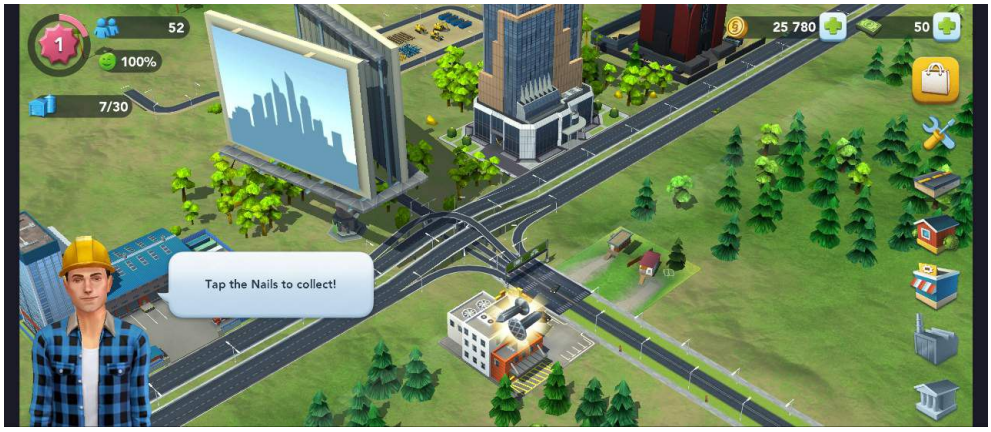

Figure 5.5. A screenshot of the game SimCity BuildIt

5.4 Tracing Results

I have tested the listed games and recorded several API call traces from each game. Following is an exhaustive list of used OpenGL ES calls used by the games tested. The API calls are sorted by the frequency of occurrence in descending order

- glVertexAttribPointer
- glBindBuffer
- glActiveTexture
- glDrawElements
- glBindBufferRange
- glBindTexture
- glUniform4fv
- glBindBufferBase
- glTexParameteri
- glInvalidateFramebuffer
- glUseProgram

[4] `https://www.ea.com/games/simcity/simcity-buildit`

- glBufferSubData
- glUnmapBuffer
- glFlushMappedBufferRange
- glBindFramebuffer
- glEnableVertexAttribArray
- glDisableVertexAttribArray
- glViewport
- glScissor
- glDrawElementsBaseVertex
- glEnable
- glDisable
- glBlendFuncSeparate
- glCullFace
- glUniform3fv
- glStencilOpSeparate
- glStencilFuncSeparate
- glClear
- glDepthFunc
- glDepthMask
- glClearDepthf
- glStencilMask
- glColorMask
- glClearStencil
- glClearColor
- glPolygonOffset
- glFrontFace
- glDrawArrays
- glBlendEquationSeparate
- glUniform1i
- glFenceSync
- glClientWaitSync
- glBufferData
- eglSwapBuffers

All OpenGL ES functions utilized by the tested games either modify state in the rendering server, or render elements in the rendering server. None of the functions is blocking for the client, with the sole exception of the combination of `glFenceSync` and `glClientWaitSync`. The former sets up a synchronization fence and the latter blocks the client until the server reaches the previous synchronization fence. However, these functions were always used to wait on the rendering of the previous frame, which leaves one frame of acceptable delay to report the completeness of the previous frame to the client, which should not be limiting in our use-case.

Chapter 6
Design

In this chapter, we delve into the intricacies of designing the library. We explore the Android rendering stack and how we can modify it to achieve the goals set forth in chapter ??.

6.1 High level overview

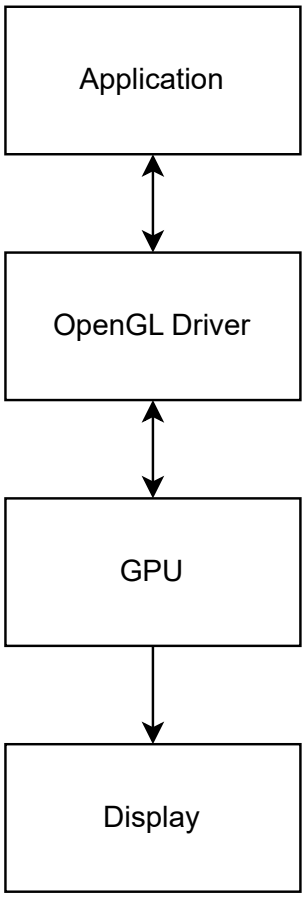

Figure 6.1. The layers of a standard OpenGL rendering stack.

One of the important goals of this thesis was to interoperate with existing applications without modification to said applications. OpenGL is a specification by Khronos group, a consortium of various vendors. Each vendor implements OpenGL and provides it to customers in a driver package as a shared library. The architecture of a regular OpenGL rendering stack is shown in figure 6.1. We can pretend to be a driver vendor and implement a shared library which will behave as a regular driver OpenGL implementation from the viewpoint of client applications. Internally however, the library forwards OpenGL API calls to a different device

over the network and utilizes native OpenGLES and EGL only to present rendered frames.

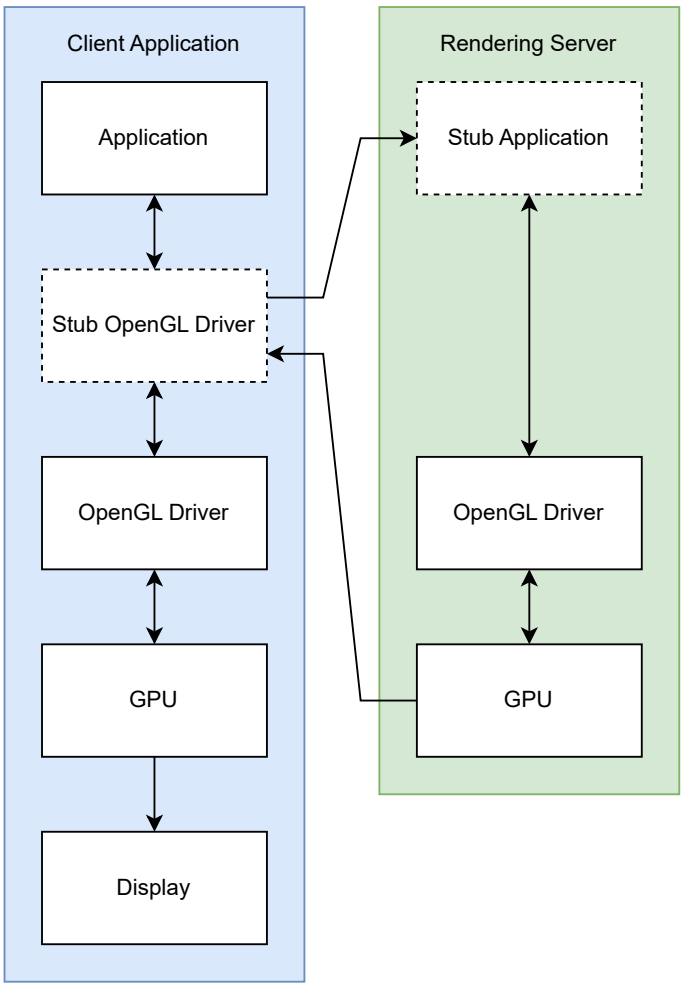

Figure 6.2. Server and client in our OpenGL offloading rendering stack.

We call the remote rendering device the rendering server. Figure 6.2 shows how we can slot-in to the existing OpenGL rendering stack. A stub driver injected into the client application offloads rendering calls to the rendering server, where a stub application uses the rendering stack on the server device to render the frame. The rendered frame is then sent back to the client stub driver, which uses the existing OpenGL infrastructure to display the frame. The communication between the server and client is implemented over a network socket.

6.2 Serialization

A key part of this thesis is serialization of all OpenGL calls. By assigning each function a unique ID and serializing it with the function's arguments, we can reconstruct which function and with what arguments was requested and perform the call on the server. The specific value of the ID is arbitrary and serves only for the client and server to transmit which function was called. A simple sequence of increasing integers in declaration order of the functions is sufficient.

Although OpenGL ES was created as a reduced variant of standard OpenGL, and EGL was designed to be a simpler alternative for OpenGL state management

to existing libraries, I still had to implement 171 different function calls and decide for many more that they are safe to omit. Implementations for many of these functions are very similar. On the client side, each function has to serialize its ID and arguments to the network socket. On the server side, the application reads the id of the next function from the socket, deserializes arguments for said function and calls it. If it is a function that returns some information to the driving program, that information also has to be serialized and sent back to the client, which then deserializes it and returns it to the application.

As implementing all functions would be a tedious and error-prone work, which would have to be redone if the communication format were to change, I implemented several approaches for generating these functions automatically. As OpenGL is a C library, it comes in two parts. The shared object library we are going to create, and one or more C header files which describe the API of the library. I implemented a generator which would create the serialization and deserialization functions from the C header definitions. This worked great for many functions and would have been the preferred solution. Unfortunately, C headers only describe functions within the semantics of the C language. Some functions, such as `glGenTextures` or `glGetAttachedShaders`, require more information, which cannot be described in C headers, to properly implement and call.

Arrays are the most problematic structures of all. The C language does not support passing arrays directly in functions argument. Instead, passed arrays undergo pointer decay, and only a pointer to the first element is passed. The length of the array has to be passed as a different argument, or the end of the array is marked by a sentinel value, such as the nul byte in C-style strings. When an array is passed in OpenGL, a pointer type is present in the header file, but in order to determine how many elements should be passed, documentation of the specification has to be consulted. Considering how many OpenGL functions accept an array and how many different kinds of arrays are present, this approach rapidly devolved into implementing most functions manually by hand.

Specification, documentation and C headers are not the only description of OpenGL API available. Because OpenGL is a very popular API, many tools have been created which help in debugging and tracing of OpenGL applications. These tools also faced the challenges described in the previous section and created their own description of the OpenGL API, with added information to describe the semantics of the API which cannot be described using C headers alone.

One such tool, apitrace, maintains a description of OpenGL APIs using Python objects. It then generates its serialization and deserialization code from this description. Unfortunately apitrace's serialization is intended for a different use-case. Because apitrace is primarily a record-replay debugger for various graphics APIs, it records information about out arguments only after the function is called. One of the added semantics in the OpenGL API is that some functions behave differently if some of their out arguments are null pointers (e.g. `eglGetConfigs`). Therefore we must also encode if a pointer was null or not into the serialized stream. apitrace is a complex tool which supports multiple graphics APIs and its code is flexible to support this use case. This flexibility is however a source of complexity and because we support only a single API, a simpler approach was chosen.

A compromise between generating code from C headers and modifying apitrace's generation code is to use apitrace's OpenGL description, but write a custom generator. The custom generator is much simpler than apitrace and better suits the needs of this thesis. However the better description of the API from apitrace's Python objects can still be utilized.

Chapter 7
Implementation

We have discussed the theoretical background and the design of the implementation. This chapter delves deeper into the details of implementation and how to use the final software.

The implementation itself is split into two main parts. The first part is a generator which generates serialization and deserialization code for most functions in the OpenGL ES and EGL APIs. The second part is a combination of a library which replaces OpenGL ES and EGL in the client and a server application, which evaluates the offloaded OpenGL calls. Some apitrace definitions include C code snippets for calculating array lengths, which limited the choice of language to C or C++. C++ was chosen because it allowed for a slightly more ergonomic implementation of the serializer. A portion of the client code is written in Zig and heavily utilizes its compile-time reflection capabilities.

7.1 Generator

Due to the size of the OpenGL ES API, a large portion of the final code is generated using a Python script. The goal of the generator is to take apitrace definitions and generate code for the client, and corresponding code for the server.

```
# Example function as listed by apitrace
GlFunction(Void, "glGenTextures", [(GLsizei, "n"),
  Out(Array(GLtexture, "n"), "textures")])

// Generated client stub code
extern "C" void APIENTRY glGenTextures(GLsizei n, GLuint *textures) {
  ser_de.ser(static_cast<uint32_t>(130));
  ser_de.ser(n);
  ser_de.flush();
  bool textures_present = ser_de.deser<bool>();
  if (textures_present) {
    size_t len = ser_de.deser<size_t>();
    for (size_t i = 0; i < len; i++)
      textures[i] = ser_de.deser<unsigned int>();
  }
}

// Corresponding code generated for the server
void glGenTexturesImpl() {
  int n;
  n = ser_de.deser<int>();
```

```
    GLuint textures[n];
    glGenTextures(n, textures);
    ser_de.ser(textures != nullptr);
    if (textures) {
      size_t len = n;
      ser_de.ser(len);
      for (size_t i = 0; i < len; i++) {
        ser_de.ser(textures[i]);
      }
    }
    ser_de.flush();
  }
```

The generator is implemented in the file `generator.py`. The sources also include several Python files from the apitrace [7] project, these files include the definitions, or supporting code used by the definitions.

apitrace represents types in the OpenGL API by nesting Python objects, each adding a modification onto the previous, until the last object, which represents a trivial type. Similarly, functions are also objects with fields for the output type, function name and a list of argument types. apitrace uses this object structure in conjunction with the visitor pattern to generate its serialization code. I decided to use a simpler approach using recursive functions and Python's run-time type reflection facilities.

Generating code works for most functions whose implementations are similar to each other. However some functions include constructs which are not used by many other functions and implementing their required functionality in the generator would be more work than to implement them manually. An example of a very special function is `glShaderSource`, which expects the shader string in a rope-like data structure, composed of either 0-terminated, or length-delimited strings depending on if the lengths argument is present. However as the string is concatenated by the serialization process anyway, the server complement of this function is significantly simplified. All manually implemented functions are located in the the file `manual.cpp`.

As mentioned in chapter 2, the semantics of many functions cannot be expressed using C headers, which is why apitrace definitions were used. The functions `glFinish` and `glFlush` have semantics which are out-of-scope even for apitrace. The generated implementations for these functions were modified to satisfy the OpenGL specification, but are not part of `manual.cpp` due to the small size of the modifications.

7.2 Serialization

Serialization and deserialization is implemented in the file `serde.cpp`. It is important to send the rendering commands as quickly as possible, but the amount of data these commnds need is very little. The system is latency sensitive but the calls used to draw each frame do not require transporting large amounts of data.

I decided to implement a simplicity-focused serialization system by writing each value in network-endian to the network socket. This approach is significantly more efficient than possible textual encodings or the encoding used by apitrace, which includes additional information to allow playing back traces from different versions of apitrace.

Another aspect of serialization is buffering of data. As mentioned above, most OpenGL calls comprise of small amounts of data but a scene is rendered using many such calls. Sending each call independently would result in substantial overhead from the operating system's network stack. Buffering of data is performed to reduce the total number of packets sent by incresing their average size.

7.3 Presentation

After a frame is rendered on the remote device and the client receives it, it has to be displayed to the user. One set of APIs which can be used to display the image and we know is present is EGL and OpenGL ES. A full-screen triangle is drawn and the received image is used as its texture, displaying the frame to the user.

There is one problem with this implementation. We are using OpenGL ES and EGL functions inside our library, which implements the same set of functions. As a result, using the required functions in the usual manner is not possible. POSIX specifies a second method for accessing dynamic library functions, `dlopen` and `dlsym`. These function manually open a shared library and look-up symbols by name. Using the reflection and compile-time evaluation tools offered by Zig, the following excerpt automatically populates a structure of function pointers to OpenGL ES functions.

```
const gles_so = std.c.dlopen("/usr/lib/libGLESv2.so", 1);
inline for (@typeInfo(@TypeOf(gles)).Struct.fields) |field| {
    @field(gles, field.name) = @ptrCast(std.c.dlsym(
        gles_so,
        (field.name ++ [_]u8{0})[0..field.name.len :0]
    ));
}
```

The result is a structure whose fields have the same name and type as the function they represent, and the value is a function pointer to the native implementation of said function. An analogous structure exists for EGL functions. Using these functions is very similar to their normal usage — `gles.glDrawArrays(c.GL_TRIANGLES, 0, 3);`.

7.4 Configuration

Although the implementation was designed to be as transparent to the end user as possible, some aspects require or offer some configuration. This section describes what configuration options exist and how to modify them.

7.4.1 Native Shared Object Location

In the previous section we have explained that the client implementation must load OpenGL ES and EGL implementations manually using `dlopen`. In order to load the libraries, their locations on the client device must be known. The expected location of the libraries is configured in the file `src/client.zig` at lines 39 and 40 for EGL and OpenGL ES respectively. The default location should work on most Linux systems, but some distributions may place system libraries in different locations.

7.4.2 Network Options

The server and client parts of this thesis communicate together over the network. In order for the client to connect to the server, the client must be configured with the server's IP address. The IP address is configured in the file `src/serde.cpp` at line 74. The port number can also be configured in the same file and is defined at line 32.

7.4.3 Buffer Size

The implementation uses sending and receiving buffers to join multiple requests together before sending them over the network, reducing the overhead of sending many individual network packets. Increasing the buffer size reduces the number of system calls, improving performance. Increasing the buffer size however also increases memory consumption and delays the execution on the remote server until the buffer is flushed, reducing performance.

Choosing a size for a buffer is always a trade-off, optimal buffer sizes for one setup may result in poor performance on a different setup. Tuning the buffers is accomplished by modifying the values in `src/serde.cpp` at lines 37 and 38 for output and input buffers respectively.

7.5 Building

The software is provided in source code form and is implemented in a compiled language. Before using it, the source files must first be converted to an executable application and a binary library by building the project. The project has a few libraries which must be installed before compilation.

7.5.1 Dependencies

The run-time dependencies of the project were intentionally kept at a minimum. Adding dependencies to the client library might cause conflicts if the application also used the same dependencies. For compiling this project the following dependencies have to be installed locally.

- Zig 0.11.0:[1] Used for compiling both Zig, but also C++ source code.
- XCB [14]: Library for registering an X window with the X server for the server application
- EGL and OpenGL ES: Required by the server to execute the offloaded commands.

[1] `https://ziglang.org/download/#release-0.11.0`

7.5.2 Compilation

Compilation is done using the Zig build system. With the dependencies installed, the project is compiled with the following command.

```
zig build -Doptimize=ReleaseSafe
```

Both the server application and the client library are produced with the previous command and are stored in the `zig-out` folder.

7.6 Usage

Using the In order to use the software, the server application must be started first.

```
./zig-out/server
```

Once the server application is running, the game can be started. The following command forces the dynamic loader to use functions from our library instead of loading them from the system-provided implementation.

```
LD_PRELOAD=./zig-out/lib/libclient.so <application>
```

Chapter 8
Results

In this chapter we will take a look at the final library and how it performs at offloading. We will examine the graphical fidelity of the rendered image and the performance of an application running through our OpenGL ES offloader.

8.1 Testing Hardware

All testing was done on a laptop computer equipped with an Intel Core i5 7300U with Intel HD Graphics 620 and 8GB RAM, running Linux 6.6.8.

8.2 Software

The performance of the offloading library was evaluated on a simple testing application, es2gears[1]. es2gears is part of the Mesa demos package of OpenGL demo applications.

8.3 Performance

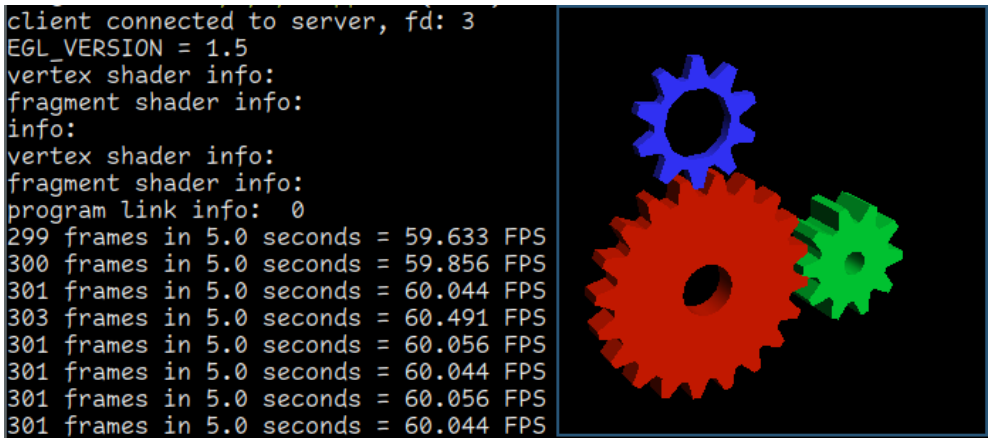

Figure 8.1. es2gears running at 300

- 300 through OpenGL ES offloader

Running the testing application through the network offloader resulted in full performance of 60 FPS and no visual artifacts are present. In these conditions the offloading layer is fully transparent to the user.

[1] `https://cgit.freedesktop.org/mesa/demos/tree/src/egl/opengles2/es2gears.c`

```
client connected to server, fd: 3
EGL_VERSION = 1.5
vertex shader info:
fragment shader info:
info:
vertex shader info:
fragment shader info:
program link info:  0
90 frames in 5.1 seconds = 17.822 FPS
71 frames in 5.0 seconds = 14.146 FPS
107 frames in 5.0 seconds = 21.391 FPS
115 frames in 5.0 seconds = 22.972 FPS
112 frames in 5.0 seconds = 22.200 FPS
112 frames in 5.0 seconds = 22.337 FPS
115 frames in 5.0 seconds = 22.972 FPS
```

Figure 8.2. ES2gears running at 1920

- 1080 through OpenGL ES offloader

Increasing the window size and thus the rendering resolution has a negative impact on the performance. The offloading layer is not able to supply rendered frames from the server to the application fast enough and performance is degraded to 22 FPS. Furthermore at this window resolution the application is stuttering leading to a degraded experience to the user.

The implemented offloading method correctly runs over a network socket and produces identical images to and application running natively. Unfortunately the performance becomes a limiting factor at greater resolutions.

Chapter 9
Conclusion

In conclusion this thesis has sought to review existing methods for collaborative rendering, analyze the Android OpenGL ES rendering stack, propose and partially implement collaborative rendering.

In chapter 6 we have analyzed the OpenGL ES rendering stack and proposed implementing collaborative rendering using a virtual driver as shown in figure 6.2.

This solution for collaborative rendering on a remote device has been implemented. The CPU part of the application is kept on the low-power device, but the more power-intensive GPU part is fully offloaded over a network socket. This thesis thus shows that the idea of collaborative rendering is possible by modifying the existing OpenGL ES rendering stack without modifying the applications.

Performance of the offloading solution was also tested. At lower screen resolutions the implementation performed well and was transparent to the user. However at larger screen resolutions the implementation slowed down and slowed down the entire application.

9.1 Future Work

OpenGL is a complex API to implement. In it's current state the work presented here does not implement the full OpenGL ES specification. Furthermore the performance of the offloading library is disappointing and causes a regression in user experience.

Both of these aspects could be addressed in possible future work.

9.1.1 Compatibility

Following is a list of OpenGL features currently missing in this library. Implementing them would increase compatibility with currently incompatible applications.

- BLOB arguments
- `eglGetProcAddress`
- returning arrays
- sync objects
- buffer mapping

9.1.2 Performance

As discussed in chapter 8, the performance of the offloading library is not good and causes performance regressions on larger display resolutions even with simpler applications. The following is a list of proposed features which should improve the performance of the whole system.

- FrameBuffer compression
- asynchronous network operations
- relaxing synchronization primitives
- implement I-frame rendering, described in section 3.3.2
- Buffer size tuning
- UDP sockets instead of TCP

References

[1] Thomas C G, and A. Devi. A Study and Overview of the Mobile App Development Industry. *International Journal of Applied Engineering and Management Letters*. 2021, 115-130. DOI 10.47992/IJAEML.2581.7000.0097.

[2] The Khronos Group. *OpenGL Overview*. 2011. `https://www.khronos.org/opengl/`.

[3] The Khronos Group. *OpenGL ES Overview*. 2011. `https://www.khronos.org/opengles/`.

[4] The Khronos Group. *EGL Overview*. 2011. `https://www.khronos.org/egl/`.

[5] freedesktop.org. *Wayland*. `https://wayland.freedesktop.org/`.

[6] Google. *Android Open Source Project: Surfaceflinger and windowmanager*. `https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager`.

[7] José Fonseca. *apitrace*. 2022. `https://github.com/apitrace/apitrace`.

[8] James Gettys, Philip L. Karlton, and Scott McGregor. The X window system, version 11. *Software: Practice and Experience*. 1990, 20 (S2). DOI 10.1002/spe.4380201404.

[9] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai. *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 2015. DOI 10.1145/2742647.2742657.

[10] Lei Yang, Shiqiu Liu, and Marco Salvi. A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum*. 2020, 39 (2), 607-621. DOI https://doi.org/10.1111/cgf.14018.

[11] Google. *Android GPU Inspector*. `https://developer.android.com/agi`.

[12] Google. *ANGLE*. `https://chromium.googlesource.com/angle/angle/+/main/README.md`.

[13] Baldur Karlsson. *RenderDoc*. `https://renderdoc.org/`.

[14] freedesktop.org. *XCB*. `https://xcb.freedesktop.org/`.

[15] Chao Wu, Yaoxue Zhang, Lan Zhang, Bowen Yang, Xu Chen, Wenwu Zhu, and Lili Qiu. *ButterFly: Mobile collaborative rendering over GPU workload migration*. In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017. 1-9.

[16] The Khronos Group. *Khronos Registry*.
`https://registry.khronos.org/`.

[17] Jorge Rodríguez. *docs.gl*.
`https://docs.gl/`.

[18] OpenAI. *ChatGPT: A Large-Scale Generative Model for Open-Domain Chat*. 2021.
`https://openai.com/chatgpt`.

Appendix A
Attachments

A.1 Source Code

All source code is bundled in the archive `source.tar.gz`. The contents of the archive are listed below.

```
source.tar.gz
+-- build.zig
+-- gen
|  +-- debug.py
|  +-- eglapi.py
|  +-- eglenum.py
|  +-- generator.py
|  +-- glapi.py
|  +-- gles2_whitelist.txt
|  +-- gles3_whitelist.txt
|  +-- glparams.py
|  +-- gltypes.py
|  +-- stdapi.py
+-- src
   +-- client.cpp
   +-- client.zig
   +-- glsize.hpp
   +-- serde.cpp
   +-- server.cpp
   +-- signatures.zig
```

Appendix B
Glossary

ABI	Application Binary Interface
AGI	Andorid GPU Inspector
API	Application Programming Interface
BLOB	Binary Large OBject, A large amount of opaque binary data, without information about internal structure
DRI	Direct Rendering Infrastructure, A framework for managing access of unprivileged user-space programs to graphics hardware
EGL	Embedded-Systems Graphics Library
OpenGL	Open Graphics Library
OpenGL ES	OpenGL for Embedded Systems
OS	Operating System
PC	Personal Computer
PCIe	PCI express, A high-speed point-to-point interconnect used between the CPU and other components inside of a computer
POSIX	Portable Operating System Interface, IEEE standard for compatibility between operating systems
SoC	System on a Chip, A single chip which includes functionality of CPU, GPU and more, which are usually implemented using more chips.
Vulkan	A modern cross-platform graphics API, intended to replace both OpenGL and OpenGL ES by granting developers lower-level access to graphics hardware features
XCB	X protocol C-language Binding, A library for communicating using the X protocol from C programs