Building temporally coherent 3D maps with VGGT for memory-efficient Semantic SLAM

Anonymous Preprint

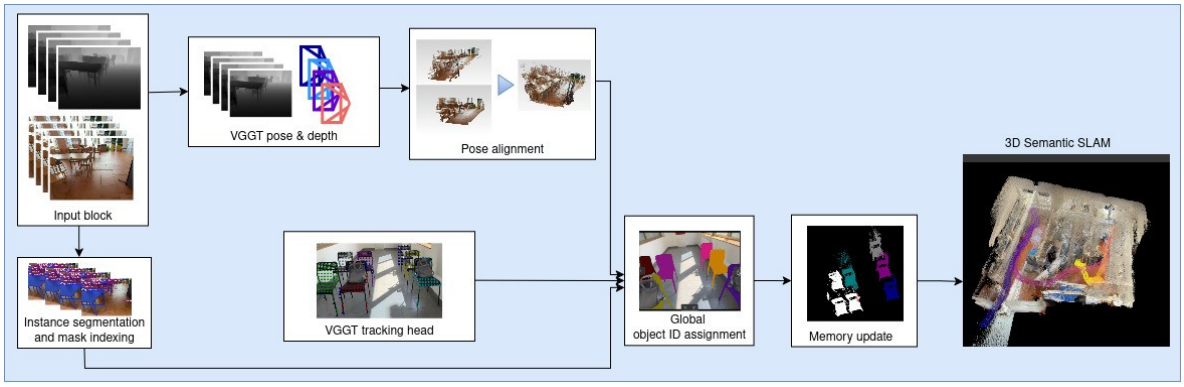


Figure 1. Overview of the proposed pipeline: memory-efficient, temporally coherent semantic SLAM with VGGT.

Abstract

We present a fast, spatio-temporal scene understanding framework based on Vision Gated Generative Transformers (VGGT). The proposed pipeline is designed to enable efficient, close to real-time performance, supporting applications including assistive navigation. To achieve continuous updates of the 3D scene representation, we process the image flow with a sliding window, aligning submaps, thereby overcoming VGGT's high memory demands. We exploit the VGGT tracking head to aggregate 2D semantic instance masks into 3D objects. To allow for temporal consistency and richer contextual reasoning the system stores timestamps and instance-level identities, thereby enabling the detection of changes in the environment. We evaluate the approach on well-known benchmarks and custom datasets specifically designed for assistive navigation scenarios. The results demonstrate the applicability of the framework to real-world scenarios.

1. Introduction

3D scene understanding underpins assistive navigation: users must operate in cluttered, unfamiliar, and evolving indoor environments. A practical system therefore needs a temporally coherent 3D map that stays stable across viewpoint change, partial occlusions, and genuine scene state changes. Moreover, approaches should be designed for a low memory footprint and high speed. Feed-forward multi-view models such as VGGT [13] provide fast depth and pose estimation. However, applying this paradigm to long sequences is challenging for several reasons: (i) memory requirements grow rapidly with sequence length; (ii) temporal dependencies are not modeled explicitly; and (iii) the target application demands sustained spatio-temporal consistency across hundreds of frames while also incorporating semantics — i.e., handling open-world objects — and adapting to rearrangements and genuine scene changes.

In realistic deployments, input arrives as a continuous video stream and the map must be updated online. Methods that rely on global, post-hoc optimization over full sequences presuppose offline access to all data and are ill-suited for real-time guidance. Accordingly, we target an online, streaming-friendly formulation that incrementally maintains geometry and semantics as frames arrive. [1]

We focus on indoor environments, a primary deployment setting for assistive navigation, where mapping is challenged by reflective and transparent surfaces, repeated structures, motion blur from handheld sensor devices, and frequent occlusions. We primarily target a monocular RGB setup; when auxiliary depth from commodity mobile sensors (e.g., LiDAR) is available, we use it only to stabilize metric scale, and we report results for both configurations.

Beyond integrating semantics into a geometrically accurate 3D map suitable for SLAM, the semantic layer should

[1] We will publish the code in a public repository.

provide a firm basis to perform tasks with real user impact; accordingly, we consider the broadly relevant problem of finding an empty seat in indoor environments as a representative challenge. This can be challenging in unfamiliar spaces and because seat state can change even when the surrounding scene appears static.

2. Related work

Recent years have seen rapid progress in close to real time 3D scene representations from motion. We build on VGGT [13], a pretrained, feed-forward vision transformer that, from RGB images alone, predicts depth, a point cloud, and camera intrinsics and poses (extrinsics). The authors benchmark on an NVIDIA H100 GPU and report 8.75s for a 200-frame sequence. The memory footprint exceeds 40GB at 200 frames and increases approximately linearly with sequence length on this setup.

VGGT Long [2] extends VGGT's memory-limited, locally consistent point-map generation to kilometer-scale scenes. It partitions the sequence into overlapping chunks ($\tilde{6}$0-75 frames), aligns each chunk to the previous via Iteratively Reweighted Least Squares (IRLS) optimization, and mitigates drift through loop detection and Sim(3) alignment of overlapping pointclouds, followed by a global optimization over the chunk transforms. Because it requires multiple passes and access to the full sequence—including a final global adjustment over all frames—the method is not suited to streaming input; moreover, dynamic objects are typically down-weighted and effectively treated as outliers due to low confidence.

FastVGGT [8] accelerates VGGT via training-free token merging in the Global Attention module, achieving a 4× speed-up on an NVIDIA H100 GPU. Whereas VGGT, provided 80GB VRAM, runs out of memory at around 300 frames, FastVGGT increases the feasible sequence length to over 1 000 frames ($\simeq 3.5\times$).

To address true streaming/online use, StreamVGGT [15] is designed to process incoming frames incrementally. However, because its spatio-temporal tokens scale with sequence length, both memory usage and latency grow with the number of frames. It replaces global self-attention with temporal causal attention and caches historical keys/values as an implicit memory. The cached-token design implies that memory footprint grows with sequence length, which the authors explicitly list as a challenge for long sequences and resource-constrained devices.

VGGT-SLAM [6] extends feed-forward VGGT to long sequences by running VGGT on overlapping keyframe windows to form submaps, then aligning them via 15-DOF homographies on the SL(4) manifold to address the projective ambiguity that arises with uncalibrated monocular, offline input. The pipeline augments sequential alignment with loop-closure constraints and solves a global factor-graph optimization on SL(4).

Spatial Tracker V2 [14] unifies 3D point tracking, monocular video depth, and camera pose estimation in a feed-forward pipeline, achieving state-of-the-art results. The authors report up to 50× faster runtime than dynamic 3D reconstruction approaches, however, memory usage grows with sequence length, and a ∼150-frame input may require more than 40GB of GPU VRAM.

3. Methods

3.1. Global alignment

The number of input frames that VGGT can process for 3D generation is constrained by GPU memory. To scale to longer sequences, we construct submaps with VGGT and then align them, thereby making efficient use of memory. We partition the input video stream into non-overlapping blocks of consecutive frames. To align submaps globally and efficiently, we keep a rolling global reference extrinsic matrix and exploit that VGGT expresses all poses within in a block of images in the coordinate system of the first frame of the block (see Fig. 2 for illustration).

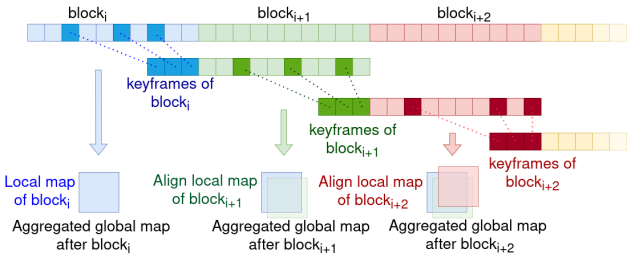


Figure 2. The input stream is partitioned into blocks, with key frames serving as anchors for alignment.

Let each block B_i contain n RGB frames and the corresponding depths maps too:

$$\mathcal{I}^i := \{\mathbf{I}_t^i\}_{t=0}^{n-1}, \qquad \mathcal{D}^i := \{\mathbf{D}_t^i\}_{t=0}^{n-1}.$$

For each block we select k keyframes ($0 < k < n$), which will provide the basis for alignment:

$$\mathcal{K}^i \subset \{0, \ldots, n-1\}, \quad |\mathcal{K}^i| = k, \qquad \mathcal{K}^i := \{I_t^i \mid t \in \mathcal{K}^i\}.$$

The frames with the most detectable features are selected as keyframes to suppress motion-blur–induced noise. In the implementation we used the ORB feature detector [7]. We compose a current frame list $\mathcal{I}^{current}$ by concatenating the previous block's keyframes with the current block's frames and corresponding depths:

$$\mathcal{I}^{current} := \big((\mathbf{I}_t^{i-1})_{t\in\mathcal{K}^{i-1}}, (\mathbf{I}_t^i)_{t=0}^{n-1}\big),$$

$$\mathcal{D}^{current} := \big((\mathbf{D}_t^{i-1})_{t\in\mathcal{K}^{i-1}}, (\mathbf{D}_t^i)_{t=0}^{n-1}\big).$$

For $\mathcal{I}^{current}$ we predict the depth maps and the extrinsics using VGGT depth and pose heads:

$$\mathcal{D}^{current}_{\text{VGGT}} := \{\mathbf{D}^{current}_{\text{VGGT},j}\}_{j=0}^{n+k-1},$$

$$\mathcal{E}^{current} := \{\mathbf{E}^{current}_{j}\}_{j=0}^{n+k-1},$$

where

$$\mathbf{E}^{current}_{j} \;=\; \big[\,\mathbf{R}_j \mid \mathbf{t}_j\,\big] \in \mathbb{R}^{3\times 4},$$

where **R** denotes rotation and **t** denotes translation.

The poses within the first block B_0 get aligned by VGGT. For a pair of consecutive blocks B_{i-1}, B_i, where B_{i-1} is the last aligned block and B_i is the current, unaligned block, we first calculate the depth scale factor and then, the Sim3 similarity transform.

Although the depth data of a LiDAR sensor is noisy, and not as fine-grained as VGGT's depth estimate, it can be used to obtain a consistent scaling. We apply depth thresholding to the LiDAR depth data to eliminate sensor measurement errors in unreliable ranges (very near and very far) or where the depth estimation confidence of VGGT is too low, by a mask M_j, that retains only the values between the thresholds for all $D_j \in \mathcal{D}^{current}$. To determine the s_j scaling factor we apply least-squares method, as follows:

$$s_j \;=\; \underset{s}{\operatorname{argmin}} \left\| M_j\big(s\, D^{current}_{\text{VGGT},j} - D^{current}_{,j}\big)\right\|_2^2$$

for $j = 0, \ldots, n{+}k{-}1$, where $D^{current}_{\text{VGGT},j}$ and $D^{current}$ are the depth maps for frame $j \in [0, n+k-1]$ of the frames in the current frame list, and M_j masks valid pixels, between near and far ranges. We then take the median of the scale values $\tilde{s} = \text{median}\{s_j\}_{j=0}^{n+k-1}$ over the block to improve robustness, and rescale all $D^{current}_{\text{VGGT},j}$ VGGT depth maps in $\mathcal{D}^{current}_{\text{VGGT}}$ and the translation component of all extrinsics in $\mathcal{E}^{current}$ accordingly:

$$\hat{\mathbf{D}}^{current}_{j} \;=\; \tilde{s}\,\mathbf{D}^{current}_{\text{VGGT},j}, \qquad \hat{\mathbf{E}}^{current}_{j} \;=\; \big[\,\mathbf{R}_j \mid \tilde{s}\,\mathbf{t}_j\,\big].$$

For the similarity transformation we use our updated extrinsics. Let us denote the extrinsic matrix of the global graph by $\mathbf{E}_{\text{ref}}$. As VGGT aligns all camera poses in $\mathcal{E}^{current}$ with respect to $E^{current}_0$, we pick $E^{current}_0$, to be the corresponding extrinsics for the current frame list:

$$\mathbf{E}^{current}_{ref} \;=\; \mathbf{E}^{current}_{0}$$

To align the current block's VGGT poses to all previous, already aligned blocks, we apply a right-multiplicative transformation on the extrinsics:

$$\Delta \;=\; \big(\mathbf{E}^{current}_{ref}\big)^{-1}\,\mathbf{E}_{\text{ref}}.$$

We update the extrinsics ($\forall E_j \in \mathcal{E}^{current}$) in the current lists by this transformation:

$$\hat{\mathbf{E}}_i \;=\; \mathbf{E}_i\,\Delta, \qquad \forall i,$$

which aligns the block to the reference while preserving intra-block relative poses. This means, that also the corresponding frame's extrinsic matrix is updated as follows:

$$\hat{\mathbf{E}}^{current}_{ref} \;=\; \mathbf{E}^{current}_{ref}\,\Delta \;=\; \mathbf{E}_{\text{ref}}.$$

Therefore, as a last step, we select the key frames from the current block into the keyframe memory and the global reference extrinsic matrix is set to the (updated) extrinsic matrix of the first keyframe of this current block:

$$\mathbf{E}_{ref} \;=\; \hat{\mathbf{E}}^{current}_{ref}.$$

To smooth the camera-pose trajectory, we apply the Curvature-Corrected Moving Average (CCMA) [10]. For each sample index n, CCMA uses the $2k{+}1$-point neighbourhood $n-k, \ldots, n+k$ along the trajectory, where k is a hyperparameter; we therefore apply it independently per block. Because the alignment transform is obtained by composing per-frame transforms, smoothing every transform would compound noise. We maintain both raw (unsmoothed) and smoothed extrinsics: the smoothed pose is used for submap alignment, while the raw pose is used for point-cloud unprojection. We also compute smoothed extrinsics to produce filtered object point clouds. This step has three hyperparameters: (1) k_1, the number of neighbouring points for curvature correction; (2) k_2, the number of neighbouring points for smoothing; and (3) the CCMA kernel. Unless otherwise stated, we keep k_1 and k_2 at their default values and use a Hann (a.k.a. Hanning) kernel.

3.2. 3D segmentation

We propose a method to represent real-world objects as 3D point clouds, where the points of each point cloud are labeled with the instance type and ID. To ensure correspondence with reality, these labels are maintained consistently throughout the sequence, thereby defining global objects in the 3D representation over time. To add semantic information into the 3D representation we apply instance segmentation within the 2D frames, and instance tracking both inside and across blocks. For 2D segmentation, we use a foundation model, namely YOLOv9e [12] from Ultralytics [4], which is pretrained on the MS COCO dataset [5]. Our goal is to produce per-frame instance masks that are assigned to a global ID. We first perform instance segmentation on 2D RGB frames using the foundation model, and post-process the masks with a slight erosion to reduce noise after inverse projection back into the 3D representation. To exploit the contextual information inherently captured by VGGT, and ensure that tracking remains robust even in challenging scenarios with numerous overlapping and occluding objects, we use the tracking head for tracking instance masks across frames. To improve computational and memory efficiency, instead of tracking all mask pixels, we apply a uniform grid-based subsampling. The VGGT tracking head returns per-frame point trajectories with a confidence score for all grid

points. We only retain points with confidence scores above a threshold (set to 0.1). In each frame, each preserved point is assigned to a local instance mask or to the background, yielding an initial association. Sampling instance masks and the resulting point-to-mask assignments are illustrated in Fig. 3.

Figure 3. Tracked points sampled from instance masks. Each color corresponds to the assigned instance mask, while dark blue dots indicate unassigned (propagated) points.

To aggregate 2D instances into 3D objects we perform a voting. We define a tracklet as a temporal sequence of detections consistently assigned to the same global ID. Thus, each tracklet therefore corresponds to a single real-world object across frames. The tracked sample points from the masks of the previous frame are propagated to the current frame using the VGGT tracking head, and for every 2D instance mask in the current frame we measure the support from the tracklets, defined as the number of propagated points from a given tracklet (global ID) that fall inside the mask. We then perform mutual assignment: for each mask the tracklet that has the largest support on that mask is selected as a candidate, and then each tracklet is assigned to the mask (out of the ones where it became a candidate) on which most of its points fall. This conservative strategy (favoring false negatives over false positives) mitigates occlusion errors and avoids erroneous merges. We also enforce a type-consistency constraint: a mask and tracklet are matched only if their object classes agree (e.g., a bag may not become a chair). If a mask has no valid mutual match among the tracklets, then a new tracklet is initialized and a new global ID is allocated to it if it happens on the first frame of a block, otherwise a default ID ("untracked") is assigned to it; if a tracklet receives no supporting points in a frame, we record an empty state to preserve temporal alignment and allow for later re-identification. Unmatched tracklets may get resolved via re-identification.

For re-identification across temporal gaps or block boundaries for the very same objects, the point cloud of all newly initialized tracklet is compared against the point clouds of previously established tracklets. A cost matrix is constructed using the 3D intersection-over-union (IoU) of bounding boxes of the point clouds. If the overlap exceeds a predefined threshold, the new sub-tracklet is merged with the existing tracklet, and all stored identifiers are updated accordingly, both at the instance level and at the global object level.

Objects may leave and re-enter the field of view, so we maintain a persistent memory of previously seen instances. When a new tracklet appears, we perform re-identification to check whether it corresponds to a past object. For each 2D instance, we compute the median point of its 3D point cloud (PCD) for every frame in the block—yielding up to one median point per frame—and label these with the global ID. We do the same for the current block's instances that could start new tracklets. We then compare historical and current median-point clouds using Chamfer distance; if the closest match is below a threshold (a hyperparameter, 30 cm in our setup), we match the instance to the prior tracklet. This approach also handles the type of noise occurring when 2D instance masks cover holes, which would otherwise appear as artifacts during 3D projection.

3.3. Change detection

Apart from the static elements of the environment, there are realistically changing parts. We target to handle changes in the state or presence of objects — for example, a chair transitioning from unoccupied to occupied, or vice versa — where the actual movement does not occur within the camera's field of view. The proposed pipeline detects changes by assigning timestamps to global objects, providing a memory mechanism through continuous updating.

Frames arrive sequentially, thus we assign incrementally an index to each frame, and each segmented instance is assigned the index of the frame in which it appears. This enables us to track the temporal occurrence of each instance. We associate the instance with its corresponding global ID (i.e., the object it represents) as described in subsection 3.2, which allows us to maintain a memory of the object and its state changes over time. Each object can be in one of three states: (1) RECENT, (2) RETAINED, or (3) REMOVED. We associate each object with a confidence value c. An object is considered RECENT if it is observed in the last block, and its confidence is set to $c = 1$.

The states of the objects are updated at each block. The main steps are described in Alg. 1. To update the state of each object o with 3D points P_o, in the subsequent frames, we check the visibility. We project all points onto the 2D image plane of the current camera pose $\mathbf{T}_{ts}$, where ts is the timestamp (or index) of the frame, yielding the projected image $\hat{I}_i$. We keep only those points with depth $z_{\text{proj}} > 0$

(i.e., in front of the camera plain) and with pixel coordinates inside the image bounds; this subset is denoted by Ω_i^o. If $|\Omega_i^o| = 0$ the object is not in the field of view, meaning it is not observed in the latest block, and becomes RETAINED. On this state change no penalty is applied the last value of c is maintained, which may then be gradually decayed over time.

For objects that are within the field of view, we further evaluate their visibility using the depth image $\mathbf{D}_i$. A projected pixel $p \in \Omega_o^i$ is considered *visible* if

$$z_{\text{proj}}(p) \;\leq\; z_{\text{obs}}(p) + \delta,$$

where $z_{\text{proj}}(p)$ is the depth of the projected 3D point, $z_{\text{obs}}(p)$ is the observed depth at the corresponding pixel location, and δ is a tolerance value (e.g. $1mm$). This formalizes the intuition that if no other objects are present between the expected object and the camera, the object's mask should be visible in the image. If the mask is not observed under these conditions, the object is considered missing.

The fraction of visible pixels is then defined as

$$f_{\text{vis}} \;=\; \frac{\#\text{ visible pixels in } \Omega_o}{|\Omega_o|}.$$

If $f_{\text{vis}} \geq \tau_{\text{vis}}$ threshold value for visibility ($\tau_{\text{vis}} = 0$ by default) but no detection is associated with object o in the current frame, the memory confidence is decayed as

$$c \;:=\; c - \eta, \qquad \eta, c \in [0, 1].$$

The parameter η reflects the confidence in forgetting: smaller values make the system less certain about forgetting (slower decay), while values closer to 1 increase the confidence and lead to faster forgetting. If the confidence c of an object becomes zero, then its state is set to REMOVED.

To avoid penalizing small boundary fragments that detectors frequently miss, decay is skipped when the projected area fraction falls below a threshold ($\tau_{\text{area}} = 0$ by default). Objects with a RETAINED or REMOVED status may later become active again via re-identification.

3.4. Ego-location and object positions

In the streaming scenario, we consider the position of the user as the camera pose corresponding to the last frame index. To determine the distances of detected objects from the current pose (i.e. the user holding the sensor) and the distances among objects we use a matching approach as follows. Whenever an object is updated, its associated point cloud is updated as well. A median point is then computed over this point cloud, considering all points belonging to the object (unlike the per-instance median used in Section 3.2). This median serves then as the object's estimated location.

In each iteration, i.e. in each block, we compute the relative positions for all objects currently considered valid, using the position of the user and the object's centroid obtained from the median, using Euclidean distance.

Algorithm 1: Object Visibility and Confidence Update. We update object visibility states as the last step of processing the current block of frames.

Input: Sequence of frames with object detections
Output: Updated object states with confidence values

```
foreach object o do
    Reproject o into current block (using current poses);
    if o is visible in the reprojected view then
        Compare depth of o with scene depth;
        if scene depth < object depth then
            Increase uncertainty value of o as it may be occluded;
    Determine state of o: in-field-of-view / hidden;
    if o not present where expected then
        Update confidence of o (simulate forgetting);
```

4. Results

4.1. Experimental setup and metrics

For evaluation, we use the TUM RGB-D dataset [11], a standard benchmark for indoor perception. It provides calibrated, time-synchronized RGB-D sequences across diverse layouts with accurate motion ground truth, making it well suited for assessing visual odometry, SLAM, and map building—the backbone of navigation stacks. We also evaluate on the 7-Scenes dataset [9], which includes dense point clouds, allowing us to measure accuracy, completion, and Chamfer distance, mirroring the VGGT-SLAM protocol. We report average values in Table 4 measured on sequences: *stairs, pumpkin, heads*. For navigation and SLAM we report Absolute Trajectory Error (ATE) [11]. We evaluate odometry using the *evo* toolkit [3].

For semantics and 3D reconstruction, we target a use case relevant to visually impaired users: identifying unoccupied seats in public indoor settings. As, to our knowledge, no widely adopted benchmark exists for this task, we provide qualitative results on our own dataset. We present visualizations of reconstructed 3D maps — both RGB and semantic — and include scenarios with environmental changes (e.g., a bag displaced from a chair).

Measurements were performed on a computer with Ubuntu 22.04, equipped with an AMD Ryzen Threadripper Pro 5955WX CPU, NVIDIA RTX 4090 GPU with 24GB of VRAM, and 512GB of RAM.

Similarly to the evaluation protocol used in VGGT-SLAM [6], we apply a transformation to align the predictions with the ground-truth coordinate system, enabling di-

rect comparison.

4.2. Quantitative results

VGGT can process about 60 frames within the 24 GB VRAM limit. For the proposed pipeline we measured VRAM usage with NVIDIA's Nsight Systems tool [1]. Although the implemetation has not been optimized, we report the measurements in Table 1., alongside with the total running time and the used CPU RAM as a reference, as well as the number of frames of the individual scenes.

In Tables 2 and 3 we report the absolute trajectory error (ATE) as root mean square error (RMSE) for the 7-Scenes and for the TUM RGB-D sequences, respectively. Our approach works remarkably well on the TUM floor scene, which proved to be a very challenging subtask for VGGT SLAM, because, as they note, the high number of images containing only the flat floor lead to non-unique solutions for the homography matrix, and ultimately divergence. By using shorter block sizes and thereby limiting the focus of the VGGT reconstruction to the more recent frames, we successfully achieved much better convergence.

4.3. Qualitative results

Beyond the quantitative results, the method's potential is well illustrated by how it performs on difficult cases. In Fig. 4, we show the point clouds produced by VGGT-SLAM and by our method for the same input.

4.4. Discussion

In instance tracking, when an instance has no tracklet based on the tracked points, we only initialize a tracklet if this occurs at the very first frame in the frame list. The reason is that the VGGT tracking head is initialized from the first frame of the block. This means that for intermediate frames within the block (e.g., when an object is occluded), no new tracklet is created that could later be re-identified. Instead, such objects loose or not get a global object ID and remain untracked but detected objects, if they are not visible at the beginning of the block. A possible way to address this would be to wait for data within each block and decide, using some method, which frame in the block should be used for initializing tracking, or to allow initialization from multiple frames. However, the latter approach has the drawback that each initialization and tracking step takes about 0.5 seconds, which would introduce too much overhead. The former also incurs computational overhead and, on its own, may not eliminate all failure cases: with a block size of 10, an object can remain untracked for about 0.5s in the worst case and thus becomes "untracked" — we know it is present but cannot associate it with a specific instance — which has a negligible impact on the assistive navigation application.

4.5. Conclusion

We introduced a memory-efficient pipeline for temporally coherent 3D semantic mapping based on Vision Gated Generative Transformers (VGGT). By processing video input in blocks and aligning submaps, the method overcomes the high memory footprint of standard VGGT, enabling long-sequence operation. Our integration of semantic instance masks with VGGT's tracking head yields object-level consistency across time, supporting robust change detection in dynamic environments. With the representation now containing all necessary information, the remaining step is to communicate it effectively to the user for assistive navigation. The framework is designed to enable the processing streaming input and achieves close to real-time performance on a desktop GPU, making it well suited for assistive navigation and other applications where both geometric accuracy and semantic awareness are required. Future work will explore the integration of moving, dynamic objects in the representation.

Table 1. RAM, VRAM, and running time measurement on the TUM RGBD dataset

Sequence	360	desk	desk2	floor	plant	room	rpy	teddy	xyz
Number of frames	740	570	620	1220	1120	1350	690	1400	790
Max RAM (GB)	12.4	10.1	10.8	18.7	17.3	20.3	11.6	21.0	13.0
Max VRAM (GB)	17.8	17.8	17.8	17.7	17.8	17.8	17.8	17.8	17.9
Total time (s)	134.53	110.09	115.61	211.66	193.63	231.14	134.57	247.91	147.51

Table 2. Comparison of the root mean square error of VGGT-SLAM and the proposed method across 7-Scenes sequences.

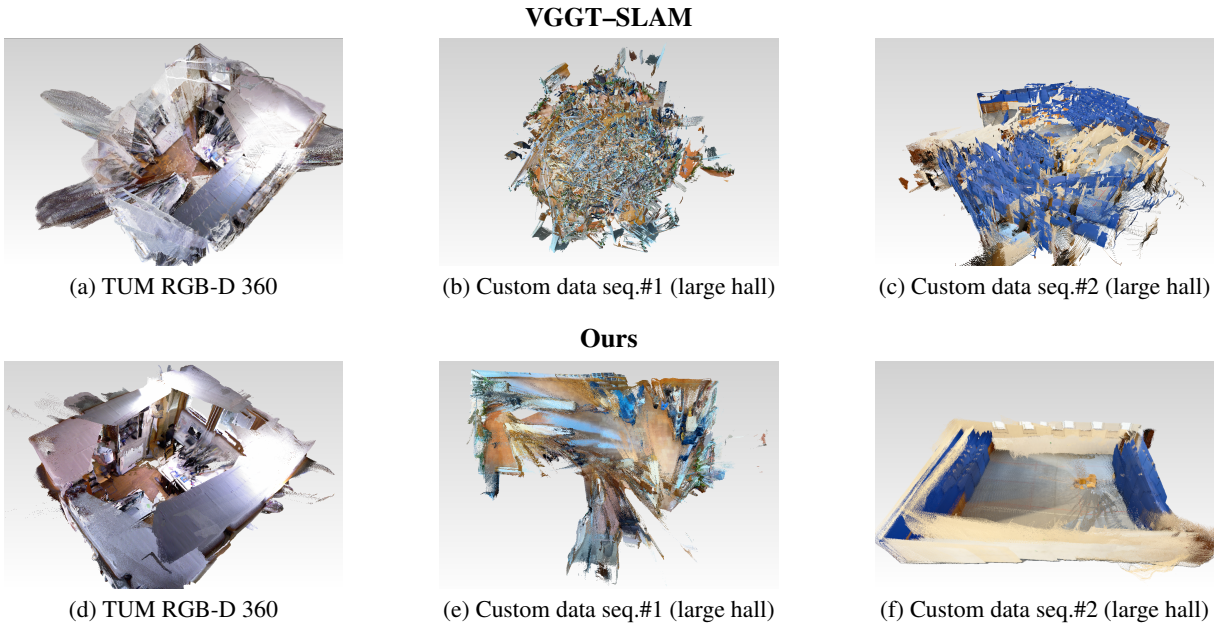
Method	chess	fire	heads	office	pumpkin	kitchen	stairs	Avg
VGGT-SLAM Sim(3)	0.037	0.026	0.018	0.104	0.133	0.061	0.093	0.067
VGGT-SLAM SL(4)	0.036	0.028	0.018	0.103	0.133	0.058	0.093	0.067
Ours	0.055	0.047	0.034	0.120	0.152	0.063	0.032	0.072

Table 3. Comparison of the root mean square error of VGGT-SLAM and the proposed method across TUM RGB-D sequences.

Method	360	desk	desk2	floor	plant	room	rpy	teddy	xyz	Avg
VGGT-SLAM Sim(3)	0.123	0.040	0.055	0.254	0.022	0.088	0.041	0.032	0.016	0.074
VGGT-SLAM SL(4)	0.071	0.025	0.040	0.141	0.023	0.102	0.030	0.034	0.014	0.053
Ours	0.118	0.042	0.039	0.063	0.077	0.081	0.029	0.076	0.034	0.062

Table 4. Root mean square error (RMSE) of the reconstruction evaluated on 7-Scenes

Method	Acc. ↓	Complet. ↓	Chamfer ↓
VGGT-SLAM Sim(3)	0.052	0.062	0.057
VGGT-SLAM SL(4)	0.052	0.058	0.055
Ours	0.0566	0.0668	0.06179



(a) TUM RGB-D 360 (b) Custom data seq.#1 (large hall) (c) Custom data seq.#2 (large hall)

(d) TUM RGB-D 360 (e) Custom data seq.#1 (large hall) (f) Custom data seq.#2 (large hall)

Figure 4. Point-cloud comparison on identical inputs across four sequences. Top row: VGGT–SLAM; bottom row: our method. Vertically aligned pairs correspond to the same input.

References

[1] NVIDIA Corporation. Nsight Systems. https://developer.nvidia.com/nsight-systems, 2025. Accessed: 2025-09-20. 6

[2] Kai Deng, Zexin Ti, Jiawei Xu, Jian Yang, and Jin Xie. VGGT-Long: Chunk it, loop it, align it – pushing vggt's limits on kilometer-scale long rgb sequences. *arXiv preprint arXiv:2507.16443*, 2025. 2

[3] Michael Grupp. evo: Python package for the evaluation of odometry and slam. https://github.com/MichaelGrupp/evo, 2017. 5

[4] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. Ultralytics yolo. https://github.com/ultralytics/ultralytics, 2023. Version 8.0.0. License: AGPL-3.0. 3

[5] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing. 3

[6] Dominic Maggio, Hyungtae Lim, and Luca Carlone. Vggt-slam: Dense rgb slam optimized on the sl(4) manifold. *arXiv preprint arXiv:2505.12549*, 2025. 2, 5

[7] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski. Orb: An efficient alternative to SIFT or SURF. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571, 2011. 2

[8] You Shen, Zhipeng Zhang, Yansong Qu, and Liujuan Cao. Fastvggt: Training-free acceleration of visual geometry transformer, 2025. 2

[9] Jamie Shotton, Ben Glocker, Christopher Zach, Shahram Izadi, Antonio Criminisi, and Andrew Fitzgibbon. Scene coordinate regression forests for camera relocalization in RGB-D images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2930–2937. IEEE, 2013. 5

[10] Thomas Steinecker and Hans-Joachim Wuensche. A simple and model-free path filtering algorithm for smoothing and accuracy. In *2023 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–7. IEEE, 2023. 3

[11] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, 2012. 5

[12] Chien-Yao Wang and Hong-Yuan Mark Liao. Yolov9: Learning what you want to learn using programmable gradient information. *arXiV*, 2024. 3

[13] Jianyuan Wang, Minghao Chen, Nikita Karaev, Andrea Vedaldi, Christian Rupprecht, and David Novotny. Vggt: Visual geometry grounded transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025. 1, 2

[14] Yuxi Xiao, Jianyuan Wang, Nan Xue, Nikita Karaev, Yuri Makarov, Bingyi Kang, Xing Zhu, Hujun Bao, Yujun Shen, and Xiaowei Zhou. Spatialtrackerv2: 3d point tracking made easy. *arXiv preprint arXiv:2507.12462*, 2025. 2

[15] Dong Zhuo, Wenzhao Zheng, Jiahe Guo, Yuqi Wu, Jie Zhou, and Jiwen Lu. Streaming 4d visual geometry transformer. *arXiv preprint arXiv:2507.11539*, 2025. 2