DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

3D Environment Reconstruction Based on an Externally Tracked Camera in Virtual Reality

Lukas Bonauer

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

3D Environment Reconstruction Based on an Externally Tracked Camera in Virtual Reality

3D-Rekonstruktion der Umgebung anhand einer extern getrackten Kamera in Virtual Reality

Author:	Lukas Bonauer
Supervisor:	Prof. Gudrun Klinker
Advisor:	Sandro Weber
Submission Date:	2017-09-15

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 2017-09-15 Lukas Bonauer

Acknowledgments

I want to thank my advisor Sandro Weber for pointing me in the right direction when needed and for answering all my questions and Frieder Pankratz for introducing me to the wonders of Ubitrack.

Abstract

Room-scale virtual reality has arrived on the market and with it the size of somebody's room has become a limiting factor for immersive experiences. Rectangular boundaries displayed in VR roughly indicate where the available area ends, but they cannot account for smaller obstacles or more complex room geometry.

In this thesis, the potential use of the integrated camera of a current VR headset for online 3D reconstruction is investigated in order to allow a more accurate display of room boundaries. The feature-based SLAM algorithms PTAM and ATAM are adjusted to incorporate data from the SteamVR tracking system and the resulting sparse point clouds are visualized in a virtual scene. The results show that the quality of the reconstructions of both algorithms still suffer from considerable noise and require further tweaks and optimizations in order to be usable. Surprisingly, it is found that while the added information about the camera poses does contribute to a more globally consistent result, it seems to increase the noise in the point clouds. Overall, the approach provides a decent foundation for future work.

Contents

1 Introduction

Virtual reality is on the rise again, with consumer hardware ultimately becoming powerful enough to provide a satisfactory user experience through affordable head-mounted displays. With room-scale positional tracking, walking around in your living room while immersed in a virtual game world is no longer just a dream of the future. But the available space is limited and as a result, the real world still plays a necessary role in the experience. VR systems typically allow the user to define a rectangular boundary of the available space, which is displayed as a warning in the virtual scene.

But what if the real world could be modeled more accurately than that? If a detailed representation of the surrounding geometry were available, it would open up a range of new possibilities: The virtual warning boundary could closely fit obstacles and walls in the room, thus allowing users to make better use of the available space, for example empty spots above low furniture, with decreased risk of hitting something while immersed. Furthermore, an application could display the full 3D model of the room in the virtual scene and overlay it with virtual content, enabling many of the possibilities of augmented reality, while avoiding the technological complications of see-through head-mounted displays and having the user fully immersed in a controlled environment instead.

In this thesis, the vast research on 3D reconstruction is combined with current virtual reality systems to work towards an implementation of this idea. The front-facing camera of the HTC Vive is used in combination with its room-scale tracking system to reconstruct 3D information about the environment in real time.

1.1 Problem statement

The goal of this project is to create a framework that is able to automatically construct a detailed and textured 3D model of a VR user's environment in real time, using only

the hardware capabilities of an off-the-shelf HTC Vive headset. This framework could then be used in a utility service, enhancing the coarse Chaperone bounds displayed by SteamVR today, or as part of a full VR/AR application that can augment the reconstructed model with additional content and interactivity.

For the scope of this thesis, this ambitious goal is narrowed down to a framework capable of creating a real-time *sparse* reconstruction and rendering it as a point cloud into a VR scene, true to scale and properly aligned with the real world. Further converting the point cloud into a 3D model and texturing it is not covered here, but is left open as a future research topic.

This particular set-up is unusual and rarely studied in literature: Real-time algorithms usually assume unknown camera poses and try to estimate them at the same time as they are reconstructing the environment (SLAM). This may be because a tracked camera as the by-product of a virtual reality consumer device is a very recent innovation.

1.2 Contributions

It was decided to build the solution on top of existing algorithms in order not to reinvent the wheel. Parallel Tracking and Mapping (PTAM) by Klein and Murray [8] and Abecedary Tracking and Mapping (ATAM) by Uchiyama et al. [23] were chosen because they are open-source, relatively easy to understand and match the requirements of the problem. They serve as two alternative implementations and are later compared.

As its middleware, the project uses the Ubitrack framework [13]. The main code contribution is the development of several Ubitrack components:

- a device driver that obtains tracking poses and camera images from an OpenVR-enabled device such as the HTC Vive
- modification of PTAM to make use of known camera poses
- integration of ATAM into Ubitrack and equivalent modification to use known camera poses

1.3 Overview

The thesis is structured as follows: Chapter 2 explains important background knowledge used throughout the thesis and puts the presented solution into the global context of 3D reconstruction. Chapter 3 then gives a brief overview of related work on different categories of 3D reconstruction. It is followed by the main approach in chapter 4, which describes the developed software components and their integration into a VR scene in detail. The quality of the results and their suitability for the presented use cases are discussed in chapter 5, while chapters 6 and 7 give an outlook on the work left to do and possible developments in the future.

2 Background

This chapter explains necessary background knowledge about the topics covered in this thesis.

2.1 Camera model

An essential mathematical aspect of vision-based algorithms is their representation of the camera. Its model is usually split into its internal parameters (*intrinsics*) and its external parameters (*extrinsics*).

For the camera intrinsics, there exist many complex and generic representations that account for distortion as well as algorithms that can directly deal with them [11]. But in this context, a simple *pinhole model* is sufficient, as it simplifies the calculations and because it is possible to undistort captured images before processing them.

With a pinhole camera, the transformation from world coordinates to the image plane is linear and can be described by the 3×3 matrix

$$K = \begin{pmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{pmatrix}$$

where f_x and f_y denote the focal length (expressed in pixels in the respective dimension) and p_x and p_y the center of projection (also in pixels). K is usually called the *camera calibration matrix* [5]. The 3D reconstruction algorithms used here assume that this matrix is fixed over time and known, that is, the camera is assumed to be calibrated.

The camera extrinsics – here referred to as the *camera pose* – describe the camera's location and orientation in the world. They consist of a rotation matrix R and a translation matrix T (when using homogeneous coordinates $(x, y, z, 1)^T$), which together

describe the transformation from world space to a camera-centered coordinate space [9]. Note that T denotes the location of the world origin with respect to the camera, not vice versa.

2.2 3D reconstruction

The extraction of 3D information from different types of sensor data is a very diverse topic. A practical distinction is between *active* and *passive* methods, where the former use sensors that illuminate the scene in a controlled fashion (for example with a laser or projector), often modulated over time or space, while the latter solely rely on passive sensors, most commonly cameras [9].

Another distinction can be made between the number of vantage points involved. It is possible to infer depth information from a single vantage point – for example by looking at textures, occlusion, the defocus caused by camera lenses or the time-of-flight of a laser beam. More commonly, two or more vantage points are used: a camera paired with a projector (structured light [18]), a stereo camera rig, a moving camera, etc.

Depending on the goals and available hardware, the 3D reconstruction problem can be solved with many different algorithms, some of which are laid out in chapter 3.

The focus of this thesis lies on one particular passive multi-vantage-point method: *structure from motion*. For this method, images from a monoscopic moving camera are used as the foundation. More specifically, we look at *incremental* and *real-time* structure from motion, where the reconstruction is extended and improved while the camera is still active.

Furthermore, we focus on *feature-based* structure from motion, where the importance lies on high contrast areas of the image – salient edges and corners. Since such features are usually not present on low-textured surfaces such as single colored walls, feature-based reconstructions result in *sparse point clouds*, with 3D points only where enough feature points could be detected. In contrast, *dense* methods make use of all available pixel data and produce much more detailed point clouds and full depth maps, at the cost of more computation power and complexity. In order to be feasible for real-time reconstruction, they need to make heavy use of parallelization [12].

Common tasks in feature-based structure from motion include:

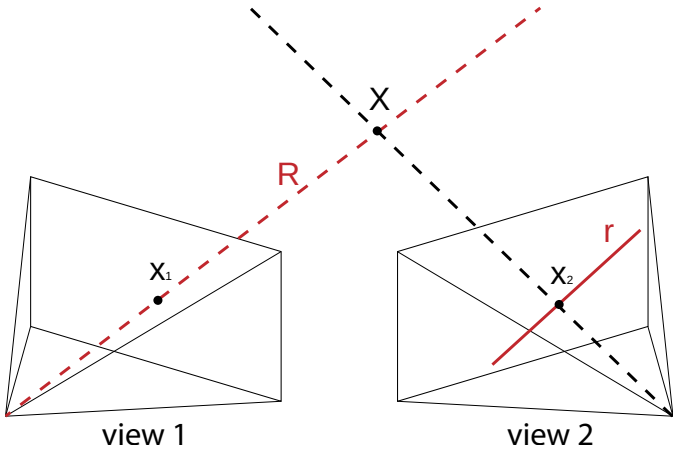

Figure 1: Epipolar relationships between two views. Two measurements x_1 and x_2 of the 3D point X. The ray R through x_1 projected onto the image plane of view 2 is the epipolar line r, which intersects x_2.

- **Feature extraction** From a given image, find a set of the most salient features and return their projected points. A very popular algorithm for this is FAST [17].

- **Correspondence / Feature matching** Given two views and a set of projected points for each view, find pairs of points that represent (approximately) the same location in the world.

  This task is facilitated if the transformation between the two views is known: By taking the first view and constructing a 3D ray from the camera origin through a point on the image plane, this ray will correspond to a line on the image plane of the second view, known as the *epipolar line* (see figure 1). If the spatial relationship between the two image planes is known, the location of the epipolar line can be calculated. Since a potential feature match has to lie somewhere on that line, this reduces the possibilities to a one-dimensional search space. As a result, both computational complexity and the amount of false positives is reduced, since similar looking features at different spots are less likely to be misidentified as matches.

- **Triangulation** Given the 2D projections of a 3D point in two views with known poses, find the 3D coordinates of the point. This is equivalent to finding the

intersection between the rays from the two camera origins through the respective projected points. However, under the influence of noise, those rays generally do not intersect, so it is necessary to find the most probable point of intersection [6].

The distance between two views used for correspondence and triangulation is called the **baseline** and has a significant impact on the quality of reconstruction. If the baseline is too small, the rays are almost parallel and the noisy intersection has a large margin of error [9]. But if the baseline is too large, the quality of feature matching is impaired, as fewer points may be visible in both views and their visual disparity increases when viewed from more different angles.

When a video camera is used for feature-based structure from motion, performing triangulation every frame is impractical. Not only due to performance concerns, but also because the baseline is too small for triangulation. This is why algorithms have to decide which frame to promote to a **keyframe** first and only perform triangulation between keyframes, thus ensuring a good baseline.

The additional data between keyframes can be used to assist feature matching: due to the small timestep, the relative movement of feature points on the image plane is minimal, so they can be matched more easily by a technique called **trail tracking**.

Note that the most common use case of real-time structure from motion is visual SLAM (simultaneous localization and mapping), where 3D reconstruction is coupled with an estimation of the camera pose. In our use case – because of the use of the SteamVR tracking system – the camera pose is known with high accuracy, thus (in theory) reducing the problem to mapping only.

2.3 OpenVR and SteamVR

OpenVR is an API developed by Valve in order to allow applications to interface with virtual reality hardware without directly programming for a specific hardware vendor [24].

At the time of writing, Valve's SteamVR is the only known runtime that implements OpenVR and the HTC Vive is the only OpenVR-enabled hardware that is also equipped with a camera.

SteamVR tracking (also called *Lighthouse tracking*) uses two stationary base stations, which are installed in opposite corners of the tracking space. They send out beams of infrared light in a 120° cone with an effective range of about five meters. Each tracked object – in a standard setup the head-mounted display and two controllers – contains up to 32 infrared sensors spread out across its surface, which detect the beams of the base stations.

Tracking is achieved by the base stations sending out precisely timed signals: First, a pulse is sent out in all directions, synchronizing all sensors in the room. Next, each base station sweeps its beam across the room, first horizontally then vertically. The sensors measure the delay between when they were hit by the pulse and the sweep and send this data to the host (the computer running SteamVR), which calculates the angles between the base station and the sensors and from this the position and orientation of each object relative to the base station.

While the information from a single base station is enough to calculate the pose, sensors need to have a line of sight to the station, so using two stations makes tracking more robust to occlusion, for example caused by the human who wears the VR headset standing in the way. Additionally, the accuracy of the pose is increased if a device is visible to both base stations.

Note that the base stations do not need to be connected to the host or the tracked devices, but they need to be placed in view of each other so they can synchronize themselves and automatically calculate their placement relative to each other.

By combining the data from the infrared sensors with inertial measurement units (IMUs) on the devices, SteamVR achieves consistent tracking with submillimeter accuracy at an update rate of 1000 Hz [25].

The camera used in this project is part of the head-mounted display of the HTC Vive. Its out-of-the-box usage at the time of writing is somewhat limited: If enabled, its video is displayed in VR in a tiny window next to one of the hand controllers in the settings menu of SteamVR. It is also used for the “room view” mode, which overlays a stylized version of the camera image in the virtual world, as shown in figure 2. This allows some rough interaction with the real world without taking off the headset. Because the camera is angled slightly downwards on the headset, the overlay does not fill the top section of the user’s field of view. The overlay can also be configured to only be

visible when the user comes close to the edge of the play area (as part of the *Chaperone* system), but due to the monoscopic nature of the camera, the projection appears flat rather than as actual geometry, so knowing when exactly you will hit a small obstacle is difficult. Also, since the camera is several centimeters away from the user's eyes, the view is not perfectly aligned with what the user would see with their own eyes.

The camera is exposed to VR applications through OpenVR's *tracked camera* interface, which allows access to its video stream as well as to its pose, as determined by the SteamVR tracking system. The transformation from the pose of the whole head-mounted display to the pose of the camera is already applied by the runtime, presumably by applying a pre-programmed offset and rotation.

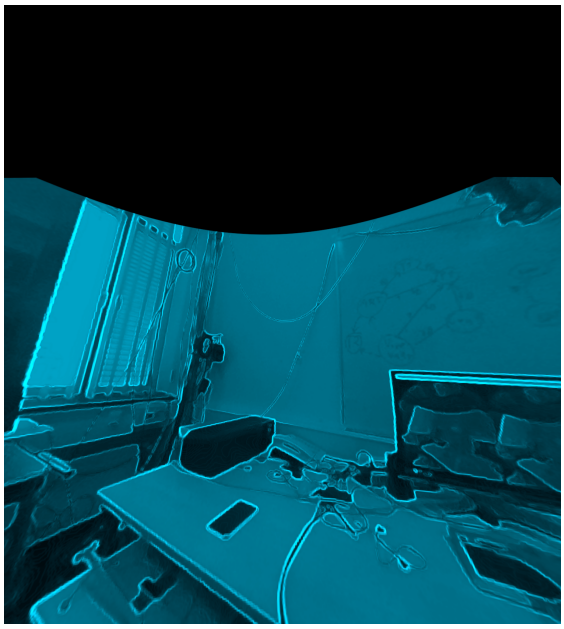

Figure 2: Room view overlay of SteamVR

2.4 Ubitrack

Ubitrack is a framework developed by the Forschungsgruppe Augmented Reality at the Technical University of Munich and heavily used in all components of this project. It is designed as a generic middleware for dynamic and heterogeneous tracking environments and its goal is to provide an optimal estimate of geometric relationships between arbitrary objects [13].

Ubitrack has many features which are not relevant to this project, such as the automatic deduction of spatial relationships at runtime or the distribution of tracking via multiple networked devices. This section only includes a brief summary of the concepts relevant to the approach, for more details please refer to the respective literature on Ubitrack [13, 15, 16].

A scene in Ubitrack is defined in a **spatial relationship graph** (SRG). It specifies the arrangement of devices and objects in the scene with respect to each other. Formally,

nodes represent coordinate systems and edges represent transformations between them.

These transformations are estimated by **measurements**, which can come from various sources: They may come from an actual sensor, be read from a file or be inferred by an algorithm. An edge also has various properties, like the data type of its measurements, for example a full 6 DoF pose (3D position and rotation) or a 3×3 intrinsics matrix. The data may also contain information about the measurement uncertainty (like a covariance matrix for a pose), allowing the system to propagate information about the accuracy of values through the graph. Some edges are also considered static: their values are fixed over time, for example the transformation between rigidly attached objects or the camera intrinsics [7, 13].

Measurements are taken at discrete points in time, so they need to include a timestamp. Since sensors usually generate measurements asynchronously, great care must be taken when combining measurements from multiple sources. If differences in the timestamps of measurements are not accounted for, tracking accuracy is greatly reduced [15]. Hence, edges are also marked with their synchronization mode: A **push** edge generates an event every time a measurement is made and sends it to interested consumers. Conversely, a **pull** edge provides a time-continuous function that can be queried for measurement values at any given time. In practice, this works either if the value is constant or if a *push-pull-conversion* has been applied – for example by interpolating or extrapolating sensor data.

While the formal model asserts edges as transformations between coordinate systems, in practice there is some leeway in what they may represent: Data types also include raw image data and the button type – a scalar value indicating that some command has been sent. A "button" edge is usually necessary when a tracking scenario requires user interaction and – by convention – connects the abstract nodes "Event" and "Event Space", which do not represent real objects, but are necessary for button events to fit into the model of the framework.

The actual computations in Ubitrack are performed in **components**, which are decoupled algorithms that implement a specific subtask, for example sensor drivers, vision algorithms, camera calibration or debug output. A component defines **patterns**, which are subgraphs of an SRG. A pattern represents the signature of an algorithm: It defines its inputs and outputs, and also places constraints on the geometric relationship

between them [16]. When visualizing a pattern, **input edges** are drawn with a dashed line, **output edges** with a solid line[1].

Figure 3 shows one of the simplest possible patterns: the inversion of a pose. It takes a transformation from coordinate system *A* to *B* through its input edge *Input Pose*, which is of type `POSE_6D` and internally represented by a 4x4 transformation matrix. The software component behind the pattern inverts the matrix, which then represents a transformation from *B* to *A*, and provides the result through its output edge *Inverted Pose*. Both edges have the synchronization mode `AUTO`, which means the pattern can be used on both a push or a pull edge.

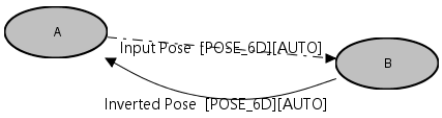

Figure 3: Example of a pattern: Inversion

Finally, a **data flow network** (DFN) connects multiple patterns in order to form a full SRG. Each input edge needs to be connected to an output edge with compatible data type and synchronization mode, so the underlying component knows where it can get its input data from. Usually, a DFN contains at least one pattern with only output edges (the sensor data) and at least one pattern with only input edges (the consumer processing the data that the application is interested in). The DFN is what Ubitrack uses at runtime to propagate data between the components and to answer queries by the application.

[1] This corresponds to the convention used by the graphical tool *trackman*. In most literature about Ubitrack, the visualization is actually converse: input edges are drawn with solid lines, output edges with dashed lines. In order to be consistent with the included screenshots, *trackman*'s version is used in this thesis.

3 Related work

3D reconstruction based on imagery has been studied for decades. There is no consensus on the naming in literature: *structure from motion*, *multi-view geometry* and *multi-view stereo* often appear while describing the same thing, while some differentiate the terms and see one as a subset of another.

More importantly, the most glaring distinction between methods is whether they are designed for real-time use or allow long-running calculations. The former usually work incrementally, processing new images as they come in, while the latter receive a fixed set of images to work with.

For long-running structure from motion, there exist ready-to-use tools such as openMVG [10] or COLMAP [19, 20].

Real-time algorithms that work on video input most commonly solve the SLAM problem – estimating both the camera position and a map of the world at the same time. Feature-based examples include MonoSLAM [4], PTAM [8] and ATAM [23]. Other approaches part ways with the reliance on feature points and calculate depth information for every pixel, such as DTAM [12] and REMODE [14].

Using the tracked camera of the HTC Vive results in an unusual set-up: It combines real-time mapping with a fully calibrated (internally and externally) moving camera. To the author's knowledge, no algorithms with published implementations exist that fit this scenario (while Carceroni et al. [3] use GPS data as priors for camera locations, these do neither include orientations nor is the presented algorithm designed to work in real time). Since SLAM is the problem most closely related to the goal of this thesis, it builds on two established algorithms from that field – PTAM and ATAM – and augments their calculations with reference poses from the external tracking system.

4 Approach

In the scope of this thesis, an application was implemented that can read camera data from the HTC Vive and run either PTAM or ATAM, while displaying their resulting point clouds in a VR scene. This pipeline can be broken down into several decoupled components.

Section 4.1 describes the acquisition of camera images and reference poses from the VR headset through OpenVR and its integration into Ubitrack. It also describes the various options for camera calibration and image undistortion in the context of OpenVR's API.

After an undistorted image and its pose have been acquired, the 3D reconstruction algorithms come into play. Section 4.2 explains the original PTAM algorithm and then focuses on the modifications that were made so it can be used in combination with the reference pose. Section 4.3 does the same with the modifications of ATAM, and also highlights key differences to PTAM. Both algorithms can be used interchangeably as they operate on the same kind of data and they both provide a sparse point cloud as output.

Finally, section 4.4 examines the integration of the algorithms into an interactive VR scene and the visualization of the point clouds obtained from PTAM and ATAM.

4.1 OpenVR for Ubitrack

The first step is to interface with the OpenVR API to obtain the camera images and tracking information from the HTC Vive. This has been achieved by implementing a Ubitrack component to expose the data to the SRG. While the API provides a variety of tracking data, only access to the relevant part has been implemented in the scope of this thesis: the tracked camera. The camera of the HTC Vive has an unusual resolution of 612x460 pixels and runs at 60 Hz.

4.1.1 Initialization

OpenVR is designed to allow multiple applications to access the HMD at the same time. It allows each process to independently connect to and disconnect from the OpenVR runtime. Upon initialization, the application can specify one of the following application types:

- "`VRApplication_Scene` – A 3D application that will be drawing an environment.
- `VRApplication_Overlay` – An application that only interacts with overlays or the dashboard.
- `VRApplication_Background` – The application will not start SteamVR. If it is not already running the call with VR_Init will fail [...].
- `VRApplication_Utility` – The application will start up even if no hardware is present. [...] This application type is appropriate for things like installers." [24]

Only one application of type *Scene* can be active at the same time. For the purpose of 3D reconstruction, the *Background* mode is sufficient (if SteamVR can be assumed to be already running when the process is started). For other use cases, the application type can be set as a parameter on the component.

Note that the initialization is scoped to the entire process, so if Ubitrack is embedded into a VR-enabled application (as exercised in section 4.4), the component has to skip both initialization and shutdown, since it would otherwise interfere with the hosting application.

4.1.2 Polling rate

OpenVR currently only exposes a polling interface to the camera, so the images have to be polled at regular intervals. Valve recommends polling approximately every 16 ms[1]. However, when the polling rate is merely equal to the camera framerate, it has proven to be difficult to synchronize them reliably. This results in missed frames and irregular delays between new measurements. This problem can be mitigated by making use of

[1] see `https://github.com/ValveSoftware/openvr/blob/5d0574bf6473130d25dd296ad30206ccd148590b/headers/openvr.h#L3231` (accessed 2017-08-17)

the frame sequence number that is provided in the frame header of each API calls. This enables the polling thread to only copy the new image data once the sequence number has actually increased, thus it can poll at a much higher frequency with minimal additional CPU cost, reducing the likelihood of skipping frames and the overall latency between the time a new camera frame is available and the time it is pushed to the output port.

In the Ubitrack component, the polling rate can be set as a parameter. Latency-critical applications can set it to a high frequency in order to get measurements as early as possible and also to ensure no frames are skipped. In contrast, applications that do not need to process the full 60 frames per second can set it to even lower values, intentionally skipping every nth frame while saving processing power.

4.1.3 Camera calibration

Many computer vision algorithms – including the ones used later for 3D reconstruction – need knowledge of the camera intrinsics as well as images that are free from distortion. OpenVR provides some functionality to facilitate this, but Ubitrack also has existing camera calibration components that can be used instead.

Images can be requested from OpenVR in one of two modes: distorted or undistorted. The distorted mode uses the native image of the camera – where the distortion is caused by the fish-eye lens. In undistorted mode, the image is internally transformed with some unknown parameters and only then returned to the caller.

Additionally, the API can be queried for the *focal length* and the *center of projection* (but only in undistorted mode), which is sufficient to build the intrinsic matrix. Hence, the Vive camera can be fully calibrated without any additional steps if you are willing to rely on the values used by OpenVR.

Alternatively, undistortion and intrinsics can also be done with the help of Ubitrack. This requires a camera calibration step, where images of a calibration grid from multiple angles are used to calculate the distortion parameters as well as the focal length and center of projection. After calibration, the parameters are stored in a *camera model file*, which can later be used in the actual application.

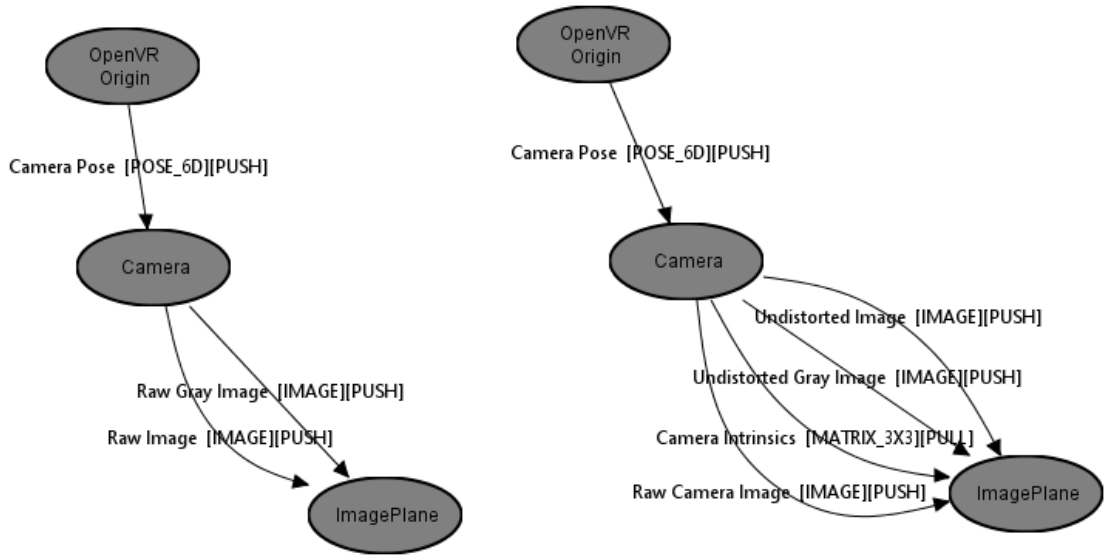

Figure 4: SRG patterns of the OpenVR component. On the left: *uncalibrated*, right: *file calibrated* or *software calibrated*.

The Ubitrack component leaves the choice of calibration to the user, providing three different versions of the pattern:

- **uncalibrated**: This pattern only outputs the raw image data obtained from OpenVR. It can be configured to use either distorted or undistorted mode. It is intended to be used in the camera calibration SRG.
- **file calibrated**: This pattern takes a *camera model file* generated in the calibration step as input, undistorts the image based on the model file and outputs that and the intrinsic matrix.
- **software calibrated**: This pattern uses the undistorted images as well as the intrinsic matrix provided by OpenVR and outputs both.

The impact of the calibration mode on the spatial relationship graph of the component is visualized in figure 4.

4.1.4 Image data

Images are read in RGBA mode from OpenVR with 8 bits per channel. Note that the alpha channel is unused in all modes described here.

In all versions of the pattern, the unmodified data is pushed to the *Raw Image* port (see figure 4). In the calibrated version, the undistorted image is also pushed to the *Undistorted Image* port and converted to grayscale (*Undistorted Gray Image* port).

4.1.5 Pose data

Each camera frame is augmented with tracking information about its pose, consisting of a 3x4 transformation matrix and a vector for the camera's linear and angular velocity. OpenVR provides these in the frame header struct and already applies the transformation from the headset's origin to the camera's position on the headset. The pose data is already in sync with the camera frames and is pushed together with the image data.

In the Ubitrack component, only the transformation matrix of the pose is currently provided as an output port (the *Camera Pose* edge in figure 4). While the velocity could theoretically be used later for further processing, this is not done in the current implementation. If no pose is available for a frame (because the trackers are obstructed or out of range of the base stations), the frame is skipped entirely.

4.2 Adaptation of PTAM

The original Parallel Tracking and Mapping (PTAM) algorithm by Klein and Murray [8] is designed for simultaneous localization and mapping (SLAM), so it works without external knowledge of the camera pose. It takes a video stream and incrementally estimates the camera pose (tracking) and a 3D map of the world (mapping) relative to each other in real-time. The tracking works at framerate, while the mapping runs on a separate thread, allowing more computationally expensive tasks such as bundle adjustment to be executed to improve accuracy.

For this project, only the mapping aspect of PTAM is relevant for the 3D reconstruction since the camera poses are already known from the OpenVR tracking system. PTAM's tracking component is thus redundant in this scenario. However, because

tracking and mapping are closely interconnected, it is beneficial to understand both of them before looking at the necessary adaptations for this use case.

Note that before the start of this project, the original implementation of PTAM had already been modified by the Forschungsgruppe Augmented Reality at TUM, albeit with a different purpose. It was already converted to a Ubitrack component and some integration of a reference pose had been started. For this thesis, several changes have been made, including better handling of different frame sizes and a full use of the reference pose.

4.2.1 Overview over PTAM

In this section, the original PTAM algorithm is briefly described. Details that are not relevant for further discussion have been left out or simplified. For a full description, please refer to Klein's and Murray's original paper [8].

PTAM works with undistorted grayscale images and assumes that the camera intrinsics are known.

A core idea behind PTAM is to split tracking and mapping into two separate threads. This allows tracking to be performed continuously at framerate, while the mapping thread can run much more computationally expensive tasks in parallel. The mapping thread runs continuously in the background and prioritizes its work based on what new data it has to work with:

1. if a new keyframe is available, integrate it into the map
2. run local bundle adjustment
3. run global bundle adjustment
4. find new map points in old keyframes and reevaluate outliers

Initialization

To bootstrap both tracking and mapping, an initial map has to be acquired in a special initialization step: The user presses a key to indicate they want to start tracking (first keyframe). Then they translate the camera sideways and press a key again once the baseline is wide enough (second keyframe).

In the meantime, PTAM tracks the trails of salient feature points (FAST corners [17]) and uses their locations in the first and second keyframe to get matching stereo pairs for the five-point algorithm [21], which gives an initial estimate of the camera movement between the first and second keyframe.

Using those two pose estimates, the first map points are inserted by triangulating the stereo pairs in the two keyframes. Next, to get a good initial map, bundle adjustment is run to refine the map.

Lastly, PTAM attempts to find a dominant plane (for example the surface of a table) in the points and aligns the map's coordinate system with it.

Tracking

Incoming images are first decomposed into four pyramid levels, each with half the width and height of its predecessor. This is done primarily so points viewed from different distances can be matched better by finding their pixels in a different pyramid level. For example, after moving closer to an object until it appears twice as large in the image, it now occupies roughly the same amount of pixels in pyramid level 1 as it did before the movement in pyramid level 0 – because at pyramid level 1 the image is scaled down by a factor of two.

For each pyramid level, FAST corner detection [17] is used to find feature points in the image. These feature points are then used to find existing map points in the current frame (referred to as *patch search* in PTAM) – the essential step to obtain observations of a world point from multiple viewpoints.

For this to happen, each map point is first projected into the camera coordinate system of the current frame (based on the camera intrinsics and a prior estimate of the camera's current pose). Next, a patch template is generated – a small 8x8 pixel region of the image in which the map point was first discovered – and warped according to what the square patch would look like from the current perspective. The pyramid level which matches the size of the warped patch best is selected and all feature points of that level within a fixed radius of the projected map point are searched. The best match is selected (based on its zero-means sum of squared differences score) and used as a measurement, if its score passes a fixed threshold.

To improve accuracy and performance, some further refinements are made to the measurements, which are left out here for simplicity reasons.

Once all measurements have been made for the current frame, the new camera pose is estimated by minimizing the overall reprojection error of the points (answering the question "Which pose of the camera best explains the new image positions of the map points?").

Lastly, the tracker assesses the current tracking quality by looking at the percentage of successful measurements and uses that to decide if it is good enough to submit a new keyframe or bad enough to require relocalization.

If the tracking quality is good enough and the current pose is far enough away from the closest keyframe, the current frame is submitted to the mapping thread as a new keyframe.

If relocalization is required, PTAM attempts to recover by comparing a heavily downscaled and blurred version of the current frame to all other keyframes and selecting the most similar image. Then, it finds a rotation-only estimate of the transformation between the selected keyframe and the current frame by using the ESM algorithm [1].

Mapping

New keyframes Once a new keyframe has been submitted by the tracking thread, the already measured features are used as the foundation for new map points. PTAM only filters out features points that are far enough away from existing map points (on the image plane) and uses them as candidates for new map points.

To extract depth information from those candidates, it searches for corresponding features in the closest existing keyframe by performing *epipolar search*. If a match was found, it triangulates the 3D point and inserts it into the map.

Bundle adjustment When the mapping thread is not busy adding keyframes, it starts bundle adjustment to improve the accuracy of the map. Due to the inherent complexity of bundle adjustment, it first performs it locally – only considering the most recently added keyframe and the four keyframes closest to it, and only including the map points visible in those keyframes. Once that has converged, global bundle adjustment is considered, but it is aborted as soon as a new keyframe arrives.

Data association refinement At lowest priority, the mapper tries to find map points in older keyframes where they have not been attempted to be measured yet. This increases the number of viewpoints from which the points are visible and can thus lead to a better optimization in the next bundle adjustment run.

The mapper also reconsiders outlier decisions: Points that have previously been judged as "not fitting the data" are given a second chance and points that have not fit well into the map during bundle adjustment are marked as outliers.

4.2.2 Integration of the reference pose

The addition of the reference pose from OpenVR simplifies and changes some aspects of PTAM. First and foremost, the coordinate systems of the map and OpenVR have to be aligned, so the reference poses correspond to the keyframe poses. This requires several changes:

At the initialization stage, the 5-point-algorithm, which usually calculates the displacement of the camera for the initial two keyframes, is replaced with the already known values of the reference pose at the respective times. The world origin is thus no longer at the origin of the first keyframe, but identical to the origin of the reference poses.

The search for a dominant plane in the initial map and the alignment of the coordinate system with it is redundant now as well – after room setup, OpenVR's coordinate system is already aligned with the floor, so it can be removed.

PTAM's tracking system is not completely removed. While its main purpose – to determine the camera pose – is in theory unnecessary due to the more reliable and drift-free reference pose, the system still remains active so it can decide when a new keyframe should be added to the map and so it can update important data about the underway keyframe that the mapping thread uses (for example, the set of FAST corners).

Furthermore, in practice, the accuracy of the map may actually improve when PTAM's internal tracking is used despite the reference pose, as long as the tracker can measure enough points. This phenomenon is discussed in more detail in chapter 5.

As long as a reference pose is available, relocalization should never be necessary. Even when the tracker cannot find any existing points, the frame can still be eligible for

a new keyframe.

When the mapper runs bundle adjustment (no matter if local or global), it originally minimized errors by adjusting both the keyframe poses and the map points. However, this is now detrimental to the mapping result, as it breaks the alignment of the coordinate system with the reference pose. Bundle adjustment needs to be restricted so all keyframe poses remain fixed (only the map points are changed). Even in this mode, it still remains an important optimization, because the initial triangulation of a point only uses two viewpoints, but by the time bundle adjustment is executed, it may have also been observed by more recent keyframes or even found in older keyframes by the *data association refinement* process described in the previous section. These additional measurements can be leveraged by bundle adjustment for a more accurate and consistent result.

Note that because the reference pose is available even before initialization, a good point of time for the second keyframe could be determined automatically (when there is a sufficient baseline and enough tracked features for a high-quality initial triangulation) instead of requiring the user to press a key again. This has not been implemented.

4.2.3 Integration into Ubitrack

The spatial relationship graph of the Ubitrack component is shown in figure 5. At startup, the component is idle and waits for an event on the *Command Inputs* port, which starts the initialization stage of PTAM. All further user interaction is done through this port – primarily initialization completion and resetting the tracker.

Images are received through the *Image* push input port. They are assumed to be free of distortion and in grayscale format, analogous to the original PTAM algorithm. Each received measurement is directly processed by the tracker. It first pulls the corresponding measurement from the *Reference Pose* input port and then proceeds as described in the previous sections. Note that the *Camera Intrinsics* port is only read from once at startup, since the intrinsics are assumed to be constant over time.

These inputs are compatible with the OpenVR component described in section 4.1. Its *Undistorted Gray Image* and *Camera Intrinsics* outputs can be directly connected to the *Image* and *Camera Intrinsics* inputs of PTAM, respectively. OpenVR's *Camera Pose*

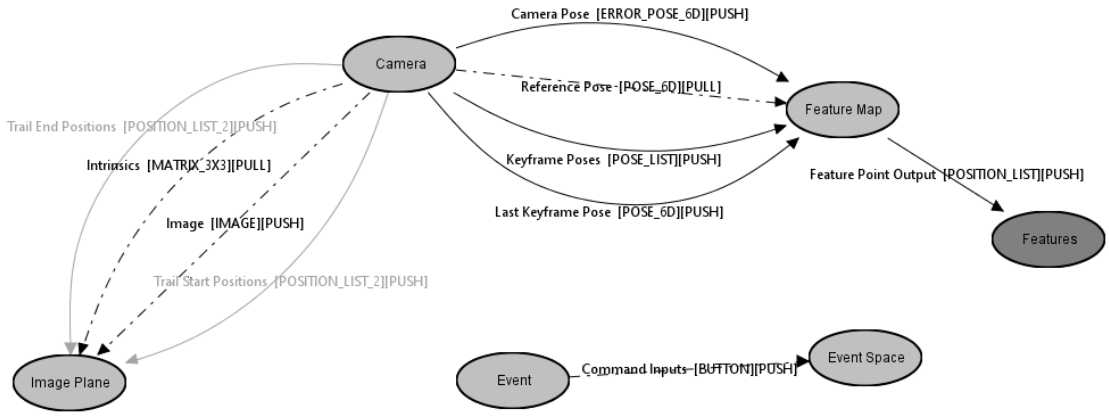

Figure 5: SRG pattern of the modified PTAM component

output has to be inverted and converted to a *pull* edge (using a buffer) before it can be used as the *Reference Pose* input.

While designed for use with OpenVR, the modified PTAM component is resolution and framerate agnostic and can also be used with any other setup that provides a camera with known intrinsics and known pose.

The output ports are also visualized in figure 5. The most important one is *Feature Points*, which is a sparse point cloud – a list of 3D positions – containing all the map points of the ongoing reconstruction. It is pushed every time the map is updated (when either a new keyframe is added or bundle adjustment has completed).

The *Camera Pose* output contains the internal state of the camera in PTAM. This is usually equal to the reference pose, but it differs if PTAM is configured not to use the reference pose for its frame-by-frame tracking, but only for initialization and in case its internal tracking is lost.

Information about the set of keyframes is provided through the *Keyframe Poses* and *Last Keyframe Pose* ports, which are filled with a complete list of keyframes in order of insertion and the most recent keyframe, respectively. They are updated whenever a new keyframe has been added. Depending on the needs of the consumer of the component, only looking at the most recent keyframe may be more appropriate.

Right now, only the keyframe poses are provided as output, which is mostly useful for visualization (section 4.4), but the data could easily be extended to also include the

full images or even the set of visible map points per keyframe, which would be helpful for further postprocessing of the reconstruction.

Finally, for debugging, the component provides pairs of 2D image positions through *Trail Start Positions* and *Trail End Positions*. These are only relevant during PTAM's initialization stage, where the initial map is built from trail tracking. Both lists are pushed at the same time and have the same length, the first one contains the pixel coordinates of the tracked features in the very first keyframe, the second one for the current frame.

4.3 Adaptation of ATAM

After several instabilities and problems with PTAM arose, which are described in chapter 5, adaptation of a second SLAM algorithm for comparison became apparent. ATAM was chosen because of its recency and its clean code design, promising easy extensibility and integration into Ubitrack.

4.3.1 Overview over ATAM

Abecedary Tracking and Mapping (ATAM) [23] is an open-source toolkit for visual SLAM that was developed for the ISMAR tracking competition 2015. According to its authors, it is designed for beginners and to be easily modified. It is based on very similar principles as PTAM and works with the same setup: a single moving camera in an unknown environment.

The toolkit also comes with modules for camera calibration and video capture, but its main component is the SLAM algorithm – which is split into frame-by-frame tracking and keyframe-based mapping, just like PTAM. It is also a feature-based solution and uses bundle adjustment for optimization. However, there are a few key differences to PTAM:

- ATAM does not use epipolar search to match feature points between keyframes. Instead, it tracks their trails frame-by-frame with the Lucas-Kanade feature tracker [2] (KLT), which produces feature correspondences over a period of time. While PTAM only uses KLT in its initialization stage, ATAM uses the resulting trail start and end positions for each triangulation.

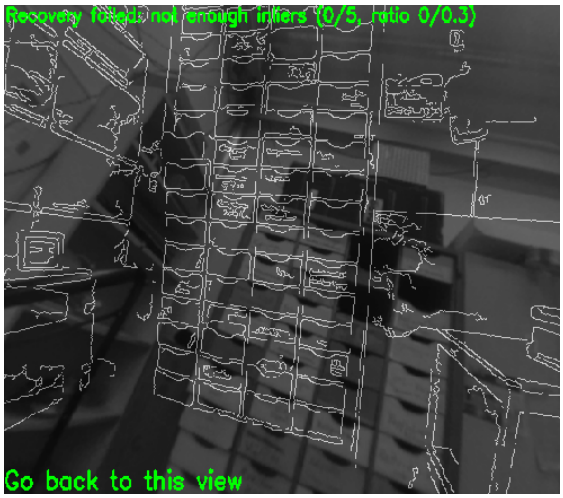

Figure 6: Example of the interactive relocalization process of ATAM

- ATAM performs both tracking and mapping on the same thread – which is made possible by not requiring expensive epipolar search. Only bundle adjustment is run on a separate thread.
- ATAM does not do any data association refinement – past keyframes are not searched for possible observations of new map points.
- PTAM does not insert new map points at locations where there is an existing map point in very close proximity. ATAM does not have this check, so its point clouds often contain noisy batches of many points in a small space for longer mapping runs, which should have ideally been recognized as a single feature point.
- When tracking is lost, ATAM starts an interactive relocalization process. It automatically selects the keyframe closest to the pose where tracking was lost, but also allows the user to manually select a keyframe in the interface. An overlay of the edges in the selected keyframe is displayed to the user and they are instructed to move the camera back to the corresponding location, while ATAM attempts to refind the mapped features that were visible in the keyframe to continue tracking. Figure 6 shows an example of such an overlay.

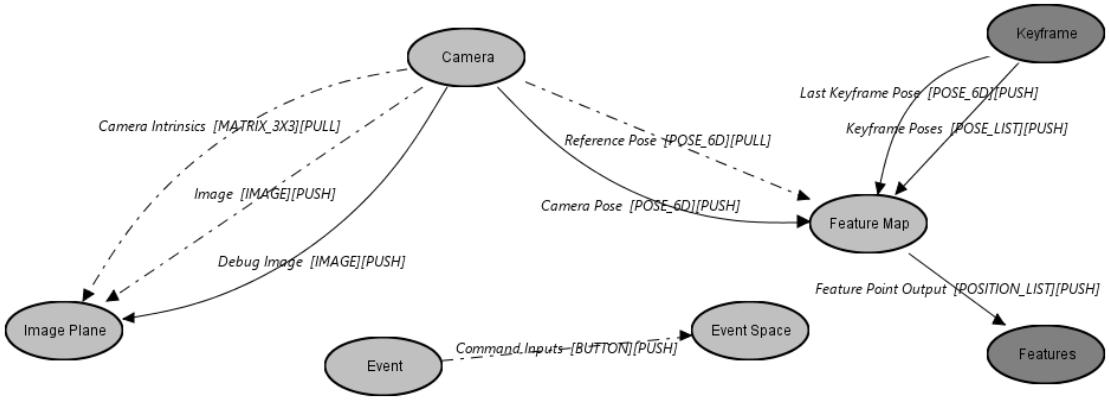

Figure 7: SRG pattern of the modified ATAM component

4.3.2 Integration of the reference pose

Due to its similarity to PTAM, the adaptation of ATAM to make it operate with a reference pose required almost identical steps as with PTAM, which are described in section 4.2.2. One notable difference is that the tracker still plays a central role, since the trails produced by KLT are directly used for mapping.

4.3.3 Integration into Ubitrack

ATAM's integration into Ubitrack had to be built from the ground up. The SRG pattern of the component, shown in figure 7, was modeled to be mostly equivalent to the pattern of modified PTAM. The input and output edges are the same, with the exception of a new *Debug Image* output port. This allows direct display of ATAM's internal state for debugging purposes, showing the received image in the background with various overlays. Most of the data is the one displayed in the original algorithm, for example the currently tracked feature points are shown (white, if they already exist in the map, blue if they are new point candidates).

Another small contribution that was made is a bugfix in the *cvsba* library [26] used by ATAM for bundle adjustment: Switching bundle adjustment to the mode where it only optimizes the map points and not the poses caused the wrapper to accidentally discard all the changes made to the map points, because of a small typing error.

4.4 Visualization in virtual reality

While the point cloud is in the process of reconstruction, it is desirable to view it directly within virtual reality. This enables an interactive reconstruction process, where the user can move the camera to areas of interest that have not been reconstructed with enough detail yet.

An immersive visualization of the feature points also provides a much more intuitive experience than displaying them on a regular screen. The user can judge the accuracy of the approximated geometry by physically moving around it and touching the real surfaces with their hands or controllers.

4.4.1 Technical framework

Unity was chosen as the game engine to display the generated point clouds. An integration of a Ubitrack DFN into Unity has already been developed at the chair [22].

To make this work, the desired output edges in the SRG are connected to an *Application Sink* component with a specific name, which can be used by a Unity component to read the data into the application.

The current implementation takes the generated point cloud from either ATAM or PTAM and puts it through a custom vertex shader that renders it as dots. Note that Unity imposes a technical limit of 65535 vertices per mesh, which can be circumvented by splitting the point cloud into multiple meshes. However, due to the sparseness of the point clouds, this has not been necessary in the applied scenarios so far.

The second type of data that is displayed is the set of keyframe poses, which both ATAM and PTAM provide as output. These are displayed as small floating cameras and help to understand which angles have been used for the reconstruction so far.

4.4.2 Interactivity

The reconstruction process can be started and stopped by pressing spacebar on the keyboard or the trigger button on one of the Vive controllers. After moving the camera horizontally, the same button must be pressed again. If successful, the rest of the reconstruction is automatic and points should start appearing in the scene.

The reconstruction can be stopped with the same keys (PTAM) or with the *R* key on the keyboard or menu button on the Vive controller (ATAM). To save the point cloud to a file, the 8 key can be used. With ATAM, it is also possible to view the current camera image and the active trails by using the touchpad.

4.4.3 Displaying stored point clouds

Separated from the immersive live reconstruction, stored point clouds can also be loaded and displayed in VR. This allows manual comparison of different reconstructions. In the scene, stored point clouds matching some file name pattern can be cycled through with the menu key on the controller.

5 Evaluation

In this chapter, the quality of the reconstruction pipeline will be assessed. Section 5.1 investigates the accuracy of the different camera calibration options available in OpenVR. It is followed by a qualitative comparison between the two main reconstruction algorithms PTAM and ATAM in section 5.2. Next, the effect of using the reference pose on accuracy is discussed in section 5.3. Finally, the general suitability of the approach for bringing the real environment into VR is evaluated in section 5.4.

Quantitative tests for the accuracy of the generated point clouds are difficult to make: An objective measure of the quality of a reconstruction would require comparison to a ground truth, which was not available for the test environment. While acquisition of ground truth data with a high-quality scanner would be possible, that would be beyond the scope of this thesis.

Consequently, most of the results are derived from working with the algorithms, from the analysis of their principles and from the interpretation of the generated example point clouds.

5.1 Comparison of camera calibration modes

Accurate camera calibration is an important prerequisite for precise reconstruction results. As described in section 4.1.3, there are multiple paths to obtain an undistorted image and the camera intrinsics that are necessary for the reconstruction: (a) use OpenVR's internal undistortion algorithm and its hardcoded intrinsic parameters or (b) use OpenVR's undistortion algorithm, but do manual camera calibration to get intrinsic parameters or (c) use the raw distorted image and do manual camera calibration to get undistortion and intrinsic parameters.

Table 1 lists the resulting values (focal length and center of projection) for the camera intrinsics of each calibration mode. For manual calibration, images of a known chess-

	Mode		Intrinsics			
	OpenVR undistortion	Manual calibration	f_x	f_y	p_x	p_y
(a)	yes	no	279.60	279.60	-302.80	-224.04
(b)	yes	yes	279.79	279.18	-305.91	-236.49
(c)	no	yes	278.68	278.76	-303.20	-237.64

Table 1: Camera intrinsics of different calibration modes

board structure were taken from different angles and used to compute the parameters. Images were added until convergence was reached. Note that mode (c) also resulted in distortion parameters, which are only significant for that mode and are not displayed in the table.

The focal lengths of each mode differ very little – the hardcoded values deviate less than one pixel – while the focal point shows differences of up to four pixels, suggesting some inaccuracy of the static parameters included in OpenVR.

A more precise analysis would be possible by calculating the reprojection errors of a fixed set of 3D points.

5.2 Comparison of PTAM and ATAM

While testing, PTAM showed several shortcomings, some of which were difficult to reproduce consistently and to debug, which motivated the adaptation of ATAM as an alternative. Notable problems include:

Lack of synchronization As its name suggests, PTAM's tracking and mapping run in parallel. Despite continuously accessing shared resources (most importantly the map points), the code does not contain traces of any attempts to synchronize such access or using concurrent data structures. This enables many race conditions throughout the algorithm. While it is unclear how frequently race conditions in PTAM occur in practice and how much they contribute to the experienced instability problems, they make the algorithm very unreliable.

In contrast, ATAM only performs bundle adjustment in a separate thread and takes

accordant measures to ensure thread safety: Relevant keyframes and map points are copied while protected by a mutex, bundle adjustment is run only on this copy and the results copied back inside a mutex as well. Additionally, no new keyframes may be inserted while bundle adjustment is active to prevent incorrect associations when copying back the data.

Retrofitting proper synchronization into PTAM would be a possible but time-consuming task, as shared resources are accessed in many places in the program and resource-intensive operations such as bundle adjustment would most likely have to be redesigned so performance is not impacted too much.

Instability Occurring both in the original PTAM algorithm and its modification, the map is occasionally corrupted during longer runs. It is unknown if this happens because of the concurrency issues mentioned above or because of another bug. The corruption usually originates as a numerical error during bundle adjustment, after which every subsequent adjustment of the map is unsuccessful, producing a barrage of errors and sometimes resulting in a full crash. Resetting PTAM and starting over with an empty map fixes the problem.

Crashes In the modified version of PTAM, its initialization process crashes the entire application for a particular set of reference poses for the initial two keyframes. In the test environment, it could be reproduced every time the camera was pointed in one particular direction of the room when the initialization key was pressed. The crash has been traced down to occur within the initial bundle adjustment that is performed immediately after trail tracking has finished. It may be caused by an oversight during modification or have a different cause. The original algorithm had no reference pose and used hardcoded coordinates for the initial two keyframes (placing the world origin at the first keyframe), so the issue did not apply there. A possible workaround would be to use the original hardcoded coordinates for initialization and transform the map right after the initial bundle adjustment has finished, so its coordinate system matches the one used by the reference poses again.

ATAM has much more concise and simpler code than PTAM. On the one hand, this is possible because it was written much more recently than PTAM and makes

heavy use of built-in features of the *OpenCV* and *cvsba* libraries. Its up-front design to be targeted at beginners and modifiable was also noticeable during its adaptation and integration into Ubitrack: Redundant parts of the code such as image acquisition, camera calibration and rendering to a window could easily be removed.

On the other hand, ATAM also lacks several features of PTAM, as described in section 4.3.1. One drawback that is noticeable in the resulting point clouds is the noisy batches of map points that should have been recognized as a single feature, which PTAM manages to do considerably better. Other than that, point clouds generated by PTAM and ATAM look visually very similar.

Another drawback of ATAM is its reliance on feature trails: For a new point to be added to the map, it has to have a complete trail, whose start and end positions are used for triangulation. Since new trails are only started when a keyframe is added, a trail needs to be tracked across all frames between two subsequent keyframes. If a trail is lost due to shaky camera motion or obstruction, the candidate is discarded. This enforces careful operation of the camera to ensure that enough new trails "survive" between keyframes, otherwise expansion of the map is completely hindered. Contrast this with the epipolar search in PTAM, which only relies on the measurements of the new keyframe and which considers matches in all other keyframes in the map (not just the most recent one). Evidently, this comes at the cost of increased computation time, but since PTAM performs this step on the mapping thread and it is still orders of magnitude faster than bundle adjustment, the spent time is negligible.

In summary, PTAM tends to produce reconstructions with slightly better quality, but with less detail and an implementation that is much more unreliable and prone to crashing than ATAM. Furthermore, ATAM is also much more convenient to work with with respect to code quality and maintainability.

5.3 Impact of the reference pose

Despite the reference pose from OpenVR being very precise and globally consistent, its use as a replacement for the internal tracking system seems to increase the noise in the reconstructed map points instead of decreasing it. This holds true for both PTAM and ATAM, but the induced noise in ATAM is much more noticeable.

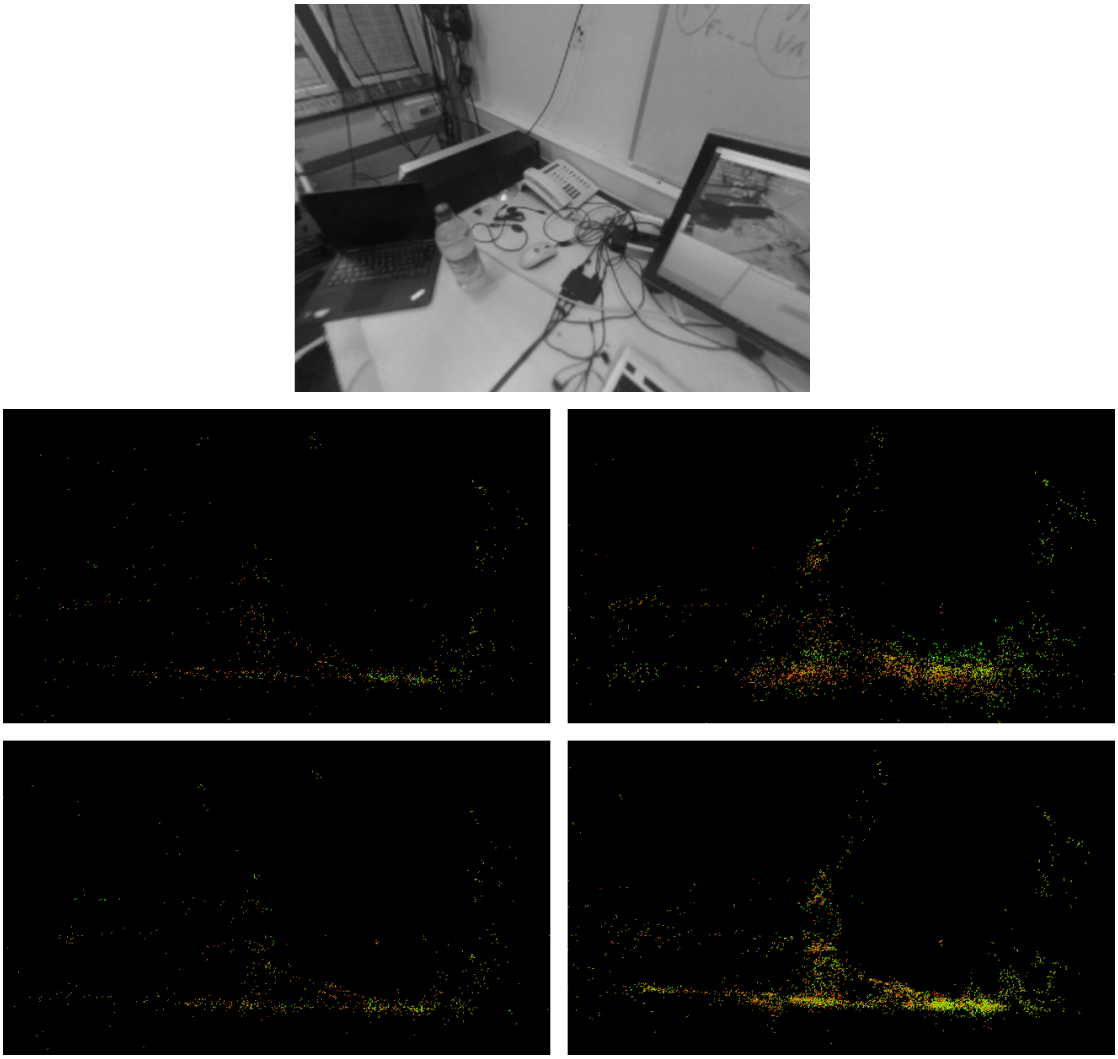

Figure 8: Comparison of point clouds, shown as an orthographic projection from the front of the table. On the left: PTAM, on the right: ATAM, at the top: tracking with reference pose, at the bottom: original tracking

Figure 8 shows a side-by-side comparison of point clouds by PTAM and ATAM operating in two modes: On the top, the modified algorithms that always use the reference pose as described in the approach. On the bottom, the algorithms using their original tracking modules. Here, the reference pose is only used to initialize the map at the correct origin and scale and for relocalization when tracking is lost. Each picture is an orthographic projection of the respective point cloud viewed from the side. The data was generated by capturing a short image sequence as well as their corresponding reference poses and playing them back in real time for each version of the algorithm.

Especially the point clouds of ATAM show a drastic difference: On the bottom, the flatness of the table is much better captured and the general shape of the water bottle in the scene can be recognized, despite the orthographic view.

However, this seeming higher accuracy of the map is contrasted with a lack of consistency with the world. While the shape of objects and surfaces appears to be better reconstructed with internal tracking, their location and orientation in the world are no longer accurately represented. A potential cause for this could be that as soon as the internal tracking pose deviates from the reference pose, measurements are no longer made in the same coordinate system as the reference pose and thus drift away further and further from their "real" location.

This has been tested with ATAM in "internal tracking mode": In the beginning, the internal pose remains relatively close to the reference pose (about 2-5 cm distance in position). But as soon as about a dozen keyframes have been added and bundle adjustment has run a couple of times, the distance quickly increases to 15 cm or more. Both tracking and mapping continue without any problems, but when viewing the point cloud in VR, the reconstructed geometry is often quite far away from its real counterpart.

5.4 General observations

Beyond their differences in the details, PTAM and ATAM largely rely on the same principle: feature-based real-time visual SLAM. But are the scans they produce suited for the pursued goal of warning the VR user of obstacles or even allowing interaction with the real world?

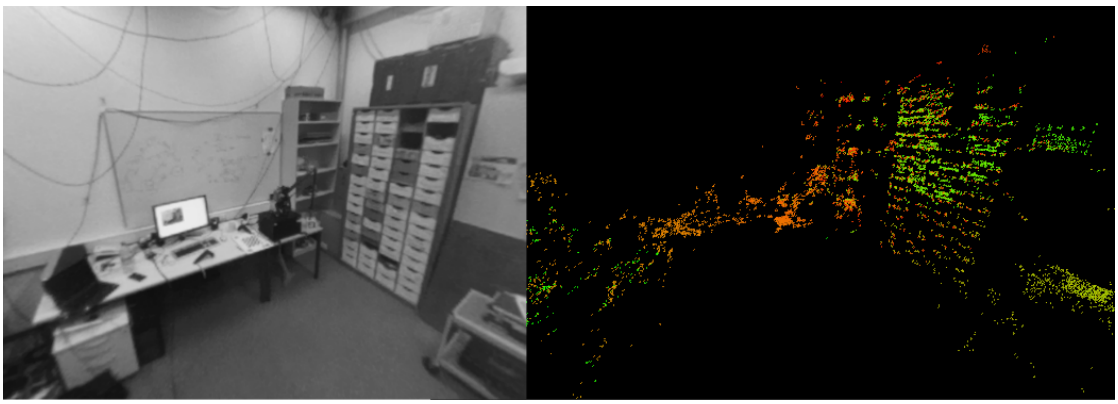

Figure 9: Example of a scene (left) and its point cloud (right) reconstructed with ATAM

As a matter of principle, the feature-based nature of the algorithms makes them very weak with untextured objects [9]. Areas of the room with low-contrast colors barely get any feature points, so they are mostly missing in the reconstruction. An example of this can be seen in figure 9: While high-detail areas like the table and drawers have plenty of map points, the largely single-colored wall and floor have almost none.

When SLAM is used with localization of the camera as the primary goal, this is less of a problem: Tracking continues to work as long as there are any visible high-contrast areas. But here, the focus lies exclusively on the resulting map, and the holes in the map become glaringly apparent. When these reconstructions would be used to warn of obstacles (after further processing), entire objects may be missing from the model.

As of right now, the reconstruction process is also not ready to be used by users not familiar with the algorithms. Care has to be taken to move the camera smoothly enough at the beginning to get a good enough initialization and to make sure that moving it into new areas is done at a pace where mapping can keep up. For example, when ATAM requires relocalization, the user has to stop exploring and go back to a good section of the map before continuing. However, these are limitations imposed by the current implementation and could still be made more robust and "out of the way" of the user by automatically reinitializing the algorithms with the help of the reference pose whenever required.

6 Future Work

A lot remains to be done to make the software produce usable reconstructions and to incorporate it into VR experiences. In order to move beyond sparse point clouds, algorithms that generate actual 3D meshes from them can be deployed. They can connect neighboring points to create surfaces and use image data from the keyframes to generate textures for them.

It is questionable if the quality of the point clouds as of now is sufficient for that. A likely prerequisite would be another post-processing step that reduces the noise and – for example – detects the batches of points often created by ATAM and merges them together. This may be enough to at least find larger flat surfaces in the scene with acceptable accuracy.

Another problem is the current choice between local consistency and global alignment, depending on whether the reference pose is used for tracking or not. Further investigation has to be done to identify the cause of the noise introduced by using the reference pose.

Once the issues with regard to accuracy have been resolved, the algorithm can be embedded as a background service to replace SteamVR's Chaperone system. Realistically, it would use a stored map and display parts of it that the user is too close to, while having a separate application to generate such a map. If the performance overhead can be made small enough, it could even be possible to continuously run mapping in the background (while the user is engaged in a different VR experience) in a mode that tries to detect changes in the environment, so an update of the scan can be made, for example when furniture has been moved.

If it is found that the quality of the scan is not satisfactory despite all efforts of post-processing, a switch to a completely different algorithm should be considered. Aforementioned options that directly produce dense reconstructions and rely on prob-

abilistic models may be better suited for an accurate real-time map. If that still does not produce good results, the "incremental" and "real-time" requirements for the reconstruction could be dropped altogether, settling for a long-running one-time computation, which would in turn require a more intricate guidance of the user to take the appropriate pictures.

7 Conclusion

In this thesis, two options to do 3D reconstruction in real time with a camera tracked by SteamVR have been explored. For this, a Ubitrack driver to obtain camera and pose data from OpenVR was implemented, as well as components that run a modified version of PTAM or ATAM based on this data and a VR application to control and display them. The differences between the two algorithms have been highlighted, most notably PTAM's instability and ATAM's simplicity and smaller feature set.

Modifications to both algorithms revolved around the reference pose, which was not present in their original design. It has been observed that completely replacing their internal tracking with the reference pose paradoxically leads to an increase of noise in the point clouds, but it still helps them with staying aligned with the real world.

The current implementation is a decent start and can be used for a rough estimate of the environment, but it leaves a lot of work to be done in order to reach a high-fidelity 3D model and to further integrate it into VR. Once that is achieved, tight spaces can be used to full capacity and would no longer be such a big limitation for virtual reality enthusiasts around the world.

List of Figures

Bibliography

[1] S. Benhimane and E. Malis. "Homography-based 2d visual tracking and servoing." In: *The International Journal of Robotics Research* 26.7 (2007), pp. 661–676.

[2] J.-Y. Bouguet. "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm." In: *Intel Corporation* 5.1-10 (2001), p. 4.

[3] R. Carceroni, A. Kumar, and K. Daniilidis. "Structure from motion with known camera positions." In: *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2006, pp. 477–484.

[4] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse. "MonoSLAM: Real-time single camera SLAM." In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1052–1067.

[5] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004.

[6] R. I. Hartley and P. Sturm. "Triangulation." In: *Computer vision and image understanding* 68.2 (1997), pp. 146–157.

[7] P. Keitler, D. Pustka, M. Huber, F. Echtler, and G. Klinker. "Management of tracking for mixed and augmented reality systems." In: *The Engineering of Mixed Reality Systems*. Springer, 2010, pp. 251–273.

[8] G. Klein and D. Murray. "Parallel Tracking and Mapping for Small AR Workspaces." In: *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*. Nara, Japan, Nov. 2007.

[9] T. Moons, L. Van Gool, M. Vergauwen, et al. "3D reconstruction from multiple images part 1: principles." In: *Foundations and Trends® in Computer Graphics and Vision* 4.4 (2010), pp. 287–404.

[10] P. Moulon, P. Monasse, R. Marlet, et al. *OpenMVG. An Open Multiple View Geometry library.* `https://github.com/openMVG/openMVG`.

[11] E. Mouragnon, M. Lhuillier, M. Dhome, F. Dekeyser, and P. Sayd. "Generic and Real-Time Structure from Motion." In: *BMVC*. Vol. 7. 4. 2007, p. 6.

[12] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. "DTAM: Dense tracking and mapping in real-time." In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2320–2327.

[13] J. Newman, M. Wagner, M. Bauer, A. MacWilliams, T. Pintaric, D. Beyer, D. Pustka, F. Strasser, D. Schmalstieg, and G. Klinker. "Ubiquitous tracking for augmented reality." In: *Mixed and Augmented Reality, 2004. ISMAR 2004. Third IEEE and ACM International Symposium on*. IEEE. 2004, pp. 192–201.

[14] M. Pizzoli, C. Forster, and D. Scaramuzza. "REMODE: Probabilistic, monocular dense reconstruction in real time." In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE. 2014, pp. 2609–2616.

[15] D. Pustka. "Construction of data flow networks for tracking in augmented reality applications." In: *Proc. Dritter Workshop Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR*. 2006.

[16] D. Pustka, M. Huber, M. Bauer, and G. Klinker. "Spatial relationship patterns: Elements of reusable tracking and calibration systems." In: *Proceedings of the 5th IEEE and ACM International Symposium on Mixed and Augmented Reality*. IEEE Computer Society. 2006, pp. 88–97.

[17] E. Rosten and T. Drummond. "Fusing points and lines for high performance tracking." In: *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*. Vol. 2. IEEE. 2005, pp. 1508–1515.

[18] D. Scharstein and R. Szeliski. "High-accuracy stereo depth maps using structured light." In: *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2003, pp. I–I.

[19] J. L. Schönberger and J.-M. Frahm. "Structure-from-Motion Revisited." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[20] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm. "Pixelwise View Selection for Unstructured Multi-View Stereo." In: *European Conference on Computer Vision (ECCV)*. 2016.

[21] H. Stewenius, C. Engels, and D. Nistér. "Recent developments on direct relative orientation." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 60.4 (2006), pp. 284–294.

[22] Ubitrack. *Ubitrack/integration_unity3d: unity3d integration of ubitrack*. URL: `https://github.com/Ubitrack/integration_unity3d` (visited on 08/19/2017).

[23] H. Uchiyama, T. Taketomi, S. Ikeda, and J. P. S. do Monte Lima. "[POSTER] Abecedary Tracking and Mapping: A Toolkit for Tracking Competitions." In: *Mixed and Augmented Reality (ISMAR), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 198–199.

[24] Valve. *API Documentation – ValveSoftware/openvr Wiki*. URL: `https://github.com/ValveSoftware/openvr/wiki/API-Documentation` (visited on 08/04/2017).

[25] Valve. *Welcome to Steamworks*. URL: `https://partner.steamgames.com/vrtracking` (visited on 08/27/2017).

[26] D. Zeng. *willdzeng/cvsba: cvsba: an OpenCV wrapper for sba library*. URL: `https://github.com/willdzeng/cvsba` (visited on 09/02/2017).