SLAM from RGB Image Sequences based on 3D Gaussian Splatting

Andrej Tibensky

TU Delft

SLAM from RGB Image Sequences based on 3D Gaussian Splatting

by

Andrej Tibensky

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 15, 2025 at 11:30.

Student number:	5882133	
Project duration:	March 3, 2024 – August 15, 2025	
Thesis committee:	Prof. Michael Weinmann	TU Delft, supervisor
	Prof. Nergis Tömen,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Contents

1 Introduction

Simultaneous localization and mapping (SLAM) is a long-standing problem in computer vision that aims to produce a scene representation from sensor inputs while simultaneously localizing the sensor during its movement through the scene. SLAM systems have wide-spread use in autonomous systems, robotics, virtual reality (VR), and augmented reality (AR). Many SLAM methods rely on depth data from a depth sensor, which facilitates camera tracking and scene mapping in comparison to purely visual SLAM methods. However, high-quality depth sensors still provide an increased cost and physical weight of the hardware used for capture. In contrast, purely visual SLAM only relies on the availability of a camera, thereby offering a lower cost approach with additional applications. Therefore, in the scope of the thesis, we have decided to focus on the more difficult but more accessible monocular RGB setting.

Traditional visual SLAM methods [17, 11, 33, 42, 43, 5] rely on the availability of correspondences between RGB monocular images for camera tracking and sparse mapping using sparse point clouds. However, such sparse point-based methods can not produce high-quality renderings of the scene. In order to improve the rendering quality, later methods introduced new scene representations, based on surfels [19, 30, 82, 62], signed distance functions [45], and optimized depth maps [73]. Although these new representations improved visual synthesis quality, they still remained sparse and could not produce photorealistic renderings.

To introduce a dense scene representation and improve the rendering quality even further, later methods based their scene representation on Neural Radiance Fields (NeRF) [39]. NeRF-based methods [70, 81, 55] approach dense mapping by fitting a differentiable neural scene representation to the observations. This is achieved through a multilayer perceptron (MLP) that predicts density and color at a given coordinate in the scene. This scene is then sampled along camera rays to render an image. Although this denser representation improved view synthesis quality, it was still far from photorealistic. Additionally, the use of an MLP significantly slowed down view synthesis, increased memory usage, and introduced catastrophic forgetting in larger scenes. Subsequent methods [90, 91, 85, 24, 58] attempted to improve the view synthesis speed, rendering quality, and performance in larger scenes by introducing additional explicit components, such as voxels, quad trees, or point clouds. However, these methods still struggled with photorealistic rendering quality, memory usage, and real-time performance.

To improve rendering quality and view synthesis speed, recent methods base their scene representation on 3D Gaussian Splatting (3DGS) [32], which represents the scene using 3D Gaussians that can be efficiently projected to the screen to render an image. 3DGS-based SLAM methods [22, 38, 86, 16] provide photorealistic rendering quality with fast view synthesis. However, these methods still struggle with tracking without depth priors and large memory overhead.

Without the availability of depth measurements, monocular RGB SLAM faces significant challenges in both tracking and mapping. For tracking, the availability of accurate depth measurements can be used for both camera pose initialization, through iterative closest point algorithms [16], and camera pose optimization based on a loss term between rendered depth and the depth measurements. This information results in significantly faster optimization and correct scale of the scene when compared to the RGB setting. As for mapping, the depth priors are usually converted to point clouds that are used

for map initialization and densification. Depth losses can also be used for optimizing the positions of scene elements once the cameras are optimized.

Our approach aims to address the limitations of existing SLAM methods by combining the strengths of sparse and dense scene representations while mitigating their weaknesses. For this, we use a dense 3DGS-based mapper with a sparse point-based camera pose estimator. Unlike traditional sparse point-based methods, our method achieves photorealistic rendering through a dense 3D Gaussian-based scene representation. Furthermore, our explicit dense scene representation allows for faster training and rendering speed than NeRF-based methods while improving visual synthesis quality.

We enhance our dense scene representation with a sparse point-based representation that we use for camera pose estimation. Inspired by recent developments in Structure-from-Motion [79], our sparse scene is optimized based on multi-view image correspondences from a pre-trained multi-view point tracker. These point tracks are predicted simultaneously over a window of N frames, rather than the frame-to-frame flow used by traditional methods. This provides our method with higher-quality correspondences which results in better camera pose estimation and sparse mapping than 3DGS-based methods in the monocular RGB setting. This sparse map can then also be densified, using a monocular depth predictor, and used for densification of our dense scene to improve rendering quality compared to other 3DGS-based methods. Finally, the use of a light-weight point tracker along with a dense Gaussian scene representation results in lower GPU memory usage, allowing for applications with limited compute capability.

In this thesis, we will provide details of our dual SLAM method and show that it has comparable tracking and reconstruction performance to the latest SLAM methods, such as DROID-Splat [21] and Splat-SLAM [59], which were developed concurrently to our research. Both methods use a 3DGS-based dense mapper with a camera pose and per-pixel disparity predictor (DROID-SLAM and DSPO respectively) as opposed to an explicit point-based sparse scene representation for tracking. For our evaluation, we used the Replica dataset [68], a synthetic dataset with high-quality images, and the TUM dataset [69], a real-world dataset that has more difficult scenes. To test tracking performance, we measure the relative mean squared error (RMSE). To test mapping performance, we measure the peak signal-to-noise ratio (PSNR), the structural similarity index measure (SSIM), and the learned perceptual image patch similarity (LPIPS) and compare them with recent methods. Finally, we also report our GPU memory usage and frames per second.

We will show that our method achieves competitive tracking and view synthesis performance on the synthetic Replica dataset [68] when compared to state-of-the-art methods. We will also show that our method achieves comparable tracking and view synthesis performance on the real-world TUM dataset [69]. Finally, we will show that our method achieves state-of-the-art GPU memory usage compared to state-of-the-art dense mapping monocular SLAM methods while retaining comparable training speed.

In summary, our contributions are as follows:

- We present a near real-time monocular RGB SLAM system with a dual scene representation
- We present a novel camera pose estimation and refinement module with a sparse point cloud optimized using multi-view point correspondences from a neural multi-view point tracker
- We present a 3DGS-based scene enhanced with depth information from a neural depth predictor aligned to our optimized point cloud
- In the scope of extensive evaluations on synthetic and real-world datasets, we demonstrate comparable performance of our method despite its lower GPU memory usage thanks to our sparse point-based scene and light multi-view point tracker

2

Related Work

In this chapter, we will provide a brief overview of the major developments in simultaneous localization and mapping. For this purpose, we will structure the overview mostly chronologically by first discussing traditional explicit algorithmic methods in Section 2.1, followed by the evolution of NeRF-based methods in Section 2.2, and lastly discussing the most recent methods using 3D Gaussian Splatting in Section 2.3.

2.1. Traditional Visual SLAM Methods

The exploration of SLAM began in the field of robotics, with methods [65, 40] relying purely on sensor depth measurements. Due to the computational overhead of processing full RGB images, the field of computer vision has been mostly focused on Structure-from-Motion (SfM) algorithms that were fundamentally offline in nature. Respective methods [17, 75, 53] focused on extracting simple geometric features, such as edges and corners, which could then be used for global optimization of a sparse map and camera poses. These developments laid the groundwork for early visual SLAM methods [11, 33, 42] that relied on sparse scene representations through improved image features and feature matching algorithms. However, these methods only provided a sparse scene representation in terms of sparse point clouds or line sets that were limited to capturing only some parts of the scene. Later methods built upon these concepts by introducing denser scene representations such as depth maps, surfels, truncated signed distance functions, and voxel grids. Furthermore, advances in deep learning led to later methods that incorporate neural components into their approaches. This section will present the details of selected traditional visual SLAM methods [7] before explaining modern dense differentiable methods in the later sections.

2.1.1. Sparse Feature-based SLAM

One of the first successful methods that managed to apply the SLAM methodology to the pure visual domain was MonoSLAM [11], which combined real-time SLAM from the robotics domain with advances in visual Structure-from-Motion. The system used an Extended Kalman Filter (EKF) [26] as its probabilistic framework to represent the camera poses as a 13-element vector comprising position, orientation, linear velocity, and angular velocity. The EKF predicted new camera poses using a constant-velocity model that was then optimized through a sparse map of 3D feature points. These sparse points were triangulated based on image features extracted using Shi-Tomasi corner detection [63]. These feature points were stored as the coordinates of the camera in which they were first observed along with their inverse depth from the camera, as traditional 3D point representation struggled with distant features. Although MonoSLAM demonstrated the feasibility of real-time visual SLAM, its reliance on simple features and an inefficient EKF framework limited its scalability to larger environments.

Next, Parallel Tracking and Mapping (PTAM) [33] introduced a significant advancement by separating the tracking and mapping into separate threads. This allowed for faster and more robust camera pose estimation while increasing the map accuracy. PTAM uses the FAST feature detector [56], which detects corners by examining pixel intensities in a circle around each candidate point. However, the simple FAST features without descriptors still limited the accuracy of the system.

To address robustness even further, ORB-SLAM [42, 43, 5] leveraged ORB (Oriented FAST and Rotated BRIEF) [57] features. ORB combined the FAST feature detector [56] with the BRIEF feature descriptor [3] that computes binary string feature descriptors by comparing pixel intensities at selected pairs of locations around a feature point. These binary string descriptors enable fast feature matching using Hamming distance calculation through simple XOR operations. ORB improves these descriptors by selecting pairs of locations that maximize variance and minimize correlation, as this makes the descriptors more resistant to rotation.

ORB-SLAM [42] took full advantage of these features for tracking, mapping, and loop closure in a three-thread architecture with sparse keyframe-based maps for global optimization. Later versions of ORB-SLAM [43, 5] added support for stereo and depth input, and multi-map capabilities for better handling of large scenes. Despite these improvements, the sparse features provided insufficient constraints for dense reconstruction and could only be used for sparse maps without high-quality visual synthesis.

2.1.2. Direct Frame-to-frame Methods

To solve the limitations of sparse methods and introduce a dense form of scene representation, some methods moved toward operating on every pixel of a frame rather than sparse extracted features. One of the first methods was DTAM (Dense Tracking and Mapping) [44], which represents the scene using detailed textured depth maps at selected keyframes. These depth maps are iteratively estimated by globally minimizing a photometric error calculated by projecting points into overlapping images based on camera poses and previously predicted depth. The method then iteratively refines camera poses by minimizing photometric error between the current projection of the dense scene and the original frame.

LSD-SLAM [14] extended the direct approach to work in large-scale environments while achieving faster optimization speed. The method uses a three-thread structure for camera tracking, depth map estimation, and map optimization, achieving robust pose estimation while constructing an accurate dense map. LSD-SLAM was able to address accuracy in larger scenes by introducing a novel direct tracking method that optimized scene scale on top of camera poses for explicit scale-drift detection. The camera poses are estimated by constructing a pose-graph of keyframes with associated semi-dense depth maps that focus on high-gradient regions. These semi-dense depth maps were included in tracking through a probabilistic approach to correct for noisy depth values, improving tracking accuracy and mapping in large scenes while achieving real-time performance.

2.1.3. Surfel-based SLAM

Parallel to direct methods, advances in consumer RGB-D cameras led to some approaches addressing the problem of dense reconstruction through surface elements (surfels). Surfels are points in 3D space with an associated radius, orientation, confidence, and color. These surfels can then be projected into camera space for image rendering, with low-confidence surfels being removed. This dense surface representation is particularly useful for the SLAM setting because of the efficiency of its rendering.

Surfels were first introduced by Henry et al. [20], to effectively combine RGB and depth information from consumer RGB-D cameras. The method achieves this by combining point clouds acquired from depth measurements with visual SIFT features and surfel parameters. These enhanced point clouds can then be used for camera pose alignment and loop closure through an Iterative Closest Point (ICP) algorithm. Keller et al. [30] expanded the original surfel representation by adding support for dynamic scenes. This is achieved by creating a model of high-confidence observations that are stable across multiple views. Dynamic objects can then be detected through outlier detection in the ICP algorithm.

ElasticFusion [82] aimed to improve performance in large scenes by introducing a continuously deformable scene. This is achieved by refining the scene through non-rigid deformations. These deformations are applied through nodes distributed in the scene based on photometric and geometric loss calculated during global loop closure. Each node has an associated affine transformation which is applied to surrounding surfels that have a similar timestamp.

BAD SLAM [62] took full advantage of the modern hardware computing performance to implement direct scene optimization through global bundle adjustment of all surfels and camera poses. This was achieved by calculating the geometric and photometric residuals between the renderings and the original keyframe images. The global bundle adjustment then attempted to minimize these residuals by first updating the surfel normals and then updating the surfel positions and descriptors. The camera parameters could then be optimized based on the refined scene through an additional global bundle

adjustment.

2.1.4. Dense Volumetric SLAM

Parallel to direct and surfel-based methods, some methods aimed to address the problem of dense mapping through volumetric scene representations. Unlike previous approaches, volumetric methods aimed to represent the full 3D scene rather than just focus on the object surfaces.

One of the first volumetric methods was KinectFusion [45], which used a truncated signed distance function (TSDF) [8] to represent the scene. Signed distance functions represent a surface implicitly, by storing distances to the surface at vertices in the scene. Vertices inside the surface have positive distances, those outside have negative distances, and vertices on the surface have a distance of 0. The vertex map which holds the TSDF values is initialized based on depth measurements, with normal vectors computed based on cross products with the neighboring vertices. Rendering is done by marching a ray for each pixel in the camera starting from minimum depth until a zero crossing is found, indicating a surface. Only vertices within a truncation band around the surface are updated, while vertices outside the band are removed. Camera poses are estimated using an ICP algorithm based on the incoming RGB-D frame and the rendered geometric and photometric information.

BundleFusion [10] expanded the original vertex-based TSDF scene representation with local and global optimization. This was achieved by extracting visual SIFT features that were then used for improved camera pose estimation and loop closure. The method groups frames into chunks of 11 frames and optimizes their poses within these chunks. Global optimization then aligns all chunks using representative keyframes. Both local and global optimization are then solved through an efficient GPU-based solver, achieving real-time performance while improving tracking accuracy.

Voxel Hashing [47] later expanded the TSDF scene representation by storing SDF values in a voxel grid of 256 blocks. A hash function maps 3D world coordinates to buckets of multiple entries to handle collisions while providing constant-time access. New buckets are initialized around the truncation band based on depth sample footprints. This allows for higher level of detail in more complex areas. Rendering is done by per-pixel ray marching with TSDF values interpolated from the neighboring points. Each voxel block is processed by a GPU multiprocessor with one thread per voxel.

InfiniTAM [25] aimed to improve the rendering efficiency of Voxel Hashing by introducing an optimized ray casting pipeline. This pipeline starts by projecting visible voxel blocks to establish minimum and maximum ray lengths for each pixel. The ray casting is then applied with step size decisions based on truncation band status and SDF values, with interpolation only applied near surfaces. This allows for a more efficient rendering suitable for mobile GPU architectures. InfiniTAMv3 [54] introduced a unified framework that supports loop closure with local and global bundle adjustment for both volumetric and surfel-based scene representations. Further extensions of Voxel Hashing focused on optimizing the data structure to avoid collisions during the interaction with the hash map [67] and its combination with an additional point-based representation for the dynamic parts of the scene [77].

2.1.5. Deep Learning SLAM

The advances in deep learning resulted in various approaches that use neural components within their frameworks. These consisted of neural camera pose prediction, depth map prediction, or both. Although not a full SLAM system, PoseNet [31] was the first demonstration that neural networks could solve pose estimation problems. Later, DeepVO [80] combined CNN feature extraction with LSTM temporal modeling for an end-to-end deep learning approach to camera pose estimation from RGB image sequences. The first major method to introduce neural components into SLAM was CNN-SLAM [72], which combined CNN-predicted depth maps with the traditional surfel-based LSD-SLAM [14]. This allowed for optimizing a dense scene representation with correct scene scale and accurate camera poses from monocular RGB input.

CodeSLAM [2] instead used the latent space of a Variational Auto-Encoder (VAE) for a compact learned depth map scene representation. The VAE is used to predict a 128-dimensional latent representation of the image. This code can then be used along with the image to predict a dense depth map with camera poses estimated using traditional optimization methods. This resulted in dramatically reduced dimensionality and real-time performance, due to only storing the camera pose and latent code for each frame, while retaining dense scene reconstruction from the runtime depth map decoder.

DeepFactors [9] later expanded this learned approach by introducing loop closure and global optimization of camera poses and latent codes by minimizing three error metrics. These are geometric

and photometric loss between the source frame and a target image warped into the frame, and reprojection error between observed and known feature locations using BRISK [35] features. DeepTAM [44] used neural networks for both scene depth map-based representation and camera pose estimation. An encoder-decoder model is pre-trained offline to predict dense optical flow and relative camera poses between two frames. The model can then be used at run-time to predict a camera pose update between a keyframe and the current frame. The camera pose update is then refined at multiple resolutions. Keyframe images are then accumulated in cost volumes and used along with the current keyframe to predict a depth map using pre-trained CNNs.

DROID-SLAM [73] instead used the predicted dense frame-to-frame flow, stored in a frame-graph, to optimize per-frame camera poses and inverse depth. The flow prediction is based on RAFT [74], extracting features and constructing a full correlation volume between the two frames. The correlation volume is then used to predict updates of the depth and camera pose. The flow predictions are then optimized through a global Dense Bundle Adjustment (DBA) layer that maps them to depths and calculates losses over the entire frame-graph to optimize the optical flows.

GO-SLAM [89] later extended DROID-SLAM by introducing loop closure and global bundle adjustment for more robust camera pose estimation. Loop closure is achieved by computing a co-visibility matrix between a recent window of keyframes and all historical keyframes to find new connections for the frame-graph. The global bundle adjustment runs in a separate thread to allow continuous optimization of camera poses and predicted depth maps of all historical keyframes. The connections between recent keyframes and historical keyframes then allow for global alignment through local bundle adjustment.

2.2. Neural Radiance Fields SLAM

Recently [84], visual SLAM methods based on Neural Radiance Fields (NeRF) [39] became popular. NeRF is a novel view synthesis approach representing the scene implicitly as a function through a Multilevel Perceptron (MLP) that maps a 3D location and 2D viewing direction to RGB color and volume density. NeRF renders an output image by querying the MLP along camera rays where the predicted volume density represents the probability of ray termination at that queried position. The predicted colors are multiplied by their corresponding density and integrated for all queried positions along the camera ray to get the color of the corresponding pixel in the output image. The MLP is then optimized through backpropagation based on rendering loss between the rendered image and the ground-truth image. The method uses a coarse network to produce query points along the rays that are used in the fine network, skipping empty spaces to improve performance. NeRF-based SLAM methods can be divided into fully implicit methods that only use an MLP and hybrid methods that use the MLP along with additional scene components to store information about the scene to be used in rendering.

2.2.1. Fully Implicit Methods

Fully implicit methods represent the scene entirely with the MLP as described in the original NeRF formulation without using any explicit scene components such as voxel grids or 3D points. iMAP [70] was the first SLAM method to use a simplified NeRF-based approach for dense mapping by removing the view direction and only considering the position of the query point. The method jointly optimizes camera poses and network parameters based on rendering and depth loss from ground-truth RGB-D images. Later, NeRF– [81] added optimization of the shared intrinsic camera parameters along with the per-frame camera poses. These shared intrinsic parameters were then used to calculate the viewing direction, taking advantage of the full NeRF mapping approach. NeRF-SLAM [55] added DROID-SLAM [73] to NeRF-based mapping to achieve better tracking and support for purely monocular RGB input, taking into account depth and pose uncertainties during the NeRF-based mapping. DROID-SLAM runs in a front-end thread to process new frames and continuously optimize an active window of keyframes through bundle adjustment. The NeRF-based mapper is run in a second thread to continuously optimize the implicit scene representation based on rendering and depth loss.

Although these first approaches proved that NeRF-based dense mapping can be used in the SLAM setting, they struggled with scalability due to catastrophic forgetting in larger scenes, which resulted in the initially explored parts of the scene degrading in quality.

2.2.2. Hybrid Implicit and Explicit Methods

In order to improve performance in larger scenes and on longer image sequences, some methods expanded the original NeRF-based scene representation by adding explicit components, such as voxel grids, feature planes, or 3D points. These components stored feature vectors that were passed to the MLP during rendering and were optimized alongside the MLP to provide better visual synthesis quality in larger scenes without impacting computational overhead.

NICE-SLAM [90], was one of the first methods to expand the implicit scene representation with feature grids that held geometry and color features. The method used the Convolutional Occupancy Network (ConvONet) framework [51], which processes 3D input, such as a point cloud, using a CNN to extract geometric features stored in a 3D grid structure. These features are then processed by an MLP to output occupancy values of 0 for empty space and 1 for occupied space. This approach was used to pre-train three MLP decoders on synthetic indoor scenes to predict occupancy. After training NICE-SLAM discards the CNN and instead optimizes the geometric feature grids directly at three levels of detail. This enabled faster tracking, even in regions that were previously unseen, by skipping regions with empty occupancy. The method also employs separate feature grids at 16 levels of detail for color representation that are used by the MLP during rendering. The geometry feature grids, color feature grids, and MLP are jointly optimized through local bundle adjustment based on rendering and depth losses. Camera poses are optimized separately based on geometric loss that down-weights regions with high depth variance. NICER-SLAM [91] later expanded the original method by removing pre-trained networks and direct density predictions for geometry encoding. The method instead uses an MLP to predict SDF values based on features stored in a 32^3 dense voxel grid. These SDF values are then transformed to density values based on a locally adaptive parameter. This allows for better handling of areas with fewer observations.

Before the improvements introduced by NICER-SLAM, Vox-Fusion [85] introduced a simpler octree-based voxel scene to hold embeddings. These features were then passed to an MLP to predict SDF values for traditional volumetric depth and color rendering. The ray-marching was expanded with an improved point sampling based on ray-voxel intersection tests to avoid wasting computation on empty space. The voxel scene and the MLP were then jointly optimized based on geometric and rendering losses, a free-space loss to enforce correct SDF values in free space between the camera and the surface, and an SDF loss to ensure an accurate surface representation within the truncation area around surfaces.

DIM-SLAM [36] was the first RGB-only dense neural implicit SLAM. The method used hierarchical feature volumes for scene representation rather than a voxel-based representation. Traditional volumetric rendering was done by passing features at query point locations to a two-headed MLP to predict the density values for geometric details and color values for rendering. Co-SLAM [78] later expanded this representation through a multi-resolution hash-based feature grid with separate MLP decoders to predict TSDF and RGB values. Unlike previous methods, Co-SLAM used joint coordinate and parametric encoding as inputs to the MLP decoders instead of only using features.

ESLAM [24] used three axis-aligned multi-scale feature planes for storing color and geometric features to limit memory growth in larger scenes. The coarse-scale planes enable efficient reconstruction of large free spaces, while the fine-scale planes enable high-quality rendering around surfaces. The rendering is based on traditional ray-marched volumetric methods with each query point projected onto all feature planes to get a set of features which are concatenated and passed to separate MLP decoders. These MLP decoders output color and TSDF values with the TSDF values converted to density through a learned sigmoid function.

With the move towards more explicit representations, Point-SLAM [58] introduced a point-based scene representation of 3D points with geometry and color features that can be decoded by a pair of MLPs. Unlike voxel-based volumetric approaches, Point-SLAM only samples 5 query points along the pixel-rays. The geometry and color features of each query point are calculated as an average of its 8 closest scene points weighted by their inverse distance. The features are then passed to the two MLPs to predict density and color values for rendering.

Finally, GlORIE-SLAM [87] combines a neural point cloud representation with a Disparity, Scale, and Pose Optimization (DSPO) layer. The neural point cloud is similar to Point-SLAM with deformable point coordinates and a decoder that takes projected features to produce a rendering. The DSPO layer is similar to DROID-SLAM in that it stores and optimizes per-frame depth maps and camera poses. Unlike DROID-SLAM, which uses a dense pixel-wise flow based on RAFT [74], GlORIE-SLAM uses a

sparse flow based on geometric features. These sparse flows are then combined with a predicted depth map and later optimized with a global bundle adjustment that minimizes reprojection errors similar to DROID-SLAM.

2.3. 3D Gaussian Splatting SLAM

The natural next step for scene representation was to discard the neural component and instead use a fully explicit representation [6, 32]. To achieve this, 3D Gaussian Splatting (3DGS) [32] uses 3D Gaussians for dense representation. Each Gaussian has an associated 3D mean, scaling matrix, rotation matrix, opacity, and color that get optimized during training. Gaussians can be split, fused or removed during optimization. An image can be rendered, through a custom rasterizer, by splatting these Gaussians into camera space and summing their colors for each pixel.

Unlike similar traditional SLAM methods that rely on sparse point clouds or surfels, 3DGS presents a dense Gaussian scene representation with an efficient differentiable rasterizer. This allows for fast learning-based scene optimization that provides photorealistic rendering quality. In contrast with NeRF-based methods that rely on a high number of slow neural network evaluations for each rendered pixel, 3DGS achieves real-time rendering and training speeds through its efficient rendering and scene optimization. Next, we will present an overview of 3DGS-based methods, divided into methods that rely on ground-truth depth priors and methods that work in the monocular RGB setting.

2.3.1. Depth-dependent Methods

The first 3DGS-based methods required depth priors to initialize, densify, and optimize their Gaussian camera parameters and camera poses. This is in contrast to some later methods, which were able to optimize the Gaussian scene without access to depth priors. GS-SLAM [83] was the first SLAM approach to use a 3D Gaussian scene representation. The method performs joint optimization of camera parameters and Gaussian parameters through gradient-based updates. GS-SLAM relies on depth priors for the initialization of Gaussians at pixel locations, with an adaptive densification strategy that adds new Gaussians in under-reconstructed areas based on depth loss. For camera tracking, GS-SLAM uses a coarse-to-fine technique that selects reliable Gaussians to optimize camera poses, resulting in faster optimization and robust estimation.

Next, Gaussian-SLAM [86] added 3D Gaussian sub-maps with associated sets of keyframes to enable scalable mapping in larger environments. Each sub-map contains a set of Gaussians that can be optimized independently based on the sub-map's kyframes. This approach allowed for better memory management as only the current sub-map's Gaussians needed to be kept in memory. This resulted in state-of-the-art visual synthesis results but sub-par tracking, especially due to the lack of Gaussian pruning mechanisms. Gaussian-SLAM attempted to remedy the tracking issue by using DROID-SLAM for tracking, although this still did not match other depth-based approaches.

SplaTAM [29] simplified the model by using only a single radius parameter, forcing the Gaussians to be isotropic, and using only three parameters for RGB color, eliminating the complexity of anisotropic scaling and spherical harmonic color synthesis. To improve reconstruction quality, SplaTAM uses opacity silhouette masks, derived from rendered depth discontinuities, to densify the scene with new Gaussians. However, this results in slower training and poorer performance compared to methods using full anisotropic Gaussians.

GS-ICP SLAM [16] uses an Generalized Iterative Closest Point (G-ICP) algorithm for fast camera pose estimation that directly aligns depth measurements with the existing Gaussian scene. These aligned depths can then be used to initialize and densify the Gaussian scene. The tracking and mapping can then be run simultaneously, achieving state-of-the-art camera pose estimation and incredibly fast training performance at over 107 FPS while maintaining competitive reconstruction quality.

2.3.2. Monocular RGB Methods

Later works introduced support for monocular input as well as ground truth depth. MonoGS [38] is the first 3D Gaussian Splatting based SLAM approach to support both RGB-D and monocular RGB inputs. MonoGS optimizes camera poses directly on the Gaussian map instead of using a separate tracking system. For Gaussian initialization in new parts of the scene, MonoGS uses depth projection when depth is available and random initialization in the monocular RGB setting. Finally, MonoGS adds an anisotropic regularizer to contain highly elongated Gaussians. This regularizer attempts to constrain

Gaussians that have large scale variations between their axes.

Next, Photo-SLAM [22] enhanced Gaussian primitives with ORB [57] features. These ORB features can be used for geometric tracking in addition to the implicit spherical harmonic features used for realistic rendering. The Gaussian map is initialized from matched ORB features and densified during optimization using inactive feature points. An important innovation is the addition of multi-level optimization over a Gaussian pyramid of images with increasing resolution to speed up initial optimization and improve performance.

In order to compete with older methods, Splat-SLAM [59] introduces a separate tracking module that functions as a front-end for Gaussian Splatting. This tracking module is based on GIORIE-SLAM's [87] Disparity, Scale, and Pose Optimization (DSPO) layer that takes in frame-to-frame dense optical flow and jointly optimizes camera poses along with per-pixel disparities. These can then be passed to the Gaussian Splatting backend for dense mapping and further optimized with global bundle adjustment and loop closure. Similarly, DROID-Splat [21] combines Gaussian Splatting with DROID-SLAM as a pose estimation and per-pixel disparity predictor. These are also optimized with global bundle adjustment and loop closure.

The progression of visual SLAM methods shows that dense methods such as 3DGS provide the best rendering quality while traditional visual SLAM methods provide the best camera pose estimation in the RGB setting. Similarly to the concurrently developed methods [59, 21], our method aims to bridge this gap in the RGB setting by combining a dense 3DGS-based scene representation for rendering with a novel sparse point-based scene for camera pose estimation.

Inspired by recent developments in Structure-from-Motion [79], our method aims to optimize camera poses of incoming frames using a sparse point-based scene. The main difference between our method and traditional visual SLAM methods is the use of simultaneously optimized sparse multi-view image correspondences from a pre-trained multi-view point tracker [28]. The benefit of sparse multi-view correspondences is that they are more consistent across a set of frames compared to frame-to-frame correspondences while also reducing GPU memory overhead compared to full frame methods. Our method uses these correspondences to triangulate and optimize a sparse point cloud while optimizing camera poses of incoming frames. This point cloud can then be used in conjunction with a monocular depth predictor to initialize and update our dense 3DGS-based scene based on incoming frames. This provides better rendering quality compared to the NeRF-based SLAM methods or the purely 3DGS-based SLAM methods.

3

Preliminaries

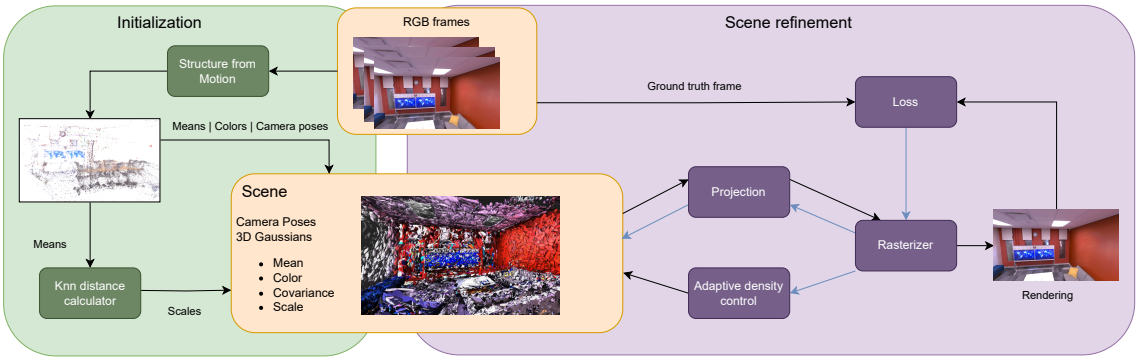


Figure 3.1: This figure presents an overview of 3D Gaussian Splatting as presented by Kerbl et al. [32]. The left side illustrates the initialization of the Gaussian scene using a Structure-from-Motion algorithm to get camera poses along with an initial point cloud with point colors and Gaussian scales. The right side illustrates the scene refinement through rendering from each camera pose, calculating loss with the ground truth frame, and back-propagating this loss to the Gaussian parameters. We show the execution flow with black arrows and gradient update propagation with blue arrows.

In this Chapter, we provide some preliminary overviews of the algorithms and models that we use in our approach. In Section 3.1, we provide an overview of 3D Gaussian Splatting by Kerbl et al. [32] that we adapt in our approach as a dense scene representation for photo-realistic rendering. In Section 3.2, we provide an overview of CoTracker [27, 28], a multi-view point tracker network that predicts multi-view image correspondences which we use for camera pose estimation and sparse point cloud triangulation in our approach. In Section 3.3, we provide an overview of Unidepth [52], which is a monocular depth estimation network that we use to update our dense scene representation based on incoming frames. Finally, Section 3.4 presents Structure-from-Motion algorithms that we use for optimizing camera poses and our sparse scene representation.

3.1. 3D Gaussian Splatting

3D Gaussian Splatting [32] seeks to create a photorealistic 3D scene representation from a set of frames $\mathcal{I} = (I_1, \ldots, I_{N_I})$ with $I_i \in \mathbb{R}^{3\times H\times W}$. This representation is made up of 3D Gaussians given by:

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

Each 3D Gaussian has a mean $\mathbf{m} \in \mathbb{R}^3$, covariance matrix $\Sigma \in SO(3)$, and opacity α. This allows for a compact representation with fast optimization. For directional appearance (color), 3DGS uses spherical harmonic coefficients $\mathbf{SH} \in \mathbb{R}^{16}$. These can be replaced with RGB color values or latent vectors to trade appearance for performance. 3DGS allows for fast synthesis of novel views due to the method's custom tile-based rasterizer that includes an efficient backward pass for fast optimization. Figure 3.1 shows an overview of 3D Gaussian Splatting.

During rasterization, 3DGS first splits the screen into 16x16 tiles before culling Gaussians against the view frustum and the tiles to only render Gaussians with at least 99% confidence of overlapping the tile. 3DGS also skips Gaussians that are too close or too far from the near plane. The Gaussians are then sorted based on depth and tile ID. Due to this, 3DGS does not need to perform any per-pixel sorting, which greatly improves performance at the cost of negligible appearance drops from incorrect alpha blending. After the Gaussians are sorted, 3DGS can perform per-tile rendering with separate per-pixel threads.

The rendering is done front-to-back by first projecting the 3D Gaussians into image space. Given a viewing transformation W 3DGS gets the covariance matrix Σ' as:

$$\Sigma' = JW\Sigma W^T J^T$$

where J is the Jacobian of the affine approximation of the projective transformation and Σ is the covariance matrix of the Gaussian. Xy et al. [66] show that the last column and row of Σ' can be skipped to obtain a 2x2 variance matrix which has the same properties. Once the 3D Gaussians are projected into image space, the color $\mathcal{C}_p$ of a pixel p can be rendered as:

$$\mathcal{C}_p = \sum_{i \in \mathcal{N}} c_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j)$$

The only stopping criterion for rendering is the saturation of α, which allows for an unlimited number of primitives to receive gradient updates. Per-pixel depth can be rendered similarly.

The training of 3DGS for a scene starts by using SfM for camera calibration and point cloud generation. This needs to be done ahead of time and is not suitable for an online system such as SLAM. We give the details of our initialization in the next chapter. The Gaussians for 3DGS are initialized at point cloud coordinates with an isotropic covariance and scale equal to the mean of the distance to the three closest points. This ensures good initial coverage of the scene. The optimization is done by rendering the scene from a given viewpoint and comparing the rendered image to the ground truth. The mean, opacity, and color are then optimized using Stochastic Gradient Descent. The problem with optimizing covariance matrices $\Sigma \in SO(3)$ is that they only have a physical meaning when they are positive definite. This cannot be easily constrained during optimization. The covariance matrices can instead be represented as a scaling matrix S and a rotation matrix R that get the corresponding Σ as:

$$\Sigma = RSS^T R^T$$

3DGS then optimizes a scale vector s and a rotation quaternion q that can be optimized during SGD and trivially converted back to S and R during rendering. The quaternion q needs to be normalized to obtain a valid unit quaternion. The loss function for SGD is $\mathcal{L}_1$ combined with a D-SSIM term:

$$\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda \mathcal{L}_{\text{D-SSIM}}$$

where λ is set to 0.2.

To better represent the scene, the original 3DGS formulation by Kerbl et al. uses an adaptive scheme for densification in under-reconstructed areas that have missing geometric features and over-represented areas that have very large Gaussians. This is done by cloning the Gaussians in under-reconstructed areas that are then moved in the direction of the positive gradient which increases the volume and number of Gaussians. On the other hand, the large Gaussians in over-reconstructed areas are split into two Gaussians with half the scale, and their position sampled as a PDF of the original Gaussian. This keeps the volume the same but increases the number of Gaussians. Culling is done by removing Gaussians with a small opacity before every densification step to prevent unnecessary rendering. This adaptive densification results in good reconstruction quality while keeping the number of Gaussians low for efficiency.

3.2. Multi-view Tracking

Multi-view tracking (MVT) is the process of tracking a set of query points across a sequence of RGB images. The output is a collection of point tracks, which are sets of per-frame 2D coordinates that

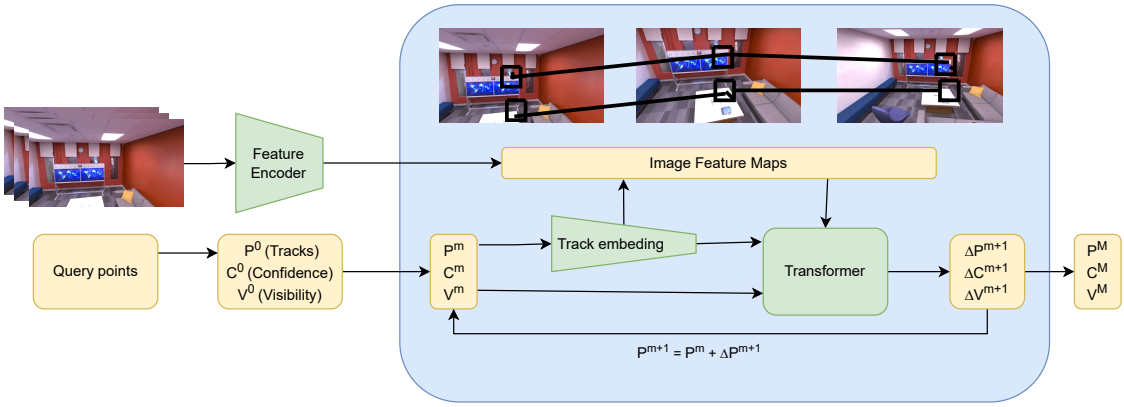


Figure 3.2: This figure presents an overview of CoTracker [27, 28], a multi-view point tracking model. The tracks are initialized to the given query points and iteratively updated by a transformer to obtain reliable image correspondences between the frames.

correspond to that point. These tracks can then be used for pose estimation and point triangulation. In addition, each observation has an associated visibility score that evaluates the quality and visibility of its corresponding point in that frame. This allows the model to track points even if they get occluded in some frames. For our method, we use the CoTracker and CoTracker3 [27, 28] multi-view trackers as they perform well under various conditions. Both versions work by iteratively refining an initial track using a 4D correlation map fed into a transformer network. An overview of CoTracker3 is provided in Figure 3.2.

The method starts by computing dense feature maps for each frame in the window. The images are down sampled multiple times with a feature map being computed at each scale. This provides both the fine details at finer scales and broader context at coarser scales. Next, a local feature patch is extracted around each query point in the query frame and the corresponding estimated points in all other frames. These feature patches are then compared via inner products to compute correlation scores, forming a 4D correlation map for refinement. One addition in CoTracker3 is the use of a multi-layer perceptron to project and simplify the features before the next step. This reduces the parameter count and computational overhead.

The main loop of this method is an iterative refinement of per-frame point track coordinates using a transformer. The inputs to this transformer are the displacement information, the correlation features, and the visibility estimates. The displacement information is the difference between the current and the query point location to get the current total displacement. The displacement information is encoded using Fourier encoding. The correlation features are sampled from the 4D correlation map and encoded in CoTracker3. The visibility estimates are fed directly without an encoding. The transformer network outputs updates to the coordinates and visibility. The updates are then added to the point coordinates and visibility predictions before the next iteration.

A significant improvement of CoTracker3 comes from the fact that it is trained on more data. The training is initially done on synthetic data (e.g., Kubric dataset) that contain precise annotations. This is followed by semi-supervised training on real data. For this, CoTracker3 leverages pseudo-labels from multiple teacher models (versions of CoTracker3 and TAPIR) on unlabeled real videos. This allows CoTracker3 to perform better on real data than other multi-view trackers.

3.3. Monocular Metric Depth Estimation

Monocular Metric Depth Estimation (MMDE) is the task of predicting per-pixel metric depth from a single RGB image. For this task, we use the Unidepth [52] model, illustrated in Figure 3.3. Unidepth introduces a novel approach to monocular depth prediction that does not require ground truth camera intrinsic parameters. This is useful because our monocular SLAM system does not have access to ground truth camera intrinsic parameters, and they need to instead be optimized at runtime. To achieve this, Unidepth proposes a pseudo-spherical camera representation defined by azimuth, elevation, and logarithmic depth to decouple the angular and depth components.

The model consists of an encoder, a camera module, and a depth decoder. The encoder, based

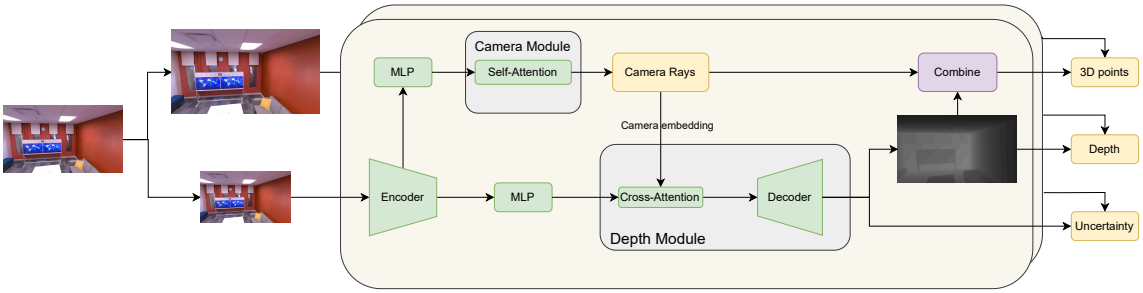


Figure 3.3: This figure presents an overview of Unidepth [52], a Monocular Metric Depth Estimation. The top line shows the camera module which predicts camera rays from an encoded frame. The bottom line shows the depth module which predicts a depth map with uncertainty from the encoded frame and its predicted camera rays. A 3D point cloud is also calculated from the predicted camera rays and depth map. This process is done at 2 levels of detail that are combined to produce the final output.

on the Vision Transformer (ViT), divides the image into patches that are encoded into tokens and processed by transformer layers to produce feature maps at multiple scales. These multi-scale feature maps are then passed to the camera module, which outputs camera parameter updates.

The camera parameters are initialized using a pinhole model. The intrinsic parameters can then be used to project the pixels back into camera space, providing a direction and scale of the scene depth. From this, an azimuth and elevation angle can be computed to capture every pixel's orientation relative to the camera. This information is then encoded using a spherical harmonic encoding and passed to the depth module along with the initial multi-scale features.

The depth module takes the features and encoded pixel rays to produce metric depth. The multi-scale features are first averaged over the scale dimension and passed to a cross-attention mechanism along with the camera module outputs to produce conditioned features. An FPN-style decoder then takes these features and progressively up-samples them while refining them with self-attention layers. The final output is a log-depth map which is exponentiated to get the final depth map.

3.4. Structure-from-Motion

Structure-from-Motion (SfM) [76] is a method for producing and optimizing a sparse 3D point cloud $X = \{\mathbf{x}^j\}_{j=1}^{N_\mathbf{x}}$ of $N_\mathbf{x} \in \mathbb{N}$ 3D points $\mathbf{x}^j \in \mathbb{R}^3$ and camera poses $\mathcal{P} = \{P_1, \dots, P_{N_I} \mid P_i \subset \mathbb{R}^{3\times4}\}$ from a set of unordered RGB images $\mathcal{I} = \{I_1, \dots, I_{N_I}\}$ with $I_i \in \mathbb{R}^{3\times H\times W}$. This is achieved by extracting features from the images, estimating their relative poses and intrinsic parameters, and using these to triangulate 3D points. These 3D points and camera poses are then further optimized using bundle adjustment and pose refinement.

The aim of bundle adjustment is to jointly optimize all camera poses and 3D point locations based on their 2D observations in each camera. This is achieved by setting up a least-squares problem:

$$E = \sum_j \rho_j \left(\left\| \pi \left(\mathbf{P}_c, \mathbf{X}_k \right) - \mathbf{x}_j \right\|_2^2 \right)$$

with π a function that projects the 3D point locations $\mathbf{X}_k$ into camera space using the camera parameters $\mathbf{P}_c$. The resulting point projections are compared with the given 2D point coordinate $\mathbf{x}_j$ in camera space using a robust loss function ρ_j that can be observation-specific for outlier handling. This problem is then solved using the Levenberg–Marquardt solver with a chosen loss function and camera model.

Pose refinement aims to refine initial pose estimates for a set of cameras based on their intrinsic parameters, visible 3D points, and the points projections in the cameras. This is achieved by setting up the same non-linear least-squares problem as bundle adjustment, but keeping the 3D point locations fixed. The aim is to minimize the sum of reprojection errors across all 3D points and their 2D observations by updating only the camera parameters.

4

Methodology

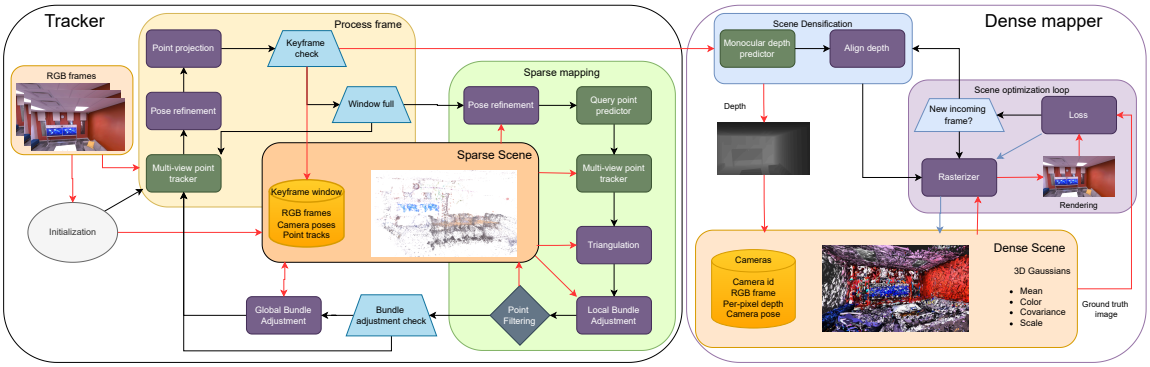


Figure 4.1: Overview of the full pipeline of our SLAM approach. Black arrows indicate subsequent execution, red arrows show the flow of data, and blue arrows show the flow of gradients.

The goal of our approach is to simultaneously estimate accurate camera poses, and to create a photorealistic dense scene representation from an RGB video. An overview of our approach is provided in Figure 4.1. For the first task, we optimize camera parameters using a sparse point based representation that is optimized based on multi-view point tracks from a pre-trained network as described in Section 4.1. We first initialize the sparse scene and camera poses based on an initial window of frames. Next we use this scene to estimate camera poses of new incoming frames, adding them to the frame window if they meet certain criteria. Once we fill the keyframe window, we densify the scene by triangulating new points. We periodically run a global bundle adjustment to optimize the sparse scene and all camera poses.

In a separate thread, we optimize a dense scene representation, consisting of 3D Gaussians, for photorealistic rendering as described in Section 4.2. These 3D Gaussians are initialized and densified based on predicted depth maps that are aligned to the sparse scene representation. The scene is then continuously optimized based on keyframes from our tracking front-end. In the following sections, we provide in-depth explanations of different components of our SLAM approach.

4.1. Tracking

Given a 2D RGB video $\mathcal{I}$ of N frames $\mathcal{I} = \{I_1, \ldots, I_{N_I}\}$ with $I_i \in \mathbb{R}^{3\times H\times W}$, our method aims to optimize accurate camera projection matrices $\mathcal{P} = \{P_1, \ldots, P_{N_I} \mid P_i \subset \mathbb{R}^{3\times 4}\}$. These matrices consist of a rotation matrix $\mathcal{R} = \mathbb{R}^{3\times 3}$ and a translation vector $\mathcal{T} = \mathbb{R}^3$. Additionally, we optimize shared intrinsic camera parameters that are used to project 3D points into 2D camera coordinates. The intrinsic parameters consist of the focal lengths and the principal point in the x- and y- directions. Optionally, we optimize extra distortion coefficients which model real-world camera lenses that deviate from an ideal pinhole camera. We assume that the input video was captured on a single camera and therefore these intrinsic parameters are shared for all frames.

We optimize the camera poses and intrinsic parameters using a sparse point-based scene representation $X = \{\mathbf{x}^j\}_{j=1}^{N_\mathbf{x}}$ of $N_\mathbf{x} \in \mathbb{N}$ 3D points $\mathbf{x}^j \in \mathbb{R}^3$. This scene is triangulated using image correspondences between multiple frames predicted by a pre-trained multi-view point tracking model. The camera parameters and the scene can then be optimized through local and global bundle adjustment by minimizing the error between these point tracks and the 3D point projections.

This scene needs to first be initialized along with the initial set of cameras and an initial shared camera intrinsic matrix. We describe this process in Section 4.1.1 and Figure 4.2. This point-based scene can then be used, in combination with a multi-view point tracker, for estimating the poses of new incoming frames. We describe this process in Section 4.1.2 and Figure 4.3. Finally, as we observe new parts of the scene, we also need to update our point-based representation so that it can be used to estimate the poses of new incoming frames, as described in Section 4.1.3 and Figure 4.4.

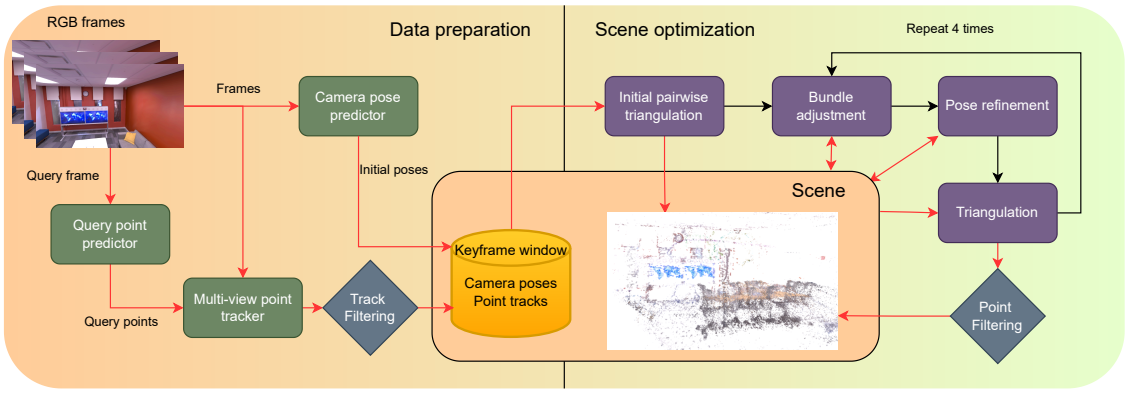


Figure 4.2: Overview of the initialization of our sparse point cloud based scene used for tracking. The left side shows the data preparation of our first window of frames, while the right side shows our iterative optimization of the initial scene. Black arrows indicate subsequent execution and red arrows show the flow of data

4.1.1. Initialization

As explained in Section 4.1, our tracking module optimizes the camera poses of the incoming frames based on the camera poses of previous frames, the shared camera intrinsics, and a point-based scene representation. Therefore, we need to initialize all of these scene components using a window of the first N_f frames before we can start optimizing the camera poses of new incoming frames.

The overview of our scene initialization is shown in Figure 4.2, which starts with predicting an initial set of camera poses of the window frames and initial shared camera intrinsic parameters using a pre-trained multi-view camera predictor [79]. Next, we predict image correspondences between the window frames using a pre-trained multi-view point tracker [28]. The camera parameters and image correspondences can then be used to triangulate the 3D points in our scene. Finally, we iteratively optimize our scene, camera poses, and intrinsic camera parameters by minimizing the error between the projected 3D points and the predicted point tracks.

First, we need an initial set of rough camera poses and initial shared camera intrinsic parameters, which we obtain using a multi-view camera pose estimator. This pose estimator takes a set of encoded images and outputs a set of camera poses and camera focal lengths. The estimator initializes the poses to world origin coordinates and encodes them. The image and camera encodings are combined to create input tokens. These input tokens are then fed into a deep transformer network that iteratively refines the camera parameter prediction for $n = 4$ iterations. We found that more iterations do not seem to improve performance. The output focal lengths are then multiplied by the smaller image dimension, halved, and averaged to get an initial focal length component of the intrinsic parameters as:

$$f = \frac{\tilde{f} * min(H_{image}, W_{image})}{2}$$

Additionally, we initialize the principal point of our intrinsic matrix to the center of the image and the optional distortion coefficients to 0. These intrinsic parameters are then shared for all frames as we assume a single camera for the entire sequence.

Next, we predict a set of image correspondences between the frames of our initial window using a query point predictor and a multi-view point tracker. We select the middle frame as the query frame as it should have the highest overlap with the other frames in the window. This is the case if the camera is moving or rotating in a constant direction, which is the most common case. Additionally, camera movements that change direction result in high overlap for any query frame, therefore the middle frame still works well. Having sufficient overlap between the query frame and all other frames ensures that a high number of the predicted query points are visible and trackable in other frames, which improves stability of correspondences. The query frame is passed to a query point estimator to predict reliable query points for tracking. Our testing has shown that a combination of a neural query point predictor [12] and a geometric query point predictor [37] produces the best results. These query points are tracked over the frame window using our pre-trained multi-view tracker described in Section 3.2. Each observation in a track has an associated visibility score and confidence score, output by the multi-view tracker. These visibility and confidence scores are then used to filter out unstable or obscured tracks. Following the original method [28], we filter tracks with visibility scores less than 0.05 and confidence scores less than 0.5.

The filtered tracks are then used to triangulate an initial set of 3D points for every pair of query and non-query frame through a method based on direct linear transformation [71]. This process takes a set of N camera projection matrices $\mathbf{P}_i \in \mathbb{R}^{3\times4}$ along with a set of image correspondences $\mathbf{p}_i = [u_i, v_i]^T$ and outputs a set of $N-1$ point clouds through pairwise triangulation. The method forms a linear system from point observations and camera projection matrices with the constraint that each observation must lie on the projection of the corresponding 3D point expressed as:

$$\tilde{\mathbf{p}}_i \times (\mathbf{P}_i \mathbf{X}) = \mathbf{0}$$

where $\tilde{\mathbf{p}}_i$ is the normalized homogeneous 2D point and $\mathbf{X}$ is the normalized homogeneous 3D point. This constraint can then be reformulated into the linear system:

$$\mathbf{A} = \sum_{i=1}^{N} [(\mathbf{I}_3 - \tilde{\mathbf{p}}_i \tilde{\mathbf{p}}_i^T)\mathbf{P}_i]^T [(\mathbf{I}_3 - \tilde{\mathbf{p}}_i \tilde{\mathbf{p}}_i^T)\mathbf{P}_i]$$

where $\mathbf{I}_3$ is the 3×3 identity matrix and $\tilde{\mathbf{p}}_i \tilde{\mathbf{p}}_i^T$ is the outer product of the 2D observations that projects camera rays into the observed direction. Minimizing this system gives the 3D points best aligned to point tracks and camera poses. Using these triangulated points and camera poses, we compute cheirality masks to ensure positive depth in both camera coordinate systems for all pairs of query and non-query frames. These are calculated as:

$$\mathbf{P}_i \mathbf{X} \cdot [0, 0, 1]^T > 0 \quad \forall i$$

which holds true for all points with a positive depth in both camera coordinate systems.

We additionally compute the angle between the two camera-to-point rays and use it to produce triangle masks based on a minimum viewing angle. This is because small triangulation angles are very sensitive to noise, with small errors in the 2D observations resulting in large changes to the triangulated 3D point location. For each triangulated point $\mathbf{X}$ the angle θ between the camera centers $\mathbf{C}_1$ and $\mathbf{C}_2$ is calculated as:

$$\cos(\theta) = \frac{\|\mathbf{r}_1\|^2 + \|\mathbf{r}_2\|^2 - \|\mathbf{b}\|^2}{2\|\mathbf{r}_1\|\|\mathbf{r}_2\|}$$

where $\mathbf{r}_1 = \mathbf{X} - \mathbf{C}_1$ and $\mathbf{r}_2 = \mathbf{X} - \mathbf{C}_2$ are the ray vectors from each camera center to the 3D point, and $\mathbf{b} = \mathbf{C}_2 - \mathbf{C}_1$ is the baseline vector. We filter points based on this triangulation angle using a threshold t_t. The point cloud with the highest number of inliers, based on visibility, confidence, triangulation angle, and cheirality masks, is chosen for pose refinement.

Before refining the camera poses, we first filter the best point cloud based on reprojection error, which measures the consistency between the 2D observations and the 3D points. High reprojection error indicates that the image correspondences do not align with each other or with the scene as a whole. We start by projecting each 3D point into each camera using the camera parameters as:

$$\begin{bmatrix} \bar{X} \\ \bar{Y} \\ \bar{Z} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_1 \\ R_{21} & R_{22} & R_{23} & t_2 \\ R_{31} & R_{32} & R_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$
$$\tilde{x} = K \cdot f\left(\begin{bmatrix} \bar{X}/\bar{Z} \\ \bar{Y}/\bar{Z} \end{bmatrix}\right) \tag{4.1}$$

where XYZ are the coordinates of the 3D point, R and t are the camera rotation and translation, $\tilde{x}$ is the projected point in camera space, f is an optional distortion function, and K is the shared camera intrinsic matrix. The resulting projections $\tilde{x}$ can then be compared with the predicted tracks x, and the outputs are normalized and squared to obtain a final reprojection error for every observation as:

$$e_i = \|\tilde{x}_i - x_i\|_2^2 \tag{4.2}$$

Observations with a reprojection error higher than a threshold t_{rpo} are masked for optimization and points with fewer than three stable observations are filtered.

Finally, we optimize the initial camera poses, shared camera intrinsic parameters, and the filtered point-based scene based on the predicted point tracks using bundle adjustment. This is achieved by setting up a least-squares problem that minimizes the equation:

$$\sum_j \rho_j \left(\left\| \pi\left(\mathbf{P}_c, \mathbf{X}_k\right) - \mathbf{x}_j \right\|_2^2 \right) \tag{4.3}$$

with π the function that projects the 3D point locations $\mathbf{X}_k$ into camera space using the camera parameters $\mathbf{P}_c$ described in 4.1. The resulting point projections are compared with the given 2D point coordinate $\mathbf{x}_j$ in camera space using a loss function ρ that optionally weighs down outliers. This problem is then solved using the Levenberg–Marquardt solver with a chosen loss function. In our method, we use the Cauchy loss function that is less sensitive to outliers by applying a logarithmic penalty to large errors. We then refine only the poses using Equation 4.3 with the scene points remaining constant.

Next, we perform robust triangulation based on the optimized camera poses and the original point tracks. For this we use a RANSAC-based method with local refinement. The method begins by generating a set of initial triangulation hypotheses based on a set of randomly selected 2-view combinations using the original two-view triangulation. Inliers in each hypothesis are then selected based on triangulation angles and cheirality constraints using the methods described above. We additionally consider the angular reprojection error, measuring the consistency between observed image rays and projected 3D points as:

$$\epsilon_{\text{angular}} = \arccos(\text{clamp}(\mathbf{r}_{\text{obs}} \cdot \mathbf{r}_{\text{proj}}, -1, 1))$$

where $\mathbf{r}_{\text{proj}} = \frac{\mathbf{P}_i\mathbf{X}}{\|\mathbf{P}_i\mathbf{X}\|}$ is the normalized projected ray. Each hypothesis is re-triangulated using the inlier tracks in all views instead of just the initial 2-view pair. The optimal hypothesis is selected based on a residual indicator calculated as:

$$\mathcal{I} = N_{in} + \frac{\epsilon_{max} - \bar{\epsilon}_{in}}{\epsilon_{max}}$$

where N_{in} is the number of inliers, $\bar{\epsilon}_{in}$ is the mean inlier angular reprojection error, and ϵ_{max} is the maximum acceptable angular reprojection error equal to 2 degrees in our method. This method balances inlier count with sparse map accuracy.

The triangulation, filtering, bundle adjustment, and pose refinement steps are repeated three times to get a final refined scene initialization. We store the camera poses of the window frames, shared intrinsic camera parameters, point tracks, and the 3D triangulated points as a scene for the optimization of new incoming frames.

4.1.2. Keyframe Selection

Once the scene is initialized, we can start the online optimization of new incoming frames. An overview of our keyframe selection is provided in Figure 4.3. Our approach optimizes the scene over a sliding

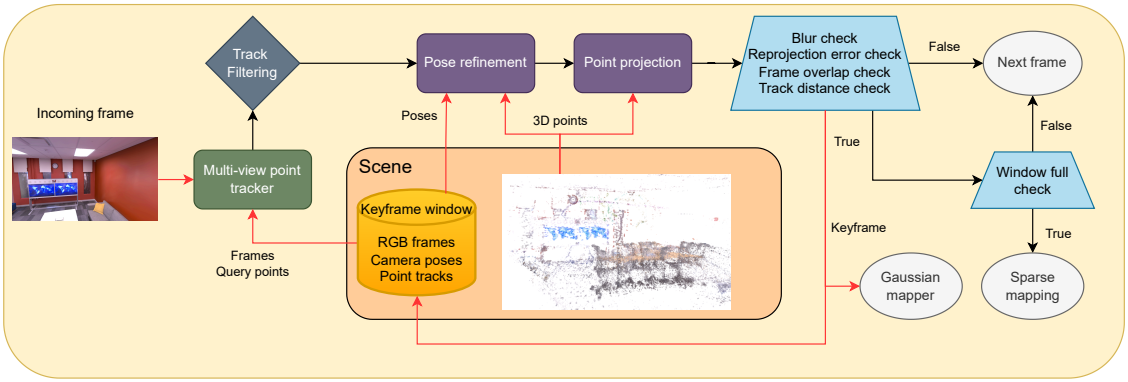


Figure 4.3: This figure presents an overview of our per-frame camera pose estimation and our keyframe selection. Black arrows indicate subsequent execution and red arrows show the flow of data

window of N_f keyframes. Each existing frame in the window has an optimized pose from the initialization or the previous optimization step.

For each new frame, we first add it to the end of the window and then run a fast multi-view point tracker [28], described in Section 3.2, over the whole window plus the new frame. The tracker takes in the RGB images along with the coordinates of a set of query points in a query frame. The output is a set of coordinates for each query point in each frame of the window along with their visibility score. We use the last sparse mapping keyframe in the window as a query frame. For query points, we use the uv coordinates of the most recently triangulated points in this query frame. In order to speed up performance, we select a subset of these query points based on frame coverage. For this, we divide the frame into a grid and first try to get an equal number of points in each cell. Cells that do not have enough points overflow into the neighboring cells to maintain good coverage.

Once we have the tracks, we load the camera poses for all existing keyframes in the frame window and initialize the new frame's pose to the last optimized frame's pose. These poses are then used along with the point tracks to refine the camera pose of the most recent frame. For this, we use the pose refinement, based on bundle adjustment, as described in Section 4.1.1 and Equation 4.3. The pose refinement takes the shared camera intrinsic parameters, per-frame camera poses, and a set of 3D points along with their uv coordinates in the frame. These are used to refine the camera pose by minimizing the sum of the reprojection errors across all points and tracks. This pose is saved in a frame dictionary and used for projecting the visible points to calculate the reprojection error and point depths. We then use this pose to project the visible 3D points and calculate the reprojection errors as described in Section 4.1.1 and Equations 4.1 and 4.2.

As explained in Sections 3.2 and 7.1, we have found that our tracker struggles with point tracking on real-world frames with high motion blur. To mitigate the motion blur issue, we calculate the motion blur of every new frame and reject it if it is greater than a threshold t_b. We calculate the blur based on edge and movement detection. We use Sobel filters to calculate the X and Y gradients which imply vertical and horizontal edges respectively. These filters are applied using a 2D convolution on grayscale frames. Combining these gradients allows us to calculate the overall edge strength for a frame. We then compare this edge strength of the most recent frame to the previous frames in the window and reject the frame if the blur is much higher. This makes our approach adaptable to different scenes and even different parts of the same scene.

We have also found that point tracks over frames with very low motion tend to be difficult to optimize. This is due to small updates to these point positions becoming unstable with small changes sometimes resulting in large changes of the resulting triangulation. To mitigate the low-motion issue, we compute the distances of the track coordinates between the new frame and the last keyframe using Equation 4.2. If the median distance drops below a threshold t_d, we also reject the frame.

Finally, we add the frame to the keyframe window if one of three criteria is met. The first criterion is if the ratio between visible and non-visible track points drops below a threshold t_o. This ratio balances the speed of our approach, by lowering the number of the slow triangulation steps, with the accuracy of our camera poses. Next, we check if the reprojection error is greater than a threshold t_{rp}. We have found that the reprojection error tends to increase as the pose estimation error increases. Furthermore,

both errors tend to accumulate between sparse densification steps. Therefore, we attempt to limit the number of frames with a high reprojection error, to prevent error accumulation, by filling up the keyframe window and triangulating new points. Finally, we check if we went more than t_f frames without a keyframe, to retain the accuracy of non-keyframe camera poses.

If the mean reprojection error becomes greater than a threshold t_{rpr}, we restart keyframe selection with stricter criteria. We provide all the thresholds in Chapter 5. Once the frame window becomes full, we can move onto sparse mapping.

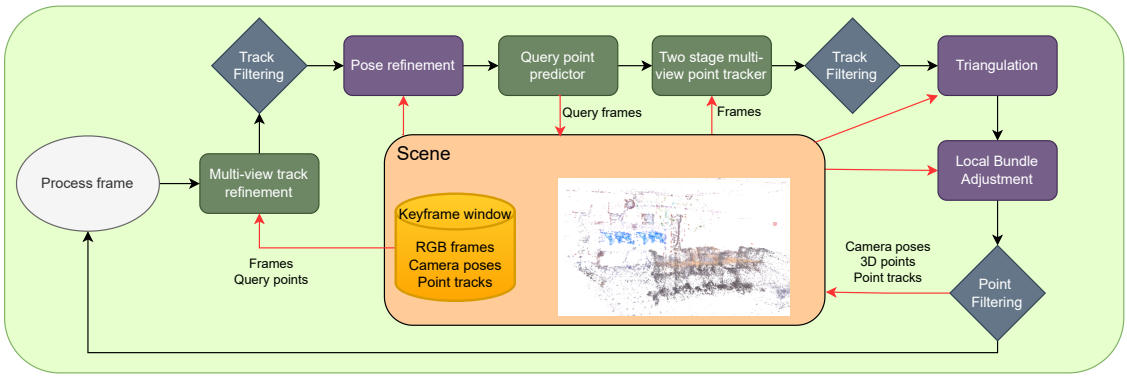


Figure 4.4: This figure presents an overview of the densification of our sparse scene representation which consists of refining the camera poses of the current keyframe window and then triangulating new points based on predicted image correspondences. Black arrows indicate subsequent execution and red arrows show the flow of data

4.1.3. Sparse Map Densification

As described in Section 4.1.1, we initialize our sparse point-based scene on a first set of frames. However, these frames do not observe the entire scene and only see a small part of the scene. Therefore, as we observe new parts of the scene based on new incoming frames, we need to densify the scene so that it can be used for the pose optimization of further incoming frames. We do this whenever our frame window becomes full to balance the speed of our approach with camera pose estimation quality. We start this densification by refining the existing point correspondences from our keyframe selection, which we then use for additional camera pose refinement. Next, we predict new query points and track them in all frames of the window so that they can be used to triangulate 3D points in the newly observed parts of the scene. Finally, we optimize the new camera poses and 3D points using local bundle adjustment and filter the points based on reprojection error as described in Section 4.1.1. Optionally, we run a global bundle adjustment over all keyframes and 3D points before we continue optimizing new incoming frames. We present an overview of our sparse map densification in Figure 4.4.

First, we take the multi-view point tracks of the last triangulated points across the full keyframe window from the last keyframe selection step. Next, we refine these tracks using our multi-view point predictor, described in Section 3.2, at a higher resolution. This provides higher-quality image correspondences. We then use these tracks to optimize the camera poses using pose refinement, as described in Section 4.1.1 and Equation 4.3.

Next, we select the last and midpoint keyframes as query frames to be used to predict query points using a query point predictor. We use the last keyframe as it should observe the most new points in the newly observed part of the scene. Additionally, we use the middle keyframe to ensure high overlap with the early frames in the window. This improves the stability of correspondences so that they can be used for local bundle adjustment of camera poses. Our testing has shown that a combination of a neural query point predictor [12] and a geometric query point predictor [37] produces the best results for both synthetic and real-world scenes. These new query points are tracked using the two stage multi-view tracker over the keyframe window. We use the tracks along with the optimized camera poses to triangulate new points in newly observed parts of the scene using the RANSAC-based algorithm described in Section 4.1.1.

These new triangulated points are then projected into each camera to calculate the reprojection error for filtering as described in Section 4.1.1 and Equations 4.1 and 4.2. Observations with a reprojection error higher than a threshold t_{rpo} are masked for optimization and points with fewer than three

stable observations are filtered. We also use the visibility and confidence scores predicted by the point tracker [28] to filter out unstable or obscured tracks. Following the original method [28], we filter tracks with visibility scores less than 0.05 and confidence scores less than 0.5. Next, we run a local bundle adjustment, as described in Section 4.1.1 and Equation 4.3 on the points that were triangulated in the last window and in the current window. We keep the oldest pose and last window's points constant as they should already be optimized and only optimize the new points and camera poses. These optimized points are then again filtered based on reprojection error to remove any badly optimized points.

Every B sparse mapping steps, we run a global bundle adjustment as described in Section 4.1.1 and Equation 4.3 to optimize all camera poses, the shared intrinsic camera parameters, and all 3D points. Global bundle adjustment is performed every two sparse mapping steps. After the fifteenth mapping step, the interval increases and adjustments occur every four steps. Our testing has shown that additional global adjustments are necessary during the early optimization stages to ensure a robust initial scene. Once the scene becomes stable, we can decrease the frequency of global bundle adjustments to enhance speed. This adaptive approach balances camera pose quality with faster performance.

4.2. Dense Mapping

The second goal of our approach is to create a dense photo-realistic 3D scene representation from the given 2D RGB video. For this, we use 3D Gaussian hyper primitives based on 3D Gaussian Splatting [32], described in detail in Section 3.1 with an overview shown in Figure 4.5. The main advantage of 3D Gaussian Splatting is that it provides a dense scene representation, unlike traditional SLAM methods. Furthermore, this scene representation improves training and rendering speed, compared to dense NeRF-based methods [39, 84, 87], while improving visual synthesis quality.

The original setting for 3D Gaussian splatting required a point cloud optimized over all frames for initialization. We do not have access to this point cloud in the SLAM setting, as the frames are processed sequentially. Other 3DGS-based methods initialize and densify these Gaussians randomly [38], or based on frame-to-frame image correspondences [22]. Concurrent methods use monocular depth predictions [59, 21], predicted based on frame-to-frame correspondences. Our method instead uses monocular depth predictions, aligned to our sparse point-based scene, which we optimize using multi-view image correspondences as described in Section 4.1. The original 3DGS setting also required camera poses of all frames for optimization. Other 3DGS-based methods optimize these based on the Gaussian scene [38], or based on frame-to-frame image correspondences [22]. Concurrent methods [59, 21] predict camera poses using a neural module, based on frame-to-frame flow. Our method uses sparse scene for camera pose estimation as described in Section 4.1.2. The camera poses of keyframes are then forwarded to our Gaussian mapper along with the visible sparse points for densification and optimization of Gaussians.

We provide an overview of our dense Gaussian scene and our rendering pipeline in Section 4.2.1. Next, we provide the details of the initialization and densificitation of this scene in Section 4.2.2. Finally, we provide an overview of the optimization of our dense scene in Section 4.2.3.

4.2.1. Rendering

As explained in Section 4.2, our approach uses a dense scene representation based on 3D Gaussian Splatting [32]. This scene representation is made up of 3D Gaussians given by:

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

where x is a 3D coordinate and Σ is a covariance matrix representing the scale and rotation of the Gaussian. Each 3D Gaussian has a mean $\mathbf{m} \in \mathbb{R}^3$, a covariance matrix $\Sigma \in SO(3)$, and an opacity α. This provides a compact representation that can be efficiently optimized. For directional appearance (color), we use spherical harmonic coefficients $\mathbf{SH} \in \mathbb{R}^{16}$.

These Gaussians can be used to efficiently render a scene using a custom tile-based rasterizer [32]. During rasterization, the screen is first divided into 16x16 tiles, which are associated with their visible Gaussians. The Gaussians are then culled against the view frustum and the associated tiles. Before rendering, the 3D Gaussians must first be projected into camera space as:

$$\Sigma' = JW\Sigma W^T J^T$$

where Σ is the covariance matrix of the corresponding Gaussian, Σ' is the projected covariance matrix, W is the viewing transformation, and J is the Jacobian of the projective transformation. The last column and row of the projected covariance matrix Σ' can then be skipped to obtain a 2x2 variance matrix with the same properties. These projected Gaussians are then sorted, based on depth and tile ID, and rendered front to back for each tile. For each pixel p in the output image, we render its color $\mathcal{C}_p$ as:

$$\mathcal{C}_p = \sum_{i \in \mathcal{N}} c_i \alpha_i \prod_{j=1}^{i-1} (1 - \alpha_j)$$

where $\mathcal{N}$ is the number of visible Gaussians for the tile corresponding to the pixel and c_i and α_i are the color and opacity of the Gaussian at index i.

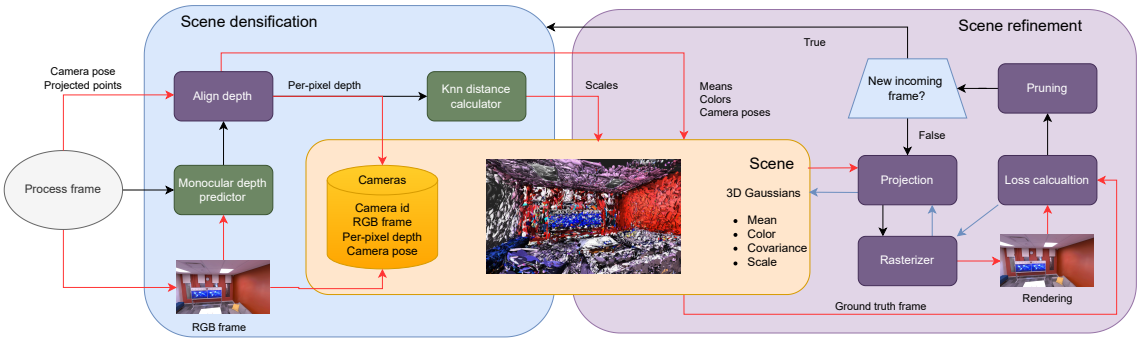


Figure 4.5: This figure presents an overview of our dense 3D Gaussian Splatting module. The left side shows our scene densification that is performed whenever we receive a new frame from the camera pose estimation module. The right side shows the continuous refinement of the dense Gaussian map based on rendering and gradient updates. Black arrows indicate subsequent execution, red arrows show the flow of data, and blue arrows show the flow of gradients.

4.2.2. Initialization and Densification

As explained in Section 4.2.1, our dense scene representation uses 3D Gaussians to render photorealistic images. However, this Gaussian scene needs to first be initialized based on an initial window of frames. Furthermore, these initial frames do not observe the entire scene, but only see a small part of the scene. Therefore, as we observe new parts of the scene, based on new incoming frames, we need to extend and densify the scene so that it can be used for the rendering of these new frames.

We both initialize and densify the 3DGS map using the same strategy. Whenever a new frame is received from the tracking thread, the Gaussian mapper uses Unidepth [52], a pre-trained monocular depth predictor, to generate an initial dense depth map. This map is then aligned to the projected sparse points provided from the sparse tracking map by calculating the scale difference between the depths and distances. Before we can use the depth map for densification, we first need to down-sample it with the down-sampling rate r_d depending on the image resolution. We do this to limit the number of Gaussians, which speeds up optimization and lowers GPU memory usage. We then project the depth values into the scene to get the means of our new Gaussians.

Next, we calculate, for each Gaussian, the mean distance to its k nearest neighbors. For this, we use a fast GPU-accelerated algorithm designed for 3D point-cloud processing. The algorithm encodes points using Morton codes [41] to preserve spatial locality, sorts them, and partitions them into boxes of size 1024. For each point q, we calculate the squared distance $d^2(q,p)$ to a point p as:

$$d^2(q,p) = (q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2$$

The algorithm maintains the set of k smallest distances for each point q that are updated based on newly examined points. First, the algorithm examines the immediate spatial neighborhood of the query point q based on the Morton encoding. Next, all boxes that have a distance greater than the current smallest distance are pruned. The points in the remaining boxes are then examined to get the k nearest points. Finally, for each point q_i, we calculate the mean distance d_i to its k nearest neighbors as:

$$d_i = \frac{1}{3} \sum_{j=1}^{k} \sqrt{d^2(q_i, p_j)}$$

We set k to 3 because it leads to a faster calculation without affecting performance or robustness to outliers. This mean distance is then used to initialize the isotropic scales of Gaussians. This approach provides good coverage of the newly explored scene areas without initializing unnecessary Gaussians, which could negatively impact quality and performance. The rotations are initialized to 0 and the opacities to 0.1. Lastly, the spherical harmonic coefficients are acquired from the raw pixel color at the corresponding pixel location in the original image.

Finally, we prune the existing Gaussians every 200 optimization steps to improve rendering quality and speed. We remove Gaussians with an opacity below 0.005 on the basis of our tests. This process removes Gaussians that do not sufficiently contribute to the rendering to improve memory overhead and rendering speed. We also remove Gaussians whose scale is beyond a certain threshold as these Gaussians tend to negatively affect rendering performance. Our tests have found that a threshold of 2.5 meters for synthetic datasets and 10 for real-world datasets performs well. This threshold is multiplied by the scene scale during optimization, where the scene scale is the mean ratio between the predicted depth values and the depth values from our sparse scene. This is done under the assumption that the scene should be represented with Gaussians that represent their corresponding elements.

4.2.3. Optimization

While waiting for a new densification frame, our mapper optimizes the existing Gaussians over the previously received keyframes. This is done by first rendering the Gaussian scene from the frames predicted in the tracking thread. For this we use the original differential Gaussian rasterizer from 3D Gaussian Splatting [32]. Next, we calculate the loss between the rendered image and the ground truth. This loss consists of an absolute loss term $\mathcal{L}_1$ combined with a D-SSIM term:

$$\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{\text{D-SSIM}}$$

with λ equal to 0.2 following the original paper [32]. Further details of the renderer are discussed in the preliminaries section, while the optimization parameters are detailed in the implementation details section.

5

Implementation Details

We implemented our approach with Python and various libraries. In Section 5.1, we discuss the libraries that we use in our approach. Next, in Section 5.2, we provide the implementation details of the pre-trained models that we use for initial camera prediction and multi-view point tracking. Finally, in Section 5.3, we provide any parameters that are used in our camera pose estimator or dense Gaussian mapper. All experiments were run on a Ubuntu 22.04 desktop with an AMD Ryzen 5 5600x 6-core CPU, RTX 3070 GPU with 8GB VRAM and 64GB RAM.

5.1. Libraries

Our approach is written in Python 3.10 taking full use of PyTorch [50] which provides Python bindings for the torch library [50]. Torch is a library of various deep learning algorithms, with strong GPU support written in CUDA [48]. We use PyTorch 2.5 [50] on top of CUDA 12.4 [48] to get the latest features for machine learning and optimization. Our dense mapping module uses the Differential Gaussian Rasterization library [32], written in CUDA [48], for fast rendering and Gaussian optimization. We also use simple-knn [4] to get the scales of newly initialized Gaussians in the scene.

Our tracking module uses the pycolmap library [60, 61] for python bindings of the original COLMAP [60, 61] code written in C. We use this library for per-frame pose estimation, full window pose refinement, local and global bundle adjustment, and point filtering after bundle adjustment. COLMAP allows for setting selected points and cameras as constant, which is very useful for local bundle adjustment and keeping the scene aligned during optimization. COLMAP additionally provides various camera types including simple radial, which has one extra radial distortion parameter, and OpenCV which has four extra radial distortion parameters. We use the simple radial camera for synthetic datasets and the OpenCV type for real-world datasets to better simulate the real-world camera distortion effects. COLMAP also offers various loss functions, such as Cauchy loss, to better handle outliers. The bundle adjustment optimizer uses the Pyceres library [1] as a solver. We also use the evo library [15] for camera pose evaluation with scene alignment and scale alignment. Finally, we use the torchmetrics [46] implementation of LPIPS [88] to calculate the perceptual similarity between images.

5.2. Model Implementations

Our approach uses pre-trained models for initial camera prediction [79] and multi-view point tracking [28] in sparse mapping and pose refinement. We provide the details of these models based on the original papers [79, 28]. The initial camera predictor takes ResNet50 [18] features from a DINOv2 [49] backbone that are passed to a self-attention layer and a cross-attention layer for 8 iterations. Both attention layers have 8 attention heads. The main loop then refines these encodings using another self-attention layer before passing them to a multi-level perceptron that outputs pose encoding updates. This loop is run for 4 iterations to get refined pose estimates.

We also use a pre-trained multi-view point tracker [27, 28], described in Section 3.2, for our sparse mapping and camera refinement. This point tracker takes a set of frames $t = 1, \ldots, T$ along with a set of query points $q \in \mathbb{R}^2$ in a query frame t_q and outputs the locations of these points in all other frames. All

frames are first downsampled by a factor $k = 4$ for efficiency and passed to a 2D convolutional encoder that outputs a set of feature maps Φ_t^s for each frame at $S = 4$ decreasing scales with $s = 1, \ldots, S$. Next, for each point, a square neighborhood of feature vectors is extracted as:

$$\phi_t^s = \left[\Phi_t^s\left(\frac{x}{ks} + \delta, \frac{y}{ks} + \delta\right) : \delta \in \mathbb{Z}, \|\delta\|_\infty \leq \Delta\right] \in \mathbb{R}^{d \times (2\Delta+1)^2}, \quad s = 1, \ldots, S, \quad (1)$$

where the feature map Φ_t^s at scale s is sampled using bilinear interpolation around the point (x_t, y_t). A 4D correlation can then be defined as:

$$\langle \phi_{tq}^s, \phi_t^s \rangle = \text{stack}((\phi_{tq}^s)^T \phi_t^s) \in \mathbb{R}^{(2\Delta+1)^4}$$

for every scale $s = 1, \ldots, S$. To reduce the dimensionality of the features, we project them using an MLP as:

$$\text{Corr}_t = (\text{MLP}(\langle \phi_{tq}^1, \phi_t^1 \rangle), \ldots, \text{MLP}(\langle \phi_{tq}^S, \phi_t^S \rangle)) \in \mathbb{R}^{p \cdot S}$$

These correlation features are then combined with their positional embedding into tokens and passed to a transformer that outputs the position updates. This transformer consists of 12 self-attention layers and 12 cross-attention layers with 8 heads each. This transformer is run for 10 iterations.

5.3. Hyperparameters

Our camera pose estimation module and our dense Gaussian mapper include various hyperparameters that influence the performance. We selected these parameters based on our testing or based on previous work if we believed that the setting was equivalent.

Our camera pose estimation module optimizes the camera poses of incoming frames and adds them to a keyframe widow of size N_f until the window becomes full. We set $N_f = 8$ for the synthetic Replica dataset [68] and $N_f = 6$ for the real-world TUM dataset [69] because it has a lower frame rate. This balances training speed with camera pose accuracy in scenes with large camera translation or rotation. This is because larger frame windows fill up slower, which results in faster optimization due to fewer sparse map densification steps. However, larger frame windows also increase the likelihood of low overlap between the window frames due to the movement of the camera causing parts of the scene visible in the early frames to come out of view in later frames, especially in parts of the scene with large amounts of camera rotation. This limits the number of visible image correspondences between the first and last frames of the window, which increases the effect of outliers and decreases camera pose accuracy.

Our initialization triangulates a set of point clouds from a set of pairs of query and non-query frames. We calculate the triangulation angle between the camera poses of the two views observing each point and filter out points with a triangulation angle below a threshold t_t to decrease sensitivity to small errors in the 2D observations. We start by setting this threshold to 16 degrees and then we divide the threshold by 2 until at least one point cloud contains more than 200 points that were not filtered. The resulting threshold is then used to filter the point cloud and to select a point cloud with the highest number of inliers.

Throughout optimization, we also filter observations that have a reprojection error greater than a threshold t_{rpo} as these observations are inconsistent with the scene. Our tests have shown that observations with a reprojection error below 1 for synthetic datasets and below 4 for real-world datasets work well. This is because of the larger error in our correspondences in the real-world dataset with a lower threshold resulting in an insufficient number of observations.

For our camera pose estimation module, we use two parameters to reject the frames from being selected as keyframes. The first parameter is the motion blur threshold t_b for new incoming frames, as we have found that our point tracker struggles with blurry frames. Our testing has found that a threshold t_b of 500 for real-world scenes works best. We do not check for the threshold in synthetic scenes as these do not contain motion blur. The second parameter is the distance threshold t_d, which prevents low-motion frames from being used as keyframes. This is due to the instability of the predicted tracks with frames that have low motion. Our testing has shown that a threshold t_d of 6 prevents unstable tracks without affecting performance in other parts of the scene.

Furthermore, we have three keyframe selection thresholds. The first is the ratio t_o between visible and non-visible track points. Our tests have shown that having an initial threshold that decreases by t_{od}

for every keyframe added to the window works better. This initial threshold is reset after every sparse map densification. The initial thresholds t_o that performed best in our tests were 0.8 for synthetic scenes and 0.9 for real-world scenes. The decrements t_{od} that worked the best were 0.01 for synthetic scenes and 0.02 for real-world scenes. Next, we check if the reprojection error is greater than a threshold t_{rp}. We have found that the reprojection error threshold also performs better when increased by t_{rpi} for each keyframe added to the window and reset after every sparse map densification. Our tests have shown that an initial threshold of 0.4 works best for synthetic scenes, while 2.0 works best for real-world scenes. The increment t_{rpi} of 0.01 worked well for both scene types. Finally, we check if we went more than t_f frames without a keyframe, to retain the accuracy of non-keyframe camera poses. Our tests have shown that a threshold of 6 for both scene types works best.

If the mean reprojection error becomes greater than a threshold t_{rpr}, we restart keyframe selection and set all new frames as keyframes. Our testing has shown that a threshold $t_{rpr} = 6$ works well for both synthetic and real-world scenes.

Finally, for dense Gaussian Splatting mapping, we use different learning rates for position, feature (color), opacity, scaling, and rotation parameters chosen based on our testing. This accounts for the sensitivity to changes of each parameter. These are $1.6e^{-6}$ for position, 0.0025 for feature, 0.05 for opacity, 0.005 for scaling, and 0.001 for rotation. For DSSIM, we use a λ of 0.2 following previous work [38]. We down-sample the depth map before projecting by a down-sampling rate r_d of 10 for the higher-resolution Replica dataset and 5 for the lower resolution TUM dataset. We prune Gaussians every 200 iterations based on their extent and opacity. We prune large Gaussians with an extent that is greater than a prune threshold (mean depth * 2.5). We also prune transparent Gaussians that have an opacity of less than 0.005.

6 Results

In this chapter, we first provide an overview of the Replica [68] and TUM [69] datasets used to evaluate the performance of different methods in the scope of our evaluation. Next, we discuss the metrics used to evaluate the tracking performance and view synthesis performance of our method. Next, we provide a qualitative and quantitative evaluation of our method's tracking and view synthesis performance. This is followed by an evaluation of our method's speed and GPU memory usage. Finally, we provide an ablation study of our point tracking module.

6.1. Datasets

Following prior work, we test our approach on two popular indoor scene datasets. The Replica dataset [68] consists of 8 rendered indoor scenes. These are 5 office scenes and 3 living room scenes. Each scene consists of 2000 consecutive frames from a rendered video with their associated ground truth depth maps and poses. Each frame has a resolution of 1200 by 680 pixels. The advantage of this dataset is that the images, depth maps, and camera poses are of high quality, which allows for testing the approaches in a theoretical best-case scenario. The disadvantage is that the results might not be representative for real-world scenarios. This is mainly due to the fact that the frames do not exhibit motion blur or lighting exposure effects which would be present on a real-world dataset captured using a physical camera.

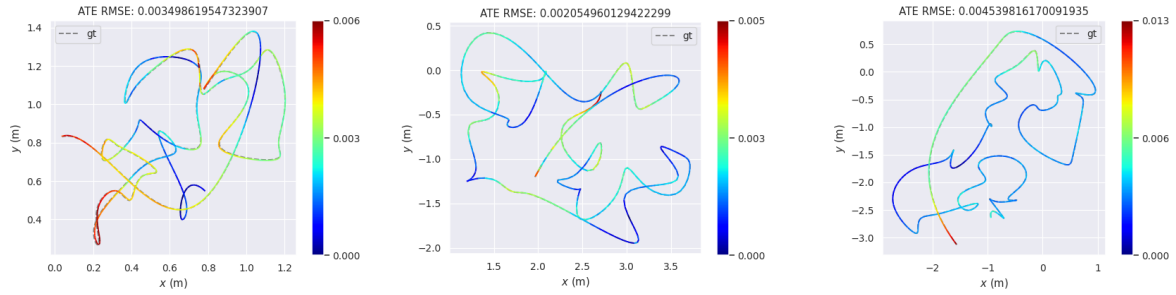
As a second dataset we use the TUM-RGBD dataset [69] which consists of 5 real scenes captured using a handheld camera with a depth sensor. These scenes range from 591 to 3512 frames with their associated captured ground truth depth maps and camera poses. Each frame has a resolution of 640 by 480 pixels. This is a widely used dataset due to its ability to test the performance on difficult real-world scenarios. The disadvantage of this dataset is that the images are of low resolution and can include blur. In addition, the ground truth depth maps and camera poses might include errors. This might mean that the results are worse than they would be on data captured with higher-quality devices.

6.2. Metrics

Following prior work, we measure various metrics to evaluate and compare the performance of our pose estimation and reconstruction with state-of-the-art methods. For evaluating pose estimation, we measure the absolute trajectory error (ATE) by calculating the root mean square error (RMSE) between our predicted poses and the given ground truth poses. The RMSE is calculated by first getting the squared difference between the ground truth camera translation and the estimated camera translation of every frame. These are then summed with the RMSE given as the square root of this sum according to:

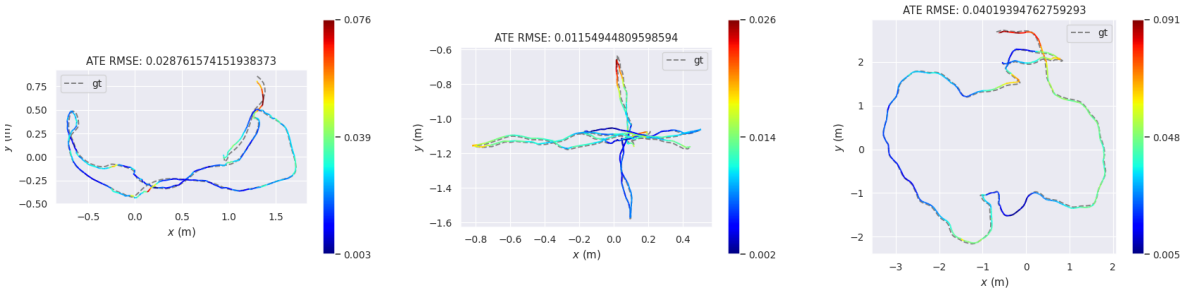
$$\mathrm{RMSE}(\hat{\theta}) = \sqrt{\mathrm{MSE}(\hat{\theta})} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left((\hat{\theta}-\theta)^2\right)}$$

where θ is the ground truth camera translation and $\hat{\theta}$ is the predicted camera translation. However, these camera poses cannot be compared directly due to the lack of ground truth depth maps and



(a) Office 1 (b) Room 2 (c) Office 3

Figure 6.1: Visualization of our predicted camera positions compared to the ground truth camera position on 3 scenes from the synthetic Replica [68] dataset. The relative mean squared error (RMSE) is reported in meters with lower values being closer to the ground truth.



(a) f1 Desk (b) f2 xyz (c) f3 Long office

Figure 6.2: Visualization of our predicted camera positions compared to the ground truth camera positions on 3 scenes from the synthetic TUM [69] dataset. The relative mean squared error (RMSE) is reported in meters with lower values being closer to the ground truth.

poses during training in the RGB setting. This results in a mismatch of scene scale and misalignment of coordinate frames between the ground truth and predicted scene. Therefore, the estimated poses need to first get aligned and scaled to the ground truth poses before calculating the RMSE. For this metric, lower values indicate a lower error.

For evaluating view synthesis performance, we use three different metrics, namely the peak signal-to-noise ratio (PSNR), structural similarity index measure (SSIM), and learned perceptual image patch similarity (LPIPS). This allows us to compare both the direct per-pixel error and the perceived error that is closer to human perception.

Peak signal-to-noise ratio (PSNR) is the ratio between the power of the ground truth signal and the power of the corrupting noise calculated as:

$$\mathrm{MSE}(\hat{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left((\hat{\theta}-\theta)^2\right)$$

Following general practice, PSNR tends to be expressed as a logarithmic quantity in decibels calculated as:

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

where MSE is the mean squared error and MAX is the maximum value of a pixel in our case being 1. For this metric, higher values indicate a better reconstruction.

The structural similarity index measure (SSIM) measures perceptual similarity between the ground truth and the rendered image. SSIM makes use of perceptual phenomena to better approximate how a human would perceive the difference between images. This is done by extracting information from

Method	R0	R1	R2	O0	O1	O2	O3	O4	Avg
NICER-SLAM [91]	1.36	1.60	1.14	2.12	3.23	2.12	1.42	2.01	1.88
MonoGS [38]	-	-	-	-	-	-	-	-	14.54
Photo-SLAM [22]	0.35	1.18	0.23	0.58	0.32	5.03	0.47	0.58	1.09
GlORIE-SLAM [87]	0.31	0.37	**0.20**	0.29	**0.28**	0.45	0.45	0.44	0.35
Splat-SLAM [59]	**0.29** 0.29	0.38 **0.33**	0.24 0.25	**0.27** 0.29	0.35 0.35	**0.34** 0.34	**0.42** 0.42	**0.43** 0.43	0.34 0.34
DROID-Splat [21]	-	-	-	-	-	-	-	-	**0.27**
Ours	0.49 0.57	0.58 0.63	0.21 0.26	0.44 0.49	0.35 0.36	0.41 0.56	0.44 0.62	0.68 0.9	0.46 0.55

Table 6.1: Comparison of the pose estimation performance achieved by different SLAM methods on the Replica [68] dataset. We report the relative mean squared error (RMSE) in centimeters for better readability. Best results are highlighted as first, second, and third. Additionally, best results are highlighted in **bold**. For our method and Splat-SLAM [59], we report both the RMSE of only keyframes in the top row and of all frames in the bottom row and highlight based on the better result. The results of all methods are taken from [59, 21].

spatially close pixels in both images to better evaluate structural similarity. SSIM also uses luminance masking in bright regions and contrast masking in regions with high texture due to distortions becoming more difficult to perceive in these regions. The simplified variant for SSIM is:

$$\mathrm{SSM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_1)}$$

where μ_i and σ_i are the mean and variance of the pixel sample from a 2D convolution and c_i works to stabilize the division for a weak denominator. Following previous work [16], we set c_1 to 0.01^2, c_2 to 0.03^2, and the window size of the convolution to 11. The left side of the denominator handles luminance masking while the right side handles contrast masking. The method outputs a float scalar value that goes from -1 to 1 with 1 indicating perfect similarity, 0 indicating no similarity, and -1 being perfect anti-similarity.

Learned Perceptual Image Patch Similarity (LPIPS) [88] calculates the perceptual similarity between two images using a multilayer neural network to encode both the ground truth and the rendered image. The loss is then calculated between the patches that are output by the network layers. This approach tends to be better aligned with human perception. The original LPIPS can use AlexNet [34], VGG [64], or SqueezeNet [23] as the backbone. Following other methods [87, 59, 21], we use the AlexNet backbone. The output is a floating point scalar value that is generally between 0 and 1 where 0 means perfect perceptual similarity and higher values indicate higher perceptual dissimilarity.

6.3. Experiments

In this section, we present the results of our experiments and compare them with other RGB SLAM methods. We first show the qualitative and quantitative evaluation of the tracking performance on the Replica [68] and TUM [69] datasets. Then we show the qualitative and quantitative results on the Replica and TUM datasets for absolute and perceived rendering quality. Finally, we show the quantitative evaluation of the training speed and GPU memory usage of our method compared to others.

6.3.1. Tracking Results

First we discuss the qualitative results of our approach. We conduct this by examining graphs which show our predicted camera poses aligned to the ground truth camera poses on 3 scenes from the Replica and TUM datasets. The root mean square error (RMSE) is reported in meters. These graphs can help us examine the failure states of our approach and evaluate its limitations.

Method	f1/desk	f2/xyz	f3/off	Avg.-3	f1/desk2	f1/room	Avg
MonoGS [38]	3.8	5.2	2.9	4.0	75.7	76.6	32.8
Photo-SLAM [22]	**1.5**	1.0	**1.3**	1.3	–	–	–
GlORIE-SLAM [87]	1.6	**0.2**	1.4	**1.1**	2.8	4.2	2.1
Splat-SLAM [59]	1.65 1.92	0.22 0.23	1.44 1.41	**1.1** 1.19	2.79 3.05	4.16 4.43	2.05 2.21
DROID-Splat [21]	1.6	**0.2**	1.7	1.7	**2.3**	**3.3**	**1.8**
Ours	2.88 3.18	1.05 1.21	4.0 5.9	2.64 3.43	13.81 14.45	39.61 39.54	12.27 12.86

Table 6.2: Comparison of the pose estimation performance achieved by different SLAM methods on the TUM [69] dataset. We report the relative mean squared error (RMSE) in centimeters for better readability. Best results are highlighted as first, second, and third. Additionally, best results are highlighted in **bold**. For our method and Splat-SLAM [59], we report both the RMSE of only keyframes in the top row and of all frames in the bottom row and highlight based on the better result. The results of all methods are taken from [59, 21].

Figure 6.1 shows the plots of our method compared to ground truth on three scenes from the Replica dataset. Our approach performs consistently in most parts of the scene. We see an increased error in parts of the scene with camera movement in the viewing direction. These tend to appear at the end of the scene, such as the middle left in Figure 6.1a, middle of Figure 6.1b, or the bottom left in Figure 6.1c. We have found that this can result in degenerate point tracks that degrade the pose refinement and triangulation of our method. We have also found that some earlier parts of the scene do not have large errors until the end of the scene video, when their error suddenly increases, as in the bottom left of Figure 6.1a. We believe that this is caused by the alignment of the scene to ground truth poses made prior to the calculation of the RMSE described in Section 6.2. In the process of minimizing the error in latter parts of the scene through this alignment, some earlier parts of the scene have their error increased. This is especially prevalent in parts of the scene with fast camera movement which tends to result in fewer keyframes in that section of the scene. These sections then have a smaller effect on the total error, which results in them being misaligned.

Figure 6.2 shows the plots of our method compared to the ground truth reference on three scenes from the TUM dataset. The method tends to perform consistently in most parts of the scene, although the RMSE is lower than on the Replica scenes. The performance is mainly worse around the start of the scene. We believe that this is caused by our window-based optimization and local bundle adjustment, which can sometimes incorrectly change the scale of the scene based on incorrect point tracks. This scene scale is then mostly aligned by our global adjustment, however slight scale changes can remain if the point tracks become too misaligned. These scale differences can then accumulate, resulting in the early parts of the scene getting misaligned compared to the later parts of the scene. This occurs in the real-world scenes that are more blurry or noisy compared to the synthetic scenes. This can result in degenerate point tracks that degrade pose refinement and triangulation, which makes local and global bundle adjustment difficult.

In the scope of the quantitative evaluation, we present the Relative Mean Squared Error (RMSE) in centimeters between our predicted camera poses and the ground truth poses. For this we use the evo library [15] which first aligns and scales the scene to the ground truth scene scale before calculating the RMSE. This is done because it is impossible to accurately estimate the ground truth scale without ground truth camera poses or depth maps. We then compare these RMSE values with various state-of-the-art methods on the Replica and TUM datasets. We present the RMSE for keyframes and all frames following Splat-SLAM [59]. The error values for keyframes are lower due to the additional optimization in the sparse densification step of our method.

Table 6.1 shows the root mean square error (RMSE) of our method on the synthetic Replica dataset [68] compared to the results of other SLAM methods. We can see that our method is comparable to state-of-the-art methods [21, 59], achieving top-three performance on all but two of the scenes. Of

Figure 6.3: Rendering results on 3 scenes from the Replica [68] dataset. The first row shows rendered images from our method while the second row shows the ground truth images.

these, our method achieves the second best result on four out of the eight scenes. Furthermore, our method is within 0.2 millimeters of the best method on the room 2 and office 3 scenes, while still staying within 1 millimeter on the office 1 and office 2 scenes. For the remaining four scenes, our method still remains within 2 millimeters of the best result. Our method seems to struggle on the Office 4 scene, which we believe is because the scene tests handling of a transparent surface using a large glass wall that is particularly difficult for our method. Other methods seem to handle these conditions better than our method, although Splat-SLAM [59] also achieves its worst result on this scene. Finally, the average performance of our method only falls behind the concurrent state-of-the-art DROID-Splat [21] by 2 millimeters while outperforming the older Photo-SLAM [22] and NICER-SLAM [91] methods. The authors of DROID-Splat [21] and MonoGS [38] do not report their per-scene performance so we were unable to fully compare the performance of our method against their methods.

Table 6.2 shows that our method is behind the state-of-the-art methods on the real-world TUM dataset [69]. This is especially visible on the desk 2 and room scenes that are often skipped by older methods due to their difficulty. We believe that the worse performance on the TUM dataset is mainly caused by our multi-view tracker not being trained on real-world data sequences. This can result in degenerate tracks especially in parts of the scene with camera specific visual artifacts. The main artifact with which our method seems to struggle is the motion blur caused by fast camera motion. This seems to be quite common in this dataset, especially in the desk 2 and room scenes. Another visual artifact is the change in lighting conditions due to camera exposure.

Although these degenerate tracks have limited impact on individual frame camera pose accuracy and sparse scene quality, their effects compound over longer sequences, which leads to substantial error accumulation. Other methods solve error accumulation through continuous global bundle adjustment. In contrast, our method only performs global bundle adjustment periodically, allowing errors to accumulate between adjustment steps. Furthermore, the addition of a loop closure module, present in the concurrent methods, would also improve the handling of error accumulation. When compared to the older methods, Table 6.2 shows that our method performs better than MonoGS on all scenes except the office scene and is within 5 millimeters of Photo-SLAM on the xyz scene.

6.3.2. Mapping Results

First, we discuss the qualitative results of our mapping and rendering quality by examining renderings from our method compared to ground truth images on three scenes from the synthetic Replica [68] and real-world TUM [69] datasets. These renderings can be useful for examining parts of the scene that are difficult to represent using our method and parts of the scene where our method performs well.

Figure 6.3 shows that our approach can generalize across various synthetic scenes. This includes the representation of lighting and different materials in the scene. The two room scenes have difficult lighting conditions that can cause problems for older methods. Our method performs well on these scenes because of the Gaussian representation's ability to represent different surface and subsurface material properties. The room scenes also contain blinds which tend to present a difficult optimization

Figure 6.4: Rendering results on 3 scenes from the TUM [69] dataset. The first row shows rendered images from our method while the second row shows the ground truth images.

problem for older NeRF-based approaches [70, 91, 58]. Our method is able to represent these in high quality due to the higher-resolution explicit Gaussian representation. Furthermore, the office scenes tend to include large regions with homogeneous appearance, especially on walls or tables. NeRF-based approaches [70, 91, 58] must evaluate their neural scene representation multiple times along each camera ray even in these visually uniform areas. In contrast, our approach can represent these visually uniform areas with a sparse set of anisotropic Gaussians. This improves GPU memory usage and rendering speed while retaining high rendering quality.

Figure 6.4 shows that our approach struggles on the real-world TUM dataset [69]. This seems to be mostly related to the accuracy of our pose estimates and our sparse scene representation. We can also see that our approach struggles on parts of the scene with large numbers of objects. This is most likely caused by imperfect depth information coming from our tracking module that is related to the quality of our camera poses. The reconstruction seems to especially struggle in areas with large amounts of motion blur due to the effect it has on our tracker and the general difficulty of using blurry images for accurate mapping.

Next, we examine the quantitative rendering results of our approach on 8 scenes from the Replica dataset and 5 scenes from the TUM dataset. For this purpose, we use the Peak Signal-to-Noise Ratio (PSNR), which measures absolute pixel-wise error, with higher values being better. The second metric that we use is the Structural Similarity Index Measure (SSIM), which evaluates perceptual and structural similarity, with higher values being better. The final metric that we use is the Learned Perceptual Image Patch Similarity (LPIPS), which uses a neural network to encode the images and measure their similarity. For this metric, lower values are better. These metrics allow us to compare both the direct per-pixel error and the perceived error that is closer to human perception for a better understanding of our method's performance.

Table 6.3 presents the measured rendering metrics on 8 scenes from the Replica dataset [68] for our method compared to various state-of-the-art methods. Our method is able to outperform the NeRF-based NICER-SLAM [91] and GIORIE-SLAM [87], and the older 3DGS-based methods of MonoGS [38] and Photo-SLAM [22]. We can also see that the other concurrent method of DROID-Splat [21] has the best average performance. However, because they do not report their per-scene performance, we cannot fully compare our method's performance beyond the average metrics for the whole dataset.

The only method that seems to outperform our per-scene performance is Splat-SLAM [59] which was developed concurrently. We are still able to reach higher PSNR and SSIM values compared to Splat-SLAM [59] on 4 out of the 8 scenes and lower LPIPS on 2 scenes. This is in spite of our method having worse pose estimation on some of these scenes. Our method mainly falls behind on the Office 4 scene due to worse pose estimates of our tracking front end. However, our results are comparable

Method	Metric	R0	R1	R2	O0	O1	O2	O3	O4	Avg
NICER-SLAM [91]	PSNR↑	25.33	23.92	26.12	28.54	25.86	21.95	26.13	25.47	25.41
	SSIM↑	0.75	0.77	0.83	0.87	0.85	0.82	0.86	0.87	0.83
	LPIPS↓	0.25	0.27	0.18	0.17	0.18	0.20	0.16	0.18	0.19
MonoGS [38]	PSNR↑	-	-	-	-	-	-	-	-	31.22
	SSIM↑	-	-	-	-	-	-	-	-	0.91
	LPIPS↓	-	-	-	-	-	-	-	-	0.21
Photo-SLAM [22]	PSNR↑	29.77	31.30	33.18	36.99	37.59	31.79	31.62	34.17	33.30
	SSIM↑	0.87	0.91	0.93	0.96	0.95	0.93	0.92	0.94	0.93
	LPIPS↓	0.11	0.08	0.07	0.06	0.06	0.09	0.09	0.07	0.07
GIORIE-SLAM [87]	PSNR↑	28.49	30.09	29.98	35.88	37.15	28.45	28.54	29.73	31.04
	SSIM↑	**0.96**	**0.97**	0.96	**0.98**	**0.99**	**0.97**	**0.97**	0.97	0.97
	LPIPS↓	0.13	0.13	0.14	0.09	0.08	0.15	0.11	0.15	0.12
Splat-SLAM [59]	PSNR↑	32.25	**34.31**	35.95	**40.81**	40.64	**35.19**	**35.03**	**37.40**	36.45
	SSIM↑	0.91	0.93	0.95	**0.98**	0.97	0.96	0.95	**0.98**	0.95
	LPIPS↓	0.10	0.09	**0.06**	**0.05**	**0.05**	**0.07**	**0.06**	**0.04**	0.06
DROID-Splat [21]	PSNR↑	-	-	-	-	-	-	-	-	**39.66**
	SSIM↑	-	-	-	-	-	-	-	-	**1.0**
	LPIPS↓	-	-	-	-	-	-	-	-	**0.03**
Ours	PSNR↑	**33.29**	33.61	**37.1**	39.08	**41.61**	33.94	34.76	32.98	35.8
	SSIM↑	0.94	0.94	**0.97**	0.97	0.98	0.95	0.94	0.94	0.95
	LPIPS↓	**0.07**	**0.07**	0.07	0.07	**0.05**	0.1	0.1	0.12	0.08

Table 6.3: Rendering performance on the Replica [68] dataset. We present the PSNR (decibels), SSIM, and LPIPS as described in section 6.2 where arrows indicate whether lower or higher values are better. Numbers are taken from the respective papers. Our method performs comparably to state-of-the-art approaches across various metrics. Best results are highlighted as first, second, and third. Additionally, best results are highlighted in **bold**

on the 3 remaining scenes.

Table 6.4 presents the quantitative rendering results on the TUM dataset [69]. As we can see, our method is once again comparable to the NeRF-based GIORIE-SLAM [87], and the older 3DGS-based MonoGS [38] and Photo-SLAM [22] on the f1/desk, f2/office, and f1/room. This is in spite of our method's worse camera pose estimation when compared to GIORIE-SLAM [87] and Photo-SLAM [22].

Our method seems to perform much worse than GIORIE-SLAM [87] and Photo-SLAM [22] on the xyz scene although we still achieve a better PSNR value than MonoGS [38]. We believe that this is because unlike other scenes that contain both camera rotation and translation, the xyz scene contains very little rotation, with most of the scene consisting of camera translation in the viewing direction. As mentioned above, camera movement with low rotation results in the triangulation being highly sensitive to noise, with small changes to the 2D observations leading to large errors in the 3D point. This seems to especially affect our global bundle adjustment, which results in frequent scene scale adjustments. These adjustments to the camera poses and the sparse scene then led to the pruning of older Gaussians because their size and distance to camera became inconsistent with the optimized sparse scene. Our method performs the worst on the desk 2 scene. We believe that this is caused by the above mentioned movement in the viewing direction toward the end of the scene along with inaccurate camera poses and triangulated scene caused by difficult camera artifacts. These issues result in scene scale alignment, which causes the pruning of older Gaussians, negatively affecting rendering performance.

When it comes to the two concurrently developed state-of-the-art methods Splat-SLAM [59] and DROID-Splat [21], we can see that our method gets outperformed on all scenes. This seems to be due to their significantly better pose estimates, as seen in Table 6.2, which would also result in higher-quality depth information from their trackers.

Overall, we can see that our method performs comparably to the other methods on the synthetic Replica dataset [68]. This is especially the case for rendering, where our method manages to out-

Method	Metric	f1/desk	f2/xyz	f3/off	f1/desk2	f1/room	Avg.
Photo-SLAM [22]	PSNR↑	20.97	21.07	19.59	–	–	–
	SSIM↑	0.74	0.73	0.69	–	–	–
	LPIPS↓	0.23	0.17	0.24	–	–	–
MonoGS [38]	PSNR↑	19.67	16.17	20.63	19.16	18.41	18.81
	SSIM↑	0.73	0.72	0.77	0.66	0.64	0.70
	LPIPS↓	0.33	0.31	0.34	0.48	0.51	0.39
GlORIE-SLAM [87]	PSNR↑	20.26	25.62	21.21	19.09	18.78	20.99
	SSIM↑	0.79	0.72	0.72	0.80	0.78	0.77
	LPIPS↓	0.31	0.09	0.32	0.38	0.38	0.30
Splat-SLAM [59]	PSNR↑	25.61	**29.53**	26.05	23.98	24.06	25.85
	SSIM↑	0.84	0.90	0.84	0.81	0.80	0.84
	LPIPS↓	0.18	0.08	0.20	0.23	0.24	0.19
DROID-SPLAT [21]	PSNR↑	**26.42**	28.08	**27.84**	**25.21**	**25.11**	**26.53**
	SSIM↑	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**
	LPIPS↓	**0.12**	**0.08**	**0.11**	**0.17**	**0.18**	**0.13**
Ours	PSNR↑	20.42	18.92	20.93	17.44	18.66	19.27
	SSIM↑	0.73	0.64	0.74	0.64	0.67	0.72
	LPIPS↓	0.35	0.45	0.38	0.52	0.43	0.43

Table 6.4: Rendering performance on TUM-RGBD [69]. Numbers are taken from the respective papers. We present the PSNR (decibels), SSIM, and LPIPS as described in section 6.2 where arrows indicate whether lower or higher values are better. Best results are highlighted as first, second, and third. Additionally, best results are highlighted in **bold**

perform one of the concurrent state-of-the-art methods. However, our method still struggles on the real-world TUM dataset [69] in both camera pose estimation and rendering quality. This is mainly due to the performance of our point tracker on real-world images. We will provide more information on the limitations of our method in the next chapter.

	SplaTAM [29]	**GIORIE-SLAM** [87]	**MonoGS** [38]	**Splat-SLAM** [59]	**Ours**
GPU Usage [GiB] ↓	18.54	15.22	14.62	17.57	6.21
Avg. FPS ↑	0.14	0.23	0.32	1.24	0.75

Table 6.5: Comparison of SLAM methods on the room 0 scene of the Replica dataset. We present the GPU memory usage in GiB and average frames per second. The numbers are taken from [59].

6.3.3. Memory and Runtime

Table 6.5 shows the maximum GPU memory used during optimization and the average frames per second (FPS) of our method compared to the state-of-the-art methods on the room 0 scene from the Replica dataset. Due to our tracker only tracking a lower number of points between frames, our method has a relatively low GPU memory usage of just 6.2 GiB. This is much lower than all the other methods, such as GIORIE-SLAM [87] or Splat-SLAM [59], which require 15.2 GiB and 17.6 GiB of GPU memory respectively. This is due to the larger complexity of their modules for estimating camera poses and full depth maps. The older methods of SplaTAM [29] and MonoGS [38] also require high GPU memory usage of 18.5 GiB and 14.6 GiB respectively. This is due to their much weaker pruning approaches and relying on lower-quality poses which can result in unnecessary Gaussians in the dense map. This makes our method more suitable for portable settings with limited GPU memory.

We also show that while our method performs slower than Splat-SLAM, it still performs faster than the older methods. We believe that this is also due to our tracker only optimizing over a lower number of points with depth maps being aligned to the scene instead of being optimized directly from the correspondences. The 3D Gaussian splatting is very fast and therefore the tracker is the only bottleneck.

The speed of our tracker depends on the number of keyframes, which depends on the degree of camera movement between consecutive frames. We found that in scenes with low camera movement, our tracker can perform at over 3 FPS. We believe that this makes our method useful for real-time settings.

Method	f1/desk	f2/xyz	f3/off	f1/desk2	f1/room
CoTracker1 [27]	3.78	1.54	4.55	33.82	68.51
CoTracker3 [28]	2.88	1.05	4.0	13.81	39.61

Table 6.6: Comparison of our method with different versions of our multi-view tracker on the TUM dataset. Both versions of the tracker are using the blur filter evaluated in Table 6.7.

Method	f1/desk	f2/xyz	f3/off	f1/desk2	f1/room
Our w/o Blur Filter	3.18	1.19	4.18	24.11	48.14
Our w/ Blur Filter	2.88	1.05	4.0	13.81	39.61

Table 6.7: Comparison of our method with and without the motion blur frame filter on the TUM dataset. For this experiment, we used the CoTracker3 [28] multi-view point tracker evaluated in Table 6.6.

6.3.4. Ablation Studies

As explained in Section 4.1, our front-end tracker approach relies on image correspondences from a pre-trained multi-view point tracker [28] for camera pose estimation and sparse scene triangulation. This pre-trained point tracker has two versions that differ in model architecture and training strategy, details of which are provided in Section 3.2. The older CoTracker1 [27] was trained only on synthetic image sequences with ground truth point tracks. In contrast, the newer CoTracker3 [28] expanded the training strategy by incorporating real-world image sequences with pseudo-labels. These pseudo-labels were produced by averaging the unreliable predictions of various multi-view trackers and multiple versions of CoTracker. Although these pseudo-labels were more reliable than the original predictions, they still contained errors. The difficulty of multi-view point tracking is especially present on frame sequences that contain camera artifacts, such as motion blur or light exposure variations.

Table 6.6 shows the ATE RMSE camera pose estimation results of our method, comparing the two versions of CoTracker, with both versions using our blur filter evaluated below. As we can see, the newer CoTracker3 performs much better on all scenes compared to the older CoTracker1. This is especially the case on the difficult desk2 and room scenes, where CoTracker3 achieves an improvement of 38 and 29 centimeters respectively. Furthermore, we show that CoTracker3 achieved an improvement of 0.5 centimeters on the xyz and office scenes and 0.9 centimeters on the desk1 scene. This shows the effects of the expanded training strategy on performance in real-world scenarios. However, as mentioned before, this expanded training strategy still encounters errors on particularly difficult image sequences.

In an attempt to limit the remaining effects of these blurry frames, our method rejects frames that are too blurry from becoming keyframes. We explain the details of this in Section 4.1.2. We present an evaluation of this blurry frame rejection in Table 6.7, reporting the ATE RMSE camera pose evaluation on the real-world TUM dataset using our model with the CoTracker3 [28] multi-view point tracker evaluated above. The table shows that adding the filter significantly improves performance on the most difficult desk2 and room scenes with improvements of 10 and 9 centimeters respectively. The improvement is less pronounced on the xyz and office scenes at 1.4 and 1.8 millimeters respectively. We believe that this smaller difference is due to these two scenes consisting of a large number of frames with most frames containing limited motion blur. Finally, the desk1 scene also contains few sections with significant motion blur, resulting in our blur filter only improving performance by 3 millimeters. These ablation studies show the difficulties of predicting image correspondences on real-world image sequences with large amounts of camera artifacts. Furthermore, we present our handling of these artifacts and the improvements our handling provides for performance on real-world image sequences.

7

Limitations and Future Work

In this chapter, we discuss the limitations of our system and the limitations of the individual components. For this, we focus on the multi-view point tracker and its failure states while examining practical examples. Next, we discuss future improvements on our work that could solve some of these limitations while improving pose accuracy, rendering quality, and speed.

7.1. Limitations

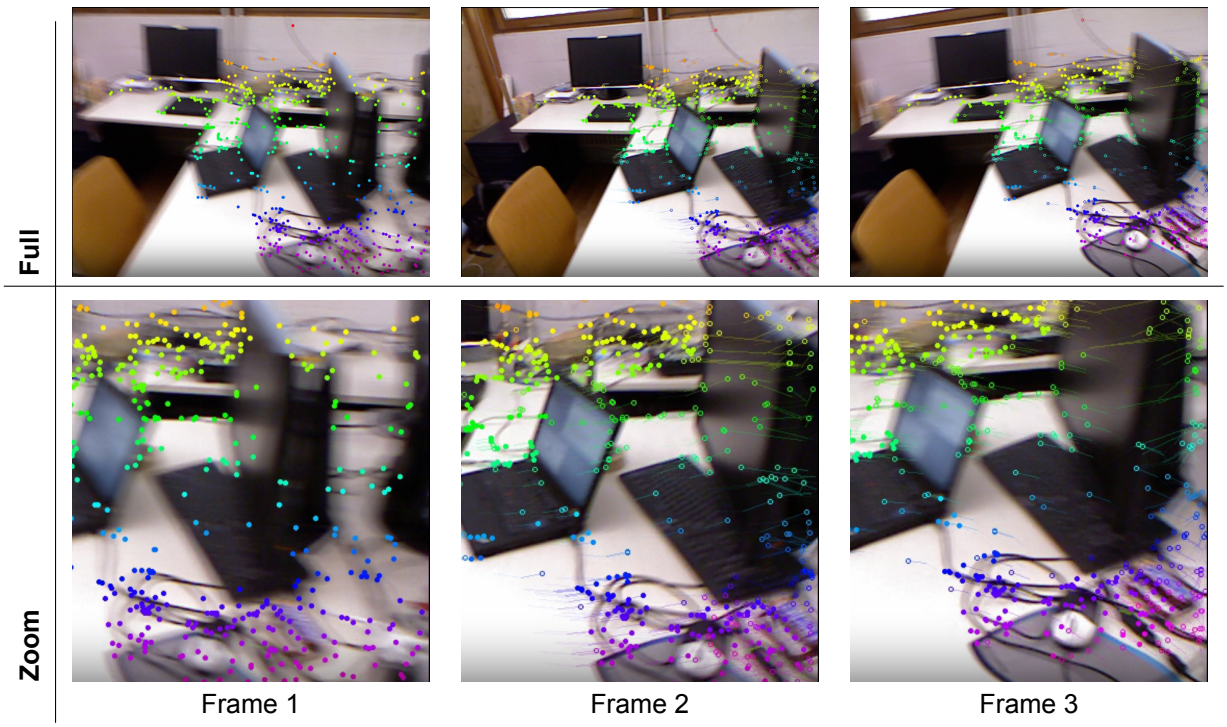


Figure 7.1: Illustration of the failure states of our pre-trained multi-view point tracker [28] on three frames from the desk 1 scene of the real-world TUM dataset [69]. The top row shows the full three frames while the bottom row shows a close-up look of the bottom right section of each frame. The circles in these frames represent tracked points with empty circles indicating that the corresponding point was filtered out.

As discussed in Section 3.2, our current multi-view point tracker was mainly trained on synthetic data. This is due to the difficulty of acquiring real-world frame sequences with accurate multi-view point tracks. As a result, we have found that the tracker struggles on real-world data. This is especially visible in parts of the scene with fast rotation or translation that result in large amounts of motion blur. To remedy this, CoTracker3 [28] uses pseudo-labels on real-world videos that are output by a collection of point trackers and versions of itself. Although this somewhat improves the accuracy of these tracks,

we still found that the tracker struggles in difficult parts of the scene, which can often result in error accumulation that impacts camera pose estimation and sparse scene point triangulation.

Figure 7.1 shows practical examples of the failure states of CoTracker3 [28] on three frames from the desk scene of the real-world TUM dataset [69]. The top row shows the full three frames while the bottom row shows a close-up look of the bottom right section of each frame. The first frame is the query frame with the points in this frame being tracked across the two other frames. The points in each frame are represented by colored circles that share the same color between frames. Empty circles correspond to points that were filtered out by our filtering approach. These frames contain large amounts of motion blur caused by a right-to-left camera motion.

The close-up view of frame 1 shows that there are approximately 10 cyan and light-blue query points at the bottom of the monitor stand. However, when we look at frame 2, we can see that the number of points at the bottom of the stand suddenly increased to almost 20. These additional points correspond to the query points on the right side of frame 1 that should have left out of view due to the camera motion. The point tracker instead seems to have incorrectly matched these points to the bottom of the monitor stand. A similar issue appears across the right side of the frame with many points that should have come out of view getting matched to objects within the view of the later frames. The last frame shows that some of these points correctly move to the right. Although the figure shows that all of these points were filtered out by our filtering approach, we believe that these issues are present in other points as well. Furthermore, the large number of filtered points results in a less constrained triangulation problem, which increases the effect of the remaining outliers while ignoring large parts of the frame. We have only observed this behavior on real-world scenes with a large amount of camera motion and the resulting motion blur, which indicates that these limitations stem from the domain gap between synthetic training data and real-world scenarios. We believe that addressing these challenges requires additional training data that specifically captures these failure states.

7.2. Future Work

In this section, we provide some suggestions for future works that could improve on our method. The first and most important improvement is an improved multi-view point tracker. As discussed throughout our paper, the current state-of-the-art point trackers [27, 13] struggle when tested on real-world data due to the limited amount of available real-world training data and the challenge of annotating real data at scale. The main advantage of synthetic data is that it provides perfect ground truth annotations of every point in a frame along with easy scalability. Furthermore, unlike existing real-world annotated datasets that focus on specific scenarios or environments, synthetic data can include a greater variety of conditions. However, the main disadvantage of synthetic datasets is that they lack complex real-world camera artifacts. This leads to a large domain gap in performance between synthetic and real-world data.

CoTracker3 [28] attempts to address this gap by using semi-supervised training on real-world data labeled by other trackers and other versions of itself. The model was first pre-trained on a synthetic dataset for 50 000 iterations on 32 NVIDIA A100 80GB GPUs and then trained for a further 15 000 iterations on the real-world data with pseudo labels. The real-world data consisted of 100 000 internet-like videos that mainly feature humans and animals. Although this approach has a large positive impact on real-world performance, the model still struggles on frames with complex camera artifacts, such as motion blur, sensor noise, and challenging lighting conditions. We believe that a larger dataset that focuses on difficult indoor scenes with complex camera artifacts could improve the performance of the model. Furthermore, the use of physics-based camera degradation models on synthetic scenes could generate large-scale training data with realistic blur, noise, and distortion while preserving perfect ground truth annotations.

The second improvement would be the addition of a custom bundle adjuster for local and global adjustment. Our current approach uses COLMAP [60, 61] for both local and global bundle adjustment. However, this significantly slows down our model due to the different data structures used in COLMAP and inefficiencies introduced due to the more generalized nature of COLMAP's use case. A custom bundle adjuster could be significantly more efficient while allowing more control of the internal optimization pipeline. Having a custom bundle adjuster could also allow for a multi-threaded approach with a separate global bundle adjustment thread running along the tracking and mapping threads, further speeding up the framework at the potential cost of higher GPU memory overhead.

Another limitation of our system is the minimal frame overlap required for a frame window of a select size. We found that larger frame windows tend to produce better results as long as there is sufficient overlap between the first and last frames. This is because the last triangulated points are used as query points for pose refinement. This is exacerbated by our sparse mapping step, which adds two additional point track predictions every time the frame window becomes full.

We believe that both of these problems could be solved by triangulating new points whenever the criteria for adding a new keyframe are met. This would lead to better co-visibility between the new frame and the last sparse mapping frame, which would also be the latest keyframe in the frame window. This is currently not possible in our approach, due to our sparse mapping step taking ten times longer than our per-frame pose estimation. This is caused by two additional point tracking steps, triangulation, and local bundle adjustment. However, if we were able to speed up the multi-view point tracker and local bundle adjustment, then this per-keyframe triangulation might become possible.

Finally, many traditional and even recent methods use loop closure to improve scale consistency during optimization. This is especially important for methods that do not use additional ground truth sensor information such as depth or accelerometer data. The methods that implement loop closure do so by calculating the similarity of the current frame to previous frames. These older frames can then be used to calculate visual flow and be used for pose optimization across the whole scene. This would be difficult with our current framework, as the older frames would have different amounts of overlap with the individual frames in the frame window. We believe that this problem could be mitigated by using the per-keyframe triangulation described in the previous paragraph. The result could be better performance on difficult scenes with minimal speed decrease.

8

Conclusions

We proposed a monocular RGB SLAM system with a dual scene representation. For our camera pose estimation module, we used a sparse point-based scene representation with a novel approach to scene optimization using image correspondences from a pre-trained multi-view point tracking network instead of using only frame-to-frame image correspondences. We combined this sparse point-based scene with a 3D Gaussian map for high-quality dense scene representation. We extended the 3D Gaussian Splatting [32] representation with a monocular depth prediction that we align using our sparse scene to initialize, densify, and optimize the dense 3D Gaussian map.

Our experiments showed that our camera pose estimation performs comparably to concurrent state-of-the-art methods on the synthetic Replica dataset while outperforming some older methods. We also showed that our rendering quality performs better than older methods and even outperforms a concurrent state-of-the-art method. We examined the shortcomings of our method on real-world data by discussing the failure cases of our multi-view point tracking approach. Finally, we proposed improvements to our method that should improve our performance and provide further research directions. Our method has shown the potential of using jointly optimized multi-view image correspondences combined with a 3DGS-based scene for fast and memory efficient visual SLAM systems. We believe that future methods can expand on our work by addressing the limitations of the current multi-view point tracking modules and introducing additional scene optimization techniques.

Bibliography

[1] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. *Ceres Solver*. Version 2.2. Oct. 2023. URL: https://github.com/ceres-solver/ceres-solver.

[2] Michael Bloesch et al. “Codeslam—learning a compact, optimisable representation for dense visual slam”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2560–2568.

[3] Michael Calonder et al. “BRIEF: Binary Robust Independent Elementary Features”. In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792.

[4] Camenduru. *Simple-knn*. 2023. URL: https://github.com/camenduru/simple-knn.

[5] Carlos Campos et al. “ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual–Inertial, and Multimap SLAM”. In: *IEEE Transactions on Robotics* 37 (2020), pp. 1874–1890. URL: https://api.semanticscholar.org/CorpusID:220713377.

[6] Guikun Chen and Wenguan Wang. “A survey on 3d gaussian splatting”. In: *arXiv preprint arXiv:2401.03890* (2024).

[7] Weifeng Chen et al. “An Overview on Visual SLAM: From Tradition to Semantic”. In: *Remote Sensing* 14.13 (2022). ISSN: 2072-4292. URL: https://www.mdpi.com/2072-4292/14/13/3010.

[8] Brian Curless and Marc Levoy. “A volumetric method for building complex models from range images”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 303–312. ISBN: 0897917464.

[9] J Czarnowski et al. “DeepFactors: Real-time probabilistic dense monocular SLAM”. In: *IEEE Robotics and Automation Letters* 5 (2020), pp. 721–728. DOI: 10.1109/lra.2020.2965415. URL: http://dx.doi.org/10.1109/lra.2020.2965415.

[10] Angela Dai et al. “BundleFusion: Real-time Globally Consistent 3D Reconstruction using On-the-fly Surface Re-integration”. In: *ACM Transactions on Graphics 2017 (TOG)* (2017).

[11] Andrew J Davison et al. “MonoSLAM: Real-time single camera SLAM”. In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1052–1067.

[12] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. “SuperPoint: Self-Supervised Interest Point Detection and Description”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2018.

[13] Carl Doersch et al. “TAPIR: Tracking any point with per-frame initialization and temporal refinement”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2023, pp. 10061–10072.

[14] Jakob Engel, Thomas Schöps, and Daniel Cremers. “LSD-SLAM: Large-Scale Direct Monocular SLAM”. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 834–849.

[15] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM.* https://github.com/MichaelGrupp/evo. 2017.

[16] Seongbo Ha, Jiung Yeon, and Hyeonwoo Yu. “Rgbd gs-icp slam”. In: *arXiv preprint arXiv:2403.12550* (2024).

[17] Christopher G. Harris and J. M. Pike. “3D positional integration from image sequences”. In: *Image Vis. Comput.* 6 (1988), pp. 87–90. URL: https://api.semanticscholar.org/CorpusID:3222870.

[18] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[19] Peter Henry et al. “RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments”. In: *International Symposium on Experimental Robotics*. 2010. URL: https://api.semanticscholar.org/CorpusID:2647907.

[20] Peter Henry et al. “RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments”. In: *ISER*. 2010, pp. 477–491. URL: https://doi.org/10.1007/978-3-642-28572-1_33.

[21] Christian Homeyer, Leon Begiristain, and Christoph Schnörr. “DROID-Splat: Combining end-to-end SLAM with 3D Gaussian Splatting”. In: *arXiv e-prints* (2024), arXiv–2411.

[22] Huajian Huang et al. “Photo-SLAM: Real-time Simultaneous Localization and Photorealistic Mapping for Monocular, Stereo, and RGB-D Cameras”. In: *ArXiv* abs/2311.16728 (2023). URL: https://api.semanticscholar.org/CorpusID:265466452.

[23] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size”. In: *arXiv preprint arXiv:1602.07360* (2016).

[24] Mohammad Mahdi Johari, Camilla Carta, and Franccois Fleuret. “ESLAM: Efficient Dense SLAM System Based on Hybrid Representation of Signed Distance Fields”. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2022), pp. 17408–17419. URL: https://api.semanticscholar.org/CorpusID:253735147.

[25] O. Kahler et al. “Very High Frame Rate Volumetric Integration of Depth Images on Mobile Device”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.11 (2015).

[26] R. E. Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552. eprint: https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/82/1/35/5518977/35_1.pdf. URL: https://doi.org/10.1115/1.3662552.

[27] Nikita Karaev et al. “Cotracker: It is better to track together”. In: *European Conference on Computer Vision*. Springer. 2024, pp. 18–35.

[28] Nikita Karaev et al. “CoTracker3: Simpler and Better Point Tracking by Pseudo-Labelling Real Videos”. In: *arXiv preprint arXiv:2410.11831* (2024).

[29] Nikhil Varma Keetha et al. “SplaTAM: Splat, Track & Map 3D Gaussians for Dense RGB-D SLAM”. In: *ArXiv* abs/2312.02126 (2023). URL: https://api.semanticscholar.org/CorpusID:265609060.

[30] Maik Keller et al. “Real-Time 3D Reconstruction in Dynamic Scenes Using Point-Based Fusion”. In: *2013 International Conference on 3D Vision* (2013), pp. 1–8. URL: https://api.semanticscholar.org/CorpusID:17260589.

[31] Alex Kendall, Matthew Grimes, and Roberto Cipolla. “PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 2938–2946. DOI: 10.1109/ICCV.2015.336.

[32] Bernhard Kerbl et al. “3D Gaussian Splatting for Real-Time Radiance Field Rendering”. In: *ACM Transactions on Graphics (TOG)* 42 (2023), pp. 1–14. URL: https://api.semanticscholar.org/CorpusID:259267917.

[33] Georg Klein and David Murray. “Parallel tracking and mapping for small AR workspaces”. In: *2007 6th IEEE and ACM international symposium on mixed and augmented reality*. IEEE. 2007, pp. 225–234.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[35] Stefan Leutenegger, Margarita Chli, and Roland Y. Siegwart. “BRISK: Binary Robust invariant scalable keypoints”. In: *2011 International Conference on Computer Vision*. 2011, pp. 2548–2555. DOI: 10.1109/ICCV.2011.6126542.

[36] Heng Li et al. “Dense RGB SLAM with Neural Implicit Maps”. In: *ArXiv* abs/2301.08930 (2023). URL: https://api.semanticscholar.org/CorpusID:256104917.

[37] David G Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International journal of computer vision* 60 (2004), pp. 91–110.

[38] Hidenobu Matsuki et al. “Gaussian Splatting SLAM”. In: *ArXiv* abs/2312.06741 (2023). URL: https://api.semanticscholar.org/CorpusID:266174429.

[39] Ben Mildenhall et al. “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis”. In: *Commun. ACM* 65 (2020), pp. 99–106. URL: https://api.semanticscholar.org/CorpusID:213175590.

[40] Michael Montemerlo et al. “FastSLAM: a factored solution to the simultaneous localization and mapping problem”. In: *Eighteenth National Conference on Artificial Intelligence*. Edmonton, Alberta, Canada: American Association for Artificial Intelligence, 2002, pp. 593–598. ISBN: 0262511290.

[41] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[42] Raul Mur-Artal, José M. M. Montiel, and Juan D. Tardós. “ORB-SLAM: A Versatile and Accurate Monocular SLAM System”. In: *IEEE Transactions on Robotics* 31 (2015), pp. 1147–1163. URL: https://api.semanticscholar.org/CorpusID:206775100.

[43] Raul Mur-Artal and Juan D. Tardós. “ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras”. In: *IEEE Transactions on Robotics* 33 (2016), pp. 1255–1262. URL: https://api.semanticscholar.org/CorpusID:206775640.

[44] Richard Newcombe, Steven Lovegrove, and Andrew Davison. “DTAM: Dense tracking and mapping in real-time”. In: Nov. 2011, pp. 2320–2327. DOI: 10.1109/ICCV.2011.6126513.

[45] Richard A Newcombe et al. “Kinectfusion: Real-time dense surface mapping and tracking”. In: *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee. 2011, pp. 127–136.

[46] Nicki Skafte Detlefsen et al. *TorchMetrics - Measuring Reproducibility in PyTorch*. Feb. 2022. DOI: 10.21105/joss.04101. URL: https://github.com/Lightning-AI/torchmetrics.

[47] M. Nießner et al. “Real-time 3D Reconstruction at Scale using Voxel Hashing”. In: *ACM Transactions on Graphics (TOG)* (2013).

[48] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 12.4*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[49] Maxime Oquab et al. “Dinov2: Learning robust visual features without supervision”. In: *arXiv preprint arXiv:2304.07193* (2023).

[50] Adam Paszke et al. “PyTorch: an imperative style, high-performance deep learning library”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[51] Songyou Peng et al. “Convolutional Occupancy Networks”. In: *European Conference on Computer Vision (ECCV)*. 2020.

[52] Luigi Piccinelli et al. “UniDepth: Universal Monocular Metric Depth Estimation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, pp. 10106–10116.

[53] Marc Pollefeys et al. “Visual modeling with a hand-held camera”. In: *International Journal of Computer Vision* 59.3 (2004), pp. 207–232.

[54] V. A. Prisacariu et al. “InfiniTAM v3: A Framework for Large-Scale 3D Reconstruction with Loop Closure”. In: *ArXiv e-prints* (Aug. 2017). eprint: 1708.00783.

[55] Antoni Rosinol, John J. Leonard, and Luca Carlone. “NeRF-SLAM: Real-Time Dense Monocular SLAM with Neural Radiance Fields”. In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2022), pp. 3437–3444. URL: https://api.semanticscholar.org/CorpusID:253107311.

[56] Edward Rosten and Tom Drummond. “Machine Learning for High-Speed Corner Detection”. In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443.

[57] Ethan Rublee et al. “ORB: An efficient alternative to SIFT or SURF”. In: *2011 International Conference on Computer Vision* (2011), pp. 2564–2571. URL: https://api.semanticscholar.org/CorpusID:206769866.

[58] Erik Sandström et al. “Point-SLAM: Dense Neural Point Cloud-based SLAM”. In: *2023 IEEE/CVF International Conference on Computer Vision (ICCV)* (2023), pp. 18387–18398. URL: https://api.semanticscholar.org/CorpusID:258049300.

[59] Erik Sandström et al. “Splat-slam: Globally optimized rgb-only slam with 3d gaussians”. In: *arXiv preprint arXiv:2405.16544* (2024).

[60] Johannes Lutz Schönberger and Jan-Michael Frahm. “Structure-from-Motion Revisited”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[61] Johannes Lutz Schönberger et al. “Pixelwise View Selection for Unstructured Multi-View Stereo”. In: *European Conference on Computer Vision (ECCV)*. 2016.

[62] Thomas Schöps, Torsten Sattler, and Marc Pollefeys. “BAD SLAM: Bundle Adjusted Direct RGB-D SLAM”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 134–144. URL: https://api.semanticscholar.org/CorpusID:196201321.

[63] Jianbo Shi and Tomasi. “Good features to track”. In: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 1994, pp. 593–600. DOI: 10.1109/CVPR.1994.323794.

[64] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[65] Randall Smith, Matthew Self, and Peter Cheeseman. “Estimating Uncertain Spatial Relationships in Robotics”. In: *Uncertainty in Artificial Intelligence*. Ed. by John F. LEMMER and Laveen N. KANAL. Vol. 5. Machine Intelligence and Pattern Recognition. North-Holland, 1988, pp. 435–461. DOI: https://doi.org/10.1016/B978-0-444-70396-5.50042-X. URL: https://www.sciencedirect.com/science/article/pii/B978044470396550042X.

[66] Ewa Splatting et al. “Ieee Transactions on Visualization and Computer Graphics Ewa Splatting”. In: URL: https://api.semanticscholar.org/CorpusID:9389692.

[67] Patrick Stotko et al. “SLAMCast: Large-Scale, Real-Time 3D Reconstruction and Streaming for Immersive Multi-Client Live Telepresence”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.5 (2019), pp. 2102–2112. DOI: 10.1109/TVCG.2019.2899231.

[68] Julian Straub et al. “The Replica Dataset: A Digital Replica of Indoor Spaces”. In: *arXiv preprint arXiv:1906.05797* (2019).

[69] J. Sturm et al. “A Benchmark for the Evaluation of RGB-D SLAM Systems”. In: *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. Oct. 2012.

[70] Edgar Sucar et al. “iMAP: Implicit Mapping and Positioning in Real-Time”. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021), pp. 6209–6218. URL: https://api.semanticscholar.org/CorpusID:232320654.

[71] I.E. Sutherland. “Three-dimensional data input by tablet”. In: *Proceedings of the IEEE* 62.4 (1974), pp. 453–461. DOI: 10.1109/PROC.1974.9449.

[72] Keisuke Tateno et al. “ CNN-SLAM: Real-Time Dense Monocular SLAM with Learned Depth Prediction ”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2017, pp. 6565–6574. DOI: 10.1109/CVPR.2017.695. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.695.

[73] Zachary Teed and Jia Deng. “DROID-SLAM: Deep Visual SLAM for Monocular, Stereo, and RGB-D Cameras”. In: *Neural Information Processing Systems*. 2021. URL: https://api.semanticscholar.org/CorpusID:237278112.

[74] Zachary Teed and Jia Deng. “RAFT: Recurrent All-Pairs Field Transforms for Optical Flow”. In: *European Conference on Computer Vision*. 2020. URL: https://api.semanticscholar.org/CorpusID:214667893.

[75] Carlo Tomasi and Takeo Kanade. “Shape and motion from image streams under orthography: a factorization method”. In: *International journal of computer vision* 9.2 (1992), pp. 137–154.

[76] Bill Triggs et al. “Bundle adjustment—a modern synthesis”. In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[77] Leif Van Holland et al. “Efficient 3D Reconstruction, Streaming and Visualization of Static and Dynamic Scene Parts for Multi-Client Live-Telepresence in Large-Scale Environments”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*. Oct. 2023, pp. 4258–4272.

[78] Hengyi Wang, Jingwen Wang, and Lourdes de Agapito. “Co-SLAM: Joint Coordinate and Sparse Parametric Encodings for Neural Real-Time SLAM”. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2023), pp. 13293–13302. URL: https://api.semanticscholar.org/CorpusID:258352757.

[79] Jianyuan Wang et al. “VGGSfM: Visual Geometry Grounded Deep Structure From Motion”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2024, pp. 21686–21697.

[80] Sen Wang et al. “DeepVO: Towards end-to-end visual odometry with deep Recurrent Convolutional Neural Networks”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. Singapore, Singapore: IEEE Press, 2017, pp. 2043–2050.

[81] Zirui Wang et al. “NeRF−−: Neural Radiance Fields Without Known Camera Parameters”. In: *arXiv preprint arXiv:2102.07064* (2021).

[82] Thomas Whelan et al. “ElasticFusion: Dense SLAM Without A Pose Graph”. In: *Robotics: Science and Systems*. 2015. URL: https://api.semanticscholar.org/CorpusID:16413702.

[83] Chi Yan et al. “GS-SLAM: Dense Visual SLAM with 3D Gaussian Splatting”. In: *ArXiv* abs/2311.11700 (2023). URL: https://api.semanticscholar.org/CorpusID:265294572.

[84] Kaiyun Yang et al. “SLAM Meets NeRF: A Survey of Implicit SLAM Methods”. In: *World Electric Vehicle Journal* 15.3 (2024). ISSN: 2032-6653. URL: https://www.mdpi.com/2032-6653/15/3/85.

[85] Xingrui Yang et al. “Vox-Fusion: Dense Tracking and Mapping with Voxel-based Neural Implicit Representation”. In: *2022 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)* (2022), pp. 499–507. URL: https://api.semanticscholar.org/CorpusID:253223971.

[86] Vladimir V. Yugay et al. “Gaussian-SLAM: Photo-realistic Dense SLAM with Gaussian Splatting”. In: *ArXiv* abs/2312.10070 (2023). URL: https://api.semanticscholar.org/CorpusID:266348788.

[87] Ganlin Zhang et al. “Glorie-slam: Globally optimized rgb-only implicit encoding point cloud slam”. In: *arXiv preprint arXiv:2403.19549* (2024).

[88] Richard Zhang et al. “The unreasonable effectiveness of deep features as a perceptual metric”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 586–595.

[89] Youmin Zhang et al. “GO-SLAM: Global Optimization for Consistent 3D Instant Reconstruction”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2023.

[90] Zihan Zhu et al. “NICE-SLAM: Neural Implicit Scalable Encoding for SLAM”. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), pp. 12776–12786. URL: https://api.semanticscholar.org/CorpusID:245385791.

[91] Zihan Zhu et al. “NICER-SLAM: Neural Implicit Scene Encoding for RGB SLAM”. In: *ArXiv* abs/2302.03594 (2023). URL: https://api.semanticscholar.org/CorpusID:256627303.