Interactive Multi-View Stereo Reconstruction

Masterarbeit im Fach Informatik

vorgelegt von

Darius Rückert

geb. am 10. April 1993 in Bamberg

angefertigt am

Institut für Informatik
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: M. Sc. Matthias Innmann, Dr. Thomas Hach

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 15. Juni 2017

Abgabe der Arbeit: 11. Januar 2018

Contents

Chapter 1

Introduction

1.1 Motivation

Medicine, film industry, robotics, city planning, virtual environment, earth observation, archeology, augmented reality, reverse engineering, animation, and human computer interaction are only a few fields that use 3D reconstruction systems [39]. In game development, all kinds of assets ranging from stones to human faces are more often scanned than modeled by hand [10]. Scanning takes usually significantly less time, while the produced model still exhibits fine details baked into high resolution textures. Google uses 3D reconstruction algorithms to automatically generate 3D models of the earth surface from satellite images [29]. Virtual and augmented reality systems require a reconstruction of the surrounding scene for displaying obstacles and integrating virtual objects. Some systems, for example Microsoft's Mixed Reality [53], only use RGB cameras mounted on the device for stable real-time tracking of head movement.

A special interest for researchers are reconstruction systems that rely on the images of a single RGB camera. Such systems bring 3D reconstruction to a large consumer base, because the required hardware is already widely spread through photographic cameras and smartphones. Several companies and universities already provide commercial and non-commercial software that produce good models from RGB images (see Section 2), but most of these products are tied to an offline reconstruction process, which means that their computation time ranges from several minutes to multiple days. On the other hand, existing real-time RGB reconstruction systems suffer from low quality output models as a result of using low resolution input images or by dropping expensive optimization techniques such as global bundle adjustment (Section 3.4).

In this work, we present a real-time monocular RGB reconstruction system (see Figure 1.1) that uses the same steps found in common offline reconstruction software. The goal is to provide a low resolution preview of the scanned object to the person controlling the camera (the operator) such that the preview can be inspected and undesired holes or artifacts can be removed by taking more images from different angles. During the scan, some frames are marked as *keyframes* and stored in full resolution on the disk. These keyframes can then be used in existing reconstruction software to create a high quality 3D model of the scene.

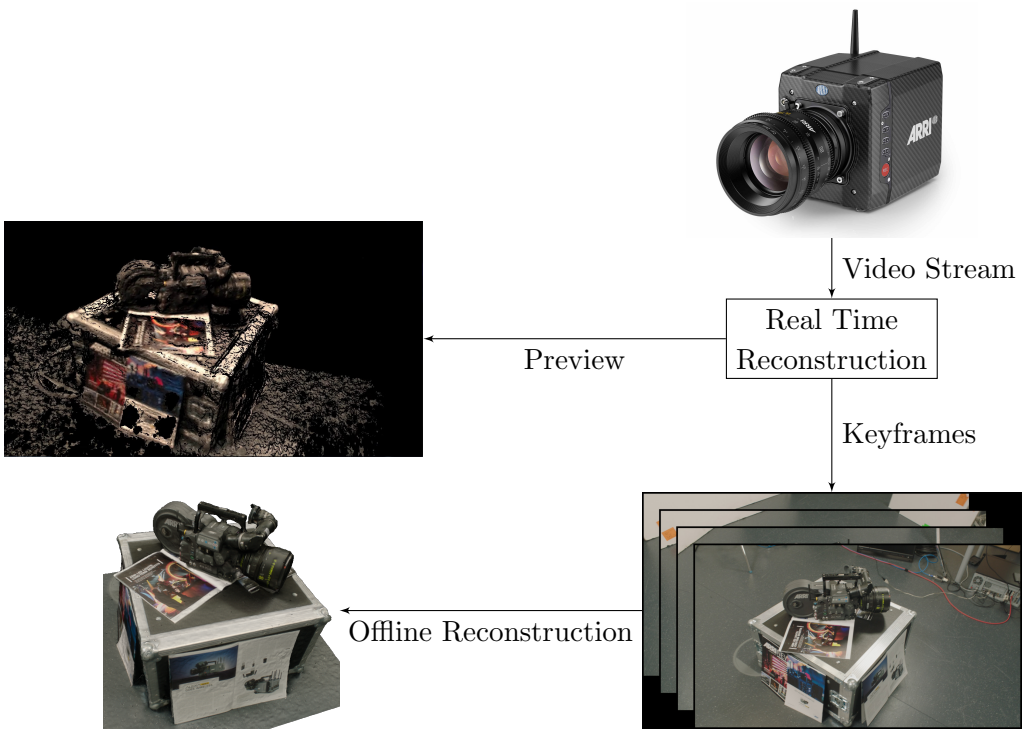

Figure 1.1: Overview of our interactive multi-view stereo reconstruction system. A low resolution preview of the scene is created from a live video stream and a few input *keyframes* are selected and stored in full resolution. These keyframes can then be used by offline reconstruction software to generate the final 3D model.

One use case of our system is film making, because scans of different assets are required and a good video camera is already present. The advantage in comparison to the traditional method of using a separate SLR camera is that the real-time preview of the reconstruction allows scanning by less experienced operators, because it is immediately visible if the reconstruction succeeds. This is an important property from a commercial point of view, because the assets might be only available for a limited period of time and the risk of a failed offline reconstruction should be reduced as much as possible. Furthermore, the color space of the scanned object is identical to the movie's color space, because the same camera is used.

1.2 Challenges

When building an interactive 3D reconstruction system that uses RGB images as input, several challenges arise:

1. **Robust monocular camera tracking:** For every frame, the camera pose relative to the reconstructed scene is computed. The tracking must be robust when previously seen parts of the image disappear due to occlusion and camera movement. *Drift*, which occurs when tracking errors accumulate over multiple frames, must be avoided.

2. **Reconstruction preview:** The reconstructed scene is rendered and displayed to the operator. This preview should be similar to the final model reconstructed by an offline process, but can be of lower resolution and exhibit some degree of noise. The operator should be able to identify weakly reconstructed parts of the scene such as holes.

3. **Interactive 3D reconstruction:** The complete reconstruction process should run at around 20-30 frames per second. All algorithms used in the process must be chosen and implemented carefully such that the total frame time does not exceed 50 ms.

4. **Ensuring success in offline reconstruction:** We want to ensure that the offline-generated 3D model from the selected keyframes is at least as good as the preview. This allows the operator to stop scanning when the desired quality level is reached, without the risk of a failing offline reconstruction.

1.3 Contributions

The challenges described in the previous section are solved in the following way:

1. **Robust monocular camera tracking:** The camera tracking is achieved by searching previously reconstructed 3D points in the new image and computing the corresponding camera pose (Section 4.4.2). For this purpose, we use SIFT keypoints and descriptors, which are invariant to translation, rotation, and scale (Section 4.2). Reprojection errors resulting in camera drift are reduced by using a nonlinear optimization over multiple frames (Section 4.7).

2. **Reconstruction preview:** A preview of the scene is generated by computing a depth map for every frame (Section 5.2) and fusing them into a truncated signed distance field (TSDF). This TSDF is rendered using ray casting (Section 5.4), resulting in a dense preview of the scene without the need of building a polygon mesh. The operator can directly spot holes and weakly reconstructed regions by inspecting the shaded preview surface.

3. **Interactive 3D reconstruction:** The input image stream is downscaled to a resolution of 960×540 and time-consuming steps are implemented on the GPU, enabling a real-time reconstruction of the scene. Only when a keyframe is selected, additional steps are performed, introducing a short delay of around 30 ms (see Section 6.1.4).

4. **Ensuring success in offline reconstruction:** Even though we can not guarantee a successful offline reconstruction, we try to maximize the success rate by using the same processing pipeline as common offline systems. This is one of the main differences to existing real-time reconstruction systems, because we do not replace expensive offline techniques with faster algorithms of lower quality.

Chapter 2

Related Work

2.1 Offline Reconstruction

Probably one of the most relevant publications in the field of large scale offline reconstruction is the work by Snavely et al, which was released as an open source project in 2008 under the name **Bundler** [73] [72]. The goal of their work was to compute camera parameters and a sparse reconstruction of the scene from unorganized and uncalibrated Internet photo collections (see Figure 2.1). This problem is also called *structure from motion* (SfM). They start their reconstruction by computing SIFT keypoints and descriptors for every image [46]. Next, for each pair of images, they match keypoint descriptors, using the approximate nearest neighbors (ANN) [5] and discard geometric outliers by robustly estimating a fundamental matrix [33] with RANSAC [19]. All pairwise matches are then organzied into *tracks*, where a track is a connected set of matching keypoints across multiple images. The actual reconstruction is started by computing the essential matrix of a single image pair with the five point algorithm [61] and triangulating the corresponding image matches. After that, the remaining images are added incrementally to the already reconstructed scene. Between each step, they use bundle adjustment [76] to reduce the overall reprojection error.

Figure 2.1: The Colosseum in Rome is reconstructed from 2,106 unorganized images by **Bundler** [72].

Another open source 3D reconstruction system is **Multi-View Environment** (MVE) [23]. In comparison to Bundler, MVE implements the complete end-to-end pipeline from photos of a scene as input to textured triangle meshes as output (see Figure 2.2). After the structure from motion part is completed, which is almost identical to Bundler, they use a Multi-View Stereo (MVS) approach [27] to reconstruct depth maps for every view. Next, the depth maps are merged into a hierarchical signed distance function (SDF), which is optimized for samples with different scales [22]. Finally, the isosurface is extracted from the implicit function [38] and converted to a triangle mesh.

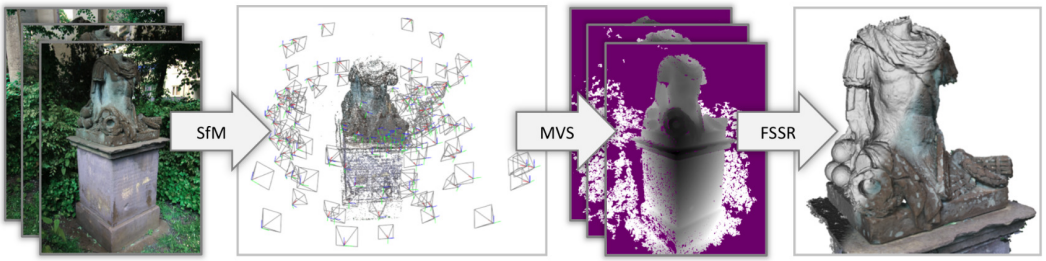

Figure 2.2: The reconstruction pipeline of **Multi-View Environment**. A set of input images is converted in multiple steps to a textured triangle mesh [23].

As the demand on user-friendly 3D reconstruction software increased over the years, several commercials products were released. One example is **PhotoScan**, developed by the Russian company Agisoft LLC [3]. PhotoScan is now widely used in the film industry [67] [18] and in game development [42] [17] [50]. Even though no publications describing the internal reconstructions algorithms were released, we can see from the manual [2], developer comments [70], and console output that a standard incremental SfM combined with a MVS depth map pipeline, similar to MVE, is used. The success of PhotoScan is most likely a result of providing the full end-to-end pipeline including mesh processing tools for automatically creating textured triangle meshes with a simple and self explaining user interface (see Figure 2.3).

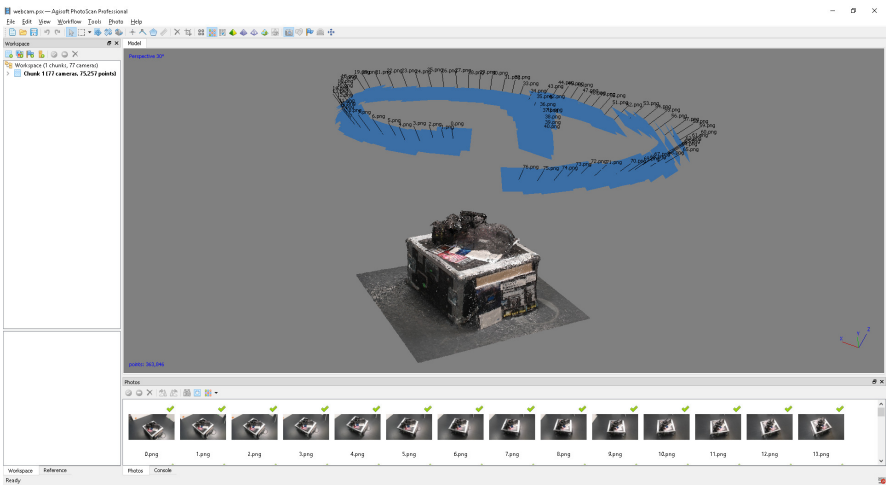

Figure 2.3: The user interface of **PhotoScan** [3].

2.2 Online Reconstruction

Online reconstruction systems are able to generate a 3D model of the scene, while the image stream is captured by a camera. This creates a strict constraint on processing time per frame, resulting in models of lower resolution compared to offline reconstruction systems. There exist a few fundamentally different methods for online reconstruction. These are explained in more detail in the following sections.

2.2.1 Active Depth Cameras

A stable method for online reconstruction, known as **KinectFusion** [58], relies on depth maps, captured from the handheld Kinect depth sensor. KinectFusion stores the scene in a voxel grid, where each voxel contains the signed distance to the surface. This allows direct rendering through ray casting and simple mesh generation with Marching Cubes [45]. The camera tracking is achieved by generating a 3D point cloud from the depth image and aligning this point cloud to a ray-casted point cloud of the scene with the iterative closest point algorithm [7]. Next, the depth map is added to the SDF with volumetric integration [12]. Small errors in the initial depth map are reduced by fusing depth maps over multiple frames and with a bilateral filter [75]. As a result, the reconstructed surface is smooth, even though the input images are noisy (see Figure 2.4). A disadvantage of the original KinectFusion implementation is the requirement of a dense three dimensional voxel grid. An improvement, known as **VoxelHashing** [59], was introduced to overcome this problem, which only reserves memory for voxels containing the surface. This is achieved by using a hash function that maps every discretized point in space to a one dimensional array of fixed size [74]. The hash-based system is more flexible than the original KinectFusion method, because it is able to reconstruct larger scenes and does not need initial scene bounds (see Figure 2.5).

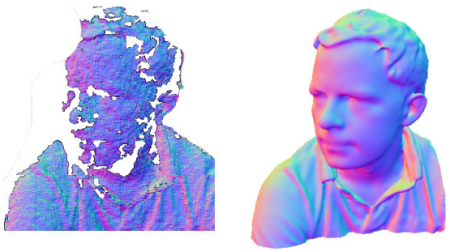

Figure 2.4: The noisy input image of the Kinect depth sensor (left) is fused into a smooth output model with volumetric integration (right) [58].

Figure 2.5: Online reconstruction of a 20m wide scene with **VoxelHashing** [59].

2.2.2 Keypoint-Based Camera Tracking

One of the first real-time reconstruction system, which relies only on the input of a single RGB camera, is **PTAM** (Parallel Tracking and Mapping) [40]. They inherently differentiate between finding the camera pose (Tracking) and sparsely reconstructing the scene (Mapping). In the tracking step, the camera pose is estimated by the previous frame's camera pose and a decaying velocity model. Next, the already reconstructed 3D points are projected to the estimated camera and compared to FAST-10 corner points [68] in a small radius. Given the 2D to 3D point correspondences, a camera pose update is computed by iteratively minimizing a robust objective function [37]. While tracking, some frames are identified as *keyframes* based on a heuristic of the tracking and map quality. When a new keyframe is encountered, new 3D points are added to the scene by triangulating image matches with the previous keyframe. This ensures that the scene dynamically grows when the camera moves to new locations. The mapping stage manages the scene and refines it through local and global bundle adjustment. The final result is a real-time tracking of the camera and a reconstructed sparse point cloud (see Figure 2.6).

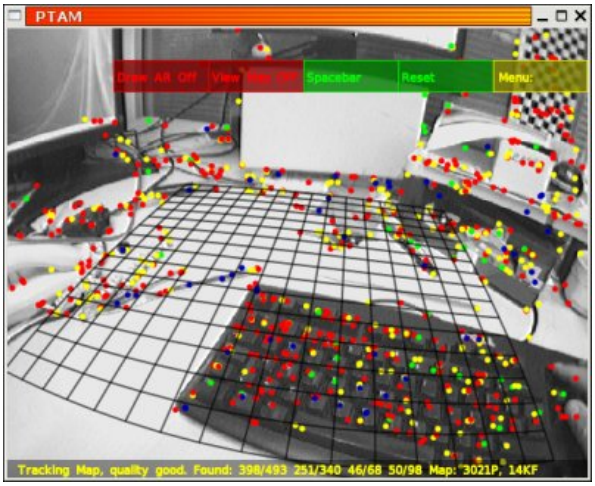

Figure 2.6: Real-time camera tracking with **PTAM**. The colored points are the 3D points of the scene projected to the current image [41].

More advanced simultaneous localization and mapping (SLAM) systems, such as Microsoft's **MonoFusion** [66] are able to generate dense 3D models in real time (see Figure 2.7). MonoFusion's camera tracking is using FAST feature points similar to PTAM, but the algorithms used in the reconstructions are more robust. They initialize the global map by computing an essential matrix for every frame with the five-point algorithm [61]. Camera tracking is

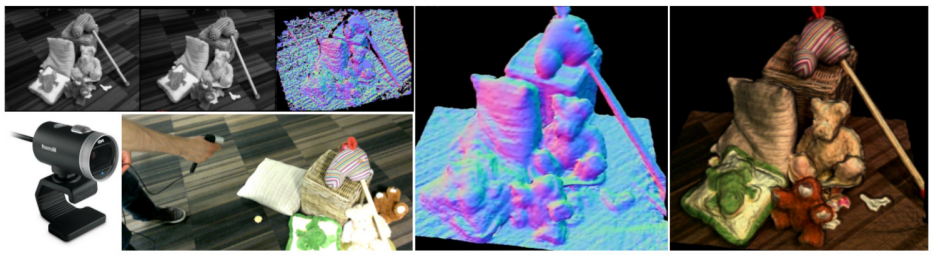

Figure 2.7: Small reconstructed scene with **MonoFusion**. The left half shows the input image and the estimated depth map for the current frame. In the right half, the output model can be seen with normal and color shading [66].

achieved by projecting previously reconstructed points to an estimated camera position and applying the perspective three point algorithm [26] in a RANSAC [19] setting. After the new pose is computed, the closest previous keyframe with a sufficient baseline is selected to compute a dense depth map for the new frame. They use a real-time variant of PatchMatch stereo [51], which first guesses random depth values for every pixel and then propagates the current value to neighboring pixels, minimizing an energy function based on the NCC score of image patches [25]. As a last step, the depth maps are fused into a truncated signed distance field and rendered with ray casting similar to KinectFusion [58] (see right image of Figure 2.7).

2.2.3 Direct Image Alignment

As a contrast to keypoint based tracking, direct methods exists that use visual odometry (VO) to optimize the geometry directly on the image intensities, enabling the use of all information in the image [57]. Such methods provide higher accuracy and robustness in environments with few keypoints. **Large-Scale Direct monocular SLAM** (LSD-SLAM) [15] uses direct image alignment coupled with filtering-based estimation of semi-dense depth maps as originally proposed in [16]. They represent the global map as a pose graph of keyframes as vertices with 3D similarity transforms as edges. This system is able to detect scale change in the environment and correct accumulated drift. LSD-SLAM runs in real time on a CPU and produces semi-dense reconstructions, which can be seen in Figure 2.8.

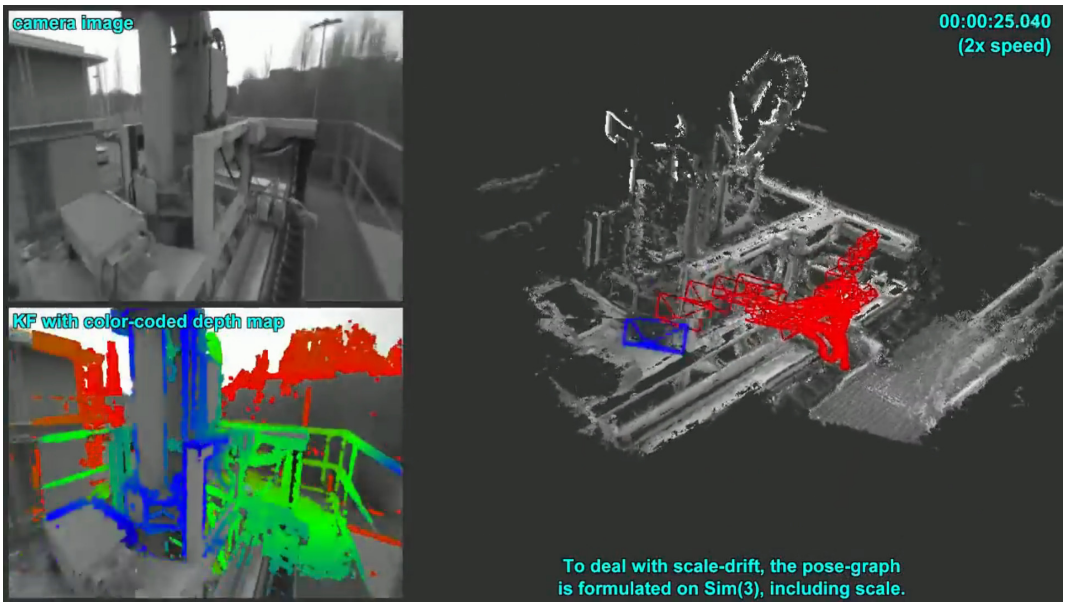

Figure 2.8: Semi-dense reconstruction of an outdoor scene with **LSD-SLAM**. On the left side the input image with its estimated depth map can be seen. The right side shows the final model in gray and the camera positions of the keyframes in red. The image was taken from an online video [14].

Chapter 3

Background

3.1 Camera Projection Model

A camera model is a mapping between the 3D world and a 2D image. In this work, we will use a finite projective camera [33] extended by a lens distortion model (Section 3.2). A point in space with coordinates $\boldsymbol{x} = (x, y, z)$ is mapped to a point on the image plane $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y})$, where the line joining $\boldsymbol{x}$ to the center of projection intersects the image plane. Assume that

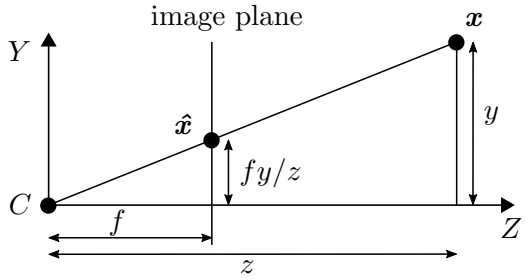

Figure 3.1: Projection of the point $\boldsymbol{x}$ to the image plane.

the camera center C is placed at the origin and its principal axis (the camera direction) is the z-axis. One can see by similar triangles that the point $\boldsymbol{x} = (x, y, z)$ is mapped to the point $\hat{\boldsymbol{x}} = (fx/z, fy/z)$ on the image plane, where the focal length f is the distance between the camera center and the image plane (see Figure 3.1). If the image point is represented by a homogeneous vector, the projection can be expressed as a linear mapping and therefore be written in terms of matrix multiplication:

$$\begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & & \\ & f & \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.1}$$

The expression (3.1) assumes that the origin of coordinates in the image plane is at the principal point, the pixels of the CCD chip are perfectly squared, and the image axes are perpendicular.

We use a more general projection model, also known as a **finite projective camera** [33]:

$$\begin{bmatrix} \hat{x} \\ \hat{y} \\ z \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ & f_y & c_y \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.2}$$

The 3×3 matrix in Equation (3.2) is called the *intrinsic matrix* $\boldsymbol{K}$. The variables c_x, c_y describe the location of the principal point, which is the intersection of the principal axis with the image plane. The parameter s is referred to as the *skew* parameter. Most cameras will have a skew of zero, however in certain unusual instances, such as the image axes being not perpendicular, it can take non-zero values.

We have assumed that the camera is placed at the origin looking in positive z-direction. An arbitrary camera pose can be allowed by applying a rigid transformation on the world points before projecting them onto the image plane. This rigid transformation maps points from world to camera space (or view space) and can be expressed as the 3×4 view matrix

$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix},$$

where $\boldsymbol{R}$ is a 3×3 rotation matrix and $\boldsymbol{t}$ a 3×1 translation vector. The full projection model, mapping world points to the image, can be seen in Figure 3.2. The distortion operation, transforming from *Normalized Image Space* to *Distorted Normalized Image Space*, is covered in the next section.

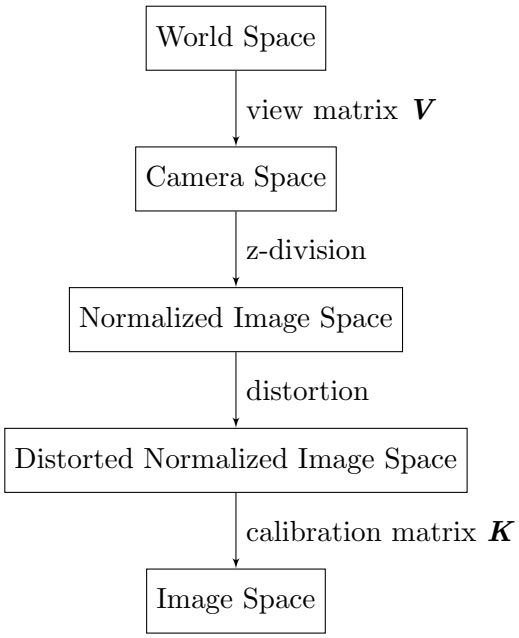

Figure 3.2: Projection of a point in world space to an image. If the distance between the camera and the 3D point is known, every step is invertible and therefore an image point can be projected back to the world.

3.2 Lens Distortion

Distortion is present in all images taken by cameras that deviate from the pinhole camera model; Therefore by all cameras which consist of one or more lenses. The lenses of professional SLR-cameras usually suffer from far less distortion than inexpensive lens system, which are used, for example, in smartphones and webcams. We generally differentiate between two kinds of distortion: *Radial distortion* and *decentering distortion* [8].

3.2.1 Radial Distortion

Radial distortion is a symmetric distortion that either magnifies or minifies points depending on their distance to the optical center (the radius). It can be seen in almost every camera, but is especially pronounced in single and wide angle lenses. When looking through a magnifying glass, we observe that lines far of the center are bent inwards (see Figure 3.3). This kind of distortion is also known as pincushion distortion. The opposite, barrel distortion, bends lines away from the optical center. Radial distortion is mainly caused by two physical properties of lens design. First, the focal length of the lens varies over the surface and second, the addition of a stop causes the chief ray to bend, depending on the distance from the optical center. This results in image parts that are more magnified than others, even though each point is still focused [71] [34].

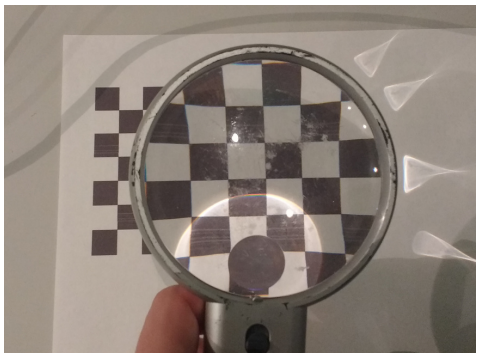

Figure 3.3: The straight lines of a checkerboard are bent when viewed through a magnifying glass.

3.2.2 Decentering Distortion

Decentering distortion is a non-symmetric distortion and is usually less pronounced than radial distortion. Points are not only moved outwards or inwards, but are also shifted along a vector perpendicular to the line connecting the optical center with the image point. The latter part of the distortion is called the tangential component, because its vector is tangent to the circle formed by the optical center and the image point. In contrast to radial distortion, decentering distortion is not a property of lens systems, but a result of misaligned lens elements. A slight tilt of one or more lenses in respect to the optical axis produces a distortion pattern in which one line through the optical center suffers no tangential distortion, while its perpendicular line undergoes maximal tangential distortion (see Figure 3.4) [8] [77].

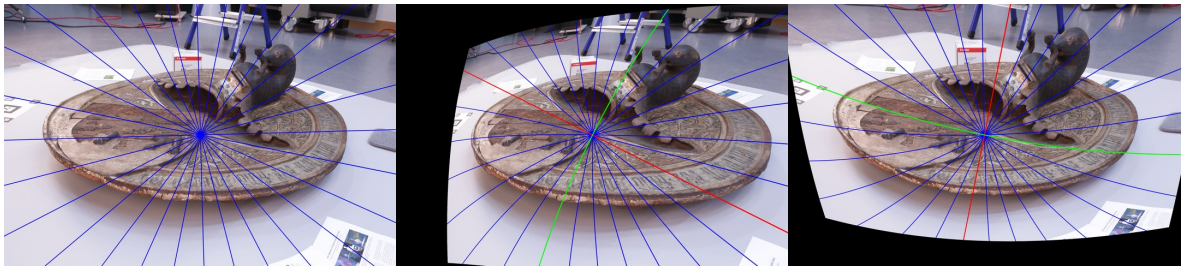

Figure 3.4: Decentering distortion adds a tangential distortion component. The red line is the axis of minimal tangential distortion. The green line is the axis of maximal tangential distortion.

3.2.3 Distortion Model

Compensating for distortion is very important for accurate 3D reconstruction purposes. The most commonly used mathematical model of lens distortion was introduced in 1966 by Brown [8], which is based on Conrady's model from 1919 [11]:

$$x_d = x_u + (x_u - x_c)d_r + d_{tx}$$

$$y_d = y_u + (y_u - y_c)d_r + d_{ty}$$

$$d_r = (K_1 r^2 + K_2 r^4 + K_3 r^6 + ...)$$

$$d_{tx} = (2P_1(x_u - x_c)(y_u - y_c) + P_2(r^2 + 2(x_u - x_c)^2))(1 + P_3 r^2 + P_4 r^4 + ...)$$

$$d_{ty} = (2P_2(x_u - x_c)(y_u - y_c) + P_1(r^2 + 2(y_u - y_c)^2))(1 + P_3 r^2 + P_4 r^4 + ...)$$

(x_d, y_d)	:	distorted image point
(x_u, y_u)	:	undistorted image point
(x_c, y_c)	:	distortion center = principal point
d_r	:	radial distortion
K_n	:	radial distortion coefficients
d_{tx}, d_{ty}	:	tangential distortion
P_n	:	tangential distortion coefficients
r	:	$\sqrt{(x_u - x_c)^2 + (y_u - y_c)^2}$

This model adds two terms d_r, d_t to the undistorted points x_u, y_u to obtain the distorted points x_d, y_d. These two terms describe the radial and tangential lens distortion and consist of an infinite series of polynomial coefficients. In our system, we use three radial distortion coefficients and two tangential distortion coefficients.

3.2.4 Inverse Distortion

Inverting the distortion function is required to convert points from the distorted image back to the undistorted image. However, Brown's model is a non-invertible mapping, so no analytical solution exists [13]. Reasonable approximations of this problem can be achieved with iterative algorithms. The compact formulation of Brown's distortion model with $\boldsymbol{p}_u$ being the undistorted point and $\boldsymbol{p}_d$ being the distortion point is:

$$\boldsymbol{p}_u d_r(\boldsymbol{p}_u) + d_t(\boldsymbol{p}_u) = \boldsymbol{p}_d \tag{3.3}$$

A simple iterative algorithm, also used in OpenCV [21], assumes that the radial and tangential distortion does not change between the estimated solution $\boldsymbol{p}_n$ and the real solution $\boldsymbol{p}_u$.

$$d_r(\boldsymbol{p}_u) = d_r(\boldsymbol{p}_n)$$

$$d_t(\boldsymbol{p}_u) = d_t(\boldsymbol{p}_n)$$

Putting this into equation (3.3) and solving it for $\boldsymbol{p}_u$ with $\boldsymbol{p}_u = \boldsymbol{p}_{n+1}$ results in the iterative method:

$$\boldsymbol{p}_{n+1} = \frac{\boldsymbol{p}_d - d_t(\boldsymbol{p}_n)}{d_r(\boldsymbol{p}_n)}$$

Another way of solving equation (3.3) is to convert it into a nonlinear least squares problem and solve it with classic optimization algorithms like Levenberg-Marquardt [49].

$$\min_{\boldsymbol{p}_u} |\boldsymbol{p}_u d_r(\boldsymbol{p}_u) + d_t(\boldsymbol{p}_u) - \boldsymbol{p}_d|^2$$

These iterative algorithms need a good initial guess to find the correct solution. We assume that the overall distortion is relatively small, so a good initial value is $\boldsymbol{p}_n = \boldsymbol{p}_d$. In practice, both iterative algorithms converge in about five iterations. The nonlinear optimization method yields better or equally good solutions compared to OpenCV's simple algorithm, but also take more computation time. In this work, we use the simple algorithm, because the precision is good enough for our purpose and we strive for real-time performance.

3.3 Camera Calibration

Camera calibration is a necessary step in computer vision in order to extract metric information from 2D images. Existing calibration methods can be roughly classified into two categories [81]:

1. **Reference object based calibration.** Camera calibration is performed by observing a calibration object whose geometry in 3D space is known with very high precision. A calibration object can be anything, but it is advantageous to use easily detectable patterns such as checkerboards.

2. **Self-calibration.** Self-calibration or auto-calibration methods compute the internal camera parameters while reconstructing a static scene. Correspondences between three images are sufficient to recover both the internal and external parameters, which allow to reconstruct 3D structure up to a similarity [48] [32].

In this work, we use the popular reference object based calibration by Zhang [81]. First, multiple images (at least two) of a known planar pattern are taken by either moving the pattern itself or moving the camera. The pattern must be detectable in all images. We use a black and white checkerboard as a calibration pattern and detect it with OpenCV's `findChessboardCorners` [21] method (see Figure 3.5).

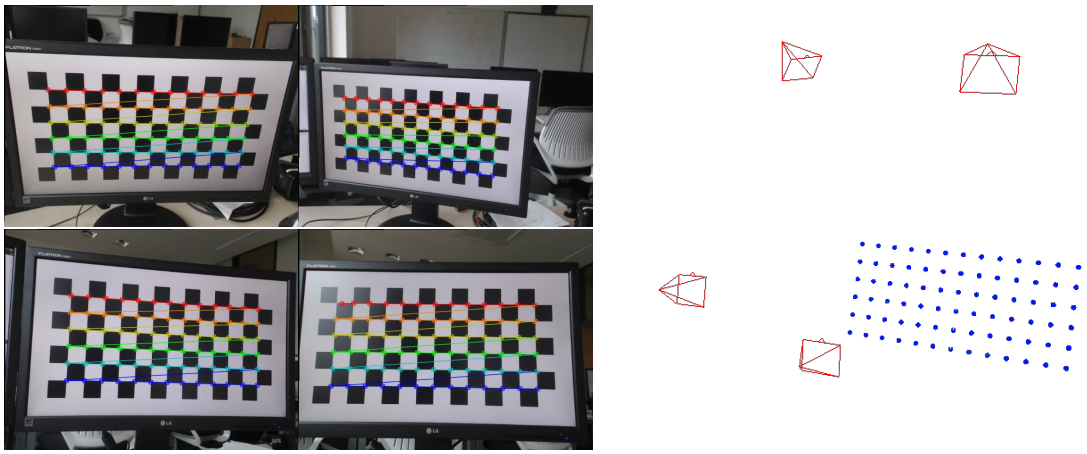

Figure 3.5: Camera calibration from four images of a planar pattern. On the right side the computed camera poses can be seen in red and the 3D point of each checkerboard corner in blue.

Without loss of generality, we define the model lying in the x-y-plane with $z = 0$. Therefore, a model point is related to its image point by a homography. The homography is a 3×3 matrix with 8 degrees of freedom and is computed by solving a linear system of equations. Each model-to-image point correspondence yields two equations, thus the calibration pattern has to have at least 4 detectable points. Having multiple homographies (for each image one) allows us to directly estimate the intrinsic camera matrix $\boldsymbol{K}$, again by solving a linear system. Given $\boldsymbol{K}$, the camera pose is computed for every input image. As a last step, the camera pose and $\boldsymbol{K}$ are refined with bundle adjustment (Section 3.4). The nonlinear optimization also allows us to find the parameters of the distortion model described in Section 3.2.

In the remainder of this thesis, we assume that the camera is calibrated, which means that the calibration matrix $\boldsymbol{K}$ and the distortion parameters are known. In practice, we calibrate the camera once before reconstruction by taking multiple images of a checkerboard displayed on a computer screen, as seen in Figure 3.5.

3.4 Bundle Adjustment

Consider a situation in which a set of 3D points X_j is viewed by a set of cameras with parameters P^i. We wish to solve the following reconstruction problem: Given the set of image points x_j^i corresponding to the 3D points X_j, we want to find the camera parameters P^i as well as the world points X_j such that

$$\text{project}(P^i, X_j) = x_j^i, \tag{3.4}$$

where *project* is the world to image projection defined in Section 3.1. If the measurements are noisy, equation (3.4) will not be satisfied exactly. In this case we seek the *Maximum Likelihood* [31] solution, assuming that the measurement noise is Gaussian: We wish to find the 3D points X_j and camera parameters P^i that project the world point X_j to exactly an image point $\hat{x}_j^i$ and minimize the geometric distance between the observed point x_j^i and the projected point $\hat{x}_j^i$.

$$\begin{aligned} \min_{P,X} \sum_i \sum_j ||\hat{x}_j^i - x_j^i||_2^2 \\ \hat{x}_j^i = \text{project}(P^i, X_j) \end{aligned} \tag{3.5}$$

These equations, which involve minimizing the reprojection error, are known as *bundle adjustment*, because the bundle of rays between each camera center and the set of 3D points is adjusted [33]. Direct solutions with arbitrary projection functions for equation (3.5) do not exist, therefore a nonlinear iterative optimization technique with a good initial guess has to be applied. In this thesis, we use the Levenberg-Marquardt [55] algorithm implemented in the open source library *Ceres* [1].

There exist multiple variants of bundle adjustment depending on which parameters are optimized in the procedure. When the camera is calibrated the general bundle adjustment equations (3.5) can be simplified by projecting the world points only to normalized image space (see Section 3.1). This significantly speeds up the process, because the expensive distortion model (Section 3.2.3) does not have to be evaluated. Care has to be taken that the resulting reprojection error of equation (3.5) is not given in pixels, but rather in normalized image units. Usually this is not an issue, but, for example OpenCV, still converts the error back to pixels by multiplying with the focal length [21].

3.5 Perspective-Three-Point Problem

The Perspective-n-Point (PnP) problem is originated from camera calibration [81]. Also known as pose estimation, it is to determine the position and orientation of the camera with respect to a scene object from n corresponding points [26]. The Perspective-Three-Point (P3P) problem is the minimal version of PnP with exactly three control points that results in at most four solutions.

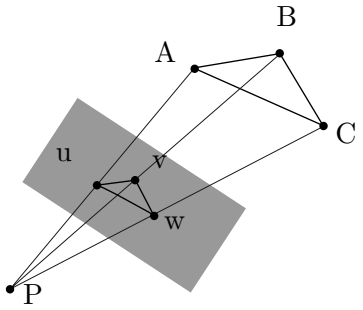

Figure 3.6: Three world points A,B and C are projected to their corresponding image points u,v,w. The rays connecting the world- and image points all intersect in the camera's optical center P.

Instead of directly solving for the view matrix V, a possible set of distances $\{|PA|, |PB|, |PC|\}$ is computed and converted to a pose configuration (see Figure 3.6). The P3P equation system, seen after this paragraph, is based on the law of cosines that links affine distances with angles.

$$Y^2 + Z^2 - YZp - a'^2 = 0$$

$$X^2 + Z^2 - XZq - b'^2 = 0$$

$$X^2 + Y^2 - XYr - c'^2 = 0$$

$$|PA| = X,\ |PB| = Y,\ |PC| = Z,$$

$$a = \angle BPC,\ b = \angle APC,\ c = \angle APB$$

This equation system can be solved with Wu Ritt's zero decomposition [78], resulting in a 4th degree polynomial, giving us up to four real solutions. The distances are then converted to 3D points in camera space by multiplying each distance with the normalized vector of each image point in normalized image space (see Section 3.1). The final solution, consisting of a rotation matrix and a translation vector, is computed by finding a rigid transformation that maps the 3D world points to the camera space points. This problem is also known as *Procrustes Problem* [30] and can be solved with singular value decomposition [28]. Finally, given the four possible pose configurations, a forth point is projected to each solution and the pose producing the lowest reprojection error is selected.

3.6 Triangulation

Triangulation describes the method of finding the position of a point in 3D space given its image in two views and the camera parameters of both views (see Figure 3.7).

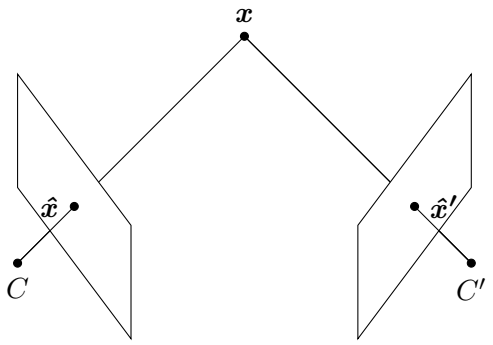

Figure 3.7: A point $\boldsymbol{x}$ is seen by two cameras. The rays starting from each camera center passing through the imaged points $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{x}}'$ intersect at the world point $\boldsymbol{x}$.

If there are small errors in the measured image points, a perfect triangulation is not possible, because the rays passing through each camera center and the image points do not intersect [33]. We start the triangulation by backprojecting each image point to normalized image coordinates (Section 3.1) resulting in the following two equations

$$\hat{\boldsymbol{x}} = \boldsymbol{V}\boldsymbol{x}$$
$$\hat{\boldsymbol{x}}' = \boldsymbol{V}'\boldsymbol{x},$$

where $\boldsymbol{x}$ is the target 3D point, $\boldsymbol{V}$, $\boldsymbol{V}'$ are the view matrices of each camera and $\hat{\boldsymbol{x}}, \hat{\boldsymbol{x}}'$ are the normalized image points in homogeneous coordinates. The homogeneous scale factor can be eliminated by a cross product to give three equations for each image point (one equation for each component of $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$):

$$\hat{\boldsymbol{x}} \times \hat{\boldsymbol{x}} = \boldsymbol{0} = \hat{\boldsymbol{x}} \times \boldsymbol{V}\boldsymbol{x}$$
$$\hat{\boldsymbol{x}}' \times \hat{\boldsymbol{x}}' = \boldsymbol{0} = \hat{\boldsymbol{x}}' \times \boldsymbol{V}'\boldsymbol{x}$$

Only four of these six equations are linearly independent, resulting in a 4×4 homogeneous system of linear equations

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} \hat{x}\boldsymbol{v}^{3T} - \boldsymbol{v}^{1T} \\ \hat{y}\boldsymbol{v}^{3T} - \boldsymbol{v}^{2T} \\ \hat{x}'\boldsymbol{v}'^{3T} - \boldsymbol{v}'^{1T} \\ \hat{y}'\boldsymbol{v}'^{3T} - \boldsymbol{v}'^{2T} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \boldsymbol{0},$$

with $\boldsymbol{v}^i$ being the i-th row vector of $\boldsymbol{V}$. The solution is overdetermined, because four equations are given for only three unknowns. If the rays do not intersect in space, no solution of $\boldsymbol{x}$ can satisfy all four equations. The triangulation method described in this section is also known as *linear triangulation* and does not give a geometrically optimal solution. Therefore, after the estimated 3D point is computed, we apply bundle adjustment (Section 3.4) to further optimize the world position.

Chapter 4

Camera Tracking and Sparse Reconstruction

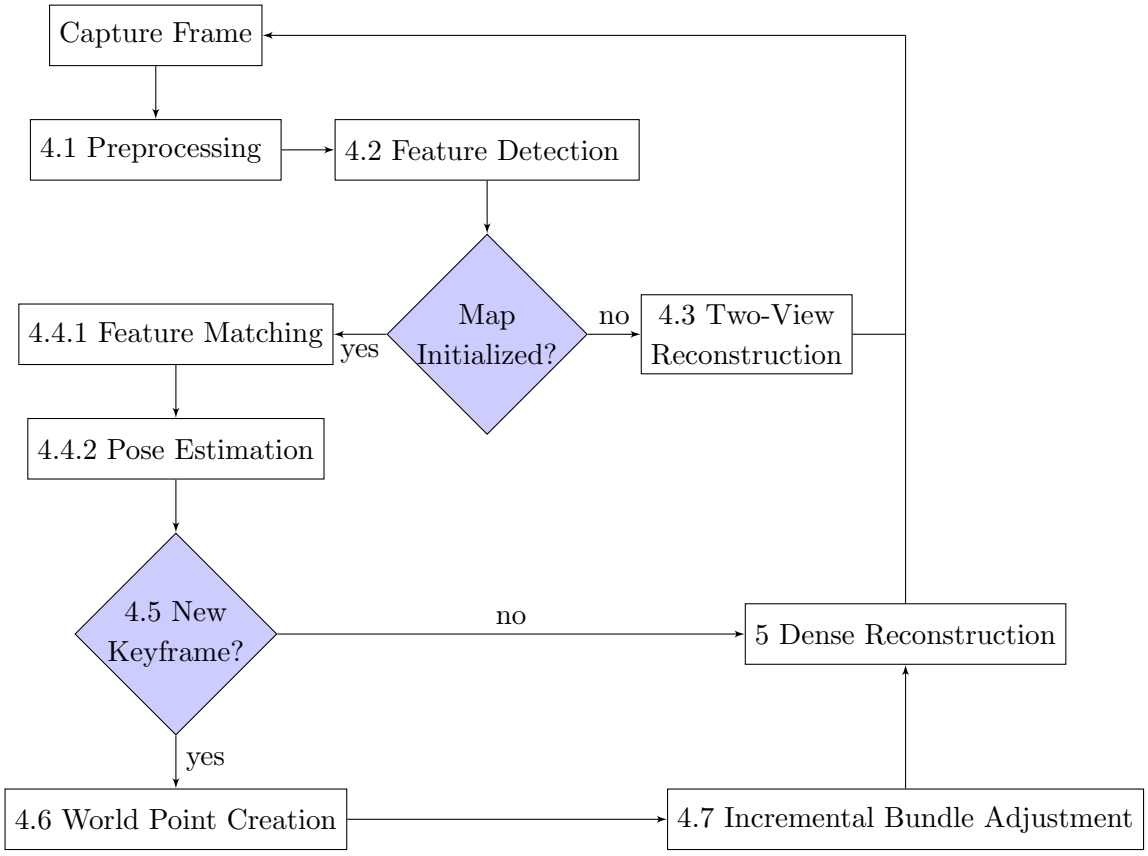

Figure 4.1: Overview of the camera tracking and sparse reconstruction system. The new frame is captured from the camera (top left) and processed in multiple stages to compute the camera pose and a sparse reconstruction of the scene. The information is then passed to the dense reconstruction (Chapter 5) for further processing and rendering.

4.1 Preprocessing

The images captured from the camera have to be preprocessed before they are passed to further stages. The exact preprocessing operations depend on the input format of the image and the required formats in sparse and dense reconstruction. Our complete reconstruction system runs in real time at a resolution of 960×540 pixels. The images provided by our test cameras were all larger than the target resolution. For example, the *Alexa Mini* streams images of 1920×1080 pixels. Therefore, they have to be downscaled by a factor of 2 in x- and y-direction. After scaling, the images are undistorted (Section 3.2) to simplify the camera projection model (Section 3.1) and allow the usage of highly distorted lenses. While undistorting, the skew parameter s is set to zero. As a last step, the images are converted to the correct binary format. Feature detection (Section 4.2) expects a 32-bit floating point grayscale image and the dense reconstructions operates on 8-bit RGBA images. The alpha channel does not contain any information, as it is only used to pad the 24-bit RGB pixels to a 32-bit alignment. This increases performance, because a texture fetch always results in exactly one memory transaction. All preprocessing operations described here are parallelized and implemented on the GPU. An overview of all steps and processing time of a 1920×1080 8-bit RGB input image is shown in Table 4.1.

Preprocessing Step	Time (ms)
Copy Host $\rightarrow$ Device	2.14
Scale $1920 \times 1080 \rightarrow 960 \times 540$	0.02
RGB $\rightarrow$ RGBA	0.02
Undistortion	1.07
RGBA $\rightarrow$ Gray	0.02
Total	3.28

Table 4.1: Preprocessing of an 1920×1080 8-bit RGB input image. All steps are implemented on the GPU and the timings were measured on a GTX 1080. Most of the time is spend in the host-to-device memory transfer and undistortion.

4.2 Feature Detection

Published in 2004, the Scale-Invariant Feature Transform (SIFT) still counts as one of the best feature detection algorithms for image matching and object recognition [46]. SIFT features are often used in offline reconstruction systems, because the extracted features are invariant to image scale and rotation, are highly distinctive, and a single feature can be matched with high probability to features from different images. SIFT was patented in the USA by the University of British Columbia [47], which restricts its use in commercial applications. In the following, a summary of the original paper [46] is given.

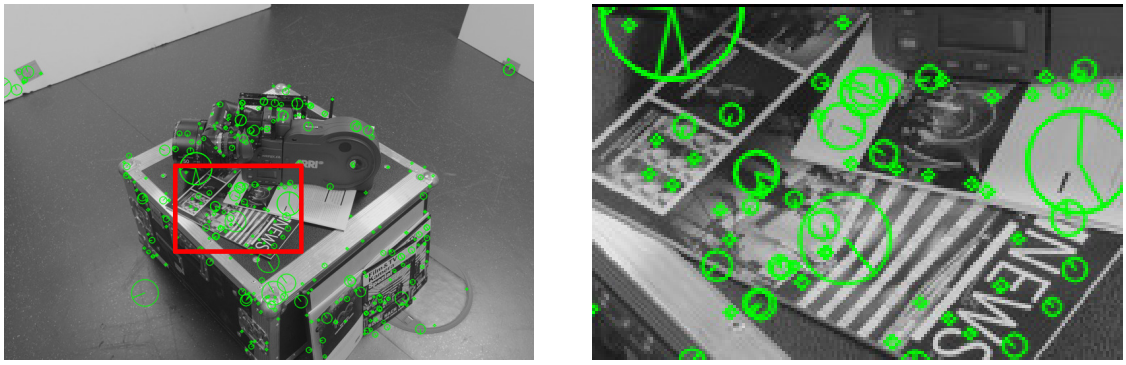

Figure 4.2: 344 SIFT keypoints were detected on a black and white image. The radius of the circle represents the scale and the line through the center shows the orientation of the keypoint.

4.2.1 Scale-Space Extrema

The first stage of keypoint detection is to identify locations and scales that can be repeatably assigned under differing views of the same object. Detecting locations that are invariant to scale-change of the image can be accomplished by searching for stable features across all possible scales, using a continuous function known as *scale space* [79]. The scale space of an image is defined as the function $L(x, y, \sigma)$, which is produced from the convolution of a variable-scale Gaussian $G(x, y, \sigma)$ with the input image $I(x, y)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

The most stable image features are produced by the maxima and minima of the Laplacian of Gaussian $\sigma^2\nabla^2 G$, compared to other possible image functions such as the gradient or Hessian [54]. For efficiency reasons, the difference-of-Gaussian function $D(x, y, \sigma)$ is used in keypoint detection, because it is a close approximation to the Laplacian of Gaussian and can be computed cheaply by subtracting nearby scales separated by a constant factor k.

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

In order to detect local extrema, each sample point is compared to its eight neighbors in the current image and nine neighbors in the scale above and below. If the value at the sample point is larger or smaller than all neighbors, it is selected as a candidate keypoint.

4.2.2 Keypoint Refinement

Once all candidate keypoints have been found, a 3D quadratic function is fit to the location, scale, and ratio of principal curvature to determine the interpolated extremum position $\hat{\boldsymbol{x}}$ [9]. In low contrast regions, for example a monochromatic wall, local extrema are found due to very small texture changes or image noise. These features cannot be detected reliably and are prone to give wrong matches. Therefore, a minimum contrast is enforced by rejecting all extrema with a difference-of-Gaussian value $D(\hat{\boldsymbol{x}})$ below a user defined *contrast threshold*.

Additionally, keypoints lying on edges are not stable enough for correct matches. Even though edge points exhibit a high contrast, they can shift along the edge due to noise-artifacts similar to how low contrast points can shift on a plane. Therefore, the *edgeness* e is computed by inspecting the ratio of principal curvatures $e = k_1/k_2$, with $k_1 > k_2$. If e exceeds a certain threshold, the keypoint is discarded.

4.2.3 Orientation Assignment

By assigning a consistent orientation to each keypoint based on local image properties, the keypoint descriptor can be represented relative to this orientation and therefore achieve invariance to image rotation. Given the scale of a keypoint, the Gaussian smoothed image L is selected and the gradient magnitude $m(x, y)$ and orientation $\theta(x, y)$ is computed for samples around the keypoint using central differences:

$$\Delta x = L(x+1, y) - L(x-1, y)$$

$$\Delta y = L(x, y+1) - L(x, y-1)$$

$$m(x, y) = \sqrt{\Delta x^2 + \Delta y^2}$$

$$\theta(x, y) = \tan^{-1} \frac{\Delta y}{\Delta x}$$

Each orientation sample is added to a histogram of 36 bins, where each bin corresponds to a 10° interval. The samples are weighted by their gradient magnitude and a Gaussian-weighted circular window to give more importance to samples that are close to the keypoint. The rotation angle with the highest peak in the histogram is stored in each keypoint. If more peaks are within 80% of the highest peak, additional keypoints are produced each with a different rotation assigned. Finally, a parabola is fit to the three histogram values around the peak to obtain an orientation with sub-bin accuracy.

4.2.4 Local Feature Descriptors

The local feature descriptors encapsulate all the data required to match different keypoints. As mentioned in the beginning, the descriptor is invariant to rotation, scale, and some amount of brightness and contrast change.

Similar to orientation assignment, the gradient magnitude and orientation is sampled in a small neighborhood (i.e. 16×16 pixels) of each keypoint in the smoothed image. The gradient orientations are computed relative to the orientation assigned in the previous step. The initial window is then subdivided into 4×4 regions. For each region, a Gaussian weighted gradient histogram with 8 bins is created and the histograms are concatenated into a single vector with 128 elements ($4 \times 4 \times 8$). It is important to avoid all boundary effects in which the descriptor abruptly changes as a sample shifts smoothly from being within one histogram to another or from one orientation bin to another. Therefore, trilinear interpolation is used to distribute the value of each gradient sample into adjacent histograms and bins. The feature vector is then modified to reduce the effects of illumination change. First, the vector is normalized to account for a change in contrast, because a constant factor multiplied on the gradient is canceled out by normalization. After that, the influence of large gradient magnitudes is reduced by clamping each value to 0.2 and renormalizing the vector.

4.2.5 GPU Implementation

The SIFT algorithm is well parallelizeable, which means that for large images the performance scales roughly linear with the number of available cores. We implemented SIFT for the GPU by porting each step of OpenCV's [21] CPU implementation to the graphics hardware. The timings of each step in the algorithm for an 960×540 image can be seen in Table 4.2.

Step	Time (ms)
1. Initial Blur	0.049
2. Gaussian Pyramid	0.235
3. DoG Pyramid	0.118
4. Keypoint extraction	0.195
5. Orientation assignment	0.231
6. Descriptor computation	0.219
Total	1.050

Table 4.2: Timings for each step of the SIFT algorithm on an 960×540 image. The input image with the 344 extracted keypoints can be seen in Figure 4.2.

Roughly one third of the time is spent for computing the Gaussian and difference-of-Gaussian pyramid. The time spent in orientation assignment and descriptor computation depends on the number of features extracted. In our test image (Figure 4.2) only 344 keypoints were found, because large parts of the image are covered with a low contrast background.

4.2.5.1 Gaussian Filter

Before SIFT keypoints can be detected on the image, the *scale space* has to be computed by applying Gaussian filters with different sigmas. For our input images of size 960×540 we use eight octaves and three layers per octave, giving us a total number of $8 \cdot (3 + 2) = 40$ scale space images to search the keypoints.

Figure 4.3: Two-pass Gaussian filter on the GPU.

The traditional way of computing the Gaussian convolution with an image on the GPU is to separate the filter and split the operation in two passes (see Figure 4.3): In the first pass, the source image is read, the row-wise convolution is computed, and the result is written to a temporary image. This temporary image is read in the second pass, the column-wise convolution is computed, and the result is written to the destination image [65]. Efficient implementations, for example, NVIDIA's library on image and signal processing *NPP* [62], partition the image into tiles, where each tile is convolved by a single thread block. This reduces global memory reads, because values of neighboring pixels can be transmitted through shared memory. The side effect is that the tiles have to overlap depending on the filter radius. Overlapping tiles introduce pixels that have to be read two or more times from memory. A general optimization technique is to increase tile size until the occupancy (limited by shared memory) falls below a critical value. In our tests with different convolution methods, we found that the performance peaks at around 75% occupancy.

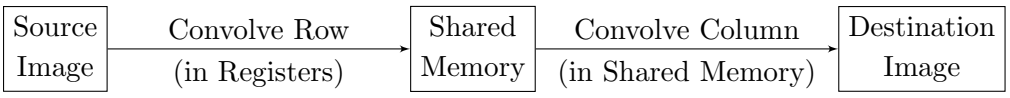

Figure 4.4: Our single-pass Gaussian filter on the GPU.

The disadvantage of two-pass methods is the requirement of writing the row-convolved image back to global memory. The upper bound of the performance is given by the memory bandwidth of the GPU. A single-pass convolution, which goes without the read and write to a temporary image, can theoretically be twice as fast as two-pass methods, assuming enough shared memory is present. With that idea in mind, we have implemented a single-pass convolution (see Figure 4.4): A rectangular tile is read into thread-local registers, the row-wise convolution is computed by shuffling the elements to neighboring threads, and the result is written to the destination image after a column-wise convolution is computed in shared memory. When comparing our method to the two-pass implementation of *NPP* [62], we found that for small filter radii our initial prediction was correct, showing that the single-pass convolution is roughly twice as fast as the two-pass method (see Figure 4.5). Only at large filter sizes, a two-pass convolution is still faster, because the number of overlapping pixels per tile grows faster on single-pass methods. The point when the two-pass convolution surpasses

our method is roughly at a radius of 12, corresponding to a filter window of 25×25 pixels. In this work, we use our single-pass convolution to create the SIFT scale space, because a filter radius of 4 is enough for stable feature detection.

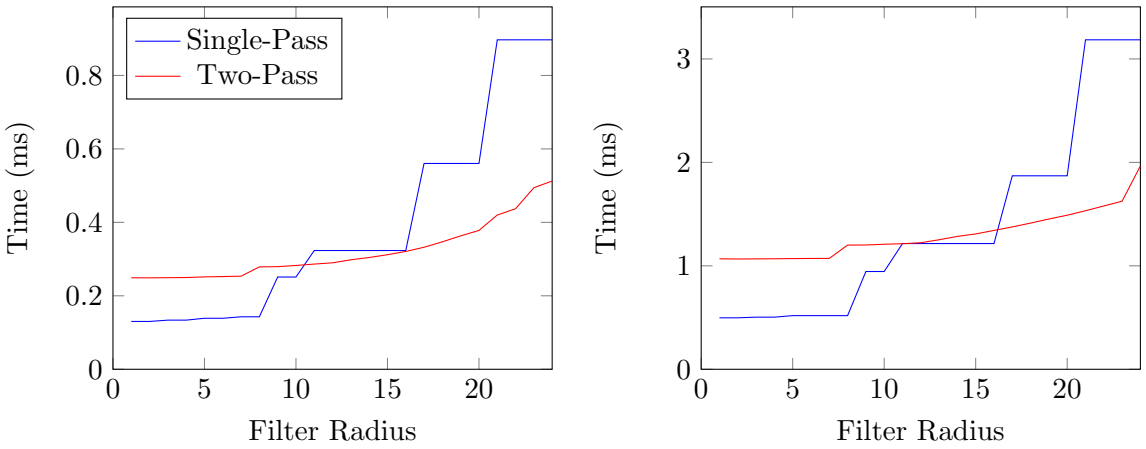

Figure 4.5: Single- and two-pass convolution time on a full HD image (left) and 4K image (right). For larger filter sizes, we implemented the single-pass method only when the radius is divisible by four, which results in a step-wise increase of the blue graph.

4.2.5.2 Keypoints and Descriptors

The keypoint extraction is one of the most difficult stages to implement efficiently on the GPU. When starting one thread per pixel that samples the neighborhood, each value in scale space is read 27 times from global memory. To make it even worse, the keypoint refinement (Section 4.2.2) is applied iteratively, allowing for keypoints to move to neighboring pixel locations when the computed delta of the quadratic fit exceeds 0.5. In this case, the warp diverges (assuming pixels close by are no extrema), because a single thread has to load additional values from global memory. We solve these problems by caching small three-dimensional tiles of the scale-space in shared memory. This reduces global memory traffic by a large margin, because the extrema detection can be performed on shared memory. When a keypoint is interpolated to a nearby pixel, the required values are most likely already present. Only when the interpolated point lies outside of the loaded tile, additional global memory reads have to be performed.

The remaining steps are implemented similar to the CPU version of OpenCV. The main difference is that we start multiple threads (at least one warp) for each keypoint to ensure that the sampling in scale-space is coalesced.

4.3 Two-View Reconstruction

At the start of the scan no knowledge of the scene is available. A bootstrapping phase is used to initialize the sparse reconstruction. Similar to MonoFusion [66], we use a two-view reconstruction, which is the minimal structure from motion problem. The goal is to find the relative pose (translation and rotation) between a camera pair and a set of 3D-points that project to each image. A robust two-view reconstruction method is required, because subsequents steps might fail, if the scene does not contain enough points or exhibits large reprojection errors. Figure 4.6 shows the result of the two-view reconstruction explained in this section.

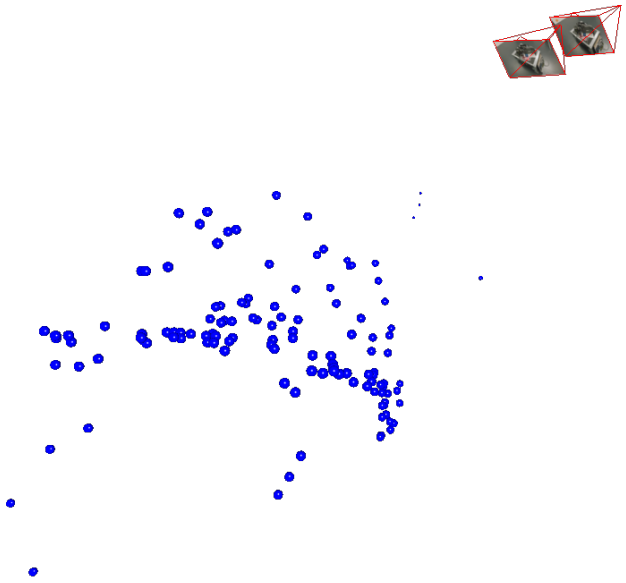

Figure 4.6: Sparse point cloud and relative camera transformation reconstructed from two images.

We use a classic two-view reconstruction method present in state of the art offline [23] and online [66] reconstruction systems. Candidate matches are computed between the SIFT features of the image pair and an essential matrix is estimated with the 5-point algorithm. The relative pose is extracted from the essential matrix and valid matches are triangulated (Section 3.6) to obtain the corresponding 3D world points. As a last step of the reconstruction, bundle adjustment (Section 3.4) is ran to minimize the reprojection error. The quality of the current reconstruction is evaluated by inspecting the number of good feature matches and the median triangulation angle between the camera positions and world points. The current two-view reconstruction is discarded, if the number of good matches is fewer than 80 and the median triangulation angle is smaller than 4°. The latter condition ensures a scale and rotation independent minimum base line, which is required for stable triangulation [33]. In the following, two-view feature matching and relative pose estimation is described in detail.

4.3.1 Two-View Feature Matching

Given two sets of SIFT feature descriptors, each belonging to the keypoints of an image, the goal is to find corresponding points belonging to the same 3D world point. As described in Section 4.2, a SIFT feature descriptor is a 128 element vector. The Euclidean distance between two feature vectors describes the similarity between the associated keypoints. Therefore, matching is performed with a nearest neighbor (NN) search in 128-dimensional space.

In addition to classifying false matches by thresholding the distance to the nearest neighbor, we use the ratio between the nearest and the second nearest neighbor as described in [47] and used by [73]: A match is accepted, if $\frac{d_1}{d_2} < 0.7$, where d_1 and d_2 are the distances of the two nearest neighbors. In Figure 4.7, an image pair with 71 matched SIFT keypoints is shown.

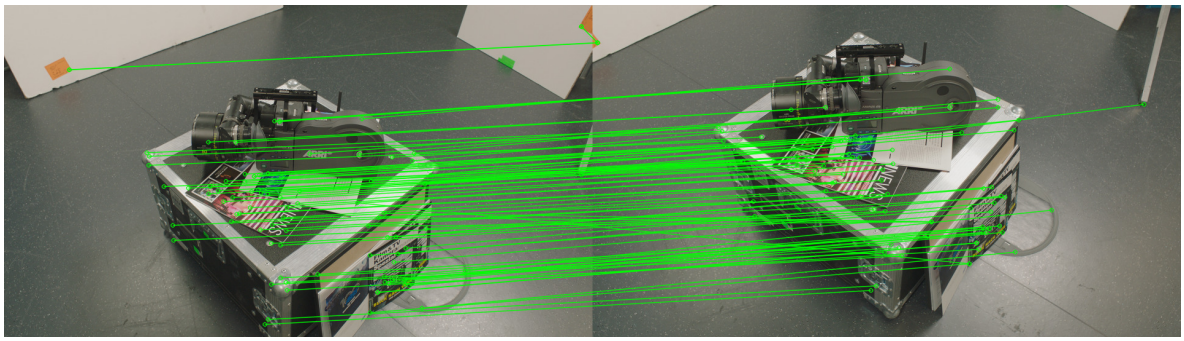

Figure 4.7: Two-view feature matching on images captured from different view points. The 71 matches shown here all pass the ratio test, but a few outliers still remain.

A brute force implementation of the k-NN search, with k being the number of desired nearest neighbors, of n points in one image and m points in the other image has a complexity of $O(k{\cdot}n{\cdot}m)$. Tree-like acceleration structures reduce the best-case complexity to $O(k{\cdot}n{\cdot}\log(m))$. One of these acceleration structures is the multidimensional binary search tree or *kd-tree* [6]. Even though kd-trees are commonly used for NN-searches in 2D and 3D, they do not perform well in high dimensional spaces such as the SIFT descriptor space [5]. Bentley suggests as a good rule of thumb to use k-d trees only if $n > 2^{2k}$ [6]. In the case of SIFT descriptors, this would mean that at least 2^{256} keypoints are required.

To overcome the problem of dimensionality, approximated nearest neighbor search (ANN) was introduced [5]. ANN samples multiple different k-d trees and terminates early after a fixed number of checks. As a result of the early termination, the actual nearest neighbor is not found every time. Nevertheless, ANN search is widely used in offline reconstruction systems such as MVE and Bundler [23] [73].

In a real-time environment, the construction and traversal of the k-d trees used in the ANN search is not feasible, because the processing time exceeds the 100 ms mark and parallelization is non-trivial. Fortunately, our input images are small and the number of feature points is usually fewer than 1000. At such magnitude, a brute-force search takes less than 1 ms when implemented on a GPU.

Our GPU implementation of the NN search starts by computing a distance matrix, where each entry (i, j) gives the distance in L_2-norm between descriptor i in the first image and descriptor j in the second image. For efficient computation, the distance matrix is divided into tiles of 32×32 elements and each tile is computed by a single thread block in shared memory. After the distance matrix is built, the k nearest neighbors can be found by inspecting each row of the matrix. The simple approach of sorting every row and selecting the first k elements works, but is not efficient because k is usually small (< 4). Instead, we sweep over each row and keep track of the k smallest values. For further parallelization, we use multiple threads per row, where each thread computes the k smallest elements of its subset and merges the local lists with shuffle instructions. An overview of our measurements for different match sizes and GPUs can be seen in Table 4.3.

Match Size	264×281	344×351	1123×1169
GTX 970	0.065	0.089	0.829
GTX 1080	0.032	0.044	0.384

Table 4.3: Total time in ms to match SIFT features between two images. The required time increases quadratically with the number of descriptors.

4.3.2 Two-View Relative Pose Estimation

Given a set of feature matches obtained through a nearest neighbor search, the relative transformation between both camera poses is computed. This transformation can only be computed up to scale, therefore the final reconstruction will be *Euclidean* instead of *metric* [33]. The geometry of two calibrated views is captured in the 3×3 essential matrix $\boldsymbol{E}$. Each correct match, consisting of a point from each image $\boldsymbol{x}_0, \boldsymbol{x}_1$, satisfies the epipolar constraint:

$$\boldsymbol{x}_1^T \boldsymbol{E} \boldsymbol{x}_0 = 0 \tag{4.1}$$

If at least 5 matches are known, up to 10 candidate essential matrices can be computed with Nister's implementation of the 5-point algorithm [61]. The amount of potential solutions depends on the number of real roots when solving a 10th degree polynomial. Each candidate essential matrix is then decomposed into two rotation matrices and translation vectors, giving a total number of 4 possible transformations per essential matrix. For each relative transformation, the five initial points are triangulated (Section 3.6) and projected to each image. Only one solution over all essential matrices and pose configurations is geometrically valid, which means that all 3D points lie in front of both cameras. All other configurations have at least one point that is behind one or both cameras.

The 5-point algorithm is the minimal algorithm for the two-view relative pose problem. If one of the feature matches is incorrect, the algorithm still produces an essential matrix that satisfies (4.1) and passes the projection test described in the previous paragraph. These wrong configurations are identified by wrapping the algorithm in a RANSAC procedure [19].

Potential matches are classified as inliers, if the normalized epipolar constraint is smaller than a constant term

$$|\boldsymbol{x}_1^T \frac{\boldsymbol{E}\boldsymbol{x}_0}{|\boldsymbol{E}\boldsymbol{x}_0|}| < \frac{2\epsilon}{f_x + f_y},$$

with ϵ being a user defined value in pixels that approximates the allowed distance of $\boldsymbol{x}_1$ to the epipolar line $\boldsymbol{E}\boldsymbol{x}_0$. We use $\epsilon = 2$ to aggressively filter out wrong matches. If more image noise is expected by the cameras or the resolution changes, ϵ has to be adapted accordingly. As a result, we obtain the relative pose between the two views and a set of matches that satisfy the epipolar constraint (4.1). These matches are triangulated to finalize the two-view reconstruction and it is proceeded as described in Section 4.3.

4.4 Incremental Pose Estimation

In the last section, the initialization process via two-view reconstruction has been described. For every further image, the relative pose of the camera to the current scene is computed (see Figure 4.8). As a first step, 2D image features are matched to already reconstructed 3D points (Section 4.4.1). Next, the new pose of the camera is computed from these matches (Section 4.4.2).

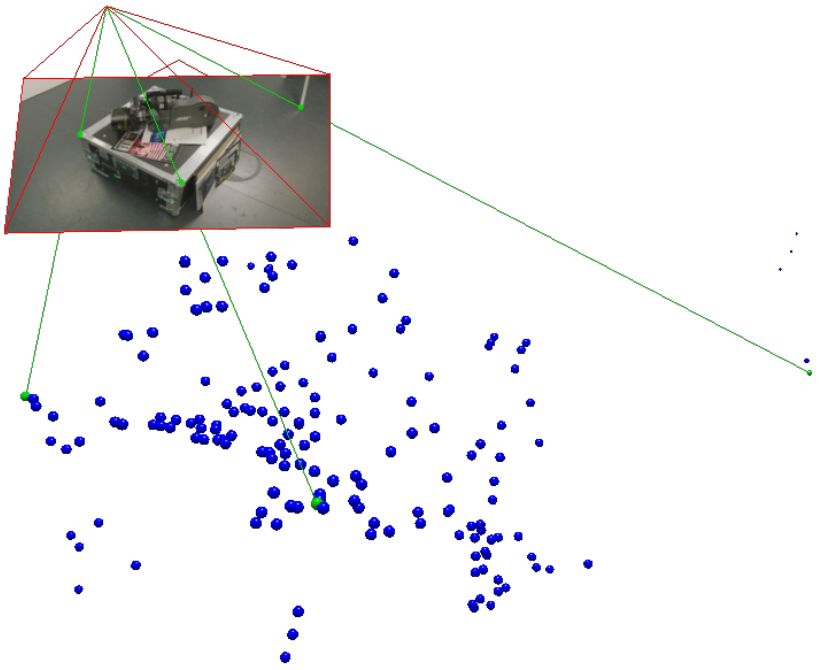

Figure 4.8: Given a sparse scene representation (blue points), the camera pose (red lines) can be computed with a few image-to-world feature matches (green lines). Three correct matches are sufficient (see Section 3.5), but if more matches are available, the pose can be estimated more robustly.

4.4.1 Image-to-World Feature Matching

Matching the previously reconstructed 3D points to the feature points of the current image is similar to two-view feature matching (Section 4.3.1). It is not directly clear which feature descriptor should be used for each 3D point, because every point is seen by at least two cameras and therefore can be described by more than one SIFT descriptor. Averaging all SIFT descriptors for a point did not work, because too many correct matches where filtered out by the ratio test (see Section 4.3.1). Instead, we use the feature descriptor of the last image that observed the 3D point. This allows our scene to adapt to new viewing angles and increases the matching behavior, because SIFT feature points are only marginally invariant to perspective transformation.

As the scene grows, more and more world points have to be matched against the current image. This increases processing time until a real-time reconstruction is not possible anymore. Furthermore, fewer good matches can be found, because more correct matches are filtered out by the ratio test (see Section 4.3.1). To tackle both problems, we guess the new camera pose, project all world points to the estimated pose, and match the image features only to points, which project into a radius of 75 pixels. As a pose guess, we take the previous camera transformation without further heuristics, such as a decaying velocity model [40].

4.4.2 Pose Estimation

From the previous step, we have obtained a set of 2D to 3D correspondences. We use the perspective three-point problem (Section 3.5) to directly compute the transformation of the camera in relation to the scene. As in the two-view relative pose estimation (Section 4.3.2), the feature points exhibit image noise and not all given matches are correct. The P3P algorithm does not produce correct results in both cases. Therefore, we handle outliers by wrapping the procedure in a RANSAC algorithm [19] that classifies points as inliers, if the introduced reprojection error is smaller than 2 pixels. To account for noisy inputs, nonlinear optimization (see Section 3.4) is performed before the inlier check is executed. The best pose is then optimized a second time with all previously detected inliers. In our experiments, we observed that tracking becomes unstable if fewer than 20 inliers remain after the RANSAC inlier check. In such an unstable case, the computed camera pose *jumps* unrealistically from frame to frame even when the operator does not move the camera. This phenomenon can be explained by small errors in previous keyframe poses, keypoint localization, and triangulation.

4.5 Keyframe Selection

When the camera is moved through the environment, unseen parts of the scene become visible and previously seen parts disappear. A stable reconstruction has to *grow* the internal scene by adding data of the newly seen geometry. We use a keyframe-based approach, which can also be found in other real-time reconstructions systems such as [66] and [15]. Some frames are marked as keyframes after pose estimation. For every new keyframe, additional points are triangulated and added to the scene (see Section 4.6).

The selection process of deciding which frames should be marked as keyframes is a vital part of the reconstruction. Selecting too many, rapidly grows the complexity of the scene. Selecting too few degrades the tracking, because not enough world points are reconstructed. In MonoFusion [66], new keyframes are generated if the number of tracked points falls below a certain threshold. This method tries to ensure that always a certain tracking quality is met and works well when the camera is quickly moved to new parts of the scene. A disadvantage is that keyframes might be generated with very small baselines, for example, when the camera is only rotated. In our system we use the following constrains for new keyframes:

1. Number of matches > 30

2. Triangulation angle to last keyframe $> 4°$

The first constraint ensures that the tracking is stable enough, such that the pose of the keyframe has been estimated correctly (see Section 4.4.2). The second constraint sets a lower bound to the relative camera movement as it was done in the two-view reconstruction (see Section 4.3).

4.6 World Point Creation

When a new frame is marked as a keyframe (see Section 4.5), new 3D points of previously unseen parts of the geometry are added to the scene. This is required for stable tracking, because if not enough 3D points can be matched to the image, a precise pose estimation is not possible (see Section 4.4.2). To generate these new world points, we use two-view reconstructions between the new keyframe and the last two keyframes. In contrast to Section 4.3, no essential matrix has to be computed, because the relative pose between the keyframes is already known.

After matching the SIFT feature points for both images, we eliminate wrong matches with the ratio test and by thresholding the distance to the epipolar line. The remaining matches are correct with a high probability and are added to the scene by triangulation (see Section 3.6). For features which are far away, the triangulation angle is small and a precise depth estimation is not possible. Therefore, reconstructed points with a triangulation angle below $2°$ are discarded.

4.7 Incremental Bundle Adjustment

Every time a new keyframe is created, the complexity of the scene grows. Each new world point and every new camera adds additional unknowns in bundle adjustment (Section 3.4). This increases the required time of the nonlinear optimization until a real-time reconstruction is not possible anymore. In our tests, we observed that the processing time exceeded 100 ms after 40 keyframes and around 5000 points were added to the scene.

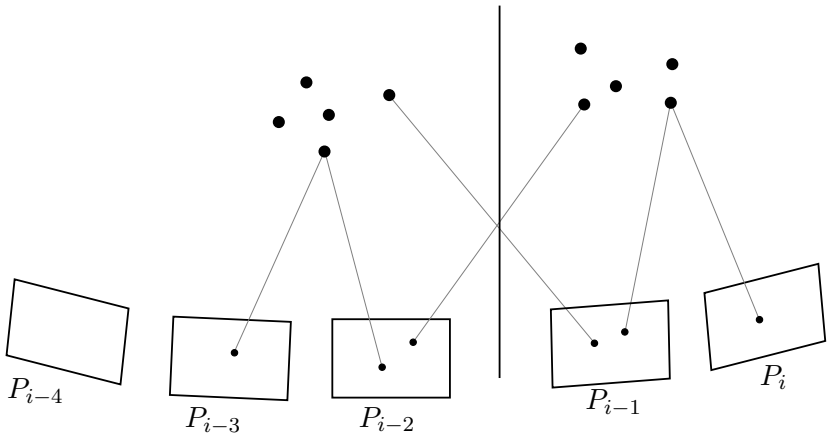

Figure 4.9: The scene is semantically divided into an old part (left) and a new part (right). Only *new* camera poses and 3D points are optimized in local bundle adjustment. All old parameters are kept constant during the optimization.

Incremental bundle adjustment [56] optimizes only the last few frames, whereas old camera poses and old 3D points are held constant (see Figure 4.9). If both the 3D point and the camera are constant, the reprojection error does not have to be evaluated in the optimization. As a result, the Jacobian Matrix size stays roughly constant and the processing time does not increase with larger scenes. Mathematically, two additional terms are added to the original bundle adjustment equation (4.2):

$$\min_{P,X} \sum_{i}^{n} \sum_{j} ||\text{project}(P^i, X_j) - x_j^i||_2^2 \tag{4.2}$$

$$\min_{P} \sum_{i}^{n} \sum_{j} ||\text{project}(P^i, X_j) - x_j^i||_2^2 \tag{4.3}$$

$$\min_{X} \sum_{i}^{N} \sum_{j} ||\text{project}(P^i, X_j) - x_j^i||_2^2 \tag{4.4}$$

The first new term (4.3) only minimizes the camera pose P^i of the last n keyframes, while the world points X_j are kept constant. In the second new term (4.4), the camera pose is constant and only the world points are optimized.

Every 2D to 3D correspondence has to be either included in one of the three equations above or discarded. This decision is made by classifying each 3D point and each keyframe either as *old* or *new*. A keyframe is old, if it has been created more than n keyframes before the current frame. A world point is old, if it has been added more than N keyframes before the current frame. For our real-time reconstruction system, we chose $n = 5$ and $N = 10$ as a trade-off between precision and performance. Figure 4.10 shows the difference in processing time and reprojection error for global and local bundle adjustment. Global bundle adjustment (blue line) gives the best results in terms of reprojection error, but the required time increases linearly with the number of keyframes. The processing time of incremental bundle adjustment (red line) stays constant at around 11 ms over the complete scan and results only in a slightly worse reprojection error. If no optimization is used at all (black line), a larger error is visible.

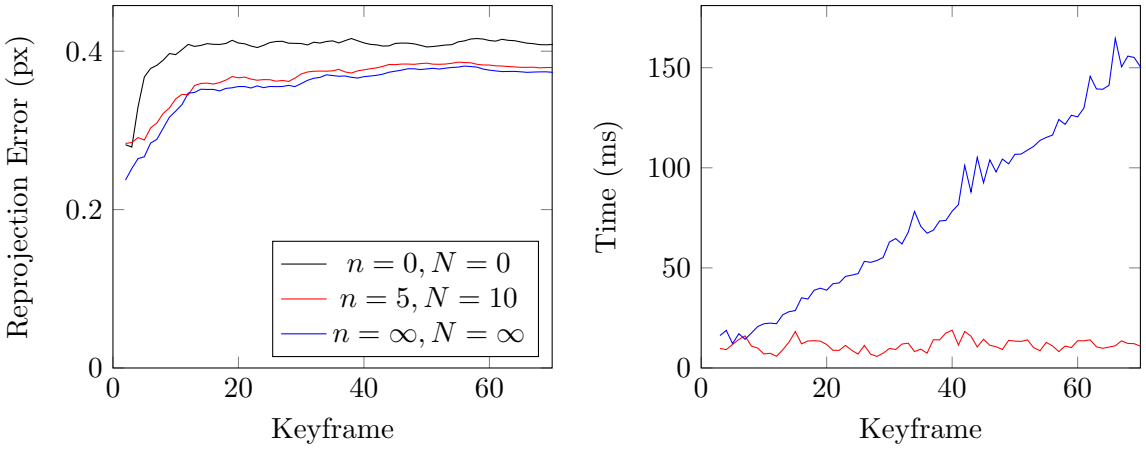

Figure 4.10: Average reprojection error and processing time for different optimization methods over a scan with 70 keyframes.

Chapter 5

Dense Reconstruction

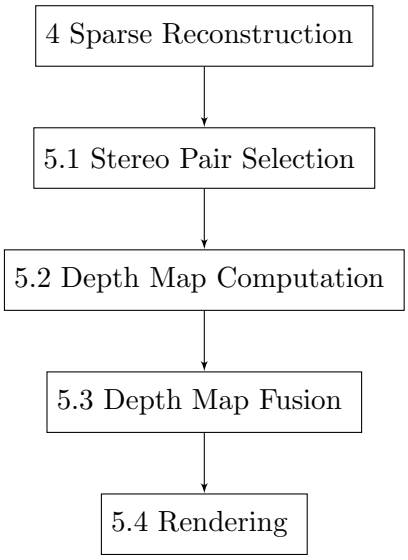

Figure 5.1: Overview of the dense reconstruction. A depth map is computed for every frame and fused into a signed distance field (SDF) representation of the scene. The SDF is then ray casted to create the preview of the reconstruction.

The dense reconstruction is executed after the pose computation for every frame (see Figure 4.1). It takes as input the pose and RGB-image of the new frame and all previous keyframes. As output, the dense reconstruction produces a rendered preview image of the scene from the current view point of the camera. In difference to [63], this process does not influence the tracking and can be disabled, if no preview is required. An overview of all steps can be seen in Figure 5.1 and Table 5.1. First, one of the keyframes is selected and used to compute a depth map with two-view stereo matching. The depth map is then fused with all previous depth maps using a signed distance field (SDF). As a last step, the SDF is ray casted from the current view point to extract the surface and generate a preview of the scene.

Step	Time (ms)
1. Select Stereo Pair	0.02
2. Depth Map Computation	15.49
3. Depth Map Fusion	2.86
4. Rendering	11.88
Total	30.25

Table 5.1: Processing times of the different dense reconstruction steps for a 960×540 image on a GTX 1080.

5.1 Stereo Pair Selection

Every new frame, which is passed to the dense reconstruction, is assigned to one keyframe for depth map computation. Just like in two-view reconstruction (Section 4.3), features between the images should allow for easy matching and the baseline should be large enough for stable triangulation. These two properties are in contrast with each other, because a larger baseline hinders matching and a small baseline introduces large triangulation errors [33].

Following the work of [66], one approach is to group all keyframes that are within a baseline B of the new camera pose. From this group, a histogram is plotted with the ratio of all 3D points originally detected in each keyframe to those that can be successfully matched in the new frame. Ultimately, the farthest keyframe with a score above 0.65 is selected for stereo matching.

We use a slightly different approach in our system: The triangulation angle is computed between the new frame and the latest keyframe. If this angle is above $4°$ and more than 30 common world matches are present, the keyframe is selected for stereo matching. Otherwise, the process is repeated with the next older keyframe. This method ensures a sufficiently scale-independent baseline (see Section 4.3) and also selects recent keyframes, which are likely to have overlapping parts in the images.

5.2 Stereo Depth Map Computation

After selecting a previous keyframe for the current frame, a depth map is computed between these images with stereo matching. Existing stereo matching algorithms can usually be classified as local or global methods. Global approaches formulate an energy function that is minimized by taking all image pixels into account. Local methods identify corresponding pixels only based on the correlation of local image patches. Between both worlds, approximative global methods minimize a global energy function with greedy or discrete approaches, such as dynamic programming [69] [36].

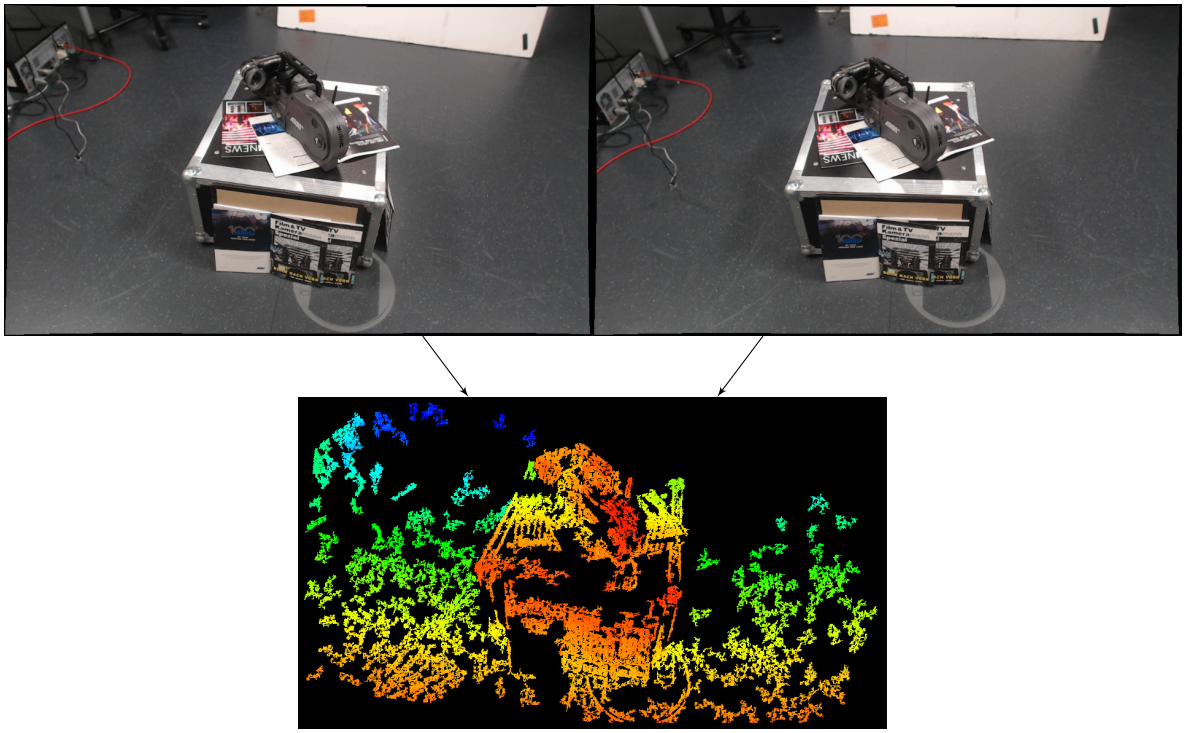

Figure 5.2: Depth map computed from two input images with stereo matching. Red pixels indicate small depth values (close to the camera) and blue pixels indicate large depth values.

In our real-time reconstruction system, we use a local patch-based method that directly estimates depth values for every pixel without having to rectify the images [44]. Figure 5.2 shows the depth map computed by our stereo matching algorithm. In the next section, the mathematical background of depth reprojection and matching cost is presented. After that, the stereo matching algorithm is described, which is based on MonoFusion [66] and consists of the following steps:

1. Random initialization of the depth map (Section 5.2.2).
2. Propagate low-cost depth values to nearby pixels (Section 5.2.3).
3. Invalidate outliers by thresholding and segmentation (Section 5.2.4).

5.2.1 Matching Cost

Given a depth estimate at a specific image point, a cost function is defined that describes the quality of the estimated depth value. Good estimates (correct depth values) should yield low costs and bad estimates (wrong depth values) should yield high costs. A small patch in the first image around the 2D point $\boldsymbol{p}_{i1}$, augmented with a depth value d, is projected to the corresponding 2D point $\boldsymbol{p}_{i2}$ in the second image. For a calibrated camera, this transformation can be described with a single 3×4 matrix $\boldsymbol{T}$:

$$\boldsymbol{p}_{i2} = \boldsymbol{T}\boldsymbol{p}_{i1}$$

$$\boldsymbol{T} = \boldsymbol{K}\,\boldsymbol{V}_2\,\boldsymbol{V}_1^{-1}\boldsymbol{K}^{-1},$$

where $\boldsymbol{K}$ is the intrinsic matrix and $\boldsymbol{V}$ the view matrix (see Section 3.1). The RGB intensities at each pixel of the patches are extracted and compared with the *zero-mean normalized cross correlation* (ZNCC) [25]:

$$C = \frac{1}{N} \sum_{\boldsymbol{p} \in \Omega} \frac{(f(\boldsymbol{p}) - \bar{f})(g(\boldsymbol{p}) - \bar{g})}{\sigma_f \sigma_g} \tag{5.1}$$

The patches are given by the set Ω. The intensity of a point $\boldsymbol{p}$ in image f is $f(\boldsymbol{p})$. The mean and standard deviation for the first image is $\bar{f}$ and σ_f. Equation 5.1 handles only single channel images. The ZNCC score for RGB images can be computed by either concatenating each channel, resulting in larger sums or computing the similarity measures for each channel independently and taking the average. We compute the ZNCC score by taking the average of each channel, as it was suggested by [25].

5.2.2 Random Initialization

The depth map computation starts by estimating the valid depth range $[d_{min}, d_{max}]$, which contains all interesting parts of the image. If no previous knowledge is available, this valid depth range cannot be computed automatically and has to be given by the user. MonoFusion [66] estimates this range in the bootstrapping phase and leaves it constant over the reconstruction. We use the sparse feature matches obtained in the tracking stage (Section 4.4.1) to compute the valid depth range for each stereo pair individually. This dynamically changing range should be more stable than MonoFusion's approach, if the distance of the camera to the scene changes over time. Another advantage of having a sparse set of feature matches is that for some pixels the depth value is already known. We use this knowledge to initialize 2×2 patches of the depth map around each feature match.

For all remaining uninitialized pixels, k random depth values are computed by uniformly sampling the valid depth range $[d_{min}, d_{max}]$. The depth value that produces the lowest ZNCC cost is chosen. Increasing the amount of samples k increases the depth map quality after the initialization step (see Figure 5.3), but also increases the computation time. In our real-time system, we use only one random sample per pixel $k = 1$, because small depth errors are compensated over multiple frames in later stages of the dense reconstruction (see Section 5.3).

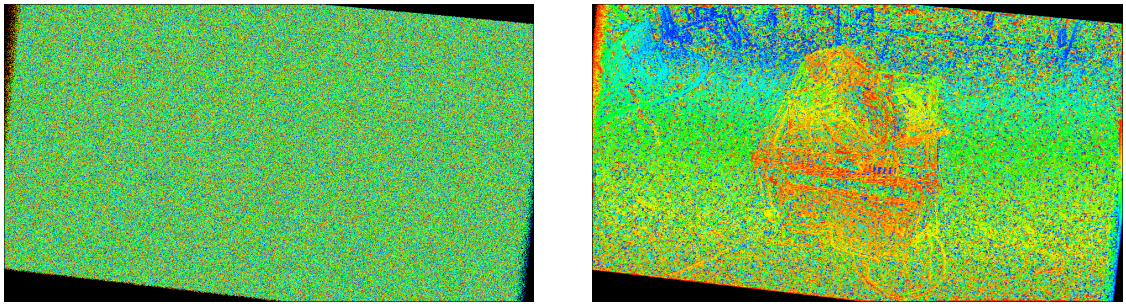

Figure 5.3: Random depth map initialization for a different number of samples. Left: 1 sample. Right: 10 samples.

The general idea behind the random initialization is that we model the world as piecewise planar faces with constant depth. The probability that at least one depth value in that region was estimated correctly increases with region size and number of sampled depth values. For example, given a 16×16 region with constant ground truth depth, we would like to reconstruct that depth in a range of up to two meters with a precision of 5 mm. The total number of possible depth values is $l = 2000mm/5mm = 400$. If we take $k = 5$ samples for every pixel, the region contains at least one correct depth sample with a probability of 95% [66].

5.2.3 Spatial Propagation

In the previous section, it was shown that for a given region of constant depth at least one pixel has been guessed correctly with high probability. These correct values are propagated to its neighboring pixels, so that every pixel in the region has the same, correct depth value. In an iterative spatial propagation scheme, each pixel reads the depth estimates of its neighbors and choses the value that produces the lowest cost according to the ZNCC score. When the total cost cannot be reduced by propagating depth estimates anymore, this process has converged and spatial propagation is finished.

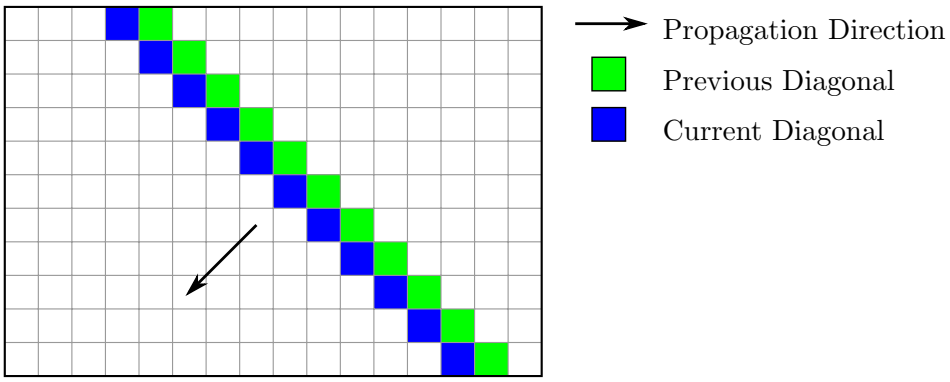

Figure 5.4: Each pixel in the current diagonal (blue) reads the depth of the right and upper neighbor (green). The depth producing the minimal ZNCC score is selected and stored in the current pixel.

In our system, we use an approximative version of spatial propagation. The image is processed diagonal by diagonal so that each pixel only reads the neighbors that were part of the previous diagonal (see Figure 5.4). Correct depth values are propagated quickly through regions of constant depth and we terminate the iteration after two sweeps in opposite directions, because further sweeps only marginally decrease the overall cost (see Figure 5.5).

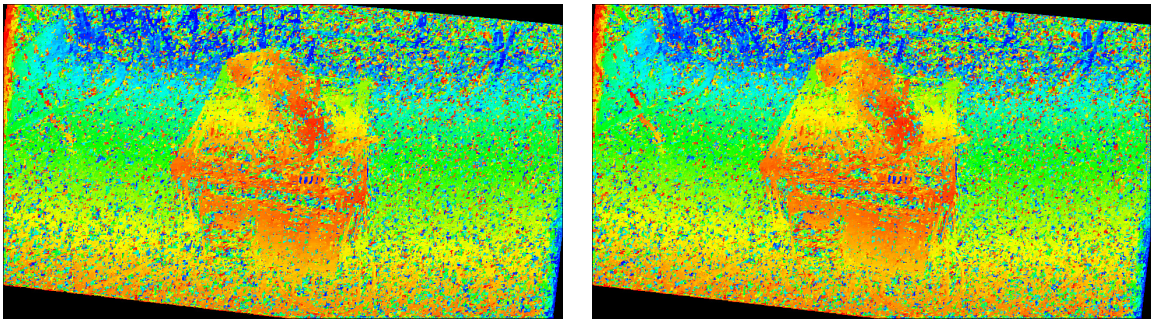

Figure 5.5: Spatial propagation on a random initialized depth image. The left image shows the result after the diagonal down sweep and the right image after the up sweep.

5.2.4 Outlier Removal

In the right image of Figure 5.5, it can be seen that the spatial propagation does not produce correct depth values for every pixel. Those outliers usually appear in small patches of similar depth, which are different to its surrounding pixels. A simple and efficient way to remove incorrect matches is to invalidate all depth values that produce costs over a fixed threshold t. In this work, we use a threshold of $t = 0.5$ for the ZNCC similarity measure. Additionally, all pixels are invalidated where either the pixel itself or the projection to the second image is fewer than 10 pixels away from the border. The depth image after thresholding and border removal can be seen in Figure 5.6.

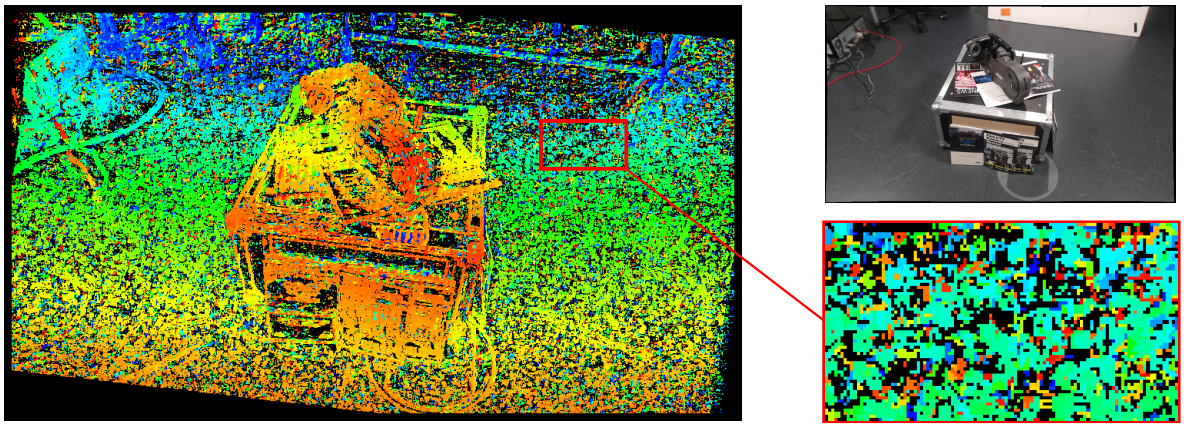

Figure 5.6: Invalidation of all depth values where either the ZNCC cost exceeds a threshold or the pixel is too close to an edge.

Some incorrect depth values in Figure 5.6 produce low costs and pass the previous filtering. This can be explained by image noise, errors in camera alignment, occlusion problems, and the random initialization. Especially in textureless regions (the floor in Figure 5.6) many outliers remain, because patches produce low matching costs, even though they triangulate to incorrect 3D positions.

Fortunately, these wrong matches occur for individual pixels or very small regions. When a large region of similar (or equal) depth is found, it is most likely correct. These large connected regions of similar depth can be found by image segmentation with region growing. A naive

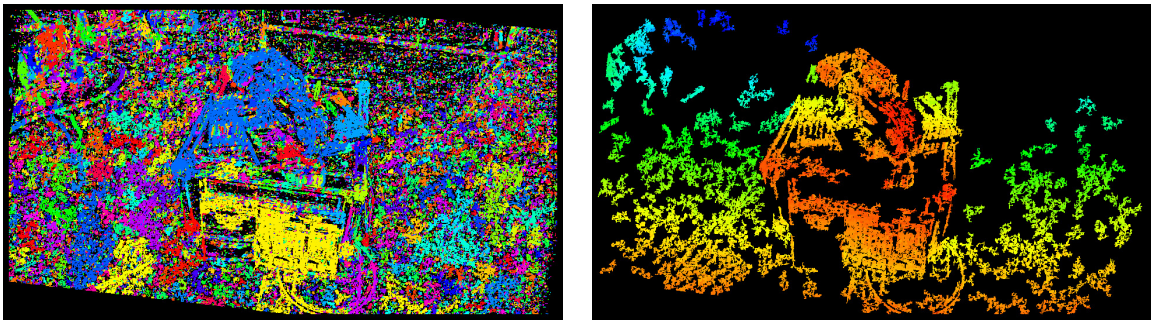

Figure 5.7: Outlier removal with image segmentation. Left: Segmentation displayed by assigning a random color to each region. Right: Final depth map after removing all regions with fewer than $r_{min} = 100$ elements.

implementation of region growing starts by assigning a unique *region id* to every pixel and merges neighboring regions, if the depth difference falls below a given threshold. The algorithm terminates when no more regions can be merged. Ultimately, all pixels belonging to a region with fewer than $r_{min} = 100$ elements are invalidated. Figure 5.7 shows the image segmentation and the final depth map after small regions have been removed. No visible outliers remain in the depth map.

5.2.5 GPU Implementation

Our multithreaded CPU implementation of the stereo matching algorithm, described in the previous sections, takes around 200 ms per frame. As this is not fast enough for our real-time application, an efficient GPU implementation is required.

One bottleneck in all steps is the ZNCC cost computation for a given pixel and depth value. The naive implementation of starting one thread per pixel is expensive, because the thread has to loop over all pixels in the patch and sample the projected pixels in the second image. This results in uncoalesced global memory reads, because when neighboring pixels have different depth values, the projected pixels are at different locations. Our solution is to start 8 threads per pixel that can load the complete region Ω to shared memory in one or two cycles. The sum of the ZNCC cost is then computed in parallel with a shuffle-based reduction.

The spatial propagation, described in Section 5.2.3, requires global synchronization every time a diagonal is processed, because the read data depends on the previous diagonal. We solve this problem by borrowing a technique used in parallel scan implementations [80]: Blocks are assigned to work elements based on their scheduling order and intra-block synchronization is performed through atomic operations in global memory. This ensures that the new diagonal is processed as soon as the previous one is finished and in total only one kernel has to be launched.

The region growing algorithm, described in Section 5.2.4, assigns a unique region id to every pixel and merges neighboring regions, if the depth difference falls below a given threshold. We implement this on the GPU with a divide and conquer approach: The complete image is recursively divided into smaller tiles until each tile is of size 1×1. For those minimal tiles, the initial segmentation of assigning each pixel to a unique region id is the only possibility and therefore correct by definition. In the conquer step, neighboring tiles are merged until only one tile containing the complete image is left (see Figure 5.8).

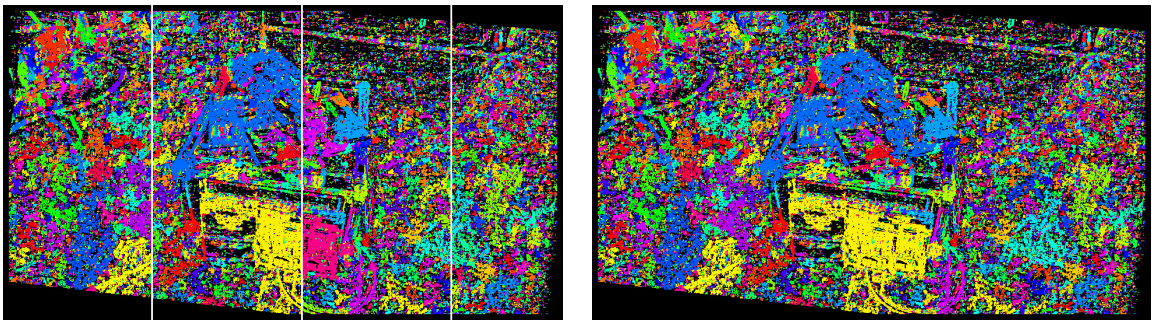

Figure 5.8: The segmentation is already computed on four vertical tiles (left image). A merging process combines the remaining tiles until only one tile covering the complete image remains (right image).

In Table 5.2, an overview of the processing time is given for all steps in the depth map computation. The total required time for one frame is 15.49 ms. Most of the time is spent for the initialization and spatial propagation.

Step	Time (ms)
1. Initialization	5.12
3. Spatial Propagation	7.14
4. Cost Filtering	0.12
5. Region Segmentation	3.41
Total	15.49

Table 5.2: Processing times of different steps in depth map computation for a 960×540 image, measured on the GTX 1080.

5.3 Depth Map Fusion

A depth map is computed for every new frame (see Section 5.2). These incoming depth maps are fused into a signed distance field (SDF) representation of the scene. The SDF is stored in a regular voxel grid, where each voxel contains the signed distance to the surface. The surface is represented with distance values of zeros, free space as positive values and occupied space as negative values [58]. Truncated signed distance fields (TSDF) store the distance only in voxels close to the surface. All other voxels contain an invalid value. Figure 5.9 shows a TSDF in two dimensions.

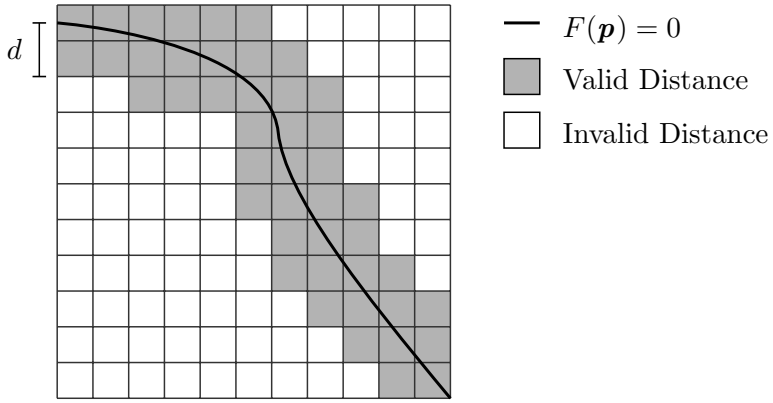

Figure 5.9: A truncated signed distance field on a regular grid. Voxels close by the surface $F(\boldsymbol{p}) = 0$ contain a valid distance value. Voxels further away than the truncation distance d are displayed in white and do not contain a valid distance value.

In three dimensional scenes, only a few voxels contain distance data. Most of the voxels will contain an invalid value, because the distance to the closest surface is larger than the truncation distance. We use Nießner's VoxelHashing approach [59] that allocates memory only for voxels close to the surface by mapping them to a one dimensional array using a hash function. For efficiency reasons, the hashing and allocation is performed on *voxel blocks*, each consisting of $8 \times 8 \times 8$ voxels. For more details and implementation hints, the reader is referred to the original paper [59]. The reference implementation is available on GitHub [60].

The depth map for each frame has to be integrated into the existing TSDF. For that purpose, every depth value is projected into the scene and voxels closer than the truncation distance are selected. For each selected voxel, the surface distance and a per-pixel interpolation weight is computed. The per-pixel weight W_R is proportional to $\cos(\Theta)/R_k$, where R_k is the depth value and Θ the angle between the associated pixel ray direction an the estimated surface normal [58]. The normal for each depth sample is estimated by the difference from neighboring samples.

In addition to a weight for every depth map element, a weight W_k is stored for every voxel, which is proportional to the uncertainty of surface measurement. A large voxel weight indicates that the stored distance is with a high probability close to the ground truth distance. Given the depth value weight W_R and the voxel weight W_k, each voxel can be updated by linear interpolation between the old and the new value [12]:

$$F_k = \frac{W_{k-1}F_{k-1} + W_{R_k}F_{R_k}}{W_{k-1}W_{R_k}}$$

$$W_k = W_{k-1} + W_{R_k}$$

As more and more data points are added to the scene, the voxel weights increase and small errors in depth map computation are averaged out.

5.4 Visualization

In the previous section, a method was described to create a truncated signed distance field (TSDF), by fusing the incoming depth maps. The surface of the reconstructed mesh can be viewed by displaying the isosurface of the TSDF at $F(\boldsymbol{p}) = 0$.

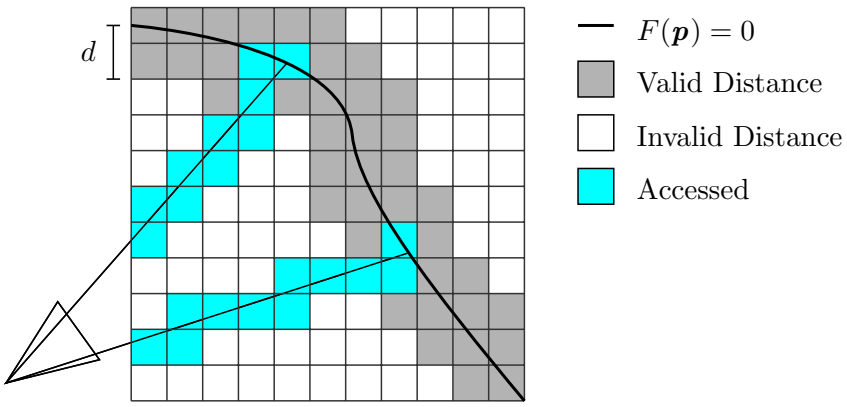

Figure 5.10: Ray casting the TSDF for a given camera pose. All accessed voxel are marked in blue.

We use the approach that is implemented in VoxelHashing [59]. The scene is visualized by ray casting the TSDF from the current camera position (see Figure 5.10) [64]. Each pixel's corresponding ray is marched, starting from the minimum depth for the pixel and stopped when a zero crossing is found indicating the surface interface. Marching also stops if a back face is found, or ultimately when reaching maximum depth [58]. When a surface point is found, the gradient is evaluated using numerical derivatives. The gradient direction is then used as surface normal for lighting calculations. The final, shaded output of the ray casting is shown in Figure 5.11.

Figure 5.11: Dense preview of the scene generated by ray casting the TSDF. The numerical normal estimates allow Phong shading of the surface.

Different acceleration structures can further speed up marching through empty space [64]. Newcombe et al [58] found that *ray skipping* provides a useful acceleration for truncated signed distance fields. In ray skipping, the fact is utilized that near $F(\boldsymbol{p}) = 0$ the fused volume holds a good approximation to the true surface distance. Using the known truncation distance d, the ray is marched by sampling the TSDF values with a step-size of $y < d$. A step y must pass through at least one non-truncated value before stepping over the surface zero crossing [58]. As a result, less voxels have to be read from memory, which is visualized in Figure 5.12.

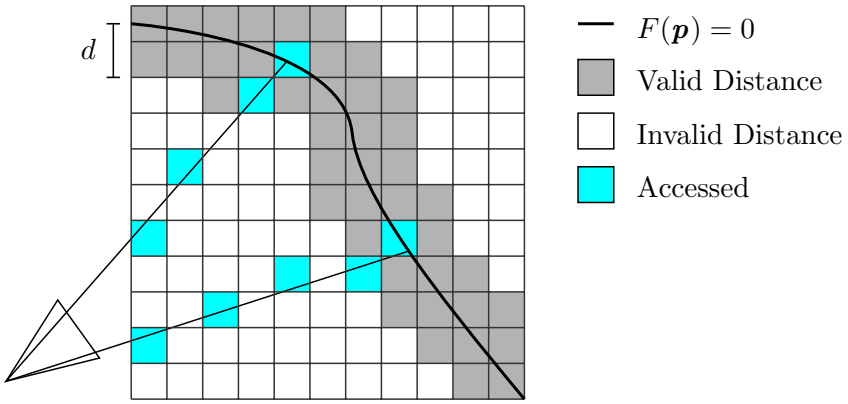

Figure 5.12: Voxels are skipped in *ray skipping*, if the distance to the previous voxel is smaller than the truncation distance d.

Chapter 6

Evaluation

In this chapter, the results of our reconstruction system for different scenes and cameras are presented. All videos were captured at ARRI in the context of *FOR3D* [20], a composite between research institutes and industry partners. The scan setup is shown in Figure 6.1. The object being scanned is placed on the floor between three white area light sources. The elevated placement of the lights reduces shadowing artifacts of the object and operator. The images captured by the camera are sent to a computer and a live preview of the scene is reconstructed by our system. This preview image is sent back to a display mounted on top of the camera. The cables for data transfer are attached to a high metal pole, yet we used a second person holding the cables to ensure they do not appear in front of the camera.

Figure 6.1: Scan setup for evaluating our reconstruction system. The object being scanned is lit from all sides by bright white area light sources.

Two different cameras (see Figure 6.2) were used to capture two different scenes, resulting in four scanning configurations. The first camera is a Logitech HD Pro Webcam C920 [43] with an input resolution of 1920×1080 pixels. In our tests, we set the camera to 1280×720, because the full HD mode was only able to stream 10 frames per second. The second camera is an ARRI ALEXA Mini [4], which is mainly used in the film industry. The live image transmission of the ALEXA requires a special PCI-express card that stores incoming frames

in a FIFO buffer and writes to memory with DMA. This complete setup is able to stream 24 frames per second in 1920×1080 pixels to the PC. The transferred image file format is 10-bit RGB stored in 4 bytes per pixel.

Figure 6.2: The two different cameras used in our tests. Left: Logitech HD Pro Webcam C920 [43]. Right: ARRI ALEXA Mini [4].

The first scanned scene (see Figure 6.3 left) consist of a black box with aluminium edges and an old ARRI film camera placed on top. A few magazines were added to the sides and on top of the box to cover large textureless regions and increase the number of recognizable feature points. The second scene (see Figure 6.3 right) is built of a suitcase decorated with office supplies, books, papers and a dark backpack, which leans on the far side of the suitcase. In the following, the first scene will be called *Box* and the second scene *Suitcase*.

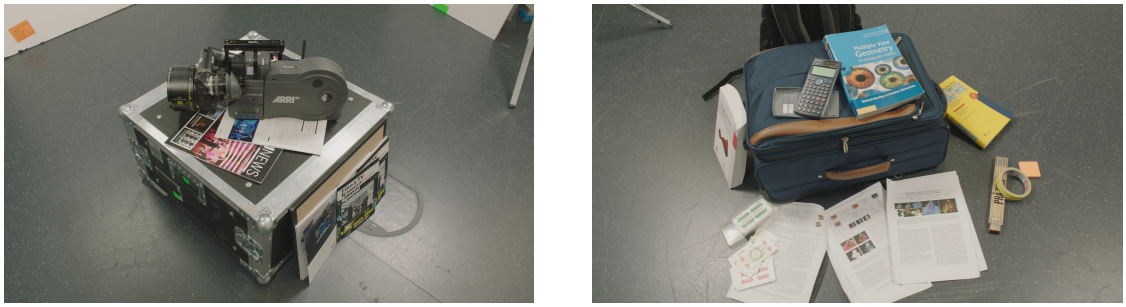

Figure 6.3: The *Box* scene (left) and *Suitcase* scene (right).

In the next section, our real-time reconstruction system is evaluated for both scenes and cameras. Then the offline reconstruction results of the cameras are compared and the important outcomes of the evaluation are discussed in the last section.

6.1 Real-Time Reconstruction

The main contribution of this thesis is a reconstruction system that generates a dense preview of the 3D model of the scene in real time. The stability of the camera tracking and sparse reconstruction (Chapter 4) is evaluated in the next section by measuring the amount of good image-to-world matches. Preview images of the dense reconstruction (Chapter 5) are shown in Section 6.1.2. A detailed comparison of the real-time reconstruction between both of our tested cameras is given in Section 6.1.3. After that, the processing time for each step is presented and the change in this processing time is evaluated during a scan.

6.1.1 Tracking

The camera tracking is initialized by a two-view reconstruction between the first and the second keyframe. The method described in Section 4.3 is stable and gives good results for all our test scenes. For other scenes, if the number of good two-view matches is too low, the initial reconstruction is too sparse and the camera tracking is unstable (see Section 4.4.2). A minimum match threshold was added to prevent this scenario, but this means our system will fail in textureless or low contrast environments.

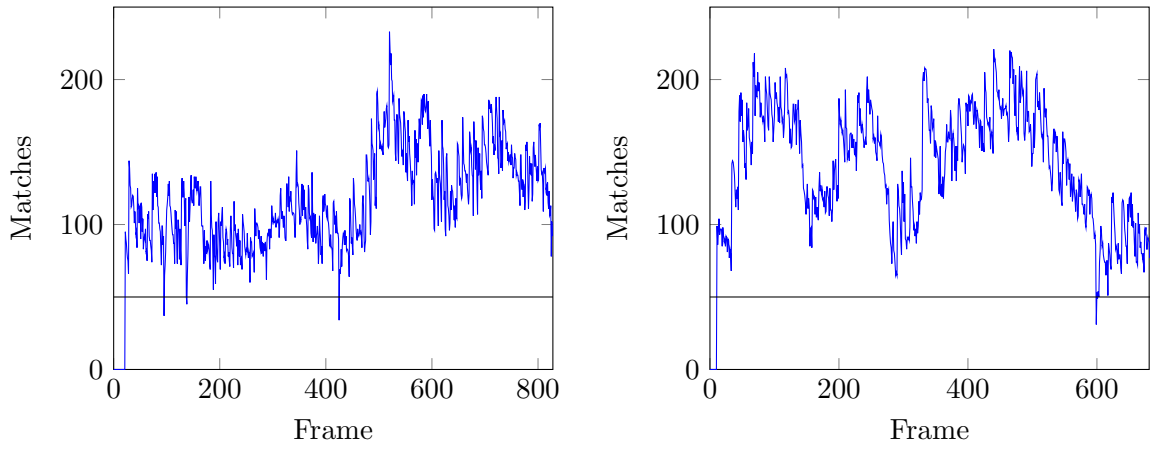

Figure 6.4: Number of image-to-world matches in the *Box* (left) and *Suitcase* (right) scene. Only in a few frames, less than 50 matched could be found (black horizontal line). One example is frame 425 of the *Box* scan (see Figure 6.5)

The keypoint-based tracking, which matches image features to previously reconstructed 3D points, is stable, if enough matches can be found. In all of our test configurations, between 80 and 200 good matches were found for most of the frames (see Figure 6.4). Only for a few rare cases fewer than 50 feature points could be matched, yet there were still enough to compute a valid camera pose. These outlier frames were caused by images exhibiting strong motion blur, which reduces the number of detected feature points (see Figure 6.5).

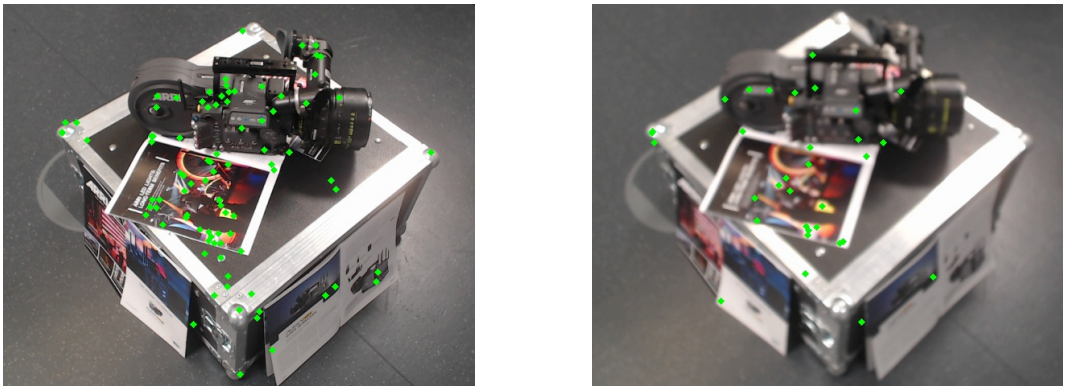

Figure 6.5: 109 good matches (green points) have been found in frame 420 (left). In frame 425 only 34 matches have been found, because the image exhibits motion blur.

We found that when running the reconstruction in real time, the tracking stability and preview quality depends on the processed frame rate. If the system processes only 5 frames per second, around 80% of the incoming 24 frames per second are dropped. Thus, the pose estimation gets less stable, because the distance of the camera between two consecutive frames is increased (see Section 4.4.2). As a result, the operator has to compensate for low frame rates and move slower to achieve the same reconstruction quality.

In cases when the tracking fails to compute a valid camera pose for the current frame, the system is in an ill state and a valid pose must be recovered as fast as possible. In that case, we use the last correctly computed camera pose as an estimate and notify the operator that tracking has failed. The camera can then be moved back to a previous position for pose recovery. This process is intuitive and the recovery worked in most occasions. Only when the tracking quality is already bad for multiple frames, a valid recovery is not possible anymore. In such a case, a complete restart of the scan is required.

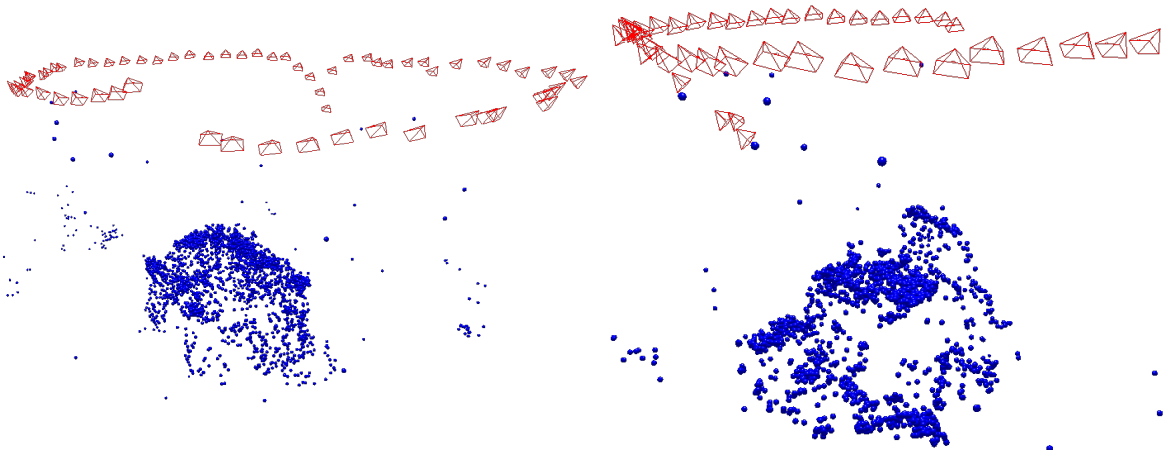

Figure 6.6: Sparse point cloud and keyframe camera poses for the *Box* (left) and *Suitcase* (right) scene. The *Box* scan consists of 65 keyframes and 3710 points. The *Suitcase* scan consists of 43 keyframes and 2290 points.

Figure 6.6 shows the reconstructed sparse point cloud and computed camera poses of the keyframes. The rough outlines of the scanned objects can already be seen and the circular camera trace is visible. A few outliers remain in the sparse point cloud, for example, in the right image between the cameras. These outliers can not be detected automatically, because they satisfy the epipolar constraint (see Section 4.3) and introduce only a small reprojection error.

Another factor that influences tracking stability is bundle adjustment, which is performed when a new keyframe is added. If only a few iterations are performed or the number of variables is too small (see Section 4.7), reprojection errors accumulate over time. When the operator scans the object with a circular trajectory, an accumulated error results in a loop closure problem. That is, when parts of the object are reconstructed a second time at a different location. We observed this behavior after completely removing bundle adjustment in the sparse reconstruction. In Figure 6.7, the left bright edge of the box is reconstructed a second time with a slight offset.

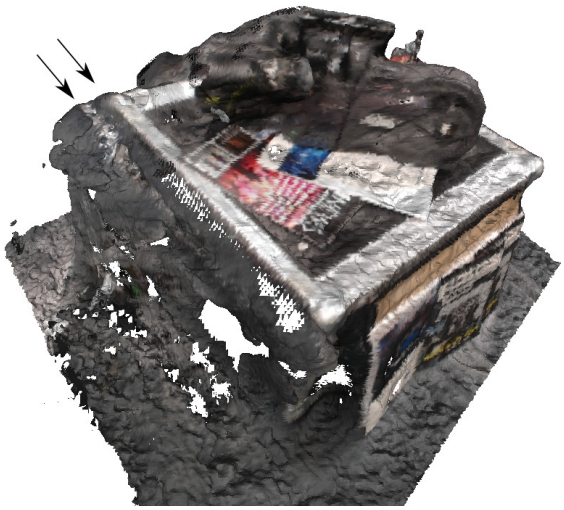

Figure 6.7: Loop closure problem after removing bundle adjustment. The edge of the box is reconstructed two times at different locations (black arrows).

6.1.2 Dense Reconstruction

The dense reconstruction starts with computing a depth image for every incoming frame (see Section 5.2). Outliers are removed aggressively by thresholding and segmentation, but the resulting depth maps also have many correctly estimated values removed. Furthermore, the scene is approximated by regions of constant depth, introducing step-like artifacts for surfaces not orthogonal to the viewing direction. As a summary, the computed depth maps are sparse, noisy, but contain few to none outliers (see Figure 6.8).

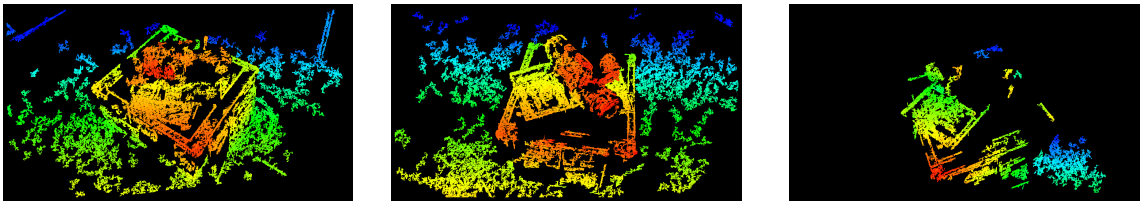

Figure 6.8: Depth maps generated for different frames of the *Box* scene. The colors are scaled so that the smallest depth value in each image is red and the largest value is blue.

For the depth map fusion, we use the original VoxelHashing implementation [60]. This framework was initially developed to fuse a stream of depth images taken from an active depth sensor like Microsoft's Kinect [52]. When comparing the properties of our computed depth maps to the measured values of an depth sensor, we find similarities in the amount of noise and outliers, but an active depth sensor usually produces denser depth maps, especially in textureless regions.

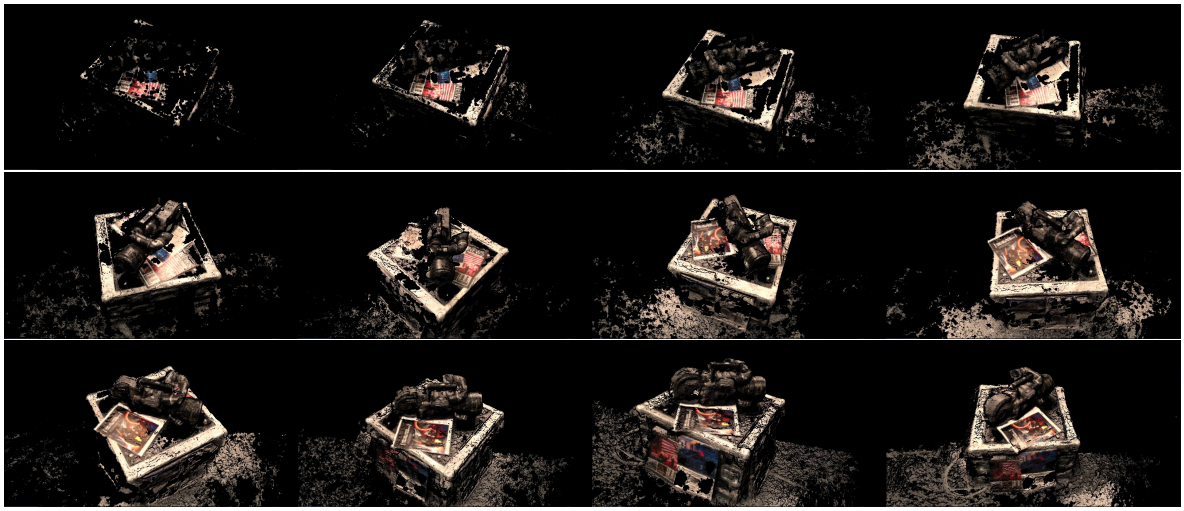

Figure 6.9: Raycasted output of the real-time reconstruction as shown to the operator. At the beginning (top left) many holes are visible. These are filled up as more images are taken from different angles.

In Figure 6.9, an image sequence of the *Box* scene is given, which shows the final output of the dense reconstruction. From top left to bottom right more and more depth maps are fused, holes are filled, and noisy surfaces are smoothed. Only in regions where either not enough depth values were estimated or the lack of texture causes high noise levels, the reconstructed surface does not match the ground truth. However, that did not negatively impact the scanning process, because these artifacts only appeared in less important parts of the scene such as the background or the floor. In Figure 6.10, a close up comparison of the reconstructed surface is shown. The dark floor and the background geometry is noisy and contains holes. The box and camera surface is reconstructed accurately.

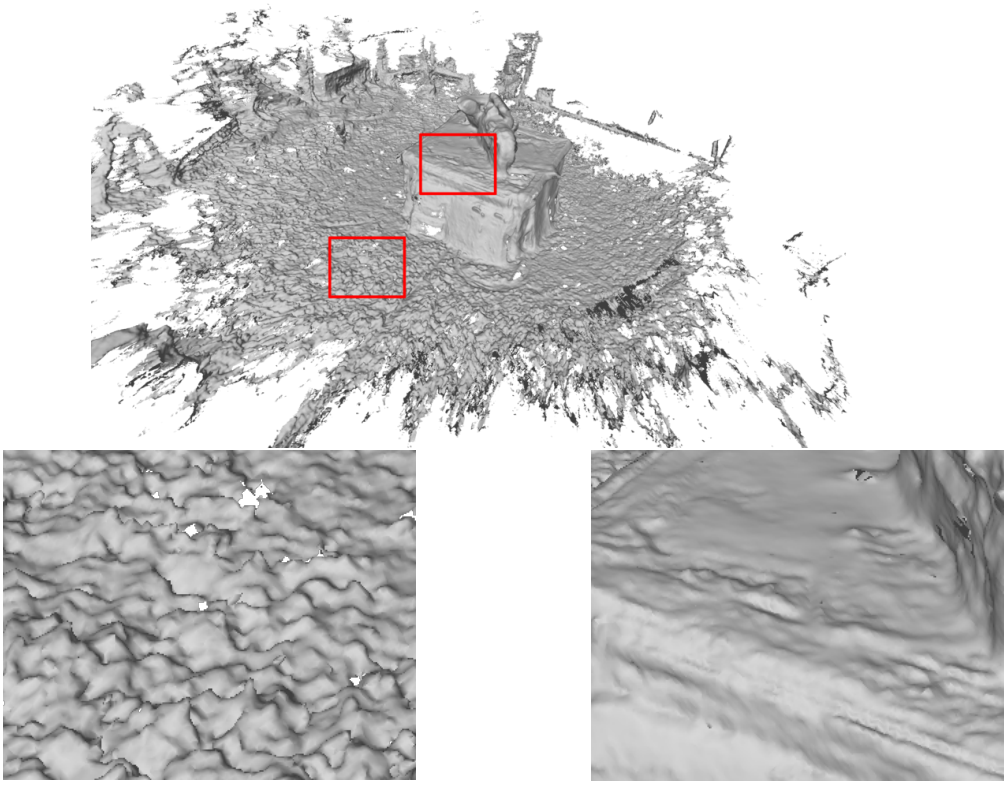

Figure 6.10: Real-time reconstructed surface of the *Box* scene without colors. The floor and background objects are noisy and contain holes. The surface of the box and the camera is reconstructed well, because depth values from hundreds of frames are fused in the volumetric integration stage.

6.1.3 Camera Comparison

As previously noted, two different cameras where used in our tests: A Logitech webcam and an ARRI ALEXA Mini. To compare the reconstruction of both cameras, the webcam was mounted onto the ALEXA and both images were streamed simultaneously to the PC (see Figure 6.11).

Figure 6.11: The webcam is mounted on top of the ALEXA to capture frames from the same viewing position.

In every timestep, the images of both cameras were captured and stored on the hard disk. After the scan was completed, we reconstructed the scene from video as if it was a live input stream. This process ensured that the same number of frames were taken from both cameras with each frame being shot from the same camera location. In Figure 6.12, the same frame of the webcam (left) and ALEXA (right) is shown. It can be seen that these images do not match perfectly, because the webcam's optical center is slightly above the ALEXA's and the field of view differs by a few degrees.

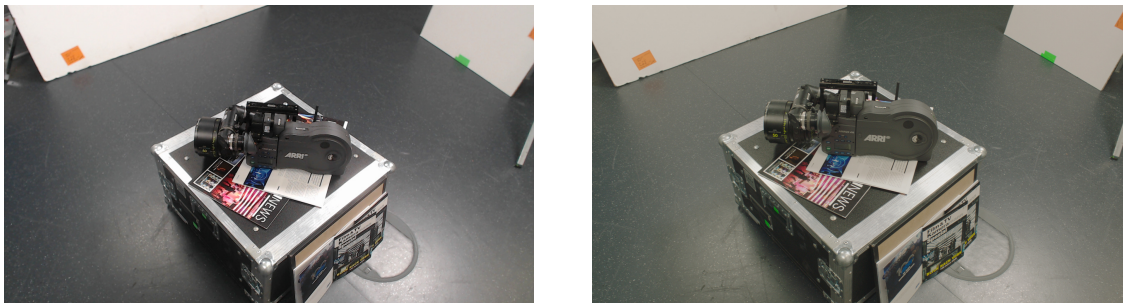

Figure 6.12: One frame captured from the webcam (left) and ALEXA (right) at the same time.

For the real-time reconstruction, both images were downsampled to an equal resolution of 960×540 pixels. Surprisingly, we did not find a notable difference in tracking stability and dense reconstruction quality. Figure 6.13 shows the number of matched feature points for both scenes and cameras. We can see that the graphs for each camera behave similar, increasing and decreasing at roughly the same frames. The low frequency graph fluctuations are caused by the operator's camera movement. For example, the valley around frame 300 of the *Suitcase* scan was created as the camera was moved towards the object for some close up images. At this point, many previously tracked background features disappear, which reduces the number of good feature matches.

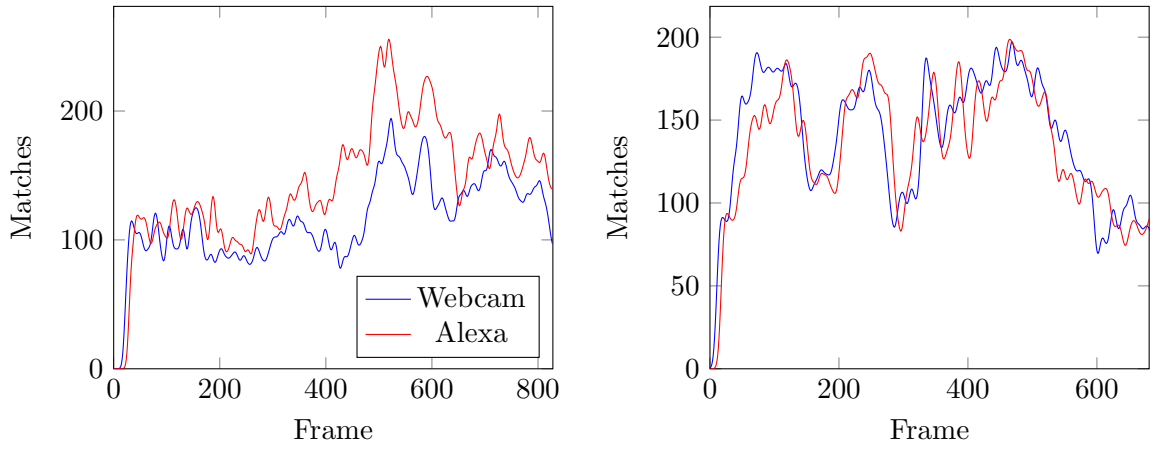

Figure 6.13: Image-to-world feature matches for the *Box* scene (left) and *Suitcase* scene (right). Roughly the same amount of matches were found over the complete scan for both cameras.

Figure 6.14 shows the online-reconstructed surface for both cameras and scenes. The actual object is sharply reconstructed in all scans and similar noise patterns are visible in the background and textureless regions.

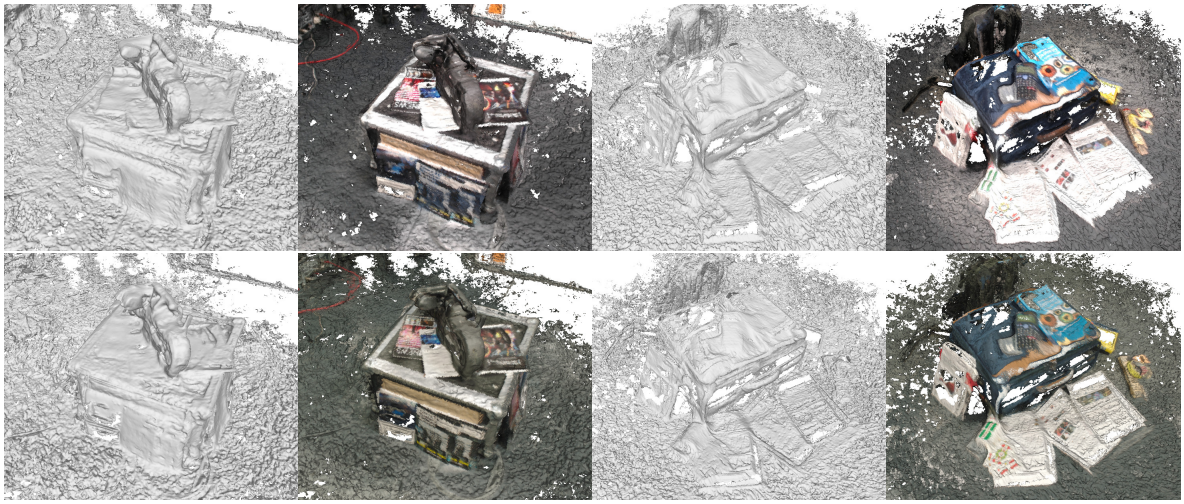

Figure 6.14: Reconstructed surface of the webcam (top row) and ALEXA (bottom row). Similar noise patterns and holes appear in both images.

6.1.4 Timings

An overview of the time consumed for each step of a regular frame is given in Table 6.1. If a keyframe is detected (see Section 4.5), two additional steps have to be performed: The triangulation of new world points and incremental bundle adjustment. The total processing time of a keyframe is shown in Table 6.2.

Step	Time (ms)
Preprocessing	3.51
Feature Detection	1.05
Feature Matching	0.38
Pose Estimation	4.21
Depth Map Computation	15.49
Depth Map Fusion	2.86
Rendering	11.88
Total	39.38

Table 6.1: Average processing time of each step in our real-time reconstruction system.

As more keyframes are added, the number of points in the scene increases. These additional points have to be considered in image-to-world feature matching (Section 4.4.1), which increases processing time. Figure 6.15 shows the total time for each frame over a video sequence of 813 images. It can be seen that the time increases slowly, yet is still fast enough

Step	Time (ms)
Non-Keyframe Steps	39.38
World Point Creation	10.28
Incremental Bundle Adjustment	17.51
Total	67.17

Table 6.2: Average processing time of a keyframe. The *Non-Keyframe Steps* are shown in Table 6.1.

for interactive 3D reconstruction. The peaks in the graph indicate the points in time when a keyframe was added. The peak size differs over the scan, because incremental bundle adjustment can be terminated early when the change in error is small.

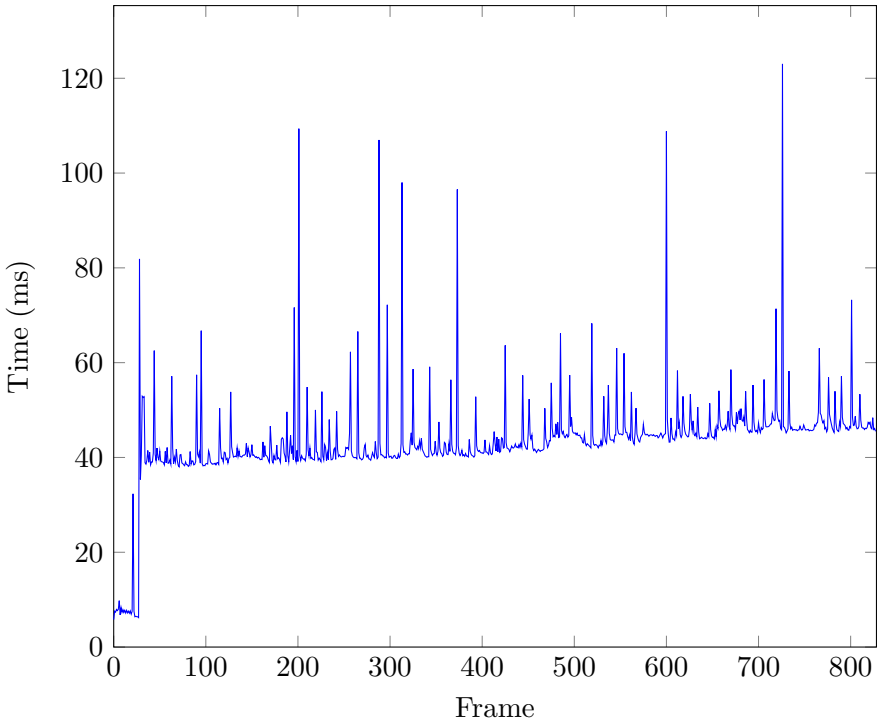

Figure 6.15: The processing time per frame increases slightly over the scan. Additional steps have to be performed for keyframes resulting in large peaks in the graph.

When testing our system with a live input video stream, we found that reading the image from camera to memory also had a non-negligible impact on the frame rate. In our implementation, capturing a single image takes around 50 ms for the webcam and 30 ms for the ALEXA. These timings were not added to Table 6.1, because they were hidden by an additional thread, which reads the next frame while the previous one is processed.

6.2 Offline Reconstruction

As mentioned in the introduction of this thesis, the extracted keyframes of the real-time reconstruction can be stored in full resolution and later be used by an offline reconstruction system to create a high quality output model. We use Agisoft PhotoScan [3] as an offline system with all settings set to high. Three different keyframe types were stored while scanning the object. Small keyframes of size 1280×720 and 1920×1080 were saved directly as they were transmitted to the computer by the webcam and ALEXA (see Section 6). The ALEXA provided an additional functionality of storing high resolution raw images with 2880×1620 pixels on the camera itself. It is currently not possible to stream the raw images to a computer, thus we selected them manually after the scan was completed by comparing them to the low resolution keyframes. In Figure 6.16, the final meshes of the offline reconstruction are shown for the *Box* scene and in Figure 6.17 for the *Suitcase* scene. The time required to create these models ranged from 30 minutes to over 3 hours depending on image resolution and number of keyframes.

Figure 6.16: Offline reconstructions of the *Box* scene for the webcam (top) and ALEXA (bottom).

In both scenes, a difference in quality between the cameras can be seen in geometry and texture. ALEXA's meshes (lower row) appear sharper and capture more details than the webcam's meshes. For example, in the *Suitcase* scene in the upper row, books and paper edges are blurred, forming a smooth transition to the background. The edges of the ALEXA reconstruction are sharper, making individual objects easier to distinguish. When looking at the magazine placed on top of the box, the headline is blurred on the top and readable on the bottom. These quality difference between the reconstructed models of our cameras matches the expectation, because higher resolution input images carry more information, increasing both geometry and texture quality.

Figure 6.17: Offline reconstructions of the *Suitcase* scene for the webcam (top) and ALEXA (bottom).

The successful offline reconstructions show that the keyframe selection process of Section 4.5 works. To further test how keyframe selection affects the offline reconstruction, we scanned an advent wreath scene with non uniform camera motion. The operator started with a slow velocity, increased the speed over time, and then stood still for multiple seconds on the same spot. We reconstructed the model by choosing the images in two different ways: The left image of Figure 6.18 shows the keyframe locations and final model when selecting the keyframes through our real-time reconstruction (see Section 4.5). All cameras could be aligned and the computed model is of reasonable quality. The right image of Figure 6.18 was created by passing every tenth frame of the input video to PhotoScan. The camera alignment and dense reconstruction fails resulting in a broken 3D model.

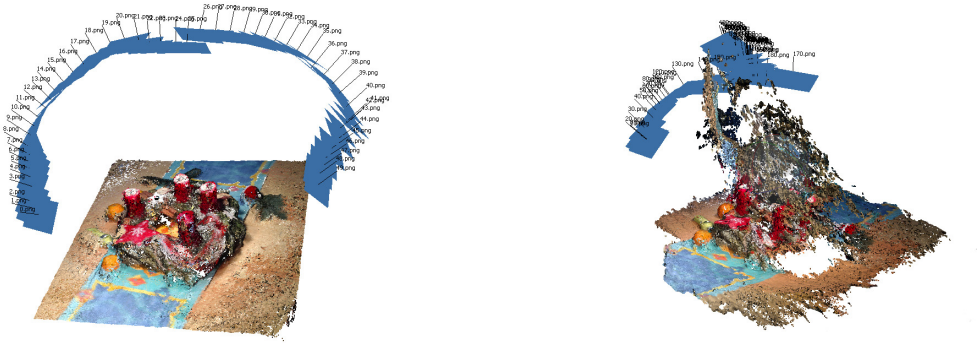

Figure 6.18: Left: Running the real-time reconstruction and using the selected keyframes for offline reconstruction. Right: Choosing every tenth frame as input for PhotoScan. The same number of images were used in both reconstructions.

6.3 Discussion

In this chapter, we have shown that the proposed challenges (Section 1.2) could be solved and good results were produced for our test configurations. The real-time system displays a preview of the scene and reduces the amount of input data to a handful of keyframes, which can then be used by an offline reconstruction program. The preview itself looks good, except in low-contrast or texture less regions, because large parts of the depth maps are filtered out.

The keyframe selection process and subsequent offline reconstruction produces good results for all our tested scenes. We showed on an example (Figure 6.18) that offline reconstruction systems have trouble computing valid camera poses when the camera is moved non-uniformly and frames are chosen naively. It might be beneficial for large data sets, for example, a drone scanning the surface, to automatically select *good* keyframes with a real-time system before passing the images to offline reconstruction.

Another interesting result of testing our reconstruction system with two different cameras is that the preview generated in real time does not differ between an inexpensive webcam and a high end movie camera (see Figure 6.14). The reason is that the images are downsampled to the same size and all the additional information is lost. In the future, when processing power increases or algorithms are optimized this might change again, because downsampling is not required anymore and good cameras can be used to its full potential. For now, we believe that if only a real-time reconstruction of moderate quality is required, inexpensive cameras found, for example, in smartphones and webcams are sufficient.

Chapter 7

Summary and Future Work

We have presented a system for camera tracking and 3D reconstruction. A single RGB camera is used as input and a dense, colored surface of the object is reconstructed in real time. Some frames are marked as keyframes and are stored in high resolution. After the real-time scan is completed, these keyframes can be used by an offline reconstruction application, such as PhotoScan, to create a high resolution model of the object. The camera tracking is based on finding and matching SIFT features, which requires the scanned object to have an adequate texture. For every input frame, a depth image is estimated with stereo matching and fused into a truncated signed distance field (TSDF). The TSDF is ray casted from the current camera viewpoint to obtain the final preview of the scene. The shaded preview is shown to the operator to enable visual feedback during the scan.

In our experiments, we found that some steps can further be improved. Especially the performance should be looked at, because an increase in frame rate also increases tracking and dense reconstruction quality. In our implementation, data has to be transfered several times between CPU and GPU memory. These transfers act as synchronization points and reduce the overall performance. One example is that feature matching (Section 4.4.1) is executed on the GPU and the subsequent pose estimation (Section 4.4.2) is ran on the CPU. In this case, an array of candidate matches has to be copied from device to host memory. A GPU implementation of the P3P problem, which is the main work of pose estimation, will eliminate the data transfers. Additionally, the RANSAC procedure benefits from high parallelism, which would further speed up pose estimation.

One of the most time consuming steps is the incremental bundle adjustment (Section 4.7). A small delay is introduced, when it is executed after a new keyframe has been selected. A parallelization of this step, which optimizes the scene while further images are processed, is possible as shown in [40], but it could be more beneficial to optimize the bundle adjustment itself. The results are immediately available for further frames and our reconstruction pipeline comes closer to an offline system. The nonlinear optimization in bundle adjustment can be improved by porting it to the GPU, because fast solver exist [35] that scale well with the compute power and memory bandwidth of modern GPUs.

In camera tracking and sparse reconstruction, new points are added for every keyframe, increasing the necessary work of feature matching and pose estimation (see Section 6.1.1). Some of these points are outliers and can not be removed automatically, because they satisfy the epipolar constraint and introduce only a small reprojection error. In this case, a correct image-to-world matching is not possible. A removal of these points improves tracking stability, because fewer correct matches will be filtered out. Different heuristics of detecting these points can be examined. For example, points could be marked as *untrackable*, if they can not be matched in n successive frames.

Candidate matches can be further reduced by estimating a 3D normal for each world point. This could be achieved by either examining local feature gradients over multiple images or gathering surface information from dense reconstruction. If world points are enhanced with a normal, they can be discarded in feature matching when the angle between the estimated viewing direction and point normal is above a user-defined threshold.

There are multiple possible ways to increase the performance and quality of dense reconstruction (Chapter 5). In contrast to our algorithm, stereo matching is often preceded by image rectification, because the match search can then be performed along scan lines. The scan line matching is cache friendly, because it results in coalesced memory access patterns when executed on a GPU. One example is MobileFusion's [63] depth map computation, which is able to generate real-time depth images on the graphic processing unit of a smartphone. Another interesting task would be to revisit existing stereo matching algorithms that are used in offline reconstructions system, such as PMVS [24] and SGBM [36]. If these algorithms are reimplemented on the GPU, taking into account that correct depth estimates are already present for a sparse set of points, it might be possible to achieve reasonable frame rates with precise depth measurements.

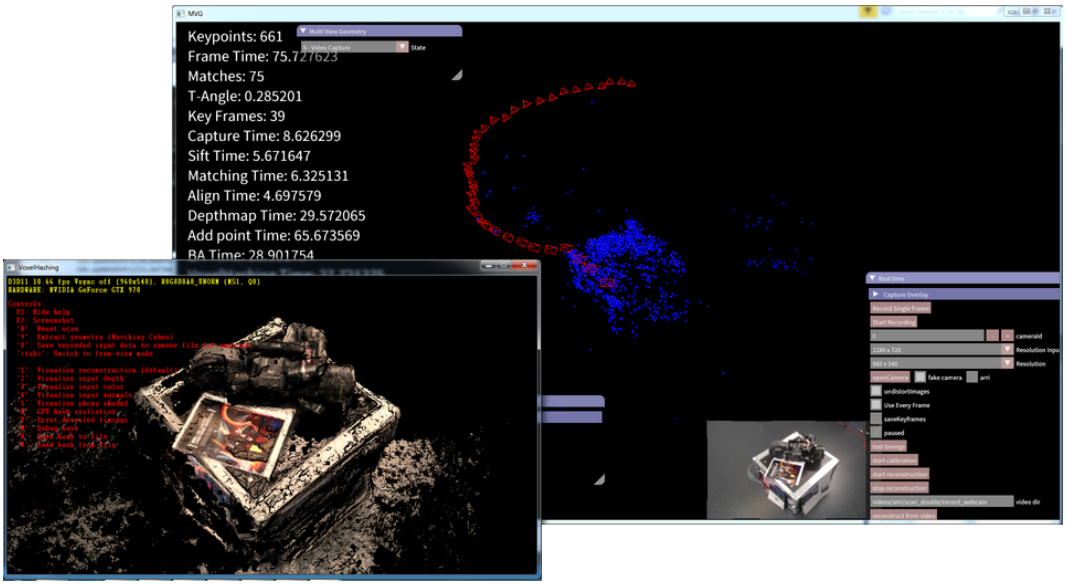

Figure 7.1: The depth maps and camera poses are sent from our reconstruction system (large background window) to the VoxelHashing framework (foreground window). Merging both system increases usability and performance.

Currently, the original VoxelHashing implementation is used in dense reconstruction for depth map fusion and rendering. This implementation requires a DirectX context and compiles with Mircosoft's Visual Studio. The rest of our system runs both on Linux and Windows with all latest libraries and compilers. Figure 7.1 shows a screenshot of our reconstruction system with the two different windows. Merging both systems removes memory transfers and synchronization points, because all data would be immediately available in only one window. Furthermore, additional optimizations can be incorporated, because VoxelHashing was originally built for reconstructing active depth sensor data instead of depth maps computed from RGB images. Differences are, for example, that a depth sensor usually produces a denser depth image with a fixed range of a few meters. Our system generates depth maps with more holes and a depth-scale that changes over time.

After VoxelHashing is integrated into our system, the information stored in the TSDF can also be used to improve camera tracking quality. This step is labeled as *dense tracking* in MobileFusion [63] and reduces tracking errors and mitigates the loop closure problem. An example of dense tracking is aligning the computed depth map for each frame to the scene with the iterative closest point algorithm.

In Section 6.1.1 we have shown that some frames experience motion blur, resulting in fewer feature matches compared to other frames. If a motion-blurred frame is selected as a keyframe, additional errors are introduced in camera tracking and further reconstruction is less stable. A *blurriness* factor could be computed for each frame by inspecting local image gradients or by computing the relative angular velocity between the camera and the scene. If the blurriness is too high, keyframe creation and depth map computation can be prohibited, ensuring that no defective data is integrated into the scene.

List of Figures

Bibliography

[1] Sameer Agarwal, Keir Mierle, et al. *Ceres Solver*. http://ceres-solver.org.

[2] Agisoft. *Agisoft PhotoScan User Manual Professional Edition, Version 1.3*. http://www.agisoft.com/pdf/photoscan-pro_1_3_en.pdf. 2017.

[3] Agisoft. "PhotoScan - http://www.agisoft.com/". In: 2010.

[4] ARRI. *ALEXA Mini*. 2015. URL: http://www.arri.com/camera/alexa_mini/camera_details/alexa-mini/ (visited on 09/28/2017).

[5] Sunil Arya et al. "An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions". In: *J. ACM* 45.6 (Nov. 1998), pp. 891–923. ISSN: 0004-5411. DOI: 10.1145/293347.293348. URL: http://doi.acm.org/10.1145/293347.293348.

[6] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007.

[7] Paul J. Besl and Neil D. McKay. "A Method for Registration of 3-D Shapes". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 14.2 (Feb. 1992), pp. 239–256. ISSN: 0162-8828. DOI: 10.1109/34.121791. URL: http://dx.doi.org/10.1109/34.121791.

[8] D. C. Brown. "Decentering Distortion of Lenses". In: *Photometric Engineering* 32.3 (1966), pp. 444–462.

[9] Matthew Brown and D Lowe. "Invariant Features from Interest Point Groups". In: *BMVC 2002: 13th British Machine Vision Conference*. Sept. 2002, pp. 253–262. URL: http://opus.bath.ac.uk/26128/.

[10] A. R. Chowdhury et al. "3D face reconstruction from video using a generic model". In: *Proceedings. IEEE International Conference on Multimedia and Expo*. Vol. 1. 2002, 449–452 vol.1. DOI: 10.1109/ICME.2002.1035815.

[11] A. E. Conrady. "Decentred Lens-Systems". In: *Monthly Notices of the Royal Astronomical Society* 79.5 (1919), pp. 384–390. DOI: 10.1093/mnras/79.5.384. URL: %5Curl%7B%20http://dx.doi.org/10.1093/mnras/79.5.384%7D.

[12] Brian Curless and Marc Levoy. "A Volumetric Method for Building Complex Models from Range Images". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 303–312. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237269. URL: http://doi.acm.org/10.1145/237170.237269.

[13] J. P. de Villiers, F. W. Leuschner, and R. Geldenhuys. "Centi-pixel accurate real-time inverse distortion correction". In: *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. Vol. 7266. Nov. 2008, p. 726611. DOI: 10.1117/12.804771.

[14] Jakob Engel. *LSD-SLAM: Large-Scale Direct Monocular SLAM (ECCV '14)*. 2014. URL: https://www.youtube.com/watch?v=GnuQzP3gty4 (visited on 09/28/2017).

[15] Jakob Engel, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: *Computer Vision – ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part II*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 834–849. ISBN: 978-3-319-10605-2. DOI: 10.1007/978-3-319-10605-2_54. URL: https://doi.org/10.1007/978-3-319-10605-2_54.

[16] Jakob Engel, Jurgen Sturm, and Daniel Cremers. "Semi-dense Visual Odometry for a Monocular Camera". In: *The IEEE International Conference on Computer Vision (ICCV)*. Dec. 2013.

[17] Epic. *Creating the Open World Kite Real-Time Demo in UE4*. GDC 2015. 2015. URL: https://www.youtube.com/watch?v=clakekAHQx0.

[18] Ian Failes. *Surviving San Andreas*. 2015. URL: https://www.fxguide.com/featured/surviving-san-andreas/ (visited on 09/28/2017).

[19] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: http://doi.acm.org/10.1145/358669.358692.

[20] FOR3D. *Schritthaltende 3D-Rekonstruktion und -Analyse*. 2017. URL: https://www.for3d.bayern/ (visited on 09/28/2017).

[21] OpenCV Foundation. *Open Source Computer Vision Library*. https://github.com/opencv/opencv. 2017.

[22] Simon Fuhrmann and Michael Goesele. "Floating Scale Surface Reconstruction". In: *ACM Trans. Graph.* 33.4 (July 2014), 46:1–46:11. ISSN: 0730-0301. DOI: 10.1145/2601097.2601163. URL: http://doi.acm.org/10.1145/2601097.2601163.

[23] Simon Fuhrmann, Fabian Langguth, and Michael Goesele. "MVE: A Multi-view Reconstruction Environment". In: *Proceedings of the Eurographics Workshop on Graphics and Cultural Heritage*. GCH '14. Darmstadt, Germany: Eurographics Association, 2014, pp. 11–18. ISBN: 978-3-905674-63-7. DOI: 10.2312/gch.20141299. URL: http://dx.doi.org/10.2312/gch.20141299.

[24] Y. Furukawa and J. Ponce. "Accurate, Dense, and Robust Multiview Stereopsis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.8 (Aug. 2010), pp. 1362–1376. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.161.

[25] Yasutaka Furukawa and Carlos Hernández. "Multi-View Stereo: A Tutorial". In: *Foundations and Trends® in Computer Graphics and Vision* 9.1-2 (2015), pp. 1–148. ISSN: 1572-2740. DOI: 10.1561/0600000052. URL: http://dx.doi.org/10.1561/0600000052.

[26] Xiao-Shan Gao et al. "Complete solution classification for the perspective-three-point problem". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25.8 (Aug. 2003), pp. 930–943. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2003.1217599.

[27] M. Goesele et al. "Multi-View Stereo for Community Photo Collections". In: *2007 IEEE 11th International Conference on Computer Vision*. Oct. 2007, pp. 1–8. DOI: 10.1109/ICCV.2007.4408933.

[28] G. H. Golub and C. Reinsch. "Singular value decomposition and least squares solutions". In: *Numerische Mathematik* 14.5 (Apr. 1970), pp. 403–420. ISSN: 0945-3245. DOI: 10.1007/BF02163027. URL: https://doi.org/10.1007/BF02163027.

[29] Google. "Google Earth". In: Aug. 2007. URL: http://earth.google.de/.

[30] John C. Gower and Garmt B. Dijksterhuis. *Procrustes problems*. Vol. 30. Oxford Statistical Science Series. Oxford, UK: Oxford University Press, Jan. 2004. URL: http://oro.open.ac.uk/2736/.

[31] John W. Harris and Horst Stocker. *The Handbook of Mathematics and Computational Science*. 1st. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997. ISBN: 0387947469.

[32] R. HARTLEY. "An algorithm for self calibration from several views". In: *CVPR, 1994* (1994). URL: %5Curl%7Bhttp://ci.nii.ac.jp/naid/10020691075/en/%7D.

[33] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004. ISBN: 9781139449144. URL: https://books.google.de/books?id=e30hAwAAQBAJ.

[34] E. Hecht. *Optics 4th Edition*. 1998.

[35] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*. Vol. 49. 1. NBS, 1952.

[36] H. Hirschmuller. "Accurate and efficient stereo processing by semi-global matching and mutual information". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. June 2005, 807–814 vol. 2. DOI: 10.1109/CVPR.2005.56.

[37] Peter J Huber. *Robust statistics*. New York, NY: Wiley, 1981. URL: http://cds.cern.ch/record/99853.

[38] Michael Kazhdan et al. "Unconstrained Isosurface Extraction on Arbitrary Octrees". In: *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*. SGP '07. Barcelona, Spain: Eurographics Association, 2007, pp. 125–133. ISBN: 978-3-905673-46-3. URL: http://dl.acm.org/citation.cfm?id=1281991.1282009.

[39] R. Khilar, S. Chitrakala, and S. SelvamParvathy. "3D image reconstruction: Techniques, applications and challenges". In: *2013 International Conference on Optical Imaging Sensor and Security (ICOSS)*. July 2013, pp. 1–6. DOI: 10.1109/ICOISS.2013.6678395.

[40] G. Klein and D. Murray. "Parallel Tracking and Mapping for Small AR Workspaces". In: *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*. Nov. 2007, pp. 225–234. DOI: 10.1109/ISMAR.2007.4538852.

[41] G. Klein and D. Murray. *Parallel Tracking and Mapping for Small AR Workspaces - Source Code*. 2008. URL: http://www.robots.ox.ac.uk/~gk/PTAM/ (visited on 09/28/2017).

[42] Hideo Kojima. *Photorealism Through the Eyes of a FOX: The Core of Metal Gear Solid Ground Zeroes*. GDC 2013. 2013. URL: https://www.youtube.com/watch?v=FQMbxzTUuSg.

[43] Logitech. *HD Pro Webcam C920*. 2012. URL: https://www.logitech.com/en-us/product/hd-pro-webcam-c920 (visited on 09/28/2017).

[44] C. Loop and Zhengyou Zhang. "Computing rectifying homographies for stereo vision". In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. Vol. 1. 1999, 131 Vol. 1. DOI: 10.1109/CVPR.1999.786928.

[45] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422. URL: http://doi.acm.org/10.1145/37401.37422.

[46] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *Int. J. Comput. Vision* 60.2 (Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[47] D.G. Lowe. *Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image*. US Patent 6,711,293. Mar. 2004. URL: https://www.google.com/patents/US6711293.

[48] Q.-T. Luong and O.D. Faugeras. "Self-Calibration of a Moving Camera from Point Correspondences and Fundamental Matrices". In: *International Journal of Computer Vision* 22.3 (Mar. 1997), pp. 261–289. ISSN: 1573-1405. DOI: 10.1023/A:1007982716991. URL: %5Curl%7Bhttps://doi.org/10.1023/A:1007982716991%7D.

[49] Donald W Marquardt. "An algorithm for least-squares estimation of nonlinear parameters". In: *Journal of the society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441.

[50] Andrew Maximov. *Future of Art Production in Games*. GDC 2017. 2017. URL: https://www.gdcvault.com/play/1024104/Future-of-Art-Production-in.

[51] Christoph Rhemann Michael Bleyer and Carsten Rother. "PatchMatch Stereo - Stereo Matching with Slanted Support Windows". In: *Proceedings of the British Machine Vision Conference*. http://dx.doi.org/10.5244/C.25.14. BMVA Press, 2011, pp. 14.1–14.11. ISBN: 1-901725-43-X.

[52] Microsoft. *Kinect Sensor*. 2010. URL: https://developer.microsoft.com/en-us/windows/kinect (visited on 09/28/2017).

[53] Microsoft. *Mixed Reality Inside-out tracking*. 2017. URL: https://docs.microsoft.com/en-us/windows/mixed-reality/enthusiast-guide/tracking-system (visited on 09/28/2017).

[54] Krystian Mikolajczyk and Cordelia Schmid. "An Affine Invariant Interest Point Detector". In: *Computer Vision — ECCV 2002: 7th European Conference on Computer Vision Copenhagen, Denmark, May 28–31, 2002 Proceedings, Part I*. Ed. by Anders Heyden et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 128–142. ISBN: 978-3-540-47969-7. DOI: 10.1007/3-540-47969-4_9. URL: https://doi.org/10.1007/3-540-47969-4_9.

[55] J.J. More. "Levenberg–Marquardt algorithm: implementation and theory". In: Jan. 1977. URL: http://www.osti.gov/scitech/servlets/purl/7256021.

[56] E. Mouragnon et al. "Real Time Localization and 3D Reconstruction". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 1. 2006, pp. 363–370. DOI: 10.1109/CVPR.2006.236.

[57] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. "DTAM: Dense tracking and mapping in real-time". In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2320–2327. DOI: 10.1109/ICCV.2011.6126513.

[58] Richard A. Newcombe et al. "KinectFusion: Real-time Dense Surface Mapping and Tracking". In: *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality*. ISMAR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 127–136. ISBN: 978-1-4577-2183-0. DOI: 10.1109/ISMAR.2011.6092378. URL: http://dx.doi.org/10.1109/ISMAR.2011.6092378.

[59] Matthias Nießner et al. "Real-time 3D Reconstruction at Scale Using Voxel Hashing". In: *ACM Trans. Graph.* 32.6 (Nov. 2013), 169:1–169:11. ISSN: 0730-0301. DOI: 10.1145/2508363.2508374. URL: http://doi.acm.org/10.1145/2508363.2508374.

[60] Matthias Nießner et al. *VoxelHashing reference implementation*. 2013. URL: https://github.com/niessner/VoxelHashing (visited on 09/28/2017).

[61] D. Nister. "An efficient solution to the five-point relative pose problem". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.6 (June 2004), pp. 756–770. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.17.

[62] NVIDIA. *Performance Primitives (NPP) Version 9.1*. 2017. URL: https://developer.nvidia.com/npp (visited on 09/28/2017).

[63] P. Ondrúška, P. Kohli, and S. Izadi. "MobileFusion: Real-Time Volumetric Surface Reconstruction and Dense Tracking on Mobile Phones". In: *IEEE Transactions on Visualization and Computer Graphics* 21.11 (Nov. 2015), pp. 1251–1258. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2459902.

[64] S. Parker et al. "Interactive ray tracing for isosurface rendering". In: *Visualization '98. Proceedings*. Oct. 1998, pp. 233–238. DOI: 10.1109/VISUAL.1998.745713.

[65] Victor Podlozhnyuk. "Image convolution with CUDA". In: (2007).

[66] V. Pradeep et al. "MonoFusion: Real-time 3D reconstruction of small scenes with a single web camera". In: *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. Oct. 2013, pp. 83–88. DOI: 10.1109/ISMAR.2013.6671767.

[67] Barbara Robertson. "Creative Robot". In: *Computer Graphics World - Issue: Volume 38 Issue 2: (Mar/Apr 2015)* (2015). URL: http://www.cgw.com/Publications/CGW/2015/Volume-38-Issue-2-Mar-Apr-2015-/Creative-Robot.aspx.

[68] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection". In: *Computer Vision – ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_34. URL: https://doi.org/10.1007/11744023_34.

[69] Daniel Scharstein and Richard Szeliski. "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms". In: *International Journal of Computer Vision* 47.1 (Apr. 2002), pp. 7–42. ISSN: 1573-1405. DOI: 10.1023/A:1014573219977. URL: https://doi.org/10.1023/A:1014573219977.

[70] Dmitry Semyonov. *Algorithms used in Photoscan*. http://www.agisoft.com/forum/index.php?topic=89.msg323. 2011.

[71] W. J. Smith. *Modern Optical Engineering: The Design of Optical Systems, Fourth Edition*. The McGraw-Hill Companies, 2008.

[72] Noah Snavely. *Bundler: Structure from Motion (SfM) for Unordered Image Collections*. 2008. URL: http://www.cs.cornell.edu/~snavely/bundler/ (visited on 09/28/2017).

[73] Noah Snavely, Steven M. Seitz, and Richard Szeliski. "Modeling the World from Internet Photo Collections". In: *Int. J. Comput. Vision* 80.2 (Nov. 2008), pp. 189–210. ISSN: 0920-5691. DOI: 10.1007/s11263-007-0107-3. URL: http://dx.doi.org/10.1007/s11263-007-0107-3.

[74] Matthias Teschner et al. "Optimized Spatial Hashing for Collision Detection of Deformable Objects". In: 2003, pp. 47–54.

[75] C. Tomasi and R. Manduchi. "Bilateral filtering for gray and color images". In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. Jan. 1998, pp. 839–846. DOI: 10.1109/ICCV.1998.710815.

[76] Bill Triggs et al. "Bundle Adjustment — A Modern Synthesis". In: *Vision Algorithms: Theory and Practice: International Workshop on Vision Algorithms Corfu, Greece, September 21–22, 1999 Proceedings*. Ed. by Bill Triggs, Andrew Zisserman, and Richard Szeliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 298–372. ISBN: 978-3-540-44480-0. DOI: 10.1007/3-540-44480-7_21. URL: https://doi.org/10.1007/3-540-44480-7_21.

[77] Juyang Weng, Paul Cohen, and Marc Herniou. "Camera Calibration with Distortion Models and Accuracy Evaluation". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 14.10 (Oct. 1992), pp. 965–980. ISSN: 0162-8828. DOI: 10.1109/34.159901. URL: %5Curl%7Bhttp://dx.doi.org/10.1109/34.159901%7D.

[78] Wu Wen-Tsun. "Basic principles of mechanical theorem proving in elementary geometries". In: *Journal of Automated Reasoning* 2.3 (Sept. 1986), pp. 221–252. ISSN: 1573-0670. DOI: 10.1007/BF02328447. URL: https://doi.org/10.1007/BF02328447.

[79] A. Witkin. "Scale-space filtering: A new approach to multi-scale description". In: *ICASSP '84. IEEE International Conference on Acoustics, Speech, and Signal Processing.* Vol. 9. Mar. 1984, pp. 150–153. DOI: 10.1109/ICASSP.1984.1172729.

[80] Shengen Yan, Guoping Long, and Yunquan Zhang. "StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP '13. Shenzhen, China: ACM, 2013, pp. 229–238. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442539. URL: http://doi.acm.org/10.1145/2442516.2442539.

[81] Z. Zhang. "A flexible new technique for camera calibration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (Nov. 2000), pp. 1330–1334. ISSN: 0162-8828. DOI: 10.1109/34.888718.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 11. Januar 2018

(Darius Rückert)