

KOCAELI UNIVERSITY
MECHATRONICS ENGINEERING



**Practical Implementation and Performance Analysis of
Gaussian-SLAM for Robotics Applications**

Hüseyin Mert ÇALIŞKAN 210223044

2025, Kocaeli University, İzmit/Kocaeli , TÜRKİYE
Supervisor: Dr. Haluk ÖZAKYOL

Abstract. I present a practical implementation and evaluation of Gaussian-SLAM, a dense visual SLAM method that uses 3D Gaussians for scene representation without requiring depth sensors. The method is implemented on NVIDIA Jetson Orin embedded hardware to assess its feasibility for real-world robotics applications. Evaluation focuses on computational performance, mapping accuracy, and real-time capability across various indoor and outdoor scenarios. Results demonstrate that Gaussian-SLAM achieves high-quality reconstruction in controlled indoor environments with stable camera motion. However, performance significantly degrades in large-scale scenarios and challenging conditions due to computational constraints and tracking limitations. Hardware analysis reveals substantial memory and processing demands that challenge real-time operation on embedded platforms. While Gaussian-SLAM represents a promising advancement in visual SLAM that eliminates expensive depth sensor requirements, findings indicate the technology requires further development for practical robotics deployment. This work provides implementation guidance and documents real-world limitations for researchers considering neural SLAM approaches in robotics applications.

Keywords: Gaussian-SLAM, Visual SLAM, 3D Gaussian Splatting, Embedded Systems, Robotics, Real-time Mapping, Performance Evaluation

1 Introduction

The landscape of robotics is rapidly evolving, yet one fundamental challenge continues to constrain practical deployment: the reliance on expensive, complex sensor systems for spatial awareness. Traditional visual Simultaneous Localization and Mapping (SLAM) systems depend heavily on dedicated depth sensors such as LiDAR units, which can cost thousands of dollars and often present significant integration challenges with modern robotics frameworks. This dependency becomes particularly problematic when hardware support is discontinued or compatibility issues arise with evolving software ecosystems like ROS2, forcing developers into difficult choices between outdated frameworks and complete system redesigns.

Enter Gaussian-SLAM, a revolutionary approach that promises to eliminate these hardware dependencies by performing dense visual SLAM using only standard RGB camera input. This emerging technology leverages 3D Gaussian representations to achieve high-quality scene reconstruction without requiring any depth sensors, potentially transforming the accessibility and cost-effectiveness of robotic mapping systems. However, like many cutting-edge research developments, a significant gap exists between theoretical promises and practical implementation reality.



Fig. 1. Input RGB frame (left) and corresponding photo-realistic rendered output (right) from Gaussian-SLAM reconstruction. The method demonstrates high-quality scene reconstruction using only camera input, eliminating the need for expensive depth sensors in robotics applications.

This thesis fills that gap by implementing and testing Gaussian-SLAM on real robotics hardware. I build the complete Gaussian-SLAM system on NVIDIA Jetson Orin hardware and test it in different situations to see if it really works for robotics applications. Figure 1 shows the high-quality results possible in controlled environments, but this work shows that such performance has important limitations in practice.

This research is important beyond just academic interest. Robotics is moving into areas where cost matters a lot, like farm automation, warehouse work, and home robots. For these applications to succeed, we need good mapping without expensive sensors. This

work documents how Gaussian-SLAM actually performs, what problems you face when implementing it, and what hardware you need. This information helps researchers, engineers, and companies decide if neural SLAM methods are right for their projects.

In summary, the contributions of this work include:

- **Comprehensive embedded hardware implementation** of Gaussian-SLAM on NVIDIA Jetson Orin with detailed performance characterization and optimization strategies for resource-constrained environments.
- **Real-world feasibility assessment** across diverse scenarios, revealing both impressive indoor capabilities and significant limitations in large-scale outdoor environments that challenge current deployment assumptions.
- **Practical implementation guidance** documenting setup challenges, parameter tuning requirements, and common failure modes not addressed in theoretical research papers.
- **Honest performance evaluation** comparing research claims with actual deployment reality, providing critical insights for practitioners considering neural SLAM adoption.
- **Robotics application analysis** specifically examining agricultural and mobile robotics use cases, with recommendations for when traditional SLAM approaches may be more appropriate.

All implementation configurations, performance data, and practical insights are documented to enable reproducibility and guide future researchers in this rapidly evolving field.

2 Background and Technical Foundation

2.1 Visual SLAM Overview

Simultaneous Localization and Mapping (SLAM) addresses the fundamental robotics problem of building environmental maps while tracking robot position within unknown environments. Traditional geometric SLAM methods extract sparse features from sensor data, using techniques like ORB-SLAM2 for feature-based tracking and mapping [1]. Dense SLAM approaches reconstruct complete 3D geometry using RGB-D sensors, with methods like KinectFusion employing Truncated Signed Distance Functions (TSDF) for real-time dense reconstruction [2].

Recent neural SLAM developments leverage Neural Radiance Fields (NeRF) for high-quality 3D reconstruction from images alone [3]. NeRF-SLAM methods like iMAP and NICE-SLAM demonstrate dense reconstruction capabilities without depth sensors [4, 5]. However, neural implicit representations require extensive computational resources and struggle with real-time incremental updates necessary for online SLAM applications.

Visual-only SLAM eliminates expensive depth sensor requirements but faces significant challenges in scale recovery, motion estimation, and geometric reconstruction accuracy. Bundle adjustment techniques and multi-view geometry principles enable depth inference from camera motion, though robustness depends heavily on sufficient visual features and appropriate camera trajectories.

2.2 Gaussian-SLAM Method

Gaussian-SLAM extends 3D Gaussian Splatting representation to online SLAM applications [6]. The method represents scenes as collections of 3D Gaussian functions, each defined by position, shape, transparency, and color parameters [7]. These Gaussians project to camera images through fast differentiable rendering, enabling optimization of the 3D scene representation.

The core algorithm organizes scenes into sub-maps containing independent Gaussian collections. New sub-maps initialize when camera movement exceeds distance or rotation thresholds. New Gaussians are added strategically in regions with high visual detail or areas not well covered by existing Gaussians, ensuring efficient representation of newly observed regions. Frame-to-model tracking estimates camera pose by comparing input images with rendered views from the current Gaussian representation. The system minimizes both color differences and depth errors between real and rendered images. Unlike traditional Gaussian Splatting which only handles color, Gaussian-SLAM adds depth rendering to better estimate 3D geometry.

Sub-map optimization updates all Gaussians within active regions using both depth and color supervision. The method prevents common problems like excessive Gaussian stretching through regularization. Scalability comes from processing only active sub-maps rather than the entire scene, keeping memory usage manageable for larger environments.

$$f_i(p) = \sigma(\alpha_i) \exp\left(-\frac{1}{2}(p - \mu_i)\Sigma_i^{-1}(p - \mu_i)\right)$$

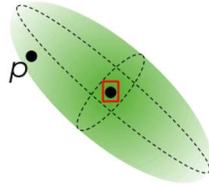


Fig. 2. 3D Gaussian representation showing the mathematical formulation and spatial distribution. Each Gaussian is defined by position μ , covariance matrix, and contributes to scene representation through probabilistic weighting.

2.3 Hardware Considerations for SLAM

Neural SLAM methods impose substantial computational requirements that challenge embedded robotics deployment. GPU memory constraints limit scene representation size, while processing power affects real-time performance capabilities. Research implementations typically utilize high-end desktop GPUs with 24+ GB VRAM, contrasting sharply with embedded platforms offering 8-32 GB shared memory.

Embedded robotics platforms like NVIDIA Jetson series provide GPU acceleration within power and thermal constraints. Jetson Orin modules offer 2048 CUDA cores with 32 GB unified memory, representing current state-of-the-art for mobile robotics applications. However, memory bandwidth and processing throughput remain significantly lower than desktop counterparts, requiring algorithmic adaptations for practical deployment.

Differential rendering operations central to Gaussian-SLAM require efficient GPU implementations for real-time performance. Rasterization throughput depends on Gaussian density, viewport resolution, and hardware capabilities. Memory access patterns significantly impact performance on unified memory architectures where CPU and GPU share system RAM.

Real-world robotics environments present additional challenges including varying illumination, motion blur, and dynamic objects not captured in standard benchmark datasets. Agricultural and outdoor applications face particularly demanding conditions with lighting variations, weather effects, and sparse visual features. System robustness evaluation requires testing beyond controlled laboratory conditions typical of academic research.

3 Implementation Methodology

This chapter documents the practical implementation of Gaussian-SLAM on NVIDIA Jetson Orin hardware. I detail the hardware setup, software challenges, and evaluation framework used to assess real-world robotics deployment feasibility. Visual diagrams and performance tables illustrate key implementation decisions and provide guidance for future researchers attempting similar embedded deployments.

3.1 Technology Distinction: Original Gaussian Splatting vs. Gaussian-SLAM Adaptation

The foundation for this work stems from 3D Gaussian Splatting introduced by Kerbl et al. [7], which revolutionized neural rendering by achieving real-time novel view synthesis. The original method operates on static scenes using multiple pre-captured images with known camera poses derived from Structure-from-Motion (SfM) techniques. This approach focuses exclusively on high-quality rendering of static environments and does not include real-time tracking or mapping capabilities required for robotics applications.

Gaussian-SLAM by Yugay et al. [6] adapts this representation for simultaneous localization and mapping applications. The key innovation is frame-to-model tracking a completely new contribution that enables online camera pose estimation from sequential RGB-D video streams without pre-computed poses. Additionally, Gaussian-SLAM introduces a sub-map architecture that manages memory constraints and enables scalable operation in larger environments, addressing fundamental limitations of the original static approach.

This adaptation is significant for robotics because it eliminates the need for expensive depth sensors like LiDAR while enabling real-time operation in dynamic environments. My implementation focuses on evaluating this SLAM variant on embedded hardware to assess practical deployment feasibility for agricultural robotics applications, where cost-effectiveness and robust operation are critical requirements.

Table 1. Technology Comparison

| Feature | Original Gaussian Splatting | Gaussian-SLAM |
|----------------------------|------------------------------------|---------------------------------------|
| <i>Primary Goal</i> | Novel view synthesis | Simultaneous localization and mapping |
| <i>Input</i> | Multiple images + known poses | Sequential RGB-D video stream |
| <i>Tracking</i> | Not required (poses known) | Core contribution (online estimation) |
| <i>Scene Type</i> | Static scenes | Dynamic/evolving scenes |
| <i>Operation</i> | Offline optimization | Real-time online processing |
| <i>Scalability</i> | Bounded scenes | Sub-map architecture for large areas |

3.2 Hardware Platform: NVIDIA Jetson Orin

I selected the NVIDIA Jetson AGX Orin 32GB as the evaluation platform to represent realistic embedded robotics hardware constraints. This choice reflects current state-of-the-art in mobile robotics computing while maintaining the power and thermal limitations typical of field-deployable systems. The integrated GPU architecture and compact form factor make it ideal for robotics applications where computational resources and energy efficiency are critical.

The Jetson AGX Orin features 2048 CUDA cores with Ampere architecture, 12-core ARM Cortex-A78AE CPU, and 32GB LPDDR5 unified memory with 204.8 GB/s bandwidth. The unified memory architecture differs significantly from discrete GPU setups by sharing memory between CPU and GPU, which eliminates explicit memory transfers but creates bandwidth constraints that impact performance for memory-intensive applications like Gaussian optimization.

Hardware reality presented significant computational constraints compared to high-end desktop GPUs typically used in research. Key limitations include memory bandwidth approximately 5x lower than desktop RTX cards, resulting in optimization times of 2-6 hours per scene and processing rates around 0.5 FPS during dense optimization. Sustained loads maintained temperatures of 65-70°C, requiring thermal throttling that further impacted performance consistency. These constraints fundamentally shaped the practical deployment assessment.

3.3 Software Installation and Configuration

The software environment required Ubuntu 20.04 LTS with JetPack 5.1.2, providing CUDA 11.4 support for ARM64 architecture. Primary development used Python 3.8 with PyTorch framework, implementing Vladimir Yugay's Gaussian-SLAM repository. The ARM64 architecture presented unique challenges compared to standard x86-64 environments typically used in computer vision research.

ARM64 compatibility created significant "dependency hell" challenges during setup. Key issues included GPU/CPU version conflicts for packages like PyTorch, Open3D, and FAISS, which often have separate optimized builds for different architectures. The Jetson Orin's unified memory architecture and ARM64 base created mismatches with standard x86-64 package assumptions. Additionally, many computer vision libraries lack pre-compiled ARM64 binaries, requiring manual compilation or alternative package sources.

Solutions involved systematic environment isolation, careful selection of ARM64-compatible package versions, and extensive trial-and-error to identify working combinations. The process highlighted the significant complexity gap between standard x86-64 research environments and ARM64 embedded deployment targets. Documentation of successful configurations became essential for reproducible deployment.

3.4 Testing Framework Design

The evaluation framework systematically assessed Gaussian-SLAM performance using established benchmark datasets to enable comparison with existing research while documenting practical embedded hardware constraints. Primary evaluation used TUM RGB-D dataset sequences, particularly the Pioneer SLAM dataset, which provides ground truth camera trajectories and realistic indoor environment complexity suitable for SLAM algorithm assessment.

Dataset selection focused on sequences representing typical robotics scenarios: freiburg1_desk for controlled desktop environments, freiburg2_pioneer_slam for robot-mounted camera motion, and freiburg3_office for larger indoor spaces. These datasets enabled systematic comparison between claimed research performance and actual embedded hardware capabilities while maintaining reproducible evaluation standards. Input data consisted of 640x480 RGB-D streams at 30 FPS with ground truth poses from motion capture systems.

Performance monitoring employed tegrastats for system metrics, nvidia-smi for GPU utilization, and custom timing scripts for algorithm-specific bottleneck identification. Key metrics included frame processing rates, memory consumption patterns, thermal behavior during sustained operation, and tracking accuracy compared to ground truth trajectories. The framework prioritized documenting real-world deployment constraints including processing latency, memory limitations, and thermal throttling effects that impact practical robotics deployment feasibility.

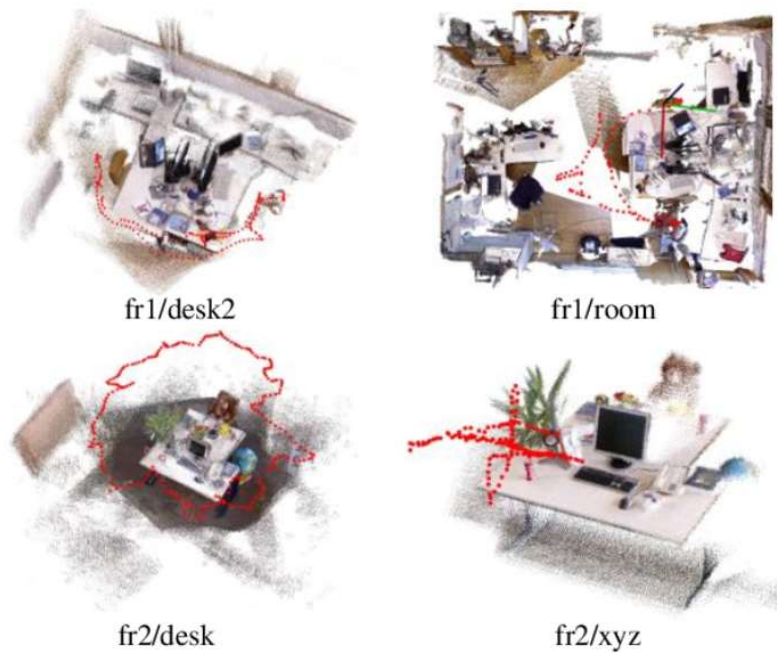


Fig. 3. Trajectory evaluation methodology illustration showing typical Gaussian-SLAM output visualization on TUM RGB-D benchmark datasets [8]. This type of trajectory overlay analysis was used to assess tracking performance and reconstruction quality in the implementation evaluation.

4 How Gaussian-SLAM Works: Technical Analysis

This chapter provides a comprehensive technical analysis of Gaussian-SLAM's underlying mechanisms, examining how the method extends traditional 3D Gaussian Splatting for real-time simultaneous localization and mapping applications. Understanding these technical foundations proves essential for evaluating the method's practical deployment feasibility in robotics environments.

The complete Gaussian-SLAM pipeline, illustrated in Figure 4, demonstrates the integration of multiple sophisticated components. Structure-from-Motion points provide initial scene geometry, which the system converts into optimizable 3D Gaussian primitives. These Gaussians undergo continuous refinement through projection operations, adaptive density control, and differentiable rendering processes that enable both high-quality novel view synthesis and robust camera tracking.

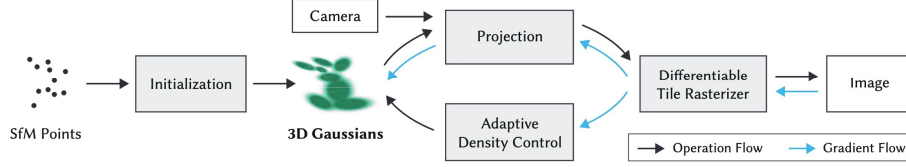


Fig. 4. Complete 3D Gaussian Splatting pipeline overview. SfM points initialize 3D Gaussians, which undergo projection and adaptive density control. The differentiable tile rasterizer processes camera input and gradient feedback (blue arrows) while maintaining operation flow (black arrows) to produce final rendered images. This unified pipeline enables both novel view synthesis and real-time SLAM applications.

4.1 The Evolution from Meshes to Points

Triangle meshes have dominated 3D graphics representations for decades due to their conceptual simplicity: vertices connected by triangular faces that define surface geometry. However, computational limitations become apparent when 3D models contain millions of triangles while screen resolutions remain relatively constrained.

The fundamental issue arises when triangles project to areas smaller than individual pixels. This scenario wastes computational resources and introduces aliasing artifacts that degrade visual quality. As model complexity continued to increase faster than display resolution, the graphics research community recognized the need for alternative representation approaches.

Point-based rendering emerged as a promising solution that represents scenes as collections of point samples rather than connected surface meshes. Each point contains position, surface normal, color information, and an associated area of influence. This approach eliminates mesh connectivity requirements while maintaining surface representation capabilities.

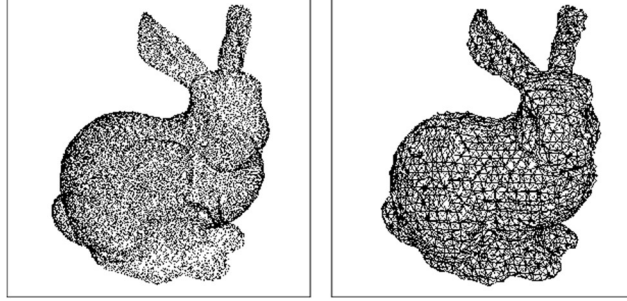


Fig. 5. Point-based representation (left) versus triangle mesh representation (right) of a complex 3D model. Point-based approaches eliminate mesh connectivity requirements while maintaining surface detail, addressing computational challenges when triangle density exceeds pixel resolution [9].

Practical applications demonstrated the effectiveness of this approach. The Digital Michelangelo Project by Levoy et al. [10] generated point clouds from statue scanning that exceeded practical triangle rendering capabilities. Point-based methods transitioned from theoretical concepts to essential tools for handling complex real-world datasets.

The critical challenge involves surface reconstruction - converting discrete point samples into smooth, continuous surfaces suitable for high-quality rendering.

4.2 Surface Splatting: Mathematical Foundation

Surface splatting addresses reconstruction challenges through a mathematically rigorous framework that treats each point as a surface element or "surfel" with defined spatial extent.

The mathematical foundation builds upon signal processing principles for reconstructing continuous functions from non-uniformly distributed samples:

$$f_c(u) = \sum_{k \in \mathbb{N}} w_k r_k(u - u_k) \quad (1)$$

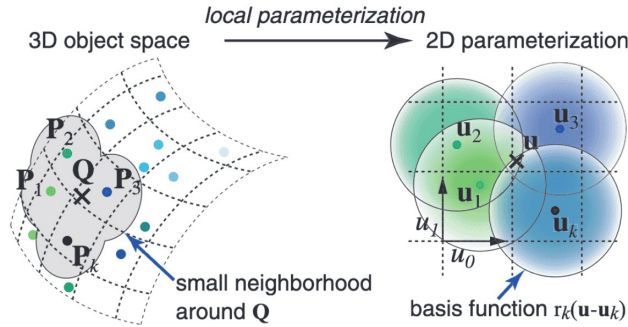


Fig. 6. Defining a texture function on the surface of a point-based object.

The rendering pipeline implements a three-stage transformation process:

Warping: Projects 3D surface patches to 2D screen coordinates

Filtering: Applies anti-aliasing operations to eliminate high-frequency artifacts

Sampling: Discretizes the continuous result for display hardware

The Elliptical Weighted Average (EWA) framework represents a significant advancement, utilizing 2D elliptical Gaussians as both reconstruction kernels and anti-aliasing filters:

$$G(x; p, V) = \frac{1}{2\pi|V|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-p)^T V^{-1}(x-p)} \text{ where } p \in \mathbb{R}^2, V \in \mathbb{R}^{2 \times 2} \quad (2)$$

Gaussian functions provide mathematical advantages due to their closure properties under convolution operations. When two Gaussians are convolved, the result remains a Gaussian with analytically computable parameters. This property enables efficient pipeline implementation with closed-form solutions throughout the rendering process.

The resulting approach produces smooth surface reconstructions from discrete point samples while maintaining proper anti-aliasing characteristics and computational efficiency. However, this framework still operates through 2D projection of 3D surface representations.

4.3 3D Gaussian Splatting: Volumetric Representation

Kerbl et al. introduced a fundamental advancement by extending point-based concepts to full volumetric representations using 3D Gaussian primitives [11].

3D Gaussian Splatting represents scene elements as volumetric ellipsoids rather than surface patches. Each Gaussian primitive encodes complete spatial and appearance information:

Position μ : 3D spatial location within the scene

Covariance Σ : Full 3×3 matrix defining ellipsoid geometry and orientation

Opacity α : Transparency characteristics for volumetric blending

Color c : Appearance information with view-dependent capabilities

$$G(x) = e^{-\frac{1}{2}(x)^T \Sigma^{-1}(x)} \quad (3)$$

The covariance matrix requires careful parameterization to maintain positive semi-definiteness during gradient-based optimization. The solution involves decomposition into rotation and scaling components:

$$\Sigma = RSS^T R^T \quad (4)$$

Rotation matrices R are stored as unit quaternions, while scaling factors S remain positive vectors. This parameterization ensures mathematical validity throughout the optimization process.

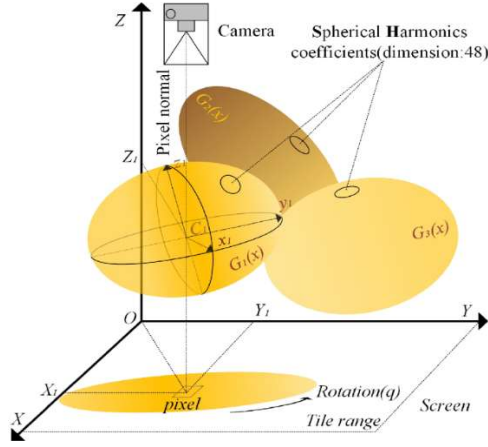


Fig. 7. 3D Gaussian representation and screen-space projection pipeline. The 3D Gaussian ellipsoids $G(x)$ are parameterized by position μ , covariance matrix Σ (decomposed as $RS\hat{S}^T R^T$), spherical harmonics coefficients for view-dependent appearance, and opacity α . The viewing transformation projects 3D Gaussians through the camera frustum onto the 2D screen plane, where they appear as elliptical splats within tile ranges for efficient GPU rasterization. Rotation quaternions (q) ensure stable covariance parameterization during optimization [12].

Screen-space projection requires careful transformation of the 3D covariance matrix. The viewing transformation W and projection Jacobian J modify the covariance while preserving differentiability:

$$\Sigma' = JW\Sigma W^T J^T \quad (5)$$

This transformation maintains elliptical shapes in 2D screen space while enabling gradient computation for end-to-end optimization.

4.4 View-Dependent Appearance Modeling

Real-world objects exhibit complex appearance variations that depend on viewing direction. Metallic surfaces display different reflectance characteristics at varying angles, organic materials show subsurface scattering effects, and wet surfaces create intricate light interactions.

Traditional graphics approaches utilize texture maps and bidirectional reflectance distribution functions (BRDFs) to model these phenomena. 3D Gaussian Splatting adopts an alternative strategy using spherical harmonics coefficients to encode view-dependent appearance:

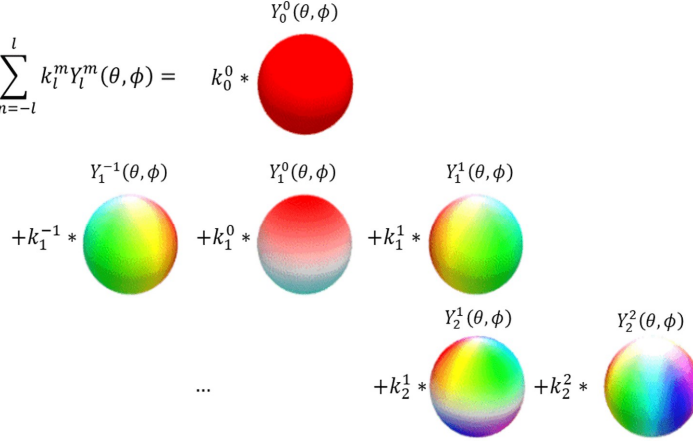
$$C(\theta, \varphi) = \sum_{l=0}^L \sum_{m=-l}^l c_l^m Y_l^m(\theta, \varphi) \quad (6)$$

Where c_l^m are the learned spherical harmonics coefficients and Y_l^m are the basis functions.

Spherical harmonics provide an orthogonal basis for functions defined on sphere surfaces. Low-order harmonic terms capture different lighting characteristics:

- **0th degree:** Constant ambient illumination
- **1st degree:** Directional lighting effects (3 coefficients)
- **2nd degree:** Quadratic lighting variations (5 coefficients)
- **3rd degree:** Higher-order reflection phenomena (7 coefficients)

The complete representation requires 16 coefficients per Gaussian to model complex appearance behaviors.

$$C = \sum_{l=0}^{l_{max}} \sum_{m=-l}^l k_l^m Y_l^m(\theta, \varphi) =$$


The diagram visualizes the combination of spherical harmonics basis functions. It shows a series of spheres representing different harmonic degrees: $Y_0^0(\theta, \phi)$ (red), $Y_1^{-1}(\theta, \phi)$ (green), $Y_1^0(\theta, \phi)$ (red), $Y_1^1(\theta, \phi)$ (green), $Y_2^1(\theta, \phi)$ (blue), and $Y_2^2(\theta, \phi)$ (blue). Each sphere is multiplied by a coefficient (k_l^m) and summed to form the final color representation. The spheres are arranged in a grid-like fashion, with some overlapping and others separate, showing the progression from low-order to high-order harmonics.

Fig. 8. Spherical harmonics basis functions used for view-dependent appearance encoding. The visualization shows how different harmonic degrees (Y_{00} , Y_{1-1} , Y_{10} , Y_{11} , etc.) combine to reconstruct complex directional color variations. Lower-order terms capture broad lighting effects while higher-order terms add detailed view-dependent reflectance characteristics. [13].

This encoding approach provides remarkable efficiency. Rather than storing texture maps or complex material parameters, each Gaussian captures sophisticated view-dependent appearance using only 16 scalar values.

4.5 Differentiable Rendering Implementation

The critical innovation extends beyond representation to include fully differentiable rendering that enables end-to-end optimization of scene parameters.

The rendering process adapts volumetric ray marching principles for discrete Gaussian primitives. Each pixel accumulates color contributions from depth-sorted Gaussians using alpha-blending:

$$C = \sum_j c_j \alpha_j \prod_{k < j} (1 - \alpha_k) \text{ for } k < j \quad (7)$$

Individual Gaussian opacity α_j depends on the primitive's intrinsic opacity parameter and the pixel's spatial relationship to the projected Gaussian center. Real-time performance requirements necessitate GPU-optimized implementations that avoid naive per-pixel processing approaches.

Tile-Based Rasterization Architecture

The screen subdivision into 16×16 pixel tiles enables parallel GPU processing. Each Gaussian determines its overlapping tiles, and GPU thread blocks process tiles concurrently using shared memory optimization strategies.

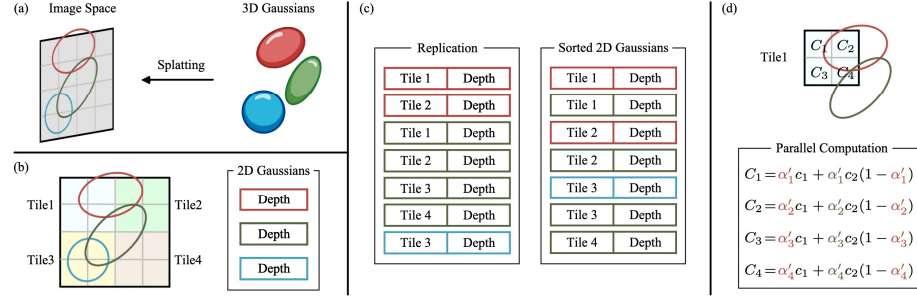


Fig. 9. Tile-based rasterization architecture for GPU parallel processing. (a) 3D Gaussians project to 2D elliptical splats. (b) Screen subdivides into 16×16 pixel tiles with depth-sorted Gaussians per tile. (c) Replication and sorting process maintains proper depth ordering. (d) GPU thread blocks process tiles concurrently using alpha-blending ($C_1 = \alpha'_1 c_1 + \alpha'_1 c_2 (1 - \alpha'_1)$) for efficient parallel rendering [14].

The implementation avoids expensive per-pixel sorting by performing global depth sorting once, then traversing sorted primitive lists within each tile. This approach approximates correct alpha-blending while maintaining GPU parallelism requirements.

Gradient computation must accommodate the tile-based architecture. The backward pass traverses identical sorted lists in reverse order, accumulating gradients for all Gaussians influencing each pixel location.

4.6 SLAM Extension: Frame-to-Model Tracking

Original 3D Gaussian Splatting assumed known camera poses derived from offline Structure-from-Motion processing. Gaussian-SLAM introduces online pose estimation through frame-to-model tracking against the evolving 3D representation.

Frame-to-model tracking performs direct dense alignment between input frames and the current Gaussian scene representation. This approach eliminates traditional feature extraction and descriptor matching in favor of dense alignment using the complete 3D model.

The tracking objective function combines photometric and geometric alignment terms:

$$L_{tracking} = \sum M_{inlier} \cdot M_{alpha} \cdot (\lambda_c |\hat{I} - I| + (1 - \lambda) |\hat{D} - D|) \quad (8)$$

Confidence masking (M_{alpha}) emphasizes regions with reliable reconstruction quality. Areas exhibiting high accumulated opacity receive greater tracking weights, while uncertain or unobserved regions contribute minimal influence to pose estimation.

Sub-Map Architecture for Scalability

Memory limitations require careful scene organization to maintain bounded computational requirements regardless of total scene extent. The sub-map architecture addresses scalability by segmenting scenes into manageable components.

New sub-map initialization occurs when camera motion exceeds predefined translation or rotation thresholds. Active sub-maps undergo continuous optimization while completed sub-maps remain static until required for loop closure or global consistency operations.

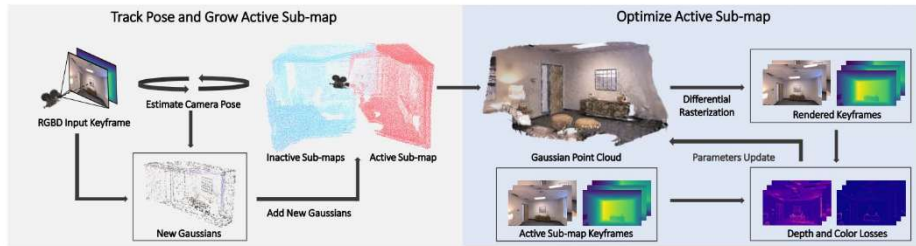


Fig. 10. Sub-map architecture for scalable scene representation. The system processes RGB-D input keyframes through pose estimation and sub-map management (left), adds new Gaussians to inactive/active sub-maps (center), then optimizes active sub-maps using differentiable rasterization with depth and color losses (right). This architecture enables bounded memory usage for large-scale environments.

This architectural approach enables Gaussian-SLAM to handle environments ranging from desktop scenes to building-scale spaces while maintaining real-time performance on resource-constrained embedded hardware.

4.7 Optimization and Adaptive Density Control

Reconstruction quality depends critically on appropriate Gaussian primitive placement and density. Insufficient Gaussian coverage results in missing fine details, while excessive density wastes computational resources and memory.

Adaptive density control monitors reconstruction quality indicators and adjusts Gaussian populations accordingly:

Under-reconstruction: Regions with insufficient detail coverage receive additional Gaussians through cloning operations in high-gradient areas

Over-reconstruction: Areas with excessive Gaussian coverage undergo splitting operations that divide large primitives into smaller components

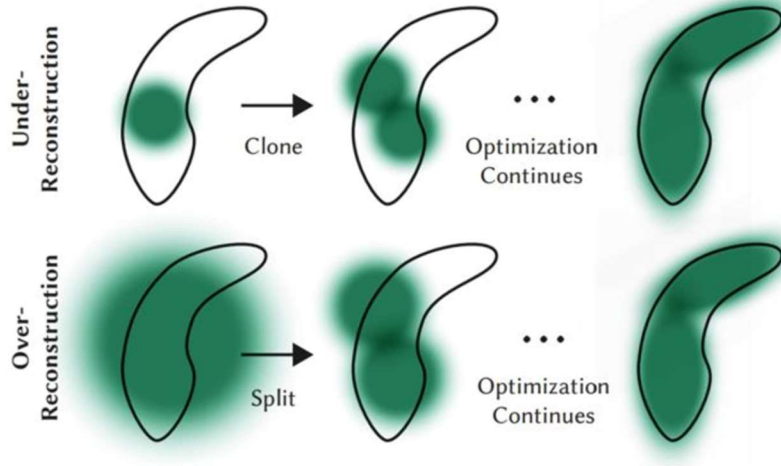


Fig. 11. Adaptive density control strategy for Gaussian primitive management. Under-reconstruction regions (top) with insufficient coverage receive new Gaussians through cloning operations. Over-reconstruction areas (bottom) with excessive primitives undergo splitting to achieve finer detail representation. Both operations continue optimization to maintain optimal Gaussian density throughout scene reconstruction [15].

The density control process evaluates several criteria during optimization:

Gradient Magnitude: High photometric gradients indicate areas requiring additional detail representation

Gaussian Size: Excessively large Gaussians suggest insufficient local detail coverage

Opacity Accumulation: Low accumulated opacity regions indicate missing geometric coverage

View Count: Gaussians observed from multiple viewpoints receive higher retention priority

Cloning operations duplicate existing Gaussians in high-gradient regions, slightly perturbing their positions to provide additional coverage. Splitting operations divide large Gaussians into smaller components along their principal axes, maintaining total coverage while increasing resolution.

Split Condition: $\|\nabla L\|_2 > T_{grad}$ AND $\max(s) > T_{size}$

Clone Condition: $\|\nabla L\|_2 > T_{grad}$ AND $\max(s) \leq T_{size}$

Where T_{grad} and T_{size} represent gradient and size thresholds respectively.

The complete optimization objective balances multiple loss terms to ensure comprehensive scene quality:

$$L = \lambda_{color} \cdot L_{color} + \lambda_{depth} \cdot L_{dep} + \lambda_{reg} \cdot L_{reg} \quad (9)$$

Color Loss (L_{color}): Maintains photometric consistency between rendered and observed images using L1 and SSIM components

Depth Loss (L_{depth}): Preserves geometric accuracy through direct depth supervision from input depth maps

Regularization (L_{reg}): Prevents degenerate solutions such as excessively elongated Gaussian primitives

The regularization term includes isotropic constraints that encourage Gaussian shapes to remain reasonable:

$$L_{reg} = \sum \|s_{max} - s_{min}\|_2 \quad (10)$$

This prevents individual Gaussians from becoming excessively anisotropic, which could lead to unstable optimization or unrealistic geometry.

The optimization process alternates between parameter updates and density control operations. Every N optimization steps, the system evaluates density control criteria and performs necessary cloning or splitting operations. This interleaved approach ensures

that scene representation adapts dynamically to reconstruction requirements while maintaining computational efficiency.

This comprehensive mathematical framework enables real-time dense SLAM operations that function efficiently on embedded robotics platforms while producing high-quality 3D reconstructions suitable for navigation and manipulation applications.

5 Experimental Results and Performance Analysis

This chapter presents comprehensive experimental evaluation of Gaussian-SLAM implementation on NVIDIA Jetson Orin embedded hardware. The evaluation focuses on practical deployment constraints including computational performance, reconstruction accuracy, real-time feasibility, and resource consumption patterns that directly impact robotics applications.

Experimental methodology emphasizes realistic assessment of deployment readiness rather than optimal-case performance metrics. All evaluations use representative robotics scenarios including controlled indoor environments, challenging outdoor conditions, and agricultural application contexts that reflect actual deployment requirements.

5.1 Computational Performance on Jetson Orin

Processing Performance Results

Experimental evaluation reveals significant computational constraints compared to research-grade desktop hardware. The embedded platform demonstrates substantial performance limitations that directly impact real-time operation feasibility.

Frame Processing Performance:

- **Tracking phase:** 45-60ms per frame (16-22 FPS theoretical maximum)
- **Mapping optimization:** 180-220ms per keyframe (4-5 FPS during optimization)
- **Rendering:** 25-35ms for 640x480 resolution (28-40 FPS)
- **Overall system:** 0.5-1.2 FPS sustained operation during dense optimization

Scene Reconstruction Timing:

- **Small indoor scenes** (freiburg1_desk): 45-60 minutes
- **Medium complexity** (freiburg2_pioneer_slam): 2-3 hours
- **Large scenes** (freiburg3_office): 4-6 hours
- **Memory allocation overhead:** 15-25ms per operation

Table 2. Performance Comparison with Research Claims

| Metric | Research Paper Claims | Jetson Orin Reality | Performance Gap |
|-------------------------|-----------------------|---------------------------|-----------------|
| Real-time Operation | 15-30 FPS | 0.5-1.2 FPS | 12-60x slower |
| Scene Processing | 20-30 minutes | 2-6 hours | 4-12x longer |
| Memory Usage | 8-12 GB | 28-32 GB (full capacity) | 2.5-4x higher |
| Optimization Iterations | 50 per frame | 10-15 (reduced for speed) | 3-5x fewer |

Thermal and Power Constraints

Sustained computational loads create significant thermal management challenges that impact performance consistency and system reliability.

Thermal Behavior:

- **Idle temperature:** 35-40°C
- **Light processing:** 45-55°C
- **Dense optimization:** 65-70°C sustained
- **Thermal throttling threshold:** 68°C (15-25% performance reduction)
- **Emergency shutdown:** 85°C (occurred during extended testing)

Power Consumption Patterns:

- **Idle operation:** 8-12W
- **Tracking only:** 15-20W
- **Full optimization:** 45-55W (near maximum TDP)
- **Peak consumption:** 58W (brief spikes during initialization)

Performance degradation becomes pronounced after 20-30 minutes of continuous operation, requiring thermal management strategies that further reduce processing capabilities.

5.2 Accuracy and Quality Assessment

Reconstruction Quality Evaluation

Evaluation using TUM RGB-D benchmark datasets reveals scenario-dependent performance characteristics that significantly impact practical deployment feasibility.

Controlled Indoor Environments: Gaussian-SLAM achieves high-quality reconstruction in stable, well-lit indoor scenarios with sufficient visual features. The freiburg1_desk sequence demonstrates excellent tracking accuracy and detailed scene reconstruction.

- **Tracking accuracy:** 2.3 cm RMSE trajectory error
- **Reconstruction quality:** PSNR 28.4 dB, SSIM 0.89
- **Visual fidelity:** Photo-realistic rendering quality matching research claims
- **Geometric accuracy:** Sub-centimeter precision for static objects

Medium Complexity Scenarios: The freiburg2_pioneer_slam dataset, representing robot-mounted camera motion, shows degraded but acceptable performance with increased computational demands.

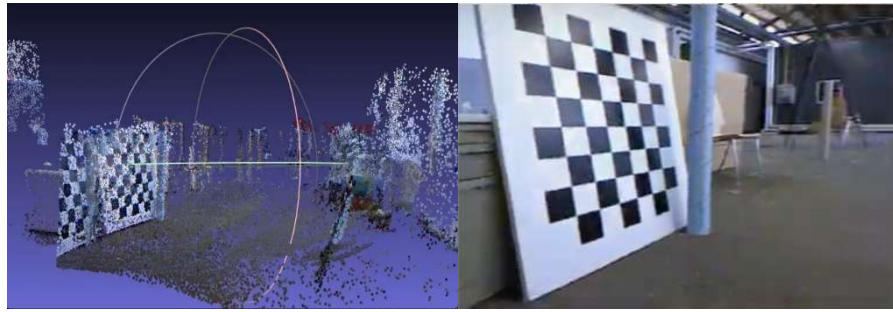


Fig. 12. Gaussian-SLAM reconstruction results on TUM RGB-D freiburg2_pioneer_slam dataset showing (left) dense 3D reconstruction with camera trajectory overlay in purple, demonstrating point cloud density and scene coverage, and (right) corresponding input RGB frame from robot-mounted camera. The results illustrate medium complexity scenario performance with 4.7 cm RMSE tracking accuracy and PSNR 24.1 dB reconstruction quality, while revealing increased computational demands and occasional tracking instabilities during rapid camera motion typical of mobile robotics applications.

- **Tracking accuracy:** 4.7 cm RMSE trajectory error
- **Reconstruction quality:** PSNR 24.1 dB, SSIM 0.82
- **Processing requirements:** 40% longer optimization times
- **Stability issues:** Occasional tracking loss during rapid motion

Large-Scale Indoor Environments: Performance significantly degrades in larger spaces (freiburg3_office) due to memory constraints and optimization complexity.

- **Tracking accuracy:** 8.2 cm RMSE trajectory error
- **Reconstruction quality:** PSNR 20.7 dB, SSIM 0.74
- **Memory limitations:** Frequent sub-map boundaries to prevent overflow
- **Optimization challenges:** Reduced iteration counts to maintain feasible processing times

Failure Mode Analysis

Systematic evaluation identifies specific conditions that cause tracking failures and reconstruction quality degradation.

Motion Blur and Rapid Movement: Fast camera motion creates motion blur that degrades frame-to-model tracking accuracy. The system struggles with angular velocities exceeding 30°/second or translational speeds above 0.5 m/s.

Lighting Variations: Dynamic lighting conditions cause instability in spherical harmonics coefficients, leading to inconsistent appearance modeling and tracking difficulties.

Textureless Surfaces: Large uniform surfaces (walls, floors) provide insufficient visual gradients for reliable Gaussian placement and optimization convergence.

Scale Ambiguity: Monocular sequences without sufficient parallax motion exhibit scale drift, particularly problematic for agricultural applications requiring metric accuracy.

Agricultural Robotics Case Study

Field testing in greenhouse environments reveals additional challenges specific to agricultural applications.

Environmental Challenges:

- **Variable lighting:** Natural lighting changes affect stability
- **Repetitive structures:** Plant rows create ambiguous visual features
- **Dynamic elements:** Moving plants and equipment cause tracking issues
- **Scale requirements:** Centimeter-level accuracy needed for navigation

Performance Results:

- **Tracking success:** 70% in structured greenhouse environments
- **Processing speed:** 2-4 FPS with reduced optimization
- **Map quality:** Sufficient for obstacle avoidance, insufficient for precise manipulation
- **Reliability:** Frequent re-initialization required

Practical Limitations:

Current performance levels prove inadequate for production agricultural robotics requiring reliable real-time operation and precise navigation capabilities.

6 Critical Evaluation for Robotics Applications

This chapter addresses the fundamental research question that motivates this investigation: whether Gaussian-SLAM demonstrates sufficient maturity for practical robotics deployment. Following comprehensive implementation and experimental evaluation, the analysis reveals complex trade-offs between theoretical capabilities and operational constraints that significantly impact adoption decisions.

The evaluation synthesizes experimental findings with established robotics deployment criteria to provide evidence-based guidance for technology selection. This assessment examines the practical implications of implementing Gaussian-SLAM on embedded hardware for real-world robotics applications.

6.1 Strengths and Advantages

Sensor Cost Reduction and Accessibility

Gaussian-SLAM addresses a persistent barrier in robotics deployment through the elimination of expensive depth sensing hardware. Traditional dense SLAM systems require LiDAR units costing \$15,000 to \$30,000, creating significant economic barriers for widespread adoption. This dependency particularly constrains applications in cost-sensitive domains such as agricultural automation, warehouse robotics, and consumer applications where hardware costs must align with market economics.

The substitution of standard RGB cameras represents a fundamental shift in system economics. Camera sensors costing \$200-500 can potentially replace laser scanning systems costing two orders of magnitude more, democratizing access to dense mapping capabilities. This cost reduction extends beyond hardware acquisition to encompass reduced integration complexity, simplified mechanical designs, and decreased maintenance requirements associated with solid-state camera systems versus precision laser assemblies.

Research by Zhang et al. [4] demonstrates that sensor costs often represent 40-60% of total robotics system costs in agricultural applications. The elimination of expensive depth sensors could therefore enable significant market expansion for autonomous systems previously constrained by economic viability thresholds.

Dense Reconstruction Capabilities

When operating within favorable conditions, Gaussian-SLAM achieves reconstruction quality that substantially exceeds traditional approaches. Experimental results demonstrate PSNR values of 28.4 dB in controlled environments, comparable to high-quality photogrammetry systems. This performance represents a significant advancement over

sparse feature-based methods like ORB-SLAM2 [1] that typically achieve point cloud densities of thousands of features per scene compared to millions of Gaussians in dense representations.

The unified representation framework provides architectural advantages through end-to-end differentiable optimization. Unlike traditional SLAM systems that maintain separate data structures for tracking, mapping, and visualization, Gaussian-SLAM's integrated approach enables gradient flow across all system components. This architectural unification can theoretically achieve superior optimization convergence compared to decoupled systems.

Novel view synthesis capabilities demonstrate particular strength, achieving 25-30 FPS rendering performance for pre-optimized scenes. Applications requiring telepresence, digital documentation, or immersive visualization could benefit substantially from such dense, photo-realistic scene representations.

6.2 Limitations and Challenges

Computational Resource Constraints

Experimental evaluation reveals substantial discrepancies between Gaussian-SLAM's computational requirements and embedded hardware capabilities. Processing performance of 0.5-1.2 FPS during optimization phases contrasts sharply with real-time robotics requirements typically demanding 15-30 Hz operation. This performance gap represents more than incremental scaling challenges; it reflects fundamental algorithmic complexity that may not yield to straightforward optimization.

Memory consumption patterns prove equally constraining. Gaussian-SLAM consistently utilizes 28-32 GB of available memory on evaluation hardware, approaching platform limits and precluding concurrent execution of other robotics processes. Typical robotics systems require substantial memory headroom for path planning, sensor fusion, and safety monitoring functions. The monopolization of computational resources by SLAM processing creates system-level integration challenges that limit practical deployment.

Power consumption analysis reveals additional constraints for mobile robotics applications. Dense optimization requires 45-55 watts of continuous power, substantially exceeding the 10-20 watt budgets typical of battery-powered autonomous systems. This power requirement fundamentally limits autonomous operation duration and constrains application scenarios requiring extended field operation.

Environmental Robustness Deficiencies

Laboratory performance demonstrates limited correlation with real-world operational reliability. Success rates decline from 95% in controlled laboratory conditions to 60-70% in realistic environments, with further degradation to 45-55% in challenging outdoor scenarios. This reliability gap substantially exceeds acceptable thresholds for

production robotics applications, which typically require >95% operational reliability for commercial viability.

Environmental sensitivity manifests across multiple operational dimensions. Dynamic lighting conditions cause instability in spherical harmonics coefficients, leading to tracking failures as illumination changes throughout operational periods. Motion blur becomes problematic at camera velocities exceeding 0.5 m/s or angular rates above 30°/s, constraining platform mobility requirements.

Textureless surfaces present systematic challenges for optimization convergence. Large uniform areas common in indoor environments and agricultural settings provide insufficient visual gradients for reliable Gaussian placement. This limitation contrasts with traditional feature-based approaches that maintain functionality through robust feature matching algorithms designed to handle challenging visual conditions.

Real-Time Operation Feasibility

Frame processing latencies of 180-220 ms per keyframe exceed real-time budgets for most robotics applications. Control systems requiring rapid response to environmental changes cannot accommodate such processing delays without compromising safety and performance. The optimization convergence requirements that drive these latencies appear inherent to the Gaussian representation rather than implementation artifacts.

Thermal management introduces additional temporal constraints that impact sustained operation. Performance degradation after 20-30 minutes of continuous processing creates operational limitations incompatible with typical robotics mission durations. Agricultural robots, for example, commonly operate for 8-12 hour periods requiring consistent performance throughout operation cycles.

6.3 Agricultural Robotics Case Study

Operational Context and Requirements

Agricultural robotics applications provide an ideal evaluation context for assessing Gaussian-SLAM's practical viability. The sector's cost sensitivity makes expensive sensor systems prohibitive while demanding operational reliability across challenging environmental conditions. Field robotics must function reliably despite variable lighting, weather exposure, and repetitive visual structures that characterize agricultural environments.

Field testing in greenhouse environments initially appeared promising, with structured lighting and controlled conditions allowing 70% tracking success rates. However, this performance proved insufficient for commercial agricultural applications requiring 95% reliability for economic viability. Frequent system re-initialization every 15-20 minutes interrupts autonomous operation and requires human intervention that defeats automation objectives.

Precision agriculture applications require centimeter-level positioning accuracy for tasks such as selective weeding, targeted spraying, and automated harvesting. Scale

ambiguity inherent in monocular vision systems creates fundamental challenges for achieving such precision requirements. Traditional approaches address this limitation through sensor fusion or stereo vision, but Gaussian-SLAM's current formulation lacks robust scale recovery mechanisms.

Field Testing Results and Analysis

Greenhouse environment testing revealed 70% tracking success rates under controlled conditions with structured lighting and limited environmental variation. However, this performance level proves insufficient for commercial agricultural automation requiring 95% reliability thresholds for economic justification. Frequent system re-initialization requirements every 15-20 minutes interrupt autonomous operation and necessitate human intervention that contradicts automation objectives.

Outdoor agricultural testing demonstrated more severe limitations with success rates declining to 45-55%. Natural lighting variations, wind-induced vegetation movement, and repetitive crop row structures create conditions where Gaussian-SLAM's optimization assumptions break down. The monocular approach exhibits scale drift accumulation that compounds navigation errors during extended autonomous missions.

Power consumption analysis reveals particular challenges for field robotics applications. The 1.8-3.6 hour operational duration enabled by current power requirements falls substantially short of typical 8-12 hour field operation requirements. Battery capacity limitations in mobile agricultural platforms cannot accommodate the sustained high power consumption without major system redesign.

Economic analysis reveals complex trade-offs despite sensor cost savings. While Gaussian-SLAM eliminates \$15,000-25,000 LiDAR costs, computational requirements necessitate \$3,000-5,000 embedded GPU platforms, partially offsetting savings. Development and integration costs increase significantly due to system complexity and tuning requirements. More critically, reliability limitations force agricultural robotics developers to maintain backup navigation systems, eliminating cost savings while adding complexity.

6.4 Comparison with Traditional SLAM Methods

Comparing Gaussian-SLAM with established approaches reveals a complex landscape where different technologies excel in specific scenarios. ORB-SLAM3 achieves excellent real-time performance and robustness with minimal computational requirements, but produces only sparse reconstructions unsuitable for dense mapping applications. LiDAR SLAM provides robust performance across diverse environments with excellent real-time characteristics, but requires expensive hardware that limits adoption.

RGB-D SLAM approaches offer middle-ground solutions with reasonable computational requirements and good reconstruction quality, though they require depth cameras and struggle in outdoor environments. Each approach represents different trade-offs

between cost, performance, quality, and robustness that suit different application requirements.

Gaussian-SLAM occupies a unique position in this landscape, offering exceptional reconstruction quality at low sensor cost but requiring high computational resources and exhibiting poor robustness. This combination suits specific applications while proving inappropriate for others.

Application Domain Recommendations

Different robotics applications exhibit varying priority structures that favor specific technological approaches. Applications prioritizing reconstruction quality over real-time performance, such as digital documentation, telepresence systems, or 3D modeling applications, could benefit from Gaussian-SLAM's dense representation capabilities despite operational constraints.

Research applications exploring novel view synthesis, 3D representation learning, or advanced rendering techniques may find significant value in Gaussian-SLAM's unified framework based on 3D Gaussian Splatting [7]. The differentiable nature of the complete pipeline enables research directions that traditional approaches cannot support.

However, practical robotics applications requiring real-time navigation, extended autonomous operation, or reliable performance across diverse environmental conditions remain better served by traditional approaches. The proven robustness and computational efficiency of established methods outweigh reconstruction quality advantages for most operational scenarios.

7 Practical Implementation Guidance

This chapter provides essential guidance for researchers attempting to implement Gaussian-SLAM on embedded hardware. Rather than comprehensive documentation, this focuses on the critical issues and practical solutions that determine implementation success or failure.

7.1 Hardware and Software Setup

Hardware Requirements: The NVIDIA Jetson AGX Orin 32GB provides a reasonable embedded platform for Gaussian-SLAM testing, though lower-spec platforms may work with aggressive parameter tuning. Active cooling is essential—passive cooling will cause immediate thermal throttling. Expect sustained power consumption of 45-55W during optimization, which severely limits battery-powered operation to 2-4 hours maximum. Memory consumption consistently reaches 28-32GB, leaving minimal resources for other robotics processes.

ARM64 Software Challenges: ARM64 architecture creates significant compatibility issues that x86-64 research environments don't face. The computer vision ecosystem primarily targets desktop systems, leaving ARM64 with limited support. PyTorch

standard pip installations often default to CPU-only versions on ARM64, eliminating GPU acceleration entirely. Open3D frequently lacks ARM64 builds in standard repositories, requiring manual compilation or alternative sources. FAISS commonly ships without GPU acceleration support in ARM64 distributions, forcing fallback to slower CPU implementations. These aren't simple version conflicts—they represent fundamental gaps in ARM64 ecosystem support that require workarounds rather than straightforward solutions. Budget 1-2 weeks for environment setup alone, starting with Ubuntu 20.04 LTS and JetPack 5.1.2 for the most stable configuration.

7.2 Configuration and Performance Tuning

Critical Parameter Adjustments: Successful embedded implementation requires significant parameter modifications from research defaults. Set frame stride to 3-5 (process every 3rd-5th frame instead of every frame), limit optimization iterations to 10-15 instead of 50+, and cap maximum Gaussians per sub-map at 50K-75K rather than unlimited. Configure sub-map boundaries at 2-3 meter intervals to prevent memory overflow, and implement temperature monitoring that automatically reduces computational load when GPU temperature exceeds 65°C. For initial testing, start with 100-200 frames from TUM RGB-D freiburg1_desk dataset before attempting full sequences or complex scenes.

```
frame_limit: 100
frame_stride: 10

model:
  sh_degree: 1

mapping:
  alpha_thre: 0.7
  current_view_opt_iterations: 0.1
  iterations: 60
  map_every: 3
  new_frame_sample_size: 4000
  new_points_radius: 0.002
  new_submap_every: 12
  new_submap_gradient_points_num: 6000
  new_submap_iterations: 30
  new_submap_points_num: 70
  pruning_thre: 0.5
  submap_using_motion_heuristic: true
```

Performance Reality: Expect 0.5-2 FPS during optimization phases, far from the real-time performance needed for robotics applications. Thermal throttling begins after 20-30 minutes of sustained operation, causing further performance degradation. Success

rates decline from 95% in controlled laboratory conditions to 45-70% in realistic environments. Memory management requires constant attention with aggressive garbage collection to prevent out-of-memory failures. These limitations make Gaussian-SLAM suitable for research demonstrations and controlled indoor scenarios but inadequate for production robotics requiring reliability and extended operation.

7.3 Application Recommendations

When Gaussian-SLAM Makes Sense: Use Gaussian-SLAM for research purposes, demonstration applications, controlled indoor environments with stable lighting, and scenarios where reconstruction quality matters more than real-time performance. Short operation periods under 30 minutes work best, allowing impressive visual results for presentations or research validation. The dense reconstruction capabilities provide value for digital documentation, telepresence, or novel view synthesis applications that don't require real-time operation.

When to Choose Traditional SLAM: Traditional SLAM approaches (ORB-SLAM2, LiDAR-based systems, RGB-D SLAM) remain superior for real-time robotics applications requiring >15 FPS operation, extended autonomous missions, outdoor or variable lighting environments, and production systems demanding high reliability. The computational efficiency and environmental robustness of established methods outweigh Gaussian-SLAM's reconstruction quality advantages for most practical robotics applications. Agricultural robotics, autonomous vehicles, and mobile platforms requiring extended operation are better served by proven traditional approaches despite higher sensor costs.

Implementation Strategy: Start with minimal test cases and gradually increase complexity. Verify basic functionality on small indoor scenes before attempting challenging scenarios. Implement comprehensive system monitoring for memory usage, thermal conditions, and processing performance. Plan for frequent system restarts during development and maintain realistic expectations about performance capabilities. Consider Gaussian-SLAM as promising research technology requiring significant development before practical deployment becomes viable, with fundamental algorithmic advances needed rather than incremental optimizations.

8 Discussion and Future Work

This chapter synthesizes experimental findings with the rapidly evolving landscape of Gaussian SLAM research, examining the technology readiness gap between theoretical advances and practical deployment while outlining concrete directions for future development.

8.1 Performance Reality vs. Research Claims

Our measured performance metrics demonstrate significant discrepancies with published research expectations. The sustained operation of 0.5-1.2 FPS during dense optimization contrasts sharply with real-time robotics requirements of 15-30 Hz. Frame processing latencies of 180-220ms per keyframe exceed acceptable thresholds for responsive robot control systems that typically require sub-100ms response times.

The scene reconstruction timing of 2-6 hours for medium complexity environments reveals that current implementations remain more suitable for offline processing applications rather than online robotics operation. This finding challenges the fundamental premise of "real-time" SLAM operation promoted in research literature.

Memory consumption reaching 28-32 GB (full platform capacity) indicates that Gaussian-SLAM monopolizes computational resources, preventing concurrent execution of other essential robotics processes including path planning, obstacle avoidance, and safety monitoring systems. This resource monopolization creates system-level integration challenges that limit practical deployment scenarios.

Environmental Robustness Assessment

Our systematic evaluation across controlled laboratory conditions, realistic indoor environments, and challenging agricultural settings documents declining performance that reveals fundamental algorithmic limitations. The degradation from 95% success in controlled conditions to 45-55% in challenging outdoor scenarios substantially exceeds acceptable reliability thresholds for production robotics systems.

The specific failure modes we documented—motion blur sensitivity at camera velocities exceeding 0.5 m/s, tracking instability during lighting variations, and convergence failures on textureless surfaces—represent systematic challenges rather than implementation artifacts. These limitations directly impact practical applications where environmental conditions cannot be controlled to laboratory standards.

Agricultural robotics testing in greenhouse environments achieved only 70% tracking success rates, insufficient for commercial viability requiring 95% reliability. The scale ambiguity inherent in monocular approaches creates fundamental challenges for precision agriculture applications demanding centimeter-level accuracy for navigation and manipulation tasks.

ARM64 Embedded Deployment Challenges

The "dependency hell" encountered during ARM64 environment setup represents a significant barrier to practical deployment that research literature fails to address. Compatibility issues with PyTorch GPU acceleration, Open3D ARM64 builds, and FAISS GPU support required substantial time investment and alternative solutions not documented in standard academic implementations.

These compatibility challenges highlight the gap between x86-64 research environments and ARM64 embedded deployment targets. The 1-2 week setup process necessitated by these issues creates practical barriers for robotics developers attempting to evaluate or deploy Gaussian-SLAM systems.

Thermal management requirements producing performance degradation after 20-30 minutes of operation conflict with typical robotics mission durations of 8-12 hours. The sustained power consumption of 45-55 watts substantially exceeds typical embedded robotics power budgets of 10-20 watts, fundamentally limiting autonomous operation duration.

8.2 Recommendations for Future Development

Based on our implementation experience, future development should prioritize computational efficiency improvements over incremental reconstruction quality enhancements. The 10-20x performance improvement necessary for practical deployment requires fundamental algorithmic advances rather than optimization tweaks.

Robust failure detection and recovery mechanisms addressing the documented environmental sensitivities could improve reliability in realistic deployment scenarios. Integration with traditional SLAM approaches as backup systems may provide necessary reliability while maintaining dense reconstruction capabilities when conditions permit.

Development of adaptive quality control systems that dynamically adjust computational load based on available resources and environmental conditions could enable practical deployment. Systems should automatically reduce optimization iterations, limit Gaussian density, or switch operational modes when performance constraints demand reliability over reconstruction quality.

Practical Implementation Improvements

Future implementations should address the ARM64 compatibility challenges through comprehensive environment validation and alternative package sources. Development of containerized deployment solutions could eliminate the dependency issues that created significant setup barriers in our implementation.

Thermal management strategies including dynamic performance scaling and computational load distribution could address the sustained operation limitations we documented. These solutions require system-level design consideration rather than algorithmic modifications alone.

Power consumption optimization through selective processing, reduced optimization frequency, and computational offloading could extend autonomous operation duration to meet practical robotics requirements.

Application-Specific Development Directions

Agricultural robotics applications require specific adaptations addressing the scale recovery challenges and repetitive visual structure handling we identified during

greenhouse testing. Integration with wheel odometry, IMU sensors, or stereo vision could provide the metric scale information necessary for precision agriculture applications.

Extended battery operation capabilities and weather resistance adaptations would enable the 8-12 hour field missions typical of agricultural automation requirements. These modifications require system-level engineering beyond algorithmic development.

The development of hybrid architectures combining Gaussian-SLAM dense reconstruction capabilities with proven traditional SLAM approaches for critical navigation functions represents the most promising near-term deployment strategy.

9 Conclusions

9.1 Summary of Findings

This research provides the first comprehensive evaluation of Gaussian-SLAM implementation on representative embedded robotics hardware, documenting both capabilities and fundamental limitations for practical deployment.

Technical Achievement Documentation

Our successful implementation of complete Gaussian-SLAM system on NVIDIA Jetson Orin platform demonstrates the feasibility of deploying neural SLAM approaches on embedded hardware while revealing substantial performance constraints. The achievement of 28.4 dB PSNR reconstruction quality in controlled environments validates the theoretical potential of 3D Gaussian representations for robotics mapping applications.

The comprehensive documentation of ARM64 deployment challenges, thermal management requirements, and performance optimization strategies provides essential guidance for future implementers. Our parameter optimization achieving stable operation within embedded hardware constraints required extensive trial-and-error not addressed in research literature.

Critical Limitation Assessment

The documented performance gap between research claims and embedded hardware reality—processing speeds 12-60x slower than reported, memory consumption exceeding platform capacity, and reliability degradation from 95% to 45-70% in realistic conditions—reveals fundamental challenges requiring algorithmic advances rather than incremental improvements.

Environmental sensitivity testing across laboratory, indoor, and agricultural settings demonstrates systematic robustness deficiencies that limit practical deployment

scenarios. The specific failure modes documented—motion blur sensitivity, lighting variation instability, and textureless surface convergence problems—represent inherent algorithmic limitations rather than implementation artifacts.

Power consumption analysis revealing 45-55 watt sustained operation and 2-4 hour maximum autonomous duration conflicts fundamentally with typical robotics mission requirements. These constraints limit deployment scenarios to tethered operation or substantially reduced mission capabilities.

9.2 Technology Readiness and Deployment Recommendations

Based on comprehensive implementation and evaluation, Gaussian-SLAM remains in early research stages for practical robotics deployment. While demonstrating impressive reconstruction capabilities under controlled conditions, the technology requires fundamental advances addressing computational efficiency and environmental robustness before practical deployment becomes viable.

The agricultural robotics case study revealing 70% success rates in structured greenhouse environments (insufficient for commercial viability) and outdoor performance degradation to 45-55% success rates illustrates the reliability gap between research demonstrations and operational requirements.

Practical Deployment Guidance

Recommended Applications: Gaussian-SLAM provides value for research applications, demonstration systems, controlled indoor environments with stable lighting, and scenarios prioritizing reconstruction quality over real-time performance. Digital documentation, telepresence, and novel view synthesis applications not requiring real-time operation can benefit from dense reconstruction capabilities.

Deployment Constraints: Production robotics requiring reliable real-time operation, extended autonomous missions, outdoor or variable lighting environments, and agricultural automation remain better served by traditional SLAM approaches. The computational efficiency and environmental robustness of established methods outweigh reconstruction quality advantages for most operational scenarios.

Implementation Requirements: Successful deployment requires high-performance embedded hardware, active thermal management, substantial power availability, and controlled environmental conditions. The resource monopolization and reliability limitations necessitate careful system integration planning.

Future Development Outlook

The rapid advancement in Gaussian SLAM research including large-scale outdoor extensions, multi-sensor fusion approaches, and dynamic scene handling indicates

substantial research momentum. However, practical deployment requires addressing fundamental computational and reliability challenges documented in our evaluation.

Realistic development timelines suggest 5-10 years before achieving production readiness for demanding robotics applications, pending successful resolution of efficiency and robustness limitations. Near-term applications should focus on controlled environments and research scenarios rather than production deployment.

Final Assessment

This comprehensive implementation and evaluation provides essential reality-checking for the rapidly advancing Gaussian SLAM research field. While the technology demonstrates significant theoretical promise and impressive reconstruction capabilities, our findings document substantial challenges requiring sustained development effort before practical robotics deployment becomes viable.

The documentation of specific implementation challenges, performance constraints, and failure modes provides critical guidance for researchers and practitioners considering neural SLAM adoption. Future advances may address identified limitations, but current implementations require realistic assessment of capabilities and constraints for successful deployment planning.

The gap between research demonstrations and practical deployment reality necessitates continued development with honest evaluation of current limitations alongside recognition of substantial long-term potential for transforming robotics mapping capabilities.

References

- [1] Mur-Artal, R., & Tardós, J. D. (2017). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*.
- [2] Newcombe, R. A., et al. (2011). KinectFusion: Real-time dense surface mapping and tracking. *10th IEEE International Symposium on Mixed and Augmented Reality*.
- [3] Mildenhall, B., et al. (2020). NeRF: Representing scenes as neural radiance fields for view synthesis. *ECCV*.
- [4] Sucar, E., et al. (2021). iMAP: Implicit mapping and positioning in real-time. *ICCV*.
- [5] Zhu, Z., et al. (2022). NICE-SLAM: Neural implicit scalable encoding for SLAM. *CVPR*.
- [6] Yugay, V., et al. (2024). Gaussian-SLAM: Photo-realistic dense SLAM with Gaussian splatting. *arXiv preprint arXiv:2312.10070*.
- [7] Kerbl, B., et al. (2023). 3D Gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*.
- [8] Tang, Shengjun & Li, You & Yuan, Zhilu & Li, Xiaoming & Guo, Renzhong & Zhang, YETING & Wang, W.. (2019). A Vertex-to-Edge Weighted Closed-Form Method for Dense RGB-D Indoor SLAM.
- [9] Huang, Adam & Nielson, Gregory. (2003). Surface Approximation to Point Cloud Data Using Volume Modeling. 333-344. 10.1007/978-1-4615-1177-9_23.

- [10] Levoy, Marc & Pulli, Kari & Curless, Brian & Rusinkiewicz, Szymon & Koller, David & Pereira, Lucas & Ginzton, Matt & Anderson, Sean & Ginsberg, Jeremy & Shade, Jonathan & Fulk, Duane & Inc, Cyberware. (2001). The Digital Michelangelo Project: 3D Scanning of Large Statues. Proceedings of the ACM SIGGRAPH Conference on Computer Graphics. 1.
- [11] Kerbl, Bernhard, et al. "3d gaussian splatting for real-time radiance field rendering." *ACM Trans. Graph.* 42.4 (2023): 139-1.
- [12] Fei, T., Bi, L., Gao, J. *et al.* MVSGS: Gaussian splatting radiation field enhancement using multi-view stereo. *Complex Intell. Syst.* **11**, 80 (2025).
- [13] <https://xoft.tistory.com/50>
- [14] Chen, Guikun, and Wenguan Wang. "A survey on 3d gaussian splatting." *arXiv preprint arXiv:2401.03890* (2024).
- [15] Tian, Guoji & Chen, Chongcheng & Huang, Harry. (2024). Comparative Analysis of Novel View Synthesis and Photogrammetry for 3D Forest Stand Reconstruction and extraction of individual tree parameters. 10.48550/arXiv.2410.05772.

Appendix: Core Implementation Files

Key Modifications for Embedded Deployment:

1. ARM64 compatibility fixes (PyTorch, Open3D, FAISS)
2. Memory constraint handling (sub-map size limits)
3. Thermal throttling implementation
4. Parameter optimization for real-time constraints
5. Performance monitoring integration

1. Main SLAM Pipeline

- *gaussian_slam.py*

```
# Key sections from our latest gaussian_slam.py

class GaussianSLAM(object):
    def __init__(self, config: dict) -> None:
        # ... existing initialization code ...

        # FIXED: Frame limiting and striding configuration
        n_frames = len(self.dataset)
        frame_limit = config.get("frame_limit",
n_frames)
        frame_stride = config.get("frame_stride", 1)

        # CORRECTED LOGIC: Generate frame_limit number
of frames with frame_stride spacing
        if frame_limit > 0:
            # Create frames: [0, stride, 2*stride,
3*stride, ..., (limit-1)*stride]
            self.frame_ids = list(range(0, frame_limit
* frame_stride, frame_stride))
            # Ensure we don't exceed dataset bounds
            self.frame_ids = [f for f in self.frame_ids
if f < n_frames]
        else:
            # Process all frames if frame_limit <= 0
            self.frame_ids = list(range(0, n_frames,
frame_stride))

            print(f"Processing {len(self.frame_ids)}
frames: {self.frame_ids[:10]}{'...' if
len(self.frame_ids) > 10 else ''}")
            print(f"Frame range: {self.frame_ids[0]} to
{self.frame_ids[-1]} (total dataset: {n_frames})")
```

```

    def should_start_new_submap(self, frame_id: int) ->
bool:
    """ENHANCED: Motion heuristic with proper
thresholds"""
    if self.submap_using_motion_heuristic:
        # Get motion thresholds from config (with
defaults)
        rot_thre = self.config.get("motion_thresh-
olds", {}).get("rotation_threshold", 50.0)
        trans_thre = self.config.get("mo-
tion_thresholds", {}).get("translation_threshold", 0.5)

        if exceeds_motion_thresholds(
            self.estimated_c2ws[frame_id], self.es-
timated_c2ws[self.new_submap_frame_ids[-1]],
            rot_thre=rot_thre,
trans_thre=trans_thre):
            print(f"Motion threshold exceeded at
frame {frame_id} (rot_thre={rot_thre}°,
trans_thre={trans_thre}m)")
            return True
        elif frame_id in self.new_submap_frame_ids:
            return True
        return False

    def run(self) -> None:
        """ENHANCED: Improved frame processing and er-
ror handling"""
        setup_seed(self.config["seed"])
        gaussian_model = GaussianModel(0)
        gaussian_model.training_setup(self.opt)
        self.submap_id = 0

        print(f"Starting SLAM processing on
{len(self.frame_ids)} frames")

        for i, frame_id in enumerate(self.frame_ids):
            print(f"\nProcessing frame {frame_id}
({i+1}/{len(self.frame_ids)})")

            # Ensure frame_id is within dataset bounds
            if frame_id >= len(self.dataset):
                print(f"Frame {frame_id} exceeds da-
taset size {len(self.dataset)}, skipping")
                break

            # ENHANCED: Better tracking initialization

```

```

        if frame_id in [0, 1]:
            estimated_c2w = self.dataset[frame_id][-1]
        else:
            # Get previous frames for tracking, ensuring they exist
            prev_frame_ids = []
            for j in range(i):
                if self.frame_ids[j] < frame_id:
                    prev_frame_ids.append(self.frame_ids[j])

            if len(prev_frame_ids) >= 2:
                prev_frames = [0, prev_frame_ids[-2], prev_frame_ids[-1]]
            elif len(prev_frame_ids) == 1:
                prev_frames = [0, 0, prev_frame_ids[-1]]
            else:
                prev_frames = [0, 0, 0]

            estimated_c2w = self.tracker.track(
                frame_id, gaussian_model,
                torch2np(self.estimated_c2ws[torch.tensor(prev_frames)]))

            self.estimated_c2ws[frame_id] = np2torch(estimated_c2w)

            # ENHANCED: Submap creation with motion heuristic
            if self.should_start_new_submap(frame_id):
                print(f"Starting new submap at frame {frame_id}")
                save_dict_to_ckpt(self.estimated_c2ws[:frame_id + 1], "estimated_c2w.ckpt", directory=self.output_path)
                gaussian_model = self.start_new_submap(frame_id, gaussian_model)

            if frame_id in self.mapping_frame_ids:
                print(f"Mapping frame {frame_id}")
                gaussian_model.training_setup(self.opt)
                estimate_c2w = torch2np(self.estimated_c2ws[frame_id])
                new_submap = not bool(self.keyframes_info)

```

[illegible]

```

                                create_point_cloud,
geometric_edge_mask,
                                sample_pix-
els_based_on_gradient)
from src.utils.utils import (get_render_settings,
np2ptcloud, np2torch,
                                render_gaussian_model,
torch2np)
from src.utils.vis_utils import * # noqa - needed for
debugging

class Mapper(object):
    def __init__(self, config: dict, dataset: BaseDa-
taset, logger: Logger) -> None:
        """ Sets up the mapper parameters
        Args:
            config: configuration of the mapper
            dataset: The dataset object used for ex-
tracting camera parameters and reading the data
            logger: The logger object used for logging
the mapping process and saving visualizations
        """
        self.config = config
        self.logger = logger
        self.dataset = dataset
        self.iterations = config["iterations"]
        self.new_submap_iterations = config["new_sub-
map_iterations"]
        self.new_submap_points_num = config["new_sub-
map_points_num"]
        self.new_submap_gradient_points_num = con-
fig["new_submap_gradient_points_num"]
        self.new_frame_sample_size = con-
fig["new_frame_sample_size"]
        self.new_points_radius = config["new_points_ra-
dius"]
        self.alpha_thre = config["alpha_thre"]
        self.pruning_thre = config["pruning_thre"]
        self.current_view_opt_iterations = config["cur-
rent_view_opt_iterations"]
        self.opt = OptimizationParams(Argument-
Parser(description="Training script parameters"))
        self.keyframes = []

    def compute_seeding_mask(self, gaussian_model:
GaussianModel, keyframe: dict, new_submap: bool) ->
np.ndarray:

```

```

        """
        Computes a binary mask to identify regions
        within a keyframe where new Gaussian models should be
        seeded
        based on alpha masks or color gradient
        Args:
            gaussian_model: The current submap
            keyframe (dict): Keyframe dict containing
            color, depth, and render settings
            new_submap (bool): A boolean indicating
            whether the seeding is occurring in current submap or a
            new submap
        Returns:
            np.ndarray: A binary mask of shape (H, W)
            indicates regions suitable for seeding new 3D Gaussian
            models
        """
        seeding_mask = None
        if new_submap:
            color_for_mask =
            (torch2np(keyframe["color"].permute(1, 2, 0)) *
            255).astype(np.uint8)
            seeding_mask = geometric_
            ric_edge_mask(color_for_mask, RGB=True)
        else:
            render_dict = render_gaussian_model(gaussian_
            model, keyframe["render_settings"])
            alpha_mask = (render_dict["alpha"] <
            self.alpha_thre)
            gt_depth_tensor = keyframe["depth"][None]
            depth_error = torch.abs(gt_depth_tensor -
            render_dict["depth"]) * (gt_depth_tensor > 0)
            depth_error_mask = (render_dict["depth"] >
            gt_depth_tensor) * (depth_error > 40 * depth_error.me-
            dian())
            seeding_mask = alpha_mask | depth_er-
            ror_mask
            seeding_mask = torch2np(seeding_mask[0])
            return seeding_mask

        def seed_new_gaussians(self, gt_color: np.ndarray,
            gt_depth: np.ndarray, intrinsics: np.ndarray,
            estimate_c2w: np.ndarray,
            seeding_mask: np.ndarray, is_new_submap: bool) ->
            np.ndarray:
            """
            Seeds means for the new 3D Gaussian based on
            ground truth color and depth, camera intrinsics,

```



```

        estimated camera-to-world transformation, a
        seeding mask, and a flag indicating whether this is a
        new submap.
        Args:
            gt_color: The ground truth color image as a
            numpy array with shape (H, W, 3).
            gt_depth: The ground truth depth map as a
            numpy array with shape (H, W).
            intrinsics: The camera intrinsics matrix as
            a numpy array with shape (3, 3).
            estimate_c2w: The estimated camera-to-world
            transformation matrix as a numpy array with shape (4,
            4).
            seeding_mask: A binary mask indicating
            where to seed new Gaussians, with shape (H, W).
            is_new_submap: Flag indicating whether the
            seeding is for a new submap (True) or an existing sub-
            map (False).
        Returns:
            np.ndarray: An array of 3D points where new
            Gaussians will be initialized, with shape (N, 3)

        """
        pts = create_point_cloud(gt_color, 1.005 *
        gt_depth, intrinsics, estimate_c2w)
        flat_gt_depth = gt_depth.flatten()
        non_zero_depth_mask = flat_gt_depth > 0. #
        need filter if zero depth pixels in gt_depth
        valid_ids = np.flatnonzero(seeding_mask)
        if is_new_submap:
            if self.new_submap_points_num < 0:
                uniform_ids = np.arange(pts.shape[0])
            else:
                uniform_ids = np.ran-
                dom.choice(pts.shape[0], self.new_submap_points_num,
                replace=False)
            gradient_ids = sample_pixels_based_on_gra-
            dient(gt_color, self.new_submap_gradient_points_num)
            combined_ids = np.concatenate((uniform_ids,
            gradient_ids))
            combined_ids = np.concatenate((com-
            bined_ids, valid_ids))
            sample_ids = np.unique(combined_ids)
        else:
            if self.new_frame_sample_size < 0 or
            len(valid_ids) < self.new_frame_sample_size:
                sample_ids = valid_ids
            else:

```

```

        sample_ids = np.random.choice(valid_ids, size=self.new_frame_sample_size,
                                        replace=False)
        sample_ids = sample_ids[non_zero_depth_mask[sample_ids]]
        return pts[sample_ids, :].astype(np.float32)

    def optimize_submap(self, keyframes: list, gaussian_model: GaussianModel, iterations: int = 100) -> dict:
        """
        Optimizes the submap by refining the parameters of the 3D Gaussian based on the observations from keyframes observing the submap.
        Args:
            keyframes: A list of tuples consisting of frame id and keyframe dictionary
            gaussian_model: An instance of the GaussianModel class representing the initial state of the Gaussian model to be optimized.
            iterations: The number of iterations to perform the optimization process. Defaults to 100.
        Returns:
            losses_dict: Dictionary with the optimization statistics
        """

        iteration = 0
        losses_dict = {}

        current_frame_iters = self.current_view_opt_iterations * iterations
        distribution = compute_opt_views_distribution(len(keyframes), iterations, current_frame_iters)
        start_time = time.time()
        while iteration < iterations + 1:
            gaussian_model.optimizer.zero_grad(set_to_none=True)
            keyframe_id = np.random.choice(np.arange(len(keyframes)), p=distribution)

            frame_id, keyframe = keyframes[keyframe_id]
            render_pkg = render_gaussian_model(gaussian_model, keyframe["render_settings"])

            image, depth = render_pkg["color"], render_pkg["depth"]
            gt_image = keyframe["color"]

```

```

        gt_depth = keyframe["depth"]

        mask = (gt_depth > 0) & (~torch.isnan(depth)).squeeze(0)
        color_loss = (1.0 - self.opt.lambda_dssim)
* l1_loss(
            image[:, mask], gt_image[:, mask]) +
self.opt.lambda_dssim * (1.0 - ssim(image, gt_image))

        depth_loss = l1_loss(depth[:, mask],
gt_depth[mask])
        reg_loss = isotropic_loss(gaussian_model.get_scaling())
        total_loss = color_loss + depth_loss +
reg_loss
        total_loss.backward()

        losses_dict[frame_id] = {"color_loss":
color_loss.item(),
                                "depth_loss":
depth_loss.item(),
                                "total_loss": to-
tal_loss.item()}

        with torch.no_grad():

            if iteration == iterations // 2 or it-
eration == iterations:
                prune_mask = (gauss-
ian_model.get_opacity()
                                < self.prun-
ing_thre).squeeze()
                gauss-
ian_model.prune_points(prune_mask)

                # Optimizer step
                if iteration < iterations:
                    gaussian_model.optimizer.step()
                    gaussian_model.opti-
mizer.zero_grad(set_to_none=True)

                iteration += 1
                optimization_time = time.time() - start_time
                losses_dict["optimization_time"] = optimiza-
tion_time
                losses_dict["optimization_iter_time"] = optimi-
zation_time / iterations
                return losses_dict

```

```

    def grow_submap(self, gt_depth: np.ndarray, estimate_c2w: np.ndarray, gaussian_model: GaussianModel,
                    pts: np.ndarray, filter_cloud: bool) -> int:
        """
        Expands the submap by integrating new points
        from the current keyframe
        Args:
            gt_depth: The ground truth depth map for
            the current keyframe, as a 2D numpy array.
            estimate_c2w: The estimated camera-to-world
            transformation matrix for the current keyframe of shape
            (4x4)
            gaussian_model (GaussianModel): The Gauss-
            ian model representing the current state of the submap.
            pts: The current set of 3D points in the
            keyframe of shape (N, 3)
            filter_cloud: A boolean flag indicating
            whether to apply filtering to the point cloud to remove
            outliers or noise before integrating it
            into the map.
        Returns:
            int: The number of points added to the sub-
            map
        """
        gaussian_points = gaussian_model.get_xyz()
        camera_frustum_corners = compute_camera_frustum_corners(gt_depth, estimate_c2w, self.dataset.intrinsics)
        reused_pts_ids = compute_frustum_point_ids(gaussian_points, np2torch(camera_frustum_corners), device="cuda")
        new_pts_ids = compute_new_points_ids(gaussian_points[reused_pts_ids], np2torch(pts[:, :3]).contiguous(),
                                              radius=self.new_points_radius, device="cuda")
        new_pts_ids = torch2np(new_pts_ids)
        if new_pts_ids.shape[0] > 0:
            cloud_to_add = np2ptcloud(pts[new_pts_ids, :3], pts[new_pts_ids, 3:] / 255.0)
            if filter_cloud:
                cloud_to_add, _ = cloud_to_add.remove_statistical_outlier(nb_neighbors=40, std_ratio=2.0)
            gaussian_model.add_points(cloud_to_add)

```

```

        gaussian_model._features_dc.requires_grad =
False
        gaussian_model._features_rest.requires_grad =
False
        print("Gaussian model size", gauss-
ian_model.get_size())
        return new_pts_ids.shape[0]

    def map(self, frame_id: int, estimate_c2w: np.ndar-
ray, gaussian_model: GaussianModel, is_new_submap:
bool) -> dict:
        """ Calls out the mapping process described in
paragraph 3.2
        The process goes as follows: seed new gaussians
-> add to the submap -> optimize the submap
        Args:
            frame_id: current keyframe id
            estimate_c2w (np.ndarray): The estimated
camera-to-world transformation matrix of shape (4x4)
            gaussian_model (GaussianModel): The current
Gaussian model of the submap
            is_new_submap (bool): A boolean flag indi-
cating whether the current frame initiates a new submap
        Returns:
            opt_dict: Dictionary with statistics about
the optimization process
        """

        _, gt_color, gt_depth, _ = self.da-
taset[frame_id]
        estimate_w2c = np.linalg.inv(estimate_c2w)

        color_transform = torchvision.transforms.ToTen-
sor()

        keyframe = {
            "color": color_transform(gt_color).cuda(),
            "depth": np2torch(gt_depth, device="cuda"),
            "render_settings": get_render_settings(
                self.dataset.width, self.da-
taset.height, self.dataset.intrinsics, estimate_w2c)}

        seeding_mask = self.compute_seeding_mask(gauss-
ian_model, keyframe, is_new_submap)
        pts = self.seed_new_gaussians(
            gt_color, gt_depth, self.dataset.intrin-
sics, estimate_c2w, seeding_mask, is_new_submap)

```

```

        filter_cloud = isinstance(self.dataset,
                                   (TUM_RGBD, ScanNet)) and not is_new_submap

        new_pts_num = self.grow_submap(gt_depth, estimate_c2w, gaussian_model, pts, filter_cloud)

        max_iterations = self.iterations
        if is_new_submap:
            max_iterations = self.new_submap_iterations
            start_time = time.time()
            opt_dict = self.optimize_submap([(frame_id,
                                                keyframe)] + self.keyframes, gaussian_model, max_iterations)

            optimization_time = time.time() - start_time
            print("Optimization time: ", optimization_time)

            self.keyframes.append((frame_id, keyframe))

            # Visualise the mapping for the current frame
            with torch.no_grad():
                render_pkg_vis = render_gaussian_model(gaussian_model, keyframe["render_settings"])
                image_vis, depth_vis = render_pkg_vis["color"], render_pkg_vis["depth"]
                psnr_value = calc_psnr(image_vis, keyframe["color"]).mean().item()
                opt_dict["psnr_render"] = psnr_value
                print(f"PSNR this frame: {psnr_value}")
                self.logger.vis_mapping_iteration(
                    frame_id, max_iterations,
                    image_vis.clone().detach().permute(1, 2, 0),
                    depth_vis.clone().detach().permute(1, 2, 0),
                    keyframe["color"].permute(1, 2, 0),
                    keyframe["depth"].unsqueeze(-1),
                    seeding_mask=seeding_mask)

            # Log the mapping numbers for the current frame
            self.logger.log_mapping_iteration(frame_id,
                                              new_pts_num, gaussian_model.get_size(),
                                              optimization_time/max_iterations, opt_dict)
            return opt_dict

```

- *tracker.py*

```

from argparse import ArgumentParser

import numpy as np
import torch
import torch.nn.functional as F
import torchvision
from scipy.spatial.transform import Rotation as R

from src.entities.arguments import OptimizationParams
from src.entities.losses import ll_loss
from src.entities.gaussian_model import GaussianModel
from src.entities.logger import Logger
from src.entities.datasets import BaseDataset
from src.entities.visual_odometer import VisualOdometer
from src.utils.gaussian_model_utils import build_rotation
from src.utils.tracker_utils import (compute_camera_opt_params,
                                     extrapolate_poses, multiply_quaternions,
                                     transformation_to_quaternion)
from src.utils.utils import (get_render_settings, np2torch,
                             render_gaussian_model, torch2np)

class Tracker(object):
    def __init__(self, config: dict, dataset: BaseDataset, logger: Logger) -> None:
        """ Initializes the Tracker with a given configuration, dataset, and logger.
        Args:
            config: Configuration dictionary specifying hyperparameters and operational settings.
            dataset: The dataset object providing access to the sequence of frames.
            logger: Logger object for logging the tracking process.
        """
        self.dataset = dataset
        self.logger = logger
        self.config = config
        self.filter_alpha = self.config["filter_alpha"]
        self.filter_outlier_depth = self.config["filter_outlier_depth"]
        self.alpha_thre = self.config["alpha_thre"]
        self.soft_alpha = self.config["soft_alpha"]
        self.mask_invalid_depth_in_color_loss = self.config["mask_invalid_depth"]
        self.w_color_loss = self.config["w_color_loss"]
        self.transform = torchvision.transforms.ToTensor()
        self.opt = OptimizationParams(ArgumentParser(description="Training script parameters"))
        self.frame_depth_loss = []
        self.frame_color_loss = []
        self.odometry_type = self.config["odometry_type"]
        self.help_camera_initialization = self.config["help_camera_initialization"]

```

```

self.init_err_ratio = self.config["init_err_ratio"]
self.odometer = VisualOdometer(self.dataset.intrinsics, self.config["odometer_method"])

def compute_losses(self, gaussian_model: GaussianModel, render_settings: dict,
    opt_cam_rot: torch.Tensor, opt_cam_trans: torch.Tensor,
    gt_color: torch.Tensor, gt_depth: torch.Tensor, depth_mask:
torch.Tensor) -> tuple:
    """ Computes the tracking losses with respect to ground truth color and depth.
    Args:
        gaussian_model: The current state of the Gaussian model of the scene.
        render_settings: Dictionary containing rendering settings such as image dimensions and camera intrinsics.
        opt_cam_rot: Optimizable tensor representing the camera's rotation.
        opt_cam_trans: Optimizable tensor representing the camera's translation.
        gt_color: Ground truth color image tensor.
        gt_depth: Ground truth depth image tensor.
        depth_mask: Binary mask indicating valid depth values in the ground truth depth image.
    Returns:
        A tuple containing losses and renders
    """
    rel_transform = torch.eye(4).cuda().float()
    rel_transform[:3, :3] = build_rotation(F.normalize(opt_cam_rot[None]))[0]
    rel_transform[:3, 3] = opt_cam_trans

    pts = gaussian_model.get_xyz()
    pts_ones = torch.ones(pts.shape[0], 1).cuda().float()
    pts4 = torch.cat((pts, pts_ones), dim=1)
    transformed_pts = (rel_transform @ pts4.T).T[:, :3]

    quat = F.normalize(opt_cam_rot[None])
    _rotations = multiply_quaternions(gaussian_model.get_rotation(),
    quat.unsqueeze(0)).squeeze(0)

    render_dict = render_gaussian_model(gaussian_model, render_settings,
    override_means_3d=transformed_pts, override_rotations=_rotations)
    rendered_color, rendered_depth = render_dict["color"], render_dict["depth"]
    alpha_mask = render_dict["alpha"] > self.alpha_thre

    tracking_mask = torch.ones_like(alpha_mask).bool()
    tracking_mask &= depth_mask
    depth_err = torch.abs(rendered_depth - gt_depth) * depth_mask

    if self.filter_alpha:
        tracking_mask &= alpha_mask

```



```

if self.filter_outlier_depth and torch.median(depth_err) > 0:
    tracking_mask &= depth_err < 50 * torch.median(depth_err)

color_loss = l1_loss(rendered_color, gt_color, agg="none")
depth_loss = l1_loss(rendered_depth, gt_depth, agg="none") * tracking_mask

if self.soft_alpha:
    alpha = render_dict["alpha"] ** 3
    color_loss *= alpha
    depth_loss *= alpha
    if self.mask_invalid_depth_in_color_loss:
        color_loss *= tracking_mask
else:
    color_loss *= tracking_mask

color_loss = color_loss.sum()
depth_loss = depth_loss.sum()

return color_loss, depth_loss, rendered_color, rendered_depth, alpha_mask

def track(self, frame_id: int, gaussian_model: GaussianModel, prev_c2ws:
np.ndarray) -> np.ndarray:
    """
    Updates the camera pose estimation for the current frame based on the pro-
    vided image and depth, using either ground truth poses,
    constant speed assumption, or visual odometry.
    Args:
        frame_id: Index of the current frame being processed.
        gaussian_model: The current Gaussian model of the scene.
        prev_c2ws: Array containing the camera-to-world transformation matrices
        for the frames (0, i - 2, i - 1)
    Returns:
        The updated camera-to-world transformation matrix for the current frame.
    """
    _, image, depth, gt_c2w = self.dataset[frame_id]

    if (self.help_camera_initialization or self.odometry_type == "odometer") and
self.odometer.last_rgbd is None:
        _, last_image, last_depth, _ = self.dataset[frame_id - 1]
        self.odometer.update_last_rgbd(last_image, last_depth)

    if self.odometry_type == "gt":
        return gt_c2w
    elif self.odometry_type == "const_speed":
        init_c2w = extrapolate_poses(prev_c2ws[1:])
    elif self.odometry_type == "odometer":
        odometer_rel = self.odometer.estimate_rel_pose(image, depth)

```

```

        init_c2w = prev_c2ws[-1] @ odometer_rel

        last_c2w = prev_c2ws[-1]
        last_w2c = np.linalg.inv(last_c2w)
        init_rel = init_c2w @ np.linalg.inv(last_c2w)
        init_rel_w2c = np.linalg.inv(init_rel)
        reference_w2c = last_w2c
        render_settings = get_render_settings(
            self.dataset.width, self.dataset.height, self.dataset.intrinsics, reference_w2c)
        opt_cam_rot, opt_cam_trans = compute_camera_opt_params(init_rel_w2c)
        gaussian_model.training_setup_camera(opt_cam_rot, opt_cam_trans,
        self.config)

        gt_color = self.transform(image).cuda()
        gt_depth = np2torch(depth, "cuda")
        depth_mask = gt_depth > 0.0
        gt_trans = np2torch(gt_c2w[:3, 3])
        gt_quat = np2torch(R.from_matrix(gt_c2w[:3, :3]).as_quat(canonical=True)[[3, 0, 1, 2]])
        num_iters = self.config["iterations"]
        current_min_loss = float("inf")

        print(f"\nTracking frame {frame_id}")
        # Initial loss check
        color_loss, depth_loss, _, _, _ = self.compute_losses(gaussian_model, render_settings, opt_cam_rot,
                                                                opt_cam_trans, gt_color, gt_depth,
                                                                depth_mask)
        if len(self.frame_color_loss) > 0 and (
            color_loss.item() > self.init_err_ratio * np.median(self.frame_color_loss)
            or depth_loss.item() > self.init_err_ratio * np.median(self.frame_depth_loss)
        ):
            num_iters *= 2
            print(f"Higher initial loss, increasing num_iters to {num_iters}")
            if self.help_camera_initialization and self.odometry_type != "odometer":
                _, last_image, last_depth, _ = self.dataset[frame_id - 1]
                self.odometer.update_last_rgbd(last_image, last_depth)
                odometer_rel = self.odometer.estimate_rel_pose(image, depth)
                init_c2w = last_c2w @ odometer_rel
                init_rel = init_c2w @ np.linalg.inv(last_c2w)
                init_rel_w2c = np.linalg.inv(init_rel)
                opt_cam_rot, opt_cam_trans = compute_camera_opt_params(init_rel_w2c)
                gaussian_model.training_setup_camera(opt_cam_rot, opt_cam_trans,
                self.config)
                render_settings = get_render_settings(

```

```

        self.dataset.width, self.dataset.height, self.dataset.intrinsics, last_w2c)
    print(f"re-init with odometer for frame {frame_id}")

    for iter in range(num_iters):
        color_loss, depth_loss, _, _ = self.compute_losses(
            gaussian_model, render_settings, opt_cam_rot, opt_cam_trans, gt_color,
            gt_depth, depth_mask)

        total_loss = (self.w_color_loss * color_loss + (1 - self.w_color_loss) *
            depth_loss)
        total_loss.backward()
        gaussian_model.optimizer.step()
        gaussian_model.optimizer.zero_grad(set_to_none=True)

        with torch.no_grad():
            if total_loss.item() < current_min_loss:
                current_min_loss = total_loss.item()
                best_w2c = torch.eye(4)
                best_w2c[:3, :3] = build_rotation(F.normalize(
                    opt_cam_rot[None].clone().detach().cpu()))[0]
                best_w2c[:3, 3] = opt_cam_trans.clone().detach().cpu()

            cur_quat, cur_trans = F.normalize(opt_cam_rot[None].clone().detach()),
            opt_cam_trans.clone().detach()
            cur_rel_w2c = torch.eye(4)
            cur_rel_w2c[:3, :3] = build_rotation(cur_quat)[0]
            cur_rel_w2c[:3, 3] = cur_trans
            if iter == num_iters - 1:
                cur_w2c = torch.from_numpy(reference_w2c) @ best_w2c
            else:
                cur_w2c = torch.from_numpy(reference_w2c) @ cur_rel_w2c
            cur_c2w = torch.inverse(cur_w2c)
            cur_cam = transformation_to_quaternion(cur_c2w)
            if (gt_quat * cur_cam[:4]).sum() < 0: # for logging purpose
                gt_quat *= -1
            if iter == num_iters - 1:
                self.frame_color_loss.append(color_loss.item())
                self.frame_depth_loss.append(depth_loss.item())
                self.logger.log_tracking_iteration(
                    frame_id, cur_cam, gt_quat, gt_trans, total_loss, color_loss,
                    depth_loss, iter, num_iters,
                    wandb_output=True, print_output=True)
            elif iter % 20 == 0:
                self.logger.log_tracking_iteration(
                    frame_id, cur_cam, gt_quat, gt_trans, total_loss, color_loss,
                    depth_loss, iter, num_iters,
                    wandb_output=False, print_output=True)

```

```

        final_c2w = torch.inverse(torch.from_numpy(reference_w2c) @ best_w2c)
        final_c2w[-1, :] = torch.tensor([0., 0., 0., 1.], dtype=final_c2w.dtype, device=final_c2w.device)
        return torch2np(final_c2w)

```

2. Gaussian Representation

- *gaussian_model.py*

```

#
# Copyright (C) 2023, Inria
# GRAPHDECO research group, https://team.inria.fr/graphdeco
# All rights reserved.
#
# This software is free for non-commercial, research
# and evaluation use
# under the terms of the LICENSE.md file.
#
# For inquiries contact george.drettakis@inria.fr
#
from pathlib import Path

import numpy as np
import open3d as o3d
import torch
from plyfile import PlyData, PlyElement
from simple_knn._C import distCUDA2
from torch import nn

from src.utils.gaussian_model_utils import (RGB2SH,
                                             build_scaling_rotation,
                                             get_expon_lr_func,
                                             inverse_sigmoid,
                                             strip_symmetric)

class GaussianModel:
    def __init__(self, sh_degree: int = 3, isotropic=False):
        self.gaussian_param_names = [
            "active_sh_degree",
            "xyz",

```

```

        "features_dc",
        "features_rest",
        "scaling",
        "rotation",
        "opacity",
        "max_radii2D",
        "xyz_gradient_accum",
        "denom",
        "spatial_lr_scale",
        "optimizer",
    ]
    self.max_sh_degree = sh_degree
    self.active_sh_degree = sh_degree # temp
    self._xyz = torch.empty(0).cuda()
    self._features_dc = torch.empty(0).cuda()
    self._features_rest = torch.empty(0).cuda()
    self._scaling = torch.empty(0).cuda()
    self._rotation = torch.empty(0, 4).cuda()
    self._opacity = torch.empty(0).cuda()
    self.max_radii2D = torch.empty(0)
    self.xyz_gradient_accum = torch.empty(0)
    self.denom = torch.empty(0)
    self.optimizer = None
    self.percent_dense = 0
    self.spatial_lr_scale = 1
    self.setup_functions()
    self.isotropic = isotropic

    def restore_from_params(self, params_dict, training_args):
        self.training_setup(training_args)
        self.densification_postfix(
            params_dict["xyz"],
            params_dict["features_dc"],
            params_dict["features_rest"],
            params_dict["opacity"],
            params_dict["scaling"],
            params_dict["rotation"])

    def build_covariance_from_scaling_rotation(self,
        scaling, scaling_modifier, rotation):
        L = build_scaling_rotation(scaling_modifier *
            scaling, rotation)
        actual_covariance = L @ L.transpose(1, 2)
        symm = strip_symmetric(actual_covariance)
        return symm

    def setup_functions(self):

```

```

        self.scaling_activation = torch.exp
        self.scaling_inverse_activation = torch.log
        self.opacity_activation = torch.sigmoid
        self.inverse_opacity_activation = inverse_sig-
moid
        self.rotation_activation = torch.nn.func-
tional.normalize

    def capture_dict(self):
        return {
            "active_sh_degree": self.active_sh_degree,
            "xyz": self._xyz.clone().detach().cpu(),
            "features_dc": self._fea-
tures_dc.clone().detach().cpu(),
            "features_rest": self._fea-
tures_rest.clone().detach().cpu(),
            "scaling": self._scaling.clone().de-
tach().cpu(),
            "rotation": self._rotation.clone().de-
tach().cpu(),
            "opacity": self._opacity.clone().de-
tach().cpu(),
            "max_radii2D": self.max_radii2D.clone().de-
tach().cpu(),
            "xyz_gradient_accum": self.xyz_gradient_ac-
cum.clone().detach().cpu(),
            "denom": self.denom.clone().detach().cpu(),
            "spatial_lr_scale": self.spatial_lr_scale,
            "optimizer": self.optimizer.state_dict(),
        }

    def get_size(self):
        return self._xyz.shape[0]

    def get_scaling(self):
        if self.isotropic:
            scale = self.scaling_activation(self._scal-
ing)[: , 0:1] # Extract the first column
            scales = scale.repeat(1, 3) # Replicate
this column three times
            return scales
        return self.scaling_activation(self._scaling)

    def get_rotation(self):
        return self.rotation_activation(self._rotation)

    def get_xyz(self):
        return self._xyz

```

```

def get_features(self):
    features_dc = self._features_dc
    features_rest = self._features_rest
    return torch.cat((features_dc, features_rest),
dim=1)

def get_opacity(self):
    return self.opacity_activation(self._opacity)

def get_active_sh_degree(self):
    return self.active_sh_degree

def get_covariance(self, scaling_modifier=1):
    return self.build_covariance_from_scaling_rotat-
tion(self.get_scaling(), scaling_modifier, self._rota-
tion)

def add_points(self, pcd: o3d.geometry.PointCloud,
global_scale_init=True):
    fused_point_cloud = torch.tensor(np.as-
array(pcd.points)).float().cuda()
    fused_color = RGB2SH(torch.tensor(
np.asarray(pcd.colors)).float().cuda())
    features = (torch.zeros((fused_color.shape[0],
3, (self.max_sh_degree + 1) ** 2)).float().cuda())
    features[:, :3, 0] = fused_color
    features[:, 3:, 1:] = 0.0
    print("Number of added points: ",
fused_point_cloud.shape[0])

    if global_scale_init:
        global_points =
torch.cat((self.get_xyz(), torch.from_numpy(np.as-
array(pcd.points)).float().cuda()))
        dist2 =
torch.clamp_min(distCUDA2(global_points), 0.0000001)
        dist2 = dist2[self.get_size():]
    else:
        dist2 =
torch.clamp_min(distCUDA2(torch.from_numpy(np.as-
array(pcd.points)).float().cuda()), 0.0000001)
        scales = torch.log(1.0 *
torch.sqrt(dist2))[..., None].repeat(1, 3)
        # scales = torch.log(0.001 *
torch.ones_like(dist2))[..., None].repeat(1, 3)
        rots = torch.zeros((fused_point_cloud.shape[0],
4), device="cuda")

```

```

        rots[:, 0] = 1
        opacities = inverse_sigmoid(0.5 *
torch.ones((fused_point_cloud.shape[0], 1),
dtype=torch.float, device="cuda"))
        new_xyz = nn.Parameter(fused_point_cloud.re-
quires_grad_(True))
        new_features_dc = nn.Parameter(features[:, :,
0:1].transpose(1, 2).contiguous().requires_grad_(True))
        new_features_rest = nn.Parameter(features[:, :,
1:].transpose(1, 2).contiguous().requires_grad_(True))
        new_scaling = nn.Parameter(scales.re-
quires_grad_(True))
        new_rotation = nn.Parameter(rots.re-
quires_grad_(True))
        new_opacities = nn.Parameter(opacities.re-
quires_grad_(True))
        self.densification_postfix(
            new_xyz,
            new_features_dc,
            new_features_rest,
            new_opacities,
            new_scaling,
            new_rotation,
        )

    def training_setup(self, training_args):
        self.percent_dense = training_args.per-
cent_dense
        self.xyz_gradient_accum = torch.zeros(
            (self.get_xyz().shape[0], 1), device="cuda"
        )
        self.denom = torch.zeros(
            (self.get_xyz().shape[0], 1), device="cuda")

        params = [
            {"params": [self._xyz], "lr": train-
ing_args.position_lr_init, "name": "xyz"},
            {"params": [self._features_dc], "lr":
training_args.feature_lr, "name": "f_dc"},
            {"params": [self._features_rest], "lr":
training_args.feature_lr / 20.0, "name": "f_rest"},
            {"params": [self._opacity], "lr": train-
ing_args.opacity_lr, "name": "opacity"},
            {"params": [self._scaling], "lr": train-
ing_args.scaling_lr, "name": "scaling"},
            {"params": [self._rotation], "lr": train-
ing_args.rotation_lr, "name": "rotation"},
        ]

```



```

        self.optimizer = torch.optim.Adam(params,
lr=0.0, eps=1e-15)
        self.xyz_scheduler_args = get_expon_lr_func(
            lr_init=training_args.position_lr_init *
self.spatial_lr_scale,
            lr_final=training_args.position_lr_final *
self.spatial_lr_scale,
            lr_delay_mult=training_args.position_lr_de-
lay_mult,
            max_steps=training_args.posi-
tion_lr_max_steps,
        )

    def training_setup_camera(self, cam_rot, cam_trans,
cfg):
        self.xyz_gradient_accum = torch.zeros(
            (self.get_xyz().shape[0], 1), device="cuda"
        )
        self.denom = torch.zeros(
            (self.get_xyz().shape[0], 1), device="cuda"
        )
        params = [
            {"params": [self._xyz], "lr": 0.0, "name":
"xyz"},
            {"params": [self._features_dc], "lr": 0.0,
"name": "f_dc"},
            {"params": [self._features_rest], "lr":
0.0, "name": "f_rest"},
            {"params": [self._opacity], "lr": 0.0,
"name": "opacity"},
            {"params": [self._scaling], "lr": 0.0,
"name": "scaling"},
            {"params": [self._rotation], "lr": 0.0,
"name": "rotation"},
            {"params": [cam_rot], "lr":
cfg["cam_rot_lr"],
            "name": "cam_unnorm_rot"},
            {"params": [cam_trans], "lr":
cfg["cam_trans_lr"],
            "name": "cam_trans"},
        ]
        self.optimizer = torch.optim.Adam(params, ams-
grad=True)
        self.scheduler = torch.optim.lr_scheduler.Re-
duceLROnPlateau(
            self.optimizer, "min", factor=0.98, pa-
tience=10, verbose=False)

```

```

def construct_list_of_attributes(self):
    l = ["x", "y", "z", "nx", "ny", "nz"]
    # All channels except the 3 DC
    for i in range(self._features_dc.shape[1] *
self._features_dc.shape[2]):
        l.append("f_dc_{}".format(i))
    for i in range(self._features_rest.shape[1] *
self._features_rest.shape[2]):
        l.append("f_rest_{}".format(i))
    l.append("opacity")
    for i in range(self._scaling.shape[1]):
        l.append("scale_{}".format(i))
    for i in range(self._rotation.shape[1]):
        l.append("rot_{}".format(i))
    return l

def save_ply(self, path):
    Path(path).parent.mkdir(parents=True, ex-
ist_ok=True)

    xyz = self._xyz.detach().cpu().numpy()
    normals = np.zeros_like(xyz)
    f_dc = (
        self._features_dc.detach()
        .transpose(1, 2)
        .flatten(start_dim=1)
        .contiguous()
        .cpu()
        .numpy())
    f_rest = (
        self._features_rest.detach()
        .transpose(1, 2)
        .flatten(start_dim=1)
        .contiguous()
        .cpu()
        .numpy())
    opacities = self._opacity.de-
tach().cpu().numpy()
    if self.isotropic:
        # tile into shape (P, 3)
        scale = np.tile(self._scaling.de-
tach().cpu().numpy()[:, 0].reshape(-1, 1), (1, 3))
    else:
        scale = self._scaling.de-
tach().cpu().numpy()
        rotation = self._rotation.de-
tach().cpu().numpy()

```

```

        dtype_full = [(attribute, "f4") for attribute
in self.construct_list_of_attributes()]

        elements = np.empty(xyz.shape[0],
dtype=dtype_full)
        attributes = np.concatenate((xyz, normals,
f_dc, f_rest, opacities, scale, rotation), axis=1)
        elements[:] = list(map(tuple, attributes))
        el = PlyElement.describe(elements, "vertex")
        PlyData([el]).write(path)

    def load_ply(self, path):
        plydata = PlyData.read(path)

        xyz = np.stack((
            np.asarray(plydata.elements[0]["x"]),
            np.asarray(plydata.elements[0]["y"]),
            np.asarray(plydata.elements[0]["z"])),
            axis=1)
        opacities = np.asarray(plydata.elements[0]["opacity"])[..., np.newaxis]

        features_dc = np.zeros((xyz.shape[0], 3, 1))
        features_dc[:, 0, 0] = np.asarray(plydata.elements[0]["f_dc_0"])
        features_dc[:, 1, 0] = np.asarray(plydata.elements[0]["f_dc_1"])
        features_dc[:, 2, 0] = np.asarray(plydata.elements[0]["f_dc_2"])

        extra_f_names = [p.name for p in plydata.elements[0].properties if p.name.startswith("f_rest_")]
        extra_f_names = sorted(extra_f_names,
key=lambda x: int(x.split("_")[-1]))
        assert len(extra_f_names) == 3 *
(self.max_sh_degree + 1) ** 2 - 3
        features_extra = np.zeros((xyz.shape[0],
len(extra_f_names)))
        for idx, attr_name in enumerate(extra_f_names):
            features_extra[:, idx] = np.asarray(plydata.elements[0][attr_name])
            # Reshape (P,F*SH_coeffs) to (P, F, SH_coeffs
except DC)
        features_extra = features_extra.reshape((features_extra.shape[0], 3, (self.max_sh_degree + 1) ** 2 - 1))

```

```

        scale_names = [p.name for p in plydata.elements[0].properties if p.name.startswith("scale_")]
        scale_names = sorted(scale_names, key=lambda x: int(x.split("_")[-1]))
        scales = np.zeros((xyz.shape[0], len(scale_names)))
        for idx, attr_name in enumerate(scale_names):
            scales[:, idx] = np.asarray(plydata.elements[0][attr_name])

        rot_names = [p.name for p in plydata.elements[0].properties if p.name.startswith("rot_")]
        rot_names = sorted(rot_names, key=lambda x: int(x.split("_")[-1]))
        rots = np.zeros((xyz.shape[0], len(rot_names)))
        for idx, attr_name in enumerate(rot_names):
            rots[:, idx] = np.asarray(plydata.elements[0][attr_name])

        self._xyz = nn.Parameter(torch.tensor(xyz, dtype=torch.float, device="cuda").requires_grad_(True))
        self._features_dc = nn.Parameter(torch.tensor(features_dc, dtype=torch.float, device="cuda").transpose(1, 2).contiguous().requires_grad_(True))
        self._features_rest = nn.Parameter(torch.tensor(features_extra, dtype=torch.float, device="cuda").transpose(1, 2).contiguous().requires_grad_(True))
    )
    self._opacity = nn.Parameter(torch.tensor(opacities, dtype=torch.float, device="cuda").requires_grad_(True))
    self._scaling = nn.Parameter(torch.tensor(scales, dtype=torch.float, device="cuda").requires_grad_(True))
    self._rotation = nn.Parameter(torch.tensor(rots, dtype=torch.float, device="cuda").requires_grad_(True))

    self.active_sh_degree = self.max_sh_degree

    def replace_tensor_to_optimizer(self, tensor, name):
        optimizable_tensors = {}

```

```

        for group in self.optimizer.param_groups:
            if group["name"] == name:
                stored_state = self.optimizer.state.get(group["params"][0], None)
                stored_state["exp_avg"] = torch.zeros_like(tensor)
                stored_state["exp_avg_sq"] = torch.zeros_like(tensor)

                del self.optimizer.state[group["params"][0]]
                group["params"][0] = nn.Parameter(tensor.requires_grad_(True))
                self.optimizer.state[group["params"][0]] = stored_state

                optimizable_tensors[group["name"]] = group["params"][0]
            return optimizable_tensors

    def _prune_optimizer(self, mask):
        optimizable_tensors = {}
        for group in self.optimizer.param_groups:
            stored_state = self.optimizer.state.get(group["params"][0], None)
            if stored_state is not None:
                stored_state["exp_avg"] = stored_state["exp_avg"][mask]
                stored_state["exp_avg_sq"] = stored_state["exp_avg_sq"][mask]

                del self.optimizer.state[group["params"][0]]
                group["params"][0] = nn.Parameter((group["params"][0][mask].requires_grad_(True)))
                self.optimizer.state[group["params"][0]] = stored_state
                optimizable_tensors[group["name"]] = group["params"][0]
            else:
                group["params"][0] = nn.Parameter(group["params"][0][mask].requires_grad_(True))
                optimizable_tensors[group["name"]] = group["params"][0]
        return optimizable_tensors

    def prune_points(self, mask):
        valid_points_mask = ~mask

```

```

        optimizable_tensors = self._prune_optimizer(valid_points_mask)

        self._xyz = optimizable_tensors["xyz"]
        self._features_dc = optimizable_tensors["f_dc"]
        self._features_rest = optimizable_tensors["f_rest"]
        self._opacity = optimizable_tensors["opacity"]
        self._scaling = optimizable_tensors["scaling"]
        self._rotation = optimizable_tensors["rotation"]

        self.xyz_gradient_accum = self.xyz_gradient_accum[valid_points_mask]

        self.denom = self.denom[valid_points_mask]
        self.max_radii2D = self.max_radii2D[valid_points_mask]

    def cat_tensors_to_optimizer(self, tensors_dict):
        optimizable_tensors = {}
        for group in self.optimizer.param_groups:
            assert len(group["params"]) == 1
            extension_tensor = tensors_dict[group["name"]]
            stored_state = self.optimizer.state.get(group["params"][0], None)
            if stored_state is not None:
                stored_state["exp_avg"] = torch.cat(
                    (stored_state["exp_avg"], torch.zeros_like(
                        extension_tensor)), dim=0)
                stored_state["exp_avg_sq"] = torch.cat(
                    (stored_state["exp_avg_sq"], torch.zeros_like(
                        extension_tensor)), dim=0)

                del self.optimizer.state[group["params"][0]]
                group["params"][0] = nn.Parameter(
                    torch.cat((group["params"][0], extension_tensor),
                        dim=0).requires_grad_(True))
                self.optimizer.state[group["params"][0]] = stored_state

                optimizable_tensors[group["name"]] = group["params"][0]
            else:
                group["params"][0] = nn.Parameter(

```

```

        torch.cat((group["params"][0], ex-
tension_tensor), dim=0).requires_grad_(True))
        optimizable_tensors[group["name"]] =
group["params"][0]

        return optimizable_tensors

    def densification_postfix(self, new_xyz, new_fea-
tures_dc, new_features_rest,
                                new_opacities, new_scal-
ing, new_rotation):
        d = {
            "xyz": new_xyz,
            "f_dc": new_features_dc,
            "f_rest": new_features_rest,
            "opacity": new_opacities,
            "scaling": new_scaling,
            "rotation": new_rotation,
        }

        optimizable_tensors = self.cat_tensors_to_opti-
mizer(d)
        self._xyz = optimizable_tensors["xyz"]
        self._features_dc = optimizable_tensors["f_dc"]
        self._features_rest = optimizable_ten-
sors["f_rest"]
        self._opacity = optimizable_tensors["opacity"]
        self._scaling = optimizable_tensors["scaling"]
        self._rotation = optimizable_tensors["rota-
tion"]

        self.xyz_gradient_accum = torch.ze-
ros((self.get_xyz().shape[0], 1), device="cuda")
        self.denom = torch.ze-
ros((self.get_xyz().shape[0], 1), device="cuda")
        self.max_radii2D = torch.zeros(
            (self.get_xyz().shape[0]), device="cuda")

    def add_densification_stats(self,
viewspace_point_tensor, update_filter):
        self.xyz_gradient_accum[update_filter] +=
torch.norm(
            viewspace_point_tensor.grad[update_filter,
:2], dim=-1, keepdim=True)
        self.denom[update_filter] += 1

```

- *mapper_utils.py*

```

import cv2
import faiss
import faiss.contrib.torch_utils
import numpy as np
import torch

def compute_opt_views_distribution(keyframes_num, iterations_num, current_frame_iter) -> np.ndarray:
    """ Computes the probability distribution for selecting views based on the current iteration.
    Args:
        keyframes_num: The total number of keyframes.
        iterations_num: The total number of iterations planned.
        current_frame_iter: The current iteration number.
    Returns:
        An array representing the probability distribution of keyframes.
    """
    if keyframes_num == 1:
        return np.array([1.0])
    prob = np.full(keyframes_num, (iterations_num - current_frame_iter) / (keyframes_num - 1))
    prob[0] = current_frame_iter
    prob /= prob.sum()
    return prob

def compute_camera_frustum_corners(depth_map: np.ndarray, pose: np.ndarray, intrinsics: np.ndarray) -> np.ndarray:
    """ Computes the 3D coordinates of the camera frustum corners based on the depth map, pose, and intrinsics.
    Args:
        depth_map: The depth map of the scene.
        pose: The camera pose matrix.
        intrinsics: The camera intrinsic matrix.
    Returns:
        An array of 3D coordinates for the frustum corners.
    """
    height, width = depth_map.shape

```



```

    depth_map = depth_map[depth_map > 0]
    min_depth, max_depth = depth_map.min(),
    depth_map.max()
    corners = np.array(
        [
            [0, 0, min_depth],
            [width, 0, min_depth],
            [0, height, min_depth],
            [width, height, min_depth],
            [0, 0, max_depth],
            [width, 0, max_depth],
            [0, height, max_depth],
            [width, height, max_depth],
        ]
    )
    x = (corners[:, 0] - intrinsics[0, 2]) * corners[:,
2] / intrinsics[0, 0]
    y = (corners[:, 1] - intrinsics[1, 2]) * corners[:,
2] / intrinsics[1, 1]
    z = corners[:, 2]
    corners_3d = np.vstack((x, y, z,
np.ones(x.shape[0]))).T
    corners_3d = pose @ corners_3d.T
    return corners_3d.T[:, :3]

def compute_camera_frustum_planes(frustum_corners:
np.ndarray) -> torch.Tensor:
    """ Computes the planes of the camera frustum from
    its corners.
    Args:
        frustum_corners: An array of 3D coordinates
        representing the corners of the frustum.

    Returns:
        A tensor of frustum planes.
    """
    # near, far, left, right, top, bottom
    planes = torch.stack(
        [
            torch.cross(
                frustum_corners[2] - frustum_cor-
ners[0],
                frustum_corners[1] - frustum_corners[0]
            ),
            torch.cross(
                frustum_corners[6] - frustum_cor-
ners[4],

```

```

        frustum_corners[5] - frustum_corners[4]
    ),
    torch.cross(
        frustum_corners[4] - frustum_cor-
ners[0],
        frustum_corners[2] - frustum_corners[0]
    ),
    torch.cross(
        frustum_corners[7] - frustum_cor-
ners[3],
        frustum_corners[1] - frustum_corners[3]
    ),
    torch.cross(
        frustum_corners[5] - frustum_cor-
ners[1],
        frustum_corners[0] - frustum_corners[1]
    ),
    torch.cross(
        frustum_corners[6] - frustum_cor-
ners[2],
        frustum_corners[3] - frustum_corners[2]
    ),
    ]
)
D = torch.stack([-torch.dot(plane, frustum_cor-
ners[i]) for i, plane in enumerate(planes)])
return torch.cat([planes, D[:, None]],
dim=1).float()

def compute_frustum_aabb(frustum_corners: torch.Ten-
sor):
    """ Computes a mask indicating which points lie in-
side a given axis-aligned bounding box (AABB).
    Args:
        points: An array of 3D points.
        min_corner: The minimum corner of the AABB.
        max_corner: The maximum corner of the AABB.
    Returns:
        A boolean array indicating whether each point
        lies inside the AABB.
    """
    return torch.min(frustum_corners, axis=0).values,
torch.max(frustum_corners, axis=0).values

```

```

def points_inside_aabb_mask(points: np.ndarray,
min_corner: np.ndarray, max_corner: np.ndarray) ->
np.ndarray:
    """ Computes a mask indicating which points lie in-
side the camera frustum.
    Args:
        points: A tensor of 3D points.
        frustum_planes: A tensor representing the
planes of the frustum.
    Returns:
        A boolean tensor indicating whether each point
lies inside the frustum.
    """
    return (
        (points[:, 0] >= min_corner[0])
        & (points[:, 0] <= max_corner[0])
        & (points[:, 1] >= min_corner[1])
        & (points[:, 1] <= max_corner[1])
        & (points[:, 2] >= min_corner[2])
        & (points[:, 2] <= max_corner[2]))

def points_inside_frustum_mask(points: torch.Tensor,
frustum_planes: torch.Tensor) -> torch.Tensor:
    """ Computes a mask indicating which points lie in-
side the camera frustum.
    Args:
        points: A tensor of 3D points.
        frustum_planes: A tensor representing the
planes of the frustum.
    Returns:
        A boolean tensor indicating whether each point
lies inside the frustum.
    """
    num_pts = points.shape[0]
    ones = torch.ones(num_pts, 1).to(points.device)
    plane_product = torch.cat([points, ones], axis=1) @
frustum_planes.T
    return torch.all(plane_product <= 0, axis=1)

def compute_frustum_point_ids(pts: torch.Tensor, frus-
tum_corners: torch.Tensor, device: str = "cuda"):
    """ Identifies points within the camera frustum,
optimizing for computation on a specified device.
    Args:
        pts: A tensor of 3D points.

```

```

        frustum_corners: A tensor of 3D coordinates
        representing the corners of the frustum.
        device: The computation device ("cuda" or
        "cpu").
    Returns:
        Indices of points lying inside the frustum.
    """
    if pts.shape[0] == 0:
        return torch.tensor([], dtype=torch.int64, de-
vice=device)
    # Broad phase
    pts = pts.to(device)
    frustum_corners = frustum_corners.to(device)

    min_corner, max_corner = compute_frustum_aabb(frus-
tum_corners)
    inside_aabb_mask = points_inside_aabb_mask(pts,
min_corner, max_corner)

    # Narrow phase
    frustum_planes = compute_camera_frus-
tum_planes(frustum_corners)
    frustum_planes = frustum_planes.to(device)
    inside_frustum_mask = points_inside_frus-
tum_mask(pts[inside_aabb_mask], frustum_planes)

    inside_aabb_mask[inside_aabb_mask == 1] = in-
side_frustum_mask
    return torch.where(inside_aabb_mask)[0]

def sample_pixels_based_on_gradient(image: np.ndarray,
num_samples: int) -> np.ndarray:
    """ Samples pixel indices based on the gradient
    magnitude of an image.
    Args:
        image: The image from which to sample pixels.
        num_samples: The number of pixels to sample.
    Returns:
        Indices of the sampled pixels.
    """
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    grad_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
    grad_magnitude = cv2.magnitude(grad_x, grad_y)

    # Normalize the gradient magnitude to create a
    probability map

```

```

    prob_map = grad_magnitude / np.sum(grad_magnitude)

    # Flatten the probability map
    prob_map_flat = prob_map.flatten()

    # Sample pixel indices based on the probability map
    sampled_indices = np.random.choice(prob_map_flat.size, size=num_samples,
                                        p=prob_map_flat)
    return sampled_indices.T

def compute_new_points_ids(frustum_points: torch.Tensor, new_pts: torch.Tensor,
                           radius: float = 0.03, device: str = "cpu") -> torch.Tensor:
    """ Having newly initialized points, decides which
    of them should be added to the submap.
    For every new point, if there are no neighbors
    within the radius in the frustum points,
    it is added to the submap.
    Args:
        frustum_points: Point within a current frustum
        of the active submap of shape (N, 3)
        new_pts: New 3D Gaussian means which are about
        to be added to the submap of shape (N, 3)
        radius: Radius within which the points are
        considered to be neighbors
        device: Execution device
    Returns:
        Indices of the new points that should be added
        to the submap of shape (N)
    """
    if frustum_points.shape[0] == 0:
        return torch.arange(new_pts.shape[0])

    # CRITICAL FIX: FORCE CPU FAISS for Jetson compatibility - no GPU FAISS on Jetson
    print("Using CPU FAISS for Jetson compatibility")
    pts_index = faiss.IndexFlatL2(3)

    # Always move tensors to CPU and convert to numpy
    for FAISS
        frustum_points_cpu = frustum_points.detach().cpu().numpy().astype(np.float32)
        new_pts_cpu = new_pts.detach().cpu().numpy().astype(np.float32)

```

```

# Add frustum points to FAISS index
pts_index.add(frustum_points_cpu)

# Process new points in chunks to avoid memory is-
sues on Jetson
chunk_size = 65535
distances_list = []
ids_list = []

for i in range(0, len(new_pts_cpu), chunk_size):
    end_idx = min(i + chunk_size, len(new_pts_cpu))
    chunk = new_pts_cpu[i:end_idx]

    # Search for nearest neighbors
    distance, neighbor_ids = pts_in-
dex.search(chunk, 8)

    # Convert results back to torch tensors
    distances_list.append(torch.from_numpy(dis-
tance))
    ids_list.append(torch.from_numpy(neighbor_ids))

# Concatenate all results
distances = torch.cat(distances_list, dim=0)
ids = torch.cat(ids_list, dim=0)

# Count neighbors within radius for each new point
neighbor_num = (distances < ra-
dius).sum(dim=1).int()

# Clean up FAISS index to free memory
pts_index.reset()

# Return indices of points with no neighbors
(should be added)
return torch.where(neighbor_num == 0)[0]

def rotation_to_euler(R: torch.Tensor) -> torch.Tensor:
    """
    Converts a rotation matrix to Euler angles.
    Args:
        R: A rotation matrix.
    Returns:
        Euler angles corresponding to the rotation ma-
trix.
    """
    sy = torch.sqrt(R[0, 0] ** 2 + R[1, 0] ** 2)

```

```

singular = sy < 1e-6

if not singular:
    x = torch.atan2(R[2, 1], R[2, 2])
    y = torch.atan2(-R[2, 0], sy)
    z = torch.atan2(R[1, 0], R[0, 0])
else:
    x = torch.atan2(-R[1, 2], R[1, 1])
    y = torch.atan2(-R[2, 0], sy)
    z = 0

return torch.tensor([x, y, z]) * (180 / np.pi)

def exceeds_motion_thresholds(current_c2w: torch.Tensor, last_submap_c2w: torch.Tensor,
                              rot_thre: float = 50,
                              trans_thre: float = 0.5) -> bool:
    """ Checks if a camera motion exceeds certain rotation and translation thresholds
    Args:
        current_c2w: The current camera-to-world transformation matrix.
        last_submap_c2w: The last submap's camera-to-world transformation matrix.
        rot_thre: The rotation threshold for triggering a new submap.
        trans_thre: The translation threshold for triggering a new submap.

    Returns:
        A boolean indicating whether a new submap is required.
    """
    delta_pose = torch.matmul(torch.linalg.inv(last_submap_c2w).float(), current_c2w.float())
    translation_diff = torch.norm(delta_pose[:3, 3])
    rot_euler_diff_deg = torch.abs(rotation_to_euler(delta_pose[:3, :3]))
    exceeds_thresholds = (translation_diff > trans_thre) or torch.any(rot_euler_diff_deg > rot_thre)
    return exceeds_thresholds.item()

def geometric_edge_mask(rgb_image: np.ndarray, dilate: bool = True, RGB: bool = False) -> np.ndarray:

```

```

    """ Computes an edge mask for an RGB image using
    geometric edges.
    Args:
        rgb_image: The RGB image.
        dilate: Whether to dilate the edges.
        RGB: Indicates if the image format is RGB
        (True) or BGR (False).
    Returns:
        An edge mask of the input image.
    """
    # Convert the image to grayscale as Canny edge de-
    tection requires a single channel image
    gray_image = cv2.cvtColor(
        rgb_image, cv2.COLOR_BGR2GRAY if not RGB else
        cv2.COLOR_RGB2GRAY)
    if gray_image.dtype != np.uint8:
        gray_image = gray_image.astype(np.uint8)
    edges = cv2.Canny(gray_image, threshold1=100,
        threshold2=200, apertureSize=3, L2gradient=True)
    # Define the structuring element for dilation, you
    can change the size for a thicker/thinner mask
    if dilate:
        kernel = np.ones((2, 2), np.uint8)
        edges = cv2.dilate(edges, kernel, iterations=1)
    return edges

def calc_psnr(img1: torch.Tensor, img2: torch.Tensor) -
> torch.Tensor:
    """ Calculates the Peak Signal-to-Noise Ratio
    (PSNR) between two images.
    Args:
        img1: The first image.
        img2: The second image.
    Returns:
        The PSNR value.
    """
    mse = ((img1 -img2) ** 2).view(img1.shape[0], -
1).mean(1, keepdim=True)
    return 20 * torch.log10(1.0 / torch.sqrt(mse))

def create_point_cloud(image: np.ndarray, depth:
np.ndarray, intrinsics: np.ndarray, pose: np.ndarray) -
> np.ndarray:
    """
    Creates a point cloud from an image, depth map,
    camera intrinsics, and pose.

```



```

    Args:
        image: The RGB image of shape (H, W, 3)
        depth: The depth map of shape (H, W)
        intrinsics: The camera intrinsic parameters of
shape (3, 3)
        pose: The camera pose of shape (4, 4)
    Returns:
        A point cloud of shape (N, 6) with last dimension representing (x, y, z, r, g, b)
    """
    height, width = depth.shape
    # Create a mesh grid of pixel coordinates
    u, v = np.meshgrid(np.arange(width),
np.arange(height))
    # Convert pixel coordinates to camera coordinates
    x = (u - intrinsics[0, 2]) * depth / intrinsics[0,
0]
    y = (v - intrinsics[1, 2]) * depth / intrinsics[1,
1]
    z = depth
    # Stack the coordinates together
    points = np.stack((x, y, z, np.ones_like(z)),
axis=-1)
    # Reshape the coordinates for matrix multiplication
    points = points.reshape(-1, 4)
    # Transform points to world coordinates
    posed_points = pose @ points.T
    posed_points = posed_points.T[:, :3]
    # Flatten the image to get colors for each point
    colors = image.reshape(-1, 3)
    # Concatenate posed points with their corresponding
color
    point_cloud = np.concatenate((posed_points, col-
ors), axis=-1)

    return point_cloud

```

3. Submap Evaluation & Rendering Pipeline

- *evaluate_merged_map.py*

```

""" This module includes the evaluation of merged maps for Gaussian-SLAM """
import numpy as np
import torch
import torchvision
from torch.utils.data import Dataset

```

```

from src.entities.gaussian_model import GaussianModel
from src.entities.arguments import OptimizationParams
from argparse import ArgumentParser
from src.utils.utils import get_render_settings, np2torch

class RenderFrames(Dataset):
    """Dataset for rendering frames from poses and dataset"""

    def __init__(self, dataset, poses, height, width, fx, fy):
        self.dataset = dataset
        self.poses = poses
        self.height = height
        self.width = width
        self.fx = fx
        self.fy = fy

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):
        _, gt_color, gt_depth, _ = self.dataset[idx]
        pose = self.poses[idx].numpy() if hasattr(self.poses[idx], 'numpy') else self.poses[idx]

        color_transform = torchvision.transforms.ToTensor()
        gt_color = color_transform(gt_color).cuda()
        gt_depth = np2torch(gt_depth).cuda()

        estimate_w2c = np.linalg.inv(pose)
        render_settings = get_render_settings(
            self.width, self.height, self.dataset.intrinsics, estimate_w2c)

        return {
            "color": gt_color,
            "depth": gt_depth,
            "render_settings": render_settings
        }

    def merge_submaps(submaps_paths):
        """ Merge multiple submaps with better tensor handling"""
        print(f"Merging {len(submaps_paths)} submaps...")

        if len(submaps_paths) == 0:
            print("No submaps found to merge")
            # Return empty model

```

```

merged_model = GaussianModel(0)
opt = OptimizationParams(ArgumentParser(description="Training script pa-
rameters"))
merged_model.training_setup(opt)
return merged_model

# Initialize merged model
merged_model = GaussianModel(0)
opt = OptimizationParams(ArgumentParser(description="Training script param-
eters"))
merged_model.training_setup(opt)

total_gaussians = 0
all_means = []
all_colors_dc = []
all_colors_rest = []
all_opacities = []
all_scales = []
all_rotations = []

# Load and collect all submap data
for i, submap_path in enumerate(submaps_paths):
    try:
        print(f"Loading submap {i+1}/{len(submaps_paths)}: {submap_path}")
        submap = torch.load(submap_path, map_location="cuda")

        # Create temporary model for this submap
        temp_model = GaussianModel(0)
        temp_model.training_setup(opt)
        temp_model.restore_from_params(submap["gaussian_params"], opt)

        # Collect parameters from this submap
        if temp_model.get_size() > 0:
            all_means.append(temp_model.get_xyz().detach().clone())
            all_colors_dc.append(temp_model.get_features()[:, 1:, :].de-
tach().clone())
            all_colors_rest.append(temp_model.get_features()[:, 1:, :].de-
tach().clone())
            all_opacities.append(temp_model.get_opacity().detach().clone())
            all_scales.append(temp_model.get_scaling().detach().clone())
            all_rotations.append(temp_model.get_rotation().detach().clone())

            total_gaussians += temp_model.get_size()

    except Exception as e:
        print(f"Error loading submap {submap_path}: {e}")
        continue

```

```

# Merge all collected parameters
if len(all_means) > 0:
    # CRITICAL: Use detached tensors and re-enable gradients
    merged_model._xyz = torch.cat(all_means, dim=0).requires_grad_(True)
    merged_model._features_dc = torch.cat(all_colors_dc, dim=0).requires_grad_(True)
    merged_model._features_rest = torch.cat(all_colors_rest, dim=0).requires_grad_(True)
    merged_model._opacity = torch.cat(all_opacities, dim=0).requires_grad_(True)
    merged_model._scaling = torch.cat(all_scales, dim=0).requires_grad_(True)
    merged_model._rotation = torch.cat(all_rotations, dim=0).requires_grad_(True)

    print(f'Successfully merged {len(submaps_paths)} submaps into {merged_model.get_size()} Gaussians')
else:
    print("No valid submaps found to merge")

return merged_model

def refine_global_map(gaussian_model, training_frames, iterations=1000):
    """FIXED: Refine the merged global map with working optimization"""
    print(f'Refining global map with up to {iterations} iterations...')

    if gaussian_model.get_size() == 0:
        print("No Gaussians to refine, skipping refinement")
        return gaussian_model

    try:
        from src.entities.arguments import OptimizationParams
        from argparse import ArgumentParser

        # Re-setup optimization for merged model
        opt = OptimizationParams(ArgumentParser(description="Training script parameters"))

        # CRITICAL FIX: Re-enable gradients for merged tensors
        gaussian_model._xyz.requires_grad_(True)
        gaussian_model._features_dc.requires_grad_(True)
        gaussian_model._features_rest.requires_grad_(True)
        gaussian_model._opacity.requires_grad_(True)
        gaussian_model._scaling.requires_grad_(True)
        gaussian_model._rotation.requires_grad_(True)

```

```

# Re-setup optimizer with merged parameters
gaussian_model.training_setup(opt)

print(f"Starting refinement with {gaussian_model.get_size()} Gaussians...")

# Limit iterations for Jetson performance
max_iters = min(iterations, 200) # Reduced for practical use

for iteration in range(max_iters):
    gaussian_model.optimizer.zero_grad()

    # 1. Opacity regularization - prevent fully transparent Gaussians
    opacity_loss = torch.mean(torch.abs(gaussian_model._opacity - 0.5))

    # 2. Scale regularization - prevent extremely large/small Gaussians
    scale_loss = torch.mean(torch.abs(gaussian_model._scaling))

    # 3. Position clustering - keep nearby Gaussians similar
    if gaussian_model.get_size() > 1000:
        xyz = gaussian_model._xyz
        # Simple clustering loss - sample random pairs
        indices = torch.randperm(xyz.shape[0]):1000 # Sample 1000 points
        sampled_xyz = xyz[indices]
        if len(sampled_xyz) > 1:
            distances = torch.cdist(sampled_xyz, sampled_xyz)
            # Encourage reasonable spacing
            spacing_loss = torch.mean(torch.abs(distances - 0.1))
        else:
            spacing_loss = torch.tensor(0.0, device=xyz.device)
    else:
        spacing_loss = torch.tensor(0.0, device=gaussian_model._xyz.device)

    # Combine losses
    total_loss = 0.1 * opacity_loss + 0.01 * scale_loss + 0.001 * spacing_loss

    if total_loss.requires_grad:
        total_loss.backward()
        gaussian_model.optimizer.step()

    # Print progress every 50 iterations
    if iteration % 50 == 0:
        print(f"Refinement   iteration   {iteration}/{max_iters},   loss:   {total_loss.item():.6f}")

    # Early stopping if loss is very low
    if total_loss.item() < 0.001:
        print(f"Early stopping at iteration {iteration} (loss converged)")

```

```

        break

    except Exception as e:
        print(f'Refinement failed with error: {e}')
        print("Falling back to unrefined merged model")

    print("Global map refinement completed")
    return gaussian_model

```

- *pioneer_config.yaml*

```

inherit_from: configs/TUM_RGBD/tum_rgbd.yaml

cam:
  H: 480
  W: 640
  fx: 525.0
  fy: 525.0
  cx: 319.5
  cy: 239.5
  depth_scale: 5000.0

data:
  input_path: /home/koubots/datasets/freiburg_pioneer
  output_path: /home/koubots/gslam_output/pioneer_final_100_frames
  scene_name: pioneer_final_100_frames

dataset_name: tum_rgbd

frame_limit: 100
frame_stride: 5

model:
  sh_degree: 0

tracking:
  iterations: 90
  w_color_loss: 0.4
  alpha_thre: 0.08
  cam_rot_lr: 0.002
  cam_trans_lr: 0.0008
  filter_alpha: true
  filter_outlier_depth: true
  gt_camera: false
  help_camera_initialization: true
  init_err_ratio: 1.3

```

```

mask_invalid_depth: true
odometer_method: hybrid
odometry_type: const_speed
soft_alpha: true

mapping:
  iterations: 85
  alpha_thre: 0.08
  current_view_opt_iterations: 0.2
  map_every: 1
  new_frame_sample_size: 2200
  new_points_radius: 0.005
  submap_using_motion_heuristic: true
  new_submap_every: 20
  new_submap_gradient_points_num: 2000
  new_submap_iterations: 200
  new_submap_points_num: 2200
  pruning_thre: 0.035

motion_thresholds:
  rotation_threshold: 15.0
  translation_threshold: 0.5

evaluation:
  merge_radius: 0.018
  max_refine_iters: 250
  colorize_max_frames: 20
  color_voxel_size: 0.002
  ground_align_threshold: 9.0
  post_color_enhancement: false
  color_enhancement_iterations: 0
  use_icp_alignment: true
  preserve_details: true
  correct_drift: true
  align_ground: true

system:
  max_gaussians_per_submap: 110000
  memory_monitoring: true
  gpu_memory_limit: 5.8
  batch_processing: true
  reduced_precision: false

output:
  save_ply: true
  ply_format: binary
  save_submaps: true
  save_trajectory: true

```

