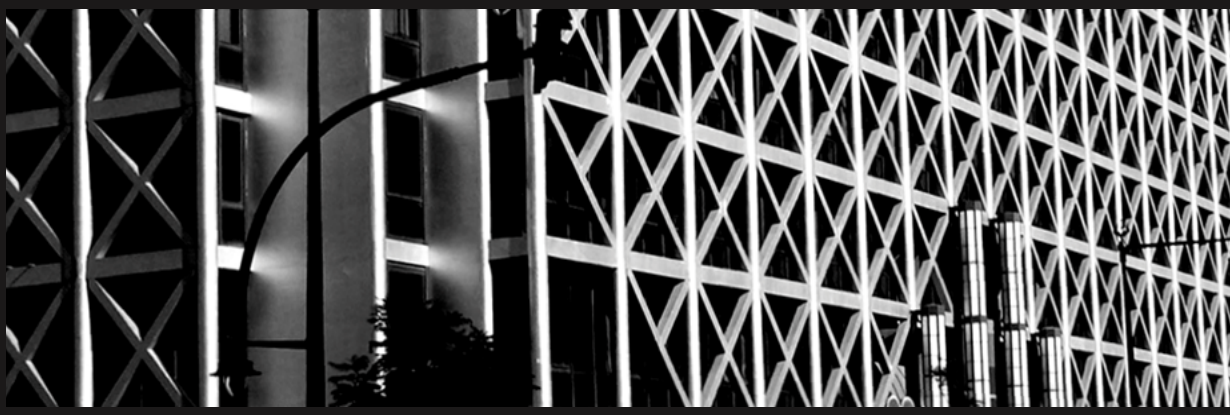
MESTRADO EM
CREATIVE COMPUTING AND ARTIFICIAL INTELLIGENCE



Faculdade de Design,
Tecnologia e Comunicação
Universidade Europeia

Lucas Terral Freire de Albuquerque Feijó

Volumetric Reconstruction using iPhones with Simultaneous Data Capture for Post-Production Adjustments

Orientador(es): Prof. Dr. Pamela Teubig, Prof. Dr. João A. Dias

IADE, Faculdade de Design, Tecnologia e Comunicação da Universidade Europeia

Faculdade de Design,
Tecnologia e Comunicação
Universidade Europeia

2025

LUCAS TERRAL FREIRE DE ALBUQUERQUE FEIJÓ

VOLUMETRIC RECONSTRUCTION USING IPHONES WITH SIMULTANEOUS DATA CAPTURE FOR POST-PRODUCTION ADJUSTMENTS

Faculdade de Design, Tecnologia e Comunicação
Universidade Europeia

2025

LUCAS TERRAL FREIRE DE ALBUQUERQUE FEIJÓ

VOLUMETRIC RECONSTRUCTION USING IPHONES WITH SIMULTANEOUS DATA CAPTURE FOR POST-PRODUCTION ADJUSTMENTS

Dissertação apresentada ao IADE - Faculdade de Design, Tecnologia e Comunicação da Universidade Europeia, para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Creative Computing and Artificial Intelligence realizada sob a orientação científica da Doutora Pamela Teubig, professora assistente do *IADE* e do Doutor João A. Dias, professor do IADE.

Acknowledgements

A special thank you is due to a number of people who have helped me to varying degrees throughout this process.

To my primary advisor Pamela Teubig; for the incentive in pursuing this research, for the guidance while adhering to academic principles and formats throughout my writing and for her appreciated patience with a working student that, more than once, struggled to balance priorities. (And also for the overtime she will most likely put on while reviewing this text.)

To my advisor João A. Dias; for valuable technical insight during meeting encounters and for the numerous suggestions for conference and publication submissions.

To IADE professors André Sabino, Cláudia Sevivas, Eulerson Rodrigues and Fábio Guimarães; for demonstrating enthusiasm about the project, for the occasional but insightful look into code files, for helping me view an unconfirmed hypothesis as still valuable knowledge and for raising important questions during mid-project milestone, respectively.

To IADE lab technician Bernardo Boffa-Molinar for being solicitous and kind during my research.

To Greta Liutkute, my professional manager during this period, that incentivized my personal development and empathized with the course demand the times it spiked.

To my family, friends and partner; for being my safe haven, my greatest support, and for always showing up even when I was determined to go full cave mode for productivity.

Keywords Computer Vision, Gaussian Splatting, Radiance Fields, NeRFs, Volumetric Reconstruction

abstract This study investigates the integration of multi-lens setups, specifically wide and ultrawide lenses, into Gaussian Splatting frameworks to enhance 3D scene reconstruction. By leveraging the fixed extrinsic relationship between the lenses, the proposed approach combines multiple RGB perspectives to improve depth accuracy, geometric precision, and texture fidelity. Unlike traditional single-lens approaches, this method utilizes the expanded field of view provided by the ultrawide lens as auxiliary data to complement the wide lens. No particular benefit was presented from using the specific approach this study (with limitations accounted for in the full text), yet during testing, reconstructions that used solely the ultrawide lens scored very competitive metrics, specially given then their sample was lower both image count and image resolution, important variables for such reconstructions. This work highlights the potential of implementing a native fisheye camera model in the state-of-the-art Gaussian Splatting pipeline.

Glossary

COLMAP A Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline that turns image groups into 3D reconstructions.

EXIF Metadata Image metadata file containing camera settings and specs in the time of capture.

Feature Matching A process that identifies corresponding features in an image group.

Fisheye Lens An ultrawide angle lens that introduces strong radial distortion to capture a wider fields of view.

Gaussian Splatting A rendering technique that uses 3D Gaussians for fast radiance field rendering.

LiDAR Light Detection and Ranging – a laser pulse technology that measures distances and maps 3D environments.

MSE Mean Squared Error – statistical measurement of the average squared difference between ground truth and output.

OpenCV An open-source computer vision library used in image processing, camera calibration, and feature detection tasks, amongst others.

PSNR Peak Signal-to-Noise Ratio – a quality metric converted from MSE, noted in decibels.

Pinhole Camera Model The simplest projection model, for no-distortion lens.

Remapping A process that adjusts image pixels to new locations, used for undistortion.

SSIM Structural Similarity Index Measure – a perceptual metric to quantify image similarity, using luminance, contrast, and structure.

Ultrawide Lens A lens with a very short focal length, with a large field of view and intense edge distortion.

Undistortion A process to correct lens distortions (as barrel or fisheye) by remapping images using camera intrinsics data.

View Synthesis Generating novel view images from existing images, often used in neural rendering and reconstruction.

Contents

Glossary **1**

List of Abbreviations **2**

List of Figures **4**

List of Tables **6**

1 Introduction **7**
1.1 Motivation 8
1.2 Objective 9
1.2.1 Main Objective 9
1.2.2 Secondary Objectives 9
1.3 Hypothesis 9
1.3.1 Improved depth accuracy with optical principles for lens-based imaging 9
1.3.2 Anisotropic Gaussian optimization 10
1.3.3 To put it simply 10

2 3D Scene Reconstruction: understanding photogrammetry, novel view synthesis and stereoscopy **12**
2.1 The breakthrough in Cotê d'Azur 12
2.2 3DGS-LiDAR, a hardware assisted approach for Gaussian Splatting model improvements 14
2.2.1 How much extra data is actually helpful? 15
2.2.2 Optimizing image overlap 15
2.2.3 Merging Point Clouds 16
2.2.4 Improvements in final output 16
2.3 Revisiting stereoscopy: a multi-lens approach for recreating a 3D scene 18
2.4 COLMAP and OpenCV as relevant tools and libraries 23
2.5 Lens distortions and important photography concepts 24

3 Multi-lens 3DGS: an approach to power 3D scans leveraging wide and ultrawide lenses with data collected from consumer-grade devices **26**
3.1 OpenSplat as the chosen base code 26
3.2 Choosing a proper dataset: *ARKitScenes* 26
3.2.1 Isolating scenes images 27
3.2.2 Ultrawide preprocessing: undistortion and cropping . . . 29
3.2.3 One scene, three scenarios 33
3.3 Generating a sparse point cloud 36
3.3.1 Feature extraction 36
3.3.2 Sequential Feature Matching 36
3.3.3 Reconstruction (Multi-Model) 37

3.4 Batch running OpenSplat with custom metric log 37

4 Methodology **40**
4.1 Perfomance Metrics 40
4.1.1 SSIM 40
4.1.2 MSE and PSNR 41
4.1.3 Time 41

5 Results **42**
5.1 SSIM Analysis 42
5.2 PSNR Analysis 44
5.3 The scores and the constraints 45

6 Conclusion **46**

Annex **52**

List of Figures

Figure 1 Comparison of images captured using telephoto, wide, and ultra-wide lenses. 10

Figure 2 3D point cloud (bottom) generated from images (top). From *3D Digitization using Structure from Motion* (Haro et al., 2012). 12

Figure 3 A conceptual comparison between NeRFs and Gaussian Splatting, originally presented in *A Comprehensive Overview of Gaussian Splatting* (Yurkova, 2024). 14

Figure 4 Representation of a traditional Gaussian Splatting workflow, originally presented in *3D Gaussian Splatting for Real-Time Radiance Field Rendering* (Kerbl et al., 2023). 14

Figure 5 Originally from *LiDAR-3DGS: LiDAR Reinforced 3D Gaussian Splatting for Multimodal Radiance Field Rendering* (Lim et al., 2024). 18

Figure 6 First stereoscopic device, with details of the pictures used for left eye (L) and right eye (R) viewing. Reproduced from the King's College London archive. 19

Figure 7 Example of a Kodak Stereo Camera, a 35mm stereoscopic film camera manufactured during 1954 and 1959. Photography by John Alan Elson, licensed under CC BY-SA 4.0. 20

Figure 8 Passive anaglyph 3D glasses mechanics illustration sourced from *How 3D Glasses Work?* (Brain, 2000). 20

Figure 9 Sourced from *Basic Principles of Stereoscopic 3D* (Reeve and Flock, 2010) 21

Figure 10 Sourced from *What are VR headsets?* (Interaction Design Foundation - IxDF, 2024) 22

Figure 11 Images with filtered depths and normals for crowd-sourced images, after Colmap Desnse Reconstruction. Presented originally in *Pixelwise View Selection for Unstructured Multi-View Stereo* (Schönberger et al., 2016). 23

Figure 12 Image undistortion process using *OpenCV*, presented originally in *OpenCV Camera Calibration* (Bradski, 2000). . . 24

Figure 13 Example of lens distortion originally presented on *Learning OpenCV* (Bradski and Kaehler, 2011) 25

Figure 14 Comparison between wide and ultrawide images, showcasing the difference in *Field of View* between the lenses. . . . 33

Figure 15 *Remap* undistortion process, with deatils for both original and undistorted images. In images c and d, there is change in the lines formed by the blinds in the window, with those bending to a more intensive degree on image c than on image d. 34

Figure 16 Comparison between original and preprocessed ultrawide images 34

Figure 17 Overlaying comparison between final preprocessed ultrawide image and original wide image, with the overlapping *FOV* applied with a *difference* blend mode and the extra *FOV* still present after undistortion and cropping represented normally, as a frame to the wide image. 35
Figure 18 Qualitative comparison of COLMAP reconstructions under different input conditions for Scene 41048085. 39
Figure 19 Side-by-side comparison of OpenSplat reconstructions of scene **41048113** 45
Figure 20 Comparison between preprocessed and raw ultrawide images. 47
Figure 21 Side-by-side comparison of COLMAP reconstructions . . . 48

List of Tables

1 SSIM per Scene and Input Type 43
2 MSE per Scene and Input Type with Best Output 44

1 Introduction

Recent advances in hardware processing power have fueled novel approaches to a somewhat old challenge: to capture and later reproduce a real world scene in three dimensions. Naturally, each approach is tailored to perform best in accordance to its expected outcome (Yurkova, 2024).

Some methods, as photogrammetry (Schönberger and Frahm, 2016) or sensor-based point cloud reconstructions, can excel in scene measurements accuracy, estimating distance between objects and from objects to sensor; with potential applications in architecture or engineering, or even Simultaneous Location And Mapping (SLAM) systems(Baruch et al., 2022), for autonomous driving or military applications.

Other methods focus on a more high-fidelity image output when rendering views from a scene, such as ray-tracing based methods, which have been state-of-the-art for image output quality, yet too resource intensive to work for real-time rendering solutions. (Mildenhall et al., 2020)

A high-fidelity image output with real-time rendering applications solution was presented to the Computer Vision community in 2023 on the original Gaussian Splatting paper (Kerbl et al., 2023). This new approach was successful in rendering novel 1080p views at over 30 FPS, while maintaining consistently high scores in important metrics for image reconstruction. It did so by reinventing the rasterization process to a non mesh-based three dimensional representation. Although traditional mesh representation methods are the current standard for editable 3D scenes and objects, non-mesh representations present potential for when the main focus is a visually accurate, non interactive reconstruction. It could be used, for example, to create background environments for a game scene or to reconstruct real-world touristic attractions in Virtual Reality with detailed accuracy, among other possibilities (Yurkova, 2024).

Current models rely on an RGB (Red, Green, Blue) or RGB-D (with depth information included in additional channel) image sequence input, from which the 3D scene is built upon, through a process that will be explored in-depth in following chapters of this thesis. Popular datasets in the field as the Mip-NeRF 360 (Barron et al., 2022) are comprised of JPG snapshots that focus on a same object/scene section but are taken from a variety of different angles. Different data is extracted from this input, from camera position to depth estimation, through algorithms as Structure from Motion (SfM) (Ullman, 1979) and COLMAP - a digital *SfM* pipeline for camera pose estimation and 3D model generation (Schönberger and Frahm, 2016), and then used in addition to the RGB data to power the reconstruction. This is ideal as it replaces expensive photogrammetry rigs with consumer-grade capture equipment as digital cameras.

Recently and evermore, smartphones manufacturers have been introducing multiple camera lenses on their devices, with different focal lengths for wider or narrower image capture, as the iPhone 16 Pro (Apple Inc., 2025) or the Galaxy S25 Ultra (Samsung Electronics, 2025). As they are attached to the same device with a fixed relative distance to each other as the phone moves throughout the scene, we have consumer-grade equipment ready to capture a scene with more then one type of RGB input per camera position. This lead us to the question: would Gaussian Splatting models be able to leverage these extra inputs for more accurate, faster, or even optimized reconstruction?

1.1 Motivation

Incremental approaches that assist Gaussian Splatting models by providing it with richer data have explored the use of depth sensors as LiDAR (Light Detection and Ranging) during capture (Lim et al., 2024) as a new data source or adopting a radiance field informed Gaussian Splatting optimization (Niemeyer et al., 2024) as a processing step. Results on these tests proved themselves positive, leading to the inference that more information could improve the model's output.

In addition to these contributions on scene reconstruction, advancements in studies and real world applications for 3D scene understanding have slowly incremented consumer-grade everyday devices with different depth sensors, from Microsoft's Kinect depth sensors powering RGB-D Objects dataset for use in object identifying algorithms (Lai et al., 2011) in the early 2010s to high-end sensors on the latest Apple's smartphone models used in capturing the ARKitScenes dataset, used for estimating 3D bounding boxes for objects in a scene and a multitude of scene understanding data to power AR applications(Baruch et al., 2022).

The latter dataset, which will be further explored in future sessions of this text, is comprised of much valuable data for scene reconstruction. Not only it provides depth sensor data and camera pose values, but also providing RGB images and camera intrinsics for both the wide and ultrawide camera lenses on the iPad Pros used to capture the data (Baruch et al., 2022).

Facing the availability of this dataset (currently licensed by Apple, Inc. under a Creative Commons agreement), the rapid growing implementation of the Gaussian Splat method - for general 3D scanning, VFX or VR/AR applications (Yurkova, 2024) - and a will to enhance this generative 3D pipeline so that it is easier to use, faster in training and less resource intensive, this study presents its objectives as follows.

1.2 Objective

In this section, the central aim of this research is outlined. Secondary supporting goals are still present and, together, this objectives will shape the methodology and analysis presented throughout the thesis.

1.2.1 Main Objective

To determine whether additional RGB data from a different focal length lens can improve the output or the performance of a Gaussian Splatting scene reconstruction model.

1.2.2 Secondary Objectives

In addition to the objective presented above and as a natural derived result from pursuing this objective, the study also aims to compare the reconstruction quality of single-lens and multi-lens models using visual and quantitative metrics; analyze computational performance in terms of memory usage, training time and rendering speed; investigate whether known optical principles as lens distortion correction can influence reconstruction quality in a multi-lens setup; identify potential limitations or trade-offs introduced by multi-lens inputs; propose future research directions based on findings including, but not limited to, the creation of new datasets with even more inputs - such as a telephoto lens, available on the Pro line up of Apple's mobile devices, for example.

1.3 Hypothesis

The main hypothesis here is clearly defined in the main objective, so in this specific topic the intent is to present part of the reasoning behind the logic that sparked that objective. Specific implementation steps of our multi-lens data as input for the model should follow on next topics as we dive into the methods and testing reports, yet this is a more generic consideration of why, indeed, would images from different focal length lenses be beneficial for this type of reconstruction.

1.3.1 Improved depth accuracy with optical principles for lens-based imaging

Different focal lengths provide varying depth cues. The different perspectives from a wide (aprox. 26 mm) and ultrawide (aprox. 13mm) can emphasize a parallax effect, which can then be leveraged for improved depth accuracy by fusing the two perspectives with triangulation, as in

$$Z = \frac{fb}{x_L - x_R}, \tag{1}$$

where Z is the depth, f is the focal length, b is the baseline (virtual in this case, derived from lens properties), and x_L, x_R are disparities from the wide and ultrawide lenses (Bradski and Kaehler, 2011).

1.3.2 Anisotropic Gaussian optimization

Ultrawide lenses capture more contextual information of a scene and its spatial relationships. This additional context can be used to guide the optimization of Gaussian anisotropy, by adjusting the covariance matrix Σ to reflect the directional uncertainty in scene geometry:

$$\Sigma = R \begin{bmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_z^2 \end{bmatrix} R^\top, \tag{2}$$

where R is the rotation matrix aligning the Gaussian to the local surface orientation, and $\sigma_x, \sigma_y, \sigma_z$ represent the spread along each axis. This extra information can make it easier for the model to determine which directions need more precision (smaller σ) and which can be looser when adjusting the shape of a gaussian, it also provides more clues about the orientation of surfaces, making R (the rotation matrix) more accurate (Kerbl et al., 2023).

1.3.3 To put it simply

The mathematics presented for those unfamiliar with it might seem overwhelming, but behind it there is quite a comprehensive logic. I invite to follow me on a quick non-measurable experiment. Try to imagine yourself in the room you see in these pictures (see Fig. 1):

(a) Telephoto lens (b) Wide lens (c) Ultra-wide lens

Figure 1: Comparison of images captured using telephoto, wide, and ultra-wide lenses.

To sum it up, if one was asked what is being shown on the television, it could refer to image c for information, since is not available on image b or a.

If one was asked about the pattern of the rug on the floor, it could refer to image a for more detailed information, since the information on images b and c is not as rich for this analysis.

Finally, if one was asked about the tattoo on subject's arm, it would likely refer to image b as information is not available on image a and it is poorly represented on image c.

It is straightforward to point which bits of information from which source could be used by a person to make inferences or predictions. The initial assumption was that if this extra information is useful to humans, there is no reason to believe it would not be useful for Machine Learning. By building two different 3DGS models (one that accounts for an ultra-wide lens in addition to the wide lens and another that does not) and analyzing their respective result metrics, I attempt to check if that assumption is valid.

2 3D Scene Reconstruction: understanding photogrammetry, novel view synthesis and stereoscopy

The state-of-the-art scene reconstruction is the focus of this chapter. Furthermore, the different models will be discussed in detail.

2.1 The breakthrough in Cotê d'Azur

This is a very active research field, and new publications are constantly released. When we talk about Gaussian Splatting, they commonly refer to the original paper (Kerbl et al., 2023) and built upon their creations and findings, hence the reason this is the study I will refer to in the text as the so called "state-of-the-art". This paper was particularly disruptive for introducing a new form of 3D representation other than commonly used points or meshes: 3D Gaussians. Here is how it impacts the scene reconstruction timeline.

Firstly the image sequence is analyzed using SfM(Snavely et al., 2008). From this image feature analysis, we gather information relative to camera position and orientation in a end up with a sparse 3D point cloud (see Fig. 2). All points in the point cloud are then turned into a Gaussian.

Figure 2: 3D point cloud (bottom) generated from images (top). From *3D Digitization using Structure from Motion* (Haro et al., 2012).

Each Gaussian - it can be thought of as a "blob" in our scene - is the 3D equivalent of a normal distribution, but anisotropic (meaning it allows for skewability on all 3 axis) and parametrized by:

- **Mean (μ)**: Interpreted as the location (x, y, z), this determines where the Gaussian is placed within the 3D scene.
- **Covariance (Σ)**: Describes how the Gaussian is shaped and oriented in space, allowing it to be skewed or stretched to align with the scene geometry. (Note: Σ here denotes the covariance matrix, not the summation symbol.)
- **Colour**: Specify what the Gaussian looks like. These can either be:
  - Generic colour values in (R, G, B), or
  - Spherical Harmonics (SH) coefficients, which encode how the Gaussian color appears when looked upon from different directions.
- **Opacity (α, mapped via $\sigma(\alpha)$)**: Determines how visible a Gaussian is when rendering a scene view. The raw opacity α is passed through a sigmoid function that normalizes it to the $[0, 1]$ range.

This parameters allow the placing and orienting of each Gaussian in our scene while also making it visually realistic, by adapting to lighting conditions and new camera perspectives.

A 3D scene is now reconstructed, yet this output needs to be rasterized to an actual image. The proposed rasterization approach, which differs from a common mesh-based render pipeline. Unlike traditional mesh-based rendering, where specifically the geometry of surfaces on a scene is defined with the aid of texture maps for added details; gaussians are "soft" primitves, blending into one another when rendered and creating a smooth representation without explicitly defined surfaces (see Fig. 3). Unlike methods that rely on planar circles with normals, 3D Gaussians model geometry using a covariance matrix Σ, avoiding the need for noisy normal estimation. The transformation of Σ into 2D for image space rendering is performed via the viewing transformation W and its Jacobian J, where $\Sigma' = JW\Sigma W^\top J^\top$. To ensure Σ remains positive semi-definite, it is expressed as $\Sigma = RSS^\top R^\top$, where R is a rotation matrix stored as a quaternion and S is a scaling matrix. This representation allows efficient gradient-based optimization and adapts to scene geometry, resulting in a compact and expressive anisotropic covariance representation suitable for novel view synthesis.

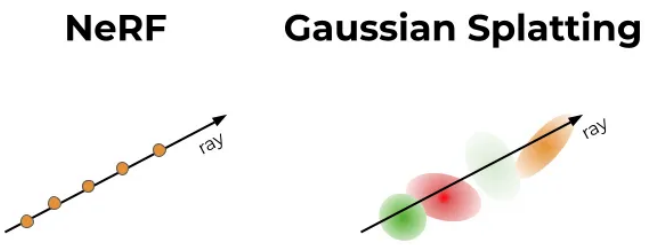


Figure 3: A conceptual comparison between NeRFs and Gaussian Splatting, originally presented in *A Comprehensive Overview of Gaussian Splatting* (Yurkova, 2024).

After rasterizing the scene into pixels, a series of iterations takes place, where the generated image is compared to the ground truth. Through successive iterations, this Gaussians are trained until they have proper values, resulting in an accurate representation of the scene from any given angle (see Fig. 4).

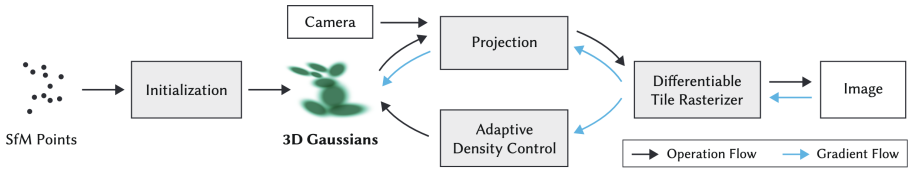


Figure 4: Representation of a traditional Gaussian Splatting workflow, originally presented in *3D Gaussian Splatting for Real-Time Radiance Field Rendering* (Kerbl et al., 2023).

This new method can be considered, depending on one's objective, a significant improvement since the later neural radiance fields state-of-the-art approach for scene reconstruction (Mildenhall et al., 2020), at its time too resource intensive to be able to power real-time rendering.

2.2 3DGS-LiDAR, a hardware assisted approach for Gaussian Splatting model improvements

Not long after the publication of the 3D Gaussian Splatting based Radiance Field Rendering, numerous attempts to make it faster, more accurate, or less resource intensive has numerously sparked across the CV field. Out of those, which consist mostly of mathematical optimization methods for the model, one that stood out in this research was *LiDAR Reinforced 3D Gaussian Splatting for Multimodal Radiance Field Rendering* (Lim et al., 2024).

The novelty here lies in utilizing multimodal inputs to enhance the model's capabilities. By collecting data from a LiDAR researchers had access to actual depth data, rather then relying solely on the estimate from the SfM algorythm applied to the scene images.

The SfM estimation was used but, in addition to it, a sensor-based ground truth depth data was factored in to the equation. After proper formatting, following a special methodolgy designed to properly integrate the new real-world collected data with the camera images for the traditional 3DGS model (Lim et al., 2024).

2.2.1 How much extra data is actually helpful?

Traditional 3DGS methodology would estimate, from a set of images, a sparse point cloud. A real-world LiDAR sensor captures a dense point cloud (a collection of 3D points that represent the geometry of a scene). Therefore, scientists had to come up with a solution to determine what quantity of data is necessary to benefit any model, mitigating overfitting risks and unnecessary resource allocation.

An algorithm was then developed to reduce the density of LiDAR point clouds while preserving critical structural features, it was named *ChromaFilter* (Lim et al., 2024). It exploits colour-based filtering, under the assumption that homogeneous regions (as walls) exhibit less colour variation than regions containing more features. Each LiDAR point is associated with an RGB tuple $\mathbf{P}_{\text{RGB}}(p_i) = [R_i, G_i, B_i]$, which is grouped and sub-sampled based on a maximum number of points per colour value (n):

$$d(p_i) = \frac{1}{k} \sum_{p_j \in N_k(p_i)} |p_i - p_j|, \tag{3}$$

where $N_k(p_i)$ represents the k nearest neighbours of point p_i. A random sampling P_{random} was then applied within each colour group. This process produced a sparse point cloud optimized for merging with the traditional SfM output (Lim et al., 2024; Schönberger and Frahm, 2016).

2.2.2 Optimizing image overlap

To find a meaningful overlap between successive frames used in SfM, the authors (Lim et al., 2024) introduced an overlap-based sampling method. Homography matrices $\mathbf{H}$ between adjacent frames were calculated using *ORB* (Oriented FAST and Rotated BRIEF) features — a fast keypoint detector and binary descriptor (Rublee et al., 2011) — with *RANSAC* (Random Sample Consensus), a robust estimation algorithm used to filter out mismatched feature correspon-

dences (Fischler and Bolles, 1981).

$$\mathbf{H}^* = \arg\min_{\mathbf{H}} \sum_{i=1}^{n} \|\mathbf{p}'_i - \mathbf{H}\mathbf{p}_i\|^2. \tag{4}$$

The overlap percentage was determined as:

$$\text{OL } \% = \frac{\sum_{i,j} I_{\text{warped,adj}}(i,j) \cap I_{\text{adjacent}}(i,j)}{\sum_{i,j} I_{\text{warped}}(i,j)} \times 100. \tag{5}$$

Images were retained if the overlap exceeded 80 %, ensuring effective SfM reconstruction while reducing redundancy.

2.2.3 Merging Point Clouds

The integration of LiDAR and SfM point clouds required alignment into a shared coordinate system. After a coarse manual alignment, the *Iterative Closest Point (ICP) algorithm* (Besl and McKay, 1992) refined the alignment by minimizing the error:

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{n} \|\mathbf{q}_i - (\mathbf{R}\mathbf{p}_i + \mathbf{t})\|^2. \tag{6}$$

The optimization iteratively refined the rotation $\mathbf{R}$ and translation $\mathbf{t}$:

$$(\mathbf{R}^*, \mathbf{t}^*) = \arg\min_{\mathbf{R},\mathbf{t}} E(\mathbf{R}, \mathbf{t}). \tag{7}$$

The alignment process achieved a 99 % correspondence between the two point clouds, providing a unified input for training the 3DGS model.

2.2.4 Improvements in final output

The integration of LiDAR point cloud data into the 3DGS framework significantly improved the overall performance of the model, both in terms of geometric precision and perceptual accuracy. Some key findings from this experiment (Lim et al., 2024) are:

- **PSNR and SSIM Improvements:** These are standard metrics when analyzing 3DGS reconstructed scenes. Incorporating LiDAR data yielded a PSNR increase of 7.064 % at a density of $n = 10$ LiDAR points, with moderate improvements in SSIM (+0.564 %). However, diminishing returns were observed as the LiDAR density increased, suggesting a dataset-specific threshold for maximizing the effectiveness of LiDAR integration.
- **Reduction of Artifacts:** The inclusion of LiDAR data effectively reduced modeling errors, particularly floating errors, which are common artifacts in 3D models generated by image-based methods. This contributed to capturing finer geometric details not accounted for in the original 3DGS model.

- **Improved Textural and Colour Accuracy:** LiDAR data enhanced perceptual aspects, such as texture and colour accuracy, by compensating for intrinsic limitations of image-based feature matching.
- **Impact on Training Time:** While introducing minimal LiDAR data (e.g., $n = 1$) increased training time by only 3.27 %, higher densities (e.g., $n = 20$) resulted in a 19.56 % increase. This highlights a trade-off between performance gains and computational overhead.
- **Optimal SH Levels for PSNR:** The highest PSNR was observed at an spherical harmonics (SH) level of 1500, indicating that the optimal SH parameter for maximizing PSNR can vary based on the dataset. Conversely, SSIM improvements were less dependent on SH level adjustments.

The study demonstrates the benefits of integrating LiDAR data into the 3DGS pipeline. By leveraging even minimal amounts of LiDAR data, the model achieved meaningful enhancements in 3D geometric reconstruction and perceptual quality, making it a practical and scalable approach for applications prioritizing quality without excessive computational demands. By analyzing the table image below, taken from the mentioned LiDAR-3DGS paper (Lim et al., 2024), as a visual summary of this article (see Fig. 5), we can view for ourselves the impact of this approach in the scene reconstruction.

Here we find a visual comparison with details of a picture of the actual real-world room used in training (referred to as Ground Truth), one generated image with same camera position in the scene renderer using a traditional 3DGS ppipeline (Kerbl et al., 2023) (here referred to as 3DGS-vanilla), and one generated image using the sensor-aided 3DGS pipeline used in LiDAR-3DGS (Lim et al., 2024).

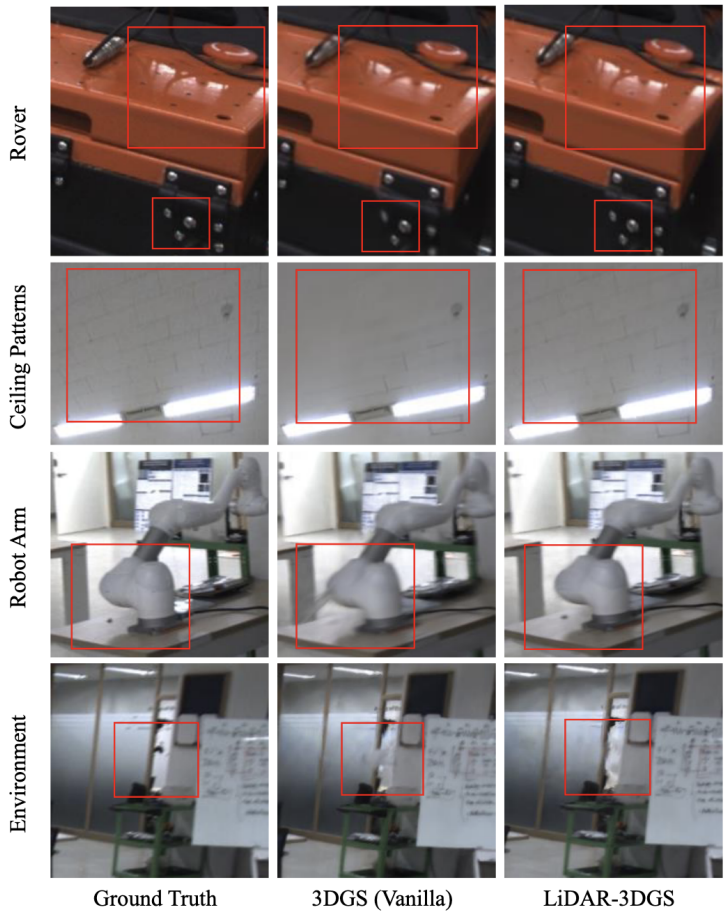
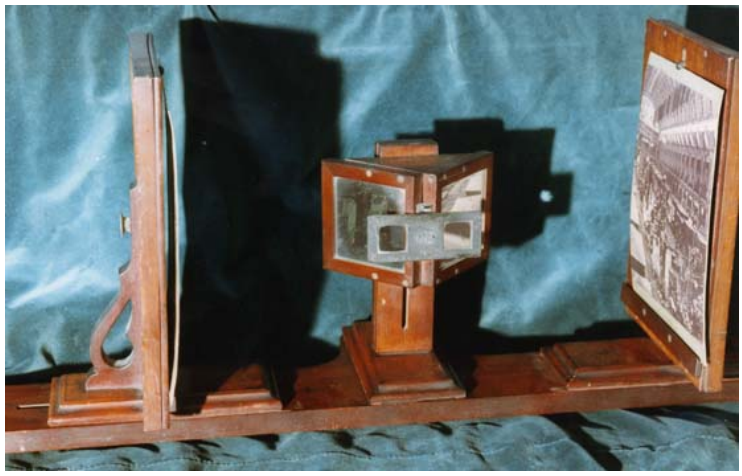


Figure 5: Originally from *LiDAR-3DGS: LiDAR Reinforced 3D Gaussian Splatting for Multimodal Radiance Field Rendering* (Lim et al., 2024).

2.3 Revisiting stereoscopy: a multi-lens approach for recreating a 3D scene

Stereoscopy is a technique for reproducing a 3D scene captured with multiple lenses. It is by no means something new. In 1838, Sir Charles Wheatstone laid out some key foundations for this method by digging into our human binocular vision system (Wheatstone, 1838). By presenting slightly different images to each eye using a mirror-based stereoscope, he simulated the disparity between the viewpoints of the left and right eyes; showing that the human brain combines these images to create the perception of depth (see Fig. 6).



(a) Wheatstone's stereoscope

(b) 1851 Great Exhibition - L

(c) 1851 Great Exhibition - R

Figure 6: First stereoscopic device, with details of the pictures used for left eye (L) and right eye (R) viewing. Reproduced from the King's College London archive.

By the 1950s, McKay (McKay, 1953) already described the spike in adoption of consumer-grade stereo cameras (see Fig. 7), increasing the popularity of the phenomena. It is one of the earliest examples of combining two images from two different lenses to simulate depth. By keeping this position relationship between lenses consistent across snapshots, depth can be simulated also in videos.

Figure 7: Example of a Kodak Stereo Camera, a 35mm stereoscopic film camera manufactured during 1954 and 1959. Photography by John Alan Elson, licensed under CC BY-SA 4.0.

Entering the 2000s, big movies studios are already constantly releasing 3D titles, powered by advancements in anaglyph stereo images (Dubois, 2001). The so called "passive 3D glasses" (typically red and cyan) use colored filters for each eye. On screen, the correspondant left and right images are individually colored then superimposed as a single image (see Fig. 8).

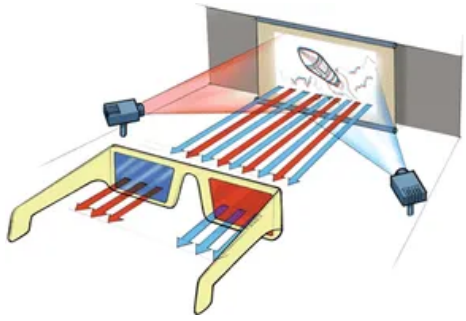


Figure 8: Passive anaglyph 3D glasses mechanics illustration sourced from *How 3D Glasses Work?* (Brain, 2000).

With the arrival of 2010, 3D cinema peaks with James Cameron's *Avatar*. The Hollywood box-office record-breaking title leveraged new, digital, active 3D glasses. This more sophistiacted devices, which required electric power, had the ability to synchronise with the screen. Lenses on each side of the glasses would block out the entire field of view of one eye, when the screen in perfect sync would display the image to be seen by the other unblocked eye. This process repeatedly alternates, so fast it is not perceived by the viewer, which sees only

a depth-enriched image (see Fig. 9).

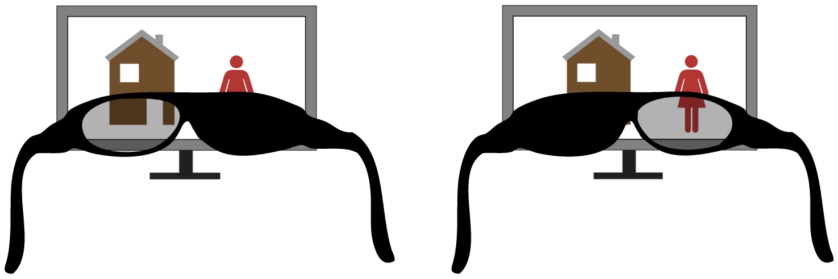


Figure 9: Sourced from *Basic Principles of Stereoscopic 3D* (Reeve and Flock, 2010)

Although useful, important for the field, and simply fun; there is no actual 3D data being generated here from the left and right images. The images are displayed in this particular way, providing one specific field of view for each eye, to create three dimensional visual experiences once our brain combines them to create a sense of depth (McKay, 1953).

Stereoscopic principles remain relevant and used in, for example, in the viewing experience on modern-day VR headsets (see Fig. 10). On this devices, it is no longer needed to filter out colors of a superimposed image or to block out alternate views, we simply have one dedicated display for each eye.

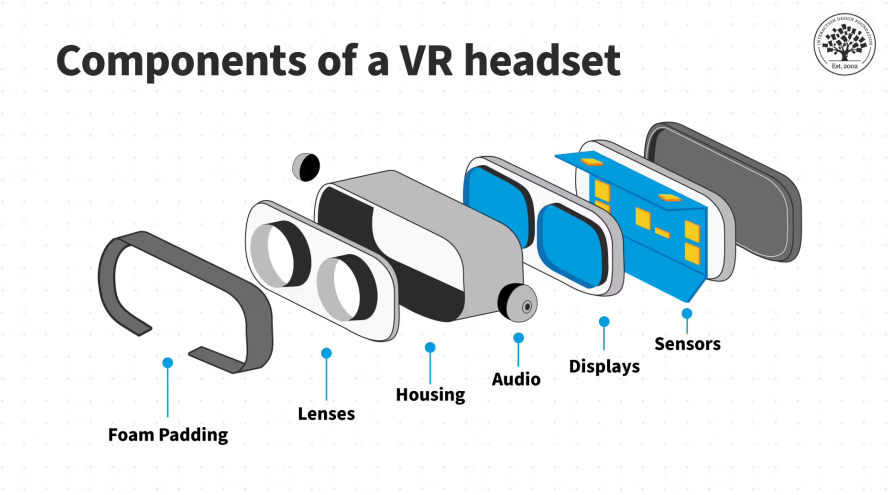


Figure 10: Sourced from *What are VR headsets?* (Interaction Design Foundation - IxDF, 2024)

Even knowing that in stereoscopy there is simply depth simulation and no depth computation, it is important to revisit this technique because, despit its output being more of an optical illusion then actual 3D data, it is clear that it leverages a fixed relationship between the two lenses in its gear to create, simulate or reproduce the sensation of depth; by outputting specific signals our biological binocular system will utilize to infer a perceived depth.

The relationship between a wide and a ultrawide camera in a modern-day smartphone, for example, is much different than the relationship of the two lenses on a stereoscopic capture equipment. However, it should be highlighted that:

There is feature overlap in stereoscopic 3D, yet the minimal consistent differences between the view of each eye provide sufficient data for the human processing of depth. Similarly, when comparing a wide and ultrawide lenses, there should be significant overlap since the field of view of a ultrawide lens encompass in itself the field of view of a wide lens, where the ultrawide provides more context on peripheral areas and the wide lens provides more detailed on the center of its field of view. Both lenses have a fixed relationship to one another across multiple frames, much like in a stereoscopic approach. From there, a question arises and prompts this research that aims to investigate wether the use of two different images (since the lens have different camera intrinsics) from the very same view point (since attached to the same device, the two lenses will be in the same extrinsic context) can help machines perceive depth. Naturally, not in the same way as stereoscopy, as that relies on the human body as a depth

decoder; but in mesaurable 3DGS and point cloud reconstructions.

2.4 COLMAP and OpenCV as relevant tools and libraries

When training 3D Gaussians for a splat, we are aiming to find visual similarity between reconstruction renders and ground truth (Kerbl et al., 2023). This process takes into account an already imported scene geometry. This training is not intended to figure out geometry. It had as input an already computated sparse cloud, and the actual aim is to represent this point cloud using the Gaussians for a lookalike representation during rendering.
It is impossible to input a video or even images into a 3DGS pipeline. The required input for this pipeline is a point cloud and its database images. If one has a video (sequence of images) that should be turned into a fully reconstructed 3DGS scene; this input would need to be turned into a sparse point cloud in order to enter the pipeline. This is done using a Structure-from-Motion (SfM) and/or Multi-View Stereo (MVS) pipeline (see Fig. 11) (Schönberger et al., 2016), good for both incremental and also global reconstruction methods. COLMAP is the standard workflow to achieve this (Schönberger and Frahm, 2016), therefore it was adopted as the study main starting point, taking as input image files and returning ready-to-use in 3DGS pipelines point cloud files. It is known for cross-platform (Windows, Mac, Linux) and wide variety of camera models (inclusing fisheye variations as *Radial Fisheye*, *Simple Radial Fisheye* and *OpenCV fisheye*) (Schönberger and Frahm, 2016).

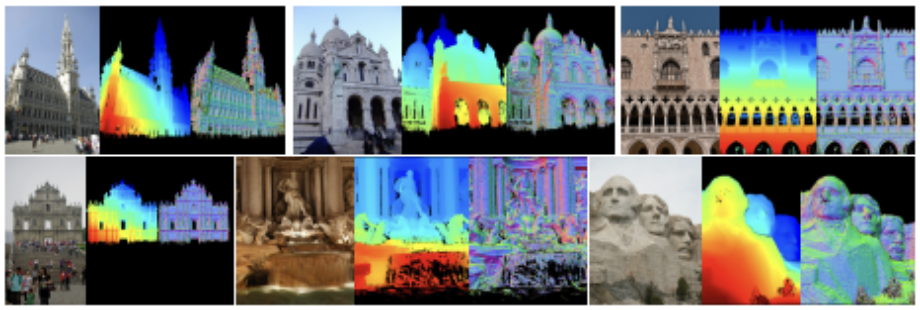
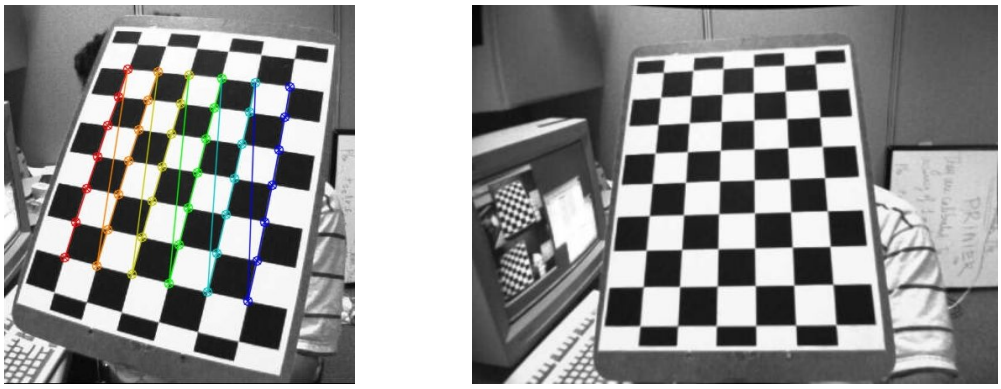


Figure 11: Images with filtered depths and normals for crowd-sourced images, after Colmap Desnse Reconstruction. Presented originally in *Pixelwise View Selection for Unstructured Multi-View Stereo* (Schönberger et al., 2016).

OpenCV (Bradski, 2000), or Open Source Computer Vision, is vast and widely adopted framework for image processing tasks and application. Born in 1999 as a project from then tech giant Intel, it is now maintaned by the active community at *opencv.org*, providing a multitude of capabilities for image manipulation, image analysis and image processing. Its open source and versatile nature, coded mainly in *C++* with bindings for *Python, Java* and other

languages, made this a framework that has currently (and for quite some time) been adopted in both academic research and industy development (Yurkova, 2024).

In the context of photogammetry and volumetric reconstruction, this framework becomes particularly useful due to its camera calibration and image undistortion modules (see Fig. 12). They help achieve greater geometrical accuracy, which is essential in *SfM* pipelines. Also, these very same modules are what enabled the chosen workflow for this study, by allowing a preprocessing method for ultrawide lens images that accounts for distortion effects.



(a) Processing barrel distortion on source image

(b) Final image after calibration

Figure 12: Image undistortion process using *OpenCV*, presented originally in *OpenCV Camera Calibration* (Bradski, 2000).

2.5 Lens distortions and important photography concepts

In the science of computer vision and photogammetry, a proper understanding of camera models and lens distortion is essential for an accurate reconstruction output. While most *SfM* pipelines rely on a traditional *pinhole* camera model, which is the simplest projection of a 3D world in a 2D image, real-life captures often present introduced distortions, from lens manufacturing (see Fig. 13) or usage. Over time, frameworks (COLMAP included) were adapted to deal with different camera models. By correctly calibrating and undistorting images, COLMAP is even able to convert one camera model to the other. Turning a *Radial Fisheye* model into a *Simple Radial* or *pinhole* model, making the data useful for model alignment (Schönberger and Frahm, 2016).

The two main types of distortion, especially in the context of this research, to be understood are **radial distortion** and **tangential distortion**. They are defined (Brown, 1966) as:

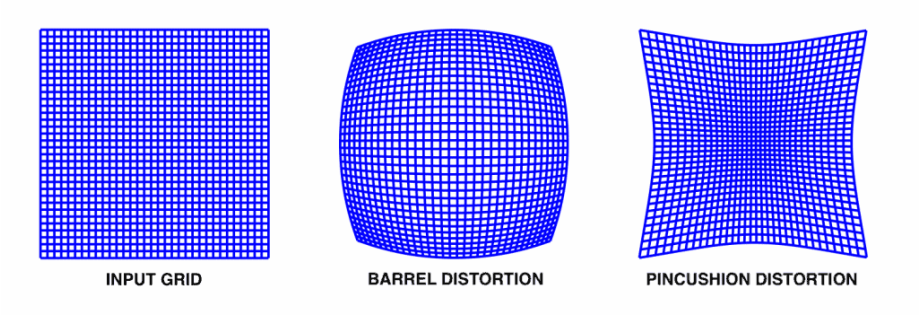


Figure 13: Example of lens distortion originally presented on *Learning OpenCV* (Bradski and Kaehler, 2011)

- **Radial Distortion:** Causes straight lines to appear curved, gets stronger as you grow out of the center. This distortion is modeled by using k_1, k_2 and k_3 as polynomial coefficients. These "wrap" the image as necessary, for perfect alignment with the 2D projection surface.
- **Tangential Distortion:** Introduces small shifts in image coordinates, often caused by lens misalignment.

Ultrawide and fisheye lenses severe distortion (Xu et al., 2024) are precisely what makes these images *Field of View* notably bigger. This provides valuable extra geometrical context when capturing a scene. However, this same big leverage turns into a blocker in the context of the pipeline, once the **Gaussians Splatting models cannot currently account for this type of distortions**. In order to bypass this while keeping model integrity (for proper evaluation), the following proposed solution was adopted.

3 Multi-lens 3DGS: an approach to power 3D scans leveraging wide and ultrawide lenses with data collected from consumer-grade devices

The project now takes on the shape of a 3D reconstruction pipeline that uses as input solely images captured with wide and ultrawide lenses through a mobile device. Using the existing methods and workflows, by making educated choices tailored to the specific object of this study, to find out if images from the same scene using different focal lens with a **fixed extrinsics relationship between lenses** would provide relevant additional context for a 3DGS reconstruction. Choices for the base code, image preprocessing, number of iterations per round. All of these were considered with this specific goal in mind, not to build the ultimate high-res reconstruction but to find out if dual lens use (regardless of frame resolution, image count or other variables) provides measurable improvements for reconstructions or not. To better answer this question isolating other variables was a key factor, allowing the research to narrow down on the actual main hypothesis.

3.1 OpenSplat as the chosen base code

Volumetric reconstruction is still a very compute-expensive task. Despite major advances in optimization, as those demonstrated in literature review; it still requires a very powerful machine or, in case of heavy adaptations for a less powerful machine, copious amount of time. Also, there were specific hardware requirements for reproducing the original 3DGS implementation (Kerbl et al., 2023), as a CUDA-capable NVIDIA graphics card and over 24 GB of RAM to mantain paper valuation quality. These specific resources were not available during the development of this process, so instead we used a different and broader codebase to test our hypothesis, named OpenSplat.

OpenSplat is a C++ implementation of 3DGS (Kerbl et al., 2023) using the Nerfstudio ecosystem (data pipeline, logging, evaluation, etc). This particular implementation was chosen for its adaptability, allowing CUDA alternatives as Apple's Metal GPUs or even the use of CPU at the expense of time. It is also a lightweight, portable and open source set up, so it was adopted as the main method for Gaussian Splatting training. The full OpenSplat code is available on the github link at the annex of this document. This communal project hosted by user *pierotofy* is based on *splatfacto* methods (Tancik et al., 2023).

3.2 Choosing a proper dataset: *ARKitScenes*

To validate the hypothesis, it was imperative to find a dataset containing images from both wide and ultrawide lenses. After extensive research, the one

dataset that stood out with all required capabilities and even some extra flare was Apple's *ARKitScenes* (Baruch et al., 2022). The full dataset is comprised of over 5000 scans of 1,661 unique scenes. All scenes were captured using not only wide and ultrawide lenses but also with LiDAR sensor, 3DoD labels (a benchmark test to show how well a reconstructed model can generate accurate images from slightly different viewpoints, names 3 Degrees of Diference), camera pose tracking files, intrinsics and more; making it the largest RGB-D dataset available (Baruch et al., 2022). Not only that, but scenes were captured using an iPad Pro 11th gen, with technical specs comparable to a wide range of competitors in the market what makes reproducing this pipeline an easy task even for consumer-level hobbyists.

3.2.1 Isolating scenes images

Upon such rich data, downsampling was necessary not only for the number of scans and scenes, as expected for a study with limited resources, but also on the very quantity of raw information present... To assure a fair comparison and isolate as much as possible the variables in the experiment we removed all extra data from the scenes folder that were not the actual images for wide and ultrawide lenses. This includes but is not limited to all camera trajectory files and 3DoD notations. Here is a copy of the official structure as presented in (Baruch et al., 2022) for the scenes scanned in *ARKitScenes* is as following:

- **60 FPS** - includes 3 low resolution assets that synchronize with no guarantee that all assets exist for all timestamps
- **lowres wide** - low resolution RGB images of the wide camera (256x192)
- **lowres depth** - low resolution depth image acquired by AppleDepth Lidar (256x192)
- **confidence** - the confidence of the AppleDepth depth image (256x192)
- **Filtered 10 FPS** - includes 2 high resolution assets that synchronize and filtered by high error between *highres depth* and *lowres depth*.
- **highres depth** - the high resolution ground-truth depth image projected from the mesh generated by FARO's laser scanners (1920x1440)
- **wide** - the RGB images of the wide camera (1920x1440)
- **10FPS** - include 1 VGA resolution asset from ultra wide camera (different timestamp from the assets in section 2)
- **ultrawide** - the RGB images of the ultra wide camera (640x480)
- **30FPS** - include 1 VGA resolution asset from wide camera

- **vga wide** - the RGB images of the wide camera (640x480)
- **Per venue** - the high resolution laser scanner point clouds which are produced by FARO lidar scanners. There are multiple point clouds for each venue, taken from different locations to increase coverage. Each scan includes 2 assets: RGBD point-cloud of the scan and transformation data used for registering multiple point-clouds. Please see the *has laser scanner point clouds* column of the metadata csv to check if those point-clouds are available, per video id (Baruch et al., 2022).

As the main purpose of the study was to test the hypothesis of reconstruction improvements upon addition of complimentary ultrawide RGB data, most of this input was discarted prior to evaluation. For our 10 chosen scenes, only the original folders for *vga wide* and *ultrawide* were kept.

The rational behind this choices was to create 3 reconstructions, using only wide, only ultrawide and both wide and ultrawide lenses. The leverage should come from the relationship of these different *Field of Views* within the same scene.

For this reason, *vga wide* was chosen and not *wide*. The latter had indeed better resolution files, but the only available resolution for ultrawide image was that of 640x480, so we wanted a matching counterpart.

On image count, the wide folder comes from a 30 FPS video scan, whereas ultrawide comes from a 10 FPS one. Image count is an important variable for structure from motion reconstruction (Schönberger and Frahm, 2016) and, in order to keep parity, *the scenario for both lenses was comprised of an equal amount of wide and ultrawide images, image count X. The scenario for wide only lens was comprised of only wide images, but keeping the final image count X. The ultrawide only scenario, however, leveraged all available ultrawide data, resulting in a final image count of X/2.*

The downsampler script used to evenly delete wide images until their count matches ultra wide count is presented below:

```python
import os
import shutil
import numpy as np
from pathlib import Path

# === CONFIG ===
input_dir = Path("/Volumes/ExtremePro/museumRoom/images")  # <- Replace with your full wide image folder
output_dir = Path("/Volumes/ExtremePro/1stRoom/images")  # <- Replace with where you want to copy the 791 images
output_dir.mkdir(parents=True, exist_ok=True)

target_count = 4000 # number of images to select

# === GET AND SORT ALL IMAGE FILENAMES ===
all_images = sorted([f for f in os.listdir(input_dir) if f.lower().endswith(('.png', '.jpg', '.jpeg'))])
total_count = len(all_images)

if target_count > total_count:
    raise ValueError(f"Target count ({target_count}) is greater than total images ({total_count})")

# === SELECT EVENLY SPACED INDICES ===
sample_indices = np.linspace(0, total_count - 1, target_count, dtype=int)
selected_files = [all_images[i] for i in sample_indices]

# === COPY SELECTED FILES TO OUTPUT ===
for filename in selected_files:
    src = input_dir / filename
    dst = output_dir / filename
    shutil.copy(src, dst)
    print(f" Copied: {filename}")

print(f"\n Done! {len(selected_files)} images copied to: {output_dir}")
```

Listing 1: Python script for downsampling the raw wide folder so that it has the same image count as the ultrawide folder

3.2.2 Ultrawide preprocessing: undistortion and cropping

Another variable of important consideration in the *Field of View* (FOV) for each lens. Ultrawide provides, by nature, a larger FOV and there for extra geometrical context for a scene. On the other hand, in order to map this larger space into a 2D image, heavy barrel distortion (see Fig. 13) is applied on all edges of the final image (Xu et al., 2024). Gaussian Splatting pipelines, up

to day this day, do not yet cope with this heavy distortions. When importing reconstructions, fisheye camera models are not accepted; and the pipeline is primarly designed specifically for *pinhole* camera models (Kerbl et al., 2023). Therefore some form of undistortion must be performed on the input images to prep then for a proper 3DGS pipeline. Undistortion methods notably rely on interpolation, and this has a steep effect in gaussian rasterization. A simple undistortion would not be enough if alone, but it still needed to be done, before paired with more methods.

COLMAP itself is currently compliant with fisheye camera models, and this was leverage to estimate camera intrinsics for the device used in the scenes scan. **This information was already present in the original dataset yet since the goal is to base reconstructions solely on RGB data, it was important to estimate it from only these images.** As the same device (an iPad Pro 11th gen) was used for all captures in the dataset (Baruch et al., 2022), this was a process that only needed to be done for one scene. Intrinsics were compared with original intrinsics values of the dataset for extra precaution, and then the estimated values were adopted for the study.

With camera intrinsics at hand, and adopting a fisheye camera model, the *remap* OpenCV function (Bradski and Kaehler, 2011) was used to mitigate barrel distortion in images. This was done via a custom script, demonstrated here:

```
import cv2
import numpy as np
import os
from pathlib import Path

# === Camera Intrinsics and Distortion ===
K = np.array([[251.97491, 0, 320],
              [0, 251.97491, 240],
              [0, 0, 1]])
D = np.array([0.55188, -0.08787, 0, 0])  # fisheye distortion
    coefficients

# === Image dimensions ===
w, h = 640, 480  # make sure this matches your image resolution

# === Paths ===
input_dir = Path("/Volumes/ExtremePro/EvalTests/Scenes
    2/41048129/full/ultrawide")
output_dir = input_dir / "undistorted"
output_dir.mkdir(parents=True, exist_ok=True)

# === New undistorted intrinsics (can tweak balance 0 -- 1 to
    trade FOV vs distortion)
Knew = cv2.fisheye.estimateNewCameraMatrixForUndistortRectify(K
```

```
    , D, (w, h), np.eye(3), balance=1)

print("Knew:")
print(Knew)

# === Prepare undistort maps once
map1, map2 = cv2.fisheye.initUndistortRectifyMap(K, D, np.eye
    (3), Knew, (w, h), cv2.CV_16SC2)

# === Undistort all PNG images ===
for img_path in input_dir.glob("*.png"):
    img = cv2.imread(str(img_path))
    if img is None:
        print(f"Failed to load: {img_path}")
        continue

    undistorted = cv2.remap(img, map1, map2, interpolation=cv2.
        INTER_LINEAR)

    out_path = output_dir / img_path.name
    cv2.imwrite(str(out_path), undistorted)
    print(f"Saved: {out_path}")
```

Listing 2: Python script for fisheye undistortion using OpenCV

After this undistortion step, the interpolation introduced was severe, negatively impacting the reconstruction pipeline. A custom cropping script was implemented to discard 0.2 of each image size. With this cropping, the resolution of ultrawide images was reduced, now at **384x288**. The script used for cropping is available below:

```
import cv2
import os
from pathlib import Path

# === CONFIG ===
input_dir = Path("/Volumes/ExtremePro/EvalTests/Scenes
    2/41048129/full/ultrawide/undistorted") change this to your
    folder
crop_percent = 0.2  # Crop 20% from each side
output_dir = Path("/Volumes/ExtremePro/EvalTests/Scenes
    2/41048129/both/uw")
output_dir.mkdir(parents=True, exist_ok=True)

# === CROPPING FUNCTION ===
def crop_center(img, percent=0.15):
    h, w = img.shape[:2]
    crop_x = int(w * percent)
    crop_y = int(h * percent)
```

```
    return img[crop_y:h - crop_y, crop_x:w - crop_x]

# === PROCESS ALL IMAGES ===
valid_exts = [".png", ".jpg", ".jpeg"]

for img_path in input_dir.iterdir():
    if img_path.suffix.lower() in valid_exts:
        img = cv2.imread(str(img_path))
        if img is None:
            print(f" Failed to load {img_path.name}")
            continue

        cropped = crop_center(img, crop_percent)
        out_path = output_dir / img_path.name
        cv2.imwrite(str(out_path), cropped)
        print(f" Cropped & saved: {out_path.name}")
```

Listing 3: Developed python script used for cropping undistorted ultrawide images

It is important to note that **one of the main bottlenecks of this investigation can be found here**. The lack of a state-of-the-art fisheye camera model for gaussian splatting pipeline forced this research to make a choice on how to ingest ultra wide data into training. Creating a custom *fisheye* camera model for implementation in OpenSplat was explored, but the high level resources, time and knowledge required reached beyond the scope of this project; apart from not providing a direct validation of the hypothesis regarding the fixed relationship between each lenses in capture, since the imporvement could be attributed to the novel camera model itself. Using *remapped* images without any cropping would preserve all of the extra *FOV* in utlrawide lenses, but the introduced interpolation would negatively affect Gaussian Training, as the current 3DGS pipeline does not currently account for such distortions... the chosen middle ground was a combination of *remapping* with OpenCV and the custom cropping demonstrated above, to discard the edges of the undistorted images where the interpolation was more noticeable. The visual comparison presented below (see Fig. 14) aims to illustrate the process of undistorting, the later cropping and the difference in *FOV* from original ultrawide to final ultrawide (undistorted and cropped) and the original wide image. A visual representation of each step is presented below, using the last frame of scene **40958748** as sample.

Upon *remapping* the ultrawide image in an undistortion attempt, there is introduced bilinear interpolation (Bradski and Kaehler, 2011), which can be observed in detail in the images presented (see Fig. 15). This can be easily visualized in areas of greater contrast, as the black line dividing the wall and the window glass, which is close to straight in original ultrawide yet notably not as straight in the undistorted detail. As it is a barrel distortion, it gradually weakens as we look closer to the image center, hence the cropping adopted after

(a) Original wide image

(b) Original ultrawide image

Figure 14: Comparison between wide and ultrawide images, showcasing the difference in *Field of View* between the lenses.

undistortion.

In the end, even after undistortion and cropping (see Fig. 16), the ultrawide images still present a larger *FOV* than their wide counterparts (see Fig. 17). The difference is drastically smaller than the one present between the two original images yet still very noticeable and measurable. This is visually demonstrated in the following images.

3.2.3 One scene, three scenarios

A total of 10 different scenes were selected for this study. Inside the *raw* folder of *ARKitScenes* (Baruch et al., 2022), they are scenes:

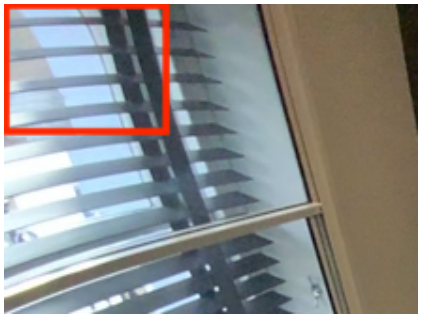
- 40777065
- 40777073
- 40958748
- 40958764
- 41007602
- 41048085
- 41048095
- 41048113
- 40753679
- 40966448
- 41048129

With each of these scenes becoming a main folder, they would contain 4 different subfolders:

- **both:** a folder containing wide and ultrawide images in their respective subfolder.
- **wide:** a folder containing wide images in their respective subfolder.
- **ultrawide:** a folder containing wide images in their respective subfolder.

(a) Original ultrawide image

(b) Undistorted ultrawide image



(c) Detail of original ultrawide image

(d) Detail of undistorted ultrawide

Figure 15: *Remap* undistortion process, with deatils for both original and undistorted images. In images c and d, there is change in the lines formed by the blinds in the window, with those bending to a more intensive degree on image c than on image d.

(a) Original ultrawide image

(b) Preprocessed ultrawide image

Figure 16: Comparison between original and preprocessed ultrawide images

Figure 17: Overlaying comparison between final preprocessed ultrawide image and original wide image, with the overlapping *FOV* applied with a *difference* blend mode and the extra *FOV* still present after undistortion and cropping represented normally, as a frame to the wide image.

A different sparse point cloud was generated for each of this scenarios (both, wide and ultrawide) for each scene, and then exported to the assigned folder.

3.3 Generating a sparse point cloud

The trainning of gaussians (both on OpenSplat and original implementation) is based on a COLMAP(Schönberger and Frahm, 2016) sparse cloud, and the settings and steps used in the point clouds processing step were as follows:

3.3.1 Feature extraction

This is the first step in COLMAP after setting the correct path to your database. Features will be extracted for each images and a few key settings are presented. The chosen model was **Simple Radial** for the low levels of distortion and integration with the OpenSplat pipeline. Some key settings were:

- **Camera model:** SIMPLE RADIAL *(f, cx, cy, k)*
- **Parameters:** Shared per sub-folder and imported from EXIF.
- **Hardware and Computing:** Apple MacBook Pro M3Pro, using *Metal* (GPU use toggled on).

3.3.2 Sequential Feature Matching

Here, features are matched across different images. This is how *SfM* calculates camera poses over time (Ullman, 1979). As the dataset is comprised of a video scan, turned into a sequence of **ordered** images, the chosen matching method was **Sequential Matching**.

- **General:** Using default COLMAP sequential feature matcher settings, except for the following customizations.
- **Hardware and Computing:** Apple MacBook Pro M3Pro, *Metal* (GPU acceleration enabled).
- **Custom Parameters:**
  - **cross_check:** enabled
  - **guided_matching:** enabled
  - **confidence:** increased to 0.99990
- Other parameters: default COLMAP values (e.g., `max_ratio: 0.8`, `max_distance: 0.7`, `max_num_matches: 32768`, `max_error: 4.0`, `min_inlier_ratio: 0.25`, `min_num_inliers: 15`, etc.).

3.3.3 Reconstruction (Multi-Model)

The reconstruction setting were kept as default, once it is already a reliable and recognized method (Schönberger and Frahm, 2016) and as a few settings might change from device to device due to hardware compatibility, the most important fields are listed below.

- **multi_model reconstruction:** Enabled (`multiple_models = True`) to allow for multiple disconnected 3D models.
- **Key Parameters:**
  - `max_num_models`: 50
  - `max_model_overlap`: 20
  - `min_model_size`: 10
  - `min_num_matches`: 15

3.4 Batch running OpenSplat with custom metric log

OpenSplat was implemented almost completely in its default state. Since the ultrawide images were already preprocessed and a **SIMPLE RADIAL** camera model was used during COLMAP, no new camera model was implemented. The only customisation performed in the code was the logging of SSIM (Structural Similarity Index) metric, which was already natively computed for training, but not originally logged on console output for monitoring. A small tweak was perfomed inside the *model.cpp* file to register SSIM values during training, through the small adjustment presented here.

```
torch::Tensor Model::mainLoss(torch::Tensor &rgb, torch::Tensor
    &gt, float ssimWeight){
    torch::Tensor ssimLoss = 1.0f - ssim.eval(rgb, gt);
    torch::Tensor l1Loss = l1(rgb, gt);
    std::cout << "SSIM: " << (1.0f - ssimLoss).item<float>() <<
        std::endl;
    return (1.0f - ssimWeight) * l1Loss + ssimWeight * ssimLoss
        ;
}
```

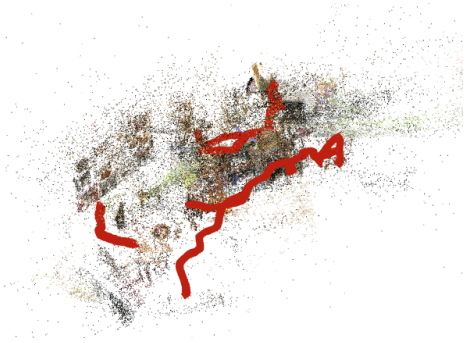
Listing 4: C++ function for logging SSIM-weighted loss for each step on console

From then on, all data was properly selected and preprocessed, minor tweaks in code for metric logging implemented, and the full environment set up complete. The following shell script was used to execute OpenSplat 30 times (for each of the 10 scenes, 3 scenarios) and collect output log for comparison and analysis. The shell script presented below iterated over each scenario (see Fig. 18) and scene and also recorded of the running time in seconds.

```bash
#!/bin/bash

SCENES_DIR="/Volumes/ExtremePro/EvalTests/Scenes 2"
BUILD_DIR="/Volumes/ExtremePro/OpenSplat/build"
OUTPUT_CSV="$SCENES_DIR/results.csv"

ITERS=7000
GAUSSIANS=100000

echo "scene,type,num_gaussians,iters,mse_final,ssim_final,
    time_sec" > "$OUTPUT_CSV"

for scene_path in "$SCENES_DIR"/*; do
  scene=$(basename "$scene_path")

  for type in both justwide justuw; do
    fullpath="$scene_path/$type"

    if [ -d "$fullpath" ]; then
      echo "Running $scene ($type)..."

      log_file=$(mktemp)
      start_time=$(date +%s)

      "$BUILD_DIR/opensplat" "$fullpath" -n "$ITERS" > "
          $log_file" 2>&1
      end_time=$(date +%s)
      elapsed=$((end_time - start_time))

      ssim=$(grep "SSIM:" "$log_file" | tail -n 1 | awk '{print
           $3}')
      mse=$(grep "Step $ITERS:" "$log_file" | tail -1 | awk '{
          prin    t $3}')

      echo "$scene,$type,$GAUSSIANS,$ITERS,$mse,$ssim,$elapsed"
           >> "$OUTPUT_CSV"

      rm "$log_file"
    else
      echo "Skipping missing folder: $fullpath"
    fi
  done
done

echo "Tests complete. Results saved to: $OUTPUT_CSV"
```

Listing 5: Bash script for processing scenes and recording metrics

(a) COLMAP reconstruction using only wide lens images (Scene 41048085)



(b) COLMAP reconstruction using preprocessed ultrawide lens images (Scene 41048085)

(c) COLMAP reconstruction using both wide and preprocessed ultrawide images (Scene 41048085)

Figure 18: Qualitative comparison of COLMAP reconstructions under different input conditions for Scene 41048085.

4 Methodology

The last section dived into the processes and steps to testing our hypothesis. Naturally, this one dives into the chosen metrics for this challenge. While specifics will be brought up along results further along this document, a contextualizing overview is presented in the subsections to prepare the ground for results. It is already established that 30 reconstructions (3 scenarios for 10 scenes) were accomplished, and now the chosen metrics and parameters to evaluate the difference between these reconstructions are presented.

4.1 Perfomance Metrics

The perfomance metrics chosen for this analysis are not strange to the science of generating images through radiance fields. In major papers on 3D Gaussian Splatting (Kerbl et al., 2023), Neural Radiance Fields (Mildenhall et al., 2020) and various improvements upon those two methods (Niemeyer et al., 2024); both **SSIM** and **PSNR** are used in comparison tables. Combining these two metrics for a better assessment is also long recommended in prior literature(Yurkova, 2024).

4.1.1 SSIM

The Structural Similarity Index (SSIM) aims to quantify similarity between images through assessments that accounts for luminance, contrast and structural information. For the purpose of this study, it is adopted to quantify the quality of reconstructions using a model that better correlates with human visual perception, providing a image quality baseline that bases more on the overall context then on a per-pixel based analysis. This SSIM equation for an analysis between two images, x and y is represented as so:

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{8}$$

In this equation,

- μ and σ denote the mean and variance of the image patches
- c_1, c_2 are constants that stabilize the division
- values range from -1 to 1, where 1 indicates perfect structural similarity.

For each scenario of the 10 chosen scenes, SSIM was calculated by comparing the **ground truth images** (in this study, either the original wide lens or the preprocessed ultrawide image, undistorted and cropped) with the rendered image from the last step in training, which was **iteration 7000** in the presented work).

4.1.2 MSE and PSNR

Peak Signal-to-Noise Ratio (**PSNR**) derives from Mean Square Error (**MSE**) to account for reconstruction quality, in decibels (dB).

The formula for MSE is defined as:

$$\text{MSE} = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2 \tag{9}$$

In this expression (Gonzalez and Woods, 2017),

- I is the reconstructed image
- K is the ground truth image
- M and N are the values for image dimensions.

MSE is already widely used in comparisons in image reconstruction or even compression tasks. For model selection and analysis it is already a reliable and standard metric, with no extra benefit perceived in conversion to PSNR decibel values(Huynh-Thu and Ghanbari, 2008). However in image comparison PSNR could be deemed easier to interpret by humans, due to its decibel scale. This conversion is performed according to the following formula.

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{L^2}{\text{MSE}} \right) = 20 \cdot \log_{10} \left(\frac{L}{\sqrt{\text{MSE}}} \right) \tag{10}$$

Here, L is the maximum pixel value (for an 8-bit image, 255).

To honor this legacy and keep parity with recent literature that analysis these types of reconstruction, **PSNR** was adopted as the nominal value that will represent this metric in both tables and text analysis.

4.1.3 Time

Besides standard quality metrics, running time was also monitored during all reconstructions in the study. This not only helps assessing the feasibility of these types of reconstructions on real-world scenarios, but also to compare the impact of having more or less cameras registered in the point cloud database in the Gaussian rasterization and training process. Naturally, the running time reflects the conditions in which the study was conducted, as per the hardware and software specifications already described in section 3.3. While it is appreciated that this conditions might not be reproduce-able by all, it is valuable in this current context as all 30 reconstructions were executed under consistent conditions. The time monitoring process is set on **Listing 5** (Bash script for processing scenes and recording metrics).

5 Results

To understand the impact of using images from both lenses as input data for a Gaussian Splatting pipeline, Structural Similarity Index and Peak Signal-to-Noise Ratio were used. Upon a batch rendering frames of frames from each scene, we are able to compare it to ground truth image, hence the use of image metrics. Even though the output is a 3D model, there is no 3D ground truth data, therefore no traditional metrics for general 3D scenes are presented here.

5.1 SSIM Analysis

The Structural Similarity Index becomes useful in this particular situation to demonstrate, in an approximate way, how humans would visually perceive the reconstruction of the scene. It accounts for image features as contrast and luminance to identify geometrical aspects such as edges or faces and which structures present in the ground truth images were also present in the reconstruction mectrics. When using SSIM metrics, the closer a result is to 1, the closer it is to a perfect match.

In the scenes analyzed, the numeric difference varies between scenes, pointing at no noticeable trend. In scenes such as 073, the reconstruction using only wide lenses scored 0.952. Meanwhile, the reconstruction using both lenses scored 0.953. Rendering, virtually, almost no difference. However, in scene 085 the jump is from 0.80 to 0.94; which is considerably above the noise threshold for a metric like SSIM. Although there are isolated examples within the dataset, when analyzing the sample's *mean* one could argue that using both lenses actually worsen reconstruction.

The performance of reconstructions using solely the ultra wide lenses was incredibly competitive. There is disparity in data between a just wide and just ultra wide reconstruction as the ultra wide extreme cropping of 60 % led to a drastic reduction in resolution, interpolation derived from artificial undistortion and half the number of images used in just wide reconstructions. Therefore, it is surprising to see them tightly close to the "just wide" scene type or even surpassing it.

Table 1: SSIM per Scene and Input Type

Scene	**Both**	**Just Wide**	**Just Ultrawide**	**Best Output**
40777065	0.8378	0.9030	**0.9050**	Just Ultrawide
40777073	0.9471	**0.9520**	0.9497	Just Wide
40958748	**0.9476**	0.9117	0.9455	Both
40958764	0.9147	**0.9526**	0.9383	Just Wide
41007602	**0.9229**	0.9027	0.8797	Both
41048085	**0.9443**	0.8021	0.9126	Both
41048095	0.8587	0.9124	**0.9469**	Just Ultrawide
41048113	0.9057	**0.9584**	0.9553	Just Wide
40753679	0.9460	**0.9496**	0.9482	Just Wide
40966448	0.8941	**0.9504**	0.9105	Just Wide
41048129	0.9137	**0.9543**	0.8617	Just Wide
Mean	0.9092	**0.9295**	0.9230	
Std Dev	0.0368	**0.0365**	0.0257	

5.2 PSNR Analysis

On Peak Signal-to-Noise ratio we measure the per pixel nuances between the two images, holding a strong baseline for comparison between reconstructions. For easier comparison across scenes, the ratio presented in this study reflects PSNR but as mean square error values, since traditional PSNR is measured in decibels.

For general PSNR analysis, the closer a computer value is to zero, the higher the similarity to ground truth. In a good performing scene, score moved from 0.05 to 0.02; whereas in scene 073 we go from 0.03 to 0.02.

Table 2: MSE per Scene and Input Type with Best Output

Scene	**Both**	**Just Wide**	**Just Ultrawide**	**Best Output**
40777065	0.0685	0.0448	**0.0331**	Just Ultrawide
40777073	0.0450	**0.0204**	0.0255	Just Wide
40958748	0.0306	0.0378	**0.0247**	Just Ultrawide
40958764	0.0436	0.0353	**0.0352**	Just Ultrawide
41007602	0.0389	**0.0378**	0.0558	Just Wide
41048085	0.0487	0.0719	**0.0428**	Just Ultrawide
41048095	0.0514	0.0340	**0.0263**	Just Ultrawide
41048113	0.0457	0.0218	**0.0193**	Just Ultrawide
40753679	0.0276	**0.0216**	0.0268	Just Wide
40966448	0.0825	**0.0269**	0.0446	Just Wide
41048129	0.0491	**0.0260**	0.0528	Just Wide
Mean	0.0475	**0.0352**	0.0362	
Std Dev	0.0172	**0.0144**	0.0114	

5.3 The scores and the constraints

While just ultrawide reconstructions use fewer and smaller input images, they achieved SSIM scores within 0.1–0.3 of just wide reconstructions in most scenes, and occasionally outperformed them (e.g., scenes 40777065 and 40958748). This suggests that the increased field of view may provide more informative spatial diversity, helping compensate for lower pixel resolution and image count.

Furthermore, the undistortion and cropping required to adapt ultrawide inputs to the Gaussian Splatting pipeline reduces effective resolution and introduces artifacts, potentially masking the full benefit of the added FOV and introducing a slight 'tunnelingeffect' (see Fig. 19). Yet, performance parity was still observed, pointing to FOV as a promising direction for future research — especially if future methods support native fisheye camera models or improved multi-view calibration.

(a) OpenSplat output for both lenses, showing in detail the tunneling effect resulted from undistortion interpolation

(b) OpenSplat output for just wide lenses, showing in detail straight lines less harmed by undistortion interpolation

Figure 19: Side-by-side comparison of OpenSplat reconstructions of scene **41048113**

6 Conclusion

The work presented in this thesis evaluated the combined use of wide and ultrawide lenses in scene reconstruction, applying the 3DGS methodology. There was not enough evidence, of statistical significance, that would justify a strong statement as saying there is measurable, positive impacts in those types of reconstruction when you combine the two different lenses.

However, a look back into the methodology and approaches chosen, mainly based on current availability for tools and systems, introduces great bottlenecks. The cropping of ultrawide lens images alone discards much of the valuable information in the extra *Field Of View* when compared to wide lens alone. COLMAP's internal undistortion feature could be implemented, but the resulting interpolation of the ultrawide images ends up harming the reconstruction (as it can be observed with a tunneling frame in reconstruction angles as seen on Fig. 21c). In a best case scenario full ultrawide data would be ingested, as we can do on COLMAP, yet current 3DGS (Kerbl et al., 2023) does not provide support for fisheye camera models. Those were the main reasons for choosing the presented methodology.

Even with all constraints imposed on the ultrawide images, they still scored quite competitively amongst other reconstructions. In some specific cases even outperforming wide only constructions, suggesting that a wider geometrical context can be more important than image resolution or even image count. When considered ultrawide only reconstrutions has half the sample size of wide only reconstructions and a much smaller resolution (640×480 for wide and 384×288 for ultrawide) the extra FOV contribution is even more notable.

A comparison between the wide lens and the cropped and undistorted ultrawide was alraedy showcased (see Fig. 17). Now, the comparison is between the cropped and undistorted ultrawide with the full resolution ultrawide image (see Fig. 19).

Here, the preprocessed image (with undistortion and cropping applied) is overlayed on top of the raw ultrawide image in *Difference* blending mode, with opacity at 90 %. The extra frame around the darker square represents all extra data that was discarded during preprocessig (see Fig. 20). Largely to avoid severe undistortion, as the ones we see on the top left corner of the image, where the door frame seems to bend. Inside the darker square, the white lines (mainly present in hard straight lines) are representing the difference after *remapping* for undistortion. It is strongly present in lines that would appear straight in real world scenario.

How 3D to 2D image projection works for fisheye lenses is already known (Xu et al., 2024) and it has been properly implemented in volumetric reconstruction pipelines (Schönberger et al., 2016) and even novel image views (Pittini,

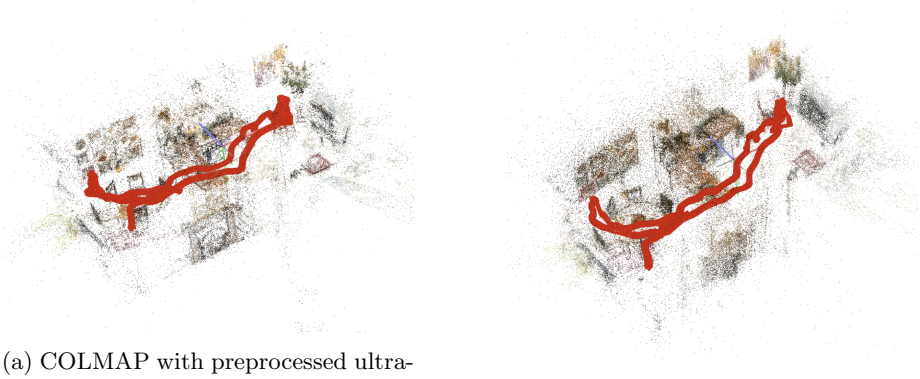
Figure 20: Comparison between preprocessed and raw ultrawide images.

2024). But the methods to deal with fisheye camera models were not yet implemented in the 3DGS pipeline. As mentioned before, Gaussian Splatting provides significant advantages over methods as NeRFs or dense *MVS* COLMAP reconstructions due to its lighter computing, faster training but specially real-time rendering at a frame rate that is acceptable for current standards (with little to no trouble in standard FPS rates as 30 or 60fps and in some implementations (Niemeyer et al., 2024) going ver 900 FPS).

In the following image comparison we can see a COLMAP reconstruction performed for the same scene using images from both lenses using the preprocessed ultrawide in one scenario and the raw ultrawide in another (see Fig. 21).

There is notable difference between the two. But we cannot fully leverage the ultrawide lens power in 3DGS just yet... Even with all undistortion steps this 'tunneling vignette' is still observerved in reconstructions that used both lenses - and it would be fair to attribute this to undistortion introduced interpolation, as this partcular effect is not present in the reconstructions that used solely wide images.

Gaussian Splatting is a very active field in the computer vision community, yet most of the efforts are currently centered in optimizing the timeline. Solutions as pruning unecessary Gaussians and initializing the point cloud with neural radiance fields (Niemeyer et al., 2024) or even tracking the changes of



(a) COLMAP with preprocessed ultrawide

(b) COLMAP with raw ultrawide

Figure 21: Side-by-side comparison of COLMAP reconstructions

Gaussians in a scene over time (Wu et al., 2023) for dynamic render or making the setup super fast and lightweight for easier implementation (Hanson et al., 2024) are important advancements that make the current pipeline more democratic as they remove hefty hardware requirements and more usable as they become lighter, more efficient and even dynamic. Yet, these advancements are on top of traditional reconstruction, in most cases using a **pinhole** camera model or something similar (Kerbl et al., 2023).

The experiment conducted in this research clearly demonstrates the potential of developing different camera models (specially *fisheye* variants) implementations in this pipeline. Projection logic is known and these camera models are more and more accessible each day. Once restricted to speficic devices as **GoPro** sport cameras or specialized DSLR lenses, today ultrawide lenses (or 0.5x lenses) are widely available across a range of different brands that produce mobile phones. While currently unable to confirm wether combining these two types of lenses results in a particular benefit, what becomes clear is that a larger *field of view* and, therefore, larger geometrical context affects greatly reconstruction quality, in some cases being more important than resolution or count.

The need for *fisheye* camera model is clear and reasoned during this text. In **nerfStudio** community, a single comment addresses fisheye models implementation in the *Frequently Asked Questions* sections, stating that **the implementation of other camera models is of interest, but not currently planned**. By highlighting the need for fisheye support through practical experimentation, this study contributes valuable groundwork that could help shape future developments in Gaussian rasterization frameworks.

References

Apple Inc. (2025). iPhone 16 Pro - Technical Specifications. Retrieved June 18, 2025, from https://www.apple.com/pt/iphone-16-pro/specs/.

Barron, J. T., Mildenhall, B., Verbin, D., Srinivasan, P. P., and Hedman, P. (2022). Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. arXiv:2111.12077.

Baruch, G., Chen, Z., Dehghan, A., Dimry, T., Feigin, Y., Fu, P., Gebauer, T., Joffe, B., Kurz, D., Schwartz, A., and Shulman, E. (2022). ARKitScenes: A Diverse Real-World Dataset For 3D Indoor Scene Understanding Using Mobile RGB-D Data. arXiv:2111.08897.

Besl, P. and McKay, N. D. (1992). A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Bradski, G. R. and Kaehler, A. (2011). *Learning OpenCV: computer vision with the OpenCV library*. Software that sees. O'Reilly, Beijing, 1st edition.

Brain, M. (2000). How 3-D Glasses Work.

Brown, D. (1966). Decentering distortion of lenses. *Photogrammetric Engineering*, 32:444–462.

Dubois, E. (2001). A projection method to generate anaglyph stereo images. Technical Report TR-EEE-01-04, University of Ottawa, School of Information Technology and Engineering.

Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395.

Gonzalez, R. C. and Woods, R. E. (2017). *Digital image processing*. Pearson, New York, fourth edition, global edition edition.

Hanson, A., Tu, A., Lin, G., Singla, V., Zwicker, M., and Goldstein, T. (2024). Speedy-Splat: Fast 3D Gaussian Splatting with Sparse Pixels and Sparse Primitives. arXiv:2412.00578 [cs].

Haro, J. d., Romo, C., and Torres, J. C. (2012). 3D Digitization using Structure from Motion.

Huynh-Thu, Q. and Ghanbari, M. (2008). Scope of validity of PSNR in image/video quality assessment. *Electronics Letters*, 44(13):800–801. Publisher: The Institution of Engineering and Technology.

Interaction Design Foundation - IxDF (2024). What are VR Headsets?

Kerbl, B., Kopanas, G., Leimkühler, T., and Drettakis, G. (2023). 3D Gaussian Splatting for Real-Time Radiance Field Rendering. arXiv:2308.04079.

Lai, K., Bo, L., Ren, X., and Fox, D. (2011). Sparse distance learning for object recognition combining RGB and depth information. In *2011 IEEE International Conference on Robotics and Automation*, pages 4007–4013, Shanghai, China. IEEE.

Lim, H., Chang, H., Choi, J. B., and Yeum, C. M. (2024). LiDAR-3DGS: LiDAR Reinforced 3D Gaussian Splatting for Multimodal Radiance Field Rendering. arXiv:2409.16296.

McKay, H. C. (1953). *Three-Dimensional Photography - Principles of Stereoscopy*. American Photography, Book Department, 2nd edition.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. (2020). NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. arXiv:2003.08934.

Niemeyer, M., Manhardt, F., Rakotosaona, M.-J., Oechsle, M., Duckworth, D., Gosula, R., Tateno, K., Bates, J., Kaeser, D., and Tombari, F. (2024). RadSplat: Radiance Field-Informed Gaussian Splatting for Robust Real-Time Rendering with 900+ FPS. arXiv:2403.13806.

Pittini, E. (2024). *Multi-view 3d reconstruction using NeRF-based approaches*. Tesi di laurea, International Conference on Virtual and Augmented Reality in Education (VARE).

Reeve, S. and Flock, J. (2010). Basic Principles of Stereoscopic 3D.

Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, pages 2564–2571. ISSN: 2380-7504.

Samsung Electronics (2025). Galaxy S25 Ultra - Specifications.

Schönberger, J. L. and Frahm, J.-M. (2016). Structure-from-Motion Revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4104–4113, Las Vegas, NV, USA. IEEE.

Schönberger, J. L., Zheng, E., Frahm, J.-M., and Pollefeys, M. (2016). Pixelwise View Selection for Unstructured Multi-View Stereo. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision – ECCV 2016*, volume 9907 of *Lecture Notes in Computer Science*, pages 501–518, Cham. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

Snavely, N., Seitz, S. M., and Szeliski, R. (2008). Modeling the World from Internet Photo Collections. *International Journal of Computer Vision*, 80(2):189–210.

Tancik, M., Weber, E., Ng, E., Li, R., Yi, B., Kerr, J., Wang, T., Kristoffersen, A., Austin, J., Salahi, K., Ahuja, A., McAllister, D., and Kanazawa, A. (2023). Nerfstudio: A Modular Framework for Neural Radiance Field Development. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings*, pages 1–12. arXiv:2302.04264 [cs].

Ullman, S. (1979). The interpretation of structure from motion. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 203:405 – 426.

Wheatstone, C. (1838). XVIII. Contributions to the physiology of vision. —Part the first. On some remarkable, and hitherto unobserved, phenomena of binocular vision. *Philosophical Transactions of the Royal Society of London*, 128:371–394. Publisher: Royal Society.

Wu, G., Yi, T., Fang, J., Xie, L., Zhang, X., Wei, W., Liu, W., Tian, Q., and Wang, X. (2023). 4D Gaussian Splatting for Real-Time Dynamic Scene Rendering. arXiv:2310.08528 version: 2.

Xu, J., Han, D.-W., Li, K., Li, J.-J., and Ma, Z.-Y. (2024). A Comprehensive Overview of Fish-Eye Camera Distortion Correction Methods. arXiv:2401.00442 [cs].

Yurkova, K. (2024). A Comprehensive Overview of Gaussian Splatting (available at https://towardsdatascience.com/a-comprehensive-overview-of-gaussian-splatting-e7d570081362 last access 18/06/2025).

Annex

Apart from the custom scripts present on the body of the text, the full OpenSplat code is available at https://github.com/pierotofy/OpenSplat. The *model.cpp* file in the repository needed a slight tweak in order to log the specified metrics. No other significant change was performed in any other repository file. The full *model.cpp* code contents as used in this study's evaluation rounds is available below.

```
#include <filesystem>
#include "model.hpp"
#include "constants.hpp"
#include "tile_bounds.hpp"
#include "project_gaussians.hpp"
#include "rasterize_gaussians.hpp"
#include "tensor_math.hpp"
#include "gsplat.hpp"
#include "utils.hpp"

#ifdef USE_HIP
#include <c10/hip/HIPCachingAllocator.h>
#elif defined(USE_CUDA)
#include <c10/cuda/CUDACachingAllocator.h>
#endif

namespace fs = std::filesystem;

torch::Tensor randomQuatTensor(long long n){
    torch::Tensor u = torch::rand(n);
    torch::Tensor v = torch::rand(n);
    torch::Tensor w = torch::rand(n);
    return torch::stack({
        torch::sqrt(1 - u) * torch::sin(2 * PI * v),
        torch::sqrt(1 - u) * torch::cos(2 * PI * v),
        torch::sqrt(u) * torch::sin(2 * PI * w),
        torch::sqrt(u) * torch::cos(2 * PI * w)
    }, -1);
}

torch::Tensor projectionMatrix(float zNear, float zFar, float
    fovX, float fovY, const torch::Device &device){
    // OpenGL perspective projection matrix
    float t = zNear * std::tan(0.5f * fovY);
    float b = -t;
    float r = zNear * std::tan(0.5f * fovX);
    float l = -r;
    return torch::tensor({
        {2.0f * zNear / (r - l), 0.0f, (r + l) / (r - l), 0.0f
            },
        {0.0f, 2 * zNear / (t - b), (t + b) / (t - b), 0.0f},
```

```
        {0.0f, 0.0f, (zFar + zNear) / (zFar - zNear), -1.0f *
            zFar * zNear / (zFar - zNear)},
        {0.0f, 0.0f, 1.0f, 0.0f}
    }, device);
}

torch::Tensor psnr(const torch::Tensor& rendered, const torch::
    Tensor& gt){
    torch::Tensor mse = (rendered - gt).pow(2).mean();
    return (10.f * torch::log10(1.0 / mse));
}

torch::Tensor l1(const torch::Tensor& rendered, const torch::
    Tensor& gt){
    return torch::abs(gt - rendered).mean();
}

void Model::setupOptimizers(){
    releaseOptimizers();

    meansOpt = new torch::optim::Adam({means}, torch::optim::
        AdamOptions(0.00016));
    scalesOpt = new torch::optim::Adam({scales}, torch::optim::
        AdamOptions(0.005));
    quatsOpt = new torch::optim::Adam({quats}, torch::optim::
        AdamOptions(0.001));
    featuresDcOpt = new torch::optim::Adam({featuresDc}, torch
        ::optim::AdamOptions(0.0025));
    featuresRestOpt = new torch::optim::Adam({featuresRest},
        torch::optim::AdamOptions(0.000125));
    opacitiesOpt = new torch::optim::Adam({opacities}, torch::
        optim::AdamOptions(0.05));

    meansOptScheduler = new OptimScheduler(meansOpt, 0.0000016f
        , maxSteps);
}

void Model::releaseOptimizers(){
    RELEASE_SAFELY(meansOpt);
    RELEASE_SAFELY(scalesOpt);
    RELEASE_SAFELY(quatsOpt);
    RELEASE_SAFELY(featuresDcOpt);
    RELEASE_SAFELY(featuresRestOpt);
    RELEASE_SAFELY(opacitiesOpt);

    RELEASE_SAFELY(meansOptScheduler);
}


torch::Tensor Model::forward(Camera& cam, int step){
```

```
80
81      const float scaleFactor = getDownscaleFactor(step);
82      const float fx = cam.fx / scaleFactor;
83      const float fy = cam.fy / scaleFactor;
84      const float cx = cam.cx / scaleFactor;
85      const float cy = cam.cy / scaleFactor;
86      const int height = static_cast<int>(static_cast<float>(cam.
            height) / scaleFactor);
87      const int width = static_cast<int>(static_cast<float>(cam.
            width) / scaleFactor);
88
89      torch::Tensor R = cam.camToWorld.index({Slice(None, 3),
            Slice(None, 3)});
90      torch::Tensor T = cam.camToWorld.index({Slice(None, 3),
            Slice(3,4)});
91
92      // Flip the z and y axes to align with gsplat conventions
93      R = torch::matmul(R, torch::diag(torch::tensor({1.0f, -1.0f
            , -1.0f}, R.device())));
94
95      // worldToCam
96      torch::Tensor Rinv = R.transpose(0, 1);
97      torch::Tensor Tinv = torch::matmul(-Rinv, T);
98
99      lastHeight = height;
100     lastWidth = width;
101
102     torch::Tensor viewMat = torch::eye(4, device);
103     viewMat.index_put_({Slice(None, 3), Slice(None, 3)}, Rinv);
104     viewMat.index_put_({Slice(None, 3), Slice(3, 4)}, Tinv);
105
106     float fovX = 2.0f * std::atan(width / (2.0f * fx));
107     float fovY = 2.0f * std::atan(height / (2.0f * fy));
108
109     torch::Tensor projMat = projectionMatrix(0.001f, 1000.0f,
            fovX, fovY, device);
110     torch::Tensor colors =  torch::cat({featuresDc.index({Slice
            (), None, Slice()}), featuresRest}, 1);
111
112     torch::Tensor conics;
113     torch::Tensor depths; // GPU-only
114     torch::Tensor numTilesHit; // GPU-only
115     torch::Tensor cov2d; // CPU-only
116     torch::Tensor camDepths; // CPU-only
117     torch::Tensor rgb;
118
119     if (device == torch::kCPU){
120         auto p = ProjectGaussiansCPU::apply(means,
121                                 torch::exp(scales),
122                                 1,
```

```
                                quats / quats.norm(2, {-1}, true),
                                viewMat,
                                torch::matmul(projMat, viewMat),
                                fx,
                                fy,
                                cx,
                                cy,
                                height,
                                width);
        xys = p[0];
        radii = p[1];
        conics = p[2];
        cov2d = p[3];
        camDepths = p[4];
    }else{
        #if defined(USE_HIP) || defined(USE_CUDA) || defined(USE_MPS)

        TileBounds tileBounds = std::make_tuple((width + BLOCK_X - 1) / BLOCK_X,
                    (height + BLOCK_Y - 1) / BLOCK_Y,
                    1);
        auto p = ProjectGaussians::apply(means,
                    torch::exp(scales),
                    1,
                    quats / quats.norm(2, {-1}, true),
                    viewMat,
                    torch::matmul(projMat, viewMat),
                    fx,
                    fy,
                    cx,
                    cy,
                    height,
                    width,
                    tileBounds);

        xys = p[0];
        depths = p[1];
        radii = p[2];
        conics = p[3];
        numTilesHit = p[4];
        #else
            throw std::runtime_error("GPU support not built, use --cpu");
        #endif
    }

    xys.retain_grad();
```

```
168
169     if (radii.sum().item<float>() == 0.0f)
170         return backgroundColor.repeat({height, width, 1});
171
172     torch::Tensor viewDirs = means.detach() - T.transpose(0, 1)
            .to(device);
173     viewDirs = viewDirs / viewDirs.norm(2, {-1}, true);
174     int degreesToUse = (std::min<int>)(step / shDegreeInterval,
             shDegree);
175     torch::Tensor rgbs;
176
177     if (device == torch::kCPU){
178         rgbs = SphericalHarmonicsCPU::apply(degreesToUse,
                viewDirs, colors);
179     }else{
180         #if defined(USE_HIP) || defined(USE_CUDA) || defined(
                USE_MPS)
181         rgbs = SphericalHarmonics::apply(degreesToUse, viewDirs
                , colors);
182         #endif
183     }
184
185     rgbs = torch::clamp_min(rgbs + 0.5f, 0.0f);
186
187     if (device == torch::kCPU){
188         rgb = RasterizeGaussiansCPU::apply(
189                 xys,
190                 radii,
191                 conics,
192                 rgbs,
193                 torch::sigmoid(opacities),
194                 cov2d,
195                 camDepths,
196                 height,
197                 width,
198                 backgroundColor);
199     }else{
200         #if defined(USE_HIP) || defined(USE_CUDA) || defined(
                USE_MPS)
201         rgb = RasterizeGaussians::apply(
202                 xys,
203                 depths,
204                 radii,
205                 conics,
206                 numTilesHit,
207                 rgbs,
208                 torch::sigmoid(opacities),
209                 height,
210                 width,
211                 backgroundColor);
```

```
212         #endif
213     }
214 
215     rgb = torch::clamp_max(rgb, 1.0f);
216 
217     return rgb;
218 }
219 
220 void Model::optimizersZeroGrad(){
221   meansOpt->zero_grad();
222   scalesOpt->zero_grad();
223   quatsOpt->zero_grad();
224   featuresDcOpt->zero_grad();
225   featuresRestOpt->zero_grad();
226   opacitiesOpt->zero_grad();
227 }
228 
229 void Model::optimizersStep(){
230   meansOpt->step();
231   scalesOpt->step();
232   quatsOpt->step();
233   featuresDcOpt->step();
234   featuresRestOpt->step();
235   opacitiesOpt->step();
236 }
237 
238 void Model::schedulersStep(int step){
239   meansOptScheduler->step(step);
240 }
241 
242 int Model::getDownscaleFactor(int step){
243     return std::pow(2, (std::max<int>)(numDownscales - step /
        resolutionSchedule, 0));
244 }
245 
246 void Model::addToOptimizer(torch::optim::Adam *optimizer, const
     torch::Tensor &newParam, const torch::Tensor &idcs, int
    nSamples){
247     torch::Tensor param = optimizer->param_groups()[0].params()
        [0];
248 #if TORCH_VERSION_MAJOR == 2 && TORCH_VERSION_MINOR > 1
249     auto pId = param.unsafeGetTensorImpl();
250 #else
251     auto pId = c10::guts::to_string(param.unsafeGetTensorImpl()
        );
252 #endif
253     auto paramState = std::make_unique<torch::optim::
        AdamParamState>(static_cast<torch::optim::AdamParamState
        &>(*optimizer->state()[pId]));
254 
```

```
255     std::vector<int64_t> repeats;
256     repeats.push_back(nSamples);
257     for (long int i = 0; i < paramState->exp_avg().dim() - 1; i
            ++){
258         repeats.push_back(1);
259     }
260 
261     paramState->exp_avg(torch::cat({
262         paramState->exp_avg(),
263         torch::zeros_like(paramState->exp_avg().index({idcs.
                squeeze()}))).repeat(repeats)
264     }, 0));
265 
266     paramState->exp_avg_sq(torch::cat({
267         paramState->exp_avg_sq(),
268         torch::zeros_like(paramState->exp_avg_sq().index({idcs.
                squeeze()}))).repeat(repeats)
269     }, 0));
270 
271     optimizer->state().erase(pId);
272 
273 #if TORCH_VERSION_MAJOR == 2 && TORCH_VERSION_MINOR > 1
274     auto newPId = newParam.unsafeGetTensorImpl();
275 #else
276     auto newPId = c10::guts::to_string(newParam.
            unsafeGetTensorImpl());
277 #endif
278     optimizer->state()[newPId] = std::move(paramState);
279     optimizer->param_groups()[0].params()[0] = newParam;
280 }
281 
282 void Model::removeFromOptimizer(torch::optim::Adam *optimizer,
        const torch::Tensor &newParam, const torch::Tensor &
        deletedMask){
283     torch::Tensor param = optimizer->param_groups()[0].params()
            [0];
284 #if TORCH_VERSION_MAJOR == 2 && TORCH_VERSION_MINOR > 1
285     auto pId = param.unsafeGetTensorImpl();
286 #else
287     auto pId = c10::guts::to_string(param.unsafeGetTensorImpl()
            );
288 #endif
289     auto paramState = std::make_unique<torch::optim::
            AdamParamState>(static_cast<torch::optim::AdamParamState
            &>(*optimizer->state()[pId]));
290 
291     paramState->exp_avg(paramState->exp_avg().index({~
            deletedMask}));
292     paramState->exp_avg_sq(paramState->exp_avg_sq().index({~
            deletedMask}));
```

```
293
294     optimizer->state().erase(pId);
295 #if TORCH_VERSION_MAJOR == 2 && TORCH_VERSION_MINOR > 1
296     auto newPId = newParam.unsafeGetTensorImpl();
297 #else
298     auto newPId = c10::guts::to_string(newParam.
            unsafeGetTensorImpl());
299 #endif
300     optimizer->param_groups()[0].params()[0] = newParam;
301     optimizer->state()[newPId] = std::move(paramState);
302 }
303
304 void Model::afterTrain(int step){
305     torch::NoGradGuard noGrad;
306
307     // When radii.sum() == 0
308     if (!xys.grad().defined()) return;
309
310     if (step < stopSplitAt){
311         torch::Tensor visibleMask = (radii > 0).flatten();
312
313         torch::Tensor grads = torch::linalg_vector_norm(xys.
                grad().detach(), 2, { -1 }, false, torch::kFloat32);
314         if (!xysGradNorm.numel()){
315             xysGradNorm = grads;
316             visCounts = torch::ones_like(xysGradNorm);
317         }else{
318             visCounts.index_put_({visibleMask}, visCounts.index
                    ({visibleMask}) + 1);
319             xysGradNorm.index_put_({visibleMask}, grads.index({
                    visibleMask}) + xysGradNorm.index({visibleMask})
                    );
320         }
321
322         if (!max2DSize.numel()){
323             max2DSize = torch::zeros_like(radii, torch::
                    kFloat32);
324         }
325
326         torch::Tensor newRadii = radii.detach().index({
                visibleMask});
327         max2DSize.index_put_({visibleMask}, torch::maximum(
328                 max2DSize.index({visibleMask}), newRadii /
                        static_cast<float>( (std::max)(lastHeight,
                        lastWidth) )
329             ));
330     }
331
332     if (step % refineEvery == 0 && step > warmupLength){
333         int resetInterval = resetAlphaEvery * refineEvery;
```

```
334         bool doDensification = step < stopSplitAt && step %
                resetInterval > numCameras + refineEvery;
335         torch::Tensor splitsMask;
336         const float cullAlphaThresh = 0.1f;
337
338         if (doDensification){
339             int numPointsBefore = means.size(0);
340             torch::Tensor avgGradNorm = (xysGradNorm /
                    visCounts) * 0.5f * static_cast<float>( (std::
                    max)(lastWidth, lastHeight) );
341             torch::Tensor highGrads = (avgGradNorm >
                    densifyGradThresh).squeeze();
342
343             // Split gaussians that are too large
344             torch::Tensor splits = (std::get<0>(scales.exp().
                    max(-1)) > densifySizeThresh).squeeze();
345             if (step < stopScreenSizeAt){
346                 splits |= (max2DSize > splitScreenSize).squeeze
                        ();
347             }
348
349             splits &= highGrads;
350             const int nSplitSamples = 2;
351             int nSplits = splits.sum().item<int>();
352
353             torch::Tensor centeredSamples = torch::randn({
                    nSplitSamples * nSplits, 3}, device);  // Nx3 of
                     axis-aligned scales
354             torch::Tensor scaledSamples = torch::exp(scales.
                    index({splits}).repeat({nSplitSamples, 1})) *
                    centeredSamples;
355             torch::Tensor qs = quats.index({splits}) / torch::
                    linalg_vector_norm(quats.index({splits}), 2, {
                    -1 }, true, torch::kFloat32);
356             torch::Tensor rots = quatToRotMat(qs.repeat({
                    nSplitSamples, 1}));
357             torch::Tensor rotatedSamples = torch::bmm(rots,
                    scaledSamples.index({"...", None})).squeeze();
358             torch::Tensor splitMeans = rotatedSamples + means.
                    index({splits}).repeat({nSplitSamples, 1});
359
360             torch::Tensor splitFeaturesDc = featuresDc.index({
                    splits}).repeat({nSplitSamples, 1});
361             torch::Tensor splitFeaturesRest = featuresRest.
                    index({splits}).repeat({nSplitSamples, 1, 1});
362
363             torch::Tensor splitOpacities = opacities.index({
                    splits}).repeat({nSplitSamples, 1});
364
365             const float sizeFac = 1.6f;
```

```
366             torch::Tensor splitScales = torch::log(torch::exp(
                    scales.index({splits})) / sizeFac).repeat({
                    nSplitSamples, 1});
367             scales.index({splits}) = torch::log(torch::exp(
                    scales.index({splits})) / sizeFac);
368             torch::Tensor splitQuats = quats.index({splits}).
                    repeat({nSplitSamples, 1});
369
370             // Duplicate gaussians that are too small
371             torch::Tensor dups = (std::get<0>(scales.exp().max
                    (-1)) <= densifySizeThresh).squeeze();
372             dups &= highGrads;
373             torch::Tensor dupMeans = means.index({dups});
374             torch::Tensor dupFeaturesDc = featuresDc.index({
                    dups});
375             torch::Tensor dupFeaturesRest = featuresRest.index
                    ({dups});
376             torch::Tensor dupOpacities = opacities.index({dups
                    });
377             torch::Tensor dupScales = scales.index({dups});
378             torch::Tensor dupQuats = quats.index({dups});
379
380             means = torch::cat({means.detach(), splitMeans,
                    dupMeans}, 0).requires_grad_();
381             featuresDc = torch::cat({featuresDc.detach(),
                    splitFeaturesDc, dupFeaturesDc}, 0).
                    requires_grad_();
382             featuresRest = torch::cat({featuresRest.detach(),
                    splitFeaturesRest, dupFeaturesRest}, 0).
                    requires_grad_();
383             opacities = torch::cat({opacities.detach(),
                    splitOpacities, dupOpacities}, 0).requires_grad_
                    ();
384             scales = torch::cat({scales.detach(), splitScales,
                    dupScales}, 0).requires_grad_();
385             quats = torch::cat({quats.detach(), splitQuats,
                    dupQuats}, 0).requires_grad_();
386
387             max2DSize = torch::cat({
388                 max2DSize,
389                 torch::zeros_like(splitScales.index({Slice(),
                        0})),
390                 torch::zeros_like(dupScales.index({Slice(), 0})
                        )
391             }, 0);
392
393             torch::Tensor splitIdcs = torch::where(splits)[0];
394
395             addToOptimizer(meansOpt, means, splitIdcs,
                    nSplitSamples);
```

```
396             addToOptimizer(scalesOpt, scales, splitIdcs,
                    nSplitSamples);
397             addToOptimizer(quatsOpt, quats, splitIdcs,
                    nSplitSamples);
398             addToOptimizer(featuresDcOpt, featuresDc, splitIdcs
                    , nSplitSamples);
399             addToOptimizer(featuresRestOpt, featuresRest,
                    splitIdcs, nSplitSamples);
400             addToOptimizer(opacitiesOpt, opacities, splitIdcs,
                    nSplitSamples);
401
402             torch::Tensor dupIdcs = torch::where(dups)[0];
403             addToOptimizer(meansOpt, means, dupIdcs, 1);
404             addToOptimizer(scalesOpt, scales, dupIdcs, 1);
405             addToOptimizer(quatsOpt, quats, dupIdcs, 1);
406             addToOptimizer(featuresDcOpt, featuresDc, dupIdcs,
                    1);
407             addToOptimizer(featuresRestOpt, featuresRest,
                    dupIdcs, 1);
408             addToOptimizer(opacitiesOpt, opacities, dupIdcs, 1)
                    ;
409
410             splitsMask = torch::cat({
411                 splits,
412                 torch::full({nSplitSamples * splits.sum().item<
                        int>() + dups.sum().item<int>()}, false,
                        torch::TensorOptions().dtype(torch::kBool).
                        device(device))
413             }, 0);
414
415             std::cout << "Added " << (means.size(0) -
                    numPointsBefore) << " gaussians, new count " <<
                    means.size(0) << std::endl;
416         }
417
418         if (doDensification){
419             // Cull
420             int numPointsBefore = means.size(0);
421
422             torch::Tensor culls = (torch::sigmoid(opacities) <
                    cullAlphaThresh).squeeze();
423             if (splitsMask.numel()){
424                 culls |= splitsMask;
425             }
426
427             if (step > refineEvery * resetAlphaEvery){
428                 const float cullScaleThresh = 0.5f; // cull
                        huge gaussians
429                 const float cullScreenSize = 0.15f; // % of
                        screen space
```

```
430                 torch::Tensor huge = std::get<0>(torch::exp(
                        scales).max(-1)) > cullScaleThresh;
431                 if (step < stopScreenSizeAt){
432                     huge |= max2DSize > cullScreenSize;
433                 }
434                 culls |= huge;
435             }
436 
437             int cullCount = torch::sum(culls).item<int>();
438             if (cullCount > 0){
439                 means = means.index({~culls}).detach().
                        requires_grad_();
440                 scales = scales.index({~culls}).detach().
                        requires_grad_();
441                 quats = quats.index({~culls}).detach().
                        requires_grad_();
442                 featuresDc = featuresDc.index({~culls}).detach
                        ().requires_grad_();
443                 featuresRest = featuresRest.index({~culls}).
                        detach().requires_grad_();
444                 opacities = opacities.index({~culls}).detach().
                        requires_grad_();
445 
446                 removeFromOptimizer(meansOpt, means, culls);
447                 removeFromOptimizer(scalesOpt, scales, culls);
448                 removeFromOptimizer(quatsOpt, quats, culls);
449                 removeFromOptimizer(featuresDcOpt, featuresDc,
                        culls);
450                 removeFromOptimizer(featuresRestOpt,
                        featuresRest, culls);
451                 removeFromOptimizer(opacitiesOpt, opacities,
                        culls);
452 
453                 std::cout << "Culled " << (numPointsBefore -
                        means.size(0)) << " gaussians, remaining "
                        << means.size(0) << std::endl;
454             }
455         }
456 
457         if (step < stopSplitAt && step % resetInterval ==
                refineEvery){
458             float resetValue = cullAlphaThresh * 2.0f;
459             opacities = torch::clamp_max(opacities, torch::
                    logit(torch::tensor(resetValue)).item<float>());
460 
461             // Reset optimizer
462             torch::Tensor param = opacitiesOpt->param_groups()
                    [0].params()[0];
463             #if TORCH_VERSION_MAJOR == 2 && TORCH_VERSION_MINOR
                     > 1
```

```
464                 auto pId = param.unsafeGetTensorImpl();
465             #else
466                 auto pId = c10::guts::to_string(param.
                        unsafeGetTensorImpl());
467             #endif
468             auto paramState = std::make_unique<torch::optim::
                    AdamParamState>(static_cast<torch::optim::
                    AdamParamState&>(*opacitiesOpt->state()[pId]));
469             paramState->exp_avg(torch::zeros_like(paramState->
                    exp_avg()));
470             paramState->exp_avg_sq(torch::zeros_like(paramState
                    ->exp_avg_sq()));
471             std::cout << "Alpha reset" << std::endl;
472         }
473
474         // Clear
475         xysGradNorm = torch::Tensor();
476         visCounts = torch::Tensor();
477         max2DSize = torch::Tensor();
478
479         if (device != torch::kCPU){
480             #ifdef USE_HIP
481                     c10::hip::HIPCachingAllocator::emptyCache()
                            ;
482             #elif defined(USE_CUDA)
483                     c10::cuda::CUDACachingAllocator::emptyCache
                            ();
484             #endif
485         }
486     }
487 }
488
489 void Model::save(const std::string &filename, int step){
490     if (fs::path(filename).extension().string() == ".splat"){
491         saveSplat(filename);
492     }else{
493         savePly(filename, step);
494     }
495     std::cout << "Wrote " << filename << std::endl;
496 }
497
498 void Model::savePly(const std::string &filename, int step){
499     std::ofstream o(filename, std::ios::binary);
500     int numPoints = means.size(0);
501
502     o << "ply" << std::endl;
503     o << "format binary_little_endian 1.0" << std::endl;
504     o << "comment Generated by opensplat at iteration " << step
            << std::endl;
505     o << "element vertex " << numPoints << std::endl;
```

```
506     o << "property float x" << std::endl;
507     o << "property float y" << std::endl;
508     o << "property float z" << std::endl;
509     o << "property float nx" << std::endl;
510     o << "property float ny" << std::endl;
511     o << "property float nz" << std::endl;
512 
513     for (int i = 0; i < featuresDc.size(1); i++){
514         o << "property float f_dc_" << i << std::endl;
515     }
516 
517     // Match Inria version
518     torch::Tensor featuresRestCpu = featuresRest.cpu().
        transpose(1, 2).reshape({numPoints, -1});
519     for (int i = 0; i < featuresRestCpu.size(1); i++){
520         o << "property float f_rest_" << i << std::endl;
521     }
522 
523     o << "property float opacity" << std::endl;
524 
525     o << "property float scale_0" << std::endl;
526     o << "property float scale_1" << std::endl;
527     o << "property float scale_2" << std::endl;
528 
529     o << "property float rot_0" << std::endl;
530     o << "property float rot_1" << std::endl;
531     o << "property float rot_2" << std::endl;
532     o << "property float rot_3" << std::endl;
533 
534     o << "end_header" << std::endl;
535 
536     float zeros[] = { 0.0f, 0.0f, 0.0f };
537 
538     torch::Tensor meansCpu = keepCrs ? (means.cpu() / scale) +
        translation : means.cpu();
539     torch::Tensor featuresDcCpu = featuresDc.cpu();
540     torch::Tensor opacitiesCpu = opacities.cpu();
541     torch::Tensor scalesCpu = keepCrs ? torch::log((torch::exp(
        scales.cpu()) / scale)) : scales.cpu();
542     torch::Tensor quatsCpu = quats.cpu();
543 
544     for (size_t i = 0; i < numPoints; i++) {
545         o.write(reinterpret_cast<const char *>(meansCpu[i].
            data_ptr()), sizeof(float) * 3);
546         o.write(reinterpret_cast<const char *>(zeros), sizeof(
            float) * 3); // TODO: do we need to write zero
            normals?
547         o.write(reinterpret_cast<const char *>(featuresDcCpu[i
            ].data_ptr()), sizeof(float) * featuresDcCpu.size(1)
            );
```

```
548         o.write(reinterpret_cast<const char *>(featuresRestCpu[
                i].data_ptr()), sizeof(float) * featuresRestCpu.size
                (1));
549         o.write(reinterpret_cast<const char *>(opacitiesCpu[i].
                data_ptr()), sizeof(float) * 1);
550         o.write(reinterpret_cast<const char *>(scalesCpu[i].
                data_ptr()), sizeof(float) * 3);
551         o.write(reinterpret_cast<const char *>(quatsCpu[i].
                data_ptr()), sizeof(float) * 4);
552     }
553
554     o.close();
555 }
556
557 void Model::saveSplat(const std::string &filename){
558     std::ofstream o(filename, std::ios::binary);
559     int numPoints = means.size(0);
560
561     torch::Tensor meansCpu = keepCrs ? (means.cpu() / scale) +
            translation : means.cpu();
562     torch::Tensor scalesCpu = keepCrs ? (torch::exp(scales.cpu
            ()) / scale) : torch::exp(scales.cpu());
563     torch::Tensor rgbsCpu = (sh2rgb(featuresDc.cpu()) * 255.0f)
            .toType(torch::kUInt8);
564     torch::Tensor opac = (1.0f + torch::exp(-opacities.cpu()));
565     torch::Tensor opacitiesCpu = torch::clamp(((1.0f / opac) *
            255.0f), 0.0f, 255.0f).toType(torch::kUInt8);
566     torch::Tensor quatsCpu = torch::clamp(quats.cpu() * 128.0f
            + 128.0f, 0.0f, 255.0f).toType(torch::kUInt8);
567
568     std::vector< size_t > splatIndices( numPoints );
569     std::iota( splatIndices.begin(), splatIndices.end(), 0 );
570     torch::Tensor order = (scalesCpu.index({"...", 0}) +
571                            scalesCpu.index({"...", 1}) +
572                            scalesCpu.index({"...", 2})) /
573                            opac.index({"...", 0});
574     float *orderPtr = reinterpret_cast<float *>(order.data_ptr
            ());
575
576     std::sort(splatIndices.begin(), splatIndices.end(),
577         [&orderPtr](size_t const &a, size_t const &b) {
578             return orderPtr[a] > orderPtr[b];
579         });
580
581     for (int i = 0; i < numPoints; i++){
582         size_t idx = splatIndices[i];
583
584         o.write(reinterpret_cast<const char *>(meansCpu[idx].
                data_ptr()), sizeof(float) * 3);
585         o.write(reinterpret_cast<const char *>(scalesCpu[idx].
```

```
            data_ptr()), sizeof(float) * 3);
586         o.write(reinterpret_cast<const char *>(rgbsCpu[idx].
                data_ptr()), sizeof(uint8_t) * 3);
587         o.write(reinterpret_cast<const char *>(opacitiesCpu[idx
                ].data_ptr()), sizeof(uint8_t) * 1);
588         o.write(reinterpret_cast<const char *>(quatsCpu[idx].
                data_ptr()), sizeof(uint8_t) * 4);
589     }
590     o.close();
591 }
592
593 void Model::saveDebugPly(const std::string &filename, int step)
        {
594     // A standard PLY
595     std::ofstream o(filename, std::ios::binary);
596     int numPoints = means.size(0);
597
598     o << "ply" << std::endl;
599     o << "format binary_little_endian 1.0" << std::endl;
600     o << "comment Generated by opensplat at iteration " << step
            << std::endl;
601     o << "element vertex " << numPoints << std::endl;
602     o << "property float x" << std::endl;
603     o << "property float y" << std::endl;
604     o << "property float z" << std::endl;
605     o << "property uchar red" << std::endl;
606     o << "property uchar green" << std::endl;
607     o << "property uchar blue" << std::endl;
608     o << "end_header" << std::endl;
609
610     torch::Tensor meansCpu = keepCrs ? (means.cpu() / scale) +
            translation : means.cpu();
611     torch::Tensor rgbsCpu = (sh2rgb(featuresDc.cpu()) * 255.0f)
            .toType(torch::kUInt8);
612
613     for (size_t i = 0; i < numPoints; i++) {
614         o.write(reinterpret_cast<const char *>(meansCpu[i].
                data_ptr()), sizeof(float) * 3);
615         o.write(reinterpret_cast<const char *>(rgbsCpu[i].
                data_ptr()), sizeof(uint8_t) * 3);
616     }
617
618     o.close();
619     std::cout << "Wrote " << filename << std::endl;
620 }
621
622 int Model::loadPly(const std::string &filename){
623     std::ifstream f(filename, std::ios::binary);
624     if (!f.is_open()) throw std::runtime_error("Invalid PLY
            file");
```

```
625
626     // Ensure we have a valid ply file
627     std::string line;
628     int numPoints;
629     int step;
630     size_t bytesRead = 0;
631
632     std::getline(f, line);
633     bytesRead += f.gcount();
634
635     if (line == "ply"){
636         std::getline(f, line);
637         bytesRead += f.gcount();
638         if (line == "format binary_little_endian 1.0"){
639             std::getline(f, line);
640             bytesRead += f.gcount();
641             const std::string pattern = "comment Generated by
                    opensplat at iteration ";
642
643             if (line.rfind(pattern, 0) == 0){
644                 step = std::stoi(line.substr(pattern.length()))
                        ;
645                 if (step >= 0){
646                     std::getline(f, line);
647                     bytesRead += f.gcount();
648                     const std::string pattern = "element vertex
                            ";
649
650                     if (line.rfind(pattern, 0) == 0){
651                         const int numPoints = std::stoi(line.
                                substr(pattern.length()));
652
653                         const char *requiredProps[] = {
654                             "property float x",
655                             "property float y",
656                             "property float z",
657                             "property float nx",
658                             "property float ny",
659                             "property float nz",
660                             "property float f_dc_"
661                             "property float f_rest_",
662                             "property float opacity",
663                             "property float scale_0",
664                             "property float scale_1",
665                             "property float scale_2",
666                             "property float rot_0",
667                             "property float rot_1",
668                             "property float rot_2",
669                             "property float rot_3",
670                             "end_header"
```

```
671                 };
672
673                 for (int i = 0; i < 6; i++){
674                     std::getline(f, line);
675                     bytesRead += f.gcount();
676                     if (line != requiredProps[i]){
677                         throw std::runtime_error(std::
                                string("PLY file's header
                                does not contain required
                                property: ") + requiredProps
                                [i]);
678                     }
679                 }
680                 std::getline(f, line);
681                 bytesRead += f.gcount();
682
683                 auto countPrefixes = [&f, &line](const
                    char *prefix){
684                     int n = 0;
685                     while(true){
686                         if (line.rfind(prefix, 0)
                                == 0){
687                             ++n;
688                             std::getline(f, line);
689                         } else {
690                             break;
691                         }
692                     }
693                     return n;
694                 };
695                 int featuresDcSize = countPrefixes("
                    property float f_dc_");
696                 int featuresRestSize = countPrefixes("
                    property float f_rest_");
697
698                 bool foundEnd = false;
699                 for (int i = 8; i < std::size(
                    requiredProps); i++){
700                     std::getline(f, line);
701                     bytesRead += f.gcount();
702
703                     if (line != requiredProps[i]){
704                         throw std::runtime_error(std::
                                string("PLY file's header
                                does not contain required
                                property: ") + requiredProps
                                [i]);
705                     }
706
707                     if (line == "end_header"){
```

```
708                 foundEnd = true;
709                 break;
710             }
711         }
712 
713         if (!foundEnd){
714             throw std::runtime_error("PLY file
                    header does not contain header
                    end");
715         }
716 
717         const size_t bytesPerPoint = sizeof(
                float) * (14 + featuresDcSize +
                featuresRestSize);
718         const size_t remainingFileSize = fs::
                file_size(filename) - bytesRead;
719         if (remainingFileSize != bytesPerPoint
                * numPoints){
720             std::cout << "Loading PLY..." <<
                    std::endl;
721 
722             float zeros[3];
723 
724             torch::Tensor meansCpu = torch::
                    zeros({numPoints, 3}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
725             torch::Tensor featuresDcCpu = torch
                    ::zeros({numPoints,
                    featuresDcSize}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
726             torch::Tensor featuresRestCpu =
                    torch::zeros({numPoints,
                    featuresRestSize}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
727             torch::Tensor opacitiesCpu = torch
                    ::zeros({numPoints, 1}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
728             torch::Tensor scalesCpu = torch::
                    zeros({numPoints, 3}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
729             torch::Tensor quatsCpu = torch::
                    zeros({numPoints, 4}, torch::
                    TensorOptions().dtype(torch::
                    kFloat32));
730 
```

```
731                     for (size_t i = 0; i < numPoints; i
                            ++){
732                         f.read(reinterpret_cast<char
                                *>(meansCpu[i].data_ptr()),
                                sizeof(float) * 3);
733                         f.read(reinterpret_cast<char
                                *>(&zeros[0]), sizeof(float)
                                 * 3);
734                         f.read(reinterpret_cast<char
                                *>(featuresDcCpu[i].data_ptr
                                ()), sizeof(float) *
                                featuresDcSize);
735                         f.read(reinterpret_cast<char
                                *>(featuresRestCpu[i].
                                data_ptr()), sizeof(float) *
                                 featuresRestSize);
736                         f.read(reinterpret_cast<char
                                *>(opacitiesCpu[i].data_ptr
                                ()), sizeof(float) * 1);
737                         f.read(reinterpret_cast<char
                                *>(scalesCpu[i].data_ptr()),
                                 sizeof(float) * 3);
738                         f.read(reinterpret_cast<char
                                *>(quatsCpu[i].data_ptr()),
                                sizeof(float) * 4);
739                     }
740                     if (keepCrs){
741                         meansCpu = (meansCpu -
                                translation) * scale;
742                         scalesCpu = torch::log(scale *
                                torch::exp(scalesCpu));
743                     }
744
745                     means = meansCpu.to(device).
                            requires_grad_();
746                     featuresDc = featuresDcCpu.to(
                            device).requires_grad_();
747                     featuresRest = featuresRestCpu.
                            reshape({numPoints, 3,
                            featuresRestSize/3}).transpose
                            (2, 1).to(device).requires_grad_
                            ();
748                     opacities = opacitiesCpu.to(device)
                            .requires_grad_();
749                     scales = scalesCpu.to(device).
                            requires_grad_();
750                     quats = quatsCpu.to(device).
                            requires_grad_();
751
752                     std::cerr << "Loaded " << means.
```

```
                                size(0) << " gaussians" << std::
                                endl;
753
754                             setupOptimizers();
755
756                             f.close();
757                             return step;
758                         } else {
759                             throw std::runtime_error("PLY file'
                                s data section is wrong size");
760                         }
761                     }
762                 } else {
763                     throw std::runtime_error("PLY file failed
                        sanity check: iteration count should not
                         begin at 0");
764                 }
765             } else if (line.rfind("comment Generated by
                opensplat")){
766                 throw std::runtime_error("PLY file does not
                    contain iteration count metadata. You can
                    edit the file to add this metadata manually,
                     by changing \"comment Generated by
                    opensplat\" to \"comment Generated by
                    opensplat at iteration 12345\", changing
                    12345 to the desired value.");
767             }
768         }
769     }
770     throw std::runtime_error("Invalid PLY file");
771 }
772
773 torch::Tensor Model::mainLoss(torch::Tensor &rgb, torch::Tensor
      &gt, float ssimWeight){
774     torch::Tensor ssimLoss = 1.0f - ssim.eval(rgb, gt);
775     torch::Tensor l1Loss = l1(rgb, gt);
776     std::cout << "SSIM: " << (1.0f - ssimLoss).item<float>() <<
          std::endl;
777     return (1.0f - ssimWeight) * l1Loss + ssimWeight * ssimLoss
        ;
778 }
```

Listing 6: Tweaked model.cpp file used in study

IADE

2025

Campus de Santos . Av. D. Carlos I, 4,1200-649 Lisboa | Portugal
Telf: (+351) 213 030 600 . iade@iade.pt

MOD.IADE-315/V01-EN; 05-05-2023