Hochschule Konstanz
Fakultät Informatik

Deformable Object Tracking

HTWG Konstanz: Teamprojekt WS 24/25 und SoSe 25 in Kooperation mit Subsequent

Arian Braun, Leonie Fauser, Jonas Kaupp, Lukas Reinke, Yoshua Schlenker

Konstanz, 31. August 2025

Contents

Introduction **3**

Methodology and technical approach **3**
Previous papers . . . 3
Comparison of existing tracking methods . . . 5
TAPIR . . . 5
CoTracker . . . 7
SpatialTracker . . . 8
Adaption and implementation . . . 9
Sliding window approach (online version) . . . 9
Video Depth Anything . . . 11
Mask Selection with Segment Anything and CLIP . . . 13
Data generation . . . 16
Ground truth . . . 16

Results and Evaluation **18**
Metrics . . . 18
Runtime evaluation . . . 18
2D tracking efficiency . . . 19
Comparison between TOF ground truth and tracker . . . 21
Results . . . 22
Runtime evaluation . . . 22
Analysis of RGB vs. RGBD Processing Performance . . . 24
2D tracking efficiency . . . 27
Comparison between TOF ground truth and tracker . . . 30

Conclusion **32**
Future work . . . 33

Introduction

Deformable object tracking is a challenging problem in computer vision, as it requires robustly estimating the position, shape, and motion of objects whose geometry changes over time. Unlike rigid object tracking, where the spatial configuration remains constant, deformable objects can undergo non-linear transformations such as bending, stretching, or twisting. These deformations significantly increase the complexity of the tracking process, especially in real-world environments where occlusions and lighting variations are present.

Accurate tracking of deformable objects has numerous applications, ranging from robotic manipulation of flexible materials and human motion analysis to biomedical imaging and augmented reality. Recent advances in machine learning, particularly deep neural networks, have demonstrated strong potential for capturing the complex dynamics of non-rigid motion. However, achieving high tracking accuracy in dynamic and unconstrained scenarios remains an open research challenge.

The goal of this project is to research and develop a system for tracking deformable objects, specifically sports equipment like resistance bands and gymnastic balls. The goal is to support physical therapists and their patients in their exercise execution with an automated, AI-driven solution.

Therefore, possible approaches are discussed in chapter 2. Disregarding most of them because they did not fit the requirements leads to one approach called SpatialTracker (Paper SpatialTracker) which we will be focusing on during the future work. Also in this chapter, the original model will be adapted, e.g, with a sliding window approach for loading longer videos and an automated segmentation for selecting the region of interest. The results and evaluation of the work will be stated in chapter 3. Two different metrics and the runtime depending on video quality and grid size of the points to track are evaluated. In chapter 4 the work is summarized and a brief outlook in future work is given.

Methodology and technical approach

In this section we will provide the theoretical knowledge and describe our progress throughout the two semesters.

Previous papers

In the beginning, the team was given five papers which dealt approximately with tracking non-rigid objects:

1. Huang et al., Tracking Multiple Deformable Objects in Egocentric Videos (Paper 1)

2. Zhang et al., Self-supervised Learning of Implicit Shape Representation with Dense Correspondence for Deformable Objects (Paper 2)
3. Henrich et al., Registered and Segmented Deformable Object Reconstruction from a Single View Point Cloud (Paper 3)
4. Wang et al., Neural Textured Deformable Meshes for Robust Analysis-by-Synthesis (Paper 4)
5. Xiao et al., SpatialTracker: Tracking Any 2D Pixels in 3D Space (Paper 5)

The task was to analyze whether one of the papers provides a useful approach for our task. Therefore, let us first consider the goals for this project:

1. Target objects of the tracking algorithm are resistance bands, gymnastic balls and smaller soft balls.
2. The work is motivated by the need to support physical therapy patients through the monitoring and assessment of correct exercise execution.
3. Most of the time one of the above mentioned objects is present in the frame. We focus more on the quality of tracking than on several objects.
4. Runtime of the approach is initially secondary.
5. The training data is mainly in 3D.

So let us have a short look on the first four papers.

Paper 1 by Huang et al. focuses on tracking multiple deformable objects in egocentric videos where the camera is mounted on a person. The method relies on template matching with learned features to track objects. This approach is problematic for tracking things like resistance bands because they undergo significant and often unpredictable deformations, including self-occlusions and changes in topology (e.g. twisting). This approach cannot handle such extreme changes as the movement of resistance bands and gymnastic ball is too complex.

Zhang et al.'s work (Paper 2) addresses self-supervised learning of implicit shape representations for deformable objects. The method is designed for learning a 3D representation from multiple views or a canonical pose. The primary goal is to establish dense correspondence for a single object, not to track its position and deformation in a dynamic video sequence. Furthermore, the objects considered in this paper, e.g. animals, are typically less flexible than a resistance band, making it unsuitable for the extreme deformations of a resistance band or a soft ball. The method would likely be too slow and would not generalize well to the continuous, unpredictable movements of sports equipment.

The paper 3 by Henrich et al. contributes to reconstructing and segmenting deformable objects from a single-view point cloud. This method is primarily concerned with 3D reconstruction, not tracking. It takes a static snapshot (a single point cloud) and tries to reconstruct the object's shape and segment it. This is a fundamentally different problem from tracking an object's movement and deformation over a sequence of video frames. While it deals with deformable objects, its single-view, static-frame nature

makes it completely unsuitable for the continuous, dynamic process of tracking gym equipment in motion.

Wang et al. propose neural textured deformable meshes for robust analysis-by-synthesis. This paper focuses on reconstructing and tracking deformable objects with a pre-trained template mesh. The method requires a canonical, or rest, shape of the object. While it can track deformations, it's designed for objects whose topology remains relatively stable and predictable, such as a face or a piece of clothing. A resistance band's deformation is often so complex that it can change its topology, twisting and folding in ways that are difficult to model with a single pre-defined mesh template. A ball, while simpler, can also be challenging due to its smooth, textureless surface, which provides few features for the method to attach to. This approach would likely fail when the object undergoes extreme changes not captured by the initial mesh.

As a short conclusion, it can be said that the first four approaches are not useful for achieving the desired results mainly due to the extreme deformation of the objects. Therefore, all of the mentioned papers except the SpatialTracker are dismissed in our future work.

Comparison of existing tracking methods

As the only option being left over is the SpatialTracker.This is the one we focus on in the next steps. In the paper of the SpatialTracker, the performance was compared with a few other tracking models. We decided to have a closer look on two of them because we considered them as relevant in the beginning. As for this, in the following paragraph, we focus on below listed models:

- Karaev et al., CoTracker: It is Better to Track Together (Paper CoTracker)
- Doersch et al., TAPIR: Tracking Any Point with per-frame Initialization and temporal Refinement (Paper TAPIR)
- Xiao et al., SpatialTracker: Tracking Any 2D Pixels in 3D Space (Paper SpatialTracker)

TAPIR

TAPIR is a deep neural network model designed for the task of **Tracking Any Point** (TAP). Its main goal is to accurately follow a specific point of interest throughout a 2D video sequence, even if that point is on a deformable object, becomes occluded, or changes its appearance. It works in two steps: **per-frame initialization** and **iterative refinement**. The figure below shows the architecture of TAPIR.

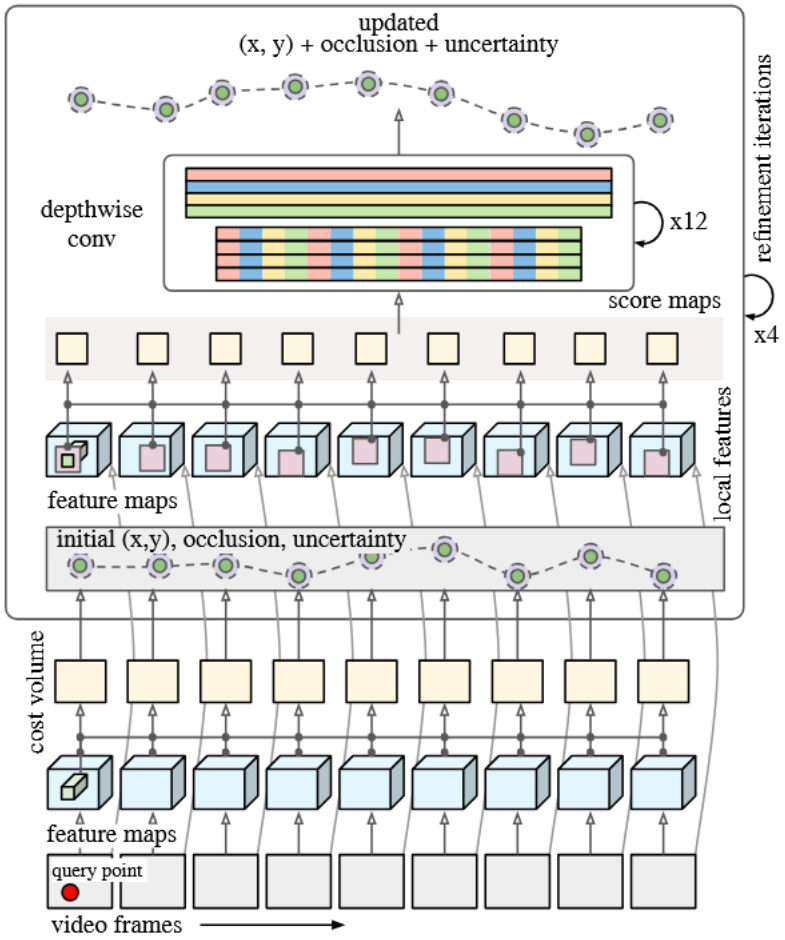


Figure 1: Tapir architecture

The lower part of the figure shows the **per-frame initialization**. It focuses on finding potential matches for a given query point in each new frame of the video. This step is designed to be robust to the challenges of deformable object tracking. The key idea is to use a matching network (CNN) that compares the query point's features with the features of every other pixel in the target frame. This step outputs three initial values:

- initial guess for the trajectory (x, y)
- probability for occlusion
- uncertainty probability

In the second stage, the **iterative refinement**, which can be seen on the upper part of the figure, the above mentioned initial guesses are improved. For each potential position found during the initialization phase, the model defines a local neighborhood window around it. This window acts as a search area for refining the point's exact location. The refinement itself is achieved by comparing the visual

features within this neighborhood to the features of the original query point. This comparison generates score maps indicate the similarity between the query point's features and every pixel within the neighborhood window. The highest-scoring pixel in this map represents the most probable refined position for the tracked point in that specific frame. This process is executed iteratively, allowing the model to correct small errors and ensure that the final trajectory is smooth and consistent over time.

Pros and Cons Tapir tracks points well. It provides smooth motion tracking that detects even the smallest changes. Only in cases of significant occlusion or excessively fast movements is the point lost. Tapir can also be run in real time. However, a major disadvantage is that only 2D trajectories are estimated and the lifting to 3D is not considered.

CoTracker

CoTracker is a paper that introduces another approach to point tracking in 2D videos. The core idea is that instead of tracking each point independently, it is more effective to track many points jointly, taking into account their dependencies and correlations. In the figure below the architecture is shown.

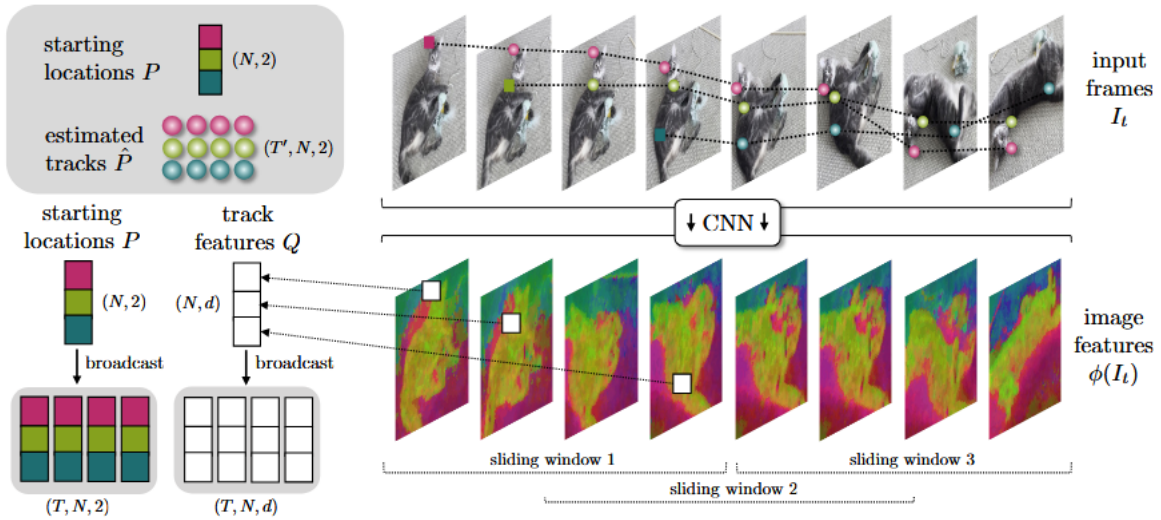


Figure 2: CoTracker architecture

Unlike TAPIR that treats each point's trajectory as an independent problem, CoTracker uses a Transformer-based network to model the relationships and dependencies between multiple points simultaneously. The heart of this system is a powerful attention mechanism that enables the network to exchange information between different tracks and across various time steps within

a given window of frames. This ability to collectively reason about the motion of multiple points makes CoTracker exceptionally robust. Furthermore, CoTracker is an online algorithm, meaning it can process video frames in real-time. To handle very long videos, it uses a sliding window approach. To achieve this, the model is trained in an unrolled fashion, like a recurrent neural network. The predictions from one window are used to initialize the tracks for the next overlapping window. This allows the model to maintain track consistency and accuracy over long durations.

The output of CoTracker is:

- Trajectory (x, y) over all frames
- occlusion probability

Pros and Cons By jointly tracking many points with a Transformer, CoTracker exploits inter-point correlations, which improves robustness to occlusions and even when points leave the field of view; it also scales to tens of thousands of points on a single GPU. Its main drawbacks are sensitivity to domain gaps—trained largely on synthetic data, it can mis-handle reflections/shadows—and, by design, joint attention over many points entails higher compute/memory than independent per-point trackers.

SpatialTracker

SpatialTracker is a method that uses the CoTracker approach and extends its 2D point tracking to the 3D domain. As input data it uses either 2D videos (RGB) or videos with depth information (RGBD). The architecture can be found below.

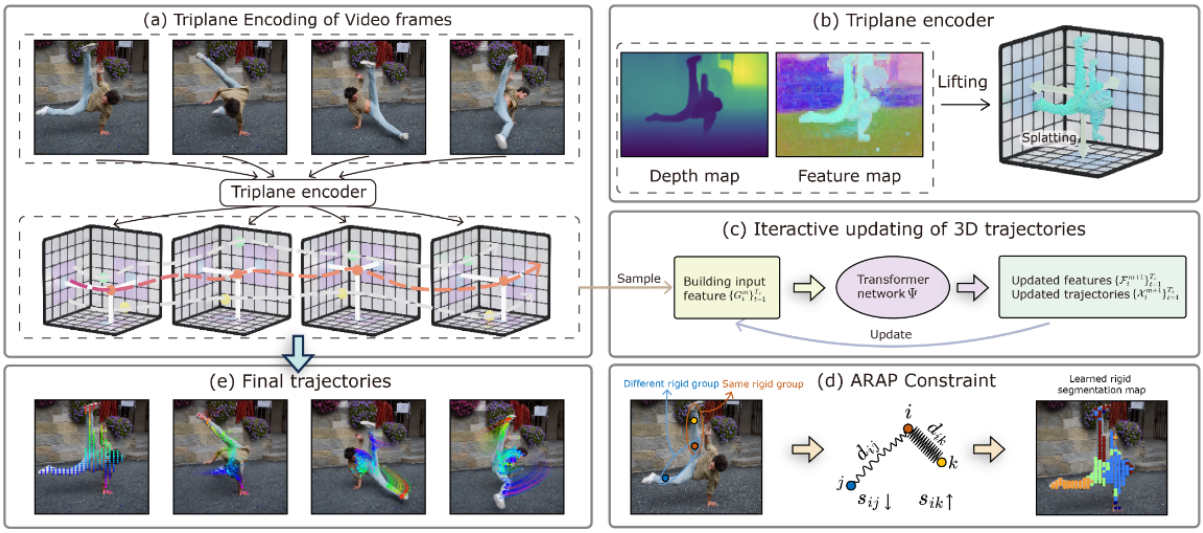


Figure 3: SpatialTracker architecture

SpatialTracker begins by estimating a depth map for each video frame and extracting dense image features, which are used to lift 2D pixels into 3D space to form a point cloud, see (a). As the monocular depth estimator (MDE) ZoeDepth is used. These 3D points are then projected onto three orthogonal planes to create a compact triplane feature representation that enables efficient feature retrieval (b). An iterative transformer network refines the 3D trajectories of query points across short temporal windows, using extracted features from the triplanes as input (c). As a last step, the model learns a rigidity embedding that groups pixels with similar rigid motion. An As-Rigid-As-Possible (ARAP) constraint is then applied. The ARAP constraint enforces that 3D distances between points with similar rigidity embeddings remain constant over time.

while a rigidity embedding combined with an As-Rigid-As-Possible (ARAP) constraint enforces locally consistent motion.

The output of SpatialTracker is:

- 3D trajectory (x, y, z)
- occlusion probability

Pros and Cons By lifting pixels to 3D (monocular depth → triplanes) and enforcing a rigidity embedding with an ARAP prior, it is robust to occlusions and out-of-plane motion and achieves strong results on long-range tracking benchmarks. Its accuracy depends on the quality/consistency of the depth input, and the pipeline incurs additional compute/memory from depth inference, triplane feature construction, and transformer updates compared to purely 2D trackers.

Adaption and implementation

As a result of the SpatialTracker providing the best tracking performance (as it can be seen in the videos in the GitHub) and also provides 3D data as output, we only focus on the SpatialTracker in follow-up work. The other models can considered in future research. This section gives an overview of the adaption we made to the already provided code base: Github SpatialTracker.

Sliding window approach (online version)

To make SpatialTracker operate reliably on long and/or high-resolution sequences, we extended the original `demo.py` into a chunked online variant (`chunked_demo.py`). Instead of processing the entire clip at once, the video is split into temporal chunks of length `--chunk_size`. For each chunk we prepend a small overlap equal to half of the model's sequence length (`--s_length_model / 2`).

The model is run on *overlap + chunk*, but only the predictions belonging to the non-overlap part are retained. This keeps peak memory usage approximately constant while preserving sufficient temporal context at chunk boundaries.

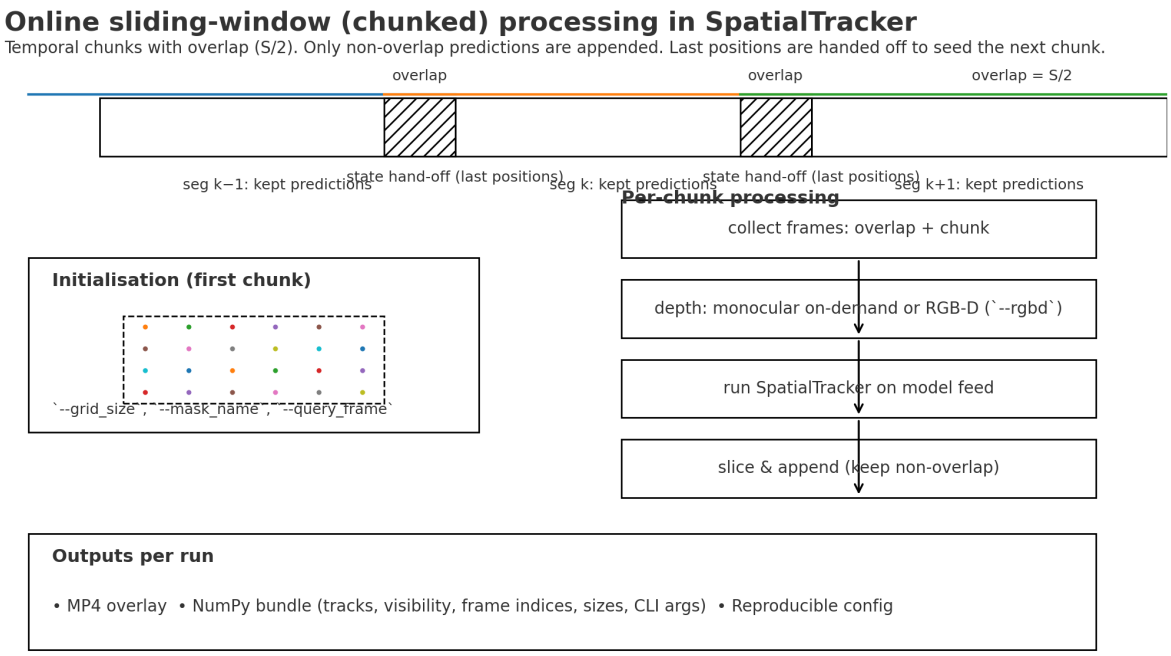


Figure 4: Online sliding-window processing

Initialisation

In the first processed segment, query points are initialised on a regular grid restricted to an optional segmentation mask (`--grid_size`, `--mask_name`). For subsequent segments we do not re-sample; instead, the last predicted positions from the previous segment are used as the queries at the new segment start. In practice this yields stable identities and avoids repeated mask processing. If no valid points are available (e.g., prolonged occlusion), the pipeline proceeds with empty/dense queries until tracks re-emerge.

Depth handling

The script supports both monocular and RGB-D inputs. By default, monocular depth is computed on demand for the frames inside each model call. When `--rgbd` is set, per-frame depth maps (pre-aligned to the RGB preprocessing) are injected directly, bypassing the MDE. This path is used for our ToF-based comparisons.

Preprocessing and outputs. We optionally sub-sample frames (`--fps`) and apply crop/downsample

operations (`--crop`, `--crop_factor`, `--downsample`) before inference; all trajectories are mapped back to the overlay/original resolution when saving. Each run exports an MP4 with overlays and a NumPy bundle with trajectories, visibility flags, frame indices, spatial metadata, and the full CLI configuration to ensure reproducibility.

Key arguments

- `--chunk_size`: frames per output segment.
- `--s_length_model`: model window; overlap is half of this value.
- `--grid_size`, `--mask_name`, `--query_frame`: first-segment initialisation.
- `--rgbd`: use external depth instead of monocular depth.
- `--fps`, `--downsample`, `--crop`, `--crop_factor`: temporal/spatial preprocessing.
- Visualisation controls: `--point_size`, `--len_track`, `--fps_vis`, `--backward`, `--vis_support`.

Example

```
python chunked_demo.py \
  --root ./assets \
  --vid_name Gymnastik_1_5s \
  --mask_name Gymnastik_1_5s_mask.png \
  --grid_size 50 \
  --chunk_size 30 \
  --s_length_model 12 \
  --fps 1.0 \
  --downsample 0.8 \
  --rgbd
```

Video Depth Anything

This section deals with depth estimation models of the SpatialTracker. To choose between different MDEs one can specify the argument `--depth_model`. The argument gets then forwarded to the `mde.py` file. All functionality for loading the MDE model and generating depth maps is implemented in this file. This script supports various state-of-the-art monocular depth estimation models such as MiDaS, ZoeDepth, Metric3D, Marigold, and Depth-Anything. Depending on the user-defined argument, the corresponding model is loaded in evaluation mode and used to infer depth maps from single RGB images. The inference process produces either relative or metric depth, and in some cases, such as with Depth-Anything, additional alignment steps are performed to calibrate the predicted relative depth to metric scale using a reference model. The script also converts the estimated depth maps into 3D point clouds using the camera intrinsics. The resulting 3D reconstructions are exported as colored point clouds in .ply format, which enables further processing.

By default the ZoeDepth estimator is used. The architecture is shown below.

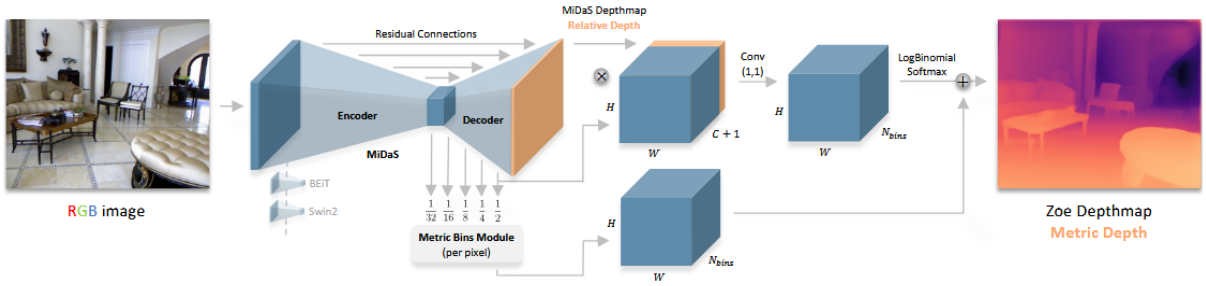


Figure 5: ZoeDepth architecture

ZeoDepth consists of two stages. First, the model is extensively pre-trained on a vast amount of data to understand relative depth, learning which objects in a scene are closer or farther from each other without worrying about specific units of measurement. Second, the model is then fine-tuned on smaller datasets that contain ground truth metric depth (exact distances in meters). Instead of simply learning a single, precise value, ZoeDepth uses a unique **metric bins module** that estimates a range of possible depth values for each pixel. In the end, ZoeDepth outputs metric depth maps.

For our work we extended the SpatialTracker by another depth estimator. We were recommended Video Depth Anything: Consistent Depth Estimation for Super-Long Videos (Paper VideoDepthAnything) by Chen et al. The depth estimator can be chosen by setting the argument `--depth_model` to either `zoe` or `depth_anything` in the file `chunked_demo.py`.

Video Depth Anything is build upon the strengths of Depth Anything to handle long-duration video sequences with high quality and temporal consistency. The authors introduce a lightweight **spatiotemporal head** on top of the Depth Anything V2 encoder that allows the network to share information across consecutive frames. Instead of relying on optical flow or camera pose (absolute depth values), they introduce a simple temporal consistency loss that encourages smooth changes in depth across frames. For long videos, the method processes clips segment by segment. The figure below illustrates the processing pipeline.

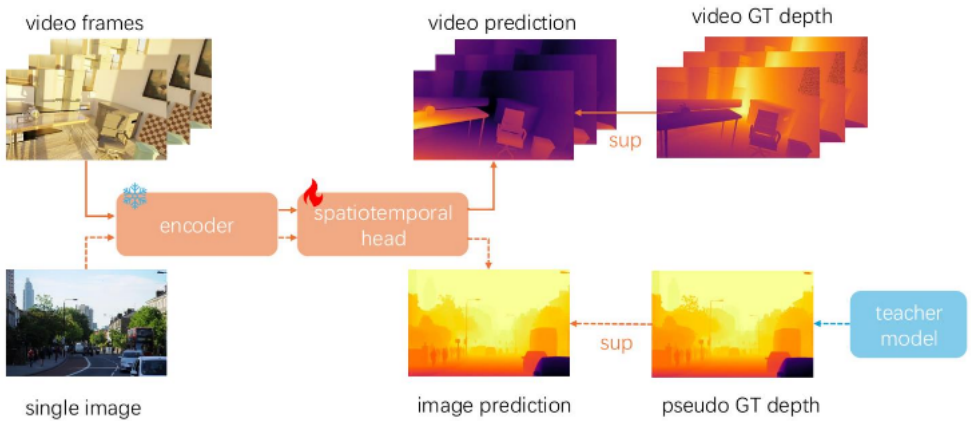


Figure 6: Video Depth Anything architecture

Mask Selection with Segment Anything and CLIP

To determine the initial positions of the tracking points, we generate masks that detect and highlight the sports equipment. For implementation, Meta AI's Segment Anything Model (SAM) [Github SAM] is used to generate segments, and then Clip (Paper CLIP) is used to select the most suitable segment for mask creation.

Context

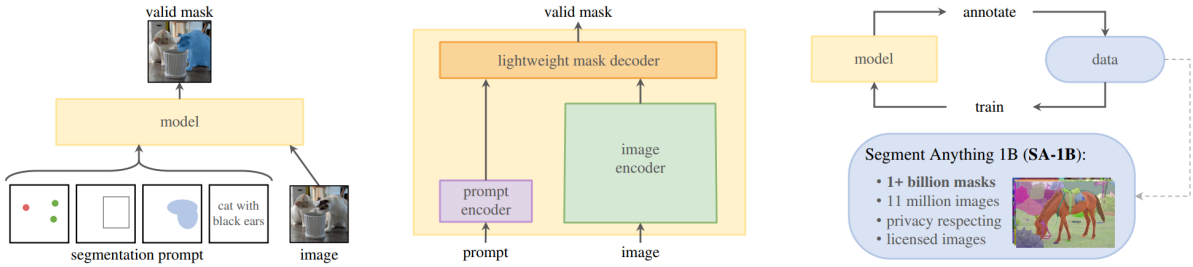


Figure 7: SAM Components

The Segment Anything Model is a model for creating segments. SAM takes an image as input, optionally with prompts such as points, boxes, or masks. The image is processed by an image encoder to extract feature embeddings, which are then combined with the prompt embeddings. A mask decoder then predicts the pixel-precise masks from the combined embeddings. The whole thing was pre-trained on a dataset with over 11 million images, so that the model can now also create a mask for unknown objects (zero-shot generalization). The masks are output as well as a score that indicates

the confidence. Text prompts are planned for use as prompts, but are not yet available. We therefore decided not to use prompts, so that segments are placed over the entire frame.

We then use CLIP (Contrastive Language-Image Pretraining) to select the most semantically appropriate mask. To do this, we selected reference images of sports equipment and used CLIP to obtain an embedding vectors for each reference image, representing their visual features. The cosine-based similarity is calculated for each segment and reference image embedding. The mask with the highest similarity value is then selected as the result:

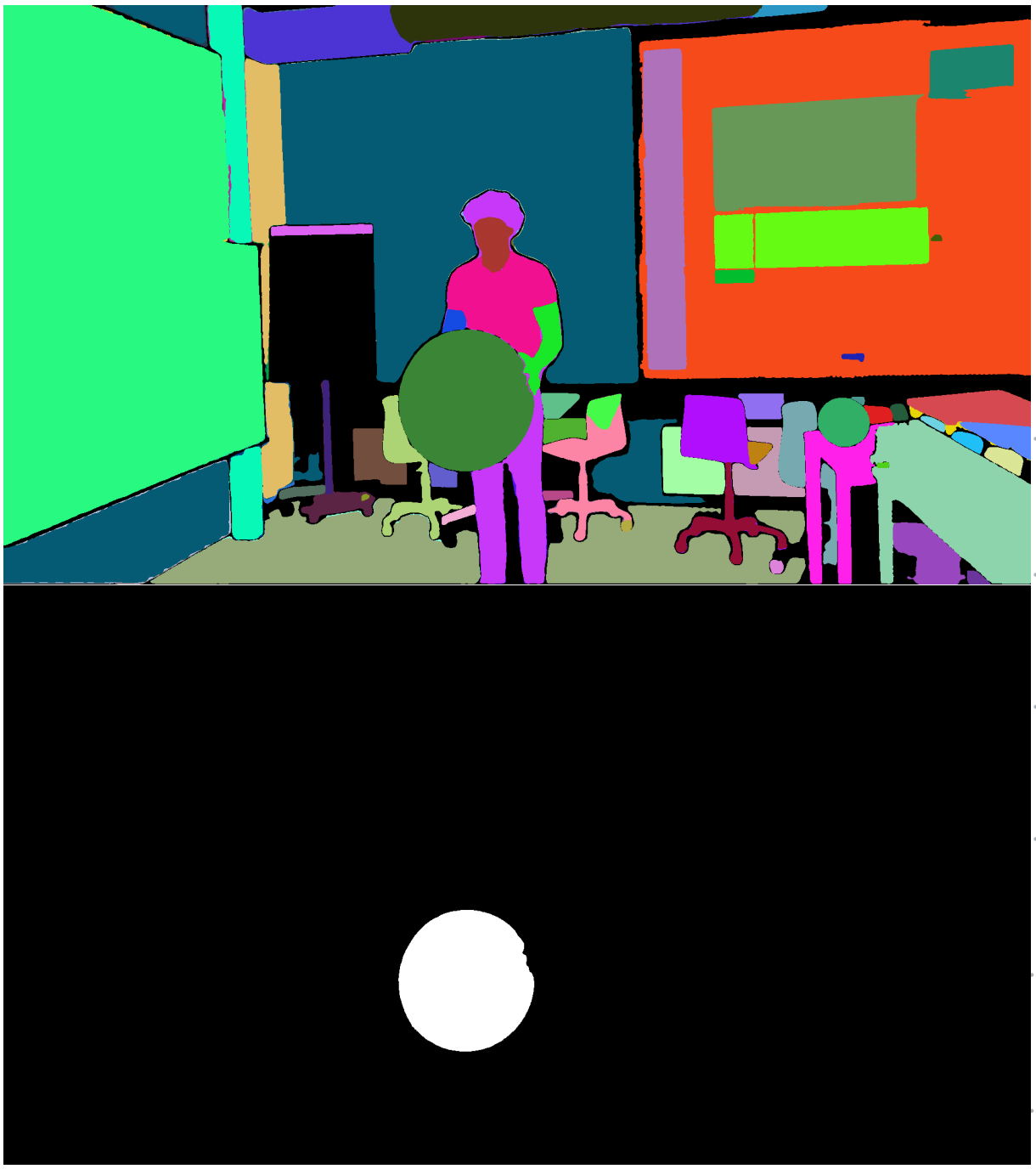
```
clip_model, preprocess = clip.load("ViT-B/32", device=device)
for i, mask_data in enumerate(masks):
            seg_mask = mask_data["segmentation"]
            seg_embed = encode_image_clip(clip_model, preprocess,
                seg_mask, device)
            score = max(torch.nn.functional.cosine_similarity(seg_embed
                , ref_embed).item()
                  for ref_embed, _ in ref_embeds)
```

Usage

Parameters to be selected as input:

`--video-path`: The path to the video. `--reference-images`: Folders containing reference images of objects to be detected. `--use-center-priority`: If set, the segment is selected based on its position (otherwise based on its shape). Priority is given to the most central location.

The program is started and a mask for the segment with the best score is selected and displayed. You can then interactively decide to select the mask with the next worst score, and so on. This allows you to select the appropriate mask even in cases of ambiguity.



Difficulties/Evaluation

We chose a gym ball, a sponge, and a softball as reference images. These objects are detected correctly. There are difficulties in detecting the resistance band, as its shape in the first frame is ambiguous and not very distinctive. As a result, other objects such as the door handle and those with small rectangular shapes are often detected incorrectly. We therefore implemented an approach that se-

lects the segment with the smallest distance from the center.

Data generation

To be able to evaluate our implementation in the next section, we need a data set. Therefore, we recorded approximately 50 videos with different features:

- Objects
  - Resistance band
  - Gymnastic ball
  - Soft ball
  - Sponge
- Video resolution
  - 1080p
  - 720p
  - 360p
- Video length
  - between 5 seconds and 30 seconds

Ground truth

To obtain reliable ground truth depth information for evaluating the tracking performance, depth images were recorded using a Time-of-Flight (ToF) camera. In our case the Femto Bolt from Orbbec was used, see below.

Figure 8: Femto Bolt depth camera

As the ToF camera outputs only individual frames, both the color stream and the depth stream had to be synchronized and merged to obtain a temporally continuous RGB-D video. This preprocessing is done by a python script (`preprocessing.py`). The camera uses an IR-sensor to generate the depth images which can be read out as a .raw file which is then converted in a NumPy array format (.npy) to ensure efficient loading and processing. Additionally, the frames of the video stream are concatenated to obtain a .mp4 video.

During experimentation, the `--rgbd` argument can be set when running the `demo_chunked.py`. It then feeds this pre-recorded depth information directly into the SpatialTracker, bypassing the monocular depth estimation module. This setup enables a direct comparison between the model's predicted depth and the actual measured depth from the ToF camera, thereby providing a robust basis for assessing the accuracy and reliability of the tracker's performance on deformable objects.

Results and Evaluation

In this chapter we want to describe our evaluation process. It was a challenge to come up with metrics which can accurately measure the tracking performance. All the metrics used in the three papers of TAPIR, CoTracker and SpatialTracker did not fit really well for our purpose. In the end, we agreed on two metrics which we will describe in the first place. Afterwards, the results of each metric are discussed. In addition, the runtime is evaluated.

Metrics

Runtime evaluation

For evaluating runtime performance, a systematic approach was designed to capture the computational load of the tracking process under different conditions. Both RGB and RGB-D data were considered to analyze differences in resource usage and efficiency.

Experimental Setup For each object recorded with the ToF camera, videos with varying length and resolution were used:

Video durations: 5 s, 10 s, 30 s

Resolutions: 720p, 360p

Grid sizes for SpatialTracker: 20, 50, 100

Measurements were performed once with pure RGB videos and once with RGB-D videos to account for the potential additional computational load from depth data.

Data Collection The runtime was measured using the Performance_Metrics class, which continuously records:

CPU usage (%)

RAM usage (percentage and absolute GB)

Disk I/O (MB read and written)

GPU utilization (%) and memory usage (if available)

The metrics were collected throughout the entire tracking run. Optionally, py-spy can be used for periodic CPU profiling to identify performance bottlenecks in the Python code.

Analysis Approach After completing the tracking runs, the collected metrics were used to generate visualizations illustrating:

Temporal evolution of system resources: CPU, RAM, disk, and GPU usage over time.

Comparison across grid sizes: Efficiency and resource peaks for 20, 50, and 100 tracking points.

Comparison across resolutions: Effect of 720p vs. 360p on runtime and resource usage.

Comparison between RGB and RGB-D data: Additional computational load introduced by depth information.

The evaluation results are presented as clear plots, enabling direct inspection of the relationships between resolution, grid size, data type (RGB vs. RGB-D), and resource consumption.

2D tracking efficiency

With this approach the tracking efficiency in the 2D space is evaluated. For this a short introduction into ArUco markers, as shown in the figure below, is needed. ArUco markers are binary square markers widely used in computer vision for camera pose estimation and object localization. Each marker consists of a unique black-and-white pattern which ensures robust detection under varying lighting conditions. By identifying the specific ID encoded in the marker's pattern, computer vision algorithms can distinguish between different markers in a scene. In our tracking application, the detection of the position of the ArUco markers establish a ground truth trajectory which can then be used to compare against the real trajectory from the SpatialTracker.

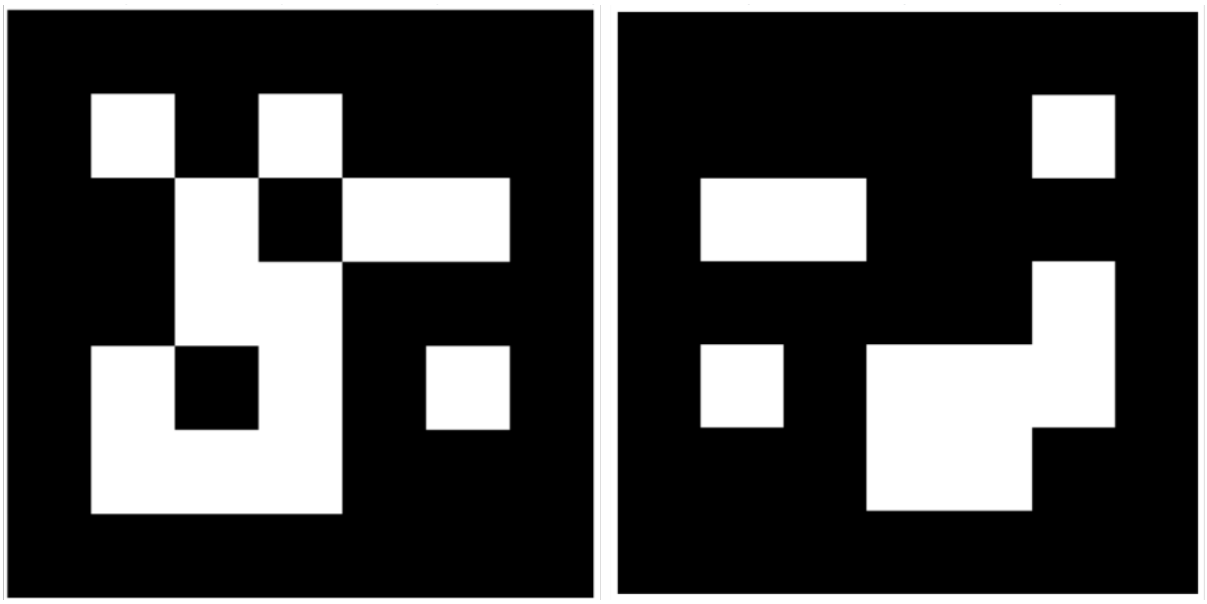


Figure 9: 5x5 ArUco markers

The figure below presents a schematic overview of the approach for one single frame. This setup is for demonstration purposes only and does not represent an actual experiment.

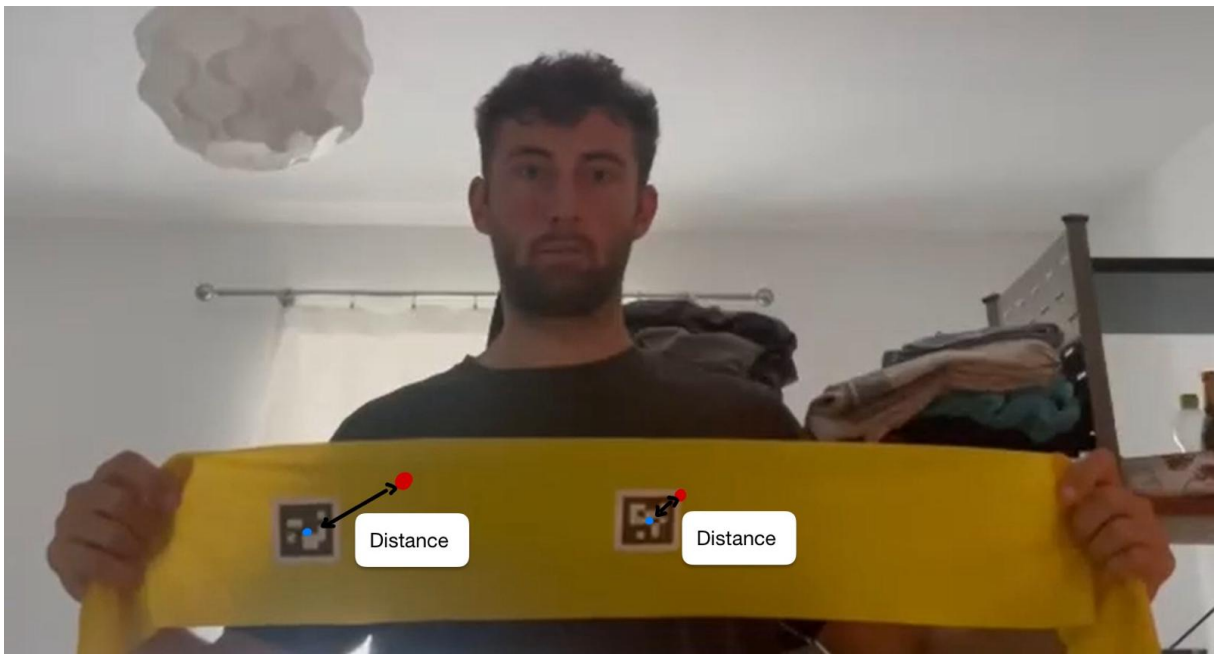


Figure 10: Schematic 2D evaluation with ArUco markers

The experimental setup involved recording three video sequences, each the two ArUco markers shown above in a size of 4cm x 4cm attached to a resistance band which is recorded approximately from a distance of 1,5 meters. The setup includes three levels of difficulty in handling the resistance band:

1. Movement only – the band is simply moved without deformation.
2. With stretching – the band is actively stretched, introducing shape changes.
3. With occlusion – the ArUco markers are fully occluded during motion. The arm is stretching so that the other side of the Theraband is visible and the markings are completely covered for 1 second.

Each video was first processed using a Python script (`arucoDetection.py`) which detects the ArUco markers in every frame. For each detected marker, the pixel coordinates of its midpoint were calculated and stored as a NumPy array file (.npy) for subsequent analysis. In above figure these are the blue points.

In the next stage, the same video was fed into the SpatialTracker. The points selected for tracking were the midpoints of the ArUco markers in the first frame of the video. SpatialTracker then computes the corresponding 3D trajectories of the points over time which were exported in JSON format for further evaluation. In the schematic these are represented by the red circles.

In the python script `evaluateTracking.py`, both stored files are loaded. The Euclidean distance between the pixel coordinates obtained from the SpatialTracker and those from the ArUco marker detection is then computed for each frame. The tracking evaluation produces several quantitative metrics, namely the maximum error, the root mean square error (RMSE), and the standard deviation of the errors. A larger distance indicates poorer tracking performance, whereas a smaller distance corresponds to better accuracy.

Comparison between TOF ground truth and tracker

We implemented an end-to-end pipeline (`eval_pipeline.py`) to compare SpatialTracker outputs against **Time-of-Flight (ToF)** depth frames. Given a clip slug, the pipeline downloads and re-encodes the RGB video, derives a first-frame object mask (from the project's segmentation), runs the chunked online variant of SpatialTracker, fetches the corresponding ToF `.npy` frames, performs a depth-aware analysis, and generates a ToF overlay video with track visualization.

Coordinate mapping (processed → ToF) Let (x_p,y_p) be tracker coordinates at the processed resolution (W_{proc},H_{proc}), and (W_{tof},H_{tof}) the ToF frame size. We map to ToF pixel indices by

$$s_x = \frac{W_{\text{tof}}}{W_{\text{proc}}}, \quad s_y = \frac{H_{\text{tof}}}{H_{\text{proc}}}, \qquad (x_{\text{tof}}, y_{\text{tof}}) = \big(\,\text{round}(s_x\, x_p),\ \text{round}(s_y\, y_p)\big).$$

Mask warp (geometric prior) The first-frame binary mask is used only as a **geometric prior**. A homography H is estimated from points that started inside the mask and are visible at $t=0$ and t:

```
\begin{bmatrix}u\\v\\1\end{bmatrix}\sim
H\begin{bmatrix}x\\y\\1\end{bmatrix},\qquad
H=\text{RANSAC}\big(\{(x_0,y_0)\leftrightarrow(x_t,y_t)\}\big).
```

The mask is warped to frame t and slightly dilated to tolerate small warp errors. Membership in the warped mask counts as a positive vote.

Local ToF depth and object depth band For a projected point (x_{tof},y_{tof}) we take a **robust local depth** as the median in a small window, ignoring zeros (no return).

To decide whether a tracked point is consistent with the ToF signal, we maintain an **object depth band** per frame. The band is estimated from ToF samples and smoothed over time using an exponential moving average. In the overlay path, depths are sampled inside the warped mask; in the metrics path,

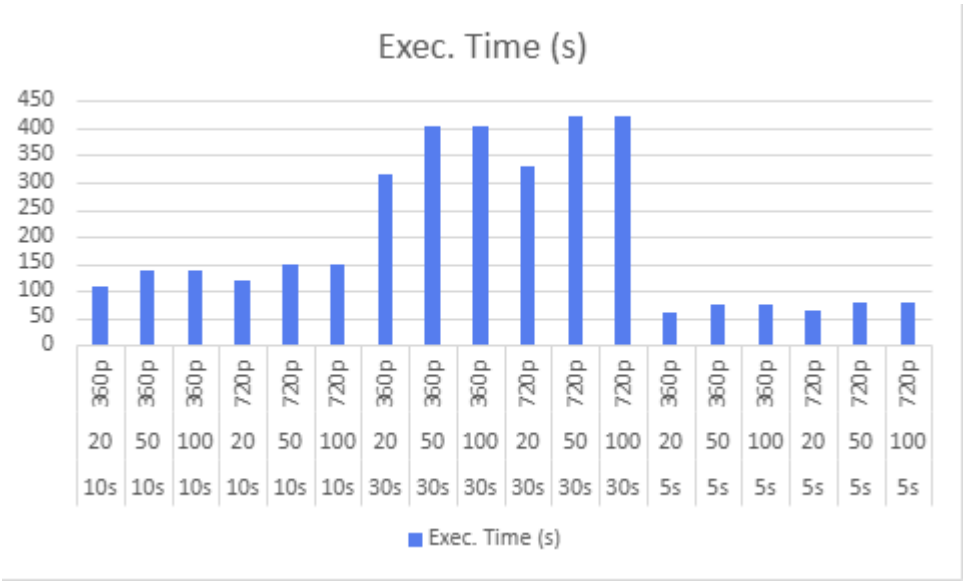
samples are taken at the current visible track locations (mask-agnostic). The band half-width uses the larger of a fixed minimum tolerance (≈6 cm) and a term proportional to the median absolute deviation (≈2.5×MAD). A point is accepted by depth if its local ToF median lies inside the current band.

Decision rule and outputs Geometry and depth are combined: a point counts as correct if it is **inside the warped mask** or **passes the depth test**. A short majority vote over recent frames reduces flicker from transient sensor dropouts. In the overlay, accepted samples are drawn **green**, rejected **red**, and optionally **yellow** if the tracker reports invisibility while ToF has no local return.

Results

Runtime evaluation

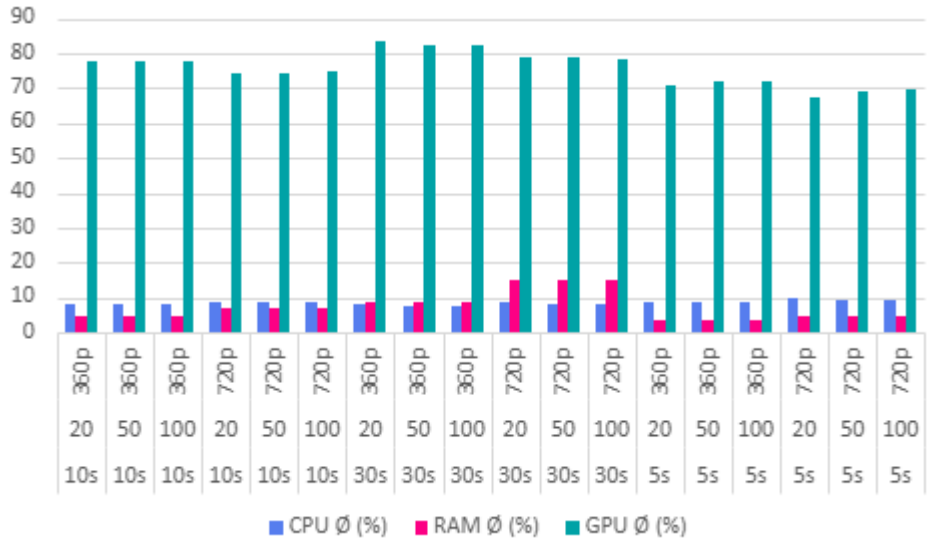
Execution Time The execution time increases with both video length and resolution. For 360p, the runtime for 30-second videos is approximately 402–313 seconds, depending on grid size. At 720p, the runtime for the same 30-second videos increases to 424–331 seconds. This shows that 720p consistently takes about 5–10% longer than 360p, regardless of grid size. The increase in execution time is due to higher computational requirements at higher resolution, even though GPU utilization is already saturated.



GPU Utilization and Memory GPU utilization remains very high, averaging between 70% and 83%, and regularly peaking at 100%. This indicates that the GPU is the main performance bottleneck during all runs. GPU memory usage also scales with video length and resolution. For 360p runs, peak

GPU memory usage reaches 22.8 GB, while at 720p it increases slightly to 23.3 GB. This suggests that higher resolution does not drastically increase GPU memory consumption, but does prolong GPU processing.

CPU Usage The CPU shows only moderate activity throughout all tests. Average CPU usage remains between 7% and 9%, with peak usage occasionally reaching 50–65%. This demonstrates that the CPU is not a limiting factor in the pipeline. The workload is dominated by the GPU, and CPU overhead is relatively stable across all configurations.



System Memory (RAM) System memory usage differs significantly between 360p and 720p. For 360p runs, RAM usage peaks between 8 GB and 22 GB. At 720p, RAM demand increases sharply, with peak usage reaching 45 GB during 30-second runs. This means that 720p resolution requires roughly twice as much RAM as 360p. Such high memory usage may exceed the capacity of typical workstations and therefore reduces scalability for longer 720p sequences.

Disk I/O Disk input and output activity remains relatively low compared to other resource demands. Data written to disk increases with video length, ranging from 35 MB for 5-second runs to about 200 MB for 30-second runs. Since both read and write operations remain small compared to modern disk bandwidth, disk I/O is not a performance bottleneck in any of the tested scenarios.

Dauer	Grid	Auflösung	Exec. Time (s)	Samples	CPU Ø (%)	CPU Max (%)	CPU Min (%)	RAM Ø (%)	RAM Max (%)	RAM Peak (GB)	Disk W (MB)	GPU Ø (%)	GPU Max (%)	GPU Mem Peak (GB)
5s	20	360p	58,59	232	8,9	41,9	4,9	3,7	4,3	7,86	35,17	71	100	20,27
10s	20	360p	109,39	433	8,4	50,9	4,9	4,7	5,3	10,51	45,89	77,8	100	21,33
30s	20	360p	313,24	1239	7,9	51,3	4,8	8,9	9,9	22,08	134,27	83,2	100	22,83
5s	50	360p	73,41	291	8,6	49,2	4,9	3,5	4,4	8,3	41,59	71,9	100	20,28
10s	50	360p	138,58	549	8,1	51,1	4,8	4,8	5,3	10,5	51,93	77,7	100	21,51
30s	50	360p	402,86	1594	7,6	52,3	4,9	8,9	10,1	22,52	106,77	82,1	100	22,56
5s	100	360p	73,51	291	8,6	40,2	4,9	3,8	4,5	8,36	35,87	71,8	100	20,28
10s	100	360p	138,22	547	8	51,4	4,9	4,6	5,4	10,66	52,16	77,7	100	21,51
30s	100	360p	402,1	1591	7,6	52,6	4,5	8,9	10,1	22,5	106,53	82,1	100	22,56
5s	20	720p	62,62	248	9,7	62,9	4,9	4,7	5,5	10,86	104,38	67,3	100	20,37
10s	20	720p	117,22	464	8,9	60,4	4,9	6,7	8	17,2	116,66	74,1	100	21,12
30s	20	720p	331,07	1310	8,7	64,3	4,8	15	19	44,95	114,57	79	100	22,87
5s	50	720p	77,63	307	9,2	56,5	5	4,8	5,8	11,69	54,84	69,3	100	21,07
10s	50	720p	147,51	584	8,7	68,1	4,9	6,7	8	17,25	132,45	74,5	100	21,33
30s	50	720p	422,8	1673	8,3	65,1	4,9	15	19,1	45,03	217,44	78,6	100	23,26
5s	100	720p	77,31	306	9,1	64,2	4,9	4,8	5,6	11,25	60,84	69,6	100	21,07
10s	100	720p	147,74	585	8,6	58,4	4,7	6,7	8	17,18	77,26	74,6	100	21,33
30s	100	720p	423,72	1676	8,3	64,8	4,9	15	18,9	44,64	218,88	78,5	100	23,26

Key Findings Execution time at 720p is consistently longer than at 360p, by roughly 5–10%.

The GPU is always fully utilized, making it the primary performance bottleneck.

The CPU does not limit performance, since its average load is low.

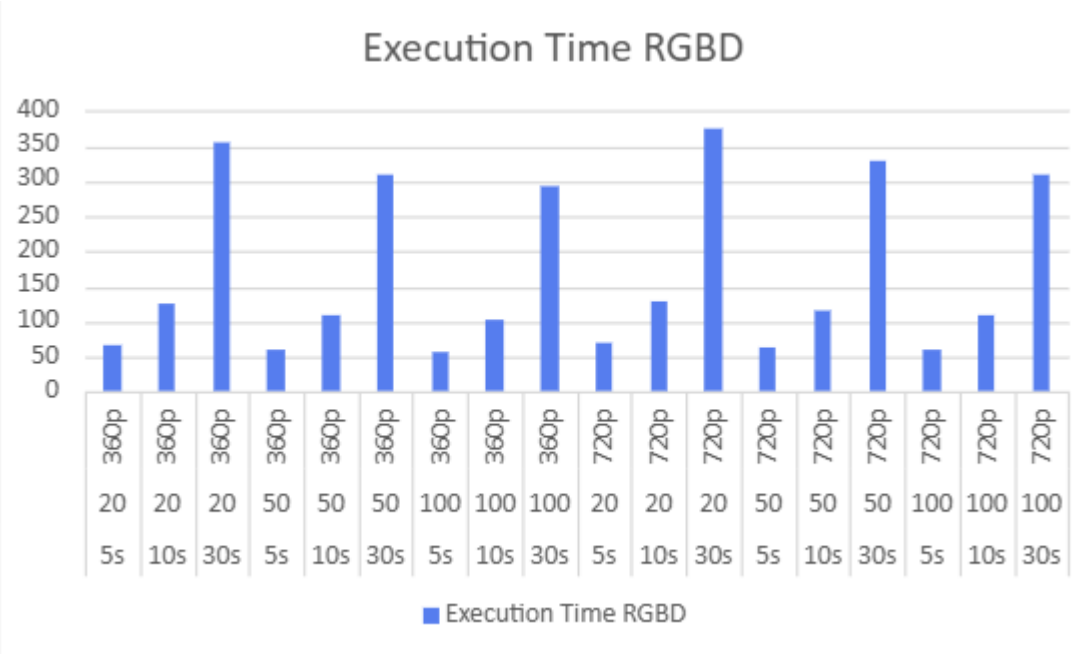
System RAM demand nearly doubles at 720p, which significantly limits scalability for longer videos.

Disk I/O has no negative impact on performance.

Analysis of RGB vs. RGBD Processing Performance

The dataset provides a comparison between RGBD (color + depth) and RGB-only video processing across different durations, grid sizes, and resolutions. The following sections summarize the main differences observed between the two modes.

Execution Time RGBD processing consistently requires more execution time than RGB-only processing. On average, RGBD runs take between 10 and 20 percent longer, depending on the resolution and grid size. This additional overhead can be attributed to the depth channel, which increases the computational workload.

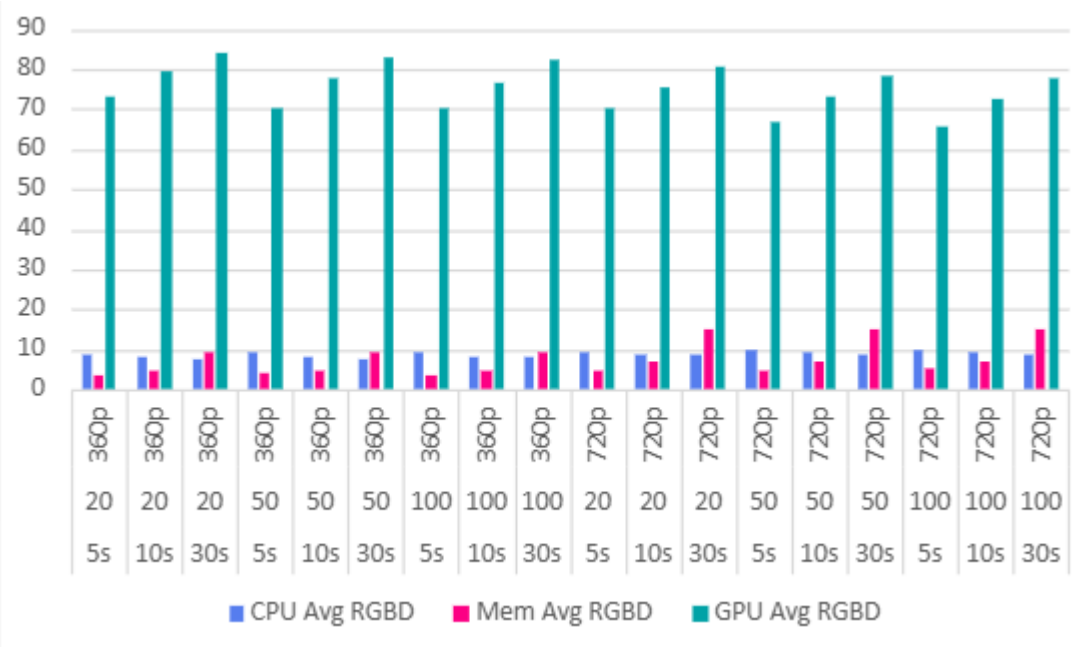


CPU Utilization The average CPU usage is very similar between RGBD and RGB, with differences usually within half a percent. Peak CPU usage is sometimes slightly higher for RGBD, but the difference is not dramatic. These results suggest that the CPU is not the main bottleneck when comparing RGBD and RGB processing. Both modes scale in a similar way with respect to CPU load.

Memory Consumption RGBD processing uses consistently more memory than RGB. The difference is typically between 200 and 500 MB on average. Peak memory usage also shows a similar gap, which reinforces the conclusion that handling depth data increases the memory footprint. For example, in the 10-second, 20-grid, 360p run, RGBD required about 10.6 GB, whereas RGB required about 10.5 GB. The memory gap becomes more noticeable at higher resolutions.

Disk I/O Disk reads are negligible for both RGBD and RGB. However, disk writes are consistently higher for RGBD. The difference is often between 20 and 40 MB more per run, which reflects the additional output data generated when depth information is included.

GPU Utilization RGBD also places a heavier load on the GPU than RGB. The average GPU utilization is consistently higher for RGBD, usually by one to three percent. In both modes, the GPU regularly reaches 100 percent peak utilization, which indicates that the GPU is the primary performance bottleneck. GPU memory usage is also higher for RGBD, with differences again in the range of 200 to 500 MB, due to the storage requirements of the depth channel.



Scaling Effects Increases in resolution from 360p to 720p lead to longer execution times and higher GPU memory consumption in both RGBD and RGB modes. However, the performance gap between them remains stable, meaning that the relative overhead of depth processing does not grow disproportionately with resolution. Similarly, increasing the grid size from 20 to 100 results in proportional scaling for both RGBD and RGB. Changes in duration, from 5 seconds to 30 seconds, primarily extend execution time in a linear fashion, with RGBD always requiring more time than RGB.

Effect of Video Length Video length has a strong and predictable influence on execution time. Longer videos require proportionally more processing time, resulting in a roughly linear increase. For example, a 30-second video requires approximately three times as much time as a 10-second video under the same conditions.

The key point is that the relative difference between RGBD and RGB remains stable across all durations. Whether the video is 5 seconds or 30 seconds, RGBD consistently requires more time and resources, but the overhead is always in the same range (10–20 percent more execution time and slightly higher memory/GPU usage).

Effect of Grid Size Grid size also significantly impacts performance. Increasing the grid size from 20 to 100 results in longer execution times and higher memory requirements for both RGBD and RGB. The more complex the grid, the more computational work is required.

However, just as with video length, the relative overhead of RGBD compared to RGB does not change. Both modes scale proportionally with grid size, and RGBD consistently consumes more time and resources by the same percentage margin.

Conclusions RGBD processing is uniformly more resource-intensive than RGB processing. It requires longer execution times, more memory, larger disk writes, and slightly higher GPU utilization. The GPU is the main bottleneck, as it often reaches maximum utilization in both modes.

The experiments confirm that both video length and grid size strongly influence absolute performance: longer videos and larger grids increase processing times and resource consumption. Nevertheless, the relative performance gap between RGBD and RGB remains stable, making the overhead of including depth data both consistent and predictable.

In practical terms, RGBD processing provides richer information at a stable cost: approximately 10–20 percent longer processing times, 200–500 MB higher memory usage, increased disk writes, and slightly higher GPU load compared to RGB-only processing, regardless of video length or grid size.

2D tracking efficiency

The following table summarizes the results for all three videos. Overall, we are looking at videos with an image size of 832x464 pixels:

Video / Condition	Marker ID	RMSE (pixels)	Std. deviation (pixels)	Max. deviation (pixels)
Movement only	0	2.53	1.34	7.69
	1	4.82	2.80	10.17
With stretching	0	2.29	1.40	6.24
	1	6.20	3.06	12.08
With occlusion	0	3.87	3.39	40.21
	1	41.59	34.04	88.28

There is a clear difference between the values of the two Aruko markers. Marker 0 is located in the middle of the band and therefore performs less stretching. Marker 1, on the other hand, is located at the edge.

The movement and stretching achieve very good values with a maximum deviation of 12 pixels. The tracking positions are therefore consistently close to the marker. This is also evident from the low standard deviations.

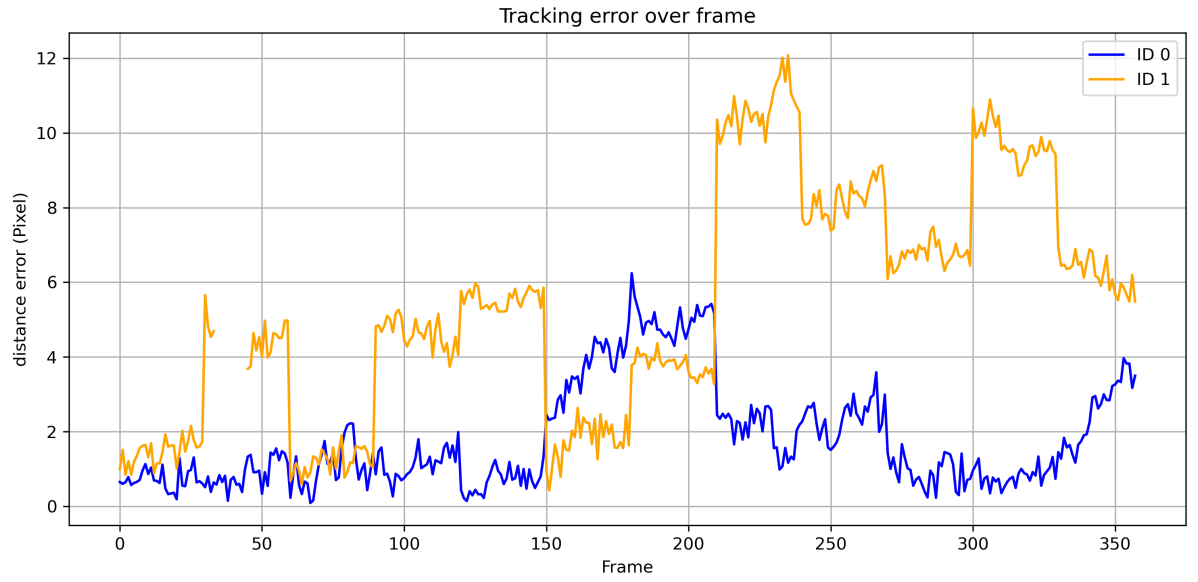


Figure 11: Tracking errors during stretching

Figure 12: Tracking error during movement

There are problems with occlusion. The maximum deviation is large and persists throughout large parts of the video. The following figure shows the progression of the deviations. When the markers are occluded, the position information is lost. The tracker then assumes incorrect position values for both markers. When the marker becomes visible again, the tracker only recognizes marker 0 and the

correct trajectories are determined there again.

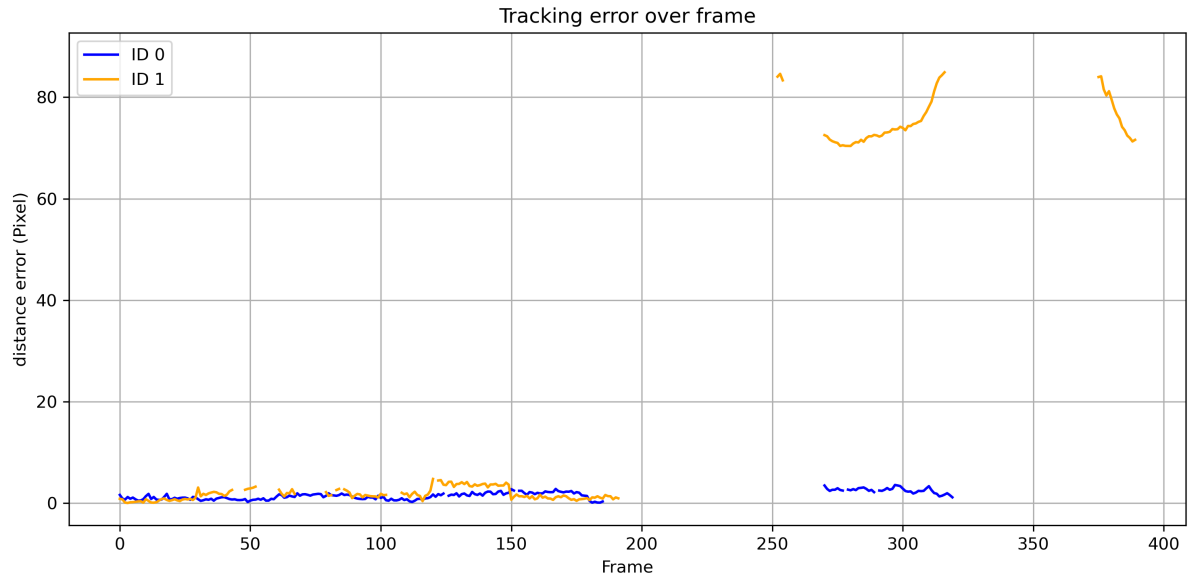


Figure 13: Error deviation during occlusion

The detection of occlusion is faulty. Occlusion is detected for a short period of time, but then another visible point is tracked (see following figure).

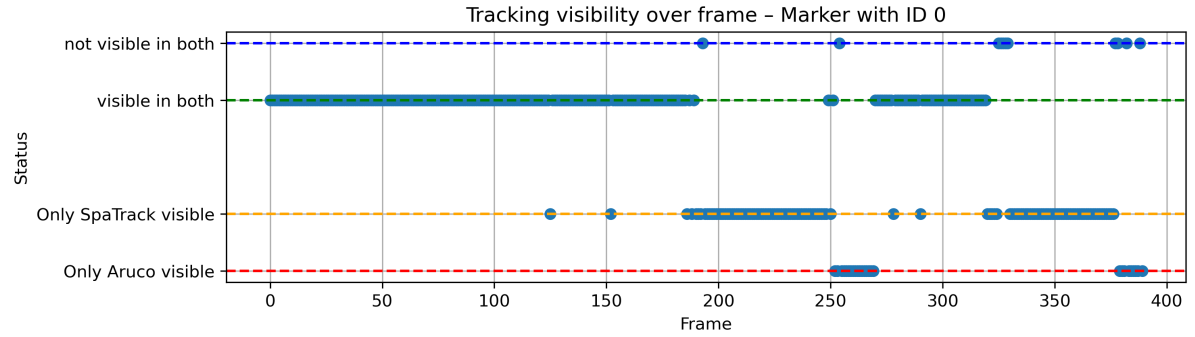


Figure 14: Comparison of visibility with Aruco Id 0

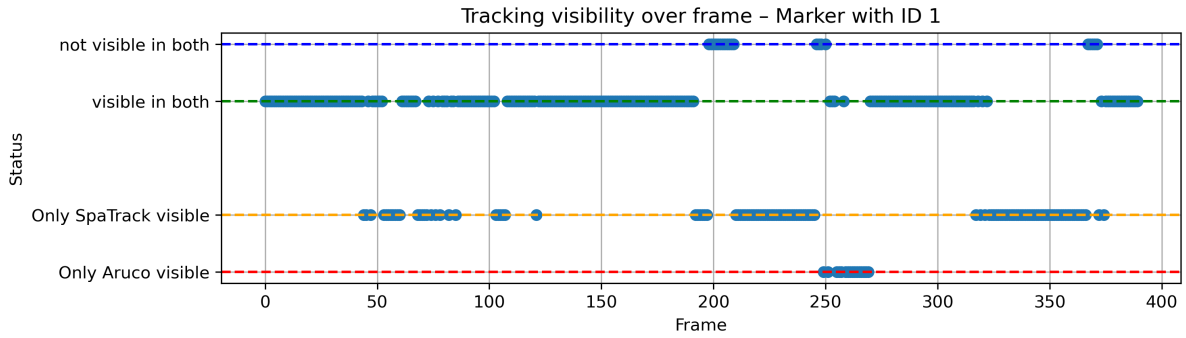


Figure 15: Comparison of visibility with Aruco Id 1

Overall, it can be said that movements and changes in shape are detected very well. There seem to be difficulties with occlusions. It should be noted that we are only using one video sequence as a reference. With this occlusion lasting approx. 100 frames, the Spatial Tracker does not show good results.

Comparison between TOF ground truth and tracker

The comparison between the ground truth provided by the TOF-camera and the results produced by our SpatialTracker shows that the result quality depends largely on the type of object being tracked. While simpler objects such as the “exercise ball” and the “blackboard eraser” were tracked consistently over the duration of the video, tracking the “Theraband” proved more challenging. Adding markings to the band improved the tracking performance to some extent, but not in a consistent manner — the results still varied considerably. This may be due to the nature of the Theraband, which is highly unstable in its dimensions and can be stretched into a line or compressed into a ball. This appears to be a complex issue that merits further investigation.

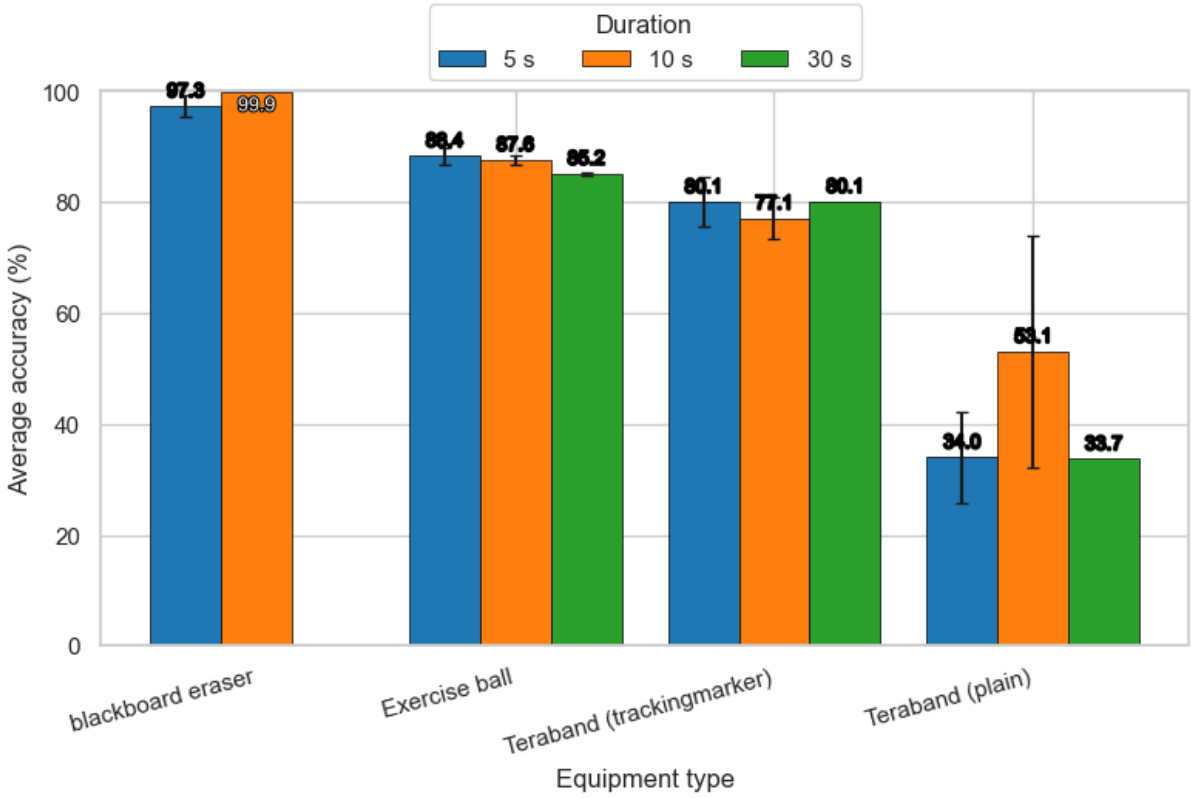


Figure 16: Average accuracy per equipment type and clip duration

In contrast, the tracking performance over time remained fairly consistent. The tracking quality in the third dimension only decreased slightly. This was not necessarily what we expected.

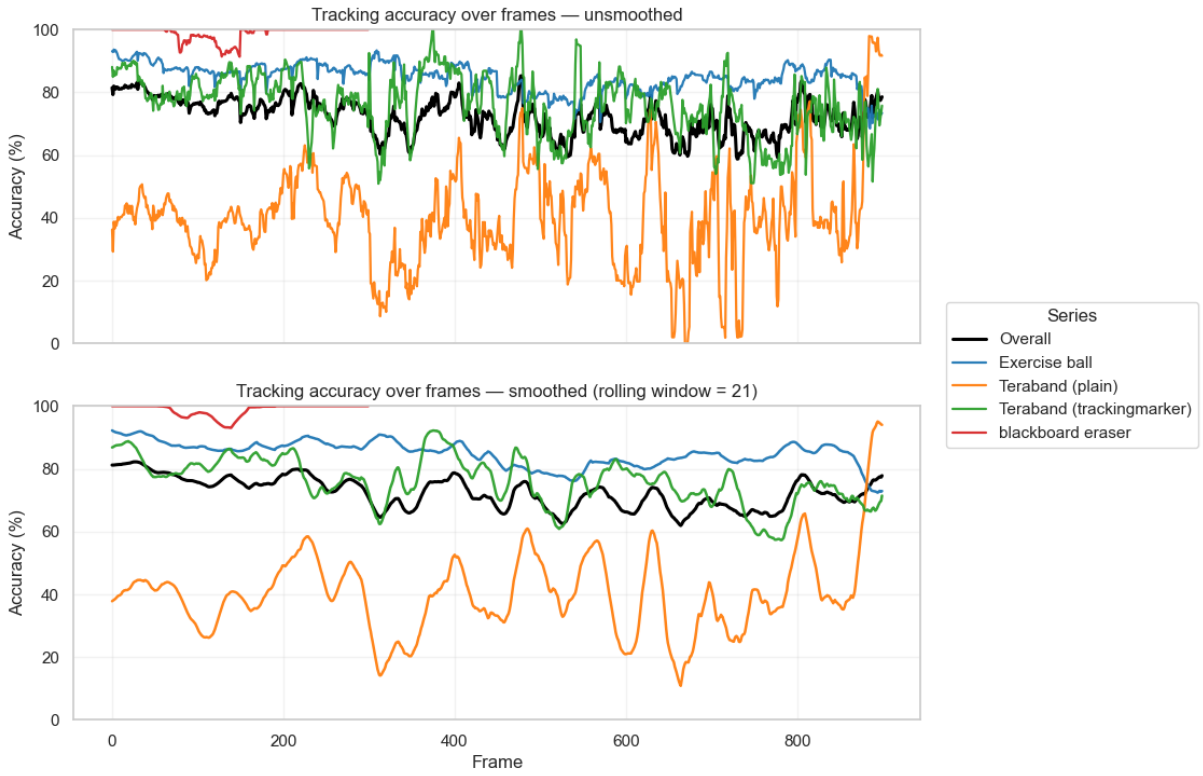


Figure 17: Average accuracy over time and per equipment type

Again, the plot above shows that tracking accuracy depends primarily on the type of workout equipment rather than on the duration.

Another point to note in the process is that it only worked reliably with good initial masks. For automated testing, some automatically generated masks — especially those for the Theraband — were of insufficient quality and had to be replaced manually. This was done to focus on the quality of tracking rather than on the inline masking.

Conclusion

Deformable object tracking remains a complex yet highly impactful area within computer vision, particularly for applications in healthcare, sports, and human-computer interaction. This project explored the challenges of tracking non-rigid objects, focusing on sports equipment such as resistance bands and gymnastic balls. By reviewing several approaches, the SpatialTracker method was identified as the most suitable for the task due to its ability to capture complex deformations over time.

In the next step, we made several adaptations to the SpatialTracker. To handle longer videos with-

out running into memory limitations, we implemented a sliding window approach. Furthermore, we added another monocular depth estimator called Video Depth Anything and a segmentation pipeline to automatically extract the region of interest. Additionally, we created a video library for testing purposes and used a time-of-flight camera to generate ground-truth depth data.

Overall, our evaluation shows that SpatialTracker can reliably track deformable objects under a range of conditions, but its performance varies strongly by object type. While simple objects such as gym balls and blackboard erasers were tracked consistently, resistance bands posed a significant challenge due to their highly variable shape, often reducing accuracy despite added markings. Runtime experiments revealed that GPU resources are the main bottleneck, with higher resolutions and longer clips increasing processing time and memory requirements, and RGB-D processing adding a predictable overhead of 10–20%. 2D efficiency tests with ArUco markers confirmed good accuracy under movement and stretching, but tracking degraded heavily under occlusion. Comparison with ToF ground truth further emphasized that accuracy depends more on object characteristics than on video length, and that reliable initialization masks are crucial for stable results.

Overall, the project lays a solid foundation for further research and development of automated tools that can support therapists and patients in exercise monitoring.

Future work

Adaptive re-sampling (e.g., after long occlusions) and learned cross-chunk consistency could further reduce boundary artefacts and drift. Runtime still scales with grid density; uncertainty-aware sparsification would improve throughput on dense masks.

As potential future work, a more detailed exploration of other tracking models could be considered. In particular, the TAPIR model was not evaluated in this project and could be an interesting topic for further research. **TODO: The CoTracker, however, is unlikely to be pursued, as it forms the foundation for the SpatialTracker and generally results in lower performance**

Another avenue for future work could involve relaxing the ARAP constraints within the SpatialTracker. However, this would require significant modifications to the algorithm. Currently, the ARAP restriction groups individual pixels together, which may be disadvantageous in cases where pixels need to move independently. This could be an advantage in case of strong deformations such as resistance bands.

TODO: another depth estimation model?