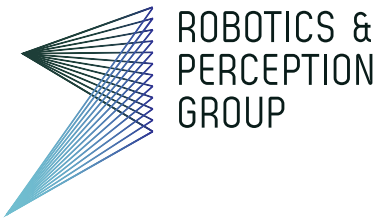
University of Zurich[UZH]

Department of Informatics

University of Zurich[UZH]

ETH zürich

Institute of Neuroinformatics



Samuel Bryner

Event-based, Direct Camera Tracking from a Photometric Map using Nonlinear Optimization

Master Thesis

Robotics and Perception Group
University of Zurich

Supervision

Henri Rebecq
Dr. Guillermo Gallego
Mathias Gehrig
Prof. Dr. Davide Scaramuzza
August 2018

Contents

Abstract

Knowing your position in an environment is crucial for a large number of applications, such as many tasks in robotics, augmented reality, surveying and others. Similarly many techniques exist to find your position, from dead-reckoning to GPS.

We use an event camera, a relatively recent sensor where each pixel asynchronously reports changes in intensity. Because these cameras report brightness changes which are caused by moving edges (if there are no changes in illumination), event cameras are suitable for motion-estimation application, and, in particular, camera tracking or ego-motion estimation.

In contrast to standard cameras, event cameras offer a very high dynamic range, very low latency, lower data rates and lower power consumption, making tracking in challenging scenarios possible.

However, due to the inherent differential nature of an event camera's output classical approaches from computer vision such as feature tracking are harder

to apply. We opted to restrict the problem to a scenario where the environment is already known and relatively static.

This is conceivably the case in augmented and especially virtual reality, where the user is usually confined to a room which can be mapped beforehand. AR and VR also require fast position updates for an immersive experience, something an event camera excels in.

We implemented a new method of tracking the full six DoF pose and velocity of an event camera by comparing the observed events with brightness increments predicted from a photometric map using a non-linear optimization framework.

Because VR and AR are consumer applications, we also investigated affordable options of generating a photometric map.

We evaluated our method on both simulated data and as well as real data. The latter consists of a dense RGB point cloud we built up using an RGB-D sensor (i.e. images and depth maps), with ground-truth poses provided by a motion capture system.

While the approach works, it is computationally expensive and we were unable to achieve real-time operation.

Nomenclature

Notation

$\boldsymbol{J}$	Jacobian
$\boldsymbol{H}$	Hessian
$\boldsymbol{T}_{WB}$	coordinate transformation from frame B to frame W
$\boldsymbol{R}_{WB}$	orientation of B with respect to W
${}_W\boldsymbol{t}_{WB}$	translation of B with respect to W, expressed in coordinate system W
${}_W\boldsymbol{p} = [p_x, p_y, p_z]^\top$	point in 3D space, expressed relative to coordinate system W
$\mathcal{U} \subset \mathbb{R}^2$	set of all (normalized) pixel coordinates, i.e. the image plane
$\boldsymbol{u} \in \mathcal{U}$	normalized image coordinates
$I(\boldsymbol{u}) : \mathcal{U} \to \mathbb{R}$	intensity image
$\Delta I(\boldsymbol{u}) : \mathcal{U} \to \mathbb{R}$	intensity change image
$\Delta \hat{I}(\boldsymbol{u}) : \mathcal{U} \to \mathbb{R}$	estimated intensity change image
$Z(\boldsymbol{u}) : \mathcal{U} \to \mathbb{R}$	depth (usually abbreviated as z)
$G(\boldsymbol{u}) : \mathcal{U} \to \mathbb{R}^2$	image gradient $G(\boldsymbol{u}) := \nabla I(\boldsymbol{u})$
$M(\boldsymbol{T}, \boldsymbol{u}) : SE(3) \times \mathbb{R}^2 \to \mathbb{R}$	photometric map
Δt	integration window
$p_k \in \{+1, -1\}$	polarity of event k
$\boldsymbol{u}_k \in \mathbb{N}^2$	position of event k
$C \in \mathbb{R}^+$	intensity threshold (contrast sensitivity)
$\pi(\boldsymbol{p}) : \mathbb{R}^3 \to \mathbb{R}^2$	projection function, projects 3D point onto camera plane
$\pi^{-1}(\boldsymbol{u}, z) : \mathbb{R}^2 \to \mathbb{R} \to \mathbb{R}^3$	back-projection function, projects 2D point from camera plane into 3D space
$\tau_{AB}(\boldsymbol{u}) : \mathbb{R}^2 \to \mathbb{R}^2$	combined warping function, $\tau_{AB}({}_B\boldsymbol{u}) := \pi(\boldsymbol{R}_{AB} \cdot \pi^{-1}({}_B\boldsymbol{u}) + \boldsymbol{t}_{AB})$

Scalars are written in lower case letters (a), vectors in lower case bold letters ($\mathbf{a}$) and matrices in upper case bold letters ($\mathbf{A}$).

Coordinate Systems

W	World frame = map frame (absolute, global position)
K	Keyframe

C	Current Camera
H	Hand (position of camera trackers)
R	Reference for relative poses in optimization

We use the notation introduced by Prof. Glocker in the course "Mechanik 3" at ETHZ to express coordinate frames, rotations and vectors. Refer to Chapter 5 "Kinematik" in the lecture script for more details [1]. Figure A.4 gives an overview of how coordinate transformations and vectors are specified. Observe that the coordinate system in which a vector is expressed is always written as index before the variable, e.g. ${}_B\mathbf{t}_{AB}$ is the vector from A to B expressed the coordinate system B. For the ease of reading, the index for the origin coordinate frame can be omitted: ${}_O\mathbf{t}_k := \mathbf{t}_k$.

Optimization Variables

$\boldsymbol{T} \in SE(3)$	camera pose
$\dot{\boldsymbol{T}} \in \mathbb{R}^{6\times6} \simeq \mathbb{R}^6$	camera velocity $\dot{\mathbf{T}} = \frac{\mathbf{d}}{\mathbf{dt}}\mathbf{T}$

Camera Intrinsics

$\boldsymbol{f} = [f_x, f_y]^T$	focal length
$\boldsymbol{c} = [c_x, c_y]^T$	principal point
W, H	camera image size

Acronyms and Abbreviations

RPG	Robotics and Perception Group
DoF	Degree of Freedom
IMU	Inertial Measurement Unit
MAV	Micro Aerial Vehicle
ROS	Robot Operating System
DSLR	digital single-lens reflex camera (synonymous with a high-end digital camera)
ETH	Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology)
ASL	Autonomous Systems Lab (ETH)
TUM	Technical University Munich
SLAM	Simultaneous Localization and Mapping
VO/VIO	Visual [Inertial] Odometry
EKF	Extended Kalman Filter, nonlinear version of the classical Kalman Filter

[1] `http://mitschriften.amiv.ethz.ch/main.php?page=3&scrid=1&pid=87&eid=1`

Chapter 1

Introduction

In virtual and especially in augmented reality, we need to know where the user is with respect to his or her surroundings. This is a much more general problem that appears in many other places as well and there is a myriad of ways to track the position of an object, a robot or a person. One approach is to use a camera mounted on the robot and program it to look at the surrounding scenery to determine it's position in the world, similar to humans and other animals.

1.0.1 Using an event camera

We are using an event camera as it provides a very high temporal resolution (in the order of microseconds) and is thus well suited for the high-speed low-latency tracking required for augmented and virtual reality applications.

To give a short introduction into event cameras: A classical digital video camera records full images with a (usually) fixed frame rate. In other words, we get a 2D array of brightness values so and so many times per second. In event camera on the other hand, every pixel is completely independent of the others and immediately reports changes in brightness. Thus, the output of an event camera is a stream of so called events, where every event consists of a the coordinate of the pixel that changed, the precise timestamp of when this change happened and a polarity, indicating if the brightness increased or decreased.

This results in a camera that can record changes with very high temporal resolution and is thus basically free of motion blur. The independent nature of the pixels also lead to an impressive dynamic range of up to 140 dB (versus 60 to 70 of standard cameras), as there is no global exposure. Another advantage is the inherently sparse nature of the data, as only changes are reported. This results in lower data rates than a classical camera, even tough the events are much faster with an update rate of up to 1 MHz.

1.0.2 Alignment of Event Frames

Using a standard camera, there are two broad methods to solve the problem of figuring out the position of a picture with respect to another one:

The first is to analyze the image and somehow summarize it. Further calculations then use these higher level data. Usually salient points are determined (called features), for example corners that are easily recognizable. One then looks for these features in other views and tries to find correspondences (i.e. features that appear in both/multiple images).

Alternatively, it is also possible to operate directly on the full image. Here, one usually compares the pixel-wise difference in intensity between two images and tries to find a viewpoint which is most similar to a given image.

The asynchronous nature of an event camera's data makes it hard to apply the first method. For example, in order to track a feature over time we would have to somehow figure out which events belong to a given feature. This is the well-known problem of data association [7].

Instead, we adopt the second approach: we integrate events over a short time window to produce an image (of brightness differences rather than absolute brightness), and then directly align the image with respect to a projection of a map of the scene, as in global image alignment methods [5]. Data association is inherited from the image representation.

1.0.3 Pre-recorded Photometric Map

As the environment in AR and VR is usually static and known we concentrate our efforts on the camera tracking problem and assume we have a map of the environment. As we are working with cameras this is a photometric map, e.g. a colored point cloud or just a single image frame with depth information.

This three dimensional map gives us the ability to create arbitrary virtual frames as we can render the scene from any perspective. For any pose we can calculate how we expect the world to look like if we took a picture there.

The problem can therefore be formulated as finding the view (and thus the pose) that is most similar to the current one by comparing the views directly on a per-pixel basis.

As events are only generated through changes in intensity and thus only through movement in an otherwise static environment, we not only estimate the pose but also the velocity of the camera. Based on the camera's velocity we can calculate the motion field (commonly referred to as optic flow) of each pixel, which can be used to compute the expected intensity change during the integration window. This expected change is compared to the one measured by the event camera, and the error is used to refine the camera's pose and velocity.

1.1 Related Work

There are other approaches to implement tracking, odometry or even full SLAM with an event camera. Apart from [7], these all either track features, rely on a filter or use a strongly discretized representation of the event stream.

Most closely related in terms of methodology is [7] where the same idea of predicting brightness changes and comparing them to integrated events is applied to feature tracking instead of full pose tracking. Small patches of a keyframe are tracked using 2D affine warpings.

Using the same problem statement, namely tracking with events on a photometric map, [6] implements a filtering approach, where every event updates the state of an EKF, instead of non-linear optimization over a batch of events

EVO [17] implements full, event-based SLAM based on the mapping from [15, 14]. In contrast to this work where we use photometric alignment, EVO relies on geometric alignment by extracting discrete edges from the map and events, thereby discarding not only polarity but also magnitude of the gradients.

The visual-inertial odometry described in [16] applies classical feature tracking to event frames generated by summing up the events while disregarding their polarity and compensating for motion blur using inertial measurements.

Another event based SLAM is shown in [9], where three probabilistic filters are used to estimate the camera motion, the intensity gradient of the scene and the scene depth relative to a keyframe.

1.2 Contributions

This master thesis presents two main contributions: (i) pose tracking with an event camera, and (ii) a comparison of methods to generate a photometric map useful for event-camera tracking, along with a new dataset.

1.2.1 Pose Tracking with Events

We developed a novel method to track full 6 DoF poses based on comparing events to brightness increments predicted from a photometric map. Our approach not only estimates absolute poses but also the current velocity (i.e. linear and angular velocity), essentially resulting in 12 DoF tracking.

This method is described in detail in Chapter 2 and tested on both simulated data sets (Section 4.2, using the event camera simulator from [12]) and real recordings (Section 4.3).

1.2.2 Affordable Methods to Generate a Photometric Map

In order to make our method available for a wide range of applications we investigated multiple approaches to generate a photometric map using commonplace

consumer technology: Digital photography (Section 3.2), video cameras (Section 3.1) and a structured light depth-sensor (Section 3.3) instead of relying on expensive sensors (like a laser-scanner or a motion-capture system).

In the course of this we created a new data set consisting of a photometric map (a dense RGB point-cloud) and multiple event-camera trajectories with ground-truth poses. This map was used for evaluating our method in Chapter 4.

Chapter 2

Methodology

2.1 Overview

In this chapter we present a principled approach for pose tracking with events on a photometric map. By combining an image's gradient with the optic flow we can predict the rate of change in intensity, a quantity which is directly related to the events rate according to the event generation model described in [6] (see Eq. 2.11).

In the following sections we go over all the parts that make up our pipeline (shown in Figure 2.1). We start at the map which gives us intensity frames with associated depth information for arbitrary poses. We then take the numeric derivative of the image to get the gradient and calculate the optic flow from the camera's velocity. The gradient and the flow are then multiplied to give an expected intensity change per pixel. These expected changes are then compared to those measured by the events.

The key idea is that the dot product of the optic flow with the intensity gradient results in an expected rate of change of intensity for a given camera pose and velocity.

2.2 The Event Image

Before we go into the details of predicting intensity changes from our model, let's look at the data we want to match: The event image.

By integrating the events over a time interval Δt we get an image that is not made of absolute intensities, but instead gives a brightness difference $\Delta I(u) : \mathbb{R}^{NxM} \to \mathbb{R}$, which can be seen as the difference of two normal images $\Delta I = I_1 - I_2$. We can also interpret it as the rate of intensity change: The more events

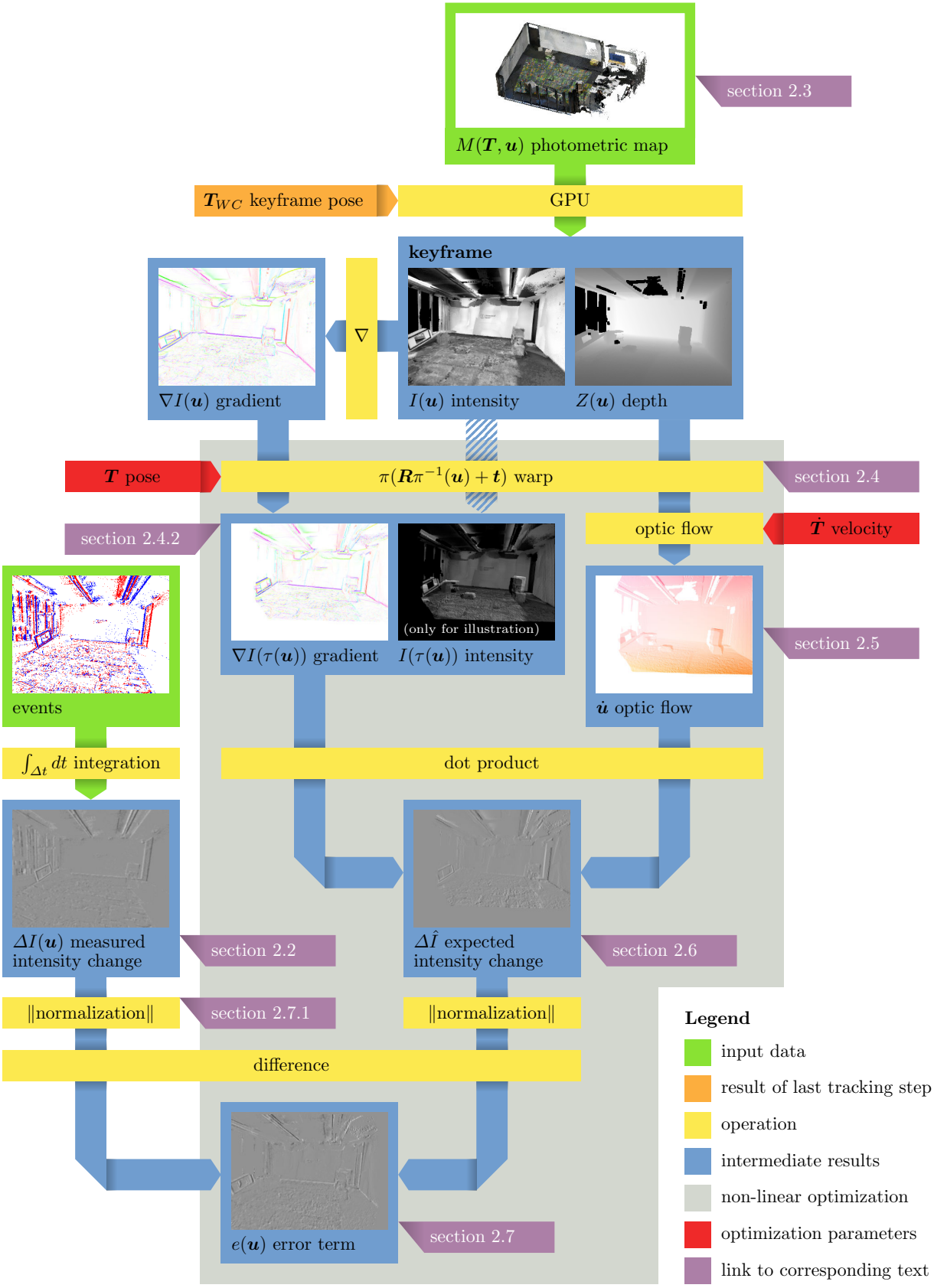


Figure 2.1: Full pipeline

at a pixel, the faster its brightness is changing.

$$\Delta I(\boldsymbol{u}) = \sum_{t_k \in \Delta t} p_k C \delta(\boldsymbol{u}, \boldsymbol{u_k}) \tag{2.1}$$

There are approaches which try to recover the absolute intensities from such a difference image, such as [8]. However, this requires expensive computations as the whole image must be taken into account. Instead, we directly work with those differences, as calculating them from absolute intensities is much easier than the other way around.

2.3 The Map

Our model is a set of 3d points with associated intensities (or color). For the simulation we used a textured mesh instead. However the actual structure of the map is not important and we can think of the model just as a function that generates an image with depth (a keyframe) given a pose:

$$\begin{aligned} M &: SE(3) \times \mathcal{U} \to \mathbb{R} \\ M(\boldsymbol{T}, \boldsymbol{u}) &:= I_{\boldsymbol{T}}(\boldsymbol{u}), Z_{\boldsymbol{T}}(\boldsymbol{u}) = I(\boldsymbol{u}), Z(\boldsymbol{u}) \end{aligned} \tag{2.2}$$

The creation of the map itself is described in Chapter 3.

From now on we will work on the image generated from the map, $I(\boldsymbol{u})$. Even tough we could directly operate on the map $M(\boldsymbol{T}, \boldsymbol{u})$ (as discussed in Section B.1), this is quite costly as the map potentially consists of many millions of points. Rendering it to a keyframe is not only a neat way to decimate the map, it also makes the math a bit more general as we now become independent of the actual source of the keyframe.

2.4 Warping

In order to use the keyframe as a map we need some way of transferring (i.e. re-rendering) it to a different viewpoint. This is achieved by back-projecting a pixel from the image plane into 3D space by using it's depth and the camera intrinsics.

Once in 3D we can apply arbitrary rotations and translations and project it back onto the new image plane, thus warping a keyframe into another perspective.

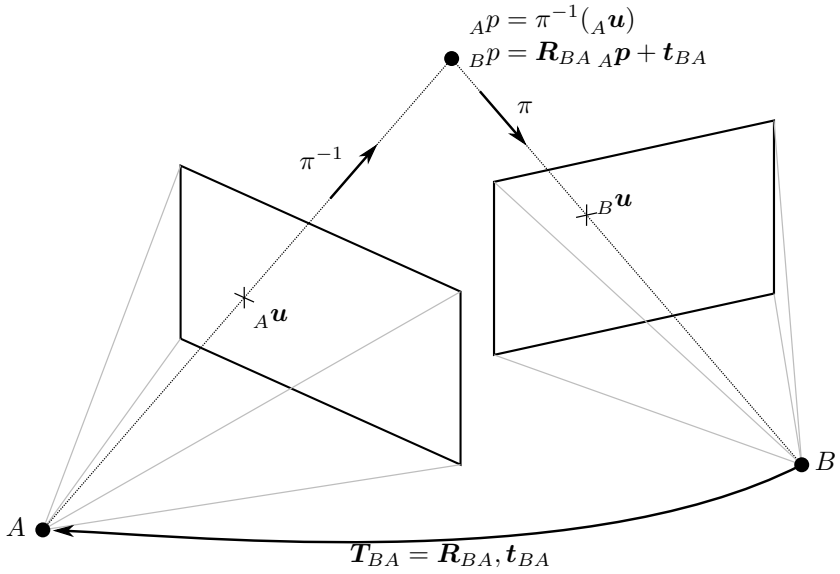


Figure 2.2: The warping operation described in Section 2.4.1, showing how a pixel in one camera frame can be transferred to that of another one.

2.4.1 Projections

The two functions required for the transferring/warping are the the projection function that maps from world coordinates to the image plane:

$$\boldsymbol{u} = \pi(\boldsymbol{p}) := \frac{\boldsymbol{f}}{p_z}\begin{bmatrix} p_x \\ p_y \end{bmatrix} + \boldsymbol{c} \tag{2.3}$$

and the back-projection π^{-1}, that does the reverse:

$$\boldsymbol{p} = \pi^{-1}(\boldsymbol{u}, z) := \frac{z}{f}\begin{bmatrix} u_x - c_x \\ u_y - c_y \\ f \end{bmatrix} \tag{2.4}$$

These can then be combined to get the warping operator, which transforms a pixel in image plane of camera A to the plane of camera B:

$${}_B\boldsymbol{u} = \tau({}_A\boldsymbol{u}, \boldsymbol{T_{BA}}) := \pi(\boldsymbol{T_{BA}} \cdot \pi^{-1}({}_A\boldsymbol{u}, z(\boldsymbol{u}))) \tag{2.5}$$

Normalized Image Coordinates

There is actually no need to keep the focal length and principal point around. It makes the formulas simpler by using normalized image coordinates by subtracting the principal point so that the center of the image is at the origin (point (0,0); instead of the top left corner) and dividing by the focal length:

$$\bar{\boldsymbol{u}} = \begin{bmatrix} \bar{u}_x \\ \bar{u}_y \end{bmatrix} := \begin{bmatrix} (u_x - c_x)/f_x \\ (u_y - c_y)/f_y \end{bmatrix} \tag{2.6}$$

From now on all image coordinates are normalized image coordinates.

The projection and back-projection functions simplify to:

$$\boldsymbol{u} = \pi(\boldsymbol{p}) := \frac{1}{p_z} \begin{bmatrix} p_x \\ p_y \end{bmatrix} \tag{2.7}$$

$$\boldsymbol{p} = \pi^{-1}(\boldsymbol{u}, z) := z \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} \tag{2.8}$$

2.4.2 Warping Gradients

Warping an intensity image based on above definition is straightforward: Take a pixel, project it into 3D space, translate and rotate it to the other frame and project it back onto the image plane.

However, we are not dealing with scalar intensities but their derivatives with respect to the pixel position, i.e. a two-dimensional gradient vector. To transfer this value it is not enough to calculate a pixel's position in another frame as the magnitude and direction of the gradient will be influenced by the transfer as well:

$$\begin{aligned} I(\boldsymbol{u}) &= I(\tau(\boldsymbol{u})) \\ \nabla I(\boldsymbol{u}) &= \nabla I(\tau(\boldsymbol{u})) \cdot \frac{\partial \tau}{\partial \boldsymbol{u}} \end{aligned} \tag{2.9}$$

This correction factor is explored in Section A.3.

2.5 Optic Flow (Motion Field)

Given linear and angular velocity of the camera, v and ω, we can calculate the optic flow at every pixel using the the feature sensitivity matrix (also known as

the 'image Jacobian' or an 'interaction matrix'):

$$\dot{\boldsymbol{u}} = \begin{bmatrix} -1/Z & 0 & u_x/Z & u_x u_y & -(1+u_x^2) & u_y \\ 0 & -1/Z & u_y/Z & 1+u_y^2 & -u_x u_y & -u_x \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \boldsymbol{J}_p \dot{\boldsymbol{T}} \quad (2.10)$$

For the derivation of this matrix see for example [3], eq. 15.3 (or 15.6 for non-normalized image coordinates).

2.6 Expected Intensity Change

We can now the combine the pieces from the previous sections to predict the expected change in intensity over a certain time interval Δt by approximating it using the optic flow $\boldsymbol{u}$ and the image gradient ∇I according to [7]:

$$\Delta I(\boldsymbol{u}) \approx \Delta \hat{I}(\boldsymbol{u}) = -\nabla I(\boldsymbol{u}) \cdot \dot{\boldsymbol{u}} \Delta t \quad (2.11)$$

We can now compare these predicted changes pixel-wise with the actual ones (Equation 2.1), indicating how well a keyframe matches a set of events:

$$e(\boldsymbol{u}) = \Delta I(\boldsymbol{u}) - \Delta \hat{I}(\boldsymbol{u}) \quad (2.12)$$

2.7 The Error Function

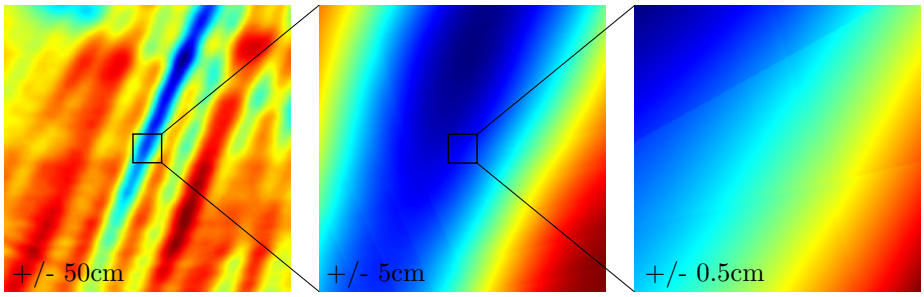


Figure 2.3: The error function plotted for translation in x and y, shown for $\pm 50cm$, $\pm 5cm$ and $\pm 5mm$

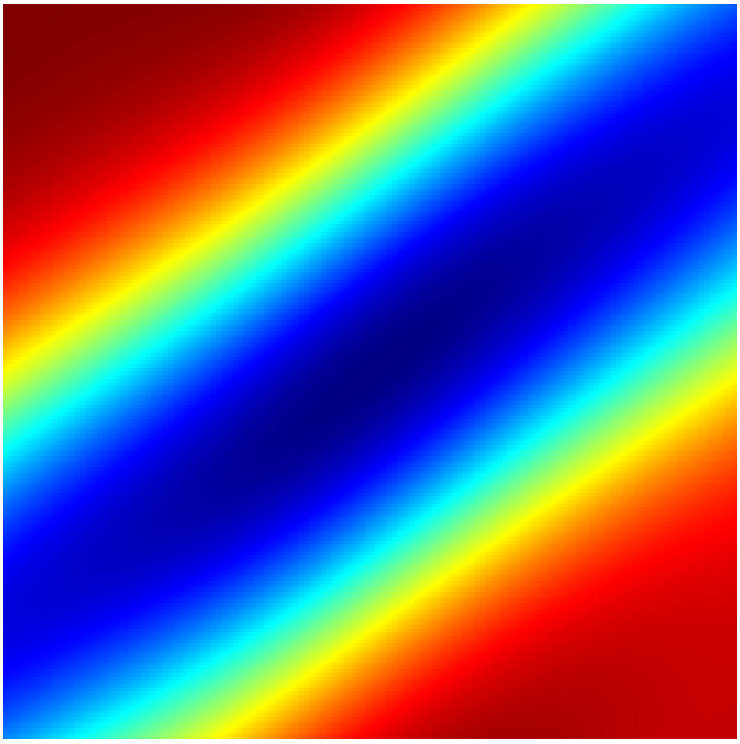


Figure 2.4: Horizontal translation ($\pm 10cm$ along the X axis) versus rotation around the vertical axis (± 0.6 around Y), clearly showing the similarity in appearance of those two motions.

The tracking task can now be stated as finding the pose and velocity that minimize the difference between predicted and measured intensity changes:

$$\underset{\boldsymbol{T},\dot{\boldsymbol{T}}}{\operatorname{argmin}} = \sum_{\mathcal{U}} \left(\Delta I(\boldsymbol{u}) - \Delta \hat{I}(\boldsymbol{u}) \right)^2 \tag{2.13}$$

This equation can be solved numerically using classical least-squares solvers, in our case Ceres [1].

2.7.1 Normalizing away Constants

You might have noticed that our error function above contains the event threshold C and the length of the time interval Δt. While the latter is chosen by us and thus known, the event threshold is not. We could try to add C as yet another parameter to optimize for, but there is a much simpler trick: We normalize both images and thus get rid of any constant terms using

$$\|I\|_{\mathcal{U}} := \sqrt{\sum_{\boldsymbol{u} \in \mathcal{U}} \|I(\boldsymbol{u})\|_2^2} \tag{2.14}$$

The final error function then becomes:

$$\underset{\boldsymbol{T},\dot{\boldsymbol{T}}}{\operatorname{argmin}} = \sum_{\mathcal{U}} \left(\frac{\Delta I(\boldsymbol{u})}{\|\Delta I(\boldsymbol{u})\|_{\mathcal{U}}} - \frac{\Delta \hat{I}(\boldsymbol{u})}{\|\Delta \hat{I}(\boldsymbol{u})\|_{\mathcal{U}}} \right)^2 \tag{2.15}$$

However, this not only removes C and δt, it divides out *any* constant part. This means that the magnitude of the velocity will also cancel, rendering $\|\dot{\boldsymbol{T}}\|$ unobservable.

Chapter 3

Generating the Map

In our scenario we assume a known and static environment, for example a VR user's living room. We therefore investigated methods which could generate a map of such an environment that do not require expensive equipment. Ideally the DAVIS itself could be used in a dense SLAM pipeline to record the map.

However, the Active Pixel Sensor of the DAVIS (the normal image sensor which takes classical intensity images) is of poor quality (very low resolution and a lot of noise). We therefore opted to use an Intel ZR300 camera which includes a high-quality wide-angle lens, a stereo-vision based depth camera and an IMU.

While there exists a large variety of SLAM algorithms, not many researchers publish their source code and even from those that do the code is often "research-quality" and can be cumbersome to set-up and use.

After looking at various options we settled on the following pieces of software to evaluate:

- **ROVIOLI and voxblox**

  Developed by the ASL at ETH, ROVIOLI [2] is a state of the art open-source visual-inertial odometry pipeline with offline loop-closure and bundle-adjustment implemented in maplab [18].

  The poses from ROVIOLI can be fed to voxblox [13], another project from ASL that uses signed distance fields to build a voxel-based map.

- **ElasticFusion**

  Another open-source project, developed at Imperial College London, ElasticFusion implements a powerful RGB-D-SLAM that builds dense maps and can correct for loop-closures in real-time [19] [20].

- **Agisoft PhotoScan**

  PhotoScan is a commercial product sold by Agisoft that uses structure from motion to triangulate camera images and recover dense point clouds and meshes.

  `http://www.agisoft.com/`

3.1 maplab: VIO with BA

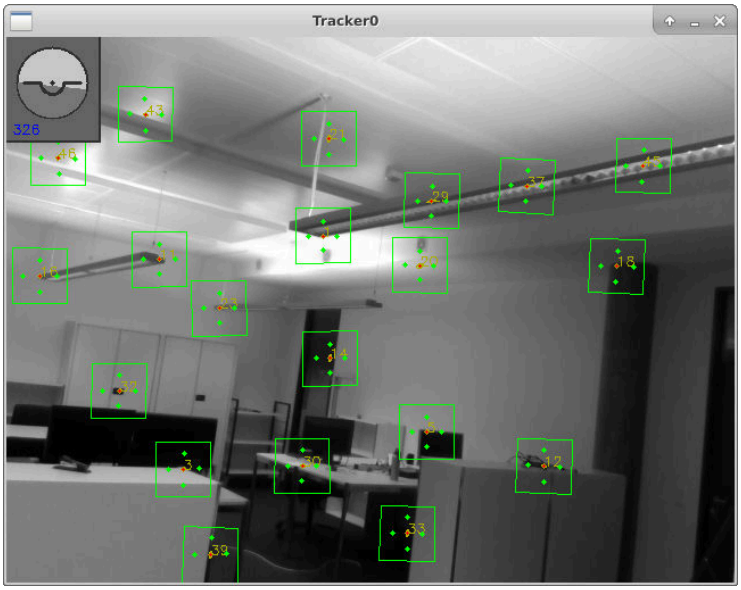


Figure 3.1: ROVIOLI running and tracking features

Maplab is a collection of algorithms and tools developed by ETH's ASL. It includes ROVIOLI, a modern visual-inertial pipeline and voxblox, a volumetric mapping library based mainly on Truncated Signed Distance Fields.

A strong advantage of maplab is its support for multiple recordings, allowing the user to record larger scenes with ease.

The Intel ZR300 is explicitly supported by ROVIOLI. For installation and usage instructions refer to the official documentation.

3.1.1 Workflow

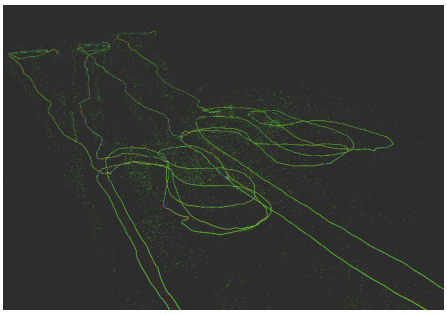
Using maplab to recover a dense map is done in three distinct steps:

1. **Take recordings with ROVIOLI**

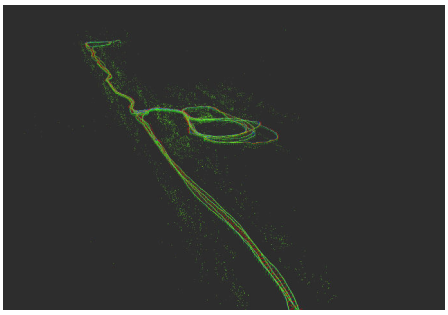
   First, a recording of the environment is taken with ROVIOLI running. This is not very computationally intensive and can easily be done by walking around with a laptop and the ZR300.

   The result is a rosbag which contains poses for every frame. To recover a dense model, the depth data provided by the stereo-vision sensor is recorded as well, even tough ROVIOLI does not use depth data and instead relies on the wide angle sensor and IMU.

   Proper calibration of the intrinsics and extrinsics of all those sensors is paramount.



(a) Raw poses output from ROVIOLI.



(b) Poses after processing in maplab.

Figure 3.2: The effect of loop-closure and bundle-adjustment on the poses.

Figure 3.3: Optimized map manually aligned to floor plan of building where the recording was taken. Even over long distances the poses are very accurate.

2. **Post-processing the poses**

   Being an odometry algorithm, ROVIOLI only provides relative poses. By loading the data from the first step into maplab, loop-detection and -closure, followed by bundle adjustment can be employed to make the poses globally consistent.

3. **Building a mesh**

   The corrected poses can now be fed to voxblox, together with the depth images, to create a colored mesh. This step can take a few hours and requires a lot of RAM.

3.1.2 Results

ROVIOLI is a very robust and accurate visual odometry package and together with maplab a versatile tool to build even large-scale maps.

Voxblox on the other hand did not produce the expected quality. We suspect this is due to the following reasons:

- Voxblox is complex software with many parameters. Better fine-tuning of those would most likely increase the quality.
- The depth data from the Intel ZR300 is very noisy and generally of poor quality. Unfortunately, it was the only depth sensor available with IMU.

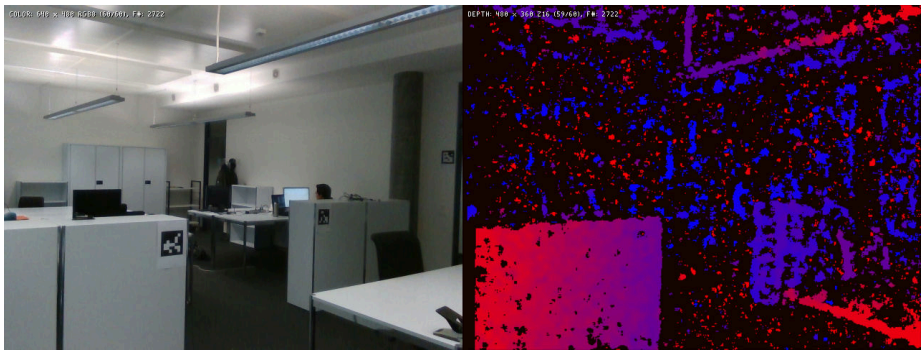


Figure 3.4: Single frame recorded with an Intel ZR300. While the pictures from the color camera (left) and the gray-scale fisheye (not shown) are of good quality, the depth (right) is very noisy and sparse.

Figure 3.5: Output of voxblox using depth data from an Intel ZR300 with poses from ROVIOLI.

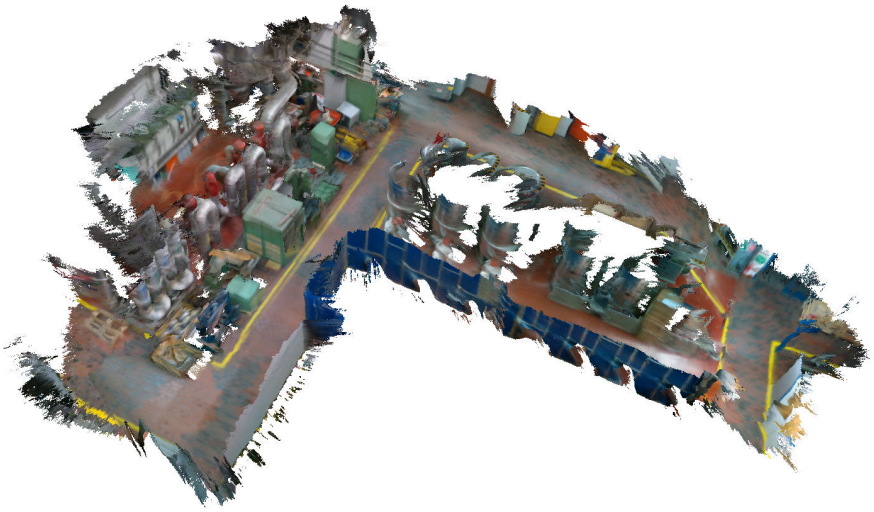


Figure 3.6: Official example from the voxblox website, showing what voxblox is capable of.
Source: `https://github.com/ethz-asl/voxblox/blob/master/README.md#example-outputs`

- At the time of the experiments, the import of depth data from ROVIO-LI/maplab into voxblox was still experimental.
- And lastly, while voxblox can produce excellent geometry (which would be for example very useful for path planning), it's voxel based model isn't well suited for our problem, as we care much more about high-fidelity textures and well defined edges than missing holes and volumes.

3.2 PhotoScan: SfM

Structure from motion is a technique to estimate relative camera poses based on normal images. It has been around for a while and various commercial solutions exist nowadays that make it very easy to use.

PhotoScan from Agisoft is one of the more commonly used options and offers a free trial-mode.

3.2.1 Workflow

The first step is to take pictures of your environment, preferably with a DSLR. It should have at least 5 megapixels and should preferably provide manual focus.

Taking pictures is the most critical step and requires a bit of practice: The algorithm works by estimating depth based on image pairs and one should therefore take care to include sufficient overlap and parallax between images. Instead

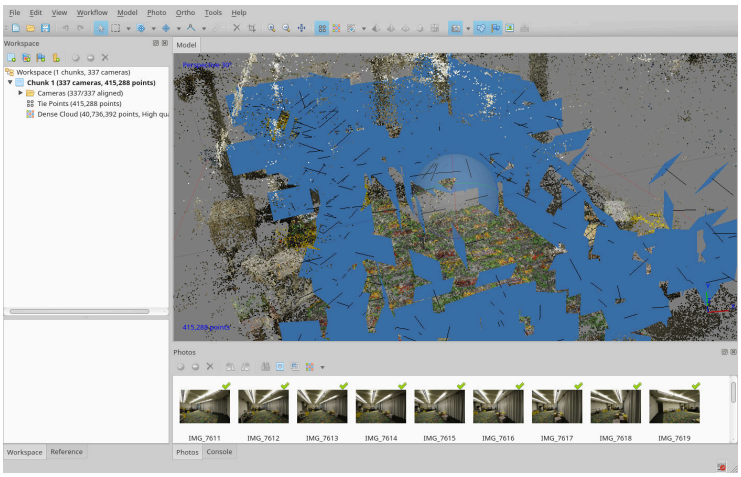


Figure 3.7: Screenshot of Agisoft PhotoScan v1.4.3

Figure 3.8: Camera poses recovered by PhotoScan for a small table scene. 120 pictures where taken trough a 16mm lens, resulting in a dense cloud with 38.87 million points.

of standing in one place and taking pictures of different areas take multiple pictures of the same area from different positions.

The software does not require camera calibration data, as these parameters are estimated together with the poses. It assumes the calibration is the same for every picture and it is therefore important to keep the camera intrinsics constant:

- Use a prime lens where the focal length is fixed or at least keep the zoom constant.
- Do not use auto-focus and instead set focus once and keep it there. Focusing moves lenses and therefore changes the intrinsics!
- The algorithm needs sharp, high-resolution images: Use a large F-stop (i.e. close your aperture) to increase the depth of field and use a tripod to prevent motion blur.
- It also helps to keep the other parameters fixed, such as exposure time and white balance.

As with all vision based algorithms it helps to have a lot of texture in your scene. A uniform white wall will not produce good data.

If required, PhotoScan supports markers in the map to recover the absolute scale of a map.

For more hints and more in-detail explanations we refer the reader to the documentation available from Agisoft at `http://www.agisoft.com/support/tips-tricks/`.

After capturing your environment, the images are loaded into PhotoScan where the software first estimates the relative poses of each image. In a second step a dense point cloud is created which can be converted into a textured mesh in a third step if required.

Processing the images can take a few hours, but can be greatly sped up by the use of GPU acceleration. The software even supports networking to distribute the workload over a cluster.

3.2.2 Results

Structure from Motion provides very good and detailed results. A scan of a single room easily leads to millions of points. As PhotoScan is commercial software it is a lot more polished and easy to use than research software. However, taking good pictures is a bit of an art and requires some practice and with all vision algorithms relying solely on normal images capturing feature-less surfaces isn't possible.

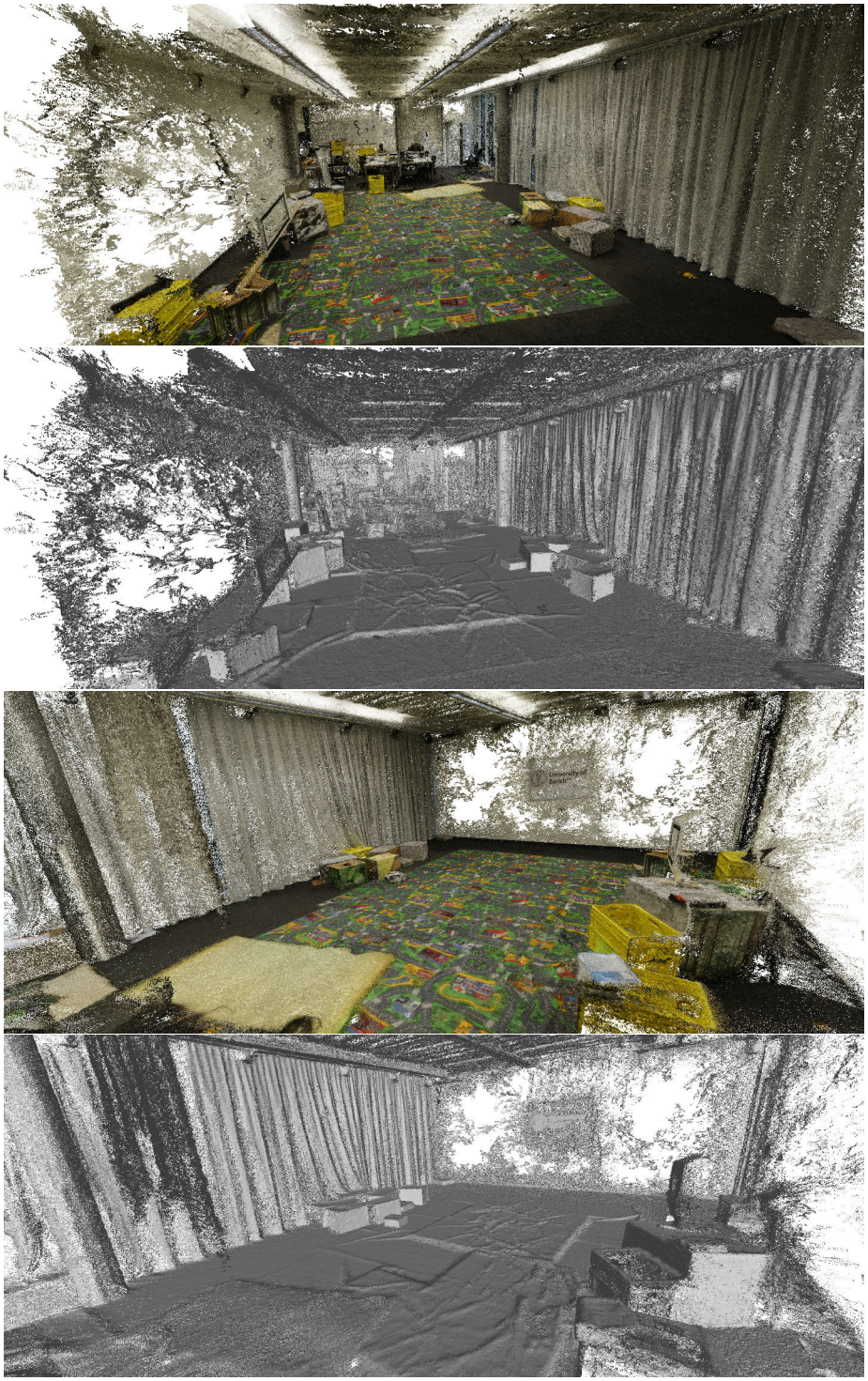


Figure 3.9: Various views of the dense cloud generated with Agisoft PhotoScan with 40.7 million points, generated from 337 pictures taken with a consumer-grade DSLR.

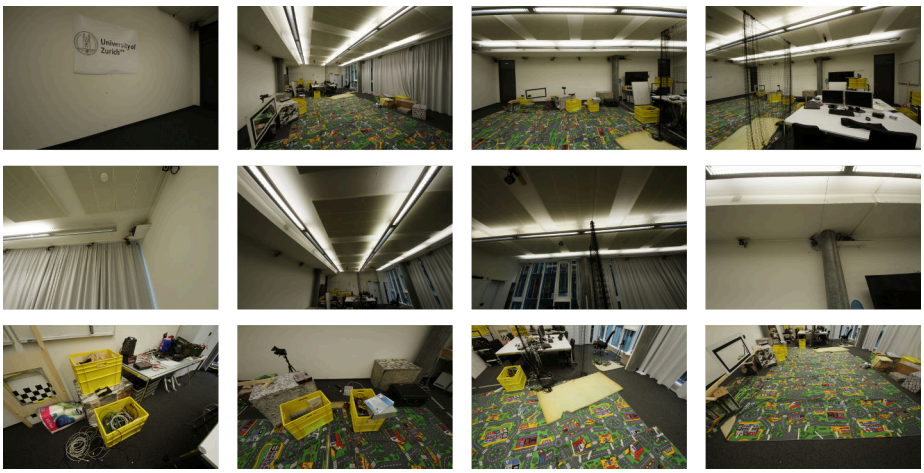


Figure 3.10: Some of the original pictures that were used to generate the model shown in Figure 3.9

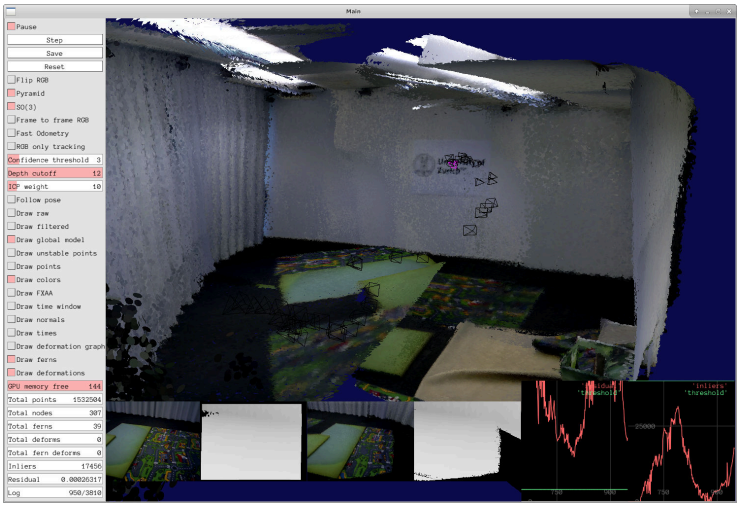


Figure 3.11: ElasticFusion, with the current pose in pink and previous poses in black. Note the bend in curtained wall on the left and how the floor was not aligned correctly.

3.3 ElasticFusion: RGB-D SLAM

ElasticFusion is based on KinectFusion and is a "real-time dense visual SLAM system capable of capturing comprehensive dense globally consistent surfel-based maps of room scale environments explored using an RGB-D camera".

Figure 3.12: A single frame from the ASUS Xtion Pro with color on the left and depth on the right. Notice how much denser and less noisy the depth image is compared to the output from the Intel ZR300 in Figure 3.4

Using ElasticFusion is very straightforward, but requires a computer with a powerful GPU.

Due to the brittleness of the tracking running it in real-time is advisable so that the current state can be observed, but ElasticFusion can also process data offline if no beefy mobile PC or long enough USB cable is available.

3.3.1 Results

Figure 3.13: A single sweep around a well-textured room, showing the quality ElasticFusion is capable of.

We first tried to run ElasticFusion using the Intel ZR300 RGB-D sensor, which we already used with ROVIOLI in the previous section. However, even after extensive parameter tuning both on the sensor side and in ElasticFusion itself without getting any useful results at all and ElasticFusion failing to close loops

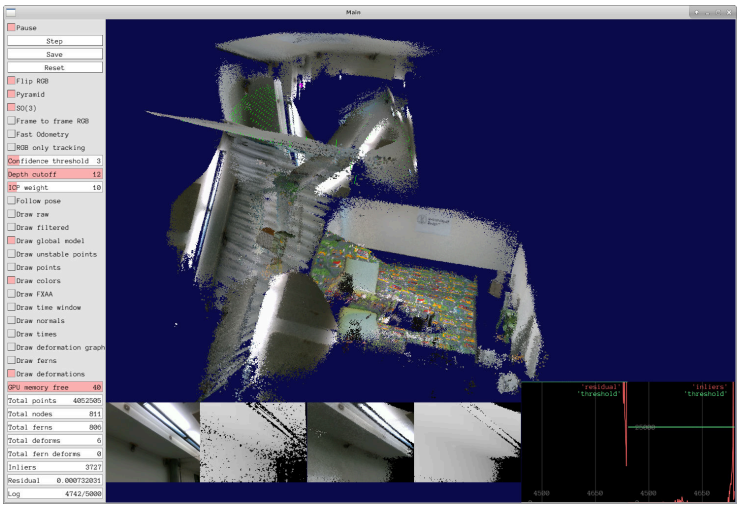


Figure 3.14: ElasticFusion is very sensitive to feature-less areas, especially with a camera that is not very wide-angled. It can diverge quite badly without being able to recover.

etc. we decided to switch to an ASUS Xtion Pro RGB-D sensor. This lead to significantly improved performance, even with the default parameters.

While reconstructing surfaces with exceptional quality, relying entirely on a single camera (with a relatively narrow field of view at that) has it's limitations and tracking is lost when few or no texture is visible, for example white walls. Unfortunately, the real-time operation of ElasticFusion means that any divergence that isn't detected by the loop-closure part will stay in the final output and cannot be corrected anymore. Thus, recording is quite brittle and you have to start over every time a hiccup occurs.

3.4 ElasticFusion with External Poses

With ElasticFusion giving excellent quality but having trouble with divergences we decided to use our motion capture system to generate accurate poses and feed them to ElasticFusion. This gives us the best possible map that might be achieved with consumer-grade hardware and more robust software and let's us evaluate the tracking without additional the errors introduced by the map building process.

As an additional benefit, the resulting map is already aligned to the ground-truth poses from the motion capture system, thus making initialization and evaluation easier.

An even simpler method to generate the map would have been to directly stitch together the raw 3D frames from the RGB-D sensor without using ElasticFusion at all. However, ElasticFusion performs additional filtering and surface regularization, giving much cleaner results which also include surface normals.

3.4.1 Workflow

1. Attach reflective markers to the camera. At least three are required, but 4 to 6 are optimal. Place them so that the markers are visible from every angle, spaced as far apart from each other and as asymmetric as possible. Import the object into the control software of the motion capture system.
2. Perform hand-eye calibration with a checkerboard to calculate the transformation between the pose recorded with the motion capture system and the optical center of the camera. Refer to **??** for details on this procedure.
3. Scan the room while recording intensity images and depth data from the camera and tracked poses from the motion capture system to a rosbag. Make sure exposure time and white balanced are fixed, as they introduce artificial gradients otherwise.
4. Export the color and depth images from the rosbag to the klg format expected by ElasticFusion. Apply the hand-eye transformation to the ground truth poses and export them as CSV.
5. Run ElasticFusion with the following parameters:

   ```
   ElasticFusion -l recording.klg -cal calibration.txt -p gt_poses.csv -d 12 -c 3
   ```

   You can play around with the `-d` (depth cutoff in meters) and `-c` (confidence threshold) parameters to include more or less points.

The calibration file used for the ASUS Xtion Pro, containing (f_x, f_y, c_x, c_y):

```
538.988802 539.938535 318.614110 241.252529
```

3.4.2 Results

The final result is shown in Figure 3.15: A dense, clean and relatively noise-free point cloud.

It is not a big surprise that using a high-end motion capture system and a good RGB-D camera offers the best results. There are no issues with feature-less surfaces, no necessity to move around in a certain way, no divergences etc.

The downside is that this is not a practical solution as it requires an expensive motion capture system and limits the map to the area covered by this system. Of course, if you already own a motion capture system you don't need any of this tracking we developed, as you could directly use the motion capture system.

3.5 Summary

We have shown three quite different approaches to generate a 3D scan of a room. While ROVIOLI with voxblox proved unsuitable for our problem, both

Figure 3.15: The final mesh used, generate with ElasticFusion in combination with a motion capture system. Auto-exposure and white-balance were enabled, leading to some unseemly edges in what are actually uniform white walls.

Agisoft PhotoScan, the commercial SfM software, and ElasticFusion, a free RGB-D-SLAM implementation, are practical methods that require relatively little training.

In the end we settled on the model generated with ElasticFusion in combination with our motion capture system, as this proved to be the most robust and accurate method, and made the alignment of ground truth data with the tracked trajectories straightforward.

Chapter 4

Experiments

4.1 Evaluation Metrics

4.1.1 Position

For positional accuracy we can simply look at the Euclidean distance between the tracked position and the true one as given by the ground-truth data.

$$e_{\boldsymbol{t}} = \|\boldsymbol{t}_{\text{meas}} - \boldsymbol{t}_{\text{gt}}\| \tag{4.1}$$

4.1.2 Orientation

Euclidean distance doesn't apply to rotations $\in SO(3)$. To measure the distance between two rotations in $SO(3)$ we use the geodesic distance, which is given by the angle of the incremental rotation from one rotation to the other:

$$\begin{aligned} \Delta\boldsymbol{q} &= \boldsymbol{q}_A * \boldsymbol{q}_B^{-1} \\ &= \boldsymbol{q}_A * \frac{\bar{\boldsymbol{q}}_B}{\|\boldsymbol{q}_B\|} \end{aligned} \tag{4.2}$$

Where the inverted quaternion can be calculated by taking the complex conjugate. From this difference we can calculate the angle, thus giving us a measure of how far apart the measured orientation is:

$$e_{\boldsymbol{R}} = 2\,\text{atan2}\left(\sqrt{q_i^2 + q_j^2 + q_k^2}, q_r\right)\Big|_{\boldsymbol{q}=\Delta q} \tag{4.3}$$

4.1.3 Velocity

Although velocities could also be compared using the euclidean distance, i.e. the 2-norm of their difference, splitting the difference into an angular part and a magnitude is probably more useful, directly telling us how far "off" our velocities are, both in directionality and in magnitude.

However, remember that we are normalizing the error term in Section 2.7.1, thus loosing a degree of freedom in the estimation! In other words, the tracker only calculates the linear and angular velocities up to scale (Although their *combined* scale).

Of course we can still meaningfully compare the direction:

$$e_{\boldsymbol{v}} = \arccos\left(\frac{\boldsymbol{v}_{\text{meas}} \cdot \boldsymbol{v}_{\text{gt}}}{\|\boldsymbol{v}_{\text{meas}}\|\|\boldsymbol{v}_{\text{gt}}\|}\right) \tag{4.4}$$

$$e_{\boldsymbol{\omega}} = \arccos\left(\frac{\boldsymbol{\omega}_{\text{meas}} \cdot \boldsymbol{\omega}_{\text{gt}}}{\|\boldsymbol{\omega}_{\text{meas}}\|\|\boldsymbol{\omega}_{\text{gt}}\|}\right) \tag{4.5}$$

Another issue with the evaluation of the velocity is the simple fact that our motion capture system only captures poses. We therefore only evaluate the accuracy of the velocities on simulated data sets.

4.2 Synthetic Data

4.2.1 The Event Simulator

In order try out our method under perfect conditions we have used the event camera simulator from [12]. It is capable of simulating arbitrary motions using splines either generated randomly or from a given trajectory.

4.2.2 Planar Scene

The simplest possible scene is just a textured plane. The trajectory is a smooth, randomly generated spline.

In a second experiment we let the tracking running only up to the second lowest level of the image pyramid, never optimizing on the full resolution images. This greatly speeds up computation with only a slight loss of accuracy.

4.2.3 Architectural Scene

The most complex simulated environment we have run is a manually set trajectory trough the model of an inner courtyard.

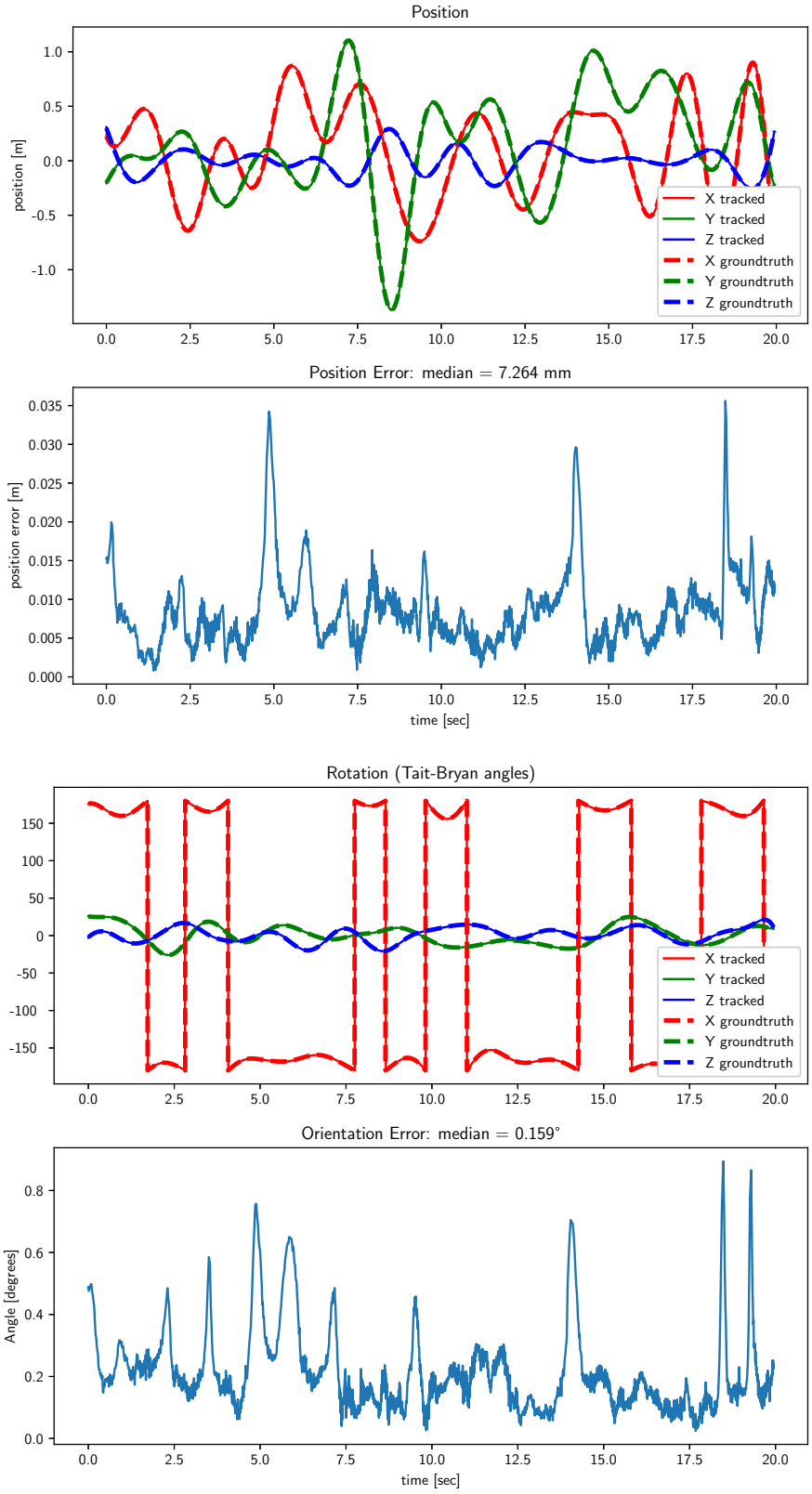


Figure 4.1: Pose tracking results for a smooth random motion over a planar texture.

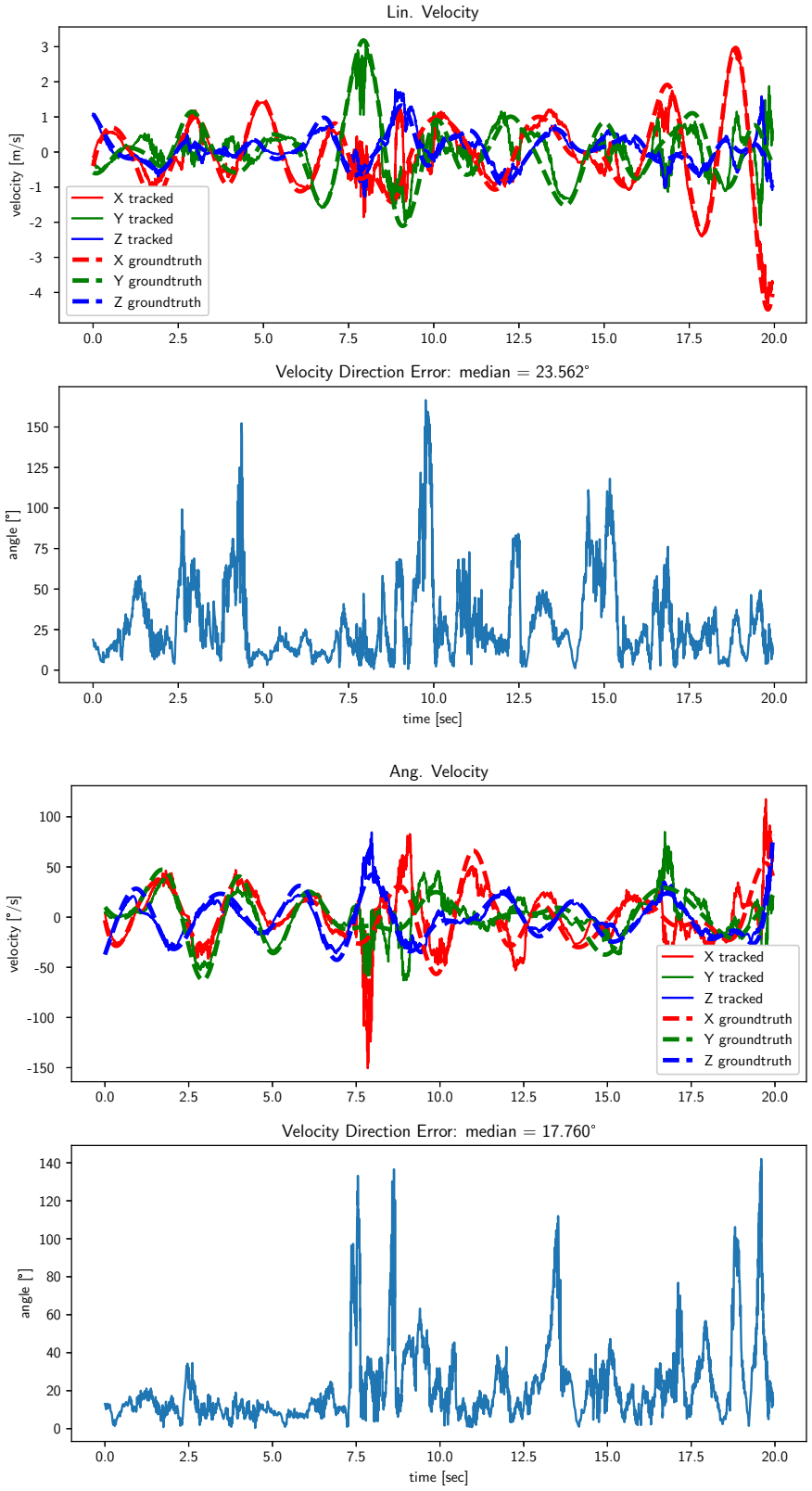


Figure 4.2: Velocity tracking results for from the planar scene. Note that the measured velocities are scaled to match the ground-truth values!

Figure 4.3: Marko Dabrovic's famous model of the Atrium Sponza Palace, Dubrovnik.
In the lower image the rendered trajectory (i.e. the ground truth) is shown as an orange line and some the recovered poses in green.

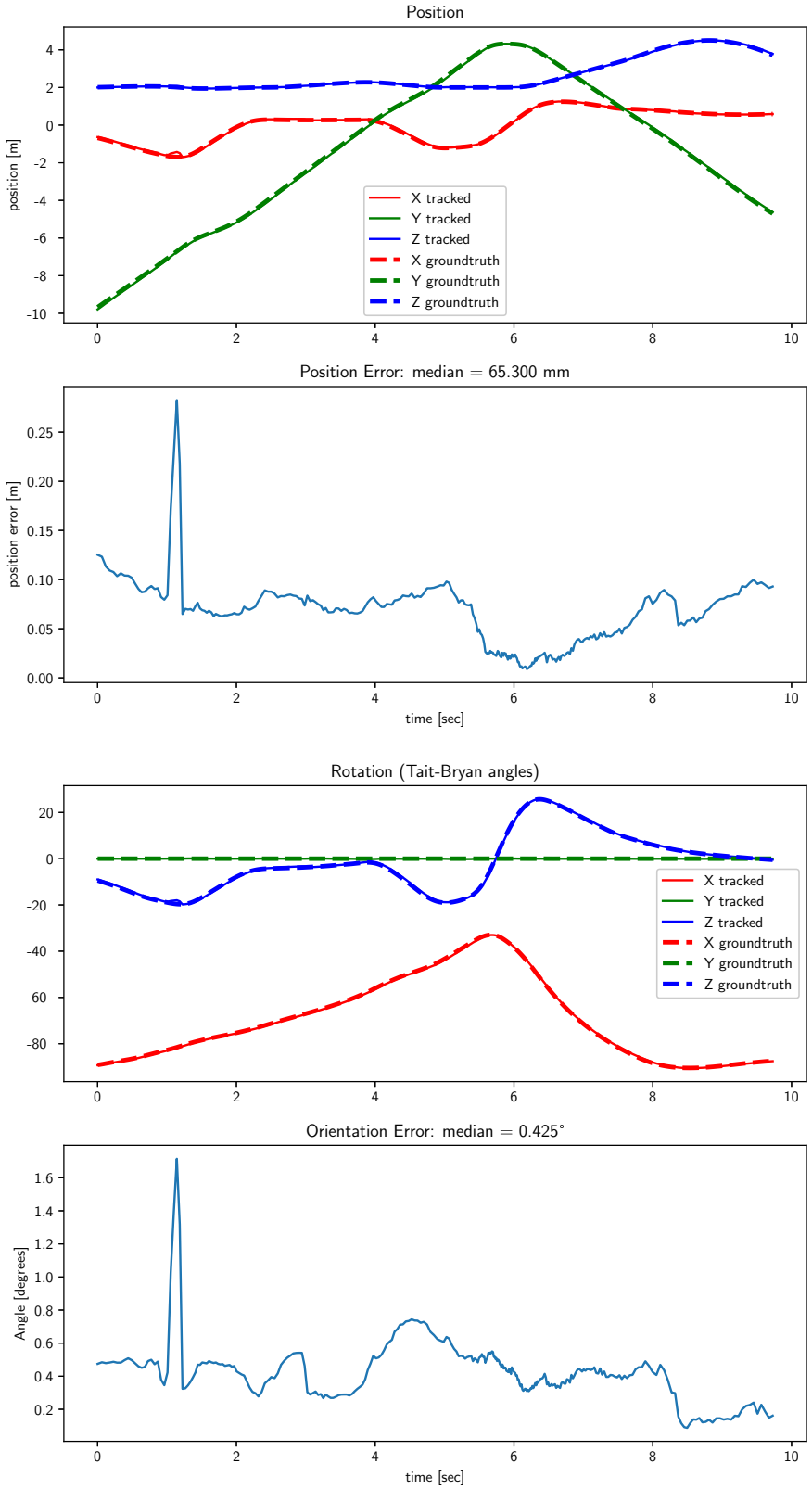


Figure 4.4: Pose tracking results for a simulated flight trough an atrium (see Figure 4.3 for a 3D view of the scene).

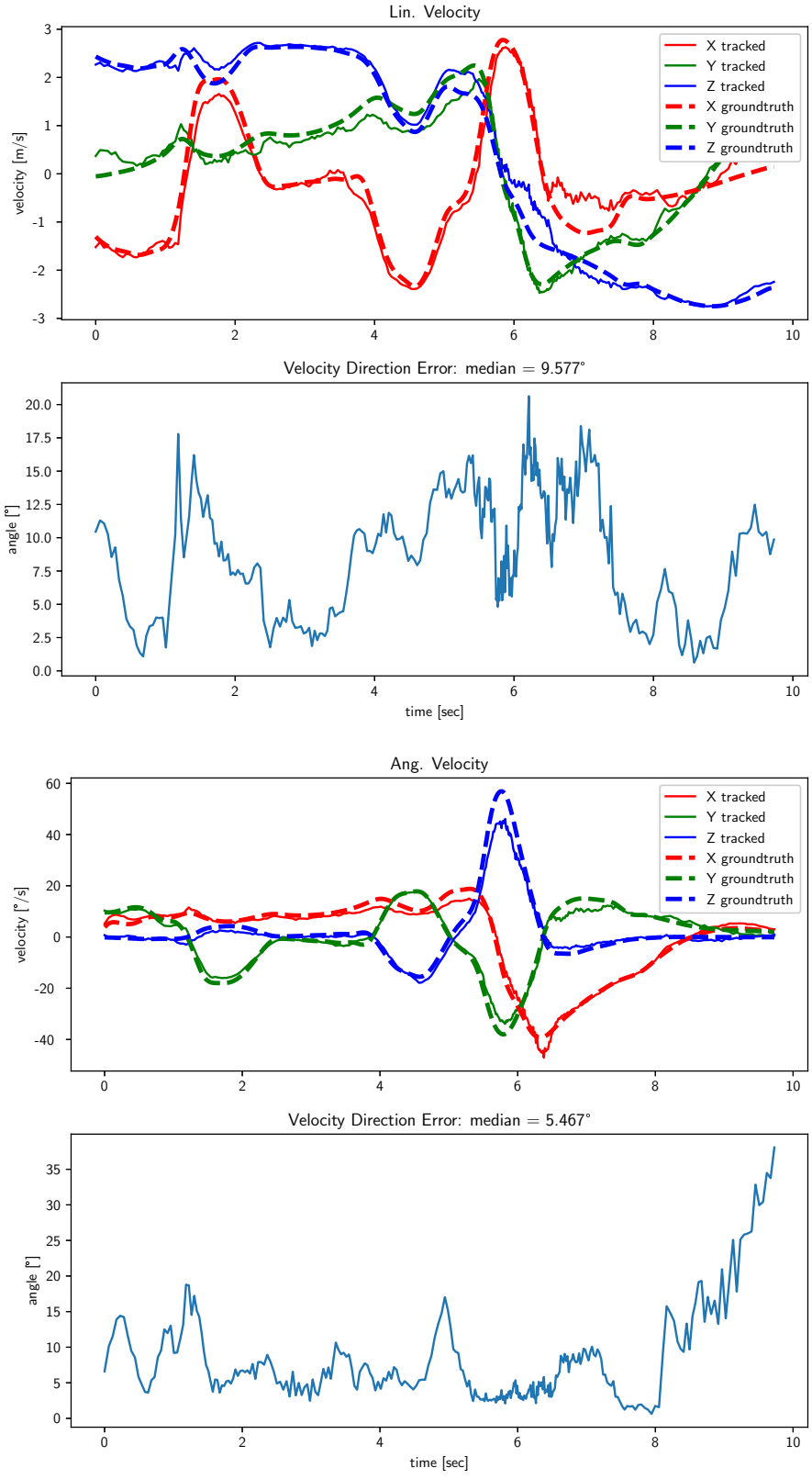


Figure 4.5: Velocity tracking results for the atrium scene, the same trajectory as shown in Figure 4.3 and Figure 4.4. Note that the measured velocities are scaled to match the ground-truth values!

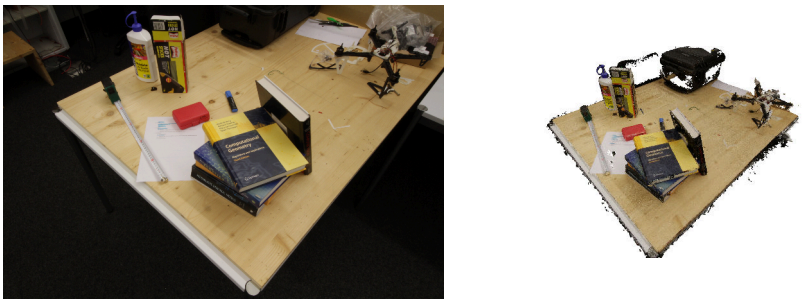


Figure 4.6: Table scene as it appeared in real life (left) and the resulting SfM model (right).

4.3 Real Data

Most of the experiments were conducted using the mesh of a room built in Section 3.4. Ground truth data was recorded using a commercial motion capture system and as mentioned only contains poses, no velocities.

4.3.1 Table Scene

More a qualitative experiment than anything else, we also tried our tracking on a scene generated solely using SfM. Unfortunately no ground-truth is available here, but it shows the tracking also works well on a map created using inexpensive means.

4.3.2 Flying Room

The real test is running our implementation on the map we generated in Section 3.4: Here, tracking is more challenging due to uniformly white walls and a floor with a quite high-frequency texture.

In this scene ground-truth data is provided by a motion capture system.

4.4 Timing

Non-linear optimization on images is slow, as even the fairly low resolution DAVIS346 camera has 90'000 pixels. The current implementation is built on top of ceres [1] and is one to three orders of magnitudes slower than real-time, depending on the settings.

4.5 Discussion

[1] track 2, tracking failed after about 18 seconds

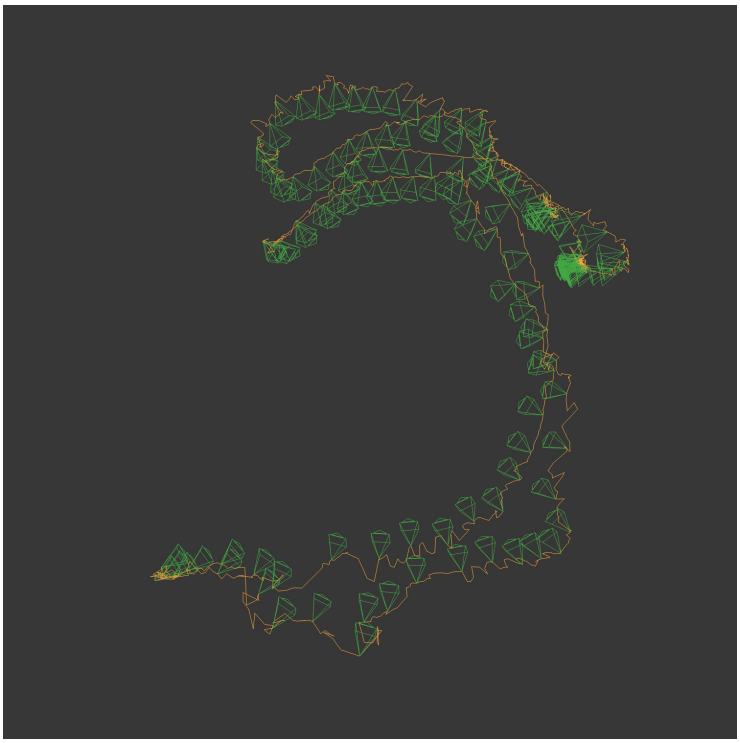


Figure 4.7: The recovered trajectory where the camera was moved around the table by hand.

Table 4.1: Computation time required for tracking.

		Processing		Full Tracking Step		
Scene	Length	Time	Events/sec	Iter./step	sec/iter.	total
Carpet 2D	20s	8771.6s	2857.2	24.656	120.06ms	3024.7ms
Carpet 2D downsampled	20s	1957.0s	12789	17.005	38.516ms	675.743ms
Boxes	20s	10772s	3160.1	25.942	132.26ms	3418.5ms
Boxes downsampled	20s	5274.7s	6433.0	27.586	59.585ms	1679.3ms
Sponza	10s	637.97s	4130.5	12.234	166.07ms	2098.6ms
Sponza downsampled	10s	229.25s	11532	17.344	41.012ms	751.63ms
Flying Room, track 1	8.32s	10058s	994.99	64.634	359.54ms	22704ms
Flying Room, track 2B	43.24s	52094s	1016.2	90.307	315.08ms	22139.2ms

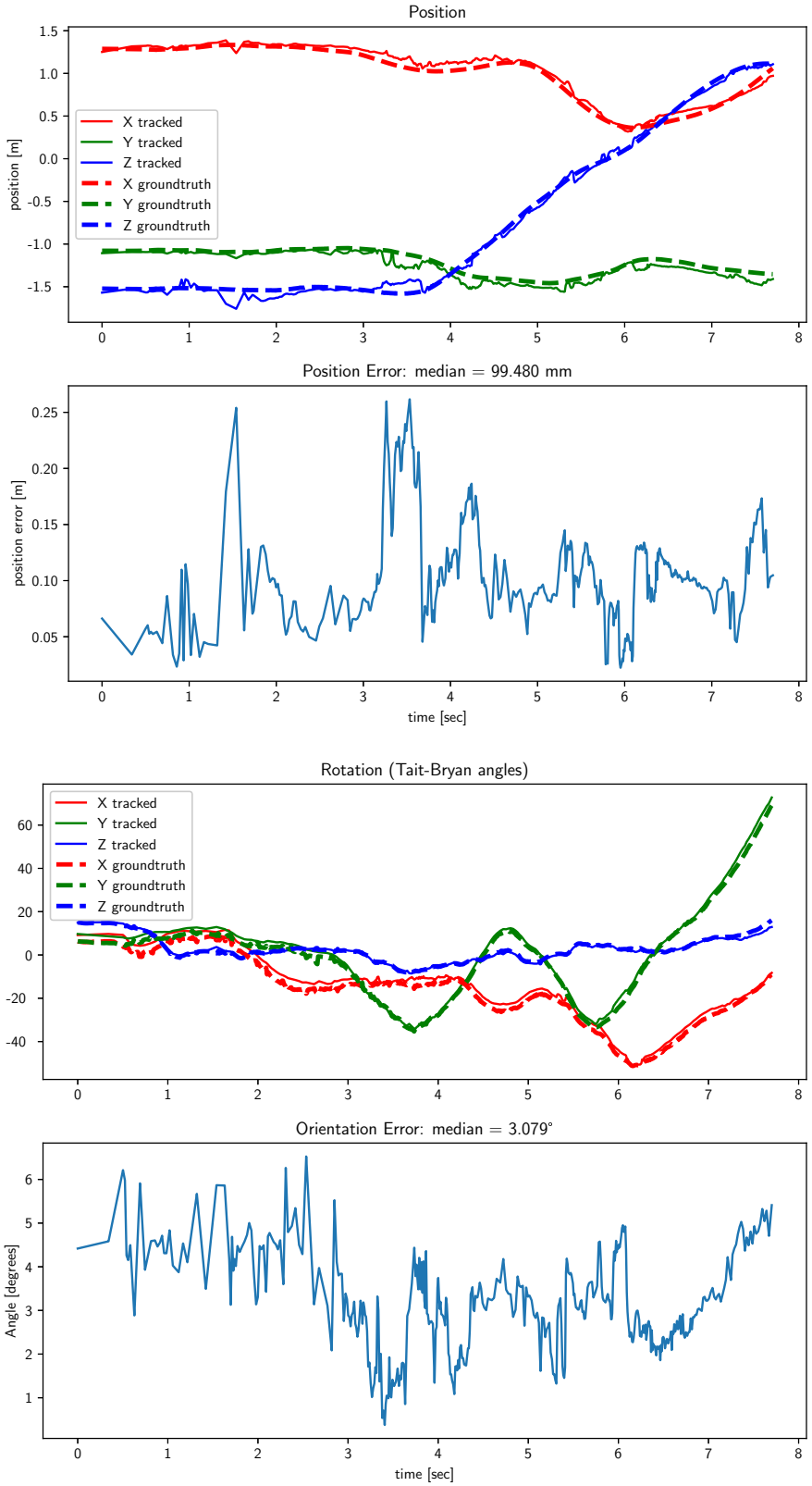


Figure 4.8: Real-world tracking performance using the mesh shown in Figure 3.15.

Table 4.2: Median accuracy of tracking of all experiments.

		Mean	Pose		Velocity Direction	
Scene	Real?	Depth	Position	Orientation	Linear	Angular
Carpet 2D	N	2.10887m	7.2640mm	0.15878°	23.561°	17.760°
Carpet 2D downsampled	N	”	11.223mm	0.20534°	20.235°	16.168°
Boxes	N	1.99285m	4.4596mm	0.20304°	17.537°	62.686°
Boxes downsampled	N	”	8.8887mm	0.28481°	17.532°	61.172°
Sponza	N	10.5994m	65.30mm	0.42514°	9.5767°	5.4670°
Sponza downsampled	N	”	69.040mm	0.52006°	11.171°	6.0394°
Flying Room, track 1	Y	3.19973m	99.480mm	3.0794°	n/a	n/a
Flying Room, track 2A	Y	3.41492m	96.410mm	3.9917°	n/a	n/a
Flying Room, track 2B	Y	”	88.232mm	3.8392°	n/a	n/a
Flying Room, downsampled[1]	Y	”	92.691mm	3.6311°	n/a	n/a

Pose tracking performs precise, though occasionally converges to a local minima instead. This is sometimes recovered in subsequent frames, but if there isn't enough texture tracking can and will fail.

Also, while poses are estimated fairly precise, the velocity estimate is very inaccurate and noisy.

Even tough the real world is much more challenging with noisy data, a noisy model, uniformly white walls etc., the tracking still performs admirably (see bottom half of Table 4.2).

Also note that even the ground truth is not perfect, as both the motion capture system and the hand-eye calibration are unlikely to be significantly more accurate than a millimeter. There is also slight jitter in the timing introduced by the recording setup.

Unfortunately, our current implementation does not run in real-time and in fact is up to several orders of magnitude slower, as can be seen by comparing the second and third column in Table 4.1. Lowering the resolution speeds up computation considerably without loosing too much accuracy (see rows marked ”downsampled”)

Chapter 5

Discussion

We've shown that accurate tracking using only events with a photometric map is possible using non-linear optimization.

Our method estimates not only pose but also velocity, which is usually very useful information (see also Section 5.2.3). However, the velocity is estimated only up to scale and is very noisy in practice, which makes its usefulness questionable.

Unfortunately, non-linear optimization is computationally expensive and we did not achieve real-time operation, let alone with low latency.

As with any tracking relying on vision only, it only works in scenes with enough texture. Additionally, our method is also quite sensitive to close objects where high optic flow can cause a single edge to dominate, leading to poor tracking or even complete loss of tracking.

In this work we only looked at tracking, so another disadvantage of our current implementation is that we rely on a map of the environment. There are other approaches to tracking with events that also include a mapping part, such as [17] and [9].

5.1 Conclusion

We implemented photometric tracking using only events and a model of the environment and explored the accuracy of this method. While the method works, it doesn't seem to be all that robust, maybe betraying the fact that we rely on intensity derivatives instead of absolute intensities, like classical photometric approaches.

In the end it seems apt to ask the question if our method as it currently stands even brings any significant advantages compared to photometric tracking with a classical camera. Due to the integration of the events into eventframes we discard a lot of the unique information provided by an event camera, while the slow tracking of the non-linear optimization negates the low latency of the sensor.

We also investigated multiple low-cost methods of generating an accurate photometric map. We have shown that it is possible to create good maps using commodity hardware, without requiring expensive equipment like a laser-scanner.

5.2 Future Work

Let's look at some possible improvements or extensions to this work:

5.2.1 Parametrization

While some measures to increase the speed of the current implementation are discussed in Appendix C, especially downsampling the images, one potentially substantial improvement was not investigated: Parallelization.

The calculation of the the Jacobian takes the most time (around 90%) and is a task that could be mostly parallelized, as every pixel is independent and only the normalization step at the end depends on all values. It is also a task well suited for running on a GPU, as there are many small, identical and independent calculations.

Further parallelization is tougher tough, as the optimization itself is inherently sequential.

5.2.2 IMU Integration

One of the natural extensions to any kind of visual odometry is the addition of inertial measurements. IMUs are cheap, fast and provide very useful information as they don't depend on features/texture being visible.

A few ways in which inertial data could be used, in ascending order of complexity:

- Instead of calculating angular velocity during the optimization, direct IMU readings could be used instead.
- Forward integration of the linear acceleration would provide the other half of the velocity and would at least provide a good starting guess to initialize the optimization with, maybe even making the velocity term in the optimization superfluous.
- Of course, integration of the IMU readings could also be used to estimate poses. However, cheap IMUs such as the one included in the DAVIS event camera are very noisy and make dead-reckoning for longer than about 100ms is unrealistic. Including a more sophisticated motion-model as described below in Section 5.2.3 might be a good or even required addition to this.
- A quite different approach would be to include the IMU terms in the error-term of optimization itself. However, finding a good weight for this

additional term relative to the photometric one might not be straightforward.

Integrating IMU data would make tracking significantly more robust and potentially even faster by providing better initialization for the optimizer and potentially reducing the numbers of parameters to estimate (Although additional terms for IMU bias would have to be added).

5.2.3 Motion Model and Filtering

Out of necessity, our approach already estimates the velocity alongside the poses. This information is actually valuable as it would allow the prediction of the next pose, thus potentially improving the optimization.

Including a motion model would also go along well with including IMU measurements, as discussed above.

While a simple constant velocity motion model might be enough, a more complex model or filter that also looks at acceleration and past poses might further improve robustness, by allowing the tracker to better detect and deal with wrong or failed alignments (for example when the visible texture isn't good enough).

5.2.4 Re-Localization

The optimization framework will get stuck in a local minima when it isn't initialized close enough to the true position. This means it works well to estimate continuous motion, where the new pose is still fairly close to the previous one and we might even have inertial data available which we can forward integrate. It completely fails when we have no coarse position available tough, such as at the start or when the tracking fails.

We dealt with this issue by manually initializing the algorithm, but for real world applications some automated way of figuring out a rough absolute position is required. This could be achieved by using classical feature matching on the DAVIS' APS sensor against precomputed features from the map.

See for example [10] for an approach employing compressed map descriptors in combination efficient search algorithms. In [4] a method using neural networks to select robust features is described.

5.2.5 Live Mapping

We implemented the tracking part. It would be interesting to extend this work to full SLAM, in order to correct and extend the map.

A mapping approach using multi-view stereo algorithm adapted for event data is presented in [**?**]. While the extracted geometry consists only of discrete lines, it might be possible to extend it to a more detailed representation containing intensities or gradients.

5.2.6 Other Error Functions

When integrating the events into an event frame, we discard a lot of useful information about the exact timing of the events. We could have just used a (fast) normal camera instead!

So it might be worth figuring out how to best make use of the high temporal resolution provided by an event camera. In general it is not clear how to match what is essentially a point-cloud (the events over time and space) to a model.

To get an accurate prediction for the events from a model we would also need to know the event threshold, which is a bigger problem than just estimating another parameter, as it turns out that in practice this threshold isn't even constant and depends on various factors such as the time to the previous event.

One idea might be to integrate the events into a three-dimensional grid (with the third axis being time) instead of the two dimensions used here. We could then match this "3D image" to one generated by simulating an event camera moving between two poses (or a pose + velocity) using a similar non-linear optimization approach.

For more information about simulating and predicting events and the problems that arise see [12].

Another interesting approach is discussed in [11], where the effects of fast motions on the image of a classical camera are taken into account, by explicitly modeling motion blur and rolling shutter deformations. Similar techniques might be applicable to event data.

A simple way to make use of the event's fine temporal resolution might be to correct for the motion while integrate the events as described in [16].

5.2.7 Comparison to Filter Approach

A very different solution to the same problem of tracking with events on a photometric map is shown in [6]. It would be very interesting indeed to compare our method with the filtering approach from [6] in terms of robustness, accuracy and computational complexity.

5.2.8 Event Selection

Selecting a good integration window for the events is important to get a good result. Too few and there isn't enough structure for good tracking, too many and motion blur becomes a problem. Further, a shorter integration window considerably increases the accuracy while at the same time lowering the throughput, thus increasing the computational load.

While our approach of choosing a constant number of events has the nice property of automatically adapting to various camera speeds and works quite well, problems arise when an object is close to the camera, causing large flow in the image and thus generating many events. This not only causes motion blur but

also leads to less events collected elsewhere in the image, so that a single edge might completely dominate the optimization.

One way of dealing with this would be to require a certain event density in every part of the image, e.g. at least N events per quadrant. However, as events are sparse it might very well happen that there are almost no events in a certain region. Thus requiring at least N events in well distributed regions of the image might be a better heuristic.

Appendix A

Additional Equations

For the optimization we need the derivative of our error function which we can calculate it by separating out the functions it is composed of.

A.1 Projections and Their Derivatives

While the derivation of the projection function $\pi(\boldsymbol{p})$ is straightforward, for it's counterpart, the back-projection function $\pi^{-1}(\boldsymbol{u})$ is less so because $\pi(\boldsymbol{p})$ is not injective and thus not invertible due the loss of depth information.

We therefore need to add the depth to the formulation of $\pi^{-1}(\boldsymbol{u})$ somehow. In the following three ways to do this are presented.

A.1.1 Projection

The derivatives of the projection function (Equation 2.3) is:

$$D_p\pi = \frac{1}{p_z}\begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix} \tag{A.1}$$

A.1.2 Back-Projection with Constant Z

The simplest way of taking the depth into account is to treat it as a constant, i.e. $\pi^{-1}(\boldsymbol{u}) = \pi_z^{-1}(\boldsymbol{u})$ and thus the derivative is simply:

$$D_u\pi_z^{-1} = \begin{bmatrix} z & 0 \\ 0 & z \\ 0 & 0 \end{bmatrix} \tag{A.2}$$

A.1.3 Back-Projection with Independent Z

However, the depth is not actually constant. We can treat it as an independent third parameter instead, i.e. $\pi^{-1}(\boldsymbol{u}) = \pi^{-1}(\boldsymbol{u}, z)$ and get

$$D_u\pi^{-1} = \begin{bmatrix} z & 0 & u_x \\ 0 & z & u_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$

This has the nice effect that if we extend the projection operator π to return the depth as a third value

$$\pi(\boldsymbol{p}) := \begin{bmatrix} p_x/p_z \\ p_y/p_z \\ p_z \end{bmatrix} \tag{A.4}$$

the functions π and π^{-1} become the true inverse of each others

$$\begin{aligned} \forall \boldsymbol{u} \in \mathbb{R}^2 : \pi(\pi^{-1}(\boldsymbol{u})) = \boldsymbol{u} \\ \forall \boldsymbol{p} \in \mathbb{R}^3 : \pi^{-1}(\pi(\boldsymbol{p})) = \boldsymbol{p} \end{aligned} \tag{A.5}$$

and therefore the Jacobians as well:

$$\begin{aligned} (D_u\pi^{-1})^{-1}\Big|_{\pi(\boldsymbol{u})} &= \begin{bmatrix} z & 0 & x \\ 0 & z & y \\ 0 & 0 & 1 \end{bmatrix}^{-1}_{\pi(\boldsymbol{u})} \\ &= \begin{bmatrix} z & 0 & x/z \\ 0 & z & y/z \\ 0 & 0 & 1 \end{bmatrix}^{-1} \\ &= \frac{1}{z} \begin{bmatrix} 1 & 0 & -x/z \\ 0 & 1 & -y/z \\ 0 & 0 & z \end{bmatrix} \\ &= D_u\pi\big|_{\boldsymbol{p}} \end{aligned} \tag{A.6}$$

A.1.4 Back-Projection with Dependent Z

Another way of treating the depth as a variable is by saying that it is actually dependent on the pixel position: $Z = Z(\boldsymbol{u})$. Treating Z as a function of u_x and u_y is closest to actual implementation where we have an intensity image with a depth image, but it is a bit strange as the depth function is not continuous and therefore the derivative $\nabla Z(\boldsymbol{u})$ is not properly defined everywhere, crucially at edges, usually the most interesting part.

The back-projection function would be

$$\boldsymbol{p} = \pi^{-1}(\boldsymbol{u}) := Z(\boldsymbol{u}) \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} \tag{A.7}$$

and its derivative:

$$D_{\pi^{-1}} = \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} \nabla Z(\boldsymbol{u})^\top + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} Z(\boldsymbol{u}) = \begin{bmatrix} \frac{\partial Z}{\partial x} u_x + Z & \frac{\partial Z}{\partial y} u_x \\ \frac{\partial Z}{\partial x} u_y & \frac{\partial Z}{\partial y} u_y + Z \\ \frac{\partial Z}{\partial x} & \frac{\partial Z}{\partial y} \end{bmatrix} \tag{A.8}$$

A.2 Image Rendering Jacobian

Instead of warping a keyframe we can also render the full map directly as described in Section B.1. Taking the derivative of this rendering function would be very expensive, as the full map and various non-linearities (such as the fragment shader) would have to be taken into account. Instead we can approximate the derivative of the rendering function using the feature sensitivity matrix, as it directly relates the motion of the camera (i.e. the derivative of its position) to changes in the image:

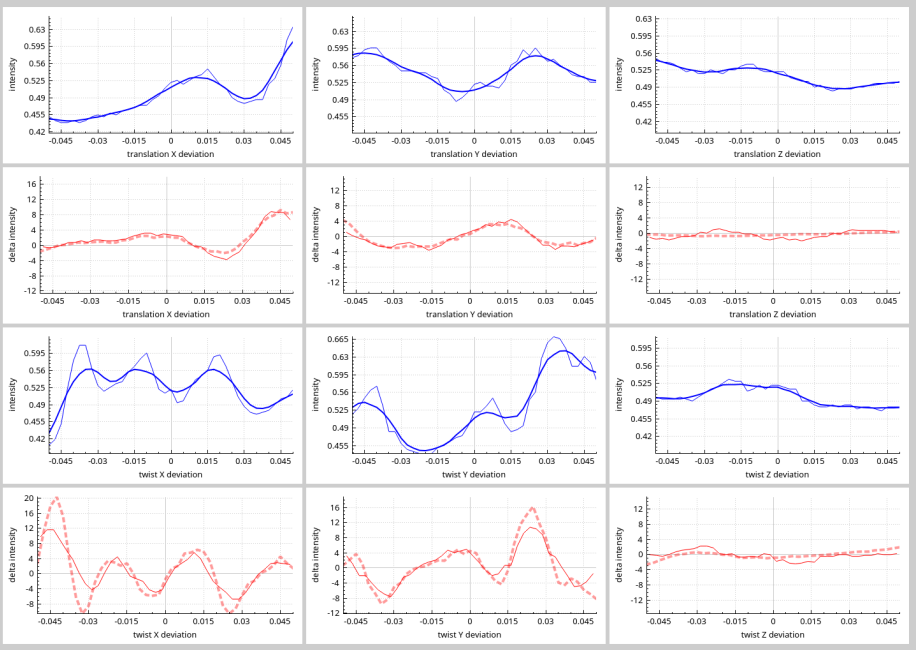
$$\frac{\partial M}{\partial \boldsymbol{T}} = J_p \tag{A.9}$$

Where J_p is defined in Equation 2.10.

Apart from this approximation not taking occlusions into account it also requires the angular derivations of the pose, which means we have to parametrize the minimization with relative twist coordinates instead of absolute quaternions.

A.3 Gradient Correction

As mentioned in Section 2.4.2, transferring an image gradient from one perspective to another is not as straightforward as warping intensities. A surface seen from another viewpoint can generate quite different image gradients. For example, rotation around the Z-axis will also rotate the gradient. Moving closer to a surface will "zoom in" on it and stretch out a gradient, thus making it weaker.



intensity values as rendered by OpenGL

smoothed intensity values (average filter)

numeric derivatives of smoothed intensity values

analytic derivatives from interaction matrix

Figure A.1: Numerical derivation of the map rendering function for a single pixel compared to the analytic solution from the feature sensitivity matrix. An interactive version of this plot can be found in the GUI under plot › check renderer Jacobian

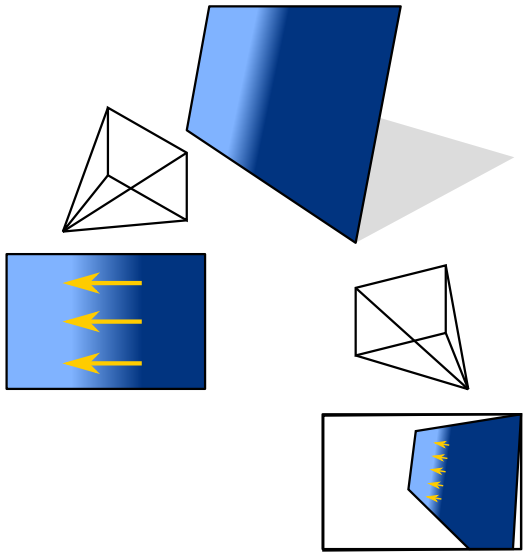


Figure A.2: A surface with an intensity gradient viewed from two different perspectives. Notice how the image gradient changes magnitude and direction depending on the camera's position.

We therefore have to take this effect into account when warping a gradient by multiplying the warped gradient with the Jacobian of the warping operator $\tau(\boldsymbol{u})$:

$$\begin{aligned} I({}_A\boldsymbol{u}) &= I(\tau_{AB}({}_B\boldsymbol{u})) \\ \nabla I({}_A\boldsymbol{u}) &= \nabla I(\tau_{AB}({}_B\boldsymbol{u})) \cdot \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} \end{aligned} \tag{A.10}$$

Where the correction factor is

$$\begin{aligned} \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} &= \left.\frac{\partial}{\partial \boldsymbol{u}} \pi(\boldsymbol{R}\pi^{-1}(\boldsymbol{u}) + \boldsymbol{t})\right|_{{}_B\boldsymbol{u}} \\ &= \left. J_\pi \right|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \left. J_{\pi^{-1}} \right|_{{}_B\boldsymbol{u}} \end{aligned} \tag{A.11}$$

and the 3D point ${}_A\boldsymbol{p}$ in frame A is:

$${}_A\boldsymbol{p} := \boldsymbol{R}_{AB}\pi^{-1}({}_B\boldsymbol{u}) + \boldsymbol{t}_{AB} \tag{A.12}$$

The calculation of $\frac{\partial \tau}{\partial \boldsymbol{u}}$ will of course depend on the which version of π^{-1} we use, i.e. how the depth is parametrized (see also Section A.1).

A.3.1 Constant Depth

The simplest case is where Z is just a constant. Substituting Equation A.1 and A.2 in Equation A.11 leads to

$$\left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} = \frac{1}{p_z}\begin{bmatrix}1 & 0 & -p_x/p_z\\ 0 & 1 & -p_y/p_z\end{bmatrix}\Bigg|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \begin{bmatrix}z & 0\\ 0 & z\\ 0 & 0\end{bmatrix} \tag{A.13}$$

Translation Only

If we restrict the motion to translation only and drop the rotation (i.e. $\boldsymbol{R}$ is identity), ${}_A\boldsymbol{p}$ becomes

$$\begin{aligned} {}_A\boldsymbol{p} &= \pi^{-1}({}_B\boldsymbol{u}) + \boldsymbol{t}_{AB} \\ &= z\begin{bmatrix}u_x\\ u_y\\ 1\end{bmatrix} + \boldsymbol{t}_{AB} \end{aligned} \tag{A.14}$$

and Equation A.13 simplifies to

$$\begin{aligned} \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} &= \frac{1}{p_z}\begin{bmatrix}1 & 0 & -p_x/p_z\\ 0 & 1 & -p_y/p_z\end{bmatrix}\Bigg|_{{}_A\boldsymbol{p}} \begin{bmatrix}z & 0\\ 0 & z\\ 0 & 0\end{bmatrix} \\ &= \frac{1}{z+t_z}\begin{bmatrix}1 & 0 & -\frac{u_x+t_x}{z+t_z}\\ 0 & 1 & -\frac{u_y+t_y}{z+t_z}\end{bmatrix}\begin{bmatrix}z & 0\\ 0 & z\\ 0 & 0\end{bmatrix} \\ &= \frac{z}{z+t_z}\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \end{aligned} \tag{A.15}$$

This makes sense geometrically: Moving the camera parallel to it's image plane (translation in x and y) will not change the appearance of any gradient, but if we move closer or further away from a surface its gradient will appear stretched out (and thus weaker) or compressed (zoomed out and thus stronger).

Also, no amount of movement will change the direction of the gradient, only the magnitude will be affected.

Rotation Around Z Only

Another simple case is where we only consider rotation around the Z-Axis. This time, $\boldsymbol{t}_{AB}$ becomes zero and $\boldsymbol{R}_{AB}$ is

$$\boldsymbol{R}_{AB} = \boldsymbol{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}(\theta) & 0 \\ 0 & 1 \end{bmatrix} \tag{A.16}$$

Where $\boldsymbol{R}(\theta)$ is the 2D rotation matrix. Our transformed point ${}_A\boldsymbol{p}$ is therefore

$$\begin{aligned} {}_A\boldsymbol{p} &= \boldsymbol{R}_z(\theta) \cdot \pi^{-1}({}_B\boldsymbol{u}) \\ &= z \begin{bmatrix} \boldsymbol{R}(\theta) \cdot {}_B\boldsymbol{u} \\ 1 \end{bmatrix} \end{aligned} \tag{A.17}$$

Using Equation A.16 and Equation A.17 in Equation A.13 leads to

$$\begin{aligned} \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} &= \frac{1}{p_z} \begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix}\Bigg|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \begin{bmatrix} z & 0 \\ 0 & z \\ 0 & 0 \end{bmatrix} \\ &= \frac{1}{z} \begin{bmatrix} I & -\boldsymbol{R}(\theta) \cdot {}_B\boldsymbol{u} \end{bmatrix} \cdot z \begin{bmatrix} \boldsymbol{R}(\theta) \\ 0 \end{bmatrix} \\ &= \boldsymbol{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \end{aligned} \tag{A.18}$$

Which again makes sense geometrically: Rotation around the camera's Z-axis is identical to rotating the image in 2D and thus rotating all gradients by the same amount.

A.3.2 Independent Depth

We can use a third, independent parameter for the depth and substitute Equation A.1 and A.3 in Equation A.11 to derive the gradient correction factor as follows:

$$\left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} = \frac{1}{p_z} \begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix}\Bigg|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \begin{bmatrix} z & 0 & u_x \\ 0 & z & u_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.19}$$

Which is essentially the same equation as for the case of a constant Z (Equation A.13), only with a third column for the additional parameter.

Translation Only

We can again restrict the motion to translation only and drop the rotation:

$$\begin{aligned} {}_A\boldsymbol{p} &= \pi^{-1}({}_B\boldsymbol{u}) + \boldsymbol{t}_{AB} \\ &= z \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} + \boldsymbol{t}_{AB} \qquad = \begin{bmatrix} zu_x + t_x \\ zu_y + t_y \\ z + t_z \end{bmatrix} \end{aligned} \tag{A.20}$$

and Equation A.13 simplifies to

$$\begin{aligned} \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} &= \frac{1}{p_z} \begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix} \Bigg|_{{}_A\boldsymbol{p}} \begin{bmatrix} z & 0 & u_x \\ 0 & z & u_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{z + t_z} \begin{bmatrix} 1 & 0 & -\frac{zu_x + t_x}{z + t_z} \\ 0 & 1 & -\frac{zu_y + t_y}{z + t_z} \end{bmatrix} \begin{bmatrix} z & 0 & u_x \\ 0 & z & u_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{z}{z + t_z} \begin{bmatrix} 1 & 0 & u_x - \frac{zu_x + t_x}{z + t_z} \\ 0 & 1 & u_y - \frac{zu_y + t_y}{z + t_z} \end{bmatrix} \end{aligned} \tag{A.21}$$

Which is the same expression as in the Equation A.15 for constant depth, only with an additional column for the new depth parameter.

A.3.3 Dependent Depth

Treating the depth as a function of the pixel coordinate is where things get hairy though. Using Equation A.1 and A.8 in Equation A.11:

$$\begin{aligned} \left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{{}_B\boldsymbol{u}} &= \frac{1}{p_z} \begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix} \Bigg|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \left(\begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} \nabla Z(\boldsymbol{u})^\top + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} Z(\boldsymbol{u}) \right) \\ &= \frac{1}{p_z} \begin{bmatrix} 1 & 0 & -p_x/p_z \\ 0 & 1 & -p_y/p_z \end{bmatrix} \Bigg|_{{}_A\boldsymbol{p}} \cdot \boldsymbol{R}_{AB} \cdot \begin{bmatrix} Z'_x u_x + Z & Z'_y u_x \\ Z'_x u_y & Z'_y u_y + Z \\ Z'_x & Z'_y \end{bmatrix} \end{aligned} \tag{A.22}$$

Where $Z'_x = \frac{\partial Z}{\partial x}$ and $Z'_y = \frac{\partial Z}{\partial y}$ respectively, to make the math more readable.

Translation Only

Removing the rotation matrix by setting it to the identity, Equation A.23 leads to

$$\begin{aligned}
\left.\frac{\partial \tau}{\partial \boldsymbol{u}}\right|_{B\boldsymbol{u}} &= \frac{1}{p_z}\begin{bmatrix}1 & 0 & -p_x/p_z\\ 0 & 1 & -p_y/p_z\end{bmatrix}\Bigg|_{A\boldsymbol{p}} \cdot \begin{bmatrix}Z'_x u_x + Z & Z'_y u_x\\ Z'_x u_y & Z'_y u_y + Z\\ Z'_x & Z'_y\end{bmatrix} \\
&= \frac{1}{z+t_z}\begin{bmatrix}1 & 0 & -\frac{zu_x+t_x}{z+t_z} = -A\\ 0 & 1 & -\frac{zu_y+t_y}{z+t_z} = -B\end{bmatrix}\begin{bmatrix}Z'_x u_x + Z & Z'_y u_x\\ Z'_x u_y & Z'_y u_y + Z\\ Z'_x & Z'_y\end{bmatrix} \\
&= \frac{1}{z+t_z}\begin{bmatrix}Z'_x(u_x - A) + Z & Z'_y(u_x - A)\\ Z'_x(u_y - B) & Z'_y(u_z - B) + Z\end{bmatrix} \\
&= \frac{1}{z+t_z}\left(\begin{bmatrix}u_x - A\\ u_y - B\end{bmatrix}\nabla Z^\top + \begin{bmatrix}Z & 0\\ 0 & Z\end{bmatrix}\right)
\end{aligned} \tag{A.23}$$

Rotation around the Z axis is left as an exercise for the reader.

A.3.4 Conclusion

Of course, all this correcting the gradient during warping is only accurate if the gradient itself is accurate in the first place. In practice however, a camera only has finite resolution and can only accurately capture signals below a certain frequency. Smooth gradients, for example a wall that gets brighter to one side, will be accurately transferred, but the gradients from sharp edges will not.

This is shown schematically in Figure **??** and can also be seen in the experiments of **??**. Moving closer to an edge will lead to an overestimate of the actual image gradient, while moving further away the gradient is correctly transformed, but only at the pixel where the gradient was in the original frame! The surrounding ones will be completely unaffected and thus wrong.

To accurately warp gradients in practice it is most likely necessary to take the discrete nature of the problem into account, instead of treating images as smooth and continuous functions.

Fortunately, we use the gradient to approximate expected changes in intensity. Using further approximations doesn't make it much worse, especially if large deviations are taken into account: For example when the keyframe is rotated by 90° we must rotate the gradients correspondingly. The more complicated formulations don't actually alter the result significantly.

A second reason that the gradient correction doesn't need to be perfect is that we have a dense map and can quickly generate arbitrary keyframes, thus keeping relative poses small and therefore also any errors introduced by bad approximations to the transferred gradient.

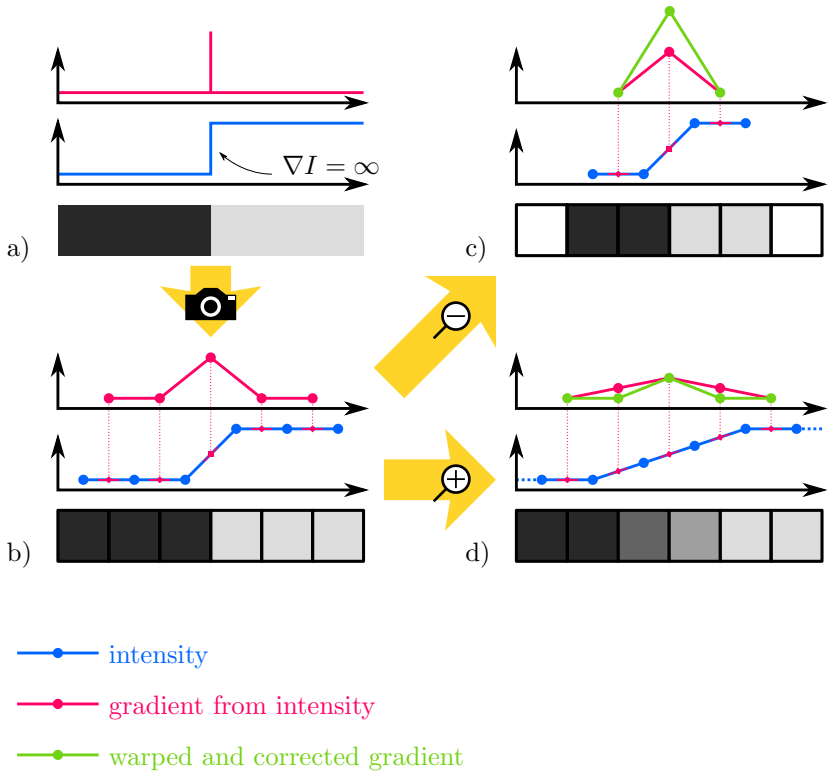


Figure A.3: Taking a picture of an infinitely sharp gradient results in wrong gradient correction when warping.
a) The original surface
b) A discretized picture of the surface from a)
c) Zooming out the discretized picture. Note how the gradient correction overestimates the actual image gradient.
d) Zooming in the discretized picture. Here the gradient correction is correct, but only at the actual edge - neighboring gradients are underestimated.

Lastly, we can even side-step the issue entirely by operating in the keyframe instead of the eventframe by transferring the velocity instead. See Section B.3 for details.

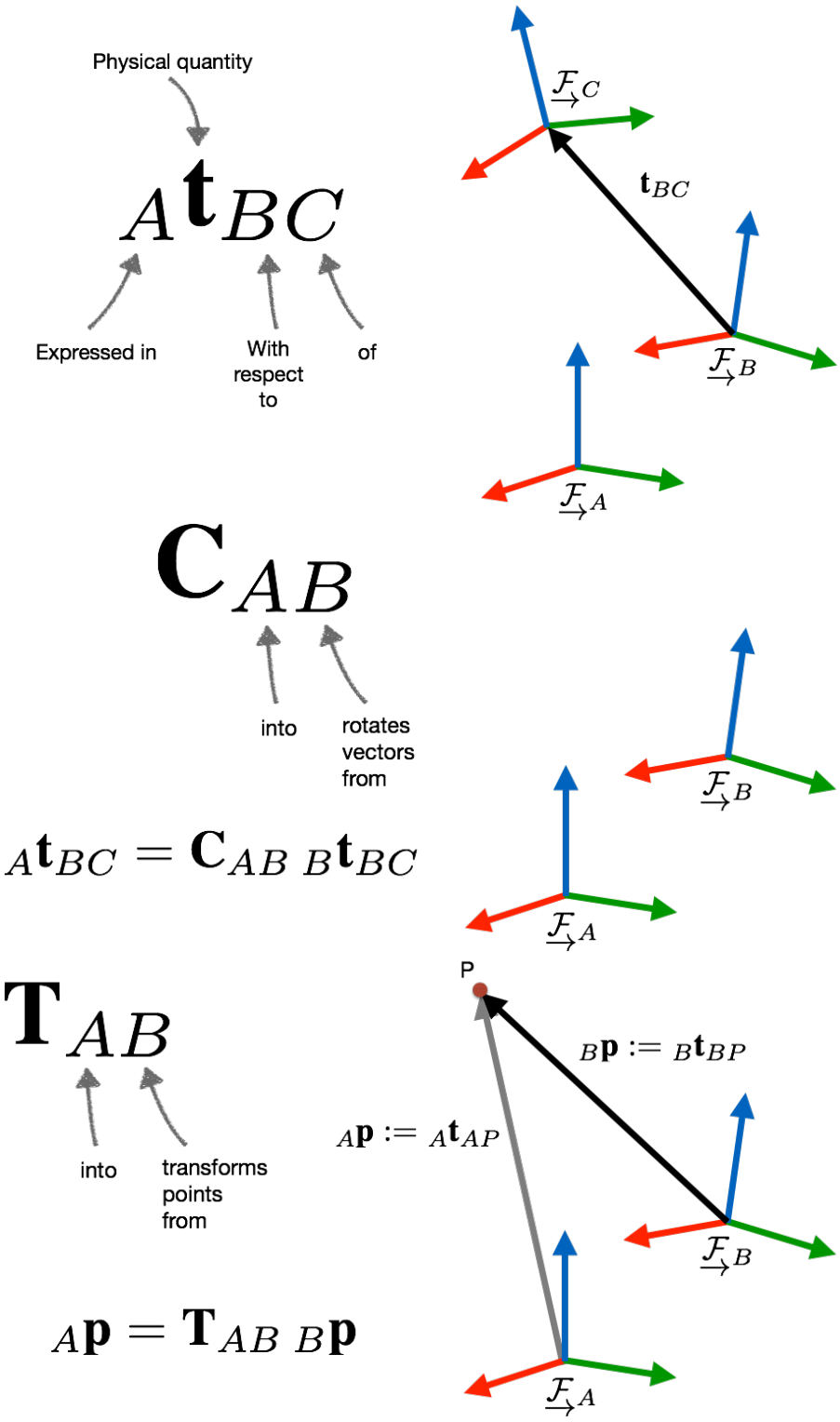


Figure A.4: Notation for frame of references. Source: `https://github.com/ethz-asl/aslam_cv2/wiki/Expressing-frame-transformations-in-code`

Appendix B

Problem Formulation Variants

While the general format of the minimization is always the same (see Equation 2.15), the implementation of $\Delta\hat{I}(\boldsymbol{u})$ can be different.

B.1 Direct Optimization

We can take advantage of the fact that we have a model which we can use to generate images from arbitrary perspectives and directly optimize on that:

$$\hat{I}_{direct}(\boldsymbol{u}) = -\nabla M(\boldsymbol{T}, \boldsymbol{u}) \cdot \dot{\boldsymbol{u}}(\dot{\boldsymbol{T}}) \tag{B.1}$$

Tough this is relatively straightforward to implement, this implies that we render a new image for every step of the minimization. Even though we can leverage a GPU for this job, it is still a relatively costly operation, especially when using a large map.

Another issue is that we need to somehow calculate the Jacobian of the rendering function M. This is addressed in Section A.2 and actually straightforward.

B.2 Warping a Keyframe

Instead of rendering the full model at every step of the optimization, we can render a keyframe $K(\boldsymbol{u})$ with depth once and re-render this. Though this does not handle occlusions, it is a much cheaper approach.

To do this re-rendering, we re-project the pixels of the keyframe into 3D space, where they are transformed using the relative transformation $\boldsymbol{T}' = \boldsymbol{R}, \boldsymbol{t}$ between

keyframe and event frame and then projected back onto the image plane:

$$M(\boldsymbol{T}, \boldsymbol{u}) \approx K(\pi(\boldsymbol{R}\pi^{-1}(\boldsymbol{u}) + \boldsymbol{t})) = K(\tau(\boldsymbol{u})) \tag{B.2}$$

To get the gradient after warping we need to take the derivative of this equation with respect to the pixel position $\boldsymbol{u}$:

$$\nabla M(\boldsymbol{T}, \boldsymbol{u}) \approx \nabla K(\tau(\boldsymbol{u})) = (\nabla K)\Big|_{\tau(\boldsymbol{u})} \cdot \frac{\partial \tau}{\partial \boldsymbol{T}} \tag{B.3}$$

This 'gradient correction' can get quite nasty, depending on how accurately it should be calculated. See Section A.3 for detailed derivations.

B.3 Warping the Event Frame

If we could warp the expected intensity changes instead, we wouldn't have to deal with warping image gradients. To do this we need to know the velocity of the camera in the keyframe's reference frame given the velocity at the eventframe.

For linear velocity, this transfer is given by:

$$\boldsymbol{v}_K = \boldsymbol{R}\boldsymbol{v}_E + \boldsymbol{R}\boldsymbol{\omega}_E \times \boldsymbol{t}_{EK} \tag{B.4}$$

and for the angular velocity

$$\boldsymbol{\omega}_K = \boldsymbol{R}\boldsymbol{\omega}_E \tag{B.5}$$

We can thus formulate our expected intensity change image function:

$$\hat{I}_{direct}(\boldsymbol{u}) = -\nabla K(\pi(\boldsymbol{R}\pi^{-1}(\boldsymbol{u}) + \boldsymbol{t})) \cdot \dot{\boldsymbol{u}}(\boldsymbol{v}_K, \boldsymbol{\omega}_K) \tag{B.6}$$

We can actually run the optimization on the velocity in the keyframe's reference frame and only have to transfer the velocities at initialization and at the end of the optimization.

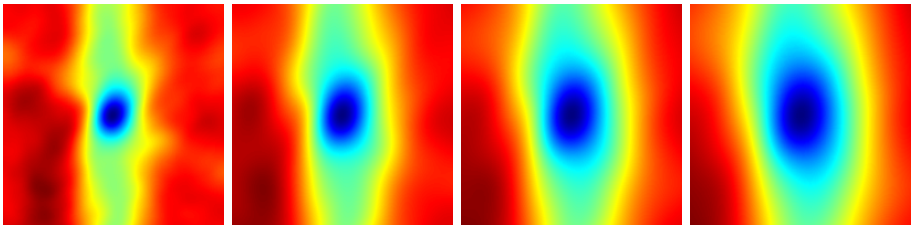
Appendix C

Implementation Details

C.1 Blurring

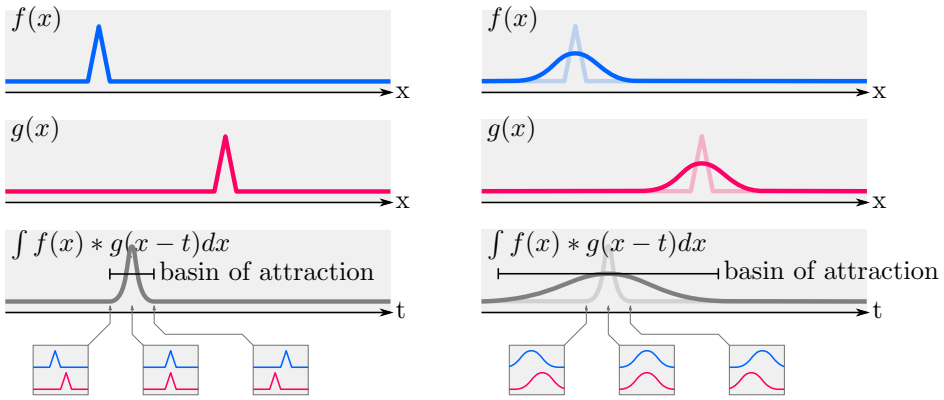
In order to increase the convergence radius and making the tracking more robust we can apply some Gaussian blur to the images. As can be seen in Figure C.1 even a small blurring kernel of 9 by 9 pixels doubles the convergence radius. This works especially well in our case because we're dealing with relatively sparse data, as events and gradients mostly occur along edges (see also Figure C.2).

However, blurring of course reduces accuracy. This can be countered by doing multiple steps, first optimizing on a blurred image to get close to the minima and then doing a second run of the optimizer without blur.



(a) No blur. (b) 9px (c) 13px ($2 \cdot 9$px) (d) 16px ($3 \cdot 9$px)

Figure C.1: The error function, plotted against an X and Y displacement of $\pm 20cm$ with various amount of Gaussian blur applied to the keyframe and event-frame.



(a) Correlating two 1D functions with only very local support and thus only a small basin of attraction.

(b) Correlating two slightly blurred 1D functions greatly increases the basin of attraction.

Figure C.2: The effect of blurring functions on correlating them.

Appendix D

Software Manual

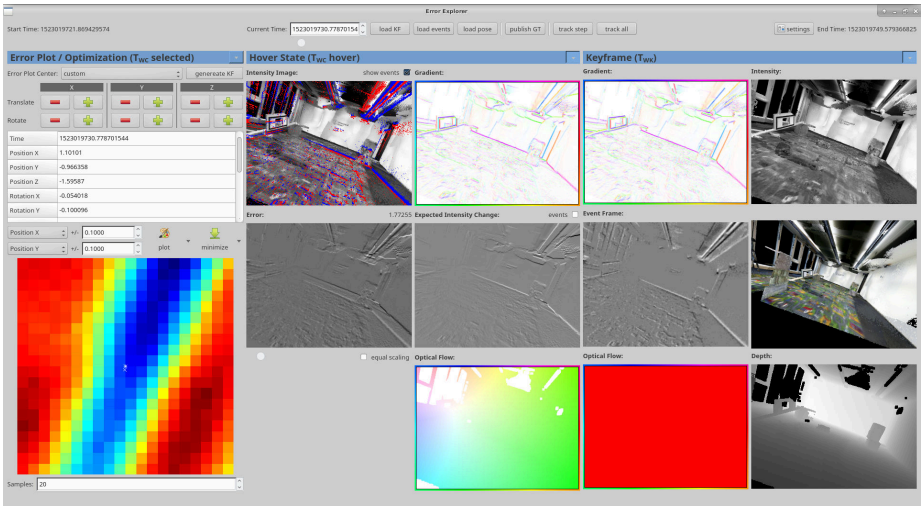


Figure D.1: Main window of tracking software with poses and keyframes loaded.

D.1 Dependencies

The software was developed and tested against **ROS Kinetic Kame** and **Ubuntu 16.04** (Xenial Xerus).

ROS dependencies that are not in the repositories:

Catkin simple: `https://github.com/catkin/catkin_simple`

Ceres: `https://github.com/ethz-asl/ceres_catkin`

glog: `https://github.com/ethz-asl/glog_catkin`

Assimp: `https://github.com/uzh-rpg/assimp_catkin`

DVS messages: `https://github.com/uzh-rpg/rpg_dvs_ros`

Other dependencies. They should all be available through the package management of Ubuntu 16.04:

- Qt 5.5 or higher (or a bit lower might be fine too)
- GCC 5.4 or higher (must support the C++14 standard)
- CMake 3.1 or higher
- OpenCV 3
- Eigen 3
- yaml-cpp
- PCL 1.7 (only for loading PLY files, could easily be replaced with assimp)
- Boost
- QCustomPlot (sudo apt install libqcustomplot-dev)
- GLM (sudo apt install libglm-dev)

D.2 Installation

Building in release mode is recommended:

```
catkin config ––cmake–args  –DCMAKE_BUILD_TYPE=Release
```

All the software is contained in a ROS package and can be installed and run using catkin.

```
catkin build direct_event_tracker
```

To launch it pass the path to a configuration file. For example:

```
rosrun direct_event_tracker direct_event_tracker cfg/main.yaml
```

D.3 Ingredients

Apart from installing all the dependencies and the software itself, you will also need several pieces of data. The path to these is set in the configuration file.

D.3.1 rosbag

A rosbag containing:

- the events (on `/dvs/events` or `/cam0/events`)

- ground truth (either on `/cam0/pose` and `/cam0/twist` or on `/optitrack/DAVIS_BlueFox`)
- IMU data (optional, on `/dvs/imu` or `/imu`)
- keyframes (optional, on `/cam0/image_raw`)

This rosbag can be either a real recording (using the `rpg_dvs_ros` package) or a simulation (from `rpg_event_camera_simulator`)

D.3.2 Map

The map can be either a point cloud (in .PLY format, must include normals; for example generated by ElasticFusion) or a mesh (in .OBJ format).

Note that there are various coordinate systems in use and you might have to rotate the model to use the correct one! When exporting as .obj from Blender for the event simulator choose 'Z Up' and 'Y forward' in the export dialog.

D.3.3 Camera Calibration

The camera calibration should include the extrinsics for the IMU. The software expects the Kalibr format:

Listing D.1: Camera calibration file example

```
cam0:
  T_cam_imu:
  - [ 0.000965, 0.999790, -0.020456,  0.002266]
  - [-0.999987, 0.000866, -0.004851, -0.002965]
  - [-0.004832, 0.020460,  0.999778, -0.024131]
  - [ 0.0, 0.0, 0.0, 1.0]
  cam_overlaps: []
  camera_model: pinhole
  distortion_coeffs: [-0.084093, 0.053358, -0.000655, -0.000167]
  distortion_model: radtan
  intrinsics: [196.718542, 196.688981, 161.805861, 143.854077]
  resolution: [346, 260]
  rostopic: /dvs/image_raw
```

'`camera_model`' must be '`pinhole`' and '`distortion_model`' must be '`radtan`'

D.3.4 Hand-Eye

Hand-Eye calibration for the ground truth (optional). This transforms poses from the motion capture system to the camera frame. Example:

Listing D.2: Hand-eye calibration file example

```
calibration:
  translation:
```

```
    x: 0.016613688563580
    y: 0.008281589803800
    z: -0.045595510429855
  rotation:
    w: -0.532291887844937
    x: 0.665903167399887
    y: -0.422674269000762
    z: 0.307546386917393
```

D.4 Configuration

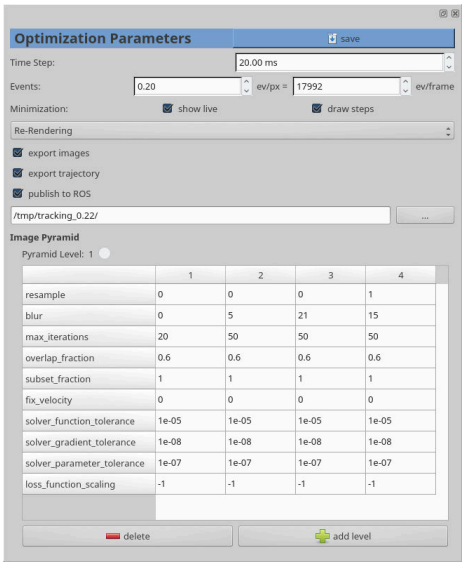


Figure D.2: Settings window showing additional configurable parameters. Can be opened in main window with the settings button on the top right.

Many of the settings are set trough a configuration file. See 'cfg/main.yaml' for an example and explanation of the parameters. Most values are required.

Some settings can also or only be changed in the GUI. Those changes are not saved.

D.5 Usage

D.5.1 Overview

After starting the application you probably want to load a keyframe and an eventframe, as otherwise some functions will crash, as they assume these exist.

To load these click on load events in the toolbar and either load keyframe from there as well or use generate KF from the error plot part on the left if there are no keyframes in the rosbag.

D.5.2 Tool Bar

At the top of the window there are various commands for loading data from the rosbag and to start tracking. It also has a slider to select a timestamp in the rosbag from which data should be loaded.

D.5.3 Poses

There are three poses that are used throughout the GUI:

Error Plot / Optimization (T_{WC} selected)
The initial state for the optimization and center state for all the plots. Can be set by clicking on the 2D error plot.

Hover State (T_{WC} hover)
Various plots allow hovering to see the warped images at the pose where the user hovers on the plot. This pose is hidden by default.

Keyframe (T_{WK}) for pose of the current keyframe
Loaded from the rosbag ground trough data when using load keyframe or copied from selected pose when using generate keyframe. This pose is hidden by default.

All pose tables can also be directly edited by double clicking on a value, can be copy/pasted or even stored/loaded from disk by using the right-click menu (this also allows to set the rosbag time slider).

Pose tables can be hidden or shown by clicking on the right of the title.

D.5.4 Error Plot

A two-dimensional plot of the error surface can be generated by clicking on the big plot button. The X- and Y-axes of the plot can be configured with the drop-down menus and the input fields on the left of the plot button. The fidelity of the graph is set trough the number of samples field at the bottom left.

The error plot is always plotted with the selected pose at the center and disturbing two parameters by the given amount.

Clicking on the plot will select that pose.

D.5.5 General Hints

- All images can be saved to disk by right-clicking on them.

- Various elements should have tool-tips that offer a short explanation when hovered over the element.
- Make sure correct Hand-Eye calibration for poses is configured. Remember to change this if you change the model (for example, when loading a simulated environment you don't want to load the hand-eye transformation used for a real model).

Bibliography

[1] A. Agarwal, K. Mierle, and Others. Ceres solver. `http://ceres-solver.org`.

[2] M. Bloesch, M. Burri, S. Omari, M. Hutter, and R. Siegwart. Iterated extended kalman filter based visual-inertial odometry using direct photometric feedback. *Int. J. Robot. Research*, 36(10):1053–1072, 2017.

[3] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*. Springer Tracts in Advanced Robotics. Springer, 2011.

[4] Marcin Dymczyk, Elena Stumm, Juan Nieto, Roland Siegwart, and Igor Gilitschenski. Will it last? learning stable features for long-term visual localization. In *3D Vision (3DV)*, pages 572–581, 2016.

[5] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. SVO: Fast semi-direct monocular visual odometry. In *IEEE Int. Conf. Robot. Autom. (ICRA)*, pages 15–22, 2014.

[6] Guillermo Gallego, Jon E. A. Lund, Elias Mueggler, Henri Rebecq, Tobi Delbruck, and Davide Scaramuzza. Event-based, 6-DOF camera tracking from photometric depth maps. *IEEE Trans. Pattern Anal. Machine Intell.*, 2017.

[7] Daniel Gehrig, Henri Rebecq, Guillermo Gallego, and Davide Scaramuzza. Asynchronous, photometric feature tracking using events and frames. In *Eur. Conf. Comput. Vis. (ECCV)*, 2018.

[8] Hanme Kim, Ankur Handa, Ryad Benosman, Sio-Hoi Ieng, and Andrew J. Davison. Simultaneous mosaicing and tracking with an event camera. In *British Machine Vis. Conf. (BMVC)*, 2014.

[9] Hanme Kim, Stefan Leutenegger, and Andrew J. Davison. Real-time 3D reconstruction and 6-DoF tracking with an event camera. In *Eur. Conf. Comput. Vis. (ECCV)*, pages 349–364, 2016.

[10] Simon Lynen, Torsten Sattler, Michael Bosse, Joel Hesch, Marc Pollefeys, and Roland Siegwart. Get out of my lab: Large-scale, real-time visual-inertial localization. In *Robotics: Science and Systems (RSS)*, July 2015.

[11] M. Meilland, T. Drummond., and A.I. Comport. A unified rolling shutter and motion blur model for dense 3D visual tracking. In *Int. Conf. Comput. Vis. (ICCV)*, December 2013.

[12] Elias Mueggler, Henri Rebecq, Guillermo Gallego, Tobi Delbruck, and Davide Scaramuzza. The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and SLAM. *Int. J. Robot. Research*, 36:142–149, 2017.

[13] H. Oleynikova, Z. Taylor, M. Fehr, R. Siegwart, and J. Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In *IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS)*, 2017.

[14] Henri Rebecq, Guillermo Gallego, Elias Mueggler, and Davide Scaramuzza. EMVS: Event-based multi-view stereo—3D reconstruction with an event camera in real-time. *Int. J. Comput. Vis.*, pages 1–21, November 2017.

[15] Henri Rebecq, Guillermo Gallego, and Davide Scaramuzza. EMVS: Event-based multi-view stereo. In *British Machine Vis. Conf. (BMVC)*, September 2016.

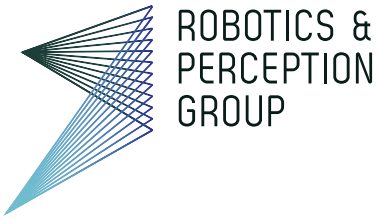
[16] Henri Rebecq, Timo Horstschaefer, and Davide Scaramuzza. Real-time visual-inertial odometry for event cameras using keyframe-based nonlinear optimization. In *British Machine Vis. Conf. (BMVC)*, September 2017.

[17] Henri Rebecq, Timo Horstschäfer, Guillermo Gallego, and Davide Scaramuzza. EVO: A geometric approach to event-based 6-DOF parallel tracking and mapping in real-time. *IEEE Robot. Autom. Lett.*, 2:593–600, 2017.

[18] T. Schneider, M. T. Dymczyk, M. Fehr, K. Egger, S. Lynen, I. Gilitschenski, and R. Siegwart. maplab: An open framework for research in visual-inertial mapping and localization. *IEEE Robot. Autom. Lett.*, 2018.

[19] T. Whelan, S. Leutenegger, R. F. Salas-Moreno, B. Glocker, and A. J. Davison. ElasticFusion: Dense SLAM without a pose graph. In *Robotics: Science and Systems (RSS)*, Rome, Italy, July 2015.

[20] T. Whelan, R. F. Salas-Moreno, B. Glocker, A. J. Davison, and S. Leutenegger. ElasticFusion: Real-time dense SLAM and light source estimation. *Int. J. Robot. Research*, 35(14):1697–1716, 2016.



Title of work:

Event-based, Direct Camera Tracking from a Photometric Map using Nonlinear Optimization

Thesis type and date:

Master Thesis, August 2018

Supervision:

Henri Rebecq
Dr. Guillermo Gallego
Mathias Gehrig
Prof. Dr. Davide Scaramuzza

Student:

Name:	Samuel Bryner
E-mail:	samuelbryner@student.ethz.ch
Legi-Nr.:	010-915-023

Statement regarding plagiarism:

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:

`http://www.lehre.uzh.ch/plagiate/20110314_LK_Plagiarism.pdf`

Zurich, 16. 8. 2018: _____