The Edward S. Rogers Sr. Department of

Electrical and Computer Engineering

University of Toronto

ECE496Y Design Project Course

Group Final Report

Optimizing 3D Gaussian Splatting for Qualcomm Snapdragon Hardware

Team Number 2024094

Team Members:
Stefan de Lasa, stefan.delasa@mail.utoronto.ca
Damian Pacynko, d.pacynko@mail.utoronto.ca
Marko Ciric, marko.ciric@mail.utoronto.ca

Under supervision of Dr. Andreas Moshovos
With Afshin Poraria (Administrator)

Submitted on: March 21st 2025

Final Report Attribution Table

Section	Student Names		
	Stefan de Lasa	**Damian Pacynko**	**Marko Ciric**
Executive Summary	ET	RS, RD, MR, ET	ET
Introduction		ET	RD, MR, ET
Background and Motivation	RD	ET	MR, ET
Project Goals and Requirements			RD, MR, ET
Final Design	RD, MR, ET	RD, MR, ET	RD, MR, ET
System-Level Overview	ET	RD, MR, ET	ET
System Block Diagram		RD, MR	
Module-Level Descriptions	RD, MR, ET	RD, MR, ET	RD, MR, ET
Testing and Verification	RD, MR, ET	RS	RD, MR, ET
Testing Table	RD, MR, ET		
Development and Testing Environment	ET	RD, MR, ET	
Frame Rate	RD, MR, ET	RS, ET	
Memory Usage	RS, ET	ET	RS, RD, MR, ET
PSNR	MR, ET		RD, ET
Summary and Conclusion			RD, MR, ET
Future Work	ET	RD, MR, ET	ET
All		CM	FP

Abbreviation Codes:

RS – responsible for research of information

RD – wrote the first draft

MR – responsible for major revision

ET – edited for grammar, spelling, and expression

OR – other

"All" row abbreviations:

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

Signatures

Name	Stefan de Lasa	**Signature**	*Stefan de Lasa*	**Date:**	March 21, 2025
Name	Damian Pacynko	**Signature**	*Damian Pacynko*	**Date:**	March 21, 2025
Name	Marko Ciric	**Signature**	*Marko Ciric*	**Date:**	March 21, 2025

Group Highlights (Author: Stefan)

As a team, we successfully implemented a 3D Gaussian Splatting renderer on a novel type of hardware (Snapdragon 8 Gen 2 System on Chip). This implementation will ease the algorithm's adoption on various platforms that are powered by Snapdragon chips, such as Meta's Quest headsets [1] and various Android smartphones [2]. In achieving this, we:

- Configured our code to compile and build on novel hardware (OnePlus 12R smartphone).
- Fixed bugs involving low-level hardware dependencies in GPU code, ultimately resulting in a functioning 3D Gaussian Splatting renderer.
- Established a remote server-based development environment, to allow team members to develop and test code in parallel.
- Created a helpful interface to allow users to move the renderer camera's position through the Gaussian scene via touch input.
- Measured the FPS of the renderer across known datasets to quantify the performance of the renderer's baseline implementation.
- Succeeded in meeting our peak memory usage requirement, even in large scenes with 3 million gaussian splat primitives.
- Achieved a 91.5% improvement in frame rate with a Display Unit module optimization

Individual Contribution - Stefan de Lasa

Before the Design Review Meeting, Stefan contributed to the project as he:

- Obtained necessary hardware for development (OnePlus 12R smartphone, powered by the Snapdragon 8 Gen 2 chip).
- Configured hardware for development, allowing new apps to be compiled and uploaded to the phone.
- Created a simple Android app that can run a test Python/PyTorch script.
    - However, the team later pivoted away from using Python and the PyTorch library since a Vulkan-based approach seemed to fit our renderer application's needs more closely.
- Configured the project's code to compile and deploy programs to the Android device/app using the Gradle and CMake build system.
    - This was difficult since the baseline renderer implementation [3] the team was building off of made several assumptions that did not work on our new hardware.

Since the Design Review Meeting, Stefan:

- Rendered a 3DGS point cloud using a Vulkan-based 3DGS renderer.
    - This involved fixing crashes and bugs to adapt the baseline renderer implementation [3] to work with the Snapdragon's limit hardware capabilities
- Tried to get the Snapdragon Profiler [4] running in MacOS and Linux environments. This would allow us to understand our renderer's resource usage and bottlenecks.
    - This faced several operating system and dependency issues. Ultimately, Stefan transferred the task to Marko to run the profiler on his Windows machine.
- Fed ground truth camera poses into the renderer so that peak signal-to-noise ratio (PSNR) measurements could test the profiler's image quality.
    - Worked to debug mismatches between expected and observed camera pose information. Tried to find ground truth adjustable Gaussian Splatting renderers for comparison.
- Developed an FPS measurement system with improved precision to validate the speed of our renderer.

Individual Contributions - Marko Ciric

Before the Design Review Meeting, Marko contributed to the project as he:

- Researched devices with the Snapdragon 8 Gen 2 SoC, ultimately deciding on the OnePlus 12R smartphone being used because it also satisfied our ideal of having 16 GB of RAM which allows for extensive testing with large 3DGS scenes
- Planned to use ExecuTorch and Qualcomm's SDK to have PyTorch interface with the Snapdragon hardware, but pivoted away from this because Qualcomm's SDK wasn't compatible with the renderer as it expected neural-network models as an input [5], which our renderer is not
- Investigated Vulkan APIs and searched for existing Vulkan 3DGS renderers, settling on a high-performance Vulkan renderer that supports cross-platform development [3] meaning that we can all use the renderer, and it will serve as a reliable base from which to develop our own 3DGS renderer for Snapdragon hardware and mobile devices

Since the Design Review Meeting, Marko:

- Implemented the **Touch Interpreter** module to allow users to configure the camera view through various touch inputs, because the baseline renderer implementation [3] was configured only for desktop inputs (keyboard and mouse) and wasn't compatible with mobile devices that use touch screen, so the camera view initially couldn't be configured
- Used Snapdragon Profiler [4] to profile the app's high-level GPU resource usage, to confirm that memory consumption is under the 6 GB requirement
- Implemented PSNR calculations intended to verify image visual quality, which included:
    - Writing the function for decoding the ground-truth image and resizing so it has the same dimensions as the rendered image on the smartphone
    - Writing the function for "capturing" our rendered image
    - Writing the function for using the PSNR standard formula [6] to compare the ground-truth image and rendered image (both now having RGBA 32-bit floating-point format and same dimensions)
    - However, Marko couldn't debug the ground-truth image resizing to crop only the necessary parts of the image out, so it only compares the exact same camera view location as the rendered image (i.e. ground-truth image is landscape, while rendered image is portrait, so matching dimensions was difficult).

Individual Contributions - Damian Pacynko

Before the Design Review Meeting, Damian contributed to the project as he:

- Configured the project's code to to compile and deploy programs to the Android device/app using Gradle and CMake build system.
- Integrated the Vulkan graphics API within the Android development environment, enabling enhanced rendering capabilities.
    - This took significant work, because we were porting a Windows Vulkan application to Android. A lot of work was done to ensure compatibility of libraries and executables with the Android development system and device hardware.
- Established a remote server-based development environment, facilitating parallel and distributed development and testing.
    - This significantly accelerated project progress because we were no longer gated by having to pass the phone around from person to person.

Since the Design Review Meeting, Damian:

- Implemented comprehensive FPS metric tracking, allowing precise monitoring and analysis of application performance and aiding in requirement validation.
- Worked extensively on implementing a module that reads a reduced splat .ply file format.
    - Developed functionality to read and process the new .ply file format, which organized vertices based on their spherical harmonics. Successfully implemented file-reading capabilities; however, extensive shader modifications were required to fully integrate this system which exceeded the project scope.
- Introduced performance optimizations aimed at increasing FPS through dynamic resolution adjustment, resulting in smoother visual performance at the cost of slightly reduced resolution, thus balancing quality and performance according to functional requirements.
    - Gained a 2x speedup in FPS by reducing the output resolution by half. This was deemed to be a good tradeoff, since the decline in overall image quality is almost undetectable at normal views, but the increase in FPS is very noticeable, especially with large Gaussian splat models.

Acknowledgements

We would like to thank Dr. Andreas Moshovos for his support as our capstone project supervisor, especially for his guidance in defining our project's scope, and in solving and debugging major issues. We would also like to thank our administrator Afshin Poraria and our communication instructor Benjamin Kinsella for their valuable feedback on course deliverables and our project's development.

I. Executive Summary (Author: Damian)

This project successfully optimized a 3D Gaussian Splatting (3DGS) renderer tailored for Qualcomm Snapdragon 8 Gen 2 hardware, enabling high-performance rendering on mobile platforms like Meta's Quest headsets and Android smartphones without reliance on large, power-intensive GPUs. Gaussian Splatting is an advanced novel-view synthesis method that efficiently creates realistic 3D images from sparse photographic data using 3D Gaussian functions (splats). It supports interactive visualization crucial for applications such as virtual reality, robotics, and autonomous vehicles.

Key accomplishments of this project include establishing a remote server-based development environment for parallel team collaboration and implementing critical modules, such as the Touch Interpreter for mobile interaction, View-Dependent Processing for efficient Gaussian splat computation, GPU-accelerated Radix sort for optimized rendering order, and a Display Unit incorporating dynamic resolution scaling for a notable performance increase.

Performance evaluations indicated significant improvements, achieving a 91.5% increase in frames-per-second (FPS) through targeted optimizations, although falling short of the intended real-time goal of 30 FPS. Memory consumption remained well within constraints, averaging 2.95 GB against the 6 GB target, thus successfully validating the renderer's suitability for mobile hardware.

However, challenges persisted, particularly in accurately quantifying the visual quality through peak signal-to-noise ratio (PSNR) due to issues matching camera viewpoints to ground-truth images. Qualitative assessments indicated generally high image fidelity, with minor artifacts observed under intensive rendering conditions.

Future recommendations include implementing robust PSNR calculation capabilities, deeper pipeline profiling to identify and target bottlenecks, and further optimizations to improve both visual quality and rendering speed, ensuring that 3DGS rendering can fully meet real-time performance standards on mobile platforms.

II. Table of Contents

1. Introduction (Author: Marko)

This report presents the motivation, system-level and module-level design, implementation, and testing for optimizing 3D Gaussian Splatting for Qualcomm Snapdragon hardware, developed as part of the final-year design course, ECE496. The report provides a comprehensive overview of the objectives and requirements of the project, the methodologies used to verify whether we achieved or failed to achieve these requirements, and the outcome of our testing, concluding with recommendations for improvements and future work.

1.1 Background and Motivation (Author: Marko and Stefan)

Novel-view synthesis (NVS) is a task in computer graphics which consists of generating new 3D renderings of a scene, given static images of the scene from different camera angles. It has applications in robotics [7], autonomous driving [8], augmented/virtual reality [9], and more.

Among the various NVS algorithms, 3D Gaussian Splatting (3DGS) is a ground-breaking, state-of-the-art technique since it achieves a rendering frame-rate over 100 frames-per-second (FPS) [10], representing a rate one to two orders of magnitudes higher than previous algorithms (e.g. M-NeRF360, INGP) with equal or better quality and comparable training time [10].

3DGS has been predominantly implemented and optimized on systems with NVIDIA GPUs, with little work done to implement it onto Qualcomm's Snapdragon system-on-chip (SoC), which instead uses an Adreno GPU. With this implementation, 3DGS could be run on devices that use Snapdragon, like the Meta Quest 3 VR headset [1] and various mobile phones, which will reduce 3D gaussian splatting's reliance on NVIDIA's GPUs, and enable its use in the 5G mobile phone market where Qualcomm's Snapdragon chips power 26.5% of the market [11], and nearly 80% of the virtual reality headset market [12]. Qin et al. [13] achieved 30 FPS at 1440p on the Snapdragon 8 Gen 2 SoC when rendering human faces. Since the scenes we are interested in contain over 10 times as many primitives [14], they will be slower to render, leaving room for FPS improvement. While others render views on a server, which requires high internet bandwidth [15].

Thus, our project aims to first implement 3DGS rendering on a OnePlus 12R mobile phone, powered by Qualcomm's Snapdragon 8 Gen 2 SoC [16]. We will then profile the implementation to understand the memory usage the renderer requires. The latter is especially important since mobile devices have less RAM compared to NVIDIA GPUs, therefore the amount of memory that can be used by the renderer for rendering the 3DGS scene is limited[1]. Additionally, most high-performance 3DGS renderer implementations available don't specifically target Android devices and hence aren't configured for use with Android apps on mobile devices. So, understanding the need for optimizing the 3DGS implementation for Qualcomm's Snapdragon hardware and for Android devices, we will also focus on hardware specific optimizations that take advantage of Adreno's tiling rendering pipeline [17].

1.2 Project Goals and Requirements (Author: Marko)

This project will aim to implement and optimize 3DGS on Qualcomm's Snapdragon 8 Gen 2 SoC to achieve real-time rendering (30 FPS), low program memory usage (6 GB of RAM maximum), and photo realistic visual quality using peak signal-to-noise ratio (PSNR) as the metric, which compares two images to assess the quality of our rendered image compared to the ground-truth image. The details of these requirements and our objective is in Table 1.

Details of the testing methodologies for these requirements are in Section 3: Testing and Verification.

ID	Project Requirement	Description
1	Frame rate when rendering Mip-NeRF360 dataset scenes: Requirement: 30 FPS Objective: 90 FPS	**Requirement and objective:** 30 FPS is considered real-time rendering for most mobile phones [18], while 90 FPS or higher would open up usage in VR headset rendering. Testing detailed in **Section 3.3 Frame rate**.

[1] e.g. Meta's Quest 2 and 3 headset only have 6GB and 8GB of RAM respectively [20] [1], with writing to disk decreasing the hardware's lifespan. NVIDIA GPUs can have anywhere from 12GB to 48GB of RAM [25] [26].

2	Memory usage when rendering entire 3DGS scenes from Mip-NeRF360 dataset Requirement: Under 6 GB (RAM) Objective: 1 GB (RAM)	**Requirement and objective:** Rendering an entire 3DGS scene with less than 6 GB of RAM ensures that the Adreno 740 GPU has enough GPU memory to run the program [19]. The Meta Quest 2 also has a 6 GB RAM limit [20] and various Snapdragon phones have around 8 GB of RAM [2]. The goal is to use only RAM (avoiding disk storage) since this will improve the renderer's performance. Testing detailed in **Section 3.4 Memory usage**.
3	Visual quality: Must achieve peak signal-to-noise ratio (PSNR) of 22 dB when rendering Mip-NeRF360 dataset scenes. Requirement: 22 dB Objective: 25.60 dB	**Requirement and objective:** PSNR measures photo quality in decibels (dB) with a higher value indicating better quality . The original 3DGS paper achieves a PSNR of 25.60 [10], but we allow a margin of error since we are developing on a system with higher memory constraints. Testing detailed in **Section 3.5 PSNR**.
4	Hardware: Snapdragon 8 Gen 2	**Requirement :** 3DGS must be able to run on a Snapdragon 8 Gen 2 device. Specifically, we will run our renderer on a OnePlus 12R mobile phone [16] which uses this Snapdragon SoC.

Table 1. Project Requirements

2. Final Design (Author: Damian)

We chose to develop an Android application utilizing the Vulkan API due to Vulkan's advanced capabilities for high-performance, low-overhead graphics rendering, which aligns closely with the computational demands of Gaussian Splatting. Vulkan's explicit control over GPU resources enables us to efficiently manage memory, parallelism, and rendering processes. This allows us to directly target our chosen hardware, the Snapdragon 8 Gen 2 SoC.

Our Vulkan rendering pipeline for Gaussian Splatting is designed to effectively transform and display a complex 3D gaussian-based representation as a smooth, 2D image on the device. Each step in the process addresses a specific challenge inherent to rendering Gaussian splats, ensuring both performance and accuracy in the final output. This section outlines this pipeline and describes the final modules we have designed and used in our project.

2.1 System-Level Overview (Author: Damian)

The **Touch Interpreter** enables interactive exploration of the scene by translating user touch inputs into camera view updates. As the user interacts with the scene, the view matrix and projection parameters will be dynamically updated, so that the rendered image reflects the desired viewpoint.

The **View-Dependent Processing** module dynamically updates the Guassian splat cloud based on the camera's position and movement determined by user input. The module performs essential computations for each splat. To optimize performance, splats that are entirely outside the camera's view and occupy no tiles are skipped, reducing unnecessary calculations and enhancing rendering speed. The module also assigns a "key" to each splat based on its depth in the view, to be used in the following stage of the pipeline.

Depth ranking is essential because Gaussian splats are semi-transparent, and their blending depends on the order in which they are rendered. The **Radix Sort** module organizes the splats by depth, ensuring that back-to-front rendering occurs. This is critical for achieving proper alpha (α) blending, which will minimize visual artifacts.

Once the splats are sorted, the **3D-2D Projection** module transforms their 3D world coordinates into 2D screen-space representation. Gaussian splats are modeled as ellipsoids in 3D space, and this step calculates their on-screen positions and sizes based on the camera's perspective. Furthermore, the splats' colors are computed using spherical harmonics, which account for lighting effects and view angles.

Finally, the **Display Unit** takes these screen-space representations and renders them to the device's display. The splats are composited onto the screen with proper blending, producing a smooth, cohesive image that maintains the aesthetic and technical qualities of Gaussian Splatting. An optimization was made to increase the final FPS by reducing the output resolution.

Together, these steps form a pipeline tailored to the specific demands of Gaussian Splatting. They address challenges like handling high data density, ensuring real-time performance, and achieving accurate transparency and visual effects, all within the constraints of our hardware platform, the Snapdragon SoC.

2.2 System Block Diagram (Author: Damian)

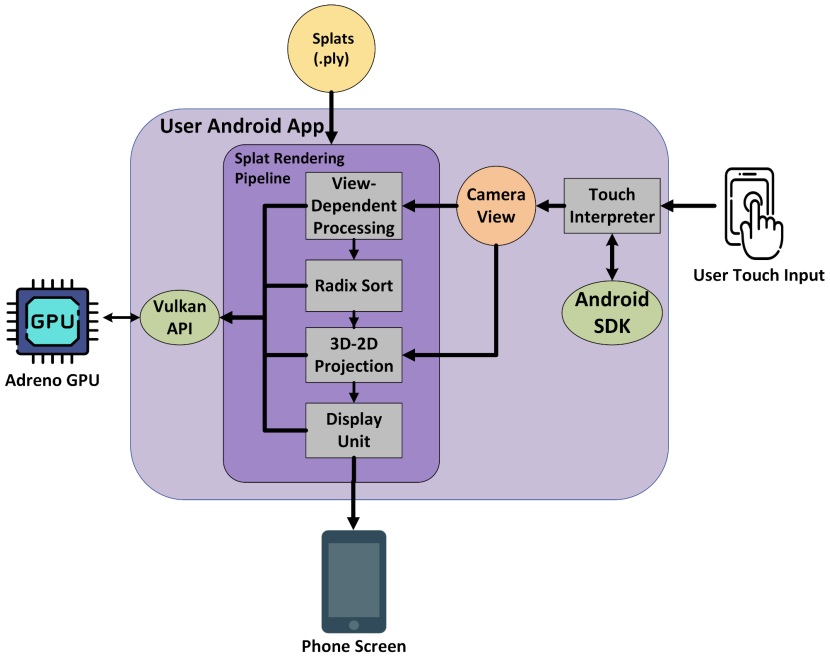

Figure 1: System Block Diagram of our Gaussian Splatting rendering application

2.3 Module-Level Descriptions (Author: All)

2.3.1 Touch Interpreter (Author: Marko)

Inputs:
- Touch input from user

Outputs:
- Updated camera view

Function:
Marko implemented this module to adapt 3DGS for usage on mobile devices. The module takes in touch input from the user to update the view of the camera based on the touch inputs registered, with the rotation and direction being the core camera parameters influenced.

Rotation is controlled by logging when the user presses down on the screen and the direction they swipe, resulting in the scene being rotated in relation to where the user swipes, i.e. if the user swipes left, the view is rotated to the left.

Direction, meaning moving the camera's position within the scene, can be controlled with the following touch inputs, with details for how it is computed in Appendix C:
- Double tapping the screen (tapping twice within 300ms) moves the camera view forward
- Holding the screen (holding for more than 500ms) moves the camera view backward
- Tapping once on the left of the screen moves the camera view left
- Tapping once on the right of the screen moves the camera view right

The touch events by the user are registered using the Android SDK GameActivity library function, *VulkanMotionEventFilter*, which filters touch events by the user, by detecting if and when an action has been made, and then storing the x-y coordinates on the screen where the action took place.

For rotation, the filter captures the initial x-y coordinates where the user pressed down, and if the user is still pressing down while moving across the screen, the filter calculates the new coordinates of where the user is relative to the original coordinates. For direction, the filter logs where and when was the last time the user pressed down, to determine which direction to move in.

The touch events detected are then passed to the renderer's handleInput function using the GameActivity function, *android_app_set_motion_event_filter*. After enabling touch event filtering, the camera view will get updated every time the user makes a valid touch event on the screen, with every new camera view then being used by the view-dependent processing and 3D-2D projection modules in the rendering pipeline.

2.3.2 View-Dependent Processing (Author: Stefan)

Inputs:

- View configuration of current camera
- Gaussian Splat model (.ply) (a complete list of splats in the scene)

Outputs:

- List splats that appear in the current camera view, and their view-dependent properties
- List of view-dependent "key" of each splat in the camera-view

Function:

As the camera moves around the scene (based on user input), the 2D image displayed on screen must update accordingly. To make this possible, this module uses updated camera information to calculate several properties of each splat in the scene. This includes:

- Which screen "tiles" the splat occupies (a tile is a 16x16 pixel section of the screen)
- Splat's depth from the camera
- The view-dependent color of the splat
- Etc.

With this information, this module creates a "key" for each splat to be used by the following Radix Sort module. This 32 bit key is composed of a 16 bit integer indicating the screen tile the splat occupies, followed by a 16 float which stores the depth of the splat from the camera.

If a splat is found to not occupy any screen tiles, none of the other calculations (e.g. depth, color, and key) are performed for it. This allows the subsequent modules (e.g. Radix Sort, 3D-2D projection) to ignore them, since they are outside the camera's view frustum and would not affect the displayed image. This significantly reduces the computation required for each frame, thus speeding up the algorithm.

It should be noted that each splat's key was originally 64 bits, with 32 bit tile information and 32 bit depth information. However, Stefan found that this resulted in the renderer program crashing or truncating relevant information, leading to an incomprehensible scene. Because of this, Stefan modified this module (and the Radix Sort module) to use 32 bit keys. See Appendix E to understand why the data loss from this truncation is justified

2.3.3 Radix Sort (Author: Stefan and Damian)

Inputs:
- List of splats in the current view and their associated "keys" (given from the view-dependent processing module)

Outputs:
- List of the same splats, but sorted in ascending order of depth from the camera

Function:
Given each viewable splat's key calculated in the view-dependent processing stage, this module sorts the splats using an efficient parallel GPU Radix Sort algorithm. Since these keys contain both tile and depth information, this module effectively outputs a list of viewable splats both in ascending order of depth from the camera, but also divided by which tiles they occupy.

This sorting crucially allows the 3D-2D projection stage to α-blend viewable splats so that the 2D output image properly captures the color of translucent splats intersecting.

2.3.4 3D-2D Projection (Author: Damian)

Inputs:
- List of splats that appear in the current camera view, and their view-dependent properties
- Camera view
- Camera's projection parameters

Outputs:
- 2D screen-space Gaussian representations

Function:
The 3D-2D projection module transforms visible 3D Guassian splats into their 2D representations suitable for rendering on screen. This process involves calculating each splat's exact position in screen-space based on current camera parameters. GPU shaders are utilized extensively for this task, performing real-time computations to apply the camera's projection and view matrices to each splat, efficiently converting 3D coordinates into accurate 2D pixel locations.

Additionally, shaders compute the colour of each splat by evaluating spherical harmonics, which encode view-dependent lighting effects. By leveraging these shader programs. The module swiftly handles complex mathematical operations, producing the final, visually coherent screen-space image displayed to the user.

2.3.5 Display Unit (Author: Damian)

Inputs:
- 2D screen-space Gaussian representations

Outputs:
- Display image on screen

Function:
This stage takes the final rendered frame, composed of thousands (or even millions) of Gaussian splats, and handles the critical task of displaying it on the device screen. The Display Unit makes use of Vulkan API calls and GPU shaders to perform the necessary final drawing steps.

To balance performance and quality, the Display Unit uses a tiled rendering approach, dividing the framebuffer into 16x16 pixel regions. Each tile is processed by a dedicated shader thread, allowing the GPU to parallelize output generation effectively.

Damian's optimization within this module was to reduce the resolution of the final image by merging every 2x2 block of pixels into a single color output. Conceptually, the splat data is still generated at full resolution, but instead of performing calculations and writing individual colors for each pixel, a shader pass in the Display Unit combines four adjacent pixels into one. This effectively halves the resolution in both width and height, yielding a quarter of the total pixel output that would otherwise be required. However, the following outcomes occur:

1. **Reduced Pixel Operations:** Because the pipeline only outputs one colour every 2x2 area, the GPU performs fewer calculations and write operations to the framebuffer.
2. **Improved Framerate:** By cutting down the total number of pixels that need to be written to the framebuffer, the optimization provides roughly a 2x improvement in FPS, as seen in Table 3. This boost is particularly noticeable in complex scenes where Gaussian splats can dominate performance if processed at full resolution.
3. **Minimal Image Quality Impact:** When viewed at standard zoom levels, there is little to no difference in quality. The Gaussian-based rendering method naturally smooths out details, and downsampling by a factor of two tends not to introduce obvious artifacts. Only when zoomed in very close does any loss of detail become apparent.

More details on image quality trade-offs are outlined in Appendix D.

2.4 Final Design Assessment (Author: Damian)

Our final renderer design successfully accomplishes the primary objective of effectively rendering Gaussian splat point clouds on Android devices. Through reduced pixel operations, we achieved a commendable balance between visual fidelity and rendering speed, consistently maintaining around 20 FPS across various scenes. This frame rate significantly enhances user interaction, enabling smooth navigation and exploration within the rendered environments, marking a notable accomplishment given the complexity of Gaussian Splatting.

Despite these successes, certain visual artifacts occasionally emerge as a consequence of the reduced pixel operations. While generally minor and unobtrusive, these artifacts highlight inherent trade-offs in our optimization strategies and hardware resources. Additionally, a limitation in our current implementation is the lack of a robust quantitative method to measure image quality. Without standardized image quality metrics such as PNSR, it becomes challenging to objectively assess and compare the effectiveness of various optimizations applied through the rendering pipeline.

Overall, the final design delivers a practical and interactive experience, allowing users to fluidly traverse and interact with complex Gaussian-based point cloud scenes. Future work incorporating precise image quality assessments and targeted optimizations based on scientific metrics will further enhance both the performance and visual accuracy of the renderer.

3. Testing and Verification (Author: Stefan and Marko)

The testing and verification of our requirements is detailed in this section. Each sub-section covers one of the requirements, whether or not it passed the metrics in Table 2, and a detailed overview of how it was tested and what changes had to be made in order to validate the requirement.

3.1 Testing Table

ID	Requirement	Method	Result and Proof
1	Frame Rate: The renderer shall have an (average) frame rate of 30 frames per second (FPS) when rendering Mip-NERF360 dataset scenes [18].	TEST: Measure average FPS over 30 seconds for 3 different scenes	FAIL. 17.55 FPS average. See Table 3 in Section 3.3.
2	Memory Usage: The renderer shall not use more than 6 GB of memory when rendering Mip-NeRF360 dataset scenes [2][20].	TEST: Measure total memory consumed to render an entire 3DGS scene using the Snapdragon profiler	PASS. 2.382 GB average. See Table 4 in Section 3.4.
3	Image Quality: The renderer shall have an average peak signal-to-noise ratio (PSNR) of at least 22 dB when rendering Mip-NeRF360 dataset scenes [10].	TEST: Measure the PSNR between ground truth images and the corresponding rendered view.	FAIL. Unable to measure accurately. See Section 3.5.
4	Hardware: The renderer shall run on hardware powered by the Snapdragon 8 Gen 2 SoC	Evaluation on final product. Renderer runs correctly on specified hardware	PASS, Verified. [16].

Table 2: Testing Table

3.2 Development and Testing Environment (Author: Damian)

Damian established a remote server environment, enabling all team members to concurrently develop, test, and verify their modules directly on the Android device. This infrastructure significantly enhanced our workflow and greatly accelerated our overall development and testing process.

3.3 Frame rate (Author: Stefan)

Damian and Stefan used a high-resolution software timer (the C++ std::chrono library) to count the number of complete frames we generate over a single second. We then average this measurement over 30 seconds, while moving the camera around various scenes. Here, our camera movement mimicked realistic user behaviour (i.e. inspecting various objects in the scene from up close and from afar). In this 30 second measurement, we captured periods of both high and low rendering workloads, depending on if the camera was pointed at a large or small amount of splats.

In our baseline implementation gathering measurements from 3 Mips-NeRF360 scenes, we found the average FPS to be 9.16 FPS, with additional details in Table 3. Additionally, Damian implemented an optimization in the Display Unit stage to improve FPS by reducing image quality (see Section 2.3.5). With this optimization, we measured an average of 17.55 FPS, marking a 91.5% improvement. However, this means that we did not meet our requirement of 30 FPS, showing there are still FPS optimizations needed for this application to be considered “real-time” for mobile applications [18].

Scene	FPS	FPS w/ optimization
Truck	9.80 FPS	20.29 FPS
Bicycle	8.47 FPS	14.32 FPS
Bonsai	9.20 FPS	18.05 FPS
Average	**9.16 FPS**	**17.55 FPS**

Table 3: Computed FPS Metrics.

3.4 Memory Usage (Author: Marko)

To validate that rendering an entire 3DGS scene consumes less than 6 GB of RAM, Marko used the Snapdragon Profiler to analyze the performance of the Vulkan renderer in rendering three separate scenes on the Snapdragon 8 Gen 2 processor. The Snapdragon Profiler was used to measure GPU activity in real-time to provide an analysis of the Vulkan renderer's workloads.

To measure the memory used to render an entire 3DGS scene, the renderer was run on the Adreno 740 GPU inside the Snapdragon 8 Gen 2 chip, and while inside a particular 3DGS scene the testing consisted of imitating a user's actions by moving the camera around the scene, i.e. rotating and moving the camera's position, and recording the GPU memory consumed by the app during this time. This test was then reproduced with two other scenes from the Mip-NeRF 360 dataset, each with varying .ply file sizes. Each test concluded with a memory usage below 6 GB as detailed in Table 4, and then the memory usage for each test was averaged out to determine that the average 3DGS scene consumes 2.95 GB of RAM thereby satisfying our 6 GB requirement.

Aside from the real-time analysis, the profiler also should have allowed for deeper GPU analysis using trace and snapshot capture which provides more detailed metrics over a span of several seconds. However, the Snapdragon Profiler was unable to inject its profiling code into the Android app when performing trace and snapshot capture, resulting in the app crashing from a segmentation fault, therefore Marko was unable to complete this deeper analysis of the renderer pipeline. Hence, it was not possible to use the profiler to identify performance bottlenecks in the pipeline to optimize, as the profiler only provided real-time analysis which had limited metrics available to analyze.

Marko's research into the official Snapdragon Profiler User Guide [21] and various other sources that Qualcomm offers for troubleshooting [22], led to the conclusion that OnePlus's Android-based OS, OxygenOS—even after enabling Developer Options and enabling "Enable GPU Debug Layers" and "USB Debugging" as directed in the user guide—did not support profiling the Vulkan workloads in detail due to the OS's advanced security layers protecting the GPU's performance from being read.

Scene	Peak Memory Usage	Number of Gaussians
Truck	2.4 GB	1,732,378
Bicycle	4.75 GB	3,616,103
Bonsai	1.7 GB	1,157,141
Average	**2.95 GB**	**2,168,540**

Table 4. Computed Peak Memory Usage.

3.5 Peak Signal-to-Noise-Ratio (PSNR) (Author: Marko and Stefan)

To validate that the rendered scene is of sufficient visual quality, a PSNR calculation should compare a ground truth test image (a real image of the scene, not used to train the model) to a rendered image of the scene taken from the same camera angle and position. Therefore, Marko implemented functions to retrieve the rendered image, convert it into a 32-bit floating point format to be compared with the ground truth JPEG that is also converted to the same format, and ultimately perform the PSNR calculation, the code for which is in Appendix F. Marko made the choice to use 32-bit floating points because each pixel is normalized to be between 0 and 1, which was a more coherent format for the standard PSNR formula to calculate with as it is also a floating point function.

However, despite Stefan's efforts, we were not able to move the renderer camera to a position corresponding to a ground truth image. While our scenes included ground truth pose information, loading these in our renderer displayed a different part of the scene, making PSNR calculations impossible. It's unclear if this was a mistake with the renderer, or the camera pose data, since Stefan and Marko were unable to find a ground truth 3DGS renderer that allowed them to troubleshoot these camera poses.

While no PSNR values were calculated, we did evaluate the renderer's image quality qualitatively. Figures 2 and 3 show that areas with low gaussian density, (e.g. the background carpet or hardwood floor) rendered very realistically and matched ground truth. However, we also observe that while areas with high gaussian density (e.g. the bonsai tree and table it sits on)

have a recognizable form, they look almost translucent, which is noticeable in both Figure 2 and 3. This latter effect suggests that there may be a persistent bug in the Radix Sort stage incorrectly displaying far gaussians as near.

In Figure 2 and 3, the left-side images are the ground truth image (Mips-NeRF360 “bonsai” scene) while the right-side images are our 3DGS rendering in a similar position.

Figure 2. Qualitative analysis of renderer’s image quality

Figure 3. Another qualitative analysis of renderer’s image quality from a different camera view.

It should also be noted that Damian’s FPS optimization is expected to lower the PSNR measurement. Since we were unfortunately unable to position the camera correctly in time for this report, we leave measuring the PSNR of both the optimized and unoptimized renderer as future work.

4. Summary and Conclusions (Author: Marko)

Our project aimed to optimize current 3D Gaussian Splatting algorithms for implementation on Qualcomm's Snapdragon hardware, specifically the Snapdragon 8 Gen 2 SoC containing the Adreno 740 GPU. Metrics of particular importance, that we defined to be our requirements for validating whether our implementation is a success or requires further improvement in the future were:

1. Real-time rendering measured by achieving an average of 30 FPS while moving inside the rendered 3DGS scene
2. Low program memory usage by requiring the rendering of the 3DGS scene to consume less than 6 GB of RAM to ensure our implementation is compatible with mobile devices that don't have powerful GPUs with large memory resources.
3. Photo realistic visual quality of our rendered images of the 3DGS scene measured by achieving a consistent PSNR of 22 dB through comparing rendered images to their ground-truth counterparts.

After thoroughly testing our design, it is evident that while we achieved significant progress in certain areas, not all our initial requirements were met. We did achieve the requirement for memory consumption as it consistently remained below the 6 GB threshold, averaging 2.95 GB across various scenes. Furthermore, targeted optimizations—such as the resolution reduction in the Display Unit module—resulted in a nearly 2x improvement in FPS compared to before the optimization, increasing the average frame rate from 9.16 to 17.55 FPS. Although this represents a substantial improvement, it still falls short of our real-time performance goal of 30 FPS.

The challenges associated with accurately measuring PSNR prevented us from quantifying visual quality through a standard metric. Qualitative analysis, however, indicates that regions with lower Gaussian density are rendered with impressive quality, while areas with higher density display a slight translucency that suggest potential issues in the depth sorting or alpha blending.

Despite these setbacks, our findings validate several core design concepts. The successful adaptation of key modules—including the Touch Interpreter, View-Dependent Processing, and Radix Sort—for mobile platforms demonstrates the viability of a Vulkan-based rendering pipeline on Snapdragon hardware. Past 3DGS implementations have rarely been developed for

Android devices, even the baseline implementation [3] that we built off, wasn't configured for Android devices. Hence, the changes made to configure and optimize the renderer for an Android device and development environment that is running on the Snapdragon SoC, is in itself an important achievement. The project also highlights the importance of robust profiling tools; Qualcomm's limited documentation and tool support for Android have underscored the need for further exploration of hardware-specific optimization techniques for profiling bottlenecks in design.

In summary, while our renderer did not fully meet the ambitious targets for real-time performance and photo realistic image quality, it has nonetheless paved the way for implementing the state-of-the-art computer graphics technique, that is 3D Gaussian Splatting, in mobile devices. The insights gained regarding module-level optimizations provide a solid foundation for subsequent improvements. Future work should focus on refining the rendering pipeline—particularly in shader computation and sorting algorithms—while incorporating a reliable method for quantitative image quality assessment. Overall, this project contributes valuable knowledge to the field and offers a promising starting point for realizing high-performance, real-time 3DGS rendering on mobile platforms that use Qualcomm's Snapdragon SoCs.

5. Future Work (Author: Damian)

To further enhance our project, we propose several key areas for development and investigation:

1. **PSNR Calculation**: Integrating a robust Peak Signal-to-Noise Ratio (PSNR) calculation framework would significantly benefit the project by providing a quantitative measure of rendered image quality. Implementing PSNR would allow future developers to objectively evaluate visual fidelity against ground truth reference, thereby guiding further refinements in rendering accuracy that our qualitative analysis just doesn't provide.
2. **Identifying Bottlenecks:** Conducting detailed profiling to capture execution times of each stage within the rendering pipeline would be highly beneficial. Accurately measuring and logging these timings will help future developers quickly identify performance bottlenecks and areas that demand optimization, thus improving the overall efficiency and performance of the application. As the famous Sir Tony Hoare once said, "premature optimization is the root of all evil".
3. **Exploring and Evaluating Optimizations:** Based on identified bottlenecks, experimenting with targeted optimization strategies would be crucial. Potential areas to explore include shader improvements, advanced memory management techniques, enhanced parallel processing, GPU workload balancing, and more sophisticated culling methods. Thorough evaluation of these optimizations is essential, as it facilitates an iterative approach, allowing developers to incrementally refine each optimization and objectively measure improvements in responsiveness, latency reduction, and overall user experience.

6. References

[1] "Data sheet Meta Quest 3 Specifications." Accessed: Sep. 17, 2024. [Online]. Available: https://www.uni-giessen.de/de/studium/lehre/projekte/nidit/goals/quest3/datasheet_quest-3.pdf

[2] "Snapdragon 8 Gen 2 Mobile Phones (Oct 2024) | 91mobiles.com," 91mobiles.com, 2024. https://www.91mobiles.com/list-of-phones/snapdragon-8-gen-2-phones(accessed Oct. 11, 2024).

[3] shg8, "3DGS.cpp," GitHub repository. [Online]. Available: https://github.com/shg8/3DGS.cpp. (Accessed: Mar. 21, 2025).

[4] "Snapdragon Profiler | Qualcomm Developer," *Qualcomm.com*, 2024. https://www.qualcomm.com/developer/software/snapdragon-profiler (accessed Sep. 17, 2024).

[5] "Qualcomm AI Engine Direct SDK", Qualcomm Inc. Available at: https://www.qualcomm.com/developer/software/qualcomm-ai-engine-direct-sdk (accessed Oct. 6, 2024).

[6] David L., "Image quality assessment," DavidL Wiki. [Online]. Available: https://wiki.davidl.me/view/Image_quality_assessment. (Accessed: Mar. 21, 2025).

[7] G. Lu, S. Zhang, Z. Wang, C. Liu, J. Lu, and Y. Tang, 'ManiGaussian: Dynamic Gaussian Splatting for Multi-task Robotic Manipulation', arXiv [cs.RO]. 2024.

[8] X. Zhou, Z. Lin, X. Shan, Y. Wang, D. Sun, and M.-H. Yang, 'DrivingGaussian: Composite Gaussian Splatting for Surrounding Dynamic Autonomous Driving Scenes', arXiv [cs.CV]. 2024.

[9] J. Ye, Z. Zhang, Y. Jiang, Q. Liao, W. Yang, and Z. Lu, 'OccGaussian: 3D Gaussian Splatting for Occluded Human Rendering', arXiv [cs.CV]. 2024.

[10] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D Gaussian Splatting for Real-Time Radiance Field Rendering," ACM Transactions on Graphics, vol. 42, no. 4, Jul. 2023, [Online]. Available: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

[11] "Omdia: MediaTek outgrowing Qualcomm Snapdragon in the 5G smartphone market," Omdia, 2021. https://omdia.tech.informa.com/pr/2024/jul/omdia-mediatek-outgrowing-qualcomm-snapdragon-in-the-5g-smartphone-market (accessed Oct. 15, 2024).

[12] E. Oyedeji, "CHART: Meta, Pico, and DPVR controlled over 90% of VR headset shipments in Q2 2024," Techloy, Oct. 02, 2024. https://www.techloy.com/chart-meta-pico-and-dpvr-controlled-over-90-percent-of-vr-headset-shipments-in-q2-2024/ (accessed Oct. 15, 2024).

[13] D. Qin et al., 'Instant Facial Gaussians Translator for Relightable and Interactable Facial Rendering', arXiv [cs.GR]. 2024.

[14] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, and G. Drettakis, 'Reducing the Memory Footprint of 3D Gaussian Splatting', Proceedings of the ACM on Computer Graphics and Interactive Techniques, vol. 7, no. 1, pp. 1–17, May 2024.

[15] Facebook VR, "Meta Horizon Hyperscape Demo," Meta.com, 2024. https://www.meta.com/experiences/meta-horizon-hyperscape-demo/7972066712871980/?intern_source=blog&intern_content=connect-2024-keynote-recap-quest-3s-llama-3-2-ai-wearables-mixed-reality (accessed Oct. 08, 2024).

[16] OnePlus, "12R Specs," OnePlus. [Online]. Available: https://www.oneplus.com/ca_en/12r/specs. (Accessed: Mar. 21, 2025).

[17] "The Evolution of High Performance Foveated Rendering on Adreno," *Qualcomm.com*, 2024.

https://www.qualcomm.com/developer/blog/2021/07/evolution-high-performance-foveated-rendering-adreno (accessed Sep. 17, 2024).

[18] “Metal Best Practices Guide: Frame Rate (iOS and tvOS),” Apple.com, Mar. 27, 2017. https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBestPracticesGuide/FrameRate.html. (accessed Oct. 08, 2024).

[19] “Qualcomm Adreno 740 Benchmark, Test and specs,” *Cpu-monkey.com*, 2024. https://www.cpu-monkey.com/en/igpu-qualcomm_adreno_740 (accessed Sep. 17, 2024).

[20] “Blog | Build with Meta Horizon OS,” Meta.com, 2024. https://developers.meta.com/horizon/blog/getting-a-handle-on-meta-quest-memory-usage/ (accessed Oct. 11, 2024).

[21] Qualcomm Inc., “Snapdragon Profiler,” Qualcomm Developer Network. [Online]. Available: https://docs.qualcomm.com/bundle/publicresource/topics/80-78185-2/sdp.html. (Accessed: Mar. 21, 2025).

[22] Qualcomm Inc., “Snapdragon Profiler Support,” Qualcomm Developer. [Online]. Available: https://www.qualcomm.com/developer/software/snapdragon-profiler/support. (Accessed: Mar. 21, 2025).

[23] “iQOO 11 Basic.”, iQOO. Available at: https://www.iqoo.com/en/products/param/iqoo-11 (accessed Nov. 15, 2024).

[24] Hongkong VT Store, “VIVO IQOO Neo 9 5G Snapdragon 8 Gen 2 5160mAh Battery 120W SuperVOOC 50MP IMX920 OIS 6.78Inch AMOLED 144Hz NFC OTA,” AliExpress, (accessed Nov. 15, 2024).

[25] “GeForce RTXTM 4080 16GB GAMING X TRIO.” Accessed: Oct. 11, 2024. [Online]. Available: https://storage-asset.msi.com/datasheet/vga/global/GeForce-RTX-4080-16GB-GAMING-X-TRIO.pdf

[26] “NVIDIA RTX A6000.”, NVIDIA, Accessed: Oct. 11, 2024. [Online] Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/proviz-print-nvidia-rtx-a6000-datasheet-us-nvidia-1454980-r9-web%20(1).pdf

7. Appendices

Appendix A: Gantt Chart History

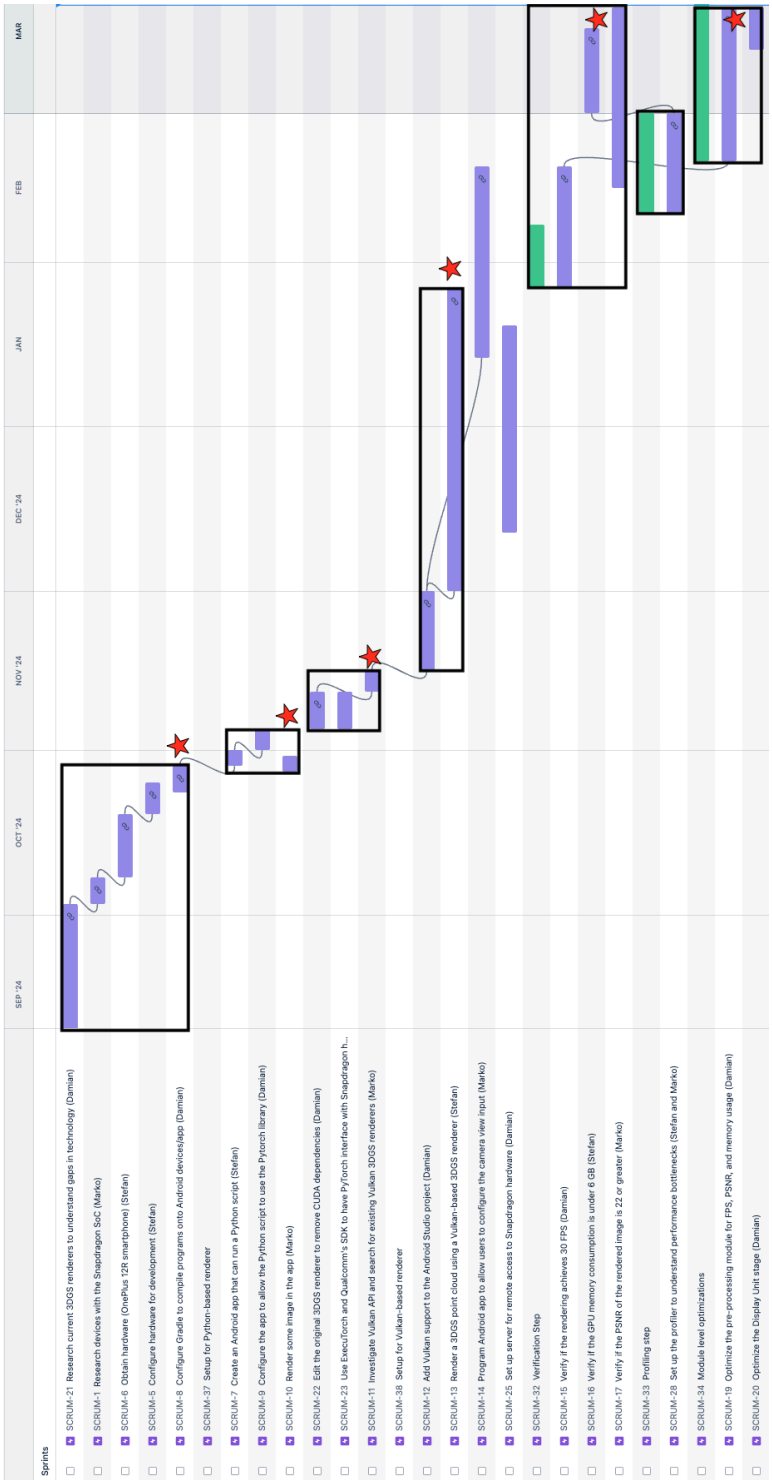

Figure 4. Updated Gantt Chart

Figure 5. Previous Gantt Chart

Please note**:**

- Green tasks (the ones without an explicit assigned team member) serve as the header or category for grouper/related tasks.
- The red stars indicate our milestones. There is an unshown milestone at the end of the final task.

Appendix B: Financial Plan

The financial plan and budget details for the 3DGS project are detailed below in Table 1, as well as contingency arrangements in the case that our supervisor was not able to fund our project. The cost of labour for an hour of work each team member puts in was calculated to be $30 per hour based on Marko's PEY internship experience in the field of graphics and hardware.

Capital Equipment						
Item	**Priority**	**Cost/unit**	**Quantity**	**Total Cost**	**Requires Funding**	**Kept/Paid for by Students**
OnePlus 12R mobile phone	1	$904	1	$904	Yes	No
Total Capital Equipment				**$904**		
Total Requiring Funding				**$904**		
Funding						
Students ($100 each)	$300					
Supervisor	$604					
Other (Specify)	$0					
Request from Design Centre	$0	(Supervisor covering all expenses)				
Total Funding	**$904**					
Student Labour						
Item	**Cost/unit**	**Quantity (# of hours)**	**Total Cost**			
Damian Pacynko	$30	199.5	$5,985			
Stefan de Lasa	$30	174	$5,220			
Marko Ciric	$30	169.5	$5,085			
Total Student Labour (unfunded)			**$16,290**			
Total Cost of Project			**$17,194**			
Total Cost Requiring Funding			**$904**			

Table 5. Financial Plan for 3DGS Project

Priority 1) OnePlus 12R: In case we are not able to obtain funds to purchase the OnePlus 12R phone, we propose to find a cheaper phone, such as the Vivo iQOO 11 Neo 9 [23] [24], that still uses the Qualcomm Snapdragon 8 Gen 2 processor, has over 12 GB of RAM, and can be developed on (has ability to deploy an Android app to it). This would not require any changes to our technical solution or work plan as they are dependent on the processor, not the phone model itself. However, this alternative would lower the funds needed in our Financial Plan.

The only aspect of our project that required funding was the physical hardware (OnePlus 12R phone), and the cost of our team's labor. The rest of the software we are using (e.g. pre-calculated 3DGS point cloud, Vulkan APIs, Android Studio, Snapdragon profiler) is free.

How the number of hours each team member dedicated was calculated using the Gantt chart in Figure 6.

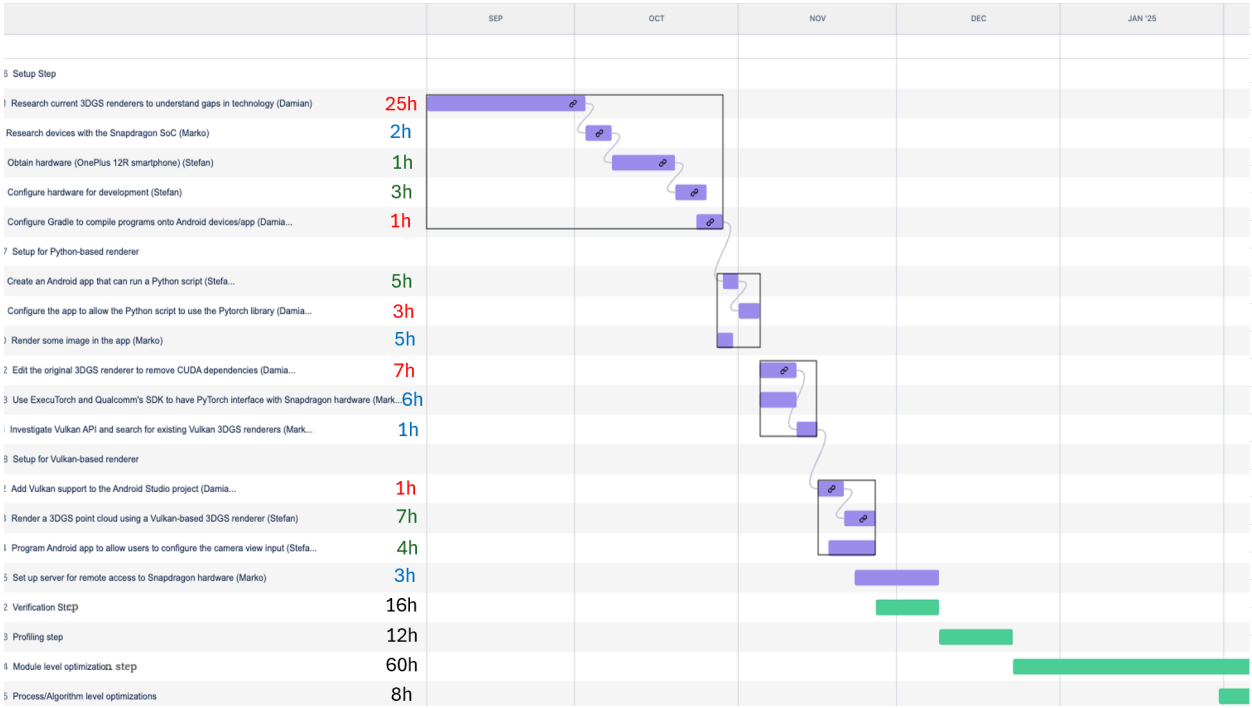

Figure 6. Estimated number of hours needed by each team member to complete project.

For the green sections each member is estimated to need an additional 96h total to complete the tasks. This estimate led to Damian dedicating 133 hours, Stefan dedicating 116 hours, and Marko dedicating 113 hours. These estimated numbers were then scaled by 1.5 to account for underproductivity and overtime work, leading to the numbers in the Financial Plan. Also to note, Damian has a much higher number of hours because he was "assigned" the initial research of the project, adding to 25 hours, but in reality all of us did research, however Damian took charge of that section because of his initial interest in the project and hence was labeled "responsible".

Appendix C: Direction computation in handleInput function for touch interpreter module

The direction is computed by:

1. Transforming the relevant vector for movement, i.e. (0, 0, -1) for forward movement, with the current camera's 3x3 rotation matrix, to determine the camera's direction in real coordinates
2. Normalizing the result so it is a unit vector in real space
3. Multiplying the resultant unit vector by a speed factor and then updating the camera view with the movement. Additionally, the speed of which the camera's position moves in the scene is configurable using the speed factor, with a default value (0.2) determined through testing as a user-friendly speed.

Appendix D: Post FPS optimization image quality comparison

As we can see below, only once we zoom in far enough, do we see the pixelation that our resolution optimization causes. However, we deem this to be a good tradeoff based on the significant jump in FPS it provides.

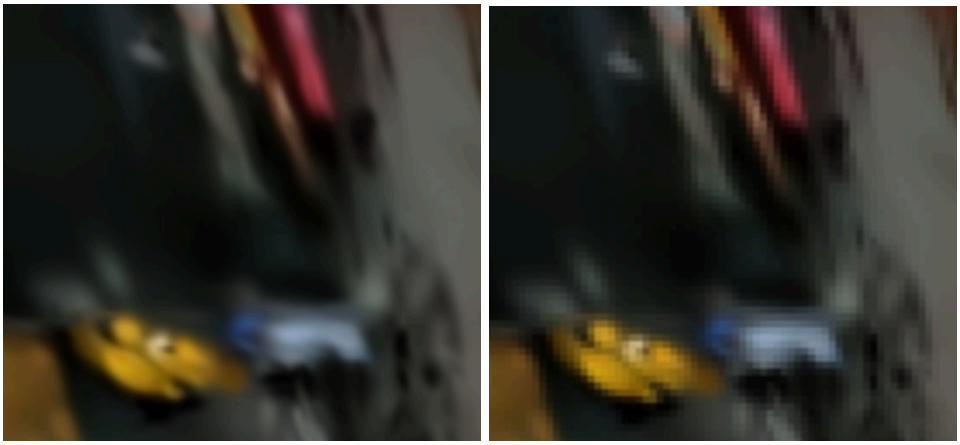

Figure 7. Comparing image quality between pre-FPS and post-FPS optimization.

In Figure 7, the left-side image is the original, full resolution rendering, achieving 10 FPS, while the right-side image is the half-resolution rendering, achieving 16 FPS, a 6 FPS increase.

Appendix E: Data truncation justification

Because the Snapdragon 8 Gen 2's GPU does not have access to 64 bit integer calculations, our implementation of the renderer's sorting pipeline needed to convert the 64 bit "keys" being sorted into 32 bits values instead. This meant truncating 32 bits of screen index values and 32 bits of depth values into 16 bits each. We believe this data loss will not have negative effect on the final scene being rendered since:

1. Our phone screen is 1080 x 2376 pixels, corresponding to 10,024 16x16 tiles. Since we only need 14 bits to store the maximum tile value (10,023), the new 16 tile data should suffice.
2. The new depth data can store data in the range of [-65,504, +65,504]. We observed a maximum splat depth of 26 when rendering our various Mips-NeRF360 scenes. Therefore, 16 bits should suffice for our depth data.

Appendix F: PSNR calculation function in renderer code

```cpp
float Renderer::computePSNR(const std::vector<float>& rendered, const std::vector<float>& groundTruth, float maxValue) {
    if (rendered.size() != groundTruth.size()) {
        throw std::runtime_error("Rendered and ground truth images must have the same number of pixels");
    }

    double mse = 0.0;
    for (size_t i = 0; i < rendered.size(); ++i) {
        double diff = rendered[i] - groundTruth[i];
        mse += diff * diff;
    }
    mse /= rendered.size();

    if (mse == 0.0)
        return std::numeric_limits<float>::infinity(); // perfect match

    float psnr = 20.0f * std::log10( lcpp_x: maxValue) - 10.0f * std::log10( x: mse);
    return psnr;
}
```

Figure 8. PSNR calculation function in Renderer.cpp.

$$
\begin{aligned}
PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\
&= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\
&= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE).
\end{aligned}
$$

Figure 9. Standard PSNR formula that code is based on.